Implementation of Probabilistic Smart Terrain in Unity

by

Joseph Korchnak

Submitted in Partial Fulfillment of the Requirements
for the Degree of
Master of Computing and Information Systems

YOUNGSTOWN STATE UNIVERSITY

December 2017
Implementation of Probabilistic Smart Terrain in Unity

Joseph Korchnak

I hereby release this thesis to the public. I understand that this thesis will be made available from the OhioLINK ETD Center and the Maag Library Circulation Desk for public access. I also authorize the University or other individuals to make copies of this thesis as needed for scholarly research.

Signature:

__________________________________________________________________
Joseph Korchnak, Student Date

Approvals:

__________________________________________________________________
John Sullins, Thesis Advisor Date

__________________________________________________________________
Abdu Arslanyilmaz, Committee Member Date

__________________________________________________________________
Alina Lazar, Committee Member Date

__________________________________________________________________
Sal Sanders, Dean of Graduate Studies Date
Abstract

Smart Objects [1] from games like The Sims are used to create Smart Terrain. Smart Terrain allows for the terrain to hold the variables a non-player character (NPC) requires to decide where to move. Probabilistic Smart Terrain [2] builds on Smart Terrain by adding the probability that given Smart Objects may not meet a need 100% of the time. This paper will cover the implementation of the Probabilistic Smart Terrain algorithm in a Unity 2D game. The game is tile based and operates in real-time. A map flooding algorithm was designed using Raycast objects within the Unity environment. A playable game was created in which the Probabilistic Smart Terrain algorithm is used to control a NPC that patrols the area guarding three treasure objects. The terrain holds the distances to each coin while the Probabilistic Smart Terrain algorithm is used to calculate the probability of where to find the player. This causes the NPC to move to the most probable direction of finding the player. This implementation will allow actual players to evaluate the apparent intelligence of the Probabilistic Smart Terrain algorithm.
Acknowledgements

Thank you Dr. Sullins for the opportunity to work on this project. Thank you to all the YSU CSIS faculty for all the knowledge you shared. Finally thank you to the best wife and friend for putting up with my stress about each class and all the YSU faculty.
# TABLE OF CONTENTS

LIST OF FIGURES .............................................................................................................. VII

CHAPTER 1  INTRODUCTION.............................................................................................. 1

1.1 Motivation...................................................................................................................... 1
1.2 Contributions................................................................................................................. 2
1.3 Research Questions......................................................................................................... 2
1.4 Organization..................................................................................................................... 2

CHAPTER 2  BACKGROUND AND RELATED WORK.................................................... 3

2.1 Smart Terrain ............................................................................................................... 3
2.2 Probabilistic Smart Terrain ........................................................................................... 5
2.3 Raycast.......................................................................................................................... 8
2.4 Map Flooding.................................................................................................................. 8
2.5 Unity.............................................................................................................................. 10

CHAPTER 3  IMPLEMENTATION IN A UNITY 2D GAME............................................ 12

3.1 Defining the original game .......................................................................................... 12
3.1.1 Modifications to the Game..................................................................................... 13
3.2 Map Flooding.................................................................................................................. 13
3.3 Representing guard decisions with probabilities ......................................................... 18

CHAPTER 4  GUARD EVALUATION.................................................................................. 20
LIST OF FIGURES

Figure 1: Signal Transmission (Sullins 2010) ................................................................. 4
Figure 2: Empty Refrigerator .......................................................................................... 5
Figure 3: Plausibility Benchmarks 1 .............................................................................. 6
Figure 4: Plausibility Benchmarks 2 .............................................................................. 7
Figure 5: Plausibility Benchmarks 3 .............................................................................. 7
Figure 6: Map Flooding ................................................................................................. 10
Figure 7: Initial Tile ........................................................................................................ 14
Figure 8: Tile List 1 ....................................................................................................... 14
Figure 9: Raycast 1 ....................................................................................................... 15
Figure 10: After Raycast 1 ........................................................................................... 15
Figure 11: Tile List 2 ..................................................................................................... 16
Figure 12: Raycast 2 ..................................................................................................... 16
Figure 13: After Raycast 2 ........................................................................................... 17
Figure 14: Tile List 3 ..................................................................................................... 17
Figure 15: Finished List ................................................................................................. 17
Figure 16: Guard Test 1 ................................................................................................ 21
Figure 17: Guard Test 2 ............................................................................................... 22
CHAPTER 1
INTRODUCTION

1.1 Motivation

Artificial Intelligence (AI) in gaming is the part of a game that connects the player to the game. The AI of the game must appear intelligent within the set rules of the game. The player needs to see realistic actions by Agents, the entities in the game. Agents are built to make the player feel like they are part of the game, as if they are facing a real enemy. The game type, rules, level layout, and any other part of the game need to be considered when building an AI. The hardware an AI will run on and the memory size allotted to the game AI also play a part in the programming process. [3] Originally games had very simple, if any, AI. The AI either follow a set pattern or react in a certain way to players’ input in the same manner. Currently, AI is getting a large amount of attention. The speed and size of games has increased tremendously. The first aspect game designers focused on was the graphics. Building more and more realistic looking games to try in engage the player. Now, graphics are reaching a pinnacle and are relatively the same level between major games. Games designers are now focusing on their game AIs to further engage the players. This paper builds on Probabilistic Smart Terrain [2] by implementing the algorithms discussed in the paper into a dynamic game. The game is built to play in real-time to test the algorithm under dynamic conditions. The game was originally a tutorial for Unity called 2D Rougelike. I modified it by changing it to real time and adding the Probabilistic Smart Terrain to the AI guards to patrol the treasure objects.
1.2 Contributions

One contribution of this thesis is the base test game with functioning enemies, which can either be extended by other researches or used by real players to evaluate the apparent intelligence of the algorithm. The map flooding algorithm introduced in this thesis is another contribution. The flooding algorithm uses Raycast objects to build a list of surrounding tiles of the tile list itself. Depending on where the coin is place, the distance will be calculated from that tile.

Unity is the game IDE used to build the game. The map flooding algorithm is written in C# as well as all the other scripts used in the game. The core assets are from a free tutorial on Unity’s official page [4]. The game displays all the information used in the Probabilistic Smart Terrain algorithm allowing a game designer to change parameters to better fit their needs. This also allows for changes to be made while the game is running and testing.

1.3 Research Questions

How to best implement the Probabilistic Smart Terrain in a Unity 2D game. Leading to testing the game AI with players to find out how to make the AI act realistically.

1.4 Organization

The organization of this thesis begins with Section 1: Introduction, which gives an overview of the thesis. Section 2: Background and Related Work, which explain the algorithm and its predecessor as well as explain some key components used in the creation of the game. Section 3: Implementation in an Unity 2D game, which explains how the
algorithm and map flooding was used to create the game. Section 4: Guard Evaluation, this section gives the observable behaviors that are considered believable as well as the guards actual paths. Section 5: Future Work, this is a look at possible game design changes and other work that can help to further the research.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Smart Terrain

“Smart Terrain” is an idea that Will Wright implemented in the AI for the game The Sims. The autonomous characters, or Sim, in the game need to satisfy certain needs that model human needs. “Smart objects” are placed in the Sim’s house that can satisfy the 8 basic needs the Sims have. [5] One of the basic needs is hunger. If the Sim becomes hungry it will need to seek out an object to satisfy their hunger. A refrigerator can be placed within the Sim’s house to satisfy that need. The adding and mixing of multiple objects that meet all the different needs of the Sim build up what is the “Smart Terrain”.

“Smart Terrain” sets itself apart by moving AI from the Sim and placing it within the terrain. Instead of the Sim deciding which need to satisfy and then picking where to go; the Sim follows the signals from the terrain to the place that will fulfill its highest need. Taking the AI away from the Sim allowed the creators to release expansions easily by adding new objects that meet needs. [1]
“Smart Terrain” does have some shortcomings. While calculating which object to move to, the Sim only uses one item to go to. This contrasts with moving towards multiple items that would increase their happiness more than the single item would. The creators did want to leave the Sims partly dumb to give a need for the player to help the Sim survive. [1] Also, the items in the game produced a signal that they would meet that need without fail. An example would be an empty refrigerator, which does not meet the hunger need.
2.2 Probabilistic Smart Terrain

Probabilistic Smart Terrain builds onto smart terrain by introducing new calculations that objects may or may not satisfy a need. This idea is meant to increase the realism of the actions performed by an autonomous character within a game. [2] The following scenarios will help to explain the difference between the two ideas.

Smart Terrain character is hungry. The character will move towards the closest refrigerator to satisfy its need. Any refrigerator in the area will satisfy the hunger need so there is no need to find anything except the closest one. Probabilistic Smart Terrain adds the idea that a refrigerator may not satisfy that need. The character will then need to calculate which way is best to find food, based both on both distances and probabilities.

The idea is to make the non-player character act as a player would, to be perceived as intelligent. Some examples include: heading to a higher probably object that is at the
same distance as a lower probably object, or moving towards a closer object even if it has a slightly smaller chance of meeting the need then moving to a farther object with a higher chance to meet the need. These would be the plausible behaviors a player would expect.

