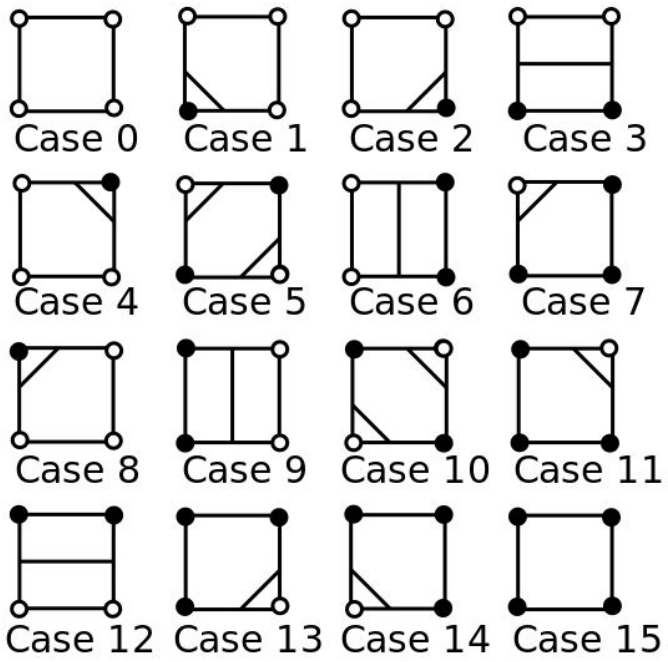
Assignments 2+3

174455 - Konstantinos Karatsenidis

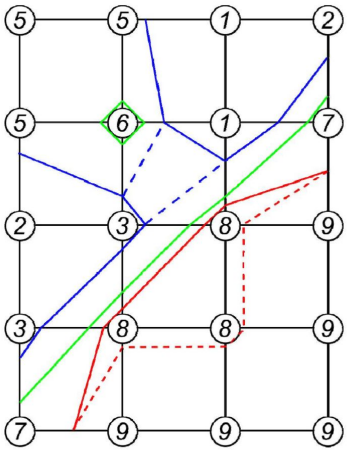
General remarks:

- Changed LoadData to keep the min and max value
- Changed DrawContour signature to keep aspect ratio
- Global contour stores 2d point tuples for marching squares contour lines
- Global vertices stores 3d points of triangles
- Global normals stores normal values of the vertices

Marching Squares



There are 7 cases that need lines drawn due to symmetry. Only case 5 (and 10) need 2 lines to be drawn. MarchingSquares selects the appropriate surrounding values and calls GetLinePoints which decides which is the current square case, and calculates the 2d points of the contour. The position of each point on the square's edge is determined by the weights of the corner values.



contour levels

- 4
- 4?
- 6-ε
- 8-ε
- 8+ε

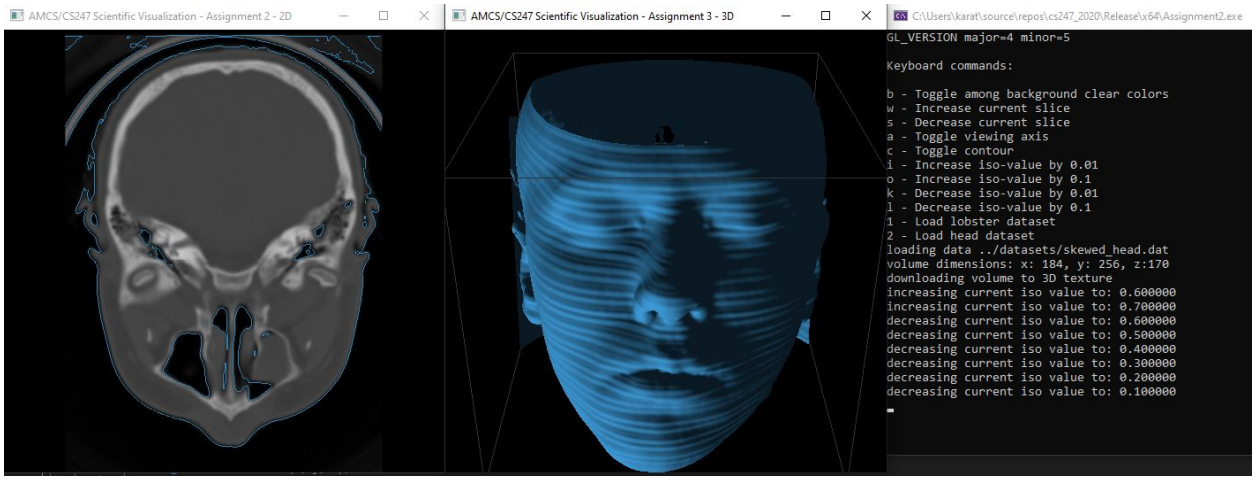
2 types of degeneracies:

