Contents

Introduction

In this assignment, I will be endeavouring to plan and program a prototype game that utilises A* pathfinding algorithms, cellular automata and marching squares procedural generation. I will first be writing a product backlog of everything that will be needed for the prototype, I will then be creating a Technical Design Document that will help me in planning my code. After this I will be using the document created to create my prototype, when I encounter an error I will add it to the error report later in this document and if the error requires me to change the Technical Design Document then I will do so. Upon completion of the prototype, I will write an improvement report that will outline what improvements I need to make or what I would like to add to the prototype in the future.

Game Prototype Description and backlog

This description outlines the mechanics to be implemented in the prototype.

The game that I am going to be creating for this prototype is an idle game. This game will be similar to that of *Non-Stop Knight,* in which the player must delve deeper and deeper into a dungeon, defeat all the enemies on that floor and the proceed to the next floor. In my prototype for this game, I will be focusing on creating a procedurally generated environment as well as incorporating A* pathfinding algorithms. As this is an idle game the user will not have much input (if any) into what the character does, this is because the player will automatically find the enemy and attack, once all the enemies have been killed the player will then proceed to the ladder to the next floor. Outside of this prototypes scope is giving the player abilities that the user can activate, this will be implemented after the prototype has been completed. The environment will be procedurally generated by using a mixture of Cellular Automata and Marching Squares. The enemies will be spawned throughout the level using a weighted random if the number generated is lower than the spawn probability than the enemy will be spawned. Finally, after the player has killed an enemy the player is rewarded with score that the player can add to a scoreboard

Figure 1 Non-Stop Knight (FlareGames, 2017)

In order to create this prototype I am going to need the following scripts:

Game Controller

The game manager is responsible for controlling gameplay elements of the game. It will keep track of the player's score and will have a function that will update the score. This Function will be called upon the death of an enemy and will add the inputted score. The game manager will also be responsible for spawning the Enemies, player and ladder into the level after it has been generated. The enemies will be spawned using a weighted random that will spawn an enemy only if they random value generated is over a determined value.

	Game Controller
Functions	
+Update Score	Updates and displays the score on screen
-GetRandomNumber	returns a random integer between 0 and 100
-Setup Level	Sets up the level for gameplay. Calls Spawn Enemies and Place Ladder on the Level Generator.
+AddScore	Adds the inputted score to the score value
+AddScore Overload	This is the overload function to the original addscore function, this version also takes in a multiplier parameter that multiplies the inputted score.
+GetCurrentScore	This function returns the current score
Variables	
+score	This will be an integer value that stores the player's score
+ProbabilityToSpawnEnemy	This variable will be ranged between 1 and 100, this value will be used to know whether to spawn an enemy.
+Enemy, Ladder, Player	These three items will all be game objects and represent the prefab of the object with the same name.

Cellular Automata

The level Generator will be using Cellular Automata to create a procedurally generated environment for the player to explore. It will create a 2D array of Integers ranging from 0 to 1 which will be utilised by the Mesh Generator to generate a mesh.

	Cellular Automata
Functions	
-Generate Grid	Fills the 2D grid Array with random Ints between 0 and 1 as a start of Cellular Automata generation
-Smooth Grid	Iterates through the grid reading how many neighbours each node has if a node has more than X amount of neighbours make it a floor tile
-GetAmountOfNeighbours	Returns an integer that represents how many neighbours the given tile has.
+GetAmountOfNeighbours Overload	The overload function of the original GetAmountOfNeighbours. This version will take an extra parameter that allows the search area to be changed.
-StartMachingSquares	This function calls the Generate mesh function in the marchingSquares class
Variables	
-Grid	This is a 2D array that holds integer values that represent the type of tile it is.
+Grid Height	An integer that represents the height of the grid
+Grid Width	An integer value that represents the width of the grid
+ Probability	This integer value will be the probability of this tile being a wall.

Marching Squares

The mesh Generator will be using Marching Squares to create a mesh for the level, this mesh will be generated using the 2D array that was created in the Level Generator class. It will turn this 2D array of ints into a 2D array of Control Nodes. Depending if a control node is active or not it will generate a mesh accordingly.

	Marching Squares
Functions	
+GenerateMesh	This function will be responsible for generating the mesh. It will initialize the squareGrid, vertices list and triangles list. It will triangulate all the squares and then create the mesh using the vertices and triangles lists
TriangulateSquares	This function will switch through all the different square configuration and create a mesh from them depending on the configuration
CreateMeshFromPoints	This will take in an unknown amount of nodes and will assign the vertices using these nodes and create a triangle from them.
AssignVertices	This function will simply give the vertices a vertex index if they don't already and will add them to the vertices list.
CreateTriangle	Adds the inputted nodes into the triangles list
Variables	
+Vertices	A list of Vector 3's, the position of the vertices in the world
+Triangles	A list of ints that holds the triangles of the mesh.
Classes	

Node class	Holds a Vector 3 for that nodes positions and an integer value for the vertex index
Control Node	The control node inherits from the Node class and has its position property set from the base node. It also contains a bool if this position is active, and float for the size of the tile.
Square	Each square will have reference to its control nodes and nodes and will have an integer value to know what configuration it is in, this configuration will change depending on which of its nodes are active.
SquareGrid	This class will create a 2D array of squares and call the other classes.

Marching Squares Configurations

The image below will be used to help me know the different marching squares triangle configurations. The corners of each square are representing the control nodes, between each control node there will be an intermediate node that will help in creating a triangle. For example in case 7 the top and the left intermediate node will be used to help create the triangle for that particular configuration.

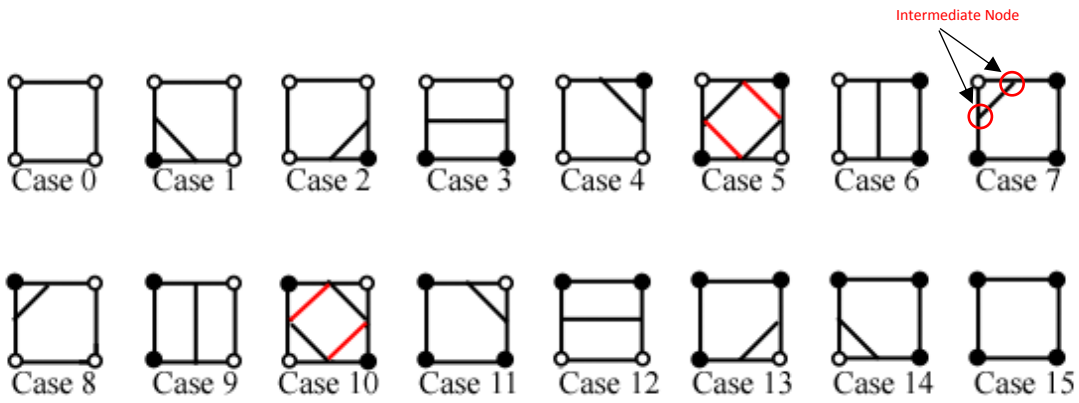


Figure 1. Marching Squares Configurations (Polytech, n.d.)

A* Grid

A* pathfinding will be used for the player to move from point A to point B. A Node class will be needed for this to be accomplished. A node will be a point in 3D space, it will hold a world position, be walkable or not walkable and hold an x and y position in a 2d array of nodes. Each node will also need 3 integer values, a G Cost, H cost and an F Cost. These costs will be used to determine the shortest path to the target node.

If the node is touching something that is walkable then the Walkable property for that node is true, if not then it is false.

	A* Grid
Classes	
+Node	**Bool** - is this node walkable
	Vector3 - this nodes world position
	2 x **Int** - the position of This node (X, Y) in the 2D node array.
	Int - This Nodes G Cost (Distance from starting node)
	Int - This Nodes H Cost (Distance from end node)
	Int - This Nodes F Cost (G Cost + H Cost)
Functions	
- Create Grid	Sets up the **Node** grid (2D Array) assigning them their world position, position in the grid (X, Y) and tells them if they are walkable or not.
+WorldToGridPos	This function turns the inputted world position into a position of the grid and returns the node at that position.
+ FindNeighbours	This function loops through the surrounding neighbours of the inputted node adding them to a list and returning a list of neighbours
Variables	
grid[,]	holds a 2D **array** of **nodes**
gridsizeX and Y	This will be the size of the grid
nodeDiameter	the diameter of each node
node radius	the radius of each node… half the diameter

gridWorldSizeXandY	The world size of the grid on the x and y
UnwalkableMask	the layer mask that will be used to check for unwalkable collisions.

A* Pathing

This script will be responsible for finding and return the path to the object that has requested it. This script will be running the A* pathfinding algorithm. The path will keep searching though all the values with the lowest F cost until it reaches the target. Once at the target the path will retrace through all its parents and add each node along the way into a list that will be the path.

	A* Pathing
Functions	
+ Find Path	This function is called to find a path between a start world position and end world position, it returns a list of nodes as the path
-RetracePath	This function retraces the path back from the end node through each node's parent until it gets to the start node. it then returns the path.
+ GetDistance	Returns the distance between two nodes
Variables	
OpenSet	<**List**>The nodes that are needing to be checked
ClosedSet	<**List**> The nodes that have already been checked
Path	a list of nodes that represents the path
ThereIsNoPath	This bool will be true if no path can be found.

A* Movement

The A star movement script will enable the player to move along the path that was returned from the A* pathfinding script. The path index variable will be used to determine which node in the path the player must move to next, once the player has arrived at the position of this node then path index will be incremented by one, this will repeat until path index is equal to the length of the list. If a path cannot be found then the can't find path variable will return true and the object that the player is trying to move to will be destroyed.

	A* Movement
Functions	
GetAndFollowPath	This function will take in a target position and will find the shortest path to the target and follow it.
Variables	
hasPath	This will be true if a path is returned
atTarget	if the player is at the end of the path this bool will be true
cantFindPath	If a path can not be found this will be true
pathIndex	this integer value will be the point in the path the character is currently moving to.
Path	This will be a list of A* nodes that will represent the path

Character Class

The Character Class will be the base for all character within the game, both players and AI. All characters will have current health, max health, score that will be awarded upon death and they will all implement the ICharacterInterface which will handle deaths differently depending on if they are a player or an enemy.

