https://doi.org/10.2352/EI.2023.35.1.VDA-397

FCLWebVis: A Flexible Cross-Language Web-based Data Visualization Framework

Nguyen K Phan, Guoning Chen; University of Houston; Houston, TX/USA
George Navarro; University of Texas; Austin, TX/USA
Reshmitha Muppala; Round Rock High School; Round Rock, TX/USA
Jonathan Chu, Sunny Kim; Klein Cain High School; Houston, TX/USA

Abstract

We present a new web-based, client-server data processing and visualization framework that supports a flexible workflow, enabling the user to customize different data processing and visualization tasks with tools implemented in different programming languages. Our framework supports server-side applications developed with different languages, allowing visualization researchers to easily make their new techniques available to the target users. The client-side of our framework is implemented in the web browser environment with customizable interface and visualizations. We describe the design of the architecture of our framework and the process of adding new user-defined tasks, followed by the demonstration of the proposed framework on a number of data processing and visualization tasks.

Keywords: *Web-based visualization system; Client-server; Scientific visualization.*

Introduction

Every year dozens of new techniques for scientific data visualization and processing are published. However, to utilize these new techniques, not only does a user need to know the programming languages and frameworks required to build the program, but also the right computing architecture and operating system designed to run these techniques are required (which can often be very expensive). In the meantime, larger projects that make use of a server-client setup may require extensive experience in both front-end and back-end development.

One recently popular way to address the above challenge and make the new visualization techniques available to the public is to create a web service that utilizes the method on the server-side and delivers the resulting visualizations through the client's web browser. This method allows anyone with a web browser to gain access. Unfortunately, not only do such web services incur very high operating cost to maintain the servers, but it also requires the researcher to have extensive knowledge of back-end and front-end web development. Rather than creating single-purpose web-services that require the visualization technique researcher to have experience in full stack development or hire an expensive web developing service, a more desirable solution is a customizable and scalable web service that researchers can use to rapidly deploy their new visualization methods as an accessible web service that anyone can use. Researchers can utilize their own servers to run the web service, or community-funded public servers can be set up to receive contributions from researchers by implementing their new methods into the public web services. If successful, such a framework can rapidly improve the public's accessibility to new visualization techniques. Before describing the design of such a framework, we start with a list of requirements for this new framework. Note that these requirements are obtained through our survey of similar frameworks and our collaborations with domain experts, which may not be complete. Nonetheless, they serve as a starting point for the development of such a framework.

Requirements

The first major requirement is a capable web application client that allows the end-user to easily view and interact with the visualizations of the data. It should also be efficient to run on low-end devices such as mobile phone browsers. However, as they can only run inside web browsers, they are extremely limited in processing power and data storage, which will greatly limit the scope and scale of the required visualization/processing tasks, hence, we move on to the next requirement. The web client should be able to transfer the majority of the heavy work to the web server. This heavy workload includes the storage, processing (data parsing, filtering, and transformation), and visualization of the user-uploaded data sets. However, if the required task or data set is sufficiently small then it is reasonable to perform the data processing and visualization on the client side which should be more responsive and provide a better user experience overall. Thirdly, to help decrease the workload and experience needed for researchers to implement their own visualization tasks into our web service, the service should be able to support the implementation of visualization and data processing tasks from different programming languages. This cross-language capability should encourage more researchers to use the framework to integrate visualization tasks using the programming language(s) that they are familiar with. The framework needs to be able to conduct efficient data transfer between executions of code written with different programming languages, in addition to having the capability to save and load the visualization state data for sharing. Lastly, there needs to be an effective abstraction of the method implementation process. Not only should the source code be well-documented and straightforward to set up, but the process of implementing the visualization task (on the client-side) and the data processing task (on the server-side) also needs to be intuitive and easy to follow.

In summary, a desirable client-server framework for web-based scientific data processing and visualization should possess the following features.

- Visualization Task Abstraction
  - *Task API*: new visualization tasks can be created and modified.

- *Flexible Workflow:* visualization tasks defined within the framework cannot be made from static class definitions and hard-coded user interface elements.
- User Session Data
  - *Save/Load State Data*: the current state of the program/visualization can be saved and loaded to prevent loss of progress.
  - *Scene Export*: Scene data can be exported into other formats.
- Infrastructure Flexibility
  - *Server/Client Setup*: Server handles data loading, processing, and filtering, and sends relevant data to the client for efficient visualization. This setup allows easy scaling.
  - *Cross-language API:* Server should allow the data loading, processing, and filtering to be done with different handlers implemented using different programming languages. (Note, this should not be confused with different languages used between the server and the client application.)

There are a few existing frameworks that aim to address the above requirements. Those frameworks provide many desired features listed above, but none of them provides all these features (see the **Related Work** section). To address this challenge, we introduce a new web-based data processing and visualization framework, **FCLWebVis** [1], that encompasses all the above features. Note that while all the above requirements are satisfied within our framework, the state of the FCLWebVis is more towards being a prototype as these features are not fully ready for practical usage (compared to those seasoned software and tools). Nonetheless, this work demonstrates the potential of our framework that hopefully will attract continued contributions to it.

Related Work

Despite many recent web-based visualization systems that only provide the front-end applications to process and visualize (mostly information) data locally, like those visualization systems created with the popular D3 library [3], there exist only a few frameworks that utilize the client-server architecture for scientific data processing and visualization, which we briefly review below. Note that while efficient content delivery and scaling are considered, they are not the main focus of our research. Hence, our related works are selected more towards their flexibility and difficulty in implementing new visualization tasks.

Paraview [1] is a flexible and powerful visualization tool designed for many visualization tasks. It supports user extension to the existing visualization and can handle extremely large data sets by performing parallel processing on shared or distributed memory machines. Unfortunately, there are certain limitations to Paraview that inspired the creation of our framework. Firstly, while Paraview can be set up in a client-server configuration, it is not a web-based framework and must be installed locally. Secondly, Paraview is limited by the system architecture and hardware of the system that it runs on, a user with a less powerful system may not be able to run Paraview with their desired tasks which might require a stronger system setup. If a researcher wants to directly share their methods, the best way is to host a web-service on their servers so that anyone can access and use it from the web browser. While Paraview can also be used as a servers that others can connect to, the host must share the actual address and logins of the server, which complicates the process and require the users to install Paraview locally on their own machines. In addition, Paraview's workflow depends heavily on the C++ VTK library and/or Python scripting, Paraview processing methods and filters mostly make use of VTK's data structures and methods. This prevents (or makes the process very challenging for) new visualization or data transformation methods that do not rely on VTK from being implemented into Paraview.

Paraview Web is a separate web-based 3D scientific visualization developed by Jourdain et al. [7]. It is specialized in visualizing large data sets by making use of one or more PWServer which is a Paraview-based visualization engine. Unlike Paraview which renders using the VTK engine, Paraview Web encapsulates the VTK object in proxies that can behave the same way as the VTK object, allowing the rendering process to work on both the local web browser and on the server-side (the user can choose either server rendering or local browser rendering). However, Paraview Web's robustness comes with many complexities. When implementing a Paraview Web visualization, the Paraview extension must be written in the VTK format and corresponding JavaScript proxies in Javascript for the web application. Furthermore, Paraview's intuitive pipeline design also isn't present. Without the task pipeline design, Paraview Web's user interface components must be defined manually by the developer, unlike Paraview's simple XML inputs. Hence, *Paraview Web is not a replacement for Paraview as a web application.* In contrast, FCLWebVis not only comes with a flexible workflow (similar to Paraview's pipeline) but also allows quick implementation of new tasks that can be executed in different programming languages. FCLWebVis also allows rapid creation of the task's user input components with XML, similar to Paraview. FCLWebVis is also not limited to only the VTK framework, allowing the use of other visualization frameworks and data structures.

