Image Snapping

Michael Gleicher

Advanced Technology Group
Apple Computer, Inc.

ABSTRACT

Cursor snapping is a standard method for providing precise pointing in direct manipulation graphical interfaces. In this paper, we introduce *image snapping*, a variant of cursor snapping that works in image-based programs such as paint systems. Image snapping moves the cursor location to nearby features in the image, such as edges. It is implemented by using gradient descent on blurred versions of feature maps made from the images. Interaction techniques using cursor snapping for image segmentation and curve tracing are presented

CR Descriptors: I.4.3 **[Image Processing]**: image processing software. I.3.6 **[Computer Graphics]**: interaction techniques.

1. INTRODUCTION

Cursor snapping is a standard method for providing precise pointing in direct manipulation graphical interfaces. Since its introduction in Sutherland's Sketchpad [30], variants of snapping have been employed by almost all object-oriented drawing, modeling and CAD applications. In this paper, we present *image snapping,* a technique that extends cursor snapping to image-based applications such as paint systems.

Image snapping retains the basic idea of traditional cursor snapping: the cursor follows the motion of the pointing device, but snaps to interesting locations that are nearby. With image snapping, features such as edges or locations of a specific color can serve as snapping targets, as shown in Figure 1.

For traditional snapping in an object-oriented drawing application we have a list of objects that provide features to snap to. To provide snapping in images, a system might apply image understanding techniques to build an analytical, object-oriented model. Such an approach is typically impractical because of the complexity of image understanding. Instead, we use image processing techniques to determine likely targets on a per-pixel basis. Image snapping searches the image made from these probabilities, called the feature map, for targets near the pointer.

1 Infinite Loop M/S 301-3J, Cupertino, CA 95014
`gleicher@apple.com`

This paper presents the methods for realizing image snapping. After an introduction to cursor snapping, we present techniques for searching an image for features. The task of identifying features is discussed with particular attention to edge detection. We then examine how cursor snapping applies to higher level tasks in image-based applications.

2. PRECISE POINTING IN DIRECT MANIPULATION

Almost all object-oriented drawing, modelling and CAD programs provide some technique to aid the user with precise positioning. Bier [3] surveys approaches to this problem. Techniques aim to give the user the required precision, yet maintain the dynamic and free feel of direct manipulation. The most successful methods enhance direct manipulation positioning by guiding the position of the cursor from which drawing operations are carried out. In such schemes, the position of the software cursor is decoupled from the the screen location specified by the hardware pointing device, which we call the pointer. The cursor follows the pointer, but snaps to nearby locations of interest. Most snapping interfaces, with the exception of Venolia's 3D system [33], move the cursor discontinuously: when the cursor gets within range, it jumps to its target.

A common cursor positioning aid is the fixed grid. While easy to implement and use, it permits only operations that are grid-aligned. Grids could be used in an image editing program; however, to obtain precision with respect to features in an initial image, the features would have to be aligned with the grid. This is especially impractical when the images are acquired from the real world.

Other snapping variants are scene sensitive, that is, the position of the cursor depends on the contents of the drawing or image. When the pointer comes within snapping range of an object, the object's "gravitational attraction" draws the cursor to its surface. Such methods were introduced in Sketchpad [30], where they were dubbed "gravity." Gravity was developed further in the snap-dragging work of Bier [3][4]. Many popular drawing applications provide object snapping. It can provide precision greater than the resolution of the input device since snap targets are computed analytically. Snapping also can facilitate reference to objects. Users can point near to objects, rather than being forced to point exactly on top of them.

Snapping systems often provide the user with feedback of the cursor location and snapping state. The user can see the target of a drawing operation before it occurs, and correct for bad snaps by moving the cursor or overriding snapping. Snapping with proper feedback tells the user what locations can be snapped to. Hudson [17] suggests many applications of snap feedback.

