TNM067 — Scientific Visualization
2. Scalar Fields and Volume Rendering

September 21, 2021

1 Introduction

This laboratory exercise will introduce scalar field visualization. The first part will include generating a scalar field based on the wave function of a hydrogen atom. The second part will then introduce two ways to visualize this scalar field using volume rendering. The first way is to use different raycasting techniques to explore the scalar field while the second way is to extract a mesh representing an ISO-surface.

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.

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 Part 1 - Hydrogen Wave Function

In the first part of the lab, we will generate a scalar field from the wave function of a hydrogen atom. From the wave function it is possible to calculate the probability of finding an electron at each position in space. The generated probability density plot will be used as a scalar field throughout the lab, on which we will test multiple scalar field visualization techniques.

3.1 The Wave Function

The wave function consists of two parts: the radial function $R_{nl}(r)$ and the spherical harmonics $Y_{lm}(\theta, \phi)$ (providing the angular part). The subscripts n, l, m are called *the three principal quantum numbers* and in this case you will use the wave function for $n = 3$, $l = 1$ and $m = 0$. Using spherical coordinates:

$$R_{3,1}(r) = \frac{4\sqrt{2}}{3} \left(\frac{Z}{3a_0}\right)^{\frac{3}{2}} \left(\frac{Zr}{3a_0}\right) \left(1 - \frac{Zr}{6a_0}\right) e^{-Zr/3a_0} \tag{1}$$

$$Y_{1,0} = cos\theta \tag{2}$$

giving the complete wave function:

$$\Psi_{3,1,0}(r, \theta, \phi) = \frac{1}{81\sqrt{6\pi}} \left(\frac{Z}{a_0}\right)^{\frac{3}{2}} \frac{Z^2 r^2}{a_0^2} e^{\frac{-Zr}{3a_0}} \left(3cos^2\theta - 1\right) \tag{3}$$

from which the electron probability density function is calculated as

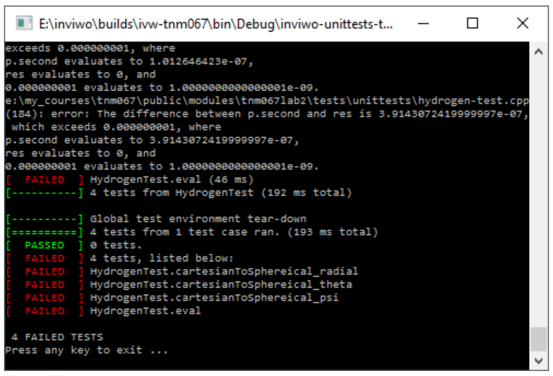
$$P_{3,1,0} = |\Psi_{3,1,0}(r, \theta, \phi)|^2 \tag{4}$$

In your code, do you have to calculate the absolute value of $\Psi_{3,1,0}(r, \theta, \phi)$? The constant a_0 is the *Bohr radius* (0.53Å) and Z is the charge of an electron. Because this case regards a single electron hydrogen, you should set:

$$Z = 1, \quad a_0 = 1 \tag{5}$$

If you feel the urge to read more about the wave function and the probability density function you can do so at http://academic.reed.edu/chemistry/roco/Density/cloud.html or read the book *Physics of Atoms and Molecules* by B.H. Brandsen and C.J. Joachain.

There is a set of equations which enables conversion between the Cartesian coordinate system and the spherical coordinate system. The angles ϕ and θ are easily mixed up because the notations are sometimes different in maths and physics, the notation here is that of physics. You can read more about spherical coordinates at http://mathworld.wolfram.com/SphericalCoordinates.html.

