RUNNING FROM WATER

Purpose	This document should explain the aim of the AI agent, the implementation choices and the environment, and all behaviours and techniques used to obtain the final result.
Creation Date	29/07/2021
Current Owner	Alice Sansoni
Last modification	11/09/2021
Software	1. **Development Software:** Unity 2020.3.3 2. **Organization Software:** Microsoft Word Version 2102 3. **Environments:** Windows 10 Home version 20H2
Data Storage and Backup	All data are saved on GitHub with an online backup repository and a copy of it on an external disk. Link to GitHub here: https://github.com/AlySansoni/AI4VGProject.git

Revision History

When	What
29/07/2021	• Initialization of all files starting from the folder used during classes • Change of terrain asset and texture • Insertion of the agent and destination on terrain with relative scripts • Creation of a first NavMesh Map to see if everything works
31/07/2021	• Insertion of the plane of water with NavMesh obstacle component and relative scripts • Change of the NavMesh adding water obstacle
18/08/2021	Added GameOver screen
19/08/2021	Modified and working GaveOver screen
23/08/2021	• Reviews and modifications of assets and scripts • Skybox added
28/08/2021	• Modified agent behaviour with speed changing in time • First cleaning of the project • Added documentation
29/08/2021	General revision and cleaning of unused files
01/09/2021	Documentation and project review
11/09/2021	Camera rotation added; documentation review

Introduction & Project Specification

Start from a map created using Perlin Noise. This map is partially covered by water whose height varies in time. This height goes from 20% to 80% of the height of the whole terrain, so that it is never completely underwater nor dry. You have to compute the path that an agent has to do to go from one corner of the map to the other using A*, checking for the water level and avoiding being submerged by it.

Goal of the AI: to find a path for the agent to move from its initial position to the destination, avoiding at runtime those parts which are cover by water.

Moving technique in the environment: A* through Navigation Mesh

Overview:

Starting from the Perlin Noise map obtained during classes, the most appropriate amplitude value (0.2 after tried different amplitude values this gave the best result) has been chosen to have a believable land with not too much slope to be travelled by the agent. Then a plane was added, with material simulating water flow, going up and down in the allowed range (between 20 and 80% of the terrain height). It goes with a fixed speed, following the uniform linear motion. After that, an agent spawner was added to instantiate the destination flag in a random corner of the map and to place the agent in a random place on the other side of the map to the flag. To make everything work, a NavMesh map was created using the terrain as a walkable area and water as a non-walkable area, adding also a NavigationObstacle component in the GameObject. The agent, as a NavMesh Agent, can walk along the map to reach the destination, waiting in the walkable part of the map when the flag is underwater and so unreachable. A third-view camera following the agent was later added; the player can use the right or left arrow (or D, A) on the keyboard to rotate the camera and have a different view of the scene. When the agent reaches the target, a Gameover canvas appears on the screen, letting you choose if you want to start again, re-instantiating the whole scene, or exit the game.

Designing AI model

The AI needed in this project is a single individual, with not a high degree of realism. The physics simulation is needed to make the agent walk the map but, as the pathfinding process, it is already included in the Unity tool of the NavMesh Map. The character is only influenced by the water plane that reduces its possibilities to reach the destination, cutting at runtime some parts of the map.

There is no need for any specific decision-making system because the agent theoretically has only three states: chase the target, wait in a safe position if the target is unavailable, and stay in the position when the target has been reached. This last part is handled in the FixedUpdate method of the script attached to the agent, changing its speed when it's close enough to the destination. Except for this, there isn't any decision to take because everything, as already said, is the responsibility of the NavMesh Map tool. The only thing done is to set the Transform of the target as the agent destination and then the NavMesh agent component is put in charge of the movement and the decision about the best path to reach the destination, including path recalculation.

In this project, there isn't a large-scale game mechanism to be understood or a higher goal to achieve so there is no need for any specific tactical or strategic AI.

