Project “Tiefsee”

Optical detection and visualization of diffuse vent in the deep sea

Moritz Bergenthal Enna Gerhard Arkadiusz Guzinski
Timo Hoheisel Jannes Knychalla Jan Lemme
Kilian Lüdemann Niklas Masemann Dennis Riemer
Kevin Ross Sheyla Saul Tjorven Schnack Clemens Wodtke

June 3, 2020

Contents

Abstract

Through diffuse hydrothermal seafloor vents water flows out of the seafloor. Though the inflowing water is transparent, it often has a different temperature or different dissolved substances than the surrounding water. This leads to a difference in the refractive indices between the various water types. This causes an optical effect, called Schlieren. Therefore, Schlieren are a way to visually detect seafloor vents. We created a software for optical detection of Schlieren. The software is based on the Horn-Schunck algorithm. It is capable of highlighting Schlieren in a video and to process live stream video data. To provide test data for the detection software a raytracer was developed, where rays get refracted based on a data from a Computational fluid dynamics (CFD) simulation. The raytracer supports 2D- as well as 3D-simulation data. The underlying CFD simulation software is Ansys. Additionally, an experiment was conducted in a tank. Its footage was used to evaluate the simulation software through side by side comparison and provide additional test data for the detection software. The detection software performed well on the footage from the simulation and the experiment, as well as on real deep sea footage with little noise and movement. Though there are still issues with the robustness of the results and with camera movements.

1 Introduction and Motivation

This paper is a result of the student project "Projekt Tiefsee", a project worked on by computer science students at the University of Bremen. The team working on "Projekt Tiefsee" consists of 13 students and is supervised by two professors, Ralf Bachmayer and Gabriel Zachmann.

Low temperature diffuse flow contributes to approximately more than half of the heat emitted around hydrothermal vents in the deep sea. However, as it is difficult to determine where exactly diffuse flow is happening, research on those have been relatively sparse (Bemis, Lowell, and Farough, 2012). Most research is concentrating on focused flow, because those are easy to spot and measure. Our main goal is the optical detection and visualization of diffuse vent in deep sea. This can aid further deep sea research and running missions with Remotely operated underwater vehicles (ROVs), allowing researchers to inspect diffuse vents discovered by the detection software.

To reach this goal, we formed three subgroups. The group "Experiment" has worked on recreating deep sea vents in a controlled lab environment described in Section 3: Experimental recreation. The group "Simulation" has worked on recreating deep sea vents digitally using simulation tools and rendering software, see Section 4: Simulation. Finally, the group "Detektion" has worked on creating a tool to analyze video material of deep sea exploration and a detection algorithm. Video footage from previous missions as well as material created by the other two subgroups was used during the development. Their work is explained in Section 5: Optical detection.

2 Background

Hydrothermal vents

There are several types of hydrothermal vents. Black smokers are typically created by a concentrated flow of water up to 450 °C and transport sulfides and other minerals. When these sulfides contact the surrounding cold water they precipitate, creating the signature black chimney like structures. White smokers are still focused, yet they typically have lower temperatures and consist of different materials (Schön, 1999).

In comparison, diffuse vents have lower flow rates and are distributed across much larger areas. They can have temperatures between less than 1 °C and up to 80 °C. Mixing with the surrounding water may also play a role in its lower temperatures (Bemis, Lowell, and Farough, 2012).

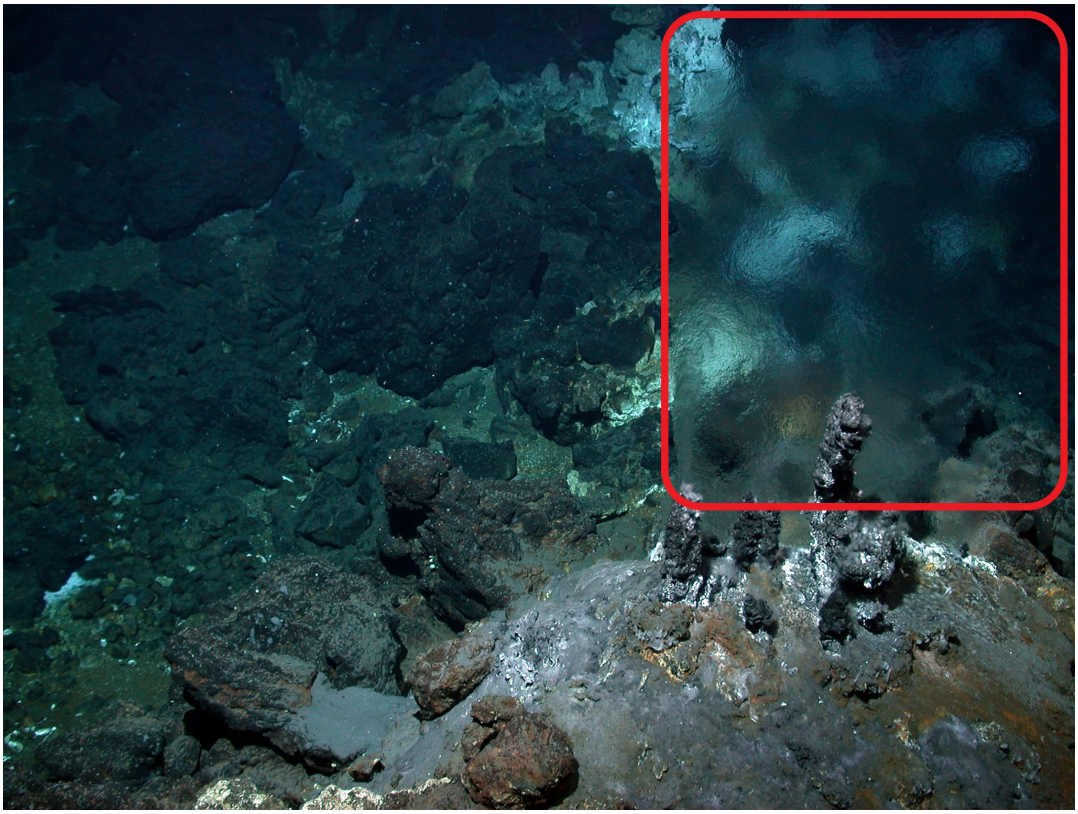


Figure 1: Example of diffuse/hydrothermal seafloor vent and the caused Schliere, highlighted by a red box (Center for Marine Environmental Sciences (MARUM), University of Bremen, CC-BY 4.0)

Schlieren

Schlieren are an effect occurring because of a difference or gradient in the refractive index of a transparent medium, the medium is optically inhomogeneous. This gradient has the effect of refracting light rays. Therefore, Schlieren are visible by small distortions on pictures. Commonly known examples of Schlieren are the flickering air above a hot street, fire or an engine shown in Figure 2.

Figure 2: NASA’s SR-71A aircraft taxiing on the ramp at NASA’s Dryden Flight Research Center, Edwards, California, heat waves from its engines blurring the hangars in the background (NASA Dryden Flight Research Center, 1995).

In the following Schlieren are regarded in the context of diffuse hydrothermal seafloor vent because Schlieren are an optical indicator for these vents and hence can be used to detect them. An example of a schliere caused by a diffuse/hydrothermal seafloor vent is shown in Figure 1.

3 Experimental recreation

3.1 Introduction

In this section the efforts to recreate hydrothermal vents and the typical Schlieren effect in an experiment and creating videos with those are described. Everything reported in this section was conducted by the subgroup "Experiment", consisting of Moritz Bergenthal, Enna Gerhard, Jan Lemme and Tjorven Schnack.

The ultimate goal was to create videos of optical effects occuring while artificially creating Schlieren, which could then be used for testing and verification of the detection software as well as testing and verification of the simulation software. This final realization is described in Section 3.4: Recreating hydrothermal vents.

The first goal was to create the Schlieren effect, which itself occurs when fluids or gas with have inhomogeneities in their refractive indices and is described in Section 2: Schlieren. Therefore, the objective in the first test series was to find a simple and repeatable method to create different refractive indices in water. Those tests are further described in Section 3.2: Visualization of different refractive indices.

In parallel, a few different inflow systems for inserting fluids into an aquarium were developed and tested. Those efforts are further described in Section 3.3: Creating diffuse water vents.

3.2 Visualization of different refractive indices

To achieve different refractive indices the decision to not necessarily insert water with different properties than the water in an aquarium was made. In this test series, different solid, but transparent materials were used as well as warm and later on salt water. The test series can be seen as a prototype for later series. Before conducting large scale experiments, it was decided to evaluate different techniques and approaches in a smaller test environment.

3.2.1 Test setups

Instead of using a large tank, the first two test series were conducted inside of a smaller aquarium. The main advantages include a much easier access with being able to reach the ground, requiring less water. The aquarium used in the first and second test series had the dimensions $40\,\text{cm} \times 24\,\text{cm}$. A larger container with the dimensions $120\,\text{cm} \times 100\,\text{cm} \times 76\,\text{cm}$ was used for the third test series and the final experiment. This improved results drastically as it reduced side effects such as affecting circulation inside of the container, and an influx of salt water or different temperatures would not be as noticeable. Overall, an even larger tank might result in further improvements, however the results in a larger tank were already satisfying.

We evaluated two different background images during the first two test series, one with a checkerboard and one with a photograph of gravel to create a more natural effect. During the

third series, we explored plain backgrounds of black and gray color. We used a grid of black lines on a white background for the fourth series to create higher contrast.

Video capturing was performed by a *GoPro Hero 5* and *GoPro Hero 6* camera in 1080p with 60 Frames per Second (FPS) for the first and fourth test series respectively. In the second, it was substituted with the camera of a *Motorola Moto G6*. It is a good idea to have an alternative camera system to be able to account for the equipment suddenly being unavailable. For the third test series, we evaluated the camera of a *BlueROV*. This was insufficient because it created too many artifacts and was not focused well. Finally, we co-recorded the third and fourth experiment with a *Rollei 400* in 1080p with 30 FPS.

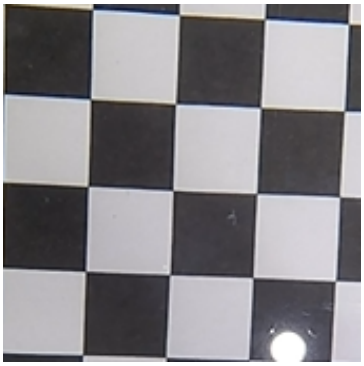
Starting with the second experiment, we covered the aquarium to reduce light from the room as well as noise and reflections which improved video quality significantly.

We tried using a green cross line laser during the third test series, but it did not have a visible effect on the appearance of the Schlieren.

3.2.2 Different Materials

We explored different materials during the first three test series. The following is an account on the process of selecting the most prospective candidates.

A wide selection of prospective materials and shapes of objects that could be used was evaluated during the first test series. The result was a set of seven candidates to create different refractive indices. Polyurethane (PU) was used in a 3D printer to create a drop, wave and lens.

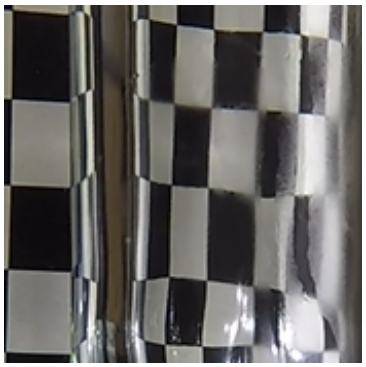


(a) Empty aquarium for comparison

(b) PU drop

(c) PU Wave

(d) PU Lens



(e) Empty bottle

(f) Filled bottle

(g) Water, 50 °C

(h) Oil

Figure 3: Results of the first test series

ID	Material	Outcome	Figure
1.0	None	Empty aquarium for comparison	Figure 3 (a)
1.1	PU Drop	Fairly opaque, some lens effect	Figure 3 (b)
1.2	PU Wave	Shadow appearing behind the sample	Figure 3 (c)
1.3	PU Lens	Nearly invisible, lens effect	Figure 3 (d)
1.4	Empty Bottle	Lens effect	Figure 3 (e)
1.5	Filled Bottle	Lens effect	Figure 3 (f)
1.6	Water 50 °C	Diffuse distortion and flickering	Figure 3 (g)
1.7	Oil	Sticking to the tube, rising quickly	Figure 3 (h)

Table 1: Results of the first test series

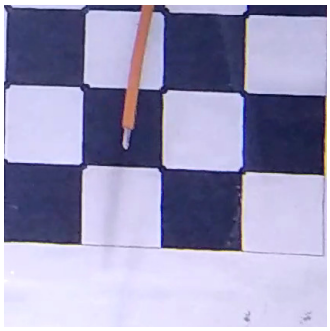
Hot water clearly showed the best results with realistic diffuse distortions and flickering. We were not able to create realistic movement with probes out of Polyurethane and the type and shape of distortion was very different. Oil did not lead to well visible effects, especially as the bubbles surfaced very quickly. Furthermore, it was much harder to handle and to clean up.

With hot water having been the best medium of the first test series by far, we decided to focus on different types of water inflow for the second series. This included further tests with hot water, cold water created by ice as well as salt water.

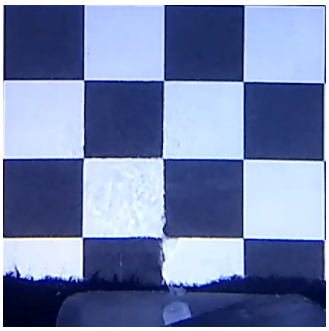
The salt water was created using completely dissolved salt. This only consisted of regular cooking salt (NaCl).

(a) Empty aquarium for comparison

(b) Empty aquarium with a gravel background, for comparison



(c) Warm water 48 °C (with thermometer)



(d) Warm water 60 °C focused from below

(e) Ice Cube

(f) Ice Cubes

(g) Salt water, 240 ‰

(h) Salt water, 240 ‰, gravel background

(i) Salt water, 38 ‰

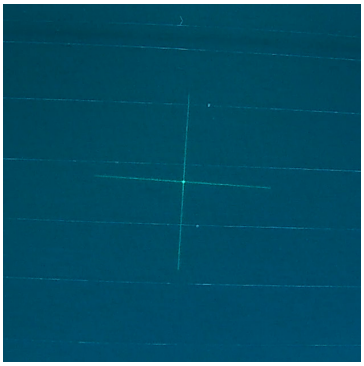
(j) Salt water, 19 ‰

Figure 4: Results of the second test series

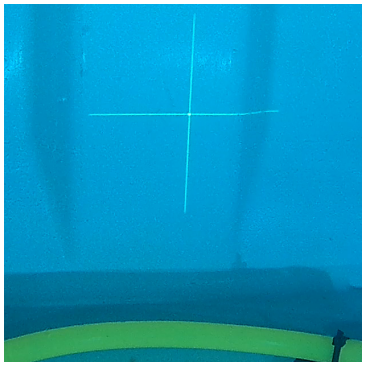
ID	Material	Outcome	Image
2.0	None	Empty aquarium for comparison	Figures 4 (a) and 4 (b)
2.1	Water 48 °C	Minor Schlieren visible	Figure 4 (c)
2.2	Water 60 °C	Schlieren clearly visible	Figure 4 (d)
2.3	Ice cube	No distortions visible	Figure 4 (e)
2.4	Ice cubes	No distortions visible	Figure 4 (f)
2.5	Salt water 240 ‰	High visibility, somewhat diffuse	Figures 4 (g) and 4 (h)
2.6	Salt water 38 ‰	More diffuse than 2.4	Figure 4 (i)
2.7	Salt water 19 ‰	Less visible, more diffuse than 2.5	Figure 4 (j)

Table 2: Results of the second test series

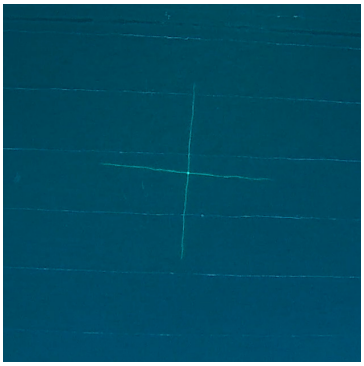
The second test series introduced salt water which offered good results. Salt water should be inserted with increasing salinity so there is less floating around at the bottom during consecutive tests. Hot water also produced some realistic Schlieren. Ice cubes were not feasible as they took too long to dissolve.



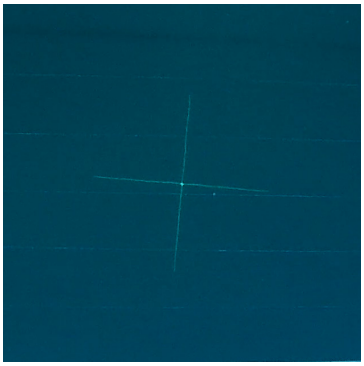
(a) Empty tank for comparison



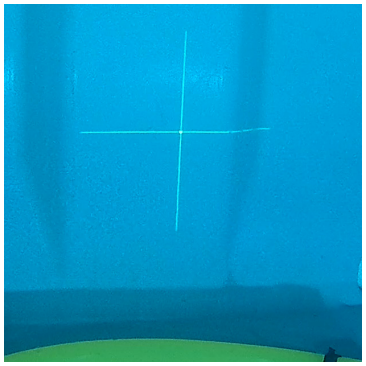
(b) Empty tank with a gray background, for comparison



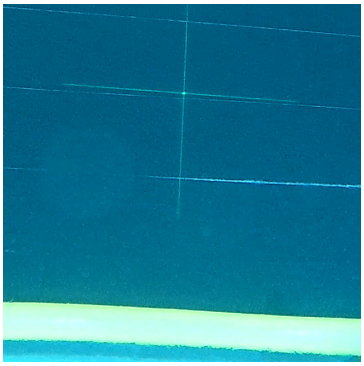
(c) Warm water



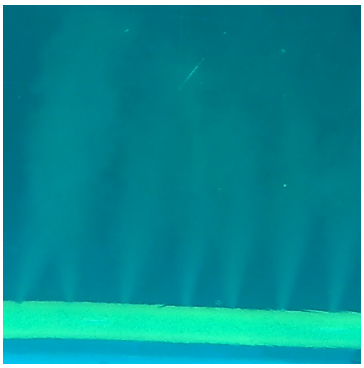
(d) Salt water, 10 ‰



(e) Salt water, 20 ‰



(f) Salt water, 40 ‰, gravel background



(g) Tinted water

Figure 5: Results of the third test series

ID	Material	Outcome	Image
3.0	None	Empty tank for comparison	Figures 5 (a) and 5 (b)
3.1	Warm water	Air caused distortions	Figure 5 (c)
3.2	Salt water 10 ‰	Barely visible	Figure 5 (d)
3.3	Salt water 20 ‰	Partially visible	Figure 5 (e)
3.4	Salt water 40 ‰	Some Schlieren visible besides the background	Figure 5 (f)
3.5	Tinted Water	Visible outflow of colored fluids	Figure 5 (g)

Table 3: Results of the third test series

The third test series mainly faced problems with the test setup. On the one hand the pump for the inflow system was very difficult to operate and on the other hand the camera system produced low quality and noisy videos. Therefore the results of this test series were not good and unusable. The only visible results was from water tinted with food coloring, though it was not realistic.

Overall salt and warm water have produced the best results throughout all test series.

