behaviorism : A Framework for Dynamic Data Visualization

Angus Graeme Forbes, Tobias Höllerer, and George Legrady

Abstract—While a number of information visualization software frameworks exist, creating new visualizations, especially those that involve novel visualization metaphors, interaction techniques, data analysis strategies, and specialized rendering algorithms, is still often a difficult process. To facilitate the creation of novel visualizations we present a new software framework, *behaviorism*, which provides a wide range of flexibility when working with dynamic information on visual, temporal, and ontological levels, but at the same time providing appropriate abstractions which allow developers to create prototypes quickly which can then easily be turned into robust systems. The core of the framework is a set of three interconnected graphs, each with associated operators: a scene graph for high-performance 3D rendering, a data graph for different layers of semantically-linked heterogeneous data, and a timing graph for sophisticated control of scheduling, interaction, and animation. In particular, the timing graph provides a unified system to add behaviors to both data and visual elements, as well as to the behaviors themselves. To evaluate the framework we look briefly at three different projects all of which required novel visualizations in different domains, and all of which worked with dynamic data in different ways: an interactive ecological simulation, an information art installation, and an information visualization technique.

Index Terms—Frameworks, information visualization, information art, dynamic data.

"But the greatest thing by far is to be a master of metaphor. It is the one thing that cannot be learned from others; and it is also a sign of genius, since a good metaphor implies an intuitive perception of the similarity in dissimilars." Aristotle [3]

"We shall define a database as the model of an evolving physical world." J.R. Abrial [1]

1 Introduction

Developers of visualization projects are primarily engaged in designing and implementing "good metaphors" at different levels of a project, including the organization of the data, the visual presentation of the data and the techniques for interacting with the data. Despite various approaches to define methodological principles, there is a lack of consensus about appropriate, sufficiently detailed methodologies for creating effective visualizations quickly and robustly. Moreover, the types of projects which come under the rubric of information visualization continue to expand, and methodological principles need to address related visualization domains such as information art, knowledge visualization, and network visualization, among others. The kinds of data that need to be represented are often heterogeneous and increasingly complex in nature, and software frameworks need to allow developers to create specific ontologies that work with different combinations and categories of data. Many projects also incorporate temporal elements, and the use of animation to create interactive narratives which enhance information visualization tasks is another issue that needs to be addressed by the software framework. This paper examines these issues and introduces a new information visualization framework called *behaviorism* which address some of the concerns inherent in the above issues in order to facilitate the creation of novel visualizations.

behaviorism provides a wide range of flexibility that lets developers work with dynamic information on visual, temporal, and ontological levels, but at the same time provides appropriate abstractions which allow developers to create prototypes quickly which can then easily be turned into robust systems. The core of the framework is a set of three interconnected graph structures, each with associated operators: a high-performance scene graph to render 3D graphics, a data graph to store and analyze different layers of semantically-linked heterogeneous data, and a sophisticated timing graph to control scheduling, interaction, and animation. In particular, the timing graph provides a unified system with which to create and schedule behaviors that can organize both data and visual elements, as well as the behaviors themselves. The data graph holds various kinds of `Node` objects which represent data elements and `Relationship` objects which represent the connections between `Nodes`. The scene graph holds `Geom` objects which contain methods to draw themselves based on `Node` objects they reference from the data graph. The timing graph holds `Behavior` objects which control both data operations on the data graph, including the retrieval, filtering, processing, and analysis of `Nodes`, and visual operations on the scene graph, including the layout, rendering, and animation of `Geoms`.

- *Angus Graeme Forbes is with the Media Arts & Technology Department at the University of California, Santa Barbara, E-mail: angus.forbes@mat.ucsb.edu.*
- *Tobias Höllerer is with the Department of Computer Science at the University of California, Santa Barbara, E-mail: holl@cs.ucsb.edu.*
- *George Legrady is with the Media Arts & Technology Department at the University of California, Santa Barbara, E-mail: legrady@arts.ucsb.edu.*

Manuscript received 31 March 2010; accepted 1 August 2010; posted online 24 October 2010; mailed on 16 October 2010.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org .

2 Motivation and Related Work

2.1 Methodological Concerns

An issue discussed recently within the information visualization literature is the fact that there is no universal methodological framework that guides visualization designers and developers to create or prototype all flavors of visualization projects easily and quickly. For example, [10] surveys common taxonomies, guidelines, and reference models and notes that they tend to be data-oriented approaches that do not address "creativity and problem-solving challenges that occur in the design process." In a recent survey of challenges and unsolved problems in human-centered visualization environments, [21] bemoans the incompleteness of visual taxonomies and the lack of a working methodology to create effective visual metaphors, making it difficult to manage the constant proliferation of new data. Some authors have pointed to the fact that any overarching visualization methodology is necessarily complex and incomplete. The processes involved in information visualization tasks are not fully understood, either in terms of cognition or perception, and finding effective visualization strategies is enmeshed in social, cultural, and linguistic constructions [5]. A more pragmatic article proposes a model which emphasizes the importance of correctly characterizing the domain problems for a visualization project, and then iteratively addressing threats to a tiered set of common implementation issues [27]. Other recent articles describe "design ideation" strategies from the domain of architecture and graphic design that might help to generate new communicative ideas, for example see [6, 7]. The *behaviorism* framework provides an effective programming framework in the absence of an exhaustive methodological framework. By making it easier to characterize data and generate visual and interaction metaphors, developers can more quickly test out ideas, using different methodologies to create them.

Complicating this awareness of various methodological issues is the expansion of the domain of information visualization to include broader conceptualizations of the targeted tasks and goals of visualization projects. Early discussions of information visualization, such as [30], attempted to categorize both the kinds of data types that visualization projects worked with as well as the tasks users would carry out using them. This categorization generally assumed that data was static and amenable to a straightforward mapping to visual encoding. More recently, there have been a number of re-evaluations of this assumption, leading to new considerations of what inputs to visualization projects are appropriate, what types of data processing might need to be done to transform raw data into visual encodings, and also what kinds of outputs are acceptable and valuable. For instance, [22] discusses the positive effects of aesthetics on information visualization, pointing to projects which have an "extrinsic" focus and an "interpretive mapping" of the raw data to its visual output. In particular, it claims that broadening the scope of information visualization to include extrinsic concerns allows for "high-level interpretations of complex datasets." Similarly, a recent paper measures how certain kinds of "visual embellishment" in fact increase even the fundamental comprehension of data [4].

