DynaViz: A Haskell Realtime Graphing Library

Greg Jones, Alex Mordkovich
{gregj, amordkov}@stanford.edu

December 17th, 2011

1 Introduction

DynaViz (short for "dynamic visualization") is a Haskell library for visualizing streaming data. Existing data visualization tools such as MATLAB and Microsoft Excel work with finite data. That is, a user furnishes a finite-length file containing her data, which the software then processes and presents graphically. However, these tools do not support visualizing unbounded data streams, or visualizing data in real time. Thus, we created DynaViz to visualize data online, as it arrives, without requiring a finite dataset.

2 The Case for Realtime Data

Realtime data visualization has numerous applications. Today, vast quantities of publicly available digital data are generated every second, from the stock market to social media sources such as Facebook and Twitter. Monitoring and understanding this data is interesting from both a personal and a business standpoint. More generally, it is often instructive to monitor the vitals of any complex computerized system whose state is constantly changing or being updated.

On the other end of the spectrum, we may be interested in data that is generated on less accessible time scales. For example, historical data that accumulates over the course of many years or decades. Or data that is collected by particle accelerator sensors, which pick up millions of events per second[1]. Replaying, or simulating, such data at human-scale speed can allow scientists to better digest and analyze the information. Note that although such data can be statically plotted against time to make it presentable to a human, doing so "wastes" a dimension of spatial representation; if the nature of the data is two-dimensional, then a 3-D plot must be used to visualize it. And if the data has three (or more) dimensions, then there is no way to statically plot it against time. With DynaViz, two-dimensional data can be played back in two-dimensions, with animation capturing the change over time. Theoretically, then, with 3-D rendering support from the underlying framework, DynaViz would be able visualize 3-D data streams.

Moreover, collecting data from a (digital) experiment is often an iterative and error-prone process. Without online data visualization, the experiment is first run and its results are accumulated into a file, which is then provided to the graphing software. If the experimenter then realizes that the wrong data was being collected, that it was not being collected correctly, or that insufficient information was being collected, she must re-run the experiment and repeat the cycle. With online data visualization, the experimenter gets immediate feedback on the quantity and nature of data being collected. The data collection "cycle" can thus be significantly shortened.

3 View from the Top

DynaViz exports a minimal interface to client Haskell programs wishing to visualize their data. A client initializes its interaction with the DynaViz library by invoking the `initialize` entry point. DynaViz takes this opportunity to spawn off a `gloss` thread, which drives the animation, and then returns to the client. (`gloss` is further explained in the next section.) The client can now create graphs and add data to those graphs via the `newGraph` and `newData` entry points, respectively. These functions simply update DynaViz internal graph state. Meanwhile, the `gloss` animation thread calls back into DynaViz to get new frames to display. This callback generates a frame from the current internal graph state. A `TVar` is used to synchronize access to this shared graph state. This interaction is illustrated in Figure 1.

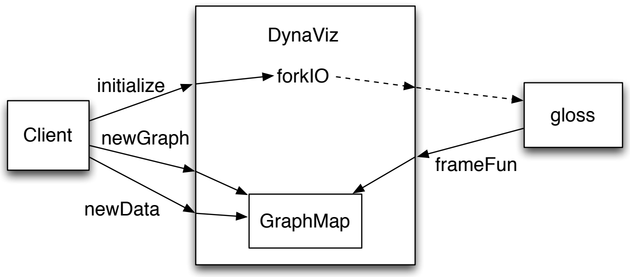

Figure 1: The client–DynaViz interface

4 What is `gloss`?

`gloss`[4] is a realtime drawing and animation framework for Haskell. Much like Processing[2] for Java, `gloss` abstracts away all the difficult aspects of getting an animation up and running. The idea is to hide all the IO driven, monadic aspects of animation from the programmer. With a pure functional draw loop, a beginning Haskeller can easily generate some impressive results, before going on to learn the more complicated aspects of the language.

With the aid of `gloss`, drawing something to the screen requires four simple parameters: a timestep, and a window width and height, and a function that renders a single image of the animation. `gloss` even implements several drawing primitives to make the implementation of the "frame function" easier. Circles, lines, and polygons can all be drawn with simple functions. Matrix transformations can be applied easily with translate, scale, and rotate functions.

5 DynaViz and `gloss`

DynaViz builds heavily on top of `gloss`. `gloss`'s feature set was *almost* perfect for implementing DynaViz. It abstracted away the low level details of windowing and animation, and provided primitives for rendering graphs. It meant we never had to touch GLUT[3], and our UI worked cross-platform automatically.

