Path finding games using A* algorithm

A MINI PROJECT REPORT

18CSC305J - ARTIFICIAL INTELLIGENCE

Submitted by

ABHINAV D TRIVEDI [RA2011003011181]
JEYANTH PRAKASH [RA2011003011166]
V ALLEN JEROME[RA2011003011167]

Under the guidance of

DR. M ARUNA

Assistant Professor, Department of Computer Science and Engineering

in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

in

COMPUTER SCIENCE & ENGINEERING

of

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
Deemed to be University u/s 3 of UGC Act, 1956

S.R.M. Nagar, Kattankulathur, Chengalpattu District

MAY 2023

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that Mini project report titled **"Path finding games Using A* algorithm**'' is the bona fide work of (**ABHINAV D TRIVEDI (RA2011003011181) JEYANTH PRAKASH (RA2011003011166) , V ALLEN JEROME (RA2011003011167)**) who carried out the minor project under my supervision. Certified further, that to the best of my knowledge, the work reported herein does not form any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE

GUIDE NAME
GUIDE
Assistant Professor
Department of Computing Technologies

SIGNATURE

Dr. M. Pushpalatha
HEAD OF THE DEPARTMENT
Professor & Head
Department of Computing Technologies

Title : Path finding in games using A* algorithm.

Problem Statement: One of the greatest challenges in the design of realistic Artificial Intelligence (AI) in computer games is agent movement. Pathfinding strategies are usually employed as the core of any AI movement system. Pathfinding strategies have the responsibility of finding a path from any coordinate in the game world to another. Systems such as this take in a starting point and a destination; they then find a series of points that together comprise a path to the destination. A games' AI pathfinder usually employs some sort of precomputed data structure to guide the movement. At its simplest, this could be just a list of locations within the game that the agent is allowed move to. Pathfinding inevitably leads to a drain on CPU resources especially if the algorithm wastes valuable time searching for a path that turns out not to exist.In this project we have presented the use of A* algorithm for path finding in a 2D game due to it being complete and its ability to find the optimal path at a low cost.

ACKNOWLEDGEMENT

We would like to thank our lab in-charge Dr. M Aruna for giving us the opportunity and imparting us with the knowledge to do this project.

INDEX

1. What is A*?

It is a searching algorithm that is used to find the shortest path between an initial and a final point.
Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue.
What A* Search Algorithm does is that at each step it picks the node according to a value-'f' which is a parameter equal to the sum of two other parameters – 'g' and 'h'. At each step it picks the node/cell having the lowest 'f', and process that node/cell.
We define 'g' and 'h' as simply as possible below
g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.).

Algorithm

We create two lists – Open List and Closed List (just like Dijkstra Algorithm)

// A* Search Algorithm
1. Initialize the open list
2. Initialize the closed list
   put the starting node on the open
   list (you can leave its f at zero)

3. while the open list is not empty
   a) find the node with the least f on
      the open list, call it "q"

   b) pop q off the open list

   c) generate q's 8 successors and set their
      parents to q

   d) for each successor

i) if successor is the goal, stop search

ii) else, compute both g and h for successor
successor.g = q.g + distance between
successor and q
successor.h = distance from goal to
successor (This can be done using many
ways, we will discuss three heuristics-
Manhattan, Diagonal and Euclidean
Heuristics)

successor.f = successor.g + successor.h

iii) if a node with the same position as
successor is in the OPEN list which has a
lower f than successor, skip this successor

iV) if a node with the same position as
successor is in the CLOSED list which has
a lower f than successor, skip this successor
otherwise, add the node to the open list
end (for loop)

e) push q on the closed list
end (while loop)

In our project we have used the Euclidean distance to calculate the heuristics.