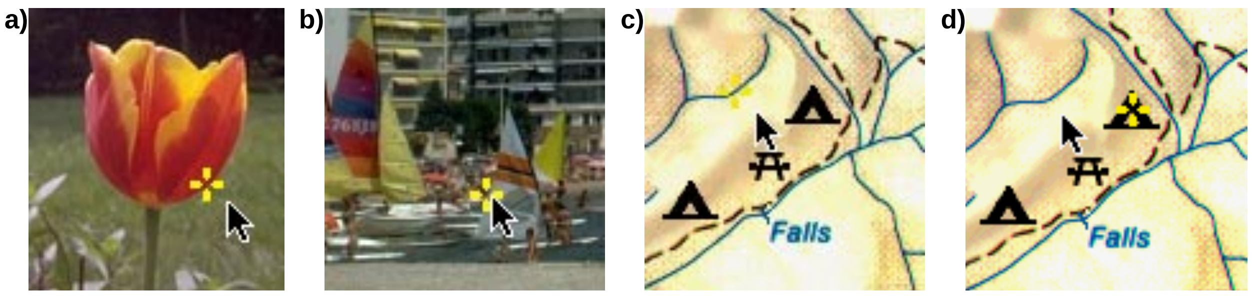

Figure 1: Examples of Image Snapping. Image snapping attracts the cursor to image features such as object edges (**a** and **b**), pixels of a specific color (blue pixels in **c**), or the centers of particular shapes found with template matching (campground symbols in **d**). The cursor state and pointer position are both always shown.

2.1 Implementing Snapping

The central piece of a snapping implementation is the function that computes the position of the cursor given the position of the pointer. In a traditional snapping implementation, this cursor function has arguments of the pointer position and the scene objects. The basic implementation of a cursor function iterates over the objects, finding the location on each that is closest to the cursor. The closest result is returned. Complications arise from the need to cull objects for better performance and computing the points that might be snap targets.

Computing snap points requires finding the point on each object that is closest to the pointer. Sutherland is "intentionally vague" [30,p.30] in describing how to do this for a line and circle, noting that each new object type requires a new method. Defining this method for more complex objects, such as curves, can be difficult. If a snapping implementation is to support snapping to object intersections, the cursor function must compute the intersection of any pair of objects, increasing the difficulty of adding new object types.

Gleicher [11] presents a generalized approach to implementing snapping using numerical constraint solving. The cursor is considered as a free floating point that minimizes its distance to the pointer location, as if it were connected by a damped spring. Each object has an implicit function that measures the distance of a point to the object, so that f(x,y)=0 defines the object's edge. When the cursor is close to an object, it is snapped to the object by treating the implicit function as a constraint on the cursor position. Intersections of objects are handled by solving multiple constraints simultaneously. The constraint snapping the cursor to an object is removed if the spring connecting the cursor to the pointer must pull too hard. We can think of generalized snapping as a physical process: the cursor is a ball that we position by pushing it around on a surface. Objects are grooves in the surface. If the ball comes close to a groove, it will fall in. If we push the ball gently, or in a direction that allows it to stay in the groove, it will roll within the groove. If we push harder, the ball will come out of the groove.

3. IMPLEMENTING IMAGE SNAPPING

The image snapping algorithm must compute the following: given the location of the pointing device, return the precise position of where the user is most likely to be pointing. We assume that positions correspond to positions in the image, that is, that the transformation from screen coordinates to image coordinates has already been applied.

The image snapping function searches a feature map for snap targets near the pointer. We postpone discussion of how to obtain the feature map until Section 4. For now, we imagine that some oracle will tell us if each pixel is part of a feature. This will not, in general, be a binary (thresholded) result. Varying values can indicate probabilities and varying amounts of feature in the image region represented by the pixel. We prefer to avoid thresholding the feature maps as it allows for anti-aliasing and also because picking a proper threshold to reject improbable pixels can be difficult to do statically (see [27] for a discussion).

We use the convention that in the image map, a maximum value indicates the absence of a feature and a minimum value indicates the most certain features. We can think of such an image map as a height field where the image features are canyons.

