Visualization 2IV35

Oliver Schinagl (...)
Wilrik De Loose (0601583)

January 10, 2008

Contents

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 distributed to implement such techniques on top of the existing code.

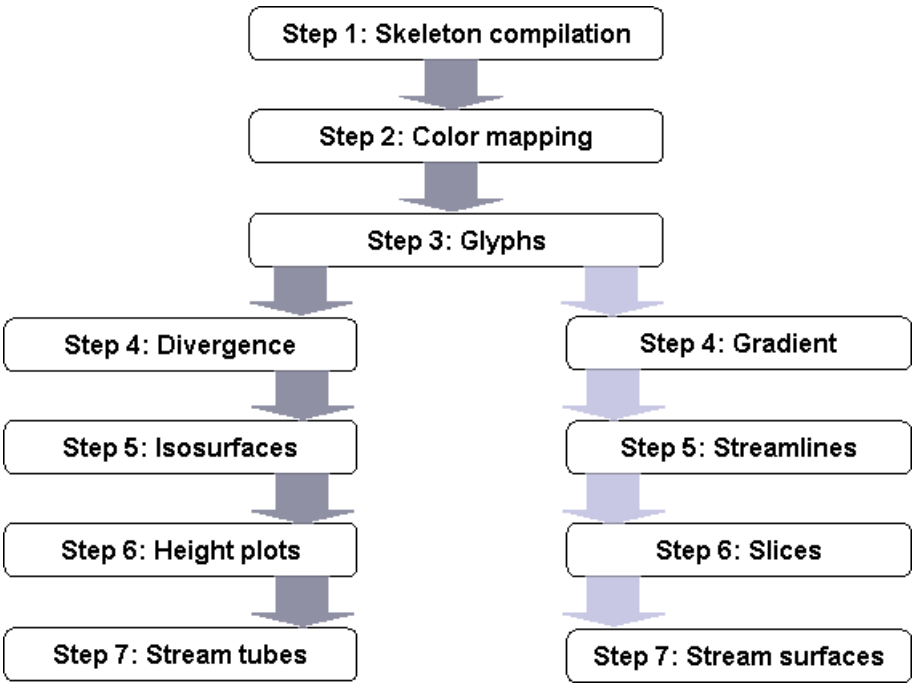

Figure 1: Practical assignment

The above figure illustrates the different assignments. After doing step 3 there were two different ways to continue the practical assignment. We chose to take the left branch of the figure which, amongst others, involved imple-

menting isosurfaces and hight plots.

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

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

We end our report 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 hight 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. That 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 of what we were doing, we created a file for every new technique we implemented. The second thing we did is we split up the calculations, the rendering functions and user interactions from each other.

With this new structure we were able to manage the code and add additional functionality. We chose C as our programming language and used the GTK+ library to create the graphical user interface. We also used SVN to manage our project. This was very useful since we both had different development environments and also did a lot of work at home.

Chapter 3

Color mapping

In chapter 2 we saw a figure (figure 2) which showed a fluid in motion. 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. We already 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 we saw the smoke using a grey scaled colormap. If we know that the values at the vertices ranges from 0 to 1, we can use the value 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, we added a legend at the top of the screen. 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.

You are also able to set the number of used colors. At default this value is set to 256 colors, but you can easily set that amount to 16 using a slider. You will see big bands of colors appear. This way, the line between certain values becomes more visible.

3.2 Implementation

We've implemented two actual colormaps and three which only contain one RGB color, red, green or blue (useful for isolines for instance). The "Wilrik" colormap implements a fire elemental color scheme. The other one is called "Oliver" and is a repeated band of colors which can show quite well where the fluid's in motion.

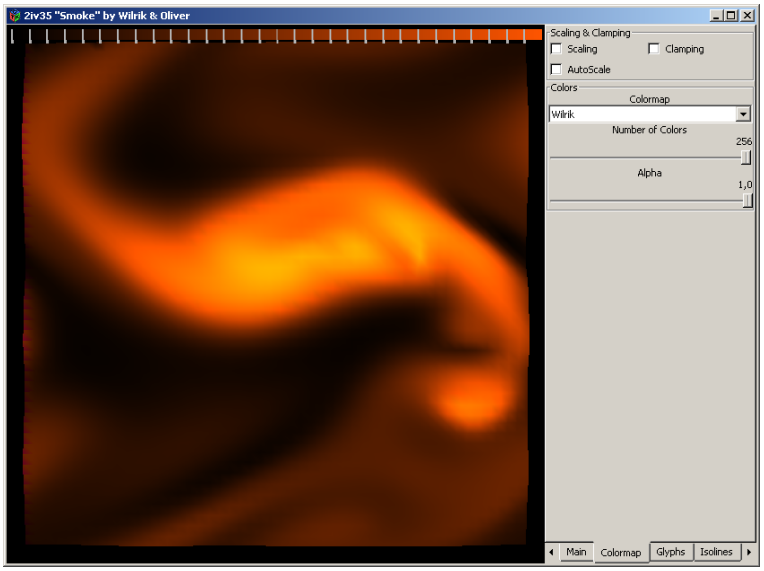

Figure 3: A fire colormap

The fire color is determined as follows:

