Dive In! Enabling Progressive Loading for Real-Time Navigation of Data Visualizations

Michael Glueck[1,2], Azam Khan[2], Daniel Wigdor[1]

[1]Dept. of Computer Science
University of Toronto
{mglueck|daniel}@dgp.toronto.edu

[2]Autodesk Research
Toronto, Ontario, Canada
{firstname.lastname}@autodesk.com

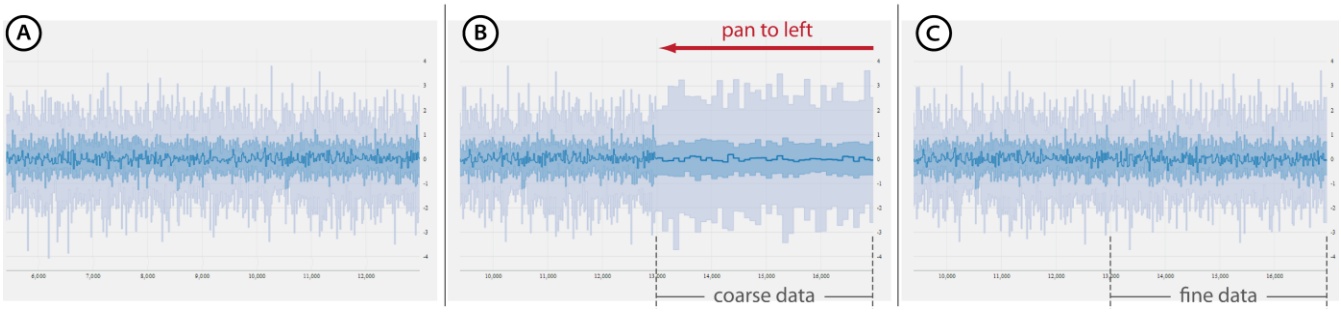

Figure 1: The Splash framework enables real-time navigation for client-server visualization systems by progressively loading data. (a) A user viewing part of a dataset (b) pans to the left and a coarse version of missing data is downloaded and displayed immediately, until (c) the fine data are finished downloading and are displayed. Splash streamlines the process of creating and automates retrieving these level-of-detail versions for both data curators and visualization developers.

ABSTRACT

We introduce Splash, a framework reducing development overhead for both data curators and visualization developers of client-server visualization systems. Splash streamlines the process of creating a multiple level-of-detail version of the data and facilitates progressive data download, thereby enabling real-time, on-demand navigation with existing visualization toolkits. As a result, system responsiveness is increased and the user experience is improved. We demonstrate the benefit of progressive loading for user interaction on slower networks. Additionally, case study evaluations of Splash with real-world data curators suggest that Splash supports iterative refinement of visualizations and promotes the use of exploratory data analysis.

Author Keywords

Client-server; data visualization; progressive-loading; real-time interaction

ACM Classification Keywords

H.5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

CHI 2014, April 26 - May 01 2014, Toronto, ON, Canada

ACM 978-1-4503-2473-1/14/04…$15.00.
http://dx.doi.org/10.1145/2556288.2557195

INTRODUCTION

In client-server visualization systems, data is stored on remote servers and then transmitted on-demand to client applications. This architecture provides flexible, scalable access to data across a variety of platforms and devices. However, supporting real-time visual exploration remains a complex undertaking, especially as the size of a dataset grows. For example, if the volume of data is too large to display coherently on the screen at one time, a developer must allocate additional development resources toward implementing methods of data reduction, the creation of a coarser-grain level-of-detail (LOD), and most importantly navigation, such as zooming and panning. If data is reduced carelessly or if visual exploration of the data is omitted due to the added development overhead, the opportunity to find errors, inconsistencies, or anomalies is lost. Even powerful automated methods of analysis, such as statistical measures and tests, or machine learning-based models, can only reveal a partial perspective on a dataset. A visual verification step that allows results to be explored, inspected, and navigated is desirable and beneficial [12].

User interaction in client-server systems typically occurs as a stepped transaction: user input invokes a network request for additional data, which must be fulfilled before an update can be displayed. Subject to network conditions, such as latency and throughput, these network requests become a dominant factor hindering a smooth user experience. If a user must wait for the system to respond, the result is a bottleneck to interaction. Latency has long been known to negatively impact user performance [23]; as little as 10ms of latency has been found to be noticeable when interacting with touch screens [25].

Navigation operations such as resizing the view, zooming, scrolling, or panning all require downloading additional data to the client. Common strategies, such as *a priori* client-side caching, benefit smaller datasets, but do not scale well as the size of data increases. Facilitating real-time navigation with vast server-stored datasets is non-trivial: it generally requires (1) that the client system can randomly access any portion of the dataset, and (2) that the server and client systems work in tandem to filter, aggregate, or resample the data on-the-fly to render a visualization in real-time.

Visual patterns and features seem to automatically jump out at the human eye; this is precisely why data visualization lies at the core of *exploratory data analysis*, an approach that foregoes *a priori* assumptions of the data model and allows patterns to emerge through visual exploration [31]. Not only does visual exploration serve to debug and validate the results of automated methods but also, more importantly, it supports opportunistic discovery. We believe real-time navigation is critical to visual exploration, as it facilitates an uninterrupted interaction dialog with the data, and encourages diverse and flexible questions to be asked and answered. As Cleveland put it: "To regularly miss surprises by failing to probe thoroughly with visualization tools is terribly inefficient because the cost of intensive data analysis is typically very small compared with the cost of data collection." [7]

In this paper we present Splash, a framework that enables easy development and modification of client-server visualization systems to include smooth and continuous, rather than stepped, navigation (see Figure 1). This is accomplished through two key modifications to the traditional client-server model. First, on the server-side, Splash streamlines the specification of a LOD hierarchy and the precomputation of a multi-scale version of the dataset. Second, on the client-side, Splash manages fetching these LODs from the server by automatically selecting an appropriate target LOD to display and then progressively downloading increasing resolutions until this target is reached. Thus, real-time, on-demand navigation is achieved by ensuring the highest resolution data can be rendered at interactive framerates and by minimizing the duration of the first network request associated with an interaction.

We start by demonstrating the user performance benefits of enabling real-time navigation for visual search tasks common in exploratory data analysis. Building on these positive results, we discuss considerations informing the design of the Splash framework, detail the developer experience, and describe the architecture of the framework. Next, we report on deployment case studies evaluating the process of selecting a useful LOD hierarchy and aggregate measures. Finally, future directions are discussed.