3.3 Creating diffuse water vents

To insert water into an aquarium an inflow system was needed. These systems should produce a diffuse water vent that is as close as possible to actual hydrothermal vents in the deep sea.

An inflow system consists of two parts, the supply line, which has to apply water pressure, and the actual inlet were the fluid enters the aquarium. For applying pressure, two different electrical pumps as well as just potential energy though height difference were evaluated.

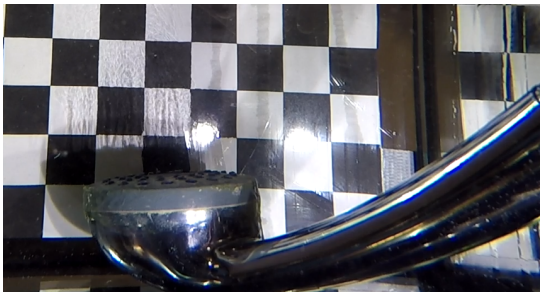
For the first experiment we chose a shower head and pumped hot water out of a bottle using height differences. (Figure 6 (a))

In the second experiment we arranged a meandering tube with regular drilled holes. (Figure 6 (b))

Due to insufficient water output through the small holes we switched to let the water flow out of the ending of the hose. (Figure 6 (c))

Based on this experience, we used a bigger hose with bigger holes lined up on the top in the third test series. (Figure 6 (d))

For the next and final experiment we printed an inlet with a 3D printer, which is described in the next section. This was the result the experience gained in the first three test series



(a) Shower head as an initial test candidate, first test series

(b) Small meandering tube to allow for a more precise outflow, second test series

(c) Tube ending accounting for increased water output, second test series

(d) Bigger tube with larger holes to sustain increased water output, third test series

Figure 6: Different types of inlets

3.3.1 3D printed inlet

For the final experiment, we decided to create an inlet distributing water instead of a single line over a larger area. This custom inlet was created out of Polylactide (PLA) using a 3D printer. To reduce the effort of recreating this by the subgroup “Simulation”, we decided to create a grid of 9 by 9 holes with a distance of 1 cm and a diameter of 2 mm. These were placed on a cuboid with the dimensions 10 cm × 10 cm × 2 cm. The top-layer has been printed separate, with the holes drilled in afterwards. The top-layer was glued to the cube, and a separate hole was drilled at one side to glue a hose connector for the inflow system tube in place.

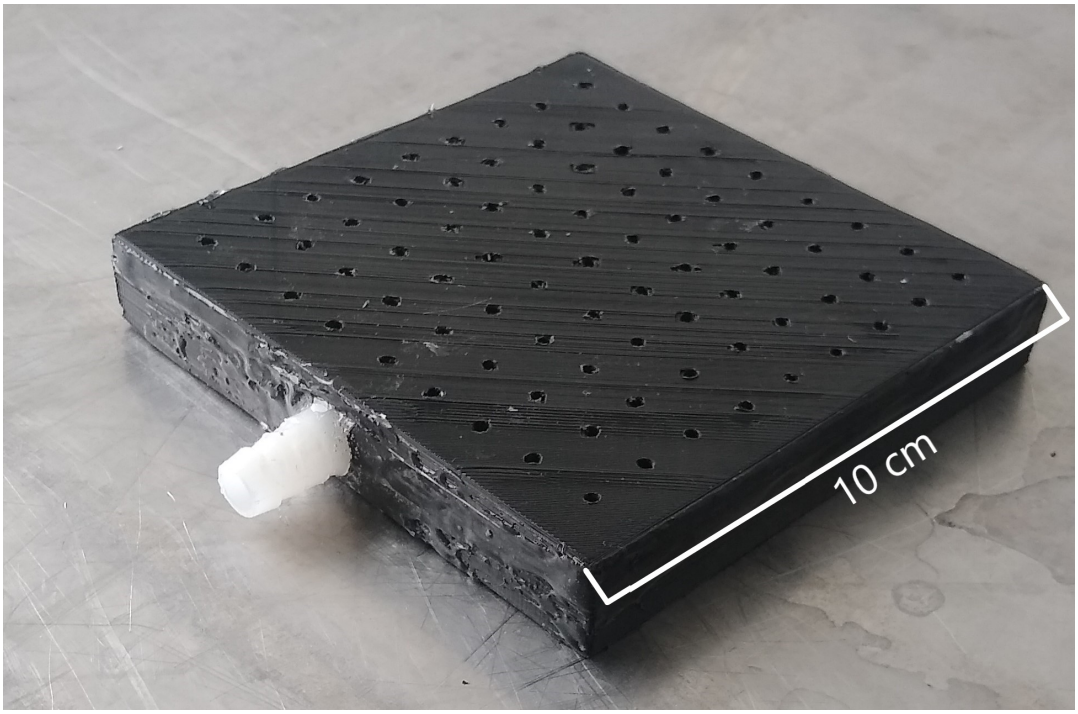


Figure 7: 3D printed inlet used in the final test

3.3.2 Inflow system

In the end, we decided to build our own gravity based inflow system as pumps were not producing a constant enough water stream and were harder to control. A wooden tower with a height of about 1.5 m forms the carrier for a canister. This canister lays with its outlet directed to the floor on top of the tower and held in place with 4 pieces of rigid tube, which are screwed to the sides of the tower. The rear top of the canister was cut open to refill water and replace water flowing out with air.

The outlet of the canister and the flow regulation ball valve are connected by a short piece of rigid tube which is hold in place by hose clamps. We connected a tube with a diameter of 8 mm and a length of about 2.5 m to the outlet of the ball valve, again hold in place by a hose clamp. This tube can be connected to the 3D printed inlet using its hose connector.

Figure 8: Wood tower and canister of the inflow system for the final test series

3.4 Recreating hydrothermal vents

With the lessons learned and the experience gained in the previous tests we conducted this final large scale experiment. It is the core experiment and based on its results we made the following evaluations.

3.4.1 Setup

We created the setup as depicted in Figure 9: Experiment setup based on the experiences gathered in the first three test series. The main change compared to the previous test series included the larger 3D printed inlet (Section 3.3.1: 3D printed inlet) and the use of the gravity based wooden tower inflow system (`b` in Figure 9; Section 3.3.2: Inflow system).

The inlet was placed inside a tank with the dimensions 120 cm × 100 cm × 76 cm. It was filled with tap water ($d = 38$cm).

We used a board with a 2 cm grid of 3 mm thick lines. Light was provided by a *Suptig LED* light (`y`) with about 500 lm to illuminate the inside of the tank. In order to be on the safe side, we used two cameras. A *GoPro Hero 6* (`x`) was placed in the center and a *Rollei 400* (`x'`) was placed at an angle to capture the vents.

The 3D printed inlet (`a`) was placed at a distance of $s = 20$cm from the background and 3.7 cm above the ground. The camera of the *GoPro* was $h = 7$cm above the ground. The light (`y`) was mounted directly on top of it, at a height of $l = 17$cm. The *Rollei* was inside of a case, that lifted it about $h' = 2$cm above the ground. After going through every test at a camera to inlet distance of $c = 75$cm once, where repeated the tests with a distance of $c = 45$cm.

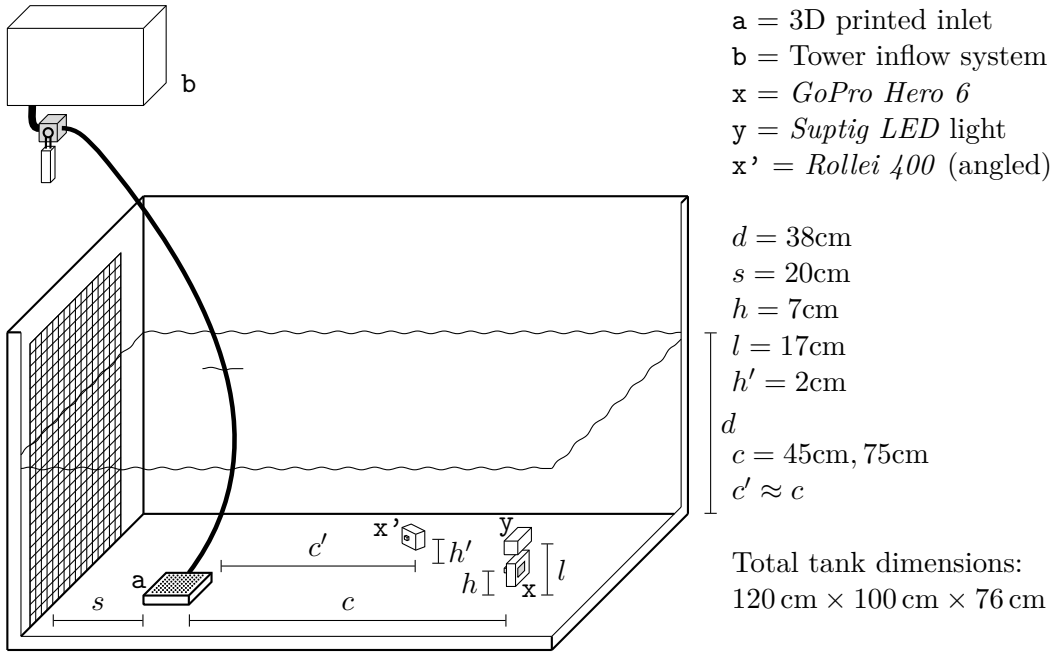


Figure 9: Experiment setup

3.4.2 Execution

We conducted four different tests with the following properties:

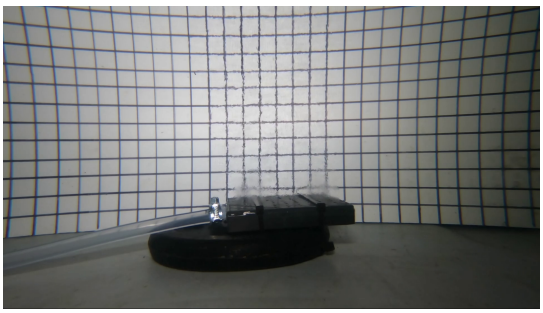
Name	Properties
Warm water	Maximum temperature at the inlet was50 °C.
Salt water 1	Salinity of 10 ‰. 17 °C.
Salt water 2	Salinity of 20 ‰. 17 °C.
Salt water 3	Salinity of 40 ‰. 17 °C.

Table 4: Listing of tests and properties

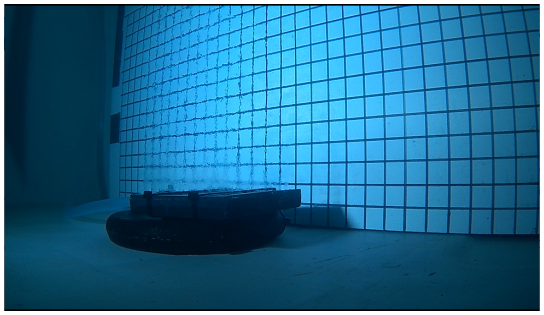
The initial water temperature was 16.3 °C

Each test was executed twice. The first time with cameras further away, the second time with closer cameras, with five minutes between two tests to minimize side effects between the tests.

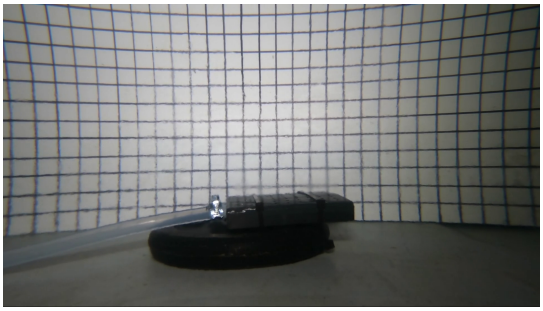
3.4.3 Results



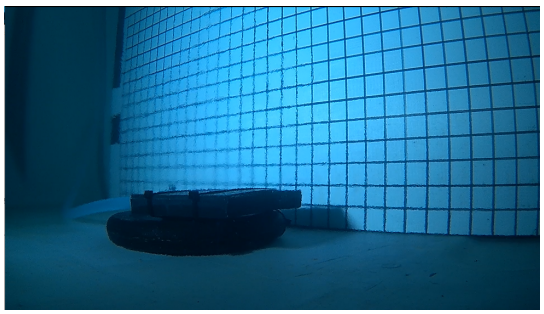
(a) Warm water, direct



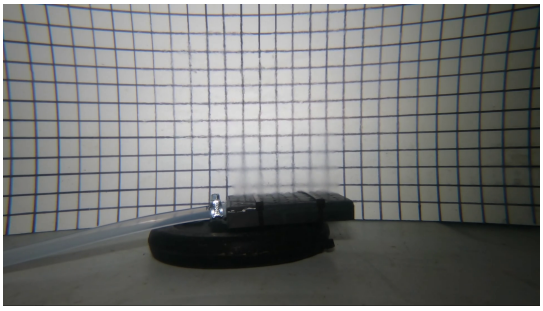
(b) Warm water, angled



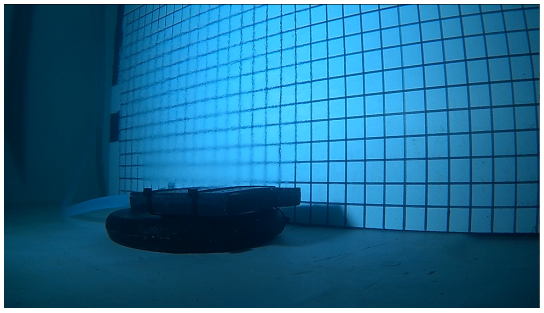
(c) Salt water 1, direct



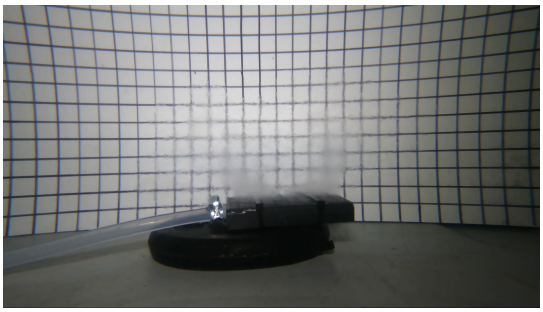
(d) Salt water 1, angled



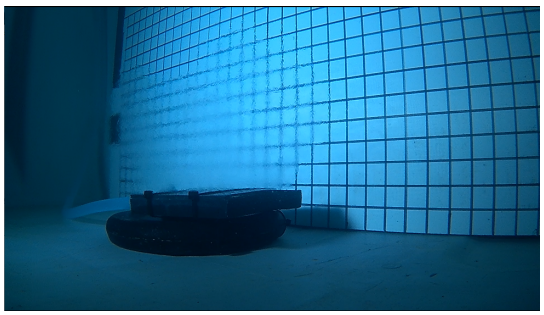
(e) Salt water 2, direct



(f) Salt water 2, angled



(g) Salt water 3, direct



(h) Salt water 3, angled

Figure 10: Snapshots of different videos created in the core experiment. Each left and right image origin from the same test and same time stamp. The left images was taken with the *GoPro Hero 6*, the right image was taken with the *Rollei 400*. Due to different white balance possibilities a heavy bluish cast can be seen on the images from the latter.

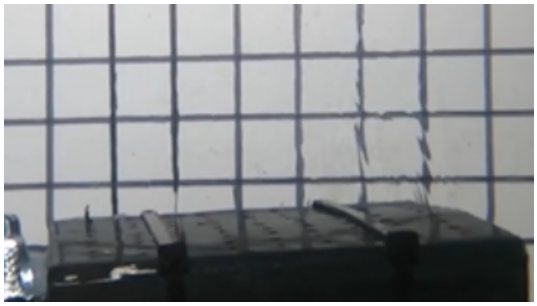


Figure 11: Constant distortion at the end of the warm water test

Warm water In the result video the Schlieren effect is clearly visible, shown in Figures 10 (a) and 10 (b). The inflowing water heads to the surface. The variety of intensities of the effect through the duration of this test is noticeable. When the inflow is strong the effect is visible not just above the inflow but up to the surface. Towards the end the inflow decreases, accordingly the Schlieren effect gets more subtle and becomes barely visible. There is also a point where a distortion is still visible just above the inlet, but there is no high frequently flickering visible, and the distortion does not change much. It looks like a still image, as in Figure 11.

After the test the temperature in the tank ranged from 16.6 °C to 17.4 °C. The further away from the inlet and the lower to the ground the colder the temperature.

Salt water 1 The Schlieren effect is also clearly visible, shown in Figures 10 (c) and 10 (d). In comparison to the warm water the inflowing water appears a little more opaque. Another noticeable observation is that initially the water heads to the surface, but the water stops rising and falls off to the sides, forming a plume. Then the plume shrinks to a consistent level. When the inflow stops, the whole plume falls off, spreading out on the ground besides the inlet. There, Schlieren are still visible even after the inflow stopped. In the five minutes between the tests this effect dissolved.

Salt water 2 The results Figures 10 (e) and 10 (f) of the second salt water test match the results of the first salt water test. As expected, the higher salinity increases the observed effects. The inflowing water is more opaque, and the plume shrinks to a smaller level. It is possible that both can be ascribed to the higher density of the more salted water.

Salt water 3 This test result Figures 10 (g) and 10 (h) also correlates with the two previous salt water tests. The inflowing water is even more opaque, and the consistent plume level is just above the inlet.

A noticeable observation in the result videos is a clear crisp boundary between the salt water plume and the surrounding water. Based on that, you can assume that the salt water does not mix immediately with the surrounding water. This effect is probably also present in the both previous salt water tests but not as clearly visible due to the lower salinity.

The individual tests have a flow characteristic in common. The effects are more intensive towards the sides of the inlet. This is well visible in Figure 10 (g). We assume that this corresponds to turbulences occurring when the inflow "jet" is inserted into still water. Though it is only possible to give an average value for the inlet velocity at each hole. By knowing the total inflow area A and the amount of inflowing water per time f it is possible to calculate the average inlet velocity v.

On the surface of the inlet are 81 holes, each with 2 mm diameter, in total $2\,\mathrm{L} = 2000\,\mathrm{cm}^3$ inflowing fluid and the duration of the inflow was 40 seconds.

$$f = \frac{2000\,\mathrm{cm}^3}{40\,\mathrm{s}} = 50\,\frac{\mathrm{cm}^3}{\mathrm{s}} \tag{1}$$

$$A = 81 \cdot \pi \cdot \frac{2\,\mathrm{mm}^2}{4} = 254,502\,\mathrm{mm}^2 = 2,54502\,\mathrm{cm}^2 \tag{2}$$

$$v = \frac{f}{A} = \frac{50\,\frac{\mathrm{cm}^3}{\mathrm{s}}}{2,54502\,\mathrm{cm}^2} = 19,646\,\frac{\mathrm{cm}}{\mathrm{s}} = 0,19646\,\frac{\mathrm{m}}{\mathrm{s}} \tag{3}$$

3.5 Connecting experiments with simulation and detection

3.5.1 Detection

After experimentally creating videos of hydrothermal vents, those videos were given to the subgroup "Detektion" for evaluation and afterwards they were compared with a frame from the original video which was analyzed. The first results we got were from late February 2020. While the subgroup was still working on the algorithm, those results were not final. The results of the algorithm are bare point clouds and there is no distinction between image segments with or without vents, like drawn bounding boxes on an image. Apart from those still missing features, the point density is clearly higher in the areas of the outflow. Regarding this, the algorithm from late February 2020 in total can be considered as "working".