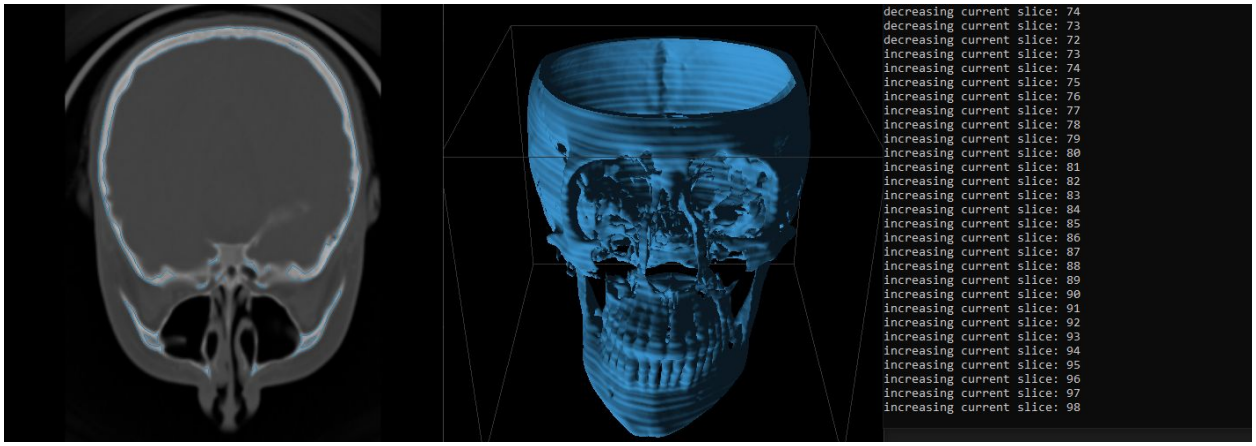
- isolated points (*c*=6)
- flat regions (*c*=8)

Avoiding degeneracies can be achieved with setting an appropriate epsilon value while calculating the `cubeIndex`. The idea is to introduce perturbations - if node value is equal to the contour, use value-e.

Marching Cubes

With a similar concept, but in 3d space, `MarchingCubes` for each cube, created by adjacent points in space, collects the vertex and normal values of its edges. Then it finds the appropriate cube case and according to the `edgeTable3D` calculates the triangle points for the vertex and the normal values. The values are normalized between [-1, 1].