Mayavi Project [10] is a visualization frameworks that competes with Paraview in some way. They offer very similar features to Paraview, boasting a flexible treeview workspace similar to Paraview's intuitive pipeline. It also allows extensive customization and the integration of new visualization tasks using Python scripting. However, similar to Paraview, it is mostly limited to Python scripting and not configurable as a web service.

VisIt [4] is an open source scientific data visualization and analysis tool that is interactive and scalable. In a way, it is similar to Paraview as it mainly relies on the VTK framework for data representations and processing. It also possesses a similar underlying pipeline system while not as robust. Nonetheless, VisIt boasts about 60 more importable data formats than Paraview!

Recently, Bock et al introduced the **Openspace** [2], which is an extensive and flexible visualization framework for visualizing astronomical data. Its customizable and modular design ensures the support of any present and future astronomy visualization tasks. Unfortunately, Openspace's module's customization scope is limited to the core structure defined in the Openspace-Core package, which limits its potential to astronomy-related visualization tasks only. Compared to Paraview, Openspace has a much more intuitive user-interface setup and better accessibility (i.e., accessible via any web browser), it has to compensate with a more restricted architecture, supporting a more limited set of data structures and less robust user-interface inputs.

[1] source code available at: https://github.com/MangoLion/FCLWebVis

Liu et al. [8] created a robust web application for the **visualization of 3D ocean eddies.** The web application utilizes ThreeJS [14] to display different 3D rendering tasks of ocean eddies' properties with various interactive features. The eddy extraction and tracking are performed on a separate C++ program that utilizes VTK to process ocean eddy data and can take hours to complete before the user can explore the results via the web application.

Table 1 compares the frameworks reviewed above and our new FCLWebVis in terms of the desired features listed in the **Introduction** section . As can be seen, only our new framework provides all these desired features. Note that even though existing tools and software do not provide all the desired features, they do offer easier interface for extension and intuitive user interaction, compared to FCLWebVis, which is still a prototype framework.

Table 1: Comparison of different frameworks

	Task Abstraction		Infrastructure Flexiblity		
	Task API	Flexible Workflow	Server Client	Web -based	Cross-language API
FCLWebVis	✓	✓	✓	✓	✓
Paraview	✓	✓	✓	-	-
Paraview Web	-	-	✓	✓	-
Visit	✓	✓	✓	✓	-
Mayavi	✓	✓	✓	✓	-
OpenSpace	✓	-	✓	✓	-

Design of FCLWebVis

FCLWebVis first starts with the basic definition of the base components, including the data source which is the dataset or its transformations and the task which transforms the data source and/or visualizes it. Unlike Paraview which relies mainly on the Visualization Toolkit (VTK) as the foundation for it's data representations and algorithms, FCLWebVis has more loosely defined data representations. Each data source type is a simple data container by definition, and the developer decides which task can utilize the source type as the input or as the output to other sources. A data source can be defined as a VTK data type and can interact with tasks that perform VTK methods on those data types, but this setup allows FCLWebVis's data sources and tasks to not be limited to VTK and it's data representations. A developer can easily define a new data type and task that utilizes it, and incorporate it into existing data sources and tasks.

To ensure the web service is accessible from any modern browsers running on different device types (laptop, mobile phones and tablets), the Web Application is a React JS application hosted on a Node JS server using Express JS. **Node JS** is the back-end JavaScript run-time environment that we use to implement all server instances of FCLWebVis (the processing server, web application host server, and the load balancer). **React JS** is a widely used open-source JavaScript front-end library used to build and display responsive user interface (UI) HTML elements and facilitate user interactions for web applications. To deliver the files needed to run the web application to the user's browser over the internet, we make use of **Express JS** which is a Node JS framework for content delivery. Aside from serving the web application files to the user's browser, the web application itself needs to be able to maintain a reliable connection between the web application with the processing server and transfer a potentially large amount of data between the two, and allow for the easy restoration of disrupted or dropped connections, the **Socket IO** framework is used as the main communication method over the internet. Socket IO is a popular bi-directional low latency, event-based communication framework.

While FCLWebVis supports the cross-language implementation of visualization tasks, we also take into account situations where cross-language scripting is not needed. Firstly, if the new task (visualization or data filtering/processing) is written in Javascript then Node-Gyp Addon API is not needed. Secondly, if the new task is in the form of an executable program (rather than compilable source code) then the task can be integrated using simple pipe communication instead.

Next we describe the architecture of FCLWebVis following the above design.

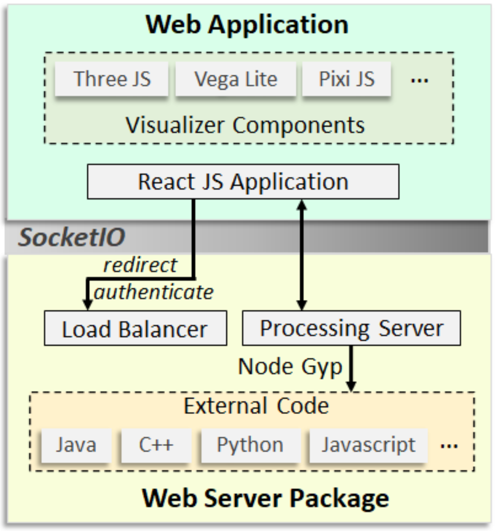


Figure 1. *The System Architecture of the web application and the server package. Note that the dashed boxes indicates that the contents are of the same types.*

FCLWebVis Architecture

Overview Our framework utilizes a client-server architecture as illustrated in Figure 1. Recall that the design goal is to create a web-based visualization framework that can accommodate any type of visualization task, the tasks themselves can be executed in any programming language or framework on the server-side. The service will also keep track of each user's workspace and allow the saving and loading of user sessions. In the following, we will describe the detailed architecture of the web-based client side and the server side, respectively.

Web Application

To address the requirement of displaying different visualization types and allow easy implementation of new visualization frameworks, **the View Panel** renders all visible tasks provided that the task has a corresponding visualization component whose type matches the view type of the current view panel. Different view types are supported based on the type of visualization. The view panel utilizes ThreeJS [14] for displaying 3D visualizations, such as surface visualization or volume rendering. For 2D tabular data visualization, such as drawing histograms and scatter plots, FCLWebVis uses Vega-Lite [12] which is a robust visualization grammar framework. And for other dynamic 2D visualizations, Pixi JS can also be used. It is important to note that new view types can be easily defined with a new ReactJS View Panel Component. For example, a natural language processing task may require a custom type of view that displays text HTML elements, then a new view component can be defined that accepts the task's output and visualizes it as a set of text elements of different colors.

Client to Server Connection. Once the user accesses the React web app, the app will contact the processing server using Socket IO. Socket IO will handle the user id, and the server creates a temporary workspace with that id. Any subsequent request (data set upload/download, visualization tasks on a data set) will be executed inside that workspace. At any time, the user can download the snapshot of the entire workspace from the web app (stored in the main memory of the web app) for viewing later. Lastly, the user's workspace data stored on the processing server can be saved and resumed at a later time.

Web Server Package

The server package holds the required components on one or more server machines required to maintain the web service. It consists of the following two main components.

