Project documentation Artificial intelligence for videogames

BY FEDERICO FIORIO 08276A

1 INTRODUCTION

The aim of the project is to design and implement an A.I. technique that could be used in a Videogame enviroment.

This project is about a technique called procedural content generation, in particular the generation of trees on a terrain.
In a nutshell procedural content generation is a method to create game content in an algorithmic way, the content provided can be generated on its own or can be used to assist designers to perform a better job.

The main idea behind this project it's to automatically generate trees with the help of L-systems, and then, fill an entire terrain enviroment with these random generated trees.

2 THE DESIGN

The project is based on the cocept of L-systems.
L-systems or Lindenmayer systems are a parallel rewriting systems and a type of formal grammar.
When we define a L-system, we define three main elements to describe them:

- The variables, or in other words, the type of elements that our system can interpret;
- The axiom, or in in other words, the initial sentence of the system;
- The rules, or in other words, they describe how elements change inside the system e.g: (A –> AB)

Lindenmayer a.k.a. the creator of the L-systems utilized these methods to describe plant development, in the same way I used L-systems to describe the structure of a tree in a 3D environment.
To create randomness inside the creation of trees, some rules can be ignored in this way the trees won't look exactly the same, they might be similar but not the same.
Other things that can give the idea that the trees are different from one to another are the height and the angle between branches.
In this way it's impossible that two trees will look exactly the same, and in a forest-like environment,

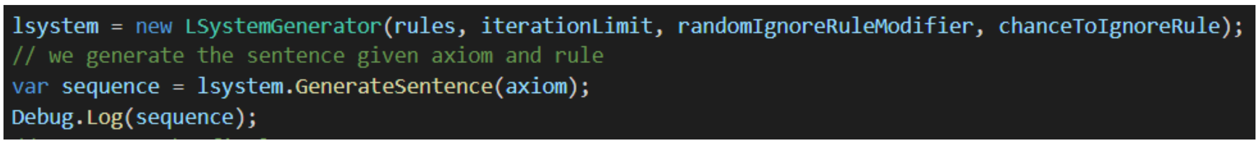
with many trees the player will be amused by the randomness of nature, like in the real world and will not find the same tree in the same map.

The spawn of multiple trees on a playable environment will might take some time and some computational power, but this process will be taken in the loading screen, for example, between two different environments.
I want that the technology inside this project can be used in different environments and so there is the possibility to spawn different types of trees, maybe we can have a summer-like environment with trees with leaves and another environment that might be more like winter, with balding/dead trees.

The terrain on which the trees will be spawn it's basically a procedural content generated environment with Perlin noise, but the Perlin noise part won't be in the scope of this project.
The terrain will have slopes and hills, it can be also flat if desired, but I assume for this project to work on environments without floating platforms or anything like that.

3 IMPLEMENTATION

The implementation rely on the base code given by professor Gadia about the processing of a sentence in the L-systems.
I used the code of professor Gadia to generate the final sequence that is later parsed and transformed into a tree.
I will not go into details of this part.



```
lsystem = new LSystemGenerator(rules, iterationLimit, randomIgnoreRuleModifier, chanceToIgnoreRule);
// we generate the sentence given axiom and rule
var sequence = lsystem.GenerateSentence(axiom);
Debug.Log(sequence);
```

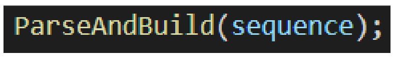
Fig. 1 – Professor Gadia's code

That code will generate the final sequence, something like:

$[F[X[[[+F] - FL][-X+]][* - X-][+FL] - FL][/ - X * +]]$

This sequence is then parsed and a 3D tree will be obtained.

The tree is created using the turtle graphics technique, the position and rotation of the turtle is where the next branch or leaf of the tree will be created.
The rendering of the tree happens inside the parsing method:



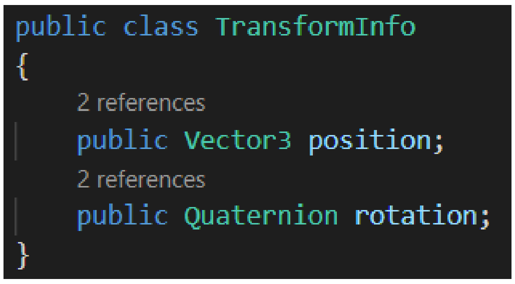
```
ParseAndBuild(sequence);
```

Fig. 2 – Parsing the final sentence and render it

In total there are 9 possible variables that the parsing method can read, 6 variables are used for rendering ('+', '-', '*', '/', 'F', 'L'), 1 variable, the 'X' it's not rendered but just used for the production rules and 2 variables ('[', ']') are used for creating different branches that the turtle can travel.