Another kind of broadening involves the use of data analysis techniques to provide more sophisticated manipulation of the raw data in order to facilitate human reasoning about a particular problem. In this view, the mapping of raw data to visual representation is mediated by iterations of clustering, filtering, and analyzing data. That is, an effective visualization project may involve the creation of new types of data through aggregation, sampling, and other types of processing [36].

A third kind of broadening involves what constitutes raw data. The raw data can often come from disparate sources, and needs to be organized in a way amenable to analysis operations and visual representations. For instance, [36] discusses the challenges of integrating space and time, looking at systems which need to handle streams of dynamic, constantly arriving data, and which need to perform some kind of automated "information synthesis" prior to being visually encoded. According to [21], generating effective "time dependent visualization techniques" is an ongoing challenge: "Watching objects in motion generally provides more insight than static images, but also requires more cognition on behalf of the viewer. The transient nature of a dynamic visualization can make some things not only easier to see, but also more difficult to see." The dynamism inherent in this kind of data may need to be handled differently than static data, and visual representations of it might more naturally be displayed as animations. A recent article explores the often under-utilized role of time and narrative in information visualization and argues for the importance of motion as a primary perceptual form that aids comprehension of visual representations. In particular, it draws parallels to film editing and examines the connection between dynamics and narrative and ways in which the movements between frames and editing techniques influence meaning [25]. Another interesting possibility arises if data is thought of having some form of agency. For example, [17] explores the idea of "organic information" in which data simulates certain properties of living organisms to create emergent data and/or emergent visualizations. A recent paper also mentions that simulations, or perhaps hybrids between information visualization systems and simulation engines, might function as a "creativity support tool" to inspire novel, creative approaches to problems [31].

2.2 Visualization Frameworks

Historically, there is a correlation between methodological approaches and the software toolkits that are created to assist in creating visualizations. That is, the approaches and concerns of the software toolkits echo those methodological concerns. For instance, an influential article concerned with developing a basic taxonomy of common static data types and appropriate visualizations [30] led, perhaps indirectly, to a host of visualization frameworks which focused on providing tools to simplify the creation of visualizations that addressed this taxonomy, for example [9, 14, 29]. Various models that describe the need for a transformation of raw data to visual data (for example, see [10, 8]) are paralleled by frameworks that manage that transformation. For instance, [20] enables developers to create custom data filtering and processing methods. Programmers can then create visualizations by defining "executable chains that can then be run to manipulate visual data and perform animation." That is, the visualizations are created through direct transformations of the data model.

Issues central in the visual analytics community, such as being able to generate knowledge through statistical analysis and being able to keep track of the reasoning processes, are paralleled by frameworks which address those concerns. For instance, [32] supports the process of analytical reasoning through "foraging" and "sensemaking" tasks which are used to posit and judge hypotheses about the data. A knowledge database is used to keep track of sensemaking tasks, such as recognizing correlations between different data elements. Other approaches, such as [35], emphasize the availability of a statistical toolkit which can be used to analyze the data.

Current or ongoing concerns in the information visualization community involve the use of time and animation and strategies for thinking about how new visual analogies can be generated quickly. Visual languages (such as [37, 11]) or general prototyping programming frameworks (such as [24, 28]) aim to let developers "sketch" visualization and interaction ideas quickly before deciding whether or not to commit to developing a more robust implementation. Another framework enables the overlap of time-dependent processes with information visualization techniques, allowing users to create "process visualizations" which incorporate dynamic and temporal data [13].

behaviorism echoes the current methodological concerns by allowing developers create rich visual representations of raw or processed real-time, heterogeneous data and also addresses the likely possibility that new concerns will become important in the future. It aims to be a robust creativity support tool providing flexible mechanisms for developing visualizations that need to provide more sophisticated integration between elements of time, graphics, and data. In particular, it allows developers to quickly create visual narratives which directly or indirectly map the source data. In so doing, it is able to support the development of a wide variety of visualization projects, including ambient data visualization, data art projects, social networking visualizations, simulations, as well as traditional interactive information visualization projects.

2.3 Design Decisions

behaviorism was designed to address the various concerns related to the complexity that dynamic data introduces into a project. These include programming issues that arise from modeling complex data, concurrency issues that arise from working with multiple input streams and user inputs, and scalability issues that arise from working with large data sets. *behaviorism* is primarily structured using separate graph components which address the organization of data, the rendering of visual elements, and the scheduling of processes. Developers can create and reassess the data, visual, or temporal models of the visualization system at any stage of the design without disrupting other aspects.

The organization of data is an important design decision in and of itself which potentially affects performance and scalability. For complex visualization projects, the relevant information that needs to be visualized may exist at the crossroads of raw information, statistical analyses, and user interaction. In other words, the data may be dynamic in multiple ways, and a primary task of an information visualization designer is to develop an effective model of the data which enables successful visualization representation. Due to memory or processing constraints it may not be feasible to keep all raw data within the data graph; a node may represent an aggregation of information or an abstraction or summary of data stored remotely or within an external database. In order to manage large amounts of data, specialized data elements can be used which simplify or combine sets of discrete elements, discarding the original information either programmatically or in response to user-interaction. Similarly, visual elements can be grouped together to minimize rendering time if needed. The data, scene, and timing graphs are all built on concurrent data structures which can be safely updated by different threads. All updates to the graphs are handled by different types of scheduled events (called `Behaviors`, discussed in detail below). These events, though potentially executing asynchronously, update the graphs within a synchronized rendering loop which regulates their interaction with data and visual elements and minimizes problematic side effects. In order to provide the flexibility necessary to render novel visualizations of dynamic data, *behaviorism* includes a real-time, hardware-accelerated, 3D graphics engine with mechanisms to link visual elements to data as well as methods to create text, images, video, and user interface elements. Animation is particularly well-suited to demonstrate changing data and narrative elements, and *behaviorism* supports the animation of visual elements, both supporting transition effects and the use of animation as a meaningful indicator of particular qualities of data.

Defining a visualization project as a set of graphs controlled by behaviors is similar in some ways to dependency graphs used in 3D modelling applications (described for instance in [19]). Core differences include the ability to programmatically update and restructure the graphs during runtime via user-interaction of incoming data events. Additionally, there is an emphasis on interactive information visualization representations which run in real time. *behaviorism* also bears a resemblance to media engines and rich internet application environments such as [2] and [26]. While an information visualization framework could certainly be built upon them, these environments in and of themselves are more generalized programming platforms and as such do not incorporate, for instance, a dedicated data graph for organizing and analyzing data.