	Character
Variables	
-Max Health	(**int**) The max amount of health the character can have
+Current Health	(**int**) The state of the character's current health
+Model	This will be the game object of this particular character
Interface	
ApplyDamge	All character must have an implementation of this function that will vary how damage is dealt to them
OnDeath	all characters must have an implementation of this function that will do different things when a certain character type dies

Player Class

The player class will inherit from the Character class, its max health, current health and model variables will be set via the characters constructor. The player will also inherit from the character interface meaning there are certain functions that it has to implement. The Player class will have a unique variable for movement speed.

	Player Class
Variables	
+Movement Speed	(**float**) How fast the player can move
Functions	
OnDeath	when the player is killed the game will end
Apply Damage	When damage is applied to the player the inputted value will be subtracted from his current health.

Enemy Class

The Enemy class will inherit from the Character class, its max health, current health and model variables will be set via the characters constructor. The Enemy will also inherit from the character interface meaning there are certain functions that it has to implement. The Enemy class will have a unique variable for the score that will be rewarded to the player when this enemy is killed.

	Enemy Class
Variables	
+ScoreReward	This integer value will be the score awarded to the player when this enemy dies.
Functions	
OnDeath	when the Enemy is killed score will be rewarded to the player
Apply Damage	When damage is applied to the Enemy the inputted value will be subtracted from his current health.

Player

The player is the centrepiece of any game. In this instance, the player has health, movement speed and a damage variable. Along with health, the player has a Max health value that will serve as a cap for the health. The movement speed will control how fast the player can move and will be fed into the A* pathfinding. When the player is within a certain radius an enemy it will begin to attack it.

	Player
Variables	
maxHealth	This value will be fed into the playerClass Constructor
speed	This value will be fed into the playerClass Constructor
Target	this will be a game object reference for the target
Functions	
Find Target	Finds an enemy, if no enemies are available then it will find the ladder to move to the next floor
MoveTowardsTarget	This function will use the target variable and send it to the A*Pathfinding in order to get and follow a path. If the player is at the target then deal damage to the target.

UI Leaderboard

This leader board will show the top 10 highest scores from when the scene was loaded. When the player pressed the add current score button, the current score will be added to the leaderboard and the leader board will be updated. The top 10 in the leader board list will be shown to the player and LINQ Query will be used to find them. In this prototype, the leader board will be reset each time the game is a played, however, in the final product the game will feature an online leader board that would require an SQL database and some PHP (this is outside the scope of this prototype).

	UI Leader board
Functions	
AddCurrentScore	This function will be assigned to a button press and will add the current score to the leader board, it will also call the update leader board function.
UpdateLeaderboard	This function will update the leader board so that it shows the top 10 scores.
Variables	
Highscores	This variable will be a list of ints that will hold the high scores
HighscoresText	this will be an array of UI texts within the game scene.
delegate	this delegate will be used to call the update leader board function

Assets

As this is a prototype most of these items (if any) will not be implemented into the build, however, if the game was to continue beyond the prototype stage these are the items the game would need. As you can see there is no terrain in the below table, this is because the terrain will be generated through code, however, there is a need to have textures for the terrain.

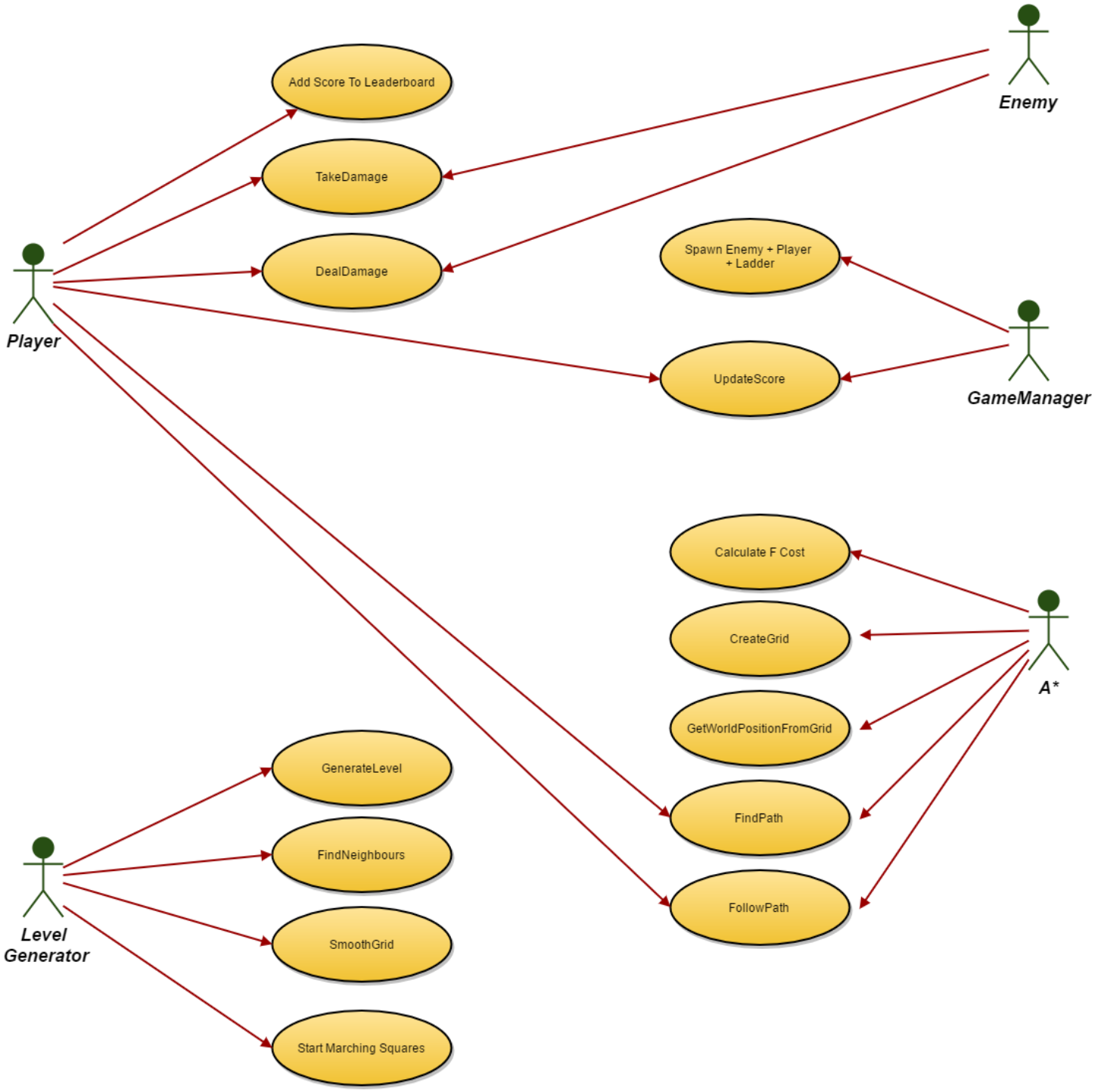
Models	**Textures**	**Music**
Player	Player	Attack SFX
Enemy	Enemy	Background music
Ladder	Ladder	Walking SFX
	Cave	UI music
	Lava	
	UI	

Backlog Schedule

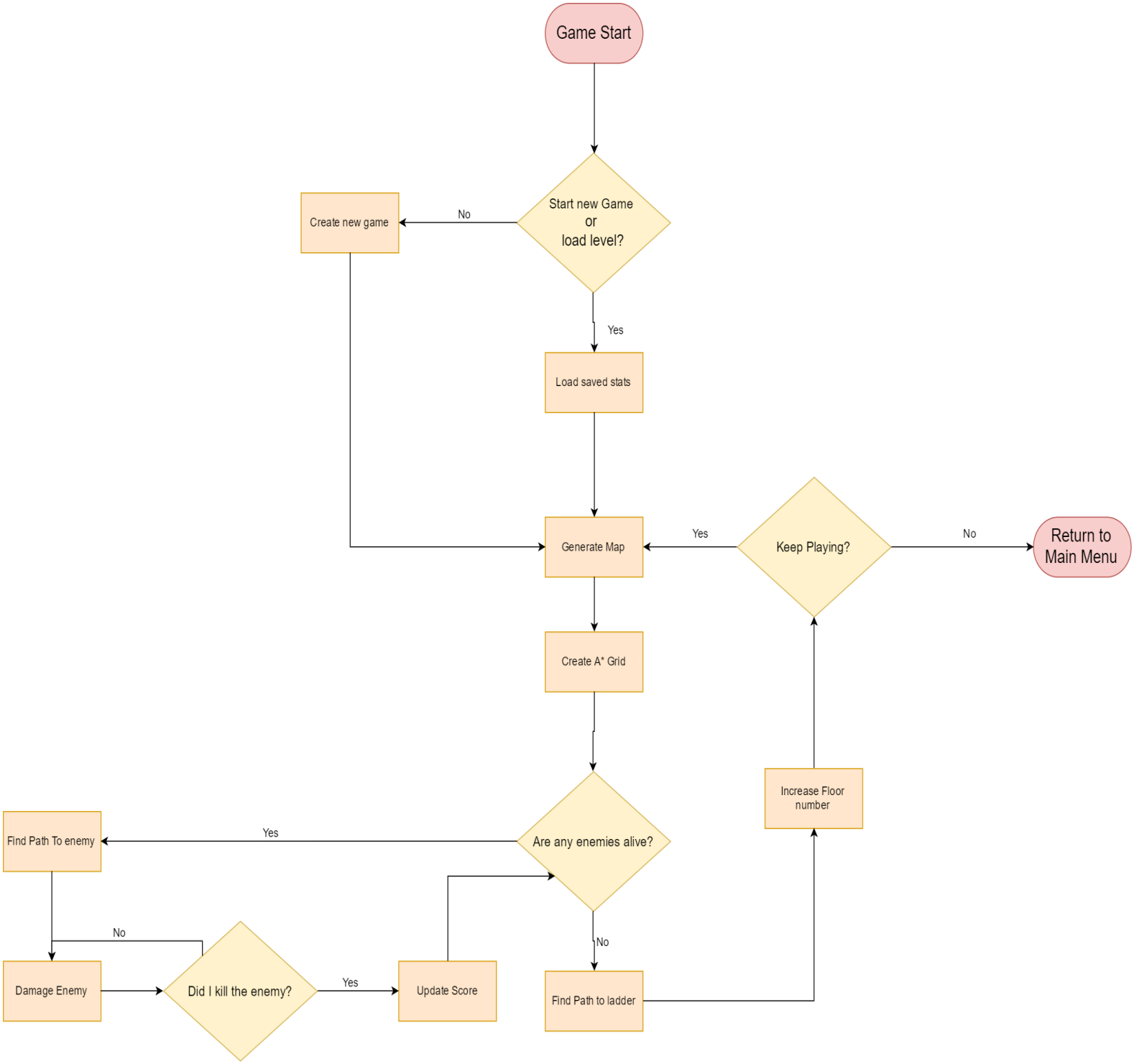
The following times represent how long each item will take to complete after the planning stage. If these times were to include the time that will take to plan tasks they would be much longer.

