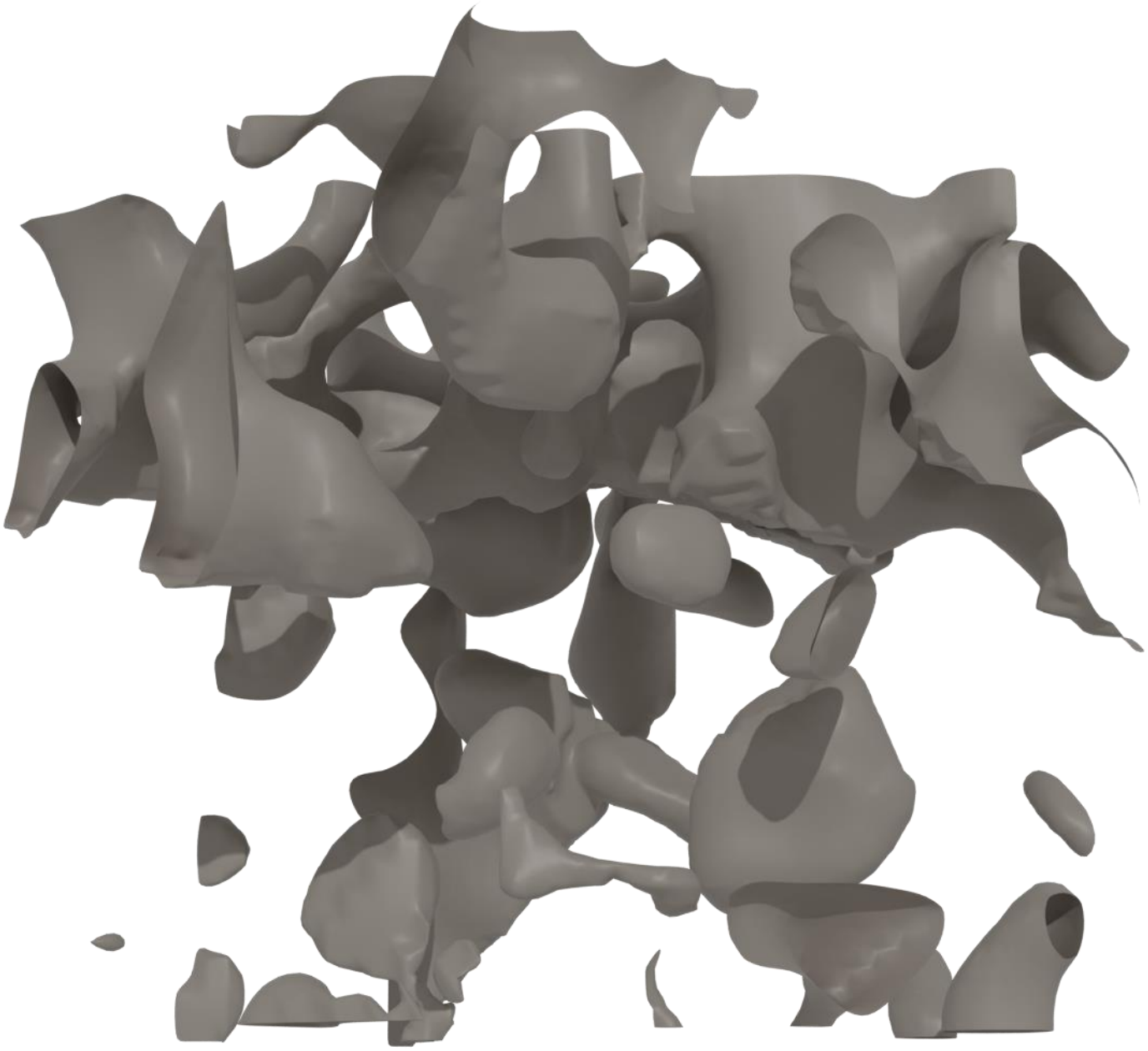
Scala Infinite Stream



A scala implementation of the Marching Cube Algorithm

3D Mesh generated by our implementation, rendered in blender.

The Algorithm

The marching cube algorithm created in 1987 by Lorensen and Cline, is a computer graphics technique used to construct 3D surfaces from volumetric data.
It divides a 3D space into cubes and determines the surface intersections within these cubes.
this algorithm is used in medical imaging to create 3d models from scan such as MRI or even in video game for dynamic terrain.

