Visualization 2IV35

Oliver Schinagl (0580852)
Wilrik De Loose (0601583)

January 11, 2008

Contents

1 Introduction **3**

2 The skeleton code **5**
2.1 Grid-based smoke 5
2.2 Internal structure 6

3 Color mapping **7**
3.1 Description 7
3.2 Implementation 8
3.3 Difficulties 9

4 Glyphs **10**
4.1 Description 10
4.2 Implementation 11
4.3 Difficulties 12
4.4 Quake root 12

5 Divergence **14**
5.1 Description 14
5.2 Implementation 14
5.3 Difficulties 15

6 Isosurfaces **16**
6.1 Description 16
6.2 Implementation 16
6.3 Difficulties 18

7 Height plots **20**
7.1 Description 20
7.2 Implementation 20
7.3 Difficulties 22

8 Streamtubes **24**
8.1 Description 24
8.2 Implementation 24
8.3 Difficulties 26

9 Conclusion **27**
9.1 Visualization 27
9.2 The course 28

Chapter 1

Introduction

The visualization course focuses on techniques and algorithms used to visualize large data sets. The code of a 2D fluid simulator was supplied to implement such techniques on.

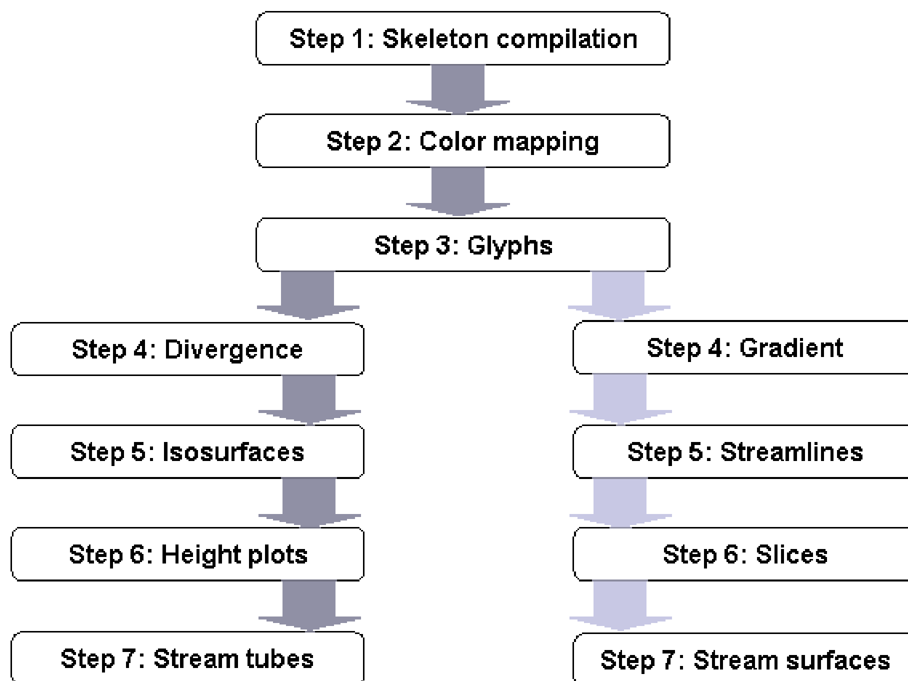

Figure 1: Practical Assignment

The above figure illustrates the different assignments. After step 3 there were two different paths to continue the practical assignment. The left branch of the figure which, amongst others, involved implementing isosurfaces and

height plots was chosen.

Every chapter in this report discusses a particular objective from the practical assignment. The next chapter will briefly talk about the skeleton program supplied and how the code was adapted to be able to add functionality. Chapter 3 will discuss the color mapping, chapter 4 is about glyphs and so on.

A few subjects for each practical assignment will be discussed. Each chapter gives a short description of each technique with a couple of screenshots, it talks about how the technique was implemented and what problems were encountered, if any.

The report ends giving a detailed conclusion about visualization in general and the visualization course.

Chapter 2

The skeleton code

The provided code for the course contains an implementation of a real-time fluid flow simulation. The fluid flows under the influence of a user-controlled force field. The simulation follows the Navier-Stokes equations for fluid flow. These equations describe how the velocity, pressure, temperature and density of a moving fluid are related.

2.1 Grid-based smoke

The visualization is done using a two dimensional grid as opposed to particle-based smoke visualization techniques. At every vertex a number of fluid attributes are stored. The skeleton program keeps track of the density, velocity and force. These attributes can be visualized with different techniques. Each and every attribute has its own 'preferred' visualization technique.

Using these values at the cell's vertices, all sorts of techniques, additional values and other useful info can be used. The values are used in colormap techniques, calculation of the divergence and the rendering of the height plots.

Figure 2: Fluid visualization

The above screenshot shows us the fluid movement using a grey colormap. This simple yet effective visualization technique is explained in chapter 3.

2.2 Internal structure

The first assignment was to simply compile the code. This was fairly easy. However, the code was a bit unstructured. Everything was put into one big file. To be able to add additional functionality without losing sight it was needed to create a file for every new technique being implemented. Furthermore the calculations, rendering functions and user interaction code were split from each other.

