UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 1851

Interactive Visualization of Electrographic Flow using WebGL

David Emanuel Lukšić

Zagreb, June 2019

The work presented here would not be possible without the great help I have received while writing this thesis. I would like to thank the team at Ablacon for patiently explaining every detail there is about anatomy, AFib, catheters, ECG signals etc. and providing funds for exciting visits to hospitals and hardware needed. Also, I would like to say special thank you to my mentor Željka Mihajlović for being very open and supportive of our collaboration with the industry and for always providing useful hints and much needed help for pushing the work forward.

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING
MASTER THESIS COMMITTEE

Zagreb, 7 March 2019

MASTER THESIS ASSIGNMENT No. 1851

Student:	**David Emanuel Lukšić (0036484690)**
Study:	Computing
Profile:	Computer Science

Title:	**Interactive Visualization of Electrographic Flow using WebGL**

Description:

Investigate electrographic flow data acquired by basket catheter in human hart. Investigate visualization approaches to electrographic flow as 2 channel image-like data as well as possibilities of utilizing 3D electrode position information in visualization process. Implement the investigated models. Implement a framework for analysis and comparison of the investigated models. Show examples of results; consider stutter-free execution. Discuss the influence of parameters of the models on the output. Evaluate the results and the implemented algorithms.
Implement the appropriate software solution. Use technologies: WebGL, three.js and typescript. Make the results of the thesis available online. Supply algorithms, source codes and results with appropriate explanations and documentation. Reference the used literature and acknowledge received help.

Issue date:	15 March 2019
Submission date:	28 June 2019

Mentor:

[illegible]

Full Professor Željka Mihajlović, PhD

Committee Secretary:

[illegible]

Associate Professor Tomislav Hrkać, PhD

Committee Chair:

[illegible]

Assistant Professor Marko Čupić, PhD

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 7. ožujka 2019.

DIPLOMSKI ZADATAK br. 1851

Pristupnik:	**David Emanuel Lukšić (0036484690)**
Studij:	Računarstvo
Profil:	Računarska znanost

Zadatak: **Interaktivna vizualizacija podataka elektrografskog toka u WebGL-u**

Opis zadatka:

Proučiti podatke elektrografskog toka dobivene kateterom košarastog oblika u ljudskom srcu. Proučiti vizualizacijske pristupe elektrografskom toku kao 2-kanalnim slikovnim podacima, kao i mogućnosti korištenja 3D informacija o pozicijama elektroda u procesu vizualizacije. Programski ostvariti istražene modele. Implementirati okvir za analizu i usporedbu istraženih modela. Prikazati primjere rezultata; uz poseban osvrt na vremensku glatkoću ostvarivanja prikaza. Diskutirati utjecaj parametara modela na ostvarene rezultate. Ocijeniti rezultate i implementirane algoritme.
Ostvariti odgovarajuće programsko rješenje. Koristiti tehnologije: WebGL, three.js i typescript. Rezultate rada načiniti dostupne na Internetu. Prezentirati ostvarene algoritme, izvorne kodove i rezultate s odgovarajućim objašnjenjima i dokumentacijom. Navesti korištenu literaturu i dobivenu pomoć.

Zadatak uručen pristupniku: 15. ožujka 2019.
Rok za predaju rada: 28. lipnja 2019.

Mentor:

[illegible]

Prof. dr. sc. Željka Mihajlović

Djelovođa:

[illegible]

Izv. prof. dr. sc. Tomislav Hrkač

Predsjednik odbora za diplomski rad profila:

Marko Čupić

Doc. dr. sc. Marko Čupić

CONTENTS

1. Introduction

Atrial Fibrilation (AF or AFib) is a common heart arrhythmia, estimated to impact 20.9 million men and 12.6 million women worldwide, having higher prevalence in developed countries. By 2030, 14-17 million AF patients are anticipated in European Union alone. This amounts to approximately 3% of adults aged 20 years or older. (Paulus et al., 2016). Historically, AF was treated using anti-arrhythmic drugs. Many different treatments are used for AF today. For this thesis, we focus on catheter ablation. More specifically, ablation guided by electrographic flow (EGF) mapping.

Electrographic flow mapping is a recent technique used to map electrical activity inside the human heart. A basket catheter (figure 1.1) is introduced through the groin, up along the inferior vena cava (IVC) into the right atrium (RA). To map the left atrium (LA), a small 1 mm diameter hole is made in the septum wall using a needle, through which the basket is introduced into the LA.

Once the basket is in place, each of the 64 electrodes on the basket (8 splines, each has 8 electrodes) samples local electrical activity at 1 kHz. This activity is a 24 bit value representing relative voltage difference (± 150 mV) to the ground electrode, which is usually located on the left leg or lower abdomen. We can think of this data as series of images where rows represent splines A through H, and columns represent electrodes 1 through 8:

$$\begin{matrix} A1 & A2 & A3 & ... & A8 \\ B1 & B2 & B3 & ... & B8 \\ C1 & C2 & C3 & ... & C8 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ H1 & H2 & H3 & ... & H8 \end{matrix} \tag{1.1}$$

These 8x8 images are then analyzed and turned into dense 200x200 flow (vector) fields using an optical flow algorithm. For more information about this step, see Bellmann et al. (2018).

Finally, the resulting flow field, also called **Electrographic Flow**, needs to be visualized in an intuitive way. The visualization must allow the medical staff and researchers to easily recognize sources, rotors and sinks and relate them to electrode locations. Additional difficulty arises from the fact that, in it's ideal shape, the basket closely resembles a sphere. When unwrapping the sphere, poles get significantly distorted. Therefore, a 3D representation is needed to visualize the proportions correctly. Ultimately, we allow for specifying 3D locations of individual electrodes. This makes the visualization method much more representative of reality. It's important to note that, for this problem, we only need to consider **time-invariant flow**. This greatly simplifies our analysis and implementation.

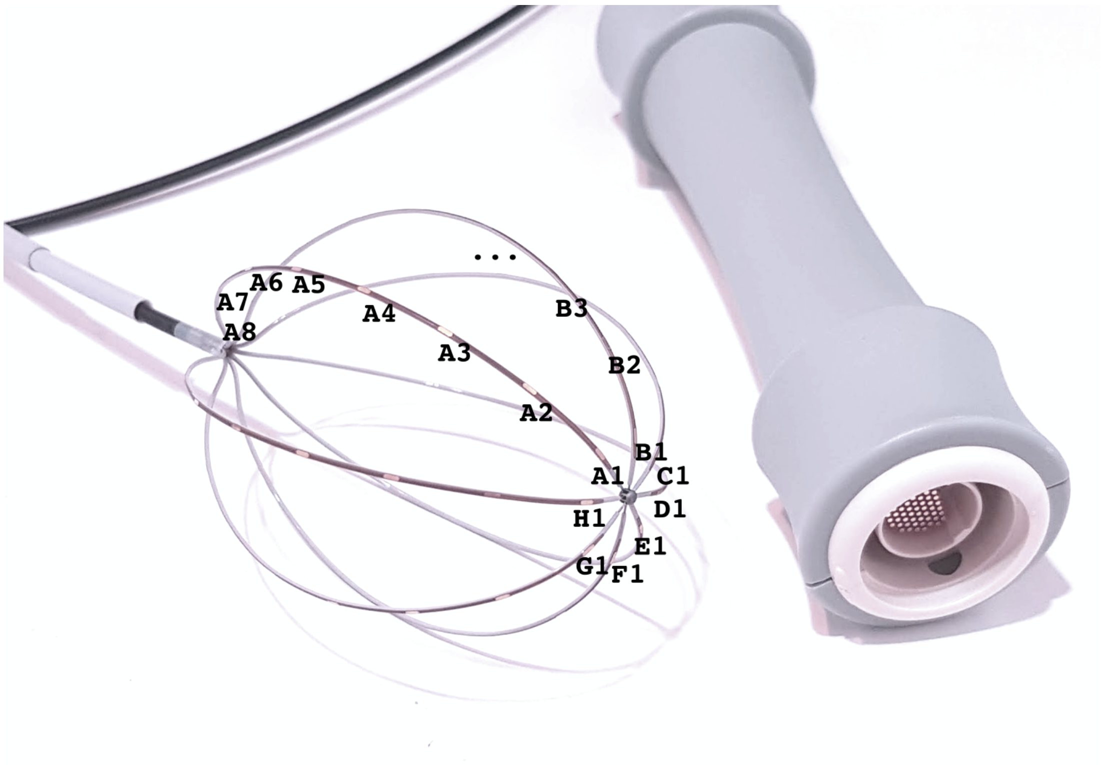


Figure 1.1: Basket catheter and its handle (output pins). Winding direction is clockwise, looking from distal (1) to proximal (8).

2. Implementation Overview

2.1. Used Technologies

2.1.1. WebGL and ThreeJS

One of the main implementation requirements was for the visualization to run in the browser, more specifically Google Chrome. This requirement directly implies that WebGL (Web Graphics Library) must be used. WebGL, is a low level JavaScript API that closely resembles OpenGL's API. It allows developers to run almost arbitrary commands on the client's GPU in order to render 3D graphics, but also some light form of general purpose graphics programming (GPGPU) can be performed as well by using textures and pixel shaders.

To simplify working with WebGL's low level API, a number of open source projects were founded. Most notable ones are ThreeJS (mrdoob et al.) and BabylonJS. I chose ThreeJS as, at the time of writing this thesis, it had more examples and support.

2.1.2. TypeScript

JavaScript is not a strongly typed language. This makes it hard to follow which objects have what data associated to them. This is especially true for medium to large graphics programs since they heavily rely on state. To help with this, Microsoft developed TypeScript. TypeScript is a language, superset of JavaScript, that aims to decorate JavaScript with types. It is also a transpiled language, which means that there is a transpiling step which takes TypeScript source code and turns it into plain (human—readable) JavaScript.

In the following example you can see TypeScript code and the JavaScript equivalent which is also valid TypeScript code:

Listing 2.1: TypeScript example

```
// TypeScript
function example(a: number, b: Vector3): Vector3 {
        return b.addScalar(a)
}

// plain JavaScript
function example(a, b) {
        return b.addScalar(a)
}
```

The main difference between these two code examples is that in the former, we know exactly the types that are coming into and out of the function. The editor can help us in this case and provides helpful hints about what is available for parameter `b: Vector3`.