3.1. Push and Pop

[,] are used for managing a LIFO stack of TransformInfo, it is used to manage the position and rotation of the turtle.

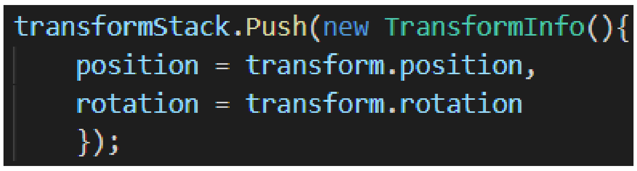


```
public class TransformInfo
{
    2 references
    public Vector3 position;
    2 references
    public Quaternion rotation;
}
```

Fig. 3 – TransformInfo struct

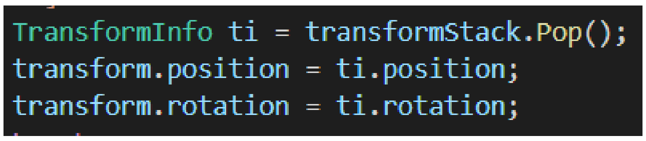
TransformInfo is a struct that stores a position and a rotation, it will be useful later to keep track of positions and rotations to correctly build a tree structure, without this stack the branches and the tree-like structure couldn't be created correctly because without it wouldn't be possible to turn back to a saved position and rotation to create branches of a tree. The **[,]** rules are used to manage this stack:

- **[** : saves current position and orientation of the turtle (push);
- **]** : retrieves last saved position and orientation, and resets the turtle to that position (pop);



```
transformStack.Push(new TransformInfo(){
    position = transform.position,
    rotation = transform.rotation
    });
```

Fig. 4 – Push: **[**



```
TransformInfo ti = transformStack.Pop();
transform.position = ti.position;
transform.rotation = ti.rotation;
```

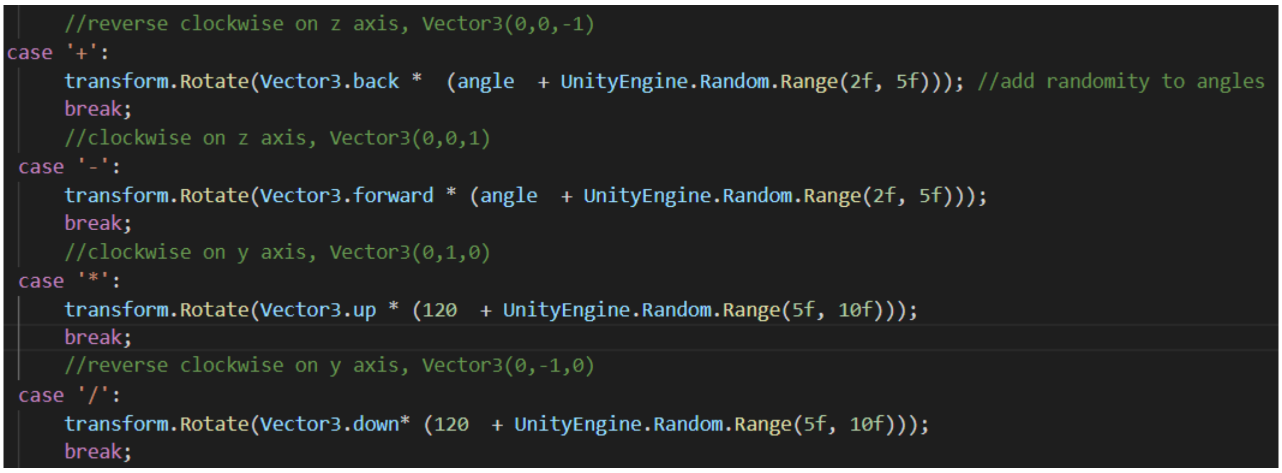
Fig. 5 – Pop **]**

3.2. Managing the turtle's rotation

Four other variables are used to menage the branches of the tree in the Unity's space, they are used to give different rotations to the branches and let the tree grow in 3D.

