Wildfire Simulation
Scientific Visualization Project Report

Group B

Johannes Kurz, Yingyan Xu, Yitong Xia, Stuart Heeb

June 12, 2023

1 Introduction

Wildfires may cause substantial environmental and property damage without adequate supervision and controls. Among various behaviors of wildfire spreading, the vorticity-driven lateral spread (VLS) is attracting attention from researchers. To better understand the cause and effect between environments and fire behaviors, it's critical to visualize and analyze the simulation data properly.

In this scientific visualization project, we visualize the VLS simulation data with the following contributions:

- Resample original curvilinear data to rectilinear data.
- Direct volume rendering of fire, water vapor and grass.
- Visualize streamlines of wind velocity.
- Compute and visualize vorticity and divergence of wind.
- Isosurface rendering of fire.
- Transfer function editing for GUI.
- Optimize GUI for fluent interaction utilizing preprocessing.
- VLS analysis with Line Integral Convolution (LIC).
- Full animation on the IEEE 2022 SciVis Contest[5] dataset[6].

2 Data

The dataset we use comes from IEEE 2022 SciVis Contest. The dataset is produced by running the Higrad/Firetec simulation that contains two parts: Higrad, an atmosphere hydrodynamics model, and Firetec[3, 4], a multi-phase fire-physics model.

The IEEE SciVis wildfire dataset consists of six scenes, given by the combination of two ignition positions (backcurve or headcurve) with three types of mountain curvature[1] (40, 80 or 320, a smaller value indicates smaller radius, i.e. more pointy).

Each scene contains time series data of nine physical parameters that describe the burning process, namely: `O2` (oxygen level), `convht` (convective heat transfer), `frhosiesrad` (fire-induced radiative heat transfer), `rhowwatervapor` (bulk density of the moisture released due to burning), `rhof` (bulk density of dry fuel), `theta` (air temperature) and [`u`, `v`, `w`] (wind velocity component along three directions). All of them are quantitative scalar data. They are stored in structured (curvilinear) grids that outline the terrain surface. Since the inefficiency of querying curvilinear data becomes obvious when the file is large (1.1GB for each frame and around 70 frames in each scene), it is beneficial to convert it to rectilinear data for further usage.

3 Goals

Our main goal was to make use of what we learned in the lecture in order to produce a visualization that allows to better understand the VLS of wildfires.

Our project can be categorized by the following timeline:

1. Gain an understanding of the dataset.
2. Explore data in ParaView[1] to determine what we can and want to visualize in order to convey meaningful information.
3. Produce visualizations in ParaView.
4. Implement visualizations in VTK[7] using Python.
5. Optimization for and implementation of interactive GUI rendering including transfer function editing.
6. Analysis of VLS behavior using our visualizations.

[1] We use "curvature" interchangeably with "radius of curvature"

4 Visualizations

In this section, we will introduce the implementation details for each module of our project.

4.1 Data Preprocessing

After our first implementation of a graphical user interface (GUI), which is described in Section 4.8, in which we wanted to be able to step through frames interactively, we noticed that computation time was too long. It took approximately 18s to render a frame completely. The reason for this was that we were using `vtkXMLStructuredGridReader` to read the full data.

As a first improvement approach, we preprocessed each `.vts` file (frame) into a `.vti` image, for the attributes where it was possible (a single `.vts` load is still required on startup). We then reused these preprocessed `.vti` files in our interactive GUI, which brough the render time down to around 5s per frame. We did not feel as though this was fast enough to ensure a interactive experience, so we considered other ways to speed this up.

To this end, we implemented a preprocessing procedure, which can be found in `preprocessing.py`. It preprocesses the data attributes into a `.npz` file using `numpy`[2]. Loading these saved values from this file format to render the scene in our GUI takes less than one second, which is satisfactory for what we wanted to achieve.

