TNM067 — Scientific Visualization
3. ISO-surface extraction

September 6, 2022

1 Introduction

In the previous lab, you explored the hydrogen atom using volume rendering implemented in existing processors in Inviwo. In this lab, you will instead implement your own ISO surface extraction. To do this we will use a technique called Marching Tetrahedra, it is similar to Marching Cubes but will work on tetrahedra instead of cubes.

Important: The labs might require more hours than available in the scheduled lab sessions. It is important that you work on the assignments outside of sessions in order to be able to finish the labs.

1.1 Change log

- 2017-09-01: Initial Version.
- 2017-09-04: Typo fixes.
- 2020-09-02: Update for 2020, Update for distance teaching.
- 2021-07-19: Typo fixes, clarifications for some tasks, changed variable name Voxel to DataPoint, new task for marching tetrahedra tests.
- 2022-07-08: Fix to match code.
- 2022-08-31: Splitting up lab2 into lab2 and lab3

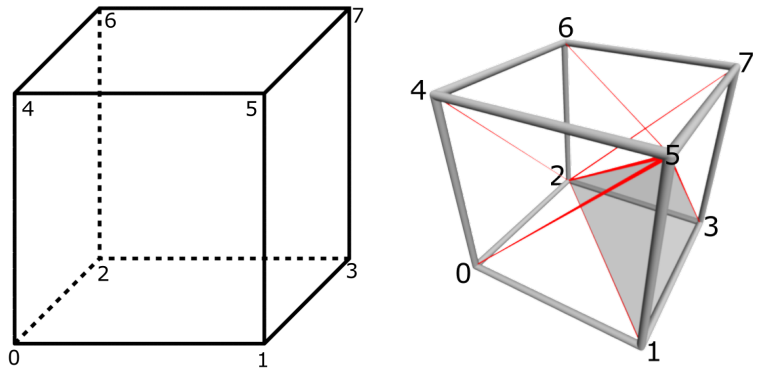
2 Documentation

On the following links you can read more about Inviwo and get help with questions related to Inviwo.

- Inviwo: http://www.inviwo.org
- Inviwo API Documentation: https://inviwo.org/inviwo/doc/
- Inviwo Issue tracker: https://github.com/inviwo/inviwo/issues

3 Marching tetrahedra

In Marching Cubes you loop over all cells in a volume. A cell can be seen as a "cube" which is constructed by 8 neighboring data points, as illustrated in figure 1(a). A case index is calculated based on whether the data points are inside or outside of the iso-surface (eg. higher or lower than the iso-value). The case index is then used as an index into a lookup table to determine if and how many triangles are to be created for this cell, which can be any number between zero and five. In Marching Tetrahedra we use a similar approach, but instead of calculating an index for the whole cell, we divide the cell into 6 tetrahedra (Figure 1(b)) and extract triangles from each tetrahedra. This will decrease the number of available cases from 256 to 16 (2^8 vs 2^4).



(a) (b)

Figure 1: On the left we have a cell consisting of 8 data points. On the right, the cell has been divided into six tetrahedra.

Another benefit of Marching Tetrahedra vs Marching Cubes is that there are no ambiguous cases. In Marching Cubes, there are a few cases where it is ambiguous how to construct the triangles. Since a tetrahedron is a Euclidean Simplex we will not have any ambiguous cases.

```
struct DataPoint {
    vec3 pos;
    float value;
    size_t indexInVolume;
};
```

```
struct Cell {
    DataPoint dataPoints[8];
};
```

```
struct Tetrahedra {
    DataPoint dataPoints[4];
};
```

Figure 2: The struct **DataPoint** stores the spatial location, the function value and the index of a data point in the volume. Eight data points are used to build up a **Cell**, each representing the corner of a cube. Finally we have the **Tetrahedra**, which is similar to the cell but contains 4 data points instead of 8, one for each corner in the tetrahedra.

In the code you are given a few structs to aid the use and access to the data. See Figure 2 for more info. The following tasks will all be implemented in *marchingtetrahedra.cpp*.

Task 1 — Implement data point index and position functions:
To create a cell, you first have to implement the two functions *calculateDataPointIndexInCell* and *calculateDataPointPos*. The first function maps a 3D index to a 1D index, which is then used to access the correct data point within the cell. The second function takes the position of the cell in the volume, the 3D index of the data point within the cell, and the dimension of the volume. This function should return the position of the data point within the volume **scaled between 0 and 1**. To test your implementations, change *ENABLE_DATAPOINT_INDEX_TEST* and *ENABLE_DATAPOINT_POS_TEST* to 1 in *marchingtetrahedra.h* and set inviwo-unittests-tnm067lab3 as start-up project.

