Appendix B

Appendix to Instantiable Visualization Techniques Framework (Chapter III)

B-1 Comparison with Previous Frameworks

Part of our framework, namely the four transformation phases that correspond to the visualization generation process (Figure II-1) is very similar to Card et al's [*Card, 1999*] reference model of visualization. Both works were developed in parallel. The main difference between our visualization creation process and Card's visualization reference model is that we have an additional step of graphical transforms. This allows us to model changes in the visual structure of the visualization that is not based on any underlying data concepts, e.g. showing state information such as selection highlighting. The three classes of objects namely data, visual structure and views considered by Card also corresponds to our three realms of data, graphical scene and output media.

The four transformation classes, however, only consist of a part of our framework. In our work we define a visualization technique to have an object selection and transformation function (*ODT model*). This model is different from any other previous frameworks. Our *ODT model* is flexible because it allows us to build techniques that can create visualization interfaces from scratch or modify exiting visualizations. By including an object definition phase before transformation, we allow any type of objects to be piped into the transformation component and as a result we can build techniques that contain transformation functions that come in any order (i.e. they do not need to follow the data → mapping → graphical → rendering phases in the visualization generation process). Visualization techniques in our framework also need not contain functions from all four classes. In addition none of the previous frameworks include a compositional syntax (chapter II-2).

Another related framework from visualization techniques was presented by Tweedie [*Tweedie, 1997*]. In her framework, Tweedie described the differences in visualization techniques by using four primary criteria: data, representation, interactivity and input/output externalizations.

For example the dynamic query technique and the Table Lens technique are described in Figure B-1:

Dynamic Queries
Purpose: Find useful sets of multivariate data
Data type: Values
Representation: A scatterplot is used to display two of the attributes, the remainder are represented as sliders.
Interactivity: Data is hidden (mechanized DM) or filtered (mechanized IM) by selecting ranges on sliders.
I=O representation: Input → Output is represented

Permutation Matrices/Table Lens
Purpose: view relations in multivariate data
Data type: values
Representation: This is essentially a graphical spreadsheet (value in each cell is encoded as height)
Interactivity: Reorder the cells (mechanized DM)
I-O Representation: Only output is represented

Figure B-1: Tweedie's description of the dynamic query and Table Lens techniques

Tweedie's framework was not very appropriate for our goals in automatic design, however, because we needed a fully instantiable language of visualization techniques i.e. the descriptive language must be complete and specific enough to generate an active visualization interface. Thus unlike Tweedie's approach we needed to describe the function and structure within each technique in much greater detail. This however does not detract from our ability to analyze and categorize the various techniques as we showed in chapter II-3.

A very desirable property of our instantiable language is that it provides a common level of primitives for describing visualization techniques (as was laid out in chapter III-1). This allows us to break down high-level visualization systems and compare their capabilities at the same level of granularity. This was not true of previous frameworks [*Tweedie, 1997*], which sometimes compared visualization systems that differ in their level of granularity. For example the Table Lens and dynamic query slider techniques shown in Figure B-1 are both at two very different levels of granularity. The Table Lens system consists of multiple different technique including an attribute value sorter, a lens technique that allows users to change the size of cells, a semantic zoom technique that changes the level of detail on elements depending on their cell size, and a column move technique. In contrast the dynamic query technique is a single technique in itself. Our framework highlights such distinctions.

In our framework we have descriptions comparable to Tweedie's data, representation and interaction categories. We however, chose to separate out "goodness" measurements of input and output externalizations from our framework because this category more pertains to the effectiveness of a design or a technique rather than to describing the structure or components within a technique. Effectiveness measurements was first introduced by Mackinlay [*Mackinlay1986a, 1986b*] and in this work we extend effectiveness criteria to cover data processing and mapping designs as well as interactive methods (chapter IV).

Both Card's and Tweedie's frameworks are functional frameworks that try to capture the semantics of visualization techniques. There are also a set of instantiation languages including the Data Explorer, Iris Explorer, and AVS systems [*Brodlie, 1991*]. These systems provide an instantiable syntax and allow designers to create visualization interfaces by building data flow diagrams that convert sets of data values into visualization renderings and interface components on the computer system. These systems were also not sufficient for the goals of this thesis for several reasons:

1) They are based on scientific visualizations.
2) They are based on the use of data flow diagrams are used to create visualization systems from scratch. In our work we need to design techniques that generate new visualization interfaces from scratch as well as modify existing visualization designs. Techniques from the latter category cannot be described using any of the three instantiation packages (the Data Explorer, Iris Explorer, and AVS systems). For a more complete description of how our framework syntax differs from data flow diagrams refer to chapter II-4; 3).
3) They consist only of low-level primitives. In contrast our framework contains both a functional description as well as an instantiable description. In our work it is necessary to group and categorize techniques based on the higher more abstract functional level. This two-level description (functional and instantiation) allows us to modularize our designer so that it can initially only consider what functional primitives it needs to use to fulfill current goals. Specifics that may affect the effectiveness of a technique but not the core functionality can be considered later once we are sure that all the required function have been included. In addition, the category of the various current visualization techniques informs our designer what roles data, mapping, graphical and rendering transforms can play in the design process and how they may be usefully combined.

By using the lower level instantiable language we allow our automatic designer system to describe a visualization in sufficient detail so that it can ultimately render an active visualization interface. Unlike previous frameworks our visualization techniques language described technique in both the functional and instantiation levels using a compatible syntax as well as establishes a systematic process to move between the two different levels of abstractions: the functional level and the instantiation level. Previous frameworks either concentrate on one level or the other. We provide a common structure for representing both.

B-2 Data Flow Diagrams

It is important to note that even though our visualization technique specification may resemble data flow diagrams, they are not strictly data flows. Data flow diagrams are commonly used to analyze and understand complex systems consisting of multiple interacting processes. Our specification language, on the other hand, is meant to describe a single process (i.e. the visualization technique process), more like a high-level flow chart. The visualization technique process **does** interact with two other processes, namely the user and the system designer, as is shown in Figure B-2. In Figure B-2, the user provides *task inputs* to the technique through the use of input devices. This input may cause system state to be updated and ultimately produce *feedback* for users in the form of visual change. The graphic designer also interacts with the visualization technique by providing *default values* to the functions within it. While our specification language does capture these relationships, its primary purpose is to encapsulate the functionality of a visualization technique in enough detail so that a working system may be generated from it. In contrast, data flow diagrams are used primarily to understand the flows or exchange of information among different processes.

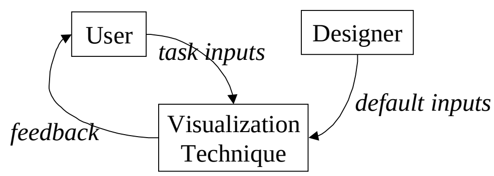


Figure B-2: Data Flow diagram showing relationships between a visualization technique, the user and the visualization designer

Other differences between our specification language and data flow diagrams include:

1. *No system state changes*: Unlike data flow diagrams we do not show system states in our specification because even though state information is important for understanding the way visualization techniques work, they are less important for capturing the functionalities that we want to perform using a technique. Including them in the specification diagram may significantly increase clutter and complexity. In any case, system state changes can be easily included in our specification diagrams as additional boxes, without changing the existing structure of the technique.

