HELLO WORLD
Development Log

by Vincent Reynier

*This document contains information on the development process of my game ‘Hello World’, playable/dowloadable on my itch.io page. It’s quite a long read so I have highlighted certain chapters which should give you a rough idea of the content of this document. If you should read only one chapter I would recommend **2.3.3**, in which I analyze the performances of two random level generation algorithms. Click on the headings below to access each chapter directly.*

1. Project Overview

Hello World is a **hexagonal-tile based management game** in which the player has to build and expand a colony of settlers in a natural environment. The game is **turn based**, each day the player gets to assign each villager to a specific task. The main goal of the game is to gather enough resources in the environment to sustain the colony and expand it over time.

The two game modes are Campaign and Random Level.

1.1. Campaign Mode

Campaign Mode consists in a series of handmade levels, in ascending order of difficulty. The first few levels act as a tutorial to the game while the more advanced levels require the player to deal with different scenarios where they will have to adapt to their environment and properly plan ahead.

Each Campaign level has its own set of **objectives** (gathering a certain amount of a particular resource, build a certain number of houses, survive for x days…). The level is beaten when all objectives are completed, and failed if all villagers die.

1.2. Random Level Mode

In this mode a level is randomly generated and one house is placed, with two villagers. Objectives are also randomly created based on the level layout and the available resources, and whenever completed another one is created to replace them. In the current version of the game Random Level Mode is endless.

2. Random Level Generation Algorithm

2.1. Generation Method

The player picks a horizontal and vertical size for the level, or one of the available size presets. The Generation process starts by filling the entire level with Grass tiles, and picking a random position for the starting tile, where the first house will be built.

Since the generation process is going to involve replacing a lot of tiles I have decided to **use a simple int[,] array to represent the level during generation**, instead of instantiating and destroying BaseTiles prefabs (BaseTile is the base type of all tiles in the game). Only when the level has been entirely generated do I actually convert this int array into a proper BaseTile array.

I have implemented two different generation method for tiles : <u>Random</u> and <u>Natural</u>. A different method can be picked for different types of tile. For example we can generate Mountains randomly but Water naturally, and each combination of methods is going to give a different feel to the level.

It is also possible to change the order in which we spawn the different types of tile. The default order is Mountains-Water-Forests-Bushes-Stone but any order will result in a playable level. And each order results in a very different kind of level. However we need to finish spawning tiles of a specific type before starting with another. We can’t interrupt Mountain generation to spawn Forests, before going back to Mountains.

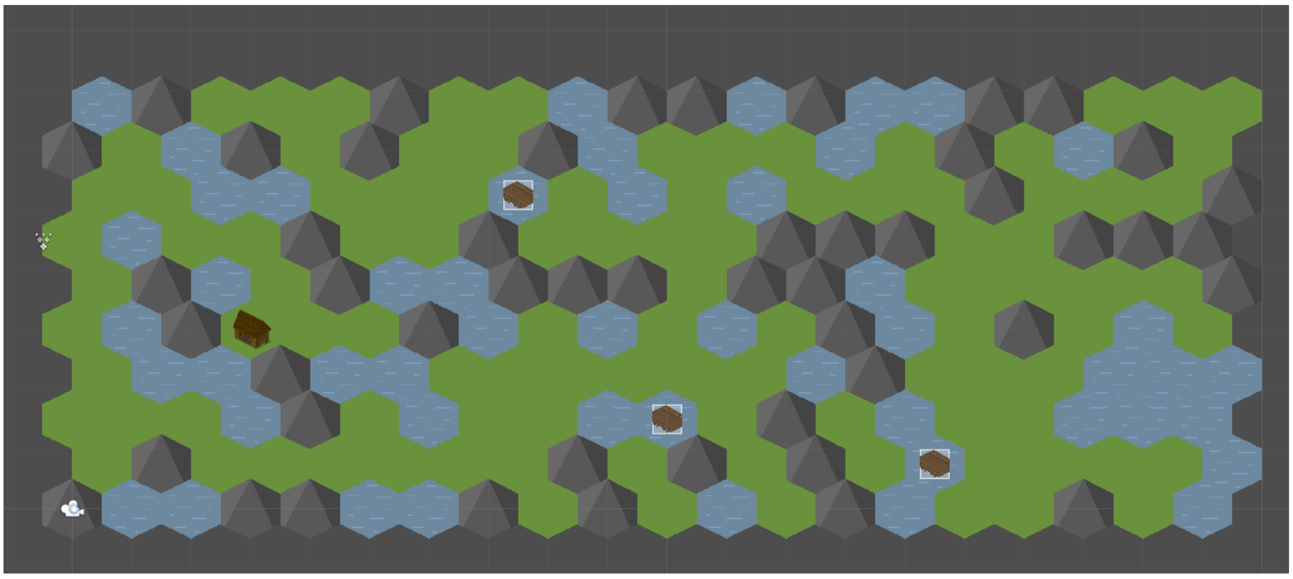
With both Random and Natural generation a **pathfinding algorithm** is used to make sure that tiles are accessible. That process is fairly straighforward for Mountains – they are totally impassable so no mountain is ever allowed to block a path – but much more difficult when it comes to placing Water tiles. Those tiles are impassable by default but the player can build a bridge across them to access the tile on the other side. See The Water Problem for more details.

For each type of tile we decide in advance how many are going to be spawned. By default each type is given the same number of tiles : we simply divide the total number of tiles in the level by the number of types, minus a few tiles to be left as Grass. When generating each type of tiles we keep placing them until the desired number has been placed, or no valid tile remains. There is a slight exception to this rule in **Corrective mode** (see the appropriate section for more details).

2.1.1 Random Generation

With this method we simply pick an available Grass tile on the level and replace it with the type of tile we are creating (say Mountains). If this type is impassable (Mountain or Water) we test if this new tile blocks access to any part of the level. If it does we turn it back into a Grass tile. Since we are using a temporary int[,] array during generation replacing a tile in the level simply requires changing the type value of its position. No instantiation or destruction of GameObject necessary.

As often with procedural generation, **true randomness tends to make levels feel very similar.** Even though each individual tile position is going to be random, the level as a whole is often going to feel the same, with a lot of scattered obstacles and resources. As a result, and while it is still satisfactory for Bushes for example– which are naturally spread out –, I decided to implement a different generation method for Mountains and Water, that would create more realistic-looking formations. But before we look into it, here is a quick note of a little optimization I implemented for the Random Generation.


Random Mountain & Water Generation

2.1.1.1 Random Generation Optimization

2.1.1.1.1 List of Available Grass Tiles

Initially, for each new tile, the Level Generator was picking a totally random position on the level. If that position was Grass it would keep going, otherwise it would go back and try a different position. This brute-force approach works fine enough at the start of the generation process, when most tiles are still Grass, but is very inefficient later on. The obvious optimization consisted in **declaring an AvailableGrassTiles list**, initially containing all tile positions (except the starting house). When a grass tile is replaced by another type it is taken out of this list, and when picking a position for the next tile to spawn the Level Generator would get a random element in the list, knowing that this position is a valid one.

This simple method doesn't however work when working in Natural Generation, for reasons that will be explained in the next section.

2.1.2. Natural Generation

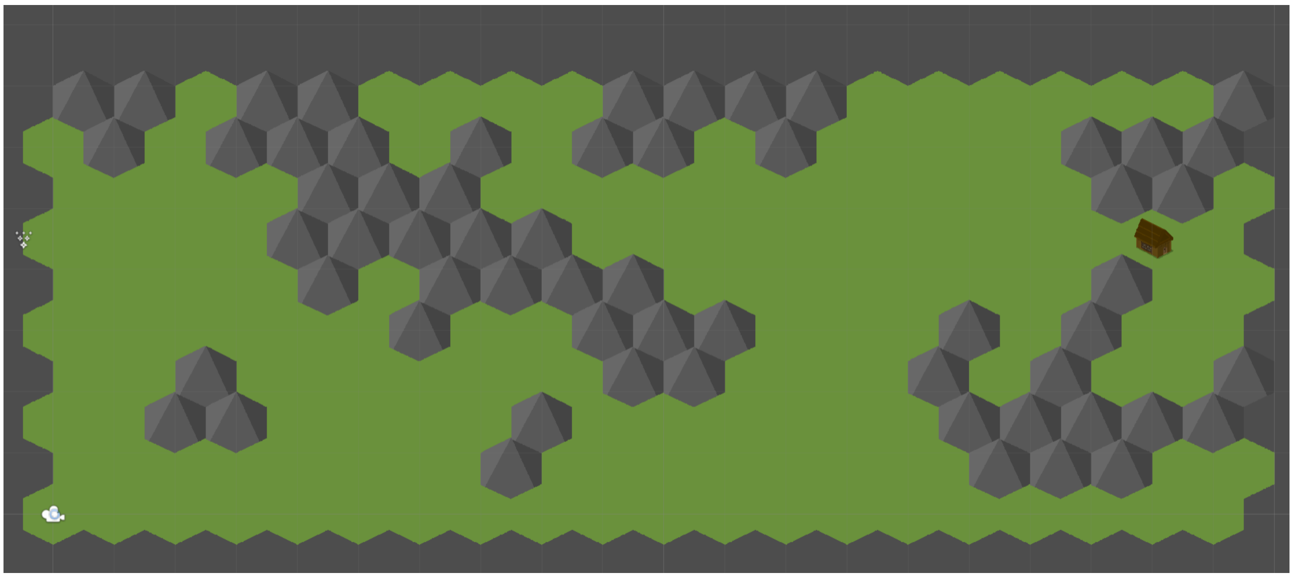
The idea of Natural Generation is to create realistic-looking levels, by spawning **mountain ranges and rivers/lakes**, instead of isolated mountains and water tiles. The process is essentially the same for Water and Mountain tiles (in the rest of this section I am going to use the term 'chain' to describe a continuous formation, independently of the type of tile we are creating). Instead of always picking a random available grass tile, the process works as follows :

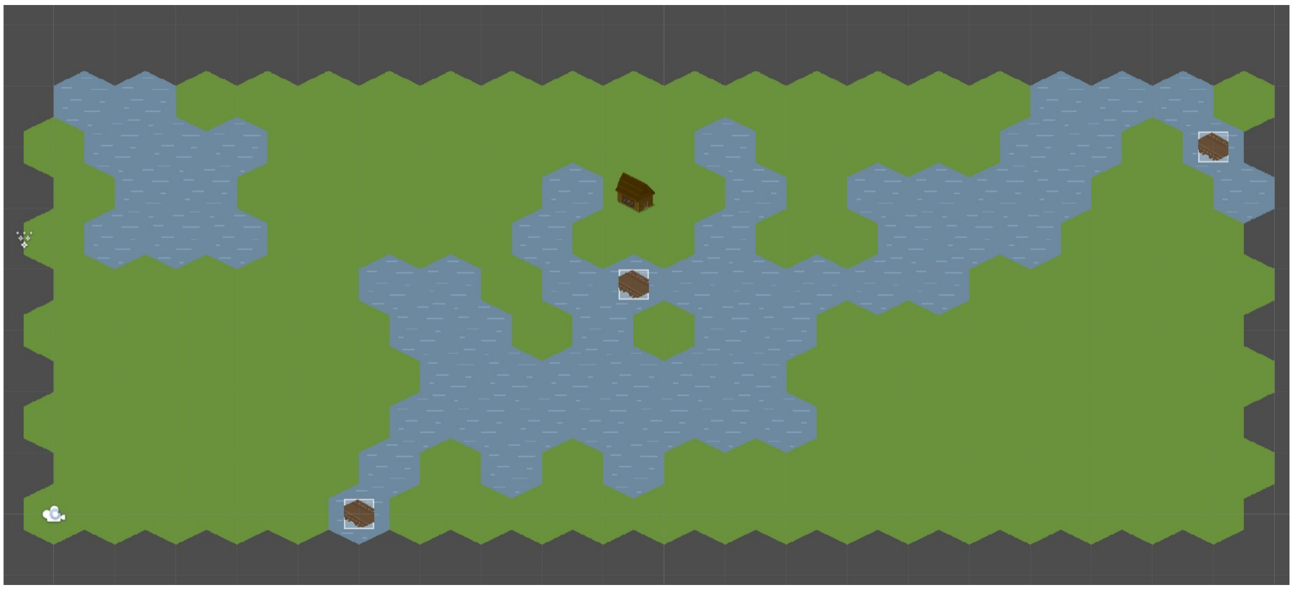
a) If no chain has been started, pick a random available grass tile (i.e. a random element of the availableGrassTile list) and place a Mountain/Water tile on it. If it blocks off a part of the level cancel it and start **a)** again, otherwise keep the tile and add its position to the openMoutainsInChain list.

b) Once a chain has been started (i.e. openMoutainsInChain has at least one element) we have a random chance to expand it, based on the CHAIN_CHANCE and CHANCE_PROPAGATION variables set in the Global Game Manager. CHAIN_CHANCE is the default chance to expand the chain and CHAIN_PROPAGATION is how slowly the chance decreases when expanding the chain. With CHANCE = 1 and PROPAGATION = 1 the Level Generator will attempt to spawn all mountains in a single chain. With CHANCE = 0.5 and PROPAGATION = 0.95 there will be a 50% chance to spawn a second tile in the chain, 45% for the third one, 40% for the fourth etc. With CHANCE = 0 we are essentially using Random Generation. If the test is not passed we break the chain and go back to **a)** to start a new one. If the test is passed we continue to **c)**

c) We pick an element in the openMoutainsInChain list as the new branching tile (see Shape of Chains for details on which element to pick). We are going to try to expand the chain from the branching tile. We look at its neighbours in a random order and try to spawn a new mountain on them. If no neighbour is valid we remove the branching tile from openMoutainsInChain and go back to **b)**. If we find a valid neighbour we spawn the new Mountain tile and add that neighbour to openMoutainsInChain. We continue until all tiles have been spawned or there is no available tile left in the level.

This method works similarly for Water tiles, except when it comes to the pathfinding. This particular point is fairly complex and is discussed in its own section, *The Water Problem*.


Natural Mountain Generation


Natural Water Generation

(In this second picture some water tiles are marked with a bridge icon. This is related to the specific pathfinding technique used for testing accessibility of water tiles. See *The Water Problem* for more details)

2.1.2.1. Natural Generation Optimization

2.1.2.1.1. List of Critical Tiles

When working on the Natural Generation method I realized that **some tiles were being repeatedly tested** even if they had been previously considered invalid. This was also the case for Random Generation but way less so. The reason for this difference is that randomly placed tiles have a lesser risk of blocking off parts of the level, since they tend to be scattered all over the level. When spawning chains of impassable tiles however, **it is much more probable to create a wall between two areas of the level**. Another reason is that when spawning multiple impassable tiles next to each other the ones in the middle risk being inaccessible.

To fix this problem I declared a criticalTiles list, initially empty. When trying to place a new tile, if it blocks off a part of the level, we cancel it AND add it to the criticalTiles list. And now, when trying to expand the chain from a branching Mountain/Water tile, **we skip any of its neighbours that have been marked as critical**.

Introducing this new list also allowed me to easily determine when there is no valid tile left on the level, so I can interrupt the Generation method. If *availableGrassTiles.Count – criticalTiles.Count == 0* then we know all remaining Grass tiles have already been tested and are not valid.

2.1.2.1.2. Shape of Chains

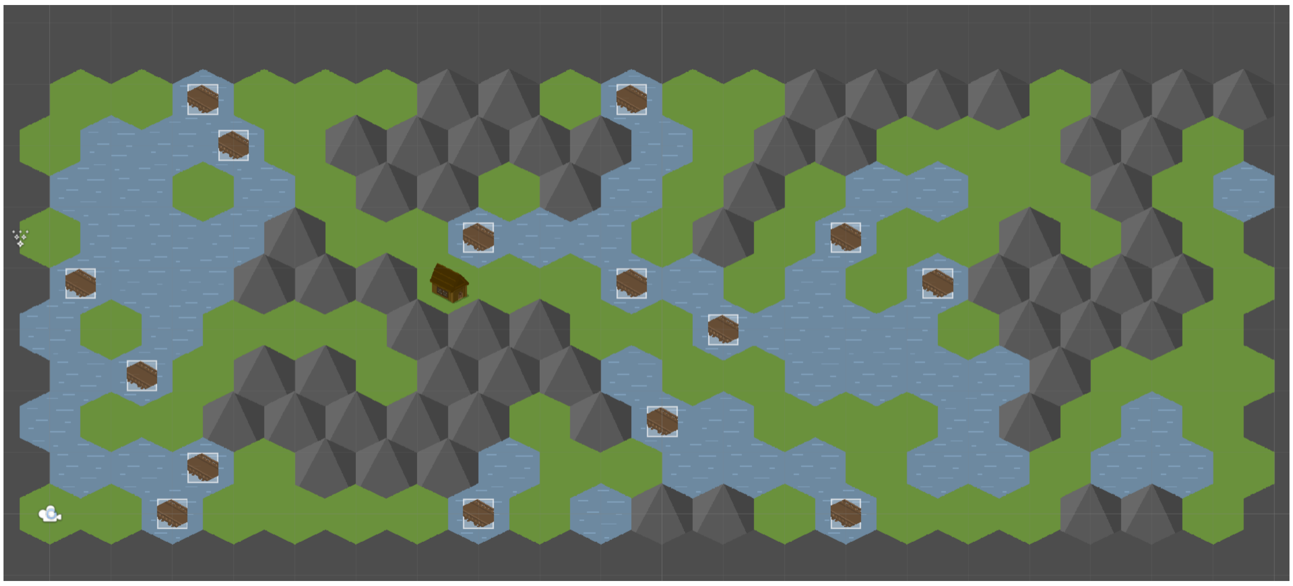
When expanding the chain, **the choice of the new branching tile is particularly relevant**. The branching tile is picked among the tiles that are already part of the chain (i.e. in the openMountainsInChain list), and the chain is expanded by spawning a new tile at one of the neighbouring positions of the branching tile. This new tile is then added to the openMountainsInChain.

This means that when building the chain the latest tile is always at the end of the list, and the first tile placed in the chain is at the first index. Say we always take the first tile in the list as the new branching tile. We are going to expand the chain concentrically around the original tile, and **end up with a lot of Mountain/Water tiles bunched up together**. Not only does this not make for very interesting formations – they are always going to be very similar – but the tiles in the middle of those formations are quickly going to become inaccessible, forcing the Level Generator to cancel some tiles, making the whole process very inefficient.

Now say we always take the latest tile added to the chain as our branching tile. This will cause the chain **to expand outwards, progressively getting away from the original tile**. In testing it didn’t take long to realize that this made for way more interesting - and varied – formations. For mountains it creates those realistic-looking ranges, and for Water it makes some fairly convincing river shapes. This is why, in my GenerateMountainsChain() method I always take the last element in the openMountainsInChain list as the new branching tile.

Finally, we could also take any random element for that list, and would end up with some middle ground between large bunched-up formations and long thin ones. This is actually what I ended up doing for the Water tiles. As mentioned in the last paragraph, picking the last element of the open list would create continuous river-like shapes, expanding in more or less one direction. I wanted the rivers to look more realistic, branching out into multiple sub-rivers on either sides. I also wanted the Level Generator to create some lakes, i.e place multiple Water tiles next to other.

Picking a random element in the open list to expand the chain from produces very satisfactory results. Rivers now expand in different directions and lakes are sometimes created along them.



Natural Mountain & Water Generation

2.2. Pathfinding Method

The one and only rule that the Level Generator has to abide by is **to generate levels where all tiles are accessible for the player**. There must be at least one path from the starting tile to any other tile. To check this accessibility I needed an efficient Pathfinding algorithm that would tell the Level Generator whether or not the tile it is trying to place is valid, in the fastest possible way.

There are two main ways to perform this check : **preemptively or correctively**.

2.2.1. Preemptive Check

By checking for accessibility preemptively, we make sure that **the Level Generator never spawns a tile that would make other tiles inaccessible**. This requires performing the Pathfinding check for every new tile during the Generation process. When the Level Generator places the last tile on the level we know that this level is valid and ready for the game to start.

Since this check will have to be performed a lot of times during the generation process (potentially once for each tile on the level) I needed a very fast pathfinding algorithm. And although I initially gravitated towards A* - by force of habit - I finally settled on a different algorithm.

2.2.1.1. Greedy Best First Search Over A*

A* has a lot of advantages. It always finds the best path between two points and is fast enough for most uses. However, after looking up several other pathfinding algorithms I decided **to use a Greedy Best First Search** instead, for the following reasons :

- A* spends time calculating and keeping track of the shortest path between the start and end points, so we can reconstruct it by backtracking. In our case **we don’t need to know what the shortest path** (or any path for that matter) is, we just want to know **if there IS at least one path** between our two tiles. A simple true of false answer is enough for us. The time spent by A* keeping track of the path would have been wasted.

- There are no movement cost between tiles, or rather this cost is always 1. As a result a straight path will always be the fastest way to travel between two points. GBFS works well in those situations since it tends to go in straight lines toward the goal by design. And in the case where an obstacle blocks the straight path, the pathfinding algorithm will always waste some time exploring tiles along that obstacle before it goes around it, whether we are using A* or GBFS. The heuristic being the distance between the current tile in the frontier and the goal, both algorithms will explore the same tiles before finding the path around the obstacle. In the worst scenario, where the only path goes around the entire level, GBFS will be exactly as efficient as A*, and faster in any other case since it performs less calculations.

2.2.1.1.1. GBFS Optimization

2.2.1.1.1.1. GBFS with Multiple Targets

What does it mean for a tile to block off a part of the level ? Simply put it means that **at least one tile is not accessible from the starting tile**. But that inaccessible tile can be anywhere in the level, and it would be inefficient to check every single tile until we find one that is not accessible.

But we don't need to, because our Preemptive Check tells us one thing for certain : at any step of the Generation process we know, by induction, that every tile is accessible. As a result **we only need to check accessibility for all neighbours of the new tile we are trying to place**. If at least one of them is inaccessible the new tile is invalid and must be cancelled.

To test if all neighbours are accessible I initially thought of calling GBFS on each of them, starting from one position (I picked top-right arbitrarily) and moving clockwise. But that is generally unecessary : **if the top-right neighbour is accessible then the right neighbour will also be, since we can reach it from the top-right neighbour**. We can apply the same logic to all other neighbours (as long as we keep testing the adjacent neighbour, meaning a clockwise or counter-clockwise order) as long as they are passable. If one neighbour is impassable (i.e. if it is a Mountain or Water tile) it will not provide accessibility to its next neighbour. So in that case we have to perform GBFS again for the next one, and keep going like that until all 6 neighbours have been proven to be accessible, either by GBFS or thanks to their direct neighbour being accessible and passable.

But this was still somewhat inefficient. It limited the number of GBFS calls significantly but we would still sometimes have to perform it multiple times when trying to place a new tile, even though all its neighbours are naturally fairly close together. That is why I decided to implement a modified version of the GBFS algorithm, that would look **for a list of targets instead of a single one**. That list would contain the neighbours of the new tile (except those outside the bounds of the level).

The heuristic doesn’t change : it is still the distance between the currently explored tile in the frontier and the new tile we are trying to place. As a result that distance is slightly incorrect (since our destinations are actually one tile away) but this doesn’t affect the search in any way. Once the GBFS has reached the new tile (*heuristic 0*) the new lowest priority in the frontier becomes 1, and the neighbours are explored immediately after.