As we are approaching the end of this project, we can now get some "final" analyzed videos from the subgroup "Detektion". One analyzed video can be seen in different processing steps and with the final result in Figure 38.

3.5.2 Comparison to simulation

The simulation software by the subgroup "Simulation" produces Schlieren videos. To test and verify the simulation software, its results will be compared to results of the experiment. Therefore, a simulation case was created that took over properties from the core experiment. The case is precisely described in the section Section 4.4.3: Recreating the experiment.

Of course, the simulation cannot depict the experiment is every aspect, for instance the inlet velocity. The simulation used an average inlet velocity modulated with a random value for every inlet hole. This is different in the experiment, as stated in Section 3.4.3: Results. Another important difference is the inflowing medium. The simulation achieved a difference in the

refractive indices for the Schlieren by a difference in temperature. In the experiment, on the other hand, we produced the Schlieren not only with warmer water but also with salted water. Naturally, the simulation results will only be compared to the warm water experiment. Nevertheless, the comparison allows a qualitative evaluation for the simulation software, to determine how realistic the simulation results are.

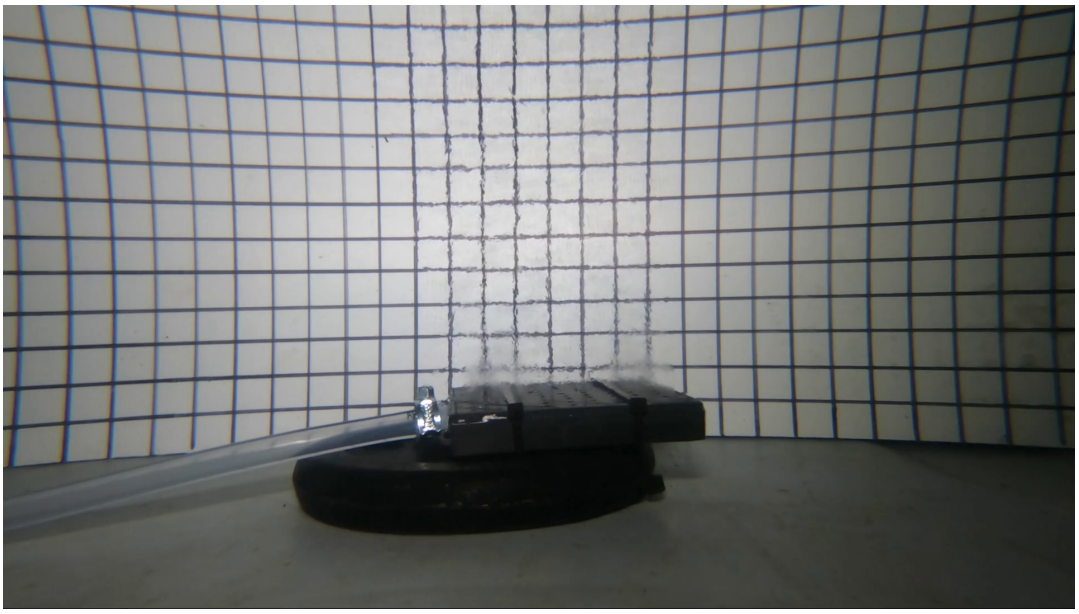


Figure 12: Experiment result

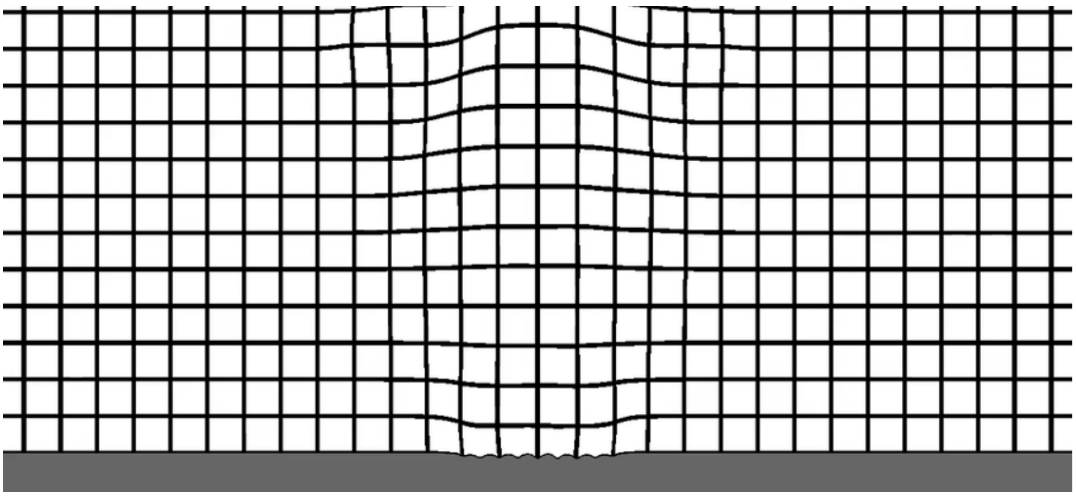


Figure 13: Simulation result

An obvious difference is that the simulation is much cleaner, meaning less noise and other factors like the fish eye effect from the experiment result. Also, the simulations do not have

an inlet structure and hose. The background in the simulation is lit perfectly even, while in the experiment the light intensity decreases towards the sides.

Regarding the Schlieren effect the simulation has two aspects. On a larger scale the simulation has distortions that clearly change the course of the background grid lines, especially the horizontal lines in the middle of the upper half. This is so significant the top grid line goes out of scope and reenters. This effect is not observable in the experiment. The other aspect is a small scale effect at the grid lines. The lines flicker and change slightly very similar to the high frequency flickering effect which is present in the experiment. But these small scale distortions of the lines are discrete, rather than continuous. This is probably due to the background which is pixel graphic. Therefore, the distortions can only occur in pixel quantities too. Interestingly these small distortion do not occur at the very middle vertical line.

Despite the mentioned differences to the experiment results the simulation produced a promising result with visible Schlieren effects.

3.6 Conclusion

3.6.1 Comparison to real seafloor vents

Our experiment reproduced Schlieren only under very specific conditions and circumstances. Hence, it did not take in to account every characteristic of a real seafloor vent. The following will describe which characteristics of real vents were not regarded by this experiment.

The first deviation involves physical properties of the inflow. The inlet properties in this experiment were rather constant. The velocity and temperature of the inflowing water were steady. Furthermore, the inlet remained unchanged throughout a test. Also, the inflowing water was either warm or had one of three different salinities. Though in reality, diffuse hydrothermal seafloor vents obviously do not have well-defined properties and are most likely a combination of those. Compounds of the inflowing water vary from just added salt (NaCl) and the temperatures are different at any geographic location where seafloor vents occur. A real inlet is not definite, like a 3D printed symmetric 9×9 hole grid inlet, and it cannot be assumed that the inflow velocity is steady. Real seafloor vent properties are very irregular and inconsistent.

Another big difference to real seafloor vents are the environmental conditions. The experiment took place in a much smaller tank with way less water volume compared to the sea. Therefore, the proportion of the inflowing water to the total amount of water is higher and the pressure conditions are different. Other factors that were not regarded are water currents and disturbances of moving object such as animals. A plane black and white grid was used as a background, while in reality a broad variety of natural backgrounds occur.

Regarding the video capturing, the experiment produced results with low noise. This is not guaranteed in real world applications. The distance from the camera to the vent is probably higher and likely to change during capturing. Movements of the camera vehicle influence the resulting video as well. Due to the limitations of the experiment, a sufficient illumination of

the scene was possible but in reality longer distances, bigger areas and back-scatter worsen the illumination and therefore also the visibility.

So on one hand our experiment was limited by physical characteristics, on the other hand real world Schlieren are much more diverse and occur in different circumstances. Hence we were not able to take every characteristic into account.

3.6.2 Future Work

While the resulting videos from Section 3.4: Recreating hydrothermal vents differ from real vents as described above, they are still by far the best we had throughout all of our test series, both in terms of video quality and similarity to real vents. With this in mind, there are no further plans on continuing our tests, besides a demonstration to represent our project to the public.

Nonetheless, there is the possibility of a subsequent project, where our work might be continued. For this, quite a few improvements come to mind.

First of all, the inlet properties could be varied more, for example by controlling and varying the flow volume and intensity. This could be done either by hand or by an actuated flow control valve, which can constantly adjust the flow volume. A self-sucking pump could also be helpful for controlling inflow pressure and volume. The properties of the inflowing water itself would have to be modified. While in our tests only distinct different salinities and temperatures were used, in a future test those could be altered during one single inflow in order to better simulate real deep sea conditions.

Next up would be an improvement of the inlet, which right now is 3D printed and afterwards got drilled for inlet holes. While this can be reproduced relatively easy in the simulation, it is not similar to real vents. In order to get the inlet more natural, the inlet structure has to be modified. An idea to resolve this issue would be to still use the 3D printed inlet, but reducing the thickness of the surface heavily before printing, creating just a mesh of PLA. This could result in a more diffuse and therefore more realistic inflow and inlet.

Another important aspect is the realism of the surrounding. Moving to a bigger tank or pool and putting the camera at an angle facing downwards, one could achieve an image without visible tank walls. Going even further, the ground could be "aquascaped" to look just like a field of hydrothermal vents in the deep sea.

In our test series, the position of the camera and thereby its distance to the inflow was consistent throughout the tests. Also, the camera models used in the tests are not proper for the use in deep sea, work-class ROVs sometimes do not even have HD-video enabled cameras. When working with an ROV, conditions like water currents and ROV thrust are changing constantly, it is impossible to achieve a constant camera position. In order to recreate more realistic videos, this has to be taken in account and one could either use an ROV or some other way of moving the camera, e.g. by hand on a stick, to get more natural results.

Using an ROV would also imply using an ROV camera, being more realistic and producing more realistic videos in comparison with those taken in the deep sea. One has to keep in mind, that the idea of a laboratory experiment is to isolate the occurring optical effect and therefore the content of the last paragraph was intentionally not taken into account in our experiments and might also not taken into account in future work.

4 Simulation

4.1 Introduction

A main part of this project is to provide simulations of diffuse seafloor vents which may be used as input data for the detection software. Moreover, they can be compared to the results of the experiments that are done for the same reason. That enables the project group to validate their results by comparing simulation and experimental results and analyzing both using the detection software.

The idea is to use a CFD-Tool to simulate seafloor vent objects based on time steps and then use a raytracer to render an image of those objects for every time step simulated. Successively rendered, the output images can be converted into a video file showing a continuous simulation of seafloor vents.

4.2 Background

The first step to be taken at the very beginning was to decide whether to use an existing ray tracing software or to build our own raytracer. Blender for example, an open source software, includes features like smoke and fluid simulation. Since no one of the simulation group has worked with Blender or a similar software before, much time would have been spent to understand the functionalities of possible tools and to decide which one would be the best to work with. Also, it could have been difficult to add new functions that might be needed in the course of the project. Due to these facts it was decided to build a new raytracer.

Another question we discussed during the implementation of the basic raytracer was how to simulate the movement of the inflowing water of a vent later on and how to calculate the corresponding refractive indices so that our output simulation will be realistic. One possibility was to use a CFD tool, with which the inflowing water of a vent can be simulated. The output of such simulations would have been a set of physical data for each cell in a custom mesh for each timestep of the simulation. We decided to use such a tool and to import the output information into our raytracer. The exact tools we used and the way we imported and worked with the CFD data are described in section 4.4.

4.3 Raytracer

In the following subsections the construction and the functionalities of the basic raytracer are explained. Further classes that were implemented to import CFD data and to build vent objects are described in section 4.5.

4.3.1 Construction

The construction of our raytracer is based on a tutorial by Dmitry V. Sokolov published on GitHub (Sokolov, 2019). Another source we used to understand the ray tracing process is a raytracing tutorial by Scratchapixel Scratchapixel, 2016. We further devided the construction into model and controller. That made it easier to split up the different tasks while programming and it ensured the clarity of the code.

The main element of the model is the class Scene. It determines all elements of a rendered image. A scene is a three-dimensional space containing various scene objects of the class SceneObject, light objects of the class Light and one object of the class Camera, which defines the section and the perspective of the image to be rendered.

Since SceneObject is an abstract class, there are several classes inheriting from it that define the exact type of a scene object. They have certain attributes to describe their position and their size. To describe the material properties of a scene object, a further class named Material is used. The following diagram shows the classes that are available to create a certain scene object.

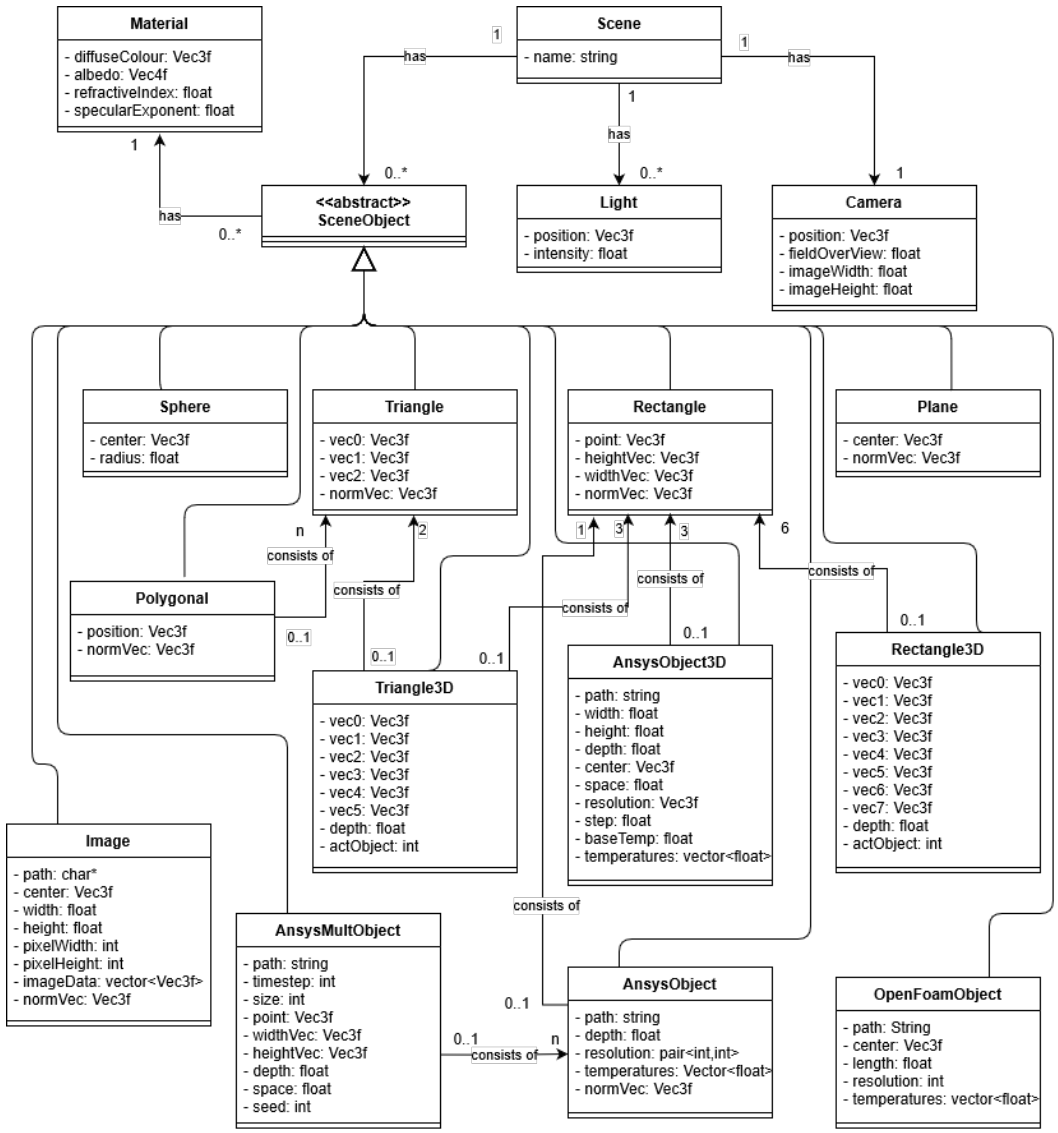


Figure 14: class diagram of the raytracer model

The controller part contains the logic of the raytracer. Its main element is the class Render. Its public function render() renders a given scene and saves the output image as a Portable pixmap format (PPM)-file. This is done by casting rays from the camera position to each point within the width and the height of the image. To enable the whole rendering process, several other functions are provided by further controller classes.

The class SceneController provides the function sceneIntersect() with the possibility to check if a given ray intersects with an object of a given scene and stores information about the hit point. Furthermore, SceneController provides a function that computes the reflection of a given ray at a certain hit point.

There are also object controller classes for each type of scene objects. They all contain equal functions with different implementations depending on the object type. For clarity, a class named SceneObjectController provides eponymous functions that distinguish between the object types and then call the matching function of the right object controller class. Accordingly, SceneObjectController is the interface between Render and SceneController and the specific object controller classes.

One of the provided functions is rayIntersect(), which checks if a given object is hit by a given ray. It is called by the previously mentioned function sceneIntersect(). Further functions enable to get the normal vector and the material of a given object at a certain point and to compute the refraction of a given ray at an object.

