universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

"Scalability of Modern Scatterplot Visualizations for Large Image Datasets"

verfasst von / submitted by

Sebastian Klaassen, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2017 / Vienna 2017

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 935
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Medieninformatik
Betreut von / Supervisor:	Univ.-Prof. Torsten Möller, PhD

Abstract

In this thesis we discuss the representation of large high dimensional image databases with modern scatterplot-based visualization techniques. We investigate scatterplot scalability both in terms of what is feasible (performance scalability) and what is reasonable (information scalability). We create two interactive scatterplot-based visualizations for large-scale image datasets, the Global View and the Interactive Cell Plot. The Global View is a desktop application for visualizing peta-scale simulations using in-situ generated image databases, developed in collaboration with the Los Alamos National Laboratory. Global View reads image databases in the Cinema database format. The loaded images are visualized by applying scripted visual mappings using our novel visual mapping scripting language. Using Global View, we compare different visual mappings for the MPAS Ocean dataset and conclude that the different mappings represent a trade-off between an intuitive viewing experience and showing multiple images simultaneously. Once a visual mapping is defined, the image database can be explored by traversing a three dimensional virtual environment of images. Our texture streaming algorithm dynamically loads and unloads images while the image database is explored, by monitoring the amount of allocated video memory. The Interactive Cell Plot is a JavaScript library for rendering large scatterplots with WebGL, developed in collaboration with the Allen Institute for Cell Science. Our efficient rendering technique enables us to render one million two dimensional data points at 60 frames per second and five million points at 25 frames per second. We evaluate visualization parameters of the Interactive Cell Plot and five different thumbnail placement strategies in a qualitative user study. Our novel algorithms for boundary- and density-based labeling minimize occlusions of data points while also minimizing the distance between label and site. We introduce density maps as an intermediate data structure for fast clustering, labeling, characteristic point detection and sample generation. By estimating density map generation runtime, our algorithm allows the user to directly control the performance-accuracy-trade-off of the density map creation algorithm.

Zusammenfassung

In dieser Arbeit befassen wir uns mit der Darstellung von großen hochdimensionalen Bilddatenbanken mit modernen Streudiagramm-basierten Visualisierungstechniken. Wir erforschen die Skalierbarkeit von Streudiagrammen sowohl in Bezug auf Machbarkeit (Performance-Skalierbarkeit), als auch in Bezug auf Sinnhaftigkeit (Informationsskalierbarkeit). Wir erstellen zwei interaktive Streudiagramm-basierte Visualisierungen für große Bilddatensätze, der Global View und der Interactive Cell Plot. Global View ist eine Desktopanwendung zur Visualisierung von Petascale-Simulationen durch in-situ generierte Bilddatenbanken, die in Zusammenarbeit mit dem Los Alamos National Laboratory entwickelt wurde. Global View liest Bilddatenbanken im Cinema Datenformat ein. Die geladenen Bilder werden durch geskriptete visuelle Zuordnungen, von unserer neuartigen Skriptsprache für visuelle Zuordnungen, visualisiert. Wir vergleichen verschiedene visuelle Zuordnungen für den MPAS-Ocean-Datensatz mittels Global View und schließen daraus, dass die verschiedenen visuellen Zuordnungen einen Kompromiss zwischen intuitiver Darstellung und der gleichzeitigen Darstellung mehrerer Bilder darstellen. Sobald eine visuelle Zuordnung definiert ist, kann die Bilddatenbank durch die Durchquerung einer dreidimensionalen virtuellen Umgebung von Bildern erforscht werden. Unser Texture-Streaming-Algorithmus lädt und entlädt Bilder dynamisch, während die Bilddatenbank erforscht wird, indem die zugewiesene Menge an Grafikspeicher überwacht wird. Der Interactive Cell Plot ist eine JavaScript-Bibliothek zum Rendern großer Streudiagramme mittels WebGL, die in Zusammenarbeit mit dem Allen Institute for Cell Science entwickelt wurde. Unsere effiziente Renderingtechnik ermglicht es uns, eine Million zweidimensionale Datenpunkte mit 60 Bildern pro Sekunde und fünf Millionen Punkte mit 25 Bildern pro Sekunde zu rendern. Wir evaluieren Visualisierungsparameter des Interactive Cell Plot und fünf verschiedene Plazierungsstrategien für Vorschaubilder in einer qualitativen Anwenderstudie. Unsere neuartigen Algorithmen für rand- und dichtebasierte Beschriftung minimieren sowohl Überlappungen von Datenpunkten, als auch die Distanz zwischen Beschriftung und Referenz. Wir stellen Dichtekarten als Zwischenstruktur für schnelles Clustering und schnelle Beschriftung, Stichprobengenerierung und Erkennung Charakteristischer Datenpunkte vor. Durch die Laufzeitschätzung der Dichtekartengenerierung gibt unser Algorithmus dem Benutzer direkte Kontrolle über den Kompromiss zwischen Laufzeit und Genauigkeit der Dichtekartenerstellung.

Contents

1 Motivation

A scatterplot visualizes tabular data with points in two-dimensional space. Each row of the table describes a single data record. Each column represents a dimension of the dataset. If the order of data records matters, then the row index represents another dimension. Datasets with more than two dimensions have to be projected to two-dimensional space.

Among all the forms of statistical graphics, the humble scatterplot may be considered the most versatile, polymorphic, and generally useful invention in the entire history of statistical graphics [19]. Since the scatterplot was first introduced by J. F. W. Herschel in 1833 [19], the type and amount of data scientists are visualizing has changed drastically. In this thesis we collaborate with the Los Alamos National Laboratory (LANL) and the Allen Institute for Cell Science (AICS) to visualize modern large-scale datasets using scatterplot-based software.

1.1 Use case 1: Visualizing peta-scale simulations using in-situ generated image databases

As scientific simulations have increased in complexity and accuracy, computing power and capacity has grown to peta-scale. The data generated by massive scientific simulations have reached the scale of petabytes and are still increasing in size. Generated data is becoming too large to analyze with conventional methods because of storage and execution time constraints. The cost of transferring data will likely be high, when the amount of data reaches petascale. Analyzing petascale-level simulation output is achieved by writing a subset of data onto disk and analyzing the data offline. A dedicated visualization machine runs a limited number of simulation time-steps out of thousands of steps generated by the simulation. In this process most of the computing time is spent on I/O operations since data has to be transferred to a powerful visualization machine almost similar to a supercomputer that does the calculations of the simulation. Options are either to look at an even smaller subset of data, which defeats the purpose of performing the original high-resolution simulation enabled by petascale computing, or not to move the raw data itself. The scientific community is reluctant to use supercomputer time for generating visualizations and coupling the simulation code with visualization. There can be conflicts in code optimization for parallel simulation code and visualization, resulting in costly and error prone replication of data and inter-processor communication. Even if the same domain decomposition could be used, duplicating data at partition boundaries might result in memory overhead. Visualization computation has to be designed to take a fraction of the overall simulation time. Appropriate color and opacity transfer functions have to be derived from the domain using an adaptive method without constantly acquiring global information. Co-processing is desirable for large scale data visualization since relevant data and geometry of the simulation are too expensive to be collected with a post processing step.

Better results can be expected with real time data communication for in-situ processing. Such simulation time in-situ visualization applications can perform rendering on smaller chunks of data with less transfer cost early in the visualization pipeline. The in-situ approach allows data sharing, so both the simulation and visualization calculations can run on the same supercomputer. Using the in-situ pipeline, visualization results are generated in the form of large image databases while the simulation is running, enabling scientists to analyze the simulation at runtime.

Early examples of in-situ generated image databases exhaust the scalability of existing image viewers and introduce new challenges to the visualization community. Domain scientists require better tools for efficiently and intuitively analyzing large high-dimensional image datasets.

A Cinema database is a set of precomputed data artifacts (i.e. images) that can be queried and interactively viewed [1]. Cinema consists of a set of specifications, exporters and viewers for image databases. ParaView recently added support for automatically generating in-situ databases in Cinema format.

Traditionally, large high dimensional datasets are visualized by showing only a small subsets of the data. Some of the most successful image viewers, such as the Cinema viewer reference implementation [1], Windows Photo Viewer or Apple Preview, show only one image at a time and let the user browse through the data space by selecting another image from the closest images in each dimension. This approach is called local view, because the user never sees more than the local neighborhood of the currently viewed image. It is the preferred approach for fine grained exploration, like when a photographer compares multiple photos of the same subject to find the best quality picture.

Using the local view approach to gain an overview of the full dataset can be very time consuming. A global view approach shows either the full dataset or a large subset, by choosing a high dimensional visual mapping (see below for a description of visual mappings) that preserves many dimensions of the dataset and by displaying many data points at a time. The global view is the preferred approach for overview tasks, like when a photographer compares a collection of photos for a photo album to make sure the color scheme is uniform throughout the album. The minimalistic design of a scatterplot allows it to display much larger datsets than other visualizations. Therefore, it is the ideal basis for the global view.

1.2 Use case 2: Browsing cells using a high dimensional scatterplot of cell properties

The initial project on cell organization by the Allen Institute for Cell Science reveals the enormous variance in the organization of the cells, raising questions about the nature of this variance and its biological origin (cell cycle, differentiation, etc.) and function [32]. They define a common coordinate framework, enabling statistical analyses appropriate for these relatively large datasets. Allen Cell's online portal provides an unprecedented view into the organizational diversity of human stem cells by combining large-scale 3D imaging data, predictive models, observations of cells, detailed methods, and cell lines available for use in labs around the world[32].

The 3D images produced by Allen Cell's imaging pipeline are dense with information, including the positions of the DNA (DNA dye), the cell membrane (membrane dye) and the tagged protein associated with the different cellular organelles. An interactive plotting tool is provided on the Allen Cell web page to analyze trends by exploring cell feature data.

The interactive plotting tool [18] is a key component in the stem cell analysis pipeline. It visualizes a dataset that is periodically expanded as more stem cells are being analyzed. Rendering large datasets, like the cell dataset (see section 4.2) in a browser at interactive frame rates is challenging. None of the existing web charting libraries are optimized for efficient rendering of such large datasets.

We ensure the scalability of the interactive plotting tool by implementing a rendering pipeline that optimizes point rendering efficiency in section 4.3. We prove the scalability of our implementation by generating and rendering a dataset, 100 times larger than the current cell dataset in section 8.1. We benchmark data throughput by rendering up to 10 million points in section 8.2. We collaborate with the software engineering team of AICS on design of a new user interface for the interactive plotting tool. In section 8.3 we optimize interface parameters for the best user

experience as part of a qualitative user study. In the second part of our user study we answer the question of how to best present thumbnails of rendered cell images, without occluding cell feature data in the plot. We cover theory and strategies for thumbnail selection and placement in section 7. Some of the presented algorithms, such as density-based thumbnail placement, depend on precomputed point density. In section 6 we introduce density maps as an intermediate data structure for clustering, label placement, computing characteristic points and sampling the kernel density of a dataset.

1.3 Implemented applications

For this thesis we created two interactive scatterplot-based visualizations:

1. GlobalView (GV) is a desktop application, designed to visualize large image collections by drawing data points as images in a 3D environment.
2. The Interactive Cell Plot (ICP) is a browser application, designed to visualize large scatterplots annotated with image thumbnails.

GV is optimized to fully utilize modern graphic acceleration and display its output on a large screen. In contrast, browser applications have to be designed in an agnostic way [25] that allows viewing in low resolution, low bandwidth and low performance environments. To achieve the best possible browser performance, ICP renders data records as points and tags some of the records with a thumbnail of the cell image.

Differences between the two visualizations are summarized in table 1.

	Global View	Interactive Cell Plot
Domain	3D	2D
Scalability focus	Number and size of images (hundreds of GBs of images)	Number of points (millions of points)
Image visualization	Images as data points	Points labeled with thumbnails
Visual mapping	Scripted	Direct mapping
Environment	Desktop (multi-platform)	Browser
Programming Language	C# (Mono)	JavaScript
Graphics Framework	OpenGL	WebGL
Research areas	I) Visual mapping	I) Density maps II) Labeling

Table 1: Differences between the two scatterplot viewers

The main contributions of this thesis are the exploration and formal definition of visual mappings (section 5 and appendix A), the analysis of different thumbnail placement strategies (section 7.2) and the computation and usage of density maps (sections 6, 7.4, 6.3 and 8.1).

Our density map based outlier detection algorithm shows clear advantages over previous density based anomaly detection methods like DBSCAN and LOF. Most importantly, it is faster on large datasets, because the time to compute a density map depends only linearly on the size of the dataset. Once a density map is computed, the time complexity of the clustering algorithm only depends on the size of the density map. The generated density map can also be used for other analytical tasks, including visualization and statistical testing [20]. Finally, density map based outlier detection has more easily tunable parameters than DBSCAN or LOF (see last paragraph of section 6.1).

We create two image viewers. The desktop viewer, GlobalView, is a cross platform image viewer that can visualize Cinema image databases of any size. The web viewer, Interactive Cell Plot, is a scatterplot library for JavaScript that outperforms commercial charting libraries in rendering performance for very large datasets. Both tools are available on GitHub. GlobalView is released under the name "global_view" [27] and the Interactive Cell Plot is released under the name "GlobalView.js" [28].

2 Related work

2.1 Visual mapping

A scatterplot displays the relationship of two quantitative variables by transforming the corresponding dataset dimensions onto x-axis and y-axis of the plot. Chi classifies this transformation in his taxonomy of visualization techniques as visual mapping [11]. A visual mapping is an abstract transform operator that takes information in a visualizable format and presents a graphical view. In the realm of multidimensional scatterplots Chi defines the visual mapping operation as choosing a variable-to-axes mapping. We expand this definition by formulating the visual mapping that maps an n-dimensional datset into a two-dimensional scatterplot as the non-linear transformation matrix $V \in R^{nx2}$. The two spatial dimensions of a scatterplot are called layout dimensions [6]. Besides the layout dimensions, other visual features of the scatterplot can be used to encode additional dimensions. Visual features [6] are also known as marks [31], graphical attributes [3] or graphical properties [16]. We summarize some of them in table 2).

It is worth noting that not all visual features are equally suitable for visualizing quantities. Cleveland and McGill have identified 10 elementary perceptual tasks to extract quantitative information from graphs and ranked them by accuracy of extraction, ordered from most to least accurate [13].

1. Position along a common scale (e.g. along an axis)
2. position along nonaligned scales
3. Length, direction or angle
4. Area
5. Volume or curvature
6. Shading or color separation

Quantities are encoded inside visual features in the form of one or more perceptual tasks. An exception are interactive or time-varying features, since Cleveland and McGill's taxonomy only describes static graphs.

Visual feature	Description
Layout	Mapping values to data point locations
Color	Coloring points by encoding values with a colormap
Opacity	Drawing points with a high value more opaque
Size	Drawing points with a high value larger
Meta-visualization	Drawing points as small charts (e.g. pie charts or bar charts)
Visibility	Hiding all but one part of a data series (e.g. animating or interactively hiding groups of points)
Motion	Highlighting or grouping points by briefly animating their locations [3]

Table 2: Visual features of a scatterplot

Images are themselves visualizations of the displayed content. By drawing images as points of a scatterplot, the Global View (GV) qualifies as a meta-visualization in the systematization of Bertini et al. [6]. This effect becomes apparent when a user zooms into an image.

Combs and Bederson describe an image browser as an application that allows users to select one or more images from multiple images by supporting both viewing of multiple images at a time and inspection of full resolution versions of individual images [14]. They developed the Zoomable Image Browser (ZIB). ZIB offers a unique advantage over many browsing systems in that the user has control of the trade-off between the number of images displayed and the resolution of those images. We have implemented this focus-and-context technique into GV. By zooming in, the visualization transitions seamlessly from a global overview to a local comparison view, and finally to a single image view. Combs and Bederson compare ZIB to three more image viewers in a comprehensive user study. The browser that performs almost as well as ZIB is the classic 2D grid viewer of ThumbsPlus [15]. The other two browsers are Simple LandScape, a 3D browser that allows users to fly through a forest of images and PhotoGoRound, a rotatable 3D viewer that uses a "lazy Susan" metaphor. Both 3D browsers, as well as GV, show images as billboards. They always face the viewer, independent of viewing angle. Every viewer in Comb and Bederson's user study can be implemented using GV with corresponding visual mappings.

2.2 Density maps

Density based clustering and outlier detection methods like DBSCAN [17] and LOF [8] are computed based on the probability density function (PDF) of the sample points. They sample the PDF at each data point by measuring distances to neighboring points.

Clustering by fast search and find of density peaks (CFSFDP) computes local density around every point and searches for high density points to detect cluster centers [33]. Mehmood et al. extend CFSFDP by computing a kernel density estimate (KDE) via the heat distribution method (CFSFDP-HD) [30]. The KDE is a non-parametric way to estimate the probability density function. CFSFDP-HD samples the PDF by computing the KDE with a Gaussian kernel from neighbors closer than a maximum point distance called bandwidth. Gan and Bailis implement fast KDE based clustering, by only focusing on the density threshold that separates clusters and pruning branches of the computation that go beyond the threshold [20].