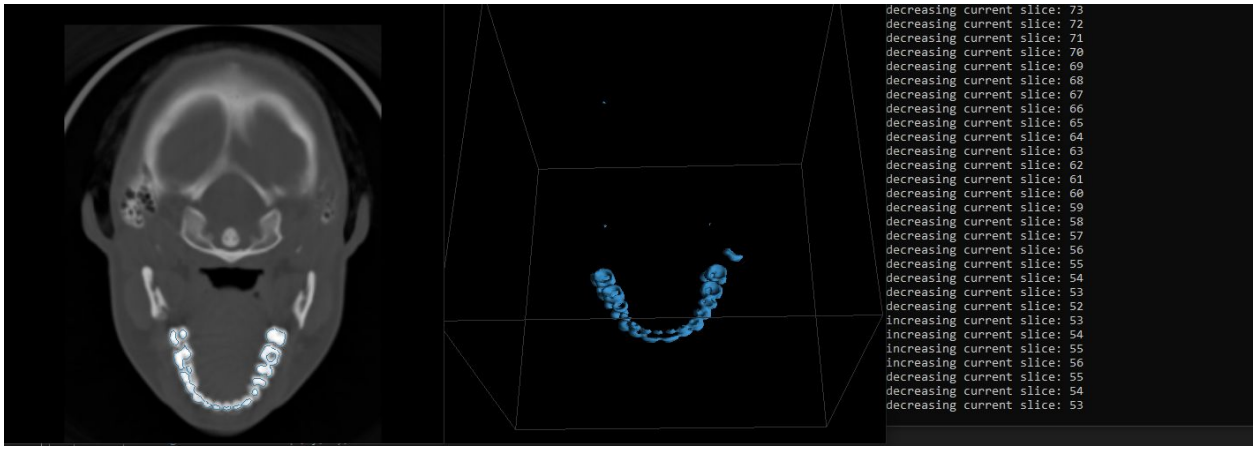


Table offset provides a simple way to access the neighbors of a point in an iterative way.
For each point, collect neighboring points coordinates p and values val.
Using these values, determine cube case cubeIndex.

```
853     for (int x = 0; x < vol_dim[0] - 1; x++) {
854         for (int y = 0; y < vol_dim[1] - 1; y++) {
855             for (int z = 0; z < vol_dim[2] - 1; z++) {
856                 Vertex p[8];
857                 double val[8];
858                 uint8_t cubeIndex = 0;
859                 for (uint8_t i = 0; i < 8; i++) {
860                     p[i].x = x + offset[i].x;
861                     p[i].y = y + offset[i].y;
862                     p[i].z = z + offset[i].z;
863                     val[i] = data(p[i].x, p[i].y, p[i].z);
864
865                     if (val[i] < isoValue) {
866                         cubeIndex |= 1u << i;
867                     }
868                 }
```

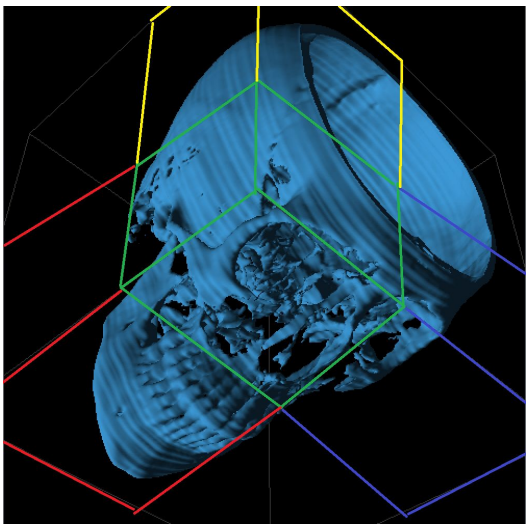
For each edge of the cubes, determine if a line should be drawn and calculate its points.

```
870             Vertex vertlist[12];
871             Vertex normalList[12];
872             for (uint8_t i = 0; i < 12; i++) {
873                 uint16_t bit = 1u << i;
874                 if (edgeTable3D[cubeIndex] & bit) {
875                     uint8_t a = points[i][0];
876                     uint8_t b = points[i][1];
877 
878                     Vertex& p1 = p[a];
879                     Vertex& p2 = p[b];
880                     double v1 = val[a];
881                     double v2 = val[b];
882                     Vertex n1 = midpointDiff(p1.x, p1.y, p1.z);
883                     Vertex n2 = midpointDiff(p2.x, p2.y, p2.z);
884                     float mu = (isoValue - v1) / (v2 - v1);
885 
886                     Vertex& vertex = vertlist[i];
887                     vertex.x = p1.x + mu * (p2.x - p1.x);
888                     vertex.y = p1.y + mu * (p2.y - p1.y);
889                     vertex.z = p1.z + mu * (p2.z - p1.z);
890 
891                     Vertex& normal = normalList[i];
892                     normal.x = n1.x + mu * (n2.x - n1.x);
893                     normal.y = n1.y + mu * (n2.y - n1.y);
894                     normal.z = n1.z + mu * (n2.z - n1.z);
895                 }
```

Table points is used to easily find the edges of each case.
For both methods, iso-value is normalized from [min, max] to [0, 1]. That means there will always be something to draw. For low values, we can see skin tissue, medium values reveal the skull, and the larger values show only the teeth. The contour line is displayed only on the slice where the object lies.

Parallelization

This algorithm can easily be parallelized, by splitting the 3d space to e.g. 8 subspaces. Each computing node can calculate the internal points of the subspace, only the adjacent perimeters need interprocess communication.



In this example, The process with the green subspace would have to send the edge values of the green cube to the red, yellow, and blue processes. It would also receive the edge values of the adjacent cubes. While waiting for communication, the internal values of the cube can run the marching cube algorithm as implemented. The 3d space can be split in many ways, depending how many processes we can run in parallel.