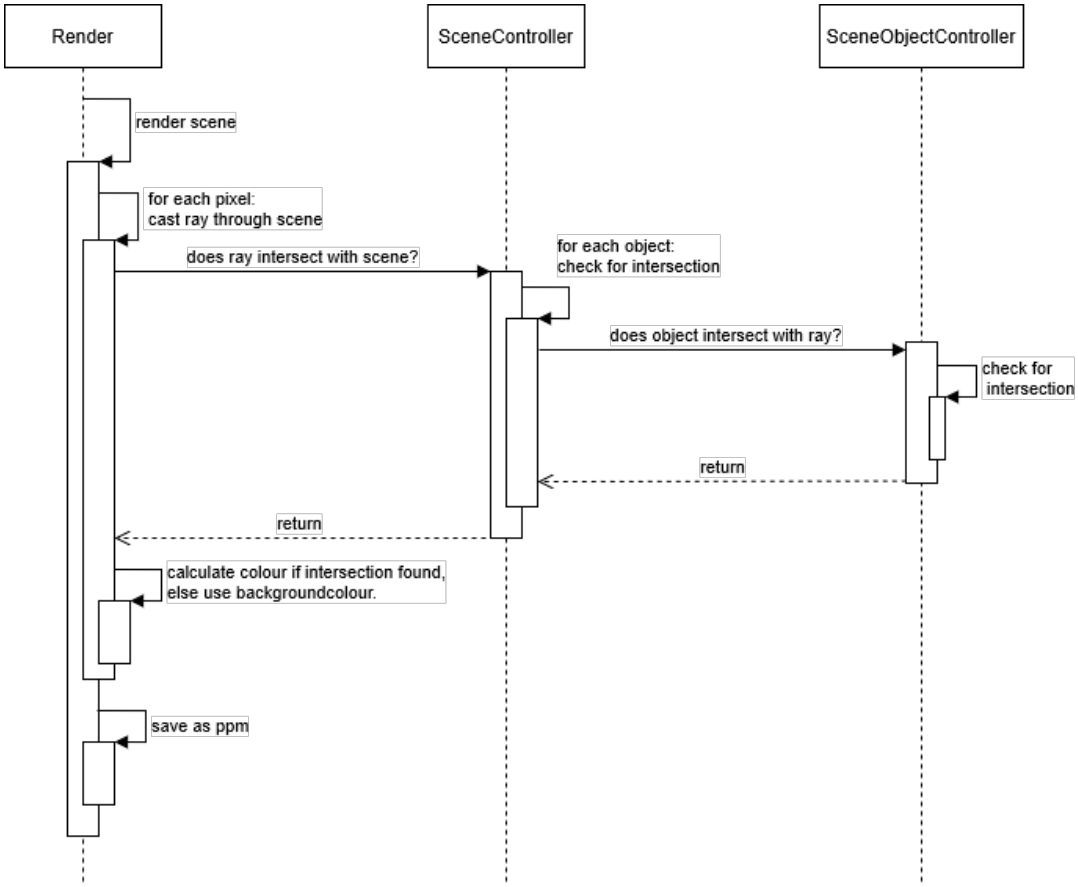
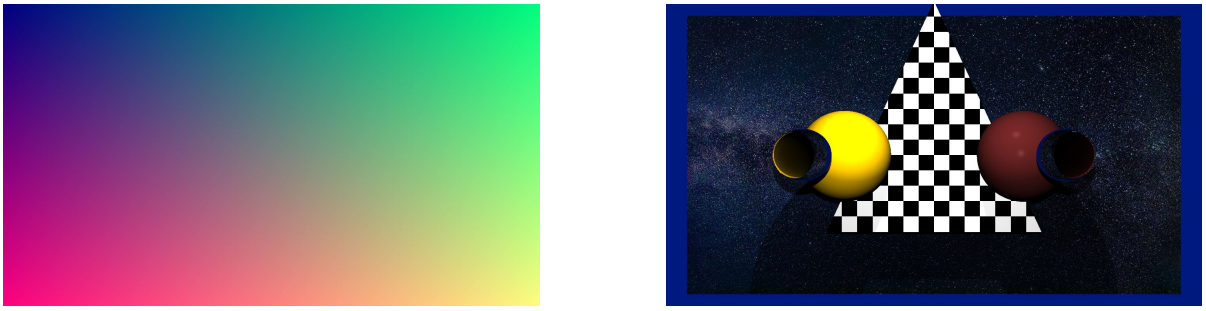


Figure 15: abstract sequence diagram of the raytracer controller

Normally, we rather want to create a sequence of pictures than just a single image. In order to maintain this, there are two options provided. The controller-class for video processing, called VideoController, provides the possibility to simulate camera movement in a scene. Its only function renderVideo() takes an amount of timesteps which should be rendered, also 3 functions for camera translations in x, z and y direction. Furthermore, it takes the scene to be rendered and starts the whole render-process for each timestep. A second option is to use

a provided loop in the main class to render a sequence of scenes that distinguish from each other so that a moving scene can be created.



(a) Beginnings (b) Completed

Figure 16: Raytracing Pictures - Step by step development of our Raytracer

The implementation of the named functions and how they exactly enable the rendering process is further described in section 4.3.2.

4.3.2 Functionality

In order to render an image, it is necessary to create a scene first. The scene contains all scene objects to be rendered, as well as the camera and the lights placed in its three-dimensional space. As mentioned before, the top level function to handle the rendering process is called render(). As its only argument, it takes a scene. To make understandable how things work, we created some Pseudo-Code.

Algorithm 1 void render (scene)

```
RGB pixelbuffer [width*height];
for all pixel : pixelbuffer do
  pixel = castRay(origin, direction, scene);
end for
```

As the idea is to shoot a ray from the camera through every pixel of a grid determined by the image width and height the camera was given, we need a way to represent this grid. That can be done by a pixelbuffer with width*height pixel entries. For each of these a ray gets cast with the cameras position as origin and direction towards the pixel position. The castRay()-function takes all these parameters for further calculations.

Often there is no object which a ray intersects with. In that case, the program will return a defined background colour. The sceneIntersect()-method handles if the ray intersects with any object, as explained in the next snippet of code. Assumimg sceneIntersect() indicates an intersection, the colour which gets returned by castRay() will be the colour of the intersected object, calculated with its refractive and reflective index, as well as the light intensity.

Algorithm 2 RGB castRay (origin, direction, scene)

```
backgroundcolour = blue;
sceneobject = null;
hitpoint = null;
normvector = null;
if (!sceneIntersect (origin, direction, scene, hitpoint, normvector, sceneobject)) then
  return backgroundcolour;
end if
material = sceneobject.Material();
refractdirection = calculateRefractionRay(hitpoint, direction, normvector, material);
refractioncolour = castRay(hitpoint, refractDirection, scene);
reflectdirection = calculateReflectionRay(direction, normvector);
reflectioncolour = castRay(hitpoint, reflectdirection, scene);
diffuseLight, specularLight = null;
for all light : scene.Lights() do
  hp, nv, so = null;
  lightdirection = light.Position() - hitpoint;
  if (!sceneIntersect(hitpoint, lightdirection, scene, hp, nv, so) then
    diffuseLight += light.Intensity() * lightdirection * normvector;
    specularLight += powerof(calculateReflectionRay(lightdirection, normvector),material.SpecularExponent()*light.Intensity();
  end if
end for
outputcolour = sceneobject.getRGB() * diffuseLight * material.getAlbedo(diffuse) + specularLight * material.getAlbedo(specular) + refractioncolour * material.getAlbedo(refraction) + reflectioncolour * material.getAlbedo(reflection);
return outputcolour;
```

Algorithm 3 bool sceneIntersect (origin, direction, scene, sceneobject)

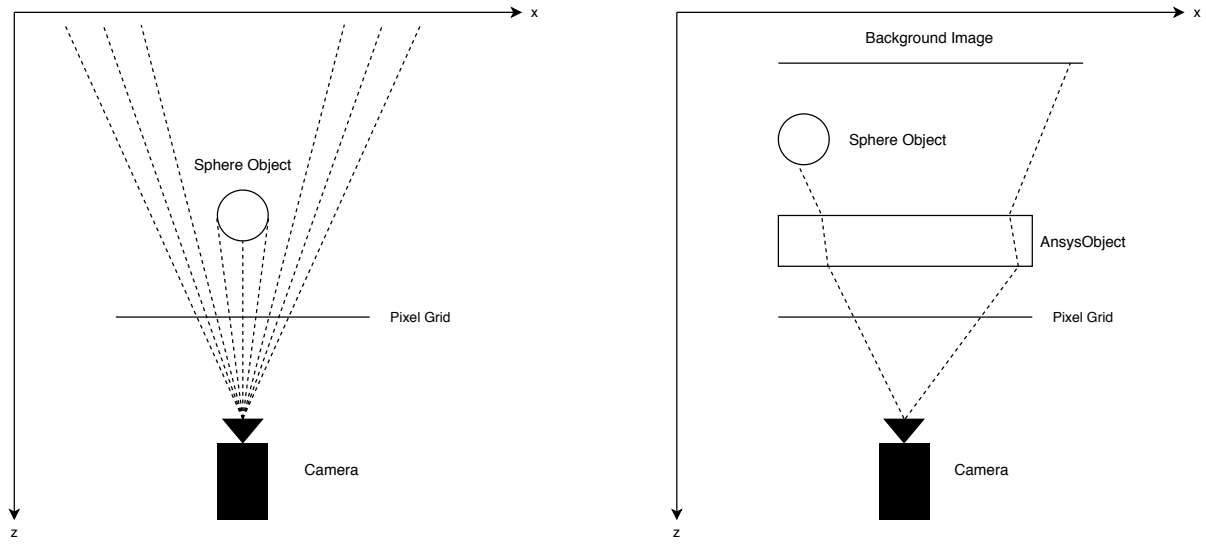
```
distance = maximum;
compDistance;
for all current : scene.getSceneObjects() do
  if (current.rayIntersect(origin, direction, scene, sceneobject, compDistance) and compDistance < distance) then
    sceneobject = current;
    distance = compDistance;
  end if
end for
return distance < 1000;
```

The sceneIntersect()-function iterates through all scene objects the scene contains and calls their rayIntersect()-function. Its procedure is explained more detailed in the next snippet. In case a rayIntersect()-function indicates an intersection, the sceneObject-variable created in the castRay()-function gets overwritten. Through all iterations of the scene objects, the one which is closest to the camera gets returned. The final return sets the distance of scene objects to the camera can be a maximum of 1000 in internal representation units.

Algorithm 4 bool rayIntersect (origin, direction, sceneobject, compDistance)

```
if (intersectionBetweenRayAndObject(origin, direction, sceneobject) == true) then
  compDistance = calculateDistanceFromOriginToHitpointWithObject(origin, direction, sceneobject);
  return true;
end if
return false;
```

Calculating if there is an intersection between a ray and a scene oject is indivual, depending on the type of the SceneObject. If it turns out that there is an intersection, the comparison distance variable gets overwritten and handled in the sceneIntersect()-method.



(a) Simple scene with a sphere

(b) Scene with a Sphere and an Ansys Object with refraction

Figure 17: Diagram of basic raytracing functionality where rays are represented by dotted lines

The basic mode of operation of the raytracer can be explained with a simple diagram as seen in Figure 17. For each pixel in the pixelgrid a ray gets cast and the closest intersection calculated and the resulting color saved in the pixelgrid. If a ray intersects with an object that has refractive properties for example an Ansys Object, the ray gets refracted and a recursive function call is made. The resulting pixel in the pixelgrid is a combination of the color of the initial intersection, the color from refraction and also reflection (not shown in the diagram).

4.3.3 First ideas for Schlieren Simulation

While working at the basic functionality of the raytracer, several ideas came up of how to simulate Schlieren of diffuse seafloor vents. This was before we started working with CFD tools. So, these ideas are more of a way trying to approximate refraction effects that look like Schlieren, but are not physically accurate.

First attempt was to shape a seafloor vent by many little triangles with different indices of refraction. This approach is similar to one of the first experiments by the experiment group of dipping several PU-shapes in water, but in this case, we have different refractive indices to get a more Schlieren-like looking result. Up to this point of time a scene object was only able to return the same material for each ray intersection with this object and due to the results of this first attempt we assume to get better results if the material which is returned depends on the point of the intersection.

Therefore, we made a second attempt with variable refractive indices. We tried to overlay the background by triangles in shape of seafloor vents. The refractive index of each point of the

triangles is calculated by a two-dimensional sine function to get Schlieren characteristic oscillation. The result of this function is multiplied by a gaussian distributed random number for a more diffuse appearance of the vent approximation. But in the end the shape of the triangles is too edgy to get close to a seafloor vent shape and the sine functions are too periodically to look as diffuse as Schlieren.

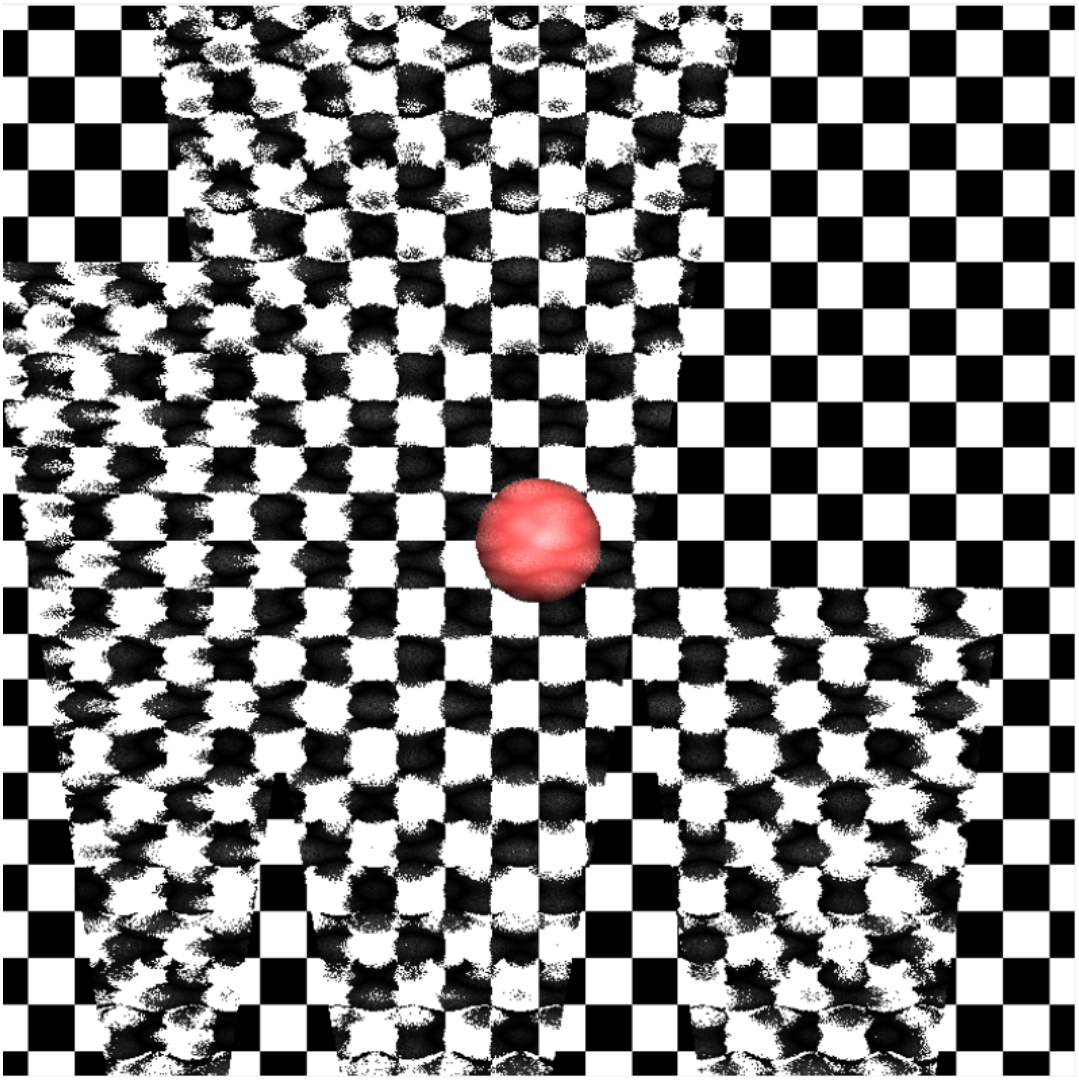


Figure 18: First ideas of Schlieren approximation with sine functions

4.3.4 Multiprocessing and simplifying the Architecture

To achieve better raytracing performance, we tried to implement multithreading with Open Multi-Processing (OpenMP). Due to difficulties with data dependencies those efforts were not successful.

In the last weeks of the project we had the idea to port the raytracer to Compute Unified Device Architecture (CUDA)-C++ to enable parallel computing on a Graphics processing units (GPUs) by *Nvidia*. To be able to move processing to a GPU, all data needed for the processing has to be either moved to the GPUs memory or has to be created there. With the previous implementation of the raytracer, mostly ignoring memory-related architecture conditions, this simply was not possible.

This resulted in one diverging branch of the codebase called "no controller". This branch simplifies the previous software architecture, a model-controller architecture, by removing most controllers. Instead of one controller class per scene object class, the methods from those controllers are integrated into the scene objects.

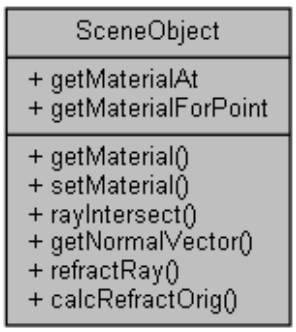


Figure 19: Abstract class SceneObject

The abstract class *SceneObject* now has four virtual functions which have to be implemented by the inheriting *SceneObjects*. Those functions were previously implemented inside the different *SceneObjectController*s and are responsible for the refracting of rays hitting those objects.

While work was continued on the main branch as well, the two versions of the raytracer diverged. The "no controller"-version has no implementations of *Rectangle3D*, *Triangle3D* and *Polygonal*.

4.4 CFD Simulation

In order to recreate the visual effects of hydrothermal vents different CFD Tools were evaluated. Since the research group had no prior experience with CFD simulation the two most common used tools in this area were considered first: Open-source Field Operation And Manipulation (OpenFOAM) (The OpenFOAM Foundation Ltd, 2020) and the Analysis System (Ansys) software (ANSYS Inc., 2020) package with the former being open-source and the latter being a commercial tool with a free but restricted student version. Due to the open-source nature OpenFOAM was first used to produce the first test cases of water flow. As it turned out OpenFOAM is difficult and time consuming to use without prior knowledge in the area of CFD which is why we switched to Ansys.

Due to having a Graphical User interface (GUI) (in contrast to OpenFOAM) Ansys was easier to learn and also more entry-level tutorials were available online that helped create the first simulation cases. At first we simulated a simple two dimensional vent in order to understand the workflow of Ansys that consists of three basic steps. The first step is creating a geometry which defines the dimensions in which the simulation case takes place but also boundaries like inlets, outlets and walls. In the next step the geometry needs to be converted into a mesh. In this step different parameters influence the way the resulting mesh looks and performs; for example, a finer mesh usually correlates with longer processing times. The third step is setting up the simulation and performing the calculation. This includes the choice of models, boundary conditions, used materials, etc. and eventually exporting or analysing the result of the simulation. Often the post-processing is regarded as a separate step in the workflow of CFD tools. As we used multiphase flow, we decided to use the volume of fluid model for our simulations. Research in several forums showed that this might be the best model to use for our purposes. Since we worked with different temperatures, the energy equation model had to be activated, too. As viscous model, we decided to use k-omega. This decision was also based on our researches.

4.4.1 First Simulation Case: Simple Vent

For our first case we created a simple two dimensional geometry with one inlet that emits water into a body of cold water in order to get a first visualization of the behaviour of water inlets.

After our initial tests with bigger inlets that were 2 cm across or bigger we realized that smaller inlets yielded better visual results and created more turbulence. Since we assumed that turbulence was needed to create Schlieren we tried to simulate smaller inlets and used a finer mesh.

