Rev. 0
07/07/2015

HEXASURFACE

A project by Alvin Sartor
& AnanasProject

SUMMARY

HexaSurface is a data structure developed for Unity Engine that can be used as powerful tool to create environments of any kind.

As the name suggests, the basic unit of the structure is the hexagon. This geometrical shape is versatile and permits us to create a graph that we can control from the top, as a single object.

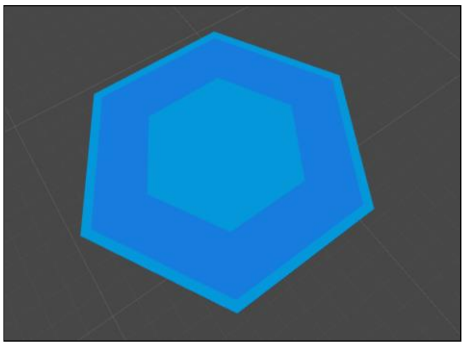
Once we have a graph, we can put quite anything on it: in this way, we will be able to create quickly randomized environments and we will have the support of the invisible data structure to perform actions. For example, we could have the need of spawning objects in a determined place, finding path in a short time, obtaining information about the terrain we are walking on, etcetera.

An important aspect of the structure is that it is born to work during run-time, so do not expect to build anything, you just have to give him the models you want to spawn and set some options, the structure will think about the graph shape, the units placing, the model choosing. How? Well… magic of course!

No, I am joking, I will explain everything on this documentation.

STRUCTURE DESCRIPTION

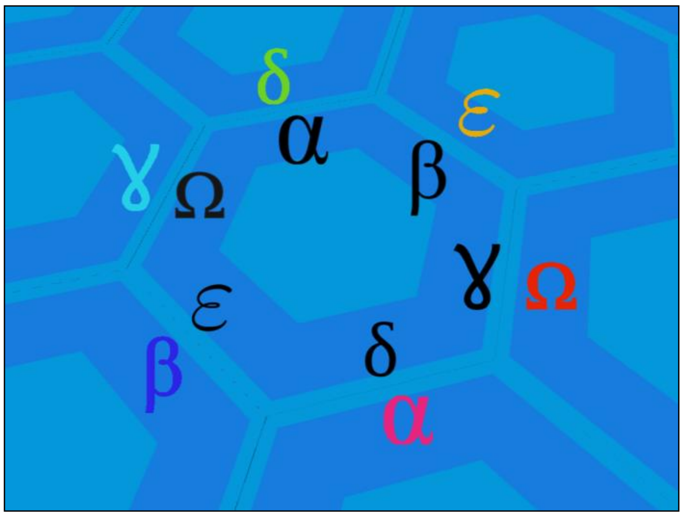
UNIT



As already said, the base unit is a hexagon. Each unit has a controlled gameObject and this is used to join the unit - that is just a class, invisible and untouchable - with the virtual world.

Linking an object to the unit class, we can locate a single unit, we can see the connections with the adjacent units and we can use the gameObject as a door to communicate with the class (with the classical Triggers, for example).

However, how do units connect to each other?



Each unit use a simple door system:

There are six doors (alpha, beta, gamma, delta, epsilon and omega, always in this order), at an entrance door, always correspond the same exit door.

For example if from one unit you move to another one through door ALPHA, you will get on the next unit from door DELTA (see at the image at the left to a clear explanation). If the intention is to go straight in one direction, you just have to get the same door, over and over.

If you are worrying about Greek letters even before starting using them, do not. You probably won't even notice them. This system is used in the lowest level of the structure, if you don't have the intention of modify the core of the structure, you won't get in contact with them.

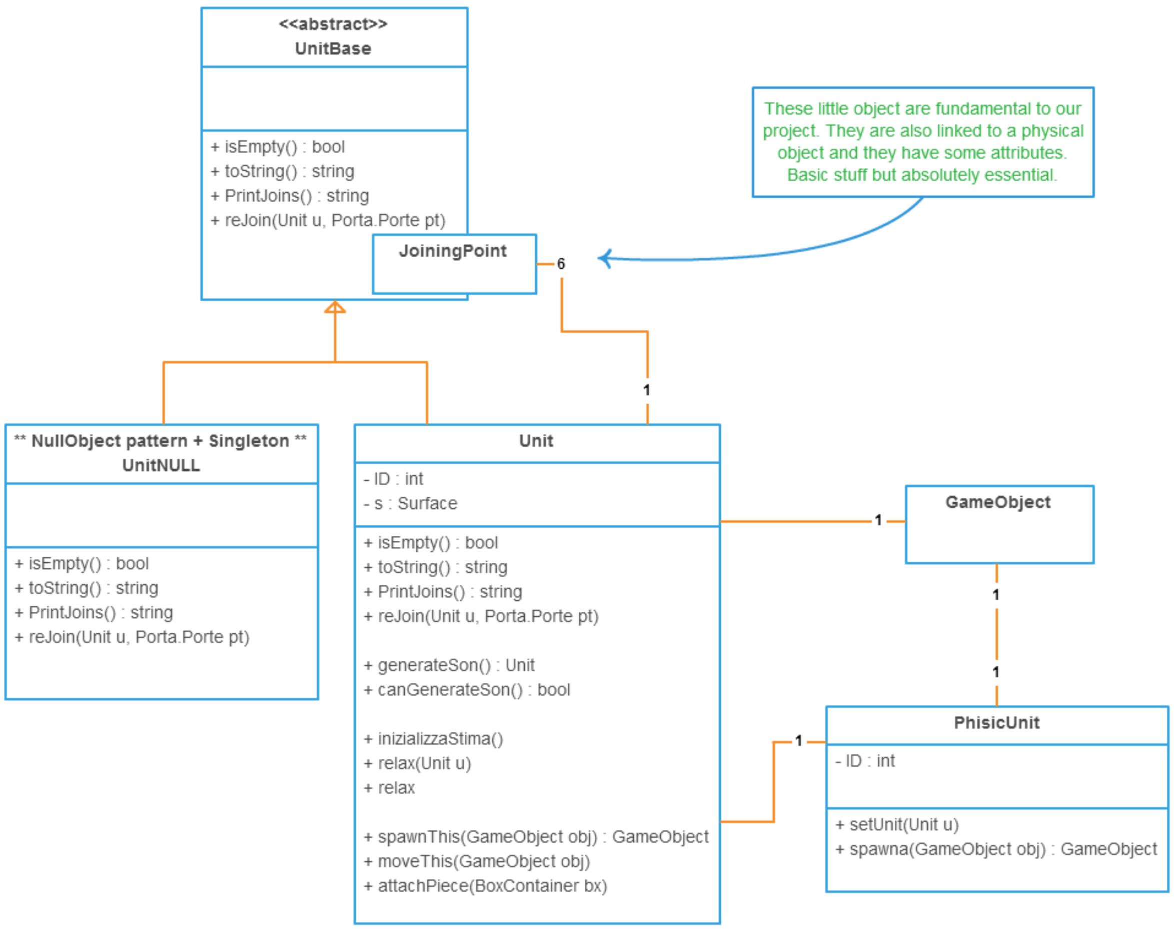
The Units-born system is made in a way to obtain a higher randomization as possible. In fact the Surface can't predict where the next Unit will be created, that choice is up to the Units.

The Surface just order to the first Unit of a determined list (*fertiliFigliazione : LinkedList<Unit>*) to create a new Unit and to return it. The Unit will check if this is possible (this process could fail for a few reasons), will spawn the Unit and then the Surface will check if it is able to give birth again, if not it will be removed from the list and the next time the Surface will ask to the next Unit.

