The Wave Function Collapse Algorithm and its Application to AI Navigation

Luke Trefry[1]

[1]Eastern Nazarene College

ABSTRACT

For so long, game developers have had to hand place waypoints, leading to linear paths being taken by Artificial Intelligence (AI) which move between them. These paths often loop, and if they don't, they only serve to move an AI from point A to point B in the same way time and time again. A more dynamic solution is needed, and the Wave Function Collapse Algorithm is suited to provide that. This solution was implemented in the Unity engine using C#. By utilizing the provided Wave Function Collapse Algorithm template, waypoints were able to be substituted for game objects. When implemented in this manner, the Wave Function Collapse Algorithm places waypoints within a defined area allowing an AI navigator to maneuver between them. Should a waypoint be inaccessible, the waypoints will regenerate. Waypoints, upon the map regenerating, are removed so that any of those colliding waypoints cannot be accessed by the AI. This creates a dynamic path for an AI to traverse. The AI only moves between active waypoints and the map is able to regenerate so that the paths laid out do not succumb to the same pitfalls that traditionally plotted paths do. The next logical step is to apply the Wave Function Collapse Algorithm to a generated map. By allowing the Wave Function Collapse Algorithm to plot waypoints dynamically and continuously, different applications can utilize this to allow an AI to traverse a map in a unique way.

Introduction

In many games and applications there exists a problem where AI navigation is linear. Paths being taken either lead the AI along from point A to point B or a set of predefined waypoints, waypoints being objects or positions which an AI navigates towards. This project was created using the Unit Engine and C# code. Additionally, it utilized the Wave Function Collapse Algorithm to place waypoints. This template which I applied to place waypoints is typically used to generate maps, but in this case tiles or other objects used to build maps were replaced with waypoints. The WFCA is the solution to linear pathing because it is able to place waypoints arbitrarily and repeatedly within a defined area, making AI navigation dynamic. The Wave Function Collapse Algorithm can plot waypoints dynamically and continuously, allowing an AI to traverse a map in a unique way.

Concept of Operations and Requirements

CONOPS: The Player

The Concept of Operations provides a top-level description if the action which are supposed to take place during the game. Below we can see the CONOPS which describe the player's interaction with the AI. The player chases the AI which is being guided by the WFCA and then collides with it for the purpose of generating a score.

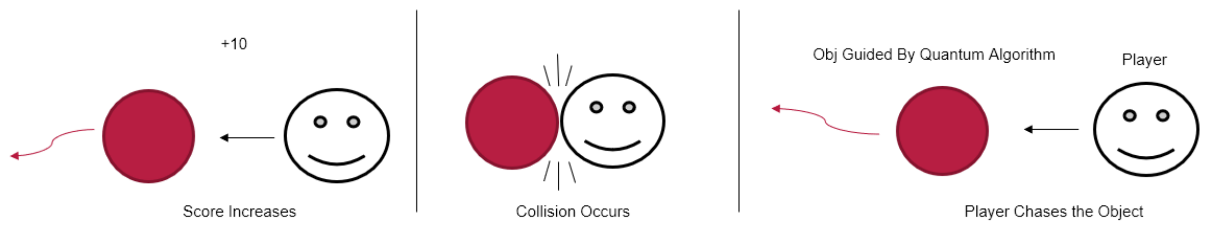


Figure 1. Player CONOPS.

Requirements: The Player

The requirements that the player character must meet are as follows:

- The player **shall** [1] be able to interact with the in-game map and object.
    - The controls **shall** [1.1] dictate which direction the player moves.
    - The collisions **shall** [1.2] allow the player to interact with the environment.
    - The score **shall** [1.3] increase upon the player interacting with the object.

The purpose of these requirements is to show that the player character is operating as intended. The player is supposed to be able to interact with the in-game map and object by controlling the player and colliding with in game object to create a score. A user is able to interact with the game by completing these tasks.

CONOPS: The WFCA

Figure 2 depicts the WFCA CONOPS. Within the first panel labeled I, a dataset is input into the WFCA. The dataset consists of possible waypoints that can appear within the map. The data set exists as a group of game objects within the Unity engine, and by placing those game objects within the provided WFCA input frame a dataset of waypoint is created.

In the second panel, labeled II, the dataset is input into the WFCA, and those waypoints are placed into super position. The WFCA, in this case, works by arbitrarily deciding whether or not a waypoint will exist in any given position. Before it does this, however, it places every object present in the dataset into "superposition," meaning that every object is occupying ever possible position simultaneously. Since there's only one object, a waypoint, this step only serves to decide whether or not a waypoint will exist in any given position.

In the third panel, labeled III, the waypoints begin to collapse, meaning that the decision which dictates where every waypoint exists is being made. As waypoints begin to take form, a path that the AI can traverse is created. The AI moves from waypoint to waypoint, and as seen in panel IV and V, does this repeatedly until a path is formed.

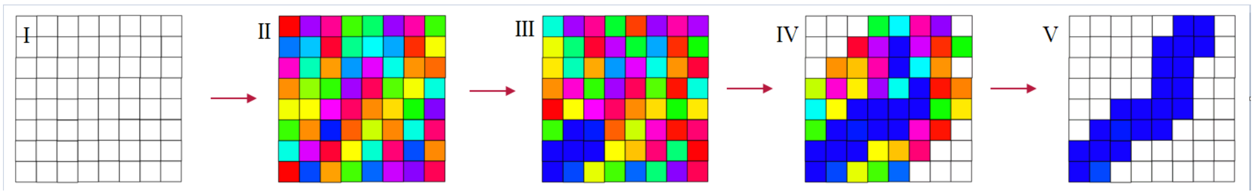


Figure 2. WFCA CONOPS.

Requirements: The WFCA

The requirements for the WFCA are as follows:

- The <u>object</u> **shall** [2] move around the map.
    - The <u>wave function collapse algorithm</u> **shall** [2.1] generate a traversable path.
    - The <u>waypoints</u> **shall** [2.2] guide the object along the path.
    - The <u>dataset</u> **shall** [2.3] determine where each waypoint can be placed in relation to other waypoints.

The decisions that went into these requirements were informed by a need to make it so that the AI would freely move around the map without impedance. It was by these requirements that it could be determined if the WFCA was working as intended. These requirements describe that the AI must be able to move around the map along a path generated by the WFCA. This path would be made up of waypoints which were input into the WFCA as a part of the data set.

CONOPS: The Map

The map, pictured below, exists to contain the interactions which the player can have. The circles and squares present within the map, which is sealed in by the large, surrounding circle, represents obstacles that the player and AI must maneuver around.