We decided to use this algorithm for this project because the volumetric data (also referred as scalar field) can get very huge which make it very complex to handle all at once.
Our “infinite stream” is the volumetric data, and we process it bit by bit.

How to use

First you will need to run the “FileStream.scala”.

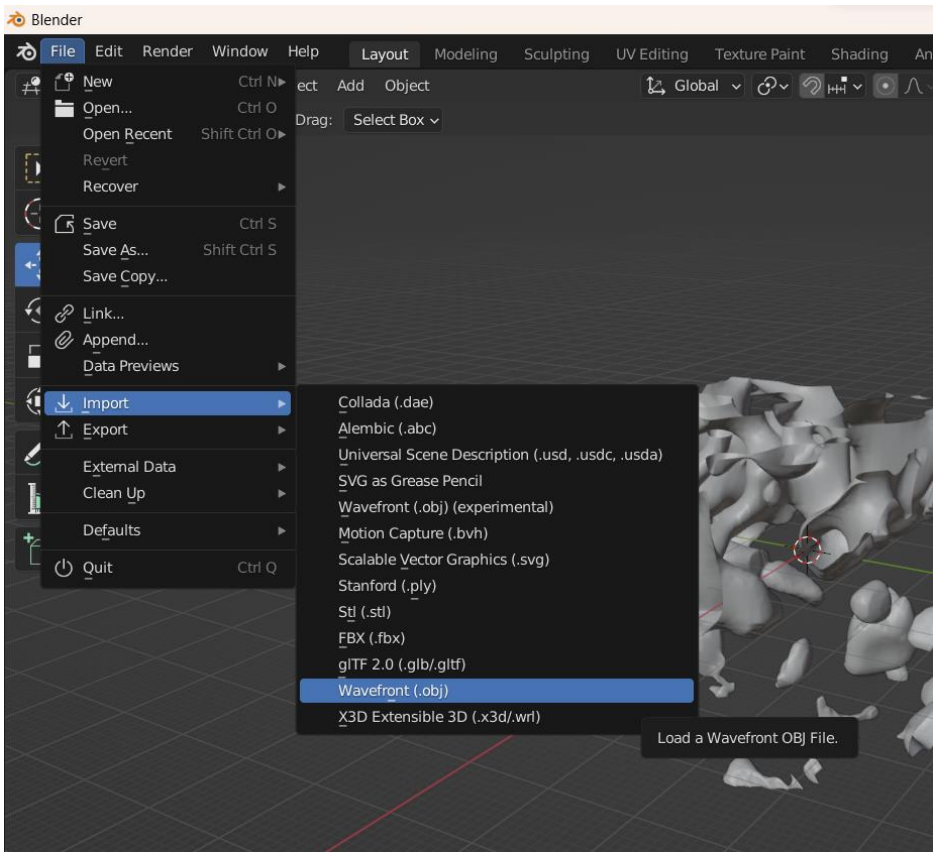
Then you will be prompted to give the name of the scalar field you wish to use two are provided in the resource folder.


```
Please select the name of your file in the 'resources' folder: ScalarField.csv
```

After that, the result mesh will be saved in the obj file format as output.obj in the root of the project.

You can open this obj file in any 3D software of your choice. (blender is a free and easy option)

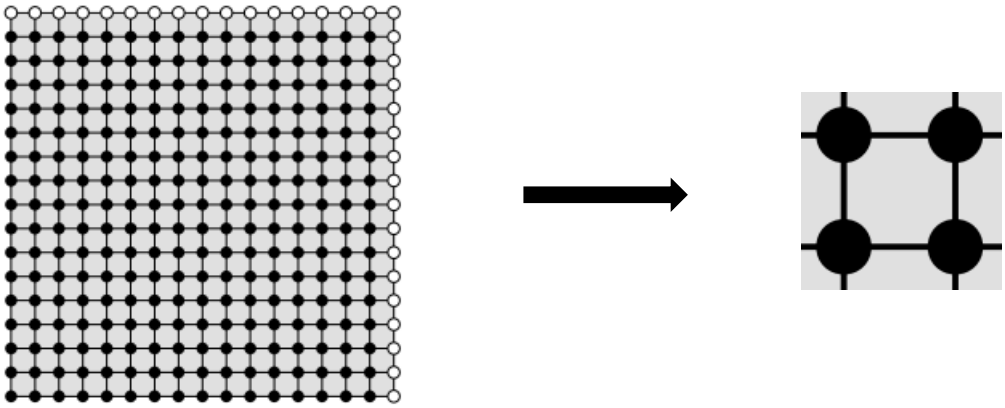
Opening result in blender



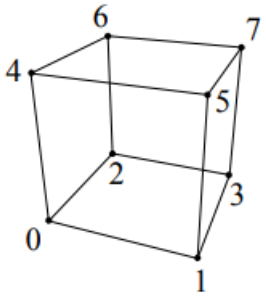
Once loaded you might need to zoom out a bit to see the mesh.