<table>
<thead>
<tr>
<th>Refrigerator with a 50% probability</th>
<th>The Non-player character (NPC)</th>
<th>Move this way</th>
<th>Refrigerator with an 80% probability</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Refrigerator with a 70% probability</th>
<th>Move this way</th>
<th>The Non-player character (NPC)</th>
<th>Refrigerator with a 75% probability</th>
</tr>
</thead>
</table>

**Figure 3: Plausibility Benchmarks 1**

Another scenario to examine is when multiple objects with a slightly lower probably at the same distance as an object with a higher probability. The NPC should move towards the multiple objects. An exception would be if an object has a 100% chance of meeting the need. No matter how high the multiple objects’ probabilities are, if they are not 100%, the player will move to the object that will fill its need.
The final test is when the NPC is moving towards an object that can meet its need, it will check lower probability objects that are reasonably close to the path it is traveling.
Once an NPC reaches a tile, it learns if that Probabilistic Smart Object can meet its need or not. That is, the probability of that tile satisfying the need should either become 100% or 0%. The NPC however, should become less certain over time, as the character does not examine it. Over time the probability of that object being able to meet the NPCs need will return to its original value. Using the refrigerator example this would simulate the refrigerator being restocked and there being a chance that the object can satisfy the characters need.

2.3 Raycast

The Raycast function allows you to test for objects within a set area. AVector2 is used as the origin of the Raycast. The Raycast type used for this implementation was a Linecast. The Linecast needs a second point to determine where the Raycast will end. The Linecast moves along a straight line until it either reaches the end point or runs into another object. The Linecast can be set to ignore objects allowing you to look for specific objects with the path. The Raycast function returns a RaycastHit object that is used to access the variables of the object hit by the Raycast. If the Raycast does not find an object, the RaycastHit object will return as null.

2.4 Map Flooding

The Probabilistic Smart Terrain algorithm needs to know the distance to the objects as well as their probability to satisfy the NPCs need. Map flooding allows you to precompute the distances to objects. The basic idea behind map flooding is that every tile
or square is 1 distance from the object. Starting with the object and spreading out adding 1
distance at each tile. This algorithm can also be used to calculate distances around walls
and objects.
The best point about map flooding is it simple and straightforward. The distances can be precomputed prior to gameplay.

2.5 Unity

Unity is a game engine made by Unity Technologies. Unity was first released in 2005 for OS X. It has grown to include 27 different platforms. Unity has progressed through six major version changes with version 2017.1 being the latest version at the time this was
written. Unity uses the entity-component-system architectural pattern. This allows for each component in the game’s scene to be an entity and each entity to have its own properties, or components, for interacting with the scene. The properties can be changed at runtime to alter each entities’ behaviors and functionality.

Unity was used to create a game that would test the algorithm’s ability in a real-time game. The algorithm was added to a modified version of 2D Rougelike [4], a game that was supplied as a tutorial to new users of the Unity platform. 2D Rougelike is a tile-turn based game where the player tries to survive as long as they can. Each move decreases your health by one point. Objects such as food and drink will replenish your health. There are also zombies that will try to attack you as you move through the board.

(Unity versions used to alter this game were 5.4.1f1, 5.5.0f3 and 2017.1.0f3.)
CHAPTER 3
Implementation in a Unity 2D game

Dr. John R. Sullins’ original tests were done in a step by step program written in Java where each step moved the Sim one tile at a time. It was a very primitive user interface more meant to alpha test the algorithm instead of providing an actual playable game. The game proposed by Dr. Sullins that used this Probabilistic Smart Terrain put the player in a role of a thief. The player would attempt to steal treasure from an area and avoid a guard that is patrolling the area. The guard would use the Probabilistic Smart Terrain to decide which way to move. Each treasure has a different value, with the higher value objects having a greater probability that the player would be in that area trying to steal the treasure. The Probabilistic Smart Terrain is used to give a believable patrol path to the guard NPC, a path that is not as predictable as the guard checking the treasures in a set order but instead focusing more on the higher value treasures.

3.1 Defining the original game

2D Rougelike is a tutorial for Unity [4]. The assets and the original code are used to help individuals learn to code in Unity. The finished 2D Rougelike game is a turn-based game where you try to avoid Zombies while picking up food items. Each time you move you lose one from your health and if you reach 0, that ends the game. Each level gets progressively harder with more enemies. To move on to the next stage you need to reach the exit on your current stage.
3.1.1 Modifications to the Game

The modifications I made turn the game into a real-time game where the zombie guard moves at a slightly slower pace than the player. The guard patrols between three treasures using the Probabilistic Smart Terrain algorithm. The algorithm calculates where the most likely place the player will be based on the treasures’ value and how long since the treasure has been checked. The guard also considers how far away the treasure is to make the most efficient path to check all the treasure objects. The wall and treasure placement are set for testing purposes but can be randomized.

The game for this research has one zombie guard and three treasures that the player is to collect. Once the player collects all the treasures they must then make it to the exit to win the current board. The functionality of the game is restricted to observing the guards patrol path. The game is played from the development mode of Unity and you can change each of the variables during game play to adjust the patrol route the guard will make. The game is for testing the believability of the guard’s patrol path using the Probabilistic Smart Terrain algorithm.

3.2 Map Flooding

Map flooding was used to establish the distances to each treasure. The flooding starts from where the treasure object is placed on the map by putting the tile the object is placed on within a list of tile objects. The tile objects store the distance to each of the treasure objects. Each tile in the list of tile objects then uses a RaycastHit2D array to collect all the surrounding tiles without walls. The next step is to check if any tiles with the array
have duplicates already in the tile list. If no duplicate is found, the tile is added to the list. The list is then traversed and if the distances of the surrounding tiles are less than the current tile's distance to the treasure, then the current tile sets its distance to the treasure to one plus that surrounding tile’s distance. The final step checks all tiles in the list versus all surrounding tiles and moves the finished tiles, that is all tiles surrounding it have less than or equal to one plus its own distance, to a finished list.

The following set of figures illustrate this process.
Raycast out from tiles in the list

![Raycast Diagram](image)

**Figure 9: Raycast 1**

After checking for null tiles and duplicates add the tiles to the list.

![Tiles Diagram](image)

**Figure 10: After Raycast 1**
Check surrounding tiles of all tiles in the list for lower distances and change those that are needed. The non-treasure tiles in this case change their distance after checking the treasures tiles distance.
Check surrounding tiles of all tiles to remove finished tiles

Since all the tiles surrounding the treasure tile are less than or equal to its distance plus one, it is removed from the list.

Current list

<table>
<thead>
<tr>
<th>Treasure</th>
<th>Tile</th>
<th>Tile</th>
<th>Tile</th>
<th>Tile</th>
</tr>
</thead>
</table>

Finished List

<table>
<thead>
<tr>
<th>Treasure Tile</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

Figure 13: After Raycast 2

Figure 14: Tile List 3

Figure 15: Finished List
Once the map flooding algorithm has completed for all three coins the game can start.

3.3 Representing guard decisions with probabilities

Game AI does not need to be intelligent AI. The main concept behind video game AI is believability. Games that have an opponent need an AI that performs in a believable way to engage players. If a game’s AI is too easy the players can perceive it as dumb and become bored with the game. On the other hand, if the AI is too hard the player may perceive it as cheating and become irritated and give up playing. The game AI needs to be hard but beatable. A player needs to believe that they may lose, but can overcome the AI by using their intelligence.

The guards are built to patrol the treasure in a believable manner using the Probabilistic Smart Terrain algorithm. By using the algorithm to set probabilities for special points on the map that contain the treasures, the guard will patrol those areas in search of the player based on their values and distances from the guard. The main areas that the guards in this implementation patrolled were the three different treasures. Each treasure had a value attached to it with the higher value being more likely for a player to be in that area. The guard also considers the amount of time that has passed since he last visited a treasure. Using the follow equation [2], a modified version of the Probabilistic Smart Terrain, the guard adds up the probability before each movement.

Original Equation

\[ \pi_i = \pi_i + a_i + t \]

- \( \pi_i \) is the original probability that object \( i \) meets the need, as defined by the level designer.
- \( a_i \) is the modified probability (either 0 or 1) created once the object was explored.
- \( t \) is the number of time units since the character last explored the object.
• \( k_i \) is the expected time to knowledge expiration for object \( i \).

\[
B(p_i, a_i, K_i, t) = a_i - (a_i - p_i) / (1 + e^{-t + K_i})
\]

Modified equation

```java
private double CalculateTileWeights(int tile)
{
    //Smart Terrain Equation
    double answer =
    (0 - (0 - ValueOfGold) / (1 + Mathf.Exp ((float)(-TimeGold + RemberGold))) / GoldArray[tile] +
    (0 - (0 - ValueOfSilver) / (1 + Mathf.Exp ((float)(-TimeSilver + RemberSilver))) / SilverArray [tile] +
    (0 - (0 - ValueOfCopper) / (1 + Mathf.Exp ((float)(-TimeCopper + RemberCopper))) / CopperArray [tile];

    return answer;
}
```

The guard value of \( a_i \) is set to 0 for patrol measures. The guard continuously searches for the player even after reaching the treasures. When the player is found, the guard would switch to a chase function to pursue and attempt to capture them. The \( p_i \) variable is set as the value of the treasure object the guard is patrolling. The higher the value the more the guard is going to patrol that treasure. The \( t \) variable is a simple counting variable to return the probability that increases at a set interval. The \( k_i \) variable is set to a certain amount to effect how long between the times the guard checks a treasure. The divisor of each of the equations is the distance to each treasure. This gives additional weight to the treasures that are closer.
CHAPTER 4

Guard Evaluation

4.1 Observable Behaviors

Artificial intelligence algorithms used for gaming are generally evaluated in terms of how playable they make a game. That is, an NPC controlled by the game AI should make a player feel as though they are dealing with an intelligent opponent because they take actions that a player would consider “plausible” for that kind of NPC. Based on [2], scenarios for the following plausible behaviors were created in order to test the implementation:

- The path travelled by the guard should be based on the value of the treasures. Specifically, the guard should check the most valuable ones more often, as there is a greater probability the player will want to steal them.