BACKGROUND AND RELATED WORK

In a visualization, navigation offers users freedom and flexibility to control how data is displayed *ad hoc*. Using Yi *et al.*'s taxonomy of user interaction tasks, navigation falls under the *explore* user interaction [32]. Prompt resolution of interaction transactions is critical to preventing bottlenecks – but how fast is fast enough? We frame our discussion along two dimensions: (1) flow between inputs, using Spence's terms *continuous* and *stepped* [30] and (2) system responsiveness, using Seow's terms *instantaneous* and *immediate* [29]. Continuous interaction characterizes inputs which can be mapped to a continuous function, such as click-and-drag panning, while stepped interactions are discrete operations, such as clicking a zoom-in button. According to Seow, instantaneous responses are those that occur within 100 to 200ms after input, while immediate responses are between 500 and 1000ms. Thus, navigation through direct manipulation should be continuous and instantaneous – what we dub *real-time*.

Splash maintains real-time navigation by automatically determining the optimal LOD and progressively downloading additional data, thus mediating the density and download size of plotted data points. Splash has benefited from a study of existing interactive visualization toolkits, real-time interaction with multi-scale data, and progressive downloading of visual information, which we summarize here.

Interactive Visualization Toolkits

Interaction has grown in prevalence with the emergence of generalized visualization toolkits, such as the InfoVis Toolkit [10] and Prefuse [17], which simplified implementation for visualization developers. Pad [26] introduced the zooming user interface, where multi-scale environments can be explored through panning and zooming navigation. Pad++ [1] utilized image-based LOD pyramids and degraded image representations to mediate visual clutter and improve responsiveness; ZVTM [27] extended these principles to interactive visualizations. More recently, the D3 toolkit [2] incorporated out-of-the-box behaviors for panning and zooming navigation for web-based visualizations.

While all of these toolkits simplify the transformation of arbitrary data into visual representations, the onus remains on the visualization developer to (a) define, generate, and manage a multi-scale version of the dataset, and (b) select, load, and cache specific LODs for display. Splash modifies process (a) above by introducing the data curator role, a domain expert who explicitly defines a multi-scale data model. This model-based LOD pyramid ensures that the important attributes defined by the data curator remain visually salient. Splash then generates and manages the resulting multi-scale dataset. Next, all parts of process (b) are automatically handled by Splash, ensuring real-time interaction is maintained. The visualization developer need only map data from Splash to the format expected by the visualization toolkit. Thus, Splash is designed to be used in tandem with existing visualization toolkits.

Real-Time Interaction with Multi-Scale Data

Custom visualization systems have addressed improving system responsiveness. The Control Project demonstrated responsiveness of direct-query visualization systems could be improved by monitoring rendering time and dynamically reducing visualization complexity to increase frame rates [19]. ATLAS ensured real-time interaction by initiating distributed computation of data aggregates prior to being requested by the visualization, by anticipating user intentions [5]. In contrast, Cloudvista employed a randomized batch-scheduler to generate aggregates for data ranges related to the current data view [6]. In both cases, on-the-fly aggregation of data must be scheduled in advance of a client-side request to maintain real-time interaction, due to the inherent scheduling latency of distributed computation. Thus, all of these systems are limited by requiring the adoption of custom data storage, computation, and visualization systems. The Splash framework supports bindings to be written quickly for existing data servers. Real-time navigation is achieved through the storage of pre-computed data aggregates on commodity hardware.

Progressive Download

Displaying a low fidelity representation prior to loading a high fidelity version has long been utilized to ensure responsive interaction [4,14,19]. It is commonly found in bitmap interlacing algorithms, such as Adam7 in the PNG format, and in geospatial tools, including the popular map website Google Maps. In addition to the obvious user experience benefits, this technique has been shown to support quicker identification of images [16] and enable faster video scrubbing [24]. Informed decision-making can be supported by sacrificing precision for speed through progressively-refined partial query results [13]. Pre-computed aggregates have also been used to facilitate real-time visual analytics for text [3]. Progressive loading clearly has broad applications; however, there is a lack of generalized support for information visualization tools. Using Splash adds progressive loading to existing interactive visualizations.

Computing LODs for arbitrary datasets in unknown domains was a challenge in the development of Splash. Unlike down-sampling images, creating lower fidelity aggregates of arbitrary data is highly task and domain specific [14]. The present work does not seek to contribute novel aggregation methods; for further reading please refer to general concepts [8,15] and data structure specific guidelines [9]. Instead, we design our framework such that the data model, LOD hierarchy, and aggregate measures are fully customizable by the data curator, thereby supporting both arbitrary data types and aggregation methods. This ensures that the useful aggregate can be selected given the type of data, user task, and data domain.

The Splash framework is designed to work with existing visualization workflows. In the next section, we investigate the benefits and disadvantages of progressive downloading to user performance in data visualization tasks.

MOTIVATING PROGRESSIVE DOWNLOAD

Progressively-downloading data enables real-time interaction by sequentially loading LODs of increasing resolution: a *coarse LOD* is quickly downloaded and displayed first while the *fine LOD* is continuously downloading, providing pleasingly smooth, on-demand interaction (see Figure 1).

Aside from improvements in smoothness, we believed progressive downloading would improve performance of certain exploratory data analysis tasks when network throughput is lower. Intuitively, the scale for potential improvement depends largely on the utility of the coarse LOD to a given task. We also considered that progressive downloading incurs an additional data download cost over non-progressive download. To understand whether users would find this an advantage or disadvantage, we designed a study to contrast user performance between progressive and non-progressive data loading methods under differing network throughput conditions and varying degrees of coarse LOD utility.

Tasks and Coarse LOD Utility

Prior user studies have investigated visual comparisons across non-navigable data views, such as multiple time-series plots [18,20,22] and animated charts [28]. Visual search while navigating has been studied as a streaming video scrubbing task [24], but not, so far as we are aware, for data visualization. We draw inspiration from these studies and propose three visual search tasks for navigation: *coarse feature*, *global feature*, and *fine feature* (see Figure 2). In each task, the participant scrolls the data visualization to center the target feature within the viewport. To generate the coarse LOD, we use an aggregate consisting of the mean and maximum/minimum, visualized as a line and a shaded range, respectively. When paired with each of these three tasks, this aggregate provides varying degrees of assistance, ranging from low, when the coarse LOD provides no help regarding the target, to high utility, when the target can be already found merely by inspecting the coarse LOD.