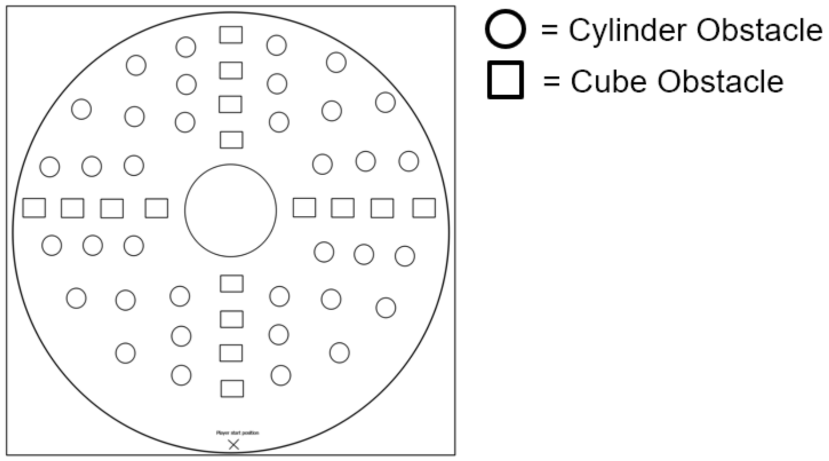


Figure 3. Map CONOPS.

Requirements: The map

- The <u>map</u> **shall** [3] contain the intended interactions that the player can have.
    - The <u>length and width</u> **shall** [3.1] be large enough to allow for the free movement of the player and object.
        - The <u>length</u> **must** [3.1.1] be at least 100 meters.
        - The <u>width</u> **must** [3.1.2] be at least 100 meters.
    - The <u>obstacles</u> **shall** [3.2] impede the player's progress.

These requirements were made so that the playability of the map could be verified, meaning that the player and AI have the necessary amount of space to move. Every design decision went into making sure that there was enough space to freely roam the map.

Architecture

The Architecture is meant to convey the complexity and detail of the project. This involves giving a detailed description of the involved code and explaining what it accomplishes and how it interacts with the larger project.

Architecture: The Map

The map is composed of the cylinder and cube game objects found within the Unity engine. They make up the surrounding walls and obstacles within the map. In some cases, the height and width of those game objects has been tweaked to add some variance to the map. The map itself contains 25 cylinders and 12 cube obstacles, each with a radius of 12.5 meters and a height of 50 meters.

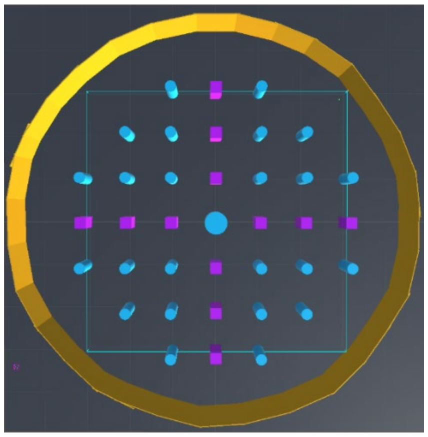
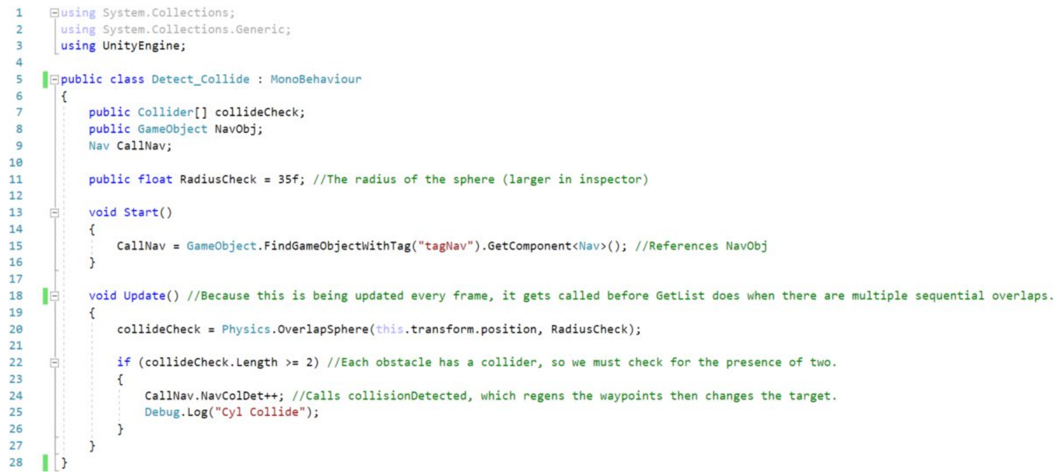


Figure 4. The Map

The Map: Code

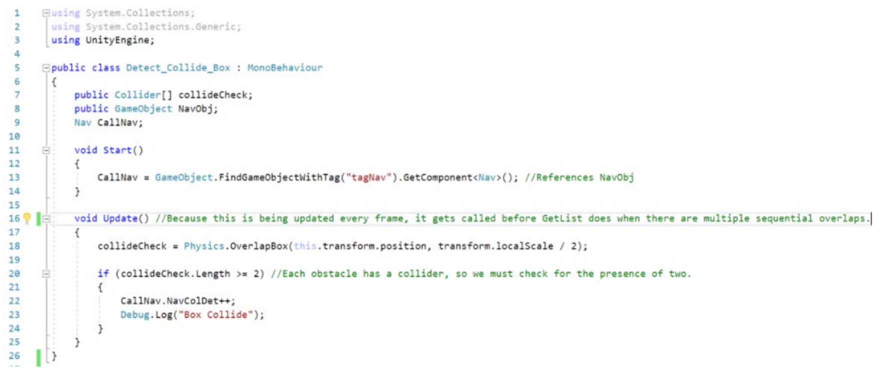
In some cases, the WFCA would place the waypoints inside of an obstacle. To correct this, each obstacle has a collider capable of detecting whether or not a collision has been made. Since the collider overlaps with the obstacle, it is necessary to check for 2 or more collisions in this case, as there is always at least one collision occurring.



