viz3: Live 3D Visualizations of Data Acquisition with Changing Boundaries

Primary Author
Center for High Performance Computing
University of Utah
Salt Lake City, Utah
dylan.gardner@utah.edu

***Abstract*—In the monitoring and viewing of live operational data from HPC compute clusters, 2D visualizations have traditionally been used to give analytical insights into a small number of sensor and metric trends. Yet, the 2D space offered by these visualizations is often not sufficient, without lossy aggregation, to visualize complete computer systems containing a multitude of sensors and sub-components that are highly dimensional or have a physical representation. On the contrary, 3D visualizations can provide exploration and new insights by viewing the relationships between all of those metrics. However, practical challenges to producing such visualizations exist, namely the dynamic nature of sensor data, the integration of data silos containing differing data dimensions, and the enabling of system administrators to produce custom visualizations. This paper introduces a software framework that addresses those challenges with a composable scheme for creating abstract 3D scenes from live operational data integrated from numerous data sources.**

***Index Terms*—3D Visualization, Data Center Visualization, HPC Monitoring, Health Metrics, Redfish**

I. Introduction

Today's HPC operation centers use 2D analytical visualization tools, such as Grafana [1], Nagios/Icinga [2], Cacti [3], and the Elastic Stack [4], to easily create analytical graphs, charts, and other visualizations of their infrastructure monitoring and health data from a wealth of data sources. These 2D visualizations give detailed insight into the operations of complex systems—showing the trends and patterns of a handful of system metrics at a time. They have served system administrators well. One fundamental limitation of these 2D visualizations however, is the lack of information density within the 2D space. This lack of information density prevents the potential for visualizations that show metrics with a large number of instances. The lack of density also prevents the ability to combine different data metrics together from separate components to provide additional insight. In other words, 2D visualizations alone cannot always visualize a complete system such as the data center thermals, PDU power usage, node thermals, GPU thermals, and the health of an entire compute cluster at once. This system-level visibility may be useful, for example, to discover abnormalities in temperature and heat distribution within a data center and relate it to node health, load and power.

3D visualizations may help fill this gap. Yet, 3D visualizations of operational systems are uncommon in practice and are largely created by hand, such as those found in [5] [6] [7]. For 3D visualizations, there is a dearth of ready made tools for administrators to produce such visualizations. Although there has been some research in producing 3D information visualizations for HPC systems in recent years [7] [8], especially with regards to node health, such tools focus on particular data sources and are inflexible to further adaption to different metrics or a differing data environment.

One of the challenges presented by operational data, particularly when creating 3D visualizations, is that it is often highly dimensional and the totality of those dimensions is rarely kept within a singular data source—metrics are stored in time-series databases such as Prometheus [9], Performance Co-Pilot (PCP) [10] and InfluxDB [11] with minimal dimensionality to differentiate instances. Administrators often store higher, more static, dimensions in relational databases. Creating 3D visualizations of many components and metrics requires the integration of many of these dimensions.

Another challenge of this data is that the number of measured items for some dimensions changes regularly. That is, as sensors and machines go offline, online or are repurposed, the boundaries of the data changes, requiring special care to ensure the layout of visualizations using this data are reactive to the introduction and removal of devices, as well as new groupings of the same data.

This paper first covers related visualization work, before describing a system for utilizing the various data dimensions of metrics, potentially stored across multiple different data sources, to declaratively produce live 3D visualizations of systems; and, show how the work is reactive to the changing and differing boundaries of operational data, while also being relatively easy to reason about and create. This system is implemented in a C++/Python framework called *viz3*. This paper then shows and describes two example visualizations produced using the software framework. Finally, the paper finishes by discussing the limitations of this work.

II. Related Work

There are numerous existing tools that provide 2D analytical visualizations of operational data. Grafana [1] is a popular tool that allows users to create dashboards containing numerous charts, tables, and gauges, with each visualization able to source data from a different singular data source. Yet, the builtin visualizations are largely two dimensional and cannot

combine data and metrics from different data sources into a single visualization to show multiple components.

In the academic world however, there has been some recent work visualizing operational data, particularly node health and performance data from HPC centers. In [7], Nguyen et al. describes a VR visualization architecture that utilizes BMC sensor information from the Redfish standard [12], as well as the Nagios monitoring system, to produce several new 3D interactive analytical visualizations that show the health and performance data of a cluster in a HPC data center. These visualizations are multivariate and interactive, allowing the user to see either power, fan speed, *or* temperature information of an entire cluster at a glance, while also being able to deep-dive into other system metrics for a particular machine by selecting it. Yet, such visualizations are not sufficient for at-a-glance status information for multiple metrics and components at once without clicking on meshes. The visualization is also not easy to create or change for a system administrator—specifically when metadata such as machine locations is stored in a separate data source than the node metrics. Similarly, in both [8] and [13], Nguyen et al. also introduced several new visualizations that combined HPC job data with compute node metrics from Redfish. These visualizations are an important step towards combining and visualizing the many components of an HPC compute cluster together, but again do not permit the at-a-glance presentation of information of more than two components at the same time with a large number of instances. They are also not general tools that can be applied to different metrics, focused specifically on job data.

Some general software frameworks aimed at easing the production of 3D visualizations have been proposed in the past. In [14], the authors describe an extension to SQL called SuperSQL that allows for a programmer to write queries that return 3D visualizations built from SQL query results. These 3D visualizations are formed by mapping the returned data columns to 3D models and mapping grouping to basic 3D layouts such as juxtaposition (putting models next to each other). Put together, with SuperSQL users can define 3D scenes and layouts of SQL-based datasets without requiring 3D programming. Yet, the inherent reliance on SQL, the static nature of the visualization, and the lack of more advanced layout algorithms found in SuperSQL makes the tool impractical for visualizing operational data.

III. Framework

The system for producing live 3D visualizations from many data sources is made up up of three different pieces, presented as follows:

- A data model and querying system that abstracts over multiple different data sources and databases to present a unified graph model of data is first described.
- Next, a declarative, composable, and reactive 3D layout and mesh tree that enables the combination of multiple sub-visualizations together, while being easy to create and reason about is shown.
- Finally, a templating system that maps 3D layout elements and mesh attributes (e.g. color, height, opacity) in the layout tree to the results of data queries in the data model is introduced.

A. Data Model

Being able to visualize and group multiple metrics and instances across different data sources, is key to creating a complete 3D visualization of a system, as single data sources often lack a total view of a system and attempting to create a unified data source is often impractical or results in data duplication.