2. *Temporal links*: We have links that show flows of data from one function to another, as well as temporal links which indicate a temporal ordering between two functions (temporal links allow us to express that certain functions have to be performed before others during execution of the visualization technique). Data flow diagrams can only have data links.

B-3 Example: Generating an Instantiation Specification for Dynamic Query Sliders

This example shows how the abstract functional *dynamic query slider* [*Ahlberg, 1992*] design presented in chapter II can be augmented to form an instantiable visualization technique. Figure B-3 shows the functional specification for the dynamic query technique. The technique starts with a user-controlled, functional description, object definition function followed by a graphical transform. The first step of the instantiation augmentation process is to determine the exact functional description function and graphical transform function to use. For the functional description we use the *threshold function* with the "*greater-than*" (>) operator. Once the objects have been defined, we need to give feedback to the user on the results of the operation. One way to do this is to attach a common identifying feature to all the selected objects. This can be achieved by using the *assign* function to set a chosen graphical property of the selected object set to a common value. Figure B-4 shows the functional specification of Figure B-3 augmented with specific instances of object definition and transformation functions.

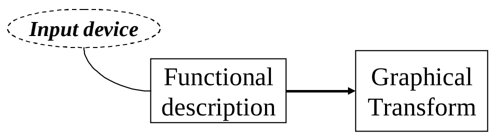


Figure B-3: Dynamic query functional specification

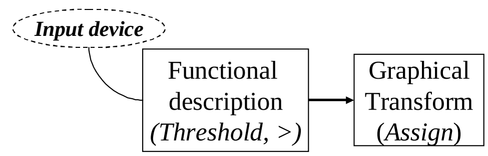


Figure B-4: Dynamic query specification with specific object-definition and transformation functions

Once the functions are chosen, we ensure that they connect correctly with one another. Sometimes translation functions must be inserted to ensure that the outputs of one function are appropriate as inputs for the following function(s). Figure B-5 shows the inputs and outputs (highlighted in gray) of the object definition and graphical transform functions that constitute the *dynamic query* technique. We start with the set of all data objects. From these objects we extract a set of data values which are fed into the *greater-than* threshold function. This function produces a set of boolean data values that are transformed into graphical values for the *assign* graphical transform function.

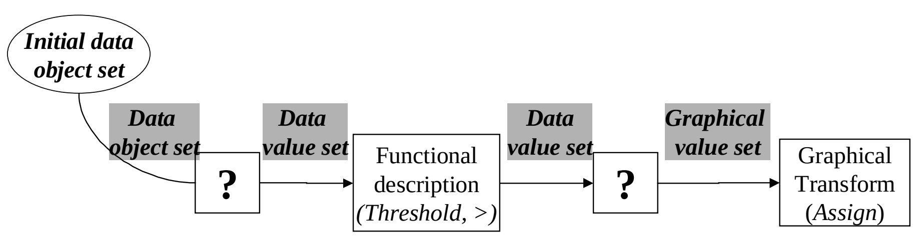


Figure B-5: Dynamic query specification with input and output types for each object definition and transformation function. The "?" boxes indicate areas where translation functions are needed to convert from one argument type to another.

Figure B-6 shows the translation functions used in this example to convert the output type of a source function to fit the input type of a destination function. The *get-values* function extracts a set of attribute values from the initial data object set. These values are fed through the threshold function, which produces a set of boolean values, indicating for each input value, whether it passed the chosen threshold. Based on these boolean values and the set of data objects considered by the *threshold* function, we identify all of the data concepts that passed the query (*boolean-to-object*). From these data concepts, we get all of the graphical objects that are used to represent them by using the *get-related-graphical-objects* function. Finally we extract the property values that we want changed from the graphical objects. These values are passed through the *assign* function that sets them to a common constant.

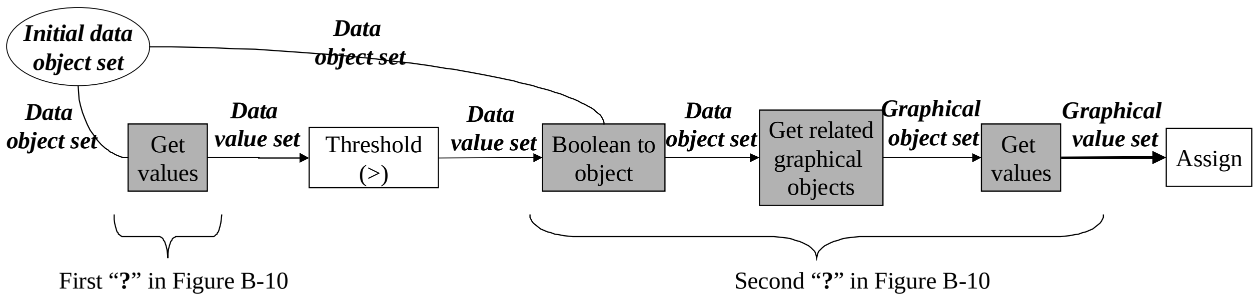


Figure B-6: Dynamic query specification with intermediate functions for inputs and outputs (Note that the visualization function classes are not shown in order to reduce the amount of diagrammatic clutter)

Figure B-7 shows the dynamic query specification with all currently unspecified function inputs in ***bold italicized*** text with light-gray background.

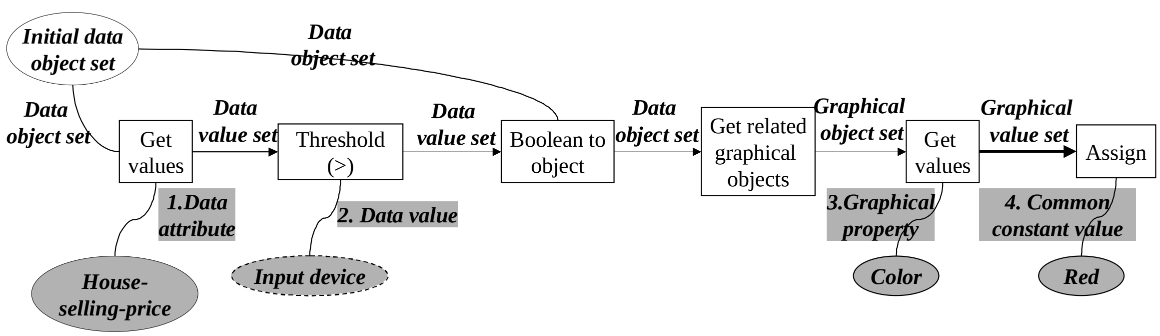


Figure B-7: Dynamic query specification with all inputs required

There are basically four necessary inputs: 1) the data attribute used to extract values for the threshold function, 2) the threshold value needed for the *threshold* object definition function, 3) the graphical property of the resulting search objects that we want to change, and 4) the graphical value we want to use as an identifying feature for all the search objects. For each of these inputs we must decide whether to provide default values (i.e. designer defined values), or whether to hook them up to an input device to get the needed values from the user. Hooking them up to a device will allow a user more flexibility in altering the functionality of the technique. On the other hand, using input devices increases the motoric load[1] of the user. In this example we have decided to provide default values for all the required input arguments except