4.2 Resample To Rectilinear Data

The original curvilinear data has an extent of $(600, 500, 61)$ along three directions. It is resampled to rectilinear data of extent $(300, 250, 150)$, i.e. halve the resolution along X and Y axis but increase the resolution along the Z axis. The resampled data looks plausible and thus we fix this extent across all of our experiments.

But there is an exception. We notice that vegetation density, i.e. `rhof`, only has obvious non-zero values at a few layers of the original curvilinear data. Thus, we extract a subset of the curvilinear data and only resample that subset to image data to the same extent.

We visualize soil and grass with the resampled subset extracted from the bottom 10 and 5 layers, respectively. We visualize the rest attributes with the full-size rectilinear data.

4.3 Direct Volume Rendering

There are four attributes that are visualized with direct volume rendering: grass, fire, water vapor, and vorticity.

4.3.1 Original Implementation

The implementation of direct volume rendering is as follows:

Firstly, we define a `vtkSmartVolumeMapper` and a `vtkVolume` to map input data to a VTK volume property.

Secondly, we set the volume property for `vtkVolume` actors, including the piece-wise transfer function and opacity function. We notice that it is tedious and inefficient to directly tune values in Python. Thus, we use ParaView as a tuning reference. After tuning in ParaView, we export the transfer function settings as a JSON file. This is merely an optimization of the workflow, but it allows us to tune the transfer function efficiently.

Then, we define a `vtkRenderer` and add all volume actors to it. We notice that the sequence of actors matters since the actor in the front will have low occlusion priority among all volumes. Thus, we carefully sort the volumes in a proper order to provide plausible visualizations.

Finally, we set the camera with desired extrinsics and intrinsics. To achieve this, we add an observer to `vtkRenderWindowInteractor` and print camera parameters during mouse interaction. Once we find a plausible view, we record those parameters and set them as default values. This allows us to maintain a consistent view across different experiments.

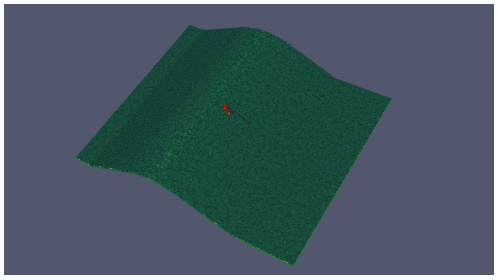
4.3.2 Improvements

During the final presentation, we receive a beneficial suggestion from the lecturer regarding solving the aforementioned occlusion problem by composing all volumes into a single volume.

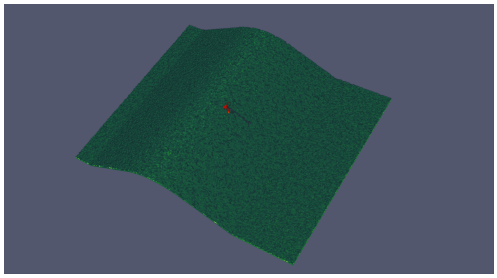
We define a `vtkMultiVolume` and `vtkGPUVolumeRayCastMapper` that composes independent data objects and volume mappers into one volume actor. Since `vtkMultiVolume` only supports `vtkGPUVolumeRayCastMapper`, we use this instead of `vtkSmartVolumeMapper` when defining volume mappers for each attribute. The visual effects have no obvious difference.

Now since all volumes are being processed simultaneously by the renderer, the occlusion problem is automatically solved and there is no need to manually sort the actors. The occlusion relationship is always plausible regardless of the viewing angle, as shown in Figure 1.

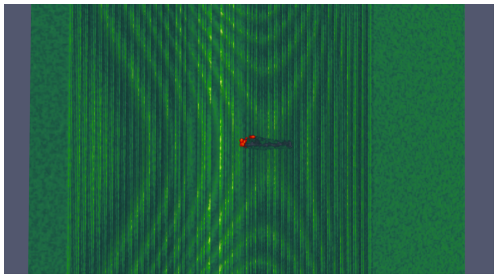
For our improved volume rendering procedure, please refer to `test_vorticity_multiVol.py`.