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

public class Detect_Collide : MonoBehaviour
{
    public Collider[] collideCheck;
    public GameObject NavObj;
    Nav CallNav;

    public float RadiusCheck = 35f; //The radius of the sphere (larger in inspector)

    void Start()
    {
        CallNav = GameObject.FindGameObjectWithTag("tagNav").GetComponent<Nav>(); //References NavObj
    }

    void Update() //Because this is being updated every frame, it gets called before GetList does when there are multiple sequential overlaps.
    {
        collideCheck = Physics.OverlapSphere(this.transform.position, RadiusCheck);

        if (collideCheck.Length >= 2) //Each obstacle has a collider, so we must check for the presence of two.
        {
            CallNav.NavColDet++; //Calls collisionDetected, which regens the waypoints then changes the target.
            Debug.Log("Cyl Collide");
        }
    }
}
```

Figure 4. Detect_Collide.cs

To achieve this a public collider was created, which returns an array of any objects that collide with the specified collider. At the start of the script, CallNav is set to be equal to any game objects that have the assigned tag “tagNav.” Then, the collider array, collideCheck returns any objects which overlap with the sphere placed within each obstacle. If the length of collideCheck, at any point, is greater than two, then the waypoints are replaced in the function Nav.



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

public class Detect_Collide_Box : MonoBehaviour
{
    public Collider[] collideCheck;
    public GameObject NavObj;
    Nav CallNav;

    void Start()
    {
        CallNav = GameObject.FindGameObjectWithTag("tagNav").GetComponent<Nav>(); //References NavObj
    }

    void Update() //Because this is being updated every frame, it gets called before GetList does when there are multiple sequential overlaps.
    {
        collideCheck = Physics.OverlapBox(this.transform.position, transform.localScale / 2);

        if (collideCheck.Length >= 2) //Each obstacle has a collider, so we must check for the presence of two.
        {
            CallNav.NavColDet++;
            Debug.Log("Box Collide");
        }
    }
}
```

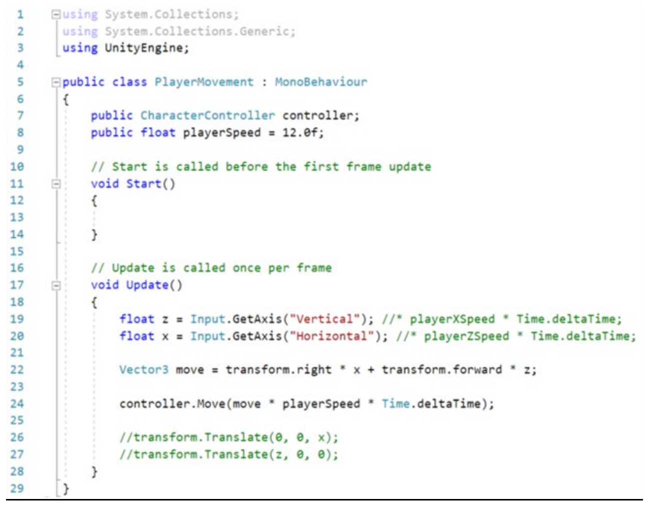
Figure 5. Detect_Collide_Box.cs

A similar pattern follows with the script Detect_Collide_Box. The appropriate game objects are referenced and, in this case, collideCheck returns a list of game objects which overlap with a box as opposed to a sphere, since this script is attached to every rectangular obstacle. Then the length of collideCheck is checked and if it is greater than two Nav will be called so that the waypoints may be replaced.

Architecture: The Player

The Player is made up of three separate entities, a parent and two children. The parent is a game object with a capsule collider and the character controller components attached to it. The first child is a capsule game object which only serves to give the player character a visible body. The second child is a camera which provides the player a view of what’s occurring in front of them. The player is capable of colliding with objects because of the attached capsule collider. Once a collision is detected the player either stops moving, if it has run into an obstacle, or generates a score, if they run into the AI.

The Player: Code:

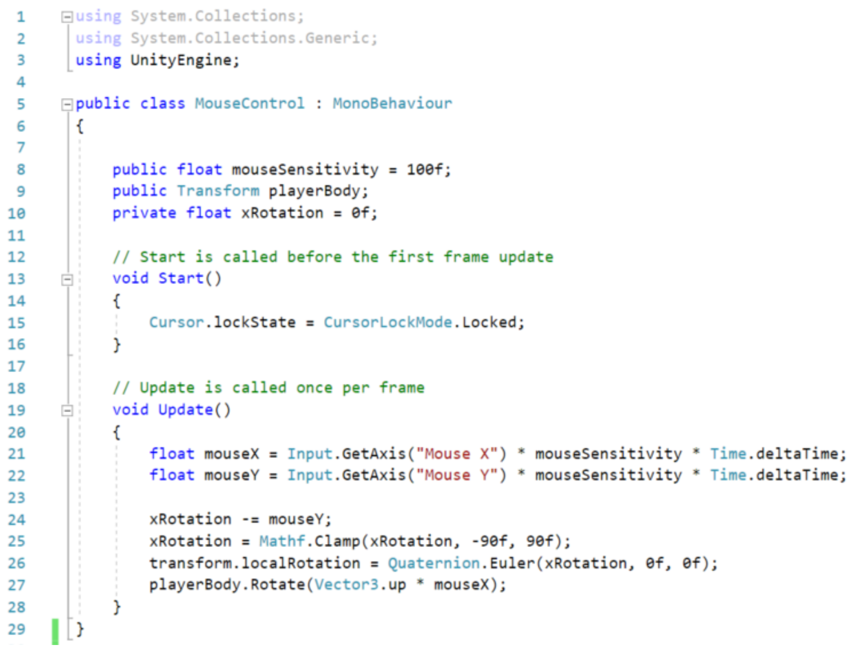


```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerMovement : MonoBehaviour
{
    public CharacterController controller;
    public float playerSpeed = 12.0f;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        float z = Input.GetAxis("Vertical"); //* playerXSpeed * Time.deltaTime;
        float x = Input.GetAxis("Horizontal"); //* playerZSpeed * Time.deltaTime;

        Vector3 move = transform.right * x + transform.forward * z;

        controller.Move(move * playerSpeed * Time.deltaTime);

        //transform.Translate(0, 0, x);
        //transform.Translate(z, 0, 0);
    }
}
```

Figure 6. PlayerMovement.cs

Results

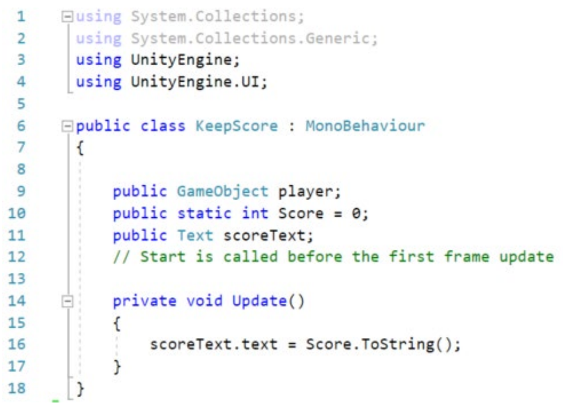
The script responsible for the movement of the player start by declaring a CharacterController, controller, and a public float, playerSpeed. Every frame, in the update function changes the values of the variables "x" and "z" by using the provided Unity input class. The vertical axis refers to the "W" and "S" keys, and by pressing the "W" key on the keyboard z is set to a positive value. Pressing the "S" key sets z to a negative value. These values are then multiplied by the transform values transform.right and transform.left. Vector3 is able to keep track of both of these transforms. This is then all placed into the controller which is responsible for the movement of the player character.