3 Implementation and Core Components

behaviorism is made up of three interlocking core components of the framework: the data graph, the scene graph, and the timing graph. The data graph stores pieces of data as `Node` objects, each of which can be linked together via one or more `Relationship` objects. The scene graph stores visual objects, or `Geoms`, within a hierarchy of modelviews. The timing graph stores scheduled events, or `Behaviors`, that update elements in the data graph and the scene graph. `Nodes` of data can be created programmatically or defined automatically by parsing the results of queries to external databases. The creation and visual appearance of `Geoms` can be based on the structure of the data graph or the information in one or more of the `Nodes`. The timing graph is defined initially by the developer, but `Behaviors` can also be created or altered by user input, or by examinations of the data graph and the scene graph. These core graph components use concurrent data structures which allow for robust asynchronous traversals and updates from different threads. The *behaviorism* scene graph uses OpenGL for hardware-accelerated rendering of the `Geoms`. The framework itself is written in Java 6 and utilizes the JOGL2 Java OpenGL bindings [34, 18]. The graph structures are accessed within the main programming thread (or sub-threads spawned from that main thread), via user interface handlers, or in response to new data events. Figure 1 shows a high-level view of the various threads creating and placing `Behaviors` within the timing graph, which then add or update elements on the data graph and the scene graph, or the timing graph itself. Additionally, `Geoms` in the scene graph can be updated by `Nodes` on the data graph.

3.1 The Data Graph

Data in *behaviorism* is semi-structured and linked together by semantic relationships defined by the developer for a particular visualization task. This layer of representation is useful for conceptually organizing the data and also for traversing the data for data processing and analysis.

3.1.1 Properties of Data

The data graph can be used for different purposes. For instance it can be used as a local repository of some portion of remote data. It can also

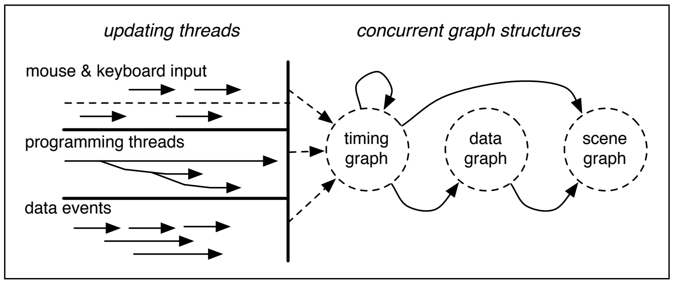


Fig. 1. High-level overview of *behaviorism* architecture. The various programming, input device, and data event threads create `Behaviors` and place then on the timing graph. When the timing graph is traversed during the rendering loop they further create and update elements the data graph and the scene graph.

be used as a dynamic model of locally generated information (e.g., for simulations or knowledge representations). The data graph has a number of properties which make it adaptable to various purposes and able to represent different kind of data. It is heterogeneous, semantic, traversable, filterable, analyzable, and dynamic. That is, it can contain arbitrary types of data, the data is connected by semantic links, it can be traversed, filtered and analyzed through special `Nodes`, and it is dynamic both in that it can receive new information from external sources, and in that it can contain data that fluctuates in response to external sources, user input, or as the result of intrinsic properties that cause it to change in some way over time. Data objects, called `Nodes`, are linkable, via `Relationships`, to other `Nodes`, which embeds them within the data graph. The basic `Node` class can be extended by adding any attributes necessary to describe the data it contains. In addition to the `Nodes` that represent raw data, a number of additional `Nodes` which special functionality can be placed in the data graph. These include, among others, `Collector` objects, which group subsets of the data graph based on particular filters, and `Analyzer` objects, whose attributes are the results of an analysis of one or more other `Nodes`. Since these `Collector` and `Analyzer` objects are themselves `Nodes`, they can be used by a `Geom` to indicate how they will be rendered.

3.2 The Scene Graph

behaviorism includes a high-performance 3D scene graph for storing information about the visual objects, or `Geoms`, attached to the `World`, which is the root of the scene graph. Similar to the original scene graph implementations (such as [33]) and those that are used in game engines (such as [12]), the *behaviorism* scene graph allows developers to position `Geoms` in relation to a parent `Geom`, and to inherit the transformations that are applied to the parent. A special camera `Geom` is attached to the `World` and positioned in absolute coordinates. Rather than being rendered, the camera controls the view of the 3D space. As the scene graph is traversed, the modelview matrix of each `Geom` is calculated and then rendered with appropriate state for lighting, blending, and material properties.

3.2.1 Properties of Geometry

All `Geoms` share a number of basic properties, defined by a set of variables and methods to update these variables: they are transformable, selectable, texture-able, shade-able, controllable, and reflective. That is, they can: be positioned in space; selected or "picked" by various inputs; overlaid with one or more textures; be bound to one or more GLSL shader programs that run on the GPU; have animation attached; and can be associated with one or more nodes of data. Some of these properties can be ignored or set to default values, or they can be customized or extended for specific situations. Less central properties which are available to all `Geoms` include color, material, and lighting. Additionally, various default state parameters such as blending information and depth testing can be specified on a per-`Geom` basis. A large set of default `Geoms` are included with Behaviorism, including

Geoms for basic 2D and 3D shapes, images, videos, and text. Any of these can be subclassed to add customized attributes or functionality, or new ones can easily be created.

3.3 The Timing Graph

The timing graph contains *Behavior* objects which define and schedule operations upon *Geom* objects in the scene graph and *Node* objects in the data graph, and which can also control or manipulate other *Behaviors* in the timing graph. A set of base packages which define a range of different types of *Behaviors* are provided with *behaviorism*. Alongside common timing mechanisms, different interfaces are available that specifically control the items in either the scene graph or the data graph.

3.3.1 Properties of Behaviors

By providing scheduled building blocks of logic, *behaviorism* aims to make it simple to program complex, temporal operations and also to make it easy to to think about and customize the various parts of a complex operation. To that end, *Behaviors* have the following properties: they are stackable, chain-able, flexible, mutable, and relative. That is, a *Behavior* can be stacked onto multiple elements; multiple *Behaviors* can be chained together to compose more complex actions; they are flexible in that they can contain arbitrary logic; they are mutable and both the strength and type of effect they have on elements can change; and they are relative in that they do not interfere with other *Behaviors* from concurrently updating or changing the same element.