How it works

like we said before the marching cube algorithm works by going trough the scalar field cube by cube. In 2D it would be like transforming an image into a grid and then looking at it square by square.



In 3D or cube would look something like this.



Each of its vertices would have a density value (we chose a value between -1.0 and 1.0) and using this we can determine the shape within the current cube.

First, we got to figure out which of the vertices are inside and outside and store that in a meaningful way.

```
def getCubeConfiguration(cube: Cube): Int =
  cube.vertices.zipWithIndex.foldLeft(0)((config, vi) => config | (vertexState(vi._1) << vi._2))
```

This function gives us an integer for each possible configuration! This will be useful later with a look up table.

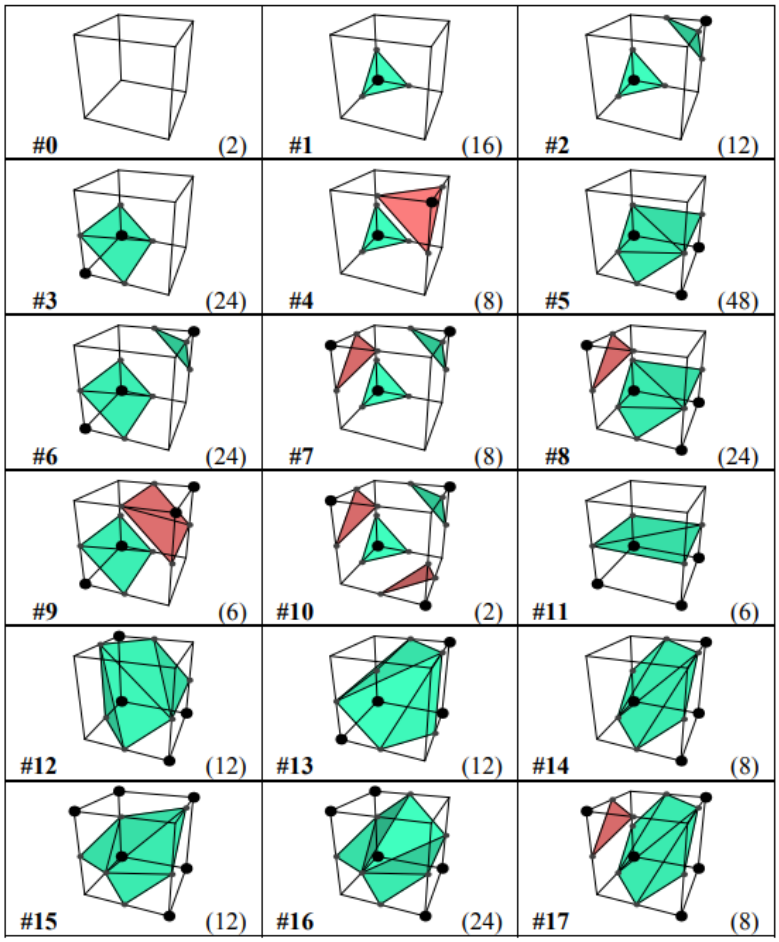
the first thing we do is check if the cube is completely empty or filled, if it is the case no geometry is needed.

```
def processCube(cube: Cube): Array[Triangle] = {
  val configuration = getCubeConfiguration(cube)
  configuration match {
    case 0 => Array.empty[Triangle]
    case 255 => Array.empty[Triangle]
    case _ => processStandardCase(cube, configuration)
  }
}
```

we then look at our look up table.

```
val sortingTable: Array[Array[Int]] = Array(
        Array(-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1),
        Array(0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1),
        Array(0, 1, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1),
        Array(1, 8, 3, 9, 8, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1),
        Array(1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1),
        Array(0, 8, 3, 1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1),
        Array(9, 2, 10, 0, 2, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1),
        Array(2, 8, 3, 2, 10, 8, 10, 9, 8, -1, -1, -1, -1, -1, -1, -1),
```