Processing Server The processing server is a NodeJS server designed to handle all data processing, data storage, and workspace export requests. Its main purpose is to receive user-uploaded data sets, and for each visualization/processing task, it performs the needed data processing operations and then returns the visualization data back to the web application. The processed data is also kept so that it may be used in future tasks that utilize them. To properly manage the data set and the processed data of each task for each user, the processing server encapsulates each user's data into a separate workspace that can be saved onto disk if the user wishes to continue working on their workspace in the future or to recover user work if the session was interrupted (such as from loss of internet, or the browser was closed unexpectedly).

To summarize, the processing server handles all data storage and processing requests. It receives requests from the user (through the web application) to upload data sets or process various data processing tasks that can be used for visualization. It also sends the needed data to the web application for visualization (oftentimes the visualization data is only a very small fraction of the original data in size). It also can also back up and restore the user's work. Communication with the web application is done mainly through Socket IO.

Node Gyp Native Addon Node-gyp is a native addon that allows the execution of visualization/data processing tasks from different programming languages. It enables the FCLWebVis framework to utilize visualization tasks that are written in other languages than Javascript. Despite being written in a different language, Node JS can still run these programs natively, ensuring efficient program execution and memory usage. Currently, FCLWebVis supports visualization tasks written in either Javascript, C, C++, Python, Java, or as a separate executable program.

Executable Tasks If the new task to be added to FCLWebVis is in the form of an executable program, then instead of using Node-gyp to communicate with the task, our framework will use pipe communication. The processing server will start an instance of the executable program and communicate using pipe to send user input and receive the program's output. However, there are several limitations to this implementation. Firstly, tasks implemented with Node-gyp API (or with native Javascript) are more efficient as the task's code is executed within the same memory space as the server process, allowing faster data transfer. Secondly, pipe communication also requires more implementation work to convert the data to and from binary for the transfer process.

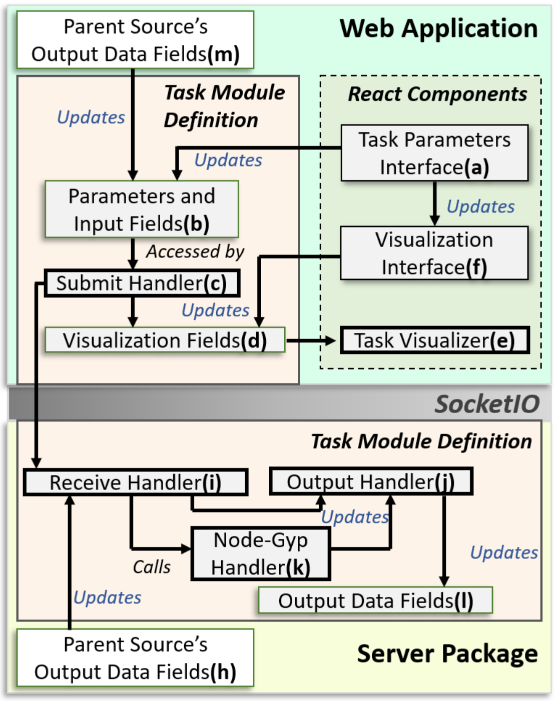


Figure 2. *The main components of an FCLWebVis Task. They are required to include a new task.*

Task Definition

An FCLWebVis's Task is the basic building block of the visualization pipeline. A task can be used to simply store data (as a source) or the results of the transformation/filtering of data (of another source). A task can also be used for analysis and visualization. To fully define and deploy a new task for the user to access, *one needs to provide the task definitions in both **server package** and **web application***. To make the process of implementing a new task into FCLWebVis (both server package and web application) intuitive, we split the task into different definition containers and React components (Figure 2). Each container and component is easy to define and implement for any researcher who wants to add their visualization method into FCLWebVis as a new task. Here we explain the roles of each task container/component, and how they interact with each other, organized based on whether they are located in Server Package or in Web Application. The complete details of how these components and containers behave and how they should be implemented are in the documentation on the GitHub page of FCLWebVis.

Server package task definition

In the processing server, the task's definition includes the task's main properties (component (b) in Figure 2), such as the task's input data type and output data type. Any referred data type needs to be defined first. An FCLWebVis data type holds a set of data fields. For example, a vector field data type holds a set of data fields for its properties from dimensions and spacing, to a large data field that stores the array of vector values.

Receive Handler receives the input data required to run the task, including the data sent by the web application, such as the user input parameters, and any uploaded data sets. It also receives the output data of the parent task (if there is one). For example, the user creates a vector field task. The task created on the server will then receive the vector field data from the web client. Since the vector field by itself does not have any parent task, no parent output data is given. Then the user creates a streamline tracing task from the vector field task. The streamline tracing task on the server will receive the streamline parameters (seeding points, step size, integration length/time) from the web client and the vector field data from the parent vector field task saved on the server.

Node Gyp Handler bridges the FCLWebVis framework with the execution of the task in a different programming language. The input data for the task needs to be properly defined so that NodeJS can send them to the task's program and receive the results. The results of executing the task will then be sent to the output handler. The Node Gyp Handler is optional, and data from the receive handler can be sent directly to the output handler in some tasks (e.g., storing data used by other tasks).

Output Handler receives the output data after the task execution. It decides which data to save to the task's output stored on the processing server and which data is necessary for client visualization. The latter data will be sent back to the web client for visualization. This means that the web application keeps a different representation of the data compared to the processing server. This visualization data is only a small fraction in size compared to the data's original representation on the processing server.

Web application task definition

This process is similar to the task definition in the processing server. The task's main properties, any new data types, and output must be defined.

Task Parameters Interface is a React component that holds the input HTML elements for the task's input parameters. FCLWebVis has a large number of default input components from text fields to list boxes that can be added with a simple line (and the data field that the element will update). New input components can also be defined by creating new React Components. Updates to the task parameter inputs will directly update the task's data fields. Additionally, data check handlers can be defined for each data field to ensure the input data is valid, before updating the data fields with the user input value. Note that input data of the tasks are defined in the task's Input Fields.

A task's parameters panel can be easily defined using existing input components (e.g., text box, drop-down, slider bars, etc.). If a new input component is needed, then it can be defined as a new ReactJS component and added to the task panel.

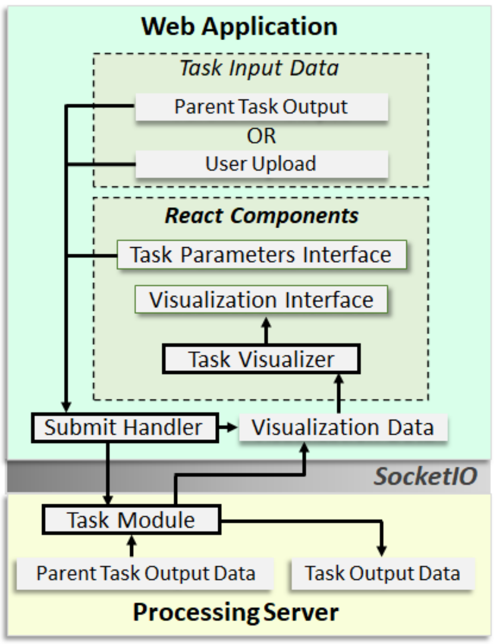


Figure 3. User Task Creation and Modification.

Submit Handler will be triggered when the user clicks the Submit button. It will determine from the task's data fields which data to be sent to the processing server, and which data to be directly sent for visualization. Some tasks may not require a data processing server at all and can be visualized directly, and in that case, the submit handler does not need to send any data to the server.

Visualization Fields are data fields of the task that are solely used for visualization. This data is often much smaller in size than the original data set.

