Navani: Navigating Large-Scale Visualisations with Animated Transitions

Kevin Pulo
APAC National Facility and ANU Supercomputing Facility
Australian National University, Acton, ACT, 0200, Australia
kevin.pulo@anu.edu.au

Abstract

When visualising datasets that are too large to be displayed in their entirity, interactive navigation is a common solution. However, instantaneous updates of the visualisation when navigating can result in disruption to the user's mental map. Animated transitions are one way of addressing this problem. This paper presents the Data–Model–View–Controller (DMVC) architecture for navigation-based interactive systems. Navani, a software framework based on DMVC for supporting animated transitions during navigation, is presented, along with a sample application of it to hierarchical data.

Keywords— **Animated transitions, navigation, large-scale visualisation, Navani, Data-Model-View-Controller**

1 Introduction

As the size of datasets continue to increase, their visualisation becomes increasingly difficult. Datasets are frequently much too large to be visualised in their entirity. Such large-scale visualisation necessitates the visualisation of subsets of the data, combined with interactive navigation. The navigation allows the user to change the subset of the data that is being visualised.

This introduces the problem of minimising the disruption to the user's mental map across navigation transitions. Navigation, by its very nature, requires changing the information that is displayed. However, simply updating the visualisation instantaneously is rarely adequate, as it results in visual elements appearing and disappearing suddenly, and the changing of the placement and visual relationships of the visual elements. This has been shown to adversely affect the user's *mental map*, and the user's performance can suffer if the mental map is not preserved [1, 2].

Immediately updating displays are commonplace in graphical applications. One simple example of this is *JDiskReport* [3], a freeware application to graphically show the disk space usage of directories in a filesystem. Users can navigate between directories in a variety of ways, and can view the usage in a variety of formats, but in all cases the on-screen changes are instantaneous. This problem is not limited to small, free utilities. For example, *Tom Sawyer Visualisation (TSV)* [4] is a sophisticated interactive graph editor and toolkit, and when coupled with *Tom Sawyer Layout* it also includes advanced graph drawing algorithms. Yet TSV does not support animated transitions of graph layouts.

This naïve approach shows no regard for preservation of the user's mental map, as such, software which behaves in this manner can be difficult and frustrating to use. Unlike software which is poorly designed (for example, with an unintuitive graphical user interface layout), it is generally not possible for users to be able to "learn" to deal with the loss of their mental map. If the display is completely updated and rearranged from the previous view of the data, then users have no choice but to completely rebuild their mental map, which requires both time and effort.

A better approach is to use *animated transitions* to help preserve the user's mental map [5, 6, 7]. Since the human visual perception system is naturally good at tracking motion, it makes sense to use motion when updating a visual display. This allows the user to offload the cognitive load of comprehending the changes to their perceptual system, which means that they are more easily able to follow the changes taking place.

Horizontal and vertical scroll bars are a good example of a simple user interface component that benefit from animation to preserve mental map. For example, if the view is shifted downwards by half a page instantaneously, then the user's only cue on the direction of the change is their action that initiated it. This issue is easily observed when viewing a display that is being rapidly scrolled up or down by another person. Since the observer does not know whether the user has chosen to scroll up or down, they have no cues at all regarding the direction of the change. Thus, when the scrolling is rapid, it is very easy for the observer to feel "lost", as they do not have time between scrolling operations to rebuild their mental map. However, if each scrolling operation results not in an instantaneous update of the display, but a brief animation that *directly shows* the up

or down movement of the displayed information, then this problem is alleviated [8]. The speed of the animation is likely to play a role, but the subject of the optimal speed of arbitrary animations has not received much attention in the literature [9, 10]. Furthermore, there may be superior ways of providing temporal context during scrolling (for example, by using afterglow effects [11]), but animated transitions are certainly a straightforward way that is an improvement on an instantaneous change.

Operating systems such as Apple's MacOS X also use animated transitions when performing operations on windows such as opening, closing, minimising and restoring, as well as when presenting an overall view of all windows with the Exposé feature [12]. In addition to being visually appealing, these animations help to inform the user of the relationships between the on-screen visual elements before and after the operation.

Despite these advantages, animation is not a panacea; benefits are not automatically obtained by arbitrary animation of transitions. The animation must be well-designed and the changes presented should be minimised. Constructing an optimal animation for a given change is a non-trivial exercise [13]. Further, animation is not always appropriate and to be effective they must be well-conceived [14]. Indeed, exceedingly poor animations are likely to be actively misleading; that is, worse than no animation.