Density based clustering methods differ from our approach in that they sample the PDF at the locations of each data point. A density map consists of regular sampled densities throughout the spatial domain.

2.3 Labeling

The problem of superimposing annotations over static content while minimizing occlusions is known in literature as label placement. Label placement is an important research area according to the ACM Computational Geometry Impact Task Force report [10]. In our case we want to annotate points selected during the thumbnail selection step. We will refer to these points as sites. Each thumbnail (i.e label) is connected to exactly one site using either a straight line, called a leader, or by annotating both thumbnail and site with identical numbers.

Boundary label placement was introduced by Bekos et al [5]. According to their classification of boundary placement strategies, our boundary thumbnail placement strategy classifies as four-sided (i.e. labeling each side of the Axis Aligned Bounding Box (AABB) of the plot) type-s (i.e. leaders are straight lines) one-to-one (i.e. each site is connected to exactly one label) boundary placement. They describe a problem where n uniform labels are evenly distributed along the AABB, which reduces label placement to the task of assigning sites to labels such that no two leaders intersect and solve it in $O(n \log n)$ time. We introduce a different algorithm in section 7.5 that optimizes label positions that are sparsely distributed along the AABB.

3 Application 1: Global View

The first scatterplot we implement is the Global View (GV). Figure 1 shows the interface of GV. The upper part shows the loaded images. Images have a fixed size and they always face the user. Placement of images is controlled by applying visual mappings that map dataset dimensions to image coordinates. Visual mappings are scripted through an integrated terminal at the bottom of the interface.

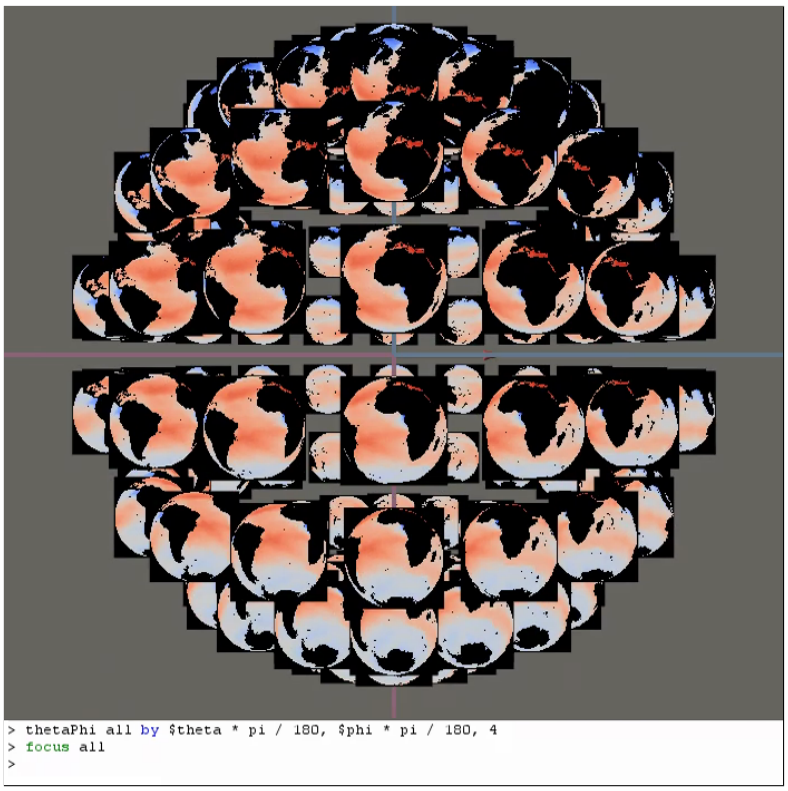


Figure 1: The GlobalView image viewer.

3.1 Requirements

We design GlobalView to fulfill the following requirements

- Cinema format
  GV is designed as a successor to the Cinema viewer [1], an image viewer for large image databases that uses the local view approach. GV supports Cinema image databases in Astaire or Chaplin (former Bacall) format. The full Cinema image database specifications are at the Cinema homepage [1]. One implication of this database format is that datasets are assumed to contain images for all combinations of input parameters. For example, if a database in Astaire format uses the input parameters $x = \{true, false\}$ and $y = \{1, 2, ..., 10\}$, then it must contain all images $(x, y) = \{(true, 1), (false, 1), (true, 2), (false, 2), (true, 3), ..., (false, 10)\}$.

- Performance scalability
  GV is used to visualize the output of large scale scientific simulations. The generated high dimensional image databases contain hundreds of high resolution images with a total size above 100GB (see 3.2). Section 3.4 describes our novel texture streaming pipeline that supports image databases that are too large to fit into main memory.
- Information scalability
  We use GV to explore how many images of the visualized high dimensional datasets can be simultaneously presented to the user. Therefore, we use a form of scatterplot that displays individual images as data points.
- Flexibility of visual mappings
  The information scalability requirement necessitates a thorough exploration of different visual mappings. We describe different visual mappings in section 5. GV uses an integrated terminal to script visual mappings using a visual mapping scripting language that we describe in appendix A.
- Platform independence
  GV is a desktop application written in Mono, a cross-platform implementation of Microsoft C#, using OpenGL for rendering. The integrated terminal uses the Fast Colored TextBox library by Pavel Torgashov [35]. GV has been tested on Ubuntu, Mac OS, Windows 7 and Windows 10.

3.2 MPAS dataset

As a working example of GV we use the MPAS dataset. The MPAS dataset is a Cinema [1] database of in-situ generated images from the Okubo-Weiss simulation [38]. The simulation was run using the Model for Prediction Across Scales (MPAS), a climate modeling framework that is developed by the Los Alamos National Laboratory (LANL) in cooperation with the National Center for Atmospheric Research (NCAR). The database was created using the ParaView Catalyst pipeline [2].

3.3 3D Interaction

Munzner describes, that even though we are living in a three dimensional reality, most of our cognitive reasoning happens in a two dimensional domain [31]. As a result, 2D visualizations are more accurate and easier to comprehend. We choose a three dimensional domain for GV, because the extra dimension gives us more flexibility for designing visual mappings of high dimensional datasets. It also allows a possible future adaption of GV for virtual reality environments.

Image locations in GV are controlled mainly by defining a visual mapping through our scripting interface (see section 5). Once images are placed in 3D space, they can be explored by moving the camera with keyboard and mouse. Whenever visual mappings are modified, images are animated towards their updated positions to avoid sudden changes in the layout of the visualization. According to the principle of congruence [36], such sudden changes are disruptive, since they prevent users from tracking changes over time [16].

Interaction is essential to exploratory visual analysis [21], [31]. Therefore we allow the user to highlight, move or remove images from the coordinates defined by the visual mapping.

3.4 Texture streaming

Modern image datasets are growing in size faster than commodity hardware. We have implemented a novel texture streaming algorithm that optimizes viewing experience on hardware with a memory hierarchy that satisfies *permanent storage size* $\geq$ *main memory size* $\geq$ *video memory size*. The only input to this algorithm is a single scalar priority value per image. The priority value indicates the level of importance of the image to the user. We use the number of visible pixels of an image as heuristic for image priority. An image that is completely occluded or out of sight has a priority of zero, while a visible image has a priority proportional to the number of visible pixels. When an image is drawn on screen, we count the number of pixels using OpenGL atomic counters.

3.4.1 Texture streaming implementation

On a background thread, texture streaming periodically queries the highest priority image that hasn't been loaded yet. It stores a full resolution version of the image in main memory and a scaled down version in video memory. The amount of scaling depends on the visible size of the image on screen. When the visible size of a loaded image changes by more than a factor of two, the image in video memory is recomputed from the full-size image in main memory. For example, this could happen due to the user zooming in or out. As soon as available memory is exhausted, we free memory by unloading low priority images. In OpenGL it is not possible to query the amount of available video memory. Therefore, we define a constant memory limit. By changing this limit the user has control over the memory imapct of GV on the host system.

To implement flicker-free animations, consecutive frames have to be prefetched before they are displayed. We implement prefetching by computing a future priority estimate, based on the expected visibility of a consecutive frame. This is done by sampling the time-dependent visual mapping with a future time value (e.g. 500 milliseconds ahead). Instead of drawing images at that future time, we only estimate which of the images will be visible to compute the future priority estimate. The future priority estimate is added to the current priority estimate, so that images receive a high priority before they become visible.

3.5 User reception

The applicability of GV is limited by the assumption of the Astaire and Chaplin specifications of Cinema that the database contains images for every combination of input parameters. However, most datasets, e.g. figure 3, contain unevenly distributed images. In order to load datasets like figure 3, we implemented a way to load non-Cinema conforming datasets. This format restriction was removed with the Dietrich specification of Cinema, following feedback we provided after implementing GV.

We successfully used GV to study different kinds of visual mappings and evaluate scalability in terms of rendering large amounts of images to screen. However, the scripting language interface and 3D camera control make GV difficult to operate. We decided to design a second scatterplot based visualization tool, with a strong focus on usability. This tool, named the Interactive Cell Plot (ICP), is described in the next chapter. Requirements of ICP are based on our observations from GV.

4 Application 2: Interactive Cell Plot

The second scatterplot-based visualization we implement is the Interactive Cell Plot (ICP). ICP is designed to replace the current Interactive Plotting application on the website of the Allen Institute for Cell Science [18]. ICP is designed to be less flexible, but easier to use than GV. It has a traditional two dimensional interface and supports only visual mappings of single input dimensions to each of x-, y- and color axes.

ICP is a library for efficiently rendering scatterplots of large datasets on a web page. For the purpose of development, we implement a sandbox interface with ICP on the left and a list of basic controls on the right, shown in figure 2. The top group of controls are used to select a dataset and assign dataset dimensions to x-, y-, and color axes. Underneath are controls for altering visualization parameters. The third group controls density-, cluster- and histogram visualizations. The fourth group consists of only one button that starts the benchmark of section 8.2. The last group contains controls for thumbnail selection and placement, see section 7.

We evaluate which of the controls to expose in the final interface on the cell science web page [18] in a user study 8.3.

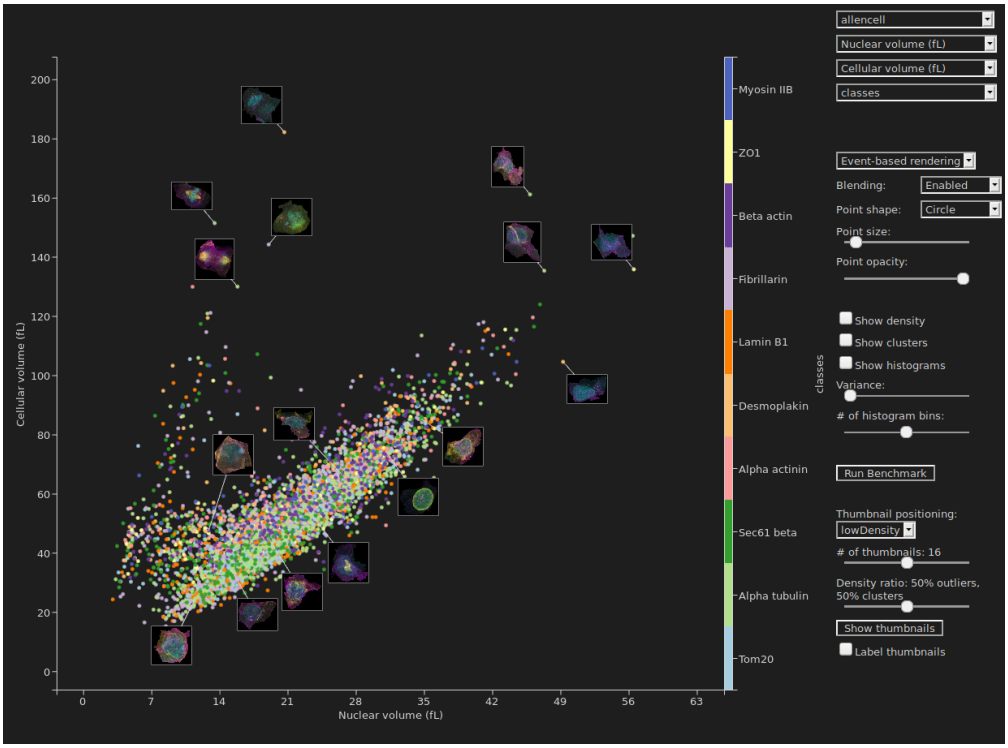


Figure 2: The Interactive Cell Plot sandbox interface.

4.1 Requirements

Based on what we observed from GV, we design ICP to fulfill the following requirements

- Deployment as a web application
  ICP is designed as a successor to the current Interactive Plotting web application [18], a scatterplot based tool for interactively exploring the cell dataset. The cell dataset is described in section 4.2.

- Annotation of images with thumbnails
  Using GV, users can immediately study all 6077 images of the cell dataset (see figure 3), but the small size of the cell images requires constant interaction (panning and zooming) to study cell image details while avoiding the loss of context. This interaction paradigm is called pan&zoom [4]. ICP replaces the pan&zoom paradigm with focus+context interaction, by superimposing larger thumbnails (focus) over a point-based scatterplot (context).

  Another reason for choosing thumbnails over images as points is the performance impact. When the cell dataset is loaded into GV (see figure 3), it takes about four seconds to asynchronously load all images from an SSD drive. The frame rate for rendering all images at once is only 4 frames per second. Because ICP is a web application, we expect frame rates to be even lower and we expect a significantly higher loading time to retrieve the 6077 images over a network connection. By rendering data points as points, not images, we achieve much better performance (see section 8.2).

- Performance scalability
  To achieve the highest possible performance, we implement ICP with WebGL. In section 4.3 we discuss techniques to optimize point rendering in a web application. In section 8.1 we show that the cell dataset can still be interactively explored with ICP after its size has been extended to 100 times the current number of cells.

4.2 Cell dataset

As a working example of ICP we use the cell dataset. The cell dataset of the Allen Institute for Cell Science [32] contains cell images and features from the Wild Type C (WTC) human induced pluripotent stem cell (hiPSC) line, produced by Bruce Conklin [29]. For this paper we are working with version 1.5 of the dataset, which consists of 6077 individual cells. Each cell has 6 properties, a cell class and a rendered cell image.

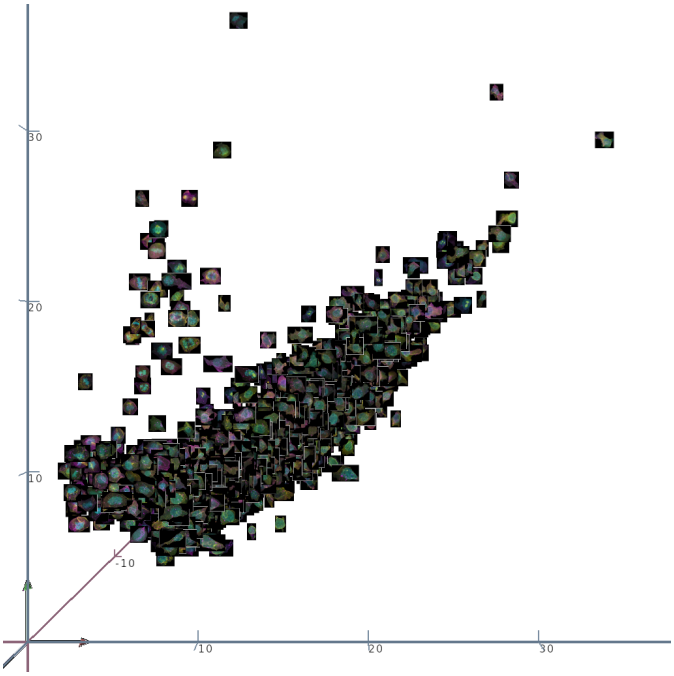


Figure 3: The cell dataset, rendered with GV.

4.3 Fast point rendering in WebGL

WebGL is an OpenGL wrapper for JavaScript. OpenGL only uses the graphics accelerator efficiently when communication with the graphics driver is low. This applies even more to WebGL applications, because browser executed JavaScript code is slower than natively executed C++ code.

When a dataset is loaded, we store the full dataset in graphics memory as one continuous buffer. Later calls to WebGL only change how data is rendered, they never change the data itself.

In the following sections we describe vertex and fragment shaders for fast point rendering in WebGL. We benchmark the presented code with up to 10 million points in section 8.2.

4.3.1 Vertex shader

Every time the user changes the view, we compile a vertex shader (see listing 2) that defines the transformation from the static data table in memory to the requested output on screen. The three dimensional point location `pos` is computed in line 6. The first two components of `pos` are x and y position of the point in normalized device coordinates. The vertex position is set to these coordinates in line 8. The third component of `pos` is the color dimension in the range $0 \leq pos_z \leq 1$. It is used as an index into a colormap in line 7. We use a two dimensional texture for the colormap, because WebGL doesn't support one dimensional textures. The variable `color` is used in the fragment shader.

Blocks {b} and {c} of listing 2 define visual mappings in the form of linear combinations of attributes. When the user switches to a different visual mapping, the new visual mapping code is

inserted into block {b} and the old visual mapping code is inserted into block {c}. The transition between old and new mappings is animated by controlling vectors `scales` and `aniScales`.

