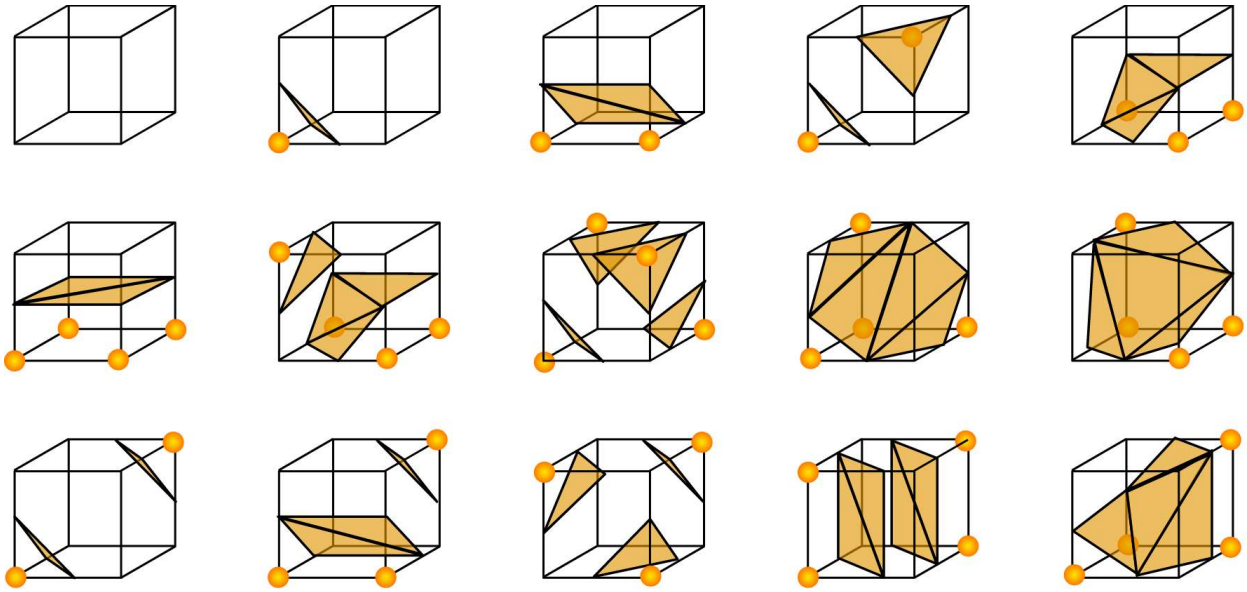
Marching Cubes Algorithm

The algorithm consists of looping through 8 points in space, each representing a value to be compared to an “iso-value”, if the value is greater than or less than the iso value then that point in space is said to be “activated” depending on the activation states of these 8 points the program then chooses a configuration from a list of possible configurations.



A list of possible configurations as seen above. The orange dots seen on the vertices of the cube are representative of “activated” points.

Generating the iso values of vertices

The generation of values or iso value for vertices is an important step in algorithm, one possibility is total randomness this would work and generate different types of polygons for cubes however there would be no level concurrence between points since neighbouring cubes have no knowledge of neighbouring/shared vertice iso values generating a larger set of cubes may result in no connections between neighbouring vertices. The solution to this is to use the position as a way to generate a random value, more specifically Perlin Noise allows for 2 values to be accepted and generate a specific noise value consistently with those values, with a quick conversion of placing the different dimension combinations we can use this to create 3-dimensional Perlin noise to use for more communicative randomization.

```
public static float PerlinNoiseValueFromPosition(Vector3 position, Vector3 area, float cubeRadius,  float filterLevel, float amplitude)
{
    float ab = Mathf.PerlinNoise(position.x + 0.01f, position.y + 0.01f);
    float bc = Mathf.PerlinNoise(position.y + 0.01f, position.z + 0.01f);
    float ac = Mathf.PerlinNoise(position.x + 0.01f, position.z + 0.01f);

    float ba = Mathf.PerlinNoise(position.y + 0.01f, position.x + 0.01f);
    float cb = Mathf.PerlinNoise(position.z + 0.01f, position.y + 0.01f);
    float ca = Mathf.PerlinNoise(position.z + 0.01f, position.x + 0.01f);

    float abc = ab + bc + ac + ba + cb + ca;

    abc *= amplitude;
    if (position.x <= -cubeRadius || position.x >= area.x - cubeRadius)  abc = filterLevel;
    if (position.y <= -cubeRadius || position.y >= area.y - cubeRadius) abc = filterLevel;
    if (position.z <= -cubeRadius || position.z >= area.z - cubeRadius) abc = filterLevel;

    return abc;
}
```