Parameter	Value
Size of Geometry	$1\,\mathrm{m} \times 1\,\mathrm{m}$
Mesh Block Size	$0.01\,\mathrm{m}$
Number of inlets	1
Inlet Size	$0.01\,\mathrm{m}$
Water temperature (cold)	$4\,^{\circ}\mathrm{C}$
Water temperature (warm)	$95\,^{\circ}\mathrm{C}$
Inlet Velocity	$1\,\frac{\mathrm{m}}{\mathrm{s}}$
Time step size	$0.04\,\mathrm{s}$

Table 5: Parameters of the first Ansys simulation

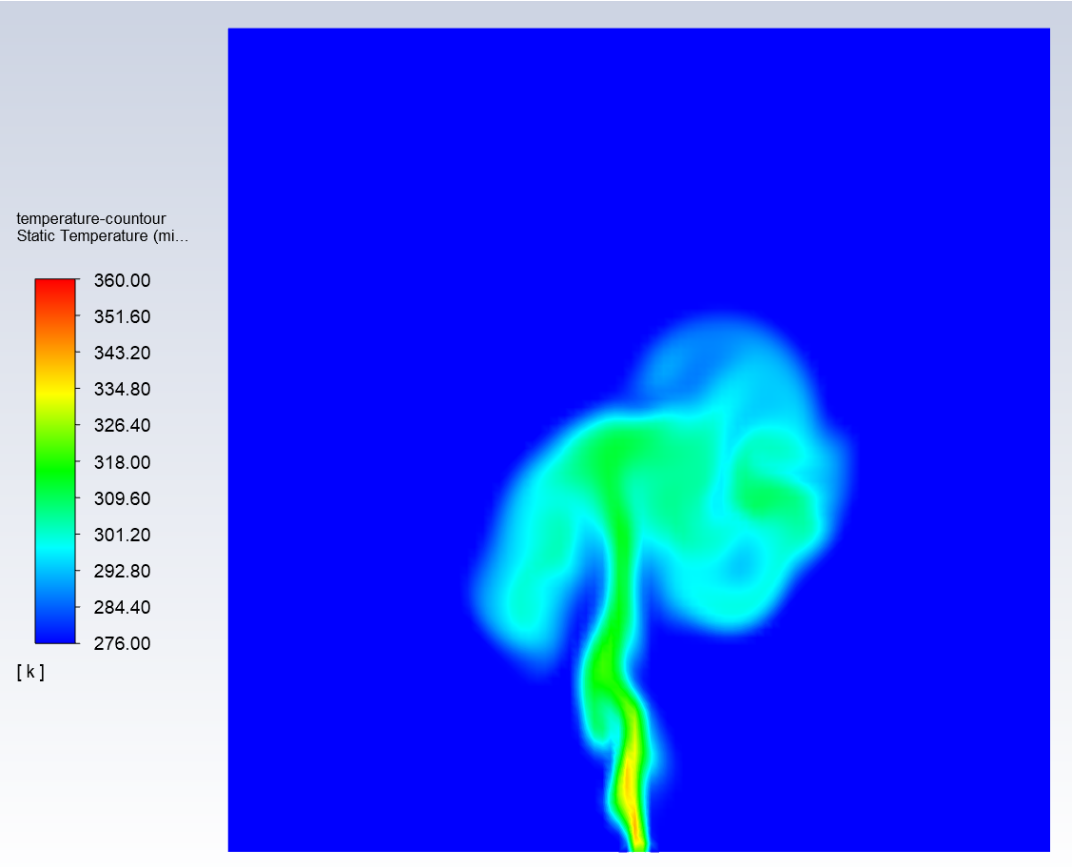


Figure 20: Simulation of a hot water inlet after 100 timesteps of 0.04 seconds (4 seconds)

4.4.2 Multiple Vents

The next step was to simulate multiple vents next to each other in order to see how they influenced each other. We started out with three vents. The first realization was that in our simulations the warm water from multiple vents joined and flowed upwards in one big conjoined stream as can be seen in Figure 21. We used the same parameters as described in the prior table.

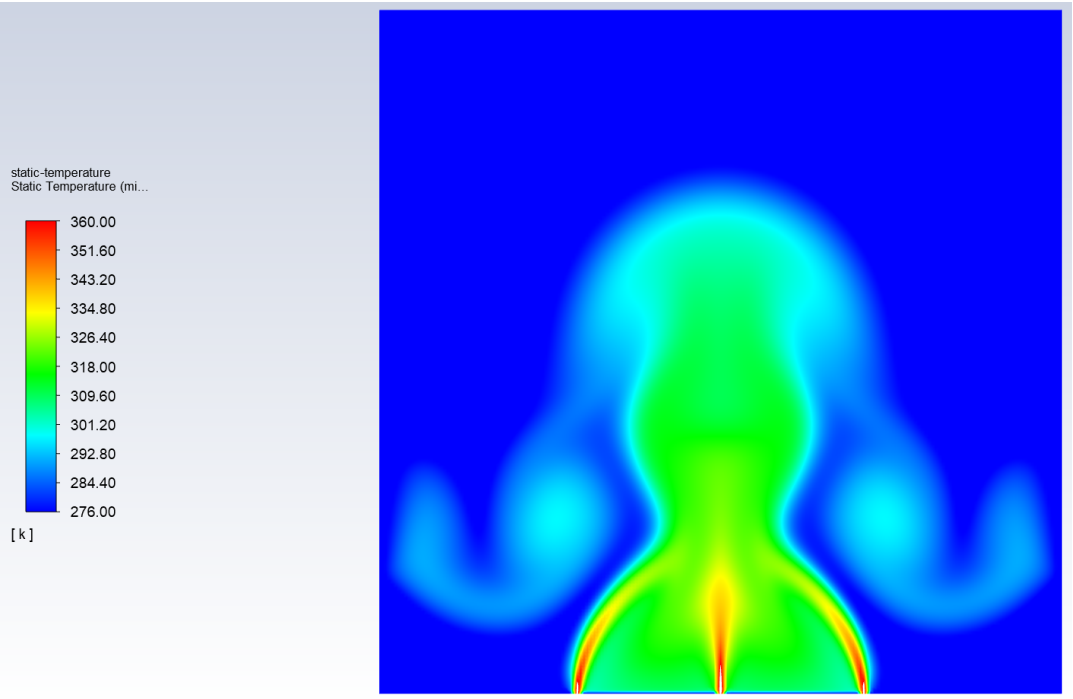


Figure 21: Simulation of three hot water inlets after 100 timesteps of 0.04 seconds (4 seconds)

We tried several methods to prevent this behaviour. The most efficient way was to put additional inlets between the hot water inlets. These additional inlets let water with the same temperature as the environment flow in. The exact impact of these inlets depends on their size and their velocity.

4.4.3 Recreating the experiment

In order to create a realistic simulation in Ansys the experiment with the best visual Schlieren effect was tried to recreate. We chose the core experiment, specifically the warm water results in 2D. Since we had the exact dimensions of the experiment setup we set up a case with the exact measurements. We chose to only recreate a box of $0.3\,\mathrm{m} \times 0.3\,\mathrm{m}$ since there was no effect at the outer areas in the experiment. Also, we realised that a consistent flow velocity wouldn't produce results which create enough turbulence which is why we modulated the velocity of the different inlets with values related to the time step. Nevertheless, the velocity was still based

on the experiment values with slight deviations. Furthermore, we used additional cold water inlets as explained in the last section to prevent the early merge of the vents.

Parameter	Value
Size of Geometry	$0.3\,\mathrm{m} \times 0.3\,\mathrm{m}$
Mesh Block Size	$0.01\,\mathrm{m}$
Number of inlets	9
Inlet Size	$0.02\,\mathrm{m}$
Water temperature (cold)	$16\,^{\circ}\mathrm{C}$
Water temperature (warm)	$50\,^{\circ}\mathrm{C}$
Inlet Velocity	$\sim 0.2\,\frac{\mathrm{m}}{\mathrm{s}}$
Time step size	$0.04\,\mathrm{s}$

Table 6: Parameters of the first Ansys simulation

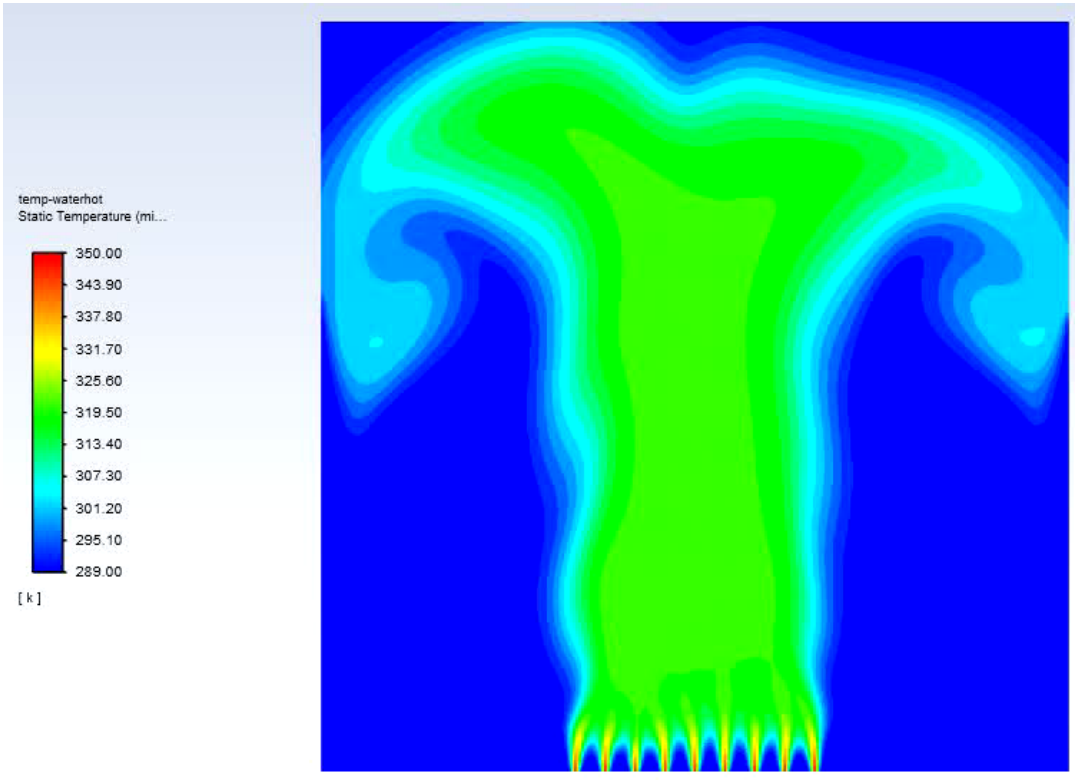


Figure 22: Simulation of the last experiment after 100 timesteps of 0.04 seconds (4 seconds)

We also tried to recreate the approach of using oil which was not effective in the first test series. Since we only did this to see if we will get the same result, we did not recreate the whole experiment but only used a single oil inlet at the bottom of a box of cold water. This simulation was unsuccessful, the substances didn't mix and there was no Schlieren effect.

We figured out that also for computed simulations using water is the best way to simulate Schlieren.

4.4.4 3D simulation

Besides the 2D simulations we also wanted to create a 3D simulation. Naturally the additional dimension increases the complexity.

The geometry modeling was done in Ansys SpaceClaim, which provides convenient functions. Though, there is one flaw. To place inlets on a plane, the plane must be separated into regions. This separation proved to be complicated in 3D. As a work around we achieved the separation by placing bumps on the plane. But this also means that the surface is no longer even and flat. We tried to keep the geometry as simple as possible. Therefore, the geometry is a cube with the inlets at the bottom arranged symmetric.

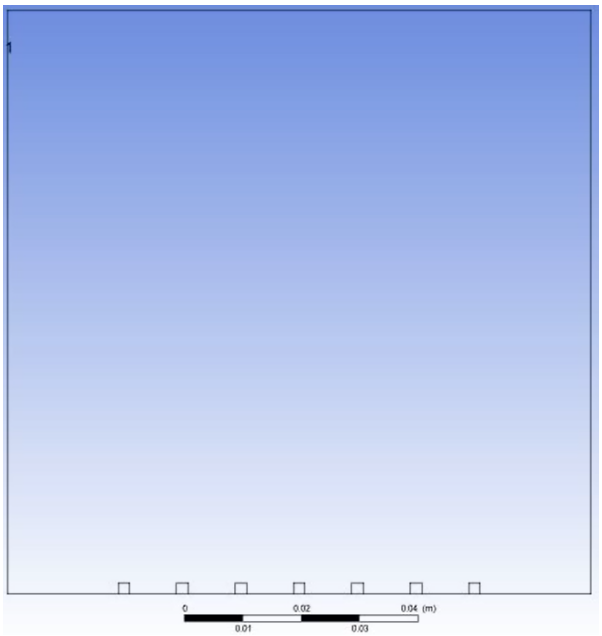


Figure 23: Slice of a 3D geometry. Squares at the bottom to separate the plane

The meshing (filling the geometry with small cells, which serve as quantities for the calculations) is not straight forward. We wanted the mesh to be regular, meaning the cells are small cubes that perfectly fill the geometry, like a grid. Despite using a method that provided this, the cell size was irregular at the previously mentioned bumps in the cube, even tough the cells would fit in perfectly. We were not able to find the reason for this. However, this irregularity only occurs at the bumps, the rest of the mesh is perfectly regular. In the meshing step the increased complexity of the 3D case brought problems. The free student license of Ansys only allows to load 512,000 cells in the Ansys Fluent solver. At a proper cell size of 1 mm (edge

length) this would only allow a cube geometry with an edge length of 80 mm. To use bigger geometries we acquired an Academic Ansys License, which restricted us in that way.

The initialization with Ansys Fluent as solver does not differ much from the 2D setups. Instead of modulating the inlet velocity to achieve a more turbulent flow, in the 3D case we slightly modulated the inflow direction. To handle the computation of the increased complexity we used high performance remote computer situated in the CGVR's lab and in the Microsoft Azure Cloud. Using General purpose Graphics processing units (GPGPUs) for the computation, that Ansys support, did not result in a significant acceleration of the computation. We tried a *Nvidia RTX2080 Super* and a *Nvidia Tesla K80*. At this point only limit for computations was free disk space. The case described below for example has a size of about 150 GB and another 150 GB for the American Standard Code for Information Interchange (ASCII) export of the temperature.

The following 3D-case orients, like Section 4.4.3: Recreating the experiment, on the core experiment.

Parameter	Value
Size of Geometry	$0.3\,\text{m} \times 0.3\,\text{m} \times 0.3\,\text{m}$
Mesh Block Size	$0.01\,\text{m}$
Number of inlets	81
Inlet Size	$0.02\,\text{m}$
Water temperature (cold)	$16.85\,°\text{C}$
Water temperature (warm)	$51.85\,°\text{C}$
Inlet Velocity	$0.2\,\frac{\text{m}}{\text{s}}$
Time step size	$0.033\,\text{s}$

Table 7: Parameters of the 3D Ansys simulation

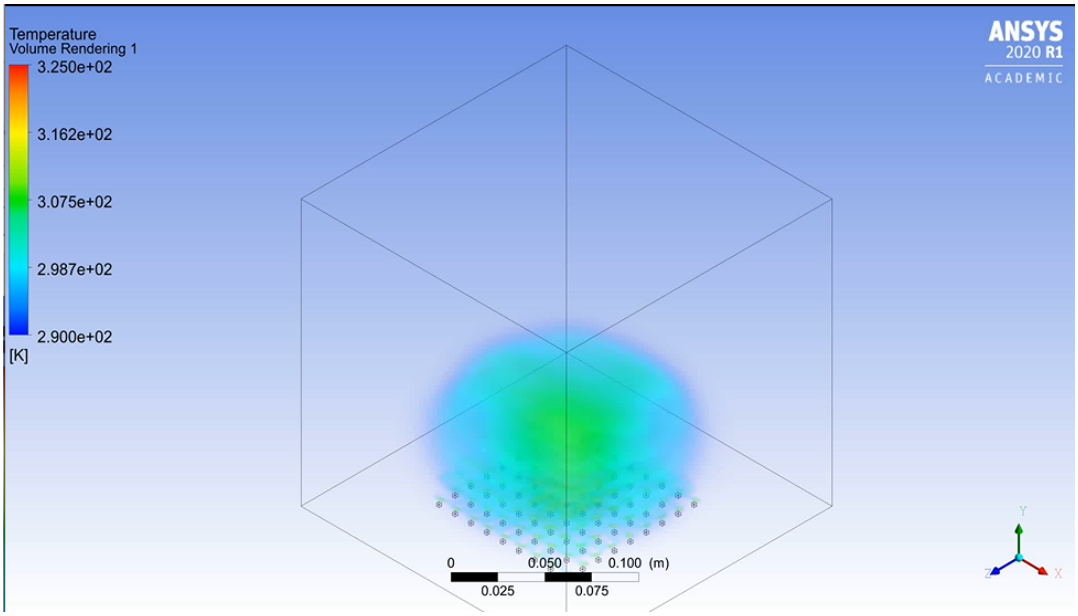


Figure 24: Snapshot of the animated results of the 3D CFD Simulations

4.4.5 Export

In order to use the temperature data of our simulated seafloor vents the data must be exported in a suitable format. Ansys already supports structured temperature data export in binary or ASCII format. Considering the downside of larger datafiles compared to the binary format, we chose the ASCII format because it could simply be imported into our raytracing software with the help of regular expressions. The format is pretty straight forward as it simply is a list of every node for every time step with the corresponding x and y component (additionally z in a 3D case) and the temperature at this point.

4.5 CFD Results in Raytracer

After finally making the decision to use CFD tools, it necessary to think about how the information they calculate shall be used in our raytracer.

Considering what we've got from the export data of the different CFD tools there are different ways these data have to be read in. In OpenFOAM exports there are just the temperatures of each cell ordered by x and y coordinates listed. In the export files generated by Ansys there are additionally to the temperatures the coordinates of each grid point specified. At first, the Ansys Data-output seemed to be ordered as well. But, with rising complexity of the Ansys simulation, the data got mixed up. Thus, the data needs to be reordered before it can be processed further.

As the CFD data is accessible by an import to our raytracer, the question came up how this data will be used.

One idea was to create a vent object that consists of many small cubes with a transparent material. The refractive index of each cube would have been depending on the CFD data for a corresponding node. Two-dimensional as well as three-dimensional simulations could have been realised this way. But, due to the fact that a huge amount of tiny cubes consisting of each twelve triangle objects would be necessary, this idea was discarded because it would be too inefficient in the current implementation of our "simple" raytracer. The more objects a scene contains, the longer time it takes to render a single image.

Similar to the schlieren approximation with sine functions we created for 2D CFD simulation a vent object that returns different materials depending on the point of the ray-object intersection. It consists of a rectangle, so that the CFD data, which is in shape of a rectangle too, can be represented in the raytracer. The relative position of the intersection point on the rectangle can be calculated which can be used to retrieve the CFD grid points of the cell that is hit by the ray. Given these points we can compute the temperature of each intersection point by linear interpolation. In knowledge of the temperature we can directly calculate the index of refraction, which is returned in the material of the specific intersection point. In order to get an even better Schlieren effect, we later changed the equation with which we calculated the refractive indices for some simulations so that the indices would have a higher variation.

Due to correctness of optics and Snell's law our vent object has an imaginary depth, which means that the ray is refracted at the intersection point and as it reaches the depth it refracts back like it would leave a 3D-object.