Dataset dimensions used by blocks {b} and {c} are defined as attributes in block {a}. Because WebGL supports only a limited number of attribute vectors and the maximum vector size is 4, we group attributes in lists of length ≤ 4. For example, if code blocks {b} and {c} require six different dataset dimensions, the attribute code inserted into block {a} is the following:

```
attribute vec4 attributeSet1;
attribute vec2 attributeSet2;
```

Listing 1: Code block for defining six unique attributes.

The assignment of dataset dimensions to attributes is done in JavaScript using the WebGL function `gl.vertexAttribPointer(...)`. View zooming and panning is also done in JavaScript, by controlling the vectors `offsets`, `scales` and `aniScales`.

```
uniform vec3 offsets, scales, aniScales;
uniform sampler2D colormap;
varying vec4 color;
{a}
vec3 main() {
  vec3 pos = offsets + vec3({b}) * scales + vec3({c}) * aniScales;
  color = texture2D(colormap, vec2(pos.z, 0.5));
  gl_Position = vec4(pos.xy, 0.0, 1.0);
}
```

Listing 2: GLSL vertex shader code for rendering data points. Attribute definition code {a} and visual mapping code for the static plot {b} and the animated transition {c} are dynamically generated.

4.3.2 Fragment shader

In the fragment shader (listing 3) we define a point shape as a function of $p \in \mathbb{R}^2$ and apply the point color defined in the vertex shader. Line 6 of listing 3 converts point coordinates from texture space $0 \leq p_1, p_2 \leq 1$ to the range $-1 \leq p_1, p_2 \leq 1$, because it is easier to formulate symetrical point shapes in this space. Most figures in this thesis use the circular point shape. The circular point shape is defined as $1 - |p|^{0.25\, pointSize}$. The factor 0.25 controls the sharpness of the point's outline.

```
varying vec4 color;
float pointShape(in vec2 p) {
  return {d};
}
void main() {
  color.a = pointShape(gl_PointCoord * 2.0 - 1.0);
  color.a = clamp(color.a, 0.0, 1.0);
  gl_FragColor = color;
}
```

Listing 3: GLSL fragment shader code for rendering data points. Point shape code {d} is a user defined function of p.

5 Visual mapping

When an image database is loaded into GV, the visual mapping is an identity matrix. All images reside at the coordinate system origin. To map dataset dimensions to spatial dimensions, a corresponding transformation has to be applied to the visual mapping matrix. Most high-dimensional visualization tools use either a hard-coded visual mapping or let the user choose which dimension to map onto each of the available visual features. Hard-coded visual mappings allow full control over the visual mapping at compile-time. GV gives the user full control over the visual mapping at runtime, by supporting the composition of transformations from within the application.

An ideal interface for composing visual mappings would be an intuitive touch- or drag-and-drop interface. Creating such an interface capable of composing a plethora of different visual mappings would require vast effort. For the sake of studying on-the-fly compositions of visual mappings we prefer the flexibility of a textual interface over the ease of use of a touch- or drag-and-drop interface. Programing visual mappings through a textual interface requires an appropriate notation. Algebraic notations are commonly used to formalize visual mappings (e.g. Ziemkiewicz and Kosara's mathematical formalization of visual mappings[39] or Kindlmann and Scheidegger's algebraic process for visualization design [26]), but to the best of our knowledge, we are the first to formally define a programmable notation of visual mappings. We propose a simple scripting language, similar to SQL, to formulate visual mapping transformations (see appendix A).

The MPAS dataset is a visualization of simulated currents in the earth's oceans. It contains views of the earth from a constant distance in regular intervals along latitude and longitude. In the following we investigate different visual mappings to present the MPAS dataset.

5.1 Identity mapping

When we load the dataset into GV, the visual mapping is initialized as an identity matrix. All images reside at the coordinate system origin. The identity mapping is shown in figure 4.

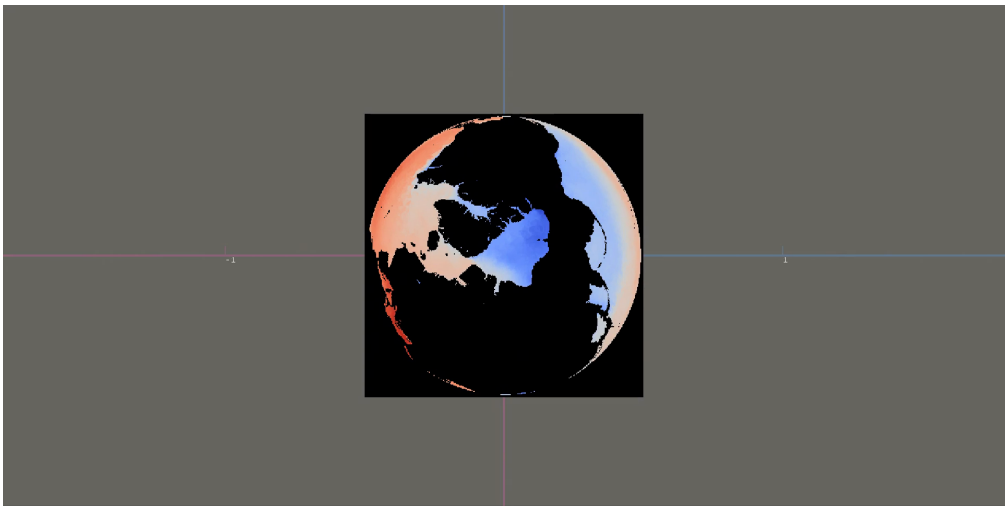


Figure 4: The MPAS dataset with an identity mapping.

5.2 Cartesian mapping

One way of aligning the spacial dimensions of the dataset is to assign latitude and longitude dimensions to x and y axes of the scatterplot. The Cartesian mapping is shown in figure 5. This mapping uses almost the entire available screen area to display all spacial views in identical size. However, the mapping of latitude and longitude to horizontal and vertical axes is unintuitive for most users, which makes this mapping hard to interpret. Below we investigate more intuitive visual mappings for latitude and longitude.

In GV we produce this mapping by issuing the command `STAR all BY $theta, $phi`. This applies a Cartesian mapping (i.e. "star" mapping) to all images by dataset dimensions longitude (i.e. camera angle "theta") and latitude (i.e. camera angle "phi"). See appendix A for details of our visual mapping scripting language.

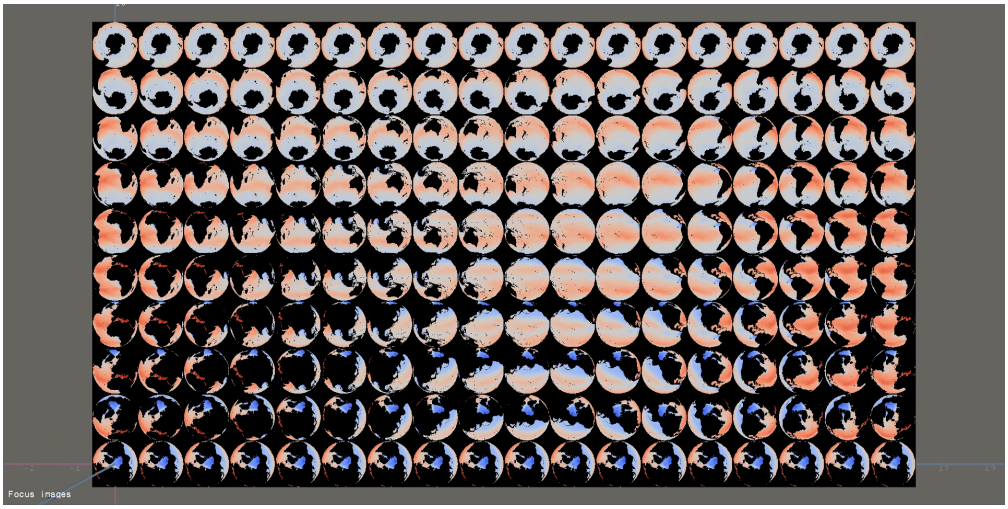


Figure 5: The MPAS dataset with a Cartesian mapping.

When the "star" mapping is applied to more than three dimensions, the high dimensional space is projected down to three dimensions using a star coordinate system [34]. Figure 6 shows the projection of a six dimensional dataset into three dimensional space. Visualizing higher dimensional data using star coordinates should be avoided because the resulting plots quickly become unreadable. The star mapping of a six dimensional dataset in figure 6 is an example of a visual mapping that exceeds the capabilities of scatterplots in terms of information scalability.

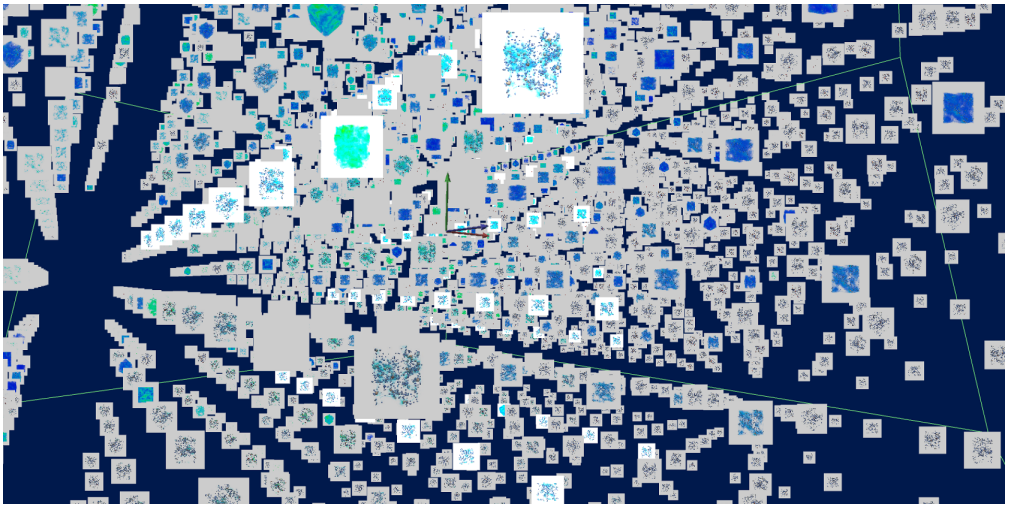


Figure 6: Projection of a six dimensional dataset into three dimensional space using a star mapping.

5.3 Spherical mapping

A more intuitive mapping is the spherical mapping. Here images are drawn at spherical coordinates according to their geographical parameters at a constant distance from the origin. The spherical mapping is shown in figure 7. The intuition behind this mapping is that every image is drawn at the point from which the view has been rendered in the simulation. The user is able to see both the current view and neighbor views. Browsing through views is done by rotating the view around the sphere of images using the mouse.

In GV we produce this mapping by issuing the command `THETAPHI all BY $theta * pi / 180, $phi * pi / 180, 4`. This applies a spherical coordinates mapping (i.e. "theta-phi" mapping) to all images by dataset dimensions longitude (i.e. camera angle "theta") and latitude (i.e. camera angle "phi"), converted to radians. The radius coordinate is set to 4 units. One unit corresponds to the size of an image. See appendix A for details of our visual mapping scripting language.

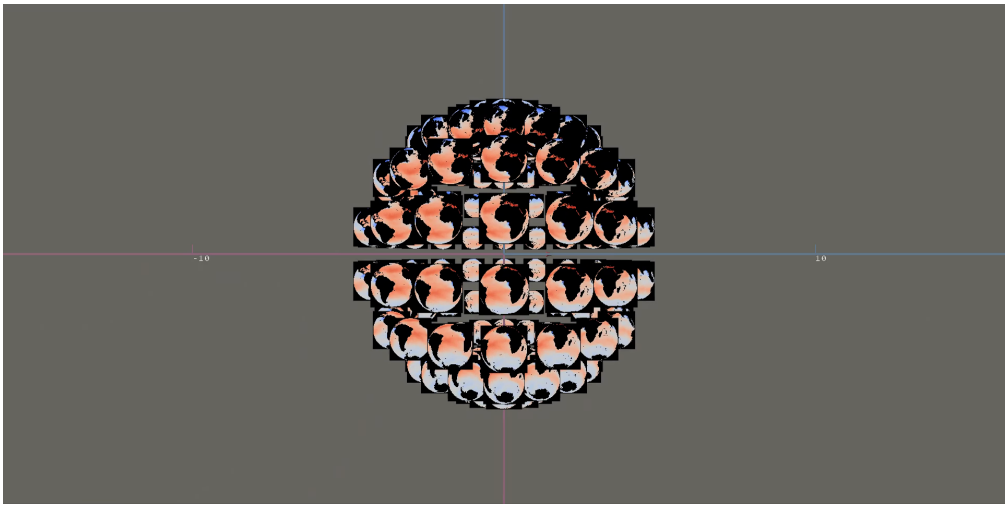


Figure 7: The MPAS dataset with a spherical mapping.

Another dimension (e.g. time) can be visualized by mapping this dimension to the x axis, to produce one sphere of images per time step. Figure 8 shows two time steps, visualized by combining the spherical mapping of spatial coordinates with a Cartesian mapping of the time dimension. In GV we produce this mapping by issuing the commands `THETAPHI all BY $theta * pi / 180, $phi * pi / 180, 4` and `X all BY #time * 10`

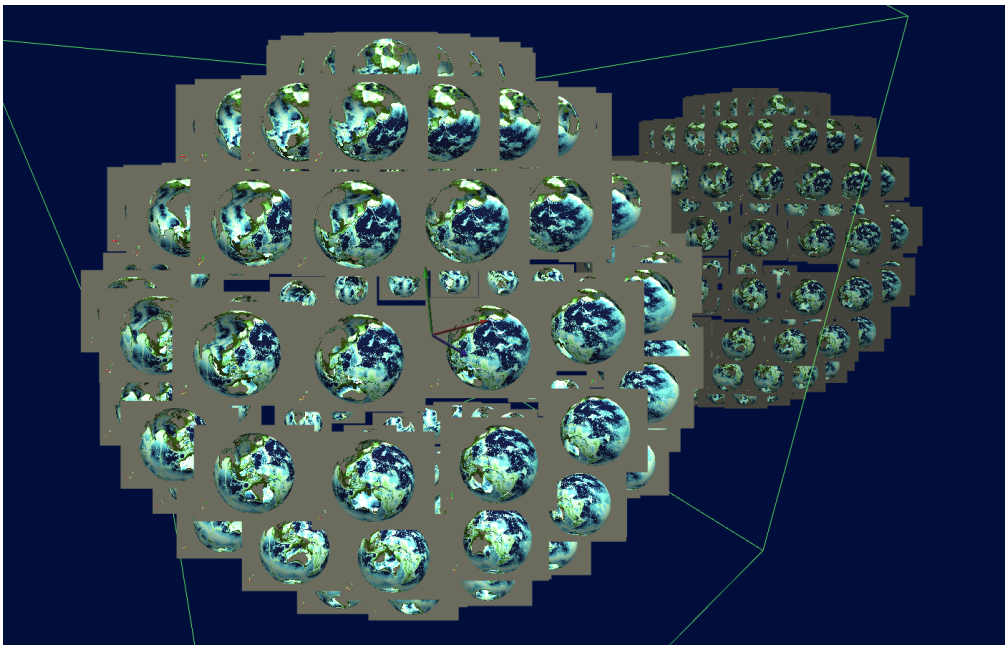


Figure 8: Two time steps rendered by combining spherical and Cartesian mapping.

5.4 Observer mapping

The most intuitive mapping for geographical coordinates is the observer mapping. Here we only show one view at a time. The displayed view is the view whose geographical coordinates most closely resemble the current virtual angle withing GV. The observer mapping is shown in figure 9. An obvious drawback from this mapping over previous mappings is that only one view is displayed at a time. This makes direct comparison between adjacent views more difficult.

Viewing a simulation with a very high spatial resolution using the observer mapping results in a viewing experience indistinguishable from observing a three dimensional object. However, storing many spatial views of the same object is infeasible, because of redundancy between adjacent views.

In GV we produce this mapping by issuing the command `LOOK all BY $theta * pi / 180, $phi * pi / 180`. This applies an observer mapping (i.e. "look" mapping) to all images by dataset dimensions longitude (i.e. camera angle "theta") and latitude (i.e. camera angle "phi"), converted to radians. See appendix A for details of our visual mapping scripting language.