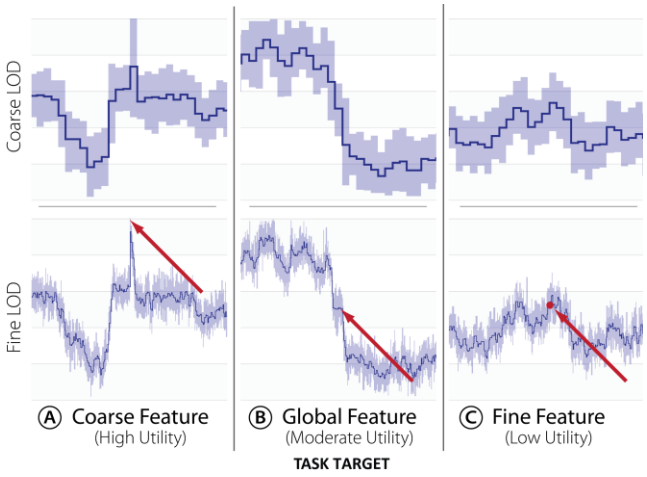

Figure 2: Example target features: (a) the maximum value of the dataset, (b) the shift in the mean value, and (c) a specific value. The coarse LOD offers varying utility to identifying the feature at the fine LOD. (Arrows are illustrative)

Coarse feature: The coarse feature target is only identifiable when it appears in the viewport, but remains salient in the coarse LOD (adapted from "random" [24] and "max" [20,22] tasks). In our study, we chose the maximum value in the dataset (see Figure 2a). This task simulates high utility, where visual confirmation of target presence can be made merely by inspecting the coarse LOD.

Global feature: The direction to the global feature target is immediately apparent based on overall trends of the data (adapted from "ordered" task [24]). In our study, this feature appears as an overall shift in the mean (see Figure 2b). This task simulates moderate utility, where the trends of the dataset inherently support user orientation and recovery from overshoots when scrolling quickly. Thus, lower latency primarily aids the user, rather than the coarse LOD.

Fine feature: The fine feature target can only be found once the finer LOD is displayed in the viewport (adapted from "random" [24] and "comparing values" [18] tasks). In our study, this feature appears as a small disc superposed only on the finer LOD (see Figure 2c). This task simulates low utility because the coarse LOD delays the display of the target. Since the task emulates careful searching of details, we posit external validity.

Hypotheses

H1: For task/aggregate pairings with high utility, completion time will be lower with progressive loading on slow networks.

H2: For task/aggregate pairings with low utility, completion time will be higher with progressive loading on slow networks.

Participants and Apparatus

Twelve volunteer participants (7 female) with mean age 26 (min: 20, max: 34) were recruited from the community, and paid $20 for a 60-minute session. Mean reported computer usage was 35 hours/week; none worked as a data analyst.

Participants performed the study in a private study room using a desktop computer configuration (2.5 GHz dual-core / 3GB RAM) running Windows 7 (64-bit), with a 24-inch LCD monitor displaying a resolution of 1920x1200 pixels. Participants were seated 20 inches away from the monitor.

A horizontal scrolling interface, with a scrollbar at the bottom, simulated an interactive time-series visualization. The viewport was 800x400 pixels (8.5x4.25 inches) in size. Download speed was simulated: 1.5Mbps for mobile and 25Mbps for broadband, both with a fixed latency of 200ms. Data were generated using a random-walk function, similar to [18], with dynamically authored target features (see Figure 2). The size of the dataset was fixed at six-times the viewport width to ensure scrolling would be required.

Design

A repeated measures within-subject design, with independent variables: *loading method* (non-progressive, progressive), *network throughput* (broadband, mobile), and *target feature* (coarse, global, fine), yielded a 2x2x3 design.

Loading method and network throughput were crossed, and target feature ordering was counter-balanced, resulting in 4 unique conditions per target feature. In each trial, target position was a randomly generated position ¼ to ¾ through the dataset. For each target feature, trials were divided into 6 blocks, each with 3 repetitions of each of the 4 conditions, yielding 12 trials per block, 72 trials per target feature, and 216 total trials. We recorded task completion time as the duration between the start of the first scroll operation to the end of the last scroll operation, thus factoring-out preparation and acquisition time.

Results

Results are shown in Figure 3. A repeated measures ANOVA with a Greenhouse-Geisser correction determined that differences in mean navigation time were statistically significant between all factors:

- loading method ($F_{(1,11)} = 123.260,\ p < .001$)
- network throughput ($F_{(1,11)} = 373.537,\ p < .001$)
- target feature ($F_{(1.294,14.235)} = 21.293,\ p = .002$)

No effect was found between blocks, indicating absence of learning effects. *Post hoc* tests using Bonferroni correction revealed a significant difference in completion time between fine feature and both coarse and global feature tasks.

These results support our first hypotheses: coarse and global features were found more quickly (28.7%, 29.8%) in the mobile throughput condition with progressive loading (H1). Progressive loading improves user performance on mobile networks to a level comparable to broadband when the aggregate is sufficiently useful for the task. We were unable to find support for our second hypothesis: although participants took slightly longer to find the fine features in the mobile throughput condition with progressive loading (5.9%), this difference was not statistically significant (H2). This suggests that even when utility is low, progressive loading does not significantly hinder user performance. In contrast, performance in the broadband throughput condition indicated negligible differences between progressive and non-progressive loading across conditions.

Thus, not only does progressive loading provide users with a smooth real-time navigation experience, it also shows clear benefits to user performance with minimal drawbacks.

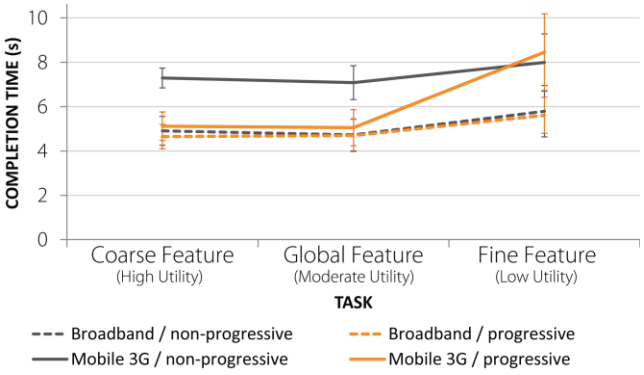

Figure 3: Task completion times. Bars indicate a 95% CI.