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

public class MouseControl : MonoBehaviour
{

    public float mouseSensitivity = 100f;
    public Transform playerBody;
    private float xRotation = 0f;

    // Start is called before the first frame update
    void Start()
    {
        Cursor.lockState = CursorLockMode.Locked;
    }

    // Update is called once per frame
    void Update()
    {
        float mouseX = Input.GetAxis("Mouse X") * mouseSensitivity * Time.deltaTime;
        float mouseY = Input.GetAxis("Mouse Y") * mouseSensitivity * Time.deltaTime;

        xRotation -= mouseY;
        xRotation = Mathf.Clamp(xRotation, -90f, 90f);
        transform.localRotation = Quaternion.Euler(xRotation, 0f, 0f);
        playerBody.Rotate(Vector3.up * mouseX);
    }
}
```

Figure 6. MouseControl.cs

The script Mouse Control is responsible for providing the player with the ability to change which way they're looking by moving the mouse. Input is taken in from the mouse via the Unity class Input. The rotation of the player is changed by taking in the x rotation of the player, which was previously determined by the variable mouseY. Then the camera is able to look up and down by multiplying mouseX by Vector3.up. The combination of these two inputs make it possible to look any two dimensional direction.



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

public class KeepScore : MonoBehaviour
{

    public GameObject player;
    public static int Score = 0;
    public Text scoreText;
    // Start is called before the first frame update

    private void Update()
    {
        scoreText.text = Score.ToString();
    }
}
```

Figure 6. KeepScore.cs

Keep score simply updates the text displayed on the screen with the score generated by the player when they collide with the AI. To do this is must covernt the int "score" to a string.

Architecture: The WFCA

The WFCA is made up of two primary components, the input and output. The input is responsible for taking in different waypoints and creating a dataset that the WFCA can use. The output provides an area for the WFCA to place waypoints.

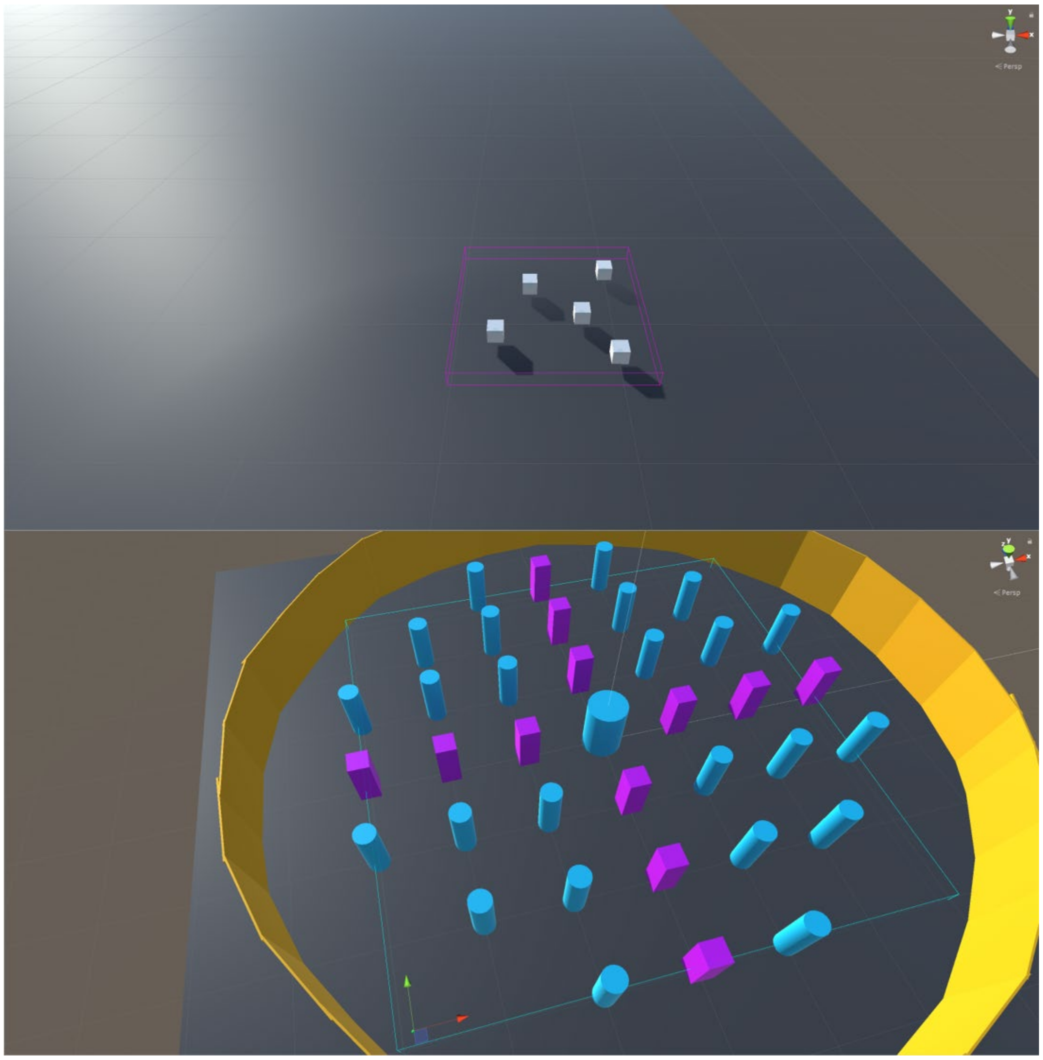
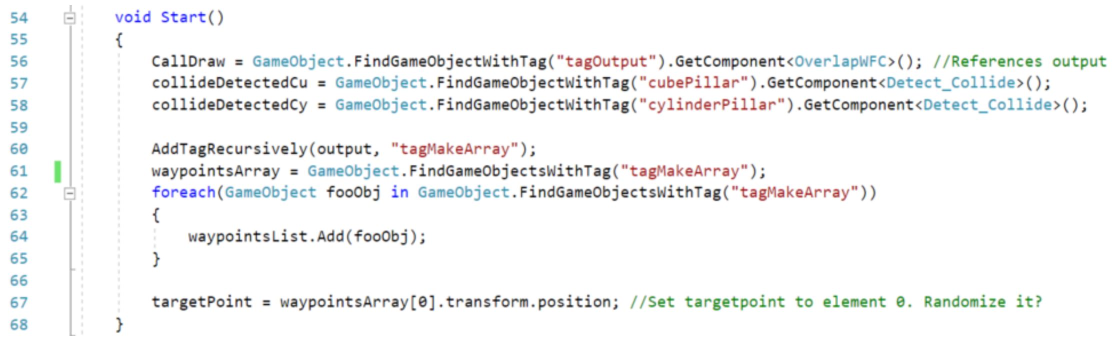


Figure 7. The input (above) and the output (below)

The input exists outside of the map and is made up of five different waypoints. The output exists within the map and is large enough to engulf the present obstacles. The input is highlighted in pink and the output is highlighted in blue.