NavMesh Map and components

The main aim of the project is to set up the NavMesh in the right way. This means to let the agent believably walk on an unregular terrain, reacting runtime at the change of water level. Thanks to the Unity tool Navigation, there is no need to use a script about movement nor steering behaviour. Any specific pathfinding script is required, too. The terrain is automatically translated into a graph exploiting the polygonal structure of the meshes inside the game itself, using their vertices to connect all areas. Each position is later associated with the polygon containing it. The tool as inputs needs those components that will later influence the map, so the terrain and the water plane. They must have the Navigation Static flag set on active and be marked as a walkable or not walkable or jump area, according to the function they will assume. Before baking the Map, we have to set up the agent parameters like its radius, its high, the maximum slope it can climb, and the step height it can jump. Then an agent is needed to walk on it, and that's why the Nav Mesh Agent component was added to the capsule in the scene. In this component all control parameters were set, as the agent type [humanoid], all steering values (as the speed and angular speed, the acceleration), and the stopping distance from the destination (set to 0 because of the distance value needed to active the canvas). The initial speed is set to 10 but is then modified during the game (see *GoSomewhere.cs* script for more) while the angular speed is always set to 80. They both are parameters chosen after a few tests, based on what was the best visual result, combined with the best value to use for the aim of the project. With the auto-braking flag, the agent will slow down when reaching the destination. Then there is the part about obstacle avoidance and the quality with which you want to check collisions. All pathfinding needed is in the Pathfinding subcomponent. With auto repath enabled the agent will attempt to acquire a new path if the existing one becomes invalid. The area mask was set to only walkable so that the agent will calculate its path only along that area type.

The main objective of this project is to disable those parts of the map that are under the level of water. To do that a NavMesh Obstacle component was added to the GameObject “Water”. Its shape was set to Box, creating a cube box of size 50 with the center set to -25. This lets you have the top side of the box at the same level of the plane, disabling all those points which are under that height. With the Carve flag set to active, the plane cuts a hole in the NavMesh around it. The Move Threshold value is the limit after which Unity treats the obstacle as moving while the Time to Stationary parameter is the time (in seconds) to wait until the obstacle is treated as stationary. The carve only stationary flag was disabled because when enabled, the obstacle is carved only when it is stationary, but here it is needed that the object continuously changes the NavMesh map while moving up and down.

Game Objects

Terrain

This GameObject is different from all the others because it is usually extremely huge and so it doesn't scale as usual. To obtain a terrain the starting point is a heightmap: a grayscale image where black represents the lowest points while white concerns those points that are the highest in the map. A gaussian 2D texture was taken as a reference. To have a more natural terrain, in this project was used "Perlin Noise", a technique that perturbates values and so creates some noise. It is a number to add to the initial information we have in order to make a regular surface bumpy. Perlin noise is a mathematical model that associates to each point of a multi-dimensional space a numerical value. Neighboring numerical values inside the multi-dimensional space are bounded in gradient so not too far from each other. To create the map a gradient-based terrain generation was used, which generates slopes (exploiting the gradient noise) and calculates heights. Those heights are obtained through interpolation of the four neighboring lattice points. Here Bicubic Noise was used for the slope function, for a better and smoother result. Also, a Fractal terrain is needed because of too regular micromanagement of the terrain itself. An easy way to produce it is to take the single-scale random terrain method and run it several times, at multiple scales, and then add all of them together. Here were used harmonics with the same noise but doubled frequencies.

The referenced script is *PerlinWalk.cs*

Agent Spawner

It is an empty gameObject used as a reference for the script *AgentSpawner.cs*. First, it spawns the destination flag in one of the four corners of the map, checking that it is a valid position to water flow. Then, following the same conditions, it places the agent on the opposite corner of the map. This map is ideally subdivided into four parts, where the game objects can be instantiated in a certain range (between 20% and 80% of the terrain height because of the minimum and maximum water level).

Agent

This GameObject represents the AI agent moving around the map. It has been used a pink capsule that acts as a Rigidbody, following the laws of physics and automatically detecting collision with other objects. X and Z rotations have been frozen to have a more believable behaviour on the terrain. It has also a capsule collider and a NavMesh agent component to make it walk on a NavMesh map (see NavMesh Map and components for more). Using the reference script *GoSomewhere.cs* it has to reach the same position as the destination, waiting for a walkable path. When it manages to do it, a game over screen must appear.

Flag (Destination)

It is a simple GameObject with no scripts attached. It only has an animator component to make it seem that wind is moving it. It is spawned in a random place on the map by the AgentSpawner and its position is passed to the agent so it can reach it.