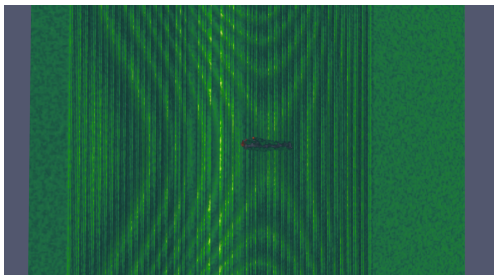
(a.1) Above view of naive method



(b.1) Above view of improved method



(a.2) Bottom view of naive method



(b.2) Bottom view of improved method

Figure 1: Comparisons between naive and improved direct volume rendering. Although both methods give plausible results at **Above** view, the naive method has a problematic occlusion relationship and thus gives wrong results when the viewing angle changes, as can be seen in the **Bottom** view. By using `vtkMultiVolume` this problem was fixed.

4.4 Streamlines

The implementation of visualizing streamlines is as follows:

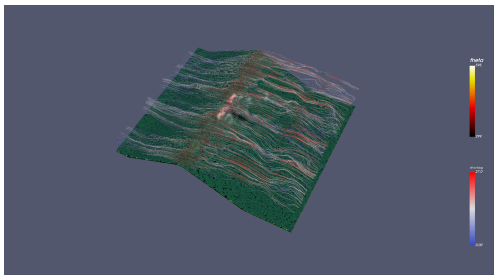
Firstly, we compose three wind velocity components into a vector field and compute its magnitude. In VTK we find `vtkImageMagnitude` does not seem to compute the magnitude correctly, thus we manually compute it by defining the formula in `vtkArrayCalculator`.

Secondly, we define the seeding curve using `vtkLineSource`. In VTK we need to set two ending points of the seeding line. We use the ParaView streamline tracer as the reference using the same coordinates. As for the criteria for placing the seeding curve, our goal of visualizing streamlines is to visualize how the curvature of the mountain influences the separation of airflow, which may induce VLS behavior.

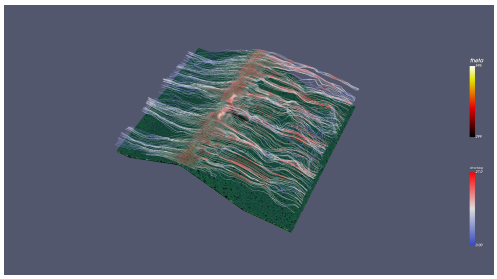
We notice that mountains with different curvatures have different heights. Thus, for different curvatures, we set the Z dimension values of seeding curves differently. We try to place the seeding line accurately at the end of the mountain peak curve with vertical distances to the surface as similar as possible.

Then, we define a `vtkStreamTracer` instance and set all parameters for the tracer as previously estimated in ParaView. We enable both forward and backward tracing for better visualization.

Finally, we need to map the tracer output into a geometry mapper. We have tried mapping wind velocity magnitude to `vtkTubeFilter`, but our program ran slowly when setting the `SetRadiusFactor` too high, while a low `SetRadiusFactor` value will result in the zoom-out view of streamlines to fade. Thus, we choose to directly map velocity magnitude to `vtkPolyDataMapper` and use the transfer function tuned in ParaView. The comparisons between the two methods are shown in Figure 2.



(a) `vtkTubeFilter` + `vtkPolyDataMapper`



(b) Direct `vtkPolyDataMapper`

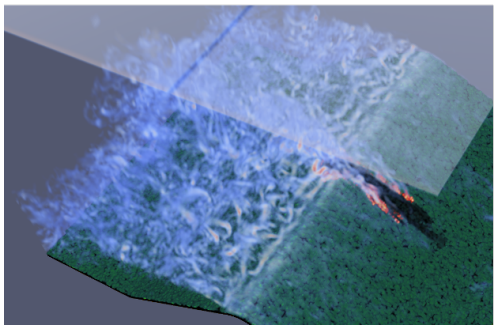
Figure 2: Comparisons between two streamline methods.