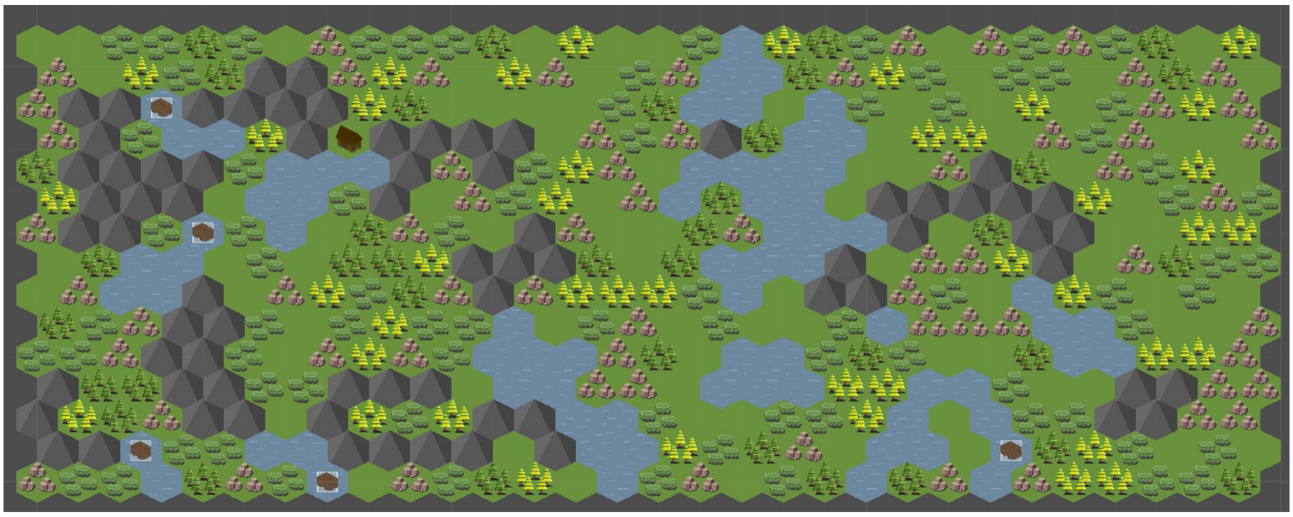
With this technique we only need to perform one search for all neighbours, reducing the computation time of the pathfinding method significantly.

2.2.1.1.1.2. Priority Queue

GBFS keeps track of visited nodes and of all the accessible neighbours of those visited nodes. This second set is called the **frontier**, and is generally implemented as a Priority Queue. Each node in it has a priority value, calculated by the Heuristic function, and **at each new step the algorithm takes the node with the best priority value** (either the lowest or the largest depending on the choice of heuristic) to explore. In this project the heuristic is the distance between each tile and the target tile, meaning that we have to take the element with the **lowest priority value** out of the frontier.

There is currently no official implementation of a Priority Queue in C#, but many programmers have published their own version of it online. For this project, since I very much see it as training project, I decided against using one of those available implementations and made my own. It is by no means optimized but does the job for this particular project. It uses a List of Tuple<T,int>, T being the Vector2Int position of each tile and int its priority. When dequeing we simply look through the list and return the element with the lowest priority.

No doubt various optimizations exist, and I will certainly look deeper into Priority Queues in the future, but there were a lot of other aspects of this project I wanted to focus on, and this implementation was fast enough for my needs.


30x15 Natural Mountains & Water With Resources

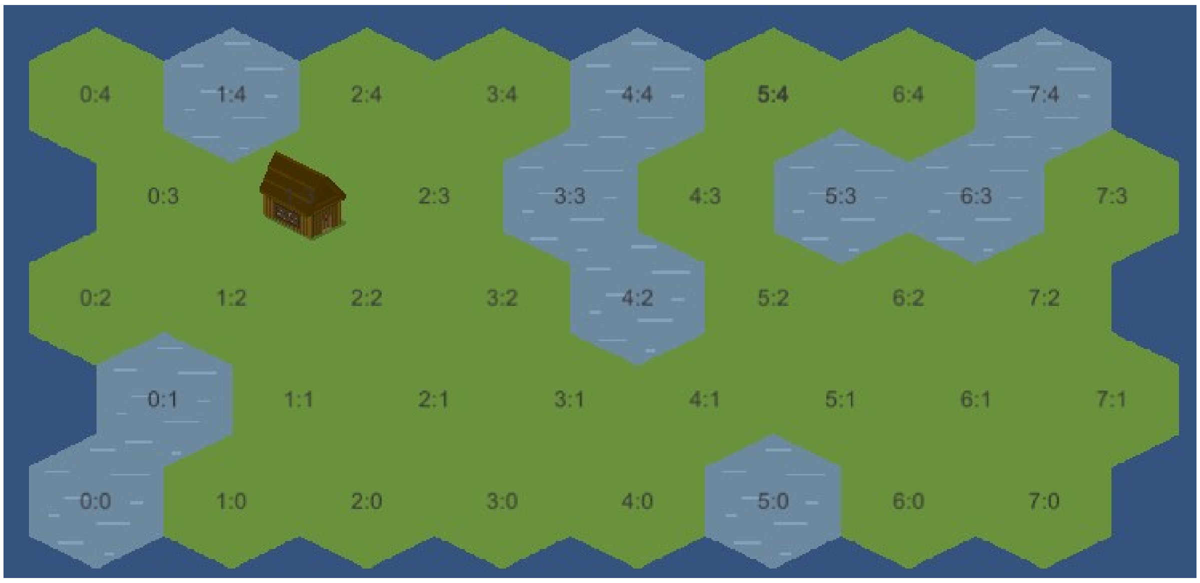
2.2.2. The Water Problem

The most complex part of designing this Preemptive algorithm was the generation of Water tiles. Contrary to Mountain tiles, which are completely impassable for the player, **Water tiles can be crossed by building a bridge on them**. Bridge have to be built in one specific direction (for hexagon tiles that means 6 possible directions), and then provide access to the tile in front of them.

For Mountain tiles the Level Generator simply had to test if the new placement would block access to any other tile, and cancel it in that case. For Water tiles however they could be placed even if they block path, **as long as the player can build a bridge on them to access the tiles inaccessible by land**. Additionaly we need to make sure that a large enough Grass area is left accessible around the starting house, to give villagers enough resources to build a bridge.

2.2.2.1. Bridge Over Crossable Water

Let’s take a look at this example screenshot, taken during the Water Generation process :



Crossable Water Tile Example

Let’s say that the Level Generator wants to place a Water tile at 4:1. Would that be valid ? In other words : if we place a Water tile at 4:1 are all 6 of its neighbours still accessible from the starting tile ?

- 3:1 : accessible on foot
- 4:2 : accessible from 3:2 or 3:1
- 5:1 : NOT ACCESSIBLE
- 4:0 : accessible on foot
- 5:0 : accessible from 4:0
- 5:2 :NOT ACCESSIBLE

(Note : as far as the Pathfinding method is concerned 4:1 is already a Water tile, which is why 5:1 and 5:2 are not accessible. We can actually observe the process using the DEBUG_MODE ingame : a Water tile is spawned at 4:1 before the Pathfinding is performed, and if the placement is found to be invalid the tile is turned back to Grass. This step is important, if we don’t put the Water tile at 4:1 before the Pathfinding call its result will be incorrect, since ***it would not count 4:1 as an obstacle****)*

If we were trying to spawn a Mountain tile, those two neighbours being inaccessible would be a deal-breaker : the tile would have to be cancelled to preserve accessibility to the right-hand side of the level. But a Water tile is different. The player could build a bridge on it. Is it the case for 4:1 ? Well, 5:2 can be accessed with a diagonal bridge from 4:0, and 5:1 with an horizontal bridge from 3:1. Spawning a Water tile at 4:1 is indeed possible, it is a **“crossable water tile”**.

One little thing to do : the tiles across 4:1 on both sides need to be marked as critical, so that we don’t attempt to place impassable tiles on them later on in the Generation Process. The access across 4:1 **needs to be preserved**.

For a counter-example let’s consider 4:3. All its neighbours can be accessed through land except for 5:4. So in order to confirm the placement of a Water tile on 4:3 we would need to find a valid access to 5:4. We look at the tile right across it, it’s 4:2, and unfortunately it’s not a passable tile. Placing a Water tile at 4:3 is not a valid option.

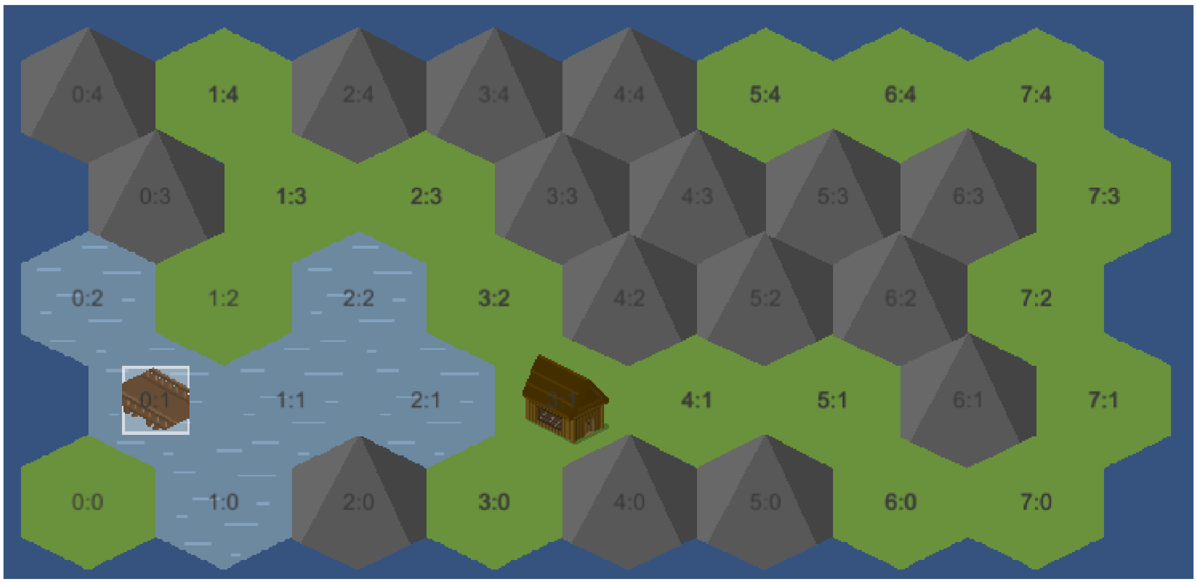
We could of course decide that such a move is valid since the player could build a bridge over 4:2 from 3:1, and then another one over 4:3. But I didn’t want the player to have to build several bridges in a row. I think that bridges are an interesting mechanic that add a level of depth and complexity to the exploration of the map, but risk becoming annoying if used too frequently.

Additionaly, if we allow multiple crossable water tiles in a row the player will not know on which tile to build a bridge and in what direction (we have to keep in mind that the level is initially fogged and that the player can only see one tile away from any accessible tile). That would require a big change in the exploration mechanic, unadvisable I think since that mechanic feels nice the way it’s currently implemented.

2.2.2.2. Resources for Bridge

Building a bridge isn’t free. It requires a certain amount of Wood and Stone. As a result we need to make sure **that the player can collect enough of those resources before they are blocked by surrounding Water tiles**. If we don’t check for that some levels will be strickly impossible to complete, the player being stuck on their starting island.

To achieve this we need to keep track of the number of grass tiles the Greedy Best First Search has explored before having nothing but impassable tiles in its frontier. If that number is less than a specified threshold we have to cancel the tile placement. For the following example we will consider that the threshold is 8. See the *‘Picking the Threshold’* section for more details on how we choose that value.



Garantying enough resources for bridge

Let’s look at the above example, focusing on two tiles : 0:1 and 4:1. In this picture the generation is over, all Water and Mountain tiles have been placed. The Generator tried to place a Water tile on 0:1 and on 4:1, respectively on the second-to-last and on the last step. Why was 0:1 accepted but 4:1 denied ? They can both be crossed to give access to the adjacent tile on the other side (respectively 0:0 and 4:1), can’t they ?

When testing 0:1 we counted the number of Grass tiles accessible from the starting tile by moving exclusively through land. There are 16 of them, which is more than our threshold of 8. In gameplay terms this means that the player will have access to 16 resources tiles to gather which is more than the minimum required (our threshold). They will be able to collect enough resources to build a bridge over 0:1 to reach 0:0. In that case 0:0 is not a particularly interesting tile to go to but it could have been the only access to a huge part of the level.

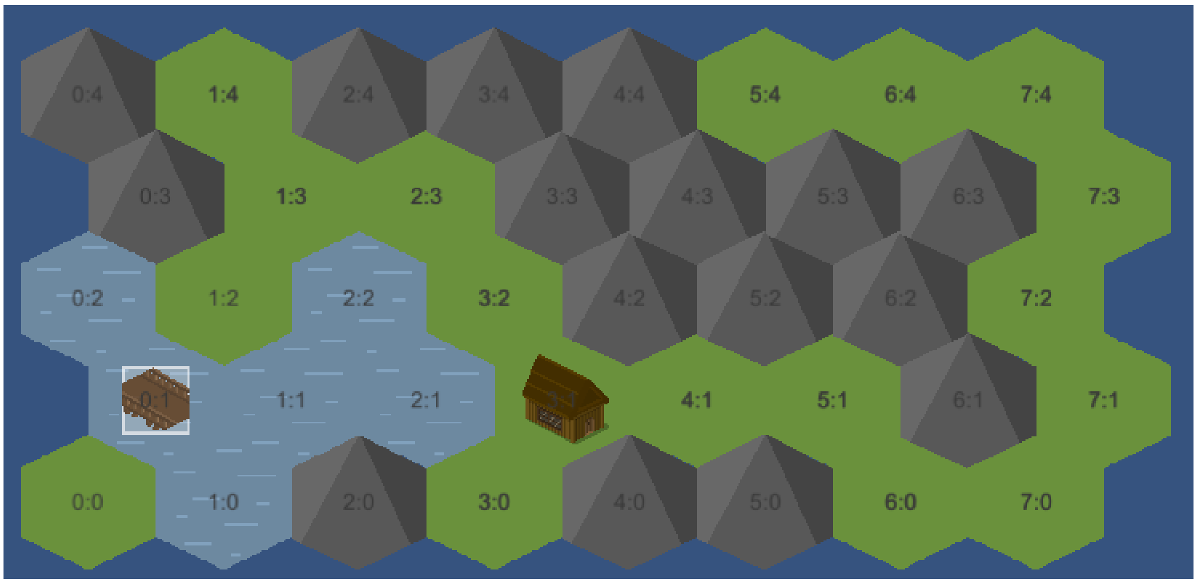
And indeed, let’s look at 4:1. That tile is the only access to the narrow valley on the right-hand side of the level. So if we place a Water tile on it, and even though it could be crossed with a bridge from 3:1 (the house) to 5:1, we only have 6 tiles accessible through land from the starting tile. Our minimum threshold is 8, so 6 is not enough. **We can’t guarantee the player will have access to enough resources to build a bridge over 4:1**. As a result we have to cancel the placement of the Water tile on it.

Finally, what would have happened if the Level Generator had tested those two tiles the other way around, 4:1 before 0:1 ? The Water tile at 4:1 would have been valid, since 8 tiles are accessible through land from the house (which is exactly our threshold value). But then 0:1 would have been cancelled, because only 6 tiles would have been left available.

2.2.2.2.1. An Interesting Programming Issue

All of this seems simple enough in theory but it provides quite an interesting problem in terms of programming. Over the Water generation process we allow some blocking Water tiles to be placed, based on the idea that a bridge can be built over them. This means that when exploring the level the **GBFS must consider those tiles as passable** when testing accessibiliy to tiles farther away in the level. Otherwise it would return that no path was found even if there is a crossable water tile in the way. If we say that those tiles are passable we have to be consistent.

The issue is that the GBFS is also in charge of counting the number of resource tiles accessible directly from the starting house (see previous section).



The Crossable Water tile Problem

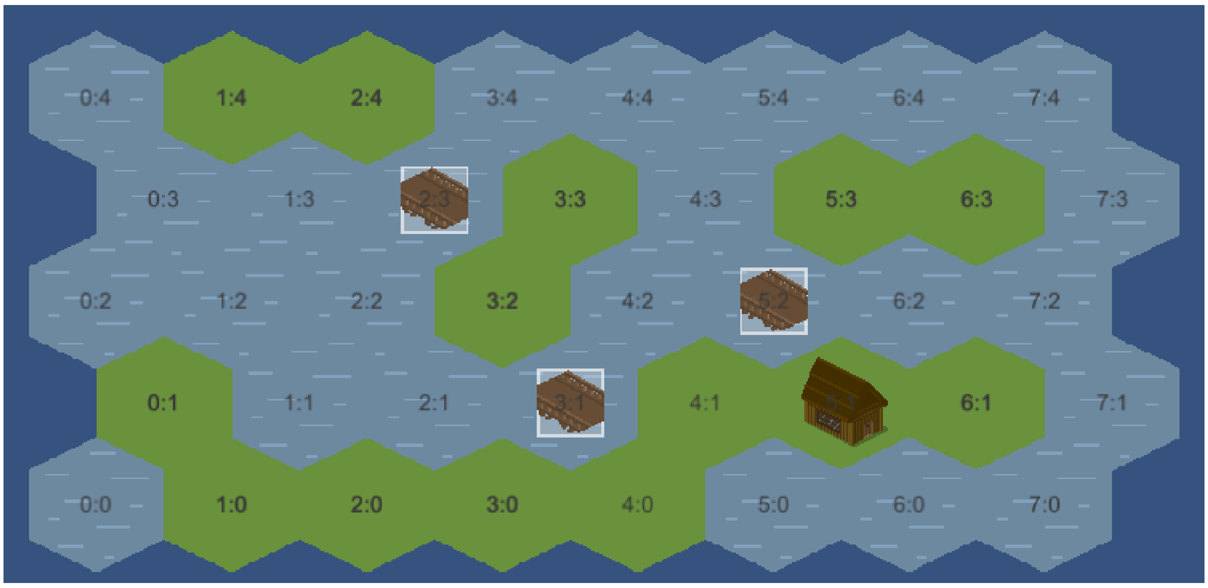
Let's go back to this example, but this time assuming our threshold is 7. We can only allow a blocking Water tile to be placed if at least 7 Grass tiles are accessible through land from the starting house. Let's try to place a Water tile on 4:1. That would leave the starting villagers with access to only 6 tiles. But the GBFS would have explored 8 tiles since it would have crossed the bridge on

0:1. It would return that there are enough resources available, which is incorrect.

Ok, so we can just stop incrementing the number of tiles explored when we cross the first bridge ? Unfortunately that would risk causing **the opposite problem**. Maybe the frontier crosses a bridge very early in the search and we end up with a lower number of accessible resources tiles than are actually present in the level.

To avoid this I modified the GBFS to give an arbitrarily high priority value (remember that for this heuristic low priority values are better) to crossable water tiles when adding them to the frontier, thus making sure that a Grass tile would always be picked from the priority queue before a Water tile. That way, when we do receive a Water tile from the frontier it means that it is all that's left in it, which in turn means we have explored the entirety of the first accessible land area. **We can stop incrementing our tilesExplored counter and it will be accurate**.

But unfortunately we are not quite out of the woods yet. If this method works fine when we are trying to place a tile at a position where it would block path, it can fail if the position is valid. Let's look at this example :



Problem with Threshold

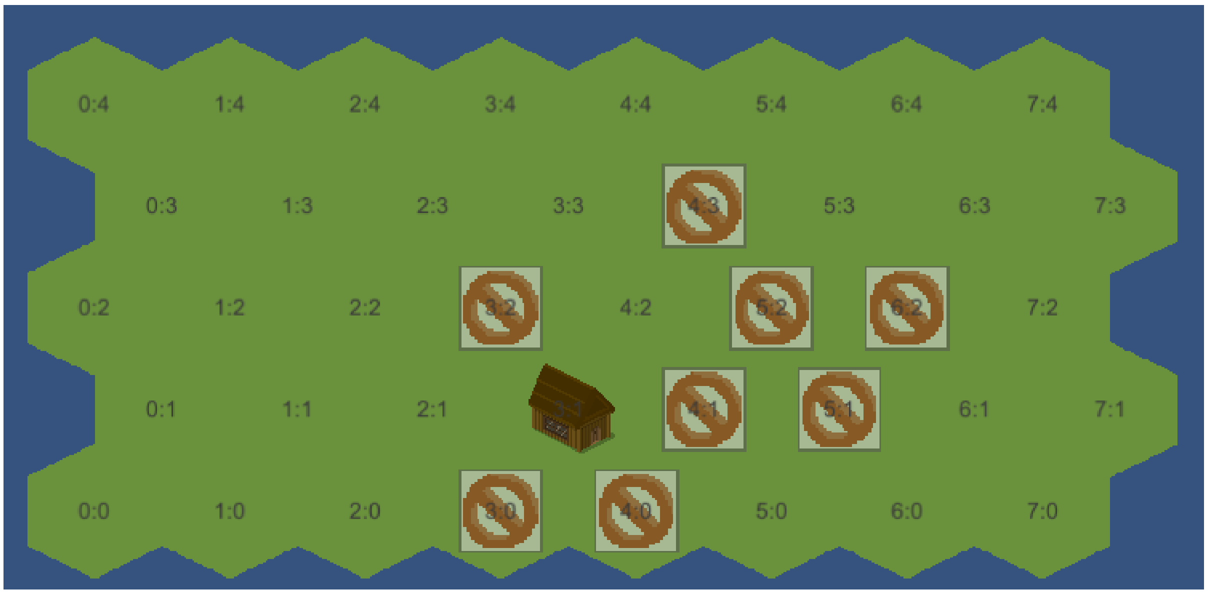
This level was generated with a threshold of 8 tiles, meaning **that 8 tiles should have been left accessible through land from the starting tile**. So how did we end up with only 7 ? The last step of the generation process was to try placing the water tile at 0:0. Considering that performing this placement takes the number of accessible tiles from 8 to 7, it should have been cancelled.

But we only check that number **if the new water tile blocks access to some part of the level**. And since 0:0 doesn't, the GBFS reaches all of its neighbours easily, and performs an early exit to return that the tile can be placed without risk. One solution would be to remove the early exit entirely, or only perform it after we have finished exploring the starting island. Not only would that be against the very concept of GBFS but it would also be extremely inefficient to explore a lot of additional tiles on every search. Imagine for example that the last Water tile placed by the algorithm was 7:1 instead of 0:0. GBFS would find an access to both its neighbours (7:2 and 7:0) in 2 steps but would then have to go the other way and explore the rest of the starting island to check if we still have access to 8 tiles.

2.2.2.2.1.1. A Simple Workaround

This is when I realized that, instead of going through all of this trouble just to end up with an inefficient algorithm, there was **a very simple solution** to this whole situation. At the start of the Generation process we just need to mark as critical a number of tiles around the starting house equal to the threshold. We define a forbidden area where no other impassable tile will be spawned, making sure the player will have access to enough resources to build their first bridge.

In the example below we can see that 8 tiles have been marked as critical at the start of the Generation Process.



Example of a Protected Starting Area

We can actually go a bit further and spawn a few forests, bushes and stone tiles in that area (critical tiles are only forbidden to impassable tiles, not resource tiles) right away. We can then start the Level Generation proper without any worry of creating an impossible level.

2.2.2.2.2. Picking the Threshold

How large does the starting island has to be to make sure the player has access to enough resources for a bridge ? Considering the wood/stone requirements for building it, 1-2 forests and 1-2 stone should suffice. Since we also want to spawn some Bushes tiles and a few unoccupied Grass tiles on the starting island, a size of 8 tiles seems appropriate. It is also worth noting that this number is the minimum, the Level Generator generally leaves a few more tiles accessible.