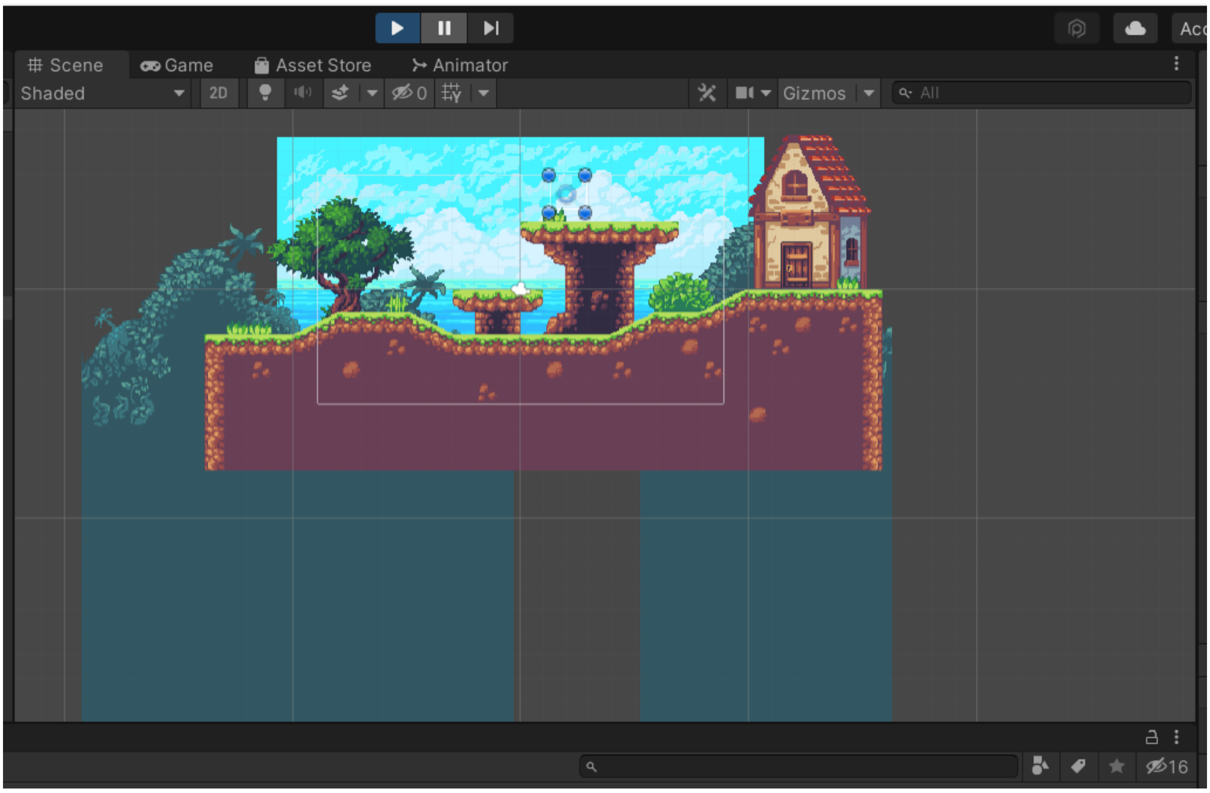
2. Implementation in Unity

We have implemented this project in unity as a 2D project. The aim of the project is to have a player object and an enemy object. The enemy object will home in on the player object using pathfinding powered by the A* algorithm.