With this new structure it was possible to be able to manage the code and add additional functionality. The programming language of our choice was C and for the graphical user interface the GTK+ library with the OpenGL extension was chosen. For project managment SVN was used. This was very useful since both students had different development environments and also did a lot of work at home.

Chapter 3

Color mapping

In chapter 2 it was seen from figure (figure 2) that the fluid had a very bright grey color. How is this color determined at every vertex?

3.1 Description

The technique that maps a value to a specific color is called color mapping. It has already been explained that the simulation is divided into cells with each 4 vertices that can contain different values, a uniform quad grid. A colormap calculates the color, given a certain colormap function, for every value at a vertex.

Example: In figure 2 the smoke was using a grey scaled colormap. If the value of vertices range from 0 to 1, the value can be used to determine each color aspect, red, green and blue. This means, for each vertex:

$$red = green = blue = value$$

To be able to reason about the colored images, a legend at the top of the screen was added. The leftmost colors indicate low values and the rightmost colors indicate high values. With such a colormap legend, it's easier to understand the produced images and say something about the value of the fluid.

Using a slider it is possible to change the number of colors. By default this value is set to 256 colors, but this is easily changed to anything below that. At lower number of colors big bands of colors will start to appear making variations more visible.

3.2 Implementation

Next to the 3 supplied colormaps 5 additional colormaps where added. Three of these where only contain one color, red, green or blue. These are not only useful for isolines but also when wanting to see height plots, but not get distracted by the flow of density. The remaining two colormaps are the "Wilrik" colormap, it implements a fire elemental color scheme and the "Oliver" colormap which may seem like a weird odd map, not like a colormap at all. It does serve a purpose however, it allows to easily see rapid changing values.

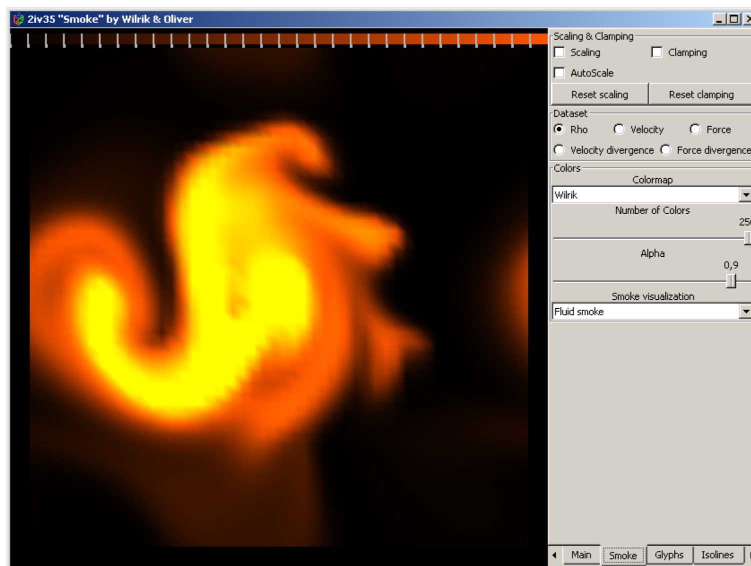

Figure 3: A fire colormap

The fire color is determined as follows:

$$red = value; green = value/3; blue = 0;$$

If the value is high, a lot of red, one third of green and no blue is taken. Low values only get a bit of red and almost no green. This gives a black to dark red to orange to almost yellow colormap giving the illusion of fire.

To create a repeating band of colors, the following formula was used:

$$value = (((int)(value * 100)) \texttt{ mod } 10)/10$$

By first multiplying our value by 100 and converting it to an integer the value get's reduced in significance. If this would be divided by 100 again the result would be 100 different bands of color. By taking the module of 10 first

however, a repeating sequence of bands is achieved. To compensate for the earlier multiplication of 100 it's required to devide by 10 once more, as color values are set to be between 0 and 1.

3.3 Difficulties

The last mechanisms to implement for this assignment were scaling and clamping. With clamping the user is able to set a minimum and maximum for the values. This is done by first enabling clamping and then clicking on in the bottom half of the color legend and drag the minimum and maximum clamping indictors, which are white. Actual data lower then the minimum or higher than the maximum calculated value are then clamped to the this minimum and maximum respectively.

The (auto)scaling mechanism was a bit more subtle. When enabled, the scaling is set in the same way as the clamping but for the top half of the legend bar, marked in red. For the auto-scaling bit, the minimum and maximum values are stored and the entire frame and is mapped to the visible colormap. This is was quite trivial, was it not that the values could also be negative. Something that went unnoticed until divergence was implemented.

Chapter 4

Glyphs

When taking a snapshot of the moving fluid a lot of information about the simulation is lost. The direction the fluid was heading for instance. Glyphs are used to take care of such problems.

4.1 Description

"Glyphs are icons that convey the value (orientation, magnitude) of a vector field in several graphical ways, such as hedgehogs or arrows."