Item	Estimated Time to Complete
Cellular Automata	1hr
Marching Squares	4hr
A* Node	30 minutes
A* Grid	2hr
A* Pathing	3hr
A* Movement	1hr
Character Class	30 min
Player Class	30 min
Enemy Class	30 min
Player	3hr
Game Controller	2hr

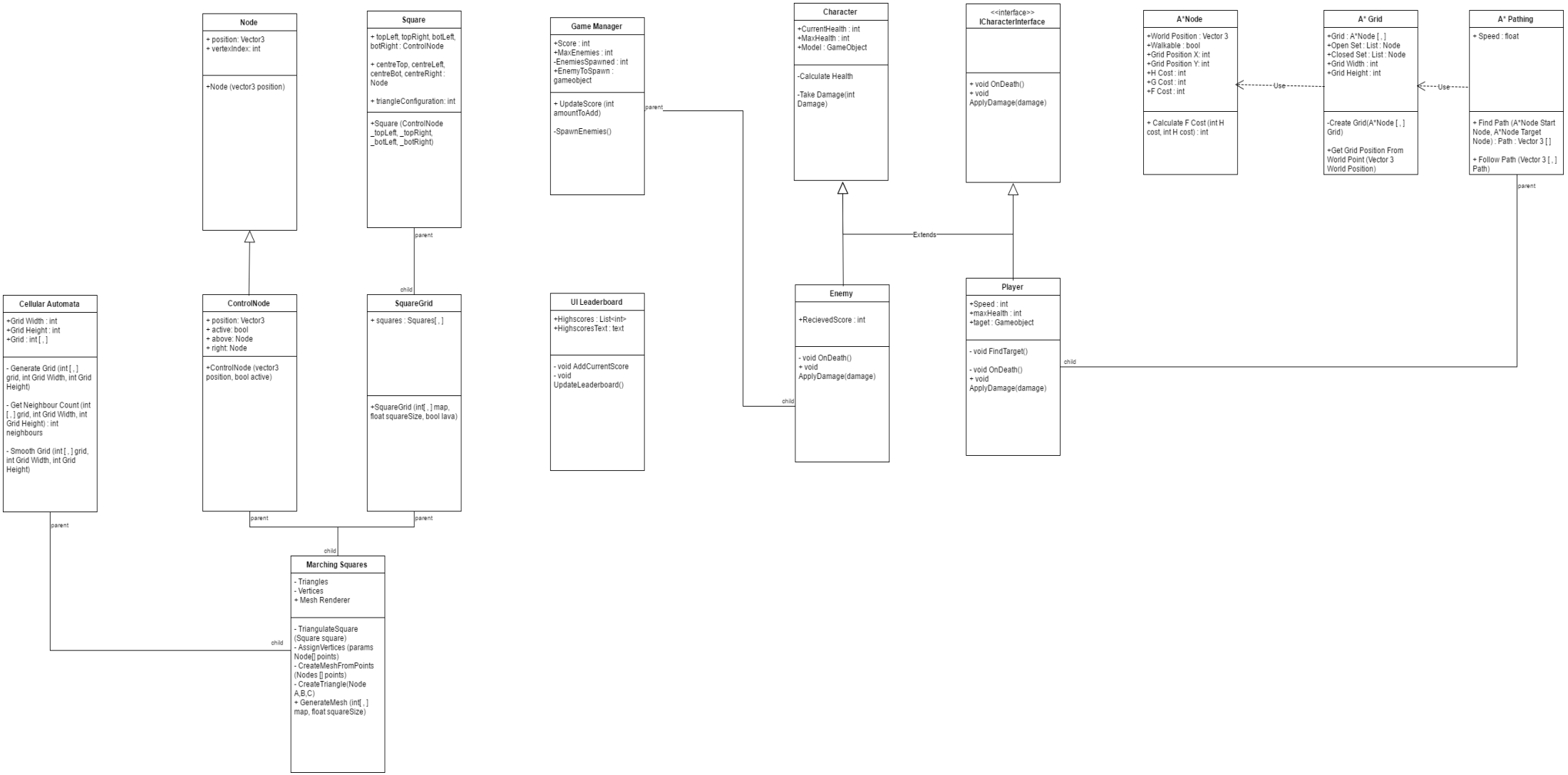
Use Cases


Add Score To Leaderboard
TakeDamage
DealDamage
Enemy
Player
Spawn Enemy + Player + Ladder
UpdateScore
GameManager
Calculate F Cost
CreateGrid
GetWorldPositionFromGrid
FindPath
FollowPath
A*
GenerateLevel
FindNeighbours
SmoothGrid
Start Marching Squares
Level Generator

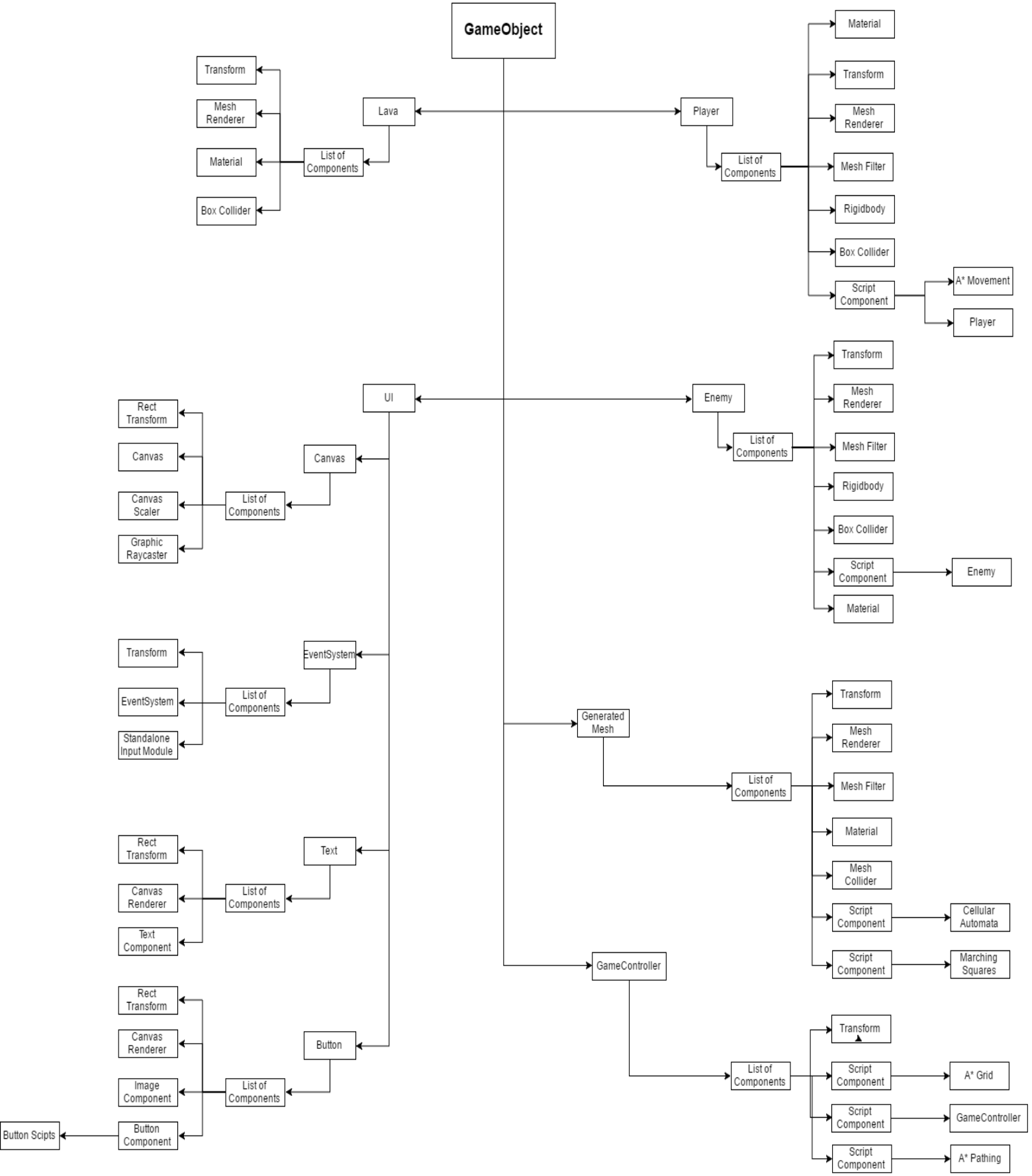
Gameplay Flow



Architectural Design (UML) **All flowcharts will be supplied as images also.**



Component Design



Components Checklist

This is checklist is a simpler version of the detailed flow chart above.

Entities	Components							
	Transform	**Mesh Renderer**	**Mesh Filter**	**Collider**	**Rigidbody**	**Script**	**Material**	**Image Renderer**
Player	✓	✓	✓	✓	✓	✓	✓	
Enemy	✓	✓	✓	✓	✓	✓	✓	
Mesh Generator	✓	✓	✓	✓		✓	✓	
Lava	✓	✓	✓	✓		✓	✓	
Game Controller	✓					✓		
Camera	✓					✓		
UI	✓					✓		✓

Interfaces Design

Entities	Character Interfaces		
	Take Damage	**Follow Path**	**Check Dead**
Player	✓	✓	✓
Enemy	✓		✓

The checklist above shows what interfaces of the character class the player and the enemy will be using. Both the player and the enemy will be able to take damage and die therefore they use the take damage and check dead interfaces. However, in this game, only the player will follow the path, and as such only, the player uses the follow path interface.

Entities	Monobehavior Interfaces
	Destroy
Enemy	✓
Player	✓

The check dead interface listed above uses the mono behaviour interface to destroy a game object and because of that the player and enemy both use this interface.