2.1.3. Python and Bokeh Plotting Library

For many years now, Python has been the leading language for professional data scientists. It has a multitude of libraries for managing and visualizing data. Most notable libraries for interactive plotting on the Web are Bokeh and Plotly's Dash. Both frameworks are a great choice, but the context in which this visualization is used requires Bokeh bindings. Bokeh comes with dozens of different plot types, layouting API and data streaming capabilities. One of the most important features is extending Bokeh through custom models[1], allowing for integration with existing third—party libraries like ThreeJS. The developed library can also be used without Bokeh (using plain JavaScript / TypeScript).

2.1.4. Dat.GUI Library

One of the most important features is ability to change visualization parameters without having to re-render the whole plot. Using `dat.GUI`[2] JavaScript library simplifies this process significantly.

[1]`https://bokeh.pydata.org/en/latest/docs/user_guide/extensions_gallery/wrapping.html`

[2]`https://github.com/dataarts/dat.gui`

2.2. Jupyter Notebook Setup

User of the developed flow visualization library needs to have Python version 3.5+ installed. Then, Bokeh and Jupyter need to be installed using the following commands:

```
pip install bokeh
pip install jupyter
```

Note: some features are available only through the JavaScript/TypeScript interface, like specifying 3D electrode positions.

In the provided code samples, `notebook_example.zip` contains fully functional Jupyter notebook sample. To run it, simply run `jupyter notebook .` in the extracted folder.

2.3. Using the Library

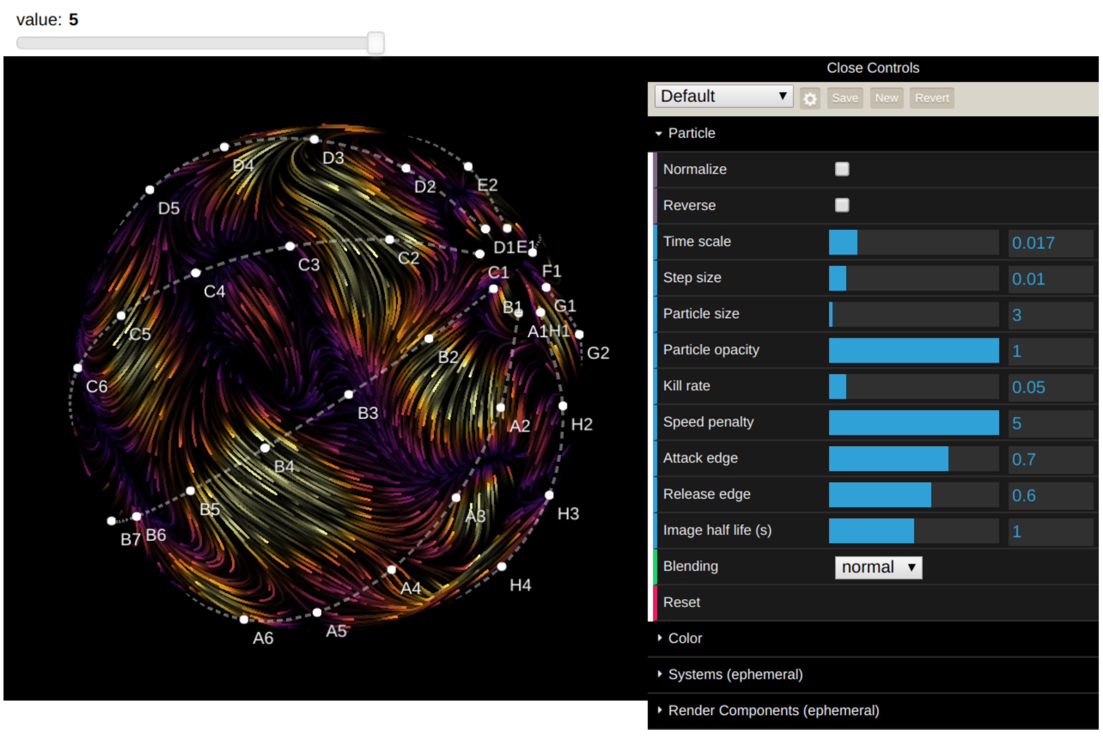
Finally we can look at how easy it is to use the library. Everything the user needs is contained in the following classes:

- `Particle2DModel`
- `Particle3DModel`
- `LIC2DModel`
- `LIC3DModel`

In the `flow_test_noise.ipynb` a usage example is given with the most important part of the code being:

Listing 2.2: Python usage example

```
from flowvis import Particle3DModel
...
model = Particle3DModel(
          ...
          width=700, height=700,
          source=flow_field_source ,
          flow='flows',
          color_map='Inferno8'
          ...
)
```



value: 5

Close Controls

Default ▾ Save New Revert

▾ Particle

Normalize	☐
Reverse	☐
Time scale	0.017
Step size	0.01
Particle size	3
Particle opacity	1
Kill rate	0.05
Speed penalty	5
Attack edge	0.7
Release edge	0.6
Image half life (s)	1
Blending	normal ▾
Reset	

▸ Color

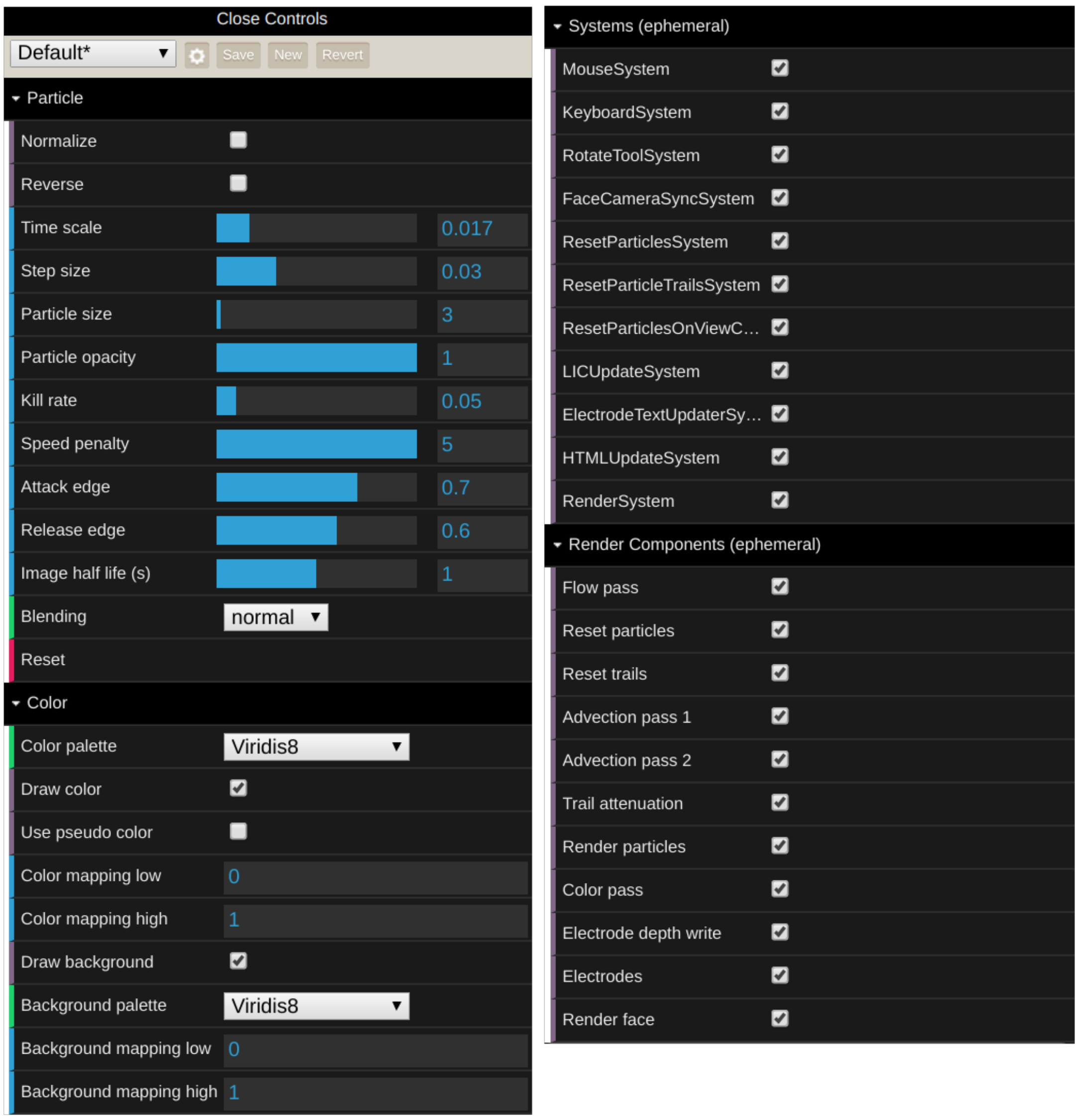
▸ Systems (ephemeral)

▸ Render Components (ephemeral)

Figure 2.1: Example notebook result, Particle3DModel, noise data

For more parameters, see appendix A.

Figure 2.1 shows the resulting visualization. On the top, a slider is shown. A slider is used when there are multiple flow maps to be shown, so the user can flip through them without having to re-render. On the right, a `dat.GUI` parameter interface is shown where the user can play with different values in real time. This has been very important when searching for optimal values, since only a small subset of combinations produces good results. Figure 2.2 shows fully expanded parameter GUI. `Systems` contains all systems which can be enabled/disabled and `Render Components` contains all components related to rendering which can be enabled/disabled.



Particle

Normalize	☐
Reverse	☐
Time scale	0.017
Step size	0.03
Particle size	3
Particle opacity	1
Kill rate	0.05
Speed penalty	5
Attack edge	0.7
Release edge	0.6
Image half life (s)	1
Blending	normal
Reset	

Color

Color palette	Viridis8
Draw color	✓
Use pseudo color	☐
Color mapping low	0
Color mapping high	1
Draw background	✓
Background palette	Viridis8
Background mapping low	0
Background mapping high	1

Systems (ephemeral)

MouseSystem	✓
KeyboardSystem	✓
RotateToolSystem	✓
FaceCameraSyncSystem	✓
ResetParticlesSystem	✓
ResetParticleTrailsSystem	✓
ResetParticlesOnViewC…	✓
LICUpdateSystem	✓
ElectrodeTextUpdaterSy…	✓
HTMLUpdateSystem	✓
RenderSystem	✓

Render Components (ephemeral)

Flow pass	✓
Reset particles	✓
Reset trails	✓
Advection pass 1	✓
Advection pass 2	✓
Trail attenuation	✓
Render particles	✓
Color pass	✓
Electrode depth write	✓
Electrodes	✓
Render face	✓

Figure 2.2: All parameters available in parameter GUI.