Moreover, every new-born Unit will join with the adjacent, to ensure the graph connection.

In this way, we are able to create jagged or regular, expanded or small structures, just modifying a little the values in the options.

UML STRUCTURE


<<abstract>>
UnitBase
+ isEmpty() : bool
+ toString() : string
+ PrintJoins() : string
+ reJoin(Unit u, Porta.Porte pt)
JoiningPoint
6
These little object are fundamental to our project. They are also linked to a physical object and they have some attributes. Basic stuff but absolutely essential.
1
** NullObject pattern + Singleton **
UnitNULL
+ isEmpty() : bool
+ toString() : string
+ PrintJoins() : string
+ reJoin(Unit u, Porta.Porte pt)
Unit
- ID : int
- s : Surface
+ isEmpty() : bool
+ toString() : string
+ PrintJoins() : string
+ reJoin(Unit u, Porta.Porte pt)
+ generateSon() : Unit
+ canGenerateSon() : bool
+ inizializzaStima()
+ relax(Unit u)
+ relax
+ spawnThis(GameObject obj) : GameObject
+ moveThis(GameObject obj)
+ attachPiece(BoxContainer bx)
1
GameObject
1
1
1
PhisicUnit
- ID : int
+ setUnit(Unit u)
+ spawna(GameObject obj) : GameObject

In this UML is possible to see the coupling between the unit, its GameObject and the script attached to the GameObject.

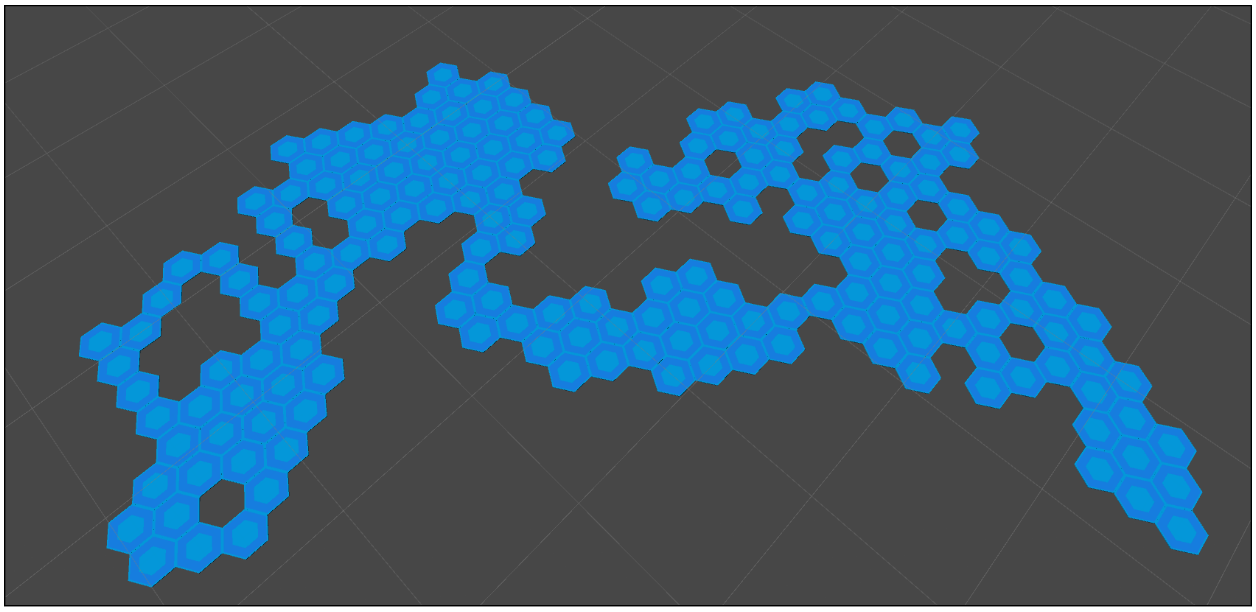
For each function/void there is a documentation in various languages that can be consulted for further information.

RELATED CLASSES

VirtualPosition -> class that represent the position of the units in the space and that can be converted in physical position, giving the size of the GameObject, to place the unit to his right place. It is represented just with two integers: X and Y.

This class is also used as key in some data structure in the Surface class, this ensure the uniqueness of the unit in the scene and to maximize the efficiency of some functions.