To search for a feature near the pointer location, we could start at the pointer location and spiral outward, stopping when either a feature pixel had been found, or when the spiral had gotten so large that anything it were to find would be out of snapping range. If the snapping radius is small, the number of pixels that need to be examined is manageable. If not, fast algorithms for computing Euclidean distance maps [9][20] can pre-compute the nearest non-zero pixel in the feature map for each pixel in the image. Such an approach to image snapping has several drawbacks:

1. It has no methods of rejecting noise;

2. It uses a preset stopping criteria, globally thresholding the feature map;

3. There is no way to trade off certainty and size for distance;

4. It stops at the first feature it finds. Another, possibly better, feature, might be equidistant, or just slightly farther away. Also, it will stop at the boundary of feature regions, rather than seeking the "best" location on the feature.

In cases where the feature map is binary, uncluttered, and assumed to be perfect, computing a map of nearest pixels using a Euclidean distance map algorithm provides a simple method for implementing image snapping. Typically, we prefer to use local stopping criteria rather than global thresholding, and perform some operation that allows the algorithm to take an entire neighborhood into account at once. The use of local extrema to define features has a long tradition, see Maxwell[23] and Cayley[7] for example, and is a commonly used in edge detection, such as [25] and [6].

We therefore, suggest a different process: follow the image map gradients to a nearby image feature point. Basically, we view the feature map as a height field, drop a ball at the pointer position, and watch it roll down hill. If the ball reaches a satisfactory feature, we have found a point to snap to. Since the downhill direction is given by the gradient (actually, the negative of the gradient), such a search is easy to implement. This method is known as steepest descent [10][28]. Issues that make steepest descent unattractive for standard numerical problems do not apply to our task of finding minima in the feature map: we are only interested in problems where we have good starting points, the discrete nature of the image provides a natural step size and accuracy bound, and the small step size and direction-based stopping criteria, discussed later, avoid troubling oscillations.

3.1 Computing Gradients and Blurring

Texts, such as Pratt[27] and Nalwa[26], contain more complete discussions of the methods for measuring gradients in images. We review the basics here. Although an image is a continuous function over a region of the plane, in practice we only have a sampled representation. Therefore, to compute derivatives, we must use discrete approximations. The simplest method is to merely look at the adjacent pixels:

$$\frac{\partial}{\partial x} f(x, y) = f(x+1, y) - f(x, y) \qquad \text{(EQ 1)}$$
$$\frac{\partial}{\partial y} f(x, y) = f(x, y+1) - f(x, y)$$

Other methods look at more pixels in order to better estimate the gradient. A common measure called the Sobel operator averages the finite differences in different directions, examining a 3x3 region around the point to be evaluated. In matrix notation, the Sobel operators are:

$$S_y = \frac{1}{4}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}, \quad S_x = \frac{1}{4}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \text{(EQ 2)}$$

where S_x is the operator for the gradient in the x direction, and S_y is the operator for the y direction. Our implementation always computes image gradients using the Sobel operators.

On a raw feature map, image gradients provide little information for snapping. A feature can only influence a point one pixel away, there is no provision for noise suppression, and no mechanism for trading off certainty and size for distance. Also, because we know nothing about the smoothness of the image function, there is no reason to believe that the steepest direction is most likely to lead us to the nearest, or darkest, edge. These problems are illustrated in the example of a 1D image shown in Figure 2.

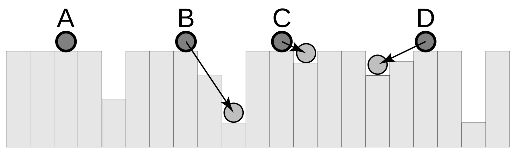

Figure 2: Cursor snapping in a 1 dimensional image shown as a height field. 4 starting points are shown. Each is given the opportunity to roll down hill. Point A does not roll anywhere since there is no gradient at its pixel. Point B correctly rolls downhill into the deep canyon. Points C and D roll down into the shallow canyons, although there are deeper ones nearby.

In short, image snapping at pixel scale [19][35], is ineffective. Abrupt changes, or high frequencies, in feature mask intensities make it impossible to sense features that are far away. Blurring the feature map to examine it at greater scale is essential for image snapping. Torre and Poggio[32] provide an alternate motivation: without blurring, evaluating gradients is an ill-posed problem.