The cross-data source querying problem is solved by creating a unified data model and querying engine that presents data stored from multiple data sources to the user and allows queries on the model to cross the underlying databases where the user defines them to be interoperable. Encoded in this model are the metrics and the shared dimensions between different metrics. To produce such a data model, a YAML configuration file must be given that describes, in terms of the underlying data model of a particular database, the metrics available within each data source, their dimensions, how those dimensions relate to each other, and how they relate to other dimensions from different data sources. Internally, each data source module within the engine interoperates between queries on the data model and the underlying databases. The data source specific encoding of the configuration ensures additional metadata about a database can be utilized to most efficiently query the configured database from a given unified data model query. A number of database modules, such as for Prometheus [9], InfluxDB [11], SQLite [15], and Performance Co-Pilot [10], are provided by the viz3 framework.

1) Representation: The model used to represent data is similar to the classical network data model [16], the precursor to the entity relational model used in modern SQL databases [17]. In this model, a directed acyclic graph (DAG) is used where the nodes in the graph represent data series, and the connections between nodes represent either one-to-many or one-to-one relationships between the series. A node with multiple parents also represents a many-to-many relationship. Logically, one could say that the incoming connections into a node constrains or derives a subset of the connected node's data series, acting as filter on the data series of the connected node. Certain metric or data dimension relationships are also encoded with certain edge patterns in the data graph. Metrics themselves, for example, are the leaves of the graph. Dimensions that are aliases of each other (here, data series with a one-to-one correspondence, or bijection, to another data series) can be encoded by having the aliased nodes have the same incoming and outgoing edges. This is seen in Figure 1 with an aliased kernel-given network interface name and administrator-defined name. Finally, category dimensions (here, data series with a one-to-many relationship to another data series), such as compute clusters, which have a one-to-many relationship to hostnames, can be directly encoded with an edge in the graph.

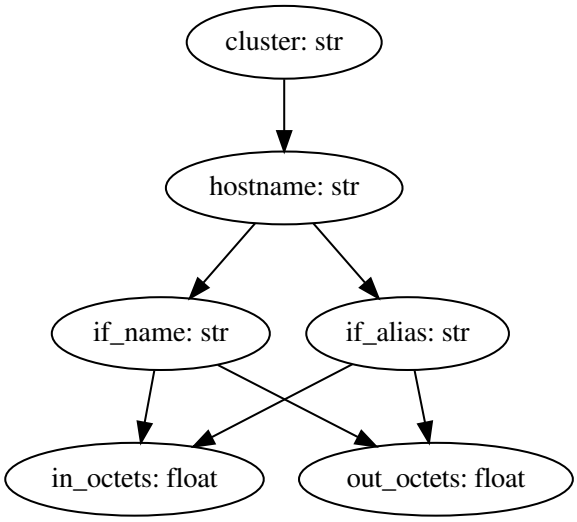

Fig. 1. Graph containing a network interface alias

The network model, rather than the relational model, was chosen for a number of reasons. First, this model was chosen to make it easier to encode the natural lack of explicit relations in NoSQL databases found in common time-series databases such as Prometheus and Performance Co-Pilot. Second, the graph structure of the network model maps naturally to the tree structure of the layout system described later. Third, this model was thought to be much easier to implement than a relational one.

2) Querying: A query against the data model is simply expressed as a path through the network graph, encoded as the node names joined onto the dot character: `machine.cpu.temperature`. At each node in the query path, the corresponding data series is filtered based on each instance of the previous node's data series. With a `machine.cpu.temperature` query for example, the query engine would retrieve all machines, then for each machine, would query the CPU node for all instances under each particular machine before querying the temperature of each particular CPU for a particular machine. The final data series and query result is simply all of the final instances from the last node's data series derived through this combinatorial process. Additionally, each selected node in the query path may have a filter set or regular expression match associated, which is ran on each resulting instance of the data series of a node to restrict the data series of the following nodes.

For convenience, a query path can also omit intermediate nodes if the path between two nodes is unambiguous. This does not change the query result and internally the intermediate nodes are added. Omissions may be beneficial for readability when using cross data source queries described below.

3) Cross-data source queries: To enable cross-data source queries, that is, using data from another data source to filter the data series in a different data source, the user must define in their configuration files what nodes across data sources are interoperable and in what direction the interoperability applies. For each interoperable pair of nodes, an edge in the corresponding direction between the different data source nodes is added to the graph. Thus, querying across data sources is done by expressing a query path through two different data source nodes. The joining between the nodes is then done locally on the client, rather than using the database. When the different data source nodes are variations of the same name, the omission of unambiguous intermediate nodes can be used for readability.

Interoperability here is defined as the values of one data series being interchangeable with another data series. This strict definition often conflicts with real data sources. For example, a hostname in one data source may use a fully qualified domain name to describe a host, whereas another may use a shorthand hostname or even use a MAC address instead. One simple technique to get around this lack of interoperability is to define a data source that encodes a dictionary mapping between the different host identifiers. e.g. using a local SQLite3 database, or the hardcoded configuration data source. More advanced techniques for remapping could be developed, for example, by allowing the user to define a regex expression or a piece of code that does this translation, but as-is the viz3 framework does not have this functionality.

4) Querying optimizations: As noted before, a naive implementation of a data source module, which queries the database based on each result of the previous data series, would result in a combinatorial number of queries to the underlying data source. This can become a performance bottleneck. One successful way to skirt around this problem, implemented in the database modules found in viz3, is to use the existing knowledge about the relationships between nodes of the same data source, found in the database-specific configuration of metrics, to do bulk queries and cache results. Not all queries from the same data source can be done in a bulk query however. If the relationship between nodes in the graph does not reflect a relationship in the data source's underlying data model, a singular bulk query cannot be issued in many cases. However, with such optimizations, for most queries the theoretical combinatorial number of requests can be eliminated.

5) Generation of configuration files: Having to define and configure every possible metric from a data source that one may want to use in a visualization, along with the relationships and dimensionality of each metric, could be a repetitive task. Each database however, does encode some information about data relationships in it's data model, for example in foreign and primary keys in a SQL database, or the set of key-value pair labels associated with a metric in Prometheus [18]. These encodings can be leveraged to help automatically define the relationships and dimensionality of metrics. The viz3 framework has scripts for many databases that can generate configuration files.