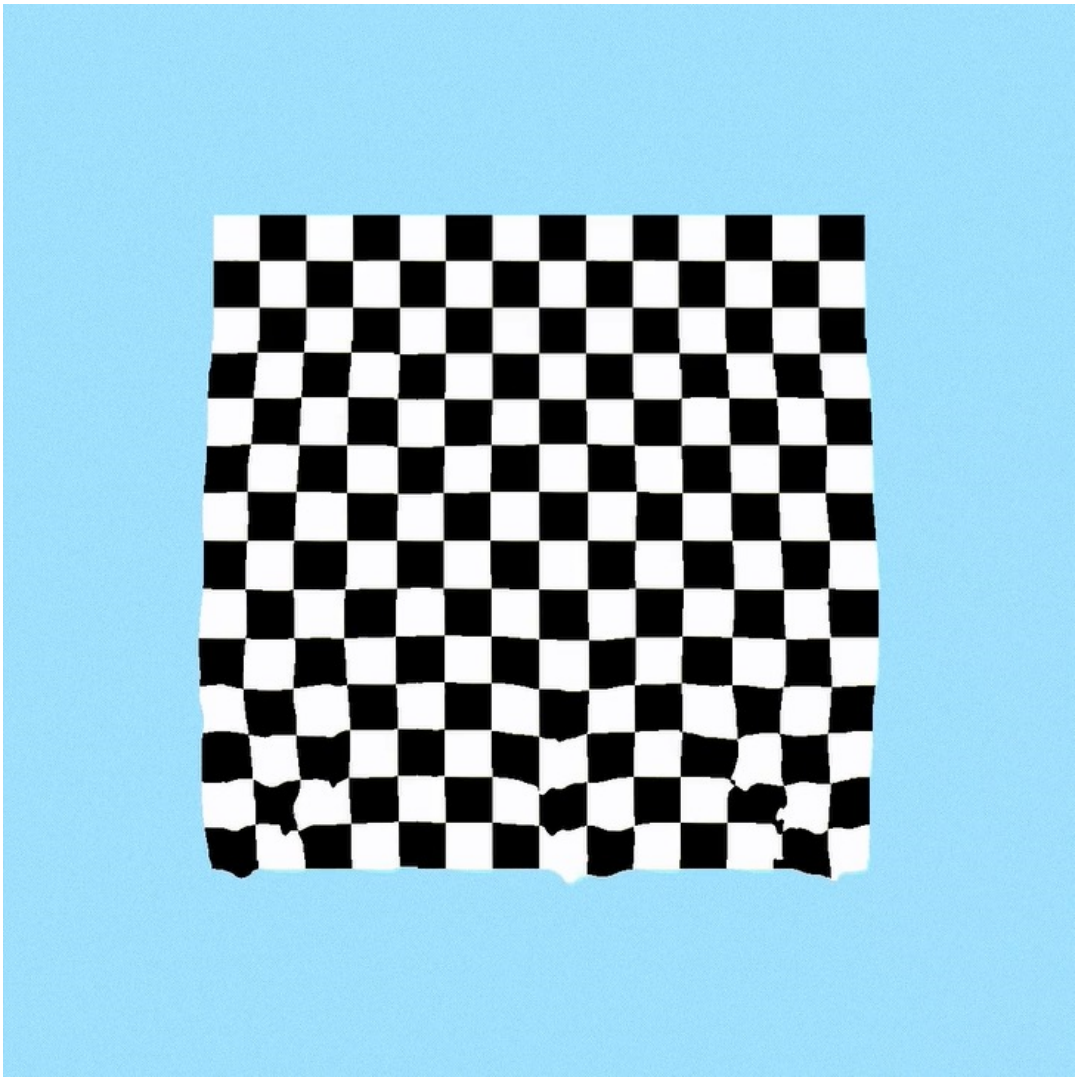


Figure 25: Ansys Object in Raytracer

For 3D simulation we have a similar approach, we want to approximate the way a ray would take in diffuse water by tracing the ray through a 3D-Grid of temperatures. In other words, we are calculating the point where a ray leaves the 3D-Object, which is comparable to the tracing in a 2D-object with its depth. The temperature at any point in the 3D-object can be computed by linear interpolation similar to 2D-objects, but with one more dimension. The tracing through the 3D-object starts at the intersection point and is followed by this iteration until it reaches the end of the 3D-object:

1. Compute refractive index of the current point
2. Follow ray in refractive direction for a specific length.

The shorter this length is the more accurate the approximation of tracing a ray through diffuse water will be.

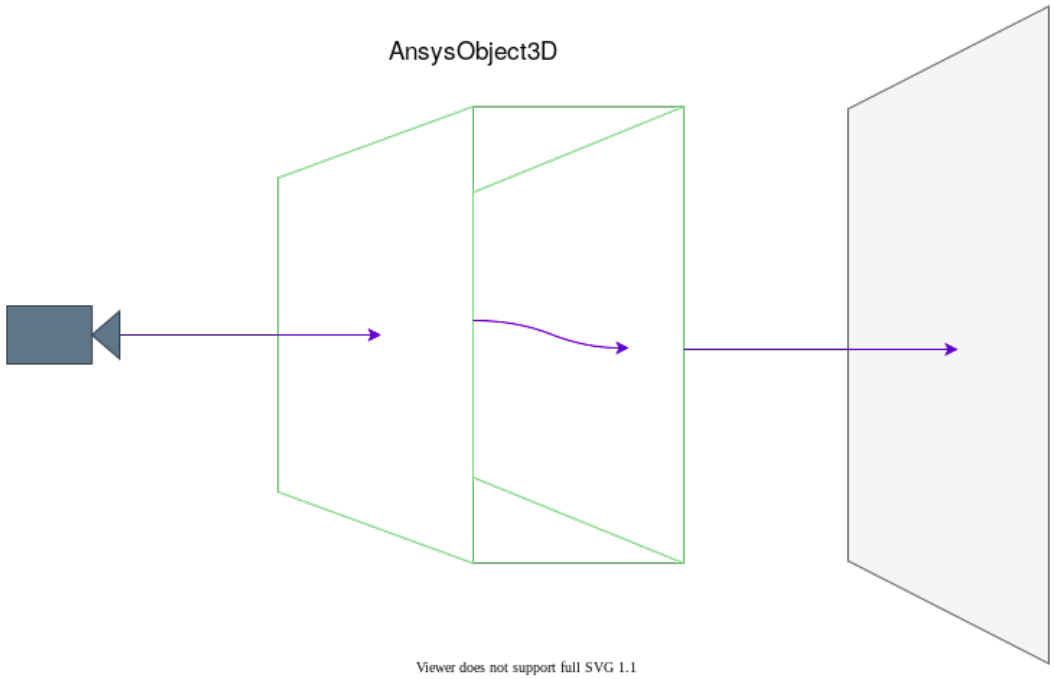


Figure 26: AnsysObject3D in Raytracer

So this is how the Ansys 3D attempt works at least in theory, but in our pipeline the data exported from Ansys needs to be processed before it comes to raytracing.

As the exported mesh is distorted at distances between its nodes and there are missing nodes at some places, the mesh is required to be reconstructed. Furthermore, nodes at some places appear to have zero as temperature value for some reason. Therefore, we apply a repair algorithm to the data, which sets the temperature of nodes that are zero to the mean value of the surrounding nodes. This happens iterative, so that repaired nodes influence other nodes that don’t have neighbours with a temperature higher than zero, until all nodes are repaired. One more option that can be applied to the 3D-grid which gets raytraced is the reduction of grid resolution. This can be very useful to lower the rendering time of frames with a AnsysObject3D.

The following picture is an output frame of the AnsysObject3D.

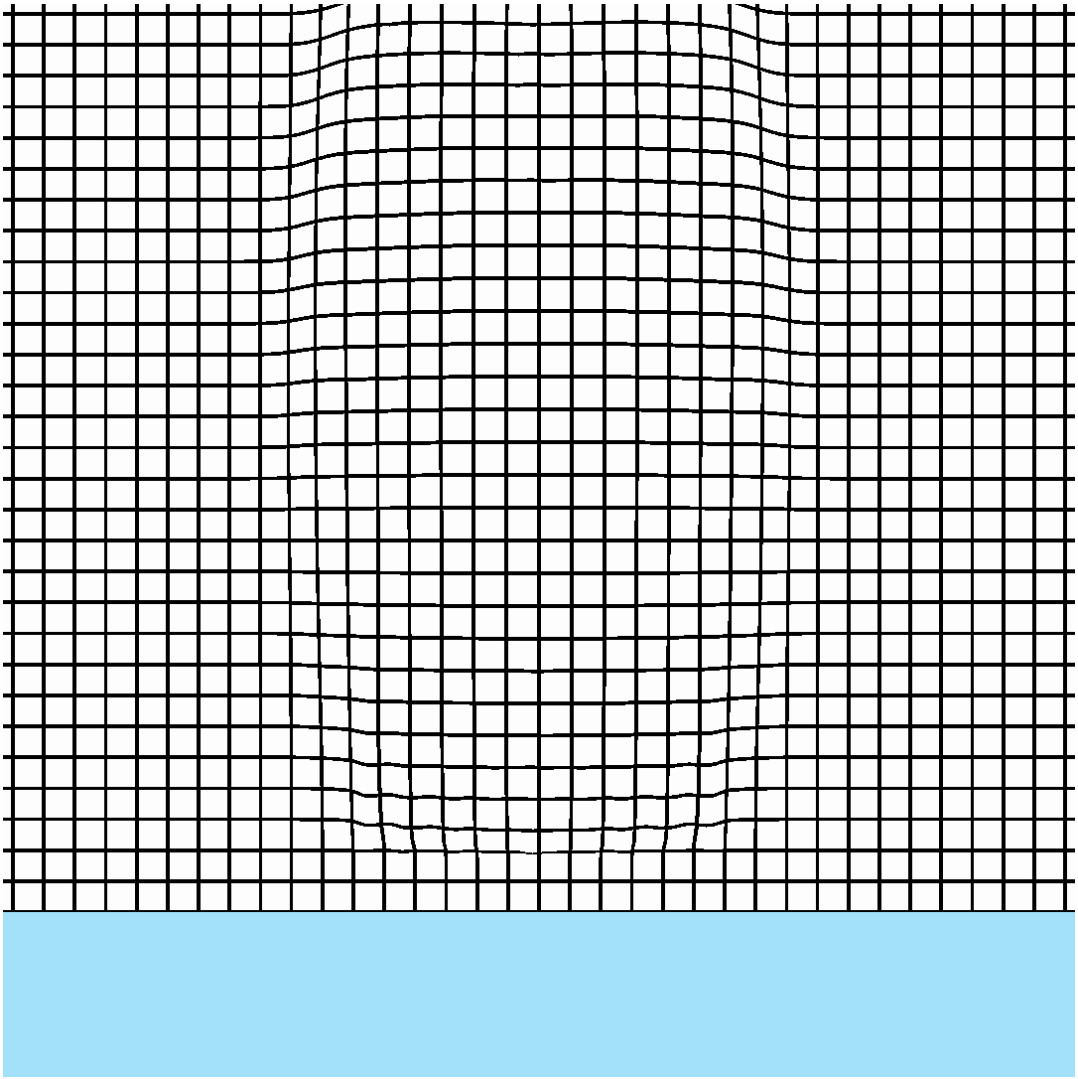


Figure 27: AnsysObject3D output

4.6 Run-Time Tests

Raytracing is a computationally intensive process. In this chapter, we describe how long it approximately takes to render a frame with our raytracer. The calculation time depends on how many objects are placed in a scene and on the amount of pixels to be rendered.
Due to the fact that during our project work the raytracer was not optimized regarding the run-time, rendering a scene may take a very long time. Some tests have shown how long the rendering process takes in dependence of the objects of the rendered scene. All tests were done on a laptop with a i7-8550U (4x 1,8 - 4,0GHz). For each test, one object was placed in the middle of a scene, covering one third of a frame with the resolution 1280x720 (720p).

SceneObject	Time	Note
Rectangle	2s	-
Triangle	2s	-
Sphere	8s	-
Image	4s	-
AnsysObject	4m 6s	consisting of 90000 (300x300) temperature points
AnsysMultObject	1h 20m	consisting of 9 AnsysObjects, each with 90000 temperature points
Ansys3DObject	2h 16m	11.4 million (225x225x225) temperature points

Table 8: results of the raytracer run-time test

It is to be noticed that the calculation time of the AnsysMultObject is already optimized by not checking each AnsysObject for intersection, but only checking the foremost one. Only if it is intersected by a ray, the next AnsysObject will be checked for intersection, too.

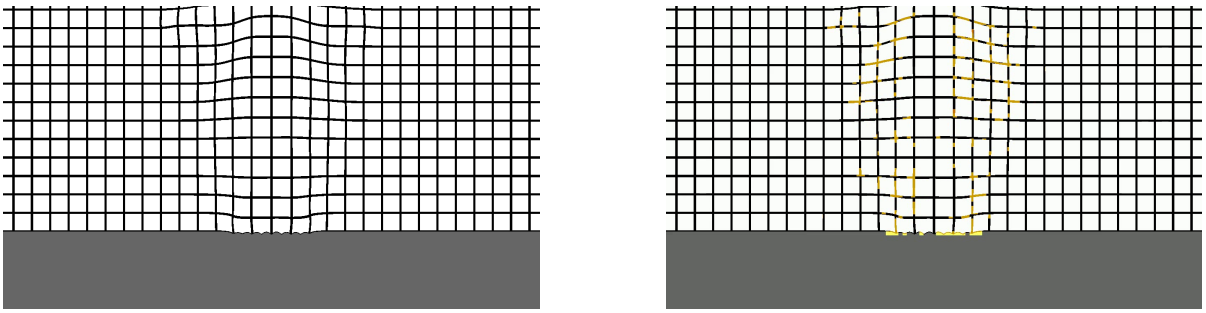
Because of the very long time that it takes to render a scene with an Ansys3DObject, certain optimizations were made to reduce this time. Otherwise, it would take several days to create a short simulaton video containing an Ansys3DObject. For example, the resolution of the three dimensional temperature points was reduced from 11.4 million to 185000 temperature points. Another aspect is to use a small depth because the deeper an object is, the more steps it takes for the ray to leave the object after intersection.

4.7 Evaluation

Many of the created Ansys simulations did not bring the expected result when importing them into our raytracer. The result always depends on how small-meshed a simulation was set. But it also depends on the width and the height the Ansys object was given in the raytracer and also where in the scene it was set, near to the camera or far away from it. These aspects have a big effect on how the simulated Schlieren may look like after the raytracing process. After a few tries we found a good way to import Ansys simulations with fitting parameters.

Since the idea of this subproject was to provide simulations which may be used as input for the detection software, which can be compared with the experiment results, we raytraced the Ansys simulation described in in front of the same background used in the experiment. The simulation looks similar, even though we could not recreate the exact same behaviour, as explained in Section 3.5.2: Comparison to simulation

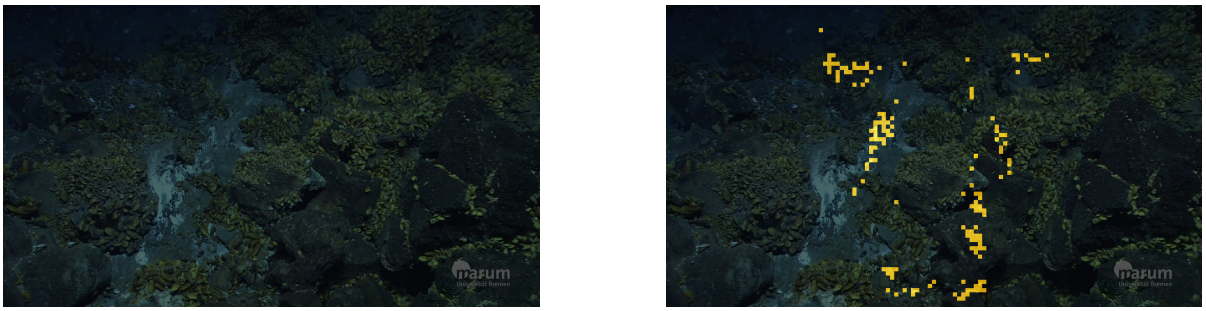
The Schlieren effect in our simulation can be seen due to the black mesh in the background. The detection software noticed these Schlieren movements, but because of the big white areas where the various refracting indices have no impact, the Schlieren effect was only noticed along the black grid.



(a) Simulation (b) Detektion

Figure 28: Comparison of detection and simulation

That made us realize that such grids are not well usable for computed simulations. We raytraced the same simulation in front of an image of the deep sea and again used it as input for the detection software.



(a) Simulation (b) Detection

Figure 29: Detection result of simulated vent in front of deep sea background image

4.8 Conclusion and Outlook

After finishing the project we can conclude that it was the right choice to build a new raytracer. That made it possible to adjust the rendering process for our needs and to easily create new classes, for example for OpenFOAM or Ansys Fluent objects, when needed. Furthermore, the total understanding of the whole program was often useful when it came to problems. Nevertheless, there is still much to do to improve the raytracer and the simulations.

The raytracer brings everything needed to render a simulation video considering imported CFD data. Only the time it takes to render a simple scene can be very long, depending on the amount and size of the scene objects and on the amount of the nodes of an Ansys object. Therefore, the efficiency brings an opportunity to further improve the raytracer. One method would be to render the frames via graphic card. Another method we already started to use at the end of our project would be multithreading.

Due to the fact we had no prior knowledge about the physical behaviour of fluids in our group, it took very much time to try out different adjustments to get good results when it came to work with the CFD tools. Because of the long time it takes to calculate a solution in Ansys, often hours were spent waiting for a solution which in spite of new adjustments was little or not at all better than the prior one. Therefore, though we now can provide simulations of a diffuse seafloor vents, they bring the opportunity to be improved by further work. Especially the selection of the models, which we chose based on our research, may be enhanced by people with a better understanding of the physical basics. Our raytracer might for example be used for future projects to import enhanced CDF data. The possibilities of Ansys Fluent and other CFD tools are huge, and with enough time and a better physical knowledge the simulation can surely become even more realistic.

5 Optical detection

5.1 Detection introduction

This section describes the process of developing an algorithm, that detects Schlieren objects in a given video or live stream source. Responsible for this section are the following students: Arkadiusz Guzinski, Timo Hoheisel, Kilian Lüdemann, Niklas Masemann, Dennis Riemer.

Within the project group, this algorithm is first used to evaluate the results from both the experiment group and the simulation group, and secondly it is evaluated by previous mentioned groups. This way all the students participating in this project could cooperate and work together at some point.

As a little overview, we will begin by explaining how we got started. Continuing with our First approach and our Learnings while experimenting with it. With this new knowledge we searched for alternatives and found a better suited algorithm for our application. Following this, we break down our Algorithm and explain how the final implementation works.

To verify our work, in section Results we compare our results with the different sources of material we were presented with, including the material from the other two subgroups experiment and simulation. Followed by that, in section Evaluation we evaluated our algorithm and how we handled the task we were given.

5.2 First approach

Before working with actual videos, we decided to start with a predetermined sequence of single pictures. Our first approach was to find Schlieren objects by running an optical flow PIV-algorithm (Python Particle image velocimetry). This algorithm gave us a velocity field between 2 frames. We then tried to extract the regions, where Schlieren objects had a significant impact in the velocity field. After successfully running this program on a sequence of frames, we then wanted to convert it to work on videos instead. The live stream functionality were planned to be implemented in the end.

5.2.1 Python Particle image velocimetry

The Python library for Particle Image Velocimetry (PyPIV) is an optical-flow algorithm for calculating a velocity field between 2 frames. It does so, by partially taking subframes out of each picture like a grid (windowing). The brightness field in the same sub frame between two frames is then correlated.