Blurring is a standard image processing operation; most image processing texts such as Pratt[27] contain full discussions. Briefly, blurring is accomplished by taking a weighted average of the neighborhood around each pixel. Writing the weights as a matrix gives a kernel that can be convolved with the image to produce the image of averages. Linear filters are described by these kernel matrices. Larger kernels typically cause blurrier images. From the standpoint of image snapping, the larger the kernel, the larger the neighborhood over which an image feature has influence on the cursor.

Our implementation uses B-Spline basis low-pass filter kernels created by successively convolving a 2x2 box by itself[15][16]. Such filters approximate Gaussian kernels.

Blurring the feature map has many benefits. Foremost, it allows the effects of a larger neighborhood to be felt at a given pixel. The high frequencies in images are often noise, so blurring can help cancel these effects. While these high frequencies might represent fine details in an image, too many small details can cause clutter. Also, blurring can add useful detail within the features. For example, thick line features get stripes down their center, large points get dots at their centers, and intersections are emphasized. These effects are illustrated in Figure 3 and Figure 4.

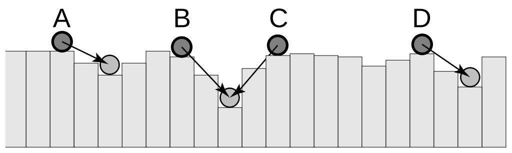

Figure 3: Cursor snapping in a blurred 1D image. A 3 pixel blurring operation has been applied to the image of Figure 2. In this case, each of the starting points snaps into a nearby deep canyon.

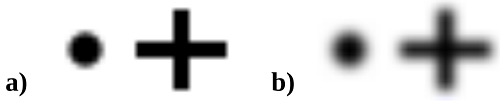

Figure 4: Blurring a simple image shows how blurring emphasizes the centers of points, lines, and intersections. The darkest points in image **b)** are centers of the circle and cross.

The benefits of blurring can be a problem. The larger area of attention can be problematic in cluttered scenes. The high frequencies in an image may not be noise, but rather, might be important features. The effect of features “migrating,” for example to the center of a wide line, may not be desired.

There is a trade-off in selecting how much blurring should be used for image snapping. A large amount of blurring allows the cursor to snap from larger distances and better reject noise at the expense of the ability to resolve fine detail.

As the blur factors grow larger, issues in how to represent the feature maps arise. While a large blur allows a feature to affect a position many pixels away, the amount of this effect can be very small. We use floating point numbers to represent feature maps; however, there is still an issue with noise rejection when the system is made sensitive enough to accommodate these very small values. We use a threshold to remove extremely small values.

3.2 Searching the Feature Map

Gradients in the feature map indicate which direction a search should proceed in order to find the best feature point. The magnitude provides little information on how to search, so we ignore it for searching. The search process moves a pixel at a time beginning with the mouse position. At each step, we evaluate the gradient of the blurred feature map to determine which direction to move the search to one of the pixel’s 8-connected neighbors. This quantization of the gradient direction means that we will not necessarily find the nearest point on the feature. We continue the process until it either moves too far from the initial starting point, meaning that there is no feature within the snapping radius, or when the search has found the best feature that it expects to find, e.g. a local extremum.

Local extrema are identified as places where the gradient vanishes [10]. Basically, when there is no downhill direction from a particular location, that location must be a minimum. Because of sampling, this criterion is not sufficient for image snapping. Local minima can actually occur between pixels. In such cases, the gradients of neighboring pixels will point at one another, as shown in Figure 5. We therefore add the additional stopping criterion that if the gradient changes direction too much between steps, a feature has been found. By placing limits on the range of directions allowed in a search, we can avoid looping and cycling. Our algorithm defines this search range from the initial step direction.

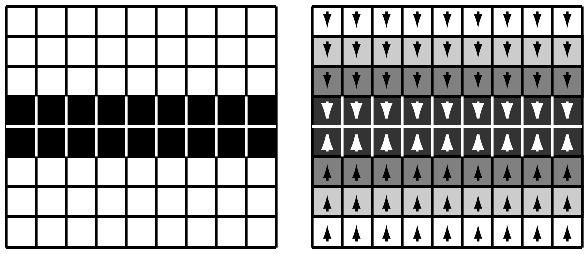

Figure 5: A 2-pixel wide feature. The left shows a 2 pixel wide feature. The right shows this feature blurred, with arrows denoting the gradient direction. Notice that the local minima is in the center of the feature, between the pixels.