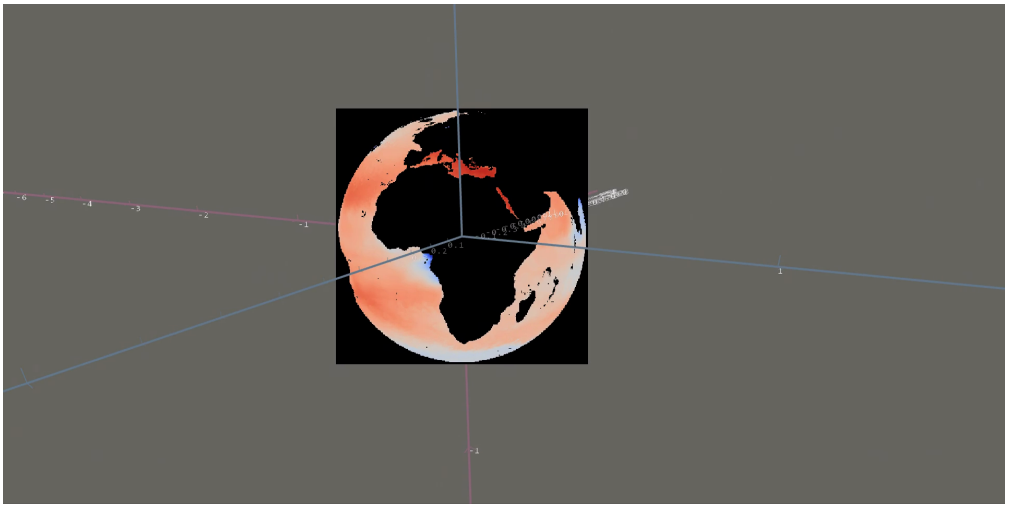


Figure 9: The MPAS dataset with an observer mapping.

5.5 Plot mapping

GV is based on a three dimensional scatterplot. We can map the two dimensional geographical coordinates of the MPAS dataset to two axes and use the third axis to visualize a third dataset dimension. This axis should be assigned to a sparse dimension to avoid cluttering the view. In figure 10 we use a Cartesian mapping of latitude and longitude to x and z axes and project the average salinity level over the visible pixels in each view to the y axis. Average salinity is a sparse dimension, because it consists of only one value per view.

The difference between a plot mapping and a three dimensional Cartesian mapping is that image locations are aided with purple lines in the plot mapping. The purple lines are important to aid cognitive perception of image locations in the three dimensional plot. They allow the user to perform spatial queries by evaluating lengths of lines, which is an elementary perceptual task for extracting information in the taxonomy of Cleveland and McGill [13].

In GV we produce this mapping by issuing the command `PLOT all BY $theta, $phi, 5 * $avgSalinity`. This applies a plot mapping to all images by dataset dimensions longitude (i.e. camera angle "theta"), latitude (i.e. camera angle "phi") and the value of average salinity times five units. One unit corresponds to the size of an image. See appendix A for details of our visual mapping scripting language.

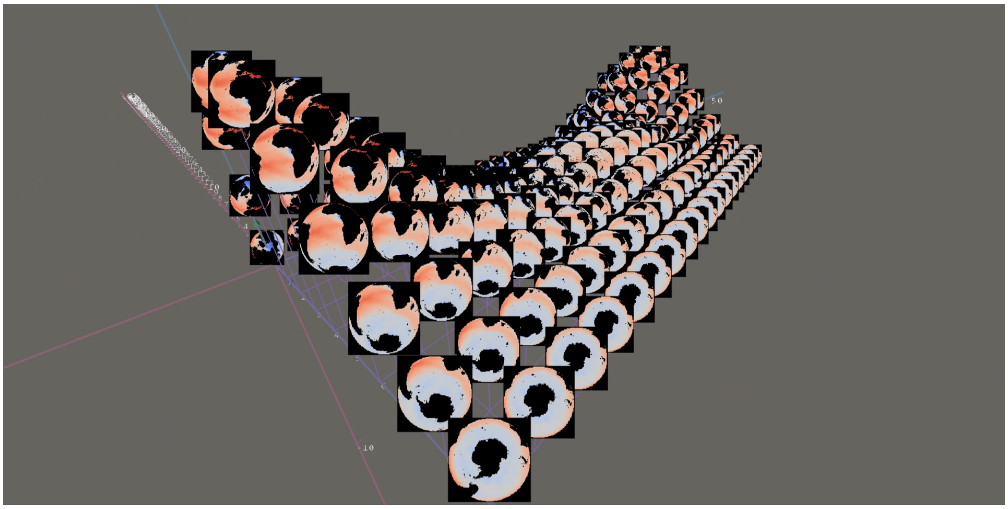


Figure 10: The MPAS dataset with a plot mapping.

5.6 Slider mapping

The slider mapping is an interactive mapping. It hides all but one image in a given dimension. In figure 11 we apply the slider mapping to the latitude coordinate of the plot mapping created in section 5.5. This results in a visualization of only one slice of the dataset. The slice is selected interactively through a slider control at the bottom of the screen.

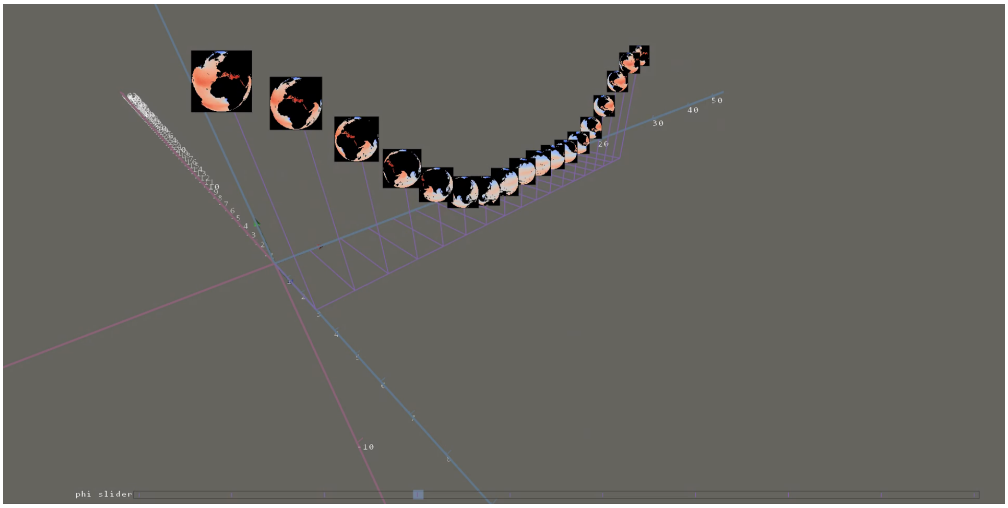
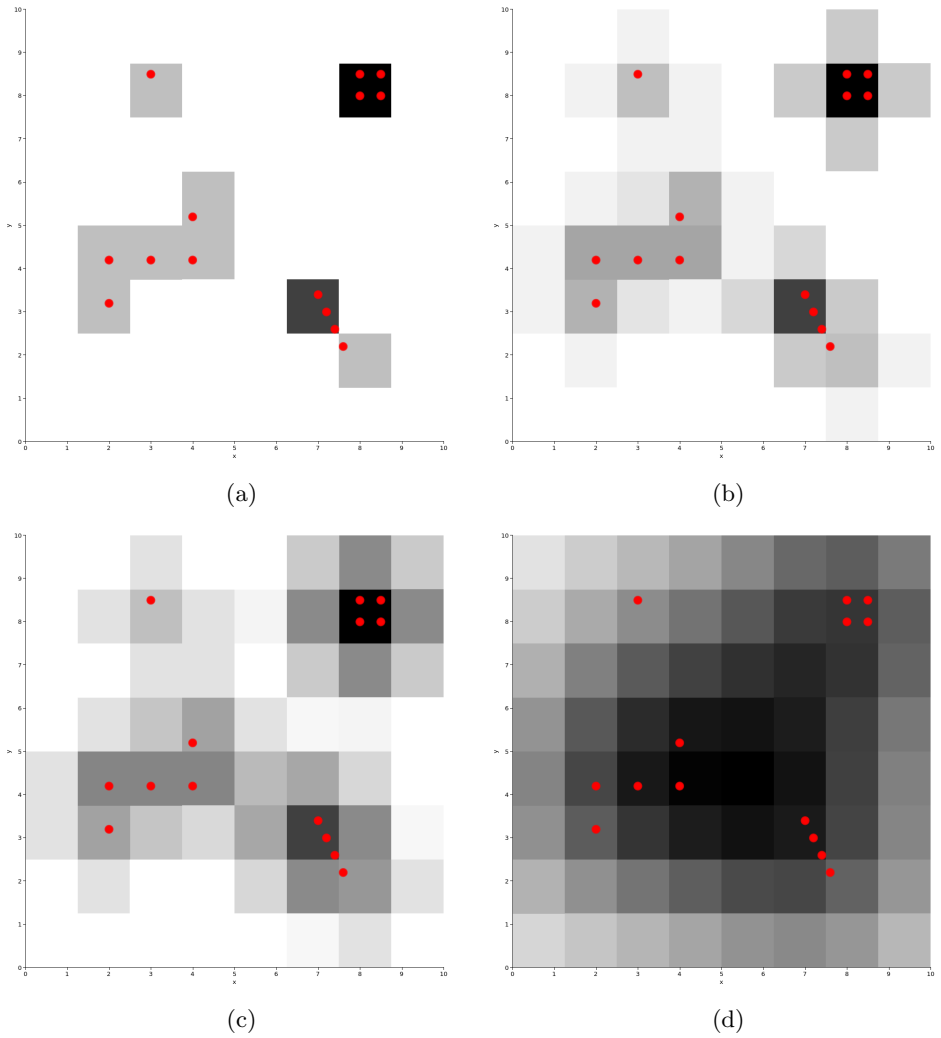


Figure 11: The MPAS dataset with a slider mapping.

6 Density maps

The point distribution of all points in a D-dimensional scatterplot can be expressed as a D-dimensional probability density function (PDF). To construct a PDF with a Gaussian kernel we have to make an assumption about the variance of the data. If the variance is assumed to be zero, then the PDF is one at every point's location and zero everywhere else. If the variance is assumed to be infinite, then the PDF is a constant.

For most datasets the exact (analytic) form of the PDF is unknown. A nonparametric way to estimate the PDF of a random variable is known as the kernel density estimate (KDE). We define the density map as a discretized KDE. Density maps with different variances are shown in figure 12. A density map with zero variance is equal to the histogram of the dataset (see figure 12a).



(a) (b) (c) (d)

Figure 12: Density maps on a two-dimensional dataset. The density maps are of size 8 by 8, with a variance of 0 (a), $8/\sqrt{200}$ (b), $8/\sqrt{100}$ (c) and $8/\sqrt{20}$ (d).

Density maps are used to analyze the dataset. For example, points lying within low point density can be considered outliers and density peaks can be considered cluster centers [33].

By using density maps to analyze a dataset, the runtime of the analysis is decoupled from the size of the dataset. It only depends on the size of the density map. We refer to the density map size as s. ICP uses two-dimensional density maps of size $s * s = 1024 * 1024$ pixel.

Optimized versions of DBSCAN and LOF compute a distance matrix to allow distance queries between any two points to be performed in constant time through a table lookup. The distance matrix of a dataset with n points is a symmetric n by n matrix. Like the distance matrix, the density map is an intermediate data structure, used to speed up operations like clustering and outlier detection. However, for large datasets the density map is typically much smaller than n by n. For example, the cell dataset is of size $n = 6077$ and in our performance benchmark (section 8.2) we evaluate datasets up to $n = 10^7$. All density maps computed for this thesis are of size $s \leq 1024$.
Density maps differ from distance matrices in the following ways:

- The density map is discretized, while distances in the distance matrix are exact. This has the disadvantage of classifying points closer than $\frac{1}{s}$ collectively, but it enables direct control over the performance/accuracy trade-off (see section 6.2).
- The density map is a projection of the dataset's PDF into 2-dimensional space. In theory a D-dimensional density map of size s^D could be computed, but the memory and performance advantages of an s^D sized density map over an n^2 sized distance matrix is diminished for $D > 2$. Instead we approach higher-dimensional computations by creating two-dimensional density maps between any two dimensions of the dataset. In other words, we are creating a density map for every unique two-dimensional scatterplot of the D-dimensional scatterplot matrix of the dataset. We then sample density within D-dimensional space by summing up samples of all two-dimensional density maps. This approach works well for tasks such as thumbnail placement (section 7.4), because the computed thumbnails are only viewed in any of the two-dimensional subspaces of the dataset. Because a scatterplot matrix is symmetric, we only create density matrices for the lower triangular scatterplot matrix, excluding the diagonal. The total memory size required by such a D-dimensional density map is $s^2 \frac{D*(D-1)}{2}$.
- The distance matrix preserves information about individual neighbors. This allows neighborhood queries to group neighborhoods into clusters or correlate a point's outlyingness with the average outlyingness in its neighborhood [8].
- The density map stores densities at any point, not just at the locations of data points. This information is utilized for placing labels (section 7.2) and generating samples (section 8.2). One could create a sparse density map with only point densities, should this extra information not be required.

6.1 Density map generation

To compute density, we need to define a measure of closeness between data points. Statistically, proximity is modeled with a Gaussian function $G(r)$. Subjective measures like "near" or "far" are quantified using the variance of the Gaussian. To model density, we use an unnormalized

Gaussian function (see equation 1).

$$G(r) = e^{-\frac{r^2}{2\sigma^2}} \tag{1}$$

The formal definition of a density map D is seen in equation 2. The density at location $\boldsymbol{x}$ within the density map is the linear combination of the measured proximities of each point $\boldsymbol{p_k}$ to $\boldsymbol{x}$. To naively compute the density at pixel $\boldsymbol{x}$ takes n operations. The runtime for naively computing a density map of s by s pixels is $O(ns^2)$.

$$D(\boldsymbol{x}) = \sum_{k=1}^{n} G(\|\boldsymbol{p_k} - \boldsymbol{x}\|) \tag{2}$$

We can compute the density map faster, by first computing a histogram of points $H(\boldsymbol{q})$ (see equation 3) and using the histogram to compute the density map (see equation 4). Computing a histogram takes $O(n)$, resulting in a total runtime of $O(n + s^4)$.

$$H(\boldsymbol{q}) = \sum_{k=1}^{n} \begin{cases} 1, & \text{if } \boldsymbol{p_k} = \boldsymbol{q} \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

$$D(\boldsymbol{x}) = \sum_{i=1}^{s} \sum_{j=1}^{s} H(\boldsymbol{q} = \begin{bmatrix} i \\ j \end{bmatrix}) * G(\|\boldsymbol{q} - \boldsymbol{x}\|) \tag{4}$$

Both equation 2 and equation 4 have similar runtimes when computing a density map of size $s = 1000$ from a dataset of $n = 1,000,000$ points ($\sim 10^{12}$ operations), but the inner sum of equation 4 can be skipped where $H(\boldsymbol{q}) = 0$. That means $O(n + s^4)$ is a worst case that applies for plots where each bin of the histogram contains points. In practice, such plots make little sense. We handle cases that would result in extreme runtimes by first estimating expected runtime and scaling down s if necessary (see section 6.2).

Another runtime optimization is implemented by restricting the sums of equation 4. The Gaussian function is an infinite function, but it's numerically bound to magnitudes above a certain threshold t. We can limit the range of $\boldsymbol{q}$ to only compute $H(\boldsymbol{q}) * G(|\boldsymbol{q} - \boldsymbol{x}|) \geq t$. Based on the unnormalized Gaussian function G in equation 1, the range of $\boldsymbol{q}$ that results in magnitudes $\geq t$ is a circular area around $\boldsymbol{q}$ with radius r_{max} according to equation 6).

$$H(\boldsymbol{q}) * G(\|\boldsymbol{q} - \boldsymbol{x}\|) \geq t \quad \text{where} \quad \|\boldsymbol{q} - \boldsymbol{x}\| \leq r_{max} \tag{5}$$

$$r_{max} = \sqrt{-2\sigma^2 \log \frac{t}{H(\boldsymbol{q})}} \tag{6}$$

Parameters for the density map computation are density map size s, variance σ^2 and threshold t. t is a constant based on floating point precision, s controls the performance/accuracy trade-off and σ^2 defines a scale for proximity.

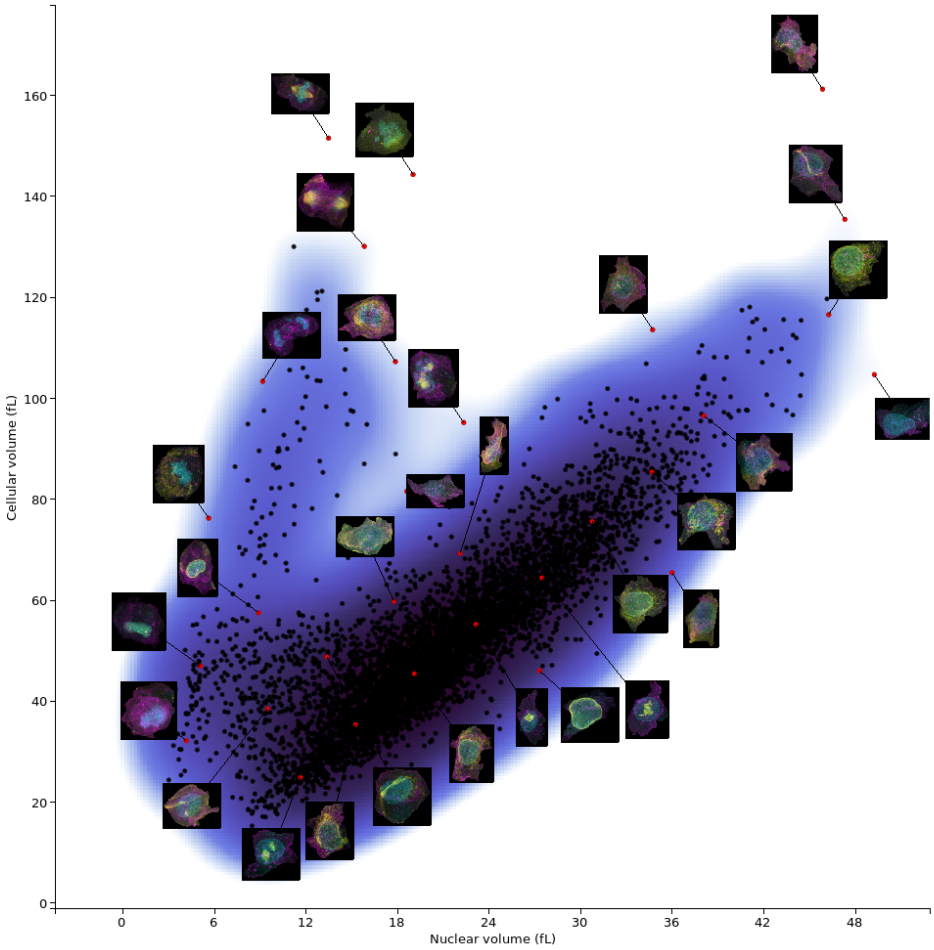


Figure 13: Density map based label placement on the cell dataset from the Allen Institute for Cell Science [32]. High densities are colormapped with a dark blue color.