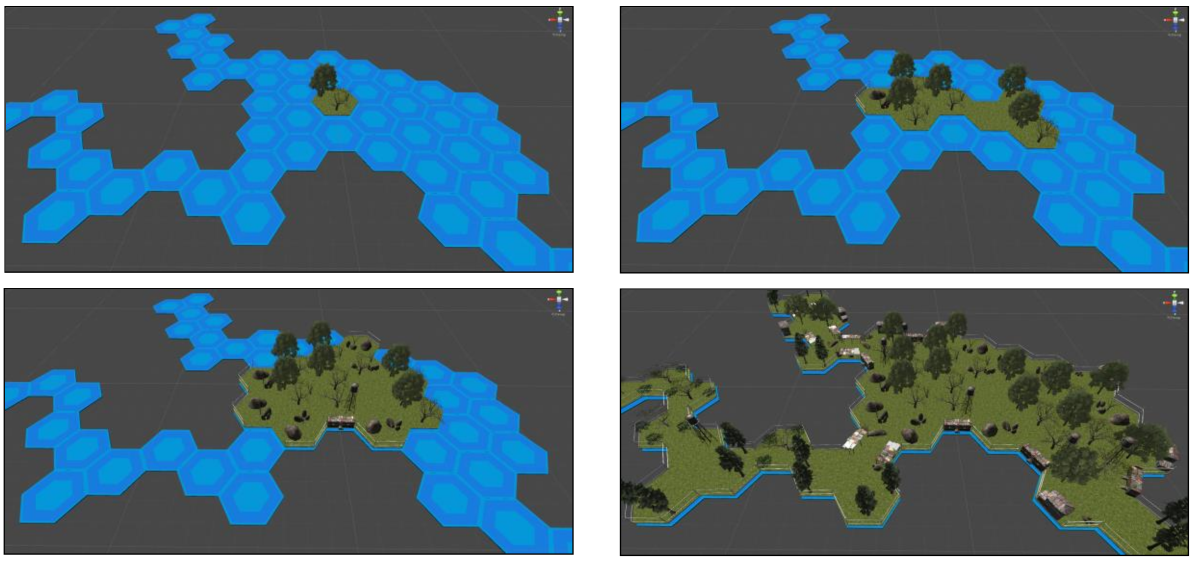
SURFACE



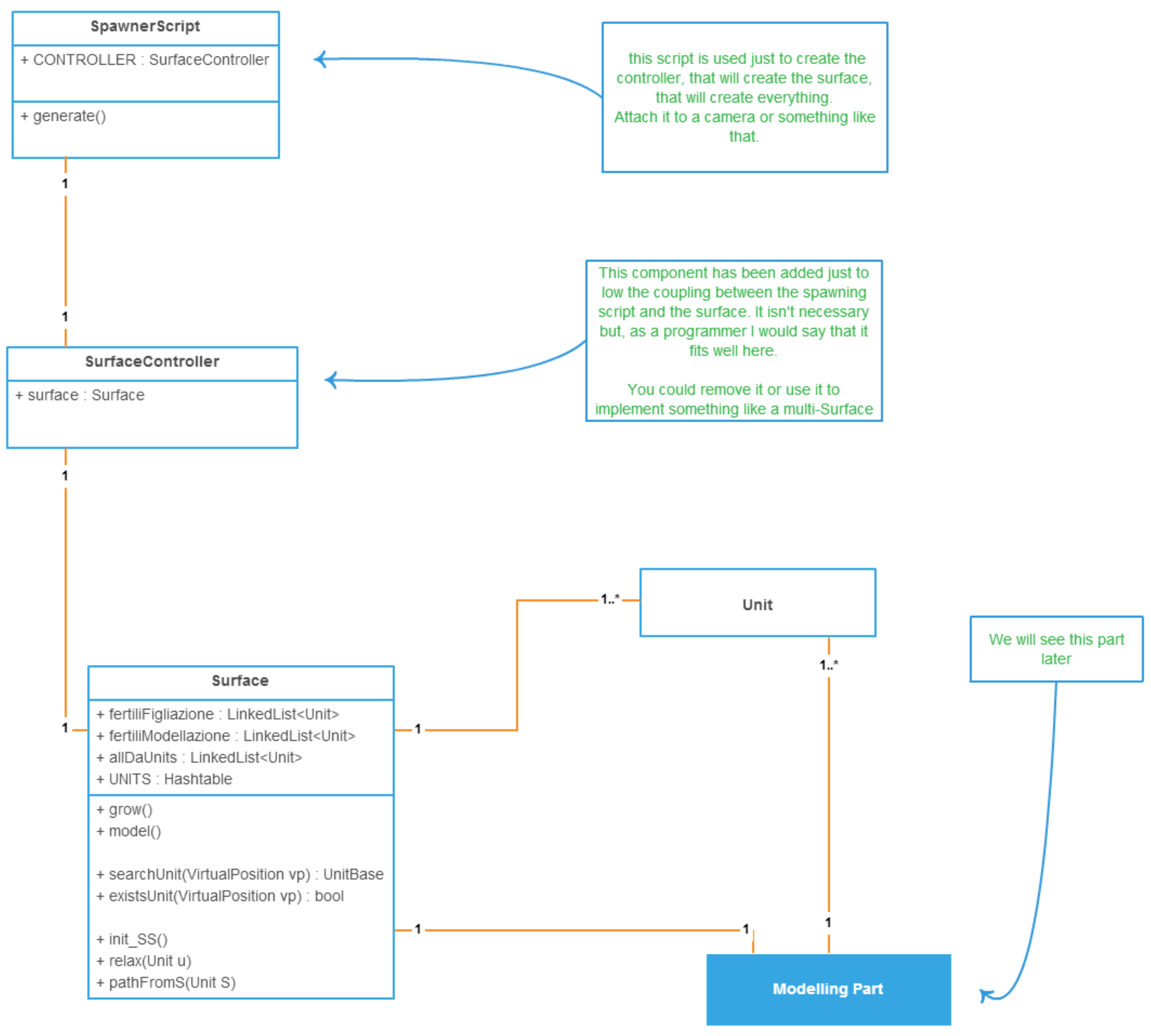
A Surface is just a class used to manage a large amount of units, through a few data structure. Its function is very easy:

1. Grow until the chosen limit is reached (or it is not possible to grow anymore)
2. Spawn the models until all units are covered
3. Do stuff if requested (path calculation, for example)

Between the born of a unit and the choose of a model there is a little delay to avoid errors. In any case, the order is the same because when a unit is spawned it is placed in a two different lists: the first one is necessary to generate other units and the second one keeps the units that don't have a model over them yet.



UML STRUCTURE


SpawnerScript
+ CONTROLLER : SurfaceController
+ generate()
this script is used just to create the controller, that will create the surface, that will create everything. Attach it to a camera or something like that.
1
1
SurfaceController
+ surface : Surface
This component has been added just to low the coupling between the spawning script and the surface. It isn't necessary but, as a programmer I would say that it fits well here.
You could remove it or use it to implement something like a multi-Surface
1
1
Unit
1..*
1..*
We will see this part later
Surface
+ fertiliFigliazione : LinkedList<Unit>
+ fertiliModellazione : LinkedList<Unit>
+ allDaUnits : LinkedList<Unit>
+ UNITS : Hashtable
+ grow()
+ model()
+ searchUnit(VirtualPosition vp) : UnitBase
+ existsUnit(VirtualPosition vp) : bool
+ init_SS()
+ relax(Unit u)
+ pathFromS(Unit S)
1
1
1
1
Modelling Part

The structure of Surface is as simple as possible to allow people personalize it.

Everyone has its own needs, so Surface, rather than satisfy a little part of the users, tries to please all of them, but giving the opportunity to be changed and modelled, to follow the idea of every single final user.

Surface is also really fast! The asymptotic computational complexity of the basic functions is very low and all the structure has been studied to be as fast as possible. Some examples?

- Unit search	->	Θ(1)	(constant)
- Path search	->	Θ(n)	(linear)
- New Unit	->	Θ(1)	(constant)

OTHER CLASSES

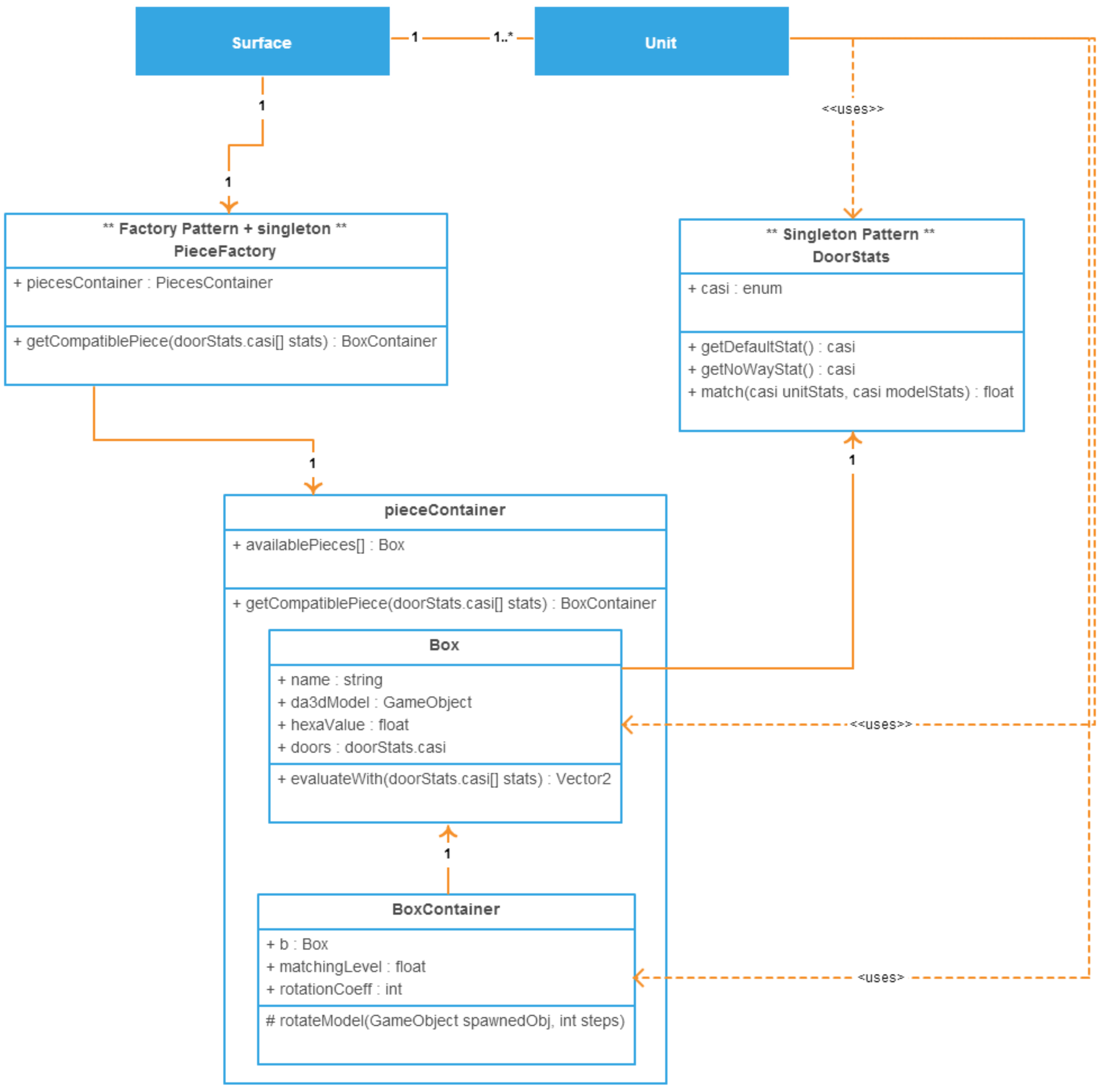
This part talks about the actual purpose of the structure: the creation of the environment.