The four variables utilize the $transform.Rotate()$**rotate** method, it adds the specified rotation to the current rotation of the turtle so I can use it to specify different angles for the branches of the tree, the turtle's rotation is changed locally to the tree's object rotation and position.
NOTE: I don't need to adjust the height of the tree because it will grow by instantiating the branches of the tree.
The four variables are:

- **+**: reverse clockwise on z axis;
- **-**: clockwise on z axis;
- *****: clockwise on y axis;
- **/**: reverse clockwise on y axis;



```
    //reverse clockwise on z axis, Vector3(0,0,-1)
case '+':
    transform.Rotate(Vector3.back *  (angle  + UnityEngine.Random.Range(2f, 5f))); //add randomity to angles
    break;
    //clockwise on z axis, Vector3(0,0,1)
 case '-':
    transform.Rotate(Vector3.forward * (angle  + UnityEngine.Random.Range(2f, 5f)));
    break;
    //clockwise on y axis, Vector3(0,1,0)
 case '*':
    transform.Rotate(Vector3.up * (120  + UnityEngine.Random.Range(5f, 10f)));
    break;
    //reverse clockwise on y axis, Vector3(0,-1,0)
 case '/':
    transform.Rotate(Vector3.down* (120  + UnityEngine.Random.Range(5f, 10f)));
    break;
```

Fig. 6 – The 4 variables of the L-System used for rotating the turtle

The *angle* can be specified in the Unity's Editor, anyway its default value it's 20 and it's less than 120 specified for the rotation on y axis.
If you think about it, it's normal because the rotation on y axis needs to be larger, otherwise all the branches of the tree will be all close together, meanwhile the rotation on the z axis needs to be smaller, otherwise the branches will not point towards the light as in nature, but they will acquire strange angles.
The $UnityEngine.Random.Range$ method keeps the trees different between each other making the result more 'realistic'.

We can better visualize the intuition between the difference angle sizes in the following image, yellow it's for y, red for x and blue for z.
As we can see if we imagine that the cylinder is the trunk or a branch of the tree, the angle needed on y it's larger than the one needed for z, in fact on y we got a range that could be 360 degrees, meanwhile on z the range will be something around a maximum of 180-220 degrees, otherwise the branches spawned will be considered not right.

That's because I want to find the right trade-off to make the tree grow higher and also larger, I don't want the branches to look too close each other (too small angles) and I don't want the tree to look like a bush, or create some circumferences (too big angles).

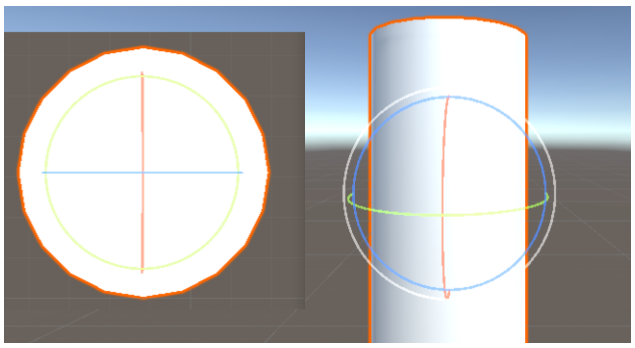


Fig. 7 – Intuition of the angles size difference on y and z axis

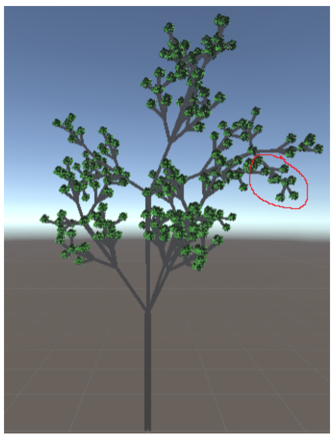
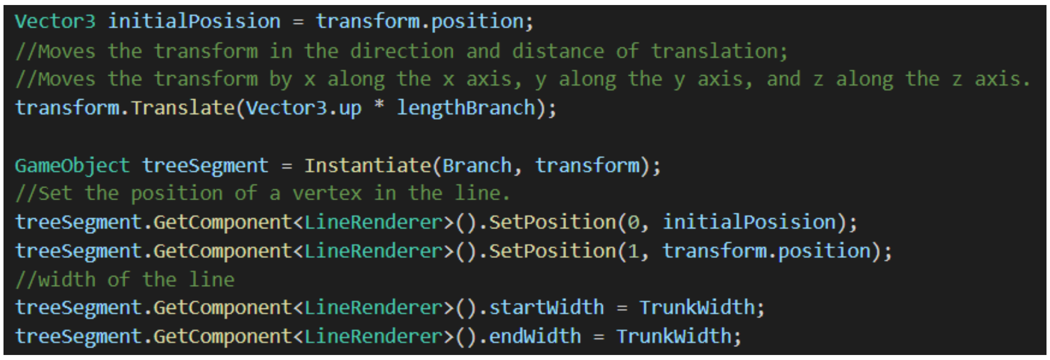