The position of the correlation peak gives the most likely shift of the brightness field from one frame to the next. This peak is fitted with sub pixel accuracy. The velocity field is calculated by dividing this position by the time between the frames. The resolution of the velocity field depends on the size and shift of the subframes. (This algorithm was developed by the

department of geophysics at Georg-August-University Göttingen (Ruebsam and Luedemann, 2019).

5.2.2 Learnings

At first the Python library for Particle Image Velocimetry (PyPIV) algorithm was a good choice for us to get into the project and understand the given context. But for the detection of underwater Schlieren objects this algorithm turned out not to be the best option. There were to many artifacts produced by the general dark environment (deep sea) and shadows on the foreground. The movement of the ROV was also at some point to great for the algorithm. We tried to come up with possible solutions for these problems, but we figured out it would become increasingly more difficult with this approach. In the following, these problems are further described.

5.2.3 Attempts of adaptation

Deep sea as background As mentioned, the PyPIV did not only detect Schlieren objects, but also had its problems with the deep sea as background. We tried to use the advantage of the darkness of the deep sea and applied Open Source Computer Vision Library (OpenCV) threshold functions to simply remove it.

Therefore we used simple thresholding with a "to zero" method. This removed the deep sea and carved out shadow-regions, which caused more problems, as it solved. Shadows under corals for example left dots, which then were interpreted as shift between pictures.

Furthermore the overall brightness in the deep sea varied too much, so that we could not choose the perfect threshold and changed to the OpenCV function otsu's binarization (OpenCV-Python-Tutorials, 2013d), which calculates a threshold between 2 peaks in a brightness histogram. This improved the results, but only for examples where two peaks existed. For very dark examples, it did not work at all, which made the general use of it impractical.

Next we tried to find a contour which should have separated the deep sea from the static foreground. Because of the nature of the objects in the foreground, which were mostly rocks, corals and bacteria, there were a lot of contours and edges. Due to this, the calculation time was very long and no contiguous contour or edge to the deep sea could be usefully calculated. The functions used for this were the openCV-functions for calculating contours, canny edge detection and image gradient computation (OpenCV-Python-Tutorials, 2013c).

Thereafter we tried to use the characteristic color of the deep sea to remove it, by applying a backprojection function given by openCV (OpenCV-Python-Tutorials, 2013a). This caused 2 problems. On one hand, we had to give an example of color distribution, which were problematic, because the bluish tint of the videos and so the deep sea background varied. On the other hand the tint made a clear color separation of the deep sea and the foreground impossible.

After that, we tried to remove video fragments, by applying image smoothing functions by OpenCV, which resulted in only a small difference. Finally, because of the overall dark theme

of the deep sea, we applied the contrast limited adaptive histogram equalization function Contrast limited Adaptive histogram equalization (CLAHE) by openCV (OpenCV-Python-Tutorials, 2013b). It is used to increase the contrast in an adaptive way, which helps especially in the border regions to the deep sea background. Because of it's adaptability, it does not overexpose the very dark regions and by that avoids producing new sources of error. This improved almost every aforementioned function and we came to the conclusion, that in our case, the clahe function in combination with a simple thresholding "to zero" method worked best for the moment. The effect of the CLAHE function can be seen below.

(a) Original image (b) Applied CLAHE

Figure 30: This is an example of how the CLAHE function brightens the dark areas

Shadows The PyPIV algorithm used gray scale images for its computation. Which is why shadows caused by the light sources of the ROV affected the computation. The light source seemed to be placed beside the camera. This caused the shadows thrown by the Schlieren objects, to flicker on the other side and therefore adding more detected spots.

We came to the conclusion, that a light source shaped like a ring around the camera would throw the shadows behind the Schlieren objects, increasing their probability of being detected by the Particle Image Velocimetry (PIV) algorithm.

Camera movement Of course there is a general shift between two frames, when the point of view moves through an environment. We first thought of removing an adaptive value constantly from the result of the computation. After that, a threshold was applied, to leave only the regions, where Schlieren objects had an impact. This is not a part of the actual algorithm.

Anomalies We found out, that the PyPIV algorithm and some filters like the simple threshold caused noise in areas, where no shift were expected. To fix this, we ran a second computation with the second and a third frame and simply compared the result areas. This removed the noise and some outlier.

5.2.4 Reconsiderations

As mentioned before, we came to the conclusion, that the PyPIV algorithm was not the best choice to work with. So in search for alternatives, we decided that the Horn-Schunck algorithm is probably better suited for our problem (Bradley Atcheson, P, and Ihrke, 2008). Examining this in practice proved the previous statement to be correct.

5.3 Algorithm

For the implementation of the algorithm and the graphical user interface, we used python. A pseudo code visualisation can be found in section Sequence diagram.

5.3.1 Software-pipeline

The following is a software-pipeline of the below described algorithm.

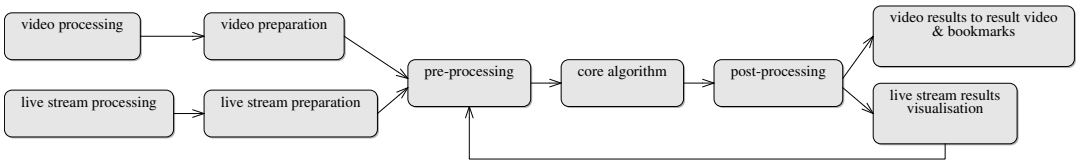


Figure 31: Software-pipeline for our algorithm

5.3.2 Preparation

First, we distinguish between preparation of videos and live streams. For a video, we have no time limits with the computation. With the live stream however, we must be either fast, or reduce the rate of calculations per frames, to achieve real time evaluation.

Video To get the images from the video for the algorithm, we used the functions which are given by OpenCV. When beginning the computation, the video is loaded and the start frame is set. This is important when we have more than one video decoding process for higher core counts. After that the video decoding process will take the first three frames of the video and stores them temporary into variables. When the three frames are loaded, a triple of the frame number of the first frame, the first frame and the third frame is put into a python multiprocessing queue. We're using the first and third frame, to get enough difference to detect, assuming a 25 FPS video. Whilst the queue expands, the frames of the queue members are always spaced by one, so all even and all odd frames are processed together. This functionality can be seen in section Sequence diagram.

Live stream When using a camera as input, the same detection algorithm is used. However only a few frames are used for detection. Then the result overlay is computed and used on every frame until the next result is available. For a more detailed description of the live stream processing branch, see section Sequence diagram.

5.3.3 Pre-processing

The pre-processing area of the main computation starts with applying the CLAHE function to both the first and the second frame, of each computation task, which can be seen at picture two of Figure 38 to Figure 39. This seemed to improve the performance of the Horn-Schunck algorithm.

Then we use a very simple way to detect movement between the two frames, by using the OpenCV function template matching (OpenCV-Python-Tutorials, 2013e). The idea behind this is to leave only the Schlieren effects between the two cutouts. A rectangle with a 10x10 padding is taken from the first frame. This is then matched within the second frame and gives fitting cutouts. If no matching area is found within the second frame, the next section will be skipped, saving computation time. For example this is the case, if only deep sea is in the view.

5.3.4 Core algorithm: Horn-Schunck

We now take our two fitting cut-outs and apply the Horn-Schunck algorithm to them. This calculates a velocity field between the frames. It starts with the assumption of no movement between the frames. If there is movement, the algorithm iteratively approaches the correct velocity field. We decided to use 8 iterations, because the result seemed to not significantly improve with more iterations and therefore we could save computation time. Horn-Schunck also applies a smoothing value alpha, where we chose 2. This seemed to work as desired and therefore no further adjustments were made. This results in two arrays, for the horizontal and the vertical velocity per pixel. By comparing these with the results from the PyPIV, we came to the conclusion, that the Horn-Schunck algorithm seemed to not have the same complication with the deep sea background and also less complications with shadows. The exact functionality can be seen in the original Horn-Schunck-paper (Horn and Schunck, 1981).

5.3.5 Post-processing

Thereafter a threshold is applied, to remove noise and small general movement. This threshold was chosen to be five plus an adaptive value, which arises from the shift between the two cut-outs. We found that bigger shifts between the frames resulted in not only shifts, but also zooming and perspective change. To take that into account, we increased the threshold by the sum of the horizontal and the vertical shift. This implies, that the algorithm is more accurate, when the movement of the point of view is low.

We found, that the algorithm produces anomalies on the edges, which may be caused by the aforementioned movement-detection function. To fix that, we simply cut the edges off by 5x5.

Clustering The resulting arrays are now distributions of points in regions of interest. These points need to be clustered for a good visualisation. We found, that established cluster algorithms would not work, because we had too many, not very separated clusters. It was more like we had one cluster, which had to be summed up and some outliers, which had to be removed. This would have been possible by declaring a maximum distance between points and therefore ignoring outliers, but we found no suitable and sufficiently fast implementation.

Groups To reduce the amount of memory-space that is used at runtime and accelerate the post processing, we decided to cluster the frame into smaller blocks, we called groups. To determine the size of the groups, we use the greatest common divisor of the frames height and width. In case the groups would be to big, a loop tries to figure out a divisor for the greatest common divisor in range from 30 to 50. After the group size has been calculated, all group elements are binarized by the trashhold of 0. Finally, the values are added to a number representing the group. Thereafter we defined an average function of how great the represent of the group needs to be as to count as a hit and all values under this average will be set to zero. The resulting 1d-array is now applied as a mask to another 1d-array consisting of original result values, which are again summed up to representatives. In Figure 38 to Figure 39 picture 3, the resulting image of the groups are shown, where a more white color represents a higher density of high values.

5.3.6 Visualisation

When a video part is ready to generate, every value of the grouped image will be replaced with a placeholder value. The threshold where the different values are replaced is calculated with the average of all groups. A higher placeholder defines stronger Schlieren effects. After the placeholders are set, the overlay image is converted to BGRA color space and the placeholders will be replaced with their actual color. To use the overlay image, we needed to convert it to its original shape. To do that, the pixel that represent the group are replicated by the greatest common divisor that is used in the group's function. While the lost pixel at the edge are padded with zeros. In the end the overlay image is set on top of the original image and is converted to BGR color space for better handling the video creation. In Figure 38 to Figure 39 picture 4 the final result is shown.

5.3.7 Graphical user interface

Figure 32: Image of the GUI

The GUI is one way to use our algorithm. It is written in python 3, using the Qt-framework[1]. In the upper left corner, an input can be chosen. After choosing such source, a preview is shown below. When a video is chosen for example, you maybe do not want to process all of it at once. Therefore in the upper right corner, you can set a begin- and an end time, between which the algorithm will calculate the result. If desired, you can queue up more than one video. For the cli preferring user, the algorithm can also be started without gui.

At the time of writing, the livestream feature is only available per command line. This is likely to be integrated into the UI soon. Same goes for video processing via command line interface.

5.3.8 Bookmarks

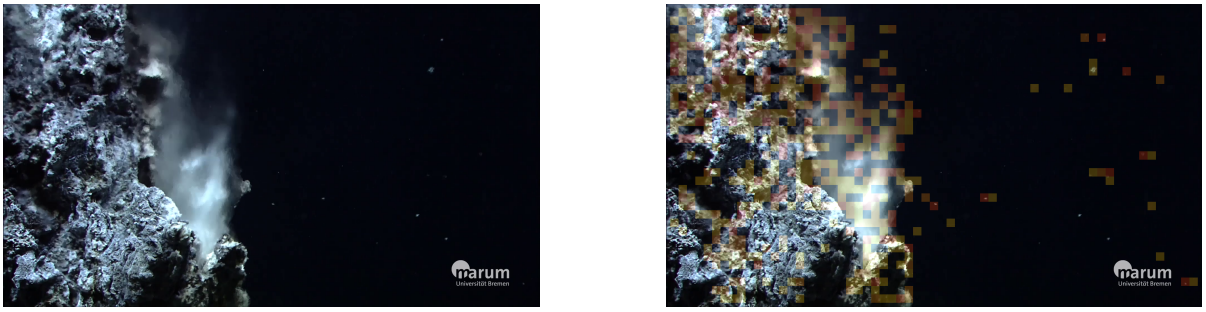
In order to improve the ability to keep track of the results, we decided to implement a bookmarks function. It generates a list of tuples, which indicates the start and end of an interval

[1] tested with Python 3.6 & 3.7, using PySide2

of frames with Schlieren objects. With this function implemented, the user is able to check the results of our program on another way and can also save the result data somewhere else. So to speak, as a safety measure, if the result video gets lost or corrupted.

5.3.9 Features

As we ran the algorithm, we found out, that we had not only detected direct vent and Schlieren objects, but also created a particle tracker. This seemed to be a problem at first, but came out as a feature. This effect can be seen below.



(a) Original image (b) Results

Figure 33: A visualisation of the particle tracker feature

5.3.10 Compare core algorithms

We can now compare the PyPIV and the Horn-Schunck algorithm. As mentioned in section Learnings, the PyPIV teached us, that in the deep sea, it is complicated to work with the varying brightness and the absence of other colors than blue. For both our algorithms it is important to generously illuminate the regions of interest. We came to the conclusion, that because of the different functionality of the Horn-Schunck algorithm, it were not as much affected by the effects of the deep sea background as the PyPIV. Therefore we were able to get rid of this problem rather simply.

Performance comparison In order to increase the accuracy, after a first calculation (called “directPIV”), another step is run through (called “adaptivePIV”). This can be extended by more such steps. Each “adaptivePIV” step increases the computation time exponentionally. For satisfactory results, at least one “adaptivePIV” had to be run. Hence, the PyPIV is not suited for our application.

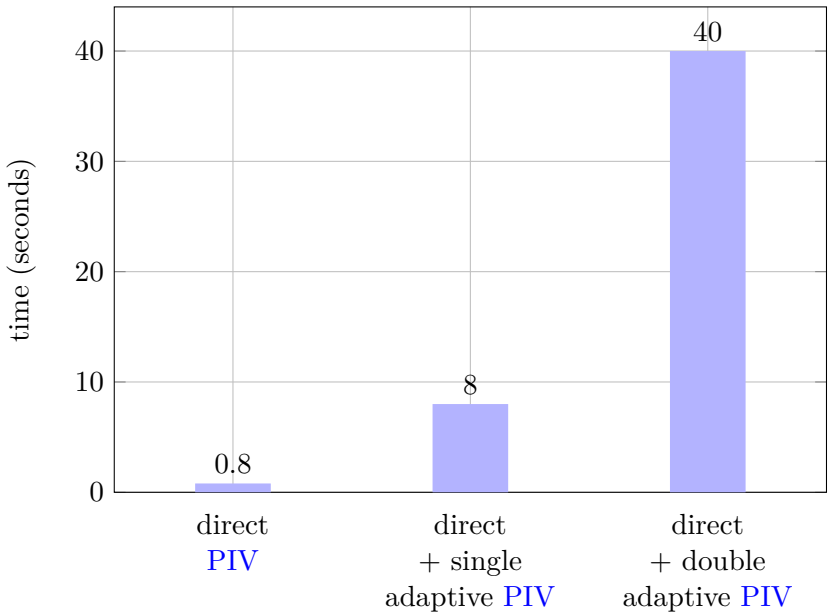


Figure 34: Performance diagram for FullHD resolution images with different PyPIV configurations

In order to increase the accuracy, the Horn-Schunck algorithm is run in several iterations. With increased iterations, the computation time does not increase as fast as the PIV does. As mentioned, we found out, that 8 iterations would be enough to compute satisfactory results. More iterations would increase the computation time without significantly increasing the accuracy.

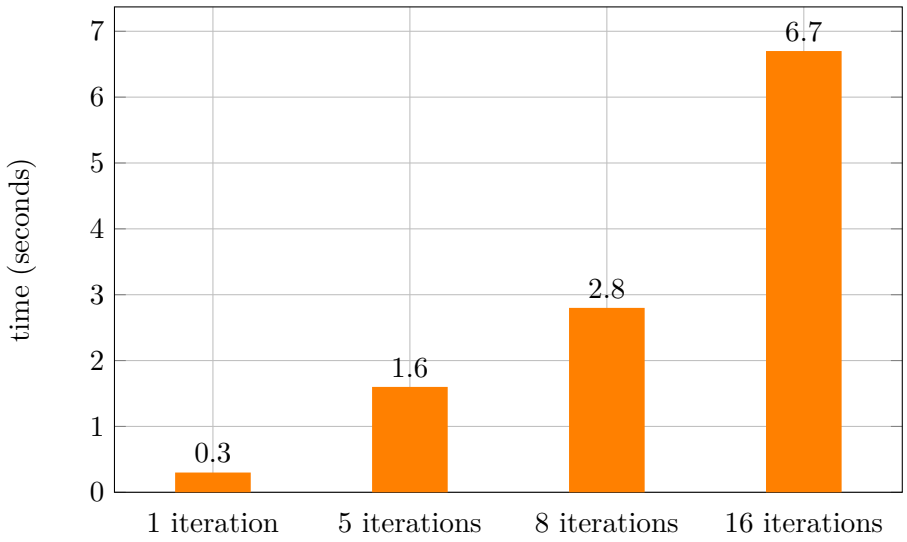


Figure 35: Performance diagram for FullHD resolution images with different Horn-Schunck configurations

These calculation times were measured, as the algorithms were run on an AMD Ryzen 7 2700X.

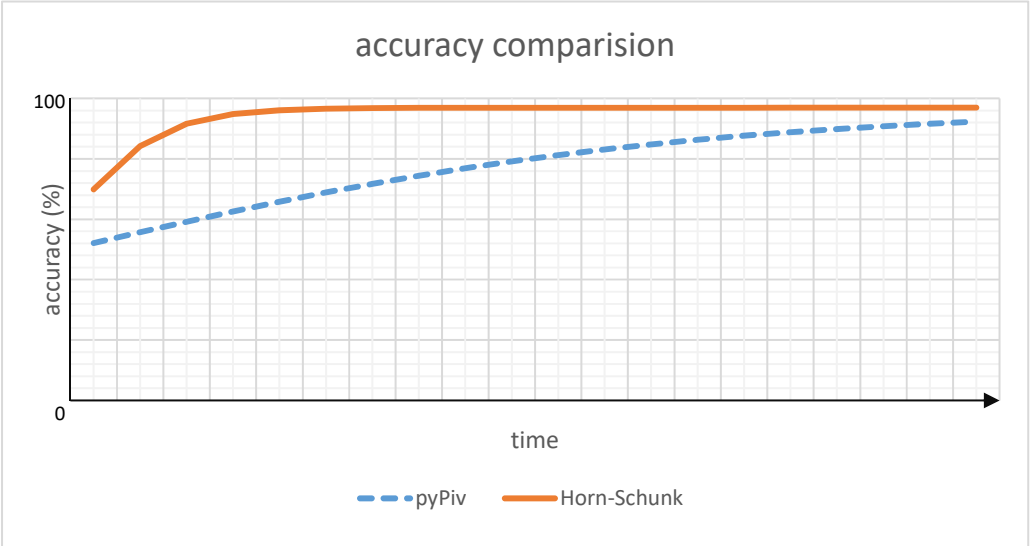


Figure 36: An accuracy comparison between PyPIV and Horn-Schunk

This diagram visualizes how the accuracy improves with more iterations. As can be seen in the diagram, the Horn-Schunk algorithm reaches a preferable accuracy in less computation time than the PyPIV.