Everything has been made in the simplest way possible for the final user: he or she has just to give the available models to the structure, set some things and press play!

Let's see how it works!

GENERAL STRUCTURE

This is the UML diagram of the remaining part. Initially it could appear messy but there is nothing to fear.


Surface
1
1..*
Unit
1
1
<<uses>>
** Factory Pattern + singleton **
PieceFactory
+ piecesContainer : PiecesContainer
+ getCompatiblePiece(doorStats.casi[] stats) : BoxContainer
** Singleton Pattern **
DoorStats
+ casi : enum
+ getDefaultStat() : casi
+ getNoWayStat() : casi
+ match(casi unitStats, casi modelStats) : float
1
1
pieceContainer
+ availablePieces[] : Box
+ getCompatiblePiece(doorStats.casi[] stats) : BoxContainer
Box
+ name : string
+ da3dModel : GameObject
+ hexaValue : float
+ doors : doorStats.casi
+ evaluateWith(doorStats.casi[] stats) : Vector2
<<uses>>
1
BoxContainer
+ b : Box
+ matchingLevel : float
+ rotationCoeff : int
# rotateModel(GameObject spawnedObj, int steps)
<uses>

PIECE FACTORY

Class that is called by the Surface class every time there is a new unit to model. It just redirect the calls to the PieceContainer class. Nothing special here, let us move over.

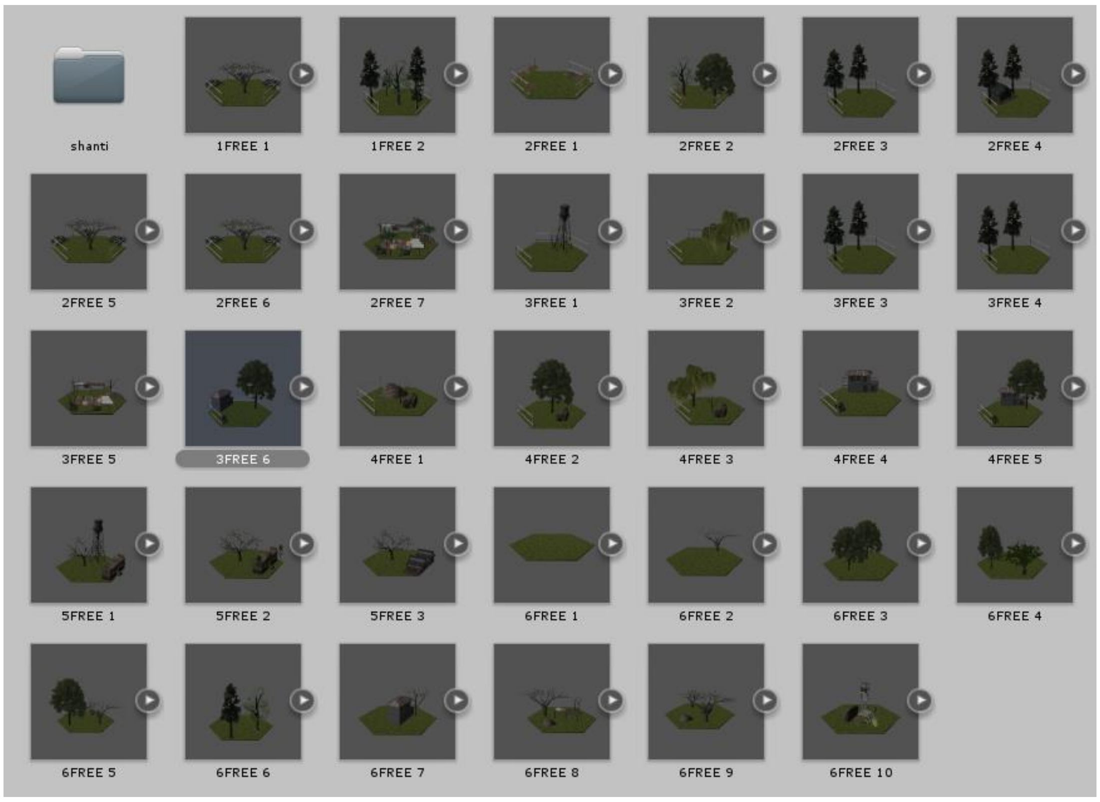
PIECE CONTAINER, BOX AND BOXCONTAINER

The PieceContainer script has to be attached to the prefab that will contain all the models that will be considered. You will find some exhaustive examples. Its duty is to match every single Unit in the Surface with the models it has stored.

Each model is saved inside a Box class and when a model is chosen, it is placed in a BoxContainer class and delivered to the Unit. Here the model will be spawned, but to do that the Unit needs some information, such as the rotation, that is why we use BoxContainers.

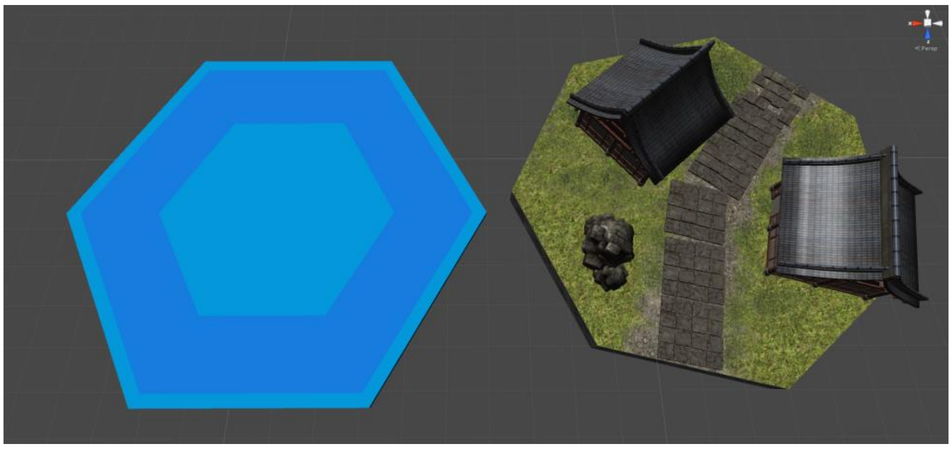
First we need some models! Some are in the project, just to show how everything works. You can create models with a 3d modelling program (such as Maya, 3dsMax or Blender) or you can use a simple hexagon and place some models over it in the Unity scene editor.

SPOILER ALERT --> how would you react if I told you that hexagons aren't really needed?