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

So, 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, almost yellow colormap.

To create a repeating band of colors, we used to following definition:

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

So we first multiply the value with 100, cast it to an integer and take that value modulo 10. We then take some colors from a look-up-table to pick the color for the band.

3.3 Difficulties

The last mechanisms to implement for this assignment were scaling and clamping. With clamping you let the user set a minimum and maximum for the values. This is done by first enabling clamping and then click somewhere in the bottom of the color legend and drag the minimum and maximum clamping indictors. Actual data lower then the minimum or higher than the maximum are set to the maximum or minimum respectively.

The (auto)scaling mechanism was a bit more subtle. When enabled, you can set the scaling in the same way as the clamping. For the auto-scaling, the minimum and maximum values are stored and the entire dataset at the current time moment is mapped to the visible colormap. This is not so hard to do, but we had not foreseen that values could also be negative. It wasn't until we implemented the divergence that we found this problem.

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 by means of several graphical icons, such as arrows."

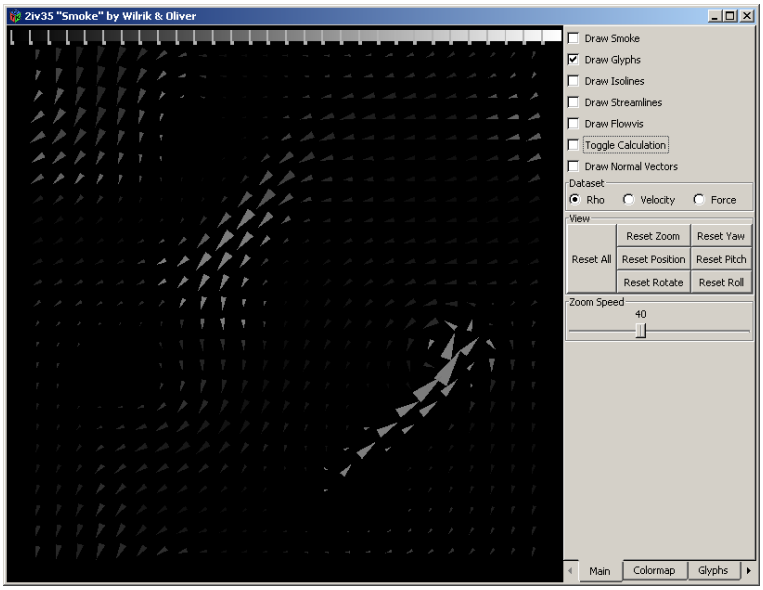

Figure 4: Triangle glyphs with a grey scaled colormap

What does it mean? The smoke in our 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 you draw this vector for each vertex you get a vector field that shows the direction and speed of the fluid at each vertex.

In the figure above (figure 4) you see triangle glyphs with a grey scaled colormap. In the lower right region you can see that the fluid is moving upwards where at the rest of the simulation the fluid is trying to get to the bottom of the screen.

4.2 Implementation

We implemented four additional glyphs besides the already implemented hedgehogs. We implemented triangles (see figure 4), 3D cones, 32x32 bitmap image glyphs (see figure 5) and s second image glyphs. For every glyph we go through a set of steps.

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

The length is calculated using the following equation of Pythagoras:

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

The angle Θ of the glyph is calculated using the inner product ($inprod$). The inner product is used to calculate the glyphs angle:

