Escape Dual Report

목차

개요

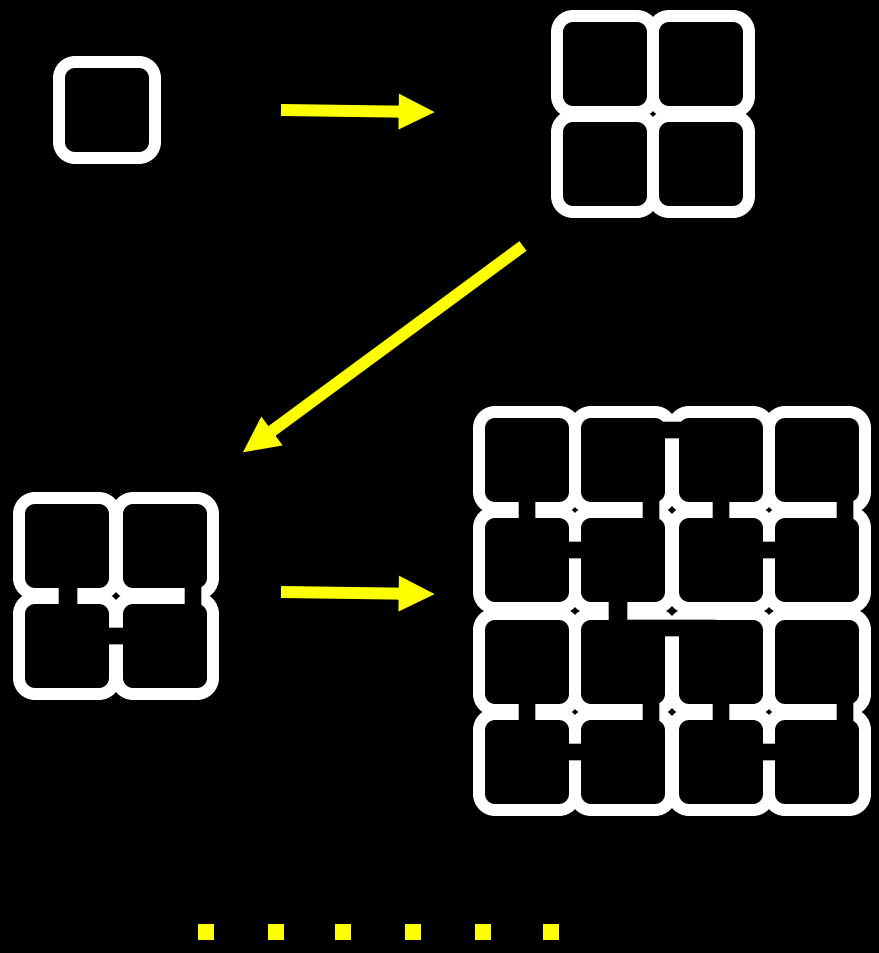
- 플레이어와 컴퓨터가 각자 서로의 미로에서 도착점에 먼저 도달하는 쪽이 승리하는 게임
- 이동 : WASD or 방향키
- 랜덤 미로 생성과 최단거리 길찾기의 응용

핵심기술

- 랜덤 미로생성 : **Fractal Tessellation algorithm**

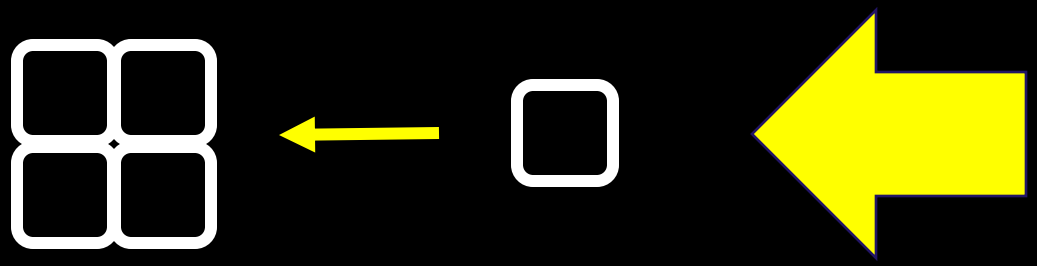
한 미로를 여러 번 복제하여 원본에 인접하도록 배치하고, n 개의 연결부위중 n-1 개의 연결부위에 통로를 만들어 새로운 미로를 생성한다.