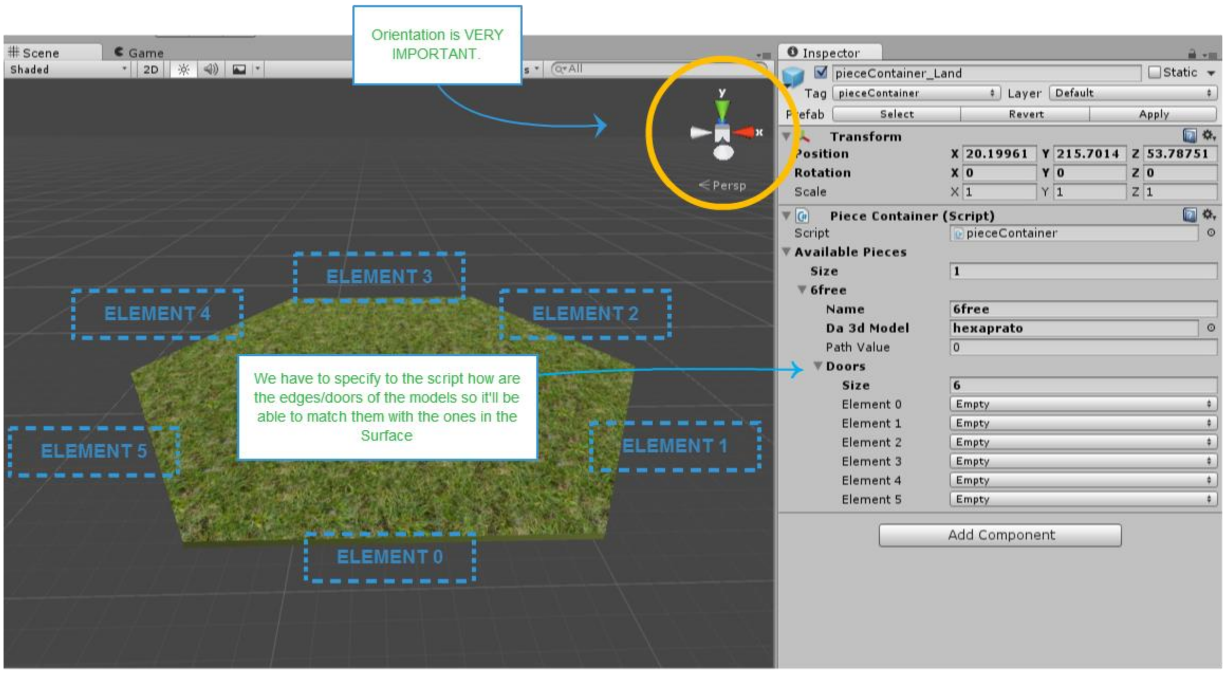


Here we have the models I used for the example before, ordered by the number of free edges. They are just a few and pretty basic: they can have free or busy edges, nothing else. Despite this, the final result is pretty awesome anyway.

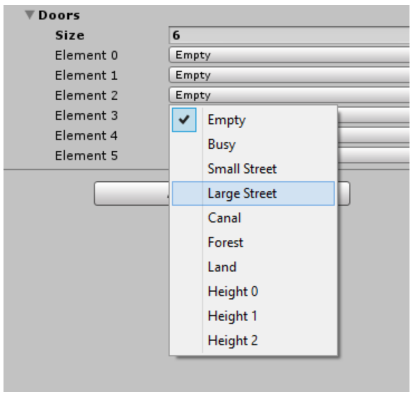
Models have to match the Unit GameObject! In short words you have to remember that we are working with hexagons. Heptagons or pentagons will not work here. Even if they are really sweet.



Now we have to add the models to the PieceContainer array in the inspector and "describe" them.


Orientation is VERY IMPORTANT.
We have to specify to the script how are the edges/doors of the models so it'll be able to match them with the ones in the Surface
ELEMENT 0
ELEMENT 1
ELEMENT 2
ELEMENT 3
ELEMENT 4
ELEMENT 5
Scene
Game
Shaded
2D
Persp
Inspector
Add Component

```
pieceContainer_Land                      Static
Tag  pieceContainer        Layer  Default
Prefab   Select   Revert   Apply
Transform
Position      X 20.19961   Y 215.7014   Z 53.78751
Rotation      X 0          Y 0          Z 0
Scale         X 1          Y 1          Z 1
Piece Container (Script)
Script              pieceContainer
Available Pieces
  Size              1
  6free
    Name            6free
    Da 3d Model     hexaprato
    Path Value      0
    Doors
      Size          6
      Element 0     Empty
      Element 1     Empty
      Element 2     Empty
      Element 3     Empty
      Element 4     Empty
      Element 5     Empty
```



```
Doors
  Size        6
  Element 0   Empty
  Element 1   Empty
  Element 2   Empty
  Element 3
  Element 4
  Element 5
```

What you should choose depends on the models, on how you want they interact, on the doors you chose.

In the next paragraph we will talk about the doors and how the surface interprets them.

DOOR STATS

This script is the kernel of our decision algorithm; in fact it has all the responsibility of the matches between the surface and the models.

Despite its importance, it is very simple to set and to use: you just have to choose the doors and to set the correlation between them.

	Unit-Side				
Model-Side		Free	Busy	LargeStr	SmallStr
	Free	10	0	0	0
	Busy	0	10	0	0
	LargeStr	0	0	10	0
	SmallStr	0	0	0	10

Let's take in consideration these four cases:
This is a simple 1 to 1 association.
In short words the algorithm will try to match the Units with the models that fits perfectly (and this is enough most of the times)

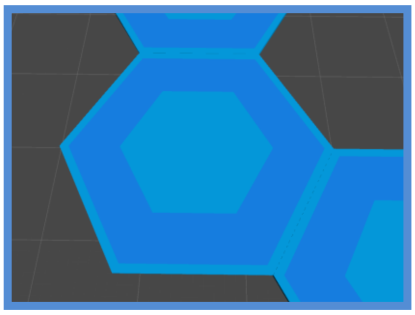
Now we have to traduce this in scripting language:

```
Public float match(casi unitStat, casi modelStat) {
        If (unitStat == modelStat) return 10;
        else return 0;
}
```

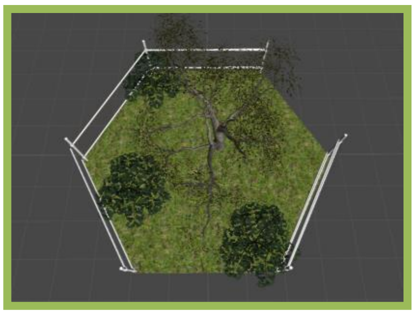
This algorithm is enough to get the most project to work fine.

But this function matches a single door in the Unit with another one in the model. How do we manage rotation? Because we know that without models rotation we should create a lot of them to cover all the possibilities that are going to be created due to the randomization.

We do that creating a matching matrix! For each model the algorithm creates a 6x6 matrix, containing all the combination. Let's see an example:



	α	β	γ	δ	ε	ω
α	0	10	0	10	0	0
β	10	0	10	0	10	10
γ	0	10	0	10	0	0
δ	10	0	10	0	10	10
ε	10	0	10	0	10	10
ω	10	0	10	0	10	10



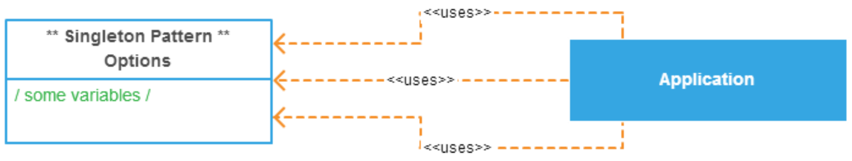
Even if the model's orientation and the unit's orientation are not the same the matrix will discover a perfect matching, just looking at the diagonals. In this way it will also find the rotation that we'll have to apply (light blue diagonal, so we need to rotate it 60° to the left).