However, there was one caveat: DynaViz needed to be able to update the animation after the client called it back with new data. Since `gloss` is handed a frame function on initialization, DynaViz needed some way to hand to that function any data it received. This obviously required some sort of synchronization between the frame function and the clients callbacks. Unfortunately `gloss` expects a pure frame function (as it should, since the whole idea is to offer the simplicity of pure functional animation). Our options were either to modify `gloss` internal event loop to listen for new data callbacks (and subsequently pass any new data into the pure frame function), or to simply stick the frame function into the IO monad) and use synchronization primitives to pass data from DynaViz to the frame function). The second alternative proved to be a much simpler change to the internals of gloss, and was the approach we took.

6 Internals

In addition to DynaViz' core synchronization module between client data and rendering engine, a significant portion of the DynaViz source code is its graph rendering implementation. Currently, three different graph types are supported: two dimensional scatter plots, time-based line plots, and bar graphs. Each of these graph types is shown in Figure 2. As seen in the figure, a graph can be drawn to fill one of the four quadrants of the window – with up to four graphs animated simultaneously – or to fill the whole window. The specific features of each graph type are discussed in the subsequent section.

DynaViz also implements a simulation layer that allows a client to "replay" historical data in virtual time. The simulation layer acts as an intermediary between a historical dataset and the DynaViz realtime graphing functionality. Given a real and virtual timestep, a historical start date, and a dataset in a text file, it kicks off a simulation that adds a data point every real timestep, stamping that datapoint based on the simulated timestep and the start date. This feature offers an incredibly simple interface to an historical data simulation.

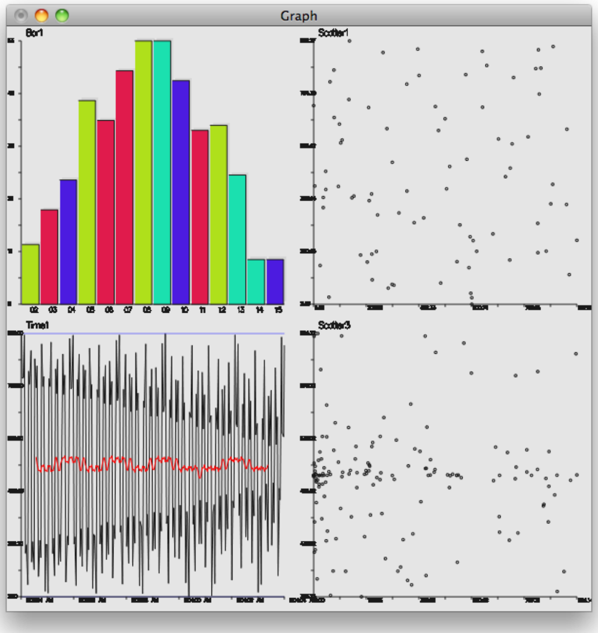

Figure 2: Screenshot of DynaViz running four simultaneous animations.

7 Features

Each DynaViz graph type offers a set of options to customize its display. The bar graph is the least customizable type. It supports a data lifetime, allowing data to be phased out after a fixed amount of time has passed. The second graph type, the scatter plot, also supports a data lifetime. Additionally, the scatter plot offers the ability to pin the x and y axes to the origin (so that the displayed range is always $[0, \texttt{MAX\_VAL}]$), or to leave one or both axes unpinned (so that the range is $[\texttt{MIN\_VAL}, \texttt{MAX\_VAL}]$).

The third graph type, the time plot, offers the most varied functionality. In addition to being able to pin the y axis (pinning the x axis has no meaning in a time plot) and provide a data lifetime, time plots also offer a moving average trendline and minimum and maximum watermarks.

All three types of graphs support data obsolescence. In a time plot, this feature has the effect of creating a sliding window which continuously moves to the right to display the most recent data. The width of this window is then the (user-provided) lifetime of a data point. To accommodate for variability in window widths, time plots employ "smart labeling" for the time (x) axis. Smart labels aim to present the most appropriate time graduations for the current time window. For example, if the visible window spans many years, the labels are graduated by whole years. On the other hand, in a time window spanning a month or two, month and day are labeled; and in a window spanning few minutes, labels are precise to the second.

8 Scalability

Since the data stream that DynaViz visualizes need not be finite, long-running visualizations pose an interesting challenge for our framework. It is naturally impractical to attempt visualize arbitrarily large amounts of data that have been accumulating over the course of a long run, due to both memory limitation and computational limitation in the frame-rendering callback. As it is natural to think of older data as having less importance to the end-user, DynaViz supports data obsolescence, whereby data points that are older than some (user-provided) age are simply discarded.

Another scalability concern is the the synchronization between the sole `gloss` animation thread and the (client) data-producing thread. Currently, a single `TVar` is used to synchronize the two. As such the same "lock" is acquired whenever a new graph or a new datum is added or when the frame-rendering callback walks through (up to four) graph states to render the graphs. However, since no two simultaneous DynaViz graphs share any state in our design, this could be improved by using finer, per-graph `TVars`. Following this direction to its logical end, we could spawn a new `gloss` thread for each graph, to be rendered in its own window. This will obviate the need for a centralized `GraphMap` bottleneck and allow for graceful scaling to a large number of simultaneous graphs, both in terms of performance and in terms of presentation.