Another issue in defining a stopping criterion is that within features the gradient is not zero. That is, within a feature, some points are more "featureful" than others. Without the direction change criterion of the last paragraph, the cursor might make a turn when it reached an edge, to search along the edge for a slightly darker spot. Dragging along a feature edge would cause the cursor to jump from one point to another. If some feature point, such as an intersection or corner, is sufficiently more featureful, its "gravitational attraction" will pull the cursor in as it approaches the feature.

Notice that our stopping criterion for a search is very different than performing a thresholding on the feature map. Rather than basing the decision on the value of the feature point alone, we use information about the neighborhood around the point, as well as the particular search.

A different problem occurs when we begin our search on a feature. Because we have no static criteria to classify a pixel as a feature or not, there is no simple way to distinguish between locations that are exactly on features and locations that are too far away from features. To perform such a classification, when we encounter a starting point which does not provide a gradient direction to search in, we check to see if neighboring pixels point towards this location before discarding it as a non-target.

3.3 Sub-pixel Positioning

When a local minima occurs between pixels, the direction change stopping criterion stops the search after the search has gone too far. In such a case, we know that the real stopping point lies between the last two pixels examined in the search. We can use the gradients at these two points to better estimate where in between the centers of these two pixels the real stopping point would be.

Given the gradients at the last two pixels of the search, the real stopping point would be the place along the line between the two pixels at which the projection of the gradient vector onto the search direction reaches a zero value. To estimate this location, we compute the projection of the two gradients onto the inter-pixel line, and approximate its change by assuming a linear transition. If the zero crossing is in-between the pixels, it is used as the sub-pixel location. This sub-pixel method only adds precision along the direction of search, so it is mainly useful for finding the centers of edges as shown in Figure 6 and Figure 7. Interpolating 4 neighboring gradients, as done for the precise edge detector of [22], might better localize points but has not been tried in our implementation.

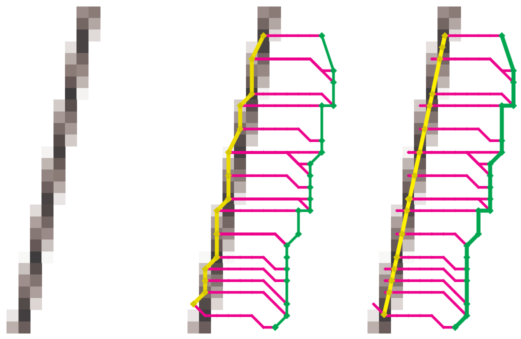

Figure 6: Snapping to a single pixel wide, anti-aliased line. The green line represents the path of the mouse, the magenta lines the search paths. In the center, the yellow line shows the pixel-aligned stopping positions. On the right, it shows the snap positions computed to sub-pixel accuracy. Notice that the points always lie on the line between the last two search points.

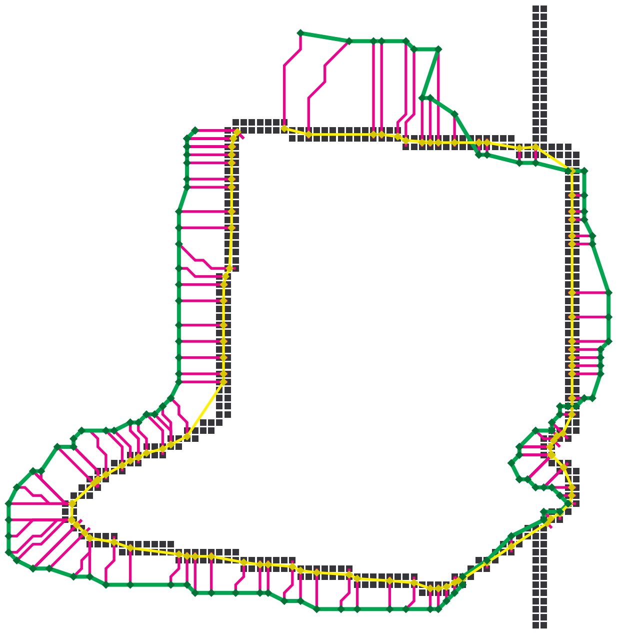

Figure 7: Snapping to the centers of the 2-pixel wide lines created by the simple edge detector applied to the segmentation image of Figure 8. Colors are explained in Figure 6.