Entities	Vector 3 Interfaces	
	MoveTowards	**Lerp**
Player	✓	
Camera		✓

The player uses the vector3 move towards interface in order to move towards the next position in the path. The camera uses the lerp function to have a smooth transition (Unity, 2017b) when the player is movement, using lerp stop the jittery movement of the camera which would irritate the user.

Entities	A* Movement Interfaces
	Get/Follow Path
Player	✓

When the player is following the path the transform interface of look at is used to make the player face in the direction that he is moving. When checking if the player is at the target position the transform position interface will be used.

Entities	Transform Interfaces	
	Look at	Position
Player	✓	✓

As the player is the only game object that will be following the path it is the only object that uses the get and follow path interface of the A* movement class.

Entities	Quaternion Interfaces
	Identity
Game Controller	✓
A*Grid	✓

The game controller uses the identity interface while spawning objects (player, ladder and enemies), this allows these objects to spawn created in the scene with no rotation (Unity, 2017a). This interface is also used in the A* Grid when checking for collisions.

Entities	Math f Interfaces		
	Clamp01	**RoundToInt**	**Abs**
A*Grid	✓	✓	
Pathing			✓

The A* grid uses the clamp01 interface to clamp the grid percentage values between 0 and 1. It also uses the round to int interface to make the returned position of the world to grid position function a whole number. The pathing script uses mathF Abs to create an unsigned variable (Microsoft, 2017b).

Entities	List and hash set Interfaces			
	Add	**Contains**	**Remove**	**Clear**
Pathing	✓	✓	✓	✓
Marching Squares	✓			

The pathing class uses a lot of list interfaces, it uses add to add items to the list, it uses contain to see if something is inside the list and hash set, remove to remove items from them and clear to remove all of its contents. The marching squares class only uses the add interface of the list, this is because it never has to remove items or check if anything is inside it.

Entities	Mesh Filter Interfaces			
	Mesh	**Triangles**	**Vertices**	**Recalculate normals**
Marching Squares	✓	✓	✓	✓

The Marching Squares class needs to access the following interfaces of the Mech Filter Component. It accesses the mesh interface to assign the generated mesh to the Mesh Filter, the triangles interface to pass in the triangles list and the vertices interface for the same reason. Finally, the recalculate normal interface is used to ensure that all the normals of the generated mesh are facing the right direction.

Entities	Mesh Collider Interfaces
	Shared Mesh
Marching Squares	✓

The Marching Squares classes uses the shared mesh interface of the mesh collider in order to set the collision mesh to the mesh that the class has generated. Doing this allows collision to happen on the generated mesh.

Entities	Unity Engine Interfaces						
	Random Range	Game Object	Find	Get Component	Load Level	Instantiate	Compare Tag
Game Controller	✓	✓	✓	✓		✓	
Camera Follow		✓	✓	✓			
Change Floor					✓		
Cellular Automata	✓						
Character Classes			✓				
Player		✓	✓	✓	✓		✓
Pathing				✓			

There are many classes throughout this project that are going to be using Unity Engine Interfaces, this is because I am using the Unity Engine. The Game Controller is going to be using the most interfaces, this is because it is responsible for spawning objects within the scene, therefore it needs access to game object and Instantiate, find (to find the mesh generator and A* grid) as well as get component to get the script components. The player uses the second highest amount of interfaces from the Unity Engine. He needs access to Game Object and find to find the enemies in the scene, then needs Get Component to deal damage to the enemies, lastly, compare tag will be used to help return if an enemy can be found.

Data Structure Design

A **List** will be used to contain the open set of A* nodes. I have chosen to use a **list** over an **array** because I will be needing to add and remove items from the open set at will, an array would not allow me to add more items then it was set to contain (Microsoft, 2017a). I chose not to use a hash set for the open set because I know there will not be too many items in the open set at any given time, meaning it does not need to be optimised for search. **Lists** will also be used in the Marching Squares script; the list will be used to keep track of the position of each vertex and the triangles. I am using a list here because I do not know how many vertices and triangles there will be at run time.

I will be using a **Hash set** for the closed set of the A* pathfinding algorithm. Hash sets are optimised for searching, this makes them the best choose to search to see if an item is already in the closed set (Stack Overflow, 2014). According to Stack Overflow (2014), a Hash set has a consistent search speed, no matter the size, whereas a list's search constantly increases with its size.

A List is also going to be used for the path, this is because we have node idea how long the path is going to be, by using a list we can dynamically decrease or increase its size. It was possible to use a **Queue** to store each position in the path, however, I felt this was unnecessary as I am already returning a list that I can easily iterate through in the movement class.

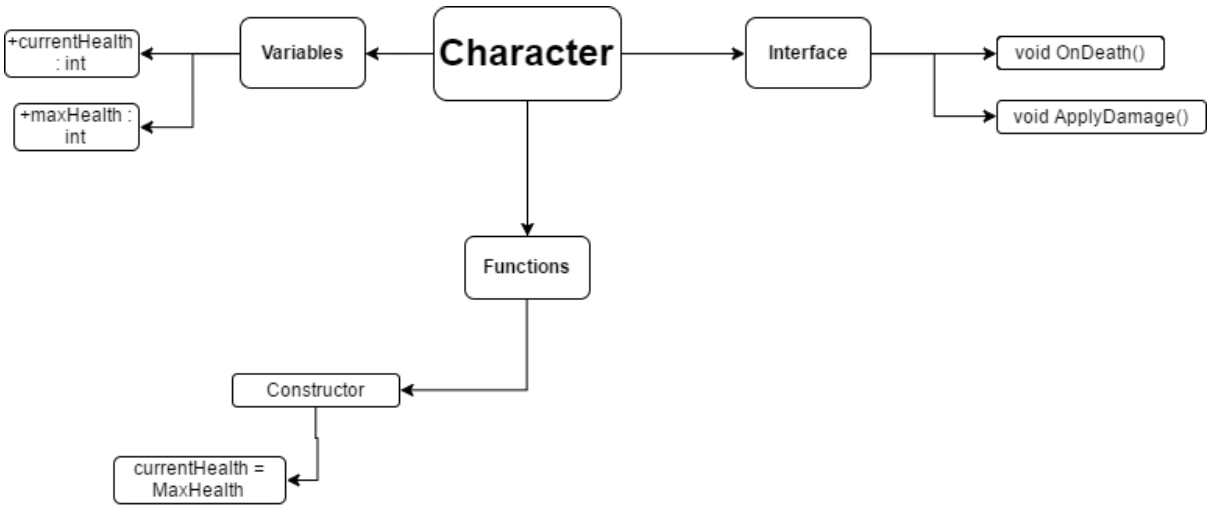
I will be creating a class for **A*Node**. These nodes will be used to keep track of their location in the world, their location in the 2D array and if they are walkable or not. Having these node classes are fundamental to the A* pathfinding algorithm.

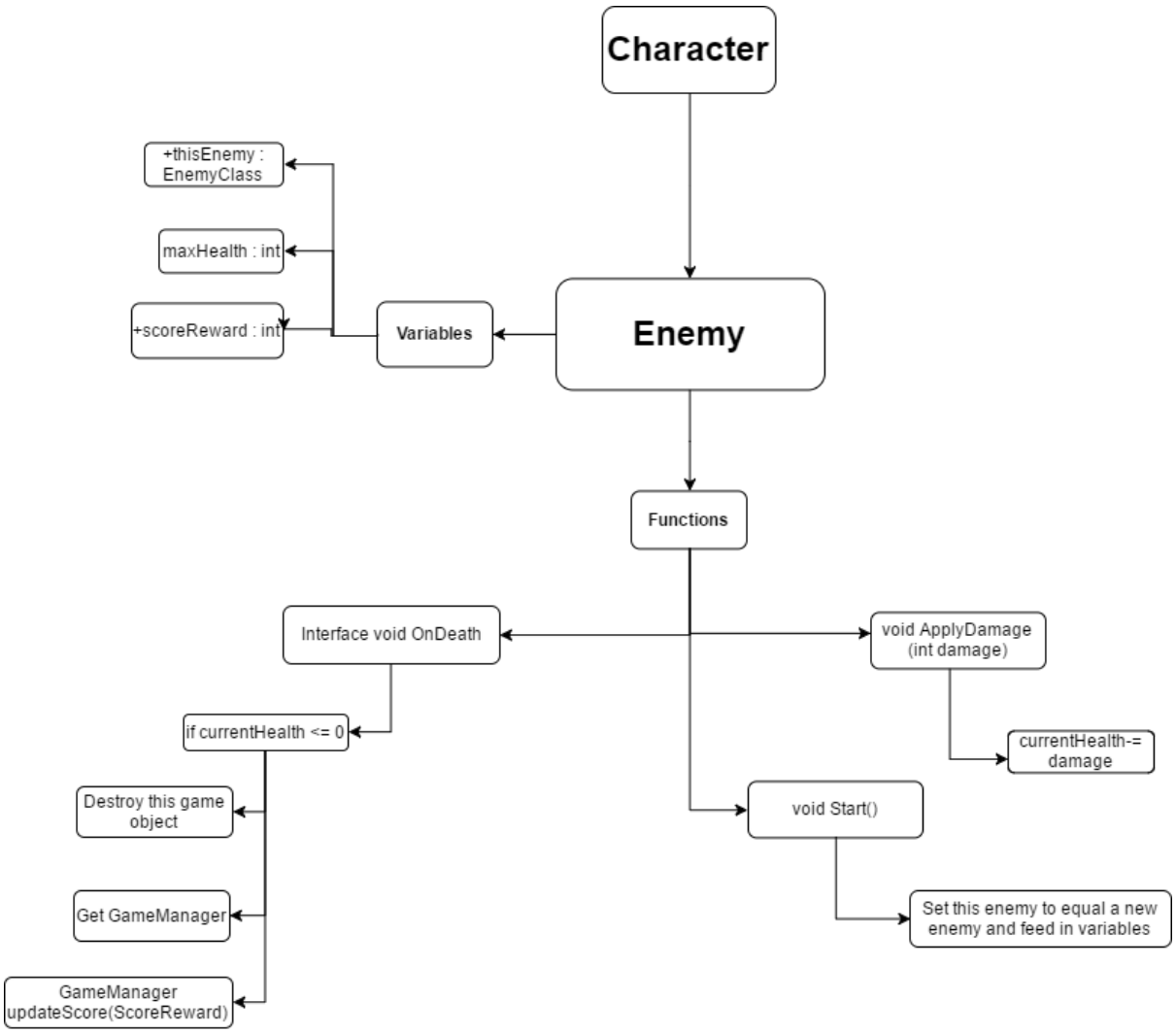
Marching Squares also requires its own form of node, this node will simply be called **node** to avoid confusion with the **A*Node**. These nodes will be responsible for holding their position in the world and their vertex index. There will then be a secondary node called the **Control Node**, these nodes will inherit from the base class of node will set their world position with the node constructor. The **Control Node** will store if that node is active and will keep track of the node directly above and node directly to the right of it.