[1] *Motoric load* (or *articulatory load*) refers to the physical effort expended by the user in manipulating physical devices such as the mouse, keyboard, or electronic pen. Example operations that result in motoric or physical effort include mouse clicks, mouse movement, key clicks, or gesturing with an electronic pen.

for the threshold value (argument 2). The default arguments are as follows: *house-selling-price* is the default search data attribute (argument 1), *color* is the feedback graphical property (argument 3), and, *red* is the common identifying color (argument 4). These arguments are shown in Figure B-7 as normal ovals.

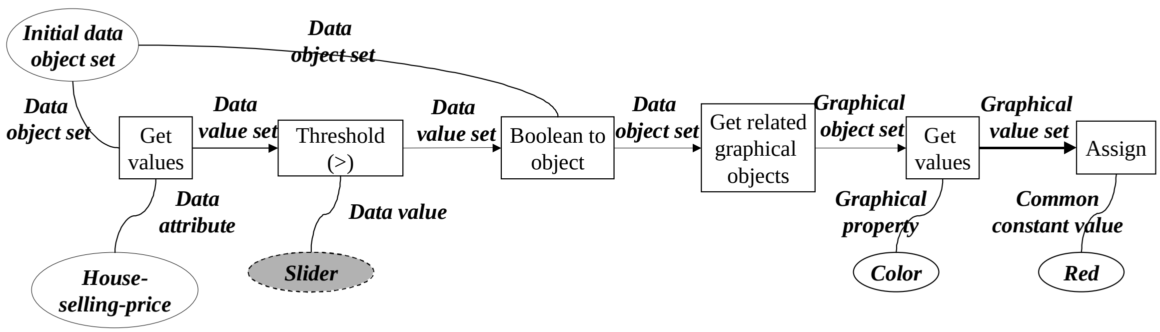
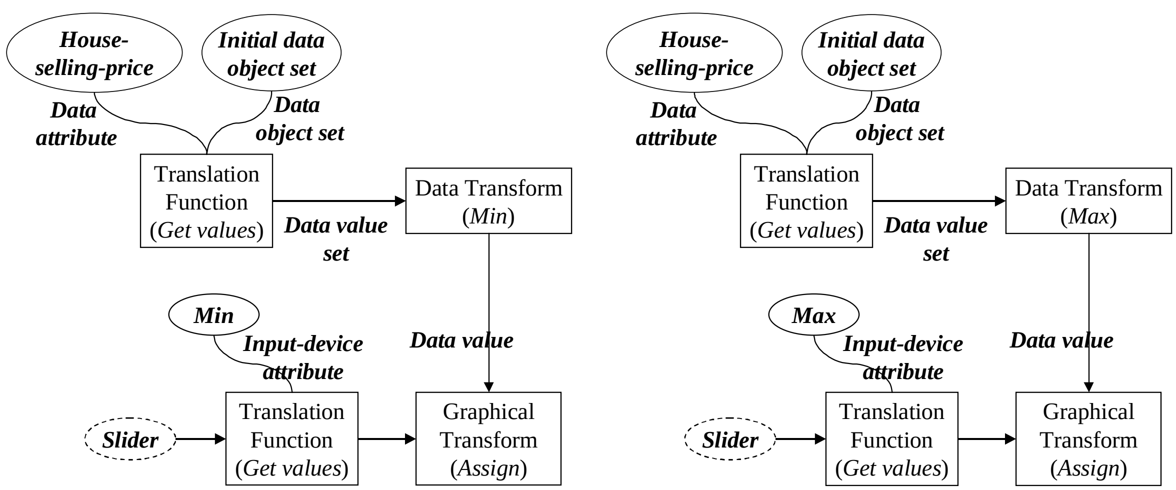


Figure B-8: Adding a slider input device for specifying the threshold constraint in the dynamic query technique

In the final steps we determine the input devices to use, and specify the initialization arguments for those devices. In this example we only have one user input value, namely the threshold value for the *greater than* object selection function. In the common dynamic query technique this input argument is attached to a *slider* input device (as is shown in Figure B-8).



(a) Specification for initializing the minimum range of the slider input device

(b) Specification for initializing the minimum range of the slider input device

Figure B-9: Initializing the *min* and *max* properties of the slider input device added in Figure B-8. The *min* and *max* values are derived by computing the *min* and *max* values of the *house-selling-price* data attribute with data transform functions.

A *slider* input device requires two initialization values, a minimum value and a maximum value, that defines the range of the slider. We set these values to be the minimum and maximum values of the *house-selling-price* attribute as is shown in Figure B-9. Initially we extract all the *house-selling-price* values from the entire data set by using the *get-values* translation function. Subsequently, we compute the *min* value using a data transformation function. This *min* value is then assigned to the *min* property of the *slider* input device which determines the minimum value on the slider range. A similar specification is used for assigning the maximum value on the slider range.

B-4 Systematic Exploration of the Instantiation Level of the Dynamic Query Slider Technique

In this section we explore the instantiation space for the dynamic query slider technique [*Ahlberg, 1994*] and discuss some of the more interesting design variations. We explore the instantiation design space by considering each of the five steps in the instantiation augmentation process (described at the beginning of this chapter III) and seeing for each step how a visualization technique can be varied:

1. Changing the specific functions used or adding more functions of the same type.
2. Changing the translation functions between object definition and transformation functions.
3. Changing how function arguments are provided (either by user or designer) as well as the default designer values.
4. Changing the type of input devices used within the design.
5. Changing how input arguments are provided to input devices.

B-4.1 Changing the Specific Functions Used or Adding More Functions of the Same Type

The first step of the instantiation augmentation process determines which specific object definition and transformation operators to use from the abstract classes described in the functional specification (e.g. functional description, graphical transform). By picking different instances of object definition and transformation functions we may generate a range of slider techniques. Figure B-10 shows the sets of alternative operators that may be used for each function class in the abstract functional specification. Different operator combinations affect the usefulness or effectiveness of the resulting technique. For example, in the slider technique shown previously, the *assign* graphical transform is used as a feedback mechanism to set the color for a group of selected or focus distributors to a perceptually salient value, e.g. *red*. Another alternative is to use the *addition* graphical transform to provide feedback by adding a constant value to the *x-position* of the focus objects thus shifting them to the right of the map. The first alternative allows users to pre-attentively see the selected objects without losing any positional context between the focus objects and the other elements in the visualization, unlike second alternative. Nevertheless the first alternative is also more susceptible to object occlusion. The second alternative like the SDM system [*Chuah, 1995*], allows users to move a set of focus objects up to the front or up above so that they can be clearly seen without occlusion and without the noise from surrounding objects.