That being said, the configuration files generated by these scripts are often not complete and nearly always present inaccuracies in the relationships since a) there is nearly always a

mismatch between the database data model and the viz3 graph data model and b) the relationships encoded in data source are often either not complete, or inaccurate. In the former case, heuristics must be used, leading to further inaccuracies. This inaccuracy issue is particularly an issue for many external production SQL databases that do not use foreign keys and for NoSQL databases such as Prometheus, where for example, labels are freely used to encode additional non-dimensional textual metrics (since only floating point metrics are allowed in the Prometheus data model [18]), making automatically detecting the relationships of labels at best an educated and often incorrect guess.

B. Layout Engine

The increased information density of 3D visualizations enable visualization designers to combine many different metrics and components together in a way that 2D visualizations do not always enable. Manual combination and composition however is often fragile, as changing or adding additional meshes can cause global positioning logic to break.

An effective composable visualization system is one where one can reason both locally about how a particular sub-layout will be positioned, while also reasoning globally about how multiple models are positioned together [19]. The layout system described here achieves this by creating a bottom-up scheme for producing and positioning 3D meshes, built on the notion that a final visualization can be decomposed into a nested tree hierarchy of positioning algorithms and 3D meshes.

In this tree hierarchy, each node contains an "element" which provides a positioning or geometric manipulation algorithm for children nodes and optionally a mesh generation function. These elements are reusable and are typically focused on a particular positioning, geometric or mesh algorithm. To produce a visualization, elements are called upon in a bottom-up depth first search fashion, with elements optionally producing a 3D mesh, as well as positioning and manipulating the meshes and position of children elements. The position of children computed by a parent element is typically based on their children element's total self-reported bounds. From the perspective of each element, the positions calculated are absolute, yet they are likely to be moved by their parent element. This absolute positioning scheme makes it easy to reason about sub-layouts of the tree. Naturally, in this system, the leaf elements of the tree are intended to be 3D meshes representing metrics, with the ancestor elements positioning either the descendant meshes or sub-layouts.

In addition, each element has key-value attributes that allow the user to customize the positioning of children and/or 3D mesh produced. For 3D mesh elements such a box shape, these attributes are used to modify the physical attributes of the resulting mesh. e.g. width, height, depth, color, opacity, etc. For layout/positioning elements, these attributes are used as parameters to modify the layout algorithm. e.g. spacing, alignment, axis, etc.

A table containing a number of common elements found in the viz3 framework are shown in Table I, along with a

```
<juxtapose axis="x" align="center">
  <plane padding="0.15" height="0.1" color="gray8">
    <grid spacing="0.25">
      <box width="1" height="0.25" depth="1" color="gray3"/>
      ... 3 more times
    </grid>
  </plane>
  <box height="0.1" width="0.75" depth="1" color="gray4"/>
  <plane padding="0.15" height="0.1" color="gray8">
    ... same as plane above
  </plane>
</juxtapose>
```

Fig. 2. An abstract 3D visualization of dual CPU sockets

```
<juxtapose axis="x" align="center">
  <plane padding="0.15" height="0.1" color="gray8">
    <grid spacing="0.25" width="2.25" depth="2.25">
      <box width="equal%" height="0.25" depth="equal%" .../>
      ...times 8
    </grid>
  </plane>
  <box height="0.1" width="0.75" depth="1" color="gray4"/>
  <plane padding="0.15" height="0.1" color="gray8">
    ... same as plane above
  </plane>
</juxtapose>
```

Fig. 3. An abstract 3D visualization of a dual CPU sockets, with relative attributes used

description of what the element does, the attributes that it has, and whether it provides a mesh. These elements are sufficient to create a wide range of abstract looking visualizations. A C++ programming interface for adding additional elements within the viz3 framework, is also available.

1) Composition: The composability of this system comes from the fact that each element serves a single purpose: it lays out its children in a particular way and/or provides a 3D mesh. This makes reasoning about the resulting layout easy since one can reason on each level of the tree hierarchy about how the layout will appear. One can also move a sub-tree anywhere within the tree, or develop it separately and insert it anywhere. Furthermore, one can easily encode this hierarchy in e.g. an XML file, making it possible to create static visualizations with just a text editor, similar to HTML.

An example of this composition can be seen in Figure 2. The resulting visualization is one of a dual CPU sockets, with cores correspondingly split into the two sockets. To produce the visualization, the definition, encoded in XML, leverages a `box` element to represent a single core, a `grid` layout element to position each core, then a special `plane` element to add a 3D plane box under the grid. The two CPU sockets are finally juxtaposed next to each other on the x axis with an intermediate box representing the memory interconnect between the two sockets. Notably, if one were to change the number of cores on each CPU socket, the layout and visualization would automatically adjust to fit the new cores. The count-agnostic nature of these elements is what is leveraged when the data model is combined with this layout system in the templating system to produce reactive 3D visualizations with relative ease.

2) Top-down constraints: As-is the bottom-up approach taken by the layout system is easy to reason about, but can suffer from too much local logic when attempting to integrate and change such a visualization within a deeper and wider hierarchy. For example, the total size of the visualization in Figure 2 is determined largely by the number of CPU cores.

Name	Description	Attributes	Provides a mesh?
box	Box mesh	width, height, depth, color, opacity	yes
cylinder	Cylinder mesh	height, radius, color, opacity	yes
nolayout	Stores attributes without positioning children; used for grouping and defining fixed top-down attributes	width, height, depth	no
juxtapose	Juxtaposes (positions side-by-side) children along an axis	spacing, axis, align, width, height, depth	no
grid	Places children in a grid pattern, sized to the largest child	spacing, axis	no
plane	Draws a flat plane (box) under children	padding, height, color, opacity	yes
padding	Creates an empty mesh of the specified size, limiting the bounds of children	padding, width, height, depth	no
street	Positions and rotates children in a fashion similar to houses on a street, while providing a "road" mesh down the middle	axis, spacing, width, height, depth, color , opacity	yes
scale	Scales children up or down to the greatest specified width, height and depth extent	width, height, depth	no
rotate	Rotates children (yaw, pitch, roll)	degrees, pitch, roll, yaw	no
include	Includes a sub-hierarchy from another file	path	no

TABLE I
A DESCRIPTION OF SOME COMMON ELEMENTS PROVIDED BY THE LAYOUT ENGINE

```
<plane padding="0.15" height="0.1" color="gray8">
  <grid spacing="0.25" width="2.25" depth="2.25">
    <box bind=".cpu" width="equal%" height="0.25"
         depth="equal%" .../>
  </grid>
</plane>
```

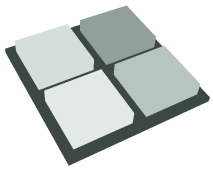

Fig. 4. An abstract 3D visualization of a CPU, with bindings used

If this increases, the visualization can become unreasonably large, and when integrated into a larger visualization, such as a fixed size motherboard, this requires scaling down the sub-visualization (a `scale` element was developed for this purpose), which can differ from the desired layout.