The combination of doors BUSY + FREE permits us to create an environment with borders and nothing else; you will have to act on the models, creating various models for the same door set, for example. This give us the possibility to have a very different environment because if more than one piece reach the same (max) matching score, the model will be chosen randomly between them. You do not have to worry about *primary agglomeration* or problems like these because they have been solved; if you do not know what it is, you can live without knowing it.

Anyway if you add some other cases, the possibilities of what you can do increase exponentially. Look at the examples part for further details.

OPTIONS

Options class is needed to manage some variables, used in various places in the application.



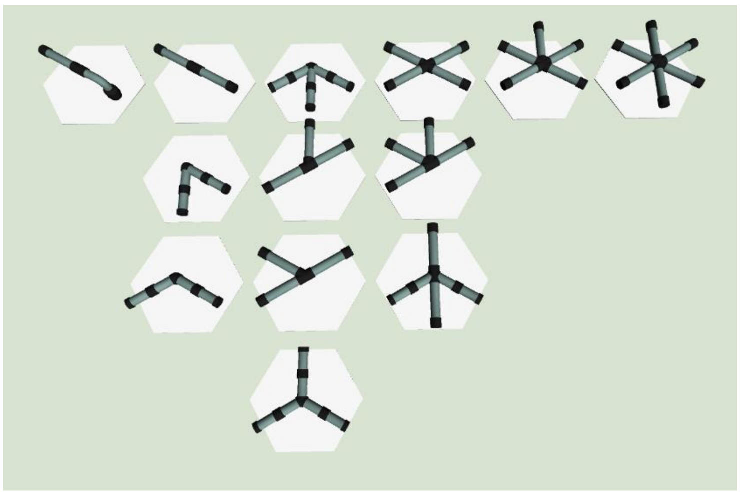
Usually options parameters are fixed before of launching the application and the Surface and the other components will use them to work, but you could take into consideration the idea of changing them run-time. Maybe could happen something cool.

Options variables are enough explained on the project.

EXAMPLES

Here I will show some examples, just to see how Surface works. You can find these examples and some other on the website and directly in the project. In this way you can interact and make experiments with them.

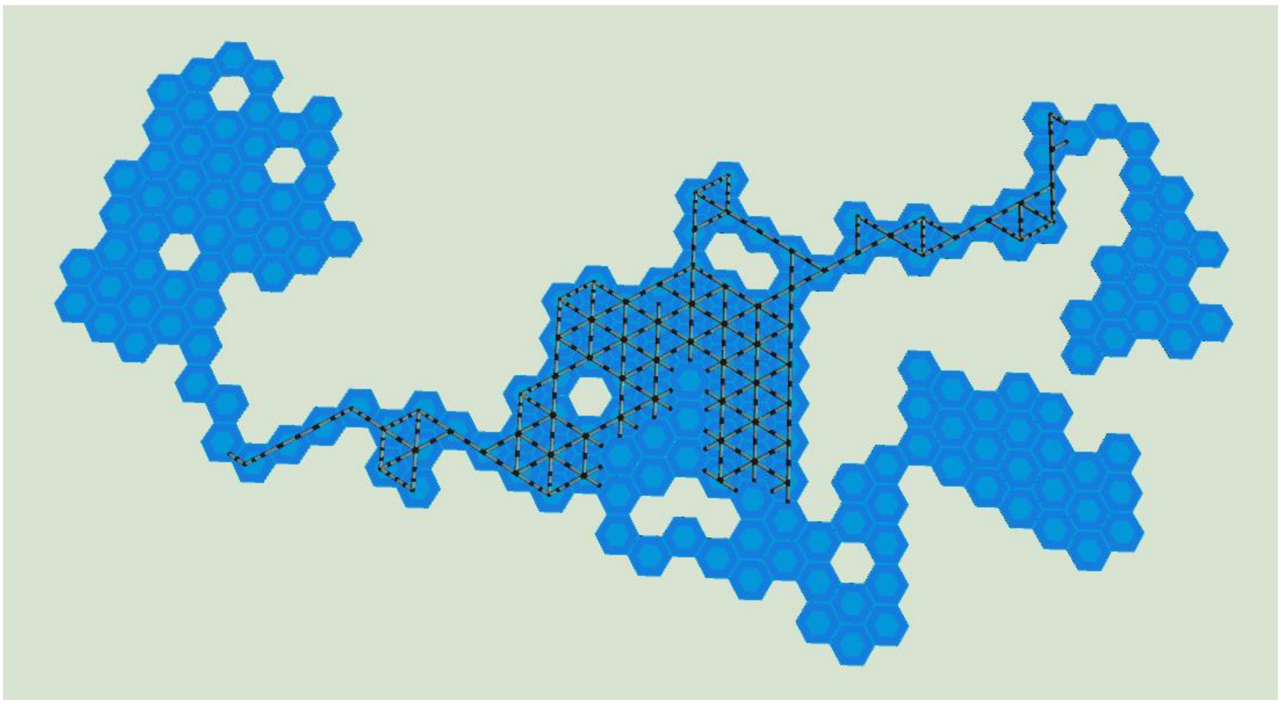
TUBES



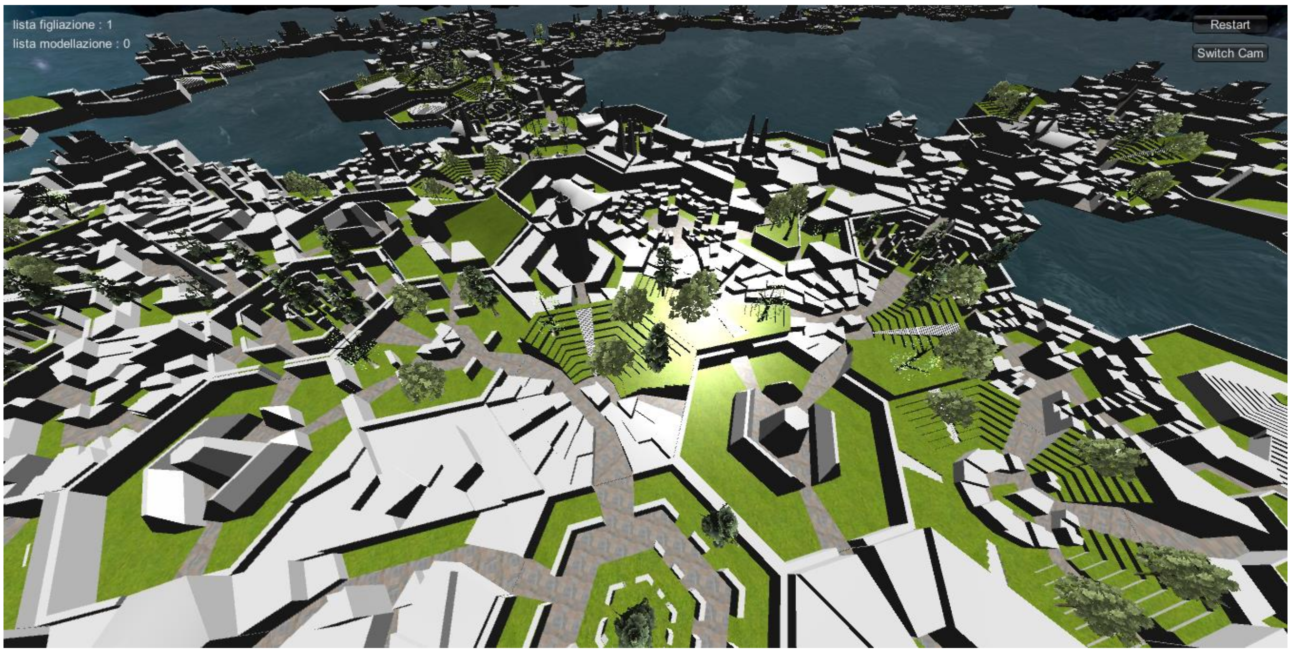
This is the most basic set of pieces you can think of.