This paper has three contributions. First, the *Data–Model–View–Controller (DMVC)* architecture is presented as a way of structuring navigation-based interactive systems. Second, *Navani* is presented, which is a software framework supporting navigation and animated transitions in large-scale visualisations. Navani does not address the evaluation of navigation or animation strategies, but provides a toolkit for animation researchers to use in their investigations. Finally, *HiePie* is presented as a sample application of Navani to hierarchical data.

2 Animation model

The animation and data model used by Navani is based on the Model–View–Controller (MVC) paradigm [15], which is a commonly found architecture for modelling and implementing interactive graphical systems. In the traditional MVC architecture, the *Model* is an abstract representation of the underlying data, the *View* is the visual or graphical representation of the Model which is presented to the user, and the *Controller* is the user interface, that is, how the user is able to adjust the Model and View. The MVC is illustrated in Figure 1.

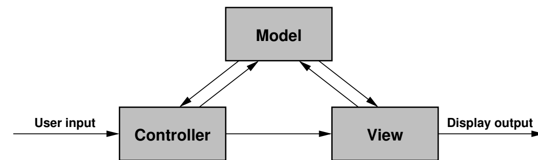


Figure 1: The Model–View–Controller (MVC).

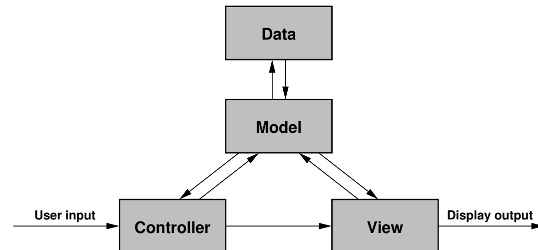


Figure 2: The Data–Model–View–Controller (DMVC).

However, the MVC is insufficient to adequately describe the operation of large-scale visualisation, since the Model cannot be visualised in its entirity by the View. It is not appropriate to include a visualisation subset within the Model, since the purpose of the Model is to be abstractly independent of any particular visualisation method. Furthermore, this would mean that simple navigation operations would be "changing" the Model, which is counter-intuitive. Similarly, adding the subset to the View is not appropriate because we would like to be able to try many different models of navigation with any given visual representation (that is, with any View).

This paper addresses this issue by presenting the *Data–Model–View–Controller (DMVC)* architecture, which is an extension of the MVC. The *Data* is the entire overall dataset, that is, what was considered the Model of the MVC. For large-scale visualisation, the Data cannot be directly visualised, and so the *Model* in the DMVC is a smaller representation of the Data, one which can be appropriately visualised. If the data presented in the Model is updated by the user, then these changes are propagated to also update the Data. This terminology is somewhat more natural, since the word "model" means a smaller representative object containing less information. This architecture is illustrated in Figure 2. The DMVC is a generalised version of the "*Zoomed Model*" presented in Structural Zooming [7].

3 Software framework: Navani

This section presents Navani, a software framework written in Java for supporting animated transitions of large-scale visualisation. Navani abstracts away the low-level details of providing animated transitions, allowing the information visualisation researcher or software developer to concentrate on the development and evaluation of the actual visualisation and transition methods.

Navani is written in Java2, and by default uses Java2D for its graphics display. Java was chosen as it is reasonably cross-platform and web-enabled, which can be important when evaluating visualisation techniques.

Navani can function as an application or a web-applet.

The `navani.Panel` class is the core user interface class. It extends `JPanel` to provide an area in which to display the visualisation, and provides another `JPanel` object for control panels and options widgets. The controls can be placed anywhere, for example in a separate window, but by default they are added underneath the main Panel area. The Panel contains objects for each of the Data, Model, View and Controller aspects of the DMVC model.

3.1 DMVC implementation

This section describes the implementation in Navani of the Data–Model–View–Controller architecture.

3.1.1 Data. The `navani.Data` class represents the overall data structure that is being visualised. It is inherited by classes that provide the necessary storage and methods for specific types of Data objects, for example, a graph or tree structure. The Data can be defined either explicitly, where some or all of the data is resident in memory, or it can be implicit, where the data structure is generated as it is needed (for example, an infinite graph or tree structure). Individual data elements in the data structure should implement the `navani.DataElement` interface, to allow them to be used by other aspects of Navani. Finally, the Data class implements a canonical Listener design pattern [16], allowing other classes that implement the `DataUpdateListener` interface to receive `DataUpdateEvent` objects whenever the underlying data is modified.