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

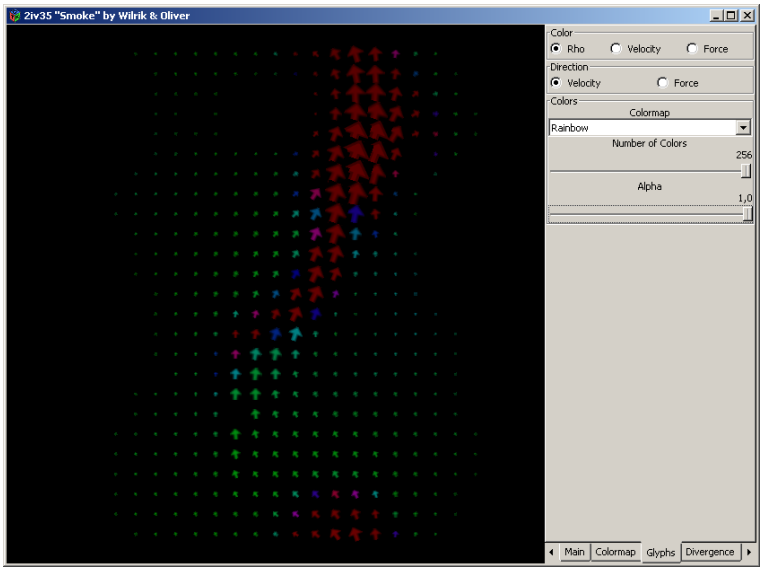

Figure 5: Image glyphs colored with a rainbow colormap

The second part of this assignment was to let the user be able to choose an alternative resolution for the sample grid. The default grid resolution is 50x50. The user is now able to alter that resolution. We use the value of the nearest-neighbor for the glyph to visualize it.

4.3 Difficulties

We had some troubles with calculating the angle of the glyph. For some reason the calculation checks the smallest angle which is between 0° and 180°. When we located this irregularity it was easily fixed.

4.4 Quake root

A nice side note to this chapter is that we don't use the default $sqrt(float)$ function that the C-library offers to determine the length of a vector for instance. We use a function that we like to call the "Quake Root" from the game Quake 3.

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

accuracy. The really interesting aspect of this function is 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's usually implemented using the FSQRT assembly instruction!

You can read more about the magical square root 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

The 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 simple, once you get it. For the divergence of the velocity it 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. We also implemented the divergence of the force field with the same formula.

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 and low values behind it.

5.3 Difficulties

Although we have a good definition of what the divergence is and what the result should look like, it is rather difficult to verify it using the simulation. This is a know difficulty. The divergence can 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.

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 sort to speak follows a given value. If the values of a field ranges from 0 to 1, a good threshold would be 0.6 for instance. The isoline visualizes the points that equal the value of 0.6.

Our program is able to define the number of contour-lines and the minimum and maximum values in between the isolines are rendered.

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);
}

So 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 in a bit.

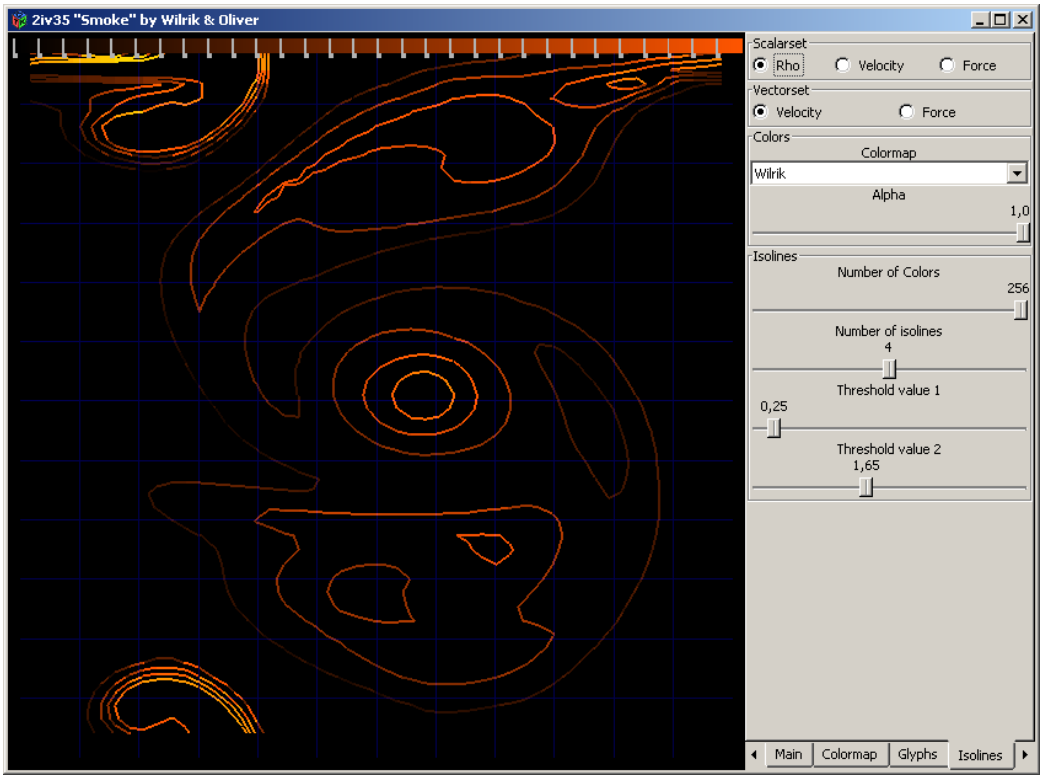

Figure 6: Three fire-like colored isolines on a blue grid

If a vertex, say 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.