원하는 크기의 미로를 만들 때까지 이를 반복한다.



Maze_Generate_Pathfind.class

```
List<List<int>> MazeMaker()
{
```



65 * 65 크기의 미로를 생성하고자 하였다.

미로 원본을 초기화하고 복사하는 과정이다.

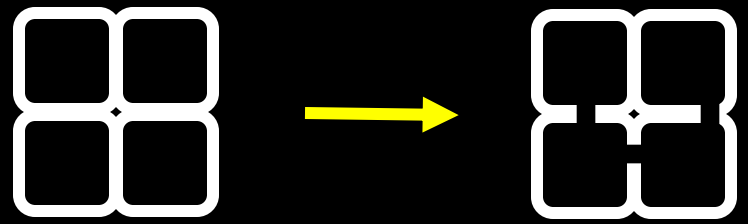
```
while (cnt < 64)//Tessellation algorithm
{
    row = 0;
    while (row <= cnt)//copy to right
    {
        col = 1;
        while (col <= cnt)
        {
            binary_Cell[row][cnt + col] = binary_Cell[row][col];
            col++;
        }
        row++;
    }

    row = 1;
    while (row <= cnt)//copy to down
    {
        col = 0;
        while (col <= cnt)
        {
            binary_Cell[(row + cnt)][col] = binary_Cell[row][col];
            col++;
        }
        row++;
    }

    row = 1;
    while (row <= cnt)//copy to diagonal
    {
        col = 1;
        while (col <= cnt)
        {
            binary_Cell[(row + cnt)][cnt + col] = binary_Cell[row][col];
            col++;
        }
        row++;
    }
```

```
//make hole inside 3 points
horizontal_rand = (int)Random.Range(1, 2 * cnt);
vertical_rand = (int)Random.Range(1, 2 * cnt);

while (horizontal_rand % 2 == 0)...
while (vertical_rand % 2 == 0)...
binary_Cell[horizontal_rand][cnt] = 0;
binary_Cell[cnt][vertical_rand] = 0;
```



n 개의 연결부위중 n-1 개의 통로를 만드는 부분이다.

중요한 점은 이전에 생성한 통로의 패턴과 다른 패턴을 사용해야 한다는 것이다.

그래야 한 위치에서 다른 모든 위치로 이동 가능한 완성된 미로를 만들 수 있다.

```
if (boolean_rand == 1)//additional horizontal hole
{
    if (horizontal_rand > cnt)
    {
        restore_rand = (int)Random.Range(1, cnt);
        while (restore_rand % 2 == 0)
        {
            restore_rand = (int)Random.Range(1, cnt);
        }
    }
    else
    {
        restore_rand = (int)Random.Range(cnt + 1, 2 * cnt);
        while (restore_rand % 2 == 0)
        {
            restore_rand = (int)Random.Range(cnt + 1, 2 * cnt);
        }
    }
    binary_Cell[restore_rand][cnt] = 0;
    boolean_rand = 0;
}
```

4개의 연결부위가 있고, 3개의 통로가 필요하다.

가로, 세로 각각 적어도 하나씩 통로가 존재한다.

이전 패턴에 세로 통로가 2개였다면,
이번 패턴은 가로 통로를 2개 생성 한다.

```
else//additional vertical hole
{
    if (vertical_rand > cnt)
    {
        restore_rand = (int)Random.Range(1, cnt);
        while (restore_rand % 2 == 0)
        {
            restore_rand = (int)Random.Range(1, cnt);
        }
    }
    else
    {
        restore_rand = (int)Random.Range(cnt + 1, 2 * cnt);
        while (restore_rand % 2 == 0)
        {
            restore_rand = (int)Random.Range(cnt + 1, 2 * cnt);
        }
    }
    binary_Cell[cnt][restore_rand] = 0;
    boolean_rand = 1;
}
```