- The path followed by the guard should be based on the treasures relative placement. Specifically, the guard should consider a group of objects more valuable than an individual object, as there is a strong probability the player will be in that area.

- The path followed by the guard should also be based on the distance of the treasures. Specifically, the guard should make a patrol route that crosses lower valued items if they are close to the path to a higher valued treasure.

- The guard should disregard the treasures and pursue the player if they are found.
4.2 Guard’s behavior

The initial path the guard takes is Gold > Silver > Gold. Gold is the highest valued item, followed by silver. So, the initial path checks Gold twice. Next the guard moves > Copper > Silver and back to Gold. With the guard needing to check the copper because it has been an extended period since it has last been there. Then checking the Silver before returning to the Gold because it is closer. This shows the plausible actions you would expect from a guard patrolling these points.
This layout is used to show that the guard will move to the Copper and Silver, which have a higher added probability, based on value and distance, before checking the Gold. The guard patrols those two treasure objects which align with the plausible actions for believability.
The guard first checks Gold then on his path to Silver he checks Copper. The same happens each subsequent patrol. This shows that even though the guard is being drawn to the next highest value object, the Silver coin, since the Copper coin is relatively close to his path he checks that treasure on the way to Silver. On his return to Gold he takes the fastest route to get there showing what would be the route between Gold and Silver if the Copper coin was not in the area. During the return to Gold, the Copper coin’s probability has very little effect on the path because of the amount of time since it was last checked.
CHAPTER 5

Future Work

The focus here is to further the research into the believability of the AI. The game itself needs to have the rest of the gameplay mechanics implemented but for future research here are different avenues to consider.

1. Have the guard AI evaluated by a group of participants that self-identify as gamers as well as a group that does not. This will give a larger range of feedback. The test should include the individuals predicting where they believe the guard should move next as well as an overall evaluation of how well they believe the guard acted.

2. Adding addition points of interest for the guard to patrol. The exit and other treasures can be added to increase the guards patrol path and options. Adding dynamic spots that the guard last saw the player at as well will allow for a more unpredictable path. The dynamic spots should return to a probability of zero over time so the guard does not continuously return to a spot the player was seen after a given period of time.

3. Adding additional guards. Multiple guards can bring in a new level of complexity. The initial addition should have guards move autonomously from one another until the come within a set distance of each other. At that point, they should share their probabilities, as in talking over what they have seen, and one guard should
retain the probabilities while the other sets the highest probability back to zero to continue the patrol.

4. Adding in the original features of Rougelike 2D. The original game allowed for the destruction of walls as well as randomly generated levels. The guard AI should be able to adapt to new paths as the player moves through the level. The initial setup could just re-flood the map at the point the player destroys the wall. This is cheating because the new probabilities that the guard will follow will change instantly to reflect the absence of the wall. This can be used to our advantage as the guard will move towards the new opening as if he heard the wall break. On the other hand, if the player was far away from the guard, the change in probabilities will be minimal until the guard gets closer to the new path. The best thing about game AI is that it does not need to be perfect, just believable. As the game grows, the changes that can be added are infinite and anything you can imagine can be added.
using UnityEngine;
using System.Collections;

public class Wall : MonoBehaviour
{
    public AudioClip chopSound1; // 1 of 2 audio clips that play when the wall is attacked by the player.
    public AudioClip chopSound2; // 2 of 2 audio clips that play when the wall is attacked by the player.
    public Sprite dmgSprite; // Alternate sprite to display after Wall has been attacked by player.
    public int hp = 3; // hit points for the wall.
    public LayerMask floorMask;

    private SpriteRenderer spriteRenderer; // Store a component reference to the attached SpriteRenderer.

    public Vector2 currentPos;

    void Awake()
    {
        // Get a component reference to the SpriteRenderer.
        spriteRenderer = GetComponent<SpriteRenderer>();
        // currentPos = new Vector2(transform.position.x, transform.position.y);
        setInitialTile(Physics2D.Linecast(new Vector2(transform.position.x, transform.position.y),
            new Vector2(transform.position.x, transform.position.y) + Vector2.up, floorMask).collider.gameObject);
    }

    void Start()
    {
    }

    public void setInitialTile(GameObject startTile)
    {
        startTile.GetComponent<Tile>()._hasWall = true;
        // startTile.GetComponent<BoxCollider2D>().enabled = false;
    }
//DamageWall is called when the player attacks a wall.
public void DamageWall (int loss)
{
    //Call the RandomizeSfx function of SoundManager to play one of
two chop sounds.
    SoundManager.instance.RandomizeSfx (chopSound1, chopSound2);

    //Set spriteRenderer to the damaged wall sprite.
    spriteRenderer.sprite = dmgSprite;

    //Subtract loss from hit point total.
    hp -= loss;

    //If hit points are less than or equal to zero:
    if(hp <= 0)
        //Disable the gameObject.
        gameObject.SetActive (false);
}

--Tile.cs--

using UnityEngine;
using System.Collections;

public class Tile : MonoBehaviour{

    public int distanceGold = 100, distanceSilver = 100, distanceCopper = 100;
    public bool _hasWall = false;

    public void setDistanceGold(int _distanceGold)
    {
        distanceGold = _distanceGold;
    }

    public void setDistanceSilver(int _distanceSilver)
    {
        distanceSilver = _distanceSilver;
    }

    public void setDistanceCopper(int _distanceCopper)
    {
        distanceCopper = _distanceCopper;
    }

    public int getDistanceGold()
    {
        return distanceGold;
    }
public int getDistanceSilver()
{
    return distanceSilver;
}

public int getDistanceCopper()
{
    return distanceCopper;
}

public bool hasWall()
{
    return _hasWall;
}

public void setWall()
{
    _hasWall = true;
}

//    public void setDistance (Transform goldPosition, Transform silver
Position, Transform copperPosition)
//    {
//        distanceGold =
//            (int) Mathf.Abs(goldPosition.position.x - transform.posit
//ion.x) +
//            (int) Mathf.Abs(goldPosition.position.y - transform.posit
//ion.y);
//        
//        distanceSilver =
//            (int) Mathf.Abs(silverPosition.position.x - transform.pos
//ition.x) +
//            (int) Mathf.Abs(silverPosition.position.y - transform.pos
//ition.y);
//        
//        distanceCopper =
//            (int) Mathf.Abs(copperPosition.position.x - transform.pos
//ition.x) +
//            (int) Mathf.Abs(copperPosition.position.y - transform.pos
//ition.y);
//        
//    }

    void Awake()
    {
        Transform goldPosition = GameObject.FindGame
ObjectWithTag("Gold").transform;
        Transform silverPosition = GameObject.FindGame
ObjectWithTag("Silver").transform;
//         Transform copperPosition = GameObject.FindGameObjectWithTag("Copper").transform;
//         setDistance (goldPosition, silverPosition, copperPosition);
//     } 
// }

--FloodTiles.cs--

using UnityEngine;
using System.Collections;
using System.Collections.Generic;
//Allows us to use Lists.

public class FloodTiles : MonoBehaviour {
    public List<GameObject> tileList = new List<GameObject>();
    public LayerMask floorLayer;
    public List<GameObject> finishedTiles = new List<GameObject>();

    public void Start()
    {
        setInitialTile (Physics2D.Linecast(new Vector2(transform.position.x, transform.position.y),
            new Vector2(transform.position.x, transform.position.y) + Vector2.up, floorLayer).collider.gameObject);
    }

    public void setInitialTile(GameObject startTile)
    {
        tileList.Add (startTile);
        if (CompareTag ("Gold"))
        {
            startTile.GetComponent<Tile> ().distanceGold = 1;
        }
        if (CompareTag ("Silver"))
        {
            startTile.GetComponent<Tile> ().distanceSilver = 1;
        }
        if (CompareTag ("Copper"))
        {
            startTile.GetComponent<Tile> ().distanceCopper = 1;
        }
        floodTiles ();
    }

    public void floodTiles()
    {
        int testExitSentinel = 0;
    }
//start the loop until no more tiles are left
while (tileList.Count != 0 && testExitSentinel != 100) {
    testExitSentinel++;

    //Get the count and valid surrounding tiles
    int tileCount = 0;

    //Add the surrounding tiles to the list
    int currentTileAmount = tileList.Count;
    for (int i = 0; i < currentTileAmount; i++) {
        RaycastHit2D[] surroundingTiles = new RaycastHit2D[4];
        //Get the surrounding tiles
        tileCount = getSurroundingTiles(surroundingTiles, tileList[i]);

        //Add the tiles to the list
        for (int j = 0; j < 4; j++) {
            if (surroundingTiles[j].collider != null) {
                //Tile to be added
                bool addTile = true;
                //Check for duplicates
                foreach (GameObject tileToCheck in tileList) {
                    if (surroundingTiles[j].collider.gameObject == tileToCheck) {
                        addTile = false;
                    }
                    if (surroundingTiles[j].collider.GetComponent<Tile>()._hasWall) {
                        addTile = false;
                    }
                }

                if (addTile) {
                    tileList.Add(surroundingTiles[j].collider.gameObject);
                }
            }
        }
    }

    //Check for finished tiles in finished Tile List
}

//Check for finished tiles in the list
foreach (GameObject finishedTile in finishedTiles) {
    for (int j = 0; j < 4; j++) {
if (surroundingTiles[j].collider.gameObject == finishedTile) {
    removeTile(surroundingTiles[j].collider.gameObject);
    currentTileAmount = tileList.Count;
}
}
}*/