Fig. 8 – Example of a branch pointing towards terrain, the shape of the tree has still grown in height

3.3. Managing the turtle's position (a.k.a spawn branches and trunk)

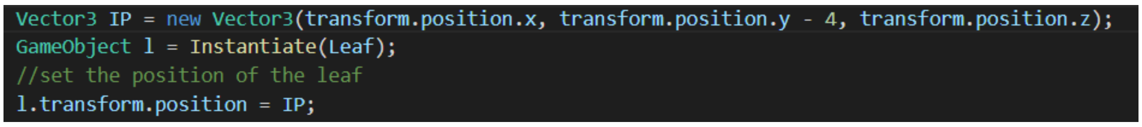
The turtle's position is handled by the variables **F** and **L**.
the **F** is responsible for moving the turtle forward and at the same time spawns a line that is a part of a branch or trunk of the tree, the turtle moves as far as the length of the line.
The **L** is responsible for creating a leaf where the turtle is located, the leaf it's a GameObject and it is created with the texture of an external package in Unity called RTP Vol.1.

I thought that for the implementation of the tree's parts, the Line component was more manageable than other GameObjects like the cylinder, in fact I can easily change the height of each line and make the trees even more random than before.



```
Vector3 initialPosision = transform.position;
//Moves the transform in the direction and distance of translation;
//Moves the transform by x along the x axis, y along the y axis, and z along the z axis.
transform.Translate(Vector3.up * lengthBranch);

GameObject treeSegment = Instantiate(Branch, transform);
//Set the position of a vertex in the line.
treeSegment.GetComponent<LineRenderer>().SetPosition(0, initialPosision);
treeSegment.GetComponent<LineRenderer>().SetPosition(1, transform.position);
//width of the line
treeSegment.GetComponent<LineRenderer>().startWidth = TrunkWidth;
treeSegment.GetComponent<LineRenderer>().endWidth = TrunkWidth;
```

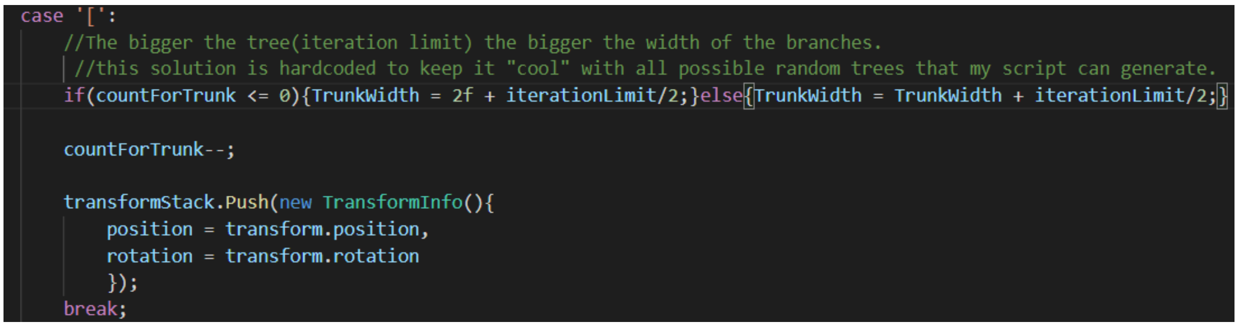
Fig. 9 – F variable



```
Vector3 IP = new Vector3(transform.position.x, transform.position.y - 4, transform.position.z);
GameObject l = Instantiate(Leaf);
//set the position of the leaf
l.transform.position = IP;
```

Fig. 10 – Leaf spawning

Obviously the trunk of a tree is larger than its branches, so I decided to change the width of the first spawned branches to make them bigger and look like the trunk of the tree.
This part it's not so modular, but I couldn't do this trick with changing the grammar of the L-system, since it is created iteratively.
So I decided to create a counter for the **[** in the L-system's sequence, until the second **[**, I assume that the branches are trunk, and so the width of the Line object is larger.
It's not the best since if I have a different structure of the tree it might produce not so beautiful results, it is also good to say that usually you don't pop until the trunk has been created, so basically 99% of time, until the first **[** it's always trunk; with the second **[** the problems might start for different structures of L-Systems, but I decided to consider only the grammars that I created, for them, this rule always works.