마찬가지로

이전 패턴에서 가로 통로가 2개였다면,
이번 패턴에서 세로 통로를 2개 생성 한다.

```
        row_idx = 0;
        while (row_idx < 65)
        {
            col_idx = 0;
            while (col_idx < 65)
            {
                temp = new Vector3((32 - row_idx % 65) * 0.5f, 0, (32 - col_idx % 65) * 0.5f);
                if (binary_Cell[row_idx][col_idx] == 1)
                {
                    Instantiate(wall, coordinate.position + temp, coordinate.rotation);
                }
                else if (binary_Cell[row_idx][col_idx] == 0)
                {
                    Instantiate(ground, coordinate.position + temp, coordinate.rotation);
                }
                col_idx++;
            }
            row_idx++;
        }

        return binary_Cell;
    }
```

미로의 정보를 담은 65 * 65 크기의 2차원 List 가 완성되었다.
원소가 1 인 경우 벽을 생성하고, 0인 경우 바닥을 생성한다.
미로의 정보를 참조해야 하기 때문에 2차원 List 를 반환한다.

핵심기술

- 최단거리 길찾기 : A * algorithm

현재 방문한 위치에서 주위 방문 가능한 이웃 노드들을 검사한다.

모든 노드는 2가지 주요 정보를 가지고 있다.

(g_cost : 출발지점에서 노드까지 이동한 거리, h_cost : 노드에서 도착지점까지 예상거리)

검사 후, g_cost + h_cost = f_cost 값이 가장 작고 방문한 적 없는 노드로 이동한다.

그리고 그 노드는 방문 완료 상태가 된다.

도착지점에 도달할때까지 이를 반복한다.

Maze_Generate_Pathfind.class

A_star_pathfind(List<List<int>> input_map, int[] input_coordinate)

```
239         List<List<int>> neighbors = new List<List<int>>();
240         List<List<int>> result_route = new List<List<int>>();
241         List<int> col = new List<int>();
242         List<int> empty_col = new List<int>();
243         List<int> convert_current = new List<int>();
244         int[] restore = new int[8];
245         string current;
246         string next_current;
247         Dictionary<string, List<int>> info = new Dictionary<string, List<int>>();
```

List<List<int>> neighbors : 모든 이웃노드들의 정보를 저장한 리스트
List<List<int>> result_route : 최종 반환 경로
List<int> col : 각 노드가 가지는 정보, neighbors 의 원소
List<int> empty_col : 최초 검사에서의 예외처리를 위한 리스트
List<int> convert_current : 현재 위치 정보를 나타내는 리스트
Int[] restore : 인접한 이웃을 검사하기 위한 행렬
String current : 현재 위치 정보를 문자열로 저장한 것
String next_current : 다음으로 방문할 위치를 저장한 문자열
Dictionary<string, List<int>> info : 노드의 좌표정보를 가진 문자열을 key, col 을 value 로 저장한 것

```
        //start point initialize
        col.Add(1);
        col.Add(0);
        col.Add(Mathf.Abs(sp_x - ep_x) + Mathf.Abs(sp_x - ep_y));
        col.Add(sp_x);
        col.Add(sp_y);
        col.Add(sp_x);
        col.Add(sp_y);
        neighbors.Add(col);
        current = sp_x + "," + sp_y;
        info.Add(current, new List<int> { 1,0,Mathf.Abs(sp_x - ep_x) + Mathf.Abs(sp_x - ep_y),sp_x,sp_y,sp_x,sp_y });
        convert_current.Add(sp_x);
        convert_current.Add(sp_y);
```

col 이 포함한 정보
col[0] : 노드 방문 여부
col[1] : 출발지점에서 현재 노드까지 이동한 거리 (g_cost)
col[2] : 현재 노드에서 도착지점까지 예상 거리 (h_cost)
col[3] : 현재 노드의 x 좌표
col[4] : 현재 노드의 y 좌표
col[5] : 부모 노드의 x 좌표
col[6] : 부모 노드의 y 좌표