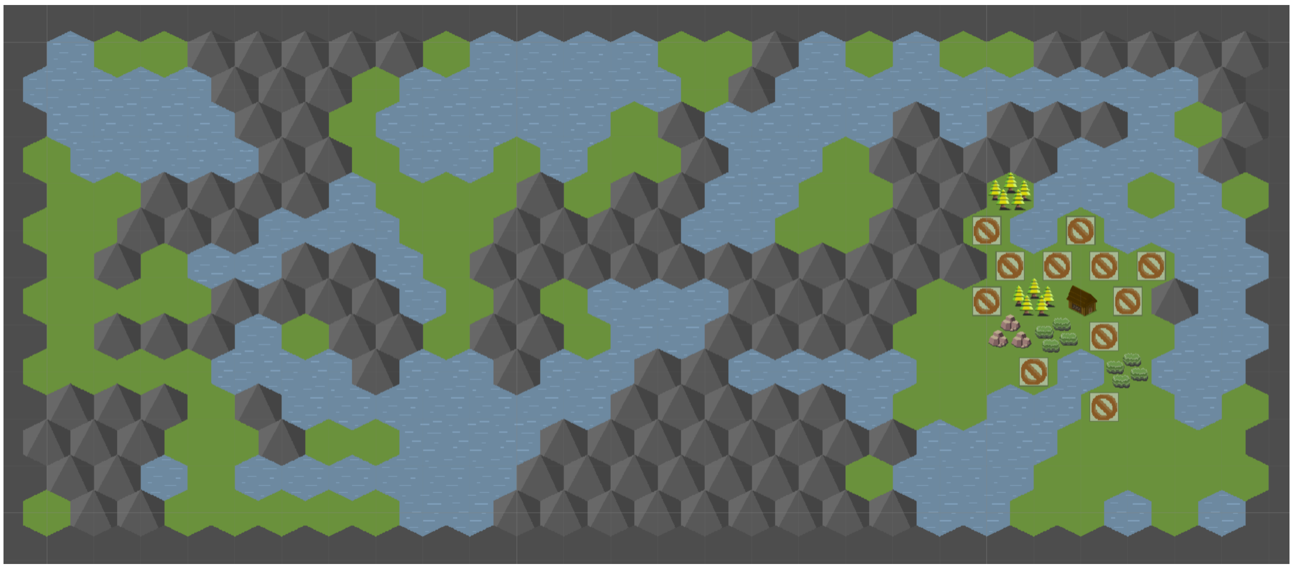
Do we need to increase this number for larger levels ? Not necessarily since we are only concerned about the size of the starting area; how big the rest of the level is is not really relevant. But it does still feel right to **introduce some proportionality** when picking the size of the starting island. On a larger level the player may need to build more bridges, as well as other types of buildings, so we are going to help them out by expanding the starting area a bit.

A linear progression would be too sharp. For the smaller level (8x5 tiles) we have decided on a starting area of 8 tiles, which is 20%. By applying the same ratio to a 100 tiles level we would get a 20 tiles island, and for the larger level available (50x25 i.e. 1250 tiles) the player would have 250 tiles to explore before having to build a bridge !

I finally settled on a **logarithmic progression**. The number of protected tiles is set to *2 * (int)log2 (xSize * ySize - 15).* This gives us the desired 8 tiles for a 40 tiles level, 10 tiles between 47 and 78, 12 between 79 and 142, up to 20 protected tiles for the largest possible level. Many other formulas would work fine, but in my opinion this one creates a nice increase of the starting island’s size **without making it too big for larger levels**.

2.3.2. Corrective Check

The Corrective method is a radically different approach to Level Generation. Instead of performing a Pathfinding check before spawning each impassable tile, we let the generation process run without performing any check. Which might result in something like this :



Unchecked Level Generation

(In this example the number of Mountain and Water tiles was artificially increased for demonstration purposes - a normal level would have way less – but this shows how a level can be generated with a lot of inaccessible tiles.)

(*Also, in all screenshots provided in this section, we can see that a few resources have been spawned aroung the starting house. This is due to the fact that after picking the spot for this starting tile we generate a few resources around it before we start the procedural generation proper. This is simply a way of making sure the player has access to enough starting resources; it is not specific to the Corrective generation method)*

The Corrective algorithm involves **destroying impassable tiles to open up a path from the starting house to all other tiles**. This works by performing a Breadth First Search from the starting house tile, until we have explored the entire level. When we get stuck – i.e when the frontier contains nothing but impassable tiles – we remove one tile from that frontier and resume the Breadth First Search from it. The main steps are as follow :

1) Pick the house as the current tile.
2) Start a Breadth First Search from the currentTile until we have nothing but impassable tiles in our frontier.
3) Pick a tile in that impassable frontier to remove, set this tile as the new currentTile and resume the BFS from **2)**.
4) Stop when the entire level has been explored.

When we go back to **2)** after picking a frontier tile in **3)**, the BFS resumes its execution where it left off. It doesn't reset its frontier nor the 'explored' boolean value of the level tiles. **All previously explored tiles remain explored** and the frontier is the same, minus the tile we just removed to be able to break the impassable frontier open. Only one Breadth First Search is performed during the Corrective Pathfinding process, it is simply interrupted when it runs out of Grass tiles to explore.

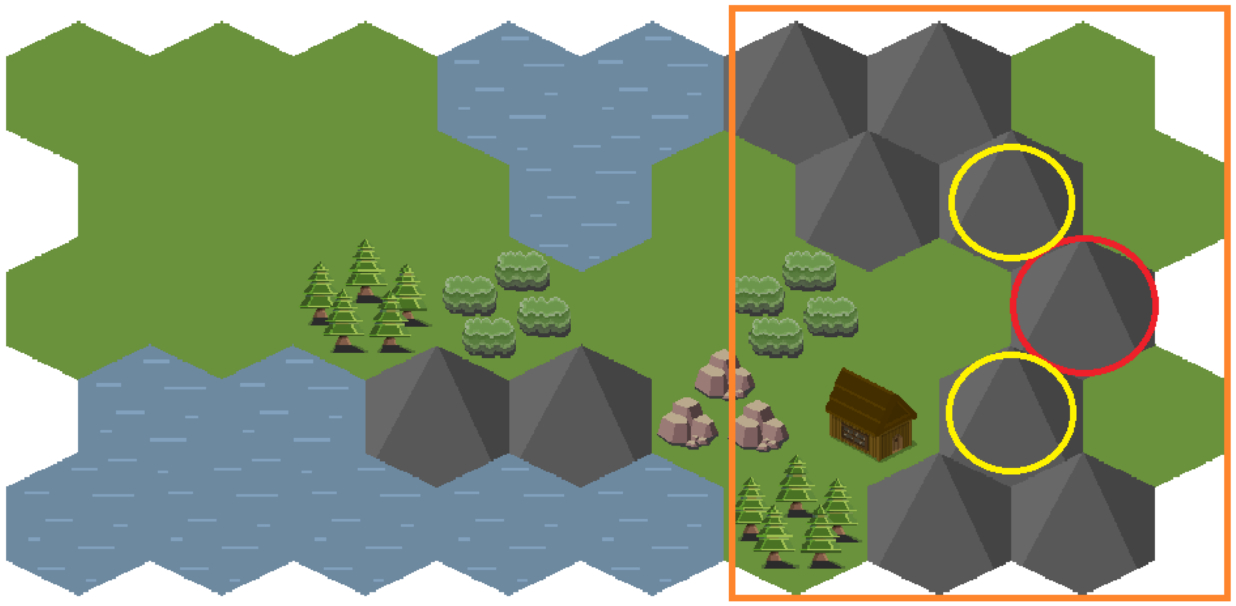
Step **3)** is where most of the optimization takes place. In the next sections I will go over the main techniques I used to implement this algorithm.

2.3.2.1. Breaking the Impassable Edge

When the Breadth First Search stops for lack of accessible Grass tiles to explore, its frontier is full of impassable tiles. In the rest of this document I will use the term *'edge'* to refer to this impassable frontier.

The main advantage of the Corrective method over the Preemptive one is its speed. It is in theory much faster to perform a single Pathfinding check after the Level Generation process than for every single tile in the level. The Corrective check, albeit much longer than each Preemptive check, will be faster than the sum of all Preemptive checks, especially on larger levels.

However this advantage comes with a additional difficulty. We need to make sure that **the Corrective algorithm does not destroy more tiles than strictly necessary**. Let's look at this example, focusing only on the area highlighted in orange :



Destroying the minimum number of tiles

How can we make those 3 Grass tiles on the right-hand side of the mountain range accessible from the starting house ? We could destroy one of the mountains circled in yellow, **but then we would also have to destroy the other one**. It would be better to destroy the one mountain circled in red, which would open up a path to all 3 Grass tiles at the same time.

Figuring out which tile to remove from the edge represents the main difficulty in implementing this algorithm. We need to do it fast while still making sure our choice is the best possible one.

2.3.2.1.1. Split Frontier

For this algorithm I implemented a custom class called Split Frontier. The Breadth First Search uses an instance of that class, instead of the usual queue, to performs its exploration. The SplitFrontier's public methods are :

- public void Add(Vector2Int newTile)
- public Vector2Int GetGrassTile()
- public Vector2Int BreakFrontier()
- public bool GrassTileLeft()

When the BFS explores a new tile it puts all neighbours of that tile in the frontier by calling splitFrontier.Add() with each of them, and then **checks if there is a GrassTileLeft() in the frontier**.

- If TRUE it sets the currentTile to splitFrontier.GetGrassTile() and continues the search.

-If FALSE it calls splitFrontier.BreakFrontier() to ask the frontier to remove a tile in the impassable edge. Afterwards there is a new Grass tile in the frontier and BFS can resume its exploration of the level.

As we can see SplitFrontier is used in a fairly similar way to a typical queue. It was very much intended that way, as a black box with a limited interface, but its inner workings are quite different from a queue.

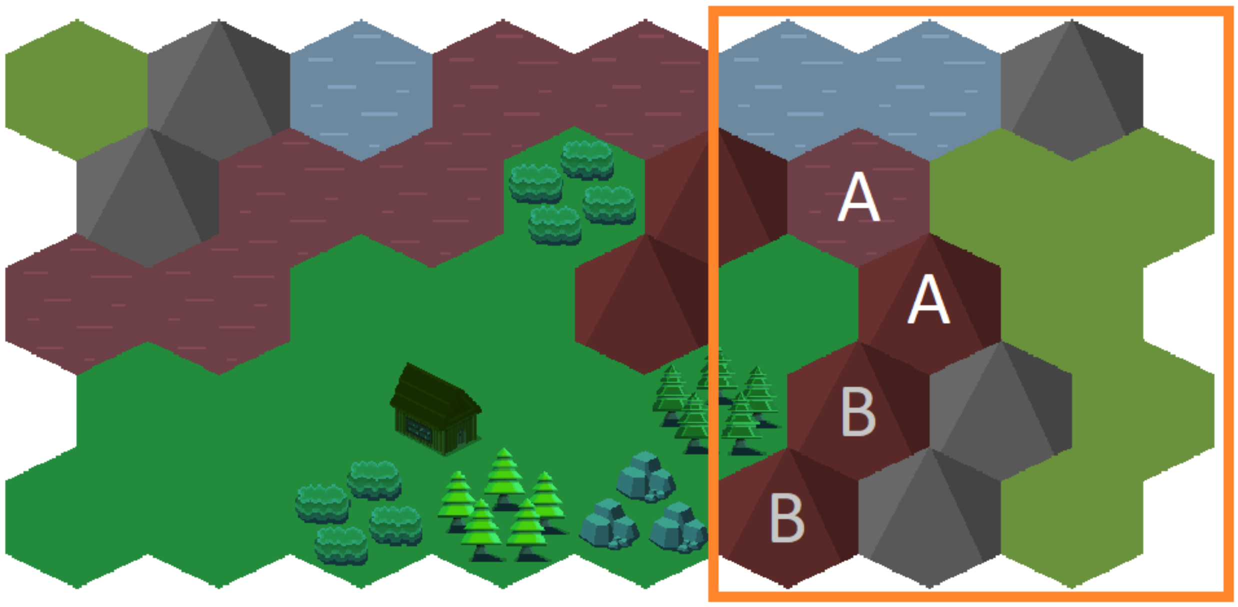
A Split Frontier keeps track of two different structures : **a queue of Grass tiles and a list of impassable tiles**. Whenever the Add method is called, the SF checks the newTile’s type and add it to either the grass queue or the impassable list. Whenever GetGrassTile() is called SR simply returns the first element of the grass queue. GrassTileLeft() is pretty obvious, it returns true if there is at least one available Grass tile in the frontier.

BreakFrontier() is in charge of **picking the best tile in the impassable edge** when the BFS gets stuck; it then moves that tile to the Grass queue so it’s available for the next step of the BFS. The *Vector2Int* return variable of BreakFrontier() is not necessarily the destroyed impassable tile, but more specifically the tile that needs to be destroyed by the LevelGenerator before BFS can resume. See *The Water Exception* for more details on that particular point.

2.3.2.1.2. Tile Grade : A and B

In order to determine which edge tile to break, I decided on a grade system for impassable tiles. An **A tile** is an edge tile that has at least 1 unexplored Grass neighbour. A **B tile** is an edge tile that has 1 or more unexplored neighbours, none of them being a Grass tile.

When picking an edge tile to destroy **we want to prioritize A-grade tiles**. The fact that they have an unexplored Grass tile among their neighbours means we can potentially open up an entire new area by breaking them. B-grade tiles on the other hand are surrounded by other impassable tiles and breaking them wouldn’t be very valuable. The following example illustrates that difference :



Grade of Edge Tiles

This screenshot was taken during the Corrective pathfinding operation, tiles are colored based on their state. The bright Green tiles around the starting house have already been explored, and the current impassable edge is made of all red tiles. Tiles that haven't been explored yet have their default color.

To understand the difference between A and B tiles **let's focus on the area highlighted in orange**. How can we gain access to the unexplored tiles on the right-hand side of the level ? By breaking one of the 4 edge tiles adjacent to them. Each of them has at least one unexplored neighbour (a Mountain tile for both tiles marked B and a Grass tile for both A-tiles). By breaking a B-tile we wouldn't make a lot of progress though. The BFS would resume for a single step before being stuck again on the adjacent mountains.

By breaking one of the A-tiles however **the BFS would gain access to the entire area on the right**, making a lot more progress. This illustrates why we always try to prioritize A-grade tiles over B-graded tiles. As a result, whenever it is called, the *splitFrontier.BreakFrontier()* method picks an A tile if possible, and if none exists settles for a B tile.

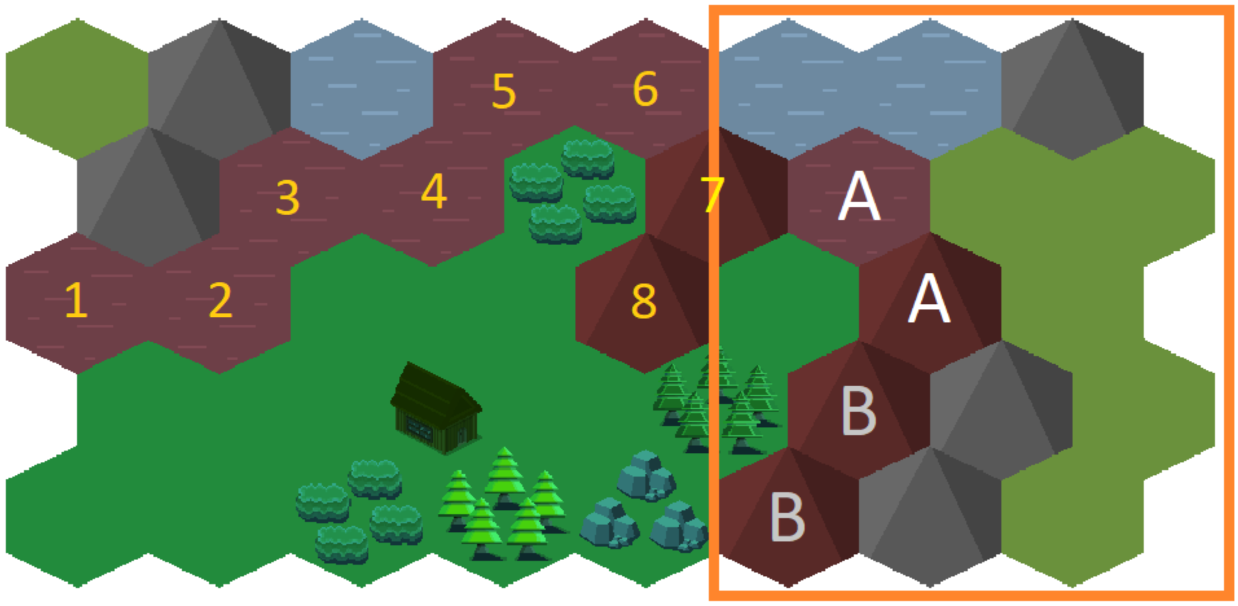
Even within each grade some tiles are better than others. SplitFrontier makes sure to pick the best tile out of the highest grade of tiles. In the previous example the upper A-tile is a better pick than the lower one since it would give access to the two Water tiles at the top in addition to opening up the Grass area to the right. But how can we identify the best tile in the quickest possible way ?

2.3.2.1.3. Picking the Best Edge Tile : Basic Method

The most simple way to identify the best edge tile is to **run through the entire edge every time BFS gets stuck**, keeping track of the best encountered tile along the way. Once the edge has been entirely explored we break the best tile and continue BFS.

If we want to speed this up a little bit we can introduce an Early Exit system, by stopping the search as soon as we find an A-tile. This will not guarantee we have found the best A-tile, but still gives satisfactory results while saving some time.

With or without this early exit however, this method is very inefficient. The split frontier will have to perform the same calculations many times over. If we go back to the previous example :



Grade of Edge Tiles – The Return

Let's say we look at all edge tiles every time BFS gets stuck. In this case let's assume we check edge tile from left to right, following the order indicated in the picture :

- Edge tile 1 : 1 unexplored neighbour, not Grass ==> GRADE B (1)
- Edge tile 2 : 1 unexplored neighbour, not Grass ==> GRADE B (1)
- Edge tile 3 : 3 unexplored neighbours, not Grass ==> GRADE B (3)
- ….
- Edge tile 8 : 0 unexplored neighbour ==> GRADE C (see next section)

Then we get to the first A tile. If we use Early Exit we stop the search right now, otherwise we keep going for the last 3 edge tiles, at which point we have our best tile (the top-most A-tile); we replace it by a Grass tile and resume the BFS.

But with this method, next time BFS gets stuck we will have to calculate the grade of all edge tiles **all over again**, even though it will not have changed for most of them since the last time. In our example, edge tiles from 1 to 5 will still have the exact same grade, making recalculating it totally pointless. It would be more efficient to save that information between each BreakFrontier() call. But let’s first look at the C-grade situation.

2.3.2.1.3.1. Trimming the Impassable Edge

Grade C doesn’t actually exist. It simply means that **an impassable tile is removed from the splitFrontier entirely**. A C-type tile has exactly zero unexplored neighbours, so why carry it around ? It will never be able to open up any new path, so we might as well discard it from the frontier.

This is actually one of the most significant optimizations in the Corrective algorithm. When we expand the impassable edge – by exploring new tiles with BFS – we simultaneously trim it by removing its unnecessary elements. This ensures that **the size of the impassable edge is always as small as possible** and allows the algorithm to remain efficient even in very large levels.

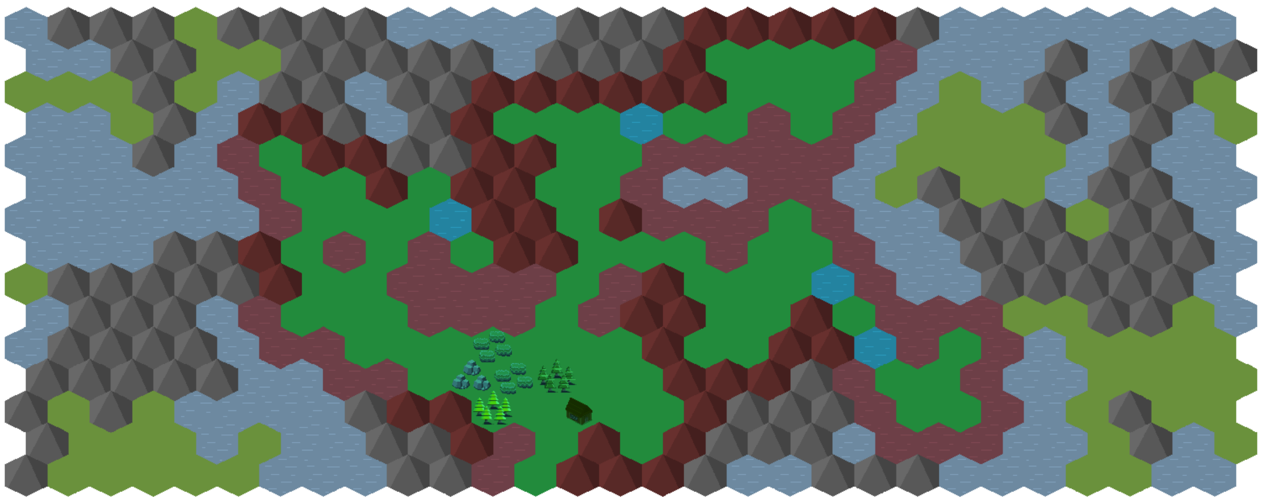
2.3.2.1.4. Picking the Best Edge Tile : Saving Grades

To save information relative to tile grades I implemented a new class called *ImpassableTile*, containing the position of the tile as well as its current number of unexplored neighbours, and unexplored Grass neighbours. When a new impassable tile is added to the frontier by the BFS, splitFrontier creates a new instance of *ImpassableTile* with that tile’s position, and initializes its unexploredNeighbours counts by looking at the state of its current neighbours. This ImpassableTile is then added to the impassableTiles list.

Now that we have initialized the grade values of the new impassable tile we need **to update them whenever a new tile is explored**, i.e. added to the frontier. So every time the Split Frontier’s Add() method is called, whether the new tile is Grass or Mountain/Water, we need to look at its neighbours. If any of them is currently in the impassable frontier list we have to decrement its count of unexplored neighbours (and unexplored Grass neighbours if the newly explored tile is Grass).

If an impassable tile sees its unexplored Grass neighbours count decreased to 0 it is demoted from A-grade to B-grade, or even to C-grade if that Grass tile was its only unexplored neighbour. Similarly an impassable B-grade tile can be demoted to C if it has no more unexplored neighbours.

By updating grades during BFS in this way, in BreakFrontier we simply have to compare the current edge tiles’ grades to pick the best one, instead of having to recalculate all of them. It is a definite improvement, but **we still have a single impassable tiles list containing both A and B-grade elements**, and looking through it – if just to compare grades – is still fairly inefficient, particularly since there are generally way more B-tiles than A-tiles. We would like to access the latter immediately without having to sim through the former.



Corrective Algorithm in Progress

2.3.2.1.5. Splitting the Split Frontier

For that reason I decided to **split the impassable tiles list in two** : an A-list and a B-list. When a Mountain/Water tile is explored and added to the split frontier we calculate its unexplored neighbours and unexplored Grass neighbours counts, and put it in either A-list or B-list. We also perform the update explained in the second paragraph of the previous section, except that we now transfer the demoted tile from A-list to B-list, and/or take it out of B-list to discard it from the frontier entirely if it has no more unexplored neighbours.

Now, when BreakFrontier() is called, we can check impassableAList.Count. If it is positive **we only have to look through A-list** and return its best element. If that Count is 0, we pick the best element in B-list. This method makes the algorithm more efficient.