A) Setting up the project -

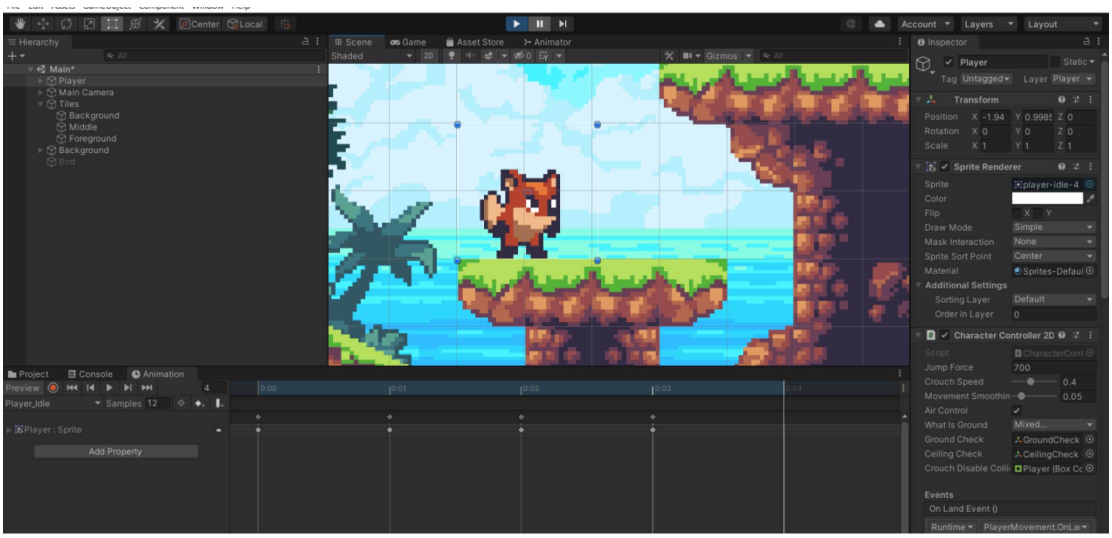
We start this project by using the base 2D game preset available in Unity. To setup the base of the project we used 2D assets available for free on the Unity Asset Store. For this project, we have utilized the Sunny Land asset from the Unity store. We follow this up by building a map using the assets we imported from the Unity Store and building the collision mesh for all objects and setting the appropriate layers for the foreground and the background.

Personal* <DX11>

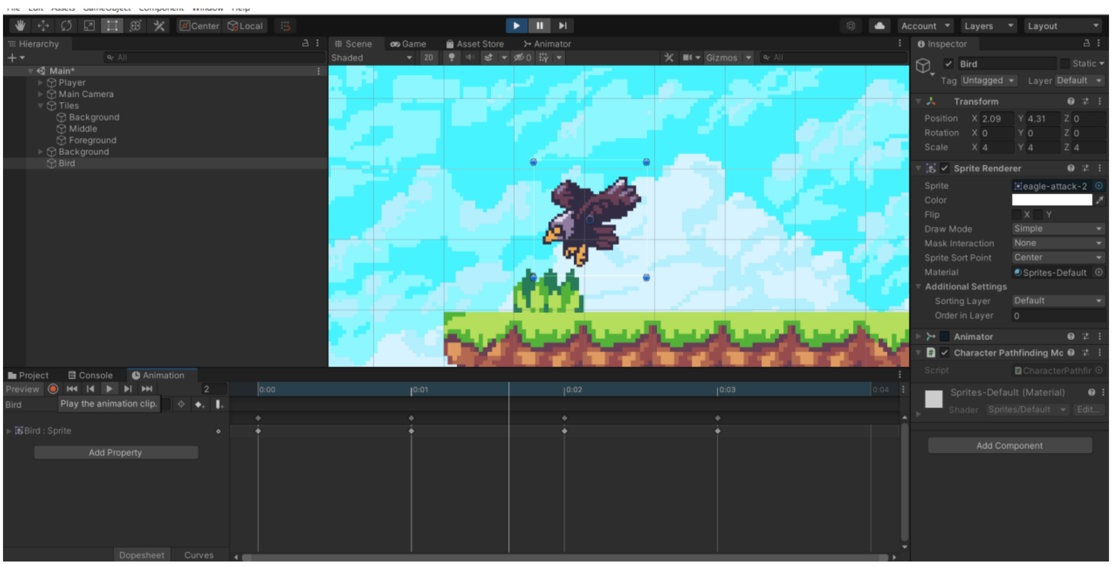


The map we will be using.

Next, we create and animate the characters. We have created two characters, one, which will be controlled by the player and the other which will be controlled by the AI. We first set up the player character. We do this by creating the game object and adding the appropriate collision meshes and player controller components. Next we set up the bird enemy AI character. We setup the collsion mesh for it as well and animate it as well.