To alleviate this problem, the layout system contains a top-down constraint system that allows each attribute in a particular element to be specified relative to another attribute value higher up in the hierarchy. More specifically, an relative attribute is relative to the closest ancestor attribute value in the hierarchy. The closest ancestor attribute value itself may also be relative so long as there is a fixed ancestor attribute value that is eventually referred to in the hierarchy above the element.

This top-down capability can be applied to the composition in Figure 2, as seen in Figure 3. In Figure 3, each grid was specified in terms of an absolute width and depth. The width and depth of each CPU core box are then modified to be an equal percentage of the total space of the grid (as defined by each element), resulting in a visualization that has a fixed size, regardless of how many CPU cores there are.

C. Mapping Data to Layouts

With a unified data model describing the relationships between metrics and a declarative and composable 3D layout engine, a data-driven 3D visualization system can be established. This is accomplished by allowing the user to "template" layout elements in a layout tree by associating a data query with an element. This template, or rather "binding", includes descendant elements. When bindings are evaluated, they result in the associated data queries being ran against the unified data model. For each instance in the data series returned, the templated element (as well as descendants) is duplicated. For example, a `box` element with a binding of `.machine` would result in a `box` being produced for every known machine. If the `box` or any templated element contained sub-elements, those sub-elements would also be duplicated for every `box`. If a binding was nested below another binding, it would be evaluated for every instance produced by the ancestor binding. Once the templates have been "exploded", the layout engine evaluates the new layout hierarchy and the visualization shown to the user is updated.

The evaluation of bindings is done periodically according to a rate set by the user. When evaluated, the templated layout tree is re-evaluated from scratch, and logically the previous exploded state is not considered—though in practice the implementation reuses and updates elements that are shared between the two states for efficiency purposes. Thus, if a bound data query returned less data series than when previously ran, then those missing corresponding layout elements are removed from the visualization. Similarly, if new instances are added from a binding, new layout elements are added. In both cases when the layout engine is ran, ancestor layout elements that position children will automatically recalculate the positions of their changed number of children, resulting in the visualization adapting to new data.

A CPU socket example similar to Figure 2, found in Figure 4, can demonstrate this idea. Here we are visualizing a singular CPU socket. Similar to dual CPU socket visualization, we have a CPU `box` within a `grid` layout element, within a `plane` element. Yet here, we have a single templated CPU `box` element that is bound to a query that will return a list

```
<juxtapose axis="x" align="center">
  <plane bind=".numa" filter="='1'" padding="0.15"
         height="0.1" color="gray8">
    <grid spacing="0.25" width="2.25" depth="2.25">
       <box bind=".cpu" width="equal%" height="0.25"
            depth="equal%" .../>
    </grid>
  </plane>
  <box height="0.1" width="0.75" depth="1" color="gray4"/>
  <plane bind=".numa" filter="='2'" padding="0.15"
         height="0.1" color="gray8">
    ... same as plane above
  </plane>
</juxtapose>
```

Fig. 5. An abstract 3D visualization of a dual CPU sockets, with bindings used

of CPU numbers, rather than having N `box` elements. When the visualization is ran however, the data query will run and the box template will be duplicated for every CPU, resulting in a similar visualization as found in Figure 2. Thanks to the count-agnostic nature of elements, the visualization declaration is adaptable to different machines with different core counts, while also making it easy to modify the layout by swapping the `grid` element for another layout element.

1) Relative Bindings: If one wanted to recreate the full dual CPU socket visualization described above, one could introduce a binding on an ancestor element to duplicate the socket box for every CPU socket, yet as described, this would result in the same number of overall CPU cores being present for *each* socket, rather than there being a half of the overall number of CPU cores for each socket. To solve this problem, descendant queries can be constrained by the results of ancestor queries as if the ancestor query path was prepended to the descendant query with a filter for each instance returned by the ancestor query path. A relative path is expressed with a leading dot. When looking at the bindings, as seen in Figure 5, one can think of the leading dot logically as a "for-each result within the ancestor query" operator.

Additionally, to facilitate the restriction of the data instances returned by a particular data query, each binding can have a filter: a set of instances returned by the data query that each instance in the returned series must match against.

Finally, the attributes of elements can also be bound to a value produced by an attribute data query, or attribute binding. That is, the value returned by the attribute data query (relative to the element binding) can be assigned to an attribute value (such as width, height, color, opacity, etc). It is unlikely that the value can be directly used, particularly when the attribute value is of a different type (e.g. color), so the viz3 framework allows a user to either use a series of predefined transformation functions to manipulate the value(s), or the framework permits the user to define their own transformations in Python. Naturally, this either requires a one-to-one relationship between the element data query and the attribute data query. Alternatively, a transformation can aggregate multiple values or a null filter with a fallback value can be defined—particularly useful when dealing with non-uniform data. For example, a `box` element bound to a `.socket` query could use a attribute binding for `.socket_temp_sensor` to change the color, size or opacity of that box based on the temperature of each socket `box` visualized. If a sensor does not exist however, but a null filter is used, then a fallback value such as a gray color can indicate the sensor is missing, rather than having no `box`. This feature is utilized in the example visualizations section below to change the opacity and color of meshes based on a percentage value.

D. Client-Server model

The layout engine in the viz3 framework is a standalone component that is designed to act as a server to multiple clients that can show the resulting visualization using different rendering technologies. This client-server model and loose coupling with 3D rendering engines (such as game engines) provides the ability to have visualizations on both the web and desktop. Furthermore, because the template explosion and layout process is done server-side, the computational burden on slower clients, often found the web, is reduced. This is in addition to the beneficial side effect of having the data that produced the visualization, which may be sensitive, being hidden to clients, a problem that existing visualization tools such as Grafana suffer from [20].