} //Get the surrounding tiles of each tile and check distance
currentTileAmount = tileList.Count;
for(int i = 0; i < currentTileAmount; i++)
{
    RaycastHit2D[] surroundingTiles = new RaycastHit2D[4];
    Debug.Log("In Flood to check distances");
    tileCount = getSurroundingTiles(surroundingTiles, tileList[i]);
    if (gameObject.CompareTag("Gold")) {
        setDistance(tileList[i], surroundingTiles, 'g');
    }
    if (gameObject.CompareTag("Silver")) {
        setDistance(tileList[i], surroundingTiles, 's');
    }
    if (gameObject.CompareTag("Copper")) {
        setDistance(tileList[i], surroundingTiles, 'c');
    }
}
// Check to remove tiles
currentTileAmount = tileList.Count;

for(int i = 0; i < currentTileAmount; i++)
{
    // Get the surrounding tiles for each tile in the list
    RaycastHit2D[] surroundingTiles = new RaycastHit2D[4];
    getSurroundingTiles(surroundingTiles, tileList[i]);
    char type = ' ';
    int currentDistance = 0;

    if (gameObject.CompareTag("Gold")) {
        type = 'g';
        currentDistance = tileList[i].GetComponent<Tile>().distanceGold;
    }
    if (gameObject.CompareTag("Silver")) {
        currentDistance = tileList[i].GetComponent<Tile>().distanceSilver;
        type = 's';
    }
    if (gameObject.CompareTag("Copper")) {
        currentDistance = tileList[i].GetComponent<Tile>().distanceCopper;
        type = 'c';
    }

    if (finishCheck(surroundingTiles, currentDistance, type)) {
        Debug.Log("Current i value " + i);
        // removeTile (tileList[i]);
        finishedTiles.Add(tileList[i]);
        tileList.RemoveAt(i);
        currentTileAmount--;
        i--;
    }
}
//End while

//Flood tiles

//Loop to remove a tile
public void removeTile(GameObject tileToRemove)
{
    int index = 0;
    int currentTileCount = tileList.Count;

    for (int i = 0; i < currentTileCount; i++) {
        Debug.Log("REmove index" + index);

        if (tileList[i] == tileToRemove) {
            tileList.RemoveAt(index);
            index--;
            currentTileCount--;
            i--;
        }

        index++;
    }
}

//Get the count and set the surrounding tiles to the array
public int getSurroundingTiles(RaycastHit2D[] surroundingTiles, GameObject tile)
{
    int tileCount = 0;

    tile.GetComponentInParent<BoxCollider2D>().enabled = false;

    RaycastHit2D upTile, rightTile, downTile, leftTile;
    bool hasUpTile = false, hasRightTile = false, hasDownTile = false, hasLeftTile = false;

    upTile = Physics2D.Linecast(Vector2.up + new Vector2(tile.transform.position.x, tile.transform.position.y),
                                new Vector2(tile.transform.position.x, tile.transform.position.y), floorLayer);

    rightTile = Physics2D.Linecast(Vector2.right + new Vector2(tile.transform.position.x, tile.transform.position.y),
                                   new Vector2(tile.transform.position.x, tile.transform.position.y), floorLayer);

    downTile = Physics2D.Linecast(Vector2.down + new Vector2(tile.transform.position.x, tile.transform.position.y),
                                  new Vector2(tile.transform.position.x, tile.transform.position.y), floorLayer);

leftTile = Physics2D.Linecast(Vector2.left + new Vector2(tile.transform.position.x, tile.transform.position.y), new Vector2(tile.transform.position.x, tile.transform.position.y), floorLayer);

// Check all tiles for null, hit current object or it has a wall on it.

if (upTile.collider != null && !(upTile.collider.GetComponent<Tile>().hasWall()))
{
    hasUpTile = true;
}

if (rightTile.collider != null && !(rightTile.collider.GetComponent<Tile>().hasWall()))
{
    hasRightTile = true;
}

if (downTile.collider != null && !(downTile.collider.GetComponent<Tile>().hasWall()))
{
    hasDownTile = true;
}

if (leftTile.collider != null && !(leftTile.collider.GetComponent<Tile>().hasWall()))
{
    hasLeftTile = true;
}

// surroundingTiles = new RaycastHit2D[tileCount];

tile.GetComponentInParent<BoxCollider2D>().enabled = true;

if (hasUpTile) {
    surroundingTiles[0] = upTile;
    tileCount++;
}

if (hasRightTile) {
    surroundingTiles[1] = rightTile;
    tileCount++;
}
if (hasDownTile)
{
    surroundingTiles [2] = downTile;
    tileCount++;
}

if (hasLeftTile)
{
    surroundingTiles [3] = leftTile;
    tileCount++;
}

Debug.Log (tileCount);
return tileCount;

}

public void setDistance(GameObject tile, RaycastHit2D[] surroundingTiles, char type)
{
    Debug.Log ("In setDistanc");

    foreach (RaycastHit2D distance in surroundingTiles)
    {

        Debug.Log ("In loop for distance");
        if (distance.collider != null) {
            switch (type) {
            case 'g':
                Debug.Log ("Gold current tile" + tile.GetComponent<Tile> ().distanceGold);
                Debug.Log ("Gold surrounding tile " + distance.collider.GetComponent<Tile> ().distanceGold);
                if (tile.GetComponent<Tile> ().distanceGold > distance.collider.GetComponent<Tile> ().distanceGold + 1) {
                    tile.GetComponent<Tile> ().distanceGold = distance.collider.GetComponent<Tile> ().distanceGold + 1;
                }
                break;
            case 's':
                if (tile.GetComponent<Tile> ().distanceSilver > distance.collider.GetComponent<Tile> ().distanceSilver + 1) {
                    tile.GetComponent<Tile> ().distanceSilver = distance.collider.GetComponent<Tile> ().distanceSilver + 1;
                }
                break;
            } // end switch
        } // end if
    } // end foreach
} // end setDistance
case 'c':
    if (tile.GetComponent<Tile>() .distanceCopper > distance.collider.GetComponent<Tile>() .distanceCopper + 1) {
        tile.GetComponent<Tile>() .distanceCopper = distance.collider.GetComponent<Tile>() .distanceCopper + 1;
    }
    break;
default:
    return;
}
}

public bool finishCheck(RaycastHit2D[] _surroundingTiles, int setDistance, char type) {
    bool toRemove = true;
    switch (type) {
    case 'g':
        foreach (RaycastHit2D tile in _surroundingTiles) {

            if (tile.collider != null) {
                int distanceToGold = tile.collider.GetComponent<Tile>() .distanceGold;

                if ((distanceToGold == 100)) {
                    toRemove = false;
                }

                if (distanceToGold > setDistance + 1) {
                    toRemove = false;
                }

            }

        }

    }
    break;
    case 's':
        foreach (RaycastHit2D tile in _surroundingTiles) {

            if (tile.collider != null) {
                int distanceToSilver = tile.collider.GetComponent<Tile>() .distanceSilver;

            }

        }

    }
    break;
}
if ((distanceToSilver == 100) || (distanceToSilver > setDistance + 1)) {
    toRemove = false;
}
break;

break;
case 'c':
    foreach (RaycastHit2D tile in _surroundingTiles) {
        if (tile.collider != null) {
            int distanceToCopper = tile.collider.GetComponent<Tile>().distanceCopper;
        }
        if ((distanceToCopper == 100) || (distanceToCopper > setDistance + 1)) {
            toRemove = false;
        }
    }
    break;
default:
    toRemove = false;
    break;
}
return toRemove;

}--SmartSelect.cs--

using UnityEngine;
using System.Collections;

public class SmartSelect : MonoBehaviour {

    public double ValueOfGold, ValueOfSilver, ValueOfCopper;
    //public bool PlayerFound;
    public double TimeGold, TimeSilver, TimeCopper;
    public double RemeberGold, RemeberSilver, RemeberCopper;

    public int[] GoldArray;
    public int[] SilverArray;
    public int[] CopperArray;
public double UpTile, RightTile, LeftTile, DownTile;

public int tileToMoveTo; // 0 = Up, 1 = right, 2 = down, 3 = left
public int TimeDelay = 0;

// Use this for initialization
void Start () {
    GoldArray = new int[4];
    SilverArray = new int[4];
    CopperArray = new int[4];

    for (int i = 0; i < 4; i++) {
        GoldArray [i] = 0;
        SilverArray [i] = 0;
        CopperArray [i] = 0;
    }

    //PlayerFound = false;
    TimeGold = 10;
    TimeSilver = 10;
    TimeCopper = 10;

    ValueOfGold = 200;
    ValueOfSilver = 150;
    ValueOfCopper = 75;

    RememeberGold = 15;
    RememeberSilver = 30;
    RememeberCopper = 50;

}

// Update is called once per frame
void Update () {
    if (TimeDelay == 0) {
        TimeDelay = ResetTime;
        TimeCopper++;
        TimeGold++;
        TimeSilver++;
    }
    TimeDelay--;

}
public int SelectTile()
{

    UpTile = CalculateTileWeights (0);
    RightTile = CalculateTileWeights (1);
    DownTile = CalculateTileWeights (2);
    LeftTile = CalculateTileWeights (3);

    if (UpTile > RightTile && UpTile > DownTile && UpTile > LeftTile) {
        return 1;
    } else if (RightTile > DownTile && RightTile > LeftTile) {
        return 2;
    } else if (DownTile > LeftTile) {
        return 3;
    } else {
        return 4;
    }

    //0 = up
    private double CalculateTileWeights(int tile)
    {
        double answer =
            (0 - (0 - ValueOfGold) / (1 + Mathf.Exp ((float)(-TimeGold + RemeberGold)))) / GoldArray[tile] +
            (0 - (0 - ValueOfSilver) / (1 + Mathf.Exp ((float)(-TimeSilver + RemeberSilver)))) / SilverArray[tile] +
            (0 - (0 - ValueOfCopper) / (1 + Mathf.Exp ((float)(-TimeCopper + RemeberCopper)))) / CopperArray[tile];
        return answer;
    }

    --MovingObjects.cs--
    using UnityEngine;
    using System.Collections;

    //The abstract keyword enables you to create classes and class members that are incomplete and must be implemented in a derived class.
public abstract class MovingObject : MonoBehaviour
{
    public float moveTime = 0.1f; //Time it will take object to move, in seconds.
    public LayerMask blockingLayer; //Layer on which collision will be checked.

    public BoxCollider2D boxCollider; //The BoxCollider2D component attached to this object.
    private Rigidbody2D rb2D; //The Rigidbody2D component attached to this object.
    private float inverseMoveTime; //Used to make movement more efficient.