Creating and animating the player character



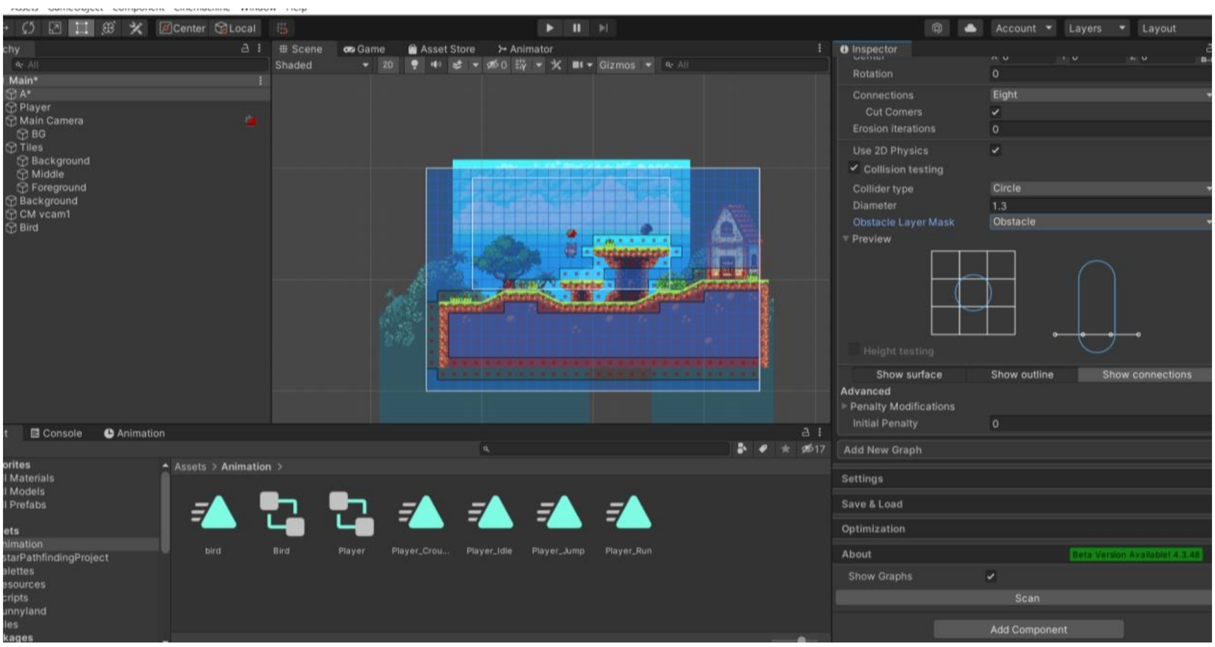
Creating and animating the enemy bird AI character.

At the end of this phase we are able to move the player around the map using our controls but thats about all that can be done in the game.

B) Implementing path finding using A*(creating the graph)

Next we implemented the pathfinding algorithm in unity. Unity has in built AI pathfinding support, however, this was only intended to be used for 3D navmeshes and not 2D. Therefore, we build a custom solution using A* algorithm utilising some inbuilt unity tools.

The game map has to be prepared or pre-processed before the A* algorithm can work. This
involves breaking the map into different points or locations, which are called nodes. To achieve this we use an inbuilt Unity tool to create a graph. We then use the layers which we had previously created to allow the tool to be able to detect the obstacles.



Detecting obstacles using inbuilt unity pathfinding tool. Red are obstacles and blue are areas which can be navigated.

C) Implementing path finding using A*(Implementing the algorithm)

We now come to the most important part of our project, the implementation of A* algorithm. This is done using a script component in C#. We utilise the graph created by the PathFinding utility in Unity to find a path from the startNode(the enemy AI characters current position) to the endNode(the player characters current position)

Algorithm -

1. Initialise openList and closedList.
2. Push startNode in openList
3. Initialise g cost to max value
4. Create function to calculate h cost(distance to end node)
5. Create function to calculate f cost(f cost= g cost+h cost)
6. Create function to find lowest f cost node
7. Loop while(openList.count>0)
8. Initialise currentnode to lowest f cost node
9. If currentnode is end node return path and exit loop
10. Remove current node from open list and add it to closed list
11. Else return null