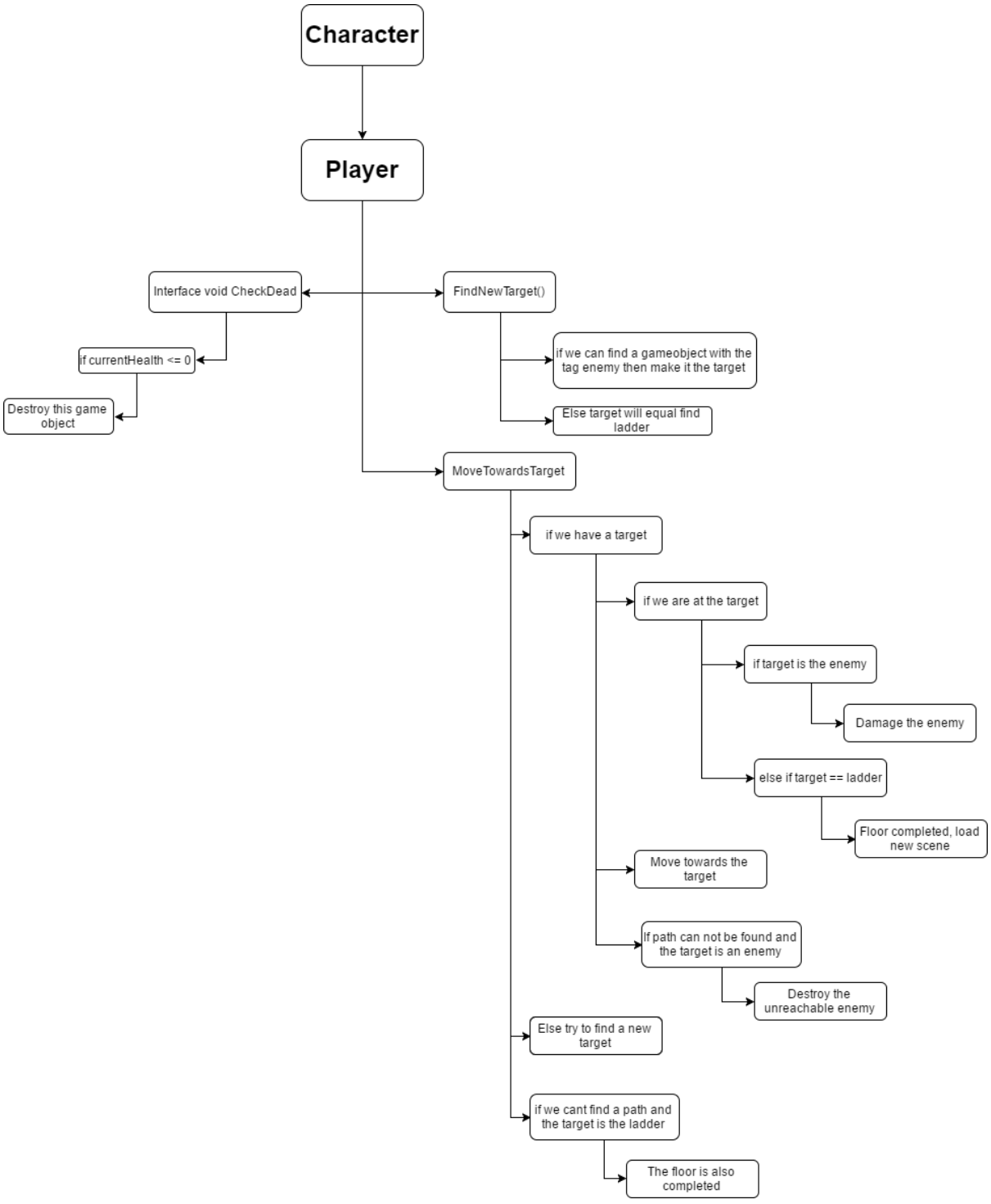
An **Array** is being used to hold all the high score text items. I am using an array here because they are much easier to use within the Unity editor and are easier to manage within it. I could have used a list in this situation but as I can easily drag and drop items into an array using the Unity Inspector I have

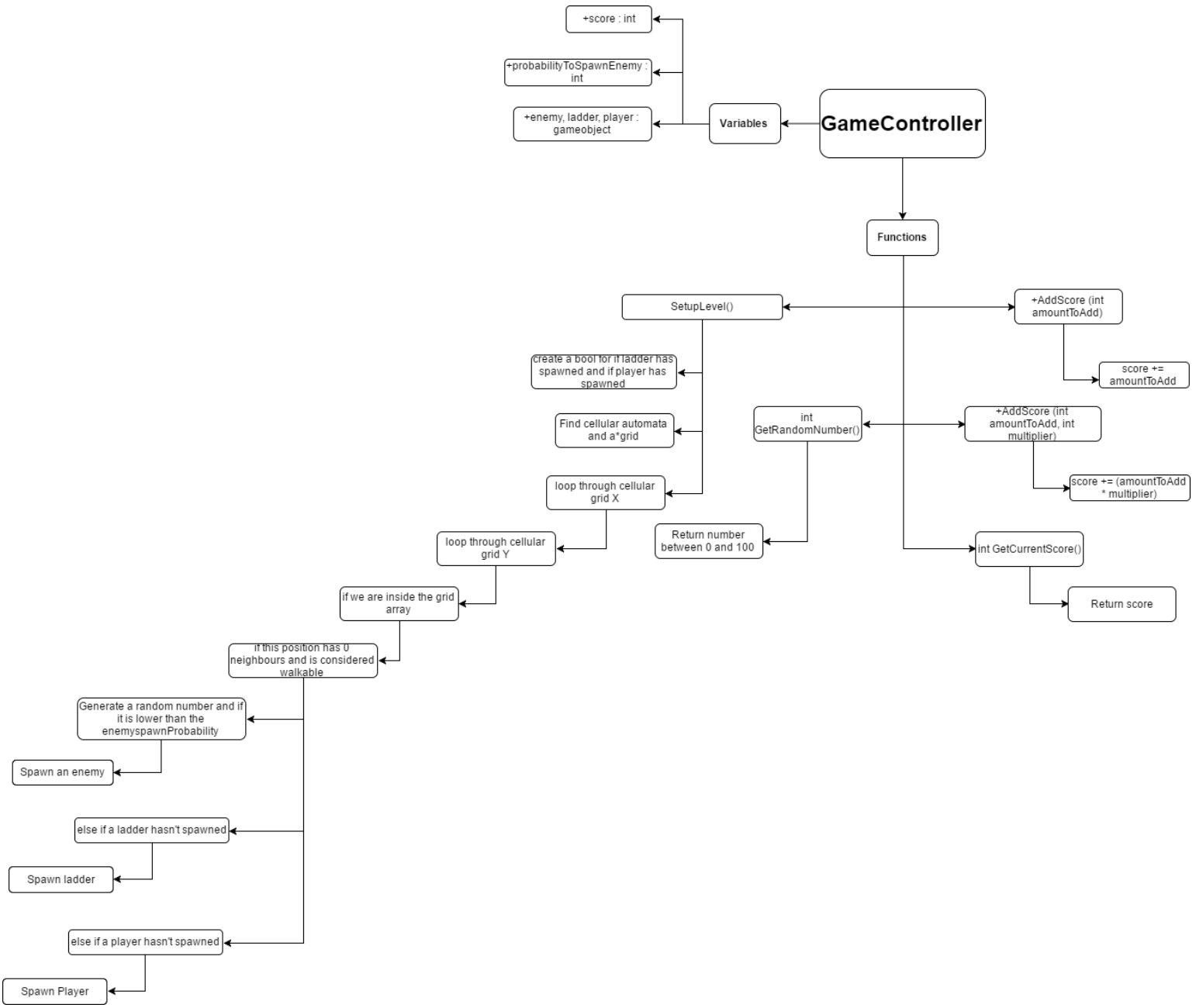
chosen to go with an array. **Two Dimension Arrays** are using in both the A* grid and the Cellular Automata classes, these arrays are used to represent the game world. I have used a 2D array here because it seemed to be the most logical solution to me and I am unsure of how else to have represented the world.