At this point we are tempted to look for even more optimization, by figuring out a way to keep track of the current best A-tile or B-tile during BFS exploration, making BreakFrontier() even simpler as a result. It would simply have to look up the best tile and break it.

As nice as that sounds it is actually more complicated than it appears. If we only keep track of the single best tile, after breaking it we would need to pick the new best tile, which would involve looking through the entire A-list or B-list. As it turns out this is **exactly what we already do in the current version of BreakFrontier()**, so it would not provide any optimization.

And keeping track of multiple best tiles is not exactly ideal either. Two methods come to mind : Sorted Lists and Linked Lists. Turning the impassable tiles lists into one of those data structures would let us know the exact order of priority among impassable tiles at any given time. It would certainly make BreakFrontier() faster but would require a whole lot of updating during BFS exploration. Any new tile explored could change the grades of multiple impassable tiles, thus forcing us to reorganize the Sorted or Linked list.

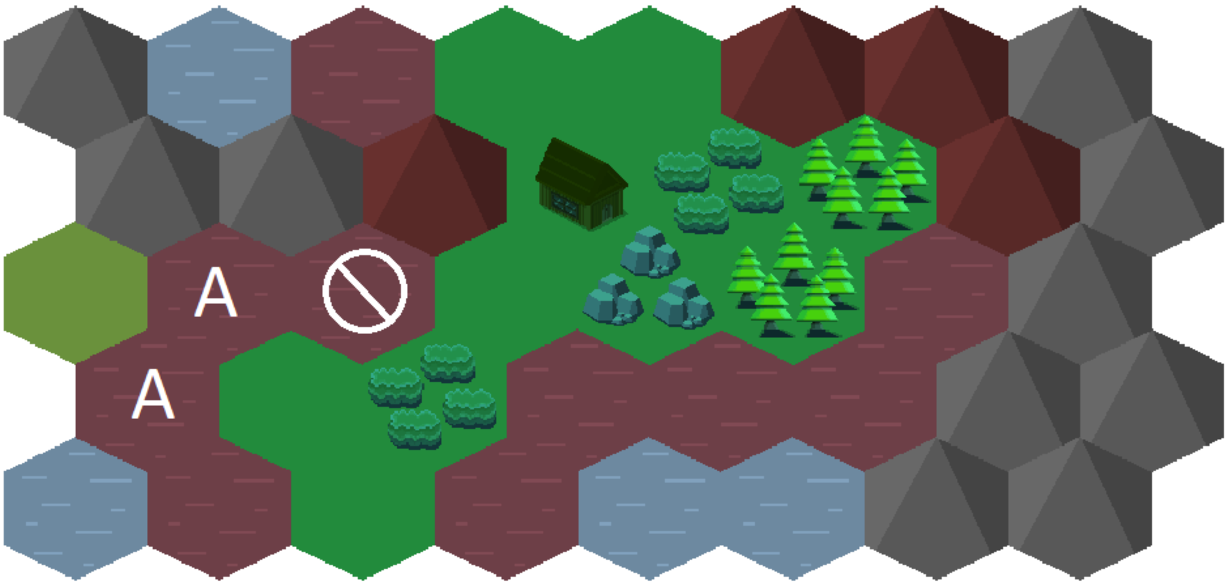
The latter would definitely be more efficient in this instance, and would be my choice if I decided to implement this method. But I spent a decent amount of time thinking of ways to implement it and none of them seemed worth implementing. The benefit they would provide for BreakFrontier() is actually fairly minimal (since exploring the impassable edge is already pretty efficient in our current method, in particular thanks to the trimming system explained previously in this document), and it would come at the cost of some additional testing and updating operations during BFS exploration.

2.3.2.2. The Water Exception

For the Preemptive generation method, we had to consider Water tiles slightly differently than Mountain tiles. A Water tile would be placed even if it blocked access to a part of the level, as long as a bridge could be built across it. We simply had to mark tiles on either side of the Water tile as critical.

In this Corrective method **the situation is reversed**. Instead of adding new tiles to the level we are removing them. As a result, no need to protect any critical tile : nothing is going to be spawned on them later down the line.

But we still have to perform an additional check in BreakFrontier(), when getting the best tile out of the A-list. If that tile is a Water tile we need to make sure a bridge can be built on it. The following example illustrates this situation :



A-grade Water Exception

The BFS is stuck and currently trying to find the best edge tile to break. We have two A-tiles, so we don't even have to look at all the B-tiles. Both of these A-tiles are Water, and the one at the top is slightly better than the other – it has 3 unexplored neighbours instead of just 2 for the other tile. So we pick the top-most water tile to break the frontier.

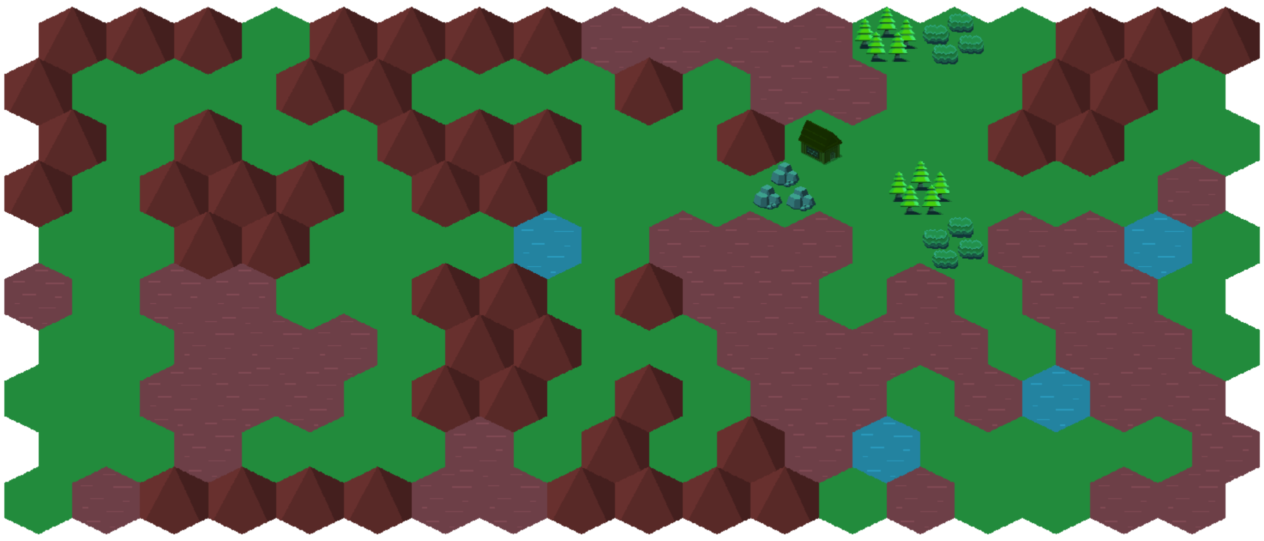
Or rather we would do that immediately if it was a Mountain tile. But since it's a Water type we have to perform an additional check : **is it possible to**

build a bridge over it in order to reach the unexplored Grass tile ? In this example it's not, the Water tile marked with the white sign blocks access.

We have two options : destroying the A-tile directly, or destroying the crossed-out neighbour. The Corrective algorithm would work either way- it would always produce a fully accessible level by removing the minimum number of tiles – but this is more of a game design decision. By removing the A-tile we remove the need for the player to build a bridge over it. By removing the crossed-out neighbour we create that need.

Because I think the Bridge system introduces an interesting level of depth to level exploration, and to the game in general as a result, I want to make sure that random levels have a few blocking Water tiles. Not too many – that would slow down the player too much and become annoying – but not too few either – to keep the Bridge mechanic relevant.

We actually have a third option : introducing some randomness. Say we have 50% chance to destroy the A-tile itself and 50% chance to destroy the impassable neighbour. But as it turns out, and after a decent amount of testing, it appeared that **always removing the crossed-out neighbour created just the right number of blocking Water tiles in the level**.



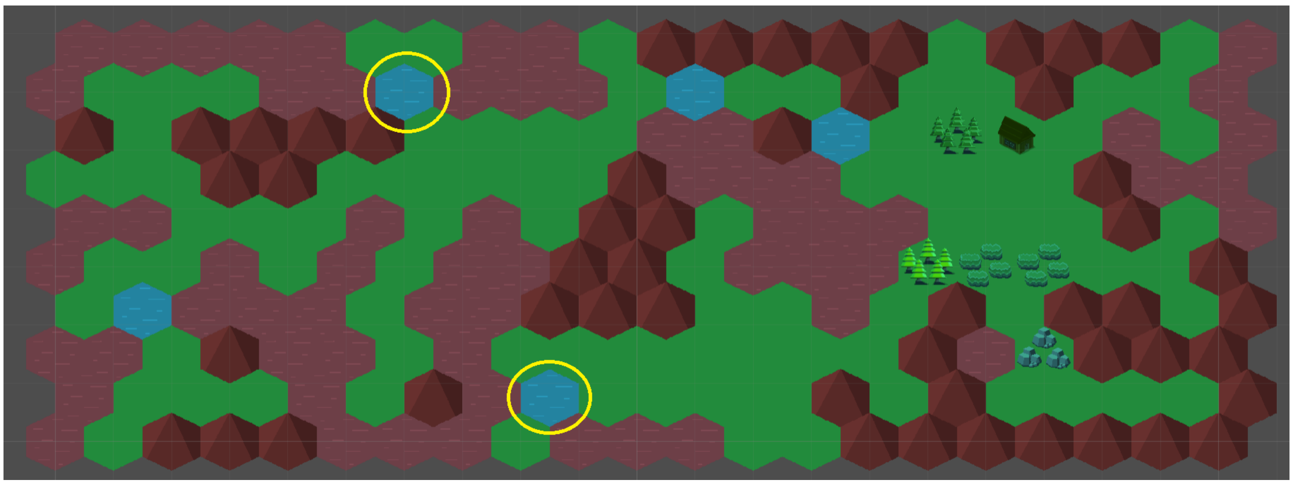
Final Level after Corrective Algorithm

In this screenshot taken in Debug mode we can see that the algorithm left 4 blocking Water tiles (displayed in blue in the picture). This feels like a decent ratio (considering the size of the level), that forces the player to build a few bridges to explore the map, without overdoing it.

As a side note, in this example, 4 is the number of ***obligatory*** *bridges, i.e. the minimum number that have to be built if we want to explore the entire level. The player can still build additional bridges to open up shorter paths to specific places. For example if they want to reach the tile at the bottom-left corner of the map, they could decide to build a horizontal bridge on the Water tile at the bottom of the large lake next to it. That would be faster than going all the way around it, through the mountain pass to the north. See Section 3.2. 'The Distance Mechanic' for more details on how distance impacts game mechanics.*

2.3.2.2.1. Pointless Bridges

At this point our Corrective algorithm guarantees that every single tile in the level is accessible from the starting house, either by land or by building bridges, while destroying as few tiles as possible. We could stop there, but there is still a little improvement we could perform. Let's look at this example :


Pointless Bridges

Blocking Water tiles – those that have to be crossed with a bridge – are indicated in blue. There are 5 of them in this level, after the Corrective algorithm was performed. Let's put ourselves in the role of the player. We start at the house, and progressively explore the level. Building a bridge over those blue Water tiles gives us access to a significant part of the level, with the exception of the two highlighted in yellow.

Those two Water tiles only give access to 1 or 2 Grass tiles, at the outer edge of the level. Nothing about this is invalid, technically speaking, but it still feels **a bit pointless**. The player will never want to spend resources to build a bridge over those tiles, it wouldn't be worth it.

As a result I decided to modify the Corrective algorithm so it would not keep blocking Water tiles next to the outer edge of a level. Whenever a Water tile is picked in the impassable A-list, if all its unexplored Grass neighbours are on the outer border of the level, the Water tile is destroyed. Running this modified version of the the Corrective algorithm on the same level would destroy the two Water tiles highlighted in yellow, only leaving blocking Water tiles that are worth building a bridge over.

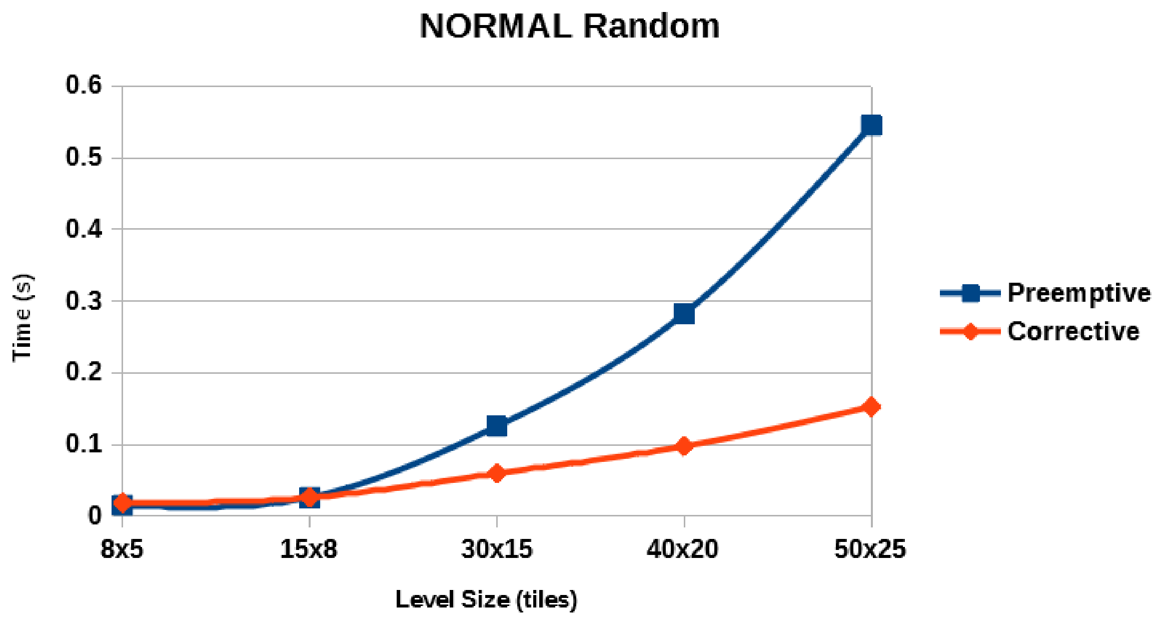
2.3.3. Conclusion on Preemptive vs Corrective

Once both methods were fully implemented I ran a number of tests, measuring execution time of the Level Generation process using *System.Diagnostics.Stopwatch*. The times reported in this section **do not take into account any other operation performed during the Scene change** by other scripts, or the initial rendering time of the newly created random level. The timer was started just before starting the level generation and stopped as soon as the entire level was instantiated.

The first series of test consisted in generating a level with an average number of impassable tiles – that being about 1/5 of the total tiles in the level – placed randomly. In this configuration blocking tiles are not very frequent and both algorithms are relatively fast as a result. The most expensive operations of both operations are performed when they encounter a blocking tile : the Preemptive algorithm will have to run its GBFS on the entire level, and the Corrective method will have to interrupt its BFS to break the impassable frontier. **The less frequent blocking tiles are, the faster both generation methods will be**.

Note: all reported Time values are an average of 3 results, to account for the randomness inherent to level generation; for example a moutain range being spawned at the start of the generation process might slow down subsequent pathfinding calls, whereas an open level where all impassable tiles are spawned at the outer edge will be generated a bit faster. In general time differences were still fairly minor though, especially for the Corrective method, whose execution time was very consistent (differences of 0.05s at most versus up to 0.5s for Preemptive).

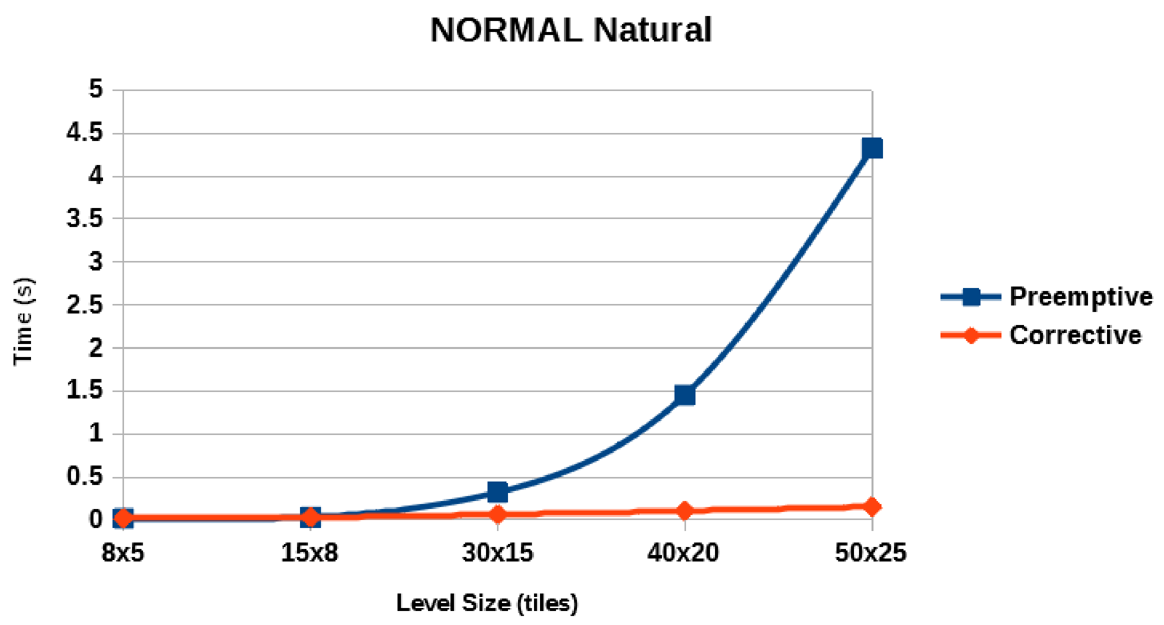
NORMAL Random	*8x5*	*15x8*	*30x15*	*40x20*	*50x25*
Preemptive	0.015275967	0.026265367	0.125403125	0.282532275	0.545264833
Corrective	0.018765867	0.0261984	0.0596467	0.0975553	0.152800233



For the smallest level, 8x5 tiles, both methods give similar results (Preemptive is actually ever so slightly faster), but as the level size increases we can clearly see that the Corrective is significantly more efficient. Its execution time goes from ~0.018s for a 40-tiles level to ~0.15s for a 1250-tiles level. In other words when the size is multiplied by ~30 the Corrective generation is only ~8 times slower. For the Preemptive method execution time is multiplied by ~36.

And differences between the two algorithms only get more obvious from now on. The second series of tests still involved an average number of impassable tiles, but this time spawned in a chain-like way.

NORMAL Natural	*8x5*	*15x8*	*30x15*	*40x20*	*50x25*
Preemptive	0.016570733	0.030765467	0.318089	1.4508602	4.323856933
Corrective	0.0198568	0.0274948	0.062459667	0.103129733	0.1531939

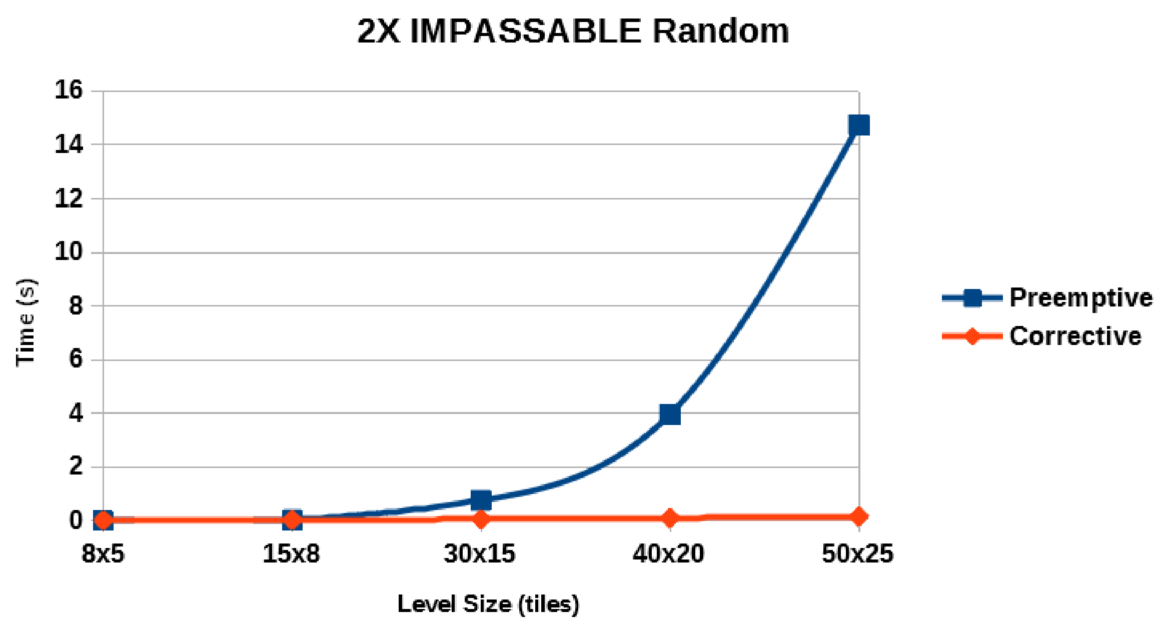


Generating Mountain and Water tiles in chains rather than in random patterns naturally tends to create more obstacles, which slows down the Preemptive method significantly. For the largest level it takes more than 4 seconds to generate the level. The Corrective algorithm, on the other hand, doesn’t really care about chains and reports execution times only slightly slower than the first series of tests.

To understand this difference we have to think of **what each algorithm has to do when they encounter a blocking tile**. The Preemptive method has to run a Greedy Best First Search over the entire level to confirm that the tile does in fact block access to one or more of its neighbours (making sure another path doesn’t exist somewhere else). The Corrective algorithm, since it keeps updating the grade of its impassable frontier tiles, only has to look up the best A-tile or B-tile in its frontier to keep progressing. Additionally, by continuously removing unnecessary tiles from its impassable frontier, it keeps its size in check even in very large levels.