4.5 Vorticity

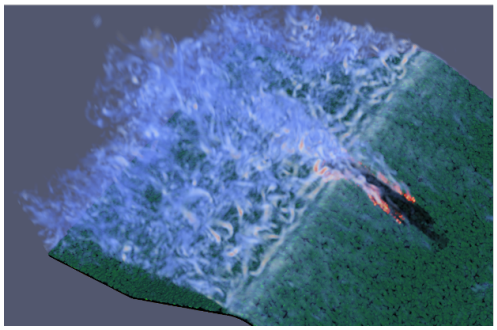
The visualization of wind vorticity was implemented as follows: We compute the cell derivative of wind velocity using `vtkCellDerivatives`. Then, `SetVectorModeToComputeVorticity` is used to compute the cell vorticity of wind velocity. Finally, we convert cell vorticity into vertex vorticity using `vtkCellDataToPointData`.

We notice that the vertex vorticity has strange ceiling artifacts. After verifying in ParaView we came to the conclusion that this is caused by boundary outlier values. We manually mask out the ceiling part, which has near-zero values and does not influence visualization. We notice that after reshaping the one-dimensional `vtkArray` into its spatial structure, the tensor is indexed in Z, Y, X order.

The visualizations of wind vorticity before and after masking artifacts are shown in Figure 3.



(a) Before masking

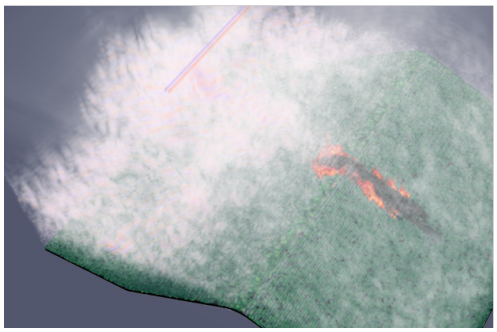


(b) After masking

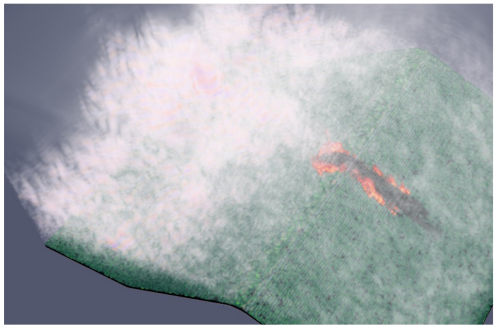
Figure 3: Visualizations of wind vorticity before and after masking artifacts.

4.6 Divergence

The implementation of the divergence visualization is very similar to visualizing vorticity. We also use `vtkCellDerivatives`, however this time we use `SetVectorModeToComputeGradient` to compute the Jacobians of the cells. We then convert the data to point data using `vtkCellDataToPointData`, then convert the results to `numpy` and compute the divergence by summing the diagonal elements of the Jacobians. Similar to vorticity we also have ceiling artifacts, so before converting the calculated divergence data back to VTK we again mask out the ceiling part to remove the artifacts, as shown in Figure 4.



(a) Before masking



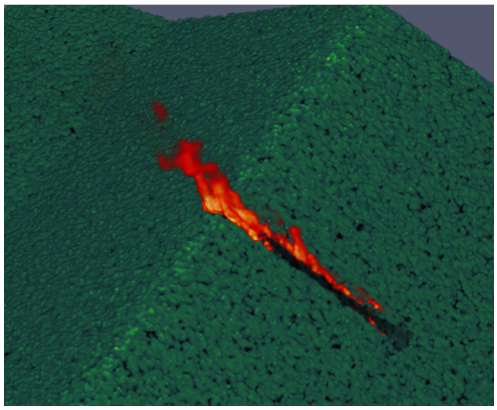
(b) After masking

Figure 4: Visualizations of wind divergence before and after masking artifacts.