6.2 Estimating density map generation runtime

Computing a density map of size s by s pixels takes $O(n+s^4)$ operations (see section 6.1). Since the runtime optimizations we introduced in section 6.1 depend on the distribution of data points, a fixed s results in vastly different runtimes for different datasets. A user is usually interested in the most accurate solution that can be computed in reasonable time, e.g. one second. To achieve this goal, we estimate an expected runtime in $O(s^2)$ time and reduce s if the expected runtime is too high, before computing the density map. The total runtime for computing a density map that can be computed in under one second is $O(n)$ (histogram generation) + $O(s^2)$ (runtime estimation) + $O(s^4)$ (density map computation) = $O(n+s^2+s^4)$. The total runtime for computing a density map that can only be computed in under one second if the density map size is reduced to $\hat{s} = \frac{s}{2}$ is $O(n)$ (histogram generation) + $O(s^2)$ (runtime estimation, returning $> 1sec$) + $O(\hat{s}^2)$ (runtime estimation, returning $\leq 1sec$) + $O(\hat{s}^4)$ (density map computation) =

$O(n + s^2 + \hat{s}^2 + \hat{s}^4)$.

In the following we will refer to the runtime estimation as estimation phase and to the density map computation as computation phase. In the estimation phase we iterate over all non-empty histogram bins and compute r_{max} (see equation 6). The number of computation phase iterations is estimated as the sum of all circular areas around non-empty histogram bins q with radius r_{max}. To compute a runtime estimate from the estimated number of iterations of the computation phase, we are measuring the runtime of the estimation phase.

Figure 14 shows runtimes and numbers of computation phase iterations measured on 15 different datasets with density map sizes between 256 by 256 and 1024 by 1024 pixels. Each of the 1900 gray points represents a randomly sized density map computed from a random dataset. Since the sampled runtimes have high variance, the graph shows 1900 averaged samples (purple points). Each purple point shows the average of $\frac{100*1024^2}{s^2}$ individual samples. We scale the number of individual samples by $\frac{1}{s^2}$ to make sure smaller density maps are measured over the same duration as larger density maps. This way we measure all density maps with the same precision. The y-axis shows the fraction of estimation phase and computation phase runtimes. We observe that estimation is between 40 and 150 times faster than computation and that there is an exponential relationship between the number of estimated computation phase iterations and the measured runtime fraction. We have fitted a logarithmic regression curve (shown in blue) using nonlinear optimization. The regression curve is a heuristic for estimating runtimes of any density map computation in $O(s^2)$ time. Since the heuristic is based on relative time measurements, it is independent of processor speed.

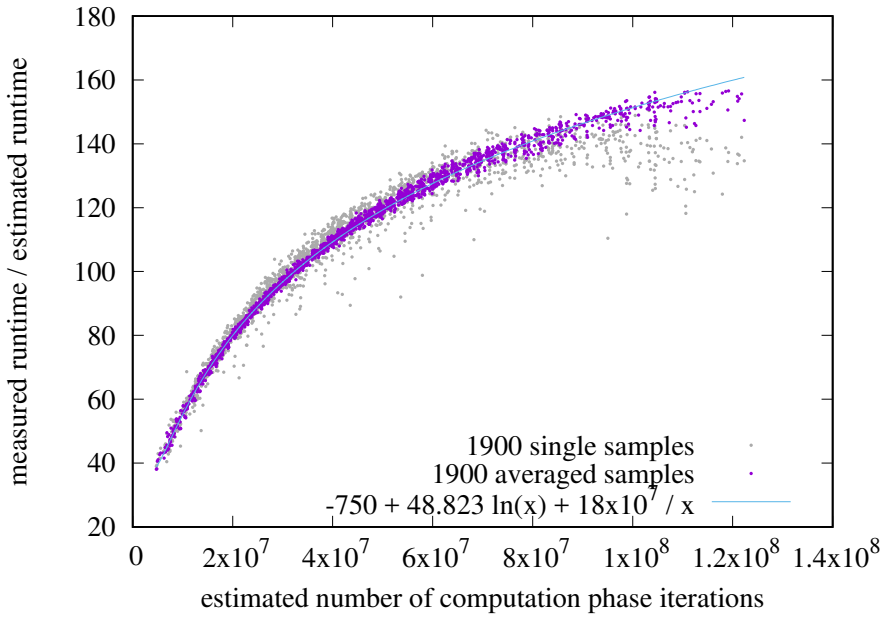


Figure 14: Measured runtime estimations for computing density maps of random size for random datasets: The x-axis shows the estimated number of computation phase iterations required to compute the density map. The y-axis shows the fraction of density map computation runtime over runtime estimation runtime. Each purple sample is an average of $\frac{100*1024^2}{s^2}$ individual samples. The blue line shows a logarithmic regression curve.

The algorithm above estimates runtimes for density maps that fully include all circular regions around non-empty histogram bins. Such density maps can be larger than s by s. We also implement a version of the density map generation algorithm, that computes only a density map the size of the bounding box around all points (s by s). For estimating the runtime of density maps, limited to size s by s, we need to extend the estimation phase to handle cropped circular areas (see figure 15). The computation of cropped circles changes the runtime of the estimation phase, which in turn results in different runtime fractions. Therefore, we use a different logarithmic regression curve to estimate cropped density maps.

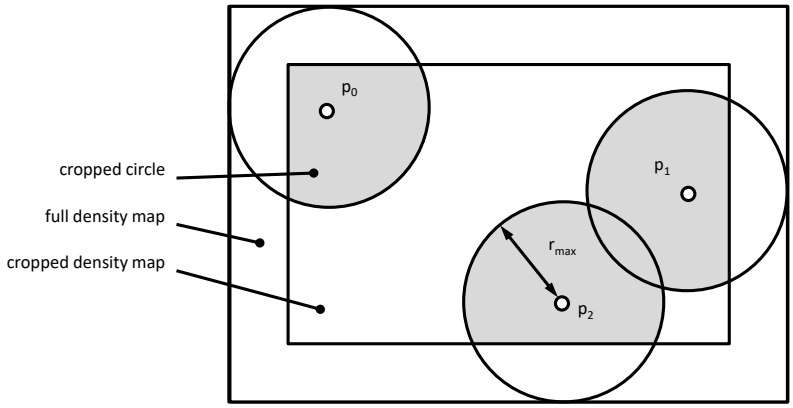


Figure 15: A depiction of the circular areas around points p_0, p_1 and p_2, that are iterated during the computation phase. When the full density map (outer rectangle) is computed, the areas around each point are full circles. When a cropped density map (inner rectangle) is computed, some circular areas are cropped.

6.3 Density map based clustering

Thresholded kernel density classification (tKDC) [20] classifies regions of higher density than a threshold t. Disjoint regions of higher density than t represent different clusters. Points that lie in regions of lower density than t are considered noise.

We implement tKDC using a flood fill method on the density map. In a first pass we find continuous regions of densities above t along the scanline through the density map. A second pass concatenates adjacent regions into clusters. Figure 16 shows the results of our method on the spiral dataset by Chang and Yeung [9]. Three detected clusters are shown in cyan, magenta and yellow. White regions have densities below t.

Our implementation of tKDC shows that density maps can be used to successfully cluster datasets. We will reserve more sophisticated implementations, like an adaption of CFSFDP for density maps, for future work.

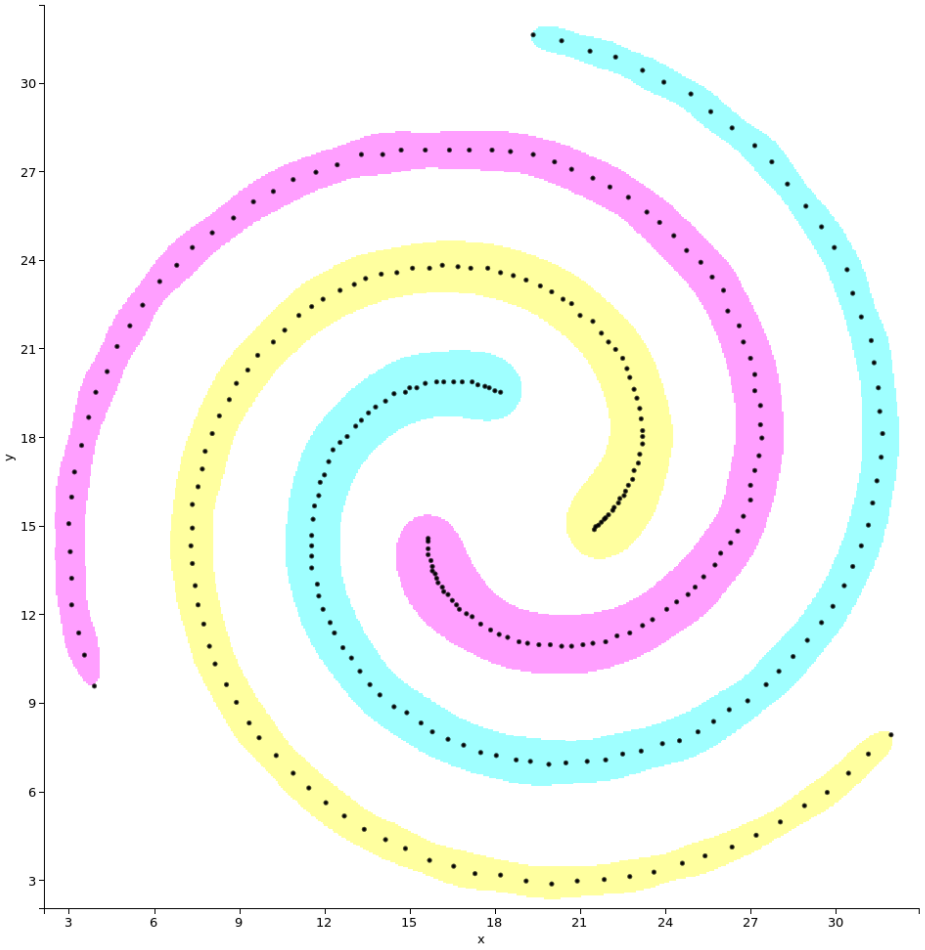


Figure 16: Density map based clustering on the spiral dataset by Chang and Yeung [9].

7 Labeling

7.1 Thumbnail selection

Rendering points instead of images saves a lot of screen real estate, but image information is crucial and needs to be exposed to the user in some way. Thumbnails are low resolution representations of individual images that are rendered either close to the plot or directly within. In this section we elaborate how to choose points to be tagged with thumbnails. The placement of these thumbnails will be covered in the next section.

How many thumbnails to expose is a subjective, data dependent problem. We will cover the selection of a reasonable number of thumbnails in our user study (section 8.3.

In a typical graph, a point is of particular importance if its location deviates from the majority of points. Such points are known as outliers. On the other hand, centers of high density regions are also considered important. The points closest to local maxima in the PDF are known as cluster centers. Both types of points characterize the dataset, making them ideal candidates to be annotated with thumbnails.

We detect such characteristic points by sorting all data points by their point density and selecting from the highest and lowest density points, depending on the requested outlier to inlier ratio. To get samples from different areas of the plot, we enforce a minimum Euclidean distance between samples. We set the minimum distance to 20% of the view diagonal and rerun the algorithm with half the minimum distance until the requested number of points can be extracted. After computing point density, each iteration computes in linear time.

Dense regions show visible patterns of selected samples with identical distance (see red points inside the high density region of figure 13), because our algorithm only probes for a constant minimum distance. These artifacts could be removed either by adding noise to the minimum distance or by using Monte Carlo sampling.

7.2 Thumbnail placement

A thumbnail should be placed near to its site without occluding important parts of the plot. Some users prefer keeping thumbnails very close to sites, while others prefer the advantage of unoccluded plots. We compare five different thumbnail placement strategies, in order of increasing thumbnail to site distance.

1. Adjacent placement places thumbnails directly at the site without regarding occlusions of other data points.
2. Density placement places thumbnails in regions of low point density within the plot, while also minimizing thumbnail to point distance.
3. Boundary placement arranges thumbnails along the rectangular border of the plot, while also minimizing thumbnail to point distance.
4. Numbered boundary placement uses the same placement strategy as boundary placement, but uses numbers instead of leader lines to connect thumbnails.
5. External placement shows thumbnails in a list outside the plot. Thumbnails are associated using numbering.

In the following we elaborate placement strategies in detail.

7.3 Adjacent placement

Placing static thumbnails directly at the site results in significant occlusions of data points (see figure 17). However, for a dynamic preview, showing thumbnails directly at the data point the mouse is hovering over is certainly the most intuitive solution. This is an established metaphor for showing textual hints in desktop applications in the form of tool tips.

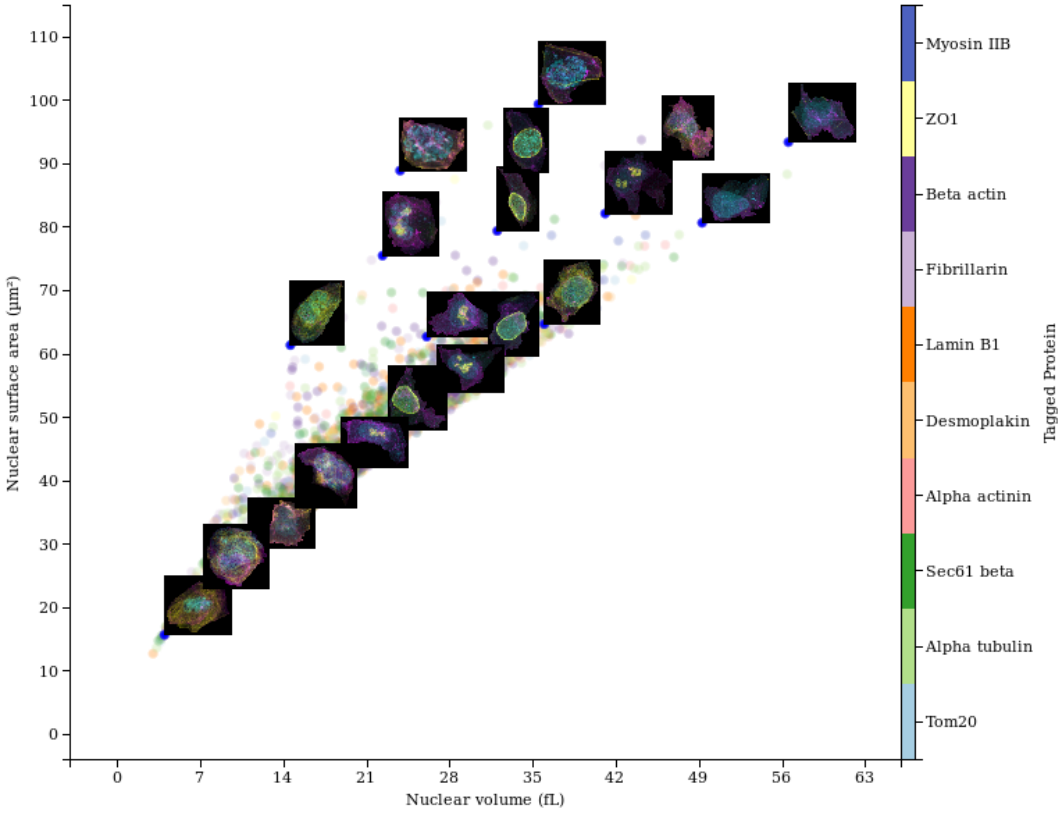


Figure 17: Adjacent placement

7.4 Density placement

The problem of finding locations within the plot that occlude as little as possible data points can be formulated as an optimization problem. The optimal location of a thumbnail is a location within low point density that is close to the site. We combine point density and site proximity at a point x into a cost function $C(x)$ according to equation 7. We scale densities to the same range as x to avoid bias. Scaled densities are denoted as $\hat{D}$. The range of $\hat{D}$ is zero to density map size $(0...s)$. The factor 10 in equation 7 controls the trade-off between density and site proximity. Alternatively the trade-off can be controlled by changing the variance of the density map.

$$C(x) = 10 * \hat{D}(x) + |r - x|^2 \tag{7}$$

We use an exhaustive search to find a location x that yields minimum cost $C(x)$ by simply checking every pixel of the density map. The runtime of exhaustive search is $O(s^2)$. There are faster ways of finding a location of low cost, e.g. using a descent method [12], but we decided the exhaustive search is fast enough for reasonable numbers of thumbnails on a density map of size $s = 1024$.