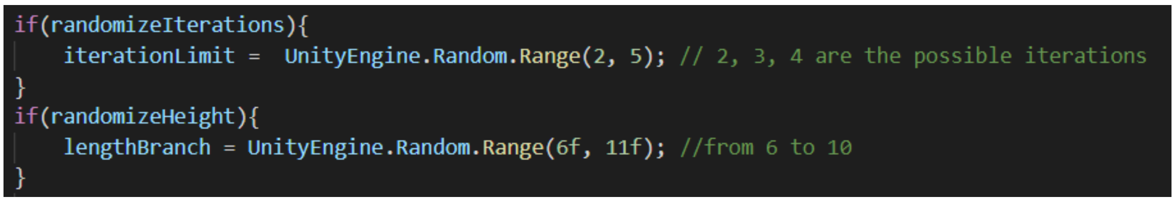


```
case '[':
    //The bigger the tree(iteration limit) the bigger the width of the branches.
    //this solution is hardcoded to keep it "cool" with all possible random trees that my script can generate.
    if(countForTrunk <= 0){TrunkWidth = 2f + iterationLimit/2;}else{TrunkWidth = TrunkWidth + iterationLimit/2;}

    countForTrunk--;

    transformStack.Push(new TransformInfo(){
        position = transform.position,
        rotation = transform.rotation
        });
    break;
```

Fig. 11 – Trunk explanation, the count it's inside the case for the '[' variable

3.4. Randomness

Randomness it's added in 4 different things, the first it's in the L-system's sequence, some rules can be ignored and so the final sequence with high probability will not be the same for each tree.
The second type of randomness it's the one talked in the previous sections that came from the angles sizes.
The third one comes from the randomness of the length of the branches and trunk that will make the tree grow higher or smaller.
The last one it's the randomization of the iterations for the last sequence.
Obviously different iterations in different trees will have different results, if we want to create an environment with similar size of trees, I would highly recommend to turn this option off, since a tree with an iteration of 4 it's a lot bigger than a tree with iteration 2.

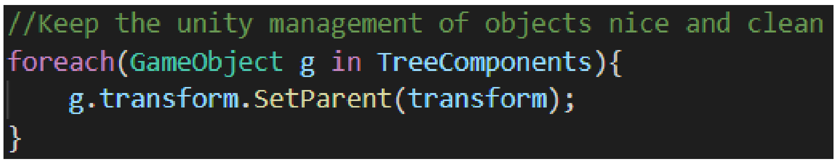


```
if(randomizeIterations){
    iterationLimit =  UnityEngine.Random.Range(2, 5); // 2, 3, 4 are the possible iterations
}
if(randomizeHeight){
    lengthBranch = UnityEngine.Random.Range(6f, 11f); //from 6 to 10
}
```

Fig. 12 – Height and iterations randomness

3.5. Unity's hierarchy

To keep every branch and every single leaf under the tree that spawned them, I need to add every GameObject spawned in a list and then use this list to add the elements to the tree's hierarchy.
This process is done with a list because otherwise, if I added every element to the hierarchy after the spawn, than the localPosition of the leaves would have broken.



```
//Keep the unity management of objects nice and clean
foreach(GameObject g in TreeComponents){
    g.transform.SetParent(transform);
}
```

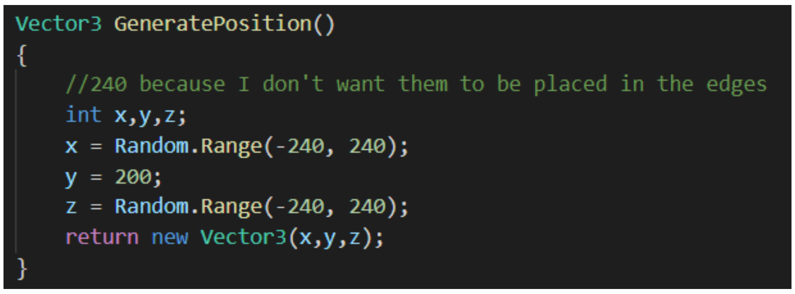
Fig. 13 – Unity's hierarchy handling

3.6. Spawning many Trees

Until now we talked about how to spawn a single tree with randomness and to keep it different from the others.
Now we will talk on how I decided to spawn every single tree on a changing ground, more specifically