    //Protected, virtual functions can be overridden by inheriting classes.
    protected virtual void Start()
    {
        //Get a component reference to this object's BoxCollider2D
        boxCollider = GetComponent<BoxCollider2D> ();

        //Get a component reference to this object's Rigidbody2D
        rb2D = GetComponent<Rigidbody2D> ();

        //By storing the reciprocal of the move time we can use it by multiplying instead of dividing, this is more efficient.
        inverseMoveTime = 1f / moveTime;
    }

    //Move returns true if it is able to move and false if not.
    //Move takes parameters for x direction, y direction and a RaycastHit2D to check collision.
    protected bool Move(int xDir, int yDir, out RaycastHit2D hit)
    {
        //Store start position to move from, based on objects current transform position.
        Vector2 start = transform.position;

        //Calculate end position based on the direction parameters passed in when calling Move.
        Vector2 end = start + new Vector2(xDir, yDir);

        //Disable the boxCollider so that linecast doesn't hit this object's own collider.
        boxCollider.enabled = false;

        //Cast a line from start point to end point checking collision on blockingLayer.
        hit = Physics2D.Linecast(start, end, blockingLayer);

        //Re-enable boxCollider after linecast
    }
}
boxCollider.enabled = true;

// Check if anything was hit
if(hit.transform == null)
{
    // If nothing was hit, start SmoothMovement coroutine passing in the Vector2 end as destination
    // StartCoroutine(SmoothMovement(end));

    rb2D.MovePosition(end);

    // Return true to say that Move was successful
    return true;
}

// If something was hit, return false, Move was unsuccessful.
return false;

protected IEnumerator SmoothMovement(Vector3 end)
{
    // Calculate the remaining distance to move based on the square magnitude of the difference between current position and end parameter.
    // Square magnitude is used instead of magnitude because it's computationally cheaper.
    float sqrRemainingDistance = (transform.position - end).sqrMagnitude;

    // While that distance is greater than a very small amount (Epsilon, almost zero):
    while(sqrRemainingDistance > float.Epsilon)
    {
        // Find a new position proportionally closer to the end, based on the moveTime
        Vector3 newPosition = Vector3.MoveTowards(rb2D.position, end, inverseMoveTime * Time.deltaTime);

        // newPosition = new Vector3((float)((int)newPosition.x), (float)((int)newPosition.y), 0f);

        // Call MovePosition on attached Rigidbody2D and move it to the calculated position.
        rb2D.MovePosition(newPosition);
    }
}
// Recalculate the remaining distance after moving.
  sqrRemainingDistance = (transform.position - end).sqrMagnitude;

  // Return and loop until sqrRemainingDistance is close enough to zero to end the function
  yield return null;
}

// The virtual keyword means AttemptMove can be overridden by inheriting classes using the override keyword.
// AttemptMove takes a generic parameter T to specify the type of component we expect our unit to interact with if blocked (Player for Enemies, Wall for Player).
protected virtual void AttemptMove<T> (int xDir, int yDir) where T : Component
{
    // Hit will store whatever our linecast hits when Move is called.
    RaycastHit2D hit;

    // Set canMove to true if Move was successful, false if failed.
    bool canMove = Move (xDir, yDir, out hit);

    // Check if nothing was hit by linecast
    if(hit.transform == null)
        // If nothing was hit, return and don't execute further code.
        return;

    // Get a component reference to the component of type T attached to the object that was hit
    T hitComponent = hit.transform.GetComponent<T> ();

    // If canMove is false and hitComponent is not equal to null, meaning MovingObject is blocked and has hit something it can interact with
    if(!canMove && hitComponent != null)
        // Call the OnCantMove function and pass it hitComponent as a parameter.
        OnCantMove (hitComponent);
}

// The abstract modifier indicates that the thing being modified has a missing or incomplete implementation.
// OnCantMove will be overridden by functions in the inheriting classes.
protected abstract void OnCanMove<T> (T component)
   where T : Component;
}

--Enemy.cs--

using UnityEngine;
using System.Collections;

using System.Collections.Generic; //Allows us to use Lists.

//Enemy inherits from MovingObject, our base class for objects that can move, Player also inherits from this.
public class Enemy : MovingObject
{
   public int playerDamage;       //The amount of food points to subtract from the player when attacking.
   public AudioClip attackSound1;  //First of two audio clips to play when attacking the player.
   public AudioClip attackSound2;  //Second of two audio clips to play when attacking the player.
   public LayerMask floorLayer;

   public Vector2 closeGold, closeSilver, closeCopper;
   public int distanceToGold, distanceToSilver, distanceToCopper;
   public Tile currentTile, TileUp, TileDown, TileRight, TileLeft;

   public float smartDistance, prevSDist;

   public Vector2 target, player;   //Transform to attempt to move toward each turn.

   private Animator animator;      //Variable of type Animator to store a reference to the enemy’s Animator component.

   //Start overrides the virtual Start function of the base class.
   protected override void Start ()
   {
      //Register this enemy with our instance of GameManager by adding it to a list of Enemy objects.
      //This allows the GameManager to issue movement commands.
      GameManager.instance.AddEnemyToList (this);

      //Get and store a reference to the attached Animator component.
      animator = GetComponent<Animator> ();

      //Find the Player GameObject using it's tag and store a referen
ce to its transform component.

distanceToGold = 100;
distanceToSilver = 100;
distanceToCopper = 100;

target = new Vector2 (1, 0);

    //Call the start function of our base class MovingObject.
    base.Start ();
}

    //Override the AttemptMove function of MovingObject to include functionality needed for Enemy to skip turns.
    //See comments in MovingObject for more on how base AttemptMove function works.
    protected override void AttemptMove <T> (int xDir, int yDir)
    {
        //Call the AttemptMove function from MovingObject.
        base.AttemptMove <T> (xDir, yDir);
    }

    public void getTiles()
    {
        Vector2 start = new Vector2 (transform.position.x, transform.position.y);
        RaycastHit2D CastUp, CastDown, CastRight, CastLeft;
        Vector2 endUp, endDown, endRight, endLeft;
        RaycastHit2D[] rayCastArray = new RaycastHit2D[4];
        int lastGold, lastSilver, lastCopper;
        int random = Random.Range (1, 3);

        lastGold = 100;
        lastSilver = 100;
        lastCopper = 100;

        endUp = start + Vector2.up;
        endDown = start + Vector2.down;
        endRight = start + Vector2.right;
        endLeft = start + Vector2.left;

        closeGold = endUp;
        closeSilver = endUp;
        closeCopper = endUp;

        boxCollider.enabled = false;

        CastUp = Physics2D.Linecast(endUp, start, floorLayer);
        CastDown = Physics2D.Linecast(endDown, start, floorLayer);
CastRight = Physics2D.Linecast (endRight, start, floorLayer);
CastLeft = Physics2D.Linecast (endLeft, start, floorLayer);

rayCastArray [0] = CastUp;
rayCastArray [2] = CastDown;
rayCastArray [1] = CastRight;
rayCastArray [3] = CastLeft;

for (int i = 0; i < 4; i++) {
    GetComponent<SmartSelect> ().GoldArray [i] = 100;
    GetComponent<SmartSelect> () .SilverArray [i] = 100;
    GetComponent<SmartSelect> () .CopperArray [i] = 100;
}

for (int i = 0; i < 4; i++)
{
    if (rayCastArray [i].collider != null) {
        GetComponent<SmartSelect> ().GoldArray [i] = rayCastArray [i].collider.GetComponent<Tile> () .distanceGold;
        GetComponent<SmartSelect> () .SilverArray [i] = rayCastArray [i].collider.GetComponent<Tile> () .distanceSilver;
        GetComponent<SmartSelect> () .CopperArray [i] = rayCastArray [i].collider.GetComponent<Tile> () .distanceCopper;

        if (rayCastArray [i].collider.GetComponent<Tile> () .distanceGold < lastGold) {
            lastGold = rayCastArray [i].collider.GetComponent<Tile> () .distanceGold;
            closeGold = new Vector2 (rayCastArray [i].collider.GetComponent<Tile> () .transform.position.x,
            rayCastArray [i].collider.GetComponent<Tile> () .transform.position.y);
        } else if (rayCastArray [i].collider.GetComponent<Tile> () .distanceGold == lastGold) {
            if (random == 1) {
                closeGold = new Vector2 (rayCastArray [i].collider.GetComponent<Tile> () .transform.position.x,
                rayCastArray [i].collider.GetComponent<Tile> () .transform.position.y);
            }
        }