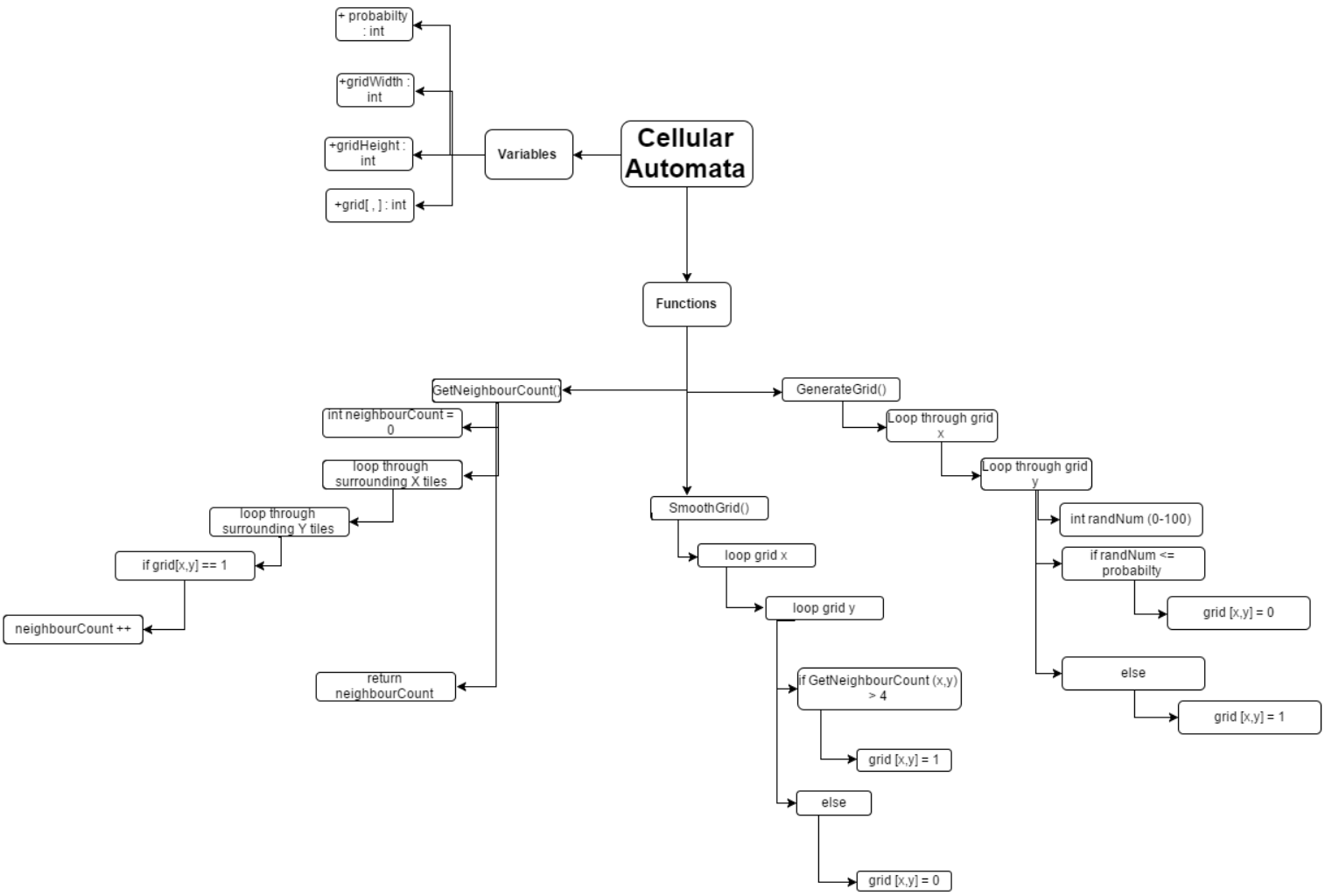
Algorithm Design (flowcharts/pseudocode)





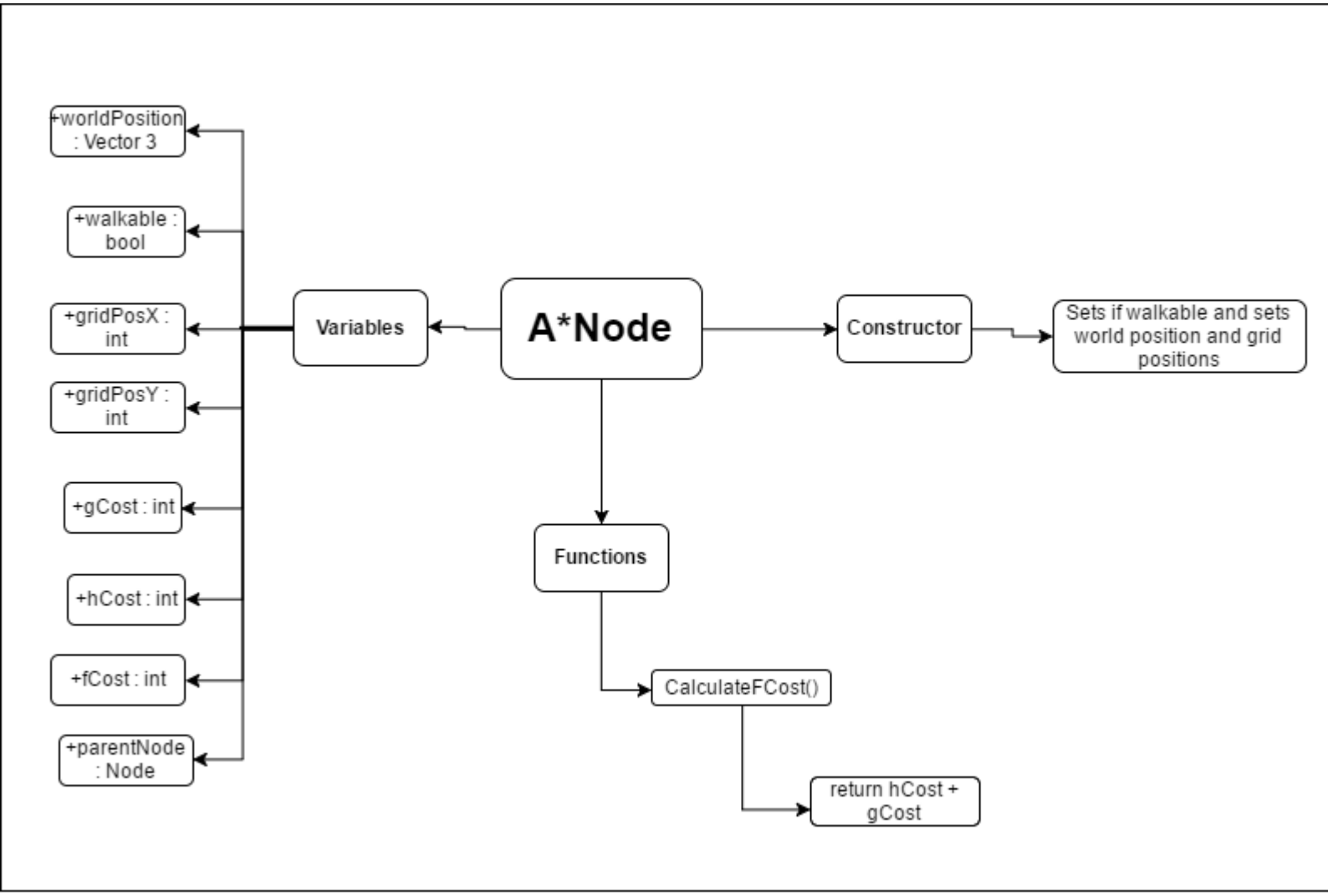

Character
Player
Interface void CheckDead
if currentHealth <= 0
Destroy this game object
FindNewTarget()
if we can find a gameobject with the tag enemy then make it the target
Else target will equal find ladder
MoveTowardsTarget
if we have a target
if we are at the target
if target is the enemy
Damage the enemy
else if target == ladder
Floor completed, load new scene
Move towards the target
If path can not be found and the target is an enemy
Destroy the unreachable enemy
Else try to find a new target
if we cant find a path and the target is the ladder
The floor is also completed

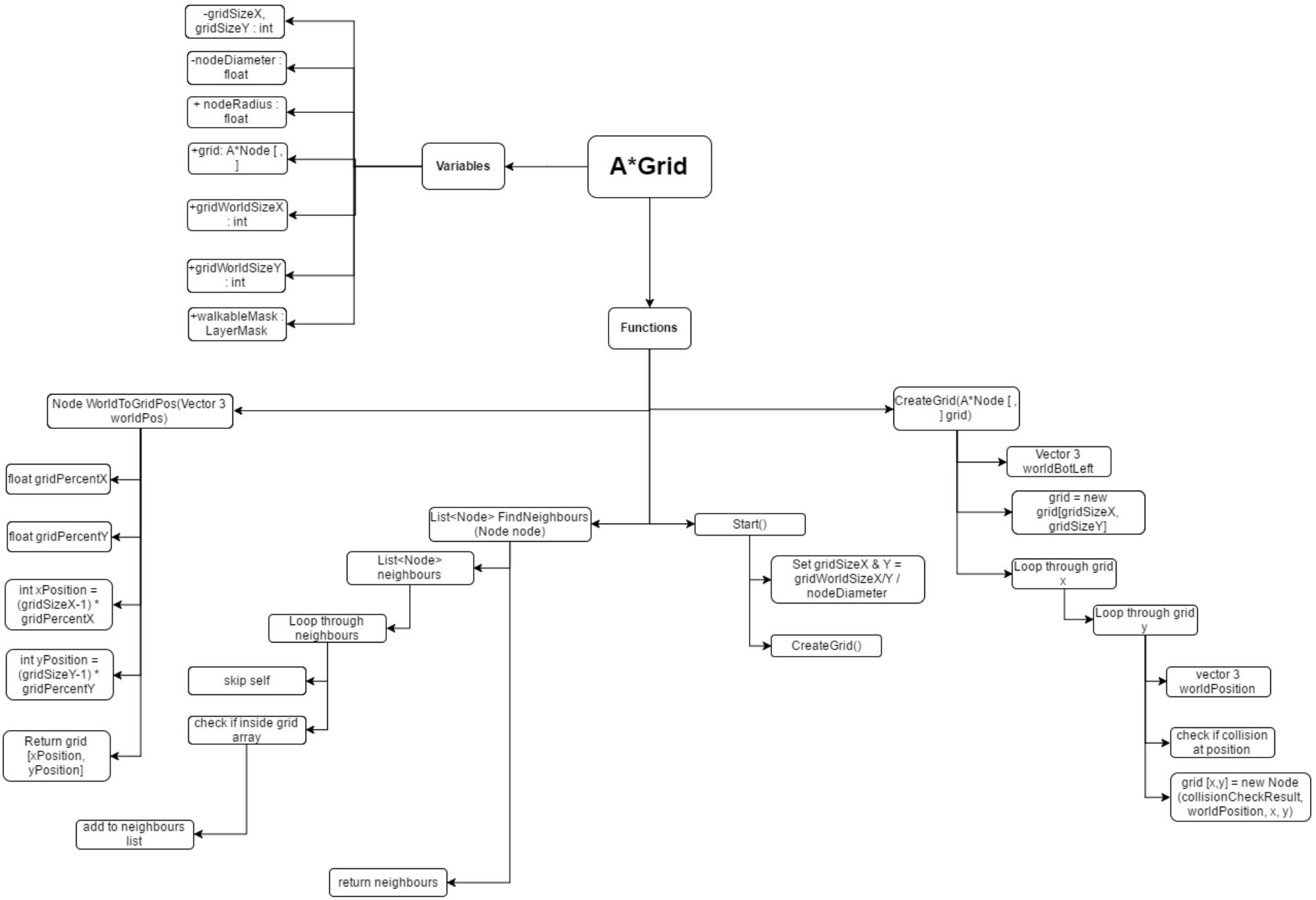

GameController
Variables
+score : int
+probabilityToSpawnEnemy : int
+enemy, ladder, player : gameobject
Functions
SetupLevel()
Create a bool for if ladder has spawned and if player has spawned
Find cellular automata and a*grid
loop through cellular grid X
loop through cellular grid Y
if we are inside the grid array
if this position has 0 neighbours and is considered walkable
Generate a random number and if it is lower than the enemyspawnProbability
Spawn an enemy
else if a ladder hasn't spawned
Spawn ladder
else if a player hasn't spawned
Spawn Player
int GetRandomNumber()
Return number between 0 and 100
+AddScore (int amountToAdd)
score += amountToAdd
+AddScore (int amountToAdd, int multiplier)
score += (amountToAdd * multiplier)
int GetCurrentScore()
Return score