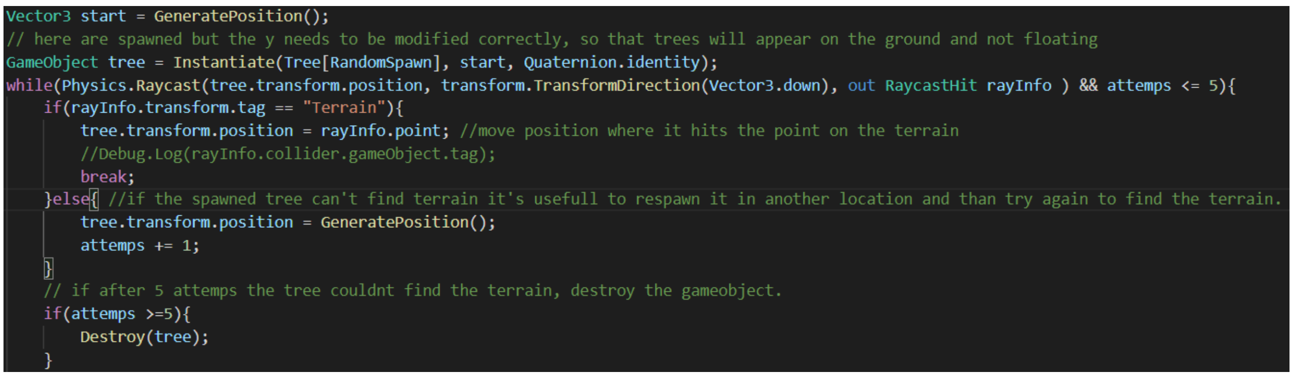
a terrain created with Perlin noise.

Taken into consideration that the terrain object it's 500x500 meters big, I wanted to spawn the trees inside it and also not on the borders of it, this is why the range for the spawn for x and y it's in a range from -240 to 240, in fact the terrain it's positioned on the origin of the scene.
The trees are then spawned on a fixed height, that is higher than the possible highest mountain created the terrain, and then, the position of the tree is changed in order to bring the tree on the ground, where on the ground means the first point, that belongs to a GameObject tagged as "Terrain", hit by a Raycast that starts from the tree and points down.
This implementation might not work if the design of the game requires some floating platforms or something similar (or just tag the floating platforms not with "Terrain"), but this is not my case.
I also implemented a way to respawn the trees that don't hit the Terrain GameObject, these trees will try to respawn a fixed number of attempts and then, if they can't find it, they will be destroyed and the map will still be created but without some trees. (this was not strictly required since the trees don't have a collider and the raycast won't hit the already spawned ones).



```
Vector3 GeneratePosition()
{
    //240 because I don't want them to be placed in the edges
    int x,y,z;
    x = Random.Range(-240, 240);
    y = 200;
    z = Random.Range(-240, 240);
    return new Vector3(x,y,z);
}
```

Fig. 14 – Generate the position of the trees



```
Vector3 start = GeneratePosition();
// here are spawned but the y needs to be modified correctly, so that trees will appear on the ground and not floating
GameObject tree = Instantiate(Tree[RandomSpawn], start, Quaternion.identity);
while(Physics.Raycast(tree.transform.position, transform.TransformDirection(Vector3.down), out RaycastHit rayInfo ) && attemps <= 5){
    if(rayInfo.transform.tag == "Terrain"){
        tree.transform.position = rayInfo.point; //move position where it hits the point on the terrain
        //Debug.Log(rayInfo.collider.gameObject.tag);
        break;
    }else{ //if the spawned tree can't find terrain it's usefull to respawn it in another location and than try again to find the terrain.
        tree.transform.position = GeneratePosition();
        attemps += 1;
    }
    // if after 5 attemps the tree couldnt find the terrain, destroy the gameobject.
    if(attemps >=5){
        Destroy(tree);
    }
```

Fig. 15 – Generate the trees and place them on the ground

3.7. Unity's editor

For better visualization of the trees, even if the terrain is huge, the trees might not scaled well with it, but this is all on the designer's shoulders, the terrain on my scene might be considered huge for some types of games and small for others, it all depends on the design of the final game.
This problem can always be adjusted by decreasing the iterations and branch's length/width, (in code, it is hardcoded because it's the right length/width to make visualize all the possible trees that the script can generate) also the leaves scale might be changed, but again, this is not a problem since it all depends on the design of the game.