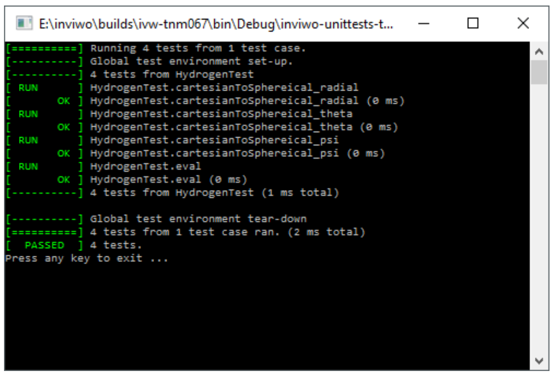


```
E:\inviwo\builds\ivw-tnm067\bin\Debug\inviwo-unittests-t...
exceeds 0.000000001, where
p.second evaluates to 1.012646423e-07,
res evaluates to 0, and
0.000000001 evaluates to 1.0000000000000001e-09.
e:\my_courses\tnm067\public\modules\tnm067lab2\tests\unittests\hydrogen-test.cpp
(184): error: The difference between p.second and res is 3.9143072419999997e-07,
 which exceeds 0.000000001, where
p.second evaluates to 3.9143072419999997e-07,
res evaluates to 0, and
0.000000001 evaluates to 1.0000000000000001e-09.
[  FAILED  ] HydrogenTest.eval (46 ms)
[----------] 4 tests from HydrogenTest (192 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test case ran. (193 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 4 tests, listed below:
[  FAILED  ] HydrogenTest.cartesianToSphereical_radial
[  FAILED  ] HydrogenTest.cartesianToSphereical_theta
[  FAILED  ] HydrogenTest.cartesianToSphereical_psi
[  FAILED  ] HydrogenTest.eval

 4 FAILED TESTS
Press any key to exit ...
```

(a)



```
E:\inviwo\builds\ivw-tnm067\bin\Debug\inviwo-unittests-t...
[==========] Running 4 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 4 tests from HydrogenTest
[ RUN      ] HydrogenTest.cartesianToSphereical_radial
[       OK ] HydrogenTest.cartesianToSphereical_radial (0 ms)
[ RUN      ] HydrogenTest.cartesianToSphereical_theta
[       OK ] HydrogenTest.cartesianToSphereical_theta (0 ms)
[ RUN      ] HydrogenTest.cartesianToSphereical_psi
[       OK ] HydrogenTest.cartesianToSphereical_psi (0 ms)
[ RUN      ] HydrogenTest.eval
[       OK ] HydrogenTest.eval (0 ms)
[----------] 4 tests from HydrogenTest (1 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test case ran. (2 ms total)
[  PASSED  ] 4 tests.
Press any key to exit ...
```

(b)

Figure 1: Task 1 and 2 are using unit test to verify your solutions.

To go from the spherical coordinate system to the Cartesian coordinate system, use:

$$x = r \sin\theta \cos\phi \tag{6}$$
$$y = r \sin\theta \sin\phi \tag{7}$$
$$z = r \cos\theta \tag{8}$$

To go from the Cartesian coordinate system to the spherical coordinate system, use:

$$r = \sqrt{x^2 + y^2 + z^2} \tag{9}$$
$$\theta = \cos^{-1}(z/r) \tag{10}$$
$$\phi = \tan^{-1}(y/x) \tag{11}$$

For task 1 and 2, set the startup project in Visual Studio to inviwo-unittests-tnm067lab2. This project contains unittests that will test code implemented in Task 1 and 2. At first, all test will fail (see Figure 1) . When you have implemented task 1, three out of four test will pass and after task 2, all test should pass. When all tests pass, change the startup project back to inviwo.

Task 1 — Cartesian to spherical conversion:
Implement the function *HydrogenGenerator::cartesianToSpherical* based on equations (9-11).

Task 2 — Implement the wave function:
Implement the function *HydrogenGenerator::eval* to evaluate equation (4). A common problem for some students with this task is to get lost in the amount of parentheses. Therefore, we strongly suggest to split the equation up on to smaller parts. Equation (12) shows and example way to divide the equation in to 5 parts whose product builds the final value.

$$\Psi_{3,1,0}(r,\theta,\phi) = \boxed{\frac{1}{81\sqrt{6\pi}}}\ \boxed{\left(\frac{Z}{a_0}\right)^{\frac{3}{2}}}\ \boxed{\frac{Z^2r^2}{a_0^2}}\ \boxed{e^{\frac{-Zr}{3a_0}}}\ \boxed{\left(3cos^2\theta - 1\right)} \tag{12}$$

4 Part 2 - Volume Rendering

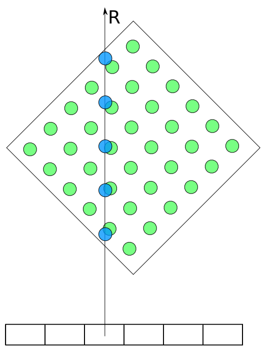


Figure 2: Casting a ray from the screen into the volume. Samples are taken along the ray which are composited to form the final pixel color.