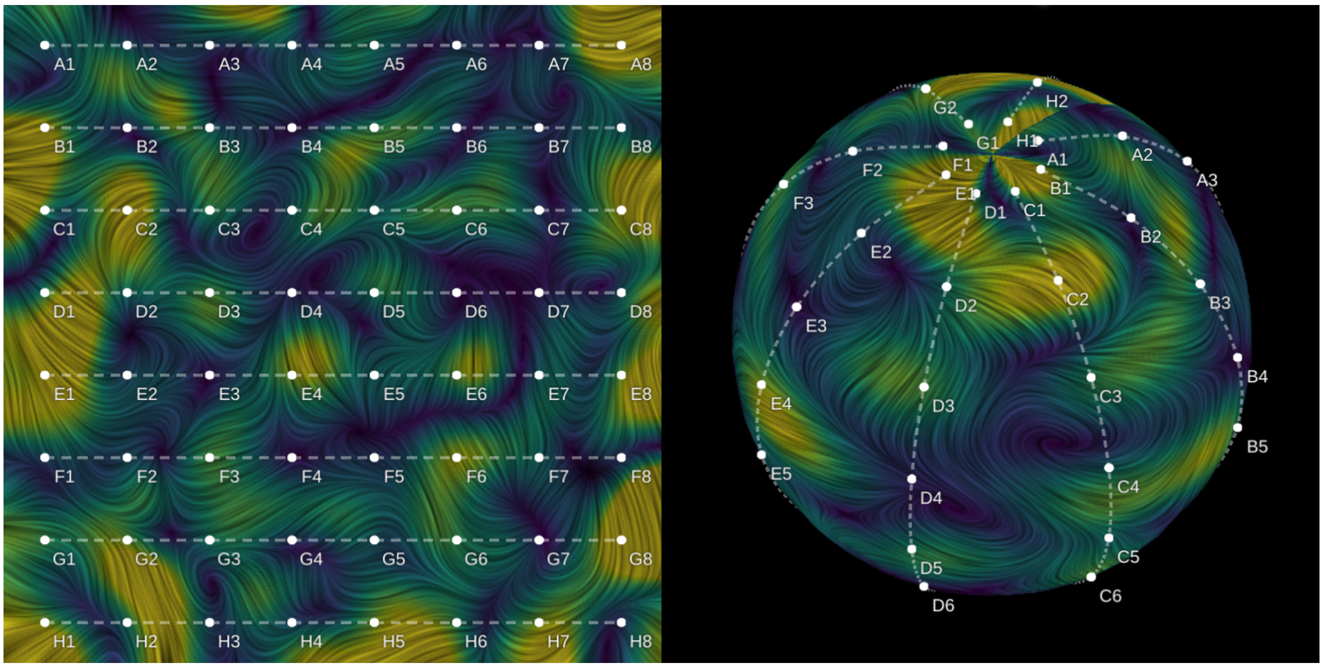


Figure 2.3: Line Integral Convolution, 2D (left) and 3D (right), noise data.

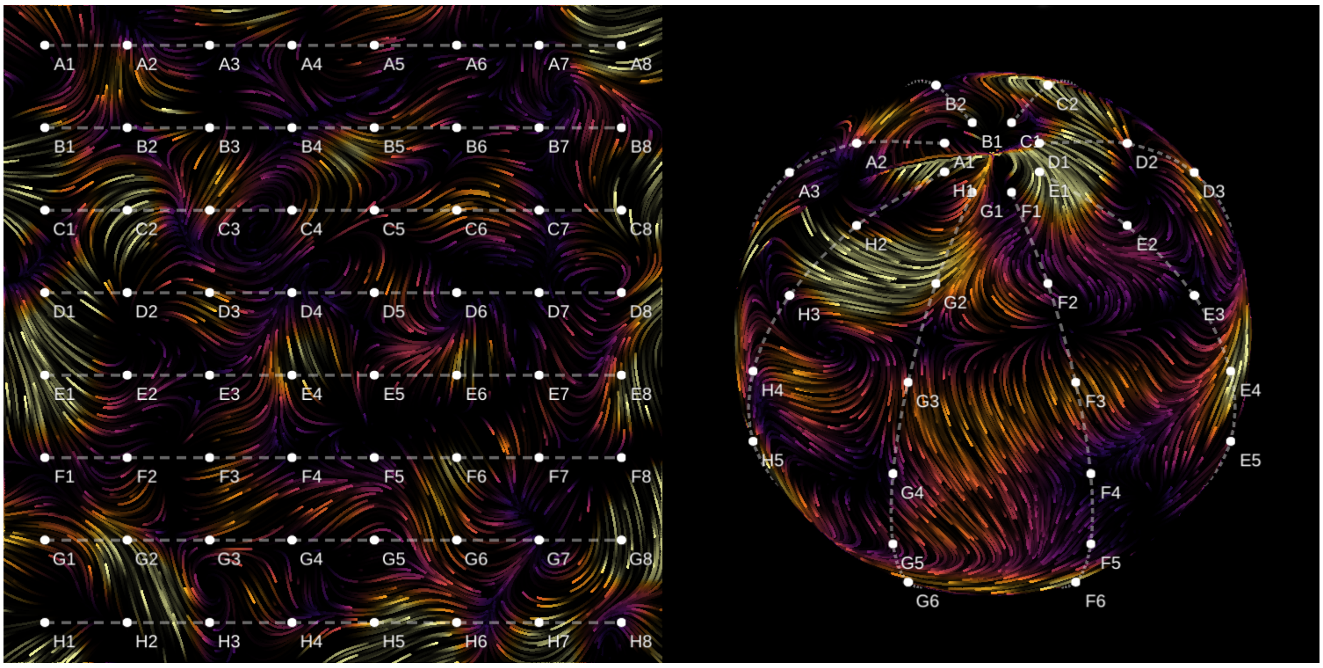


Figure 2.4: Particles, 2D (left) and 3D (right), noise data.

3. Methods

3.1. Quiver Plot

The most trivial way to visualize a given flow field is to draw arrows representing individual vectors the flow field is comprised of. We call this a quiver plot. It is a widely recognized visualization method and implemented in every major plotting library. As such, it is not the focus of this thesis, but it is interesting to note some properties. Here we show how different parameters influence legibility of the quiver plot:

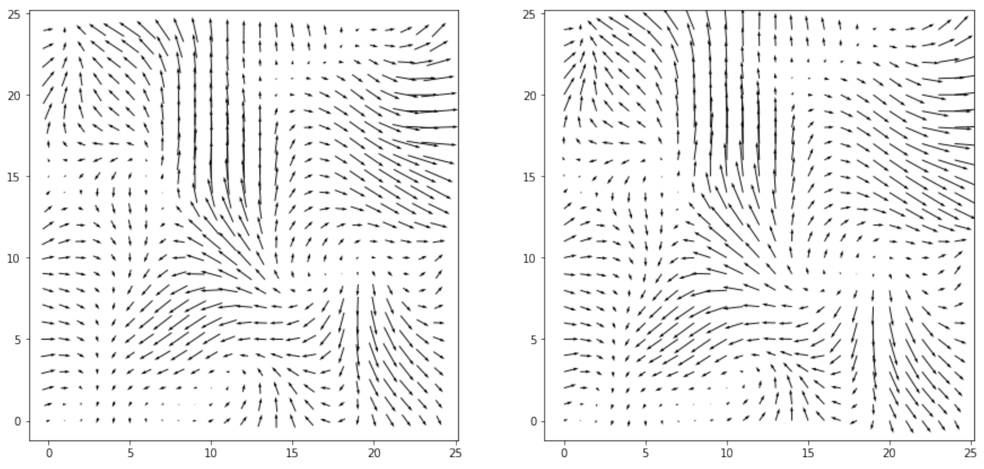


Figure 3.1: Matplotlib's quiver plot with arrows anchored using their midpoint (left) and tail (right). Midpoint anchor usually gives a better impression of the field since it doesn't shift the image.

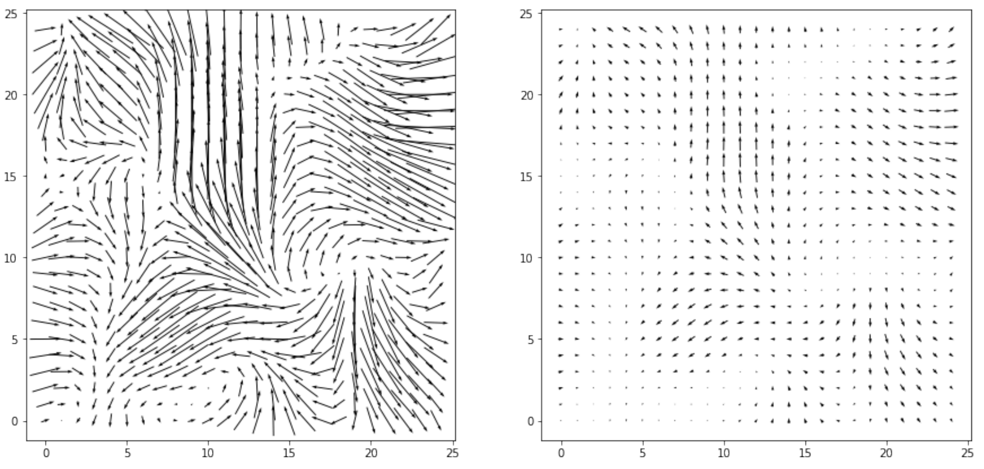


Figure 3.2: Matplotlib's quiver plot with long (left) and short (right) arrows. A great deal of issues stems from choosing the arrow length. If there is great variability in flow magnitude, some sort of compression must be used. Otherwise, short arrows are too short and long arrows conflict with each other.