Code -

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Pathfinding {

    private const int MOVE_STRAIGHT_COST = 10;
    private const int MOVE_DIAGONAL_COST = 14;

    public static Pathfinding Instance { get; private set; }

    private Grid<PathNode> grid;
    private List<PathNode> openList;
    private List<PathNode> closedList;

    public Pathfinding(int width, int height) {
```

```
		Instance = this;
		grid = new Grid<PathNode>(width, height, 10f, Vector3.zero,
(Grid<PathNode> g, int x, int y) => new PathNode(g, x, y));
	}

	public Grid<PathNode> GetGrid() {
		return grid;
	}

	public List<Vector3> FindPath(Vector3 startWorldPosition, Vector3
endWorldPosition) {
		grid.GetXY(startWorldPosition, out int startX, out int startY);
		grid.GetXY(endWorldPosition, out int endX, out int endY);

		List<PathNode> path = FindPath(startX, startY, endX, endY);
		if (path == null) {
			return null;
		} else {
			List<Vector3> vectorPath = new List<Vector3>();
			foreach (PathNode pathNode in path) {
				vectorPath.Add(new Vector3(pathNode.x,
pathNode.y) * grid.GetCellSize() + Vector3.one * grid.GetCellSize()
* .5f);
			}
			return vectorPath;
		}
	}

	public List<PathNode> FindPath(int startX, int startY, int endX, int
endY) {
		PathNode startNode = grid.GetGridObject(startX, startY);
		PathNode endNode = grid.GetGridObject(endX, endY);

		if (startNode == null || endNode == null) {
			// Invalid Path
			return null;
		}

		openList = new List<PathNode> { startNode };
		closedList = new List<PathNode>();

		for (int x = 0; x < grid.GetWidth(); x++) {
			for (int y = 0; y < grid.GetHeight(); y++) {
```

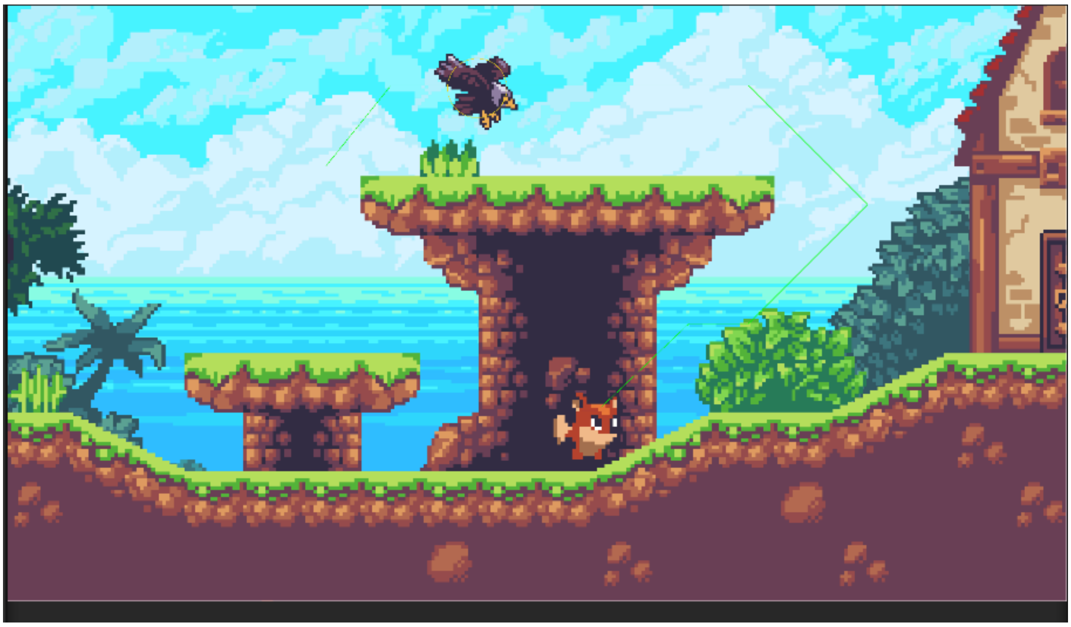
```
                PathNode pathNode = grid.GetGridObject(x, y);
                pathNode.gCost = 99999999;
                pathNode.CalculateFCost();
                pathNode.cameFromNode = null;
            }
        }

        startNode.gCost = 0;
        startNode.hCost = CalculateDistanceCost(startNode, endNode);
        startNode.CalculateFCost();