With values for vertices, the program can now compare this generated iso-value with the iso-level to see if the vertice is “activated” (or said within papers to be within the iso surface).

Mapping / Lookup table

This list of configurations however is much larger than the 15 configurations above since there are duplicates (mirrored versions) of several of them, this causes a massive list of possible polygon configurations (2^8=256 to be exact) to allow for quick mapping/lookup of these configurations. By using the earlier method of checking the point is activated, we can use activated points to represent a 1 in an 8-bit integer while unactivated points represent a 0. Using this 8-bit integer we can use it to find the appropriate index in the array of configurations.

Example:

If given points 7,6,5,4,3,2,1,0 (points in descending order) the algorithm finds that points 7, 5, and 1 are found to be “activated” then the evaluated 8-bit integer representation would be 1,0,1,0,0,0,1,0.

Points:	7,6,5,4,3,2,1,0
Activate Points:	7, 5, 1
8-bit integer representation:	1,0,1,0,0,0,1,0
Integer representation:	10100010 = 162

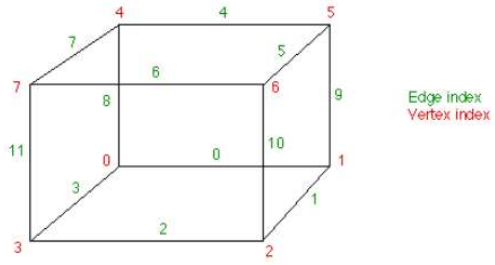
Looking at the array of possible configurations and using index 162 we are given the proper triangulation array for creating the proper polygon representation.

```
159: {10, 9, 4, 6, 10, 4}
160: {4, 9, 5, 7, 6, 11}
161: {0, 8, 3, 4, 9, 5, 11, 7, 6}
162: {5, 0, 1, 5, 4, 0, 7, 6, 11}
163: {11, 7, 6, 8, 3, 4, 3, 5, 4, 3, 1, 5}
164: {9, 5, 4, 10, 1, 2, 7, 6, 11}
165: {6, 11, 7, 1, 2, 10, 0, 8, 3, 4, 9, 5}
166: {7, 6, 11, 5, 4, 10, 4, 2, 10, 4, 0, 2}
167: {3, 4, 8, 3, 5, 4, 3, 2, 5, 10, 5, 2, 11, 7, 6}
168: {7, 2, 3, 7, 6, 2, 5, 4, 9}
169: {9, 5, 4, 0, 8, 6, 0, 6, 2, 6, 8, 7}
170: {3, 6, 2, 3, 7, 6, 1, 5, 0, 5, 4, 0}
171: {6, 2, 8, 6, 8, 7, 2, 1, 8, 4, 8, 5, 1, 5, 8}
172: {9, 5, 4, 10, 1, 6, 1, 7, 6, 1, 3, 7}
173: {1, 6, 10, 1, 7, 6, 1, 0, 7, 8, 7, 0, 9, 5, 4}
174: {4, 0, 10, 4, 10, 5, 0, 3, 10, 6, 10, 7, 3, 7, 10}
```

(source: https://www.youtube.com/watch?v=M3iI2l0ltbE, Sebastian Lague, “Coding Adventure: Marching Cubes”)

Triangulation and Polygon Representation

Once the triangulation of the polygon has been retrieved the program requires an array of 12 edge points, these points are located initially (Optional Interpolation can change this later on) in the middle of each line of the cube.



(Source: http://paulbourke.net/geometry/polygonise/, Paul Bourke, Polygonising a scalar field)

Using the initial vertices the program can find and generate the 12 edge points, through the triangulation array found from the table lookup the cube can be drawn in the space.