To enable this decoupling, clients are implemented as a simple machines that receive a stream of operations from the server to either render new meshes (found in the streaming data), remove meshes, position individual or groups of meshes, or change mesh attributes such as color, opacity and size. When utilizing an rendering engine, such as three.js [21] on the web, or Panda3D [22] on the desktop, this machine is simple to implement (around 200 lines of Python or JavaScript) and relatively performant. In the web case, the stream is implemented using a server-side event stream (SSE) [23] and for the desktop case the layout engine is simply on another thread and events are exposed directly to the game engine.

IV. Example Visualizations

A. Data Sources

The visualizations below were produced using time-series data combined with relational SQL metadata from the Center for High Performance Computing (CHPC), at the University of Utah. The various data sources used are described below:

1) SNMP: The viz3 system was configured to use CHPC's production Prometheus time-series database storing, among other things, load average, network, power, and data center temperature metrics, obtained largely though SNMP.

2) Nvidia GPUs: An InfluxDB database containing Nvidia GPU performance data, was also configured. This data, ultimately collected from a Performance Co-Pilot plugin via an unrelated data pipeline, but eventually stored in InfluxDB, was obtained in a roundabout way due to the desire to avoid adding additional collection overhead to compute nodes. viz3 however, does support pulling directly from Peformance Co-Pilot.

3) Redfish: The aforementioned Prometheus database also stores newly-collected Redfish [12] data, a new capability to CHPC, who runs several heterogeneous compute clusters where only one currently has wide support for the Redfish

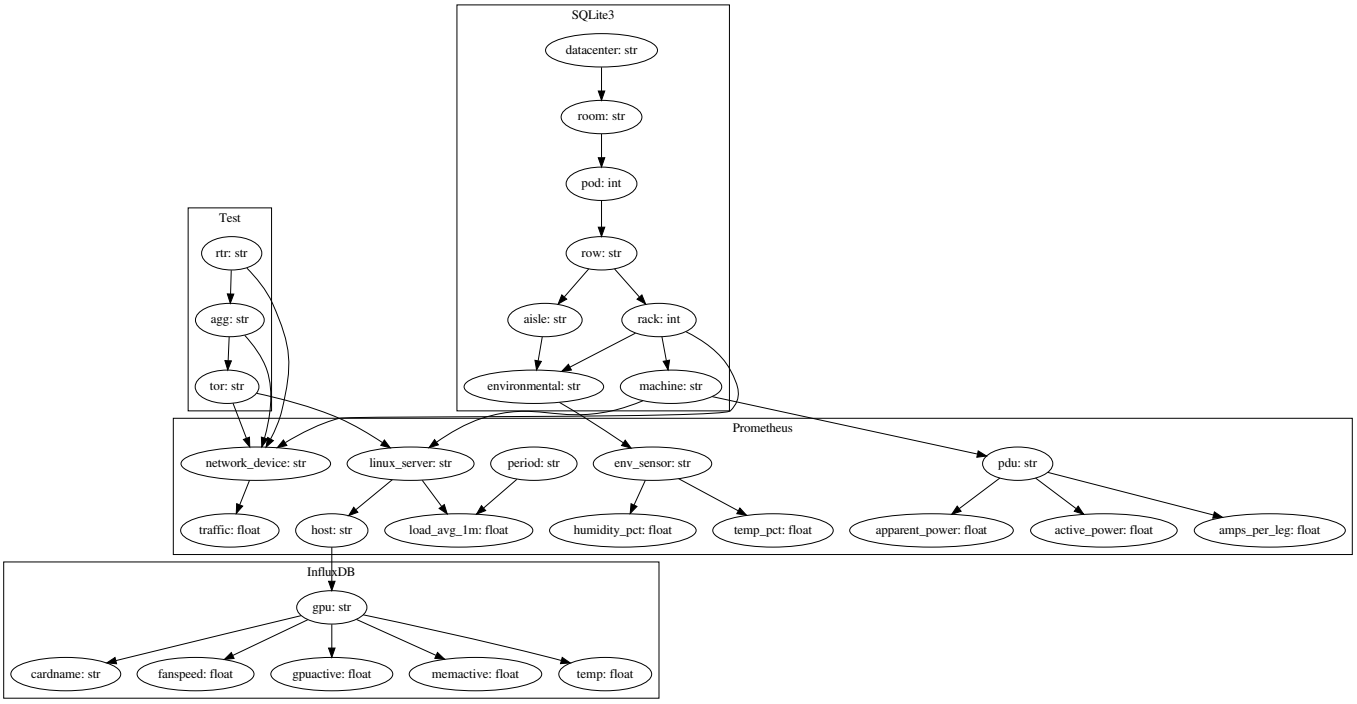

Fig. 6. Subset of the data model graph used to produce visualizations of CHPC data

standard. Data was collected using an open-source redfish-exporter tool [24]. The compute cluster has approximately 300+ nodes, spanning a dozen different hardware models, a challenge for visualizing the nodes in the data center visualization, since some motherboards have better and newer support for the Redfish standard than others. A couple dozen nodes have no support at all. Furthermore, the string locations and types of sensors reported in Redfish by each model is different, presenting a data cleaning challenge. This challenge however, was mostly solved in the viz3 data source configuration file and query filters with copious amounts of regular expressions—though missing semantics, such as missing DIMM channel and slot information in sensor names, still presented problems.

4) Infrastructure Metadata: In addition, a local SQLite3 database containing scraped and hand-tuned metadata about machine manufacturers and models, networks and the location of devices was manually created and used, since physical infrastructure metadata such as locations, as well as the machine manufacturer and model, are not stored in the Prometheus database.

5) Test Network Data: Synthetic data describing a network hierarchy of routers, aggregate switches, and physical rack switches, similar to a simplified physical topology of CHPC's network infrastructure, was created and stored in a YAML file. A test data source built into to the viz3 framework was used to query this information.

6) Configuration of viz3: Both the Prometheus and InfluxDB YAML configuration files were produced by using database relationship extraction scripts to produce a first-pass configuration, before hand tweaking was required to fix and simplify names, as well as introduce new relationships and delete incorrect relationships. The configuration for the SQLite3 infrastructure database was manually created due to the small number of relationships and metadata, and the lack of a corresponding relationship extraction script.