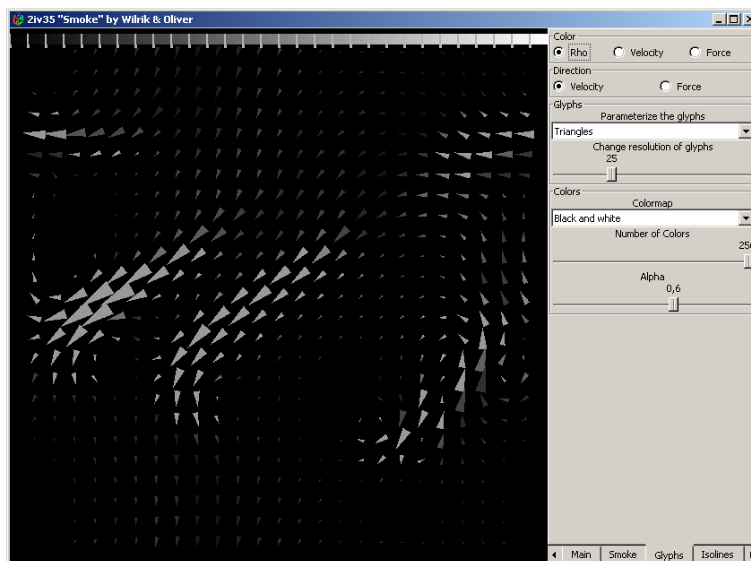

Figure 4: Triangle glyphs with a grey scaled colormap

The smoke in the fluid simulation has a direction and speed. These two values can be represented by a vector encoding the direction as the orientation and the speed as the magnitude. If this vector is drawn for each vertex a vector field that shows the direction and speed of the fluid at each vertex then becomes visible.

In the figure above (figure 4) a triangle glyph with a grey scaled colormap is visible. In the lower right region the fluid is moving upwards where as the rest of the simulation the fluid is trying to get to the bottom of the screen.

4.2 Implementation

Four additional glyphs where implemented besides the already implemented hedgehogs. Triangles (see figure 4), 3D cones, and 2 bitmap image glyphs (see figure 5). Every glyph has to go through a certain few steps explained below.

- calculate size (length)
- calculate angle (rotation)
- rotate and render object

The length is calculated using the equation of Pythagoras:

$$length = \sqrt{x^3 + y^3 + z^3}$$

The angle Θ of the glyph is calculated using the inner product (*inprod*). To calculate the glyphs angle, the inner product is used:

$$\Theta = acos(inprod) * (180^\circ/\Pi)$$

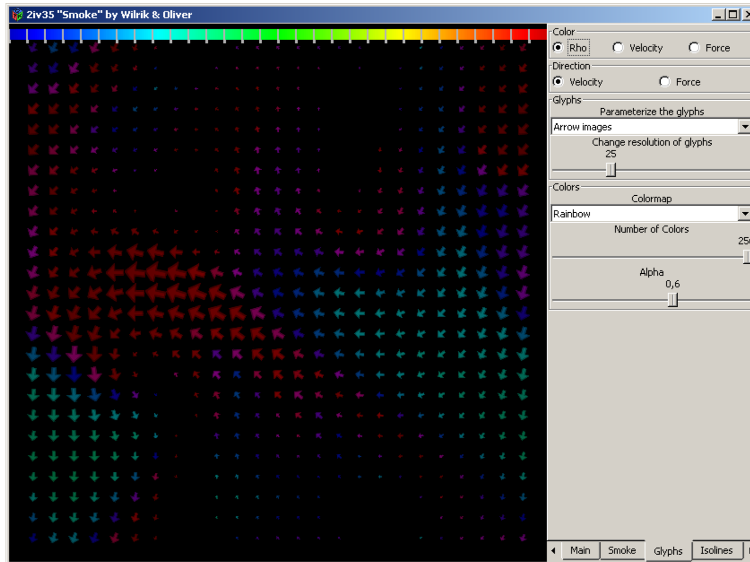

Figure 5: Image glyphs colored with a rainbow colormap

Additionally it was required for this assignment to allow the user to choose an alternative resolution for the sample grid. The default grid resolution is 50x50. To visualize this grid, the value of the nearest-neighbor for the glyph is used.

4.3 Difficulties

Troubles began with calculating the angle of the glyph. For some reason the calculation checks the smallest angle which is between 0° and 180°. After locating this irregularity the problem it was easily fixed.

4.4 Quake root

A nice side note to this chapter is that the default $sqrt(float)$ function that the C-library offers, to determine the length of a vector for example, is not used. A function that will be called "Quake Root" from the game Quake 3 is used in it's place.

The code implements the Newton Approximation of roots, which is supposed to be ran in iterations; each iteration enhances the accuracy until enough iterations have been made for reaching the desired accuracy. Interestingly this

function features a magic constant, used to calculate the initial guess. Only one Newton approximation iteration is required for a low relative error of 10^{-3}.

The function runs up to 4 times faster than the $sqrt(float)$ function, even though it is usually implemented using the FSQRT assembly instruction!

More information about the magical square root can be found online at: http://www.codemaestro.com/reviews/9

Chapter 5

Divergence

After implementing the glyphs it was time for the divergence. This is a value which can be calculated using the already known value at each vertex and that of its neighbors.

5.1 Description

Divergence shows the amount of mass which is compressed or expanded. If mass enters the field at some point, called a source point, then that point will have a positive divergence value. If mass exits the field at some point, called a sink point, then that point will have a negative divergence value.