Normal Calculations

Although the calculation of normals isn't directly correlated to the Marching Cubes algorithm, it's an important part to allow for visualization of the program/polygons. The calculation for finding the normal vector of each triangle consists of first looping through the points in each triangle. Since we have the triangulation order that will be connecting the edges we can use this to determine which points will be connected/creating a triangle, using these points we can calculate the normal vector of that triangle by using the displacement vector of points 0 to 1 and points 0 to 2 and using those 2 vectors to find the cross product to use as the normal direction.

```
//Gets normal of traingle face given 3 indexes of the points in the edge array.
1 reference
public static Vector3 SurfaceNormalFromIndices(int indexA, int indexB, int indexC, Vector3[] edges)
{
    Vector3 pointA = edges[indexA];
    Vector3 pointB = edges[indexB];
    Vector3 pointC = edges[indexC];

    Vector3 sideAB = pointB - pointA;
    Vector3 sideAC = pointC - pointA;

    return Vector3.Cross(sideAB, sideAC).normalized;
}
```

We can then increment the index of the loop by 3 to move to the next set of points within the array of edges.

```
//Goes through each triangle and calculates the normals.
1 reference
public static Vector3[] GetNormals(int[] triangleOrder, Vector3[] edges)
{
    //Calculation normals
    Vector3[] normals = new Vector3[edges.Length];
    int triangleCount = triangleOrder.Length / 3;

    for (int i = 0; i < triangleCount; i++)
    {
        //Increment by 3 for next triangle set.
        int normalTirnagleIndex = i * 3;

        int pointA = triangleOrder[normalTirnagleIndex];
        int pointB = triangleOrder[normalTirnagleIndex + 1];
        int pointC = triangleOrder[normalTirnagleIndex + 2];

        Vector3 triangleNormal = SurfaceNormalFromIndices(pointA, pointB, pointC, edges);

        normals[pointA] += triangleNormal;
        normals[pointB] += triangleNormal;
        normals[pointC] += triangleNormal;
    }

    return normals;
```

Interpolation

One optional feature of the algorithm is to use approximations to find where an edge may lie dependent on the vertex value and how close or far from the iso-level it is. If the value was significantly over the iso-level the midpoint may be further in the direction of the high-value vertice. This can give the generated polygon a smoother look.