SPLASH FRAMEWORK DESCRIPTION

When designing Splash, our goal was a framework that would be flexible and easily integrated into existing visualization workflows. Consequently, a logical separation of client and server developer roles is reflected in the architecture. Splash provides support for two distinct developer roles: the *data curator*, who manages the data on the server, and the *visualization developer*, who creates the visualization for the end user, the data analyst (see Figure 4).

The Splash framework consists of three modules. First, on the server-side, the data curator uses the *Splash Aggregator* to blueprint a multi-scale version of the dataset and define the aggregate measures used to generate LODs. Running this utility pre-computes and stores the multi-scale version of the dataset on the data server, along with the blueprint metadata (see Figure 4, Steps 1-2).

Second, the data curator implements the *Splash Data Interface*, a simple API used on the client-side to query the metadata and multi-scale data LODs (see Figure 4, Step 3). While this component must be instantiated by the visualization developer, no configuration is necessary.

Last, on the client-side, the *Splash Cache* is used by the visualization developer to route requests to the data server (see Figure 4, Steps 4-5). When initialized, the Splash Cache fetches the metadata from the data server and automatically configures the transport of data between the client and server. It seamlessly manages progressive downloading and caching of LODs. The visualization developer simply routes data requests through Splash; existing visualization tools require little to no modification as a result.

Abstract interfaces between client and server components of Splash enable support of a variety of data server technologies and visualization toolkits. Using Splash, many visualizations can be created for a single pre-computed multi-scale version of a dataset or a single visualization can mix data from multiple data sources.

We now describe each of these steps in greater detail.

Data Curator: Data Preparation and Access (Steps 1-3)

The following Python snippets illustrate the data curator's data model abstraction task. While terse, attributes of these data structures are referenced in later examples. In practice, we envision a UI would be used for configuration.

Splash Aggregator (Step 1)

The data curator starts by authoring blueprint metadata (see Figure 4, Step 1). First, the dataset is uniquely identified and the interval of the navigable data dimension(s) is defined. For example, consider data from an environment sensor, sampled every minute. Time is the only dimension. The interval of timestamps to be processed is provided by `start_time` and `end_time`:

```
dataName = 'EnvironmentSensor'
dataDimensions = [{
  'name': 'Time',
  'interval': [ start_time, end_time ] }]
```

Next, the curator defines the mapping from an existing data sample to a `datum` representation. Suppose the data source provides a sample as an array, `data`. The timestamp of the sample is stored as the *position* (along the dimension); multiple values can be linked to a position. At this stage any element of `data` can be modified, such as converting from strings to timestamps using `strptime` and `mktime`.

```
def datum( data ):
  return {
    'position': t.mktime(dt.strptime( data[0] )),
    'values': [
      { 'name': 'Temperature', 'value': data[3] },
      { 'name': 'Humidity', 'value': data[4] },
      { 'name': 'Light', 'value': data[2] } ]}
```

A LOD hierarchy defines the size of the discrete bins to be used for aggregation. In our example, the sensor data is sampled every minute and is aggregated into hourly and daily LODs, with relative sizes defined in milliseconds. Additional variables can be configured to ensure proper alignment to the start of hours and days.

```
lods = [
  { 'name': 'minutes', 'size': 60000 },
  { 'name': 'hours', 'size': 3600000 },
  { 'name': 'days', 'size': 86400000 } ]
```

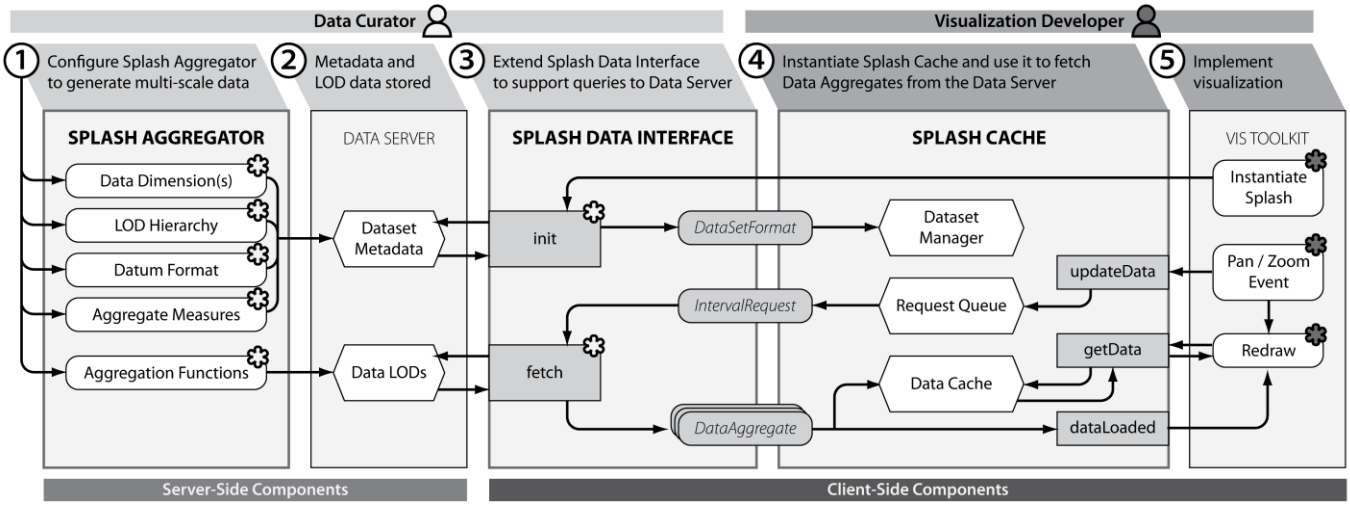

Figure 4: Overview of the Splash framework. The five steps required to integrate Splash into server- and client-side are divided amongst the data curator and visualization developer roles. Implementation responsibilities of each are indicated by asterisks.

The current LOD abstraction places two limitations on the richness of the LODs that can be expressed. First, the LODs must be a strict hierarchy: child LODs must evenly subdivide their parent LOD. Second, parent LODs at the same depth must have the same number of children LODs. Thus, semantic hierarchies, such as ontologies cannot generally be used as LOD hierarchies, since they often exhibit irregular structure. However, wide ranges of data are supported by our abstraction, from time-series to gridded geo data.