3.1.2 Model. The `navani.Model` class represents the Model, that is, the subset of the Data that is currently being viewed by the user. Each specific Data class may have several possible Models, each with a different way of representing the concept of a "subset" of the data. For example, graph data may have a model that is a subset of the nodes, with edges between nodes in the subset implicitly included, or an alternative model might be a subset of the edges, with the adjacent nodes implicitly included. The Model also includes the representation of the user's *navigation history*, that is, not only the current subset of the data, but also previous subsets. This allows the user to return to previous views, by undoing and redoing navigation operations. The Model class also implements the Listener pattern [16], allowing classes implementing the `ModelUpdateListener` interface to receive `ModelUpdateEvent` objects when the model changes. There is a `ModelUpdateEvent` sub-class for each type of navigation operation that is supported by a particular Model class.

3.1.3 View. The `navani.View` class represents the View, that is, the actual visual display of an instance of the Model. Thus, a Model may have many possible visual representations, all of which are simply different ways of displaying the same information. For example, a tree may be drawn in a traditional node-link style, where tree nodes are typically boxes and lines are used to join child nodes with their parents, or it may be drawn in the inclusion style, where the parent-child relationship is visually represented by the child nodes being wholly within the region of the parent node.

A View is simply a collection of `navani.-VisualForm` objects, each of which is a single visual entity. Typically, VisualForm objects are used to provide a visual representation of a DataElement. The VisualForm class is abstract, and must be sub-classed in order to provide visual ways of representing particular types of data. The current implementation of VisualForm stores a collection of Java2D Shape objects, along with a Color and Label. The `navani.VisualNode` subclass adds a reference to a DataElement object.

The `navani.VisualFormFactory` class utilises the Factory design pattern [16] to create VisualForm objects of a certain type. It is abstract, and subclasses must return a collection of VisualForm objects for a given collection of DataElement objects and an on-screen area in which to represent the data. Typically there is only one VisualFormFactory for each VisualForm type, but of course there may be several ways of creating VisualForms of a particular type.

3.1.4 Controller. The `navani.ViewTransitioner` class represents the Controller, that is, the algorithm for responding to navigation operations initiated by the user. It is also an abstract class, and subclasses are notified of ModelUpdateEvents when they occur. Each different view style may have many possible ways of animating transitions between views, each of which is encapsulated in a ViewTransitioner class. The simplest ViewTransitioner is the `NoAnimationTransition` class, which does not attempt to perform any animation at all, rather, it merely displays the updated view immediately. This is mostly useful for demonstrating how the mental map is easily lost in the absence of animation.

To easily support animation, VisualForm subclasses must implement an `intermediate()` method. This method takes as input the initial and final VisualForm objects and a fraction $0 < f < 1$, and returns a VisualForm object which is f partway between the initial and final VisualForms. The abstract `navani.AnimationStrategy` class specifies the type of animation that is to be performed, such as linear interpolation by `navani.AnimationStrategyLinear`. It

provides similar `intermediate()` methods for primitive numeric types, namely the floating-precision (`float`, `double`) and integer (`int`, `long`) types, as well some higher-level types such as two dimensional shapes (`Point2D`, `Line2D`, `Rectangle2D` and so on), RGB colours (`Color`), and so on. Since all of the parameters representing visual forms are composed of these basic numeric types, their `intermediate()` methods can be built up from those provided by the AnimationStrategy object. For the linear animation strategy, the `intermediate()` methods for the numeric types simply take the familiar form

$$intermediate = inital + fraction * (final - initial)$$

Other AnimationStrategy classes are possible for other types of animations, such as "slow-in"–"slow-out" animations, rectilinear motion of shapes in 2-space, etc.

Animated transitions are represented by the abstract `navani.ViewChange` class, which supports obtaining intermediate `navani.View` objects. The `navani.ViewChangeSingle` class defines a single animation between an initial and a final View, the `navani.ViewChangeMulti` class defines an animation in terms of a sequence of other ViewChange objects, and the `navani.ViewChangeReverse` class is the time-reversed version of another ViewChange object. These allow complex animations to be easily built up out of compositions of simpler ones.

3.2 Support classes

Navani features an assortment of other classes for supporting animations.

The abstract `ViewAnimator` class actions an animation with a given number of frames. The `ViewAnimatorTimed` subclass displays an animation with a delay between each frame to control the speed, while the `ViewAnimatorInteractive` subclass uses GUI buttons to step through the animation, which can be useful for debugging animated transitions.

