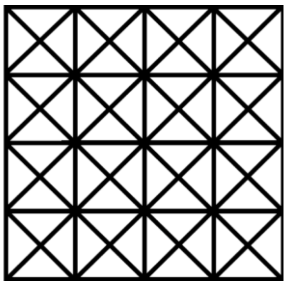
Procedural Grid Algorithm:

The grid consists of smaller planes which consists of 4 triangles. If we have a 4x4 grid ,we will have 16 planes and 64 triangles. To create this procedurally I wrote a nested for loop. The first loop defines the horizontal size of the grid and the second the vertical size.

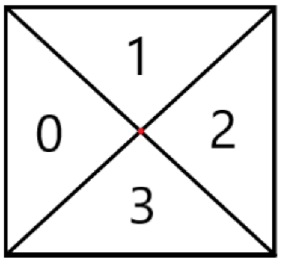
The loop will repeat as many times as the count of the planes grid. If it is a 4x4 grid it will loop 16 times. And every plane consists of 4 triangles.



The triangle meshes are being created by defining their vertices and also the triangles.

For example at index 0, we will start at the red point in the middle of the plane, which is the anchor point of the plane. The next vertices are added according to the clockwise direction. So the next point will be the bottom left and the last one the top left. The vectors would be in this case:

- Vector3 (0, 0, 0)
- Vector3 (- cellSize/2, - cellSize/2,0)
- Vector3 (- cellSize/2, cellSize/2,0)

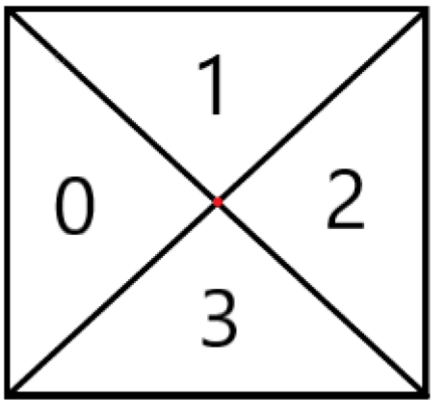


To find the triangles we need to define the indexes of the meshes in a clockwise direction so that their normal direction will be accurate. As the vertices are sorted according to the clockwise direction we only need to add the indexes of these vertices.

- Int [] { 0, 1, 2 }

While creating these triangles I didn’t share any vertices between the faces. This will make it easier whilst creating the pieces. This has the advantage that I can get add multiple faces together and create a mesh out of them without influencing anything.

I also created a list of anchor points which are the middle points of each plane. In this case red point in the image below. These anchor points will be useful during the implementation of placing the pieces inside the grid.



Pseudo Code

```
For vertical grid size
    For horizontal grid size
        Create Triangle 0
        Create Triangle 1
        Create Triangle 2
        Create Triangle 3
```

Procedural Piece Algorithm:

The algorithm for creating procedurally pieces has 6 parts;

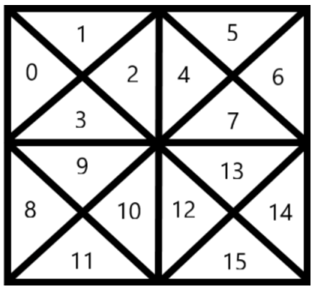
1. Creating Grid
2. Finding neighbors of all faces
3. Calculating random piece sizes
4. Creating pieces with BFS algorithm
5. Finding unused faces
6. Creating Meshes

1. **Creating Grid :**

I created a grid with the algorithm explained above.

2. **Finding neighbors of all faces.**

After I created a grid with the explained algorithm above, I created a method to find the neighbor faces of each face.



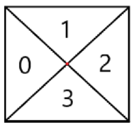
The methods find the neighbors of a given index. For example, if we take the face with the index 4. The neighbors would be 5, 7 and 2. In a more general definition;

- *Index + 1*
- *Index + 3*
- *Index – 2*

Using these definitions, I created a function. This function takes the modulo of 4 of the selected index. So there are only 4 states which are 0, 1, 2, and 3.

To continue the example 4 % 4 -> 0. So every index with 0 will have the neighbors

- *Index + 1*
- *Index + 3*
- *Index – 2*



But there are also edge cases which we need to pay attention to. For example if we try to find the neighbor of 0 we will get;

- *0 + 1 = 1 (True)*
- *0 + 3 = 3 (True)*
- *0 – 2 = -2 (False)*

We don't have a face at the index - 2 so we would get an error. Same think goes for index 8. According to the current formula we would get

- *8 + 1 = 9 (True)*
- *8 + 3 = 11 (True)*
- *8 – 2 = 6 (False)*

In this example index 8 is not a neighbor of index 6 so it should also not be added.