5.2 Implementation

The calculation of the divergence is rather trivial. For the divergence of the velocity this looks like:

$$\nabla \cdot v = \frac{\partial v}{\partial x} + \frac{\partial v}{\partial y} =$$

$$\frac{v_x(i+1,j) - v_x(i,j)}{cell_x} + \frac{v_y(i+1,j) - v_y(i,j)}{cell_y}$$

Where $v_x(i,j)$ is the x-value at the gridpoint (i, j) and similar for the y-value. The $cell_x$ and $cell_y$ are the width and height of the cell. The same method is used for the force field.

The definition of the divergence says high values should appear where matter is injected and low values where matter exits the field. This means high values should appear in front of the mouse while dragging with low values behind it.

5.3 Difficulties

Although divergence is clearly defined, it is rather difficult to verify it using the simulation. This is a know difficulty. Divergence can however be verified by setting up a test environment where the outcome of the divergence should be obvious. For instance create one source and one sink point.

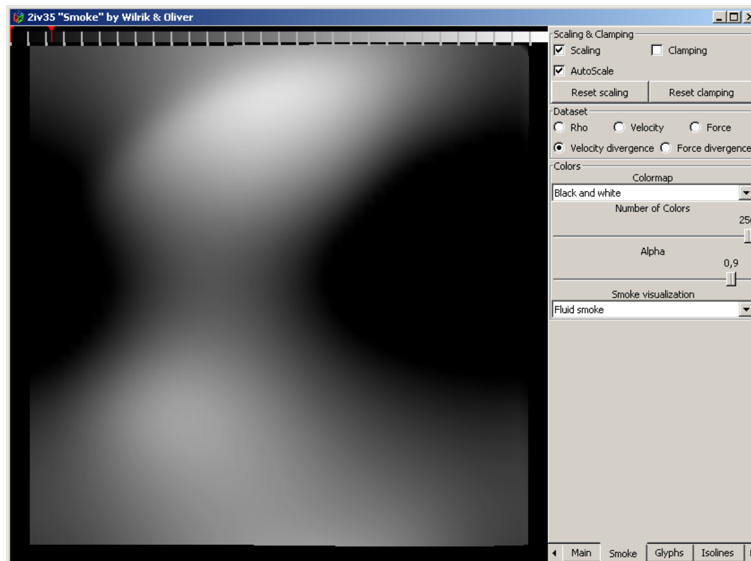

Figure 6: Velocity divergence

Chapter 6

Isosurfaces

This assignment was a really interesting one. It introduces a method to implement so called isolines. Such an isoline creates a surface that encapsulates a region with a value higher than a given threshold.

6.1 Description

An isoline is a line that more or less follows a given value. If the values of a field ranges from 0 to 1, a good threshold would be 0.6. An isoline could then visualize the points that equal the value of 0.6.

The simulation program is able to set a certain the number of isolines with a minimum and maximum values between two 'contour'-isolines.

6.2 Implementation

The algorithm that implements the isolines follows a structured pattern. In pseudo-code it looks like this:

for (each cell c_i of the dataset)
{
    for (each vertex v_j of c_i)
        store inside/outside state of v_j in bit b of *status*;
    select the optimized code from the case table using *status*;
    for (all cell edges e_j of the selected case)

```
        intersect e_j with the isovalue;
    construct the line segment(s);
}
```

The algorithm passes through every cell and then checks the four cell vertices $\{v_0, v_1, v_2, v_3\}$ of that cell. Each vertex has it's own value. With that value, the algorithm can check if a vertex is inside ($v_j \geq threshold$) or outside ($v_j < threshold$) the isosurface. The inside/outside state is then stored.