Task Visualizer is a React Component that utilizes the visualization fields of the task to visualize. The type and definition of the task visualizer component depend on the current view type. For example, if the current view panel uses ThreeJS to render a 3D scene, then the task visualizer component of the task needs to be of type ThreeJS React component to be rendered by the view; otherwise, the task will not be rendered in the current view panel.

Visualizer Interface is a React component that holds input components for adjusting the task's visualization settings. Any adjustments to the input values will send updates to the task visualizer component to update the visualization. Note that this behavior is togglable.

User Task Addition and Modification Logic

There is a need to better illustrate how the main components of FCLWebVis interact with each other. In this section, we describe the internal process that happens once the user adds a new task (or modifies an existing task), and clicks Submit, the following process will begin (Figure 3). This section is not to be confused with a developer implementing a new visualization task into the framework.

Firstly, the Submit Handler of the task will be called. It retrieves the Task Input Data and the Task Parameters Input. If the task was created from uploading a fresh data set, then the task input data is the user uploaded data set; otherwise, if the task was created as a child task of another task, then the task input data is the parent task's previous output data (step a). Note that the parent task data represented in the web client is often only the fraction of the data needed for visualization, while the main data set is stored in the processing server. This is because the web client is limited by the memory and resources of the web browser and must conserve memory. The task parameter inputs are the parameter values of each input component in the Task Parameters Interface. For tasks that do not require additional parameters (such as uploading a new data set), this component can be blank. The submit handler will then decide which data will be sent to the processing server, and which data to be directly sent to update the visualization data fields of the task.

Once the processing server receives the submitted data, it will call the corresponding task module and execute the task (task module in Figure 3). Once the task is finished and the result data is generated, the task module's output handler will decide which data to be sent to the web client and which data to be saved into storage as the current task's output data (so that future tasks created under this task can make use of the output data). The data sent to the web client will be used to update the visualization data fields of the task. When sufficient visualization data is provided, the task visualizer component will automatically update the visualization in the view panel. Due to React's efficient responsive design, the rendering update only applies specifically to the changed properties of the specific task visualizer and the web client does not have to re-render all current task visualizations.

The Practice of Implementing a New Task

After describing the necessary components that a task should have and the logic of how a task can be added and modified on FCLWebVis, in this section, we will briefly describe the practice of adding a new task to FCLWebVis. Please note that the detailed instructions, syntax, and file directory references are in the documentation attached to the project's GitHub page.

Task Registration

We begin with the web application package and create a .js file with the task name in the *tasks* folder. *registerFileType*() is used to register any new data set file types that the task will handle. Note that if the file type is already implemented then this step is not needed.

```
registerFileType({
    name: 'file_type_name',
    fields: {},
    sendToServer: false
  })
```

Then, *registerTaskType*() will register the new task and specify the task's input and output file types, task properties (such as if the task is client-side only), task input parameters, and methods (initialization, what to do when submitting to the server, and so on).

```
registerTaskType({
    name: 'task_name',
    fields: {},
    init:function(){},
    onSubmit:function(){},
    inputFileType: 'file_type_name',
    outputFileType: 'file_type_name',
  })
```

Note that some additional required steps are omitted here (but are detailed in the documentation). These steps involve modifying some of FCLWebVis's source code files *manually* to add the corresponding "include" statements and register new entries into the main index mapping. We are working to remove this limitation to allow a more self-contained packaging process for the creation of new tasks that can be conveniently added to FCLWebVis without the modification of source code.

Input Component Implementation

Here we define the React JS UI components for the new task. This should be in the same previous .js file that holds the registration methods. Next, we define a *TaskInput* component that holds the UI input elements. Each element is connected with one of the task's parameters. The following is an example XML of a streamline trace task's input:

```
<DropdownComponent name='direction'
values={['both', 'forward', 'backward']} />
<TextField name='stepsize'  />
<TextField name='length'  />
```

If we are defining a new file type, and this file should be displayed in a visualizer, then we can create a *FileInput* component that holds the input elements connecting to the visualization parameters of the file type. For example, a streamlines file type may have a geometry parameter to select whether the lines are displayed as lines or tubes.

Note that a short import statement is needed for each UI element type that will be used in the interface component. We hope to remove this step in the future by refactoring the Node JS project to make UI elements available without the need to write individual import statements.

View Component Implementation

For tasks or file types that are displayable in a visualizer, we need to define the corresponding view components for the file or task. A view component is a React JS component that will be input into the visualizer. Depending on the type of visualization framework used, this view component should be implemented accordingly based on that framework's API. For example, if a view component for displaying streamlines is a ThreeJS component then it needs to return the correct ThreeJS syntax based on ThreeJS documentation. In our current web demo, the Streamline Tracing Task's view component displays a sphere for each seeding point, and the Streamline file type's view component displays the streamlines. Here is an example code for creating some empty views and input components, and exporting them:

```
let TaskView = ({ task }) => {return null}
let TaskInput = ({ task }) => {return null}
let FileView = ({ file }) => {return null}
let FileInput = ({ file }) => {return null}
```

