ADAPTIVE SEMANTIC SNAPPING — A TECHNIQUE FOR SEMANTIC FEEDBACK AT THE LEXICAL LEVEL

Scott E. Hudson

Department of Computer Science
University of Arizona
Tucson, AZ 85721
hudson@cs.arizona.edu

ABSTRACT
This paper describes the implementation of *semantic snapping* — an interaction technique that provides semantic feedback at the lexical level while dragging a graphical object on the screen. Like conventional snapping, or *gravity fields*, semantic snapping includes a geometric component where objects in close proximity are drawn together or "snap" into position. However, semantic snapping goes further by allowing non-geometric (semantic) properties of objects to place additional constraints on snapping. Semantic snapping also provides more complex lexical feedback which reflects potential semantic consequences of a snap. This paper motivates the use of semantic snapping and describes how this technique has been implemented in a window-based toolkit. This implementation works in an adaptive manner to provide the best interactive response in situations where semantic tests are very time consuming and strain the limits of acceptable performance.

INTRODUCTION AND MOTIVATION
The interaction technique of snapping or *gravity fields* was introduced as part of some of the earliest work in interactive computer graphics. This technique is applied when graphical objects are dragged or stretched on the screen. Screen positions that correspond to correct or preferred ending points for the drag are given *gravity* — that is, when the cursor is near one of these points (or lines) it is attracted to, or snapped to, that point. For example, in a schematic diagram editor, component symbols and wires are typically attached to each other rather than being placed in arbitrary unconnected positions. Snapping allows these legal connected positions to be specified in preference to illegal unconnected positions. This allows the user to specify connections more easily and with less precise movements. As a result, for tasks that can employ snapping, it is sometimes possible to overcome the fundamental limitations of human performance described by Fitts' law [Fitt54].

This work was supported in part by the National Science Foundation under grant IRI-8702784.

Unfortunately, snapping has not been supported by the toolkits now popular for user interface development (see for example [McCo88, Lint89]) and the use of snapping in modern systems in general has not been widespread. A notable exception to this is the Gargoyle drawing editor [Bier86a]. In this system, reference objects that are not part of the final drawing may be used as guides for alignment and geometric construction. These reference objects, as well as their intersection points, are given gravity to make alignment and construction easier. Because these reference objects represent a locus of semantically interesting points, the techniques employed by the Gargoyle drawing editor (and in related work [Bier86b, Bier88]) can be seen as a form of semantic snapping. However, these techniques rely solely on geometric constraints to control snapping and provide feedback. The techniques presented here are specifically designed to compliment this approach by supporting additional control and feedback based on semantic properties that are not adequately expressed in a geometric context.

A good example of this occurs in visual programming environments such as the one shown in Figure 1. In this environment computational elements are represented by icons and dataflow between these computational elements is represented by lines. An editor for constructing these visual programs should perform type and other semantic checking. For example, the system shown in Figure 1 must ensure that connected ports send and receive data values of compatible types and that cycles are not introduced. In general, we support the view that:

It is better to prevent errors than to simply report them after they occur.

Consequently, one goal of a program editor like the one shown in Figure 1 might be to not only detect cycles and type conflicts, but prevent them. This can be done using semantic snapping. As dataflow lines are stretched between