The *Behaviors* are categorized into three types– continuous, discrete, intermittent– based on scheduling functionality, and then further defined by their effect on *Geoms*, *Nodes* or other *Behaviors*. A continuous *Behavior* interpolates a value or a set of values across a span of time. The most basic version simply determines what percentage of time has passed between its activation time and its deactivation time, performing a linear interpolation. A set of customizable easing functions are also included with *behaviorism*. By attaching an easing function to a continuous behavior we create a new value from the raw percentage of time passed between activation and deactivation. Easing functions are commonly used for pleasing transition effects, but they can in fact be used for any purpose. A discrete *Behavior* changes the state of a set of discrete variables associated with a *Behavior* or a *Geom*. For instance, a behavior might send a pulse to a *Geom* at particular intervals to trigger its visibility within the scene. An intermittent *Behavior* combines the continuous and discrete types, allowing a user to specify particular times at which to update a set of variables within a specified range.

Each *Behavior* is scheduled to execute (for discrete *Behaviors*) or to begin and end updating (for continuous *Behaviors*) at a particular times. The current time is marked at the start of each iteration of the rendering loop. Each *Behavior* will be activated when the current time is greater than or equal to its scheduled time. All *Behaviors* can be scheduled to repeat indefinitely or for a certain number of iterations. Additionally, the action they take upon repeating can be modified as needed. The value affected by the *Behavior* can be reset if needed, and the direction of the *Behavior* can be reversed. *Behaviors* can also be run in a separate thread if they may perform intensive processing which might potentially slow down the frame rate of the renderer.

In order to attach the behavior to a *Geom*, *Node* or another *Behavior*, a *Behavior* must implement either the interface *GeomUpdater*, *NodeUpdater*, or *BehaviorUpdater* (or some combination of them). Additionally, if some other use for a *Behavior* arises, a user could potentially create their own interface and have their custom *Behavior* implement that instead of (or in addition to) the provided interfaces. In general, unless custom functionality is required, *Behaviors* extend from base classes which implement a particular interface which defines the type of elements it works with. Each of the interfaces requires an update method which describes the logic necessary to update the desired aspects of a particular element.

3.3.2 The GeomUpdater Interface

If a behavior implements the *GeomUpdater* interface then any active *Geom* attached to the scene graph that is also attached to that *Behavior* will execute a method called *updateGeom*, required by the *GeomUpdater* interface. A single *Behavior* can affect multiple *Geoms* simultaneously, if desired. Likewise, a single *Geom* can have multiple *Behaviors* attached, even if the *Behavior* is of the same class. That is, a *Behavior* represents an unfixed process that can be combined together with a number of other processes. As a simple example, we can attach a *BehaviorTranslate* to an object to move it from point A to point B. At the same time, we could attach a second *Behavior* that extends from *BehaviorTranslate* to the same object to represent a gravitational force. And we could also attach a third *Behavior* that also extends from *BehaviorTranslate* that might represent a wind resistance. By so doing we could easily set up a physical simulation of, say, a projectile being fired on a windy day. That is, even though we are moving an object from a specified point to another. These points are not necessary proscribed by a rigid timeline, but can be decided by a multitude of different forces, each modeled with different *Behavior* instances. Of course, these forces do not need to represent physical models; they are flexible enough to be building blocks for any number of concepts. Figure 2 shows an example of *Behaviors* updating the scene graph.

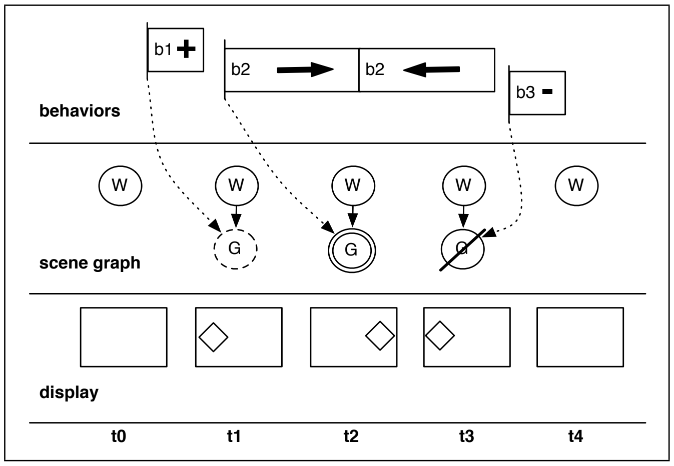


Fig. 2. Diagram of simple *Behaviors* using the *GeomUpdater* interface. At $t0$ the scene graph is empty. At $t1$ a *Behavior* adds a *Geom* to the scene graph (which is rendered as a diamond in the display). At $t2$ a second *Behavior* translates the *Geom* across the screen and back. At $t3$ a third *Behavior* removes the *Geom* from the scene graph. At $t4$ the scene graph is again empty.

3.3.3 The NodeUpdater Interface

The *NodeUpdater* interface is similar to the *GeomUpdater* interface, checking to see if a particular operation should occur at a certain time. While *Geoms* attached to the scene graph need to be re-rendered every frame of the display loop, most *Nodes* will change only occasionally. Often the data graph functions as a local repository or representation of information that is stored in another location, for instance, an SQL database or the file system or a webservices database. *Behaviors* can be set up to retrieve new information from these sources at given intervals based upon information gathered from the data graph. Figure 3 shows an example of *Behaviors* updating the *Nodes* on the data graph.

As an example, a *Behavior* implementing *NodeUpdater* could check if *Nodes* or *Relationships* are outdated and then remove them from the data graph. Another example *Behavior* could periodically re-run an analysis on a particular *Node* or set of *Nodes* to determine if the results have changed. Since *Geoms* can have data attached to them and may have their drawing methods determined by

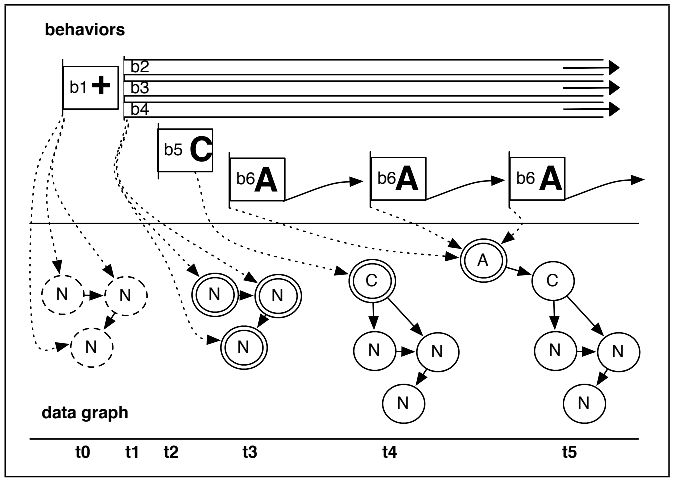


Fig. 3. Diagram of simple `Behaviors` using the `NodeUpdater` interface. At $t0$ a `Behavior` $b1$ retrieves dynamic data from an external source and defines the relationships between the data. At $t1$, `Behaviors` $b2$, $b3$, and $b4$ that update data attributes are created and scheduled to run indefinitely. At $t2$, `Behavior` $b5$ creates a `Collector` and schedules it to traverse the data graph to find nodes that match certain parameters. At $t3$, $t4$, and $t5$ a behavior $b6$ first creates an `Analyzer` and then schedules it to analyze the updated attributes of the collected `Nodes` at certain times. `Geoms` using the `Analyzer` to determine how to render themselves will be automatically updated.