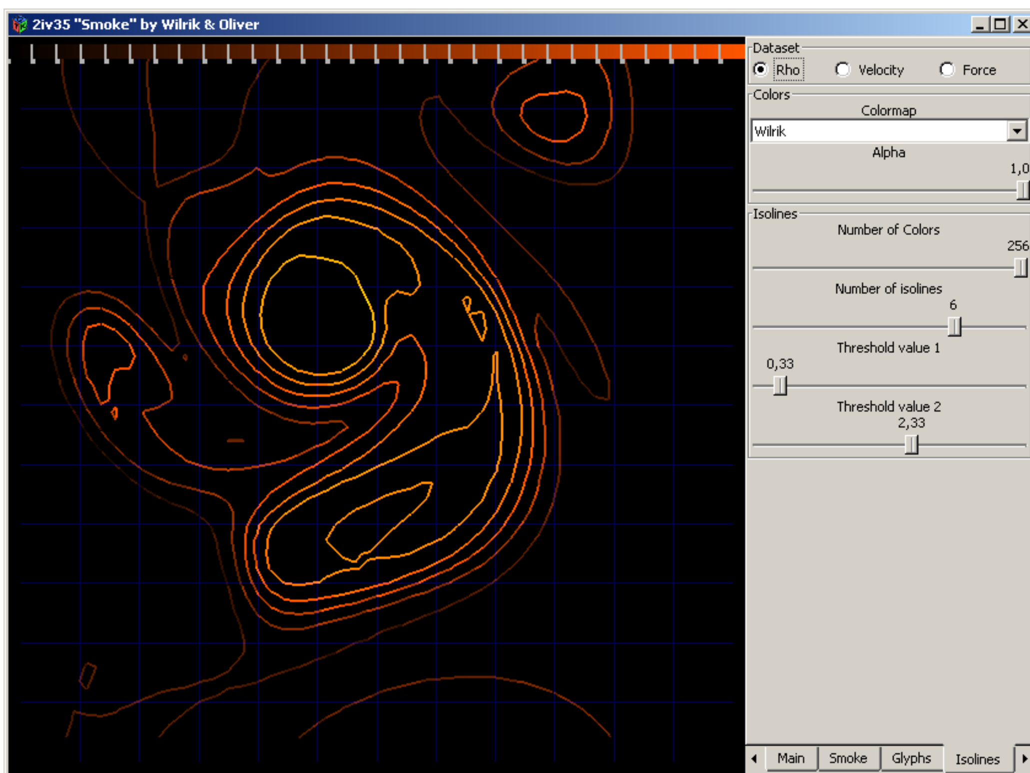

Figure 7: Fire-like isolines on a blue grid

If a vertex, v_0 is inside the isosurface, v_0 is set to 1, else it's left to 0. This is done for all four vertices which results in a 4-bit status. This means there are in total 16 different cases in which the isoline can run through a cell.

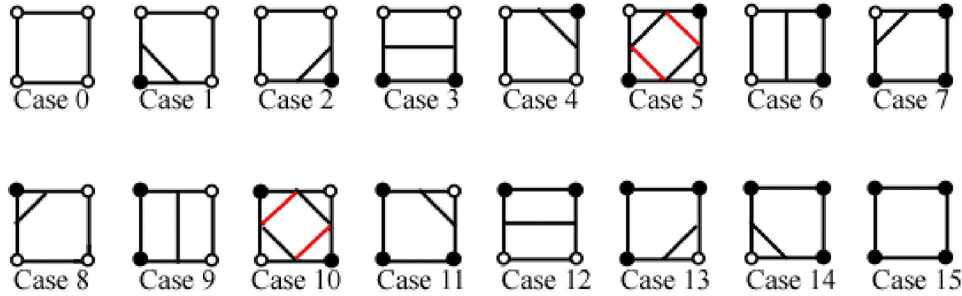

Figure 8: The 16 marching square cases

Above the image (figure 8) shows the 16 different cases of the marching squares algorithm. A white vertex indicates that the vertex is outside the isosurface and black indicates it is inside the isosurface. Every inside or outside case has its counterpart. Since some cases are equal, they can be reduced to 8. In case 0 and 15 for example, no lines have to be rendered, yet they are both different cases.

Cases 5 and 10 are also both ambiguous cases as becomes clear from the next image (figure 9).

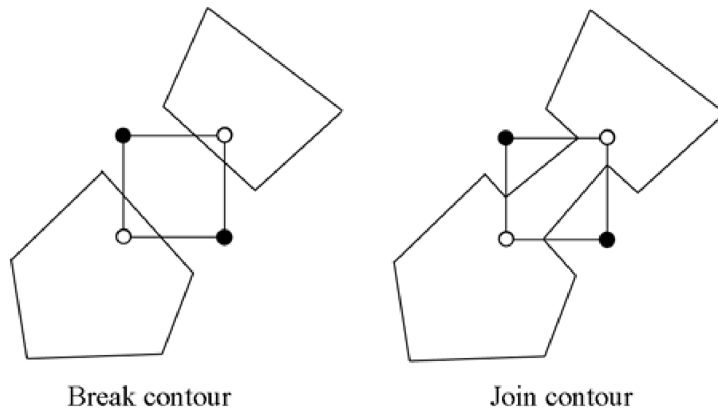

Figure 9: Two ambiguous cases in the marching squares algorithm

6.3 Difficulties

The contouring algorithm is very simple to implement following the instructions of this previous method. Initially the contour was not very smooth.

It was jaggy and had sharp edges everywhere. There was no smooth curve in the contours anywhere. The problem laid within the intersection with the isovalue. After reverting to this method, rather then something self-constructed, the isolines where nice and smooth.

Chapter 7

Height plots

Height plots are a very neat visualization technique. What are height plots, what are they used for and how where they implemented?

7.1 Description

All the previous discussed methods are implemented on a 2D grid. But sometimes it's also useful to see the value not by its color but by the height of the surface. This is exactly what a height plot does, it maps the values onto the Z-axis. Since the simulation uses a top down view, not much can be seen. Even with lightening and shadows it still left to be desired.

By rotating the field the height plots become much more visible. Rotation is achieved by dragging the mouse over the screen while holding the middle mouse button or scroll wheel, this however only controls the yaw and the pitch. Rolling can easily be done by using the shift key in addition to the mouse button. The shift key can also be used on the left mouse button to move the field up and down, left and right. This all becomes helpful when using the scroll when to zoom in and out.

7.2 Implementation

Realizing height plots would not be very visible from the default view, rotation, zoom and movement was implemented first. After having that ability height plots were constructed.

Since vertex already had x and y-coordinates, all that was needed was to add a z-coordinate. An extra array to track the height of each vertex was used, as the original simulation and its data-structures should not be disturbed. As value for the height any of the primary datasets can be chosen. When drawing a vertex the stored height from the array is then rendered using that value for the z-coordinate.
To see improve the perception of depth in the simulation some ambient and diffuse light was added. In order to let the light have effect on the surface also the normal vector was being calculated and stored for each vertex.