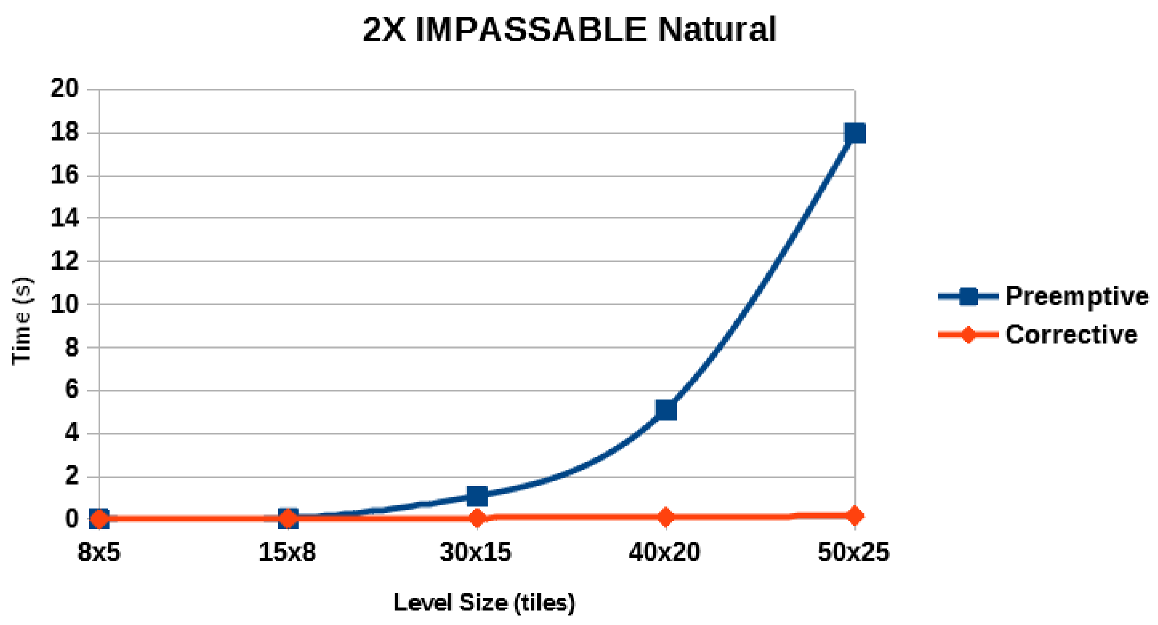
For the last two sets of tests I generated levels with twice the normal number of impassable tiles. This created a lot of blocking tiles and we can see differences in efficiency between the two methods get even larger. With randomly placed tiles first :

2X IMPASSABLE Random	*8x5*	*15x8*	*30x15*	*40x20*	*50x25*
Preemptive	0.01772375	0.046948933	0.762760025	3.9621459	14.74511733
Corrective	0.019787	0.028536233	0.062512533	0.103219833	0.160353767



The Preemptive methods takes more than 14 seconds to generate a 1250-tiles level; it only takes the Corrective algorithm 0.16 second. And finally with twice the normal number of impassable tiles AND a chain-like generation :

2X IMPASSABLE Natural	*8x5*	*15x8*	*30x15*	*40x20*	*50x25*
Preemptive	0.018962033	0.049751767	1.082858733	5.10032015	17.98546
Corrective	0.019313167	0.028311733	0.064007467	0.1046955	0.1619292



18s for Preemptive versus 0.16s for Corrective, not a particularly close competition. **For a 1250-tiles level the Corrective method is about 110 times faster**, and differences would only get bigger from there. The Preemptive algorithm shows an exponential increase of its execution time, whereas the Corrective method remains very efficient on large levels.

From the start of the project, before I even began implementing any of those two methods, it was fairly obvious to me that Corrective would be more efficient – even though I wasn’t honestly expecting such a huge difference in execution time – but I was curious to see the kind of levels each of them would generate. **The Preemptive method might be slower but could it generate more interesting levels ?**

After a lot of testing it appeared that both methods generated very similar levels. They essentially work in opposite ways (Preemptive starts with an empty level and adds tiles progressively, Corrective digs its way through an already generated level, removing tiles along its path) but **their end results are pretty much indistinguishable from one another**. As a result, because of its significant advantage in execution time I decided to stick with the Corrective algorithm, although I kept the Preemptive option in the game settings for reference and comparison.

This concludes the section on Hello World’s Level Generation. It was easily the most interesting part of this project, involving a lot of testing, research and optimization. I got to work with various algorithms, understanding their strengths and weaknesses, and I am fairly satisfied with the final result. Random levels are varied and generally feel like they were handmade, which was very much the goal. Procedural level generation has always fascinated me as a player, and I am glad to see it’s about as much fun to work on it as a developper.

3. Game Mechanics

3.1. Assigning Villagers to Tasks

In the earlier versions of Hello World, **villagers were not really living in any specific house**. Each house simply added to the ‘villagersCurrent’ variable in the Tilemap Manager based on its capacity (2 for Small House and 4 for Large House).

When the player performed any action (chopping down a tree, gathering berries, mining metal etc) the ‘villagersAvailable’ value was decremented; if that value was 0 the action requested by the player was not performed. Every day ‘villagersAvailable’ would be reset to ‘villagersCurrent’. Villagers were essentially **Action Points to be used every day**. Distance between houses and resource tiles was not considered.

After finishing the Level Generator I decided to implement a distance element to the game : the amount of resources collected during an action was reduced by the distance between the house and the resource tile, to represent the **villager having to spend some amount of time travelling between the two tiles**, and thus having less time to gather, mine, cut down trees etc. Firstly this implied modifying the Villagers system so that they would actually live in a house instead of simply being part of a global pool.

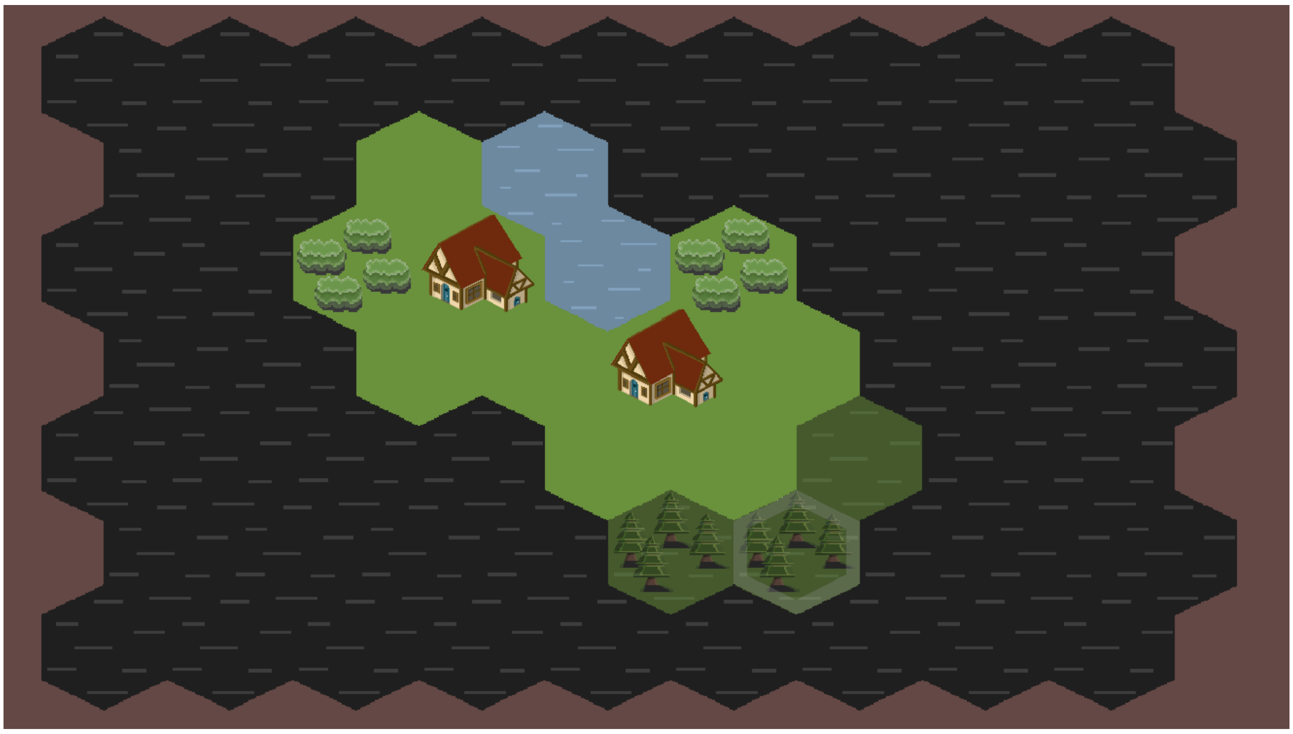
3.1.1. Picking the Best House for the Job

Then came the question of deciding which villager to assign to a task. We could let the player manually assign them : they would click on a house with at least one available villager, and then click on the resource tile to send them there. This would make my job easier as the developer, but I decided against it. I want Hello World to be **a very simple game to play**, requiring as few clicks from the player as possible. Everything that can be calculated by the game should be.

And in that case it seems fairly straighforward : we want the game to pick the closest house with an available villager. This would maximize the task’s return, since the villager would have the shortest distance to travel, giving them more time to work on the tile.

3.1.1.1. The Exploration Exception

One fairly important exception to this rule is for exploration. Efficiency is not relevant for this action : the tile is always entirely revealed after the exploration. Let’s look at this example :


Assigning villagers to exploration

Let’s assume there is one available villager in each of these two houses. The player wants to explore the highlighted Forest tile and then chop down trees in it (the tile is currently still shrouded in fog, **it needs to be explored before it can be exploited**). It would be better to use the villager in the top-left house to explore the Forest tile first and then send the villager living in the other, closer house to cut down the trees.

This is a general rule for exploration in Hello World. Exploring a new tile might reveal a new resource, that the player is going to want to gather right away. If the closest villager is always used to explore a tile the player will be left with villagers **that are farther away from the newly discovered tile**, and thus a less efficient action.

So which house to pick then ? The farthest one ? Not necessarily. That house may also be close to a lot of resources that the player might want to collect later during the turn. The best pick to perform exploration is generally going to be a house that’s far away from any interesting tile, in the middle of the village for example. This is very much going to **depend on the situation and on the level layout.**

As a result, and even though the game is still going to recommend a choice for the house to use, the player will be able to scroll through the other houses and pick the one they want. This system is also going to be available for other actions : the recommended house will be the closest one to the tile but the player will still be able to make a different choice. This will ensure that the game is still simple to play (we can still perform an action in one click if we go with the recommended house), while still giving control to the player.

3.1.1.2. Picking the Closest House

How do we determine which house is the closest to the selected tile, in a quick and efficient way ? Naturally we need **to know the distance from every house to the target tile**, i.e. the number of moves that need to be performed to reach it. We then pick the house with the shortest distance.

The Tilemap Manager holds a List of all the House tiles on the level. We only need to calculate the path for each of those houses and pick the one closest to the target tile. One little optimization however : we can quickly determine what the minimum possible distance (i.e. number of moves) is between two tiles (see formula in *‘Distance Between Two Hexagons’*). So when looking through the list of houses we can start with the one with the shortest minimum distance, calculate its actual distance from the selected tile (which is going to be different from the minimum if there are obstacles in the way) and if *actual* and *minimum* distances are equal, **there’s no need to test the other houses**. We know that we have found the closest one.

Additionaly there is no need to worry about tiles not being accessible, they always are (the Level Generation makes sure of it when creating a random level, and I did when designing the Campaign levels). However **they might not be accessible from all houses**. Imagine a level with two independent areas, separated with a Mountain range or a river, with one house in each area. Tiles from the left area will not be accessible from the house on the right-hand side of the level, and vice-versa. We pick the closest house among the houses that can reach the target tile. Let’s look at the two methods I implemented for calculating those paths.

3.1.1.2.1. First Method : Running A* Whenever The Player Selects a Tile

The simplest method is to run an A* algorithm between each house and the selected tile when the player’s mouse cursor enters the tile. That way the game can display a preview of the path from the closest house, as well as the number of moves required for the villager, and maybe an estimation of the amount of resources gained if that villager is used. The player can then scroll through the other houses if they want to make a different choice.

3.1.1.2.1.1. Calculating Path Backwards

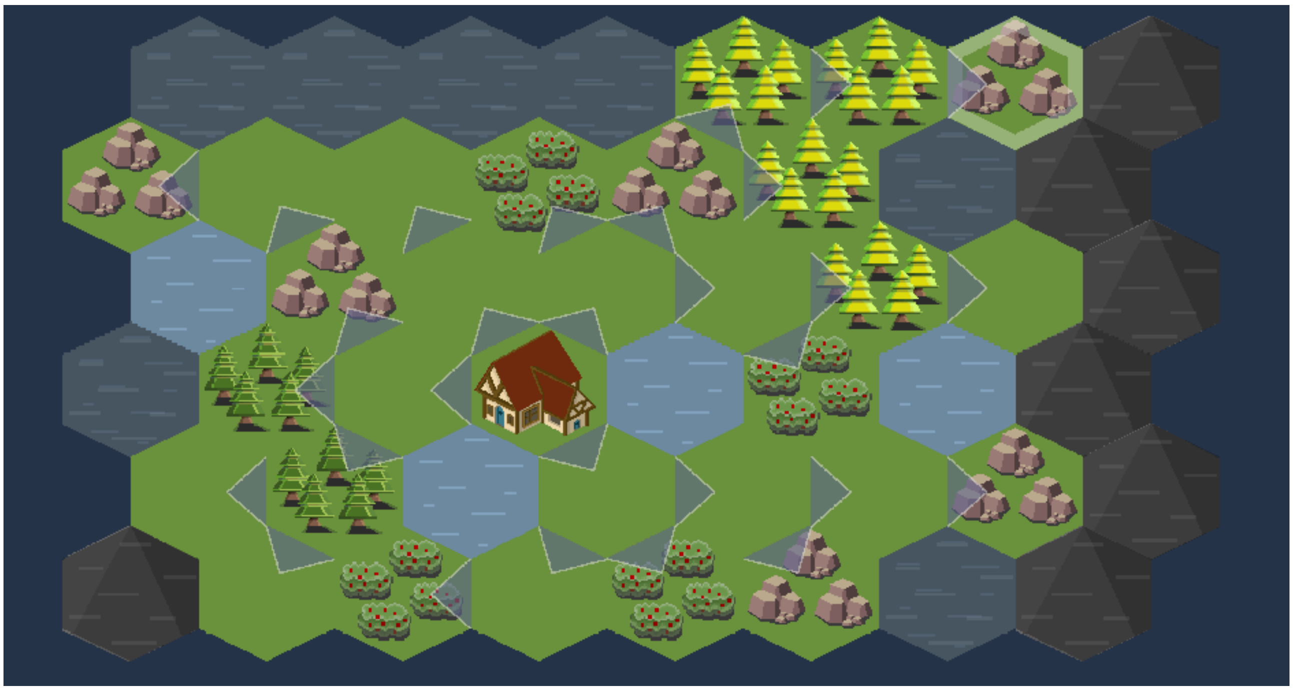
It is generally equivalent to find the path from A to B or from B to A. In this project however it is a little easier to calculate the path **starting <u>from</u> the selected tile and going <u>towards</u> the house**. This is due to some tiles being impassable, specifically Mountains in our case. When the A* algorithm explores the level it only adds neighbouring tiles to the frontier if they are passable. We can’t travel through Mountains and Water tiles (unless they have a Bridge built on them, but the Bridge tile is a different type altogether).

But what if our goal is a Mountain or Water tile ? The A* algorithm checks if it has reached its goal after getting a new tile from the frontier. And since Mountain and Water tiles are never added to the frontier they will never be reached ! A workaround would be to perform the goal check for each neighbour before adding them to the frontier, but that’s fairly inefficient.

Instead we can simply consider the selected tile as the start and the house as the goal. No additional test to perform, and the path returned by A* will be exactly the same.

3.1.1.2.2. Second Method : Preemptive Breadth First Search

Instead of calculating the path every time the player asks for it, **we could calculate all paths preemptively**, keep them saved in memory, and simply look up the specific path we want when the player selects a tile, by backtracking from that tile. In the example below BFS has been performed on all explored tiles. To find the shortest path from the house to a specific tile we simply need to backtrack from that tile, following the arrows in reverse until we get to the house.


Breadth First Search Paths

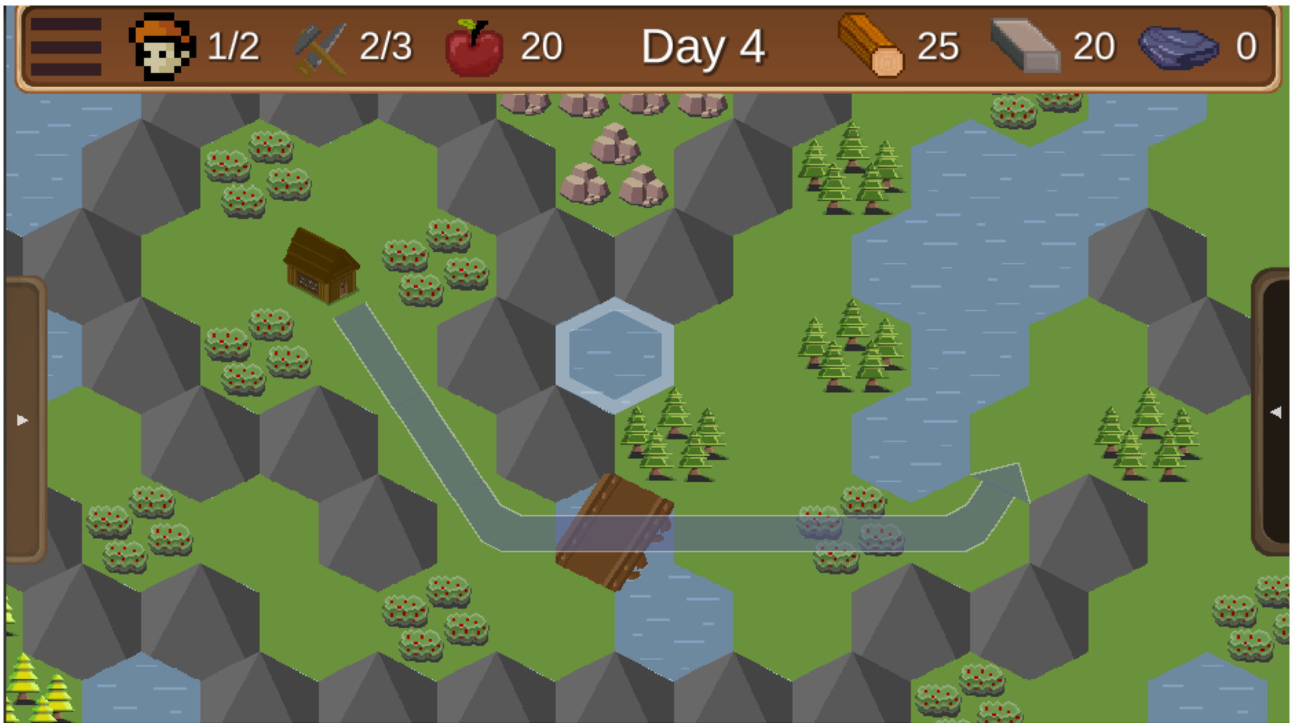
We need to make sure to update our paths :

- At the start of the level
- When a new tile is explored (to calculate the shortest path to it and since it might open up shorter paths to its neighbours)
- When a bridge is built on a water tile (it might have opened up access to an entire new area that no path was available to before)

That path update has to be performed **for each house in the level**. Shortest paths are calculated from a starting tile and are thus going to be different from every house.

Now that we have the direction to each tile from its predecessor in the shortest path, when the player selects any explored tile **we follow the arrows backwards until we reach the house**, and that is our shortest path. This is guaranteed by the fact that we run a Breadth First Search algorithm on a unweighted graph with no loop. Each tile can only be added to the frontier once and since that frontier is a Queue, when we get a new tile from it **we know it could not have been explored sooner**, meaning that we have the shortest path to it.

3.1.1.2.3. The Bridge Situation



Hmmm…

That is certainly a creative way of crossing a bridge. Once a bridge is completed it becomes a passable tile. But it is the only type of tile in the game **that is only passable from certains directions**. In this example the bridge should only be crossable from its Bottom Right or Top Right neighbours.

As a result, when performing our Breadth First Search to calculate the shortest path to all explored tiles we have to perform two additional tests :

- If a neighbour of the currentTile is a Bridge :
    - If we are not in front of it we skip it. We don’t even add an entry for it in the cameFrom list, and certainly don’t add it to the frontier.
    - If we are in front of it we set its cameFrom value to the currentTile but we only add it to the frontier if it’s completed.

- If the current tile is a completed bridge we should only add to the frontier **the neighbours that are aligned with it**. More precisely we only add the neighbour ahead of us, not the one we came from (that one has already been explored). So in this example, when we are on the bridge we should only add the adjacent Forest tile to the frontier. We add neither the Grass tile on the right – because it is not aligned with the bridge –, nor the one at the bottom left – because it has already been explored.

3.1.1.2.4. Conclusion on A* vs Breadth First Search

Those two methods are fundamentally different and each have their own advantages and disadvantages.

- On-demand A* :
  - ✔ Only calculates path when it is actually needed
  - ✔ Calculates path with early exit instead of exploring every visible tile
  - ✗ Might perform the same search multiple times

- Preemptive Breadth First Search :
  - ✔ Each path is only calculated once
  - ✔ Looking up a path is much faster than having to recalculate it
  - ✗ Wastes some time calculating path to tiles that are not going to be used by the player

In general the A* method is going to be more efficient if the player requests only a small number of paths before they explore the next tile. In that case the Preemptive BFS method would have calculated a lot of paths for nothing. However if the player performs a lot of actions before exploring the next tile **it will have been worth it to calculate all paths preemptively**, since then we only have to look up the shortest one for each new selected tile.

The longer the game goes the larger the explored area is going to be, meaning that the player will most likely perform more actions before exploring a new tile (which will cause the shortest path map to be recalculated). And finally, once the entire level has been revealed, those paths will only have to be updated if the player builds a bridge over a Water tile. The On-demand A* method would be extremely inefficient in this situation, being forced to recalculate the shortest path to the same tiles over and over again.

Gameplay, in each level, is going to **be broadly divided into two phases** : <u>exploration</u> and <u>exploitation</u>. In the first phase the player is going to reveal new tiles, in order to find resources and have room to build their first structures. Once the entire level has been explored (or at least a large enough area), gameplay is going to shift towards exploitation : managing resources that have been found, expanding the village, and working on the various objectives.

In most levels, the exploration phase is actually going to be fairly short (about 10 to 15% of the actual playtime in a level). Players are going to spend most of their time on the fully explored level, gathering resources and managing their village. And **in that case the Preemptive BFS method is going to be way more efficient**. This is why I decided to stick with it, and only kept the other method in my scripts for future reference and comparison.

A point could be made for switching between the two methods during gameplay : using On-demand A during the exploration phase and Preemptive BFS once a large enough part of the level has been explored. I decided against it because the efficiency advantage of the first method, even in the best possible situation for it, is very marginal when compared to BFS. Only having one method is also preferable* ***if we decide to add more functionality to the pathfinding later on*** *: there's only going to be one method to modify, not two.*

One last technical point in favor of Preemptive BFS : with that method each house has its own direction map to all other tiles, that it can access at any time to display a path. Before showing a new path on screen the previous one has to be removed, which in practice means destroying the individual sections forming the path.

This is made significantly easier if each house has its own list of instantiated sections. When asked to display a new path it simply destroys all of them through their reference stored in the list, and when displaying the new path on screen it adds each new section to the list. The same technique could be used in the On-demand A* method but would require more communication between scripts, and would overall be less efficient.

3.1.2. Cycling Through Houses Manually

Now that the game automatically recommends the closest house to the action, we still need to give the ability to the player to make a different choice. When the ‘NextHouse’ or ‘PreviousHouse’ button is pressed (by default set respectively to Q (or middle mouse button) and E) the game looks in the ‘houseTiles’ list for the next available house – i.e. a house in range of the selected tile and with at least 1 available villager in it – either forward (for ‘NextHouse’) or backward (for ‘PreviousHouse’).

If no other house is available the closest house remains selected. Otherwise the next/previous available house is the new selected house, and the path is updated to reflect that change.

In most cases the player will want to stick to the closest house, but there might be some exceptions :

- Exploring a tile : see the Exploration Exception
- Collecting resources for an almost empty tile : say the player wants to collect Stone from a Stone tile that only has a few resource units left. No need to use a villager from the closest house, they might be better employed on another action, and a villager from a house farther away **will still collect all the available Stone**.
- Finishing Objectives : The player might only need a small amount of a particular resource to finish an objective. Again, no need to use the closest villager, another one farther away might still have enough time to get enough resources.

3.2. The Distance Mechanic

Whenever the player selects a tile the closest house from it is selected, and the path is shown between the two tiles. The player can then cycle through all available houses and pick a different one, in which case the new path will be shown. When a house is selected, either automatically or manually, **the distance separating it from the selected tile is saved**. Once the player finally decides to perform an action on the selected tile, that tile receives the previously saved distance.