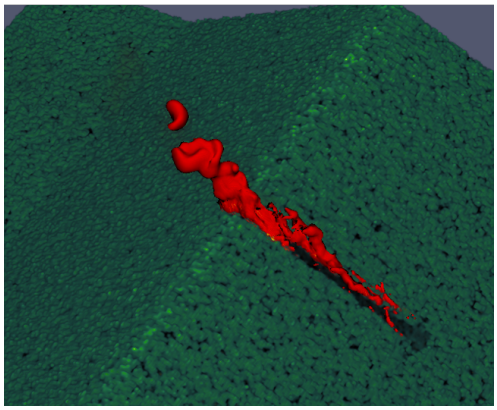
4.7 Isosurfaces

As an alternative approach to volume rendering, we also experimented with isosurfaces, as shown in Figure 5. The idea behind it was to visualize the boundary of air and fire in order to extract some interesting information like intersections or surface area. We did this by simply using the `vtkContourFilter` function applied to the temperature values (`thetaImage`), which required extensive fine-tuning of the isovalue in order to have an isosurface that is as accurate as possible. However we did not find a way to make use of the data in our analysis, so we ended up not extensively using the isosurface. In case we would have used the isosurface as a core part of our analysis, one could have implemented a pre- or post-processing function that makes for example use of other data features (e.g. oxygen concentration) to further enhance the quality of the isosurface. At the end of our project, we also real-

ized that we could have rendered grass as an isosurface directly, which would be a future improvement.



(a) Fire via volume-rendering



(b) Fire via isosurface

Figure 5: Visualizations of fire via volume-rendering and as isosurface.

4.8 GUI

4.8.1 Main Functionalities

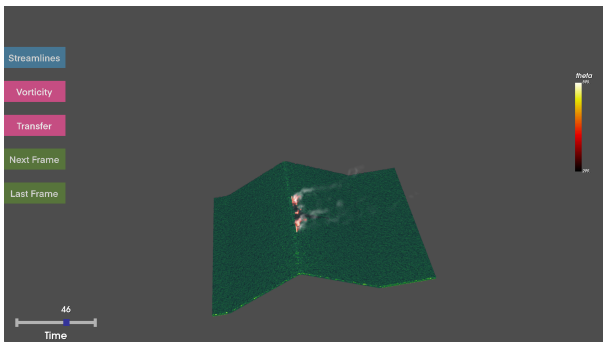


Figure 6: Graphical user interface.

The main structure of our graphical user interface (GUI) can be seen in Figure 6. It consists of the following controls:

- Buttons to (de)activate the display of streamlines and vorticity
- An additional button to show the transfer function editor for vorticity.
- A button to skip back and forth between frames one frame at a time
- A slider to quickly slide through the frames

In addition, it shows a legend of the currently display features.

4.8.2 Transfer Function Editing

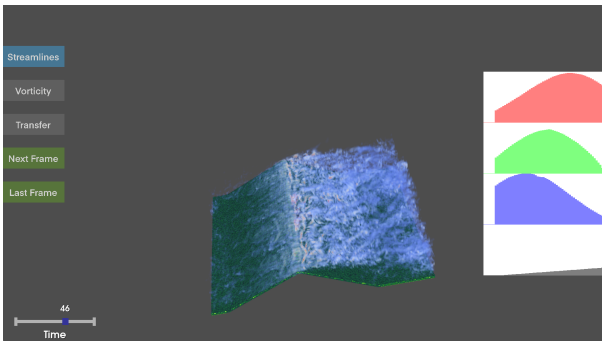


Figure 7: Transfer function editing functionality.