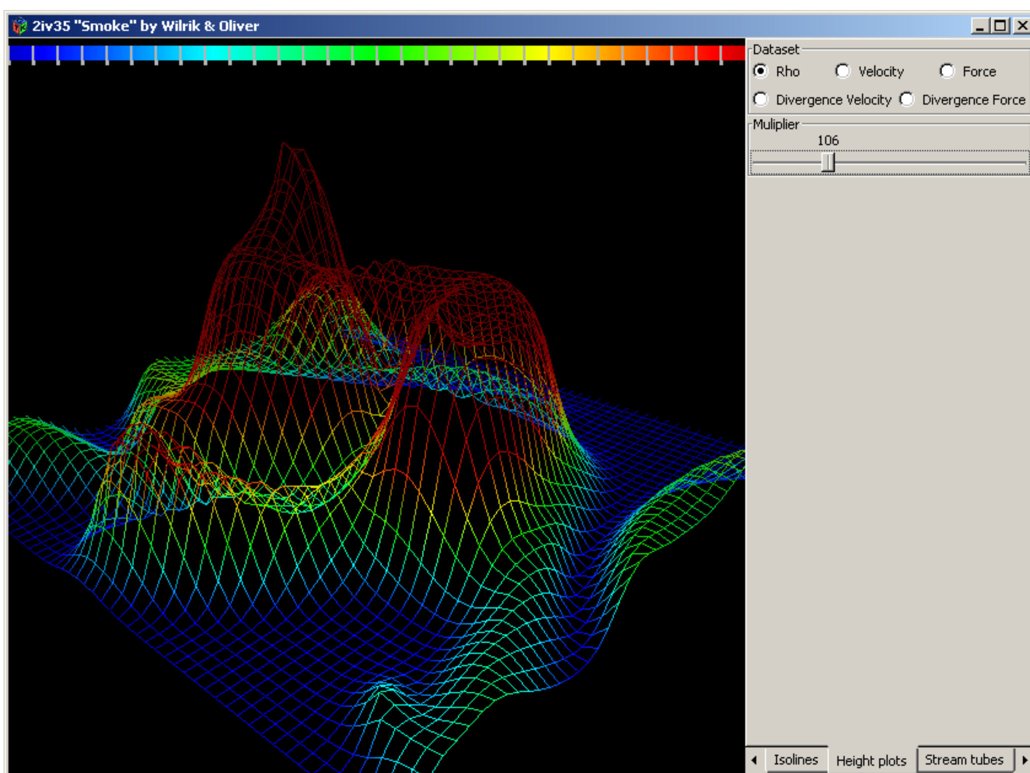

Figure 10: A height plot of the density

The above image (figure 10) shows a height plot of a fluid where the height translates to the density of the fluid. The colormap is also mapped to the density but this does not necessarily have to be the case. In figure 10, the height is visualizing the velocity of the fluid while the colormap shows differences in the density.

It can be seen in the second height plot, the bright yellow colors are not on the highest point of the height map. This means the density is not always the largest where the velocity is high.