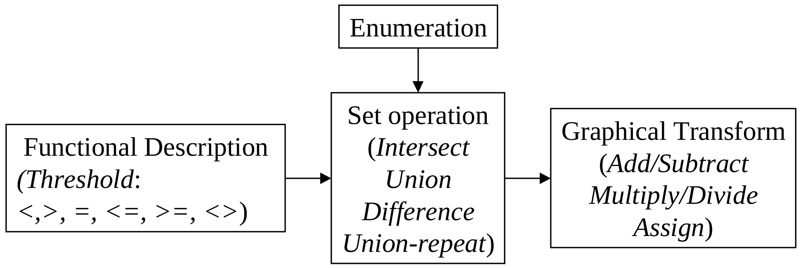


Figure B-10: Alternative object definition and transformation functions for the dynamic query slider technique

We can also experiment with expanding a function class by composing it with other functions from the same class. I.e. using multiple functional description operators or graphical transforms instead of just one. To ensure that the functional description of the technique remains unchanged however, we must only add operators that share the same general goal (e.g. *computation, summarization, feedback,* or *readability*) as the expanded operator. For example, in Figure II-13 we can use the *addition* graphical transform to alter the size of objects as well as the *assign* graphical transform to alter the color of objects. This design variation does not change the functional goal of the technique because both *assign* and *addition* functions are graphical transforms, and both functions are used for the same general goal, which is to provide *feedback* on a set of focus distributors.

B-4.2 Changing the Translation Functions between Object Definition and Transformation Functions

The second step of the instantiation augmentation phase incorporates translation functions into the design specification to ensure that the outputs of a function match the inputs of subsequent connecting functions. By using different combinations of translation functions we can vary the visualization technique design. In Figure B-11 we have enriched the dynamic query slider specification with object definition, transformation and translation functions. One variation on this design is to change the translation functions so that the *intersect* function is applied to data objects instead of graphical objects as in Figure B-12.

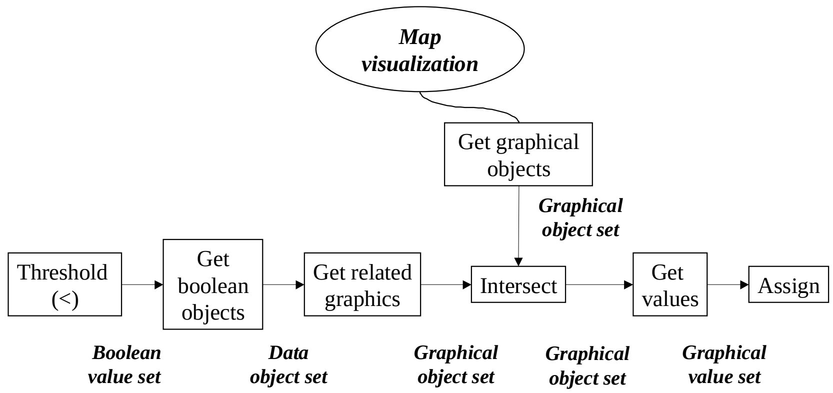


Figure B-11: Specification for the dynamic query slider technique including object definition, transformation, as well as translation functions

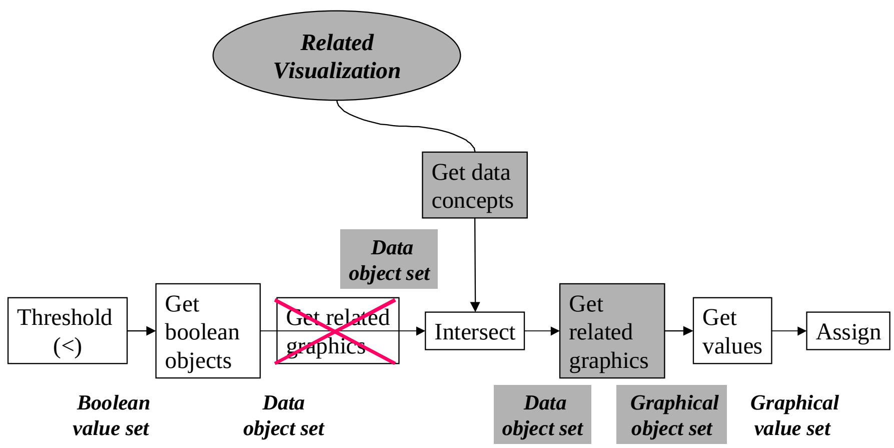


Figure B-12: In this specification we change the translation functions leading into the intersect operator so that the intersect operation is applied to data concepts instead of graphical objects as was the case in Figure B-11. Here, only those data concepts that are both selected by the *slider* and that are contained within *related_visualization* can be selected. Note that all the changes made to the specification in Figure B-11 are shown in gray in Figure B-12.

In Figure B-11 the *intersect* operator constrains the scope of the technique so that only graphical objects in the *map* region container may be colored. In Figure B-12 we constrain the scope to particular distributor data concepts instead. In this case, any graphical object may be colored as long as those objects represent data concepts that are present in *related_visualization*. In this was we can combine the object membership or query results across multiple visualization interfaces.

Translation functions may also be used to change the function arguments provided by users (through input devices) or by designers (as default values). For example, in Figure B-11 the *get-graphical-objects* translation functions is used so that the designer may enumerate the scope of the graphical object set based on a container object (i.e. the *map-visualization region*). Alternatively we may remove *get-graphical-objects* translation function and list out each individual graphical object of interest.

B-4.3 Changing How Control Arguments are Provided as well as the Default Designer Values

The third step in the instantiation augmentation process identifies the input arguments needed by the functions within a specification and connects them to a user agent or a designer agent. The specification shown in Figure B-11, for example, requires five input arguments: 1) the set of data values we want to perform the threshold operation on, 2) a threshold value for the threshold function, 3) a set of graphical objects for the *intersect* set operator, 4) a set of feedback graphical values, and 5) the feedback value used. These five input arguments are shown in Figure B-13 as text with gray background.