The abstract `ColorMapper` class, and its two subclasses `ColorMapperPalette` and `ColorMapperColormap`, allow the color of VisualForm objects to be chosen systematically.

The `DataElementComparator` class allows data elements to be compared, which allows them to be sorted according to some property. The `DataElementVisualAttributes` provides per-data-element storage of persistent visual attributes, which, for example, can be used to ensure that the colour of each data element never changes.

4 Sample application: Hierarchical data

This section presents the application of Navani to the visualisation of hierarchical data. This application

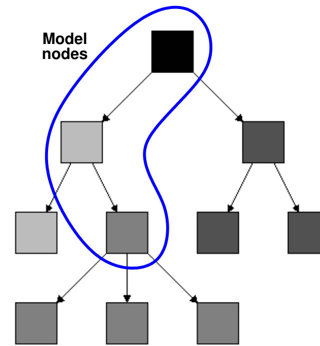


Figure 3: The tree-based Model used by HiePie.

is a work-in-progress named *HiePie*, and was the motivating driver behind Navani. It is concerned with visualising the disk space used in a hierarchical file system, for the purposes of quickly and easily locating possible files and directories to remove or relocate when free disk space becomes low. It is also expected to be a useful case study for research into navigation of large-scale visualisation with animated transitions.

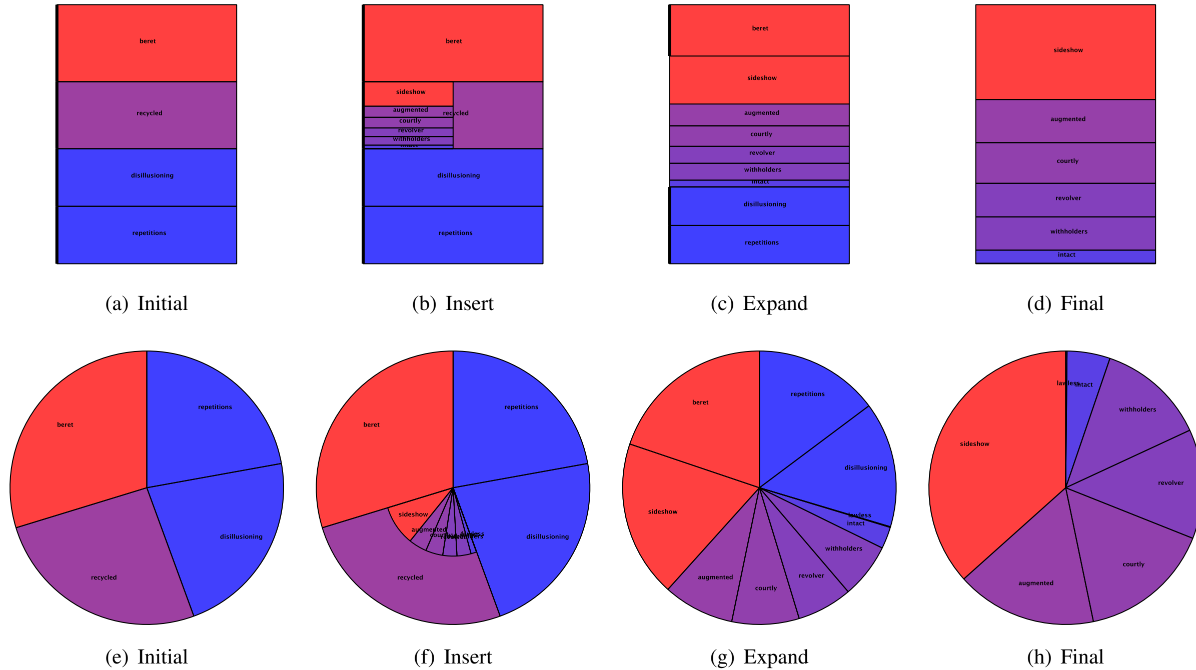
4.1 Data, model and view

Since the data for HiePie is hierarchical, it is represented by a tree structure. The `hiepie.Tree` class inherits from `navani.Data`, and the tree nodes are represented by the `hiepie.TreeNode` class, which inherits from `navani.DataElement`. Each TreeNode uses a `hiepie.Entry` object to decouple the filesystem-specific data from the tree structure.