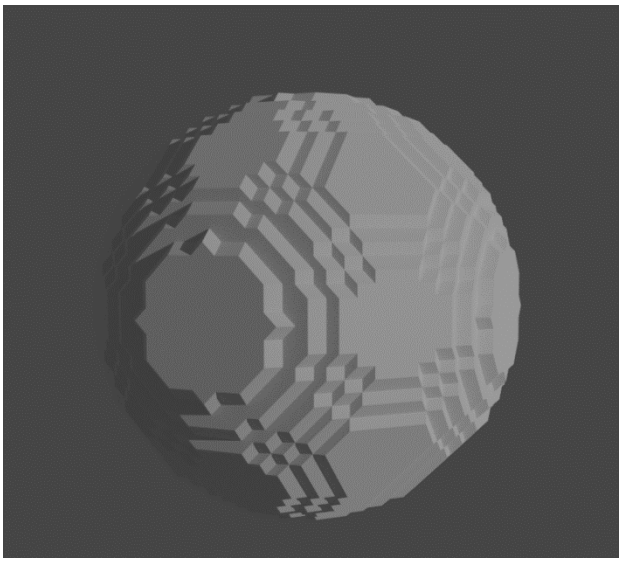
The look up table contains 256 cases. here are some of them.



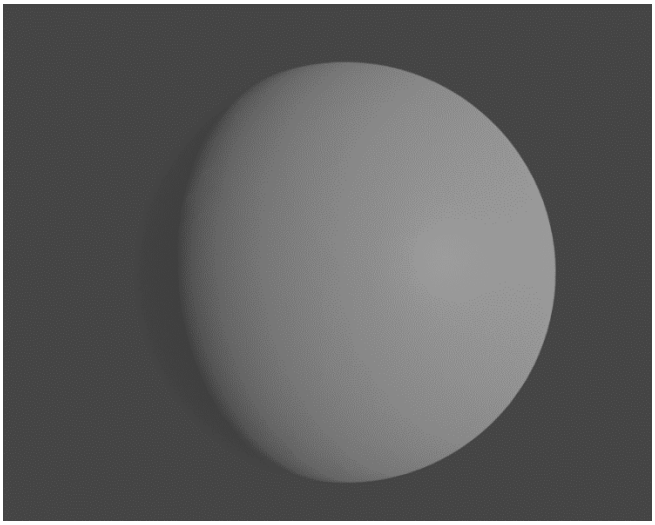
In the look up table the indices go from 0 to 11 and don't represent vertices but the edges, another look up table is needed after

```
val edgePairs: Array[(Int, Int)] = Array(
    (0, 1), (1, 2), (2, 3), (3, 0),
    (4, 5), (5, 6), (6, 7), (7, 4),
    (0, 4), (1, 5), (2, 6), (3, 7)
  )
```

Now you might be wondering why we need to also know the vertices, because we could simply take the middle of each edge and create our triangle this way like so:



The problem we face is that by always using the midpoint of the edge we get a blocky result, using the density value of both vertices to interpolate give us way better result for the same amount of vertices!



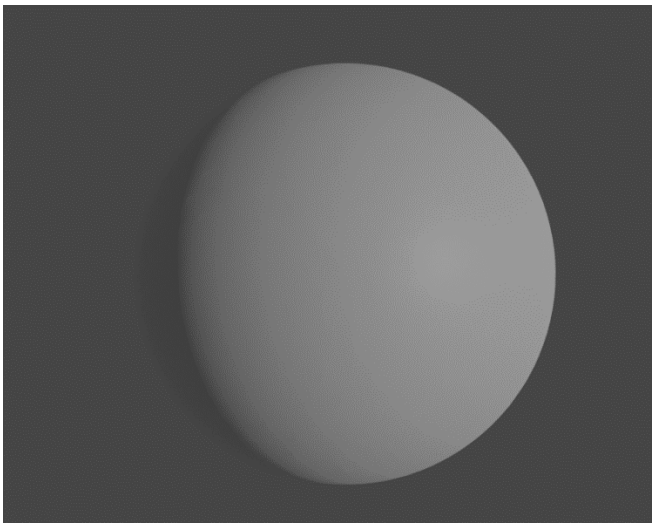
Finally, we transform our array of triangle into an obj using a recursive function.

Results

In the end we go from a 20 Mo csv file to a 2 Mo Obj file which is a 10 X size reduction!

Although it is good to note that the way we saved our obj is not optimal and could be divided by 4-5 times and that the obj file after being read by a 3D software will be saved in a way more efficient way.

Here are some result we got:



Funky stuff

Some bugs gave birth to rather interesting shape (all of them are supposed to be spheres)