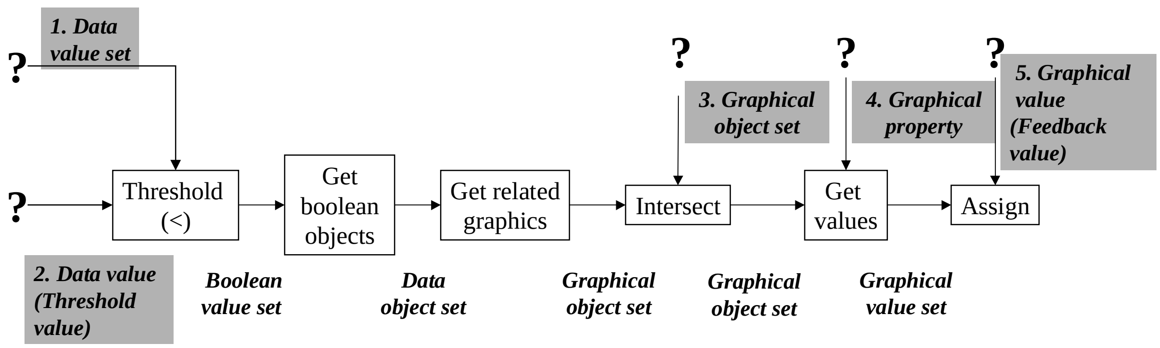


Figure B-13: This specification is similar to Figure B-11 except that here we have included all of the input arguments that are required by the various functions that have yet to be provided. Each missing argument value is indicated with a "?" symbol together with the argument type that is required. In this example there are five missing function arguments.

We can generate a variety of designs for this technique by either setting the arguments to different default values (i.e. designer defaults) or by letting users specify the arguments through input devices. One possibility is to let users provide part of the input arguments while leaving the rest as designer defaults. For example, in Figure III-12 the threshold value is provided by users through a slider input device and the set of data values for the *threshold* function are from a data attribute that is provided through a *menu* input device. All the other input arguments are provided through designer defaults. Another alternative is to give users more flexibility and let them pick the feedback graphical property (e.g. *size* or *shape* instead of *color*) and the feedback value (e.g. *blue*, or *green* instead of *red*). Note however that loading an interface with too many input controls may significantly increase the motoric and cognitive complexity placed upon users when manipulating the interface. We could also change the specification by experimenting with different default values, for example instead of using *red* as the default highlight color, we can set the highlight color to *blue* instead.

In addition to users and designers, function inputs can also be provided by other visualization techniques through composition operators as was shown in chapter II-2. However, such changes alter the functionality of a visualization technique in significant ways, and as such their use is not encouraged during instantiation specification.

B-4.4 Changing the Type of Input Devices Used within the Design

In this step we pick the input devices for each user-provided input argument (as specified in the previous step). Suppose in the previous step we attached all the input arguments shown in Figure B-13 to input devices except input argument 3 (which feeds into the *intersect* operator). This is shown in Figure B-14 which replaces each "?" in Figure B-13 with an input device or a designer default value.

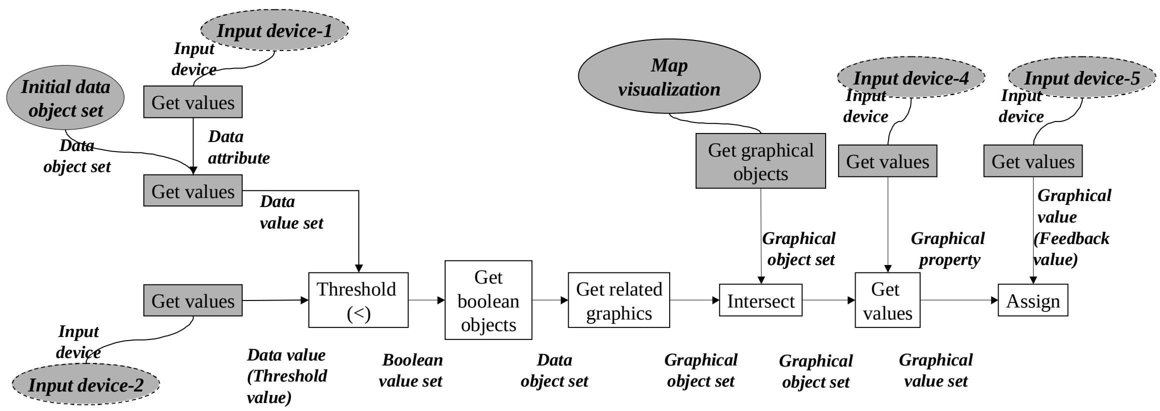


Figure B-14: Slider visualization technique with input devices

Now let us consider the types of input devices we can use to provide each of these input arguments. One way to select the threshold data attribute (i.e. *input-device-1*) is to use a *mouse* or *bounding-box* to pick an annotation object (e.g. the *x-axis*) that represents a data to graphical mapping relationship in the visualization. For example in ??, the *x-axis* annotation object represents a mapping of the *longitude* data attribute to the *x-position* graphical property. By picking this annotation object we indicate to the system that we want to perform the *threshold* operation on the *longitude* attribute. A weakness of this approach is that only graphically encoded attributes may be selected. The *mouse* and *bounding box* cannot be used to specify non-encoded data attributes because none of them are visually represented.

Other alternative input devices shown in Table III-5 include the *text-box, different menu types (option menu, scroll list, radio boxes), dial* and *slider*. These input devices are general purpose and can be used to select values, attributes, objects or containers. *Menu* input devices are especially appropriate for picking data and graphical attributes because the list of attributes is usually relatively small (< 30 attributes) and the attributes are discrete. *Dials* and *sliders,* on the other hand, are more appropriate for choosing continuous values even though they are also capable of expressing non-continuous values (e.g. alpha sliders [*Ahlberg, 1994*]). *Text boxes* are very flexible because users can type in any input argument. However, they do not indicate which arguments are valid and which are not. More information on the expressiveness and effectiveness of input devices can be found in Card et. al.'s work [*Card, 1990*].

B-4.5 Changing How Input Arguments are Provided to Input Devices

As was discussed in chapter III-1.4, input devices have attributes as well just like data and graphical objects. Some input device attributes must be initialized before they can be used. For example the *slider* input device must first be initialized with the *min* and *max* values for the slider range. To initialize an input device attribute, we construct a visualization technique specification with the same object-definition/transformation structure as all other visualization techniques we have been discussing thus far. Therefore, we may vary an input device initialization specification by using any of the previous four steps.

In summary, we have presented five steps for systematically exploring the instantiation design space. Changes in the instantiation design space allow us to expand or change the design of existing techniques while still maintaining a common functional metaphor.

B-5 Exploring the Space of Visualization Techniques

In this section we analyze three interactive visualization techniques using the five steps in the structural augmentation process (for details refer to appendix B-4). For each technique we present its instantiation specification and describe some interesting alternative designs that can be derived from varying that specification.

B-5.1 Aggregation

The aggregation technique deals with large data sets by summarizing multiple data concepts into an aggregate concept. Aggregation may be achieved in several different ways [*Goldstein, 1994*]. In this example, we examine the aggregation technique shown in Figure B-15. In Figure B-15 users may select a set of objects using a *bounding-box*. The selected graphical objects are converted to the data concepts they represent and these data concepts are aggregated (*group objects)*. In addition we also summarize a user selected attribute of the objects using the *mean* data transform function. Finally we map the new aggregate object into the visualization where the *bounding-box* was invoked (*add-object*).