Transfer function editing was implemented exemplarily for the vorticity, which is shown in Figure 7. This can easily be extended to other computations such as streamlines. The base implementation of this was taken from the third programming exercise of the course, with some adjustments and optimizations, such as automatic domain selection when the domains of the color and opacity transfer functions do not align.

4.8.3 Demo Video

Please refer to `WildFireB_GUIdemo.mov` to see a demo of the optimized GUI.

4.9 Analysis on VLS Behaviors

4.9.1 Quantitative Measurements

The spreading of wildfire can be divided into two stages: uphill and lateral spread.

We estimate the distance and duration of the two stages for all six scenes. The results are listed in Table 1 and Table 2. CSR denotes the "cumulative spread rate" and is computed as slope distance divided by duration.

Dataset Type	Stage 1 Duration ($\times 10s$)	Stage 1 CSR ($\times 10^{-1} m/s$)
back40	25	**∼6.0**
back80	31	∼5.1
back320	42	∼3.1
head40	15	**∼19.5**
head80	15	∼18.7
head320	14	∼18.8

Table 1: Stage 1 durations and CSRs. The largest CSRs are **bold**.

Dataset Type	Stage 2 Duration ($\times 10s$)	Stage 2 CSR ($\times 10^{-1} m/s$)
back40	40	**∼17.8**
back80	38	∼10.2
back320	28	∼1.3
head40	54	**∼6.8**
head80	70	∼6.0
head320	56	∼1.9

Table 2: Stage 2 durations and CSRs. The largest CSRs are **bold**.

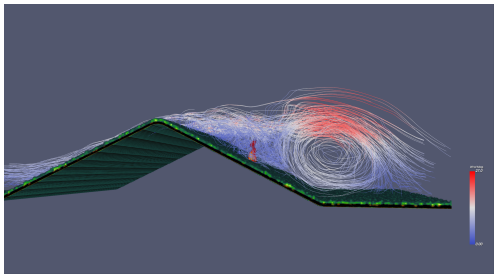
From the tables, we have two findings: (1) A pointier peak leads to faster uphill spreading at each ignition position. (2) A pointier peak leads to faster lateral spreading at each ignition position.

Finding (2) is known as "Vorticity-driven Lateral Spread (VLS)": A steep plane is more likely to cause separation of the air. And "separation of the flow creates horizontal vorticity over the leeward slope"[9]. Then "this vertical vorticity promotes lateral fire propagation across the slope"[8].

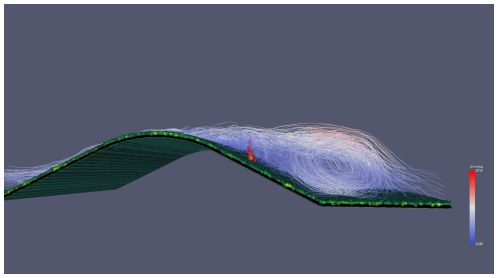
Thus, to verify finding (2), we need to visualize the difference in vorticity or vortex caused by different mountain curvatures.

4.9.2 Side-view Streamlines

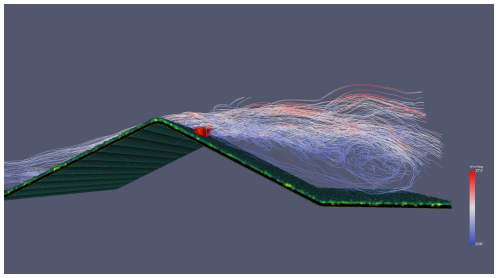
The comparisons of streamlines on different scenes at different frames are shown in Figure 8. It's clear that a more pointy mountain has stronger back-rolling winds which push the fire uphill and thereby facilitate the stage 1 spreading. And according to the last subsection vorticity caused by airflow separation is also the cause of VLS behavior.



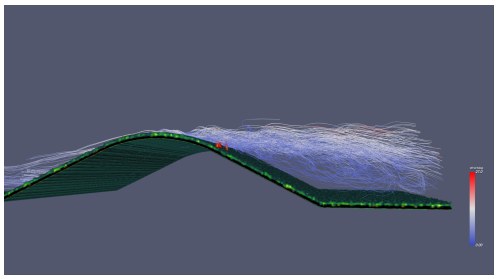
(a) backcurve40 at frame 8.