Water Plane

It is a premade asset found in the Asset Store, with an attached script for the rendering (*WaterBasic.cs*). Thanks to this code, it simulates flowing water, making it appear more realistic. It has a RigidBody component but with the IsKinematic flag checked. This means its position update is fully controlled by the script (*Water.cs*) and it is not affected by physics. In the NavMesh it is marked as a Not Walkable area and it also has a NavMesh Obstacle component, because it is what the agent has to avoid (see **NavMesh Map and components** for more). This GameObject is a plane going up and down with a fixed speed rate in a predefined range of heights (20%-80% of the whole terrain). The only exception is for the first time when water starts growing from an initial position of 3 on the y-axis.

Canvas & Event System

This GameObject is made to be shown when the Agent reaches the target. So, at the beginning of the scene, it is disabled and is later set active by the GoSomewhere script of the Agent GameObject. It is composed of:

- A black image as the background for text and buttons.
- Two text boxes telling the player he won and the target was reached.
- Two buttons:
    - Restart: to start again another simulation with different starting positions.
    - Exit: To exit the game.

The EventSystem GameObject is automatically generated by Unity to handle buttons in the scene and related inputs.

Reference script: *GameOverScreen.cs*

Main Camera

The game camera has a third-person view and was set to follow the agent with a certain offset with respect to the agent. The player can also use the right or left arrow (or D, A key) on the keyboard to rotate the camera around the y-axis. It doesn't use the hierarchy in the scene to try to avoid the quick rotation of the camera that happens when the agent is blocked and rotates on itself. This would create an annoying vision for the player.

Reference script: *CameraFollow.cs*

Scripts

PerlinWalk.cs

This is the script took from the lecture notes about the generation of the terrain. It needs a 2D texture of the base Heightmap as a reference: in this case, a gaussian one. There is the possibility to set the “MakeItFlat” flag at true and generate a flat terrain instead of a random one. This could be useful because the terrain is an asset, so every change to the structure will persist between runs. In the *Start()* method, it is called the *GenerateLandscape()* coroutine to obtain the resulting Perlin Noise terrain. This function, if the "flat" flag isn’t selected, takes points from the heightmap. It uses the terrain resolution to choose the number of points to generate. The *Amplify()* method takes the extracted points and multiplies them for a given amplitude to make them higher. Then the *CreateNoise()* function is in charge to add the random noise value. It uses a lattice grid with a random gradient at the lattice points while all heights are set to zero. To find the height values at non-lattice points you have to look at the four neighboring lattice points. Their contribution is evaluated using a dot product between the gradient vector and a vector which is drawn from the lattice point to the current point. You will obtain four heights to linearly interpolate, first horizontally and then vertically. One of the linear interpolation parameters is the call at the *Slope()* method. It computes the Perlin Bicubic Noise, which is more computationally intensive but gets a smoother result. In the end, all points to be accepted need a renormalization. The “Add Harmonics” flag is also active, so the process written above (Amplify + Create Noise) is replicated with halved the resolution and the amplitude, this means doubling the frequencies, to create a fractal terrain.

AgentSpawner.cs

This script is in charge of spawning the agent and the destination in a random position on the terrain given as a reference. It has to avoid those heights that are always under the level of water. In the *Awake()* method, at first two numbers between 1 and 2 are extracted randomly: one to choose the corner of the map on the x-axis and the other on the z-axis. According to the result, those values are passed to the *GetRandomPosition()* function to instantiate the destination in a random position. Once the target is set up, the same process happens for the placement of the agent, but on the opposite corner of the terrain. The random position comes from a random x and z coordinate in a range of around 35% of the map on each axis [so from 0 to 35 or from 65 to 100 of the x or z length], in one of the four corners according to the xZone and zZone parameters. Those values were chosen to create the furthest distance possible between the two Gameobjects. After that, with the function *terrain.SampleHeight()* (given as input a Vector3 with the world coordinates) samples the height of the terrain relative to the terrain space. This process is put inside a do-while to check if the resulting triplet is valid or not, according to the minimum and maximum water level. The maximum water level is also checked because if the flag would be spawned there, it would never be overcome by water and this is not coherent with the aim of the project.

WaterBasic.cs

Premade script found attached to Water asset in the store. It was made for the rendering of the water plane. It uses the Wave speed and Wave scale obtained by the specific shader to recreate wave effects on the plane.