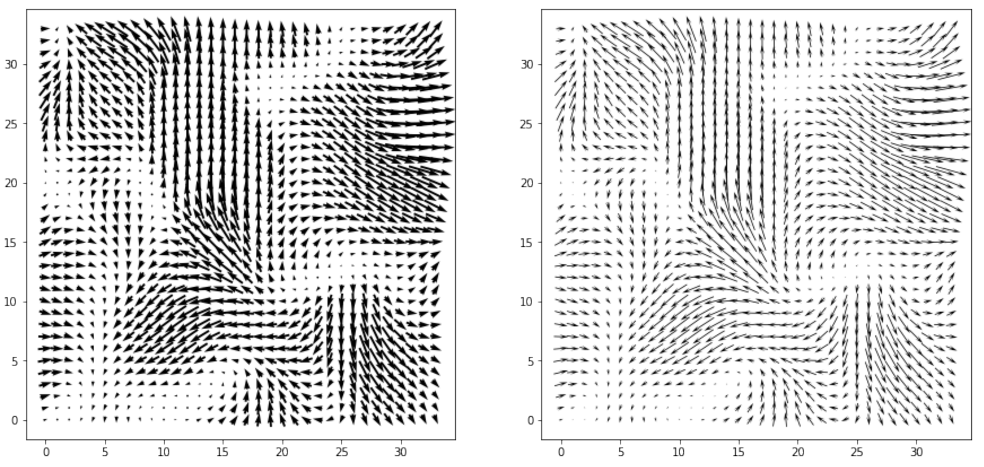


Figure 3.3: Matplotlib's quiver plot with thick (left) and thin (right) arrows. Thinner arrows are usually better for representing dense fields.

3.2. Streamline Plot

Another popular way to display flow fields, already implemented in major plotting libraries is a streamline plot. To understand the streamline plot, we first need to define what a streamline is: a family of curves that are instantaneously tangent to a given flow field. Or, explained more visually, the paths of "massless" particles traveling through a given flow field frozen in time. If the flow field was indeed time-varying, then we would have to consider other curves, listed in appendix B.

We can derive a visualization method directly from this description. First, we define a set of starting points. Then, each point $\vec{x}$ is advected along the flow field $\vec{u}(\vec{x})$ by a small dt leaving a trail:

$$\frac{d\vec{x}}{dt} = \vec{u}(\vec{x}) \tag{3.1}$$

When two trails get too close to each other, we can terminate one, so that the density stays relatively constant. Trail density is especially a problem, and it is usually solved by trying out a large number of different starting conditions until some scoring function is satisfied.

Since we are interested only in relatively short streamlines and the flow field is not changing rapidly, good results can be achieved using simple Euler's integration. For even better approximation in more dynamic fields and longer streamlines, we can use more complex integration methods like the Midpoint method or Runge–Kutta 4.

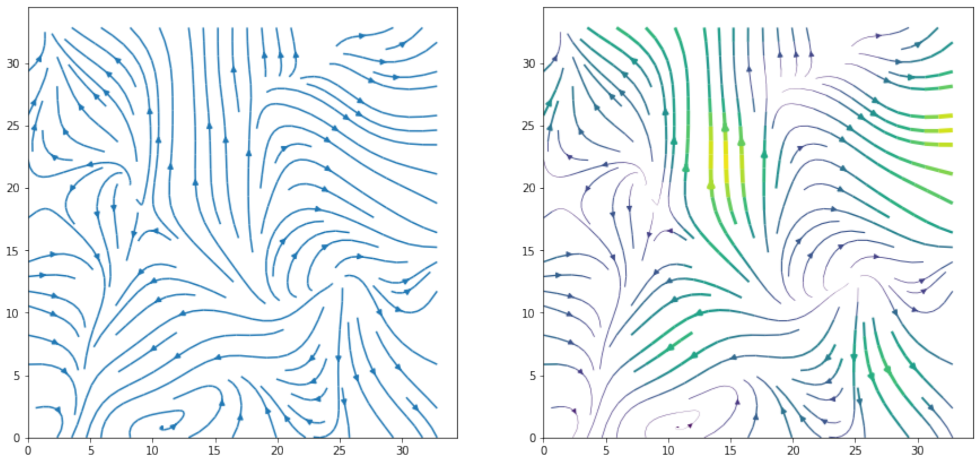


Figure 3.4: Matplotlib's streamline plot. Basic (left) and varying thickness and color (right). To show magnitude, a streamline plot can use thickness and color.

3.3. Particles Plot

The streamline plot suffers from the same issue as the quiver plot: information density. Ideally, we would like to show lots of short streamlines and animate them, so that areas that were potentially missed in one configuration, become visible in another.

A "particle" plot, is exactly that: thousands of particles moving along the flow field on a canvas, leaving a trail that slowly disappears. Since the particles tend to end up in sinks or valleys, a mechanism to reset the particles is proposed, keeping density in check. For our use case, we also needed a way to make some parts of the flow field more or less populated, depending on a probability mask. For example, particles tend to "run away" from sources, so setting high spawn probability around them is desirable.

A single particle is represented by 3 numbers: x (horizontal coordinate), y (vertical coordinate) and t "life" of the particle starting at 1. Particles themselves are stored in `particle target 1` (size $N \times 1$ where N is number of particles) as texels (see figure 3.5). On each frame, two consecutive advection passes are used. They execute following algorithm on each particle (in a shader):

```
vec2 pos = particle pos;
float life = particle life;
vec2 flow = flow vector at pos;
float prob = probability of particle appearing at pos;
if (life <= 0.0) {
    if (life < 0.0 || random() > prob) {
        pos = random vec2;
        life = 0.0;
    } else {
        life = 1.0;
    }
} else {
    // midpoint method
    flow = flow vector at half step forward;
    pos = full step forward;
    life -= (1.0 + length(flow.xy) * speedPenalty) * killRate;
}
```

Particle rendering pass takes `particle target 1` as input and renders each particle onto the `pattern target`. Users can customize how particles are rendered

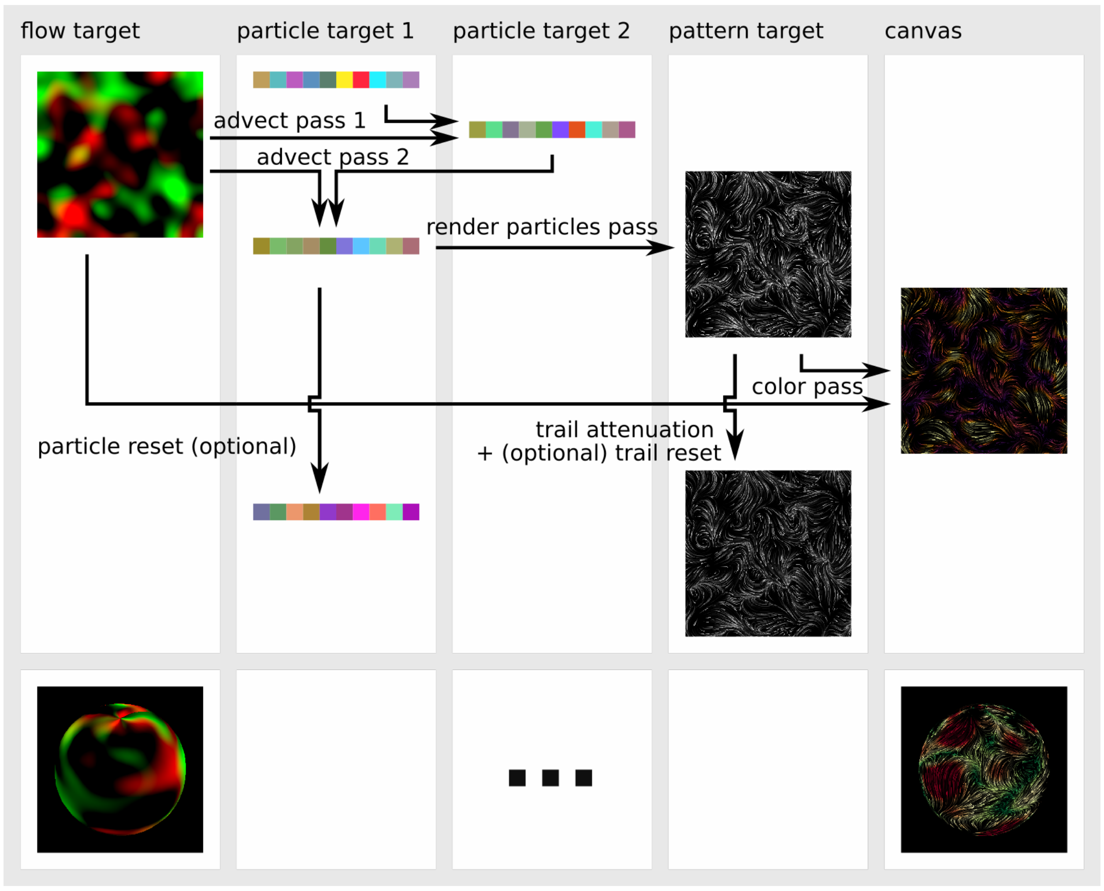


Figure 3.5: Steps in the particle algorithm, 3D version differs only in the flow field projection step.

by changing the size of particles, opacity curve's attack and decay edges and blend mode. Opacity curve has turned out to be really important for reducing the pop–in effect when resetting particles. If a particle appears with full brightness, it doesn't look smooth. To mitigate this issue, a simple linear attack and decay curve is used:

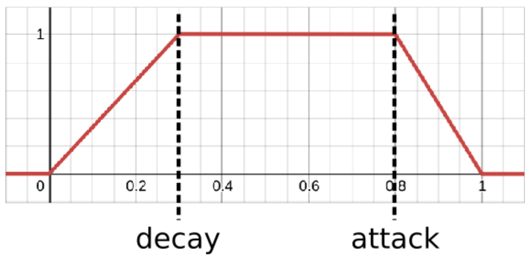


Figure 3.6: Particle opacity attack/decay curve. Life goes from 1.0 to 0.0.

3.4. Line Integral Convolution

Another technique, introduced by Cabral and Leedom (1993), describes an ingenious way of displaying flow fields using noise. Starting with an image of noise (figure 3.8 left), a special kind of convolution is used to smear the noise in the direction of flow. For each pixel, a streamline is calculated. Each location on the streamline is then averaged to generate a new color for that pixel (figure 3.8 middle). Finally, the smear pattern is colored to show magnitude. Although computationally heavy, it is ideal for implementation on a graphics card (using a shader) and a real—time implementation in WebGL is presented.

For noise generation, a 3D simplex noise algorithm by McEwan et al. (2012) is used. It takes a 3D position in space and returns a smoothly changing pseudo—random value depending on where the 3D point lies on the simplex (tetrahedron) grid. Smoothly changing noise is important because it prevents aliasing when the noise gets smeared.