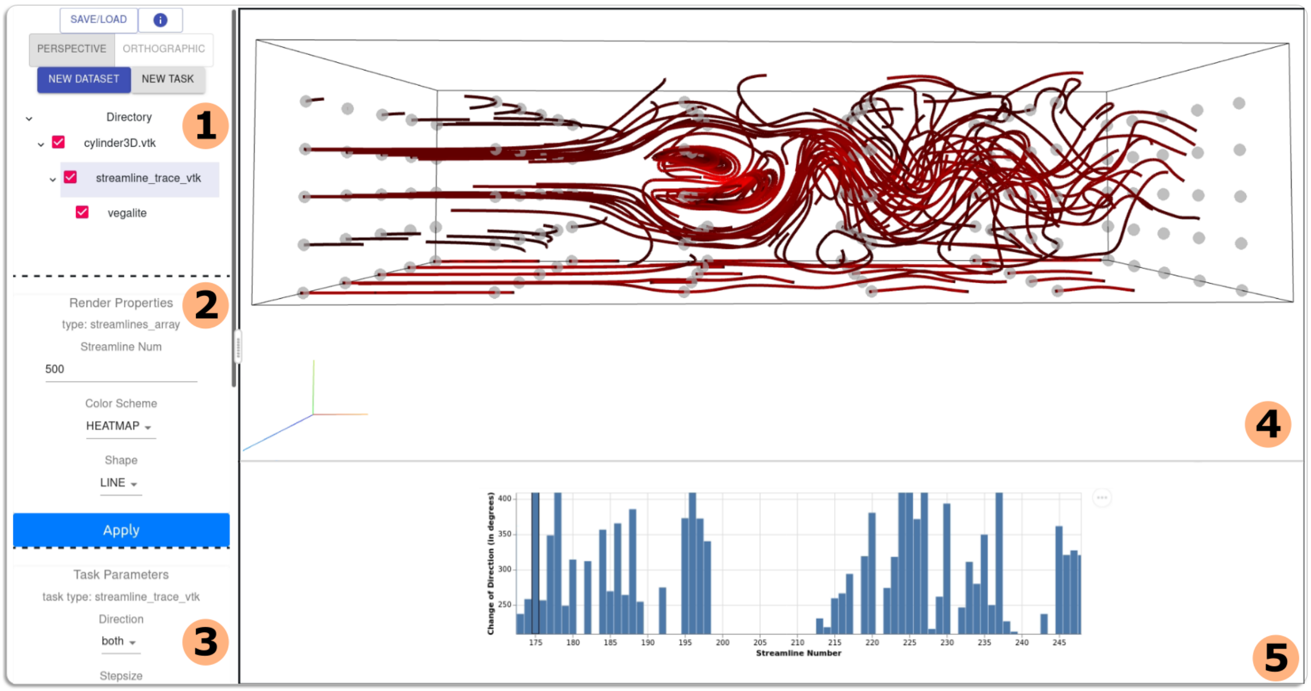


Figure 4. *Labeled interface panels and the demonstration of the streamline tracing task and Vega lite chart view on the flow past a cuboid cylinder simulation [15]. (1) The workspace directory panel holds all created tasks and data sets. The checkbox indicates if the task should be rendered in the visualization panel. (2) The render properties panel holds user input elements of the visualization data. (3) The task parameters panel holds user input elements of the current task's input parameters. (4) Visualization panel visualizes all checked tasks (provided that the task's visualization component is of the same type). This specific panel is of type ThreeJS which can display 3D visualizations. (5) The Vega-Lite chart panel is a visualization panel of type Vega-Lite, used to display a selected Vega-Lite chart task.*

```
export {
  TaskView, TaskInput, FileView, FileInput
}
```

Server Side Processing

On the server package, we first create a new method for the task in *tasks.js*. In this method, we write in Javascript what the server should do when it receives the input data for this task. Any cross-language methods can be called here for processing and receiving data (details in our documentation on the project's GitHub page and the node-gyp-addon documentation). Then, we return the new results that are sent back to the client. This result is also stored on the server locally for that user.

```
example_task: function(parameters, data){
  //parse the parameters
  //call any cross-language methods
  //return the new result back to the client
  //and also store it in user workspace here
}
```

Lastly, we write the cross-language hookup methods in the *addon* folder, based on which language the task is written in.

One current limitation of FCLWebVis is the need to modify one of the server package's source code files to add the new task's method and an import statement for the cross-language API method. We hope to remove this requirement in the future to allow seamless integration of new tasks without the need to modify source code (as the task should be added as a separate plug-in package, similar to how ParaView includes new plug-ins).

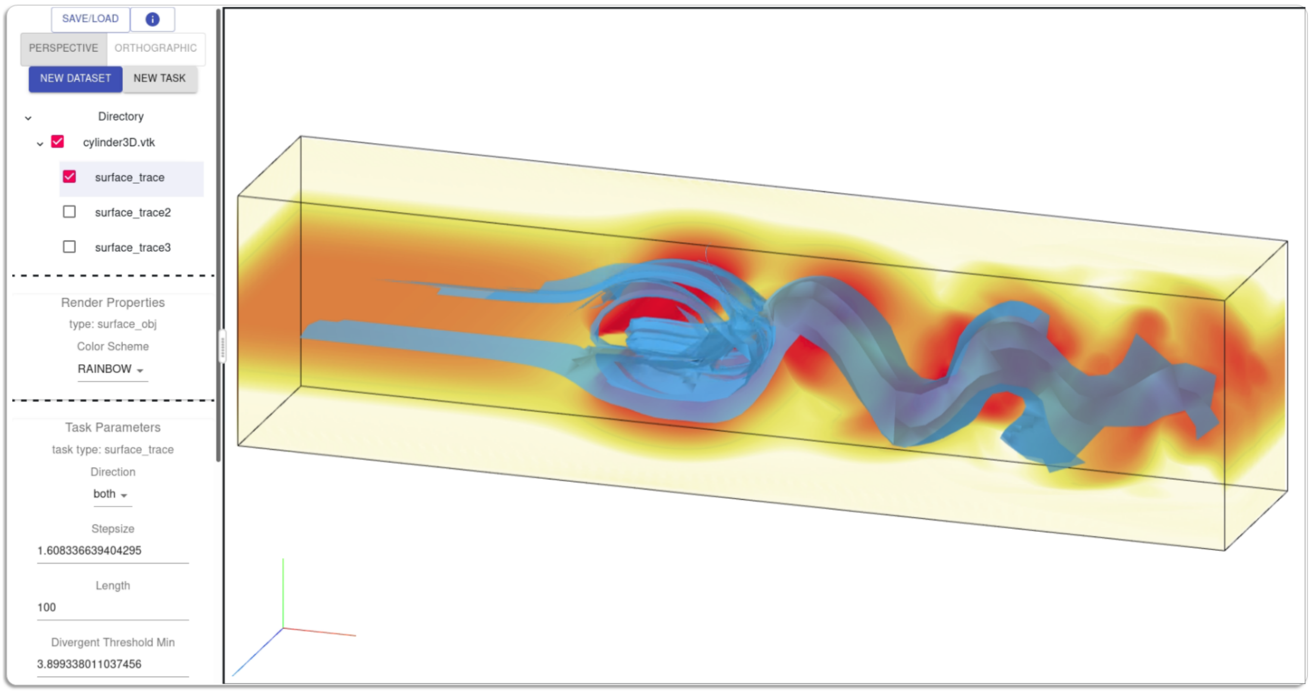
Use Cases and Applications

Next, we demonstrate how to use the proposed framework to achieve the visualization of 3D vector fields and integrated exploration of the vector fields via their abstract information.

3D Vector Field Visualization via Three JS View

FCLWebVis supports 3D data visualization with the ThreeJS visualizer. 3D Data sets by default will show a bounding box indicating the range of the data set. Additional tasks can be created for different visualizations. Figure 4 shows a vector field task that shows the vector field's bounding box, combined with a streamline tracing task that visualizes the streamlines traced from a set of seeding points. Seeding points and other integration parameters for streamline computation can be adjusted by the user. Note that since the vector field data lies inside the processing server, the streamline tracing task also needs to be executed on the server before sending the resulting streamlines back to the client.

Additionally, the streamline tracing task's visualizer component can display the seeding points by accessing the task's parameter data. Each 3D dot indicates a seeding point. Oftentimes, the task visualizer will also need to visualize the task input parameters as a guide for better adjustments and observation before executing



- ✓ cylinder3D.vtk
  - ✓ surface_trace
  - ☐ surface_trace2
  - ☐ surface_trace3

Render Properties
type: surface_obj
Color Scheme: RAINBOW

Task Parameters
task type: surface_trace
Direction: both
Stepsize: 1.608336639404295
Length: 100
Divergent Threshold Min: 3.899338011037456

Figure 5. *Demonstration of the stream surface construction and volume rendering*

the task. Additional visualization tasks for vector fields include stream surface construction [5] and volume rendering (Figure 5).

In the future, we plan to add VTK.JS as an additional 3D visualizer which is more specialized for VTK data types and processing methods, since many visualization techniques utilize the VTK framework.

User Exploration via Vega-Lite View

To demonstrate FCLWebVis's data analysis capabilities, we show a split view of the streamline visualization (using the Three.js view type) and a bar chart (using the Vega-Lite view type) of the average direction change of the first 50 streamlines (Figure 4). The Vega-Lite task component is straightforward, with two input text boxes. The first input accepts a data-prepossessing Javascript method that can transform the parent task's data (in this case, the streamline tracing task) into the desired chart data. If not provided, the output data of the task will be used as the input for the chart. The second output accepts a Vega-Lite configuration (as a file or text) that will be used to create the chart based on the input data. This allows each Vega-Lite task to fully utilize the Vega-Lite framework to create one or more interactive and customized charts. However, this requires the user to know how to write the Vega-Lite grammar and may spend some time performing prepossessing of the input data. In Figure 4, the streamlines data needs to be prepossessed by computing the average direction change of each streamline. We are actively working on integrating the Voyager 2 framework [17] to greatly simplify the Vega-Lite chart creation process by relying on Voyager 2's intuitive drag-and-drop interface and dynamic chart generation.