해당 정보를 info 에 저장한다.

```
while (convert_current[0] != ep_x || convert_current[1] != ep_y)//until arrive at goal
{
    restore[0] = convert_current[0] + 1;
    restore[1] = convert_current[1];
    restore[2] = convert_current[0] - 1;
    restore[3] = convert_current[1];
    restore[4] = convert_current[0];
    restore[5] = convert_current[1] + 1;
    restore[6] = convert_current[0];
    restore[7] = convert_current[1] - 1;
```

목적지에 도달 할 때까지 반복한다.

restore 는 현재 노드의 인접한 노드의 좌표를 검사하기 위한 사전 준비이다.

각각
(x+1, y)
(x-1, y)
(x, y+1)
(x, y-1)
을 의미한다.

```
idx = 0;
while (idx < 8)//look for neighbors
{
    res_x = restore[idx];
    res_y = restore[idx + 1];
    next_current = res_x + "," + res_y;
    //Debug.Log(current);
    //Debug.Log(next_current);
```

이후 인접 이웃을
검사하기 시작한다.

```
//look for range in map size and visitable
if (((res_x >= 0 && res_x < input_map.Count) && (res_y >= 0 && res_y < input_map.Count)) && (input_map[res_x][res_y] != 1))
{
    //Debug.Log(idx);
    if (!info.TryGetValue(next_current, out empty_col))//if empty dictionary
    {
        //Debug.Log(next_current);
        col[0] = 0;
        col[1] = info[current][1] + 1;
        col[2] = Mathf.Abs(ep_x - res_x) + Mathf.Abs(ep_y - res_y);
        col[3] = res_x;
        col[4] = res_y;
        col[5] = convert_current[0];
        col[6] = convert_current[1];
        info.Add(next_current, new List<int> { 0,info[current][1] + 1,Mathf.Abs(ep_x - res_x) + Mathf.Abs(ep_y - res_y),res_x,res_y,convert_current[0],convert_current[1] });
        mid = Binary_insert(neighbors, new List<int> { 0, info[current][1] + 1, Mathf.Abs(ep_x - res_x) + Mathf.Abs(ep_y - res_y), res_x, res_y, convert_current[0], convert_current[1] });
        neighbors.Insert(mid, new List<int> { 0, info[current][1] + 1, Mathf.Abs(ep_x - res_x) + Mathf.Abs(ep_y - res_y), res_x, res_y, convert_current[0], convert_current[1] });
    }
```

인접 노드의 좌표가 전체 List 의 범위 내에 있으며, 그 노드가 벽이 아닌 바닥인 경우, 즉 방문 가능한 경우에만 판별한다.

검사하는 인접 노드가 info 에 등록되지 않은 경우, 즉 처음 판별하는 경우 새로 등록하기 위한 절차이다.
이때, col[1] 의 g_cost 는 현재 노드에서 한 칸 이동하여 인접 노드로 온 경우로 1 을 더하여 저장한다.
col[5], col[6] 에 현재 노드를 부모 노드로 저장한다.

인접 노드의 정보를 info, neighbors 에 저장한다.
이때, neighbors 는 g_cost 의 크기에 따라 오름차순으로 정렬하여 저장한다.

```
        else if (info.TryGetValue(next_current, out empty_col) && info[next_current][1] > info[current][1] + 1)//not empty and shorter way
        {
            //Debug.Log(next_current);
            change_value_idx = 0;
            while (neighbors[change_value_idx][3] != res_x || neighbors[change_value_idx][4] != res_y)
            {
                change_value_idx++;
            }
            neighbors.RemoveAt(change_value_idx);
            info[next_current][1] = info[current][1] + 1;
            info[next_current][5] = convert_current[0];
            info[next_current][6] = convert_current[1];
            mid = Binary_insert(neighbors, info[next_current]);
            neighbors.Insert(mid, info[next_current]);
        }
    }
    idx += 2;
}
```

인접 노드의 정보가 info 에 존재하고, 더 빠른 경로를 찾은 경우
f_cost 가 가장 작은 노드들을 검사하며 경로를 찾다 보면 동일한 노드에서 이전에 검사한 g_cost 보다 새로운 g_cost 가 더 작은 경우가 존재한다.