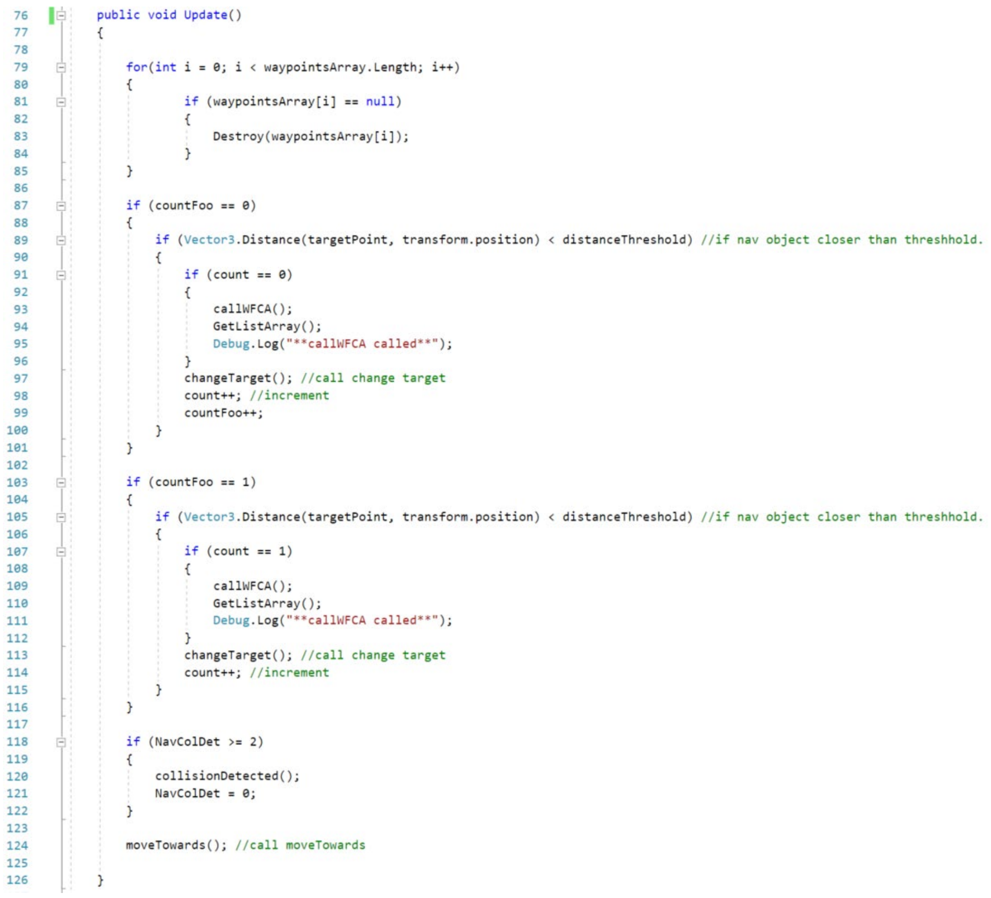
The WFCA: Code



```
void Start()
{
    CallDraw = GameObject.FindGameObjectWithTag("tagOutput").GetComponent<OverlapWFC>(); //References output
    collideDetectedCu = GameObject.FindGameObjectWithTag("cubePillar").GetComponent<Detect_Collide>();
    collideDetectedCy = GameObject.FindGameObjectWithTag("cylinderPillar").GetComponent<Detect_Collide>();

    AddTagRecursively(output, "tagMakeArray");
    waypointsArray = GameObject.FindGameObjectsWithTag("tagMakeArray");
    foreach(GameObject fooObj in GameObject.FindGameObjectsWithTag("tagMakeArray"))
    {
        waypointsList.Add(fooObj);
    }

    targetPoint = waypointsArray[0].transform.position; //Set targetpoint to element 0. Randomize it?
}
```

Figure 8. Nav.cs Start Function

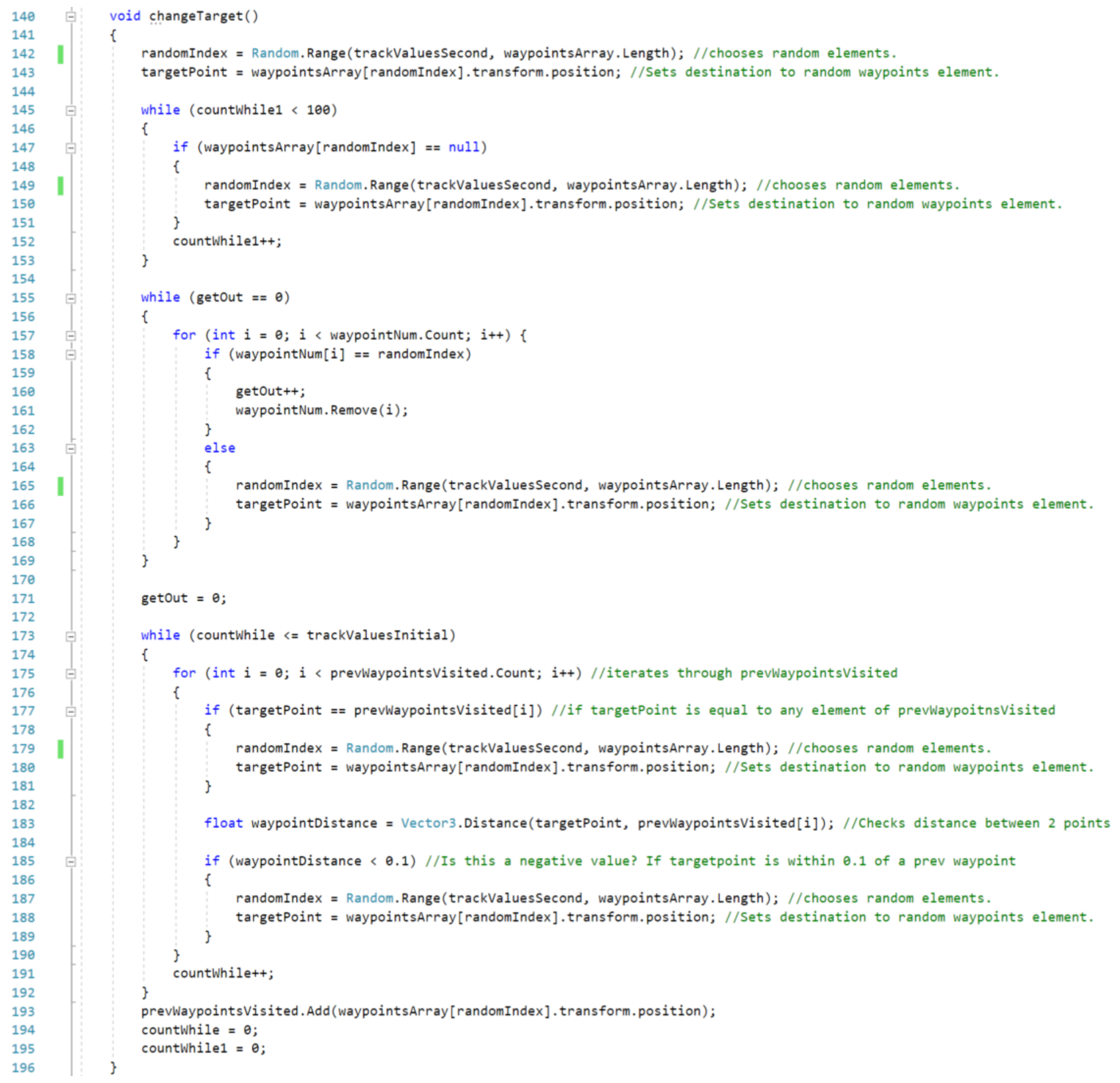
The start function is responsible for placing game objects with the appropriate tags into the corresponding variables. It also adds the tag "tagMakeArray," to the children of the output object, the waypoints. This is for the purpose of placing them into an array. Then the first target for the AI to move towards is set.



```
public void Update()
{

    for(int i = 0; i < waypointsArray.Length; i++)
    {
            if (waypointsArray[i] == null)
            {
                Destroy(waypointsArray[i]);
            }
    }

    if (countFoo == 0)
    {
        if (Vector3.Distance(targetPoint, transform.position) < distanceThreshold) //if nav object closer than threshhold.
        {
            if (count == 0)
            {
                callWFCA();
                GetListArray();
                Debug.Log("**callWFCA called**");
            }
            changeTarget(); //call change target
            count++; //increment
            countFoo++;
        }
    }

    if (countFoo == 1)
    {
        if (Vector3.Distance(targetPoint, transform.position) < distanceThreshold) //if nav object closer than threshhold.
        {
            if (count == 1)
            {
                callWFCA();
                GetListArray();
                Debug.Log("**callWFCA called**");
            }
            changeTarget(); //call change target
            count++; //increment
        }
    }

    if (NavColDet >= 2)
    {
        collisionDetected();
        NavColDet = 0;
    }

    moveTowards(); //call moveTowards

}
```