(b) backcurve320 at frame 8.



(c) backcurve40 at frame 40.



(d) backcurve320 at frame 40.

Figure 8: Comparisons of streamlines on different scenes at different frames.

4.9.3 Line Integral Convolution

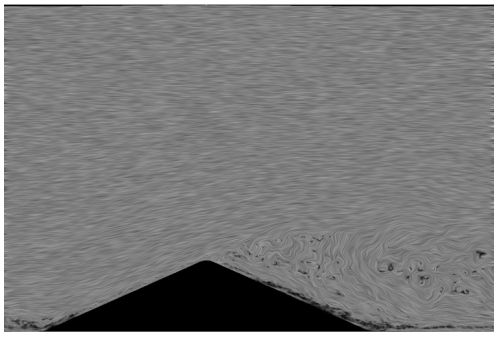
We notice that in some frames, the vorticity caused by airflow separation might not be very clear. Thus, we implement Line Integral Convolution (LIC) to show the comparisons. For the implementation of LIC, please refer to `myLIC/run_lic.py`.

The results of LIC with different kernel lengths are shown in Figure 9.

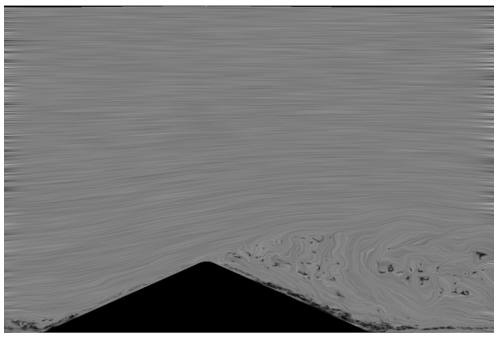
The LIC comparisons on different scenes are shown in Figure 10. It's clear that for backcurve320, the vorticity is much weaker than backcurve40, which leads to a weaker VLS behavior. This may explain why more pointy mountains have a larger lateral spreading speed.

4.10 Animations on Full Dataset

We generated all frames given in the full dataset. For each frame, we separately generated the fire, the fire including streamlines, and the fire including vorticity. For each scenario, we created a video showing all frames consecutively.

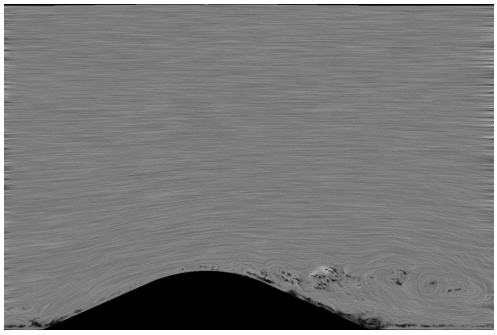


(a) LIC with kernel length 25.

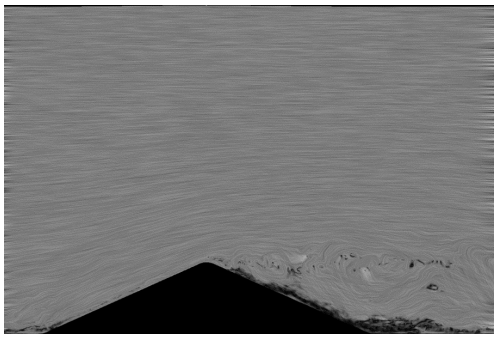


(b) LIC with kernel length 100.

Figure 9: Comparisions of LIC on different kernel lengths.



(a) LIC on backcurve320.



(b) LIC on backcurve40.

Figure 10: Comparisions of LIC on different scenes with kernel length 100.

For all frames and animations, please refer to `results.zip`.