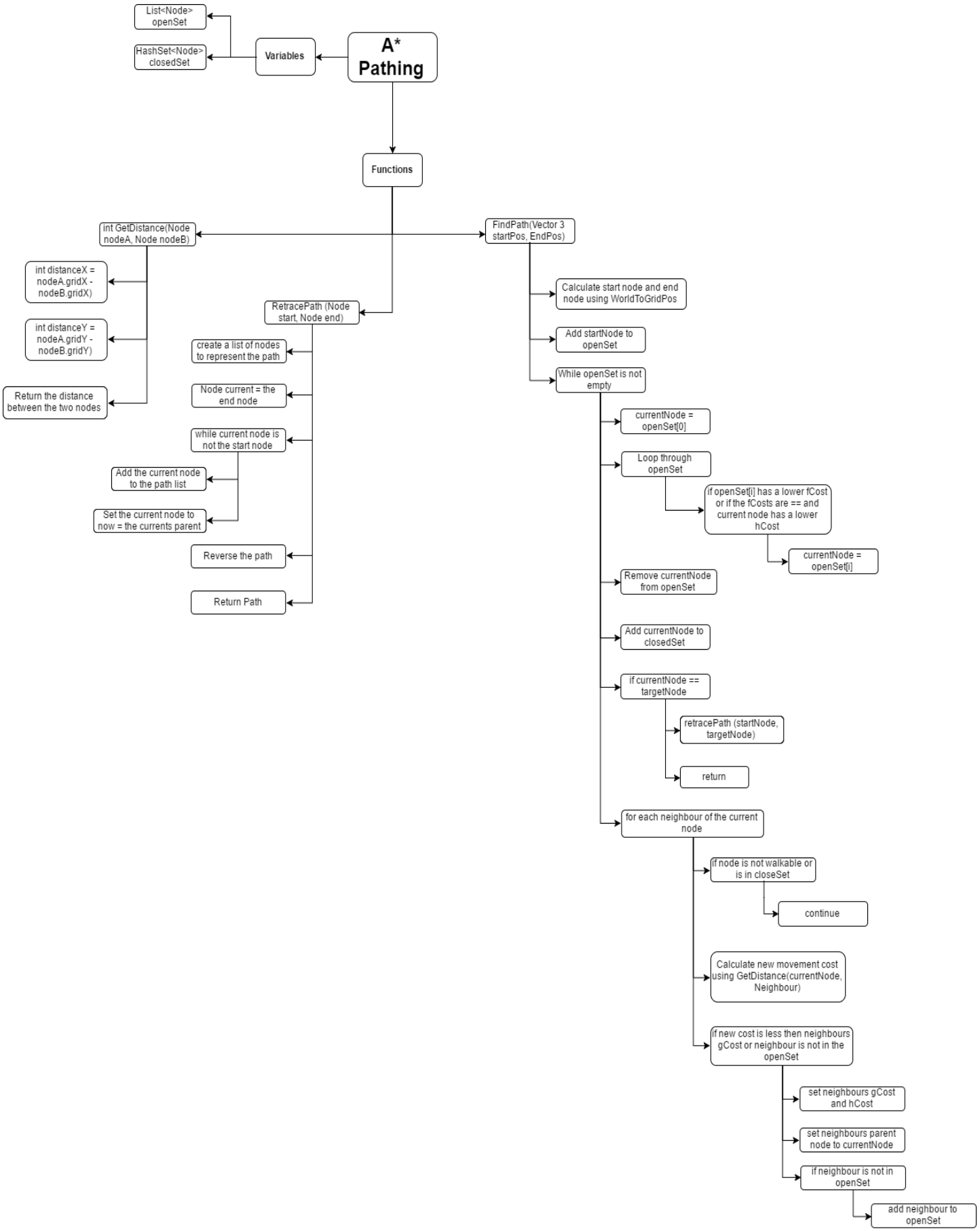

Cellular Automata
Variables
+ probabilty : int
+gridWidth : int
+gridHeight : int
+grid[,] : int
Functions
GetNeighbourCount()
int neighbourCount = 0
loop through surrounding X tiles
loop through surrounding Y tiles
if grid[x,y] == 1
neighbourCount ++
return neighbourCount
GenerateGrid()
Loop through grid x
Loop through grid y
int randNum (0-100)
if randNum <= probabilty
grid [x,y] = 0
else
grid [x,y] = 1
SmoothGrid()
loop grid x
loop grid y
if GetNeighbourCount (x,y) > 4
grid [x,y] = 1
else
grid [x,y] = 0

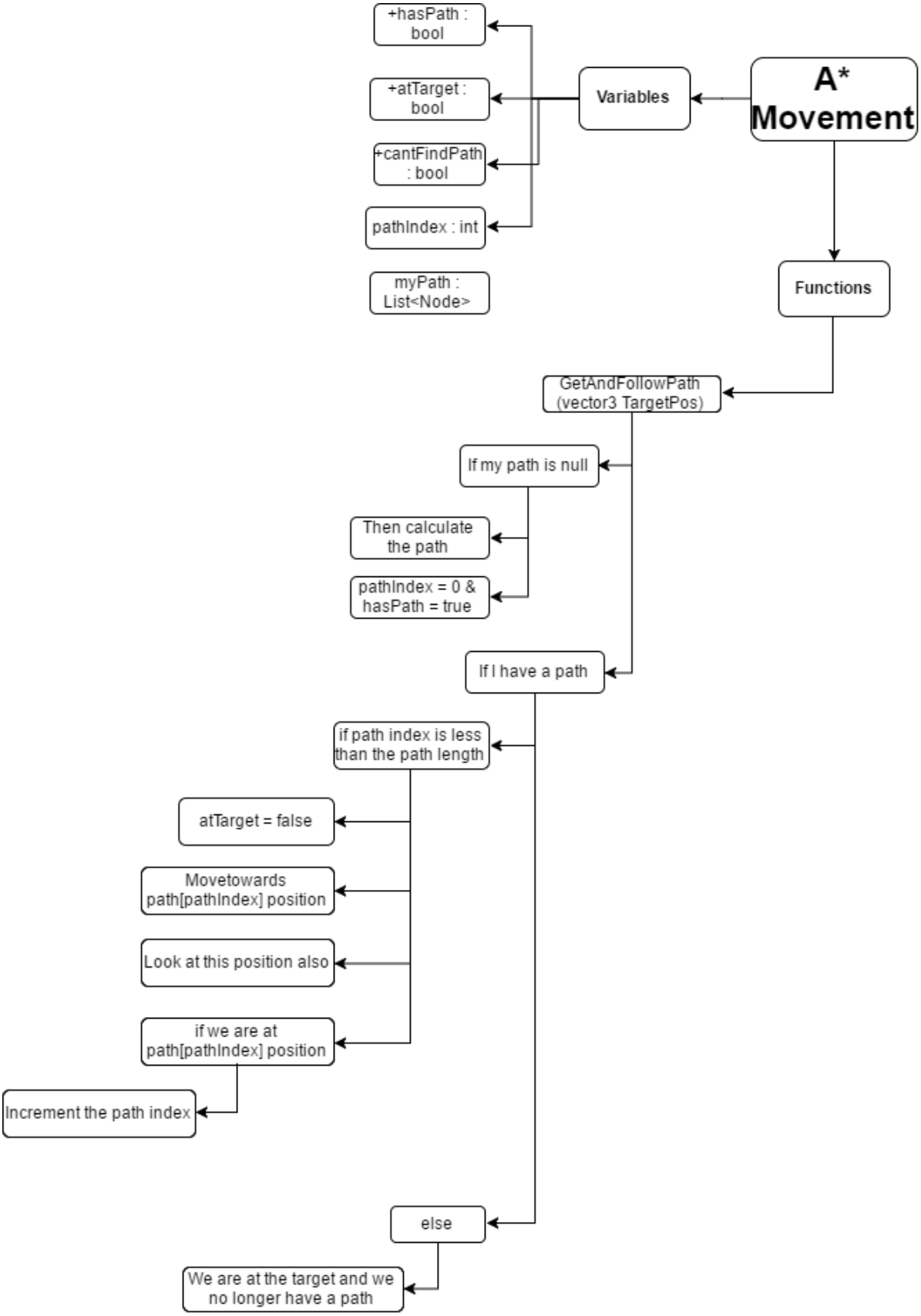
Marching Squares

- Variables
  - List <Vector3> verticies
  - List <int> trangles
  - +SquareGrid squareGrid
  - +bool Lava
- Classes
  - +SquareGrid
    - +Squares[,] squares
    - float MapWidth = width of array * squareSize
    - float MapHeight = height of array * SquareSize
    - Constructor (int[,] map, float squareSize, bool lava)
      - create 2D array of control nodes with size of the map
        - loop through map x & y
          - float yPos = randomRange (0f, 0.25f)
          - create a vector3 for the nodes position
          - bool active = false
          - if (lava = false & map[x,y] == 0)
            - controlNodes[x,y] = new ControlNode (nodePosition, active, squareSize)
            - active = true
          - else if (lava = true & map[x,y] == 1)
            - active = true
        - Initialize the squares array, remember there is always 1 less square then there is node
          - Loop through x and y of the map
            - squares [x,y] = new square (pass in the relative control nodes from the controlNodes array)
  - +Node
    - +Vector3 position
    - +int vertexIndex
    - Constructor
      - Sets the position
  - +ControlNode: Node
    - +bool active?
    - +Node above, right
    - Constructor
      - Set if node is active (wall = active)
      - Set above and right nodes
  - +Square
    - +ControlNode topLeft, topRight, botRight, botLeft
    - +Node centreTop, centreRight, centreBot, centreLeft
    - +int config
    - Constructor
      - Set the ControlNodes
      - Set the Nodes based on the ControlNodes
- Functions
  - GenerateMesh (int[,] map, float squareSize)
    - create a new sqaureGrid
    - loop through map x
      - loop through map y
        - TriangulateSquare (squareGrid.sqaures[x,y]
    - Create new mesh
    - Get the mesh filter
    - Set the mesh filter mesh to created mesh
    - Set the vertices and triangles on the mesh
  - TriangulateSquare (Square sqaure)
    - Switch through all config conditions
      - MeshFromPoints() depending on config values
  - CreateMeshFromPoints (Nodes[] points)
    - AssignVertices(points)
      - if there are 3 points
        - CreateTriangle(0,1,2)
      - if there are 4 points
        - CreateTriangle(0,2,3)
      - if there are 5 points
        - CreateTriangle(0,3,4)
      - if there are 6 points
        - CreateTriangle(0,4,5)
    - AssignVertices (Node[] points)
      - Loop through the points
        - if point does not already have a vertex index
          - This points vertexIndex = verticesList.Count
    - CreateTriangle (Node a,b,c)
      - Add a,b,c to the triangles list