Figure 9. Nav.cs Update Function

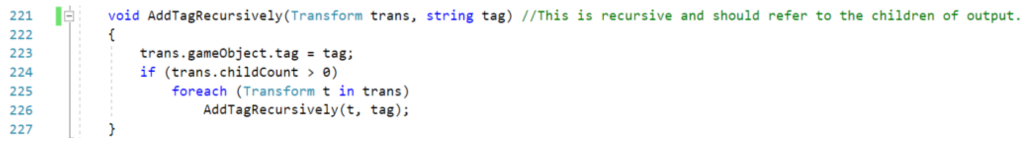
The update function is called every frame. First, it checks to see if the AI is close enough to a waypoint. Once it is, it's checked to see if the AI has visited a sufficient number of waypoints. If it did, the functions callWFCA and GetListArray are called. The target is then changed, and count is incremented. Count is responsible for keeping track of how many waypoints are visited. Then, this process repeats itself, as countFoo is also incremented. The number of waypoints colliding with obstacles is checked. If it is greater than or equal to two the waypoints are regenerated, and the variable is cleared. On every frame, no matter what, the function moveTowards is called, which simply changes the transform of the AI to move closer towards the waypoint.



```
void changeTarget()
{
    randomIndex = Random.Range(trackValuesSecond, waypointsArray.Length); //chooses random elements.
    targetPoint = waypointsArray[randomIndex].transform.position; //Sets destination to random waypoints element.

    while (countWhile1 < 100)
    {
        if (waypointsArray[randomIndex] == null)
        {
            randomIndex = Random.Range(trackValuesSecond, waypointsArray.Length); //chooses random elements.
            targetPoint = waypointsArray[randomIndex].transform.position; //Sets destination to random waypoints element.
        }
        countWhile1++;
    }

    while (getOut == 0)
    {
        for (int i = 0; i < waypointNum.Count; i++) {
            if (waypointNum[i] == randomIndex)
            {
                getOut++;
                waypointNum.Remove(i);
            }
            else
            {
                randomIndex = Random.Range(trackValuesSecond, waypointsArray.Length); //chooses random elements.
                targetPoint = waypointsArray[randomIndex].transform.position; //Sets destination to random waypoints element.
            }
        }
    }

    getOut = 0;

    while (countWhile <= trackValuesInitial)
    {
        for (int i = 0; i < prevWaypointsVisited.Count; i++) //iterates through prevWaypointsVisited
        {
            if (targetPoint == prevWaypointsVisited[i]) //if targetPoint is equal to any element of prevWaypoitnsVisited
            {
                randomIndex = Random.Range(trackValuesSecond, waypointsArray.Length); //chooses random elements.
                targetPoint = waypointsArray[randomIndex].transform.position; //Sets destination to random waypoints element.
            }

            float waypointDistance = Vector3.Distance(targetPoint, prevWaypointsVisited[i]); //Checks distance between 2 points

            if (waypointDistance < 0.1) //Is this a negative value? If targetpoint is within 0.1 of a prev waypoint
            {
                randomIndex = Random.Range(trackValuesSecond, waypointsArray.Length); //chooses random elements.
                targetPoint = waypointsArray[randomIndex].transform.position; //Sets destination to random waypoints element.
            }
        }
        countWhile++;
    }
    prevWaypointsVisited.Add(waypointsArray[randomIndex].transform.position);
    countWhile = 0;
    countWhile1 = 0;
}
```