Finally, data values are mapped to aggregation functions. The functions `meanTmp`, `stdTmp`, `maxHum`, and `minLgt` return the mean, standard deviation, maximum, and minimum, respectively, of a specific value in a list of `datum` objects. In this way, different aggregation functions can be defined for each value of interest, and more than one aggregate can be calculated for each:

```
measures = [
  { 'Temperature': [
    { 'name': 'mean', 'function': meanTmp },
    { 'name': 'std',  'function': stdTmp } ]},
  {'Humidity': [
    { 'name': 'max',  'function': maxHum } ]},
  {'Light': [
    { 'name': 'min',  'function': minLgt } ]}]
```

Since the method of binning data points for aggregation is based on the LOD hierarchy, continuously sampled data will be discretized in the process of aggregation. This may be more or less appropriate given the domain of the dataset.

Data Server (Step 2)

The Splash Aggregator utility can be run locally or in the Cloud (see Figure 4, Step 2). First, the curator-authored metadata described above is stored on the data server. Second, raw data are streamed and converted to `datum` representations, grouped into LOD bins, and then passed to the aggregation functions specified above. The results are then stored on the data server. In this way, a pre-computed multi-scale version of the dataset is generated. The Splash Aggregator utility can be re-run at any time to include additional aggregate measures or to process newly added data.

Splash Data Interface (Step 3)

To support the client-side Splash Cache, the data curator need only implement the Splash Data Interface, which consists of two API calls (see Figure 4, Step 3). First, `init` retrieves the dataset metadata from the data server and returns a `DataSetFormat` object. Second, `fetch` takes an `IntervalRequest`, which encapsulates a *query interval* along the data dimension(s) and a *target LOD*, and returns a collection of `DataAggregate` objects to the Splash Cache. Each `DataAggregate` contains values of all measures (see Figure 4, Step 1) for a sub-interval of the data dimension(s).

Visualization Developer: Splash Cache (Steps 4-5)

Splash is designed to be used with a variety of visualization toolkits, so a client-side Splash Cache module generalizes the process of requesting data (see Figure 4, Step 4-5). Internally, it maintains a dynamic data cache to store retrieved data and a queue of pending data requests.

Automatic Configuration

The Splash Data Interface is instantiated with a URL to the data server; a new Splash Cache is created using this Splash Data Interface. The Splash Cache automatically calls `init` and uses the `DataSetFormat` to dynamically create bindings for the data dimensions, LOD hierarchy, datum format, and aggregate measures (see Figure 4, Step 4). Thus, the visualization developer need not configure any dataset specific details to use Splash. They only need to know the names of the aggregate measures they wish to visualize.

Fetching and Displaying Data

Whenever the viewport of the visualization is moved or resized, the `updateData` method is called (see Figure 4, Step 4). The Splash Cache, in turn, first checks its dynamic data cache, then calls `fetch` for missing data. Each time the visualization is redrawn, the client application calls the `getData` method, which returns all available data points in the Splash Cache, and then renders the visualization. When additional data is retrieved, the client application is notified by a `dataLoaded` callback. These data points are always returned as a collection of `DataAggregate` objects, which may need to be remapped to be consumed by the visualization toolkit. In many cases, existing client visualizations can integrate Splash by simply requesting data from the Splash Cache instead of directly querying their present data server.

Target LOD Selection

Splash also manages the selection of an appropriate target LOD to display, via the `getBestLod` method. This calculation takes into account both the size of the data viewport, in pixels, and the resolution of the display, in dots per inch (DPI). Splash uses a device independent metric, *samples per inch* (SPI), to ensure that the same `IntervalRequest` will be displayed identically for the same SPI setting and visualization dimensions on a variety of devices, regardless of display size or resolution.

This target LOD is determined by traversing the LOD hierarchy from finest to coarsest LOD and comparing the interval of one LOD (`L1`) and the next coarsest LOD (`L2`), weighted by the number of subdivisions:

```
L1.size + (L2.size – L1.size) / L2.subdivs
```

If this density is greater than the currently displayed density, `L1` is returned. By default, Splash uses a two-stage progressive download: given a target LOD, the next coarsest LOD is fetched first. A configuration parameter allows the visualization developer to modify the number of coarser LODs fetched prior to the target LOD.

Usage Example

Here, we present a simplified, but working, example of a client application written in JavaScript, which highlights the use of Splash (see Figure 4, Step 5). The drawing routines of the visualization toolkit are represented as a single method call, `visToolkit.render`. For example, this function would iterate over the contents of `data`, the collection of `DataAggregate` objects, and draw a point for each value.

An additional method call, `visToolkit.getDomain`, returns the domain of the interval currently displayed. For example, the array returned might represent the interval from 12:00 until 18:00 on a particular day.

```
// Initialize Splash
dataURL = "http://datasource/EnvironmentSensor";
jsonDataSource = new JSONSplashDataInterface( dataURL );
splash = new SplashCache( callback, jsonDataSource );
splash.setDpi( 72 );
splash.setSpi( 40 );
splash.setSize( width, height );

// GetIntervalRequest, interval of data to fetch/display
function getIntervalRequest() {
  var domain = visToolkit.getDomain();
  var reqIntvl = new 1DInterval( domain[0], domain[1] );
  var reqLod = splash.getBestLod( reqIntvl );
  return new IntervalRequest( reqIntvl, reqLod );
}

// Redraw, called when data becomes available (see below)
function redraw() {
  var data = splash.getData( getIntervalRequest() );
  visToolkit.render( data );
}

// Update, called on zoom or pan
function update() {
  splash.updateData( getIntervalRequest() );
  redraw();
}

// Callback when data is available (dataLoaded)
function callback() {
  redraw();
}
```

More concretely, we will use D3 as an example visualization toolkit. Then, retrieving the domain of the current view, the `visToolkit.getDomain` function, is simply:

```
x.domain()
```

where `x` is the D3 horizontal scale object. Second, converting `DataAggregate` objects in the `data` parameter of `visToolkit.render` function to the expected format of a given D3 visualization is accomplished by a map function:

```
data.map( function(d) {
  return { u: d.intvl[0], v: d.Temperature.mean }
});
```

where the interval and mean temperature attributes of the `DataAggregate d` are renamed `u` and `v`, respectively. In a similar fashion, visualization developers can easily integrate data from Splash into other visualization toolkits.

Framework Availability

Splash will be released as a free web service, available at: http://www.splash-data.org

Data Types and Aggregate Limitations

Presently, Splash supports both densely and sparsely sampled discrete and continuous data; categorical, numeric, and string data values; and one- and two-dimensional query intervals along navigable dimension(s) of the data.