To avoid overlapping thumbnails, we keep a boolean stencil of same size as the density map and restrict the search to unmasked regions of the stencil. After placing a thumbnail, we mask the region covered by that thumbnail in the stencil. An alternative approach would be to store thumbnail regions in a data structure optimized for fast spacial query.

After every thumbnail is placed, we check every pair of thumbnails for intersections between their leader lines. If an intersection is found, we swap the thumbnails.

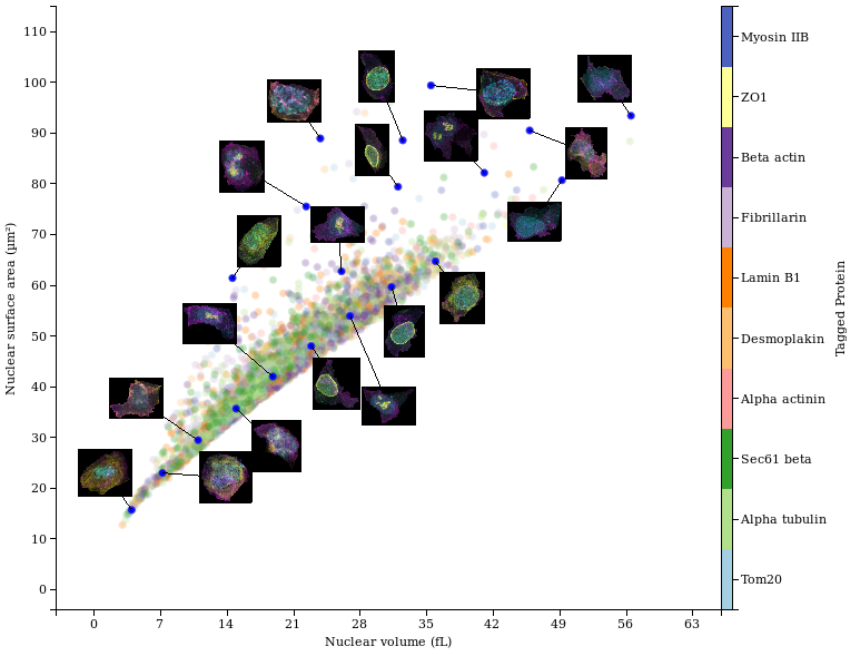


Figure 18: Density placement

7.5 Boundary placement

Boundary placement keeps thumbnails within the plot without occluding data points. Distributing labels evenly along the Axis Aligned Bounding Box (AABB) of a plot has been implemented by Bekos et al. [5]. We implement boundary placement that allows thumbnails to lie anywhere on the AABB.

First, we project sites to the AABB using a parallel projection along the second Principal Component (PC) vector of the site distribution. The first PC points towards the largest variance of the distribution. To spread out thumbnails over a wide area we project them along a direction orthogonal to the largest variance. This direction by definition is the second PC of the distribution. The projection of the site along the second PC intersects twice with the AABB, once before and once after the site. We place each thumbnail at the closest of the two intersections.

After the projection we move thumbnails along the AABB to eliminate overlaps using the ReArrange algorithm by Garderen et al. [37]. In a final pass we resolve intersections between leader lines by swapping thumbnails whose lines intersect. This pass is the same as the one we use for density placement.

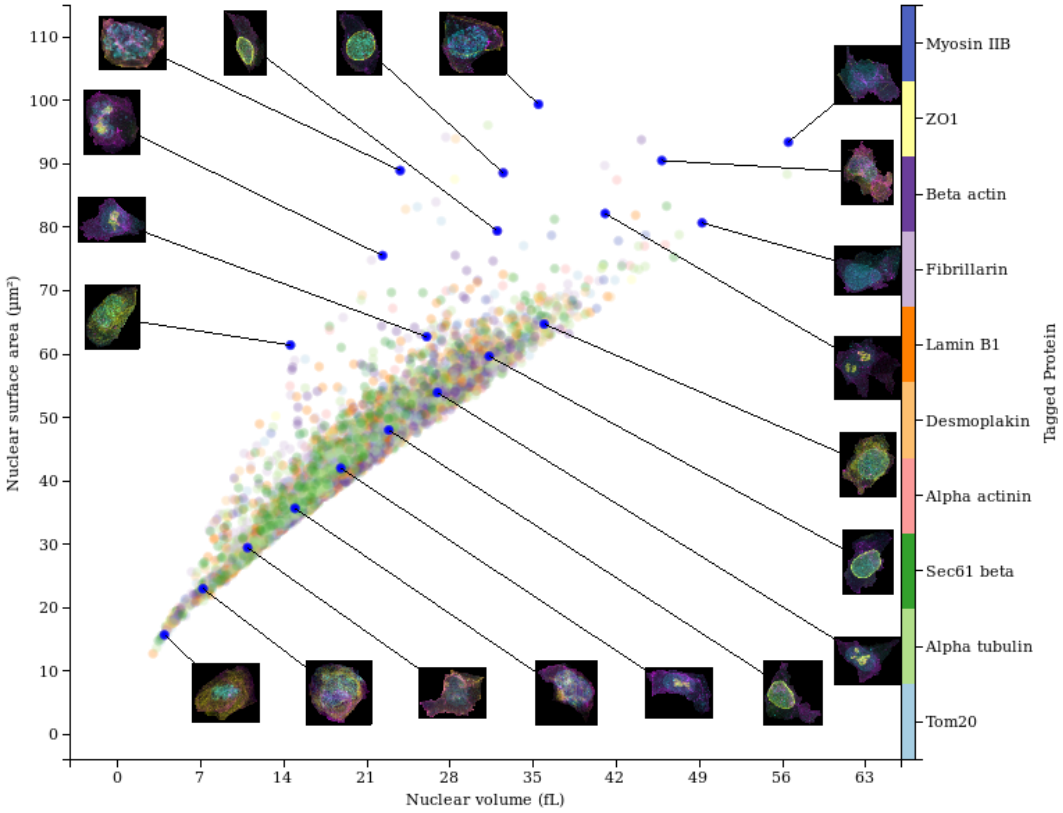


Figure 19: Boundary placement

7.6 Numbered Boundary placement

To avoid the additional clutter of long leader lines in boundary placement, thumbnails can be numbered instead. Cognitively, linking thumbnails via their numbers is harder than via leader lines. To aid this process, we highlight the corresponding number label whenever a user hovers over a thumbnail with the mouse cursor.

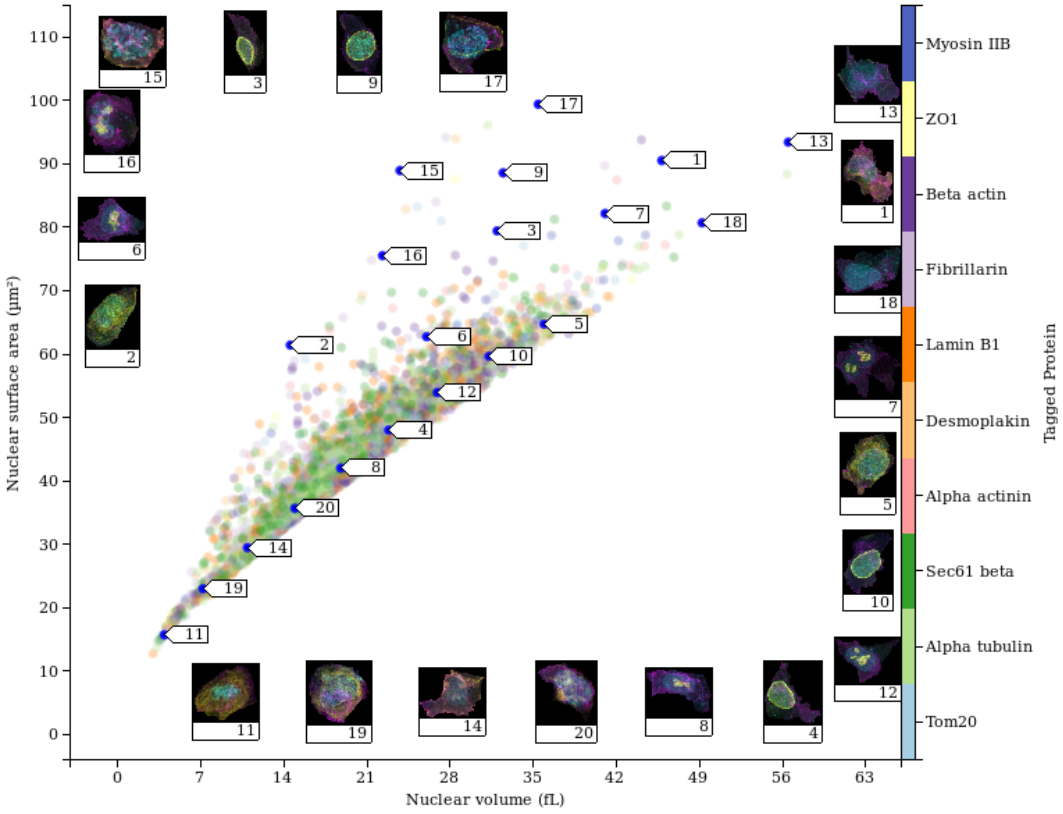


Figure 20: Boundary placement with labels

7.7 External placement

The problem of occluding important parts of the plot is avoided by placing thumbnails outside the plot. Externally placed thumbnails can have arbitrary size. The biggest drawback of external placement is that thumbnails cannot be linked with leader lines, instead external placement relies on numbering thumbnails. Cognitively, linking thumbnails via their numbers is even harder for external placement then for numbered boundary placement, because thumbnails aren't ordered along their sites.

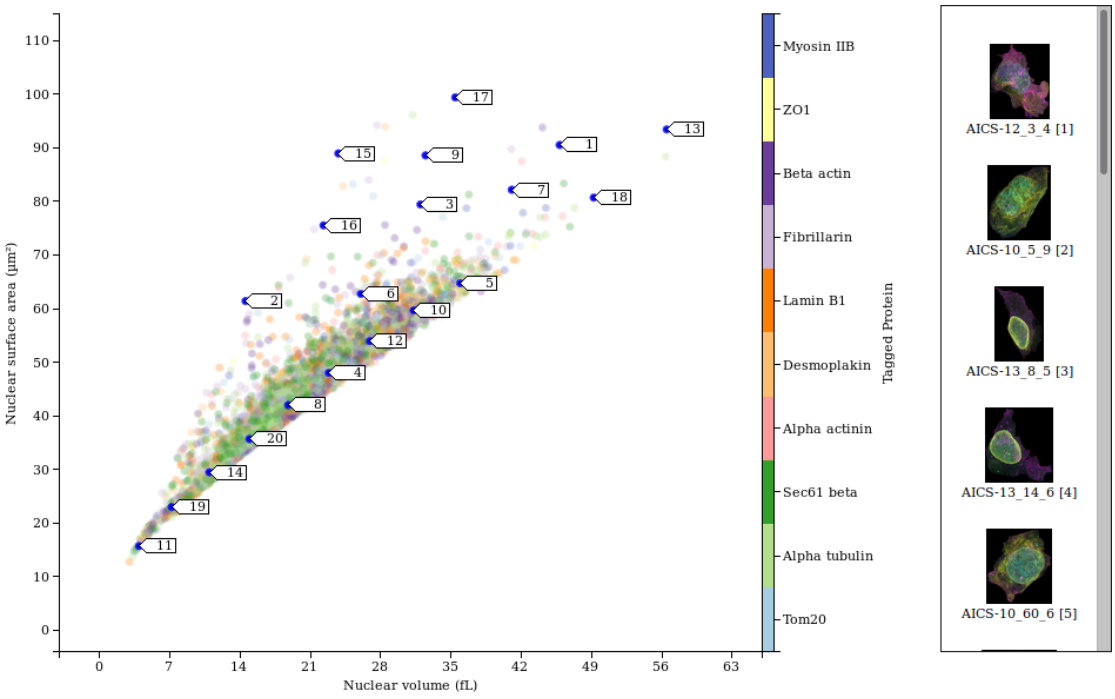


Figure 21: External placement

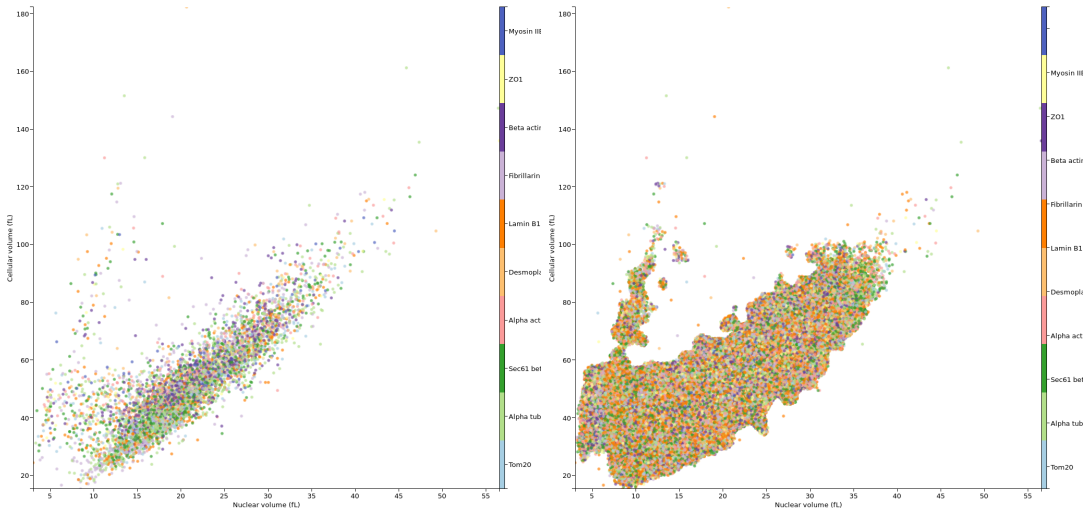
8 Evaluation

8.1 Cell dataset scalability

One key requirement of ICP is to ensure that the cell dataset can still be rendered after many more cells have been added in the future. To test performance scalability with an increasing cell dataset size, we created artificial datasets of larger sizes, but with identical point distribution.

We created those artificial datsets by adding identically distributed points to the cell dataset using the density map. The density map approximates the probability distribution of the dataset. By rejection sampling the density map, we can draw points from this distribution. The variance of the density map controls the deviation of generated points from original data points.

On our desktop machine [1] we achieved interactive frame rates for an artificial dataset with 100 times the size of the original cell dataset (see figure 22b). Our performance evaluation (section 8.2) confirms this result.



(a) (b)

Figure 22: The cell dataset with (a) the original 6077 data points and (b) 607700 data points sampled from the density map of the original dataset.

[1] CPU: Intel 3.40GHz Core i7-2600K, GPU: AMD Radeon RX 480, operating system: Ubuntu 17.04, browser: Firefox 55.0.1

8.2 Performance

The plot below shows frame rates for drawing between 1000 and 10 million two-dimensional points. Each sample in the plot represents average frame rate over a ten second interval with a constant number of points. The backbuffer size is 1024 by 1024 pixels. Thumbnails are disabled.

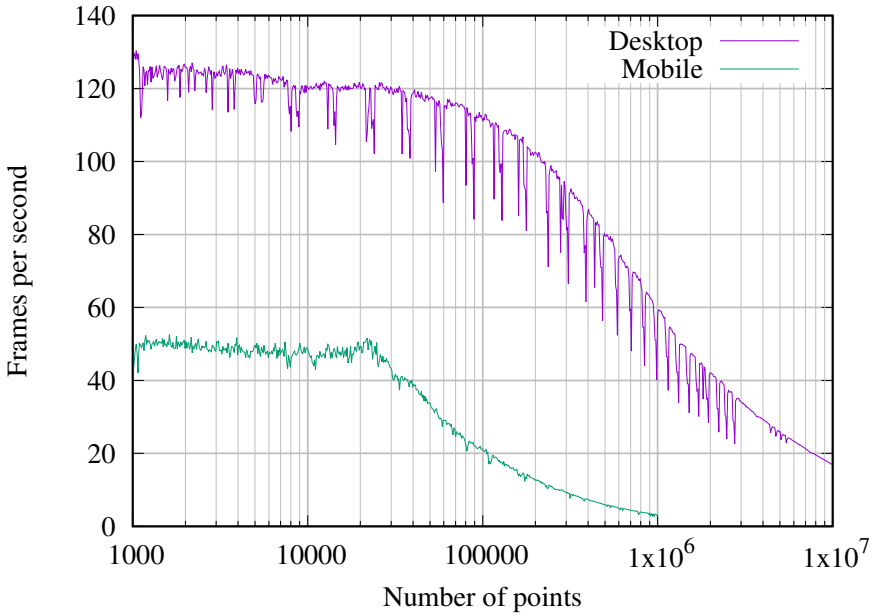


Figure 23: frame rates when running ICP in a canvas of size 1024 by 1024 pixels on a desktop machine (CPU: Intel 3.40GHz Core i7-2600K, GPU: AMD Radeon RX 480, operating system: Ubuntu 17.04, browser: Firefox 55.0.1) and on a cell phone (phone: HTC Desire 626s, browser: Chrome 59.0.3071).