These are all the possible combinations of pieces to fill the Surface completely. In fact with these 13 combinations of doors we can represent perfectly the BUSY-EMPTY doors.

The pieces just combine themselves to fill everything. The result, how you can see, looks like a thick web of tubes. There is no need to differentiate pieces or to use a strange door combinations: if you want to do something like that, you just have to let them spread and fill the entire Surface.



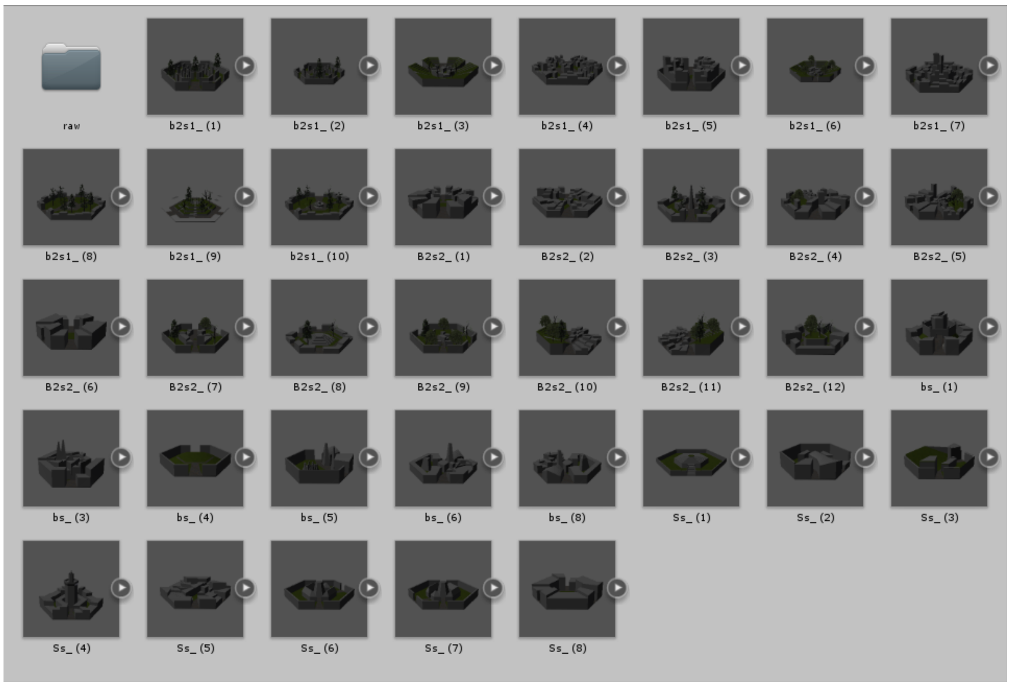
CITY



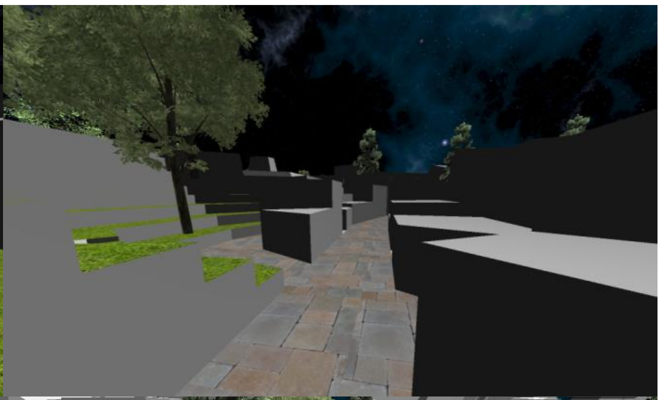
Here we have something a little bit complex. The number of models needed increases a lot because there are two different type of streets that link a piece to another one, so we used a combination like BUSY-SMALLSTREET-LARGESTREET.

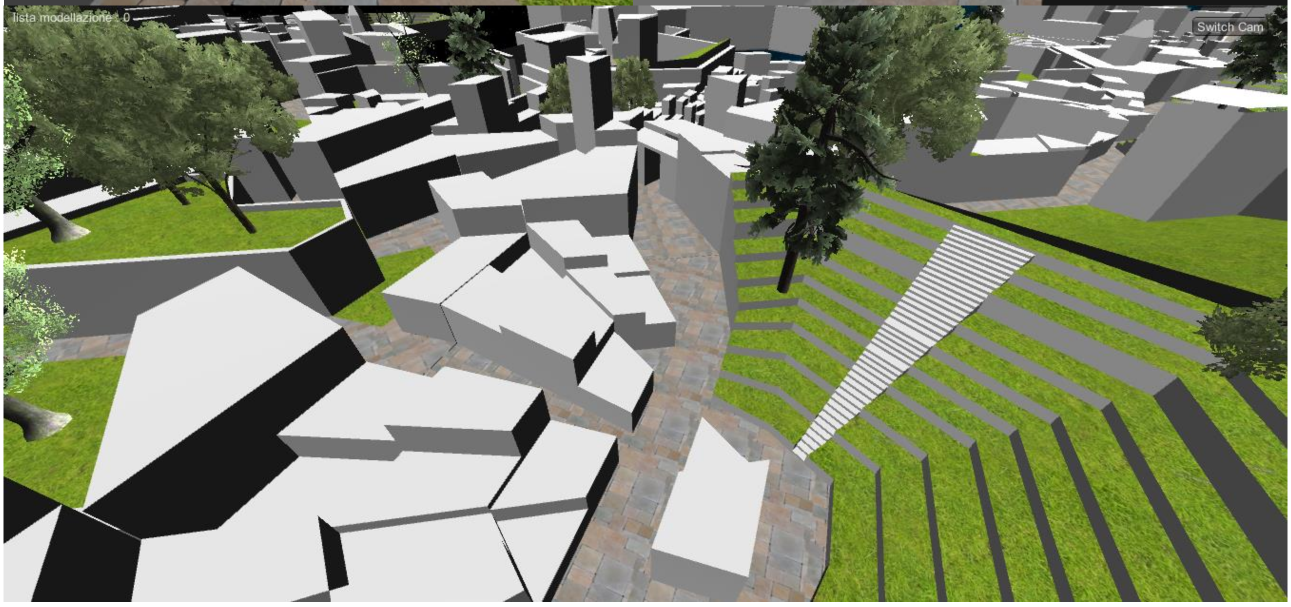
You cannot consider to create all the necessary pieces because the number would be too high (13 small street pieces, 13 large street, ~13*13 pieces that would be the combinations of the two) and the final result would be a dense link of streets, that is not realistic.