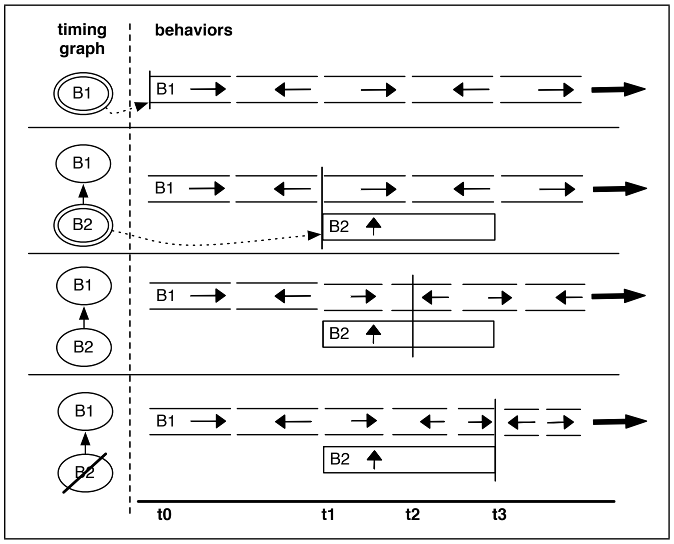


Fig. 4. Diagram of a simple `Behavior` updating a second `Behavior`. At $t0$ a `Behavior` $B1$ (using the `GeomUpdater` interface) is created which translates a `Geom` back-and-forth indefinitely at a set rate. At $t1$, a `Behavior` $B2$ (using the `BehaviorUpdater` interface) is created which updates the rate of $B1$. By $t2$, $B2$ has increased the rate of $B1$ by 50%. At $t3$, $B2$ has increased the rate of $B1$ by 100%, doubling the rate of the translation. $B2$ is then removed from the timing graph.

this data, a `Behavior` that updates an analysis node will have a ripple effect that changes the visual rendering as well. Another example `Behavior` could sample the data graph when instructed by the user. Since the data graph might be too large to conveniently analyze for particular properties, a `Behavior` which samples only certain locations of the data graph is useful for statistical tests.

3.3.4 The BehaviorUpdater Interface

The `BehaviorUpdater` interface allows a `Behavior` to be attached to any active `Behavior`. This allows for he programmatic sequencing and control of scheduling and timing objects in the scene graph. `Behaviors` implementing this interface can alter various aspects of other `Behaviors` including the timing, rate, and even their functionality. For instance, one `Behavior` might be set up to bounce an object back and forth along a trajectory every 10 seconds. A second `Behavior` could be modified to alter the original behavior such that the speed of that trajectory changes from 10 seconds to 4 seconds over a period of 1 minute. That is, every 10 seconds, the speed of the trajectory increases by a rate of 1 second. As with the continuous `BehaviorGeoms`, the continuous `BehaviorBehaviors` can be aggregated as desired, so that for instance two `BehaviorSpeeds` might act as a composite function built out of two acceleration functions.

`Behaviors` can also programmatically create and add new `Behaviors`, or remove existing behaviors, to `Geoms`. For example, a `Behavior` could direct a `Geom` to move toward some other `Geom` when a certain condition is met. If that condition is no longer met, then the `Geom` could be directed to move away from it. We can easily model this by creating a `Behavior` that listens to an `Analyzer` node on the data graph. When the analysis changes (for whatever reason) it can notify the `Behavior` that is moving the `Geom` to now move in the opposite direction. In some cases, it will make sense to use normal programming strategies to define meta-`Behaviors` of this sort, in other cases it may be convenient to have this option. Figure 4 describes an example of one `Behavior` being updated by a second `Behavior`.

3.4 User Interaction

User interaction, via the keyboard, mouse, or other input device, can be bound to a particular visual `Geom` or the entire display (in which case it gets handled by methods in the current `World` class). Default functionality is defined for common interactions such as camera manipulation and selecting or moving individual `Geoms`. For custom interactions, developers can override one or more of a series of functions that handle key presses or various device inputs, which include clicking, hovering, and dragging, among others. In addition to registering a particular function for a `Geom`, the developer is responsible for defining the `Behaviors` which then act on that `Geom`, or instead that execute some other functionality (such as updating one or more other `Geoms` or updating data `Nodes`). The framework is flexible enough so that `Geoms` which represent user interface elements can adapt to information on the data graph or the scene graph. For example, a slider element could have its minimum and maximum values change based upon the number of a particular type of `Geom` currently attached to the scene graph. While the input events are initially captured independently of the graph operations and rendering loop, their application to the visual system is always integrated via the creation of `Behaviors`.

3.5 The Rendering Loop

The main rendering loop has an active OpenGL context and runs within its own thread, controlling all of the core components. It runs as often as possible, synchronizing with the refresh rate of the display. The rendering loop is segmented into three main tasks: scheduling, rendering, and handling interactions. The rendering loop first examines the behaviors in the following order: behaviors which implement the `BehaviorUpdater` interface are examined and run if necessary; `Behaviors` which implement the `NodeUpdater` interface are examined and run if necessary; `Behaviors` which implement the `GeomUpdater` interface are examined and run if necessary. These `Behaviors` will update either elements within the timing graph, data graph, or scene graph, or change the structure of these graphs by adding or removing elements from them. After the `Behaviors` have run, the scene graph is traversed, and the modelview of each `Geom` object is updated and then ultimately rendered according to the data it refers to and the drawing instructions defined by the `Geom`'s draw