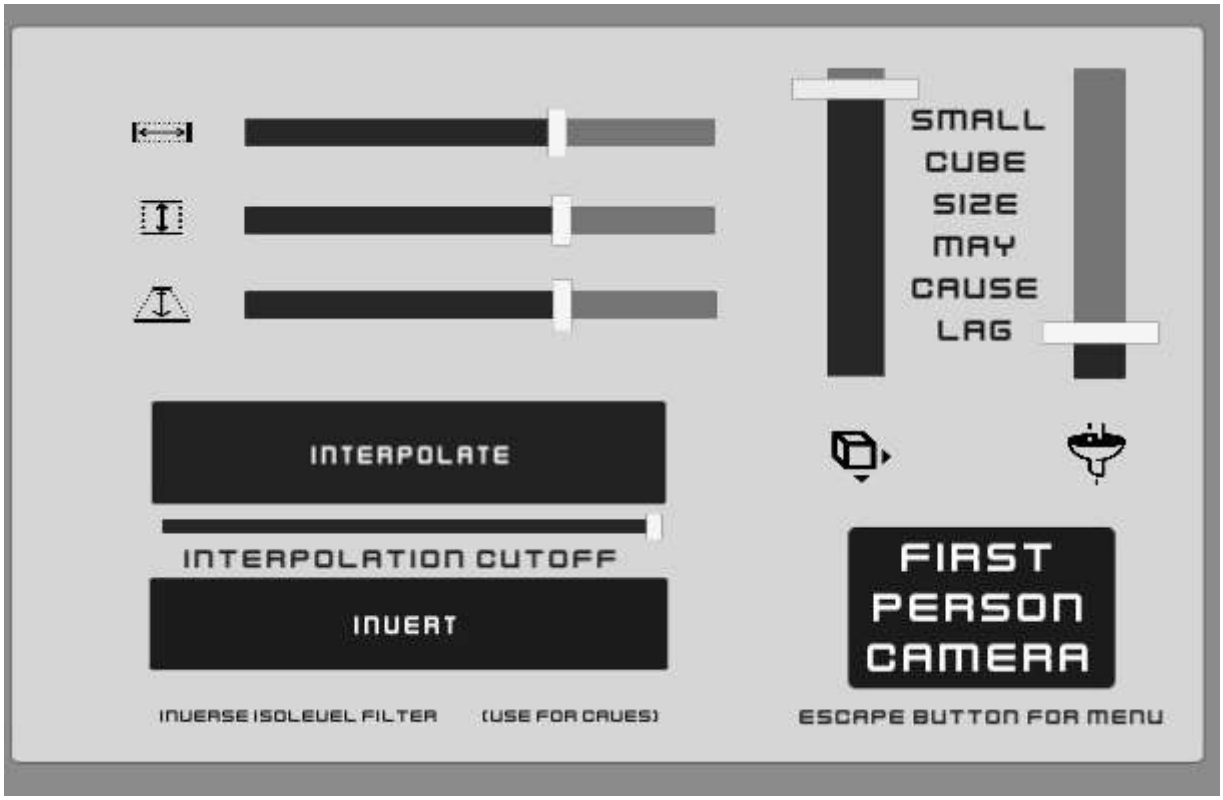
Implementation of this feature is done by sampling 2 vertices, using their values to find the exact midpoint of the line, using this midpoint we can find the ratio of the distance offset using the iso level (value) of the vertice. We can then use this percentage (mu) to correctly offset the midpoint using it as a magnitude of the direction of point2 minus point1.

```
Vector3 point;
float mu = (isoLevel - point1Value) / (point2Value - point1Value);

point.x = point1.x + mu * (point2.x - point1.x);
point.y = point1.y + mu * (point2.y - point1.y);
point.z = point1.z + mu * (point2.z - point1.z);

return point;
```

Further Details of the project

For this project I used Unity as a basis of setting up basic movement/camera. All rendering/graphics however is done though Polygon/Mesh generation supported by Unitys C# based code. I have also used free open source UI Images from https://icons8.com/ as well as open source First Person camera to allow the user to easily explore the marching cubes algorithm as a large object. The bundle I used for the First Person controller can be found here: https://assetstore.unity.com/packages/3d/characters/modular-first-person-controller-189884.

Guide of the program interface



- The Top left sliders will configure how big of a space the algorithm will sample
- The first vertical slider on the right (Cube Resize Icon) will configure the scale of the cubes generated each sample. Smaller sizes will create a more detailed mesh however will become highly computationally dependent.
- The second vertical slider (Filter icon) will determine the Iso Level, lower will accept lower level iso values to be activated, while higher iso level will filter out higher level iso values of vertices.
- Interpolate will toggle between interpolation of edge points, enabling edge points to estimate how far along the edge lines they should appear.
- Interpolation cutoff slider, will tell the algorithm how much of a difference the iso level needs to be in order for the edge point to calculate a different position rather than in the middle of the 2 vertices.
- Invert button, The invert button will swap the filter from filtering out vertices to filtering in vertices. This will create a mesh with back faces invisible (due to backface culling) but also a mesh that's created inside out. You can use this to create a “cave” like mesh, and allow your first person camera to drop inside to explore depending on how much of the space you filtered in/out.

Credits

- https://en.wikipedia.org/wiki/Marching_cubes
- https://www.youtube.com/watch?v=M3iI2l0ltbE
- https://developer.nvidia.com/gpugems/gpugems3/part-i-geometry/chapter-1-generating-complex-procedural-terrains-using-gpu
- https://people.eecs.berkeley.edu/~jrs/meshpapers/LorensenCline.pdf
- http://paulbourke.net/geometry/polygonise/
- https://www.youtube.com/watch?v=XdahmaohYvI
- https://alphanew.net/index.php?section=articles&site=marchoptim&lang=eng