+worldPosition : Vector 3
+walkable : bool
+gridPosX : int
+gridPosY : int
+gCost : int
+hCost : int
+fCost : int
+parentNode : Node
Variables
A*Node
Constructor
Sets if walkable and sets world position and grid positions
Functions
CalculateFCost()
return hCost + gCost


A*Grid
Variables
-gridSizeX, gridSizeY : int
-nodeDiameter : float
+ nodeRadius : float
+grid: A*Node [,]
+gridWorldSizeX : int
+gridWorldSizeY : int
+walkableMask : LayerMask
Functions
Node WorldToGridPos(Vector 3 worldPos)
float gridPercentX
float gridPercentY
int xPosition = (gridSizeX-1) * gridPercentX
int yPosition = (gridSizeY-1) * gridPercentY
Return grid [xPosition, yPosition]
List<Node> FindNeighbours (Node node)
List<Node> neighbours
Loop through neighbours
skip self
check if inside grid array
add to neighbours list
return neighbours
Start()
Set gridSizeX & Y = gridWorldSizeX/Y / nodeDiameter
CreateGrid()
CreateGrid(A*Node [,] grid)
Vector 3 worldBotLeft
grid = new grid[gridSizeX, gridSizeY]
Loop through grid x
Loop through grid y
vector 3 worldPosition
check if collision at position
grid [x,y] = new Node (collisionCheckResult, worldPosition, x, y)


A* Pathing
Variables
List<Node> openSet
HashSet<Node> closedSet
Functions
int GetDistance(Node nodeA, Node nodeB)
int distanceX = nodeA.gridX - nodeB.gridX)
int distanceY = nodeA.gridY - nodeB.gridY)
Return the distance between the two nodes
RetracePath (Node start, Node end)
create a list of nodes to represent the path
Node current = the end node
while current node is not the start node
Add the current node to the path list
Set the current node to now = the currents parent
Reverse the path
Return Path
FindPath(Vector 3 startPos, EndPos)
Calculate start node and end node using WorldToGridPos
Add startNode to openSet
While openSet is not empty
currentNode = openSet[0]
Loop through openSet
if openSet[i] has a lower fCost or if the fCosts are == and current node has a lower hCost
currentNode = openSet[i]
Remove currentNode from openSet
Add currentNode to closedSet
if currentNode == targetNode
retracePath (startNode, targetNode)
return
for each neighbour of the current node
if node is not walkable or is in closeSet
continue
Calculate new movement cost using GetDistance(currentNode, Neighbour)
if new cost is less then neighbours gCost or neighbour is not in the openSet
set neighbours gCost and hCost
set neighbours parent node to currentNode
if neighbour is not in openSet
add neighbour to openSet


A* Movement
Variables
+hasPath : bool
+atTarget : bool
+cantFindPath : bool
pathIndex : int
myPath : List<Node>
Functions
GetAndFollowPath (vector3 TargetPos)
If my path is null
Then calculate the path
pathIndex = 0 & hasPath = true
If I have a path
if path index is less than the path length
atTarget = false
Movetowards path[pathIndex] position
Look at this position also
if we are at path[pathIndex] position
Increment the path index
else
We are at the target and we no longer have a path

Error Reports

#	Date	Test Case/ Functionality	Error	Action To fix	Fixed (Y/N)	Date
1	25/04/17	Testing Cellular Automata	Null Reference Exception of grid array	Give the grid array a size	Y	25/04/17
2	25/04/17	Development of Marching Squares	Null Reference exception of vertices list	Initialize the list	Y	25/04/17
3	25/04/17	Development of Marching Squares	Null Reference Exception of triangles list	Initialize the list	Y	25/04/17
4	25/04/17	Development of Marching Squares	Incorrect vertex placement for triangle configurations 9 and 12, resulting in twisting.	Read through my triangle configurations.	Y	25/04/17
5	05/05/17	Developing A* grid of nodes	Incorrect position of node on the z axis	Was using y position instead of z position	Y	11/05/17
6	11/05/17	Developing A* pathing	Null reference exception of A*Node grid	Created two grid variables by mistake, removed one to fix	Y	11/05/17

7	11/05/17	Developing A* pathing	Incorrect looping through neighbours	Had less than for loop limit instead of less than or equal to.	Y	14/05/17
8	11/05/17	Developing A* Pathing	Added wrong node to neighbours list	Was adding the same node several times instead of the node at grid[x,y]	Y	14/05/17
9	11/05/17	Developing A* Pathing	Path going incorrect direction	Was not using MathF.Abs	Y	14/05/17
10	14/05/17	Developing A* Movement	Players path not resetting	Cleared the path list when the player reached their target.	Y	14/05/17
11	14/05/17	Developing A* Movement	Players pathIndex not resetting	Made the path index variable set back to 0 when the player tried to find a new path	Y	14/05/17
12	17/05/17	Making player follow path	CantFindPath variable not resetting when new target	When the player tries to find the path again this bool will set to false	Y	17/05/17

13	17/05/17	Testing	Script execution order. Certain events would fire before an event that it relied on had to happen	Changing the script execution order fixed this.	Y	17/05/17
14	17/05/17	Testing	Using the cellular automata grid to create the a star walkable areas resulted in the player being able to walk through certain walls.	Did a collision check instead, this gave more accurate results	Y	17/05/17
15	19/05/17	Testing the Game controller spawning	Player and ladder spawning in the wall.	Checked to see if the area they were trying to spawn was walkable	Y	19/05/17
16	19/05/17	General Testing	The player would be saying it was at target and reloading level before walking to the ladder.	This was happening because the at target bool did not change back to false quick enough. Making the load level if statement and if-else solved this.	Y	19/05/17

Improvement Report

If this project were to go beyond the prototype type stage there are a number of improvements that I would like to make. The first being the leaderboard, currently the leader board is local and it resets each time the scene is loaded. I will like to change the leader board into an online leader board, this will result in me having to create an SQL database, doing a tiny bit of PHP and changing the code of the in-game leaderboard to read the text from a web page. I would have liked to include this online leader board in the prototype but I needed a reason to use a LINQ Query and therefore I have not included it.

There are also a lot of visual changes that would need to be made to the game, for example, I would need the models for each character and object in the scene, have an animation for the characters moving and attack as well has to have a nice user interface for the user to interact with. Those items were out of the scope of this prototype and that is why they have not been included in this build.

There are many gameplay additions to be made to the game, currently, the user does not do anything. As this is an idle game, similar to that of *Non-Stop Knight,* the user does not do very much anyway, however, I would like to add in different attacks and abilities that the user could select, doing this gives the user some level of control over their character. Currently, the enemies are static, they do not move, I would like the enemies to be able to move around the area. Also, currently the enemies do not attack the player, having the enemies attack is essential to the gameplay, making it something that defiantly needs to be added if this game was to continue.

The prototype of this game was developed for PC, however, mobile is my preferred platform for this product. Based on my experience in developing mobile games there are currently no changes that need to be made to get this product working on mobile devices, I would just need to install the android SDK to my current version of Unity and everything should run perfectly.

Conclusion

In conclusion of this assignment, I have successfully implemented A* pathfinding algorithms, Cellular automata and Marching Squares procedural generation. This assignment has shown me the importance of carefully taking the time to plan my code instead of jumping head first into the problem. The planning that I did made it extremely easy for me to sit down and wrote the code when I got to that stage in development. In the future, I will be trying my best to implement this method before starting any programming. I will keep working on this project after I have submitted and will add in the necessary improvements. I have plans to release the final product on mobile, by doing this I can easily show people in job interviews a game that I have created using advanced programming techniques. Mobile is also the perfect platform for an idle game like mine as people can easily play it many different situations.

References

Microsoft. (2017a). List(T) Class. Retrieved May 20, 2017, from https://msdn.microsoft.com/en-us/library/6sh2ey19(v=vs.110).aspx

Microsoft. (2017b). Math.Abs. Retrieved May 20, 2017, from https://msdn.microsoft.com/en-us/library/dk4666yx(v=vs.110).aspx

Stack Overflow. (2014). HashSet vs. List performance. Retrieved May 20, 2017, from http://stackoverflow.com/questions/150750/hashset-vs-list-performance

Unity. (2017a). Scripting API: Quaternion.identity. Retrieved May 20, 2017, from https://docs.unity3d.com/ScriptReference/Quaternion-identity.html

Unity. (2017b). Vector3.Lerp. Retrieved May 20, 2017, from https://docs.unity3d.com/ScriptReference/Vector3.Lerp.html

Image References

Figure 1 - FlareGames. (2017). Nonstop Knight - Android Apps on Google Play. Retrieved May 21, 2017, from https://play.google.com/store/apps/details?id=com.koplagames.kopla01&hl=en

Figure 2 - Polytech. (n.d.). *Marching Squares Configurations*. Retrieved from http://users.polytech.unice.fr/~lingrand/MarchingCubes/algo.html