That distance is expressed in number of moves between selected house and selected tile. That number could be 1 – if house and target tile are adjacent – or it might be very large if the path zigzags around several impassable tiles, and/or covers the entire level. Such a large number of different values would make the game confusing for the player, forcing them to carefully plan every single move and perform complex calculations. In order to keep the game simple to play I decided to **introduce a threshold system for the distance**.

There are 4 of them : *Close* (0), *Medium* (1), *Far* (2) and *Very Far* (3) (with an implicit 5th one, *Out of Range*, which is never used since a house that's too far away would never be selected in the first place). **The threshold is applied negatively to action efficiency**, in a way specific to the type of tile : some substract it directly from the amount of resource collected, some perform a more complex operation.

The simplest example is the Bushes type : there are 4 bushes on it. The player can gather fruits from *'4 – distanceThreshold'* of them. If the villager assigned to the action is at a *Close* distance they can collect from *'4 – 0 = 4'* bushes. If they are at a *Far* distance however, they only have time to collect from *'4 – 2 = 2'* bushes. The final number of bushes is **the minimum between this calculated value and how many bushes actually bear fruits when the action is performed**, to make sure the villager doesn't collect more resources that are actually present on the tile. The same method is used for Fields and Orchards, respectively with the number of lines and trees.

This distance mechanic **adds a lot of depth to the game**. It previously made no difference where houses were in relation to resource tiles, and as a result there was very little strategy in deciding where to place new houses. Now the player needs to carefully consider where to build them, and where to assign new settlers. Maybe they set up a mining area in a mountain range, and therefore have a good reason to set up a little settlement next to it, so that villagers can work the mines effectively. Same with farming areas, or lumberjack villages next to forest areas. There is **more purpose in choosing where to place houses**, which makes them more unique.

3.2.1. Calculating the Distance Efficiently

When the player selects a new tile, the Tilemap Manager has to look through its list of houses, and pick the one that's closest to the tile. Therefore we need each house to **tell the Tilemap Manager how far it is from the selected tile**.

In the initial version of the Distance mechanic, houses were recalculating their distance to each new selected tile, by counting the number of steps in the shortest path to that tile. This operation wasn't particularly expensive but there is a much easier and faster way.

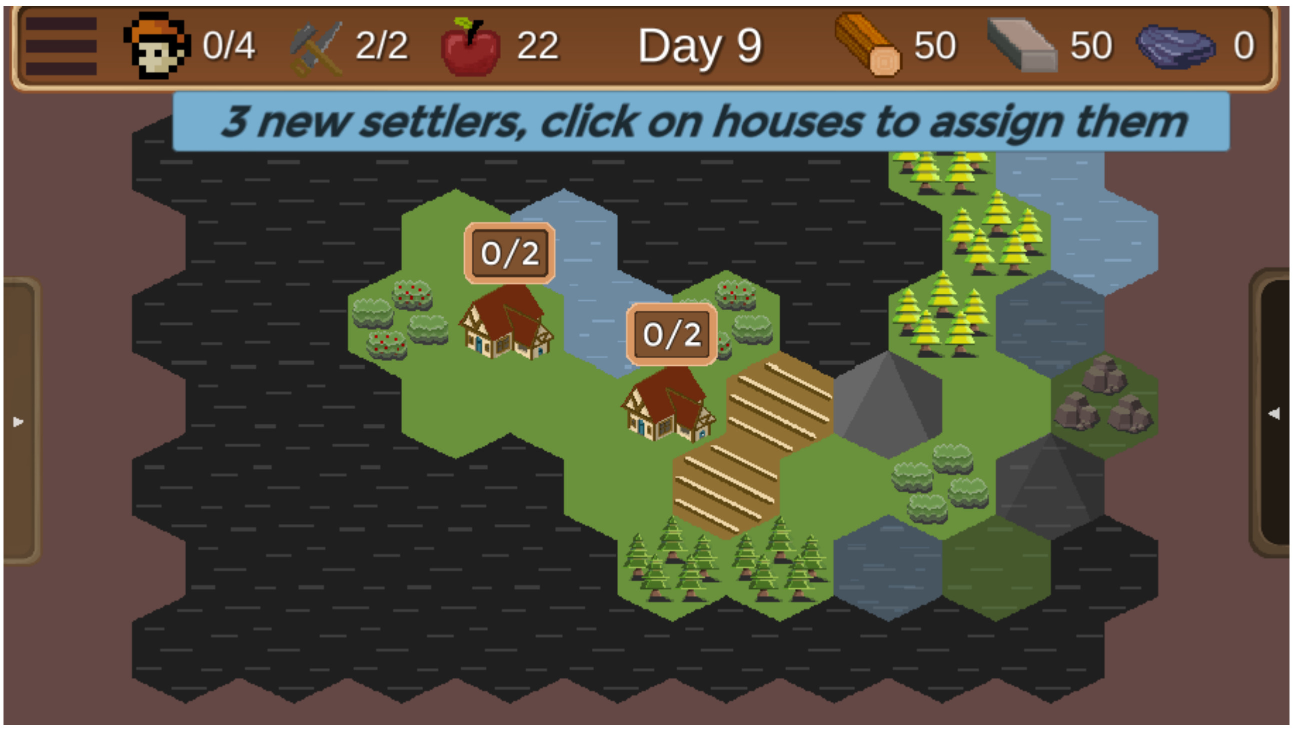
When we update the shortest path from each house to all explored tiles (this happens whenever the player explores a new tile or builds a bridge), in addition to storing the direction from the previous tile, **we also save the number of moves from the house to each tile**. The starting house has a distance of 0 to itself. Its neighbours will have a distance of 1. Those neighbours' neighbours will be at a distance of 2, and so on. In programming terms : **each tile sets its distance value to 1 + its predecessor's distance value**.

When the Breadth First Search ends we have stored **the shortest path AND the number of moves** to all accessible tiles in the level. Now whenever we need to know the distance between a house and any other tile, we simply look up the pre-calculated value in the house's path dictionary.

3.3. Caravans and Welcoming New Settlers

Every once in a while a caravan will travel through the area, carrying settlers looking for a new home. If X is the total number of available slots in houses in the village, up to X new settlers will be welcomed in the village, and assigned to houses with free slots.

If X is less than or equal to the number of new settlers there is no choice : all houses are filled to capacity with new villagers, and the additional settlers that couldn't be welcomed carry on their journey. But when X is greater than the number of settlers, meaning that the village has more available slots than settlers to place, **a choice has to be made as to which house to assign to each new settler.** Let's take a look at the following example :



Settler Assignment

On the evening of Day 9, 3 new settlers arrive in the village. We currently have two Large Houses, with a maximum capacity of 4 each. If there had been more than 3 settlers the situation would have been easy : no real choice for the player, all houses will be filled to capacity. But in this case **we have 3 settlers and 4 free slots**, which means that we have to make a choice as to which house to fill, leaving the other with 1 free slot. That choice can be made automatically or manually.

3.3.1. Automatic Mode

The simplest way is to let the game place settlers randomly. Pick a house at random, put as many settlers in it as possible, and if there are still some to place repeat with the next house and so on. Although simple, **this method might not result in the best assignment**.

3.3.2. Manual Mode

Looking back at the example above, it seems obvious that the player would rather fill the house on the right in priority. It is closer to the two fields and to the forest at the upper right corner of the map. Villagers living in this house **would be more efficient at exploiting those resources** than the ones living in the other house.

In *Manual* mode the player gets to decide where to place the new settlers. All actions (interacting with resource tiles, using keyboard shortcuts etc) are disabled while the player is assigning settlers, except left-clicking on houses. When a house is selected in that way it checks if it has any available slot, and in that case it welcomes 1 new settler. Once the player has assigned each settler to their new home the game exits SettlerPlacement mode and resumes as normal.

3.4. Objectives

Each level has its own set of objectives for the player to complete.

3.4.1. Types of Objectives

All objectives derive from the base Objective class, which defines the basic structure and behaviour of an objective : a goal value, an *Evaluate* and a *Complete* method, among others. Here are all types of derived objectives present in Hello World :

- <u>Building</u> : Build a certain number of a specific building
    - int numberGoal
    - BuildingType buildingTypeGoal

  > *Subscribed to OnBuildingCompletedEvent*

- <u>Collection</u> : Collect a certain amount of a specific resource
    - int amountGoal
    - ResourceType resourceGoal

  > *Subscribed to OnResourceCollectedEvent*

- <u>Development</u> : Reach a certain number of villagers
    - int villagerGoal

  > *Subscribed to OnVillagersArrivalEvent*

- <u>Discovery</u> : Reveal a certain number of a specific type of tile
    - int tileNumberGoal
    - int tileTypeGoal

  > *Subscribed to OnTileExploredEvent*

- <u>Exploration</u> : Reveal a certain number of tiles
    - int numberGoal

  > *Subscribed to OnTileExploredEvent*

- <u>Survival</u> : Have a least 1 villager at a specific day
    - int dayGoal

  > *Subscribed to OnNextDayEvent*

When the specific event happens during the game **each objective that listens to it update its progress and runs its Evaluate() method**. If currentValue has reached goalValue the Complete() method is called.

The ObjectiveManager of the level is in charge of initializing and updating objectives. The base class ObjectiveManager is derived into a Campaign and a Random version, which differ in the following ways.

3.4.2. Objectives in Campaign Mode

In Campaign mode objectives are set manually for each level, and will be the same every time the level is played. **Once all objectives have been completed the level ends**. In the earlier Campaign levels objectives are designed to guide the player and help them understand the game mechanics; in later levels they provide an interesting challenge based on the map layout.

Level 6 for example asks the player to reach a tile on the other side of a river (via a Discovery objective). To do so they will have to explore a large valley on the left-hand side of the level where they will find the resources necessary to build a bridge. Other levels force the player to collect a lot of food, but do not provide any bushes. All food will then have to be produced through farming, by creating and caring for Fields and Orchards.

The ObjectiveManagerCampaign initializes the level's objectives by looking at the Text file I wrote for each level, which specifies all its objectives. Here is an example of an early version of the Level 1 objectives :

```
Collection,3,10
Discovery,5,2
Collection,0,15
Survival,10|Survival,15
```

The game displays 4 objectives at once. Each line in the text file corresponds **to a specific objective slot**, from 0 to 3. The first line creates a Collection objective requiring the player to acquire 10 units of food (Food being element 3 in the ResourceType enum). The second line defines a Discovery objective asking the player to discover 2 Forest tiles. The third line is another Collection objective, requiring 15 units of Wood. Let's now look at the last line in detail.

3.4.2.1. Objective Queue

The line “*Survival,10|Survival,15*“ defines a **queue of objectives**. In Campaign levels each Objective slot contains not just 1 element, but a queue of them. Once the first objective of the queue is completed it is removed from it, and the next one takes its place. At any given time in the level only the first objective of each queue (i.e. each of the 4 slots) is active, the other ones waiting for their turn in line.

When reading the text file to initialize the level’s objectives, the ObjectiveManagerCampaign first splits the file in 4 lines, and each of these lines is then divided again in strings separated by the ‘|’ character. The line *Survival,10|Survival,15* results in the creation of an objective asking the player to survive 10 days (this objective is added in first position to slot 3 of the objectives array), and of another Survival Objective, this time of 15 days (added in second position to slot 3 of the objectives array).

Collection,3,10	==> objectivesList[0][0] = Collection(Food,10)
Discovery,5,2	==> objectivesList[1][0] = Discovery(Forest,2)
Collection,0,15	==> objectivesList[2][0] = Collection(Wood,15)
Survival,10|Survival,15	==> objectivesList[3][0] = Survival(10)
	and objectivesList[3][1] = Survival(15)

For Campaign levels I designed these series of consecutive objectives to be able to increase the total number of objectives in each level without overwhelming the player : **by only having 4 active objectives at any given time the game remains fairly simple to play**.

The other reason is to guide the player through a series of steps. Let’s say we want the player to build a Forge, as the end goal of a level. To do that they will need a certain amount of Stone. So instead of giving them a “Build a Forge” Objective right away, I start by asking them to collect the required amount of Stone. The corresponding line in the text file would be :

Collection,1,25|Building,5,1

The first numerical value is the type of goal (Stone is the second element in the ResourceType enum (so its index is 1) and Forge is the sixth element in the BuildingType enum (so its index is 5)), and the second is the amount/number of this resource/tile to reach for the objective to be completed.

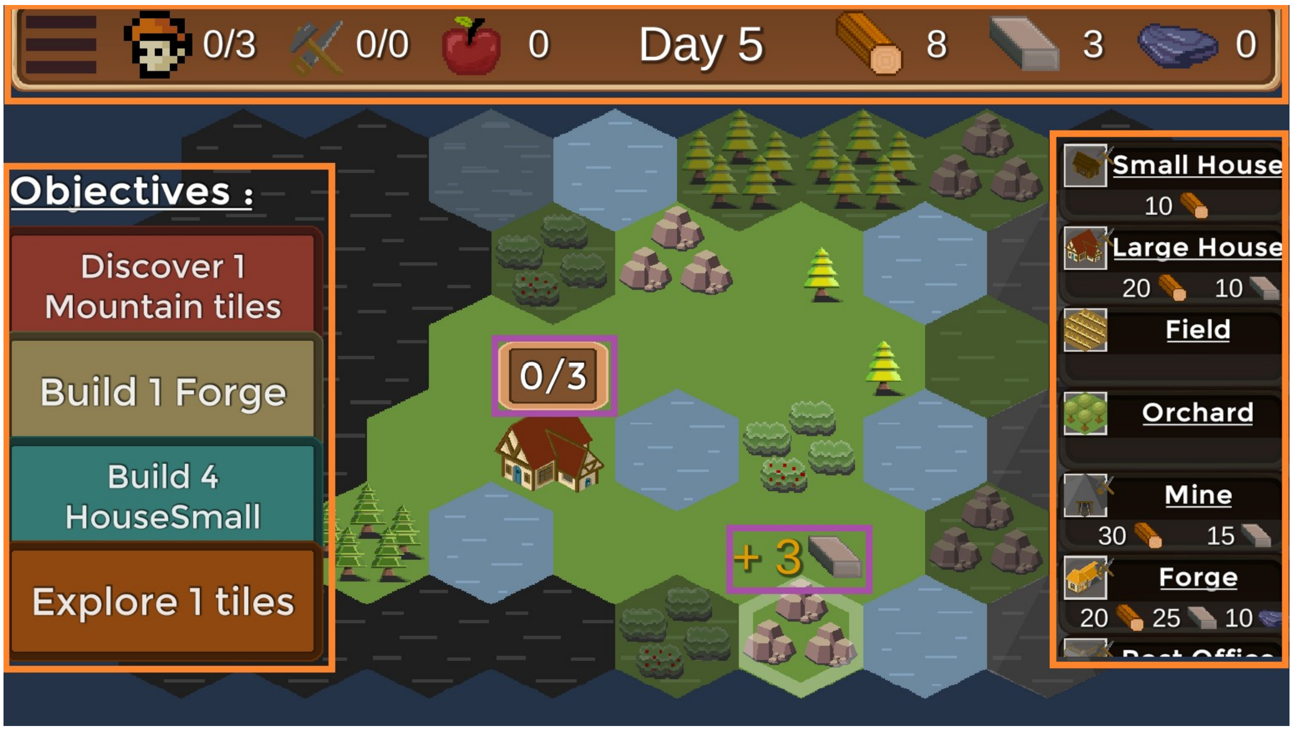
3.4.3. Objectives in Random Mode

In Random Mode objectives are not pre-set. Instead the ObjectiveManagerRandom initializes them randomly at the start of the level (after it’s been fully generated), and **every time any of them is completed by the player it generates a new one to replace it**. In this mode there is no need to use an Objective queue for each of the 4 slots : there is only going to be 1 objective in each slot at any given time.

It would certainly be possible to design an algorithm that would create consecutive series of logical objectives, similarly to what I did for Campaign mode, but it would be quite long to implement and I am not sure the benefit would be that great. With such a system Random mode would end up being fairly similar to Campaign mode; on the other hand, by picking a completely new random objective each time **this mode feels more entertaining**, more surprising, and provides a very different experience to the player than Campaign mode.

4. User Interface

4.1. Static & Dynamic UI Elements



In Hello World **some UI elements are static while others are tied to positions in World Space**. In the above example elements highlighted in *orange* are static : they always appear at the same place on the screen. They are rendered in Screen space, disconnected from the game world. Elements in *purple* are dynamic, **their position depends on where a specific object is placed in the game world**. The house tooltip for example appears a little bit above the center of the House tile it is related to. The *+3 Stone* resource Popup is also spawned at the tile location and slowly moves upwards while fading out.

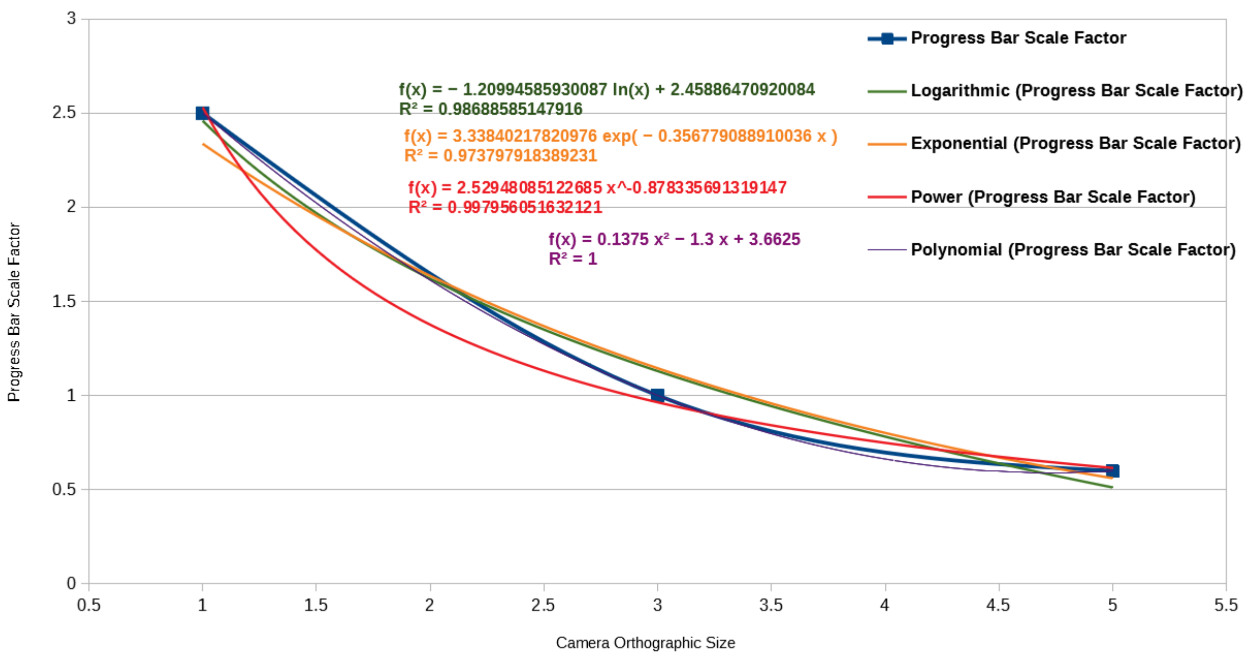
If the game view was fixed, both static and dynamic UI elements could be rendered in Screen space and there wouldn't be any problem. But since the player can move the camera around the level as well as zoom in and out, the position and size of dynamic UI elements has to be modified so that they remain in the same World position, instead of the same Screen position. If the player moves the camera to the left in the above example both the house tooltip and the resource popup should remain on top of the tile and not keep their screen position.

4.1.1. First Method : Moving and Resizing at Runtime

The first method I tried involved **recalculating the position and size of all dynamic UI elements** whenever the camera moved or zoomed (it would respectively send a *OnCameraMovedEvent* and *OnCameraZoomedEvent* to all elements that had previously subscribed to them). This required finding the right scaling factor, which I did for each type of dynamic UI elements (font size doesn't scale exactly like images for example) by hand-picking values for the two extremums and the middle point, and generating a curve based on these values.

After some testing it appeared that the right scaling function for most UI elements in the game was not quite linear. For the size of the progress bar values were :

Camera Orthographic Size	1	3	5
Progress Bar Size	2.5	1	0.6



Scaling Factor of Progress Bar Size

This resulted in this polynomial regression, indicated in purple on the graph (logarithmic, exponential and power trendlines are shown for reference). This complex calculation had to be performed by each dynamic UI elements every frame whenever the camera zoomed in or out. Visually the result was perfectly fine but it wasn’t very efficient...

4.1.2. Second Method : Rendering in World Space

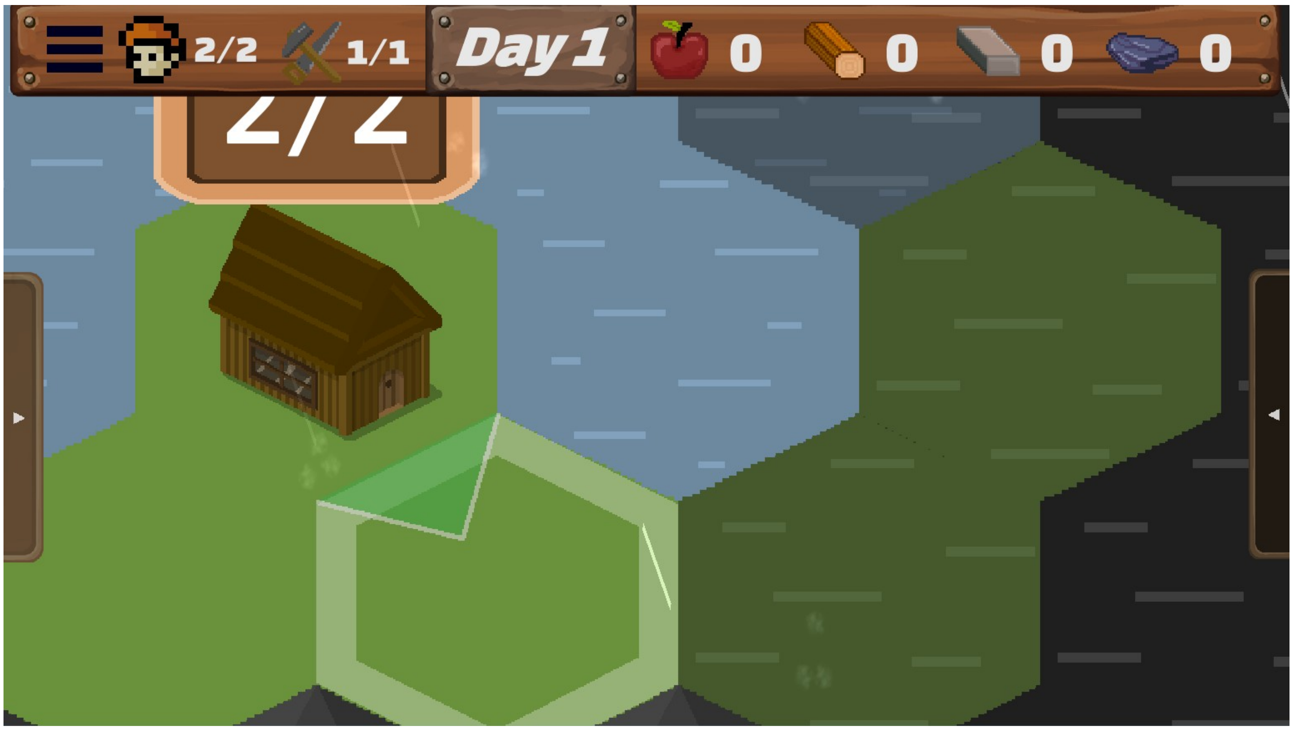
To avoid performing all these calculations I switched to a different method. I added a second Canvas to the game, **set to render in World space instead of Screen space**, and modified my scripts to instantiate all dynamic UI elements on this canvas instead. Some manual resizing was required, since all values in those elements’ prefabs were in pixels and now had to be expressed in Unity units.