We only evaluated cell phone performance up to one million points due to limitations in available video memory. Figure 23 shows that frame rates of small datasets are bound by an upper limit of about 120fps on desktop and about 50fps on mobile. This may seem unusual to people familiar with OpenGL benchmarks. These upper limits are regulated by the browser. Unlike OpenGL-based desktop applications, web applications don't have direct control over the refresh rate. Instead, a web application requests a new frame after the current frame is finished. It is then up to the browser to decide when to start the next frame. We believe that the spikes of the desktop benchmark are also caused by inconsistent browser scheduling.

We can summarize the following conclusions (omitting the spikes):

- Maximum number of points at 60fps on a desktop machine: 1 million
- Maximum number of points at 25fps on a desktop machine: 5 million
- Maximum number of points at 25fps on a cell phone: 74000
- Maximum number of points at 20fps on a desktop machine: 7.5 million
- Maximum number of points at 20fps on a cell phone: 103000

8.2.1 Comparison with other charting libraries

Most currently available charting libraries are optimized to render very large datasets. Plotly provides a benchmark of load times online [24]. They report a load time of 27 seconds for loading a scatterplot with 300,000 points using Chrome and more than 30 seconds using Firefox or Internet Explorer. When we tried to load the same dataset on our desktop machine, Chrome crashed and Firefox took several minutes to load. After the dataset was loaded, we tried to interact with the plot. Plotly achieved less than one frame per 10 seconds. ICP loads a dataset with 1 million points in under a second and renders the loaded plot at 60 frames per second.

In July 2015 Highcharts posted about a newly released boost.js module that allows them to create a chart with one million scatter points in less than 200 milliseconds. They write that 200 milliseconds is the time the chart takes to initialize, and after that the points are rendered in asynchronous chunks for about two seconds [22]. The total load time of about two seconds is longer than our initial load time. Highcharts did not post any statistics about interactivity of their one million point plot.

8.3 User Study

We conducted a qualitative user study to evaluate the layout and different thumbnail placement strategies of ICP. The main goal of this user study are the following:

1. To find the best visualization parameters, such as color scheme, point size and whether to show point density.
2. To find the best thumbnail placement strategy.
3. To find good initial values for the number and distribution of thumbnails.
4. To evaluate which controls used in the user study could become part of the user interface in the cell viewer web page.

For the first two goals, we let each subject design a view style and a thumbnail placement strategy. For the third goal, we let subjects try out different numbers of thumbnails and different inlier-to-outlier ratios using slider controls. For the fourth goal, we ask subjects about the usefulness of individual controls.

8.3.1 Subjects

6 subjects took part in our user study. They all had prior experience with scatterplots. Five subjects use scatterplots often, one subject uses them occasionally (see figure 25. 3 subjects primarily use scatterplots for presentation, 2 subjects use them primarily for exploration and one subject uses scatterplots for both presentation and exploration equally. All but one subject had previously worked with the dataset of the user study.

We performed six user studies using video conferences within two weeks. Each user study took 45 minutes. During half of the sessions we were talking to the subject alone. During the other sessions there was one other person listening. We asked subjects to verbally express the thinking process behind their decisions. A summary of what we learned from their verbal responses is shown in section 8.3.3.

8.3.2 Setup

The first part of our user study focused on visualization parameters. We did not expose any thumbnails until part 2. After asking our subjects about prior experience with scatterplots and the cell dataset, we gave each subject the opportunity to get familiar with the cell dataset by letting them browse through the dimensions of the dataset. For the remainder of the study we fix the dimensions to Nuclear Volume on the x-axis, Nuclear Surface Area on the y-axis and Tagged Protein on the color-dimension.

Then we let each subject pick one of four predefined view styles (see figure 24). We purposely implemented only minor differences between the predefined styles to motivate our subjects to pay attention to details. Style 1 has circular points on a white background. Style two differs from style 1 by having Gaussian points instead of circles. Style 3 differs from style 2 by having a dark background. Style 4 has small circular points and a grayscale visualization of point density in the background. The point density was computed using a density map.

After a subject picked a predefined view style, we let them improve the view style by exposing the following controls:

- Color schema (white or dark background)
- Point color (discreet colormap, continuous colormap, black, white or blue)
- Point shape (circle, Gaussian, cross or diamond)
- Point size (between 1 and 100 pixel)
- Point opacity (invisible to fully opaque in steps of 10%)
- Density visualization (on or off)

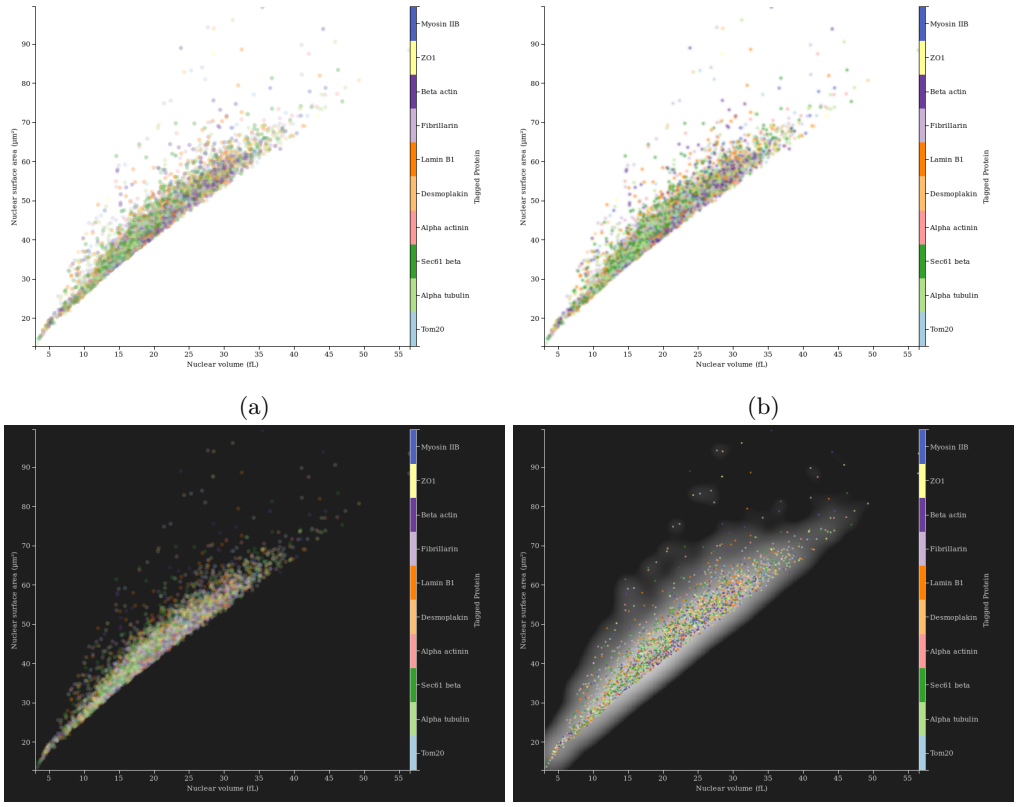
The second part of the user study focused on thumbnails. We first allowed our subjects to get familiar with some of the images in the cell dataset, by showing the cell image whenever a subject hovers over a data point with the mouse cursor.

Then we let each subject pick one of five predefined thumbnail placement strategies. The list of strategies is shown in chapter 7.2.

After a subject picked a predefined thumbnail placement strategy, we let them choose the number and distribution of thumbnails using sliders. The number of thumbnails could be chosen between 2 and 50, with a default of 20. The distribution could be chosen in form of an outlier to inlier ratio between 100% outliers and 100% inliers with a default of 50%/50%.

If a subject selected either density placement or boundary placement, we allowed them to optionally fine-tune thumbnail locations by moving them with the mouse.

We finished the user study by discussing overall user experience.



(a) (b)

(c) (d)

Figure 24: 4 different view styles presented to the subjects. a) Style 1 has circular points on a white background. b) Style 2 has Gaussian points. c) Style 3 has Gaussian points on a dark background. d) Style 4 has small circular points, with a grayscale density map in the background.

8.3.3 Results

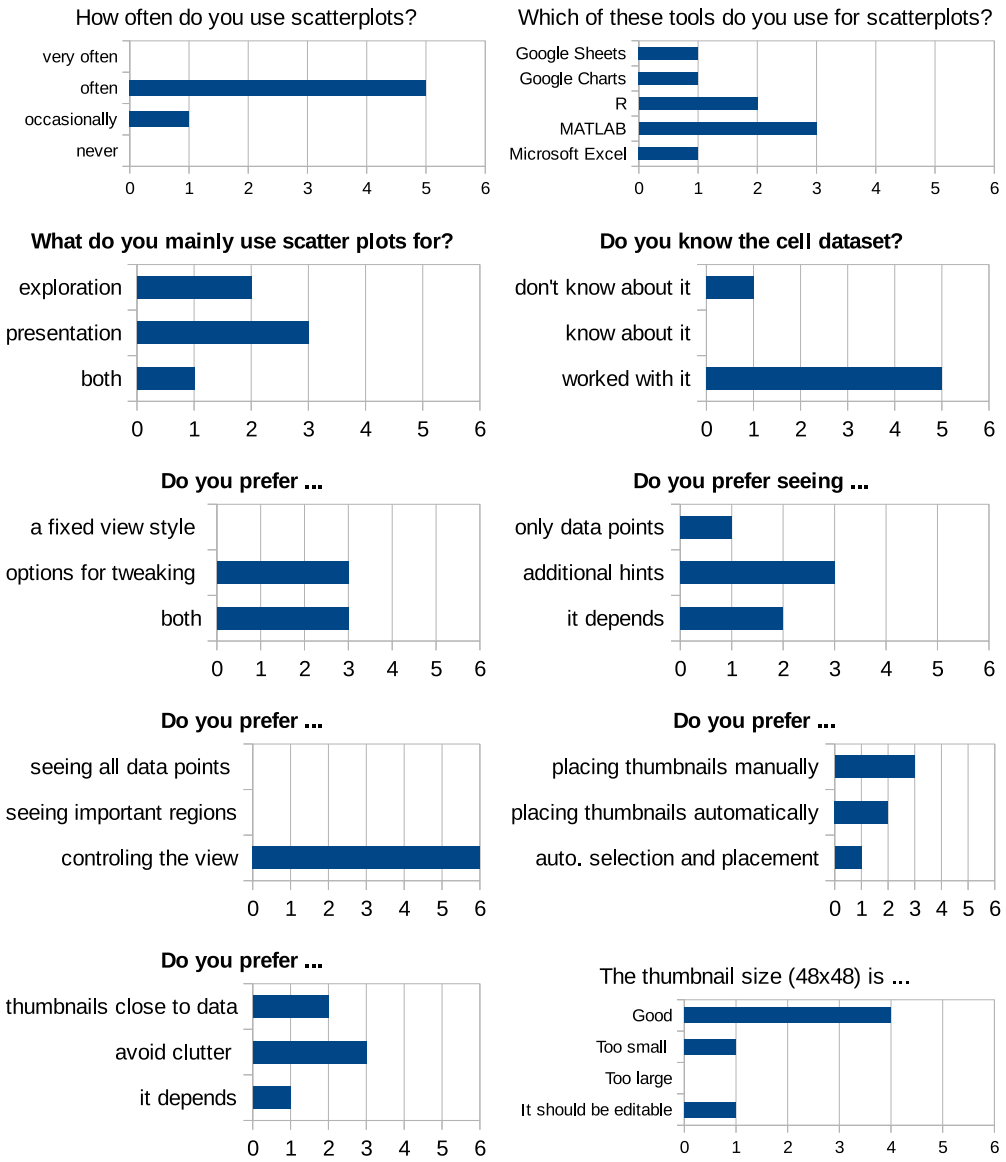


Figure 25: User study questionnaire results.

We summarize the scatterplot visualization parameters based on the choices that the subjects made when we asked them to tune parameters for the best user experience. Half of the subjects selected a point size of 7 pixels. The other choices were 5, 6 and 17. Three subjects chose a point opacity of 70%. Two subjects chose 50% and one subject chose fully opaque points. Four out of six subjects chose to color points with a discrete colormap. The other two subjects selected a continuous colormap. The majority of subjects chose not to show a grayscale density visualization behind the data points. All but one subject preferred the higher contrast of the

color scheme with a dark background. We conclude that the ideal visualization of this dataset should show circular points with a diameter of 7 pixels and an opacity of 70%, drawn using a discrete colormap on a dark background with no density visualization (see figure 26).

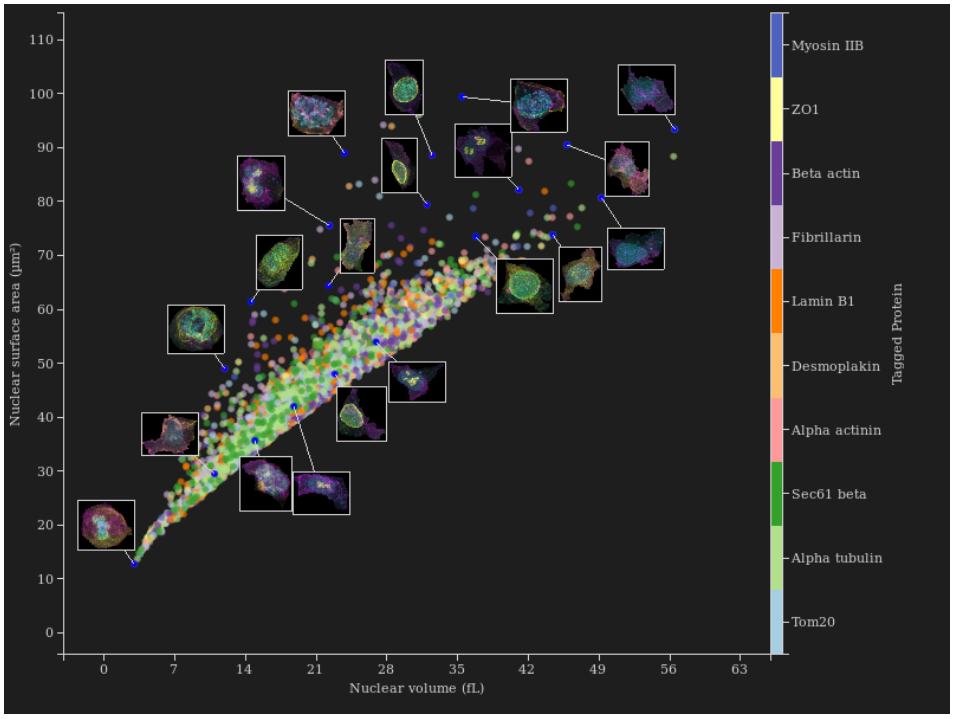
Half of the subjects picked density placement as the best thumbnail placement strategy. The other half picked boundary placement. Therefore, we deem both strategies equally suitable (see figure 26).

The preferred choice of outlier ratio for automatic thumbnail selection varied between subjects from 50% to 100% outliers. Even though two subjects chose 100% outliers, we decided to pick the average choice (75% outliers, see figure 26a) as best choice, because we believe many users would consider a selection that completely disregards clusters unsatisfactory. The average number of displayed thumbnails between all subjects is 21.67. We round this number to 20 for figure 26.

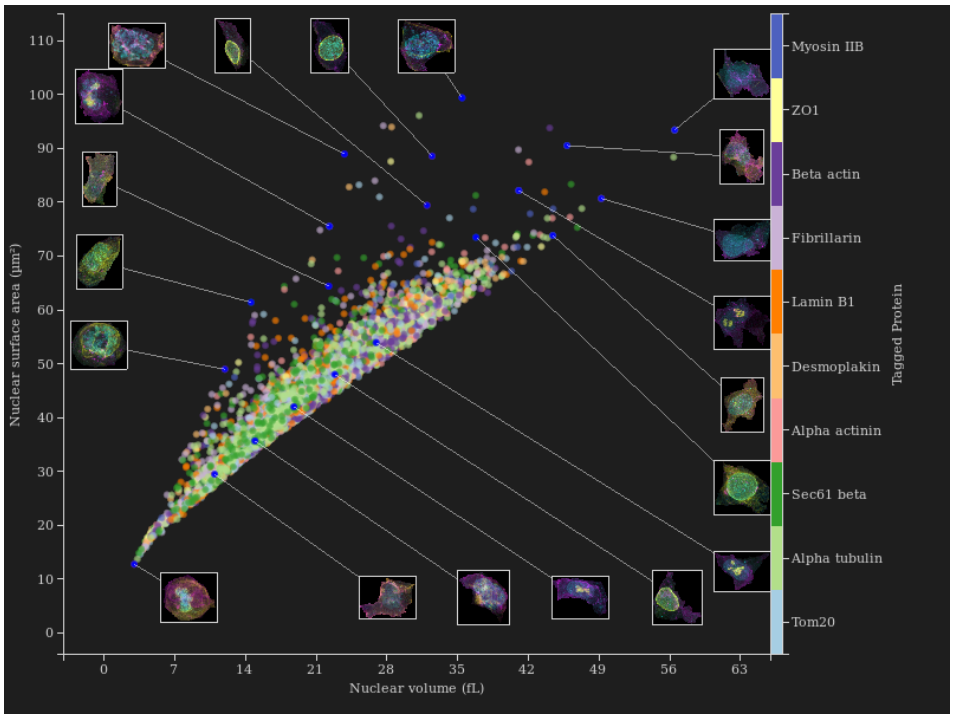
Following user suggestions, we reduced the contrast of thumbnail leader lines for boundary placement. These lines are now gray, instead of white (see 26b).