5 Contributions

Everyone on our team did a great job collaborating on this project, whether it be communicating in our group chat and via online meetings, preparing presentations, or doing their part of the implementations and analysis. The report was a collaborative effort. In the contributions, we attribute the main author(s) of the sections.

5.1 Johannes Kurz

ParaView: Initial data exploration, understanding of dataset.
VTK: Divergence, isosurfaces, animation generation, general code improvements.
Report: Sections 3, 4.6, 4.7, 6, an overhaul of the full report in terms of language and formatting.

5.2 Yingyan Xu

ParaView: Visualize fire, wind vorticity with direct volume rendering; intial transfer function tuning.
VTK: GUI code framework and optimization.
Report: Proof reading.

5.3 Yitong Xia

ParaView: Reference pipeline, transfer function tunning.
VTK: General code pipeline, direct volume rendering, streamlines, vorticity, line integral convolution, animations codes, part of animation generation, data preprocessing analysis & optimization for GUI, VLS analysis.
Report: Sections 1, 2, 4.2, 4.3, 4.4, 4.5, 4.9.

5.4 Stuart Heeb

ParaView: First experiments with the reduced dataset given in Moodle, analysis of data and its format.
VTK: Data preprocessing procedure, animation generation, GUI refinements, transfer function editing and general code improvements & debugging.
Report: Sections 3, 4.1, 4.8, 4.10, 6, an overhaul of the full report in terms of language and formatting.

6 Discussion

6.1 Limitations

Our approach does not make use of all features provided in the data and we do not think our work would benefit from direct visualization of these missing features. However, using features like oxygen concentration or heat transfer as part of for example pre- or post-processing routines might allow us to make our visualization more accurate and thereby enhance the meaningfulness of our analysis. To summarize, our approach might be limited in accuracy by not taking into account the complete provided physical model. We also consider as a limitation that we did not implement asymptotic convolution in LIC that reflects the airflow direction, which limits the accuracy of the LIC.

6.2 Future Work

In future work, we could try rendering grass as an isosurface instead of a volume. We could also incorporate the rest of the variables (*e.g.*, oxygen, heat transfers) to visualize the wildfire from more aspects and to facilitate more precise analysis. As for LIC, adding asymptotic convolution can help visualize airflow directions. It would also be convenient to integrate LIC into our GUI to interactively visualize a specified slice of the data. Furthermore, we notice some aliasing in our current GUI visualization, which can be reduced by applying a low-pass filter before resampling the structured grid to the rectilinear grid.

References

[1] J. Ahrens, B. Geveci, and C. Law. ParaView: An end-user tool for large data visualization. In *Visualization Handbook*. Elesvier, 2005. ISBN 978-0123875822.

[2] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020.

[3] R. Linn, J. Reisner, J. J. Colman, and J. Winterkamp. Studying wildfire behavior using firetec. *International journal of wildland fire*, 11(4):233–246, 2002.

[4] R. R. Linn. *A transport model for prediction of wildfire behavior*. New Mexico State University, 1997.

[5] Los Alamos National Laboratory. IEEE 2022 SciVis Contest. `https://www.lanl.gov/projects/sciviscontest2022/`, 2022. [Online; accessed 3-May-2023].

[6] Los Alamos National Laboratory. IEEE 2022 SciVis Contest data. `https://oceans11.lanl.gov/firetec/`, 2022. [Online; accessed 3-May-2023].

[7] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit (4th ed.)*. Kitware, 2006.

[8] J. Sharples, A. Kiss, J. Raposo, D. Viegas, and C. Simpson. Pyrogenic vorticity from windward and lee slope fires. *Int. Congr. Model. Simul., Gold Coast, Aust. 29 Nov.–4 Dec*, pages 291–97, 2015.

[9] J. J. Sharples and J. E. Hilton. Modeling vorticity-driven wildfire behavior using near-field techniques. *Frontiers in Mechanical Engineering*, 5:69, 2020.