Figure 10. Nav.cs changeTarget Function

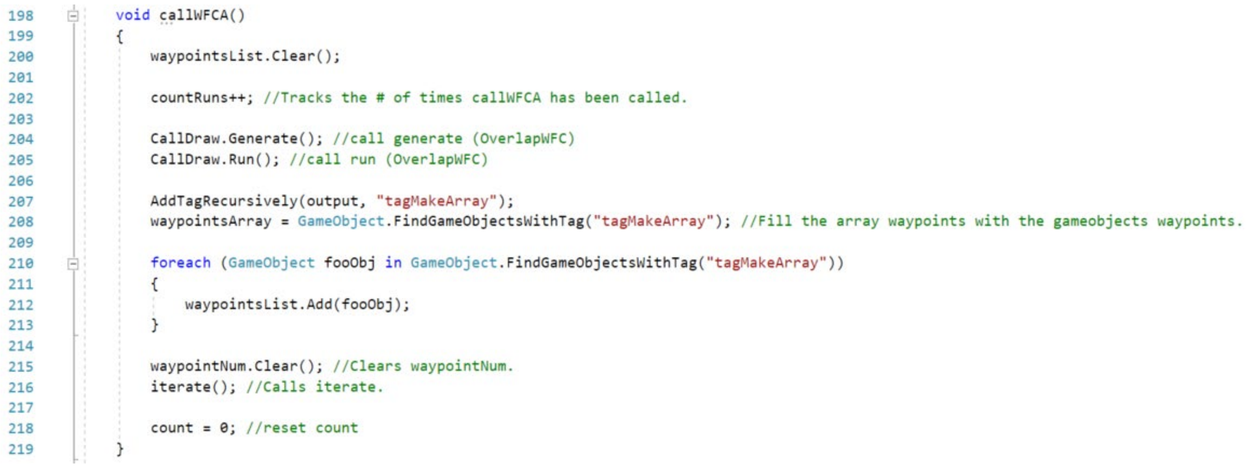
ChangeTarget is responsible for randomly selecting a waypoint. It does this by generating a random number and then setting targetpoint to the element of the waypoints array. The remaining lines of code could not be properly implemented, but were ultimately meant to prevent the AI from visiting waypoints that it had previously visited.



```
void AddTagRecursively(Transform trans, string tag) //This is recursive and should refer to the children of output.
{
    trans.gameObject.tag = tag;
    if (trans.childCount > 0)
        foreach (Transform t in trans)
            AddTagRecursively(t, tag);
}
```

Figure 11. Nav.cs AddTagRecursively Function

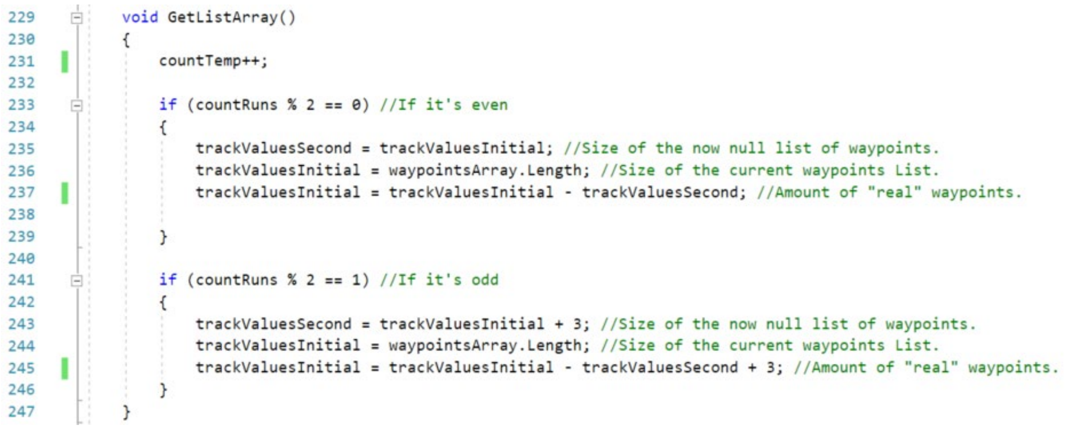
The AddTagRecursively function is what adds the tag to the waypoints. It does this recursively by attaching the tag to the transform of the game object. It then checks if there are any other children of the parents, trans.



```
void callWFCA()
{
    waypointsList.Clear();

    countRuns++; //Tracks the # of times callWFCA has been called.

    CallDraw.Generate(); //call generate (OverlapWFC)
    CallDraw.Run(); //call run (OverlapWFC)

    AddTagRecursively(output, "tagMakeArray");
    waypointsArray = GameObject.FindGameObjectsWithTag("tagMakeArray"); //Fill the array waypoints with the gameobjects waypoints.

    foreach (GameObject fooObj in GameObject.FindGameObjectsWithTag("tagMakeArray"))
    {
        waypointsList.Add(fooObj);
    }

    waypointNum.Clear(); //Clears waypointNum.
    iterate(); //Calls iterate.

    count = 0; //reset count
}
```

Figure 12. Nav.cs callWFCA Function

callWFCA is responsible for calling the WFCA and generating the waypoints. The waypoints are replaced and then the tags are added to the new waypoints.



```
void GetListArray()
{
    countTemp++;

    if (countRuns % 2 == 0) //If it's even
    {
        trackValuesSecond = trackValuesInitial; //Size of the now null list of waypoints.
        trackValuesInitial = waypointsArray.Length; //Size of the current waypoints List.
        trackValuesInitial = trackValuesInitial - trackValuesSecond; //Amount of "real" waypoints.

    }

    if (countRuns % 2 == 1) //If it's odd
    {
        trackValuesSecond = trackValuesInitial + 3; //Size of the now null list of waypoints.
        trackValuesInitial = waypointsArray.Length; //Size of the current waypoints List.
        trackValuesInitial = trackValuesInitial - trackValuesSecond + 3; //Amount of "real" waypoints.
    }
}
```

Figure 13. Nav.cs GetListArray Function

This function is responsible for making sure that the appropriate waypoints are being selected when changeTarget is called. This needs to be done because waypoints which are deleted remain in the waypoints array until the next time the waypoints are regenerated. It has to take different actions on even and odd generations, as things that are not waypoints are assigned tags by AddTagRecursively. To ensure that they are not selected, and as an extension the deleted waypoints are not selected.

Results

The code, when executed, runs a program which is able to meet the previously stated requirements. As a result of the map being spaced as it is, ample space for both the player and AI is provided. The space between each obstacle is 50 meters, many times larger than the width of the player or AI's collision box. This means that neither are constrained in any way which might be frustrating for the player.