이런 경우를 반영하여 neighbors 와 info 의 인접 노드 정보를 수정한다.

계속해서 다른 인접노드를 검사한다.

```
//move current
idx = 0;
while (idx < neighbors.Count)
{
    if (neighbors[idx][0] != 1)
    {
        f_cost = neighbors[idx][1] + neighbors[idx][2];
        break;
    }
    idx++;
}

//no way arrive to goal
if (idx == neighbors.Count)
{
    return result_route;
}
```

인접노드 검사를 끝내고, 다음으로 이동할 노드를 찾기 위한 과정이다.

neighbors 는 f_cost 의 크기에 따라 정렬되어 있으므로
순서대로 노드 방문 여부를 판별한다.
방문하지 않은 노드가 있다면 f_cost 를 저장하고 넘어간다.

만약 전부 방문했는데도 현재노드가 도착지점이 아니라면 경로가 없는 것으로 반환한다.

```
    h_cost = neighbors[idx][2];
    next_node_idx = idx;
    while (idx < neighbors.Count && neighbors[idx][1] + neighbors[idx][2] == f_cost)
    {
        if (neighbors[idx][2] < h_cost && neighbors[idx][0] == 0)
        {
            h_cost = neighbors[idx][2];
            next_node_idx = idx;
        }
        idx++;
    }

    current = neighbors[next_node_idx][3] + "," + neighbors[next_node_idx][4];
    convert_current = StrConvert(current);
    neighbors[next_node_idx][0] = 1;
    info[current][0] = 1;
    //Debug.Log("next");
}
```

만약 같은 f_cost 를 가진 노드들이 여러개 있는 경우, 그중 h_cost 가 가장 작은 노드를 찾는다.

다음 현재 노드를 확정하고, 그 노드를 방문 완료 상태로 저장한다.

```
        //route
        result_route.Insert(0, convert_current);
        while (convert_current[0] != sp_x || convert_current[1] != sp_y)
        {
            res_x = info[current][5];
            res_y = info[current][6];
            current = res_x + "," + res_y;
            convert_current = StrConvert(current);
            result_route.Insert(0, convert_current);
        }

        return result_route;
    }
```

현재 노드가 목적지에 도달 한 경우, 마지막으로 경로를 확정한다.

info 에 각 노드의 정보 중 부모 노드에 대한 정보가 담겨있다.
이를 추적하며 부모가 시작점이 될 때까지 반복하며 경로를 저장한다.

마지막으로 경로를 반환한다.

개선점

- 난이도 선택

타이틀 신에서 게임신으로 넘어가기 전에 난이도 선택을 하고 컴퓨터 오브젝트의 속도에 따라 난이도 변화를 주고 싶었다. Scene 에서 scene 으로 변수를 전달하는 방법이 있다면 사용해보는 것이 좋을 것 같다.

- 카운트 다운

게임신에 입장 하자마자 전조도 없이 진행을 하게 된다. 플레이어가 준비를 할 시간적 여유를 주고 싶었는데 timescale 이나 코루틴의 사용이 익숙하지 않았다.

- 디자인 디테일 추가

게임이라기에는 여러가지 요소가 아쉬웠다. 전체적인 디자인, 아이콘, 효과음 등이 추가되면 좋겠다.

후기

C++ 와 C# 의 여러 차이점을 알 수 있었던 좋은 기회였고, 동시에 Unity 에 익숙해지는데 좋은 경험이었다.

특히 핵심기술 A* algorithm 을 구현중에 체감 한 것이 많은데, C++ map 과 비슷한 C# Dictionary 를 사용했지만, key와 value 로 List 를 사용하다보니 vector 와 다른점을 알게 되었다.

Vector 와 달리 List 는 해시 형태로 저장되어 내부 원소가 바뀌더라도 list 자체에 영향을 주지 않는다는 점이었다.