In Figure 6 a subset of the data graph containing CHPC's Prometheus, InfluxDB, SQLite3 and test metrics and metadata that was produced from the configuration files, and used in the following visualizations is shown; Redfish metrics are too numerous to include. The names of the metrics in the graph and visualizations however, have been modified to be clearer/shorter than their real counterparts. Ordinarily required in the viz3 framework, the namespace (e.g. `redfish:` of `redfish:redfish_chassis_health`) of each node in the data graph has also been removed in the examples for better readability.

B. Network Layout

In Figure 7, a network topology showing the connections between switches and hosts is presented where, the branches represent physical links between machines or switches and the boxes represent physical machines. Each branch of the street represents a sub-network. Here, the color of each network branch is mapped to the amount of traffic flowing through the physical link, where a darker color, more purply color, represents more traffic. The color of each box, or host, is also mapped to a load average, where a high load in proportion to the number of CPUs results in a red color and a light load to a light, almost white color. The values of these metrics can be

```
<juxtapose axis="x" spacing="50">
  <street bind="rtr" color=".traffic|to_blue_purple"
          height="5" width="5" spacing="5" axis="z">
    <street bind=".agg" color=".traffic|to_blue_purple"
            height="3" width="5" spacing="15" axis="x">
      <street bind=".tor" color=".traffic|to_blue_purple"
              height="1" width="5" spacing="15" axis="z">
        <box bind=".linux_server" height="30" depth="15"
             width="15" color=".load_avg_1m|to_orange_red"/>
      </street>
    </street>
  </street>
</juxtapose>
```

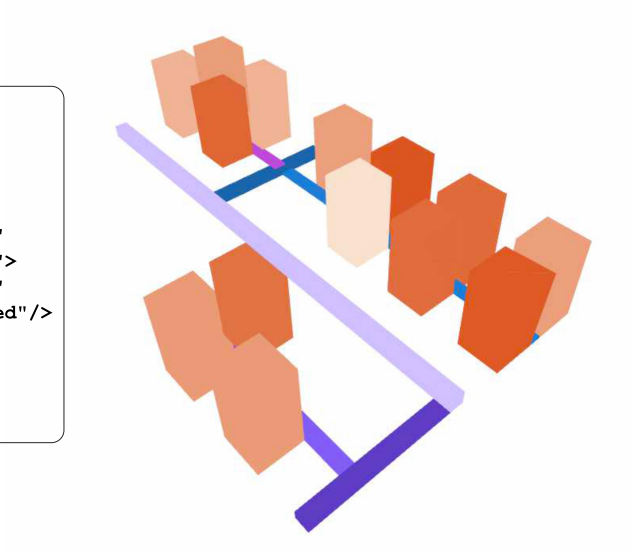

Fig. 7. A physical network visualization

obtained by hovering over the mesh. This data was organized in the layout using the test data source, with traffic information obtained from Prometheus.

To produce the visualization from an XML definition, each `street` element shown in 7 is bound to the corresponding hierarchical level in the network: the first is a router-to-router switch (`rtr` in the data model), the second is a aggregation switch (`agg`), and the last is a top-of-rack switch (`tor`), which nodes in a physical rack each connect to. Lastly, within that innermost top-of-rack `street` element, each machine is represented by a `box` and bound to a particular host (`host`) that matches the "(rtr, agg, tor)" tuple. This simple visualization demonstrates the integration of multiple different data sources and shows a basic correlation between machine load averages and network traffic.

C. Data Center Environmentals

In Figure 8, a thermal visualization of a data center room, combined with PDU rack power, as well as a node and GPU sub-visualization is shown. The non-gray colors of each mesh in the scene are color mapped within a color range, with the min and max color corresponding to an associated min and max sensor or metric value. The PDU bar-box associated with each rack, for example, is color mapped between solid blue to red from the PDU power utilization as a percentage of capacity. Furthermore, the front/back panes on each rack where temperature sensors are present (up to 3, at different heights), is color mapped between light orange to orangish-red from the expected temperature range. Both metrics are obtained from Prometheus, though their positioning in the correct rack is based on SQLite3 location data. Like the previous visualization, the associated values of a mesh can be viewed by hovering over a color mapped mesh. Alone, the visualization of data center thermals combined with power data allows system administrators to identify hot spots in a room and potentially correlate them with racks heavily drawing power.

However, within some racks, nodes from the Redfish-supported compute cluster, along with their GPUs, if present, are visualized. By zooming in on the visualization, an administrator can view these sub-visualizations, as well as see any correlations, such as thermals, between neighboring nodes. Without zooming in however, a system administrator can also see the chassis health of many nodes at once. The bright pink node in the second pod, closest row, at the top of the rack, is a node currently in critical state. Since the plane underneath the "motherboard" sub-visualization turns various colors depending on the health of the system, it is obvious from afar in the visualization that the node is in a bad state. A system administrator can then zoom into the motherboard and view the reported temperature and health data to quickly narrow down the problem. They can also see, at a glance, whether multiple nodes in bad states are located next to each other, indicating that perhaps the issue might be power related, depending on the rack PDU visualization.

The sub-visualization of the node in a bad state is shown in Figure 9. The sub-visualization is an abstract, schemorphic recreation of a 4U chassis, albeit very crude looking, created from Redfish data stored in Prometheus, positioned within the rack with SQLite3 location data. There are three color ranges used here: a blue to bright purple range is used for non-thermal values, the aforementioned orange range for temperature sensors, and green/red for health data. The front opaque wall is mapped to the inlet temperature of the chassis, with a similar wall on the back existing if this particular chassis had an exhaust sensor. Deeper, the fans (typically 4 width-wise by 2; incomplete data) and their speed, as a percentage, are shown, followed by a memory and CPU visualization, with their health and temperature data shown, if a corresponding sensor exists.