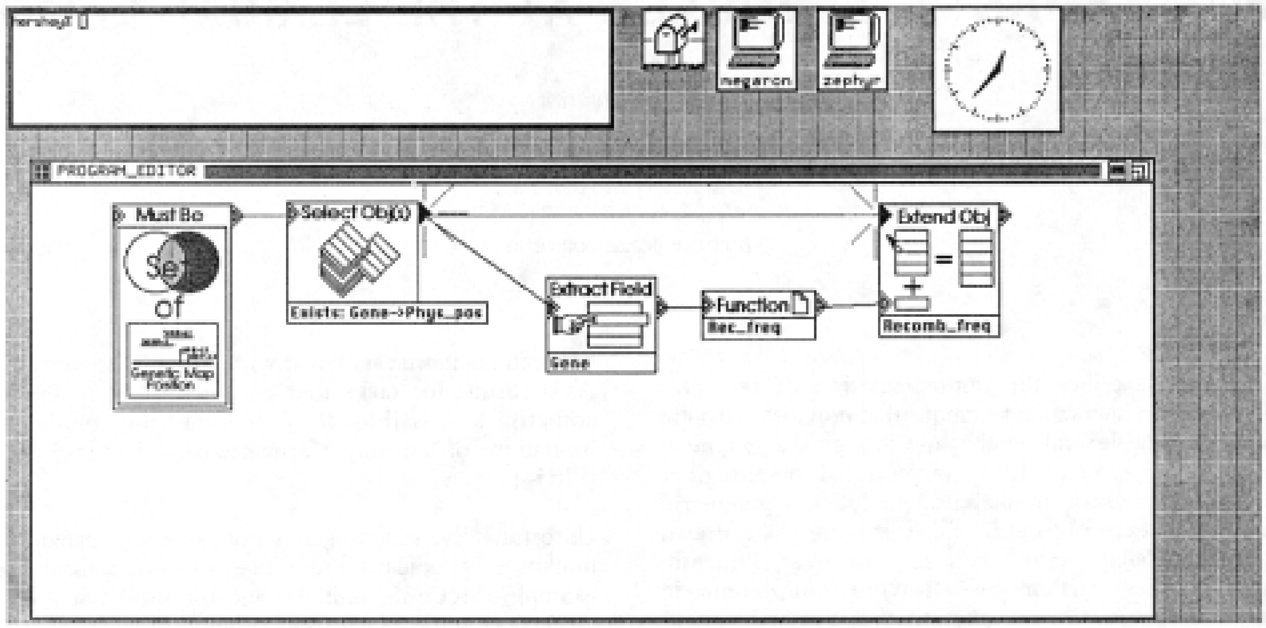

Figure 1. A Visual Programming Application

ports, they snap only to legal connecting ports and never to ports which would represent an error. Figure 1 shows the feedback for such a legal snap between the "Select" and "Extend" nodes.

As shown in Figure 2, illegal ports can supply *anti-gravity* which provides negative feedback for illegal connections. In this example, the user is attempting to create a cycle by connecting to the port at the left hand corner. The connection is of course disallowed, but the system also provides appropriate feedback indicating why the connection can't be made. In general, any desirable or undesirable property of a connection between objects can be tested and used as the basis for snapping control and feedback. This allows important but otherwise invisible properties of the objects being manipulated to be made apparent. It also allows the consequences of user actions to be indicated before the user commits to these actions. Both these properties serve to improve the directness of the interface [Hutc86].

In the next section a simple set of primitive snapping concepts will be considered along with techniques to allow the user more control over snapping. In Section 3, an implementation of semantic snapping in a window-based toolkit will be described. This implementation employs novel techniques that directly measure the progression of real time in order to adapt the system to the timing required for good interactive response. This is particularly important in situations where semantic tests are complex and strain the limits of acceptable response time. Finally, Section 4 will present conclusions.

SNAPPING CONCEPTS

In a wide range of interfaces such as graph, diagram, or schematic editors, the result of dragging is often to make a connection or relationship between two underlying objects. For example, in some systems [Myer86, Myer89, Huds89], geometric positions are used to infer constraints. Such constraints and relationships normally have implications in the underlying application domain. A primary purpose of semantic snapping is to express these important semantic implications to the user as early as possible.

Semantic snapping is built on a geometric basis — an object or set of objects that represent legal, desirable, or semantically interesting locations. These locations are often fixed, but sometimes must be computed "on the fly" based on factors such as the state of the interaction, values found in the underlying application data, and the object currently being dragged. These locations are often points, however, snapping to line segments and circles can also be useful.

Each object that supports snapping may have one or more snapping locations, or *sites*, associated with it. All snapping is done on the basis of snap sites rather than whole objects. Each site may be active or inactive at any