A line integral (also called a path integral, or a curve integral) is an integral of a scalar field along a given curve. In our case, the scalar field is the noise texture, and the curve is a streamline. For each pixel, a streamline is calculated, both forwards and backwards (by advecting in negative flow direction). The noise texture is then sampled along the streamline and averaged. We can do the averaging in a few ways. Usually, we want pixels closer to the original point to have more weight. For this purpose, a kernel function is used:

$$f(x) = sin(\pi(kx+t))^2 cos(\frac{\pi}{2}x)^2 \tag{3.2}$$

where x is the position along the curve, interval $[-1, 1]$. It is composed of two functions:

$$f_1(x) = sin(\pi(kx+t))^2 \tag{3.3}$$

$$f_2(x) = cos(\frac{\pi}{2}x)^2 \tag{3.4}$$

Function f_1 is used to create the animation of flow, because it shifts weight along the curve through time. Function f_2 is used as a modulator to emphasize samples closer to the original point.

The algorithm itself is implemented as a series of render passes for which 3 render targets are needed: flow projection target, noise target and pattern target. On figure 3.9 algorithm is visually laid out. The first steps are projecting flow and rendering noise. In 2D case this amounts to simply rendering the flow field texture itself. Then, a "LIC

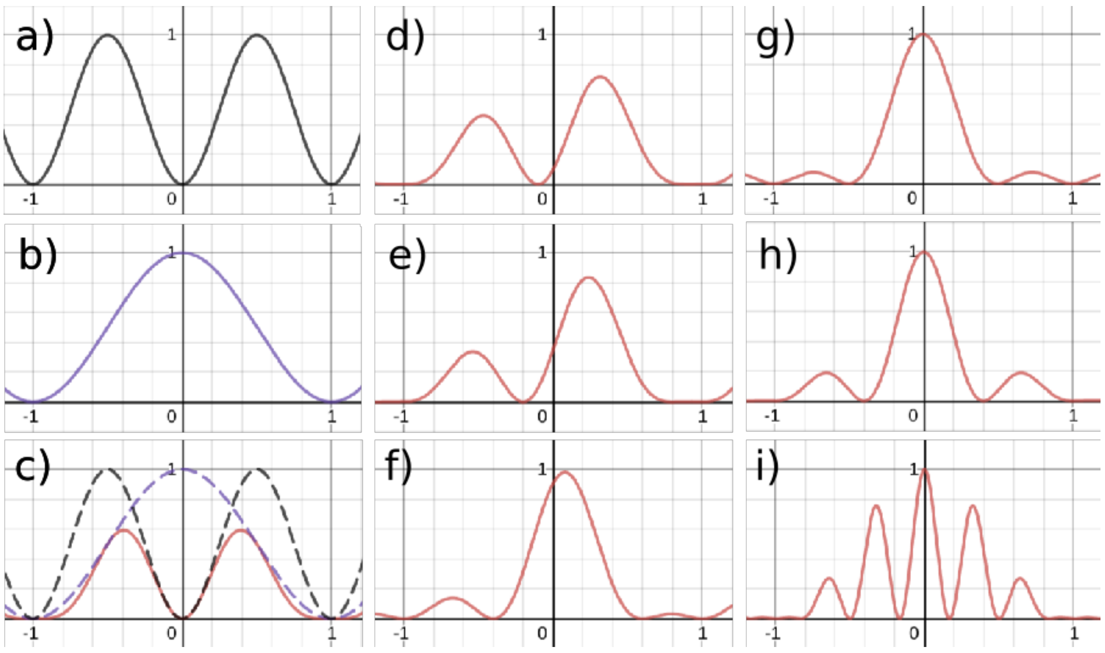


Figure 3.7: a) f_1, b) f_2, c) f (red)
d)–g) time t from 0 to 0.5
g)–i) parameter k (1.0, 1.5, 2.0).

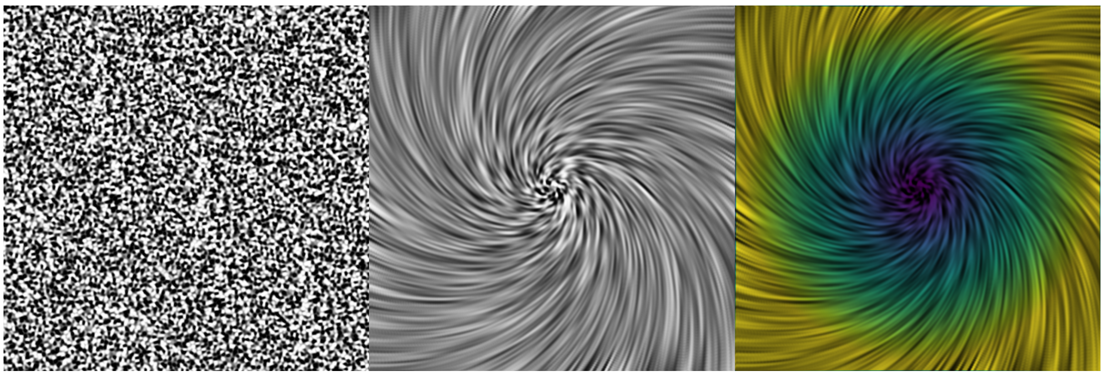


Figure 3.8: Simplex noise (left), convolution operation using a radial flow field (middle), colored pattern (right).

pass" is performed to generate the pattern. Finally, a "color pass" is performed to mix pattern and pseudocolor (color—mapping magnitude) of the projected flow field.

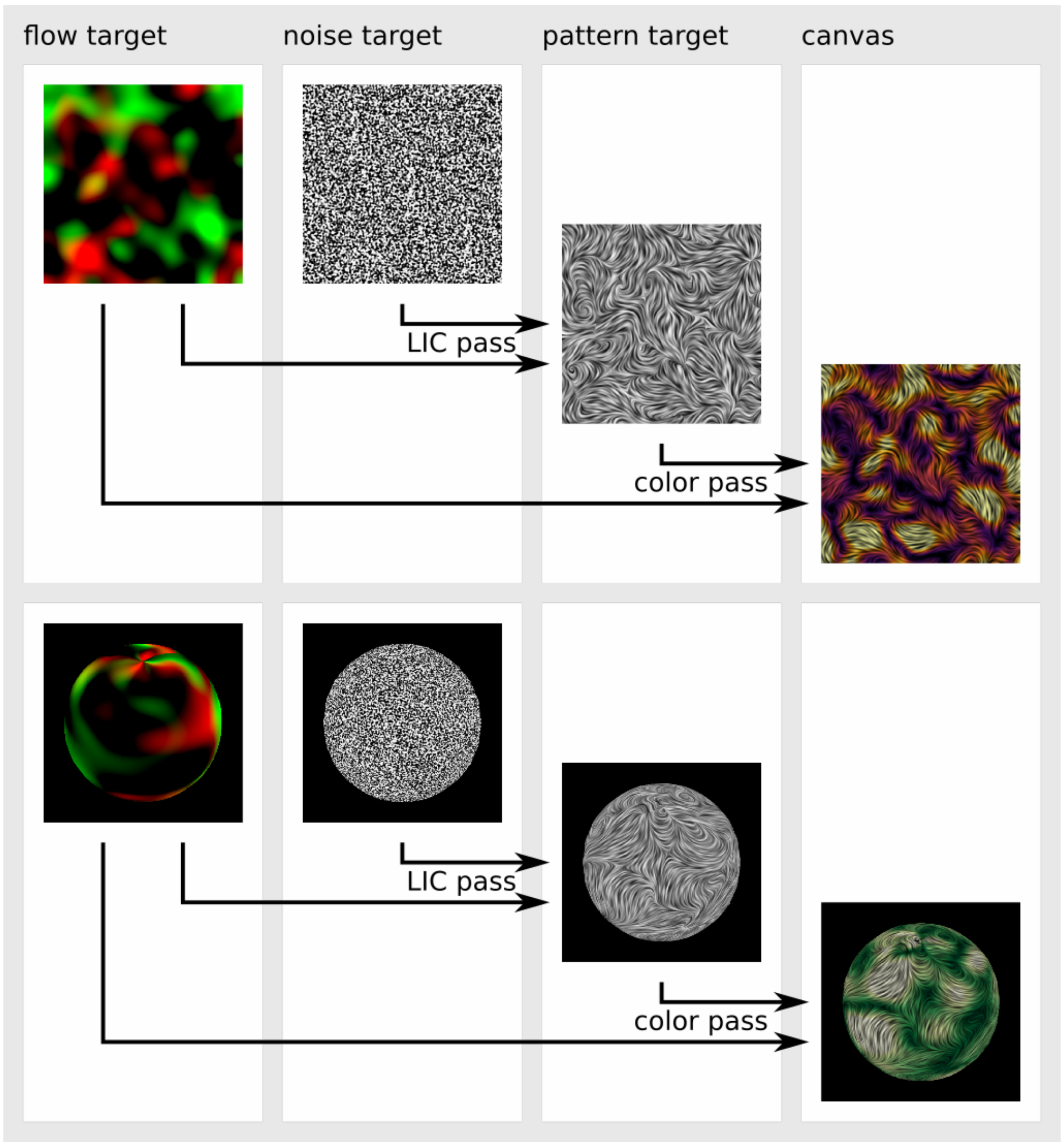


Figure 3.9: Steps in the LIC algorithm, 2D (up) and 3D (down) versions.

4. Extending to 3D

4.1. Projecting Flow

The original problem statement requires us to render the flow field as if it was wrapped around the basket in 3D. We can do this by uv mapping the flow field as a texture and projecting the flow vectors from the local space (uv) to screen space using a pixel shader. The projected flow is then used to render ordinary 2D version of flow visualization.

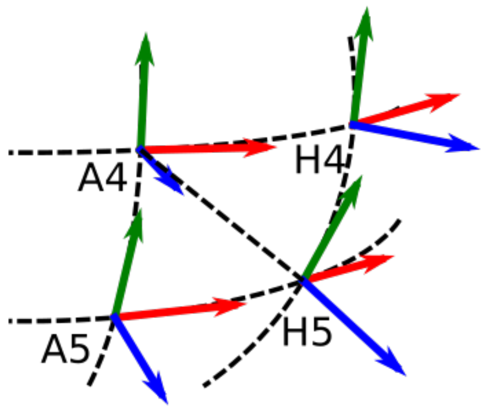


Figure 4.1: Quad (A4,A5,H4,H5) and it's vertices. Vectors $\hat{u}$, $\hat{v}$ and $\hat{n}$ are red, green and blue respectively.