Regarding to the edge cases shown above we should not use the third rule (index – 2) every time. So we also need a rule for that. From the examples above we get the edge rules;

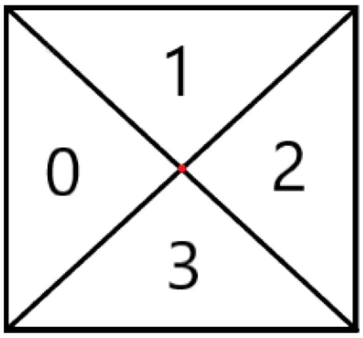
- *The index must be greater than zero*
- *Index % 8 should not be equal to zero;*

The first of these rules is general so we can use it like that. But the second will change according to the grid size. So we need to write it in a more general way which is:

- *Index % (Grid Size * 4) != 0*

So the complete rule will look something like this

```
If(index % 4 == 0)
   Neighbors.Add ( index + 1 )
     Neighbors.Add ( index + 3 )
            If( index > 0 && Index % ( GridSize * 4 ) != 0 )
                        Neighbors.Add ( index - 2 )
```



In this same way I created the rules for the other state which are;

- *index % 4 == 1*
- *index % 4 == 2*
- *index % 4 == 3*

As this part is formula based, I wrote a couple of unit tests to make sure that I wrote every formula correctly.

3. Calculating random piece sizes

To calculate the random sizes, I firstly found the amount of triangular faces in the grid. This amount can easily be found with the given multiplication.

*GridSize * GridSize * 4*

If we take a 4x4 grid for example, we will have 64 faces. These faces need to be split randomly to the number of pieces the level will have. The piece amount can go from 5 – 12. They are chosen according to the difficulty. The steps of the algorithm are:

a. Find the face amount. (which is 64 in this example)
b. Subtract the *amount of pieces* from the *faces*. This makes sure that every piece will have at least one face at the end of the algorithm
c. Get a random number between 0 – (faces / 3) and set it to the size of the piece.
d. Decrease the random number (from step c) from the faces and add 1 to it.(We add 1 because we subtracted it at the beginning so that it will have 1 face at minimum)
e. Return to step c with the next piece
f. The last piece gets the remaining faces.

4. Creating pieces with BFS algorithm

I used a slightly modified BFS algorithm to create the pieces. Besides the visited and queue lists, I used a used list. This list will hold the information if a face is already being used whilst creating a piece. The visited lists will hold the faces which are being added to the queue. There is also an createdPieces List which will hold the information of every created piece.

As we will apply the BFS algorithm multiple times the visited list is not enough to keep track of the used faces. The visited will be reset after each BFS. The used list in the other hand won't be reset.

In the 3. Step we found the sizes of the pieces which will be created. I called for ever item in the list the BFS algorithm. The BFS function has two parameter the first one defines the index of the face and the second one the amount of faces it will contain.

- *The first parameter which is index will be returned from a function which returns an unused face index.*
- *The second parameter comes from the sizes list*

The BFS function will add the index from its parameters to the queue. Then it will loop through until the pieceSize is equal to 0 and add the neighbors to the queue. If for some reason the pieceSize is still greater then 0 but there aren't any more faces in the queue, it will break from the loop, After the loop the function will add the faces which it got from the BFS to a list create Pieces List.

The simplified Pseudo code of the BFS algorithm is given below:

```
BFS(int index, int pieceSize)

    Add index to visited list
    Add index to queue list

    While(pieceSize > 0)

            If there aren't any face in queue return;

              Get the face[index]
              Find Neighbors of this face

    Add the created piece information to createdPiecesList
```

5. Finding unused faces

This step is to make sure that every face in the grid is assigned to a piece. In this step I simply go through the used list to check if all faces are being used.

For each unused face I loop trough the created pieces (This result of step 4) and check in their neighbor list if the unused face is a neighbor. (The created pieces contains a list of all of his remaining neighbors from the BFS algorithm)

If it is a neighbor apply the another BFS method to add the face to the piece and see if the new neighbors of the piece can reach another unused node.

6. Creating Meshes

Meshes are being created according to the results of the previous steps.