Once scaled properly to account for the new render mode, those elements act as would any object in the game world, so when the camera moves or zooms they stay at the same world position. No more calculations required !

4.2. Canvases & Cameras

As explained above, **Hello World’s UI elements are rendered either in Screen Space** (for the static elements like the Resources Panel and the Objectives/Construction menus) **or in World Space** (House tooltips, Building progress bars; anything that needs to be displayed at a position relative to an object in the game world). Those UI elements are part of the Screen Space Canvas and the World Space Canvas respectively.

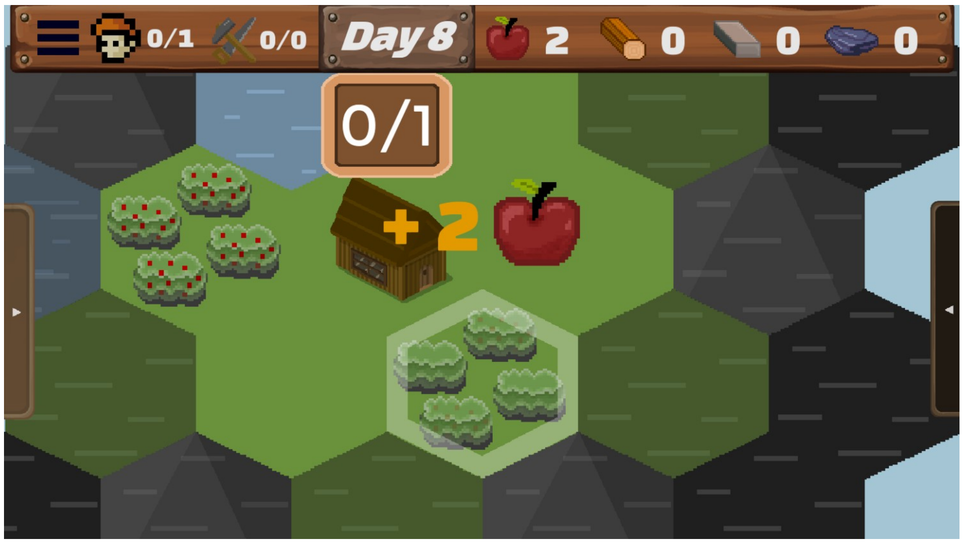
Most visual elements in the game are rendered by the LevelCamera, whose Culling Mask is set to everything *but* the UI layer. The Screen Space Canvas is rendered by the UI Camera, whose Culling mask is limited to the UI layer. Those two cameras have their ‘Depth’ values set up so that **every frame the LevelCamera renders the level first** (as well as all World Space UI elements), with a colored background for all empty areas, **and then the UICamera renders the UI on top of it**. This results in the following behaviour :



*House Tooltip being rendered behind the Resource Pane*l

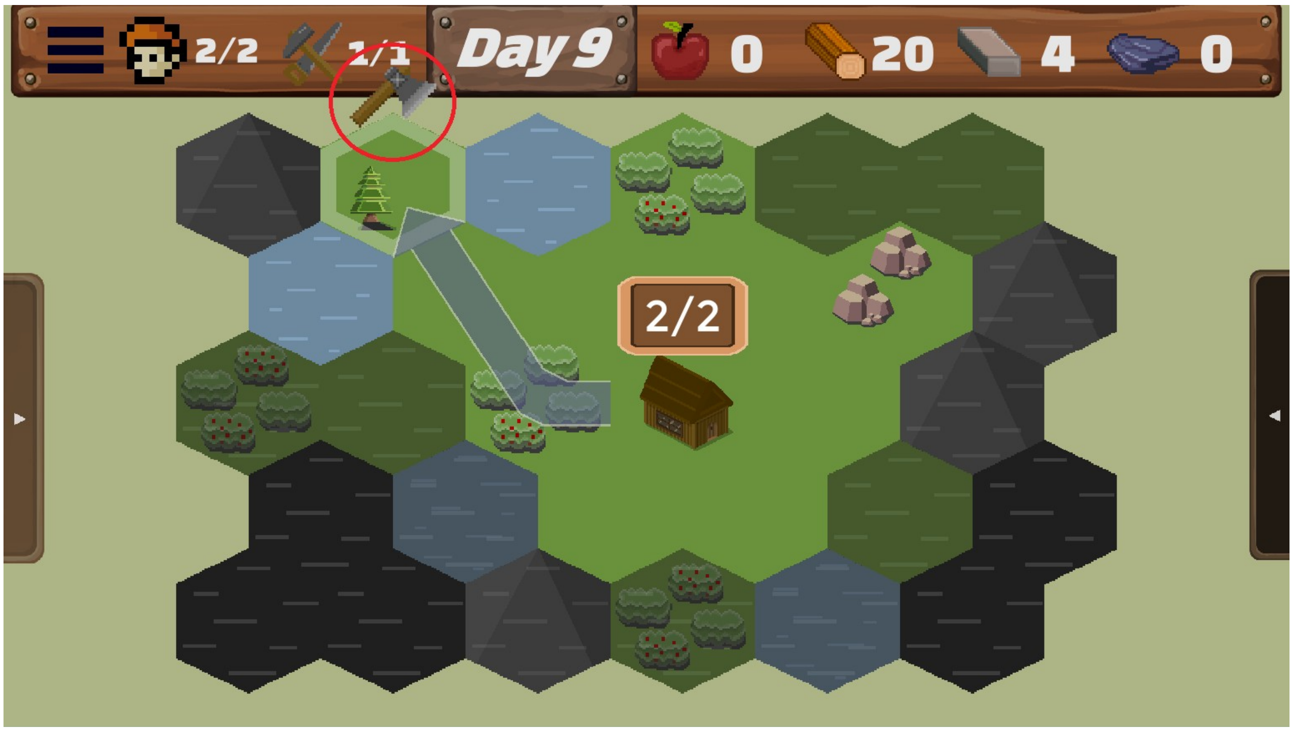
Most of the time this is what we want. If the Tooltip was rendered on top of the Resource Panel **it would obstruct important information** (in this case the Villager and Tools values). For the same reason we don't want the Building progress bar, Radial Menu and Post Office Tooltip to appear on top of the static UI. There are however two exceptions to this rule : **the Tile Action Preview and the Resource PopUps**.

Whenever the player selects a tile a preview sprite of the default action is shown on top of the tile, to indicate which action will be performed if the left mouse button is clicked. After an action is performed, if it yields some amount of resources, a little pop up indicator is displayed above the tile, moving vertically for about 1 second before fading out. In this screenshot the player has just collected fruits from the Bushes and the Pop Up has been displayed :



Example of a Resource PopUp

But what happens when the tile clicked is at the top-most row of the level ? The PopUp would appear, start going up and quickly disappear behind the Resources Panel. Similarly, there wouldn't be enough space for the Default Action Preview sprite to be displayed in full; some part of it would be behind the UI. One solution would be **to increase the amount of empty space between the top-most row of tiles and the Resources Panel**, but that wouldn't look right… In my opinion a better solution is to change the rendering behaviour of the Action Preview and PopUps, to make them show on top of the Resources Panel.



Desired Behaviour of an Action Preview being rendered on top of the UI

We could also move the Action Preview down a bit so it would fit under the Resources Panel, as well as spawn PopUps at a lower position, but this would cause them **to appear somewhere in the middle of the tile** instead of above it, which risks being confusing for the player. For all these reasons I decided to have Action Previews and PopUps render above all UI elements. The rendering order should then be :

1. Tiles and most World Space UI elements (House tooltips, Progress Bars...)
2. Static UI (Resources Panel, Objectives and Construction menus)
3. Action Preview and PopUps

This actually proved fairly complex to implement, and required delving into the particular way that Unity handles Cameras and Canvases.

Could we achieve the desired 3-step rendering behaviour with only two cameras ? The first group of World Space UI elements would have a specific layer (say *'Layer1'*), static UI elements would be set to *'Layer2'*, and Action Previews and PopUps would both be on *'Layer3'*. Then we would set the Culling Mask of LevelCamera (i.e the layers that it renders) to *'Layer1'* and *'Layer2'*, and the Culling Mask of UICamera to *'Layer2'* only. As nice and easy as this sounds it unfortunately doesn't work, before of camera ordering.

Each frame cameras are asked to render all layers present in their Culling Mask in the order defined by their *‘Depth’* field. **Cameras with lower *‘Depth’* values are rendered first**. So say we set LevelCamera’s *‘Depth’* to 0 and UICamera’s to 1. The LevelCamera would render first. During this rendering call it would render *‘Layer1’* first (all tiles, House tooltips, Progress Bars etc) then *‘Layer3’* (Action Preview and PopUps) on top of it. Once LevelCamera has finished rendering it would be UICamera’s turn; it would render *‘Layer2’* (the Static UI elements) on top of whatever LevelCamera has already rendered (by the way this involves setting the ‘Clear Flags’ field of UICamera to *‘Depth Only’* or *‘Don’t Clear’*, otherwise the UICamera would completely override what was rendered by any previous camera).

Following this order of rendering calls we can clearly see that even though *‘Layer3’* is supposed to be on top of *‘Layer2’*, because the camera rendering it (i.e. LevelCamera) renders before UICamera, it still ends up being shown behind the Static UI. In other words **the order of Layers is only relevant within each Camera rendering call** : each Camera renders layers in their specified order (*‘Layer1’*, then *‘Layer2’*, then *‘Layer3’*, skipping those that are not part of their Culling Mask), but any other camera rendering afterwards will go through the same process. So LevelCamera renders *‘Layer1’* and *‘Layer3’*, but then *‘Layer2’* is then **rendered on top of both of these layers** because it is rendered by another camera (UICamera).

We could try to have LevelCamera render after UICamera but we would end up with the opposite problem : both *‘Layer1’* and *‘Layer3’* would be rendered on top of *‘Layer2’*. Since *‘Layer1’* is the tile layer, we would see **tiles displayed above the Resources Panel**, which is about as undesirable a behaviour as we can imagine.

The main take away from all of this is that **this desired behaviour can only be obtained with a third Camera**, which is solely in charge of rendering *‘Layer3’*. This camera (let’s call it PopUpCamera) has a depth value higher than both LevelCamera and UICamera. The setup is :

- LevelCamera (CullingMask : *‘Layer1’*, Depth : 0)
- UICamera (CullingMask : *‘Layer2’*, Depth : 1)
- PopUpCamera (CullingMask : *‘Layer3’*, Depth : 2)

Then, as long as UI elements are set to the proper layer, everything should go fine. House Tooltips and similar World Space elements should be rendered behind the Static UI, and Action Preview and PopUps on top of it. Well yes, but actually no. At least not quite...

4.2.1. Canvases Layer Precedence

After some testing it appeared that in **Unity Canvases "override" the Layer of their children UI elements**. It doesn't actually matter what Layer a particular element is set to, if that Layer is different from its parent Canvas', only the Canvas' Layer will be considered.

Let's consider a Canvas set to *'Layer1'*. This Canvas has a child UI element set to *'Layer2'*. We have two Cameras in the scene, Camera1 rendering only 'Layer1' and Camera2 rendering only *'Layer2'*. If we turn off Camera2… nothing happens. The UI element set to *'Layer2'* is still rendered, even though the only Camera supposed to render it is now inactive. If we turn Camera1 off however, the UI element disappears.

For this simple test we can understand that a Canvas' layer takes precedence over any of its children elements' layers. As a result, **it's impossible to obtain our desired behaviour with only two canvases**, even if each Layer is rendered by a specific camera. **Each camera also needs its own Canvas**. As a result I had to split the World Space Canvas into two : a MainWorldSpaceCanvas (set to *'Layer1'*) and a PopUpWorldSpaceCanvas (set to *'Layer3'*). And now, as long as the Culling Masks of all three cameras are set appropriately (LevelCamera → *'Layer1'*, UICamera → *'Layer2'* and PopUpCamera → *'Layer3'*), we get the desired behaviour : *'Layer1'* is rendered behing *'Layer2'* and in turn *'Layer3'* is in front of *'Layer2'*.

Note : After some more testing – and more time spent thinking about it – it seems that this is not so much about the Canvas overriding the Layer of its children than the fact that canvases are rendered as a whole. A camera doesn't look at each individual UI elements to decide which one it should render, it only considers canvases. If a canvas' layer is part of the camera's culling mask, then this camera renders it with all its children elements. One UI element cannot be rendered independently from its parent canvas.

5. Saving and Loading

Hello World features a Save and Load system, **allowing the player to keep their progress between sessions**. Data is saved for each of the 3 available game slots. Settings are common to all 3 slots. Each slot can have one Campaign level and one Random level saved at any given time. The Start menu allows the player to start a new game, load an existing save file (if any exists) and change the game Settings.

Once a game has been started or loaded, the Main Game menu is displayed, which allows the player to access the Campaign and Random level modes. Additionally, if a Campaign level save file has been found in the game's save folder **a button appears to let the player continue that level**. The same is true for Random level mode.

Once in a level players can save their progress at any time by clicking on the Save button in the menu. Additionally an autosave is performed at regular intervals by the game itself.

Hello World's saves are stored in a persistent data path, which depends on the operating software (provided by Unity in *Application.persistenDataPath*). Settings are stored in the root save folder, in a 'settings.ini' file. Data relative to a specific Save slot is placed in that slot's folder (/save1, /save2 or /save3). Each of those folders contains a 'helloWorld.save' where the general data is stored for this save slot (see *5.2. General Game Data*). In addition each folder can hold a 'level.save' and/or a 'randomLevel.save', respectively for the current Campaign level and Random level.

5.1. Game Settings

The Settings menu lets players change all important game options : camera speed, music/sound volumes etc. An Advanced Settings is also available for more particular settings, mostly concerning Random Level generation. Most players would have no need for these advanced settings, which is why **this section is collapsed by default**, and the Random Level Debug section of the Advanced Settings would not even be available in a release copy of the game.

The Visual Debugging option slows down execution of the Random Level Generation process, showing its progression step by step. It was very useful during development and is a convenient way to show the inner working of the Generation algorithm to other people. Please note that **Visual Debugging is way more detailed in Corrective mode than in Preemptive mode**, development on the latter having been halted when it became apparent that Corrective mode was way more efficient (see *Chapter 2* of this document for more details). Preemptive mode would actually not be available in a release version of the game, and is kept in the development version for comparison purposes only.

Whenever the player clicks on the Apply button in the Settings menu, all settings are updated and saved in ‘settings.ini’. No validity check is needed when performing that save since **all UI elements of the Settings menu** (sliders, dropdowns, input fields etc) **have their bounds set to the appropriate valid values**. For example it is impossible to enter letters in the Autosave Frequency field, and the maximum numerical value allowed is 999 days. As a result when Apply is clicked we know that all Settings values are valid.

The ‘settings.ini’ file is organised in a simple way. Each line contains two fields : the Setting’s name and its value. The latter will be a numerical value, a boolean or a string depending of the field.

When loading from the ‘settings.ini’ file, when the game starts, we need to be careful. This file is deliberatly not encoded, players have access to it as long as they know where the save folder is located. This means that **they are free to modify it is any way they want**, and when accessing our Settings we need to make sure they are still valid. The file can be invalid for two reasons :

- <u>Invalid Value</u> : The user changed a field’s value, and this value is outside the valid bounds. Say they opened the ‘settings.ini’ file and changed the music volume to 250 (the valid range for this setting is [0,100]). Values can also be invalid if they cannot be parsed into the right type. For example the Camera Zoom Speed value expects a numerical value; if the user replaced it with a word, the value will be invalid.

When the LoadSettings method detects that a value is invalid for a specific field, it will **reset the in-game setting to its default value** (the GlobalGameManager holds all default values for each setting). Once the entire file has been read, if we encountered at least one invalid value, we generate a new valid ‘settings.ini’ file in place of the previous one; all valid settings are kept, only invalid ones are reset to their default values.

- <u>Invalid Field</u> : The user changed a field’s name, and this name is not a recognized Setting name. This is more problematic, since **we can’t tell if they altered an existing field or created a brand new one**. We could implement a string-analysis system to try to determine if the invalid field is close enough to an existing one and correct it, but considering the ‘settings.ini’ file was altered deliberately it seems better to re-create it from scratch for safety. So whenever we encounter an invalid Setting name in the file, we reset all Settings to their default values and create a new valid ‘settings.ini’ file.

There are other kinds of file alteration that Hello World’s loading system will be able to deal with, without having to generate a new ‘settings.ini’ file. If more than two fields are present on a line, all subsequent ones are simply ignored. Also, if the user changed the order of the fields – by moving lines up or down in the file – the loading system will still work.

5.2. Slot Specific Data

Settings are common to all save slots and are saved in the root save folder. Each Save slot has its own General Game Data, and up to one Campaign level and one Random level. Those files are stored in the subfolder dedicated to the specific game slot (/save1, /save2 or /save3).

5.2.1. General Game Data

This is the data that is not relative to the current level. It contains information on the best time saved for each level and the total number of completed levels. **This data is saved in the ‘helloWorld.save’ file**, which is created whenever a new game is started in a save slot, and updated during each session played in that slot. Since the player is not supposed to access it directly, the game uses a Binary Formatter to serialize its content.

5.2.2. Level Saving

Any level (whether it’s in Campaign or in Random mode) will contain a certain amount of data : **current resources, weather values, objectives, as well as information relative to each tile** (its type, state, whether or not it’s still fogged etc). All this information is handled by a base LevelData class. Similarly to the General Game Data, all level save files are serialized with a Binary Formatter to make it harder for the player to tamper with them.

When it comes to tile data, **some of it is present in all tiles**, while some is specific to one or several types. The former includes the tile type, its current state, whether or not it is currently fogged and if it’s accessible or not (this last one could be recalculated when loading the level but it seems easier to just save it in the file). For this common data, the save file contains 2-dimensional arrays the size of the level, containing all generic values for each tile.

Tile-specific data however, **only makes sense for certain tiles**. For example only House tiles have a field storing the number of occupants. Only Mountains have a field for the amount of metal remaining in their tunnels. Farm tiles have a few specific values concerning their watering levels and number of days without water.

For this type of data we could also use a level-size grid, but most tiles would have no use for it. It would slow down the loading process slightly (more data would need to be deserialized) for no good reason. For this reason I picked a different saving method. **All tile-specific data has its own list**, initially empty. When looking through the level grid to access each tile’s values, if we reach a tile with any specific data (say a house) that data is added to the appropriate list.

Once we have looked through the entire level, all common data has been written to the file, and we have filled all tile-specific data lists. That’s when we save those values to the file, each list on its own line. Let’s consider this example of a simple 3x2 level :

(0:1) House (2 occupants)	(1:1) House (4 occupants)	(2:1) Field (water = 3)
(0:0) Grass	(1:0) Field (water = 1.5)	(2:0) Water

The save file for these tiles would look like this (well, except in binary form) :

[Type]	*[HasFog]*	*[State]*	*[IsAccessible]*
0	(true/false)	(tile's state)	(true/false)
7	(true/false)	(tile's state)	(true/false)
9	(true/false)	(tile's state)	(true/false)
7	(true/false)	(tile's state)	(true/false)
1	(true/false)	(tile's state)	(true/false)
9	(true/false)	(tile's state)	(true/false)

[House occupants]	2	4
[Field water levels]	1.5	3

Note : the [HasFog], [State] and [isAccessible] variables are saved for each tile to their actual value when the game is saved. What's important for this particular example is not their values but simply the fact that they are fields present in all tiles, independently of their type, as opposed to the [House occupants] and [Field water levels] variables which are only present in certain tiles.

If there were more houses in the level their occupants value would have been added at the end of the [House occupants] list (similarly for Fields), in ascending order of x and y coordinates. When loading this data **we recreate the level by instantiating tiles in the order found in the file**. In this example we instantiate a Grass tile at the first grid position (0:0). When created this tile runs its Start() function, in which it initializes its values from the data found in the file (hasFog, state and isAccessible), at its position. Since the Grass type doesn't have any specific values we keep going.

Now we instantiate the House tile found on the second line. In its Start() function this tile initializes its basic values (just like the previous Grass tile), **but it also needs a value for its 'occupants' variable**. This value is not found in the grid but in the *[House occupants]* list at the end. Since this House tile is the first one to be instantiated its value is the first one in the list (2). **But how does it know it is the first one ?**

Since each tile gets initialized on its own, it doesn't know by default how many other tiles of its type have already been initialized. I give them this information by using a static int counter for each tile type, set to zero before loading the level. When a tile gets initialized it takes the value at the 'counter' index of the appropriate value list, and increments the counter to indicate that one more tile of its type has been initialized. In our example, the first House instantiated sees that the houseCounter is at 0, takes the value at index 0 of the *[House occupants]* list, and increments houseCounter to 1. When the next house gets instantiated it will take the value at index 1 (which is 4) and increments houseCounter to 2. If there were more houses the process would continue in the same way.

We use the same method for all types of tiles that have any data that's specific to them : Mountain, House, FarmTile (which is the base class for both Field and Orchard) and Bridge. **This allows the game to only save tile data for types in which it is actually relevant** (no point in having an 'occupants' entry for Grass tiles for example, or a 'Water Level' for a Mountain).

The only condition for this method to work is that tiles are loaded in the same order that they were saved in. We simply need to always look through the level in ascending order of x and y coordinate, which is the most intuitive approach anyway.

Finally it is worth noting that **this method would be made simpler by using Queues instead of List**s. Each tile could simply remove the first element of the appropriate queue, removing it in the process, and letting the next tile access the element now in first position. This would eliminate the need for static counters. But unfortunately the Binary Formatter used in this project cannot serialize Queues/Stacks, only arrays and Lists.

5.2.2.1. Campaign Level Saving

The CampaignLevelData class inherits from LevelData, adding a few values that are specific to Campaign mode. In this mode the Weather and Caravan systems use a cycling pre-set schedule. We need to keep track of the number of changes that have occurred since the start of the level in order to reset the level properly when loading it.

5.2.2.2. Random Level Saving

In Random mode Weather and Caravan systems don’t run on a cycling schedule, instead they only generate the very next set of values. It also needs to keep track of some counts used by the ObjectiveManagerRandom to make sure it creates valid objectives. Those values are saved in the RandomLevelData class, which also inherits from LevelData.