Splash does not support data visualizations where elements of the visualization are dynamic, for example, a node-edge graph visualization where nodes can be repositioned. Elements of visualizations must have fixed positions along the navigable dimension(s) of the dataset.

Despite these limitations, Splash is able to accommodate a wide variety of data visualization applications, from time-series to geographic data. Splash can be used with standard 1D visualizations (e.g., line charts, bar graphs, and dot plots) and with standard 2D visualizations (e.g., heat maps and scatterplots). For example, visualizations of financial data, weather charts, gridded census data, and fixed node-edge diagrams are all supported. Potential aggregate measures include classic summary statistics, such as the mean, standard deviation, and inter-quartile range, and more complex algorithms, like nearest neighbor clustering.

Theoretical Interaction Latency

The average interaction latency achieved by Splash is affected by two factors, the SPI and the LOD compression ratio of the multi-scale version of the dataset. The LOD compression ratio is the average number of samples aggregated at each level of the LOD hierarchy. For example, if bins of 6 samples are used to aggregate each LOD, the compression ratio would be 6. The lower the compression ratio, the more often LOD transitions occur when zooming.

The SPI parameter is used to tune the density of samples displayed in the visualization. As the SPI increases, more data is downloaded. We have found that for desktop environments, an SPI of 20 to 40 and a compression ratio between 4 and 8 yields visually pleasing transitions, while still ensuring real-time interaction across a range of SPI settings.

The impact of LOD compression ratio and SPI on interaction latency is contrasted between progressive and non-progressive loading methods for mobile 3G and broadband network conditions (see Figure 5). While interaction latency on mobile 3G benefits the most (34% to 416%), broadband also shows improvement (2% to 33% improvement).

Given device characteristics and network conditions, it may be desirable for the visualization developer to decrease the SPI or increase the number of progressively loaded LODs to maintain real-time interaction.

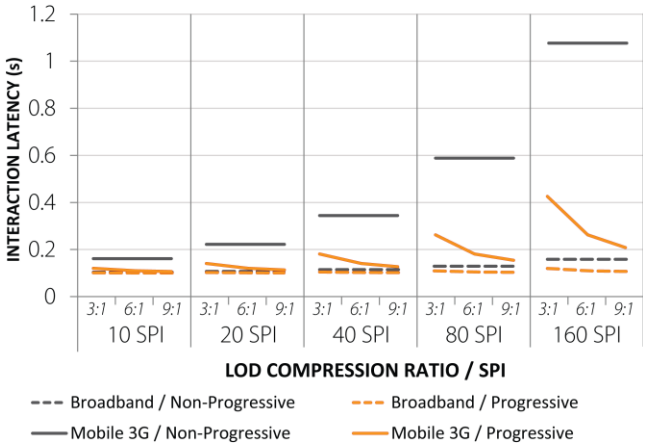

Figure 5: Modeled response times under varying LOD hierarchy compression ratios and SPI settings. Progressive loading vastly improves interaction latency.

DEPLOYMENT STUDY

We conducted a study to evaluate the data curator experience (see Figure 4, Steps 1-3). In particular, we focused on the exercise of data model abstraction and aggregation, engaging data curators to consider which attributes of their data were important for likely analysis tasks.

Three researchers were recruited for participatory case studies using their own research data. The researchers were from the domains of computer systems and bioinformatics research. Since each had a varying degree of programming experience, researchers played the role of data curators, and we the role of visualization developers. To this end, we provided each researcher with a multi-scale visualization tool, navigable by panning and zooming, with visibility toggling for each aggregate measure plotted. We did not coach or offer advice in the data model abstraction and aggregation task.

The study was conducted over three sessions. Before the first session, we asked participants to select a dataset they had previously analyzed. We started with a semi-structured interview to probe their typical workflow and analysis tools. The capabilities of Splash were demonstrated using their dataset. We asked the researchers to describe aggregate measures that would enable them to gain insights into their data in an exploratory analysis task. In the second session, we engaged the researcher in a pair-programming activity: implementing the LOD hierarchy and aggregate measures, based on measures from the first session, for the Splash Aggregator. We then ran the utility to generate the pre-computed multi-scale version of their data and presented the final visualization. In a final open-ended interview, we asked them to describe their experience, indicate difficulties encountered, and reflect on the overall utility of Splash to create interactive analysis tools.

Case Study 1: High-Performance Computing Errors

The first participant is a researcher of computer systems. He chose to analyze a dataset of memory error reports extracted from BlueGene supercomputer logs. The logs include the timestamp and number of corrected errors that occurred, categorized by the correction algorithm used: single symbol (SSE), double symbol (DSE), and chipkill errors (CK).

He reported his typical analysis is conducted in three steps. First, shell scripts and command line tools, such as `grep`, `sed`, `sort`, and `awk`, are used to pre-process the data. Second, the cleansed data is loaded into Matlab, R, or customized data structures in C and Python to compute statistical measures and tests, such as CDF plots and autocorrelation analyses. Finally, Matlab, R, or gnuplot is used to generate static visualizations for publication. Time-oriented visualizations are seldom used due to the overhead of manually reframing data plots in their current tools. However, the researcher noted that such visualizations could be useful in isolating abnormalities and discerning overall trends.

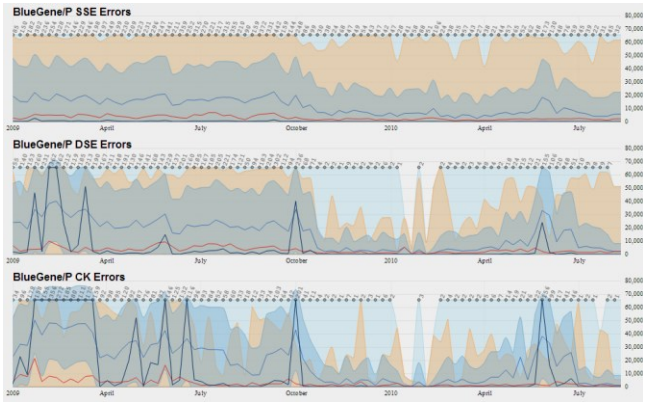

Figure 6: Screenshot of the BlueGene dataset visualization. Three errors types in stacked charts with linked navigation.

Since his analyses primarily focus on comparing distributions of corrected errors, his aggregate measures included the mean, standard deviation, median, min, and max. Additionally, the researcher wanted to gain insight into *error overflows*. The memory error logging system has a fixed-size register, capping the count at 2^{16}. He employed conditional aggregate measures to track the number of error overflows and filter error counts to exclude overflows.