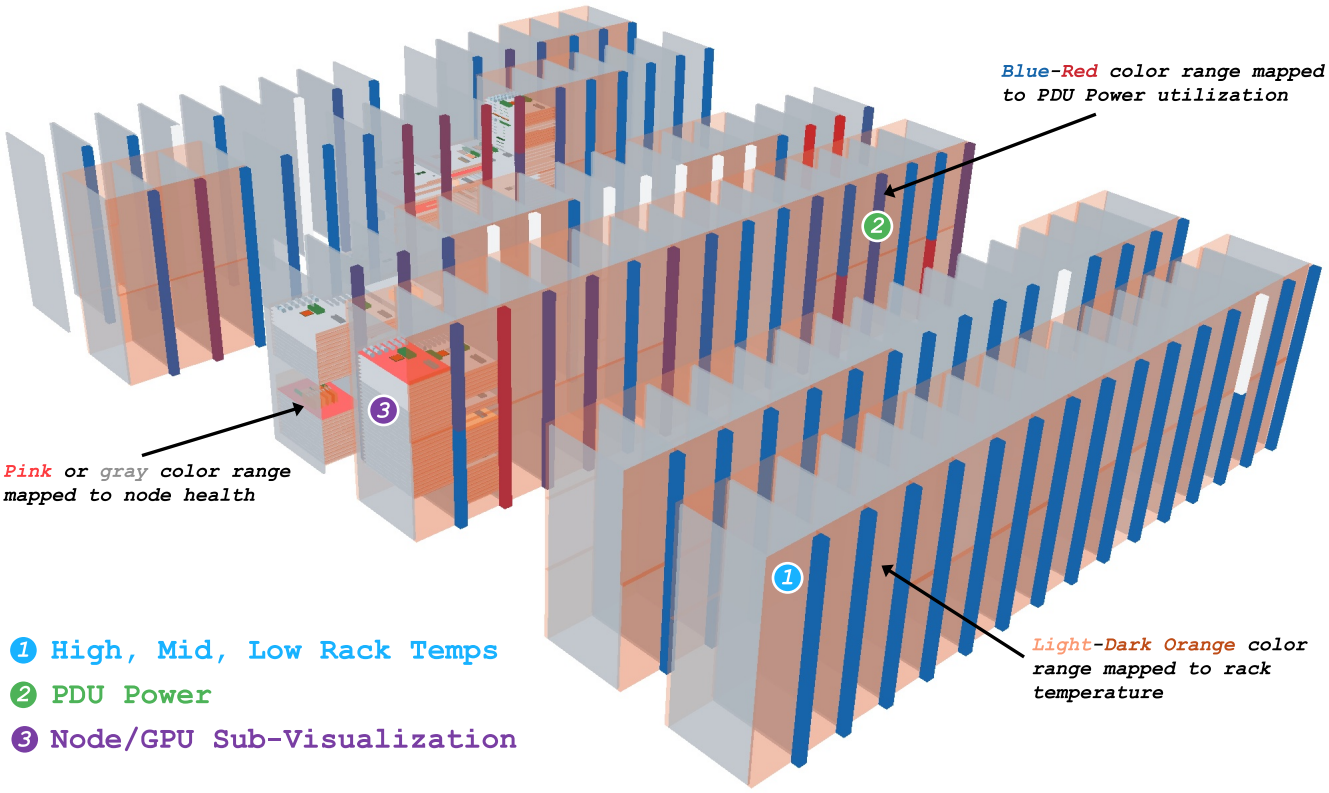

Fig. 8. Temperature and power of a data center, with a node and GPU thermal sub-visualization

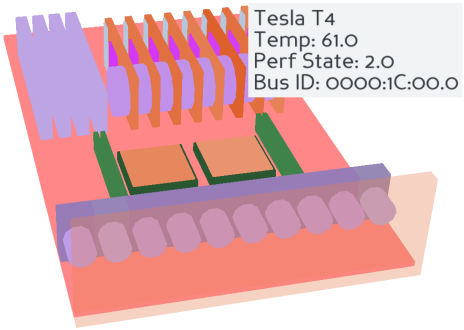

Fig. 9. Motherboard sub-visualization of node and GPU thermals, fan speed, and power draw

```
<juxatpose bind=".rack" axis="x" width="12" height="42"
           depth="20" align="left">
  <box bind=".pdu" height="100%" width="100%"
       depth="100%" color=".active_power|to_purple_range"
       opacity="0.5"/>
  <juxtapose axis="x">
    <juxtapose bind=".aisle" filter="='hot'" axis="y">
      <box bind=".env_sensor" height="equal%" width="1"
           depth="100%" color=".temp_pct|to_orange_range"
           opacity=".humidity_pct"/>
    </juxtapose>
    <box height="100%" width="100%" depth="100%"
         color="gray5" opacity="0.2"/>
    ... same juxtapose as above, except filter="='cold'"
  </juxtapose>
</juxtapose>
```

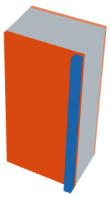

Fig. 10. Rack component of the data center visualization, without node sub-visualizations

Next, the power-supply boxes in the back show power usage, as a percentage of capacity. The 8 Nvidia GPUs at the back also report fan speed with the cylinder, utilization with the (currently purple) middle processor box, memory utilization with the horizontal box at the end, and thermals with the underlying plane. The data to produce these come from the InfluxDB data source.

With this visualization, it is clear that the bad state of the node is not the result of a processor, or DIMM stick failure since those visualization colors are green. Furthermore, if one were to hover over the processor to see the temperature of each, it would be clear that despite heavy GPU utilization and temperature (again by hovering and looking), the system is likely not in a bad state because of thermal issues, particularly because the inlet temperature is reasonable. These observations can quickly been made on near nodes as well.

To produce the final data center visualization from a (large) XML visualization file, four nested layout components were

```
<juxtapose bind="datacenter" spacing="100" axis="x">
  <juxtapose bind=".room" spacing="40" axis="x">
    <juxtapose bind=".pod" spacing="10" axis="x">
      <juxtapose bind=".row" axis="z">
        <include path="./ddc.rack.xml">
      </juxtapose>
    </juxtapose>
  </juxtapose>
</juxtapose>
```

Fig. 11. Pod and row component of the data center visualization

created:

- The outer component is the organization of the rack component into rows and pods, using relative bindings to restrict the data series of each row based on the rack's pod and room. This can be done with nested `juxtapose` elements bound to each level of the location hierarchy. This is shown in Figure 11. The rack component is imported via the `include` element.
- The next inner component is a rack model with front and back faces, along with a PDU bar. The rack model can be created by having a `box` element bound to a rack data node, sandwhiched together with the front and back faces. To produce a face for each aisle, a vertically juxtaposed wrapper element can be bound to a filtered aisle node, relative to the parent rack node, to group aisle dimensional metrics to each particular aisle (cold or hot) in a rack. This restriction can then be used with an inner `box` element bound to an environmental sensor node with a thin depth and the same width as the rack. To ensure the height of all the sensors, which may not be exactly three, matches the height of the rack, a top-down relative `equal%` attribute is used to specify that the height of each sensor `box` should be equal and add up to a parent `height=...` attribute defined above. This is shown in Figure 10.
- The following inner component is the node sub-visualization. Although the particular node visualization shown in Figure 9 is a 4U chassis with GPUs inside, other 1U or even 4-node/2U nodes have their own smaller sub-visualizations, and each XML visualization definition adapts to the mixed set of sensors and Redfish support across the dozen or so different chassis models. Each node sub-visualization is too large and complex to show here, particularly because of the heavy nested layouts required to make the visualization adaptable to inconsistent data offered by differing chassis models.
- The innermost component is a node's GPU sub-visualization. This is simply the juxtaposition of a cylinder (fan), square box (processor), and a wide box (memory) under a plane, rotated 90 degrees upwards.