Local space vectors (tangent $\hat{u}$, bitangent $\hat{v}$ and normal $\hat{n}$) are calculated for each triangle, e.g. looking at image 4.1, for triangle $A_5H_5A_4$:

$$\vec{u} = \overrightarrow{A_5H_5} \tag{4.1}$$

$$\vec{v} = \overrightarrow{A_5A_4} \tag{4.2}$$

$$\vec{n} = \hat{u} \times \hat{v} \tag{4.3}$$

Then, in each vertex we sum and normalize neighboring triangles' $\hat{u}$, $\hat{v}$ and $\hat{n}$ so that each vertex has a normalized sum of all three vectors for all 6 neighboring triangles.

In `flowProjectShader`, $\hat{u}$ and $\hat{v}$ are used to transform the original 2D flow vector (x, y) to the 3D representation of the flow vector in local object space $\vec{f}$. It is then transformed to the screen space using the projection matrix $\boldsymbol{P}$ and model view matrix $\boldsymbol{M}$. Keep in mind that this is a direction transformation, therefore the homogeneous coordinate must be 0:

$$\vec{f} = x\hat{u} + y\hat{v}$$
$$\vec{f_s} = \boldsymbol{PM}\begin{bmatrix} f_x \\ f_y \\ f_z \\ 0 \end{bmatrix} \tag{4.4}$$

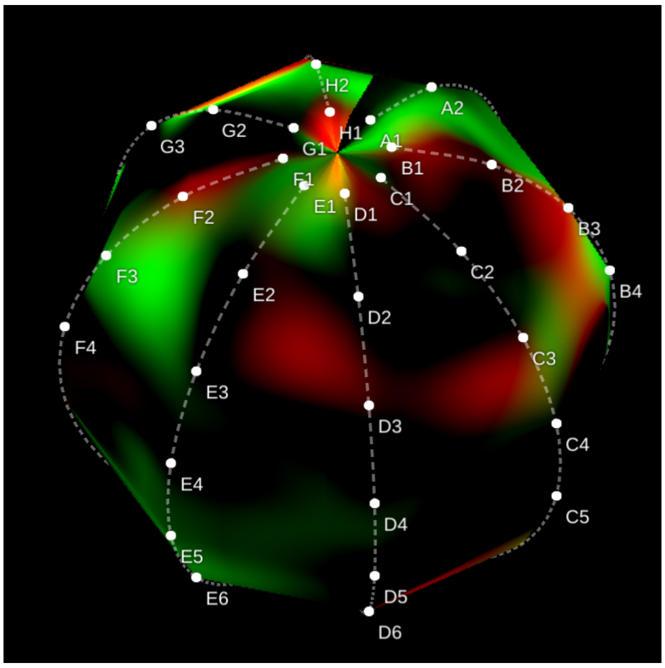


Figure 4.2: Projected flow on a linearly interpolated basket catheter. Red/green channels are projected $f_s x$,$f_s y$ components respectively.

4.2. Surface interpolation

In image 4.2 it can be seen that the basket's surface is linearly interpolated. This creates some aliasing artifacts and generally doesn't look pleasing to the eye. Although there is no information on what the surface should look like in between the electrodes, it is reasonable to assume at least a smooth surface, meaning it should exhibit at least C^1 continuity.

This constraint can be satisfied in an infinite number of ways. For simplicity, a combination of centripetal Catmull-Rom curves (implemented by ThreeJS, mrdoob et al.) is used to interpolate basket's surface.

Catmull-Rom curve, first described by Catmull and Rom (1974), is defined by 4 control points P_0, P_1, P_2, P_3 and produces a parametric curve going **through** (also called an interpolating spline) P_1, P_2. They can be safely chained while keeping C^1 continuity by overlapping first and last two points of each segment. It is important to use centripetal version of Catmull-Rom curve to ensure that no cusps or loops are formed.

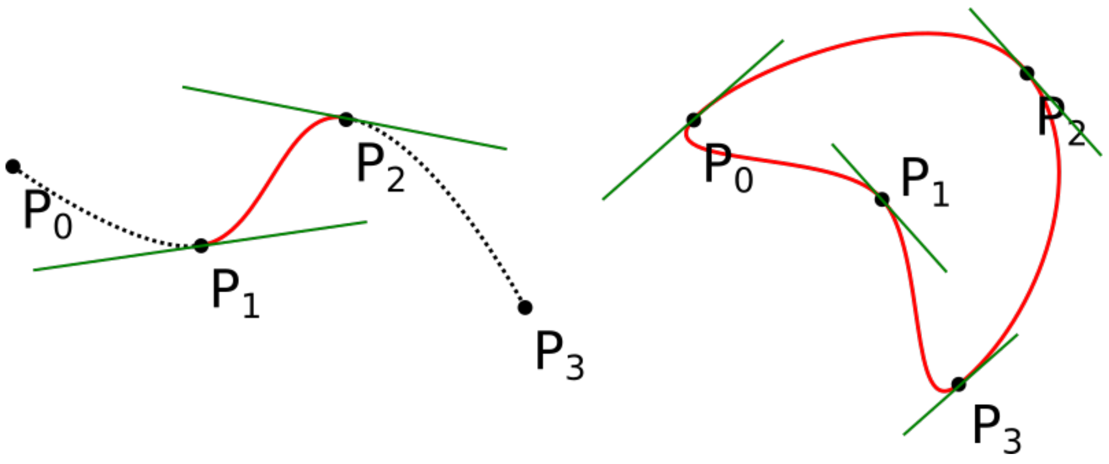


Figure 4.3: Catmull-Rom curve through 4 points, open and closed version. Green lines represent tangents constructed by the difference of two neighboring points.

Interpolation is done in three stages (figure 4.4). Firstly, the basket's splines are subdivided using Catmull-Rom curves. Secondly, "rings" are constructed across splines, through the newly interpolated points using closed Catmull-Rom curves. Finally, the top and bottom's common cap vertices are manually averaged and set to the same value to avoid rasterization mismatch due to small interpolation differences.

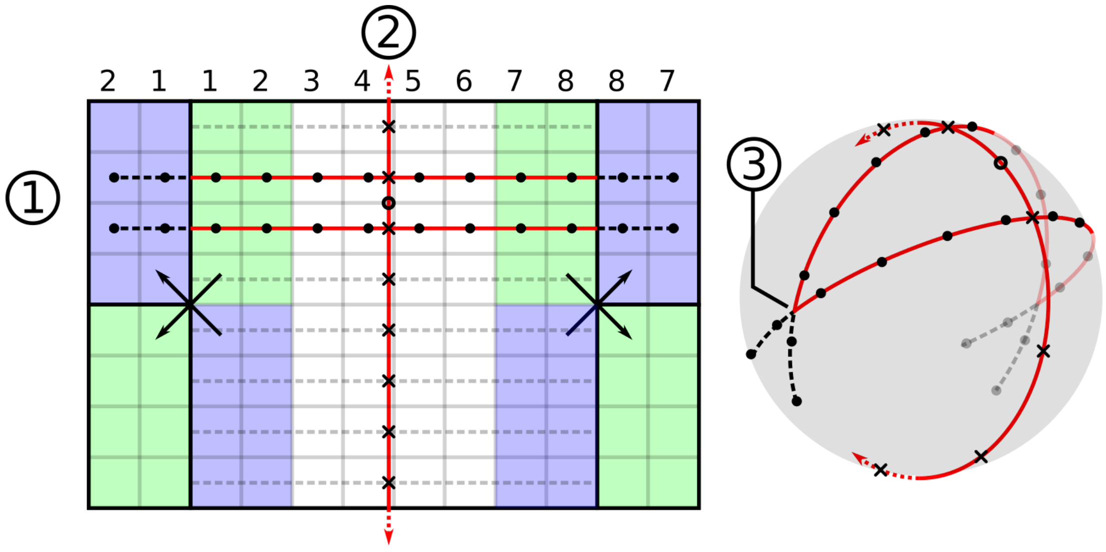


Figure 4.4: Basket surface interpolation by 1) interpolating splines, 2) interpolating across splines as closed loops and 3) averaging the cap's common vertex to avoid rasterization mismatch.

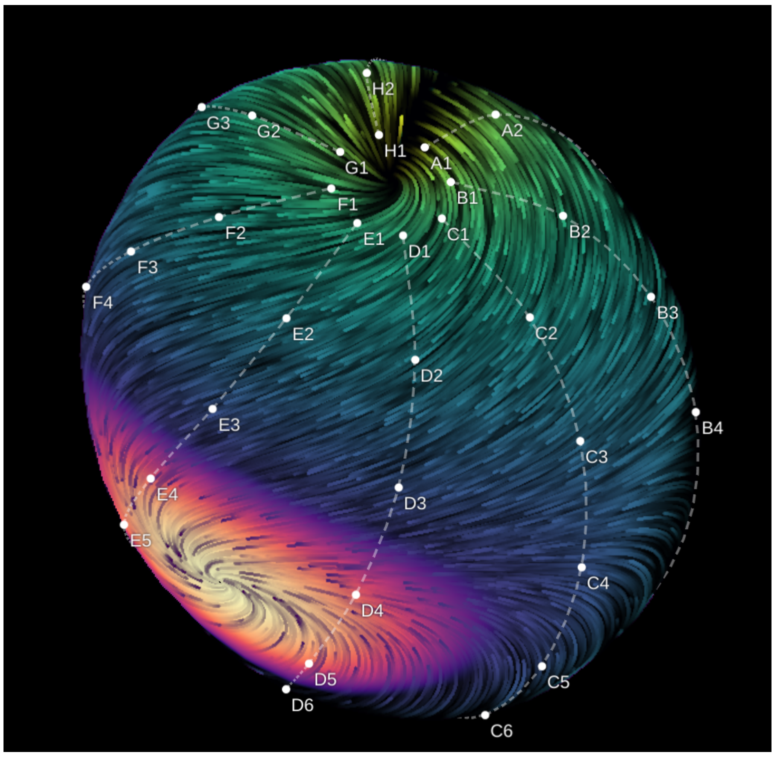


Figure 4.5: Particle plot on a distorted basket geometry, using surface interpolation by Catmull-Rom. Subdivisions: 4 between splines, 2 along splines.