Reflecting on the Splash Aggregator the researcher stated, "the process seems fairly straightforward." The researcher was pleased with the final visualization: general trends were quickly apparent, and he noted, "relative differences between error types are immediately clear" (see Figure 6). He was surprised at the large difference between the unfiltered and filtered means; commenting that "[filtering overflow values] could indicate [repeatable] hardware errors as opposed to transient [software] errors. This is easily distinguishable from the visualization." This insight would not have been apparent using his current analysis methods. Additionally, the visualization highlighted several time periods with abnormalities. The researcher was also able to confirm trends and features he had discovered in his prior analyses.

Overall, he was excited that Splash would enable more extensive hypothesis testing in the second step of his current workflow. He was very pleased with the responsiveness and freedom of navigation provided by the interactive visualization, commenting "it's great that I can quickly zoom into the details to investigate interesting trends." He said such an analysis in his current workflow is currently "more trouble than it's worth", but with Splash he felt compelled to "look closer."

Case Study 2: Expression Quantitative Trait Loci (eQTL)

The second participant is a genetics researcher of the human genome. He chose a dataset that consisted of negative-log transformed p-values for several thousand single nucleotide polymorphisms (SNPs). Locations of SNPs are scattered along a chromosome; in this case a dimension of 250 million locations. Strong p-value scores indicate which SNPs play the role of eQTLs in the human ileum.

His current analysis workflow consists of generating stationary scatterplots, called *Manhattan plots*, using Excel and visually searching for locations where clusters of strong p-values occur. This process is very time consuming because isolating the SNPs involved at specific locations requires manually reframing the plot to view details and then cross referencing to find the associated SNP marker names.

We prepared an interactive bar chart, analogous to the Manhattan plot. Local maxima and clusters of strong p-values were more important to his analysis than the distribution of p-values. The aggregate measures he chose to implement included the median, the max, and an additional label of the name of the maximum SNP in a range (see Figure 7). After exploring the data with this final visualization, the researcher commented that "the max remains by far the most useful [aggregate measure]… It helped me answer my questions about areas of concentrated strong p-values along the chromosome contrasted against 'deserts.'" He also noted, "the interactivity is fantastic… the ability to dynamically zoom in and out… is absolutely necessary."

This participant had less familiarity programming in Python than the first participant, but commented that working with the aggregates available through `NumPy` was "super simple". He said that "adding new aggregate [measures]… is straightforward if you have reasonable programming skills", but might be challenging to those in his field with little or no programming experience. Overall, he commented "the type of visualization your software provides is a necessary first step to eye-balling your data" prior to engaging more complex analyses of asymptotic or periodicity. Splash made this first step analysis far more accessible.

Case Study 3: Gene Sequence Base Frequency

The third participant is a visualization developer working in bioinformatics with plant cellular genome data. The data he chose to analyze is a gene sequence of ~360 thousand bases (A, C, T, or G) from Arabidopsis Thaliana. It is the entire sequence of chromosome M.

Current tools he uses to visualize genomic regions, such as the UCSC Genome Browser, are quite complex. Many data plots are available and while multi-scale, the interaction remains stepped and it is difficult to add custom visualizations. Our participant was interested in developing a "fun" visualization "to try out something different."

Since the physical properties of DNA are important, he wanted to develop an interactive visualization that would help visually identify regions along the chromosome where higher frequencies of certain bases occur. He explained, "promoter regions are typically more A-T rich" (see Figure 8). While this is rarely a primary research question, this visualization could be used to provide additional context for other data explorations, such as transcription factor binding site information and intron splice sites. The researcher implemented an algorithm to aggregate the normalized frequencies of each base along the chromosome.

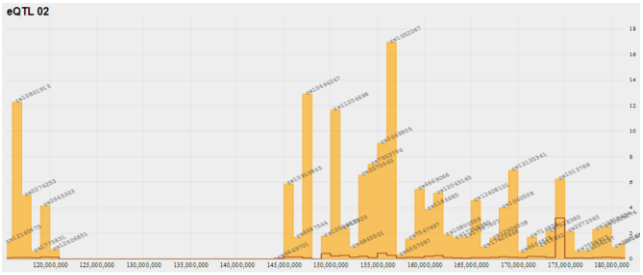

Figure 7: Screenshot of the eQTL dataset visualization. Max, median, and the name of the max SNP are plotted.

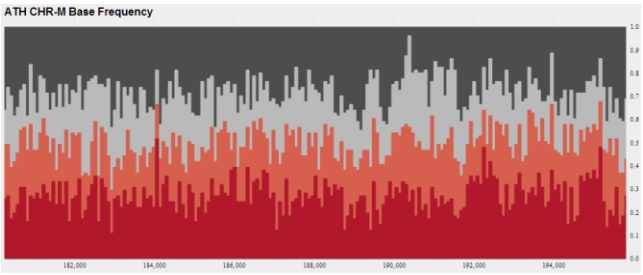

Figure 8: Screenshot of the ATH CHR-M dataset visualization. Normalized frequencies of bases are displayed.

Reflecting on the pair-programming exercise, the researcher commented that "the effort to implement [aggregate measures] was relatively minor; but, it requires knowledge of programming." He suggested a UI to facilitate the Splash Aggregator configuration steps would make it more accessible to a wider range of researchers. As a visualization developer he appreciated the reusable nature of the precomputed LOD data: "that kind of flexibility seems quite well worth [the configuration involved]." Overall, he was very positive about the "playfulness" of the real-time interaction that Splash enabled. "It helped the data come to life."

Discussion

Overall, participants agreed that preparing their data for use with Splash was intuitive and provided a much richer experience compared to their existing analysis tools. The addition of real-time interactivity enabled them to freely explore the data at multiple LODs. Of the three participants, the Case Study 1 participant was the most intrepid in utilizing non-standard measures, which we suspect relates to his familiarity with divide-and-conquer style analyses, as evidenced by the first step of his analysis workflow.

Most difficulties encountered were due to uncertainty when deciding on a LOD hierarchy. This was less of an issue in Case Study 1, where the time-oriented dataset mapped to meaningful time ranges, but for both genomic datasets (Case Study 2 & 3), the researchers felt that defining the LOD hierarchy was arbitrary. They preferred that it be automatically generated.

Based on these initial results, we believe that Splash can be easily integrated into existing early-stage analysis workflows, either introducing exploratory visual analysis to domains where it is not currently used or enhancing stationary plots with real-time navigation.