        if (rayCastArray [i].collider.GetComponent<Tile> () .distanceSilver < lastSilver) {
            lastSilver = rayCastArray [i].collider.GetComponent<Tile> () .distanceSilver;
            closeSilver = new Vector2 (rayCastArray [i].collider.GetComponent<Tile> () .transform.position.x,
            rayCastArray [i].collider.GetComponent<Tile> () .transform.position.y);
        } else if (rayCastArray [i].collider.GetComponent<Tile> () .distanceSilver == lastSilver) {
            if (random == 1) {
                closeSilver = new Vector2 (rayCastArray [i].collider.GetComponent<Tile> () .transform.position.x,
                rayCastArray [i].collider.GetComponent<Tile> () .transform.position.y);
            }
        }
    }
}
if (random == 1) {
    closeSilver = new Vector2 (rayCastArray [i].collider.GetComponent<Tile> ().transform.position.x,
                                rayCastArray [i].collider.GetComponent<Tile> ().transform.position.y);
}

if (rayCastArray [i].collider.GetComponent<Tile> ().distanceCopper < lastCopper) {
    lastCopper = rayCastArray [i].collider.GetComponent<Tile> ().distanceCopper;
    closeCopper = new Vector2 (rayCastArray [i].collider.GetComponent<Tile> ().transform.position.x,
                                rayCastArray [i].collider.GetComponent<Tile> ().transform.position.y);
} else if (rayCastArray [i].collider.GetComponent<Tile> ().distanceCopper == lastCopper) {
    if (random == 1) {
        closeCopper = new Vector2 (rayCastArray [i].collider.GetComponent<Tile> ().transform.position.x,
                                rayCastArray [i].collider.GetComponent<Tile> ().transform.position.y);
    }
}

else {
    GetComponent<SmartSelect> ().GoldArray [i] = 100;
    GetComponent<SmartSelect> ().SilverArray [i] = 100;
    GetComponent<SmartSelect> ().CopperArray [i] = 100;
}

distanceToGold = lastGold;
distanceToSilver = lastSilver;
distanceToCopper = lastCopper;

boxCollider.enabled = true;

}

public void selectTraget()
{
getTiles();

if (distanceToCopper == 2) {
    /*
    Vector3[] enemySpawnPoints = new Vector3[10];
    
    enemySpawnPoints[0] = new Vector3(18, 3, 0);
    enemySpawnPoints[1] = new Vector3(26, 22, 0);
    enemySpawnPoints[2] = new Vector3(27, 1, 0);
    enemySpawnPoints[3] = new Vector3(8, 2, 0);
    enemySpawnPoints[4] = new Vector3(0, 10, 0);
    enemySpawnPoints[5] = new Vector3(12, 17, 0);
    enemySpawnPoints[6] = new Vector3(19, 23, 0);
    enemySpawnPoints[7] = new Vector3(16, 0, 0);
    enemySpawnPoints[8] = new Vector3(28, 5, 0);
    enemySpawnPoints[9] = new Vector3(27, 16, 0);
    */
    int randomSpot = Random.Range(1, 10);
    transform.position = enemySpawnPoints[randomSpot];
    return;
    */
    GetComponent<SmartSelect>().TimeCopper = 0;
}

if (distanceToGold == 2) {
    GetComponent<SmartSelect>().TimeGold = 0;
}

if (distanceToSilver == 2) {
    GetComponent<SmartSelect>().TimeSilver = 0;
}

player = new Vector2(GameObject.FindGameObjectWithTag("Player").transform.position.x,
    GameObject.FindGameObjectWithTag("Player").transform.position.y);

if (Mathf.Abs(player.x - transform.position.x) + Mathf.Abs(player.y - transform.position.y) < 10) {
    target = player;
    return;
}

switch (GetComponent<SmartSelect>().SelectTile()) {
    case 1:
        target = new Vector2(0, 1);
break;
case 2:
    target = new Vector2 (1,0);
    break;
case 3:
    target = new Vector2 (0, -1);
    break;
case 4:
    target = new Vector2 (-1, 0);
    break;
default:
    target = new Vector2 (0, 0);
    break;
}

//MoveEnemy is called by the GameManager each turn to tell each Enemy to try to move towards the player.
public void MoveEnemy ()
{
    //Called to select the target tile to move to
    //sets the target Vector2 to that tile
    selectTarget();

    //Declare variables for X and Y axis move directions, these range from -1 to 1.
    //These values allow us to choose between the cardinal directions: up, down, left and right.
    int xDir = (int)target.x;
    int yDir = (int)target.y;

    //If the difference in positions is approximately zero (Epsilon) do the following:
    if (Mathf.Abs(target.x - transform.position.x) < float.Epsilon && Mathf.Abs(target.y - transform.position.y) < float.Epsilon) {
        int random = Random.Range (1, 3);

        if (random == 1) {
            //If the y coordinate of the target’s (player or coin) position is greater than the y coordinate of this enemy's position set y direction 1 (to move up). If not, set it to -1 (to move down).
            yDir = target.y > transform.position.y ? 1 : -1;
        } else {
            //Check if target x position is greater than enemy's x position, if so set x direction to 1 (move right), if not set to -1 (move left).
            xDir = target.x > transform.position.x ? 1 : -1;
        }
    }
//If the difference in positions is not approximately zero (Epsilon) do the following:
   else if(Mathf.Abs (target.x - transform.position.x) < float.Epsilon) {

   //If the y coordinate of the target's (player) position is greater than the y coordinate of this enemy's position set y direction 1 (to move up). If not, set it to -1 (to move down).
   yDir = target.y > transform.position.y ? 1 : -1;

   } else {

   //Check if target x position is greater than enemy's x position, if so set x direction to 1 (move right), if not set to -1 (move left).
   xDir = target.x > transform.position.x ? 1 : -1;

   //Call the AttemptMove function and pass in the generic parameter Player, because Enemy is moving and expecting to potentially encounter a Player
   AttemptMove <Player> (xDir, yDir);

   //OnCantMove is called if Enemy attempts to move into a space occupied by a Player, it overrides the OnCantMove function of MovingObject //and takes a generic parameter T which we use to pass in the component we expect to encounter, in this case Player
   protected override void OnCantMove <T> (T component) {

   //Declare hitPlayer and set it to equal the encountered component.
   Player hitPlayer = component as Player;

   //Call the LoseFood function of hitPlayer passing it playerDamage, the amount of foodpoints to be subtracted.
   hitPlayer.LoseFood (playerDamage);

   //Set the attack trigger of animator to trigger Enemy attack animation.
   animator.SetTrigger ("enemyAttack");

   //Call the RandomizeSfx function of SoundManager passing in the two audio clips to choose randomly between.
   SoundManager.instance.RandomizeSfx (attackSound1, attackSound2);
   }
   }
using UnityEngine;
using System;
using System.Collections.Generic;  // Allows us to use Lists.
using Random = UnityEngine.Random;  // Tells Random to use the Unity Engine random number generator.

public class BoardManager : MonoBehaviour
{
    // Using Serializable allows us to embed a class with sub properties in the inspector.
    [Serializable]
    public class Count
    {
        public int minimum;  // Minimum value for our Count class.
        public int maximum;  // Maximum value for our Count class.

        // Assignment constructor.
        public Count (int min, int max)
        {
            minimum = min;
            maximum = max;
        }
    }

    public int columns = 100;  // Number of columns in our game board.
    public int rows = 100;  // Number of rows in our game board.
    public Count wallCount = new Count (5, 9);  // Lower and upper limit for our random number of walls per level.
    public Count foodCount = new Count (1, 5);  // Lower and upper limit for our random number of food items per level.
    public GameObject exit;  // Prefab to spawn for exit.
    public GameObject gold;
    public GameObject silver;
    public GameObject copper;
    public GameObject[] floorTiles;  // Array of floor prefabs.
    public GameObject[] wallTiles;  // Array of wall prefabs.
    public GameObject[] foodTiles;  // Array of food prefabs.
    public GameObject[] enemyTiles;  // Array of enemy prefabs.
    public GameObject[] outerWallTiles;
//Array of outer tile prefabs.

public FloodTiles floodScript;
private Transform boardHolder;    //A variable to store a reference to the transform of our Board object.
private Transform wallHolder;
private List<Vector3> gridPositions = new List<Vector3> ();  //A list of possible locations to place tiles.
private List<Vector3> wallPositions = new List<Vector3> ();

//Clears our list gridPositions and prepares it to generate a new board.
void InitialiseList ()
{
    //Clear our list gridPositions.
    gridPositions.Clear ();
    makeWallList ();