5.3.11 Sequence diagram

The following is a sequence diagram of the complete algorithm. It is branched into the processing of a video and a live stream.

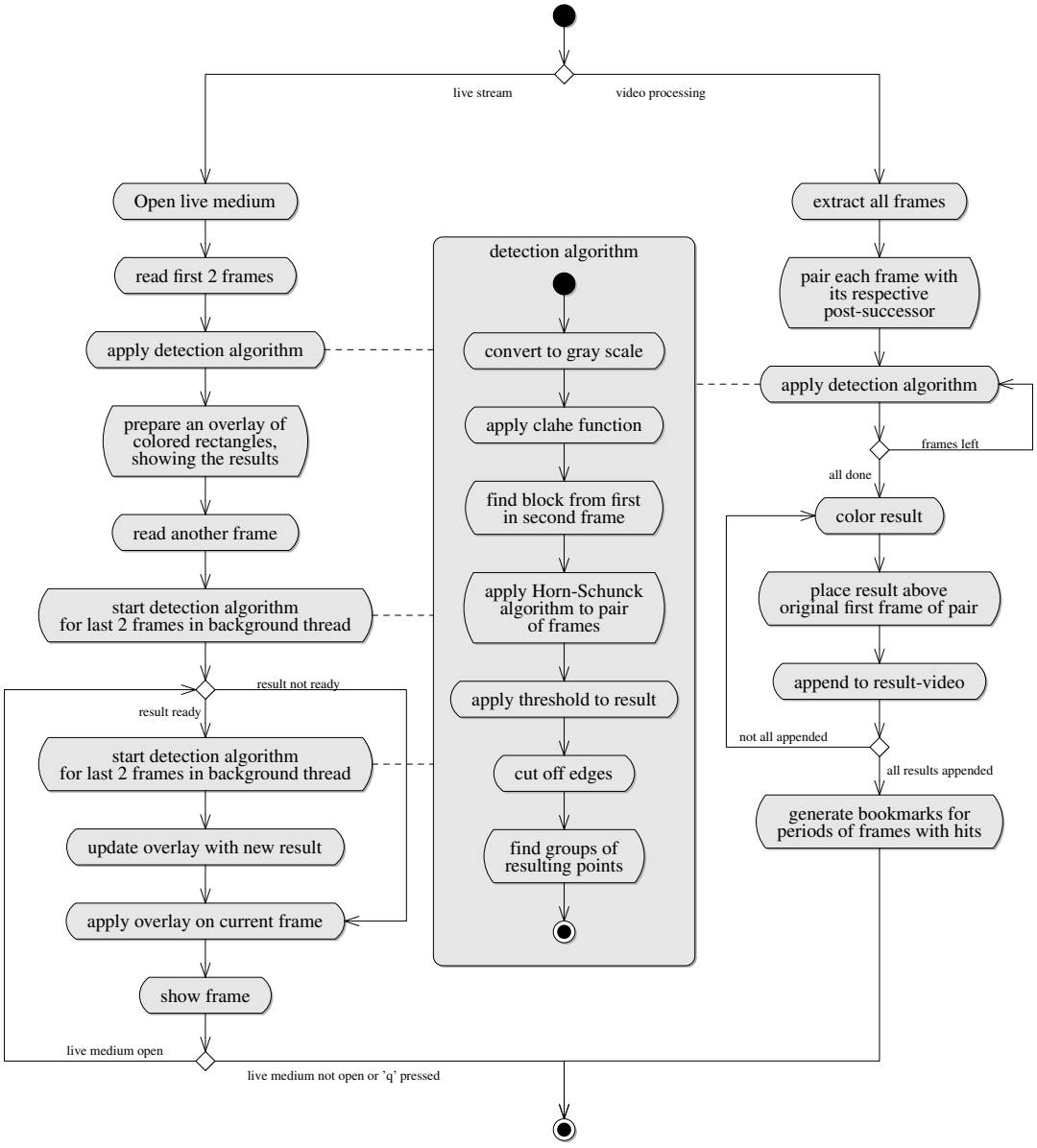
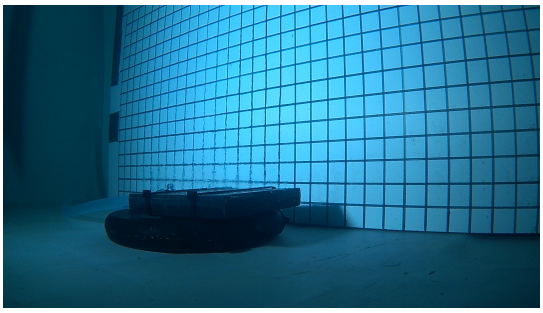


Figure 37: Sequence diagram for our algorithm

This way the original live stream is always visible in its current state, with the last computed results as an overlay. The downside is, that the results shown are actually delayed, and therefore a bit older than the visible picture. Currently this delay can be around 3 seconds with FullHD resolution.

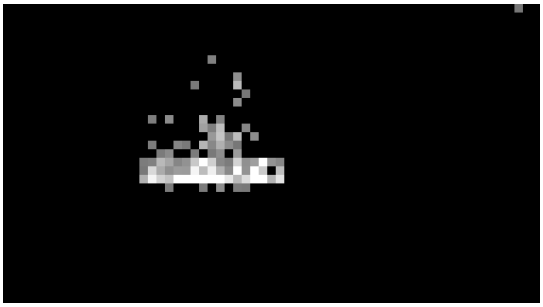
5.3.12 Processing chain

The processing chain consists of 8 steps. The most important are shown below.
The computation starts with (A) the original image.
After that, the image is converted to grayscale and a CLAHE function is applied (B).
The images are paired with their post-succesor and a block-matching function is used to get fitting cutouts between the two frames. This is thought to roughly get rid of a general shift between the frames.
These fitting cutouts are given to the Horn-Schunck algorithm, which provides us with a velocity field between the cutouts.
To the velocity field, a threshold is applied, which leaves us with a “point cloud”.
This “point cloud” is then grouped into squares (C), which represent regions of interest (This function is explained in Section 5.3.5).
In the end the regions of interest are coloured and presented as an overlay over the original image (D).

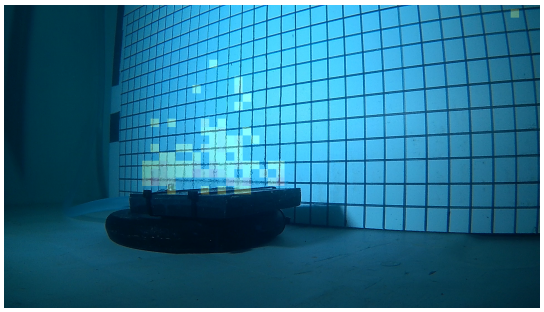


(a) Original image

(b) Applied CLAHE

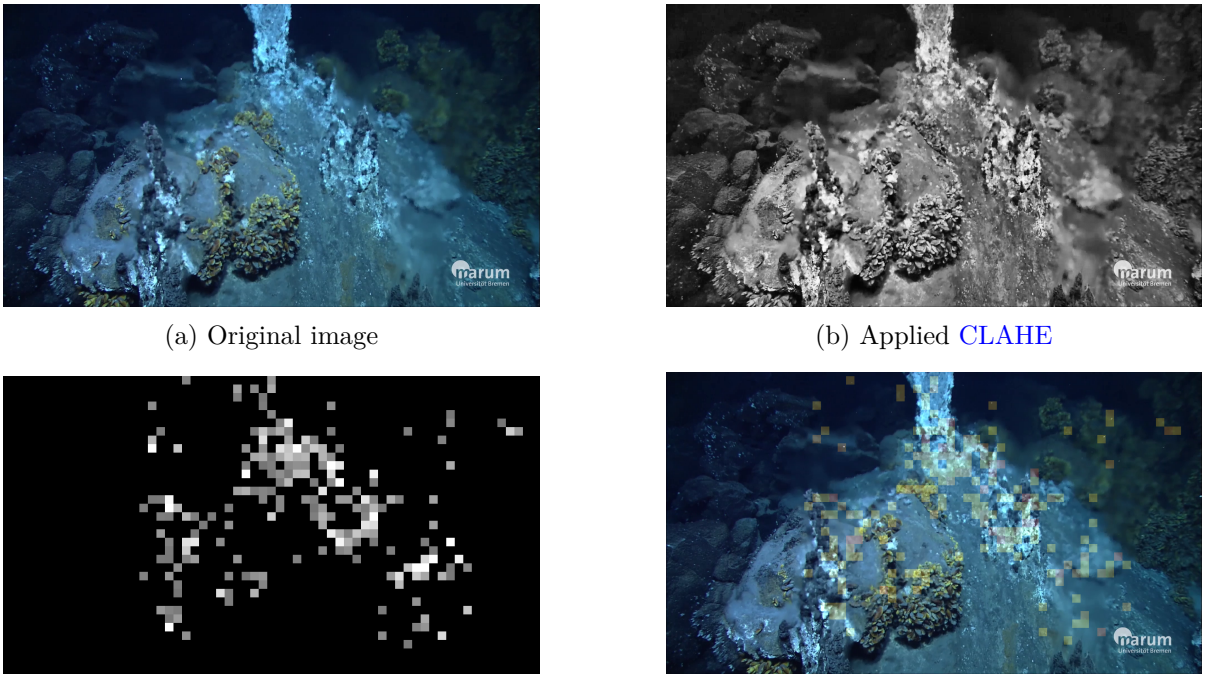


(c) Grouping



(d) Results

Figure 38: Processing chain for experimental video



(a) Original image

(b) Applied CLAHE

(c) Grouping

(d) Results

Figure 39: Processing chain for deep sea video

5.4 Results

In the following section we compare our results with the different sources of material which we were getting.

5.4.1 Experiment

The subgroup experiment provided us with a varying source of artificially created Schlieren objects. In the beginning, we used the PyPIV for evaluating this material, because the exchange with the Horn-Schunck algorithm came rather late. This resulted in very inaccurate results for detected regions. The PyPIV detected some Schlieren, but was not suited for differentiating between Schlieren objects and other effects, like interference caused by moving water surface and reflecting surfaces withing the aquarium.

Hence, the change to the Horn-Schunck algorithm improved the results significantly. Thereafter, our ability to evaluate the quality of experiment's artificially created Schlieren objects enhanced. For this reason, we came to the conclusion, that the Schlieren effects were preferably, when farther away from the background and closer to the camera. Furthermore, the last experiment contained less sources of interference, which improved the results.

5.4.2 Simulation

Based on the nearly perfect conditions, like a static background in the simulations, we had the possibility to verify our algorithm on Schlieren object-only videos. The results satisfied our expectations and verified the basic functionality of our algorithm. Regardless, it showed us that a video that contains only Schlieren objects is not sufficient for our field of application, because it would be extremely difficult for us to create comparable circumstances in the deep sea. Nevertheless, to bring the deep sea circumstances closer to the very good simulation results, we propose that adjusting the illumination of the Schlieren objects (as mentioned in the section Shadows) by a ring light around the camera, or setting the viewpoint to a higher angle, which should result in a more static background, could lead to better results.

5.4.3 Deep sea

As mentioned before, the change to the Horn-Schunck algorithm improved the results for the deep sea videos as well. The complications of deep sea photographs were described within section Attempts of adaptation and Compare core algorithms. We found out, that our algorithm performs the best, when the region of interest is generously illuminated. The area of Schlieren objects is even better detected, when there are no microorganisms floating around. Nevertheless, we value the feature of tracking particles, which consists of bacteria and minerals. Furthermore the effects were detected more accurately, where there was a static background behind the Schlieren objects. As mentioned before, our algorithm was proven to be more accurate, when the ROV movement was exiguous.

Superior results were computed, when all above mentioned conditions were fulfilled.

5.5 Evaluation

To evaluate our algorithm, we can most likely take the experiment videos as ground truth data. When computing the results for these, we evaluate our algorithm as meeting our expectations. However, when expanding our tests into the deep sea videos, we discovered aggravating circumstances, which turned out to be handled mostly well. Nevertheless, we could not solve the problem with motion-based perspective change and zooming.

The decision to use python turned out to be suitable for the most purposes within this project. Nonetheless, at some points python did not offer us enough range of functionality. This caused problems with multi-processing and multi-threading in particular. Hence, the alternative of C++ as programming language should be considered.

Our choice for Horn-Schunck as optical flow algorithm turned out to work in most cases, but it is worth considering further alternatives. Lastly, the openCV library was very valuable for us.

5.6 Conclusion

The goal of this work was to develop an algorithm to automatically detect and visualize Schlieren objects on underwater photographs.

During the development we encountered obstacles, which were both technical and location related, like varying brightness or areas with a higher amount of floating microorganisms. The requirement to be able to process live streams and at the same time detect and visualize Schlieren objects as accurately as possible was one of the major technical challenges. The common clustering algorithms for image processing, were excluded from a temporal perspective, for example.

A further challenge was the lighting, which caused strong shadows, due to the recording at depths where no daylight reaches and the resulting one-sided illumination by the ROV. Another source of interference consisted of the large number of microorganisms or minerals, whose movements caused faulty detection, which afterwards came out as a feature, because marine biology is interested in the origin and movement of those. Other movements, such as those of the camera, could be filtered out much better by means of template matching on the respective image pairs. Apart from that, when working with artificially created Schlieren objects by the experiment and the simulation group, we found out, that our algorithm worked rather great and were able to detect almost every region of interest. However, because the goal was to detect Schlieren objects in deep sea regions, we encountered aforementioned complications.

These complications cause, that the detection and the associated visualization of Schlieren objects in deep sea photographs is too inaccurate to provide a desired result without further control, but it could be used in conjunction with manual detection to highlight or further narrow down interesting areas of the image. Nevertheless, we believe that the program offers great potential for further development and is already a good tool for enhancing the material to be sifted and thus facilitating further processing.

6 Conclusion and Future Work

As mentioned in the separate sections, we have achieved a lot during the project work, though there are possibilities to improve the result of each subgroup. While the detection software worked very well for the experiment results as well as for the simulation results, there were complications when detecting Schlieren in real life deep sea pictures. That shows that on the one hand, the basic goals for each subgroup have been achieved. The simulation group, and the experiment group were able to provide pictures and videos of Schlieren which were detected by the detection software. On the other hand it is necessary to further improve the Schlieren recreations by experiment and by simulation so that they become more similar with real deep sea vents including the complications they caused. Also, the detection software has to be improved to deal with several disruptive factors so that real deep sea Schlieren will be detected without any problems.

All in all, this project brings the opportunity to be further pursued and to improve the results we could already achieve.

7 Acknowledgements

We would like to thank Gabriel Zachmann for providing high performance hardware which supported the heavy simulation computations.

Another acknowledgment goes to MARUM, Ralf Bachmayer and his group who supported us in conducting the experiments, especially Arne Kausche and Philipp Koschinsky. We also have to thank the "AG Meerestechnik", who helped us conducting the experiments in the larger tank by providing the tank, a work place and tools.

Thanks to the department of geophysics at Georg-August-University Göttingen for providing us with knowledge and the PyPIV.

Acronyms

AHE Adaptive histogram equalization. 56, 70

Ansys Analysis System. 39, 40, 42–46, 48, 49, 51–53, 72

ASCII American Standard Code for Information Interchange. 45

CFD Computational fluid dynamics. 4, 29, 37, 39, 40, 46, 47, 52, 53, 72

CLAHE Contrast limited Adaptive histogram equalization. 56, 58, 65, 66

CUDA Compute Unified Device Architecture. 39

FPS Frame per Second. 9, 57

GPGPU General purpose Graphics processing unit. 45

GPU Graphics processing unit. 39, 45, 70

GUI Graphical User interface. 40, 60

MARUM Center for Marine Environmental Sciences. 6, 72

OpenCV Open Source Computer Vision Library. 55, 57, 58

OpenFOAM Open-source Field Operation And Manipulation. 39, 40, 46, 52

OpenMP Open Multi-Processing. 38

PIV Particle Image Velocimetry. 55, 56, 62, 70

PLA Polylactide. 16, 27

PPM Portable pixmap format. 31

PU Polyurethane. 9–11

PyPIV Python library for Particle Image Velocimetry. 55–58, 61, 62, 64, 66, 69, 73

ROV Remotely operated underwater vehicle. 5, 27, 28, 55, 56, 67, 68

UI User interface. 40, 70

References

ANSYS Inc. (2020). *Ansys Fluent*. https://www.ansys.com/de-de/products/fluids/ansys-fluent. Abgerufen am 01. Juni 2020.

Bemis, Karen, Robert P Lowell, and Aida Farough (2012). "Diffuse flow: On and around hydrothermal vents at Mid-Ocean Ridges". In: *Oceanography* 25.1, pp. 182–191. URL: https://doi.org/10.5670/oceanog.2012.16.

Bradley Atcheson, Wolfgang Heidrich, Robert P, and Ivo Ihrke (2008). "An evaluation of optical flow algorithms for background oriented schlieren imaging". In: *Exp Fluids (2009)* 46.1, pp. 467–476.

Horn, Berthold K.P. and Brian G. Schunck (1981). "Determining Optical Flow". In: *Artificial Intelligence* 17.1, pp. 185–203.

NASA Dryden Flight Research Center (1995). *NASA's SR-71A aircraft taxiing on the ramp at NASA's Dryden Flight Research Center, Edwards, California, heat waves from its engines blurring the hangars in the background*. URL: http://www.dfrc.nasa.gov/Gallery/Photo/SR-71/HTML/EC95-43075-4.html.

OpenCV-Python-Tutorials (2013a). *Backprojection in OpenCV*. URL: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_histograms/py_histogram_backprojection/py_histogram_backprojection.html#backprojection-in-opencv (visited on 09/04/2020).

– (2013b). *CLAHE (Contrast Limited Adaptive Histogram Equalization)*. URL: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_histograms/py_histogram_equalization/py_histogram_equalization.html#clahe-contrast-limited-adaptive-histogram-equalization (visited on 09/04/2020).

– (2013c). *Image Gradients*. URL: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_gradients/py_gradients.html (visited on 09/04/2020).

– (2013d). *Otsu's Binarization*. URL: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_thresholding/py_thresholding.html#otsus-binarization (visited on 09/04/2020).

– (2013e). *Template Matching in OpenCV*. URL: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_template_matching/py_template_matching.html#template-matching-in-opencv (visited on 09/04/2020).

Ruebsam, Jonas and Kevin Luedemann (2019). *PyPIV*. URL: https://github.com/jr7/pypiv/tree/python3-dev (visited on 09/04/2020).

Schön, Georg (1999). *Hydrothermalquellen*. URL: https://www.spektrum.de/lexikon/biologie/hydrothermalquellen/33127.

Scratchapixel (2016). *Introduction to Ray Tracing: a Simple Method for Creating 3D Images*. https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing. Abgerufen am 12. Dezember 2019.

Sokolov, Dmitry V. (2019). *tinyraytracer*. https://github.com/ssloy/tinyraytracer/wiki/Part-1:-understandable-raytracing. Abgerufen am 08. Dezember 2019.

The OpenFOAM Foundation Ltd (2020). *OpenFOAM*. https://www.openfoam.org/. Abgerufen am 01. Juni 2020.

List of Figures