Water.cs

This script manages the movement of the water plane. It needs as reference: the plane itself and the terrain to obtain the relative heights. Here are also set the values for the range in which the objects can be instantiated in the map. In the *Reset()* method, called in editor mode, are set the reference values and water velocity on the y-axis. With the method *Start()* water plane is initialized at a height of 3. From here, with the method *FixedUpdate()* [chosen because it has the update frequency of the physics system] the current height of the plane is updated. Until it reaches the maximum water height, its position is increased with a fixed velocity, multiplied for a fixed delta time. When the limit is overcome, water position starts to decrease of the same quantity, until the minim water level is exceeded. This is an infinite loop, managed by the Boolean condition “growing”.

GoSomewhere.cs

This script is attached to the AI agent. It needs as reference: the destination, to get its position; the canvas, because it is in charge of active the Gameover screen; the water because it has to simulate an agent watching when the destination is reachable, using the y-transform of water. Then, every time the flag emerges again from water it sets the agent destination again, to be sure it immediately starts moving. In the *Start()* method, the agent's initial speed, and its destination are set, checking where the target was spawned. The *FixedUpdate()* method checks: if the target is over or under the level of water, modifying the Boolean condition “underWater”. When it is "True" and the target is reachable, it sets again the agent destination. This code also needs to check if the destination was caught up, looking at the distance between the agent's current position and the target. If so, the agent's speed and angular speed are set to zero and the *GameOver()* method is invoked. It just has to set the canvas to active.

To speed up the run and make sure the agent can reach the destination before the water rises again, a speed increment of 5 was added every 30 seconds, with a maximum speed limit of 20.

GameOverScreen.cs

This is a very simple script for the management of the buttons in the canvas. When the Restart button is clicked the *RestartButton()* method calls the Scene Manager to load the scene. This makes the whole simulation start again. When the Exit button is clicked, instead, the *Application.Quit()* method closes the game, with a debug message inside the editor to see if everything worked well.

CameraFollow.cs

This script is attached to the main camera. It takes the transform of the target as a reference for the movement of the camera. The *Update()* method changes the offset value using the user input. It is used the built-in method *Quaternion.AngleAxis()* takes an *angle* and an *axis* as parameters and returns a rotation of *angle* degrees around *axis*. To obtain the angle, it reverses the keyboard user input (-1 or +1) multiplying it for a given rotation speed. This speed was chosen by attempts for a smooth result. Then, the *LateUpdate()* method is used to be sure that the agent position is updated so that there is no competition between the two updates. To obtain a smoother result, the next position of the camera is obtained with a Lerp, so linear interpolation, between the current position and the desired one, using a predefined speed.

Conclusion & Future Improvement

This is a quite simple AI, working for most cases but not always. It starts from the fact that the terrain isn't regular so it was difficult to create a NavMesh map that could perfectly cover all areas. That's why there are a few holes in the map where sometimes the agent gets stuck. In the *GoSomewhere.cs* script there are a few commented lines about setting as agent destination the highest point in the map when the destination is under the water level. This was done to try to avoid the scattering behaviour the agent has when is trying to reach the flag but water is growing up and is very close to the agent, sometimes also stopping the agent in a dead area. For this reason, sometimes it happens that the agent goes underwater for a while ("drowning"). The problem that comes from this solution is that so the agent stops quite far from the target. So, when this one is located not much higher than the minimum water level, the agent will never be able to reach it before it goes underwater again. Something that could be combined with this idea could be to reset the agent destination after water is lower than a certain threshold, but still before the destination is available. One solution to this problem could make the agent going faster when it is colliding with the water plane, to run away from it and reach a safe position before the surrounding parts of the map become unavailable.

Another future change that could be done is randomly change the rising or falling behaviour of water before being at the minimum or maximum level, maybe after a certain number of seconds. Or changing water speed dynamically.

Something that could be also improved is the smoothness of the camera. This is because when the agent, for any reason already explained, has a scattering behaviour, the camera does the same and this could be annoying for who is watching.

There are also some unlucky circumstances where the agent will never be able to reach the destination. It is mainly because of the randomness of the flag position, which could be too close to the minimum water level with respect to the position the agent has when it starts walking again. There are also a few corners of the map where the minimum height condition is respected but all surrounding areas will always be underwater and this will lead to a dead-end area. To overcome or limit this problem, maybe it could be added a check of the neighboring heights: if in a certain range they are all under the 20% the whole area becomes unavailable.