    //Loop through x axis (columns).
    for(int x = 0; x < columns-1; x++)
    {
        //Within each column, loop through y axis (rows).
        for(int y = 0; y < rows-1; y++)
        {
            //At each index add a new Vector3 to our list with the x and y coordinates of that position.
            gridPositions.Add (new Vector3(x, y, 0f));
        }
    }
}

void makeWallList()
{
    wallPositions.Add (new Vector3(0, 3, 0));
    wallPositions.Add (new Vector3(1, 3, 0));
    wallPositions.Add (new Vector3(4, 3, 0));
    for (int i = 0; i < 8; i++) {
        wallPositions.Add (new Vector3(5, i, 0f));
    }
    for (int i = 0; i < 3; i++) {
        wallPositions.Add (new Vector3(i, 12, 0));
    }
    for (int i = 0; i < 6; i++) {
        wallPositions.Add (new Vector3(5 + i, 12, 0));
    }
    wallPositions.Add (new Vector3(6, 7, 0));
    wallPositions.Add (new Vector3(7, 7, 0));
    wallPositions.Add (new Vector3(10, 3, 0));
wallPositions.Add (new Vector3(10, 5, 0));
wallPositions.Add (new Vector3(10, 7, 0));
wallPositions.Add (new Vector3(11, 3, 0));
wallPositions.Add (new Vector3(11, 5, 0));
wallPositions.Add (new Vector3(11, 7, 0));

for (int i = 0; i < 8; i++)
{
    if (i == 4)
        continue;

    wallPositions.Add (new Vector3(12, i, 0));
}

wallPositions.Add (new Vector3(21, 6, 0));
wallPositions.Add (new Vector3(21, 16, 0));
wallPositions.Add (new Vector3(21, 17, 0));
wallPositions.Add (new Vector3(16, 18, 0));

for (int i = 0; i < 3; i++)
{
    wallPositions.Add(new Vector3(i, 12, 0));
}

for (int i = 0; i < 6; i++)
{
    wallPositions.Add(new Vector3(i + 5, 12, 0));
}

for (int i = 0; i < 7; i++)
{
    wallPositions.Add(new Vector3(i + 13, 12, 0));
}

for (int i = 0; i < 4; i++)
{
    wallPositions.Add(new Vector3(i, 15, 0));
}

for (int i = 0; i < 10; i++)
{
    wallPositions.Add(new Vector3(i + 6, 15, 0));
}

for (int i = 0; i < 5; i++)
{
    wallPositions.Add(new Vector3(i + 18, 15, 0));
}

for (int i = 0; i < 4; i++)
{
    wallPositions.Add(new Vector3(i + 24, 14, 0));
}
for (int i = 0; i < 3; i++) {
    wallPositions.Add(new Vector3(i, 20, 0));
}

for (int i = 0; i < 3; i++) {
    wallPositions.Add(new Vector3(i + 5, 20, 0));
}

for (int i = 0; i < 5; i++) {
    wallPositions.Add(new Vector3(i + 9, 20, 0));
}

for (int i = 0; i < 3; i++) {
    wallPositions.Add(new Vector3(i + 15, 20, 0));
}

for (int i = 0; i < 6; i++) {
    wallPositions.Add(new Vector3(i + 21, 19, 0));
}

wallPositions.Add (new Vector3 (27, 17, 0));
for (int i = 0; i < 3; i++) {
    wallPositions.Add(new Vector3(i + 16, 17, 0));
}

for (int i = 0; i < 3; i++) {
    wallPositions.Add(new Vector3(i + 22, 11, 0));
}

for (int i = 0; i < 3; i++) {
    wallPositions.Add(new Vector3(i + 19, 8, 0));
}

for (int i = 0; i < 5; i++) {
    wallPositions.Add(new Vector3(i + 24, 4, 0));
}

for (int i = 0; i < 3; i++) {
    if (i == 1)
        continue;

    wallPositions.Add(new Vector3(i + 19, 8, 0));
}

for (int i = 0; i < 5; i++) {
    wallPositions.Add(new Vector3(i + 24, 4, 0));
}

53
for (int i = 0; i < 3; i++) {
    wallPositions.Add(new Vector3(7, i+16, 0));
}

for (int i = 0; i < 3; i++) {
    wallPositions.Add(new Vector3(7, i+22, 0));
}

for (int i = 0; i < 4; i++) {
    wallPositions.Add(new Vector3(13, i+16, 0));
}

for (int i = 0; i < 4; i++) {
    wallPositions.Add(new Vector3(13, i+21, 0));
}

for (int i = 0; i < 2; i++) {
    wallPositions.Add(new Vector3(16, i+21, 0));
}

for (int i = 0; i < 2; i++) {
    wallPositions.Add(new Vector3(18, i+18, 0));
}

for (int i = 0; i < 8; i++) {
    if (i == 4)
        continue;
    wallPositions.Add(new Vector3(19, i, 0));
}

for (int i = 0; i < 4; i++) {
    wallPositions.Add(new Vector3(21, i+21, 0));
}

for (int i = 0; i < 4; i++) {
    wallPositions.Add(new Vector3(22, i+6, 0));
}

for (int i = 0; i < 3; i++) {
    wallPositions.Add(new Vector3(22, i+12, 0));
}
for (int i = 0; i < 3; i++) {
    wallPositions.Add(new Vector3(24, i, 0));
}
for (int i = 0; i < 6; i++) {
    wallPositions.Add(new Vector3(24, i+5, 0));
}
for (int i = 0; i < 3; i++) {
    wallPositions.Add(new Vector3(25, i+15, 0));
}
for (int i = 0; i < 3; i++) {
    wallPositions.Add(new Vector3(26, i+15, 0));
}

//Sets up the outer walls and floor (background) of the game board.
void BoardSetup ()
{
    //Instantiate Board and set boardHolder to its transform.
    boardHolder = new GameObject("Board").transform;

    //Loop along x axis, starting from -1 (to fill corner) with floor or outerwall edge tiles.
    for(int x = -1; x < columns + 1; x++)
    {
        //Loop along y axis, starting from -1 to place floor or outerwall tiles.
        for(int y = -1; y < rows + 1; y++)
        {
            //Choose a random tile from our array of floor tile pre
            //fabs and prepare to instantiate it.
            GameObject toInstantiate = floorTiles[Random.Range (0,floorTiles.Length)];

            //Check if we current position is at board edge, if so
            //choose a random outer wall prefab from our array of outer wall tiles.
            if(x == -1 || x == columns || y == -1 || y == rows)
                toInstantiate = outerWallTiles [Random.Range (0, outerWallTiles.Length)];

            //Instantiate the GameObject instance using the prefab
            //chosen for toInstantiate at the Vector3 corresponding to current grid p
            //osition in loop, cast it to GameObject.
            GameObject instance =
                Instantiate (toInstantiate, new Vector3 (x, y, 0f), Quaternion.identity) as GameObject;
        }
    }
}
//Set the parent of our newly instantiated object instance to boardHolder, this is just organizational to avoid cluttering hierarchy.
    instance.transform.SetParent(boardHolder);
}
}

//RandomPosition returns a random position from our list gridPositions.
Vector3 RandomPosition ()
{
    //Declare an integer randomIndex, set it's value to a random number between 0 and the count of items in our List gridPositions.
    int randomIndex = Random.Range(0, gridPositions.Count);

    //Declare a variable of type Vector3 called randomPosition, set it's value to the entry at randomIndex from our List gridPositions.
    Vector3 randomPosition = gridPositions[randomIndex];

    //Remove the entry at randomIndex from the list so that it can't be re-used.
    gridPositions.RemoveAt(randomIndex);

    //Return the randomly selected Vector3 position.
    return randomPosition;
}

//LayoutObjectAtRandom accepts an array of game objects to choose from along with a minimum and maximum range for the number of objects to create.
void LayoutObjectAtRandom (GameObject[] tileArray, int minimum, int maximum)
{
    //Choose a random number of objects to instantiate within the minimum and maximum limits
    int objectCount = Random.Range(minimum, maximum+1);

    //Instantiate objects until the randomly chosen limit objectCount is reached
    for(int i = 0; i < objectCount; i++)
    {
        //Choose a position for randomPosition by getting a random position from our list of available Vector3s stored in gridPosition
        Vector3 randomPosition = RandomPosition();

        //Choose a random tile from tileArray and assign it to tileChoice
        GameObject tileChoice = tileArray[Random.Range(0, tileArray.Length)];

        //Set the parent of our newly instantiated object instance to boardHolder, this is just organizational to avoid cluttering hierarchy.
        tileChoice.transform.SetParent(boardHolder);
    }
}
void PlaceEnemy(int amount)
{
    for (int i = 0; i < amount; i++)
    {
        //Choose a position for randomPosition by getting a random position from our list of available Vector3s stored in gridPosition
        Vector3 randomPosition = new Vector3(27f, 16f, 0f);

        //Choose a random tile from tileArray and assign it to tile Choice
        GameObject tileChoice = enemyTiles[Random.Range(0, enemyTiles.Length)];

        //Instantiate tileChoice at the position returned by Random Position with no change in rotation
        Instantiate(tileChoice, randomPosition, Quaternion.identity);
    }
}

void PlaceCoins()
{
    Instantiate(gold, new Vector3(17f, 18f, 0f), Quaternion.identity);
    Instantiate(silver, new Vector3(10f, 1f, 0f), Quaternion.identity);
    Instantiate(copper, new Vector3(21f, 7f, 0f), Quaternion.identity);
}

void LayoutWalls()
{
    int i = 0;
    foreach (Vector3 wall in wallPositions)
    {
        GameObject tileChoice = wallTiles[Random.Range(0, wallTiles.Length)];
        GameObject instance = Instantiate(tileChoice, wall, Quaternion.identity);
    }
}
inion.identity) as GameObject;
        i++;
        instance.transform.SetParent (wallHolder);
    }
}

//SetupScene initializes our level and calls the previous functions
to lay out the game board
public void SetupScene (int level)
{
    PlaceCoins();

    //Creates the outer walls and floor.
    BoardSetup ();

    //Reset our list of gridpositions.
    InitialiseList ();

    LayoutWalls ();

    //Instantiate a random number of wall tiles based on minimum an
d maximum, at randomized positions.
    //LayoutObjectAtRandom (wallTiles, wallCount.minimum, wallCount
    .maximum);

    //Instantiate a random number of food tiles based on minimum an
d maximum, at randomized positions.
    //LayoutObjectAtRandom (foodTiles, foodCount.minimum, foodCount
    .maximum);

    //Determine number of enemies based on current level number, ba
sed on a logarithmic progression
    //int enemyCount = (int)Mathf.Log(level, 2f);