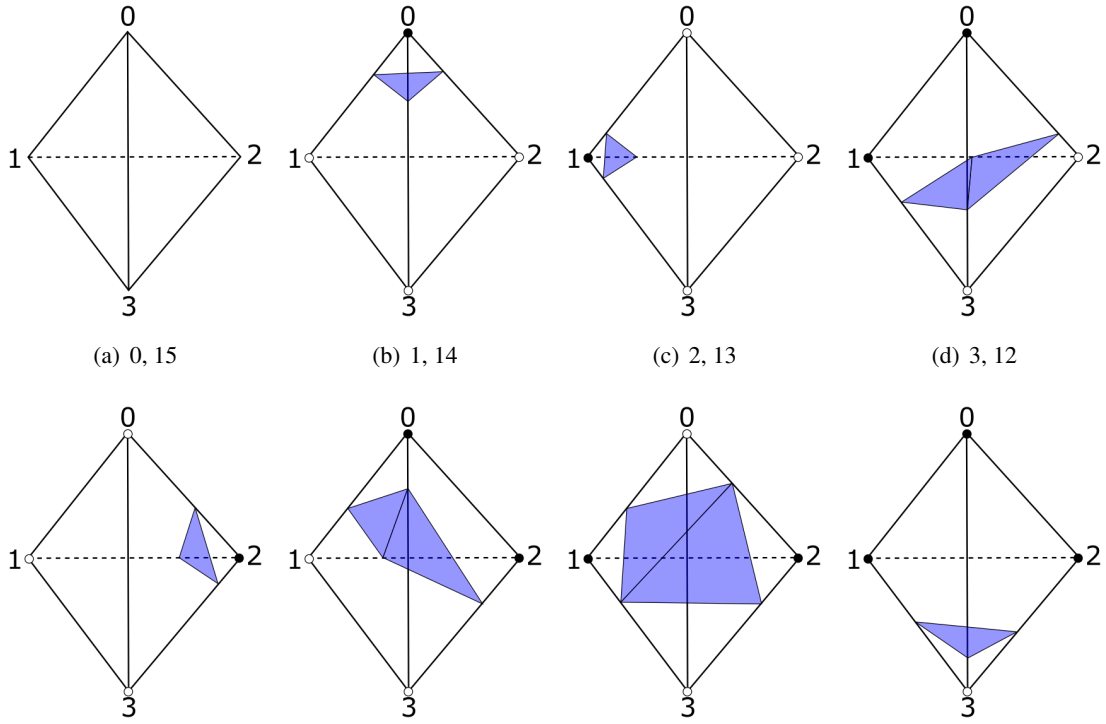
Task 2:
Create a nested for loop to loop over the 8 data points needed to construct the cell. Set the spatial position, function value and 1D-index of each data point. Use the functions implemented in the previous step to ensure you access the cell with the correct index and use the scaled spatial position. Also, note that the index used to access the cell is different from the index of the data point.

Task 3 — Divide cell into 6 tetrahedra:
Divide the cell from previous task into 6 tetrahedra using Figure 1(b) as a reference. Note: use tetrahedraIds in the code.

There is a class MeshHelper which is a wrapper for Inviwo's BasicMesh, this class is created to aid in the calculation of normals for the mesh. It has the function addVertex which takes the spatial position of a vertex and two additional integers i and j. These integers are indices of the data points that spans the edges on which the vertex lies on. The addVertex-function will only add a new vertex to the mesh if a vertex has not yet been created on that edge. The addVertex-function will return the id of the vertex, either the id of the previous vertex or the created vertex. The second method on MeshHelper that you will use is addTriangle, it takes the three indices of the vertices spanning the triangles.

Task 4 — Extract triangles from the tetrahedra:
Calculate the case id for each tetrahedra. Each tetrahedra will give 0, 1 or 2 triangles depending on which case it is. For each case, you will have to calculate the location of the vertices of the triangle(s), add the vertex to the mesh, and finally, create triangles between the vertices. MeshHelper will calculate the normals of each triangle for you, it does that by using the spatial location of each triangles vertices. You should now be able to see results if you run the *lab2-marchingtetrahedra* example workspace. If the order of the vertices is wrong, some triangles will get incorrect lighting. If this is the case for you, one way to approach this is to disable all but one case, and make sure that this case has correct order.

Task 5: (optional)
This implementation of marching tetrahedra is far from optimized. Can you think of any parts that can be optimized? Where do we have bottle necks? Are we doing any redundant calculations?



(a) 0, 15 (b) 1, 14 (c) 2, 13 (d) 3, 12

(e) 4, 11 (f) 5, 10 (g) 6, 9 (h) 7, 8

Figure 3: The different cases for marching tetrahedra.