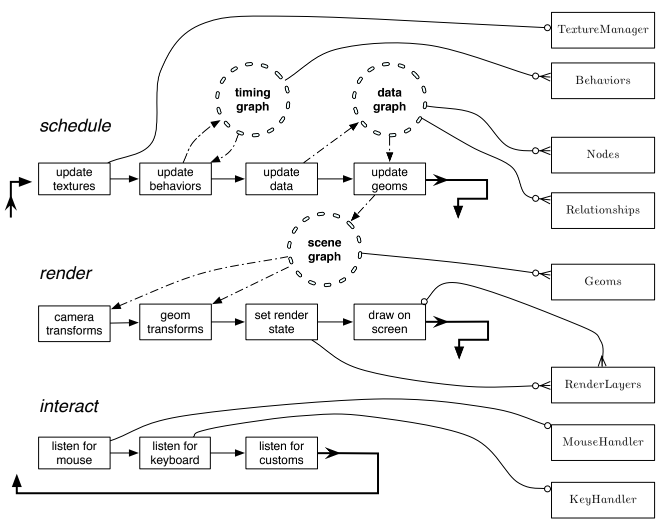


Fig. 5. Schematic of the rendering loop.

method. Once each `Geom` is rendered to the display, the system checks for mouse, keyboard, or other types of interaction with the `Geoms`. Depending on the interaction methods defined for the `Geoms`, various `Behaviors` are created which update the core components which will be reflected in the next iteration of the rendering loop. Figure 5 provides an overview of the iterated steps of each frame of the display loop.

If certain tasks, defined by `Behaviors`, such as data analysis or data retrieval, have the potential to stall the rendering loop, they can be placed in a background thread, activating when the task is complete. Potential difficulties when working with concurrent graph structures include the possibility of pathological situations during graph traversal and processing, especially when multiple actions occurring on independent threads affect the same set of elements. While there is no dedicated mechanism to automatically detect these situations, the organization of the system into separate graphs mitigates against this, making it easier to troubleshoot and refactor code which leads to problematic scenarios. A feature of the *behaviorism* architecture is that it simplifies the development of projects by not requiring to developers to think about the low-level organization of data structures. However, in certain situations, a finer control over the flow of information may be necessary. Developers can specify an ordering to the `Behaviors` so that during the rendering loop a specified `Behavior` can be guaranteed to run before others. Similarly, if the processing on a background thread becomes invalidated due to the creation of a new `Behavior` which is more current, the invalidated `Behavior` can be interrupted and discarded. The rendering loop is the sole location for coordinating changes to the elements within the graphs. This in effect flattens the graph traversals, and gives the developer the opportunity to reason about indeterminacy issues. In practice thus far it has been straightforward to identify bottlenecks and develop simple solutions to address them. For instance, one of the projects described below launches a background thread for each movement of the mouse without any loss of performance. Another project uses a `Collector` node which is only available in the rendering loop once a specified number of remote data calls has completed. Functionality to handle concurrency issues may be added to elements within data graph, the timing graph, or the scene graph as needed.

4 Example Projects

behaviorism has been used in a number of real world projects, ranging from traditional information visualization techniques, to public information art installations, to scientific modeling and simulation. In examining three example projects with different visualization goals, we highlight the effectiveness of using the *behaviorism* framework to model and to represent dynamic data.

4.1 Coil Maps

Coil Maps is an information visualization technique that uses an animated tree map algorithm to display a dynamic overview of geographic data [16]. A map is initially recursively subdivided to a specified depth. The final subdivision constitutes the leaf tiles, which cover the entire map. As new events are attached to a leaf tile, the tile increases in size, which causes its parent tile to increase in size, which in turn causes its parent tile to increase in size, and so on up until the root subdivision. This causes a change to be very apparent on the local level, but also to more subtly show changes on a global scale. For example, if initial tiling covered a map of the world, and an event arrives on, say, New York City, then that tile would immediately greatly increase in comparison to its neighboring tile, but also lesser changes would affect all of the parent tiles, up to and including a minor increase in size of the northern hemisphere and a related minor decrease in the size of the southern hemisphere. This distortion allows a viewer to see at a glance the most active regions of a map over time while retaining the perceptual grounding of the map itself.

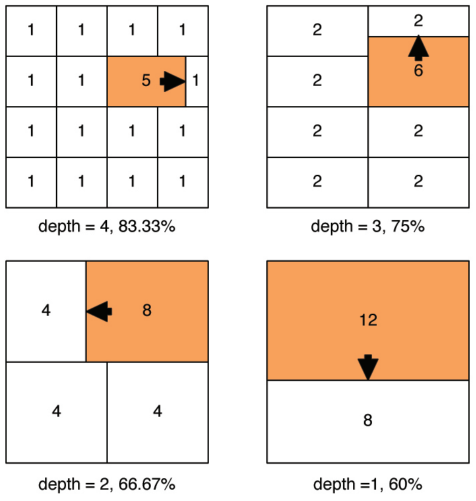
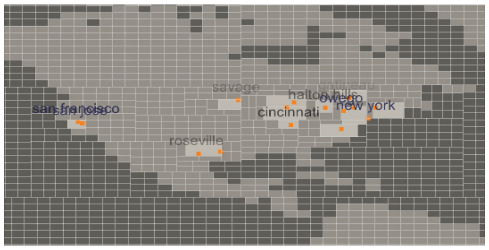
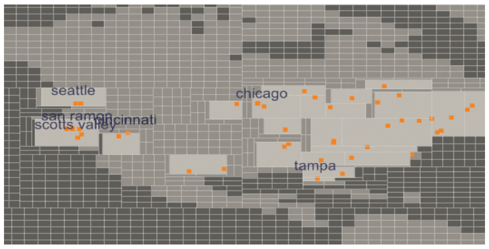


Fig. 6. Adding +4 events to a cell in the Coil Maps algorithm. Each leaf node is initialized with a uniform count; receiving new events causes an adjustment of all of the cells.

behaviorism places the incoming streaming of data into the data graph, parses it and links it to a specific tile based on its geographic location. As a tile is updated, a series of `Behaviors` are initiated which control visual indicators of the event and which animate the restructuring of the map representation. Because all of the incoming data is temporally as well as geographically located. A user can switch to an ”historical” mode and playback the events between a particular start and end time. Another version of the Coil Maps algorithm treats the tree map as an interactive zoom-able interface. Information is arranged in a grid of tiles. When a user mouses over or clicks on a tile, the tile expands, pushing the other tiles out of the way so that the user can see the information in greater detail. In this version, `Behaviors` controlling the animation and updates to the coil map data structure are linked directly to user interaction. Each movement of the mouse triggers an animation which lasts a specified amount of time. Rapid mouse movements will trigger many `Behaviors` which can take place concurrently with no loss of performance. Because of the separation of data, time, and graphics, and because of the modular formation of `Behaviors` and the ease of intertwining these elements, it is easy to create very different visualizations using, in this case, the same dynamic data structure. Figure 6 shows an example Coil Map with 16 leaf nodes and the cascading effects of updating one of the leaf nodes, causing all levels of the map shift. Each update, indicated in the figure with an arrow, is animated using `Behaviors` which control the change in data and subsequent visual changes. Figure 7 depicts screenshots from a realized project using a real-time stream of geographic data.