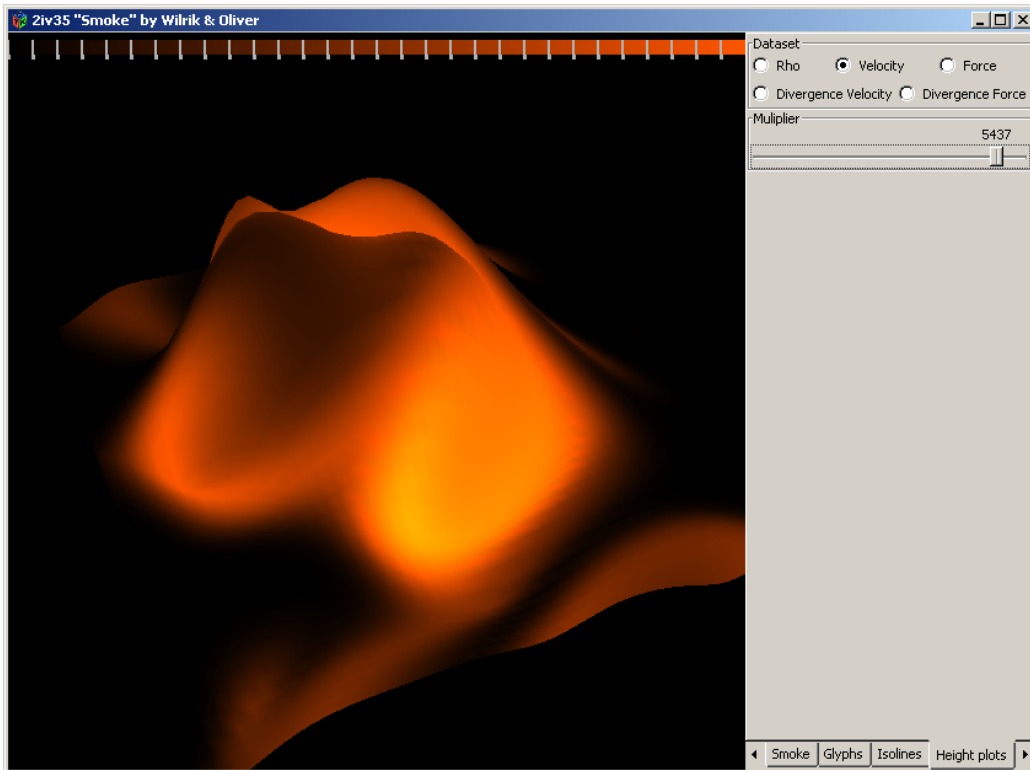

Figure 11: Velocity height plot

7.2.1 Normal vector

To calculate the normal vector for a given vertex, say v_0, firstly the two neighboring vertices in that cell are taken, v_1 and v_2. Then the difference of v_0 with v_1 and of v_1 with v_2 is taken. Lastly the cross product of these two differences is normalized to yield the result. This gives the normal vector at v_0.

OpenGL uses this vector to calculate the angle between the normal and the light source that is shining on the vertex. This angle is then used to shade a vertex, and with that an entire surface.

7.3 Difficulties

Some small issues throughout the implementation of this assignment arose, but eventually where implement correctly.

7.3.1 Rotation

The first problem encountered was due to the rotation of the field. Rotating around one axis was not to difficult. When trying to rotate the field around two or three axes, the simulation would not rotate around its center any more.

As it turns out the translation and rotation was in the opposite order. After changing the order of rotation and translation the simulation turned properly around the center.

7.3.2 Height strips

When first implementing the height plots a third dimension was simply added when rendering the triangle strips for the smoke visualization. This had as an effect that the different rows of each of the triangle strips in the simulation where not connected. In stead of a height plots, height strips were implemented.

This problem was countered first calculating the height for each vertex and storing it in an array. When rendering a vertex, the correct value is then retrieved from the array to correctly display the height.

7.3.3 Calculating the normal vector

When enabling the light in our program and adding the normal to each vertex there where some strange result. Because of the fact that the left lower neighbor was not consistently evaluated to calculate the normal vector. As a result each vertex was randomly pointing up or downwards, resulting in something that could be best explained as a checkers board.

When rendering the normal vectors the cause of the problem became immediately visible and it was quickly resolved.

Chapter 8

Streamtubes

The last of the non-optional assignments, just as the height plots, a 3-dimensional assignment. Streamlines and streamtubes can also however be used in 2D grids if needed.

8.1 Description

Imagine dropping of bit of ink into the fluid and let it flow for a while. This will result in a line that shows the path a ink particle has taken. Such a path is called a streamline. A streamtube is a 3D variant of a streamline. It consists of a number of consecutive tubes that together form a thick 3-dimensional tube.

Each streamtube has a beginning and an end. The starting point of a streamtube is called a seedpoint. From this seedpoint the tube will begin flowing with the fluid. It ends after a number of frames.

The implemented streamtubes initially were not actually flowing trough the fluid but through a frame history. Each frame's velocity or force vector components are stored in a history array and the streamtubes take a path trough this history. The 2D grid became a 3D grid with this new feature.

8.2 Implementation

Firstly the seedpoint placement was implemented. The user is able to pick an exact position for a seedpoint by with the mouse. A seedpoint is rendered

as a small sphere. From such a seedpoint either a streamline or streamtube is rendered. This is shown on figure 12.