        PathfindingDebugStepVisual.Instance.ClearSnapshots();
        PathfindingDebugStepVisual.Instance.TakeSnapshot(grid,
startNode, openList, closedList);

        while (openList.Count > 0) {
            PathNode currentNode = GetLowestFCostNode(openList);
            if (currentNode == endNode) {
                // Reached final node

PathfindingDebugStepVisual.Instance.TakeSnapshot(grid, currentNode,
openList, closedList);

PathfindingDebugStepVisual.Instance.TakeSnapshotFinalPath(grid,
CalculatePath(endNode));
                return CalculatePath(endNode);
            }

            openList.Remove(currentNode);
            closedList.Add(currentNode);

            foreach (PathNode neighbourNode in
GetNeighbourList(currentNode)) {
                if (closedList.Contains(neighbourNode)) continue;
                if (!neighbourNode.isWalkable) {
                    closedList.Add(neighbourNode);
                    continue;
                }

                int tentativeGCost = currentNode.gCost +
CalculateDistanceCost(currentNode, neighbourNode);
                if (tentativeGCost < neighbourNode.gCost) {
                    neighbourNode.cameFromNode = currentNode;
```

```
                    neighbourNode.gCost = tentativeGCost;
                    neighbourNode.hCost =
CalculateDistanceCost(neighbourNode, endNode);
                    neighbourNode.CalculateFCost();

                    if (!openList.Contains(neighbourNode)) {
                        openList.Add(neighbourNode);
                    }
                }

PathfindingDebugStepVisual.Instance.TakeSnapshot(grid, currentNode,
openList, closedList);
            }
        }

        // Out of nodes on the openList
        return null;
    }

    private List<PathNode> GetNeighbourList(PathNode currentNode)
{
        List<PathNode> neighbourList = new List<PathNode>();

        if (currentNode.x - 1 >= 0) {
            // Left
            neighbourList.Add(GetNode(currentNode.x - 1,
currentNode.y));
            // Left Down
            if (currentNode.y - 1 >= 0)
neighbourList.Add(GetNode(currentNode.x - 1, currentNode.y - 1));
            // Left Up
            if (currentNode.y + 1 < grid.GetHeight())
neighbourList.Add(GetNode(currentNode.x - 1, currentNode.y + 1));
        }
        if (currentNode.x + 1 < grid.GetWidth()) {
            // Right
            neighbourList.Add(GetNode(currentNode.x + 1,
currentNode.y));
            // Right Down
            if (currentNode.y - 1 >= 0)
neighbourList.Add(GetNode(currentNode.x + 1, currentNode.y - 1));
            // Right Up
```

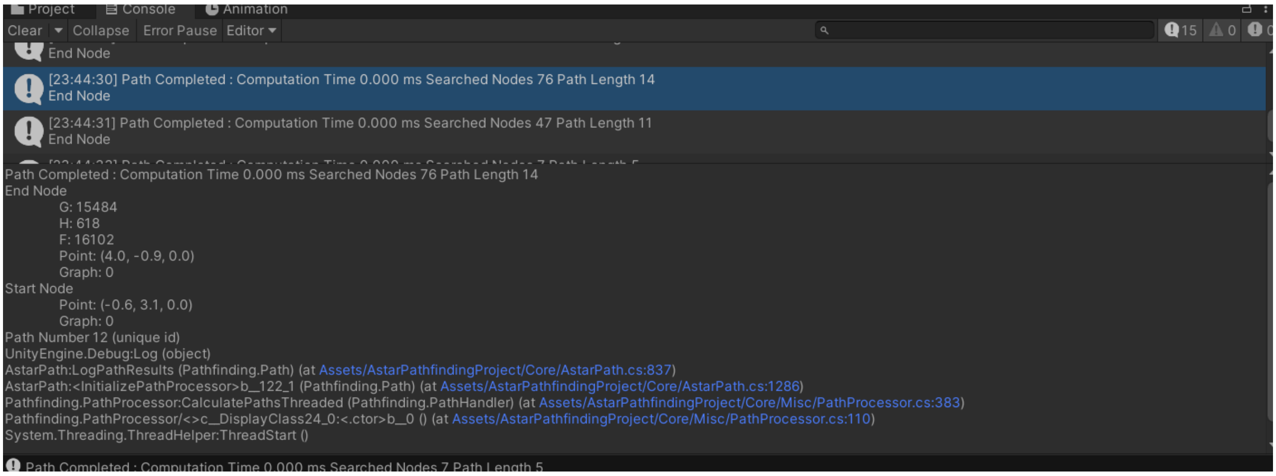
```
                if (currentNode.y + 1 < grid.GetHeight())
neighbourList.Add(GetNode(currentNode.x + 1, currentNode.y + 1));
            }
            // Down
            if (currentNode.y - 1 >= 0)
neighbourList.Add(GetNode(currentNode.x, currentNode.y - 1));
            // Up
            if (currentNode.y + 1 < grid.GetHeight())
neighbourList.Add(GetNode(currentNode.x, currentNode.y + 1));