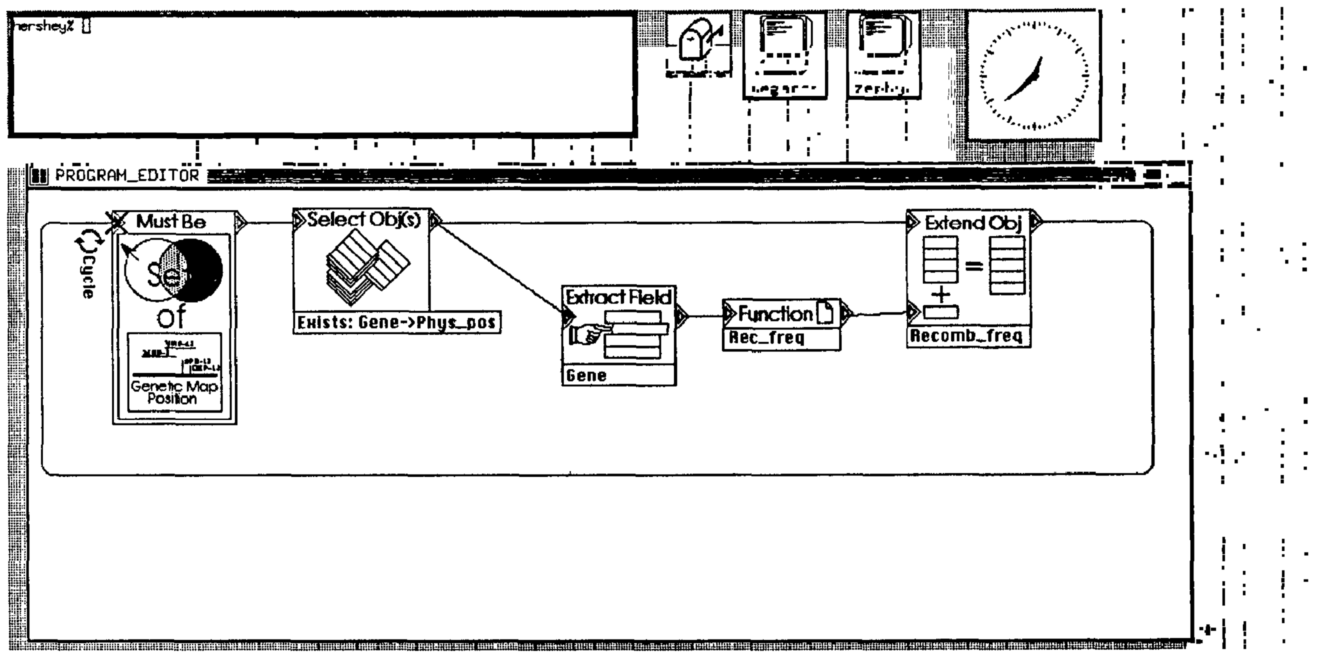

Figure 2. Anti-gravity Feedback for an Illegal Connection

given time. Activity or inactivity of a site is determined by a predicate function supplied by the site. This function has access to the internal state of the object and may test arbitrary semantic properties to determine its result. For example, the sites used in the visual programming system shown in Figures 1 and 2 are disabled when their maximum fan-in or fan-out has been exceeded.

Once a set of sites has been declared, it is the job of the snapping system to continuously search for a valid snap site and to provide appropriate feedback when one is found. This is done by finding the site closest to the cursor which: a) is active, b) is within a critical *snapping* distance, and c) passes semantic tests provided by the snap sites involved. In particular, each snap site provides a predicate function that can dynamically test whether a potential snap is semantically valid. One of the major focuses of the work presented here is a set of implementation techniques that attempts to provide the best possible response in situations where these semantic tests are time consuming.

It is also the job of the snapping system is to provide feedback about snaps. Whenever a new valid snap site is found, both the object being dragged and the object being snapped to are informed by calling a feedback function. This function is implemented by the object so that it can provide feedback that reflects both the state of the object and semantic consequences of the snap. Similarly, whenever an existing snap is broken — either by moving too far away or by explicit user action (see below) — both objects are informed so that feedback can be removed. In addition, to normal snap feedback, each site has the option of providing negative "anti-gravity" feedback. As illustrated in Figure 2, this feedback serves the purpose of an error message, indicating that a snap cannot be performed and why. Anti-gravity feedback is applied only when no valid snap can be found. Like normal feedback it is applied to the closest site which: a) is active, b) is within snapping distance, c) fails semantic tests, and d) has requested the application of anti-gravity.

In addition to the application of feedback, the snap system is responsible for resolving ambiguous snaps and for providing a user controlled reject capability. Because the active region of a snap site extends over an area rather than a single point (or line or circle), sites that have overlapping gravity fields may be difficult to select correctly. In addition, it may be possible for one cursor location to be equally close to more than one site, resulting in an ambiguous situation. This is particularly difficult to deal with if two or more sites occur at exactly the same pixel location (e.g. when both a line segment and its endpoints are snap sites). In situations where snaps are ambiguous, a priority scheme is used to select one site from among ties. However, this system is too limited to serve in all cases. To overcome this problem it is essential that a snap system also allow user controlled rejection of snaps. In the system described here, users may explicitly reject a snap by pushing another locator button or a designated keyboard key. Once a snap is rejected, the corresponding site is temporarily inactivated so that it does not interfere with selection of nearby sites. An inactivated site may be reactivated either by starting a new drag, or by moving far away from the snap site then returning.