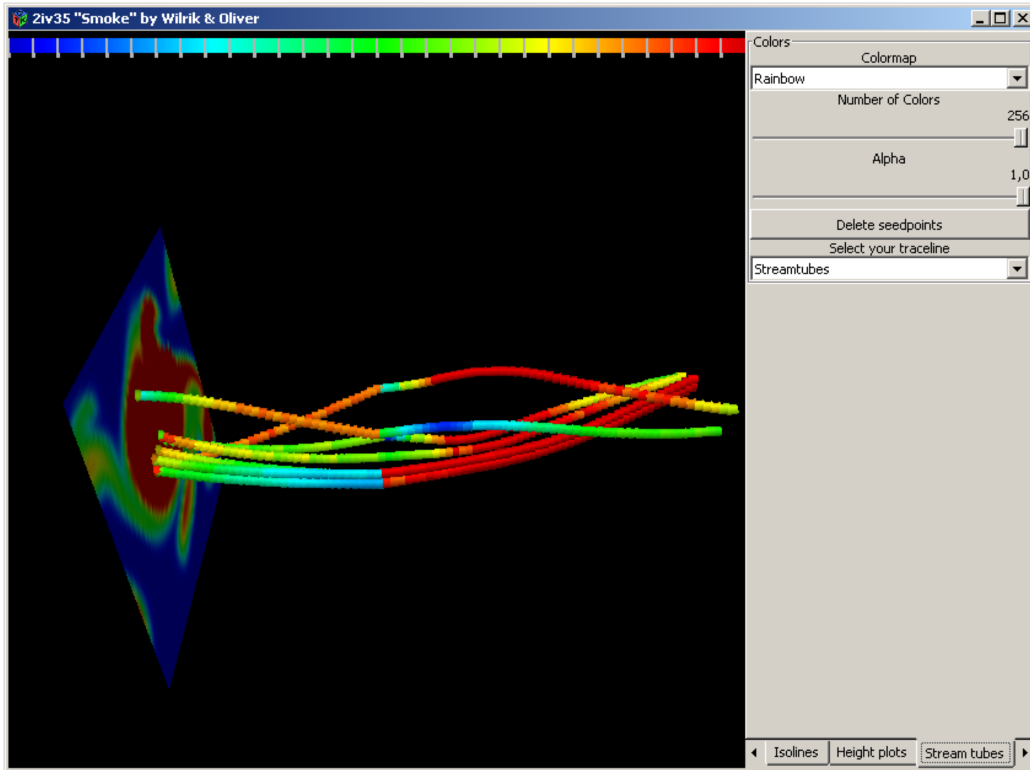

Figure 12: Eight streamtubes with a rainbow colormap

The simulation also has a history array that stores up to 80 frames. The user can go back and forth in this frame history or render all 80 frames at once to get some sort of volumetric effect of the smoke. This effect is shown in the image below (figure 13).

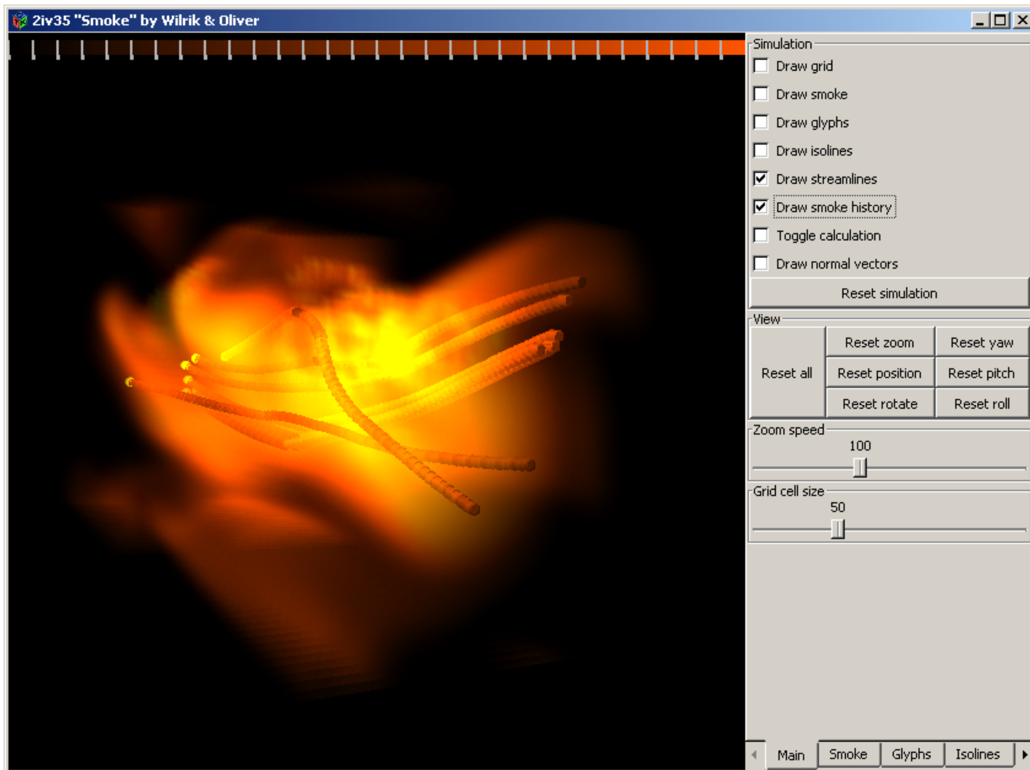

Figure 13: volumetric render of the smoke

8.3 Difficulties

Since time was becoming an issue, the implementation of streamtubes might not be fully correct.

Chapter 9

Conclusion

A lot of different visualization methods, algorithm, techniques, tips and tricks during the entire course where observed. With great enthusiasm this course was participated.

Bellow are some conclusions and recommendations about visualization and this course.

9.1 Visualization

When starting this course it was unknown what visualization was truly about. Its not about drawing nice pictures, but the entire process from getting the measured data up until the rendering process; the entire visualization pipeline.

In essence, visualization itself is a field in which many specializations of other fields are used. Physics, mathematics, computer science, computer graphics and probably many more can be thought of.

A lot of research is still being done. The translation from 3D to 2D brings difficulties. Some information is hard to visualize in two and/or three dimensions. The bigger glyphs get, the more they overlap. If you use blending, some details are lost but some new information may be gained. These examples are only a two of many more tradeoffs when visualizing large datasets.

9.2 The course

The visualization course encourages to explore the given assignments in different directions. It is stated that visualization is much about designing creative solutions to a given problem to gain different insights. And this turned out to be a positive approach for some of the assignments.

It turned out to be a positive approach to <u>some</u> of the assignments. For some other problems the insight of an expert was needed. It was therefore very useful to be able to send an e-mail for some of the troubles that arisen during the implementation.

The two practical hours after the theoretical part were a bit to crowded for the teacher however. He was unable to handle all questions from the students. This was a bit of a downside. Because of this feedback on the progress and problems were not always available during the hour.