4.1 Raycasting

Raycasting is an algorithm for performing direct volume rendering. The core of the algorithm is based on sending one ray, R, per pixel from the camera and then take samples along the ray inside a volume (see Figure 2). At each sample there is an amount of illumination, $I(x, y, z)$, from the light sources. Not all illumination is scattered in the direction to the camera, it depends on for instance on the local density, $D(x, y, z)$, at the point. It is not trivial to calculate the illumination, to do it correctly you must take the attenuation of the light source and possible shadows into account. A simplified approach is to calculate the gradient, i.e normal, at the sample point and then perform for instance Phong shading to obtain the intensity.

In OpenGL rays can be formed by rendering a proxy geometry around the volume. Keep in mind

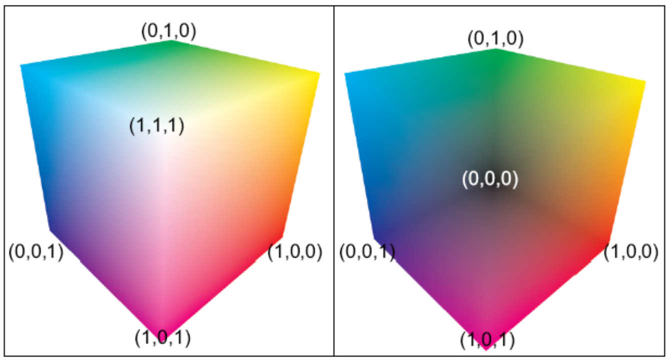
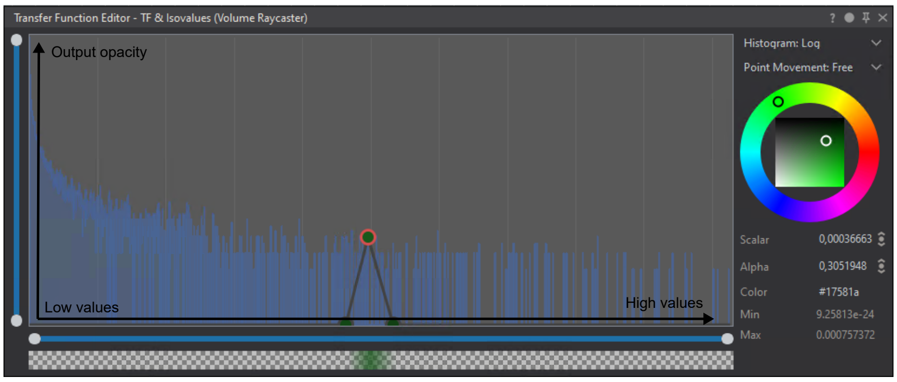


Figure 3: Texture coordinates of the front and back face rendered cubes.


Transfer Function Editor - TF & Isovalues (Volume Raycaster)
Output opacity
Low values
High values

Histogram: Log
Point Movement: Free
Scalar 0,00036663
Alpha 0,3051948
Color #17581a
Min 9.25813e-24
Max 0.000757372

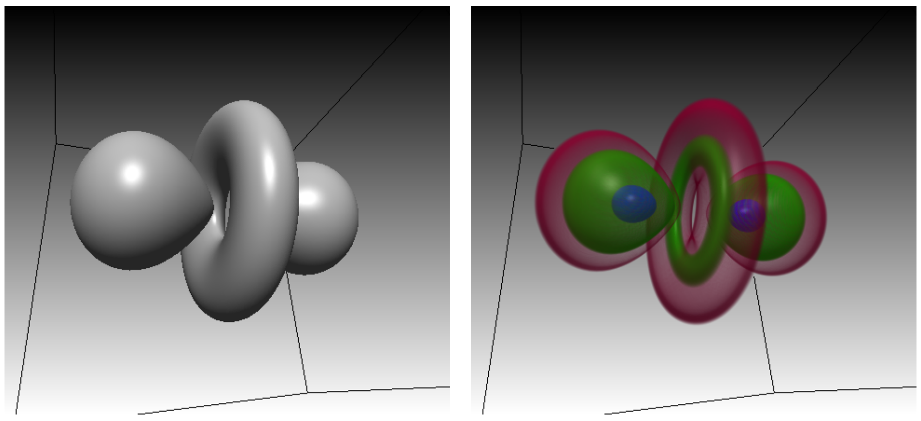
Figure 4: Inviwo's transfer function editor. Inviwo has a quite flexible transfer function editor. On the background you see blue bars representing the histogram of the voxel-intensities of the volume. You add and remove points by right clicking and move points by clicking and dragging. The location of the points on the x-axis represent what voxel-intensities it will be applied on and the height on the y-axis represent the output opacity of current sample. With the current setup all samples along the ray will have zero opacity, except those who samples whose intensity falls into the range around the green peak-point.