Additional interactivity between the Vega lite plot and the 3D view can be achieved by registering a custom event listener to the generated vega view that, when the user interacts with it, will update the corresponding data entry in the streamlines view component to display changes to the 3D view (Visualizer component). This is currently achieved using the Vega-Lite task within FCLWebVis as demonstrated in Figure 6. In this example, the user selects streamlines of interest through the Vega-Lite plots showing some accumulated characteristics of streamlines [18]. However, to achieve this linked-view setup, the user first needs to understand how to write the custom event listener for Vega-Lite by themselves. In the future, we will develop a more generalized Vega-Lite component for FCLWebVis that can allow a more intuitive, non-coding approach to set up the Vega-Lite for 3D view interaction.

QuadriFlow Mesh Quadrangulation

To demonstrate the ease of implementing new tasks, we added QuadriFlow [6] task to FCLWebVis. Quadriflow is a recent automatic quadrilateral (quad-) mesh generation method. Comparing to previous approaches, Quadriflow tends to generate quad-meshes with smaller numbers of irregular vertices and better-shaped elements. Since the authors of Quadriflow have published their method as a compilable executable program, we integrated it into FCLWebVis as a new visualization task. The integration process is very simple and straightforward. First, we define a new task definition with a new data type (e.g., quad mesh); then, we add some instructions on the server side to run the QuadriFlow executable with the input data, retrieve the output quad mesh file, and send it back to the client. Next, we define a quad mesh view component for the task of rendering the quad mesh in the Visualizer component. Figure 7 shows an example result of QuadriFlow after uploading a triangular mesh.

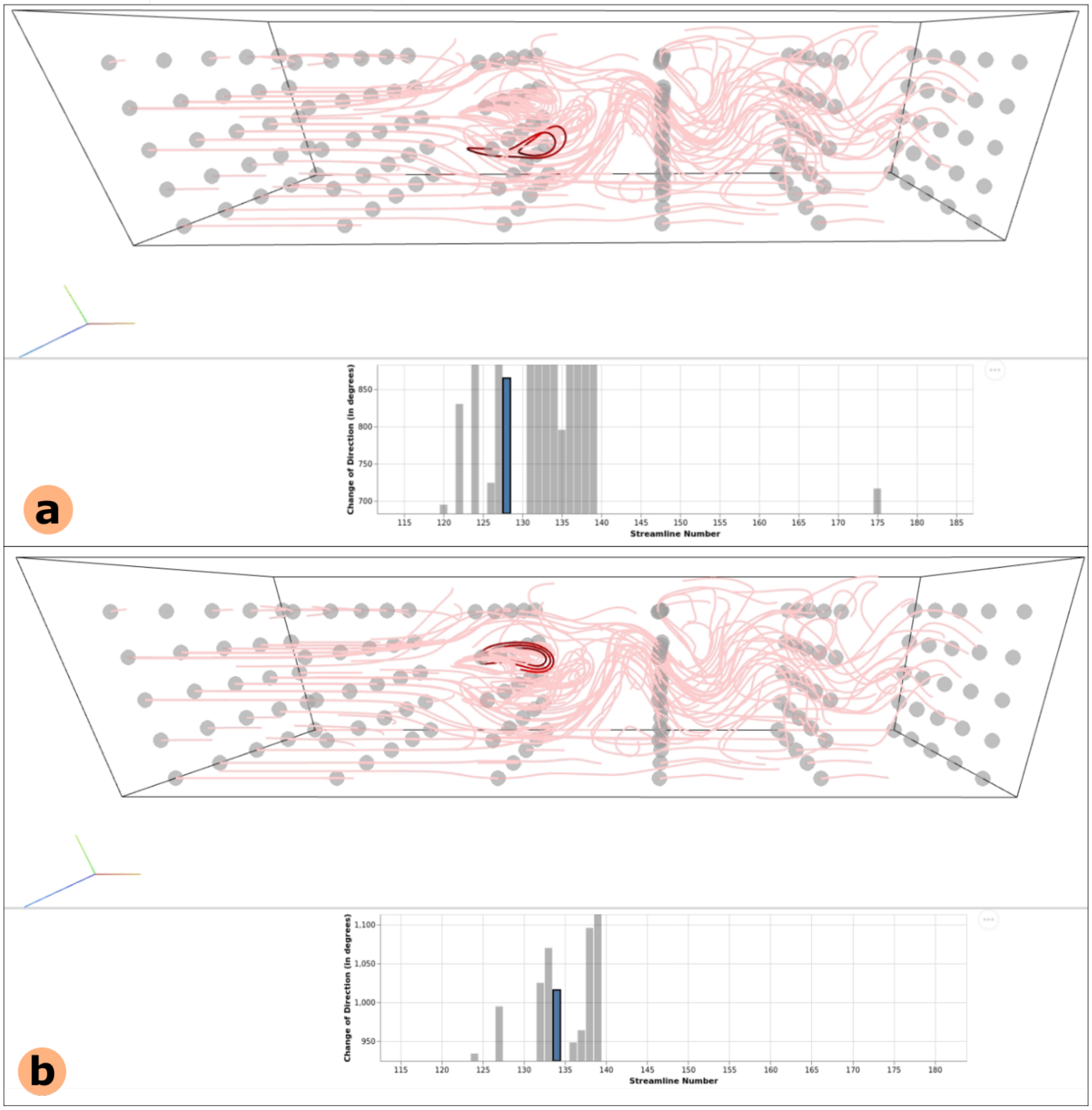


Figure 6. *Two examples of the interaction between the Vega-Lite plot and the 3D view. (a) and (b) each shows the highlighted streamline whose index is the selected bar entry on the Vega-Lite plot.*