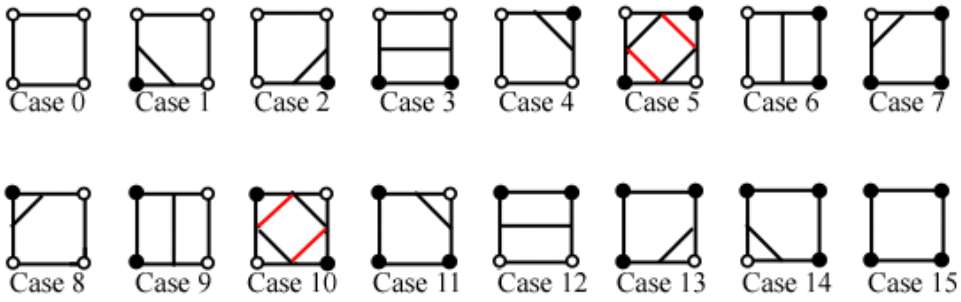

Figure 7: The 16 marching square cases

The above image (figure 7) 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 it's counterpart. So we've reduced the number of cases down to 8. In case 0 and 15 for instance, no lines have to be rendered, yet they are both very different cases.

The cases 5 and 10 are both ambiguous cases as becomes clear from the next image (figure 8).

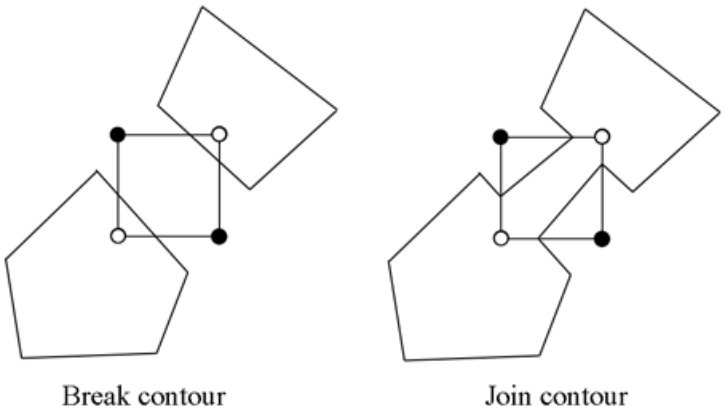

Figure 8: Two ambiguous cases in the marching squares algorithm

6.3 Difficulties

The contouring algorithm is very simple to implement. Just follow the instructions of the method. Still our contour was not very smooth. It had

sharp edges everywhere and there was no smooth curve in the contour. The problem was with the intersection with the isovalue. We did'nt exactly follow the equation that was in the study material. When we reverted to that equation the isolines where nice and smooth.

Chapter 7

Height plots

This chapter treats a very neat visualization technique, namely the height plots. What are height plots, what are they used for and how did we implement them?

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 it's color but by the height of the surface. This is exactly what a height plot does. It maps the values onto the Z-axis. This means that, initially, you won't see much differences between the version with and without the high plots implemented. This is because of the height that is being drawn onto the Z-axis that runs towards the screen.

To make the height more visible, we implemented a method to rotate the field by dragging the mouse over the screen while holding the middle mouse button. This way you can see the height of the field.

7.2 Implementation

First of all we implemented the method to rotate the field by mouse. Without this useful functionality, the height would'nt be that visible. After we implemented this feature we added the height to the program.

We created the height by adding an extra Z-coordinate to each value. We created an extra array that would keep track of the height of each vertex.

When drawing a vertex we used the stored height from the array and render the vertex using that value for the Z-coordinate.

To see more depth in the picture we also included some ambient and diffuse light to the program. In order to let the light have effect on the surface we calculated the normal vector for each vertex.

7.2.1 Normal vector

For the calculation of the normal vector for a given vertex, say v_0, we first take the two neighboring vertices in that cell, v_1 and v_2. We take the difference of v_0 with v_1 and of v_1 with v_2. Then we take the cross product of these two differences and normalize the result. This gives us the normal vector at v_0.