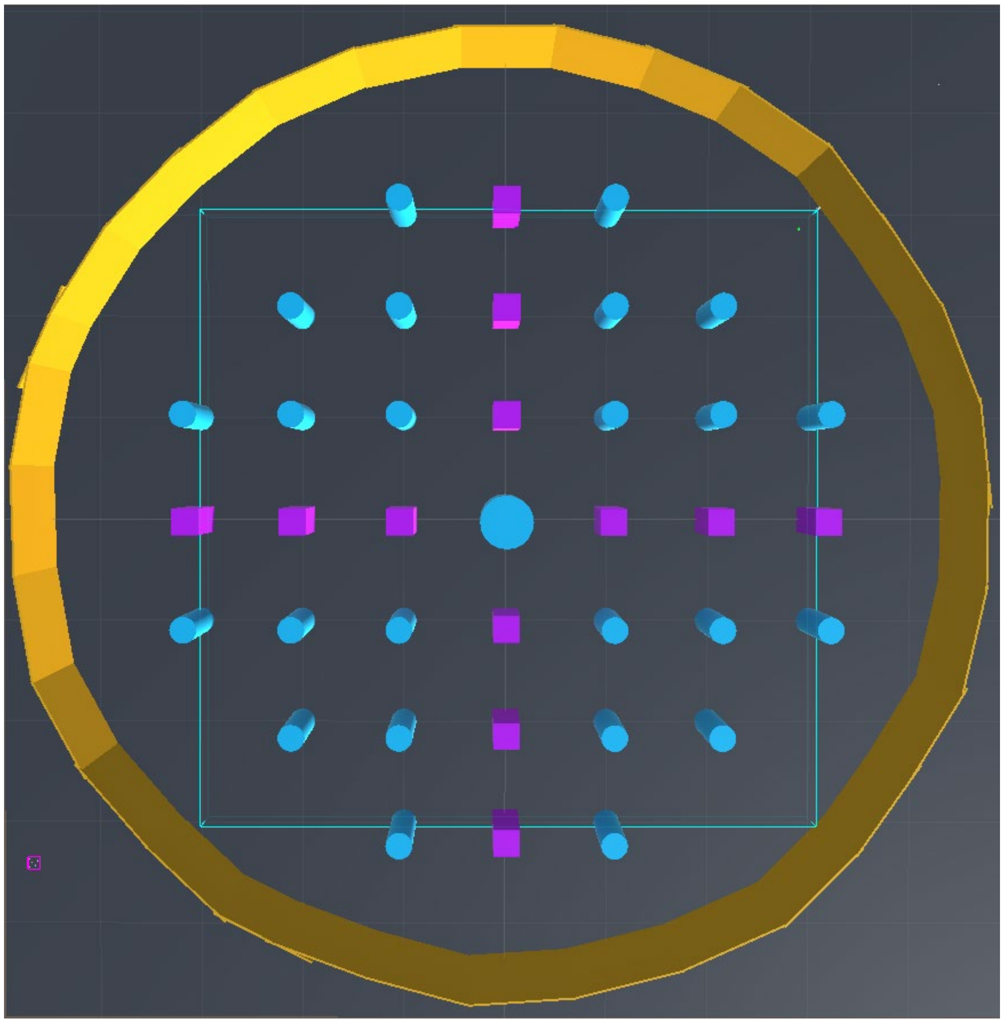


Figure 14. The Map as seen from above. In the top right corner, you can see a green dot which represents the AI.

The player is able to move around the map as the user wants it to. It moves at a speed adequate enough so that it is able to keep up with the AI. The player can move left, right, forwards, backwards or any combination of those movements. They are also able to change where they are looking by moving the mouse. These movement controls enable the player to traverse the map as they would like to. Additionally, the player is able to collide with the AI and obstacles for the purpose of interacting with the map. Colliding with the AI makes it so that a score is generated, whereby colliding with obstacles, the player's ability to collide with the AI is hindered, making the game more challenging.

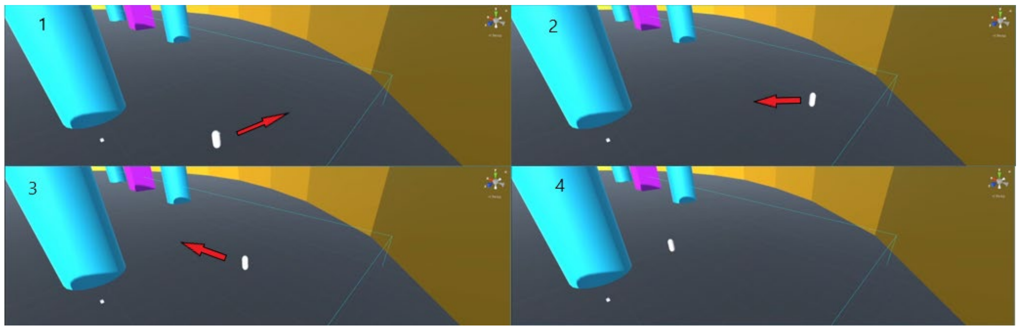


Figure 15. The Player moving in different directions as it is being guided by a user.

The AI is completely guided by the WFCA. The WFCA takes the input and creates a dataset. That dataset is then used to create different outputs. Those outputs consist of a variety of different ways in which waypoints are arranged. It is these waypoints that guide the AI along the path. The AI is able to reach a waypoint and then change its target, this is how the path is formed. Once the AI has reached enough waypoints the output is regenerated, yielding new paths for the AI to choose from. This is all done so that the deleted waypoints till present in the array do not interfere with the navigation of the AI, as they are not looked at by the random number generator which determines which waypoint in the array the AI will move towards next.

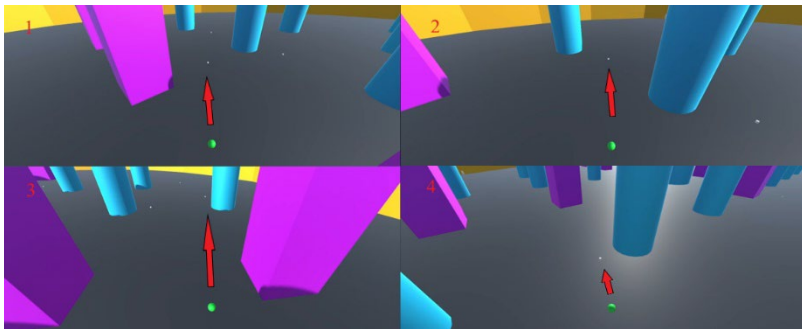


Figure 16. The AI moving from waypoint to waypoint which have been placed by the WFCA

Conclusion

Discussion

The future of this project lies within being able to combine the dynamic navigation of an AI with the ability to dynamically generate a map by using the wave function collapse algorithm. An application could simultaneously generate a unique path for an AI to traverse as well as a dynamic map which would be completely unique. This would enable the possibility for entire applications to become dynamic, creating new experiences for a player should this technology be used on something like a video game. Additionally, the problem of repetitive paths being taken has been solved as by waypoints dynamically regenerating a new path is created every time.

Summary

By allowing the Wave Function Collapse Algorithm to plot waypoints dynamically and continuously, different applications can utilize this to allow an AI to traverse a map in a unique way. The map is able to contain the intended interactions between the player, AI, and obstacles. The player is able to interact with the AI and map by moving using the provided Unity engine inputs. Finally, the AI is guided by the wave function collapse algorithm and as a result can create dynamic paths.

References

Parker, J. (2017). *Generating worlds with wave function collapse*. Generating Worlds with Wave Function Collapse - PROCJAM Tutorials. Retrieved February 28, 2022, from https://www.procjam.com/tutorials/wfc/#:~:text=Wave%20Function%20Collapse%20(WFC)%20by,patterns%20from%20a%20sample%20image.&text=The%20traditional%20approach%20to%20this,to%20alter%20your%20game%20map.

selfsame. (2016, October 29). *Unity-wave-function-collapse by selfsame*. itch.io. Retrieved February 28, 2022, from https://selfsame.itch.io/unitywfc