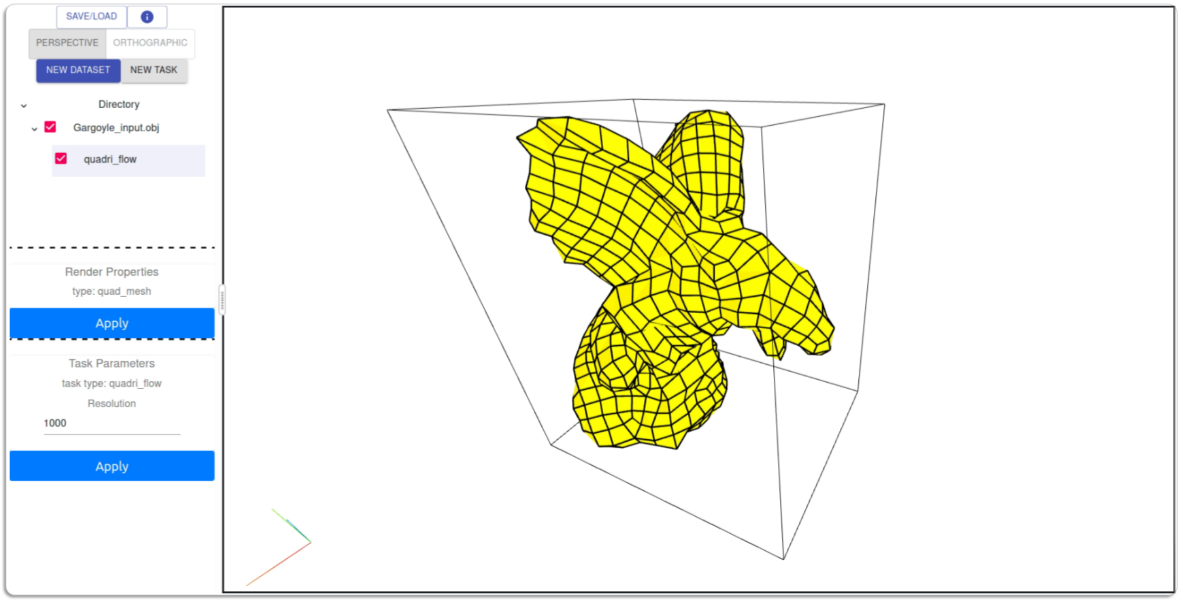


Figure 7. *Demonstration of the QuadriFlow Quadrangulation task.*

3D Vector Field Vortex Core Extraction

Another example task we implemented is the extraction of vortex core lines within a vector field dataset [11]. We decided to first write a Python script that makes use of the PythonVTK framework instead of using VTK in C++, to demonstrate FCLWebVis's cross-language capability. Similar to QuadriFlow's task, the vortex core extraction task in FCLWebVis first calls the external Python script, provides the input data and parameters, and then forwards the resulting file to the web client for visualizing the core lines in the 3D view. Note that due to this task being implemented in Python, and the original vector field data was implemented in C++ as the vtkUnStructuredGrid data type, it cannot be directly transferred to PythonVTK's vtkUnstructuredGrid and had to be re-parsed which introduced more computation time. Figure 8 shows the core lines extracted from the Benard convection flow. A core line is highlighted based on the user interaction on the Vega-lite plot showing the average λ_2 values along the individual core lines.

Performance Assessment

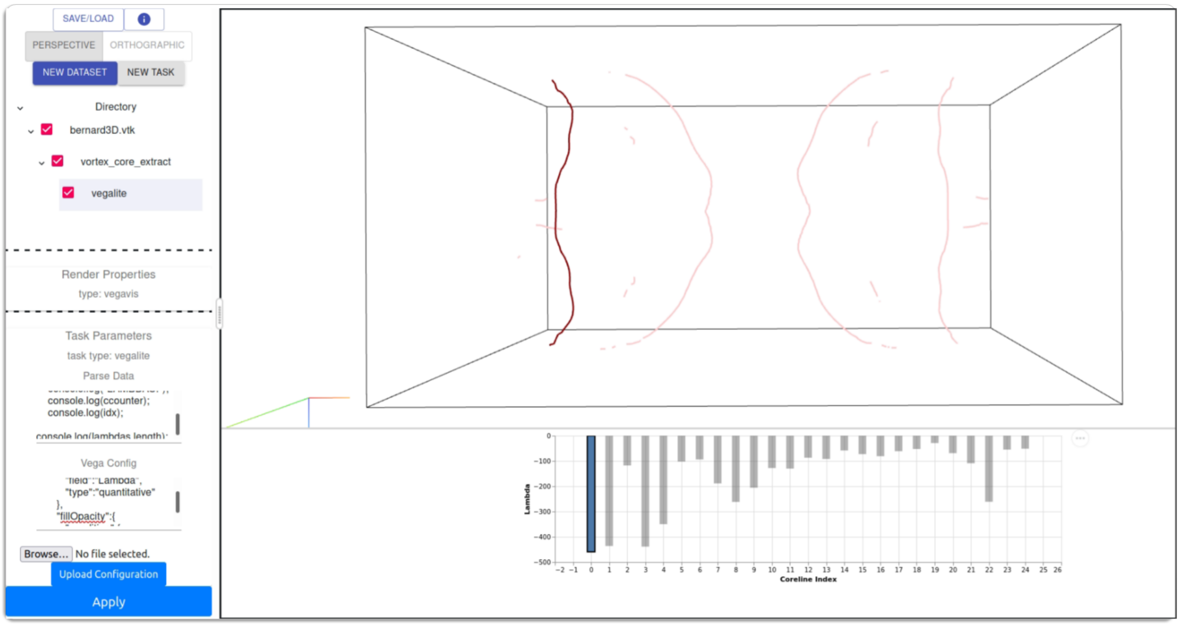
In this section, we provide an assessment of the performance of FCLWebVis.

The Web Application's performance on a normal consumer laptop is considerably efficient. The only CPU and GPU intensive operations are from the first rendering of an extensive visualization task such as streamline tracing (with hundreds of streamlines), or when the view is changed (panning or zooming) which requires re-rendering of all tasks. After a new task is rendered, additional changes to the task will only change the specific part of the task's visualization component and do not require re-rendering the view. Since the web application only keeps the minimal amount of data needed for visualization, the memory usage is negligible compared to the actual amount of memory needed to perform tasks. An average user session that makes use of multiple data sets and multiple visualization tasks do not suffer from performance issues.

The Processing Server's performance depends mainly on the operation cost of the task and the number of consecutive users. The server can manage consecutive connections, such as from multiple users uploading data at the same time, or when the resulting data is sent to multiple users at the same time. However, if the task requires the execution of non-JavaScript code (e.g., C++ or Python), the server can only execute the task concurrently due to the limitation of the Node Gyp framework. A bottleneck may happen if a user submits a computationally expensive task in which case other users will have to wait until the server finishes processing the task.

Figure 9 illustrates the processing server's response to multiple submitted client requests. Each request is timed in 4-second intervals to ensure a clear ordering (and confirm that the results will be first-in-first-out). Every request is identical and involves the upload of the Bernard 3D vector field (8MB) [16] to the processing server. The server will then parse the VTK file into the correct format in memory (e.g., VTKStructuredGrid). After loading the data set, the server will send a success message back to the client along with the basic information of the data (e.g., data range and spacing) for visualizing the bounds of the vector field. However, to test the client's ability to receive large data results, in this experiment the server will send the fully parsed vector field back to the client. Figure 9 demonstrates the server's ability to receive and send data to multiple clients at the same time, with the only major bottleneck is that the server must process each task one at a time (if the task requires executing a non-JavaScript language, in this case, the vector field is stored in C++ VTKStructuredGrid format). If the server cannot receive and send data simultaneously, then there would be a long and consistent delay for each successive client request, but in this case, each request is completed at similar times.

Since the only major bottleneck is the task execution and not on networking (e.g., uploading the input data and sending result



```
console.log(ccounter);
console.log(idx);
```

```
"type":"quantitative"
},
"fillOpacity":{
```

Figure 8. *Demonstration of the vortex core extraction task combined with an interactive Vega-Lite plot that shows the average λ_2 values of each core line.*

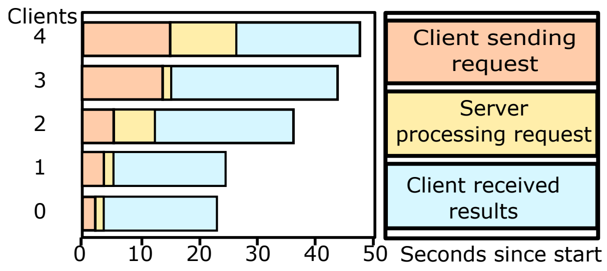


Figure 9. *Response times of the processing server to five simultaneous client requests.*

data), we can create a rough estimate of the average user wait time (AUWT) with $AUWT = (U * T_p + T_{io})/PS$ where U is the expected number of active users, T_p is the average task processing time, T_{io} is the average networking time, and PS is the number of processing servers. If the AUWT is too long, then more processing servers should be added.