6. Various Notes

6.1. Distance between two hexagons

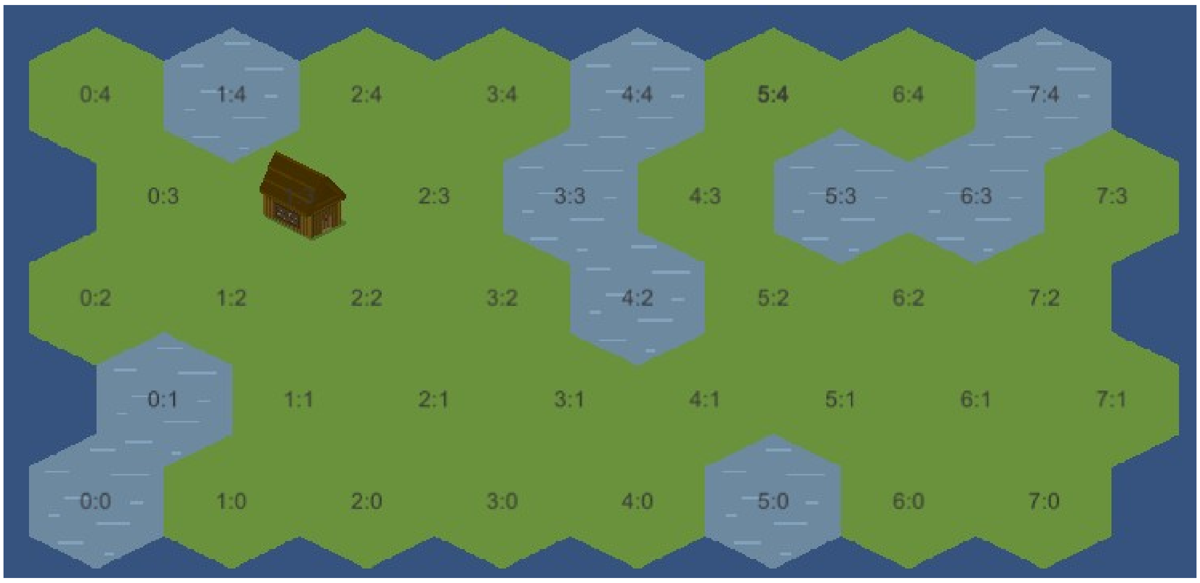
```
   FORMULA FOR DISTANCE BETWEEN TWO HEXAGONS
              (a0:b0) AND (a1:b1)

x0 = a0 - floor(b0 / 2)
y0 = b0
x1 = a1 - floor(b1 / 2)
y1 = b1
dx = x1 - x0
dy = y1 - y0
dist = max(abs(dx), abs(dy), abs(dx + dy))
```

6.2. General Hexagon Math

RedBlobGames has a fantastic blog with tons of useful information on various programming and algorithmic concepts. Their page on hexagon math is very well done :

https://www.redblobgames.com/grids/hexagons/

6.3. Hexagon coordinates : Avoiding the odd-even row distinction

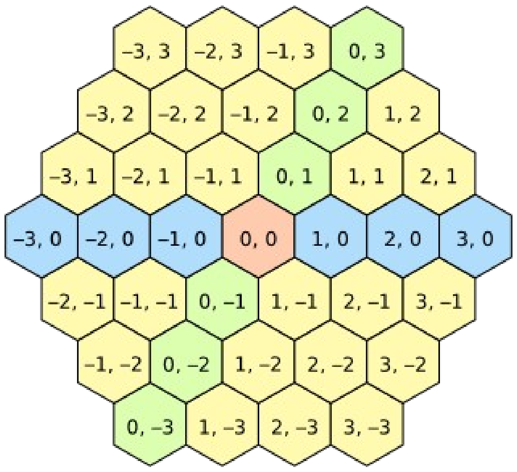


Hello World Hexagon Coordinates

Hello World uses a traditional Hexagon setup, where each column "zig-zags" vertically; when following a column upwards the first neighbour is to the top-right, the next one to the top-left, then top-right again etc. This implies that when returning the coordinates of a specific neighbour the game has to check the row index and apply an offset to the coordinates.

During my initial research on the topic I stumbled into a post on StackOverflow by an user named *Glorfindel*. He explains how to avoid the even-row distinction in an hexagon setup, by shifting everything 60 degrees to the right :

I once set up a hexagonal coordinate system in a game so that the y-axis was at a 60-degree angle to the x-axis. This avoids the odd-even row distinction.



(source: althenia.net)

The distance in this coordinate system is:

```
    dx = x1 - x0
    dy = y1 - y0

    if sign(dx) == sign(dy)
        abs(dx + dy)
    else
        max(abs(dx), abs(dy))
```

You can convert (x', y) from your coordinate system to (x, y) in this one using:

```
    x = x' - floor(y/2)
```

So dx becomes:

```
    dx = x1' - x0' - floor(y1/2) + floor(y0/2)
```

Careful with rounding when implementing this using integer division. In C for int y floor(y/2) is (y%2 ? y-1 : y)/2.

I found this to be an interesting idea, but I still decided to stick to a more traditional system. Using Glorfindel’s setup I would have had to create “slanted” levels to maintain coordinate alignment. Unless we want the first colum, from bottom to top, to have coordinates (0,0), (0,1), (-1,2), (-1,3), (-2,4) etc, which would have been more confusing in the long run.

All things considered the system I decided to use is not really that complex. We only have to write one function returning neighbour coordinates, in charge of calculating and applying the appropriate offset, and can then forget all about it. We can simply call this function asking for a specific neighbout (the top-right one for example), and never have to worry about its inner workings. All in all it seemed better than to mess with either the shape of levels or the consistency of vertical coordinates.

6.4. ‘=’ != ‘-’

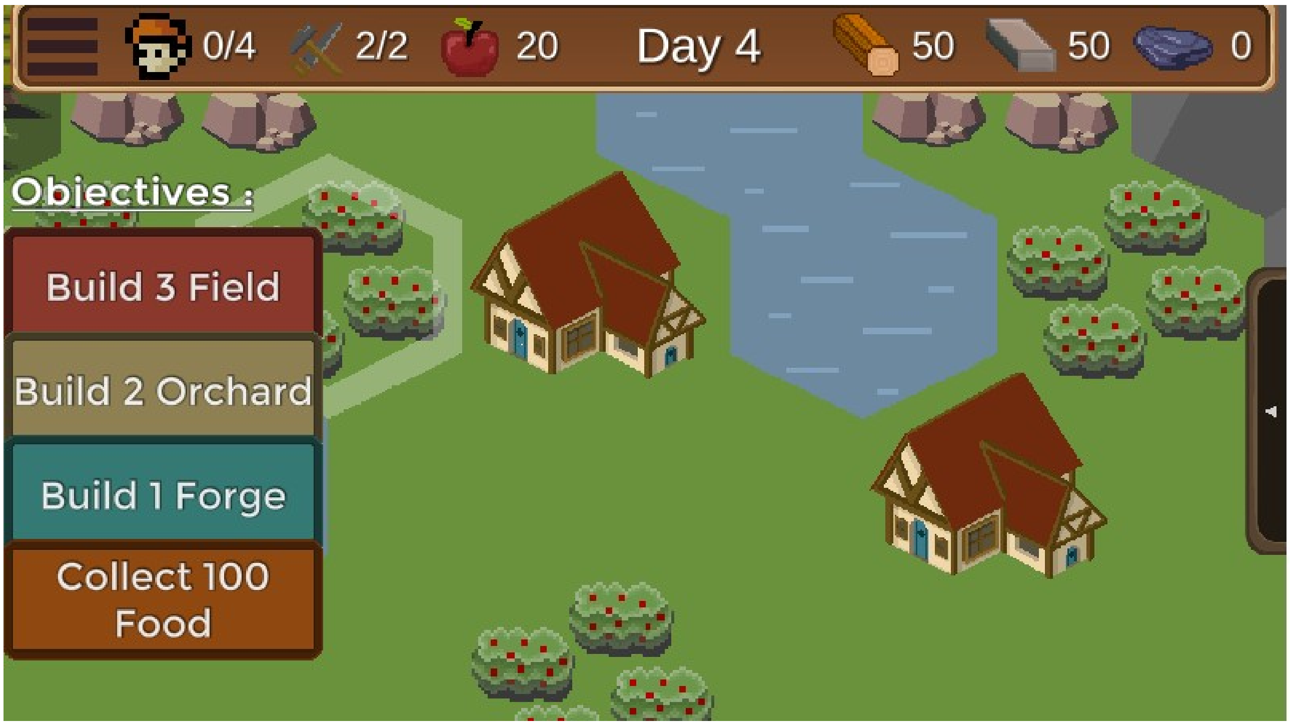
I may or may not have spent a few hours fixing a problem that crashed the game during the Level Generation, and was caused by the stupidest thing. When writing the method calculating the distance between two tiles I accidently put a ‘=’ sign instead of a ‘-’ in what was meant to be a substraction. Something like “new Vector2Int (target.x – start.x, target.y = start.y)”.

Yeah so it turns out the compiler doesn’t see anything wrong with that. I guess it’s technically legal to perform an assignation like that, but still… (no really, why would you do that). So no compiler error and when using Debug.Logs to try to identify the issue it looked like the game crashed during a yield return call. The Log before the yield was printed to the console, but not the one after, as if the coroutine was not resuming its execution.

This was actually completely misleading and I am still a bit unsure as to why the other Debug Logs were not printed to the console. The Coroutine was in fact resuming normally and the problem was in a entirely different method. So moral of the story : compiler warnings are great but don’t really on them too much !

6.5. Ignoring UI Elements with IsPointerOverGameObject

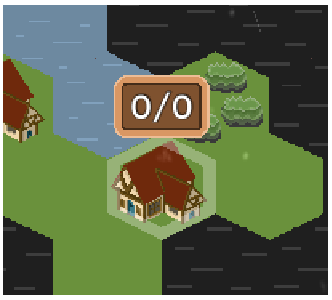
Since some parts of the UI (Construction and Objectives menu) can be displayed over the level I need to make sure that **when the player clicks on a button they do not accidently interact with a tile behing the UI panel**.



Objective Panel Over Tiles

To check if the mouse is over a UI element I use "EventSystem.current.IsPointerOverGameObject()". If it returns true then any action that would be performed on the tile under the UI element is cancelled. In the above example clicking on the "Build 3 Fields" objective would not trigger an action on the bushes behind it.

But after adding the House tooltips I ran into a slight problem. The tooltip is displayed above the house, as shown in the picture below and hovering the mouse cursor over it would cause IsPointerOverGameObject() to return true.



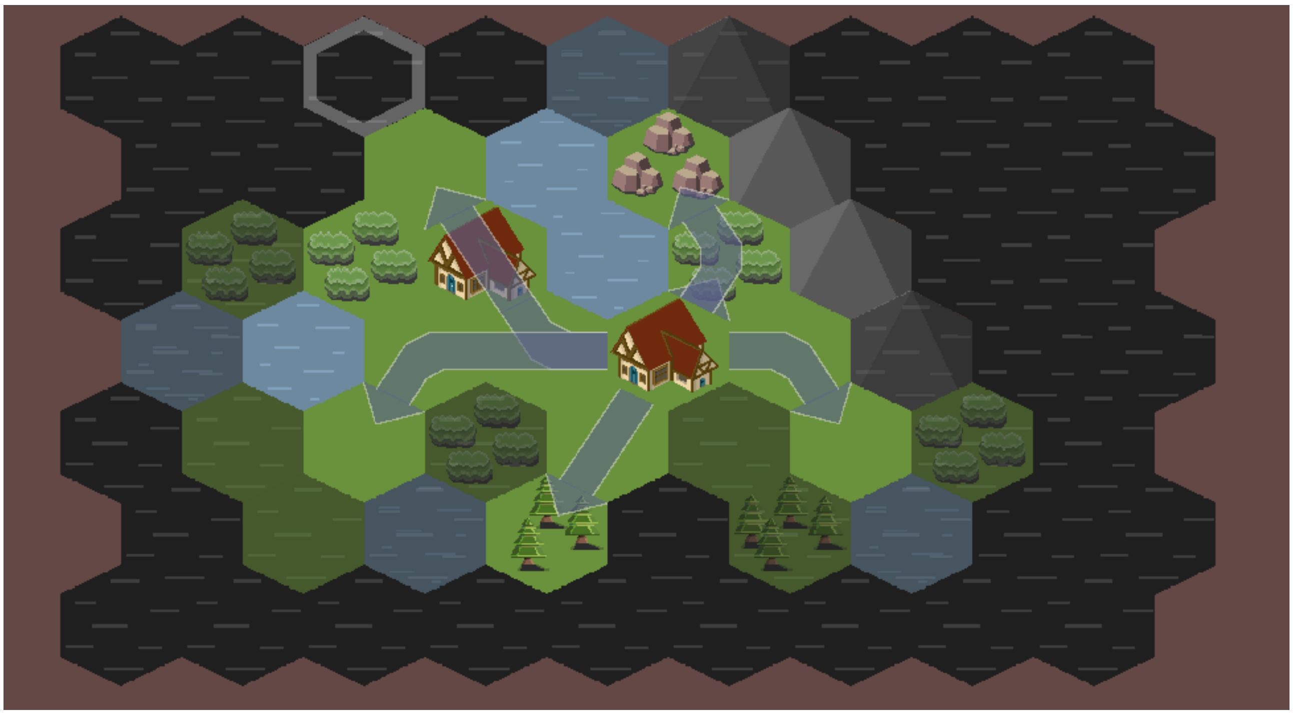
The problem concerns the tile highlight, this white outline that the game displays to indicate which tile is currently selected by the player. When a tile detects the mouse cursor entering its area (i.e.

when the OnMouseEnter() method is called on that tile), it places the tile highlight at its position.

In this example the frontier between the House and the Bushes tile is partially under the tooltip, meaning that moving the cursor there would cause the tile highlight to not be moved properly. It would stay on the House tile. I first tried to set the layer of the House tooltip to ‘Ignore Raycast’, to no avail.

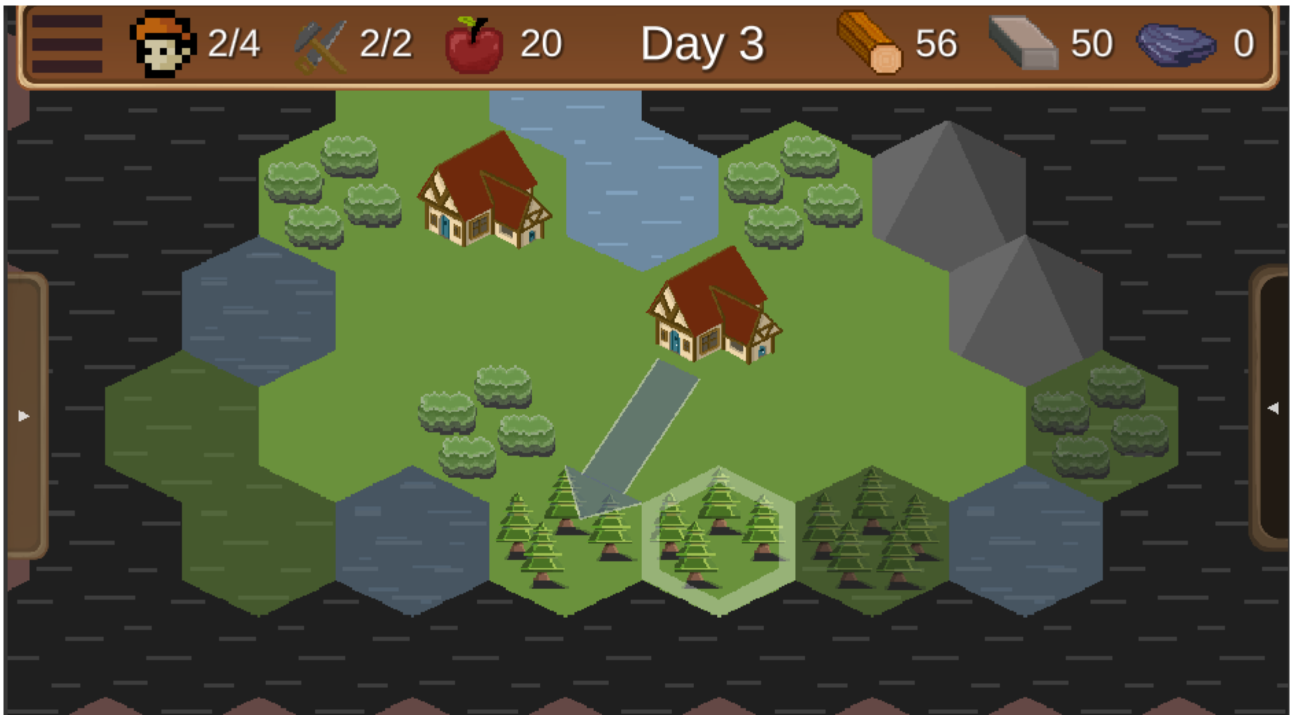
The solution was actually to untick the ‘Raycast Target’ setting **in both elements of the tooltip** (the Image background and the TMPro text component). Now the IsPointerOverGameObject() call returns false when the mouse cursor goes from the House to the Bushes tile, even when it’s over the tooltip. The white outline is now properly moved to the newly selected tile. This highlights the inner working of IsPointerOverGameObject(), which uses a Raycast to check for collision with UI elements.

6.6. Shortest Path Between Two Adjacent Tiles



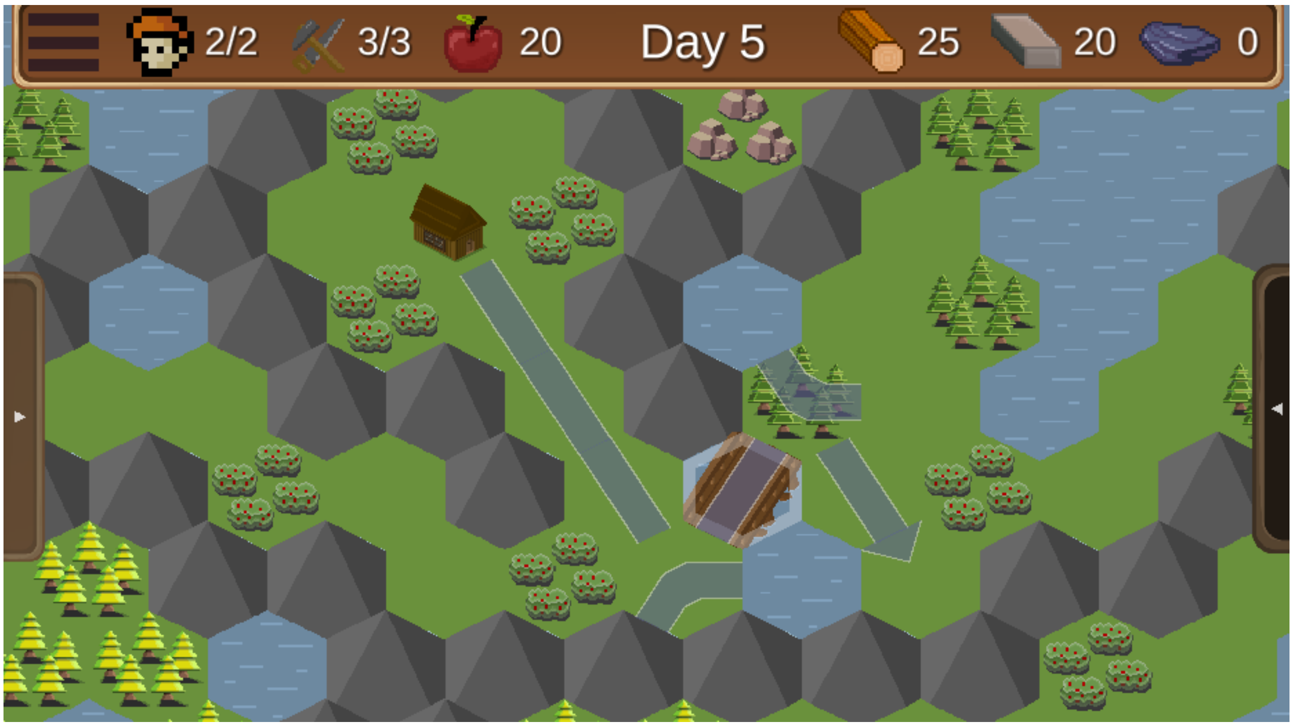
Each of these Arrow paths is made up of several sections, one for each tile in the path, picked based on the direction from the previous tile and the direction to the next one. When drawing those section sprites I assumed that I only needed 9 of them – not counting the arrow tips : 1 horizontal, 2 diagonals, and 6 for wide turns.

I deliberatly omitted the 6 sharp turns section, i.e. going from one side to a directly adjacent side. The reason for this omission was that I thought **the shortest path between two adjacent tiles was always going to be direct from one to the other**. A sharp turn would imply an additional step through another tile. Let's look at this example :



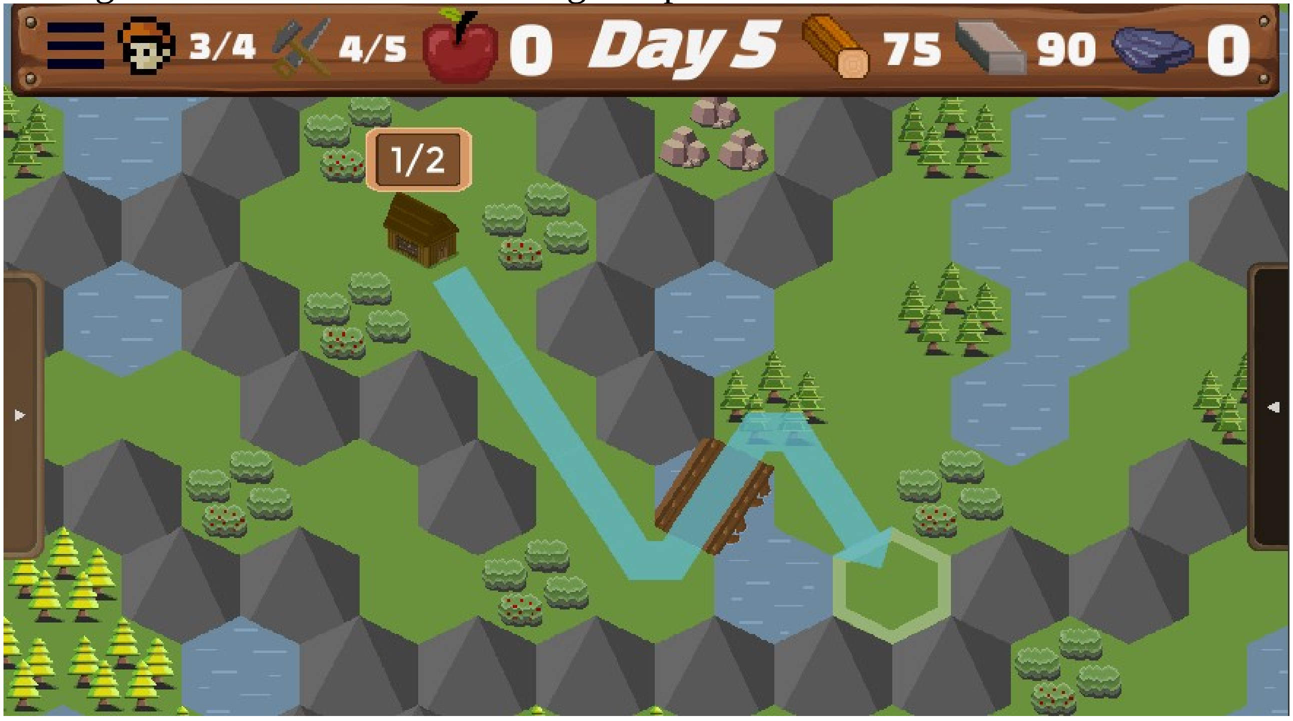
Shortest Path on Land

The shortest path from the house to the left-most Forest is obvioulsy a straight line between the two. Surely there is never going to be a situation in which it is better to go through the highlighted forest tile and then take a sharp turn to reach the forest on the left ? This is indeed correct for most tiles, but not always for bridges.



Confused Arrows

Because Bridges are only passable from specific angles we can end up in this interesting situation. The shortest path to the bridge from its Grass neighbour on the left does involve going through the other Grass tile at the bottom of the bridge. **This is a case where the shortest path between two adjacent tiles is not direct**, and because sharp-turn sections were not available in the project at the time the method displaying the path had no idea what to do, and displayed the wrong section before and after the bridge. Back to my image editing software to draw the missing sharp-turn sections !



Much better