IMPLEMENTATION

The snapping concepts introduced in the last section have been implemented in Artkit— the Arizona Retargetable Toolkit. This system is an object-oriented toolkit written in C and C++ which is designed to be hosted by a number of different underlying window systems. The toolkit currently runs under both the X and the SunView window systems and can be easily retargeted to additional window systems.

The primary challenge of implementing semantic snapping is making it run quickly enough so that good interactive response can be provided even when time consuming semantic tests need to be performed. The Artkit implementation uses four techniques to produce a fast implementation — a good geometric search algorithm, declaration and optimization of common special cases, "hiding" of work based on non-linear human perception of response time, and most importantly, adaptation based on monitoring of actual real-time performance.

Because the geometric proximity tests required for snapping (i.e., finding all the sites within the critical snapping distance) are typically faster than semantic tests, geometric searching is done first. Semantic tests are performed only on the small set of active snap sites found to be within snapping distance.

The geometric search done in the Artkit implementation is based on a *bucketing* technique. In this scheme, each window is broken up into square buckets of a fixed size and each snap site is assigned to the bucket or buckets it occurs in. Most sites have a static position and stay in one bucket, but some sites may have to have their bucket recomputed when dragging starts. The system currently uses a fixed bucket size of 32x32 pixels. This size represents a good compromise in space versus speed. If 1024x1024 is assumed to be the largest window size, there will be at most 1024 buckets in a window. Thus, space for the bucket structure itself is not unreasonable (a maximum of 4k bytes per window). Furthermore, using the high-density rejection technique described below, it is possible to guarantee that no more than 114 snap sites occur in one bucket. This corresponds to a snap site at every third pixel and is a reasonable approximation to the upper limit of what the user can handle interactively. Since the cursor may be near the corner of a bucket and hence the snapping area may overlap 4 buckets, this leads to an absolute maximum of 456 snap sites that must be considered at any time.

Large numbers of snap sites that occur very close together (within a few pixels) can place a heavy burden on the snap test algorithm, requiring many sites to be tested for each cursor movement and potentially overloading some buckets. In addition, when snap sites occur close together, it is difficult for the user to select the desired site even when the exact pixel position of the site can be determined visually. In cases of ambiguity, the system makes what amounts to a reasonable guess (the closest site with ties broken by priority) in its selection of a snap site. It then allows the user to explicitly reject this point if the guess was wrong. In order to limit the number of sites that must be considered on each cursor movement, the snap search algorithm modifies the way this guess is made in instances where sites are very close together (within 3 pixels in x and y). In particular, the system temporarily eliminates from consideration snap sites that are too close to other (higher priority) snap sites. If a site is explicitly rejected, any previously eliminated site will be placed back into consideration so that it can be subsequently picked by the user. This approach keeps the system from bogging down when overloaded, resulting in better performance when the system is saturated, with little or no effect on the end user.

Once the set of sites within snapping distance has been determined, semantic tests are performed (closest snap site first) to make a final selection. Semantic tests are performed by calling a predicate function provided by each snap site. In some rare cases, the result of the predicate depends on a continuously varying value. However, usually, the result of the predicate depends only on values that do not change over the course of a single dragging interaction and the predicate need not be evaluated more than once for each drag. In some other situations, the result of a predicate does not depend on the object being dragged and changes only rarely. Here, the underlying application can inform the snap system whenever changes occur. The system handles each of these cases — *continuous*, *single*, and *demand* evaluation — separately. For demand evaluation, the application informs the snapping system whenever the result of the semantic test might change. The system then evaluates the test function and remembers the result. By default, single evaluation is used so that the system evaluates each test function at most once and remembers the result for the duration of the drag. Only in the case of continuous evaluation does the system evaluate the semantic test function each time the cursor moves.

Human perception of response time in dragging tasks has not been widely studied and to the author's knowledge, no study has ever been made of dragging with snapping. However, early studies of remote control manipulation of objects under time delay (see for example [Sher63]) along with informal experience with dragging tasks, indicates that dragging with long lags in response is difficult to control. Conversely, delays below a small threshold tend to remain unnoticed. The system uses these facts to attempt to ensure good interactive response.

The system directly monitors real-time performance and adapts in an effort to meet response time goals. In particular, the system uses soft limits for the time spent at the beginning of a drag (usually after a locator button down event) and for the time spent after each cursor movement event. The system performs semantic tests until either a snap site is determined or the time limit has expired and a new event is waiting to be processed (note: time limits are non-preemptive). If the time limit has expired, the best snap site found thus far is used. If no valid snap sites are found within the time limit, no snap is made, but the system resumes testing where it left off when processing the next movement event. This ensures that if the cursor stays in the area of a valid snap site, the system will eventually find it. While the system cannot always meet response time goals, this approach allows the system to degrade gracefully. In situations where it cannot meet performance goals, the system either selects a nearby valid snap site or delays snapping for several movement events. In either case, disruption of normal system functioning is minimized. Furthermore, this approach allows the system to automatically respond to varying system load as well as providing transparent portability to new processors with differing speed.