5. Evaluation

5.1. Correctness Testing

One of the most important stages in developing Software as a Medical Device (SaMD) is testing. We must make sure that software that we write behaves correctly, especially in known circumstances. Since flow visualization is primarily used to visualize locations of sources/rotors/sinks, we need to test this functionality thoroughly. One problem in particular that is tricky to get right: correct mapping of flow. To get this right, we have to correctly set up 4 degrees of freedom: uv-mapping of the flow field (2 degrees) and mapping of individual vector components (2 degrees). Because of different projections in 2D (vertical inverse) and 3D (rotated by $90°$), these mappings are also different for these two cases.

To test only this (unit test), 4 different flow fields are created with a source at different location: B2, B7, G2, E7 (figure 5.1). If any of the mappings was wrong we would either see sources in wrong locations (wrong uv-mapping), or not see a source but a saddles point/rotor/sink (wrong vector component mapping).

Even if unit tests pass, that doesn't mean the software will behave correctly when combined with existing pipeline, or that we even used it correctly. Therefore, an end to end test is needed to test integration with the existing pipeline. For this purpose, a simple "mock" data generator is created. The idea is to generate a source/rotor at a known location, and test whether we see it reflected in the visualizer.

The generator takes 64 electrode locations and samples a 3D rotating wave defined as:

$$W(\vec{e}, t) = \phi(\alpha\, \vec{e} \cdot \hat{y} + \beta\, \mathrm{atan2}(\vec{e} \cdot \hat{z}, \vec{e} \cdot \hat{x})) \tag{5.1}$$

Where $\vec{e}$ is electrode location, ϕ is the activation function (saw tooth used), $\hat{x}, \hat{y}, \hat{z}$ are basis vectors for the rotating wave and α, β are divergence and rotation magnitude respectively. Figure 5.2 shows an example dataset with a pure rotor with axis of rotation (0, 1, 1) and the corresponding visualization.

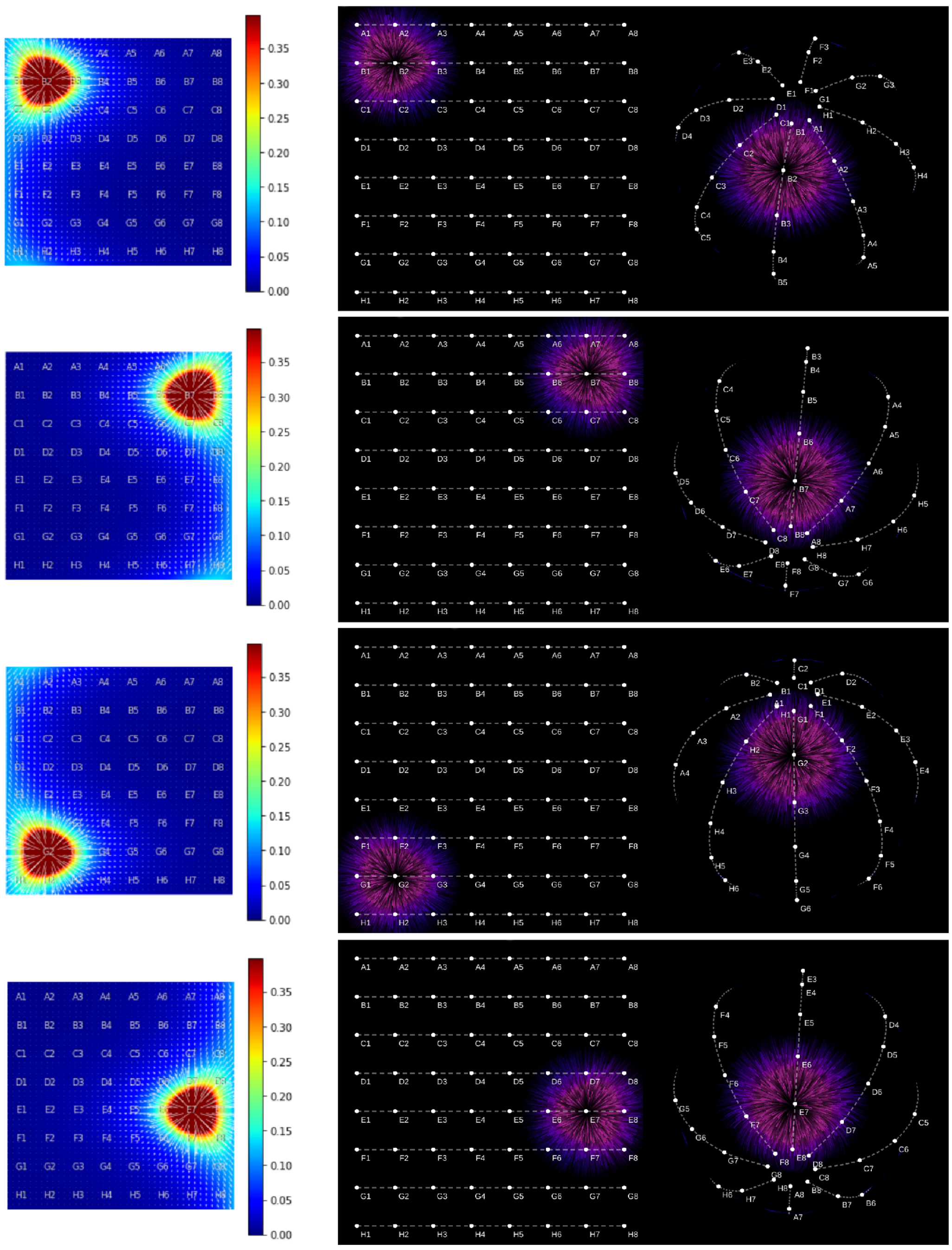


Figure 5.1: Test cases. In order from left to right: original Matplotlib plots (ground truth), 2D particle plot and 3D particle plot. A probability mask is used to concentrate particles only in the source. Colorbars are simply showing flow magnitude.

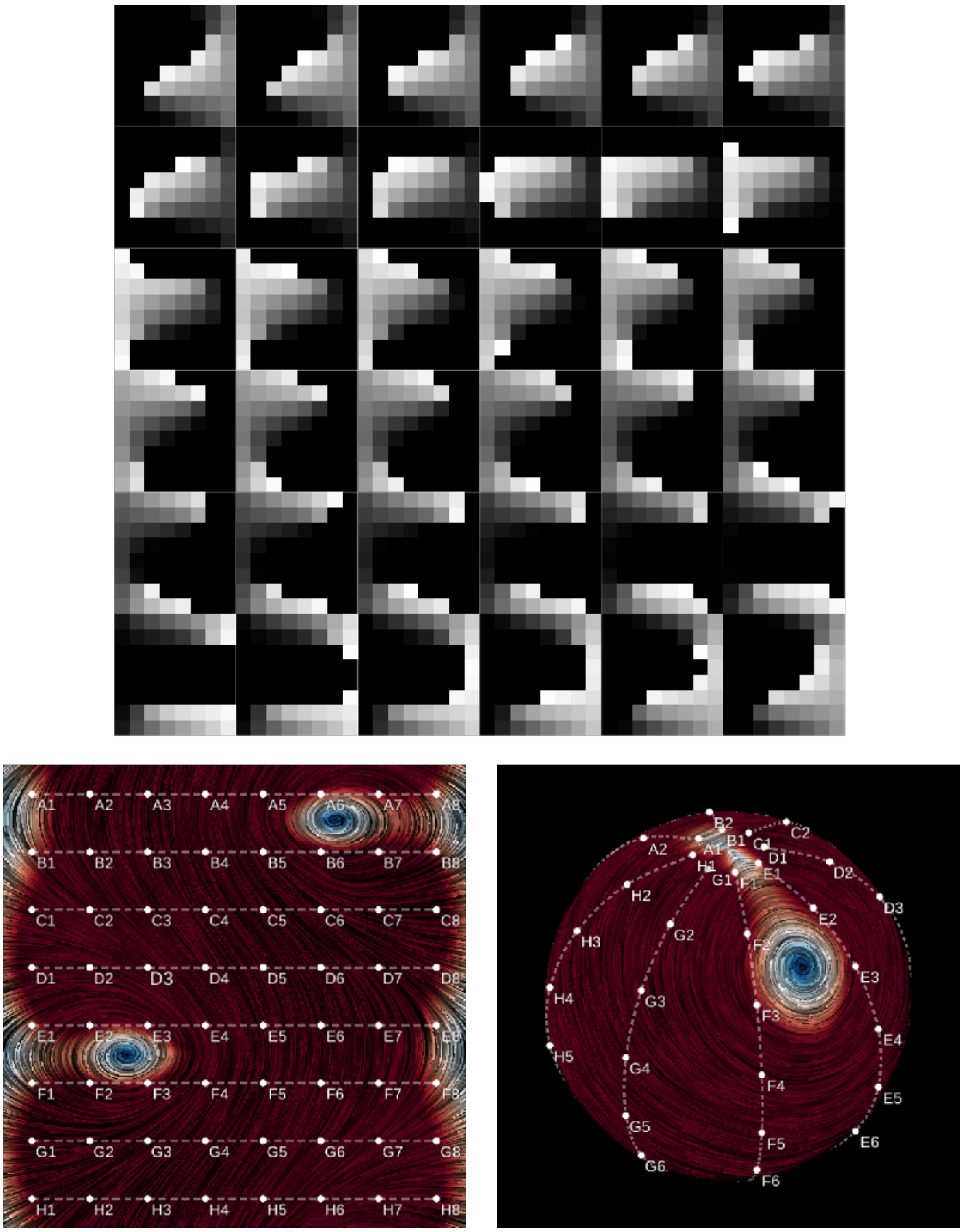


Figure 5.2: Pure rotation test data and the resulting visualization.

5.2. Performance Testing

When testing for performance, it is important to note the environment in which the software will run. Usually, plenty of GPU power will be available on hospital deployed laptops which usually have discrete GPUs on board. But this software is also going to be used by doctors/scientists with potentially less powerful GPUs. Therefore, we should optimize for performance as much as possible while aiming for at least 30 frames per second (FPS), but 60 FPS should be easily reachable given the scale of the problem.

Both particle and LIC visualization methods utilize hardware acceleration through WebGL, mostly through heavy shader use. The geometry complexity itself is very minimal (<10k triangles), so pixel shaders, fill rate and latency are expected bottlenecks.