    //Instantiate a random number of enemies based on minimum and m
aximum, at randomized positions.
    PlaceEnemy(1);

    //Instantiate the exit tile in the upper right hand corner of o
ur game board
    Instantiate (exit, new Vector3 (columns - 1, rows - 1, 0f), Qua
ternion.identity);
}

--Player.cs--
using UnityEngine;
using System.Collections;
using UnityEngine.UI;  //Allows us to use UI.
using UnityEngine.SceneManagement;

//Player inherits from MovingObject, our base class for objects that can move, Enemy also inherits from this.
public class Player : MovingObject
{
    public float restartLevelDelay = 1f;  //Delay time in seconds to restart level.
    public int pointsPerFood = 10;  //Number of points to add to player food points when picking up a food object.
    public int pointsPerSoda = 20;  //Number of points to add to player food points when picking up a soda object.
    public int wallDamage = 1;  //How much damage a player does to a wall when chopping it.
    public Text foodText;  //UI Text to display current player food total.
    public AudioClip moveSound1;  //1 of 2 Audio clips to play when player moves.
    public AudioClip moveSound2;  //2 of 2 Audio clips to play when player moves.
    public AudioClip eatSound1;  //1 of 2 Audio clips to play when player collects a food object.
    public AudioClip eatSound2;  //2 of 2 Audio clips to play when player collects a food object.
    public AudioClip drinkSound1;  //1 of 2 Audio clips to play when player collects a soda object.
    public AudioClip drinkSound2;  //2 of 2 Audio clips to play when player collects a soda object.
    public AudioClip gameOverSound;  //Audio clip to play when player dies.
    private Animator animator;  //Used to store a reference to the Player's animator component.
    private int food;  //Used to store player food points total during level.
    private Vector2 touchOrigin = -Vector2.one;  //Used to store location of screen touch origin for mobile controls.

    //Start overrides the Start function of MovingObject
    protected override void Start()
    {
        //Get a component reference to the Player's animator component
        animator = GetComponent<Animator>();

        //Get the current food point total stored in GameManager.instance between levels.
        food = GameManager.instance.playerFoodPoints;

        //Set the foodText to reflect the current player food total.
    }
}
foodText.text = "Hp: " + food;

//Call the Start function of the MovingObject base class.
base.Start();

//This function is called when the behaviour becomes disabled or inactive.
private void OnDisable ()
{
    //When Player object is disabled, store the current local food total in the GameManager so it can be re-loaded in next level.
    GameManager.instance.playerFoodPoints = food;
}

private void Update ()
{
    //If it's not the player's turn, exit the function.
    if (!GameManager.instance.playersTurn) return;

    int horizontal = 0;    //Used to store the horizontal move direction.
    int vertical = 0;      //Used to store the vertical move direction.

    //Check if we are running either in the Unity editor or in a standalone build.
    #if UNITY_STANDALONE || UNITY_WEBPLAYER
    //Get input from the input manager, round it to an integer and store in horizontal to set x axis move direction
    horizontal = (int) (Input.GetAxisRaw ("Horizontal"));

    //Get input from the input manager, round it to an integer and store in vertical to set y axis move direction
    vertical = (int) (Input.GetAxisRaw ("Vertical"));

    //Check if moving horizontally, if so set vertical to zero.
    if (horizontal != 0)
    {
        vertical = 0;
    }
    #endif
    //Check if we are running on iOS, Android, Windows Phone 8 or Unity iPhone
    #elif UNITY_IOS || UNITY_ANDROID || UNITY_WP8 || UNITY_IPHONE
    //Check if Input has registered more than zero touches
    if (Input.touchCount > 0)
    {
        //Store the first touch detected.
        Touch myTouch = Input.touches[0];
// Check if the phase of that touch equalsBegan
if (myTouch.phase == TouchPhase.Began)
{
    // If so, set touchOrigin to the position of that touch
    touchOrigin = myTouch.position;
}

// If the touch phase is not Began, and instead is equal to
// Ended and the x of touchOrigin is greater or equal to zero:
else if (myTouch.phase == TouchPhase.Ended && touchOrigin.x >= 0)
{
    // Set touchEnd to equal the position of this touch
    Vector2 touchEnd = myTouch.position;

    // Calculate the difference between the beginning and end
    // of the touch on the x axis.
    float x = touchEnd.x - touchOrigin.x;

    // Calculate the difference between the beginning and end
    // of the touch on the y axis.
    float y = touchEnd.y - touchOrigin.y;

    // Set touchOrigin.x to -1 so that our else if statement will evaluate false and not repeat immediately.
    touchOrigin.x = -1;

    // Check if the difference along the x axis is greater than the difference along the y axis.
    if (Mathf.Abs(x) > Mathf.Abs(y))
    {
        // If x is greater than zero, set horizontal to 1, otherwise set it to -1
        horizontal = x > 0 ? 1 : -1;
        else
        {
            // If y is greater than zero, set horizontal to 1, otherwise set it to -1
            vertical = y > 0 ? 1 : -1;
        }
    }

    #endif // End of mobile platform dependendent compilation section started above with #elif

    // Check if we have a non-zero value for horizontal or vertical
    if (horizontal != 0 || vertical != 0)
    {
        // Call AttemptMove passing in the generic parameter Wall, since that is what Player may interact with if they encounter one (by attacking it)
        // Pass in horizontal and vertical as parameters to specify the direction to move Player in.
        AttemptMove<Wall> (horizontal, vertical);
    }
}
protected override void AttemptMove <T> (int xDir, int yDir) {
    // Every time player moves, subtract from food points total.
    // food--;

    // Update food text display to reflect current score.
    // foodText.text = "Food: " + food;

    // Call the AttemptMove method of the base class, passing in the
    // component T (in this case Wall) and x and y direction to move.
    base.AttemptMove <T> (xDir, yDir);

    // Hit allows us to reference the result of the Linecast done in
    // Move.
    RaycastHit2D hit;

    // If Move returns true, meaning Player was able to move into an
    // empty space.
    if (Move (xDir, yDir, out hit)) {
        // Call RandomizeSfx of SoundManager to play the move sound,
        // passing in two audio clips to choose from.
        SoundManager.instance.RandomizeSfx (moveSound1, moveSound2);
    }

    // Since the player has moved and lost food points, check if the
    // game has ended.
    // CheckIfGameOver ();

    // Set the playersTurn boolean of GameManager to false now that
    // players turn is over.
    GameManager.instance.playersTurn = false;
}

protected override void OnCantMove <T> (T component) {
    // Set hitWall to equal the component passed in as a parameter.
    Wall hitWall = component as Wall;

    // Call the DamageWall function of the Wall we are hitting.
    // hitWall.DamageWall (wallDamage);
/Set the attack trigger of the player's animation controller in order to play the player's attack animation.
    animator.SetTrigger ("playerChop");

//OnTriggerEnter2D is sent when another object enters a trigger collider attached to this object (2D physics only).
private void OnTriggerEnter2D (Collider2D other)
{
    //Check if the tag of the trigger collided with is Exit.
    if(other.tag == "Exit")
    {
        //Invoke the Restart function to start the next level with a delay of restartLevelDelay (default 1 second).
        Invoke ("Restart", restartLevelDelay);

        //Disable the player object since level is over.
        enabled = false;
    }

    //Check if the tag of the trigger collided with is Food.
    else if(other.tag == "Food")
    {
        //Add pointsPerFood to the players current food total.
        food += pointsPerFood;

        //Update foodText to represent current total and notify player that they gained points
        foodText.text = "+" + pointsPerFood + " Hp: " + food;

        //Call the RandomizeSfx function of SoundManager and pass in two eating sounds to choose between to play the eating sound effect.
        SoundManager.instance.RandomizeSfx (eatSound1, eatSound2);

        //Disable the food object the player collided with.
        other.gameObject.SetActive (false);
    }

    //Check if the tag of the trigger collided with is Soda.
    else if(other.tag == "Soda")
    {
        //Add pointsPerSoda to players food points total
        food += pointsPerSoda;

        //Update foodText to represent current total and notify player that they gained points
        foodText.text = "+" + pointsPerSoda + "Hp: " + food;

        //Call the RandomizeSfx function of SoundManager and pass in two drinking sounds to choose between to play the drinking sound effect.
        SoundManager.instance.RandomizeSfx (drinkSound1, drinkSound2);
    }
//Disable the soda object the player collided with.
other.gameObject.SetActive (false);
}

//Restart reloads the scene when called.
private void Restart ()
{
    //Load the last scene loaded, in this case Main, the only scene in
    //the game.
    SceneManager.LoadScene("Main");
}

//LoseFood is called when an enemy attacks the player.
//It takes a parameter loss which specifies how many points to lose
.
public void LoseFood (int loss)
{
    //Set the trigger for the player animator to transition to the
    playerHit animation.
    animator.SetTrigger ("playerHit");

    //Subtract lost food points from the players total.
    food -= loss;

    //Update the food display with the new total.
    foodText.text = "-" + loss + " Hp: " + food;

    //Check to see if game has ended.
    CheckIfGameOver ();
}

//CheckIfGameOver checks if the player is out of food points and if
so, ends the game.
private void CheckIfGameOver ()
{
    //Check if food point total is less than or equal to zero.
    if (food <= 0)
    {
        //Call the PlaySingle function of SoundManager and pass it
        the gameOverSound as the audio clip to play.
        SoundManager.instance.PlaySingle (gameOverSound);

        //Stop the background music.
        SoundManager.instance.musicSource.Stop();

        //Call the GameOver function of GameManager.
        GameManager.instance.GameOver ();
    }
}
References