that a ray is an origin with a direction. Thus we must find out the origin and the direction for each ray. This can be done by rendering two cubes where the colors represent the coordinates. Each corner will have a coordinate and they will then be interpolated from the vertex shader to the fragment (pixel) shader, see figure 3. To get the origin a cube can be rendered with ordinary front face culling, this is where the ray will start/enter the volume. To get the direction of the ray the cube can be rendered using back face culling, the direction will then be the back face subtracted from the front face. Practically this can be done by storing the result of rendering the back face in a framebuffer and then use the front face to generate the fragments that will contain the origin of the rays.

For the following tasks in this part, you will use built in components in Inviwo to visualize the hydrogen, no programming will be needed for these tasks. You might want to save the workspace so you don't have to recreate it everytime you restart Inviwo. **If you are unsure how to connect components in Inviwo to render a volume, start by looking at the example workspace boron under File->Example Workspaces->Core->boron.inv**.

Task 3 — ISO Raycaster:
Inviwo has a processor called "ISO Raycaster" which uses a raycaster to visualize ISO surfaces of a volume. Extend your network by adding another canvas processor and the processors needed to visualize the hydrogen using the "ISO Raycaster". When you have your network you can see different ISO-surfaces by changing the ISO-value property on the "ISO Raycaster". See figure 5(a) for an example result.

Task 4 — Raycasting:
Similar to the "ISO Raycaster" Inviwo has a processor called "Volume Raycaster", this uses a transfer function, lighting and front-to-back compositing to let the user visualize multiple surfaces by using transparency. Open the transfer function editor by clicking the transfer function property in the "Volume Raycaster". Try to create a transfer function mimicking the example image in figure 5(b), which visualizes three different iso-values with different color and opacity. A good start for this is to try to make copy of the transfer function you can see in Figure 4.
Note: You can select and move several points at once by drawing a square around them.



(a) (b)

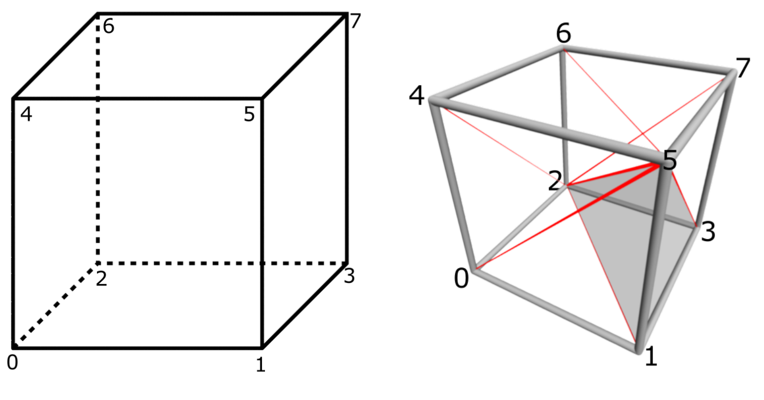
Figure 5: Possible results when visualizing the hydrogen using an ISO Raycaster and a Volume Raycaster.

4.2 ISO Surface Extraction

You have now explored the hydrogen using raycasting. In this part you will instead visualize it by extracting a mesh representing an ISO surface. 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.