The callback paradigm that `gloss` exports requires DynaViz to redraw each frame of the animation anew. Currently, DynaViz recreates the picture from the raw data for every frame. This can be improved by caching the `gloss` frame object (that is, the `Picture`) and returning it if no new data was added (and no old data had expired) since the last frame was rendered. It may also be possible to cache `gloss` objects at a finer granularity. However, determining whether cached `Picture` elements can be reused is then more involved and may require maintaining additional state, such as the bounding box of the data points in the case of a scatter plot.

Yet another optimization would involve "compressing" the per-graph data that DynaViz manages. For example, `BarGraph` datapoints are currently stored unaltered in the form in which the client thread adds them. That is, we maintain all individual instances of the objects whose frequencies the bar graph is visualizing. Compressing this data so that only object counts or relative frequencies (and max count) are stored would reduce the amount of computation that needs to be done in the frame-rendering callback.

9 Future Work

There is a long list of features that we would love to add to DynaViz in the future. A project like DynaViz naturally lends itself to iterative improvement. Countless options and customizations for each graph type can be added to the system.

9.1 Infrastructure

The core DynaViz system ought to be designed with extensibility in mind, and there are several refactoring changes we would like to make to DynaViz, with hindsight as our guide.

First, graph parameters should be pulled out as a list of `GraphParameter` typed options. Each parameter (e.g., whether to pin the x axis) should be offered as a separate type constructor for `GraphParameter`. Right now, each new parameter is explicitly passed as an argument to `newGraph`. This obviously does not work in the long term; as graphs become more customizable, the function signatures become unwieldy and difficult to maintain.

A list of available trendlines should be offered as one of these graph parameters (for time plots). Currently, the mininum, maximum, and moving average trendlines are hard-coded onto any constructed timeplot. A better solution would be to provide a basic set of predefined trendlines, and to allow clients to define their own trendline functions which, given the set of data points, compute a custom trend.

There are also interesting possibilities to explore if `DataPoint`s are treated as a polymorphic type. Currently, DynaViz supports `String` data points and `Float` data points in one and two dimensions. While the dimensionality is dictated by the graph type, we believe that data points could theoretically be of any type that conforms to a `Plottable` typeclass. The class would define the methods needed to plot points in a one-dimensional space; namely, a way to define the distance between two points, and a way to define ordering of points. One use case that comes to mind is that of word clouds, where words are plotted based on their edit distance between each other, rather than based on some numerical value of each word.

Finally, we would also like to modularize the implementation of each graph type as much as possible. The current graphing code is interspersed throughout the core DynaViz data-handling code, and is not standardized between the different types of graphs. While passing in custom graph types from a client program is beyond the scope of this project, we do think that a capable programmer should be able to recompile DynaViz with a new graph that she has written independently. Graphs should be self-contained, and should conform to a common interface to interact with the core DynaViz functionality. Once again, the key goal is to redefine graphs with extensibility in mind.

9.2 More Features

In addition to refactoring changes, we would like to add several features to DynaViz. We had initially planned to add pie charts as a separate graph type, but cut the feature last-minute due to time limitations. We would also like to add the ability to export visualizations as a video file. Both realtime and simulated dynamic visualizations could then be saved and archived for reviewing and future reference.

Additionally, we would like to add the ability to pause a simulation run, play it backward, and in general control the "playback" of the simulation. This interactivity can be built on top `gloss`'s `Game` interface, which allows handling keyboard and mouse events.

9.3 User Interface

Finally, DynaViz drastically needs a UI overhaul. While gloss did a great job of letting us get something up on screen quickly, it is rather limited in terms of its design capabilities. `gloss` lacks support for gradients and good fonts, two key aspects of any visually appealing interface. While a sleek UI may seem trifling, we consider it exceptionally important given that the *sole purpose* of DynaViz is to visualize data.

10 Conclusion

With three different graph types (with varying degree of customizability) and data points, a couple of window layout options, and a thin simulation layer, our initial implementation of DynaViz demonstrates the utility and appeal of realtime data visualization. In its current state, DynaViz represents only a fraction of what is possible with streaming data visualization; for every feature we implemented, we came up with five more possible extensions and improvements. As it matures, we hope that DynaViz will make visualizing streaming data more accessible and more fun.

References

[1] The ALICE Experiment: ALICE Data Acquisition *ALICE – A Large Ion Collider Experiment.* http://aliceinfo.cern.ch/ Public/en/Chapter2/Chap2_DAQ.html/

[2] Ben Fry and Casey Reas, *Processing.* http://processing.org/.

[3] Mark Kilgard, *GLUT – The OpenGL Utility Toolkit.* http://www.opengl.org/ resources/libraries/glut/.

[4] Ben Lippmeier, `gloss`. http://gloss.ouroborus.net/.