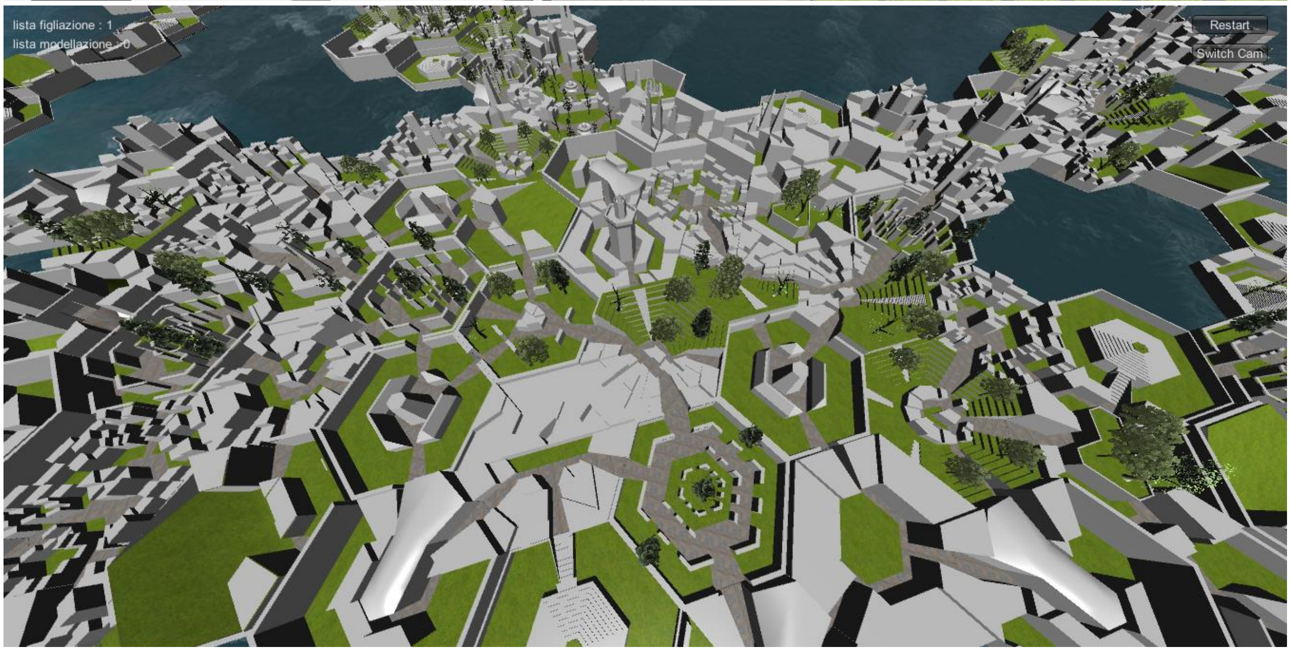
You have to mediate the number of pieces with the door combinations and eventually the algorithm, to obtain the desired result.



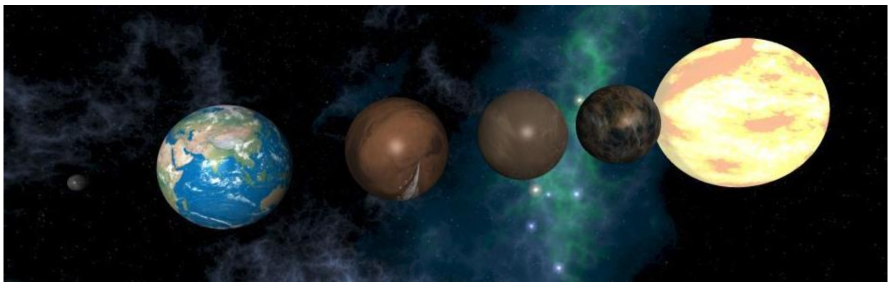
For example, there is no need to create models with six exiting streets: this model would be used everywhere in the middle of the structure, monopolizing and flattening it. For this reason, we would prefer lot of different pieces with less exiting streets but these could be chosen alternatively by the structure, permitting a nice and realistic environment as result.







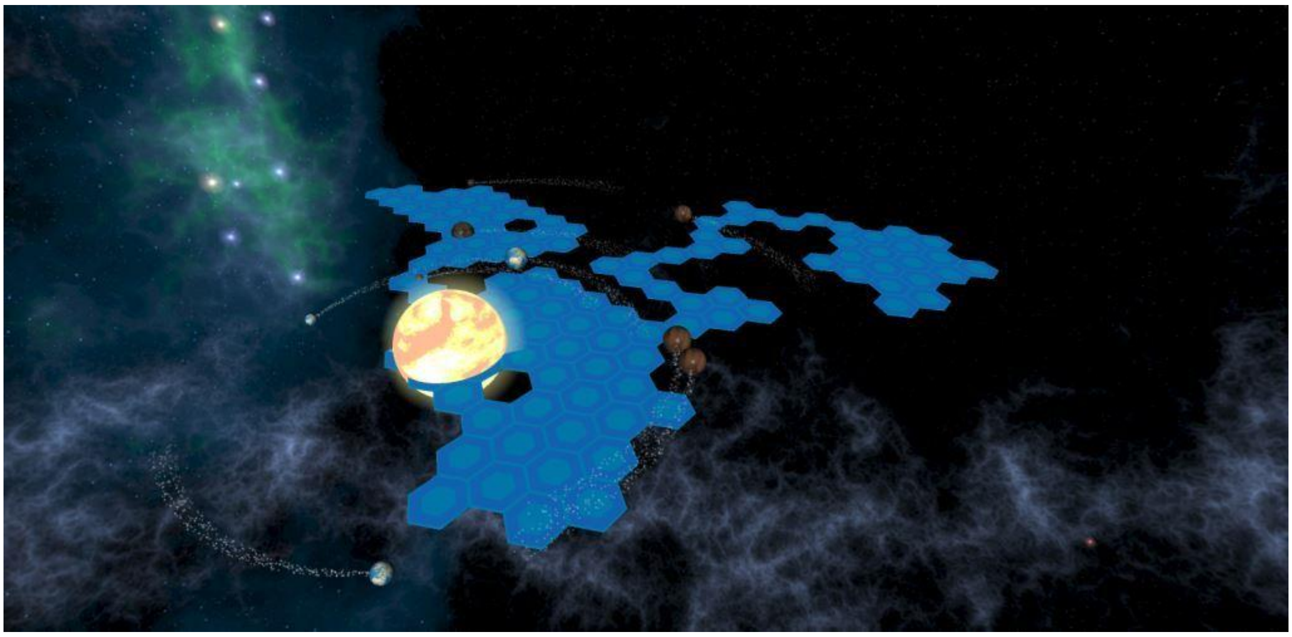
SPACE



This example has been made just to show that is possible to use not-hexagonal pieces.

Each astronomical object has a script attached to it. In this way planets will look for stars and will orbit around them, satellites will do the same. Via script is possible to manage collisions between planets or changes in the orbit, or any other law in the little universe you are creating.

In this case, Surface can be used to spawn objects randomly, after the universe has been created, like asteroids, spaceships, or whatever you want to put in the world.



FINAL CONSIDERATIONS

Surface has been created with the intention to be a universal and easy-to-programme environment generator.

Obviously it cannot be used for every kind of game but its versatility and malleability permits us to model it over the game we are going to create, granting us a wide number of uses. Hexagons are particularly useful because they can be adapted to quite everything and they can build a large network with some interesting implications.

Surface is a fast and responsive structure that we can continually interrogate, to get info over the environment or the objects that are interacting with it. Having a structure that manages quite totally the environments permits us to save a lot of time and to concentrate over other things.

Another quality of the structure is that it is totally randomized and the environment created will not be predictable. This is perfect for all that range of games that generates pieces in a random way or endlessly, but it can be used to create landscapes or entire worlds for games like GDR, shooting games, race games and a lot of others. Combining these qualities with the possibility of using the structure as active part of the environments (such as using Units as triggers to spawn things, or checkpoints randomly created, etc.), gives us infinite possibilities to deal with.