All subjects state that they expect a good visualization tool be customizable in as many as possible aspects, while providing good initial values derived from the dataset. For example, all subjects like having the software pick thumbnails automatically based on a given number and distribution of thumbnails, but half of them commented that they want to have the option to add or remove individual thumbnails by hand. One subject suggested an interactive approach, where only three thumbnails are exposed in the beginning and users could add more thumbnails as they explore the plotted data in more detail.

According to the subjects, most controls, like point shape and size, should only be exposed in a separate menu. However, controls for density visualization and number of thumbnails should be exposed in the main interface. Some subjects did not understand what the thumbnail distribution slider is used for until they started adjusting it. We got the suggestion by one subject to expose it only as a set of options, not as a slider. Examples of such options are "Random thumbnails" and "More thumbnails for outliers".



(a)



(b)

Figure 26: Two visualizations derived from the combined choices of all subjects.

9 Conclusion

We have developed two scatterplot based image database viewers that emphasize the potential of this venerable visualization tool for modern image database visualizations.

9.1 Global View

GV enables the user to interactively define custom visual mappings using our proposed visual mapping scripting language. Once a suitable visual mapping for a particular data domain has been found, GV can be used as a framework to quickly design an image viewer for this type of datasets.

The texture streaming algorithm monitors allocated video memory and dynamically allocates and deallocates memory in an unsupervised fashion. Our algorithm has great potential as a backend resource manager for image viewers. Even though we run texture streaming on a background thread, large datasets can still cause significant drops in the frame rate. These latencies are caused by texture load operations, where image data is transfered from main memory to video memory. We block the render thread during load operations, because OpenGL does not support concurrent access to the graphics accelerator. The recently introduced Vulkan API is the first cross platform graphics API that supports multithreading. We see the transition from OpenGL to Vulkan as important future work to remove the implications of load operations on the render thread.

GVs focus on performance scalability could allow this visualization to be displayed on a high resolution video wall. This would allow groups of scientists to study and present entire image databases without requiring any interaction. The concept of using large viewing environments to present both global and local information simultaneously is known as hybrid-image visualization [23]. A hybrid-image visualization represents viewing distance dependent overview+detail views in a single image. Isenberg et al. provide perceptual background for using hybrid-images to visualize data in [23].

9.2 Interactive Cell Plot

ICP is a very fast WebGL based scatterplot library. It enables scientists to present large high-dimensional datasets on their web page without compromising user experience.

WebGL is an underrepresented technology for browser-space plotting libraries. The performance evaluation of ICP opens the door for browser based visualization by showing how many points can be rendered in a browser window with current technology. We expect these numbers to increase as WebGL 2.0 gets implemented into more browsers, because we faced limitations that would allow more efficient code with WebGL 2.0.

The most significant limitation we faced was the missing support for geometry shaders in WebGL. While quadratic shapes can be drawn as points, drawing shapes with an aspect ratio other than $1 : 1$ requires the use of additional geometry. To draw such shapes as data points without a geometry shader one has two options. The first option is to render more vertices and transform the extra vertices into the point shapes using the vertex shader. This is not feasible for ICP, because the rendered data tables are too large to keep multiple copies in memory. The second options is to use the 'instancing' extension. With this extension we can construct a point shape and use instancing to replicate that shape at every data location. This technique, however, doesn't support the use of an index buffer to define what instances should be skipped. We

implemented this scenario by rendering continuous arrays of data points with one non-indexed draw call each. In the worst case scenario, a user would choose to render only every second data point. ICP would then have to issue $\frac{n}{2}$ draw calls of a single data point each. At the time of this writing 31% of browsers support WebGL 2.0 [7].

9.3 Visual mapping

Using GV we have identified three different visual mappings to visualize the spatial views of the MPAS dataset. We have outlined advantages and disadvantages of the different mappings. A mapping should be chosen by trading of the number of viewed images with the viewing experience. The Cartesian mapping shows all spacial views at once, but it is the most unintuitive to interpret. The spherical mapping shows a local neighborhood of images in a more intuitive way. The observer mapping is the most intuitive, but it only shows one image at a time.

9.4 Density maps

Density maps compute important statistics for large datasets. Our algorithm for estimating density map generation runtime gives the user full control over the performance-accuracy-trade-off. We have shown that density maps can be used to compute clusters and ideal locations for thumbnails, as well as generate new points from the dataset's distribution.

We believe density maps have the potential of becoming an important tool for future developments in the area of automated visualization design. Their potential will improve as more density-based algorithms are adapted to use density maps.

9.5 Labeling

All subjects of our user study like the number-of-thumbnails and inlier-to-outlier-ratio controls as a semi-automatic way to select characteristic points for labeling. Both boundary placement and density placement provide significantly less data occlusion than adjacent placement, which is currently used for the interactive plotting tool [18].

Our density-based thumbnail placement algorithm only considers one thumbnail at a time. Results can be further improved by optimizing all thumbnail locations at once. The $O(s^2)$ runtime of the implemented exhaustive search can be improved by implementing one of the optimization methods presented by Christensen, Marks and Shieber [12].

References

[1] James Ahrens, Dave E. DeMarle, and David H. Rogers. Cinema. `http://cinemascience.org/`.

[2] Utkarsh Ayachit, Andrew Bauer, Berk Geveci, Patrick O'Leary, Kenneth Moreland, Nathan Fabian, and Jeffrey Mauldin. Paraview catalyst: Enabling in situ data analysis and visualization. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 25–29. ACM, 2015.

[3] Lyn Bartram and Colin Ware. Filtering and brushing with motion. *Information Visualization*, 1(1):66–79, 2002.

[4] Benjamin B. Bederson and James D. Hollan. Pad++: A zoomable graphical interface. In *Proceedings of ACM Human Factors in Computing Systems Conference Companion*, 1994.

[5] Michael A. Bekos, Michael Kaufmann, Antonios Symvonis, and Alexander Wolff. Boundary labeling: Models and efficient algorithms for rectangular maps. *Comput. Geom. Theory Appl.*, 36(3):215–236, April 2007.

[6] Enrico Bertini, Andrada Tatu, and Daniel Keim. Quality metrics in high-dimensional data visualization: An overview and systematization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2203–2212, 2011.

[7] Florian Bösch. WebGL stats: WebGL 2.0. `https://webglstats.com/webgl2`.

[8] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. LOF: Identifying density-based local outliers. *SIGMOD Rec.*, 29(2):93–104, May 2000.

[9] Hong Chang and Dit-Yan Yeung. Robust path-based spectral clustering. *Pattern Recognition*, 41(1):191–203, 2008.

[10] Bernard Chazelle and 36 coauthors. The computational geometry impact task force report. *Applied Computational Geometry Towards Geometric Engineering*, 223:407–463, 1999.

[11] Ed Huai-hsin Chi. A taxonomy of visualization techniques using the data state reference model. In *IEEE Symposium on Information Visualization*, pages 69–75. IEEE, 2000.

[12] Jon Christensen, Joe Marks, and Stuart Shieber. An empirical study of algorithms for point-feature label placement. *ACM Transactions on Graphics (TOG)*, 14(3):203–232, 1995.

[13] William S. Cleveland and Robert McGill. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association*, 79(387):531–554, 1984.

[14] Tammara T. A. Combs and Benjamin B. Bederson. Does zooming improve image browsing? In *Proceedings of the Fourth ACM Conference on Digital Libraries*, DL '99, pages 130–137, New York, NY, USA, 1999. ACM.

[15] Phillip Crews. Thumbs plus. `http://www.cerious.com/thumbnails.shtml`.

[16] Niklas Elmqvist, Pierre Dragicevic, and Jean-Daniel Fekete. Rolling the dice: Multidimensional visual exploration using scatterplot matrix navigation. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1539–1148, 2008.

[17] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second*

International Conference on Knowledge Discovery and Data Mining, KDD'96, pages 226–231. AAAI Press, 1996.

[18] Allen Institute for Cell Science. Interactive plotting. `http://www.allencell.org/interactive-plotting.html`.

[19] Michael Friendly and Daniel Denis. The early origins and development of the scatterplot. *Journal of the History of the Behavioral Sciences*, 41(2):103–130, 2005.

[20] Edward Gan and Peter Bailis. Scalable kernel density classification via threshold-based pruning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 945–959. ACM, 2017.

[21] Jeffrey Heer and Ben Shneiderman. Interactive dynamics for visual analysis. *Queue*, 10(2):30:30–30:55, February 2012.

[22] Torstein Hønsi. Highcharts performance boost. `https://www.highcharts.com/blog/news/175-highcharts-performance-boost/`.

[23] Petra Isenberg, Pierre Dragicevic, Wesley Willett, Anastasia Bezerianos, and Jean-Daniel Fekete. Hybrid-image visualization for large viewing environments. *IEEE Transactions on Visualization and Computer Graphics*, 19:2346–55, December 2013.

[24] Alex Johnson, Jack Parmer, Chris Parmer, and Matthew Sundquist. Plotly benchmarks. `https://plot.ly/benchmarks/`.

[25] Unmil P. Karadkar, Richard Furuta, Selen Ustun, YoungJoo Park, Jin-Cheon Na, Vivek Gupta, Tolga Ciftci, and Yungah Park. Display-agnostic hypermedia. In *Proceedings of the fifteenth ACM Conference on Hypertext and Hypermedia*, pages 58–67. ACM, 2004.

[26] Gordon Kindlmann and Carlos Scheidegger. An algebraic process for visualization design. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2181–2190, 2014.

[27] Sebastian Klaassen. Globalview. `https://github.com/RcSepp/global_view`.

[28] Sebastian Klaassen. Globalview.js. `https://github.com/RcSepp/GlobalView.js`.

[29] Faith R. Kreitzer, Nathan Salomonis, Alice Sheehan, Miller Huang, Jason S. Park, Matthew J. Spindler, Paweena Lizarraga, William A. Weiss, Po-Lin So, and Bruce R. Conklin. A robust method to derive functional neural crest cells from human pluripotent stem cells. *American Journal of Stem Cells*, 2(2):119, 2013.

[30] Rashid Mehmood, Guangzhi Zhang, Rongfang Bie, Hassan Dawood, and Haseeb Ahmad. Clustering by fast search and find of density peaks via heat diffusion. *Neurocomputing*, 208:210–217, 2016.

[31] Tamara Munzner. *Visualization analysis and design*. CRC press, 2014.

[32] Brock Roberts, Amanda Haupt, Andrew Tucker, Tanya Grancharova, Joy Arakaki, Margaret A. Fuqua, Angelique Nelson, Caroline Hookway, Susan A. Ludmann, Irina M. Mueller, Ruian Yang, Alan R. Horwitz, Susanne M. Rafelski, and Ruwanthi N. Gunawardane. Systematic gene tagging using CRISPR/Cas9 in human stem cells to illuminate cell organization. *bioRxiv*, 2017.

[33] Alex Rodriguez and Alessandro Laio. Clustering by fast search and find of density peaks. *Science*, 344(6191):1492–1496, 2014.

[34] Jahangheer S. Shaik and Mohammed Yeasin. Visualization of high dimensional data using an automated 3D star co-ordinate system. In *Proceedings of the International Joint Conference on Neural Networks*, pages 1339–1346. IEEE, 2006.

[35] Pavel Torgashov. Fast colored textbox. `https://www.codeproject.com/Articles/161871/Fast-Colored-TextBox-for-syntax-highlighting`.

[36] Barbara Tversky, Julie Bauer Morrison, and Mireille Betrancourt. Animation: can it facilitate? *International Journal of Human-Computer Studies*, 57(4):247–262, 2002.

[37] Mereke van Garderen, Barbara Pampel, Arlind Nocaj, and Ulrik Brandes. Minimum-displacement overlap removal for geo-referenced data visualization. *Computer Graphics Forum*, 36(3):423–433, 2017.

[38] Sean Williams, Mark Petersen, Peer-Timo Bremer, Matthew Hecht, Valerio Pascucci, James Ahrens, Mario Hlawitschka, and Bernd Hamann. Adaptive extraction and quantification of geophysical vortices. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2088–2095, 2011.

[39] Caroline Ziemkiewicz and Robert Kosara. Embedding information visualization within visual representation. In *Advances in Information and Intelligent Systems*, pages 307–326. Springer, 2009.

Appendices

A Visual Mapping Scripting Language

A scripting language for visual mappings has to be simple, extensible and easily readable by humans. These requirements align with the principles of SQL. The structured query language (SQL) is a domain specific language for querying and modifying relational databases. SQL is easily understandable by non-experts. For example the SQL statement `SELECT name FROM customers` queries a list of names from the "customers" data table. Similarly, we design our scripting language to support easily comprehensible commands like `SELECT WHERE $salinity > 0.5` to select all images whose 'salinity' parameter is greater than 50% or `Y selected BY 1` to move selected images up by one unit.

The basic syntax of our scripting language is: `COMMAND ::= STATEMENT SCOPE [CLAUSE]` In the following we will elaborate the individual parts of this language.

`STATEMENT ::= CONTROL_STATEMENT | VISUAL_MAPPING_STATEMENT | SHORTCUT_STATEMENT`
`STATEMENT` refers to the operation to perform. We define three kinds of statements. Visual mapping statements apply transformations to the visual mapping of the images provided by `SCOPE`. Control statements do not modify image locations. Shortcut statements are shorter forms of other statements.
Example: `SELECT  -- Select all images`

Statement	Description / Example
`SELECT`	Select images
`FOCUS`	Move camera to fit images
`HIDE`	Hide images
`SHOW`	Unhide images
`CLEAR`	Remove all visual mappings from images
`COUNT`	Return number of images
`FORM`	Define new image group
	`FORM diagonal WHERE $theta == $phi`

Table 3: Control statements

`SCOPE ::= [GROUP | ('where' WHERE_CONDITION)]`
`SCOPE` refers to a group of images. Predefined groups are 'all', 'none', 'visible' and 'selection' (or alternatively 'selected'). The default group is 'all'. New groups can be defined using the `FORM` statement. Instead of a group, the keyword `'where'` can be used, to select all images that satisfy the condition `'WHERE_CONDITION'`.
Example: `SELECT none  --Deselect all images`

`WHERE_CONDITION ::= C#_EXPR`
`WHERE_CONDITION` refers to a boolean expression in C#. Numeric parameters of an image are referenced with the `'$'` prefix, string parameters are referenced with the `'@'` prefix and indices into sorted parameters are referenced with the `'#'` prefix.
Example: `SELECT where #flux <= 10  -- Select top 10 images of lowest flux`

Statement	Description / Example
`X`	Transform images in x-direction (Cartesian coordinates)
	`X all BY 10`
`Y`	Transform images in y-direction (Cartesian coordinates)
	`Y selected BY $intensity`
`Z`	Transform images in z-direction (Cartesian coordinates)
	`Z WHERE $intensity > 100 BY 1`
`THETA`	Transform images in θ-direction (polar coordinates)
`PHI`	Transform images in ϕ-direction (polar coordinates)
`R`	Transform images in r-direction (polar coordinates)
`STAR`	Transform images in star coordinates
`LOOK`	Show only the image whose parameters most closely matches the view angle
`SKIP`	Show only images that evaluate to true
	`SKIP all BY #t != (int)(time * 10) % 4   -- Animate 't'`

Table 4: Visual mapping statements

Statement	Description
`SPREAD`	Transform images using parameter indices as star coordinates
`RSPREAD`	Disperse images randomly in x- and y-directions
`RSPREAD3D`	Disperse images randomly in x-, y- and z-directions
`ANIMATE`	Animate the given parameter by BY with 10 frames per second

Table 5: Shortcut mapping statements

```
CLAUSE ::= 'by' BY_EXPRESSION
BY_EXPRESSION ::= C#_EXPR (',' C#_EXPR)*
```

`CLAUSE` defines arguments for the statement. It consists of the `'by'` keyword and a comma-delimited list of arguments. Control statements do not have a clause.
Example: `STAR all BY #x, #y   -- Align images in 2D Cartesian grid`