There are 2 scenes in the project, one it's for visualize the 2 kinds of trees that I created with different grammars, and the other one that is the project itself.

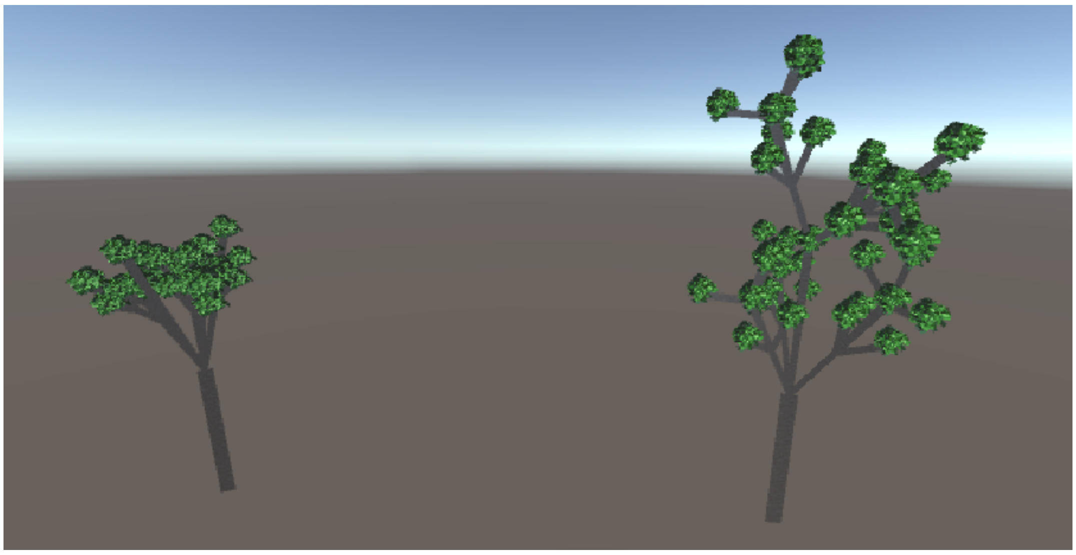


Fig. 16 – TreeSpawn scene

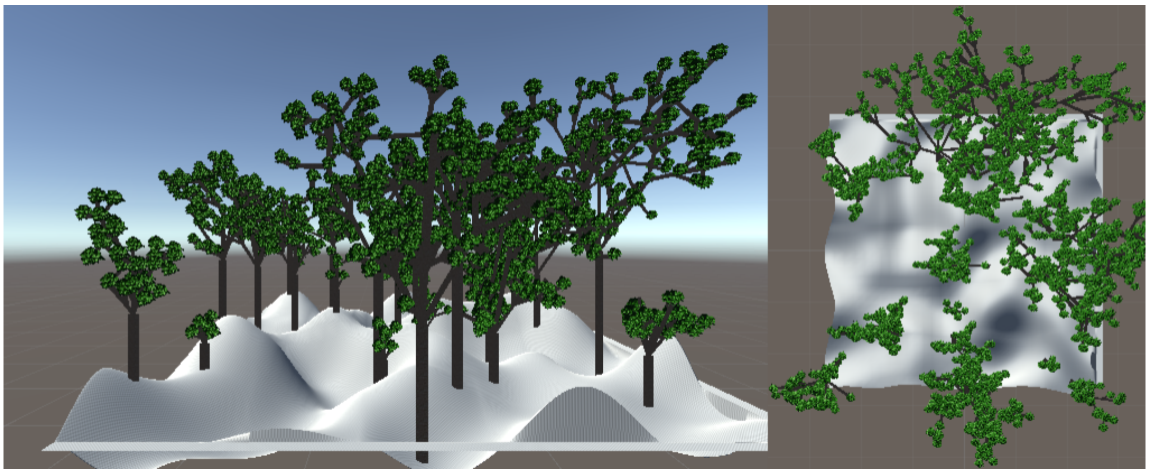


Fig. 17 – Final project