(a) after 25 events



(b) after 100 events

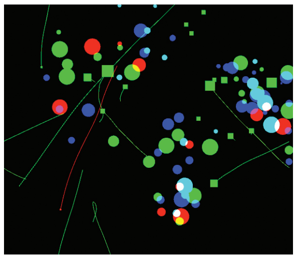
Fig. 7. An example visualization using the Coil Maps algorithm on geographic data.

4.2 Ecotone

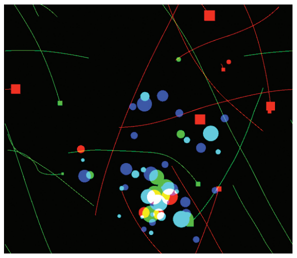
Ecotone is a nutrient dynamics simulation which explores how rational agents within a dynamic system evolve to occupy a particular niche within that system. The knowledge base for each agent is made up of a set of dynamically changing nodes situated within the data graph. Each agent retains information about the other agents in has encountered within the simulation. An `Analyzer` is used to evaluate this information in order to plan the agent's next action. Additionally, the agents can use information gathered from an encounter with a particular individual to make inferences about the species as a whole. This information is also stored on the data graph and updated as necessary. Each agent's knowledge is also stored temporally and monitored by a `Behavior`. The weights of particular inferences that are not reinforced will be attenuated and eventually removed from the data graph entirely. Users can add or remove new elements to the simulation interactively via keyboard and mouse input to test new hypotheses during runtime. Figure 8 shows snapshots from a simulation of a virtual environment over a period of time.

4.3 Cell Tango

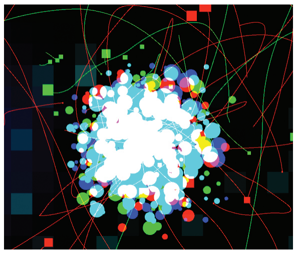
Cell Tango is an interactive multimedia artwork consisting of a series of visualizations based on a dynamically evolving collection of cellphone photographs contributed instantly by the general public [23]. These images, and the accompanying tags which categorize and describe them, are projected large-scale in the gallery, continuously shifting as new contributions are added. The layout and animation for each of the visualizations is defined by the textual associations between the photos, and additionally shows relationships with a set of similar photographs retrieved from the popular Flickr photograph-sharing database via their public API [15]. The visualizations are directly based upon a dynamic database of images and tags. A `Behavior` object controls how often the Flickr database is polled, asynchronously placing retrieved photographs into the data graph. During each of the visualizations the data graph is traversed to collect the necessary photos and tags and connections between them. Figure 9 shows an example of the Cell Burst visualization after the tags from a user-submitted cellphone photograph have retrieved related photographs from Flickr. A series of `Behaviors` governs the layout and movement of the photos as first the user-submitted photograph appears, followed by its tags,



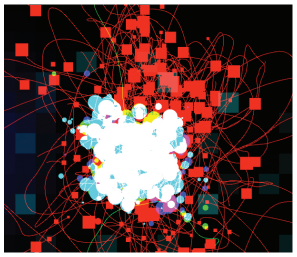
(a) after 30 seconds



(b) after 1 minute



(c) after 5 minutes



(d) after 10 minutes

Fig. 8. An example run of the Ecotone simulation. The animations of the agent's movement (indicated by the lines trailing the squares) and the growth of the nutrients (indicated by the colored circles) are controlled by a series of `Behaviors`.

and finally by the related photographs. Working with the *behaviorism* framework allowed the developers to separate the data collection and representation from the visual presentation, speeding up development time and facilitating the iterative prototyping of multiple visualizations.

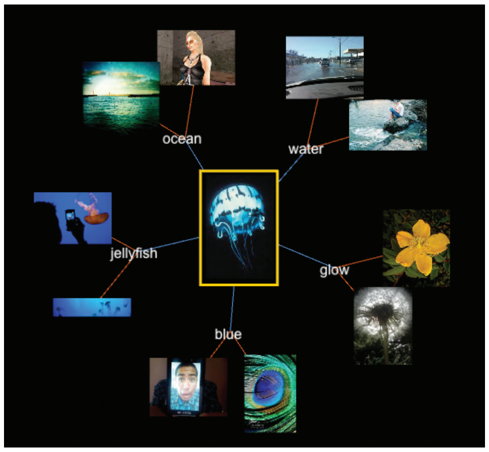


Fig. 9. Detail of "Cell Burst" view of Cell Tango project

5 Conclusion

Software frameworks for information visualization exist to simplify the creation of a project by providing abstractions which facilitate visualization tasks for a particular domain. While this facilitation can be provided by a toolkit of previous solutions to the problem of reaching some commonly desired goal, a more general purpose of a framework is to provide mechanisms which allow developers ways to reason about their goals. Since the knowledge and aims within a particular field is ever-changing, it is more important to provide frames in which to think about domain knowledge. Rather than simply providing specific tools, a framework helps developers create their own tools. Thus a framework needs to provide a high amount of flexibility. At the same time, it needs to offer some incentive to being used over coding "from scratch" using a high level language. In general, the more flexible a framework is, the more time it takes a developer to be productive with it. On the other hand, the easier it is to do something, the less likely a developer will have the necessary control to customize it, who instead will be forced to find workarounds or to bypass the intended functionality. The aim of *behaviorism* is find this sweet-spot along the continuums of flexibility and productivity. This aim is of course is exactly repeated in every information visualization project, where the goal is to take raw data and provide appropriate visualizations of the data to facilitate comprehension or discovery. If the visual metaphor is too rigid, only particular types of understandings will be available to the viewer. If the visual metaphor is too amorphous, there is not enough of a structure to work with, and the viewer may as well be looking at raw data or using low-level tools to parse and filter it. *behaviorism* makes it easier for developers to design effective metaphors for dynamic data at the various levels of the project and to provide the appropriate scaffolding to present them. By segmenting the design process into the data graph, the scene graph, and the timing graph, developers can choose to evaluate the efficacy of their decisions at any time. Changing elements from one area does not invalidate the elements from other areas. Visual elements, for instance, can be changed simply by altering the draw methods of the objects on the scene graph. Broader structural changes can be made by adding new behaviors which control the data processing or the timing of the visual representations.