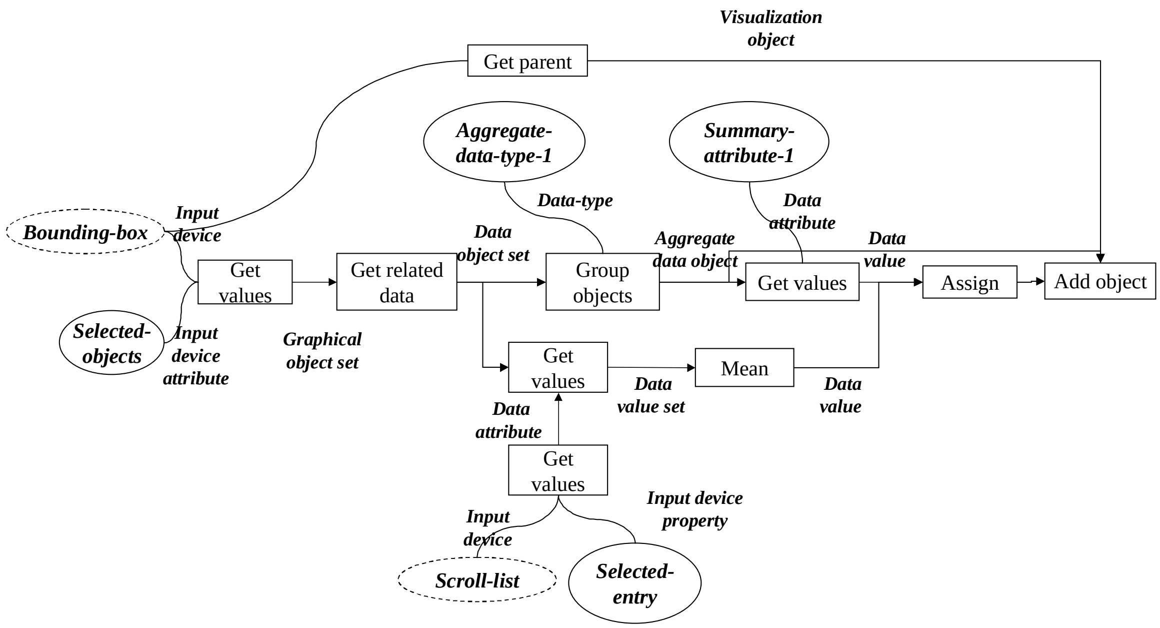


Figure B-15: One possible instantiation specification of an aggregation technique. This technique allows users to select a set of graphical objects using a *bounding-box*. It then summarizes the underlying data concepts by using a *group-objects* data transform function, thereby creating a new *aggregate-data-type-1* data-type. In addition, one of the data attributes for the selected objects is also chosen for *mean* summarization, and a new *summary-attribute-1* is created to store the mean values. This data attribute can then be mapped to a graphical property.

An interesting design variation is to use different devices to pick the aggregation objects and the visualization into which the group objects are added. This will allow us to pick objects in one visualization and have the group object appear in another visualization. Another useful variation of the aggregation technique is to remove the graphical objects that form the aggregate. Since we are adding a new group object that summarizes information on a set of individual data concepts, it may no longer be necessary to show those data concepts. Thus we add the *remove-object* mapping function that is used in tandem with the *add-object* function to reduce clutter and increase readability. Other design alternatives include adding in new input-devices for users to pick the summarization function (instead of always using *mean*), as well as the summary data-type (*aggregate-data-type-1*) and summary attribute names (*summary-attribute-1*).

B-5.2 Data Drag & Drop

Drag-and-drop is a popular input-device metaphor for moving objects from one frame or window to another. This method allows users to select an object or set of objects in an origin frame by *mouse-clicking* on them. The selected object(s) may subsequently be moved to a destination frame by *dragging* them over with the mouse. *Dragging* involves moving the mouse while having one or more of its buttons suppressed. Upon reaching the destination frame, the mouse button is released. In this work, we encapsulate the *drag-and-drop* technique as a virtual device method, just as a *bounding-box*. The *bounding-box* virtual device manages drawing of the *bounding-box*, testing and capturing all the objects encapsulated by the *bounding-box*, and erasing the box upon a mouse release event. Similarly, the *drag-and-drop* virtual device manages storing the origin frame where the initial *mouse-click* occurred (we call this object the *click object*), the transition scenes when moving objects between frames as well as keeps track of the object where the mouse release event occurred (we call this object the *release object*).

In Figure B-16 we show the instantiation specification for a drag and drop technique that is used to transfer data concepts from one visualization to another. This *data drag & drop* technique is used extensively in the Visage visualization system [*Roth, 1996*]. Upon the *mouse release* event of the *drag-and-drop* virtual device, we query it for the *click object* and the *release object*. The *drag-and-drop* virtual device in our system returns either the graphical object that was clicked on or a container object (e.g. *region*, or *visualization*). If the mouse click occurred over a graphical object, then the graphical object is returned, if not, the device will return the smallest container upon which the click occurred. Note that in Figure B-16 we only add the *click object* into the *release visualization* if the click occurred over a graphical object and not a container object. We check for this by using the *switch* function. The *switch* function is a control operator (described in appendix A-5) that allows us to pick different streams of execution based on its inputs. Similar to the click event, the release event may occur over a graphical object or over a container object. Since we are only interested in the release visualization, we use the *get-parent* translation function to query the release object for its parent visualization container.