The methods are compared on a mid-range laptop of the following configuration: Intel Core i7-8550U quad core CPU, Nvidia MX150 2GB GPU, 16GB RAM, running Ubuntu 19.04. In WebGL, performance can be tracked in a few ways (on Google Chrome):

1. Enabling "FPS meter" (figure 5.3) shows FPS and memory usage. The visualizations take only a few dozen MB at most.

2. In dev tools, "Profiling" tab can record all JavaScript callbacks and how long they take, also known as flame graph (figure 5.4).

3. Entering `about:tracing` in URL box and recording rendering activity. This method provides more advanced insights and timeline about individual calls made to the GPU.

Particle plot turned out to be a clear winner at solid stutter–free 60 FPS even at high resolutions and thousands of particles (1024x1024, 16k particles). LIC plot, on the other hand, works well for smaller resolutions (50+ FPS at 400x400, 30+ FPS at 700x700), but scales badly because of large amount of work needed per pixel (calculating a whole streamline segment). Particle plot is also a favorite, because of better perceived visual fidelity and intuitiveness (tested with target audience consisting of medical staff).

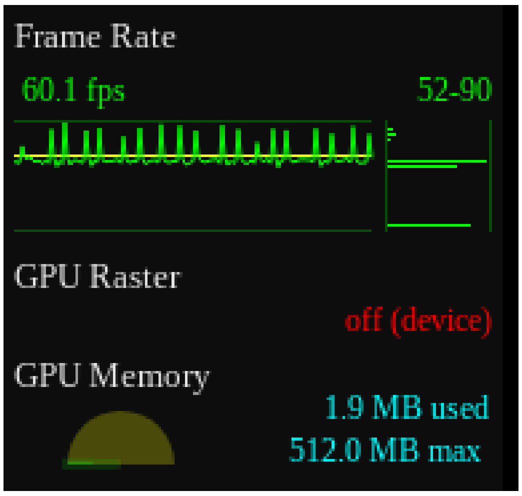
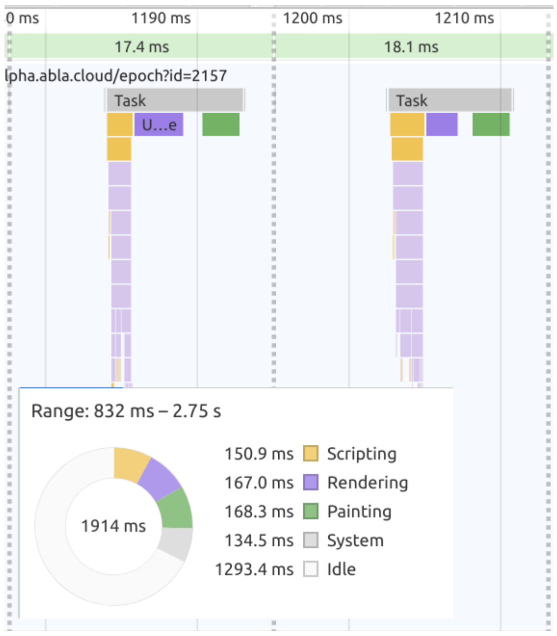


Figure 5.3: Google Chrome's "FPS meter". It also shows memory usage.



Range: 832 ms – 2.75 s

150.9 ms	🟨	Scripting
167.0 ms	🟪	Rendering
168.3 ms	🟩	Painting
134.5 ms	⬜	System
1293.4 ms	⬜	Idle

Figure 5.4: Google Chrome's "Performance" tab, flame graph. The large column is flow visualization, the other two (dark purple and green) blocks are HTML updating. This could be further optimized.

6. Conclusion

The problem of visualizing flow fields is well–known in computer graphics. In this thesis I have summarized most important methods: quiver plot, streamline plot, particle plot and line integral convolution. Implementations of the particle plot and LIC plot are presented, specialized for displaying Electrographic flow around a basket catheter. Both 2D and 3D approaches of each method are implemented. For 3D approach, methods for projecting flow and interpolating the surface are proposed, which work well for a reasonably distorted basket. Finally, the methods are tested for correctness using mocked data showing expected results.

Particle plot came out as the preferred method by the end users (medical staff) because of its intuitiveness and better visual fidelity. It is also a better choice from performance perspective, since calculating convolutions in real–time requires powerful hardware, especially for large resolutions.

The resulting flow visualization library is a fast, compact Python library, based on extending Bokeh. It is easy to use from a Jupyter notebook, but has limited application in other fields because of its specialization. Further improvements could be made, for example: optimizing HTML performance, allowing specifying 3D electrode locations from Python, additional tools like 3D annotations, orientation indicator etc.

Figure 6.1: Photo from a visit to a hospital in Hamburg, experiencing first hand what is important in an EP lab during a procedure.

Bibliography

Barbara Bellmann, Tina Lin, Peter Ruppersberg, Marit Zettwitz, Selma Guttmann, Verena Tscholl, Patrick Nagel, Mattias Roser, Ulf Landmesser, and Andreas Rillig. Identification of active atrial fibrillation sources and their discrimination from passive rotors using electrographical flow mapping. *Clinical Research in Cardiology*, 107(11):1021–1032, 2018.

Brian Cabral and Leith Casey Leedom. Imaging vector fields using line integral convolution. Technical report, Lawrence Livermore National Lab., CA (United States), 1993.

Edwin Catmull and Raphael Rom. A class of local interpolating splines. U *Computer aided geometric design*, stranice 317–326. Elsevier, 1974.

Ian McEwan, David Sheets, Mark Richardson, and Stefan Gustavson. Efficient computational noise in glsl. *Journal of Graphics Tools*, 16(2):85–94, 2012.

mrdoob et al. Threejs. `https://threejs.org/docs/`. Accessed: 2019-06-21.

Paulus, Stefano Benussi, Kirchhof, Dipak Kotecha, Anders Ahlsson, Dan Atar, Barbara Casadei, Manuel Castella, Hans-Christoph Diener, Hein Heidbuchel, Jeroen Hendriks, et al. 2016 esc guidelines for the management of atrial fibrillation developed in collaboration with eacts. *European journal of cardio-thoracic surgery*, 50 (5):e1–e88, 2016.

Appendices

A. Parameters

1. Common parameters to all visualizer models are:
   - `show_param_gui` Set to true to show parameter GUI.
   - `param_gui_closed` Set to true to close parameter GUI on load.
   - `param_gui_location` String: "right", "bottom" or "auto".
   - `normalize` Set to true to normalize the flow field before render.
   - `time_scale` Usually $1/60$ s, specifies the "speed" of the animation without influencing the look too much.
   - `source` Data source of type `ColumnDataSource` containing the flow field data.
   - `flow` Target column name where flow field data is found in the source.
   - `use_pseudo_color` Set to true to color the flow field using custom color instead of magnitude.
   - `color_map` Name of any Bokeh color palette used for foreground.
   - `color_map_low` Low cutoff for color palette.
   - `color_map_high` High cutoff for color palette.
   - `bg_color_map` Name of any Bokeh color palette used for background.
   - `atrium` "LA" or "RA". If specified "LA", the basket will rotate by $90°$ around `z` axis to mimic the orientation in real life.
   - `slider` Data source can contain multiple flow maps, and a slider can be used to flip through them.
   - `animation_delta_time` If specified >0, it will flip through the flow maps using the given delta time.
2. Common "particle plot" parameters:
   - `particle_count` Number of particles
   - `image_half_life` Time it takes for trails to lose 50% brightness.

  - `step_size` Size of the "step" a particle takes on unit magnitude flow in one second.
  - `kill_rate` Rate at which particles' life gets reduced.
  - `speed_penalty` Additional kill rate increase for particles in fast—moving flow.
  - `attack_edge` Numeric value in $[0, 1]$ range representing when particles achieve highest brightness.
  - `release_edge` Numeric value in $[0, 1]$ range representing when particles' brightness starts to fall back to 0.
  - `particle_size` Size of the particles in pixels.

3. Common LIC plot parameters:
  - `noise_scale` Scale of the simplex noise. Higher values means smaller noise specks.
  - `noise_thr_low` Threshold for noise output to be 0.
  - `noise_thr_high` Threshold for the noise output to be 1.
  - `step_size` Step size done by midpoint method.
  - `kernel_coef` Number of periods in animation kernel.

B. Types of Field Lines

Streamlines are a family of curves that are instantaneously tangent to the velocity vector of the flow field.

Streaklines are the locations of points of all the fluid particles that have passed continuously through a particular spatial point in the past.

Pathlines are the trajectories that individual fluid particles follow. These can be thought of as "recording" the path of a fluid element in the flow over a certain period.

Timelines are the lines formed by an ordered set of fluid particles that were marked in the past, creating a line or a curve that gets displaced in time as the particles move.

In a time-invariant (steady) flow, streamlines, streaklines and pathlines coincide. Definitions slightly modified from Wikipedia[1].

[1] https://en.wikipedia.org/wiki/Streamlines,_streaklines,_and_pathlines

Interactive Visualization of Electrographic Flow using WebGL

Sažetak

Elektrografsko mapiranje toka je nova tehnologija koja se koristi pri ablaciji srca. Vizualizacija dobivenog toka je važan dio operacije. U ovom radu prezentirane su dvije različite metode vizualizacije: konvolucija linijskim integralom i korištenjem čestica. Obje metode su nadograđene tako da mogu vizualizirati 3D reprezentaciju toka omotanog oko košarastog katetera. Konačno, vizualizacije su testirane korištenjem generiranih podataka, te je napravljena usporedba prema brzini izvođenja i intuitivnosti krajnjim korisnicima.

Ključne riječi: vizualizacija toka, srce, kateter, webgl, typescript, interpolacija povrsine, cestice, linijski integral, konvolucija, EGF

Title

Abstract

Electrographic flow mapping (EGF) is a technique used for aiding catheter ablation when treating atrial fibrillation. Visualizing the resulting flow map is crucial part of the process. Two different visualization methods are proposed: line integral convolution (LIC) plot and particles plot. Both methods are extended to display 3D representation of the flow map wrapped around the basket catheter. Finally, the representations are tested for correctness using mocked data and evaluated on performance and intuitiveness with target audience (medical staff).

Keywords: flow visualization, heart, basket catheter, webgl, typescript, surface interpolation, particles, line integral, convolution, EGF