In addition to real-time response goals, the system also attempts to "hide" work by making use of the non-linear human perception of response time. When a snap site is found very quickly, additional time — up to a minimum target time designed to remain unnoticed by the user — is spent evaluating the semantic functions of nearby snap sites. This allows work to be hidden in unperceived delays in hopes of shortening later delays. Similarly, informal experience indicates more user tolerance for delay at the immediate beginning of a drag (i.e., after a locator button has been pressed but before selection feedback has been provided) than during the drag itself. Consequently, the time limit at this point in the interaction is made longer. This allows the system to get a head start by evaluating the semantic functions of snap sites near the initial cursor position.

The system allows the application to provide time limits. However, it also provides defaults for these values. The current, defaults were arrived at by trail and error and are only preliminary approximations — no claim is made as to their "correctness" from a human factors point of view. A more formal and complete study is needed to determine how these default values should really be set in practice.

The four techniques described above — a good geometric search algorithm, declaration and optimization of common special cases, "hiding" of work , and most importantly, adaptation based on monitoring of actual real-time performance — serve to create an implementation which is very robust and which degrades gracefully in the presence of expensive semantic tests. This allows applications designers to use semantic snapping with confidence, knowing the system will automatically adapt to varying loads and exceptional situations.

CONCLUSION

This paper has considered an implementation of semantic snapping. Semantic snapping offers a number of advantages by increasing the level of feedback provided to the user and by allowing errors to be prevented rather than simply detected. An adaptive implementation technique for semantic snapping has also been discussed. This technique uses soft real-time limits to monitor interactive response of the system and to automatically adapt the performance of the system. This allows the system to degrade gracefully and provide the best response under high load situations.

REFERENCES

[Bier86a] Bier, E., Stone, C., Snap-Dragging, *Computer Graphics*, v20, n4, August 1986, pp. 233-240.

[Bier86b] Bier, E., Skitters and Jacks: Interactive 3D Positioning Tools. *Proceedings of the ACM SIGGRAPH Workshop on Interactive 3D Graphics*, Chapel Hill, NC, October 1986, pp. 183-196.

[Bier88] Bier, E., Snap-Dragging: Interactive Geometric Design in Two and Three Dimensions, University of California, Berkeley, Technical Report UCB/CSD 88/416, April 1988.

[Born86] Borning, A., Defining Constraints Graphically, *Proceedings of CHI '86*, Boston, April 1986, pp. 137-143.

[Card88] Cardelli, L., Building User Interfaces by Direct Manipulation, *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, Banff, Alberta, Canada, October 1988, pp. 152-166.

[Fitt54] Fitts, P., The Information Capacity of the Human Motor System in Controlling the Amplitude of Movement, *Journal of Experimental Psychology*, v 47, June 1954, pp. 381-391.

[Gold83] Goldberg, A., Robson, D., *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.

[Huds89] Hudson, S., Graphical Specification of Flexible User Interface Displays, *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, November 1989, pp. 105-114.

[Hutc86] Hutchins, E., Hollan, J., Norman, D., Direct Manipulation Interfaces, in *User Centered Systems Design,* D. Norman and S. Draper (ed), Lawrence Erlboum Associates, Hillsdale, New Jersey, 1986, pp. 87-124.

[Lint89] Linton, M., Vlissides, J., Calder, R., Composing User Interfaces with InterViews, *IEEE Computer*, v22, n2, February 1989, pp. 8-22.

[McCo88] McCormack, J., Asente, P., An Overview of the X Toolkit, *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, Banff, Alberta, Canada, October 1988, pp. 46-55.

[Myer86] Myers, B., Buxton, W., Creating Highly-Interactive Graphical User Interfaces by Demonstration, *Computer Graphics*, v20, n4, August 1986, pp. 249-258.

[Meyr89] Myers, B., Vander Zanden, B., Dannenberg, R., Creating Graphical Interactive Application Objects by Demonstration, *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, November 1989, pp. 95-104.

[Sher63] Sheridan, T., Ferrell, W., Remote Manipulative Control With Transmission Delay, *IEEE Transactions on Human Factors in Electronics*, vHFE-4, n1, September 1963, pp. 25-29.