As exemplified in the previous section, evaluation of the *behaviorism* framework has so far been largely pragmatically measured in terms of the success of visualization projects created with it, the speed and relative ease in which they were developed, and also in terms of the wide range of projects that can be created. Future work still needs to be done to evaluate specific information visualization tasks generated using *behaviorism* in comparison with other frameworks and toolkits. *behaviorism* is an open source project; the full source code is freely available at http://github.com/angusforbes/behaviorism.

Acknowledgments

The authors wish to thank everyone who provided feedback while developing projects using this framework. Basak Alper especially was influential during the early stages of development. This work was supported in part by NSF IGERT grant DGE-0221713.

References

[1] J. R. Abrial. Data semantics. In J. W. Klimbie and K. L. Koffeman, editors, *Data base management : proceedings of the IFIP Workshop Conference on Data Base Management*, pages 1–60, New York, 1974. American Elsevier.

[2] Adobe. Adobe flex. http://www.adobe.com/products/flex/, May 2010.

[3] Aristotle. *Poetics*. Translated by Ingram Bywater. Kessinger Publishing, 2004.

[4] S. Bateman, R. Mandryk, C. Gutwin, A. Genest, D. McDine, and C. Brooks. Useful junk? the effects of visual embellishment on comprehension and memorability of charts. In *CHI 2010: Proceedings of the SIGCHI conference on Human factors in computing systems*, 2010.

[5] S. Bertschi and N. Bubenhofer. Linguistic learning: a new conceptual focus in knowledge visualization. In *Ninth International Conference on Information Visualisation (IV'05)*, pages 383–389, 2005.

[6] R. Brath. Use of Analogy in Synthesizing Novel Visualizations. In *Information Visualisation, 2008. IV'08. 12th International Conference*, pages 481–484, 2008.

[7] R. Burkhard. Towards a framework and a model for knowledge visualization: synergies between information and knowledge visualization. *Lecture notes in computer science*, 3426:238, 2005.

[8] S. K. Card. Information visualization. In *The Human-Computer Interaction Handbook*. CRC Press, 2007.

[9] E. Chi, J. Konstan, P. Barry, and J. Riedl. A spreadsheet approach to information visualization. In *Proceedings of the 10th annual ACM symposium on User interface software and technology*, page 80. ACM, 1997.

[10] B. Craft and P. Cairns. Directions for Methodological Research in Information Visualization. In *Proceedings of the 2008 12th International Conference Information Visualisation*, pages 44–50. IEEE Computer Society Washington, DC, USA, 2008.

[11] Cycling '74. Max/msp/jitter. http://cycling74.com, November 2009.

[12] D. H. Eberly. *3D game engine architecture: engineering real-time applications with Wild Magic*. Morgan Kaufman Publishers, Amsterdam, 2005.

[13] K. Einsfeld, A. Ebert, and J. Wolle. Hannah: A vivid and flexible 3d information visualization framework. *International Conference on Information Visualisation*, pages 720–725, 2007.

[14] J. Fekete. The infovis toolkit. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 167–174. IEEE Computer Society Washington, DC, USA, 2004.

[15] Flickr. Flickr services. http://www.flickr.com/services/api/, June 2010.

[16] A. G. Forbes. Coil maps. In *Proceedings of the Workshop on Media Arts, Science, and Technology (MAST): The Future of Interactive Media,*, Santa Barbara, CA, January 2009.

[17] B. J. Fry. *Organic information design*. Master's thesis, Massachusets Institute of Technology, 1997.

[18] S. Gothel. Java binding for the opengl api. http://github.com/sgothel/jogl, March 2010.

[19] D. A. D. Gould. *Complete maya programming*. Morgan Kaufmann, 2003.

[20] J. Heer, S. K. Card, and J. A. Landay. prefuse: a toolkit for interactive information visualization. In *CHI 2005: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 421–430, New York, NY, USA, 2005. ACM.

[21] R. Laramee and R. Kosara. Challenges and unsolved problems. In *Human-Centered Visualization Environments*, pages 231–254. Springer, 2007.

[22] A. Lau, A. Moere, et al. Towards a model of information aesthetics in information visualization. In *Proceedings of the 11th International Conference Information Visualization*, pages 87–92. Citeseer, 2007.

[23] G. Legardy and A. G. Forbes. Cell tango. http://www.mat.ucsb.edu/~g.legrady/glWeb/Projects/celltango/cell.html, March 2010.

[24] Z. Lieberman, T. Watson, and A. Castro. Openframeworks. http://www.openframeworks.cc, November 2009.

[25] C. MacGillivray. Slices of Time-Appraising the Use of Dynamics in Design. In *Proceedings of the 2009 13th International Conference Information Visualisation-Volume 00*, pages 598–604. IEEE Computer Society, 2009.

[26] Microsoft. Windows forms / windows presentation foundation. http://windowsclient.net/wpf/, May 2010.

[27] T. Munzner. A Nested Model for Visualization Design and Validation. *IEEE Transactions on Visualization and Computer Graphics*, 15(6), 2009.

[28] C. Reas and B. Fry. Processing home page. http://www.processing.org, September 2009.

[29] W. Schroeder, K. Martin, and W. Lorensen. The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In *Proceedings of the 7th conference on Visualization'96*. IEEE Computer Society Press Los Alamitos, CA, USA, 1996.

[30] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proc. Of the 1996 IEEE Symposium on Visual Languages, IEEE Computer Society, Washington, DC*, pages 336–343, 1996.

[31] B. Shneiderman. Creativity support tools: accelerating discovery and innovation. *Communications of the ACM*, 50(12):32, 2007.

[32] Y. B. Shrinivasan and J. J. van Wijk. Supporting the analytical reasoning process in information visualization. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 1237–1246, New York, NY, USA, 2008. ACM.

[33] P. S. Strauss. Iris inventor, a 3d graphics toolkit. *ACM SIGPLAN Notices*, 28(10):192–200, 1993.

[34] Sun Microsystems. Java standard edition 6 api specification. http://java.sun.com/javase/6/docs/api/, November 2009.

[35] D. F. Swayne, D. Temple Lang, A. Buja, and D. Cook. GGobi: evolving from XGobi into an extensible framework for interactive data visualization. *Computational Statistics & Data Analysis*, 43:423–444, 2003.

[36] J. Thomas and K. Cook. Illuminating the path: The research and development agenda for visual analytics. *IEEE Computer Society*, 2005.

[37] VVVV group. Vvvv: a multipurpose toolkit. http://vvvv.org, November 2009.