V. Limitations

There are several notable conceptual and implementation limitations with the viz3 framework presented here. First, the tight integration between the data in a data source and the resulting data model results in a lack of data manipulation possibilities, making visualizations from more raw forms of data or unstructured data difficult or impossible. The viz3 framework does offer some simple, regex and dictionary based, data manipulation of metric strings in the configuration files to help with adding structure (which we heavily used in the Redfish motherboard visualization), but this manipulation does not apply to raw non-textual data. One glaring example of this is the inability to visualize instantaneous values such as CPU time and raw network octets, since an average needs to be taken to interpret such values and map them to a 3D space.

Another conceptual and implementation limitation with the mapping of data onto layouts is the lack of distinction between instances returned by a bound query being missing or simply inaccessible at the current moment. As currently designed, when an instance is not returned by a query—such as when a particular sensor goes down—the mesh or layout that was previously shown to the user is simply removed rather than shown in a down or unknown state. The current workaround for this issue is to ensure that the identifying data nodes in a data model, such as sensor names and hostnames, are stored in a more persistent database, rather than a time-series database which may not always report all instances as being present at the moment.

Similarly, the control flow offered by binding filters requires additional refinement. The rack sub-visualization, for example, requires switching out different motherboard visualization models to match the size of the chassis (e.g. 1U, 2x1U, 4x2U). The rack sub-visualization requires duplicate elements, each filtered for a distinct subset of corresponding motherboard models, to accomplish this. Such duplication significantly harms performance as the binding logic has to try every duplicated sibling element, for every data instance, despite only up to one element matching i.e. each binding filter construct is an if statement, without an else.

Finally, the directed acyclic graph model can limit queries to only processing data in singular direction, since no cycles can exist in the graph such as when nodes/metrics are bijective (one-to-one correspondence). This limitation was encountered when integrating the SQLite3 database, used in examples, with a more normal form (Boyce–Codd) database schema, requiring modifications that weakened the schema.

VI. Acknowledgements

I would like to thank Joe Breen for his guidance and support, the team at CHPC for their support and feedback, and notably, Robben Migacz, whose initial work on visualizing data centers using a tree data structure to partition a 3D space, evolved into this work. Real, more practical SQL schemas will likely encounter the same issue.

References

[1] "Grafana is the open source analytics & monitoring solution for every database." Grafana Labs. https://grafana.com/

[2] "Icinga: Monitor your entire infrastructure," Icinga. https://icinga.com/

[3] "Cacti: The complete rrdtool-based graphing solution," Cacti. https://www.cacti.net/

[4] "Elastic stack: Elasticsearch, kibana, beats & logstash," Elastic. https://www.elastic.co/elastic-stack/

[5] K. Husøy and C. Skourup, "3d visualization of integrated process information," in *Proceedings of the 4th Nordic conference on Human-computer interaction changing roles - NordiCHI '06*. ACM Press, 2006, p. 497–498. http://portal.acm.org/citation.cfm?doid=1182475.1182550

[6] R. ElHakim and M. ElHelw, "Interactive 3d visualization for wireless sensor networks," *The Visual Computer*, vol. 26, no. 6–8, p. 1071–1077, Jun 2010.

[7] N. Nguyen, L. Virgen, and T. Dang, "Hipervr: A virtual reality model for visualizing multidimensional health status of high performance computing systems," in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*. ACM, Jul 2019, p. 1–4. https://dl.acm.org/doi/10.1145/3332186.3337958

[8] e. a. Ngan V. T. Nguyen, Huyen N. Nguyen, "Jobnet: 2d and 3d visualization for temporal and structural association in high-performance computing system," in *Advances in Visual Computing. ISVC 2021.*, January 2022.

[9] Prometheus, "Prometheus: Monitoring system & time series database." https://prometheus.io/

[10] "Performance co-pilot," Performance Co-Pilot. https://pcp.io/

[11] "Influxdb: Open source time series database," InfluxData, 2022. https://www.influxdata.com/

[12] DMTF, "Redfish." https://www.dmtf.org/standards/redfish

[13] N. Nguyen, T. Dang, J. Hass, and Y. Chen, "Hiperjobviz: Visualizing resource allocations in high-performance computing center via multivariate health-status data," in *2019 IEEE/ACM Industry/University Joint International Workshop on Data-center Automation, Analytics, and Control (DAAC)*, Nov 2019, p. 19–24.

[14] T. Fujimoto, K. Goto, and M. Toyama, "3d visualization of data using supersql and unity," in *Proceedings of the 22nd International Database Engineering & Applications Symposium on - IDEAS 2018*. ACM Press, 2018, p. 141–147. http://dl.acm.org/citation.cfm?doid=3216122.3216145

[15] R. D. Hipp, "Sqlite3," 2022. https://www.sqlite.org/index.html

[16] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, p. 377–387, Jun 1970.

[17] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM Transactions on Database Systems*, vol. 1, p. 9–36, Mar 1976.

[18] Prometheus, "Prometheus: Data model." https://prometheus.io/docs/concepts/data_model/

[19] Z. Liu and J. T. Stasko, "Mental models, visual reasoning and interaction in information visualization: A top-down perspective," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 6, p. 999–1008, Nov 2010.

[20] "Security docs," Grafana Labs. https://grafana.com/docs/grafana/latest/administration/security/

[21] R. Cabello, "Threejs: Javascript 3d library," 2022. https://threejs.org/

[22] "Panda3d: The open source framework for 3d rendering and games," 2022. https://www.panda3d.org/

[23] "Using server-sent events," MDN Contributors. https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events

[24] jenningsloy318, "redfish-exporter," Mar 2022. https://github.com/jenningsloy318/redfish_exporter