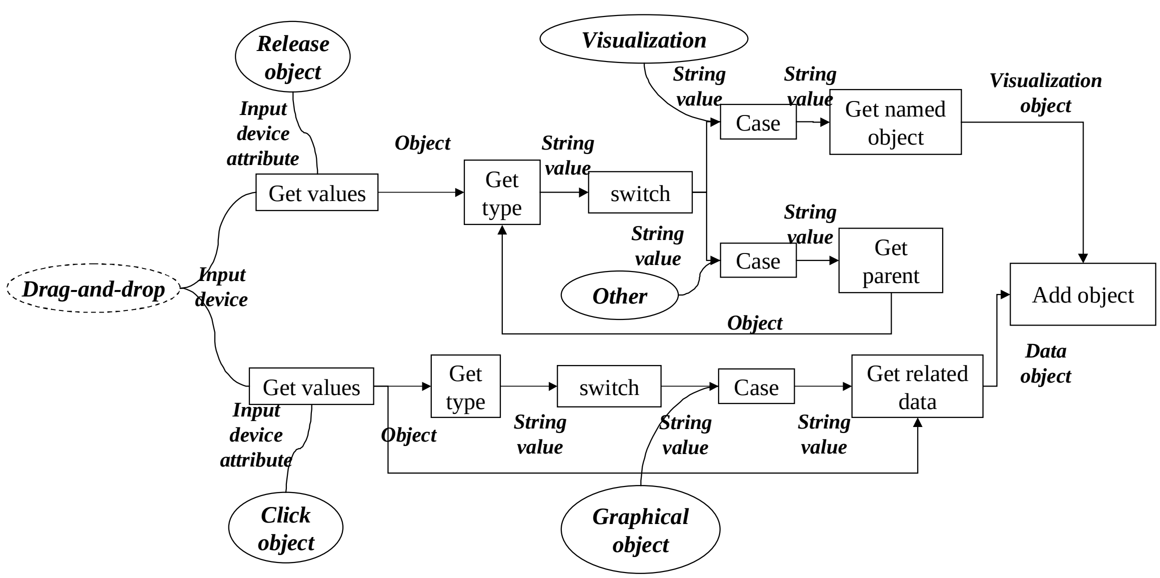


Figure B-16: Instantiation specification for Visage drag-and-drop technique. This technique allows users to pick a set of graphical objects in one visualization, and then add the underlying data concepts of the selected objects to a different visualization. The *switch* and *case* statements used above are to ensure that the user has indeed selected an origin set of graphical objects and a destination visualization container.

An interesting alternative to this instantiation design is to use different devices to pick the source graphical objects (*click object*) and the destination visualization (*release object*). For example we could use a *bounding-box* to select the initial set of graphical objects and a *mouse-click* to select the destination visualization. This design variation illustrates that the *drag-and-drop* virtual input-device is not an integral part of the functionality achieved by the technique. Another alternative is to use pre-specified source or destination arguments. For example we could make it so that elements can be selected in any visualization but they always get added into a fixed predefined visualization. Yet another alternative is to remove the selected objects from their origin visualization (using the *remove-object* mapping operator) in addition to adding them to the *release visualization*. We could also replace the *add-object* operator in Figure B-16 with the *remove-object* operator so that instead of adding data concepts to the release visualization we are removing data concepts from it.

B-5.3 Table Lens Semantic Zoom

One very useful operation in the Table Lens system allows users to interactively control the size of table cells so that the interesting ones can be expanded and viewed in greater detail while the size of the surrounding cells are contracted so that context from these surrounding cells can still be maintained. This technique is achieved through rendering transforms that are described in detail in Rao et al.'s paper [*Rao, 1994*]. Changing the size of table cells also causes the graphical object mappings within those cells to be remapped. In particular, the larger cells will have mappings to both the *text-graphical-class* and the *bar-graphical-class* while the smaller cells will have mappings only to the *bar-graphical-class*. One way to achieve this change in mapping is through the instantiation specification in Figure B-17.

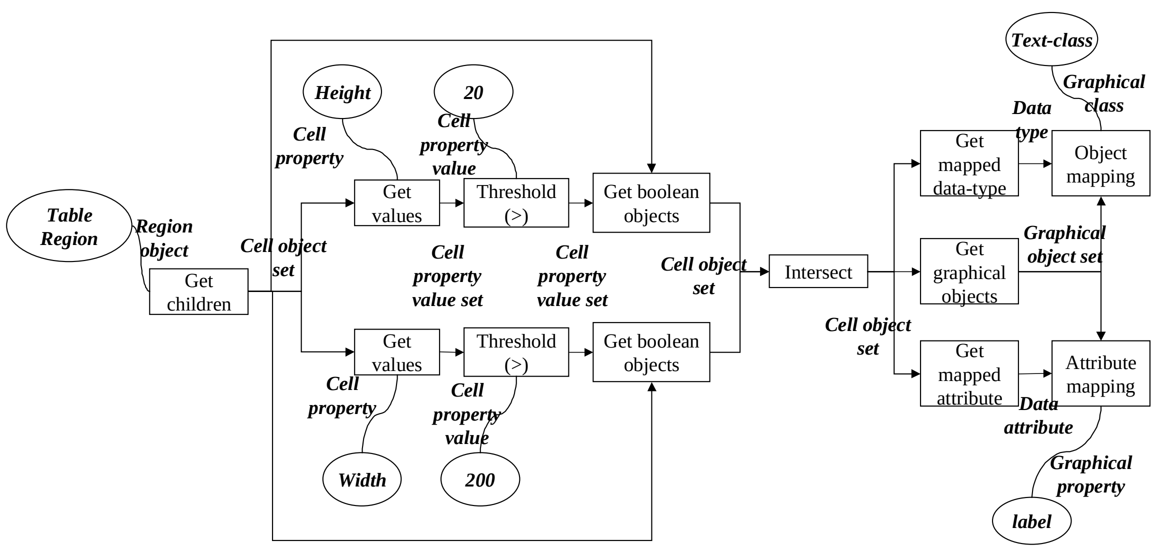


Figure B-17: Instantiation specification for the Table Lens semantic zoom operation [2]. This technique identifies the larger cells in the table (i.e. height > 20, width > 200) and adds a new *text-class* mapping for those larger cells.

---

[2] Note that in this specification we have introduced the *cell container object*, which we did not present in chapter III-1.1.2. This *cell container* however acts like any other container object and is placed below the *region* container in the graphical object hierarchy.

In this specification we query for all cells that have *height* greater than 20 pixels and *width* greater than 200 pixels (these are the larger cells). We then add mappings for the *text-class* and *label* properties to these larger cells. The other "small" cells are rendered according to the default mapping which in this case encodes all data concepts using the *bar-graphical-class*.

We can augment the Table Lens semantic zoom technique by adding in input-devices for controlling the *height* and *width* threshold values. This will allow users to interactively control what actually constitutes a large cell based on their perceptual abilities and the hardware screen settings. Another possibility is to add input-devices so that end-users may interactively choose the graphical property upon which to threshold on. A problem with this change is that it may de-couple the semantic zoom operation (Figure B-17) from the Table Lens size control operation because the semantic zoom will no longer be tied to the size of the cells.

Another design possibility is to break up the cells into several different groups (i.e. more than 2 as is the case in Figure B-17) and apply a different mapping to each group. For example instead of just having large and small cells, we could have large, medium-large, medium, medium-small and small groups. In this way the change in cell size will be more gradual and this may help users interpret the display changes when the *table lens* focus is changed. We could also threshold on several different graphical properties simultaneously, such as cell *size* and *color,* and have a large red colored cell group, a medium red colored cell group, a medium blue colored cell group, etc. In order to be useful however, this perceptual categorization must correspond to some meaningful data grouping.

B-5.4 Summary

These examples show some interesting design variations that can be achieved with current visualization techniques by making changes at the instantiation level. Alterations to the instantiation specification change the way with which the techniques are controlled, the amount of flexibility a user has in manipulating the techniques, as well as the amount and quality of feedback that is received. This section is provided as a contrast to chapter II-3 that explored the functional space of visualization techniques. In chapter II we explore the visualization techniques space by changing their functional semantics. Here we keep the semantics constant and explore the various structural forms that may be used to achieve the same functional capabilities and how these changes may improve the usability of the techniques.