CONCLUSION AND FUTURE WORK

We have introduced Splash, a framework that enables scientists to dive-in and interactively explore their data using real-time navigation through progressive downloading in multi-scale client-server visualizations. We provide empirical evidence to support the use of progressive downloading for visualizations. The results of our first study suggest that progressive loading makes data available more quickly. This immediate display of data could help mitigate common issues in multi-scale spaces, such as *desert fog* [21]. Our second study suggests Splash is easily integrated into existing analysis workflows, making exploratory data analysis more accessible to researchers.

Splash is not without its limitations. We intend to add support for more complex LOD hierarchies, including semantic and ontological hierarchies. LOD-dependent aggregate measures would enable *semantic zooming* [26]. We are also investigating real-time responsiveness tuning, where LOD selection and SPI account for network conditions.

The lack of support for features such as real-time interactivity, continuous feedback, multi-scale representations, and partial queries, among others, have been identified as continuing barriers to the wider adoption of existing scientific and information visualization tools and toolkits [11]. Supporting real-time interaction with visualizations of larger data, allowing any user to answer the simple yet valuable question, "What does my data look like" is paramount to facilitating the broader use of visualization tools. We hope that Splash will promote the use of exploratory data analysis at early stages of data analysis by mitigating many of these barriers, placing more powerful visualization tools directly into the hands of researchers and domain experts.

ACKNOWLEDGEMENTS

We would like to thank our reviewers, study participants, John Hancock, Katie Barker, John Stasko, and fellow DGP members for their insightful feedback.

REFERENCES

1. Bederson, B. B., & Hollan, J. D. (1994). Pad++: a zooming graphical interface for exploring alternate interface physics. *UIST,* 17-26.
2. Bostock, M., Ogievetsky, V., & Heer, J. (2011). D^3 Data-Driven Documents. *IEEE TVCG*, 17(12), 2301-2309.
3. Boukhelifa, N., Chevalier, F., & Fekete, J. (2010). Real-time aggregation of wikipedia data for visual analytics. *IEEE VAST*, 147-154.
4. Card, S.K., Robertson, G.G., & Mackinlay, J.D. (1991). The information visualizer, an information workspace. *CHI*, 181-186.
5. Chan, S.M., Xiao, L., Gerth, J., & Hanrahan, P. (2008). Maintaining interactivity while exploring massive time series. *IEEE VAST*, 59-66.
6. Chen, K., Xu, H., Tian, F., & Guo, S. (2011). Cloudvista: Visual cluster exploration for extreme scale data in the cloud. *SSDBM*, 332-350.
7. Cleveland, W.S. (1994). *The elements of graphing data*. AT&T Bell Laboratories.
8. Ellis, G., & Dix, A. (2007). A taxonomy of clutter reduction for information visualisation. *IEEE TVCG*, 13(6), 1216-1223.
9. Elmqvist, N., & Fekete, J.D. (2010). Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines. *IEEE TVCG*, 16(3), 439-454.
10. Fekete, J.D. (2004). The infovis toolkit. *INFOVIS*, 167-174.
11. Fekete, J.D., Silva, C. (2012). Managing Data for Visual Analytics: Opportunities and Challenges. *IEEE DEB*, 35(3), 27-36.
12. Fekete, J.D., Van Wijk, J.J., Stasko, J.T., & North, C. (2008). The value of information visualization. *Info. Vis.*, 1-18.
13. Fisher, D., Popov, I., & Drucker, S. (2012). Trust me, I'm partially right: incremental visualization lets analysts explore large datasets faster. *CHI*. 1673-1682.
14. Fredrikson, A. et al. (1999). Temporal, geographical and categorical aggregations viewed through coordinated displays: a case study with highway incident data. *NPIVM*, 26-34.
15. Goldstein, J., & Roth, S. F. (1994). Using aggregation and dynamic queries for exploring large datasets. *CHI*, 23-29.
16. Harrison, C., Dey, A.K., & Hudson, S.E. (2010). Evaluation of progressive image loading schemes. *CHI*, 1549-1552.
17. Heer, J., Card, S.K., & Landay, J.A. (2005). Prefuse: a toolkit for interactive information visualization. *CHI*, 421-430.
18. Heer, J., Kong, N., & Agrawala, M. (2009). Sizing the horizon: the effects of chart size and layering on the graphical perception of time series visualizations. *CHI*, 1303-1312.
19. Hellerstein, J.M., Avnur, R.,... & Haas, P.J. (1999). Interactive data analysis: The control project. *Computer*, 32(8), 51-59.
20. Javed, W., McDonnel, B., & Elmqvist, N. (2010). Graphical perception of multiple time series. *IEEE TVCG*, 16(6), 927-934.
21. Jul, S., & Furnas, G. W. (1998). Critical zones in desert fog: aids to multiscale navigation. *UIST*. 97-106.
22. Lam, H., Munzner, T., & Kincaid, R. (2007). Overview use in multiple visual information resolution interfaces. *IEEE TVCG*, 13(6), 1278-1285.
23. MacKenzie, I.S., & Ware, C. (1993). Lag as a determinant of human performance in interactive systems. *CHI*, 488-493.
24. Matejka, J., Grossman, T., & Fitzmaurice, G. (2012). Swift: reducing the effects of latency in online video scrubbing. *CHI*, 637-646.
25. Ng, A., Lepinski, J., Wigdor, D., Sanders, S., & Dietz, P. (2012). Designing for low-latency direct-touch input. *UIST*, 453-464.
26. Perlin, K., & Fox, D. (1993). Pad: an alternative approach to the computer interface. *CHI,* 57-64.
27. Pietriga, E. (2005). A toolkit for addressing hci issues in visual language environments. *IEEE VLHC*, 145-152.
28. Robertson, G., Fernandez, R., Fisher, D., Lee, B., & Stasko, J. (2008). Effectiveness of animation in trend visualization. *IEEE TVCG*, 14(6), 1325-1332.
29. Seow, S.C. (2008). *Designing and Engineering Time: The Psychology of Time Perception in Software*. Addison-Wesley.
30. Spence, R. (2007). *Information Visualization: Design for Interaction*, Second Edition. Pearson Education.
31. Tukey, J. W. (1977). *Exploratory Data Analysis*. Addison-Wesley.
32. Yi, J.S., ah Kang, Y., Stasko, J.T., & Jacko, J.A. (2007). Toward a deeper understanding of the role of interaction in information visualization. *IEEE TVCG*, 13(6), 1224-1231.