The `hiepie.Model` class inherits from `navani.Model`, and defines a subset of the data to be the nodes between an identified *focus node* and the root node. This is illustrated in Figure 3. This model will be generalised in the future as HiePie is improved, however it serves the current purpose of demonstrating the usefulness of defining a Model of the Data.

The standard `navani.View` class, which keeps a collection of VisualForm objects, is sufficient for HiePie and does not require subclassing.

There are two visual representations of a TreeNode: a "*Bar*" form and a "*Wedge*" form. The Bar form, defined by `hiepie.Bar` and generated by `hiepie.BarFactory`, is shown in Figure 4(a). It represents the children of the tree node by segments in a rectangle, where the size of each segment corresponds to the size of file or directory in the filesystem. The Wedge form, defined by `hiepie.Wedge` and generated by `hiepie.WedgeFactory`, is shown in Figure 4(e). This is similar to the bar form, except that each child is a wedge in a circular pie chart. Both the Bar and Wedge forms inherit from a common parent,



(a) Initial (b) Insert (c) Expand (d) Final

(e) Initial (f) Insert (g) Expand (h) Final

Figure 4: The stages of the animation for the HiePie "*AddThenGrow*" controller, used with the Insert addition method. Figures (a)–(d) show the animation for the Bar visual form, while Figures (e)–(h) show the Wedge visual form. The colour indicates the overall size of the tree node (including children), with red the largest and blue the smallest.

`hiepie.VisualNode`, which is an abstract representation of these two forms. It contains an origin (x, y) point, an offset start, a size, and an orthogonal size. For the Bar, the origin is the top-left corner of the box, the offset start and size are in vertical direction, and the orthogonal size is the horizontal width. The Width is very similar, except it is in polar coordinates — the origin is the center of the circle, the offset start is the angle at which the wedge begins, the size is the angular width of the wedge, and the orthogonal size is the radius. Thus, the only difference between the two is the visual interpretation of these parameters.

4.2 AddThenGrow animated transition

The `hiepie.AddThenGrow` controller transitions from a node to its child in two stages. First, the child nodes are added within the region of the node, and second, the child nodes are expanded until they occupy the full display. Three node addition methods are supported, *Appear*, *Fade-In* and *Insert*. Appear simply makes the child nodes appear instantaneously, Fade-In gradually fades in the colours of the child nodes, and Insert "slides" the child nodes in orthogonally. Figure 4 shows an intermediate frame in the addition and expansion stages for the Bar and Wedge forms, as well as the final view. Video files comparing all of these animated transitions are available on the web [17].

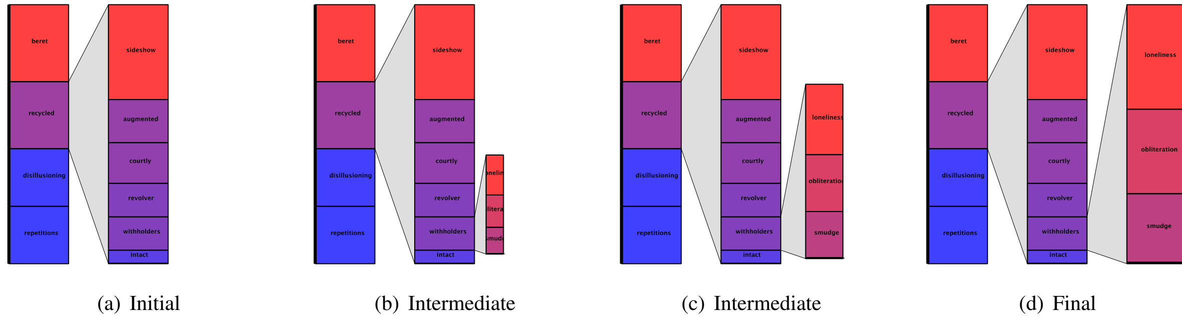
Note that the same AddThenGrow controller class is able to work with both the Bar and Wedge forms, since they are both abstractly represented by VisualNode.

4.3 BarExplorer animated transition

The AddThenGrow controller is an example of a simple animated transition facilitated by Navani. A more advanced controller is `hiepie.BarExplorer`, which is similar to the filesystem explorer in Mac OS X. It is shown in Figure 5, and a video file showing the animation is available on the web [17]. This method is similar to AddThenGrow, except that the new focus node is expanded horizontally alongside its parent. This allows the user to see several nodes concurrently. Context is provided by tethering children to parents by the `navani.VisualGlue` class. This class is a subclass of `navani.VisualForm` that draws a trapezoid between the bounding boxes of two groups of VisualForm objects. When there is no remaining horizontal space, the display scrolls to the right.