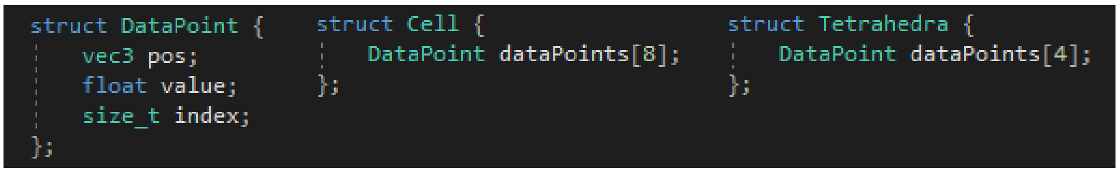
In Marching Cubes you loop over all cells in a volume. A cell can be seen as a "cube" which is constructed by 8 neighboring data points, as illustrated in figure 6(a). A case index is calculated based on whether the data points are inside or outside of the iso-surface (eg. higher or lower than the iso-value). 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 6(b)) and extract triangles from each tetrahedra. This will decrease the number of available cases from 256 to 16 (2^8 vs 2^4).

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.



(a) (b)

Figure 6: On the left we have a cell consisting of 8 data points. On the right, the cell has been divided into six tetrahedra.



```
struct DataPoint {        struct Cell {                        struct Tetrahedra {
    vec3 pos;                 DataPoint dataPoints[8];             DataPoint dataPoints[4];
    float value;          };                                   };
    size_t index;
};
```

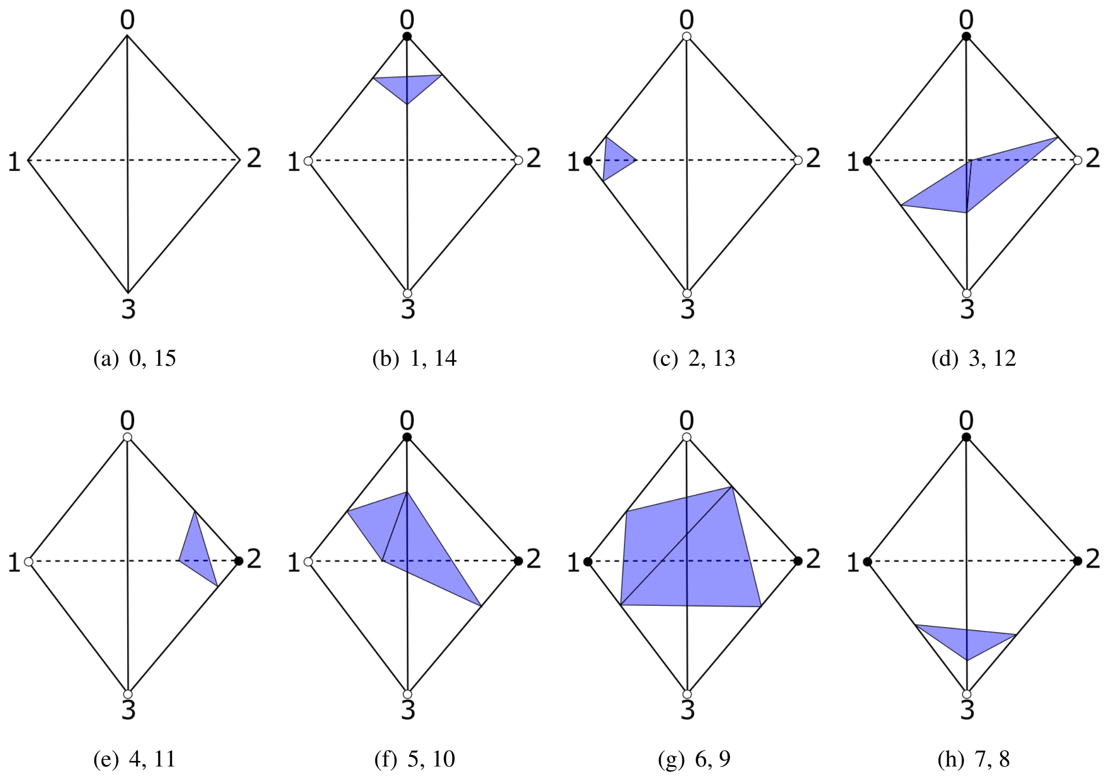
Figure 7: The struct **DataPoint** stores the spatial location, the function value and the index of a data point in the volume. Eight data points are used to build up a **Cell**, each representing the corner of a cube. Finally we have the **Tetrahedra**, which is similar to the cell but contains 4 data 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 7 for more info. The following tasks will all be implemented in *marchingtetrahedra.cpp*.

Task 5 — 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-tnm067lab2 as start-up project.

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



(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 8: The different cases for marching tetrahedra.

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 8 — 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 9: (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?