            return neighbourList;
        }

        public PathNode GetNode(int x, int y) {
            return grid.GetGridObject(x, y);
        }

        private List<PathNode> CalculatePath(PathNode endNode) {
            List<PathNode> path = new List<PathNode>();
            path.Add(endNode);
            PathNode currentNode = endNode;
            while (currentNode.cameFromNode != null) {
                path.Add(currentNode.cameFromNode);
                currentNode = currentNode.cameFromNode;
            }
            path.Reverse();
            return path;
        }

        private int CalculateDistanceCost(PathNode a, PathNode b) {
            int xDistance = Mathf.Abs(a.x - b.x);
            int yDistance = Mathf.Abs(a.y - b.y);
            int remaining = Mathf.Abs(xDistance - yDistance);
            return MOVE_DIAGONAL_COST * Mathf.Min(xDistance,
yDistance) + MOVE_STRAIGHT_COST * remaining;
        }

        private PathNode GetLowestFCostNode(List<PathNode>
pathNodeList) {
            PathNode lowestFCostNode = pathNodeList[0];
            for (int i = 1; i < pathNodeList.Count; i++) {
                if (pathNodeList[i].fCost < lowestFCostNode.fCost) {
                    lowestFCostNode = pathNodeList[i];
```

```
                }
            }
            return lowestFCostNode;
        }

}
```

Now our enemy AI character is able to track and follow the player character.



Note the faintly visible green line which is the path being calculated by our AI agent.



F, G and H values being calculated and displayed in the console.

3) Results and Conclusion

A) Limitations of A* -

A* requires a large amount of CPU resources, if there are many nodes to search through as is the case in large maps which are becoming popular in the newer games. In sequential programs this may cause a slight delay in the game. This delay is compounded if A* is searching for paths for multiple AI agents and/or when the agent has to move from one side of the map to the other. This drain on CPU resources may cause the game to freeze until the optimal path is found. Game designers overcome these problems by tweaking the game so as to avoid these situations.

The inclusion of dynamic objects to the map is also a major problem when using A*. For example once a path has been calculated, if a dynamic object then blocks the path the agent would have no knowledge of this and would continue on as normal and walk straight into the object. Simply reapplying the A* algorithm every time a node is blocked would cause excessive drain on the CPU.

A key issue constraining the advancement of the games industry is its over reliance on A* for pathfinding. This has resulted in game designers getting around the associated dynamic limitations by tweaking their designs rather than developing new concepts and approaches to address the issues of a dynamic environment [Higgins02]. This tweaking often results in removing/reducing the number of dynamic objects in the environment and so limits the dynamic potential of the game.

B) Possible Solution -

A potential solution to this is to use neural networks or other machine learning techniques to learn pathfinding behaviours which would be applicable to realtime pathfinding.

C) Conclusion -

Thus we have studied the implementation of A* algorithm for path finding in games using Unity. We have also explored its limitations.

4) Bibliography and Workflow

A) Workflow -

Importing and building of assets and animation done by - Khushi Bhakuni and Akash Dubey.

Implementation of A* and other scripting and documentation done by - Utkarsh Chaurasia and Rishi J.V.

B) Bibliography -

Assets -
https://assetstore.unity.com/packages/2d/characters/sunny-land-103349

Unity -
https://docs.unity.com/

A* algorithm implementation -

Youtube - Code Monkey, Brackeys
https://arongranberg.com/astar/