5 Conclusions

Animated transitions are a useful way of approaching the problem of preserving the user's mental map during navigation in large-scale visualisations. This paper has presented Navani, a software framework for



(a) Initial (b) Intermediate (c) Intermediate (d) Final

Figure 5: The stages of the animation for the "*BarExplorer*" controller for the Bar visual form.

supporting the developers of animated transitions. Navani is written using the Data–Model–View–Controller (DMVC) architecture. HiePie, a sample application of Navani to hierarchical data, was presented.

It is clear that the animated transitions shown in HiePie are of varying quality; for example, it seems that the additional context offered by BarExplorer provides benefits over AddThenGrow. However, the emphasis of this paper is not the quality of these particular animated navigation techniques, but rather the way in which Navani has facilitated their development and implementation. Navani removes the burden of implementing low-level animation details, thus allowing the visualisation designer to focus directly on the animated transitions. The designer is free to get on with the job of developing, evaluating and improving the animated navigation.

As such, the relative merits (or otherwise) of the specific HiePie animated navigation techniques presented in this paper will be explored in future work that is supported by Navani. For example, one disadvantage of BarExplorer is its linear nature. It is expected that a method which can provide similar animated navigation for the Wedge form will allow non-linear exploration of the tree in the plane.

References

[1] P. Eades, W. Lai, K. Misue, and K. Sugiyama, "Preserving the mental map of a diagram," in *Proc. Compugraphics '91*, pp. 34–43, 1991.

[2] K. Misue, P. Eades, W. Lai, and K. Sugiyama, "Layout adjustment and the mental map," *J. of Visual Languages and Computing*, vol. 6, no. 2, pp. 183–210, 1995.

[3] Karsten Lentzsch, "JDiskReport." `http://jgoodies.com/freeware/jdiskreport/`, 2007.

[4] Tom Sawyer Software, "Tom Sawyer Visualisation (TSV) version 6.0." `http://www.tomsawyer.com/home/products.php`, 2006.

[5] B. B. Bederson and A. Boltman, "Does animation help users build mental maps of spatial information?," in *Proc. of IEEE Symp. on Info. Vis. (Info Vis '99)*, pp. 28–35, 1999.

[6] L. R. Bartram, *Enhancing Information Visualization with Motion*. School of Computing Science, Simon Fraser University: PhD thesis, 2001.

[7] K. J. Pulo, *Structural Focus + Context Navigation of Relational Data*. University of Sydney: PhD thesis, 2004.

[8] C. Klein and B. B. Bederson, "Benefits of animated scrolling," in *Conf. on Human Factors in Computing Systems (CHI '05)*, pp. 1965–1968, 2005.

[9] A. Wallace, J. Savage, and A. Cockburn, "Rapid visual flow: How fast is too fast?," in *Proc. 5th Aust. User Interface Conf. (AUIC2004), Dunedin, New Zealand*, pp. 117–122, CRPIT Vol 28, A. Cockburn, ed, 2004.

[10] M. Lind and A. Kjellin, "Faster is better: Optimal speed of animated visualizations for decision makers," in *Proc. 9th Intl. Conf. on Info. Vis., IV05*, pp. 896–900, IEEE Computer Society, 2005.

[11] P. Baudisch, D. Tan, M. Collomb, D. Robbins, K. Hinckley, M. Agrawala, S. Zhao, and G. Ramos, "Phosphor: explaining transitions in the user interface using afterglow effects," in *Proc. 19th ACM symp. on User interface software and tech. (UIST)*, pp. 169–178, 2006.

[12] Apple Computers, Inc, "Mac OS X Exposé." `http://www.apple.com/macosx/features/expose/`, 2003.

[13] C. Friedrich, *Animation in Relational Information Visualization*. University of Sydney: PhD thesis, 2002.

[14] B. Tversky, J. B. Morrison, and M. Betrancourt, "Animation: can it facilitate?," *Intl. J. Human-Computer Studies*, vol. 57, pp. 247–262, 2002.

[15] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view controller user interface paradigm in Smalltalk-80," *J. Object-Oriented Prog.*, vol. 1, no. 3, pp. 26–49, 1988.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Reading, Mass.: Addison-Wesley, 1995.

[17] Kevin Pulo, "HiePie accompanying video files." `http://www.kev.pulo.com.au/hiepie/video/`, 2007.