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

7.3 Difficulties

We had some small issues during the implementation of this entire assignment but we managed to implement them correctly.

7.3.1 Rotation

The first problem we encountered was due to the rotation of the field. Rotating around one axis was not to difficult. When we tried to rotate the field around two or three axis', the simulation wasn't rotating around it's center any more.

It turned out we were translating and rotating in the opposite order. When we changed the order of rotating and translating the simulation rotated around it's center.

7.3.2 Height strips

When we first tried to implement the height plots we simply added a third dimension when rendering the triangle strips for the smoke visualization. This had as an effect that the different rows of the simulation where'nt connected. In stead of a height plots we implemented height strips.

We countered this problem by first calculation the height for each vertex. When rendering a vertex, we look up the correct value for the height.

7.3.3 Calculating the normal

When enabling the light in our program and adding the normal to each vertex we had some crazy outcome. This was because of the fact we did'nt consistently took the left and lower neighbor of a vertex to calculate the normal vector. As a result each vertex was randomly pointing up or downwards. This looked like some sort of checkers board.

We rendered the normal vectors and immediately saw the cause of the problem. It was simply solved by always taking the same neighboring vertices.

Chapter 8

Streamtubes

The last of the non-optional assignments. This is, just as the height plots, a 3-dimensional assignment although streamlines and streamtubes can be used in 2D grids.

8.1 Description

Imagine you drop of bit of ink into the fluid and let it flow for a while. What you will get is 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. A streamtube consists of a number of consecutive tubes that together form a thick 3-dimensional flexible tube.

Each streamtube has a begin and end. The begin point of a streamtube is called a seedpoint. From a seed point the tube will begin flowing with the fluid. It ends after a number of frames, seconds or parts. This can be variable.

The streamtubes we had to implement were'nt actually flowing trough the fluid but through a frame history. Each frame is stored in a history array and the streamtubes take a path trough history. The 2D grid became a 3D grid with this new feature.

8.2 Implementation

First we implemented the placement of the streamtubes. We chose to let the user be able to point the exact position of each seedpoint by mouse. A

seedpoint is rendered as a small sphere.

We also created a history array that stores up to 50 frames.

8.3 Difficulties

We did'nt have enough time to actually trace a streamtube.

Chapter 9

Conclusion

We saw a lot of different visualization methods, algorithm, techniques, tips and tricks during the entire course. We participated with great enthusiasm and look back to a successful semester.

We also have some conclusions and recommendations about visualization and this course.

9.1 Visualization

When we first started this course we did'nt know what visualization was truly about. It's not about drawing nice pictures, but the entire process from getting the measured data up until the rendering process; the entire visualization pipeline.

So, visualization itself is a field in which many specializations of other fields are used. Think of physics, mathematics, computer science, computer graphics and probably many more.

A lot of research is still being done. The translation from 3D to 2D brings difficulties. Some information is hard to visualize in 2 and/or 3 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 us to some of the assignments.

We say it turned out to be a positive approach to some of the assignments. For some other problems we needed the insight of an expert. We found it very useful to be able to mail some of the troubles we had during the implementation.

The two practical hours after the theoretical part were a bit to crowded for the teacher to be able to handle all questions from the students. This was a bit of a downside. Because of this we were'nt always able to show our new results or post question during the practical hour.

Chapter 10

Functionality

This final chapter lists all functionalities of our visualization program.

10.1 Simulation

Toggle grid on/off
Check the "Draw grid" box on the "Main" tab to draw a grid.

Toggle smoke on/off
Uncheck the "Draw smoke" box on the "Main" tab to disable the smoke.

Toggle glyphs on/off
Check the "Draw glyphs" box on the "Main" tab to enable glyphs.

Toggle isolines on/off
Check the "Draw isolines" box on the "Main" tab to enable isolines.

Toggle streamlines on/off
Check the "Draw streamlines" box on the "Main" tab to enable streamlines.

Toggle calculation
Uncheck the "Toggle calculation" box on the "Main" tab to pause the smoke simulation.

Toggle normal vectors on/off
Check the "Toggle normal vectors" box on the "Main" tab to render the normal vectors.

Reset simulation
Push the "Reset simulation" button on the "Main" tab to reset the smoke's attributes.

Change smoke dataset
Select either the density rho, velocity, force or the divergence to change the smoke's dataset.

10.2 Camera position

Reset all
Push the "Reset all" button the "Main" tab to reset the rotation, translation and zoom to its defaults.