B-6 Other Visualization Technique Issues

In chapters II and III we considered building visualization techniques either from primitives or through composition. However we only dealt with single techniques that are applied once to objects within a visualization. In practical situations, visualization techniques are often repeated many times over different object sets. In addition, these techniques do not usually exist in isolation but are instead integrated with other visualization techniques within a common workspace. In the next two sections we begin to study these two issues and show some of the problems that may arise.

B-6.1 Repeating Visualization Techniques

Repeating interactions is a very important issue in designing interactive systems. Interactive techniques are commonly not just executed once, but are applied repeatedly to the same or to different object sets. For example throughout a data analysis session, we may want to filter and highlight different object sets within a visualization. When this occurs we must decide what happens when a previously transformed object is being transformed again. I.e. what happens when we color-highlight objects that have already been color-highlighted. There are four possible repeat alternatives: *forgetful repeat*, *additive repeat*, *incremental repeat*, and *toggle*.

A *forgetful repeat* interaction would return all previously affected values to their original state and only show the results of the new operation. For example suppose the slider technique in Figure III-11, Figure II-12, or Figure II-13 is a forgetful interaction. In this case, each time we change the slider threshold value a new set of objects will get highlighted red. All previously highlighted objects that no longer pass the new threshold will get reset back to their original color.

In contrast, an *additive repeat* interaction would add the new focus objects to the affected object set. Thus the objects highlighted red not only include the objects currently within the threshold indicated by the slider, but also those objects that have been previously selected by the slider. Newly defined objects that are already under the influence of the interaction are not changed.

The *incremental repeat* option is similar to *additive repeats* except that the interaction is applied to all input objects irrespective of whether they have already been altered. The effect of incremental repeats is the same as additive repeats for the dynamic query slider technique (e.g. Figure III-11) because the object set color is always assigned to a fixed constant value. However, incremental and additive effects are different for the technique in that colors the selected objects as well as increases their size (e.g. Figure III-13). If we used additive repeats, the selected objects are only enlarged once. However, if we used incremental repeats, the size of objects will get larger and larger as they get selected more and more times.

Finally *toggle repeat* adds selected input objects that are not already in the applied set into the applied set and removes input objects that are already in the applied set from it. For example suppose we selected a range from 100k to 150k on the dynamic query slider interface and then we selected a range from 120k to 170k. At the end of these operations, the objects that are highlighted red are those objects that are between 100k and 120k as well as 150k and 170k. The objects between 120k to 150k get un-highlighted because they fall within both of the chosen ranges.

This issue of repeating operations has been dealt with by Wills [*Wills, 1996*], but he focused only on object selection techniques. In order to deal with a wider range of techniques we added the *incremental repeat* option that is missing from Wills' framework. Wills did not consider this option because applying multiple object selections with the *additive repeat* option or the *incremental repeat* option has the same effect. This is because the selection technique only provides users with binary feedback (either an object is colored to indicate that it has been selected or not colored to indicate that it is not selected). For techniques that provide non-binary feedback (e.g. size increase, position shifts, etc), the *additive repeat* and the *incremental repeat* options will have different effects as was discussed above.

B-6.2 Integrating Visualization Techniques within a Common Workspace

When we integrate several techniques within the same environment, we must ensure that they are consistent with each other. A consistent workspace such as the Mac or the Windows environment allows users to access the techniques within it with greater ease because once users understand the metaphor or "physics" of that workspace, they can easily pick up on how new techniques would work within that space. Trying to develop a consistent set of rules for visualization technique design, however, is often a difficult and protracted process. Some systems base their workspace upon physical metaphors from the real world. The advantage of this approach is that most people understand the basic physical laws of the real world very well and thus are able to apply that knowledge to the virtual workspace. The disadvantage is that if we follow the "real world" too closely, we become constrained by its limitations and thus may not use the flexibility afforded by the computer media to its fullest. That is why most visualization systems turn out to be hybrid systems consisting of both "real world" and "virtual world" rules. Examining different visualization technique and workspace metaphors and deciding on the right balance of "real world" and "virtual world" rules is a very interesting but large area of study. Such considerations are beyond the scope of this thesis and is left for future work.

Another very important issue to consider when integrating a set of techniques into an environment is whether they conflict with one another. As a first step towards dealing with visualization technique conflicts, we consider all possible inconsistencies that may occur between any pair of primitive visualization techniques by sensing for conflicts based on five important dimensions: the object definition set, the transformation function used, the graphical property or data attribute used with the transform function (if applicable), the graphical or data value(s) used with the transform function (if applicable), and the input devices attached to the technique (if applicable). Based on these dimensions we identified the following conflicts:

- *Resource Ambiguity*: This conflict arises for different techniques[3] that use the same feedback value and properties. For example, consider a system that allowed users to highlight objects red either by

[3] Two techniques are considered different if they have different instantiation specifications.

painting with a bounding box or querying with a slider. In such a system there is ambiguity as to how/why a red object got highlighted. Problems arise because when the primitives are different, users commonly also expect different feedback or results from them. By using the same feedback property and value, users can easily get confused as to which technique caused a particular change in the display.

- *Resource Overload*: This conflict arises when two different techniques change the same data attribute or graphical property in different or opposing ways. For example, consider a system that used sliders to highlight objects red and bounding boxes to highlight objects blue as in the painting systems. When an object falls within the applied set of both techniques, there is uncertainty as to which color should be used.
- *Resource Adjacency*: This conflict arises when two spatially adjacent visual components are changed in the same way. For example, consider two spatially adjacent components within a mark graphical object: 1) the mark outline and 2) the mark body. When both components are colored in the same way, we lose our ability to perceive the boundary of separation between them. This raises complications when the outline width is also used to represent a data attribute because we lose this data encoding whenever the outline and body color of a mark object coincide.
- *Resource Conjoins*: This conflict arises when two resources are conjoint parameters. Conjoint parameters combine together to produce emergent properties. Examples conjoint properties include the width and height of a rectangle that produces the “area” emergent property. When one conjoint parameter is changed without comparative changes in the other associated conjoint parameter(s), the emergent property shown may no longer be correct and this may cause users to misinterpret the data.
- *Resource Inconsistency*: Resource inconsistency refers to all syntactic errors in a visualization technique specification. Some example syntactic errors include using incorrect types of input arguments, or trying to use an invalidated resource. For example, suppose a *remove-objects* visualization function is used to delete certain graphical objects from a visualization. A resource inconsistency error arises if a later visualization function tries to change those deleted objects.

Dealing with conflicts in a set of visualization techniques is a difficult process because it depends on the visualization, the techniques used, and the task. If the task does not require persistence in the visualization technique effects, then conflicts with that technique may not be as important. Otherwise, we must find some way to resolve the collisions and this may get very complex when there are many techniques within the environment. We therefore leave the problem of conflict resolution for future work.