Multi-threading and Parallel Computation. Unfortunately, due to FCLWebVis's design goal of allowing intuitive and easy implementation of new tasks and cross-language scripting, the resulting high degree of freedom in the process of implementing new tasks also makes the automatic utilization of multi-threading and parallel computation very challenging. To clarify, enabling multi-threading or parallel computation on a specific task can be done by importing the required libraries and methods for such features (such as OMP for C++). We have done basic testing on simple multi-threaded tasks (e.g., streamline tracing) and obtained the expected results. The challenge here is to enable multi-threading or parallel computation out of the box without requiring the developer to incorporate them into individual tasks. ParaView does allow much easier incorporation of parallel computation and multi-threading into their filters because they mainly rely on one framework (VTK) for data representation and algorithms. We hope to achieve similar capabilities in the future, but for the moment we prioritize the ease of use and implementation of new visualization tasks in exchange for sub-optimal capabilities.

Conclusion

In this paper, we present a new web-based scientific data processing and visualization framework, called FCLWebVis. Our framework is built on the modern client-server architecture and supports a flexible workflow and user-customized new tasks. We have demonstrated the use and flexibility of our framework via a few 2D/3D data processing and visualization tasks.

It is worth mentioning that one of FCLWebVis's major goals is to enable community support of the operation cost for running visualization web services, which should encourage researchers to make their methods easily accessible by contributing to a community-supported FCLWebVis server.

Limitations and future work. While FCLWebVis is an ambitious web-based visualization framework with a high degree of flexibility and accessibility, there are still some limitations that need to be addressed. Firstly, the current implementation only has a limited number of working visualization tasks (e.g., streamline/stream surface construction and visualization, volume rendering, and vortex core extraction). More tasks for vector fields (e.g., vector field reconstruction, streamline clustering [13], and unsteady flow exploration [9]) and other data types can be added, which we are working on. Secondly, FCLWebVis can only display a maximum of two visualization panels split horizontally and do not support the dynamic creation of visualization panels yet. Thirdly, due to the current setup of the file upload process, if the user tries to upload a data set that is larger than the memory limit of the user's browser, the web application will freeze. Also, unhandled exceptions may happen during the visualization process that can terminate the web application. We plan to address these limitations in future work. In addition, we wish to add more intuitive multi-threading and parallel computation implementation to the web service to remove the need for researchers to write their own multi-threading or parallel computation methods into their tasks. Finally, while FCLWebVis does satisfy all of the listed requirements, the framework should be considered more as a working proof-of-concept prototype, as these features need more time to mature and stabilize to be ready for real-world usage. We hope that this paper can demonstrate the usefulness and potential of FCLWebVis and attract enough attention to motivate future developers to help this aspiring open-source framework reach fruition.

As a final note, our main goal is to raise awareness of the potential limitless applications of what an online, easily accessible, simple to contribute (i.e., adding new visualization methods), straightforward to setup and host, and cross-language capable web service can do to promote the use of cutting-edge visualization techniques to the general public. Hence we also welcome the development of new frameworks (or the addition of these features to existing frameworks, such as Paraview) with cross-language, online web service, and straightforward task API.

Acknowledgment. We wish to thank the anonymous reviews for their valuable feedback. This research was supported by NSF IIS 1553329.

References

[1] James Ahrens, Berk Geveci, and Charles Law. Paraview: An end-user tool for large data visualization. *The visualization handbook*, 717(8), 2005.

[2] Alexander Bock, Emil Axelsson, Jonathas Costa, Gene Payne, Micah Acinapura, Vivian Trakinski, Carter Emmart, Cláudio Silva, Charles Hansen, and Anders Ynnerman. Openspace: A system for astrographics. *IEEE transactions on visualization and computer graphics*, 26(1):633–642, 2019.

[3] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *IEEE transactions on visualization and computer graphics*, 17(12):2301–2309, 2011.

[4] Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, David Pugmire, Kathleen Biagas, Mark Miller, Cyrus Harrison, Gunther H Weber, et al. Visit: An end-user tool for visualizing and analyzing very large data. 2012.

[5] Matt Edmunds, Robert S Laramee, Rami Malki, Ian Masters, TN Croft, Guoning Chen, and Eugene Zhang. Automatic stream surface seeding: A feature centered approach. In *Computer Graphics Forum*, volume 31, pages 1095–1104. Wiley Online Library, 2012.

[6] Jingwei Huang, Yichao Zhou, Matthias Niessner, Jonathan Richard Shewchuk, and Leonidas J Guibas. Quadriflow: A scalable and robust method for quadrangulation. In *Computer Graphics Forum*, volume 37, pages 147–160. Wiley Online Library, 2018.

[7] Sebastien Jourdain, Utkarsh Ayachit, and Berk Geveci. Paraviewweb, a web framework for 3d visualization and data processing. *International Journal of Computer Information Systems and Industrial Management Applications*, 3(1):870–877, 2011.

[8] Li Liu, Deborah Silver, and Karen Bemis. Visualizing three-dimensional ocean eddies in web browsers. *IEEE access*, 7:44734–44747, 2019.

[9] Duong B Nguyen, Lei Zhang, Robert S Laramee, David Thompson, Rodolfo Ostilla Monico, and Guoning Chen. Physics-based pathline clustering and exploration. *Computer Graphics Forum*, 40(1):22–37, 2021.

[10] Prabhu Ramachandran and Gaël Varoquaux. Mayavi: 3d visualization of scientific data. *Computing in Science & Engineering*, 13(2):40–51, 2011.

[11] Martin Roth and Ronald Peikert. A higher-order method for finding vortex core lines. In *Proceedings Visualization'98 (Cat. No. 98CB36276)*, pages 143–150. IEEE, 1998.

[12] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics*, 23(1):341–350, 2016.

[13] Lieyu Shi, Robert S Laramee, and Guoning Chen. Integral curve clustering and simplification for flow visualization: A comparative evaluation. *IEEE transactions on visualization and computer graphics*, 27(3):1967 – 1985, 2021.

[14] three.js. three.js / editor, 2015.

[15] W. von Funck, T. Weinkauf, H. Theisel, and H.-P. Seidel. Smoke surfaces: An interactive flow visualization technique inspired by real-world flow experiments. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization 2008)*, 14(6):1396–1403, November - December 2008.

[16] Daniel Weiskopf, Tobias Schafhitzel, and Thomas Ertl. Real-time advection and volumetric illumination for the visualization of 3d unsteady flow. In *EuroVis*, pages 13–20, 2005.

[17] Kanit Wongsuphasawat, Zening Qu, Dominik Moritz, Riley Chang, Felix Ouk, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. Voyager 2: Augmenting visual analysis with partial view specifications. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pages 2648–2659, 2017.

[18] Lei Zhang, Duong Nguyen, David Thompson, Robert Laramee, and Guoning Chen. Enhanced vector field visualization via lagrangian accumulation. *Computers & Graphics*, 70:224–234, 2018.

Author Biography

Nguyen Phan received his B.S in Computer Science at the University of Houston-Downtown (2018) and is currently pursuing a Ph.D. degree at the University of Houston. His research interest is in data visualization with specialty in vector field (or flow) data visualization.

Guoning Chen is an Associate Professor at the Department of Computer Science at the University of Houston. He earned his Ph.D. in Computer Science from Oregon State University in 2009. Before joining the University of Houston, he was a post-doctoral research fellow at the Scientific Computing and Imaging (SCI) Institute at the University of Utah. His research interests are in Data Visualization, Geometric Modeling, Geometry Processing, and Physically-based Simulations.