The utility of sub-pixel positioning is not limited to cases where the feature detector is capable of localizing the features to this greater accuracy and the application can make use of the precision. In other cases, it adds important consistency when snapping to wide features. Consider image snapping without sub-pixel positioning on the example of Figure 5. If the search begins above the line, the stopping point will be on the bottom of the line, while if the search begins below the line, it will stop on the top. This means that if the user crosses the line while dragging the cursor along the feature, zig-zagging will occur. Sub-pixel positioning avoids this: starting from either side of the edge would lead to the same stopping point. Even if the sub-pixel position is rounded to the nearest pixel, consistency can be achieved.

3.4 Achieving Large Snap-Radii

Section 3.1 describes a trade-off in selecting the amount of blurring to apply to the feature map. To avoid losing the locality and fine detail of the features, we might prefer to use small amounts of blurring, especially when we have confidence in the map. The problem with small blurs in such cases is that the snapping radius is very small. In this section, we consider how we can achieve large snapping radii while still maintaining the features of small blurs.

We begin by blurring the feature map by a small amount (we typically use 3x3 or 5x5 kernels) and assume that when close to a feature, this blur provides good performance. This creates a small region around features that have gradient information pointing towards them. A simple thresholding criteria is possible: if the gradient has sufficient magnitude to determine a meaningful search direction, it will guide us to a good feature. If not, we need to use an alternate method to decide which direction to direct the search. As the search proceeds and nears a feature, it can use the small-blur gradients when they are available. We implement two methods for determining search directions for long snaps.

One way to enhance the snapping radius is to use different amounts of blurring. Such a technique is known as a multi-scale method, and is often used in computer vision, see [26, pp. 92-96] for a brief survey.

A simple multi-scale image snapping technique would create a very blurry version of the feature map in addition to the slightly blurred one. If no gradient is found for a given point in the small blur version, the larger blur version is checked. Unfortunately, this simple scheme will not work because the jumps in amount of blur will create discontinuities in the gradient field. At first, the search will proceed towards the best feature in the blurry image. As it gets close and switches to the less-blurry image, the feature will migrate differently causing a change in the search direction. Methods in computer vision typically address this by using a continuum of blurs [19][35]. For image snapping, we approximate these methods by using multiple blurs so that the changes among them are more gradual than jumping between a large and small blur. Computing and storing multiple blurred images is made practical by the use of an image pyramid [5][34] and interpolation.

Altering image snapping to use a multi-scale method is straightforward. We alter gradient evaluation so that if a zero value is produced at a point, evaluation looks at higher levels of the pyramid until a non-zero gradient is found. There are still discrete jumps between levels, and therefore discontinuities. These small jumps do not seem to be a problem in practice.

An alternate approach uses the distance map techniques of Section 3. Rather than building a map to the nearest feature, we create a map that stores the nearest point that has a gradient. Image snapping is trivially modified to begin by consulting the map to find the location for starting a search. Such a scheme still uses local properties to determine features and uses thresholding to determine where there is sufficient information to begin a search.

Our prototype implements both the multi-scale and distance mapping approaches to extending snap-radii. Computing the pyramid and distance map are approximately equivalent in computation time. We are able to mix the two approaches, for example to use 2 levels of the image pyramid coupled with the distance map. The advantages of the map method are that it can travel long distances quickly because it initially teleports rather than searches, it has no issues with small values, and does not require level interpolation on the pyramid. Pyramid methods may be preferable because the larger blurs reduce noise and clutter in the open spaces and allow for controlling the trade-off between distance and certainty. Pyramid methods may be easier to update incrementally, but we have not implemented this yet.

In uncluttered images, pyramids and distance maps both work well for extending the snap radius. In cluttered images, larger snap-radii may be problematic for any image snapping method because it becomes less clear which feature is the desired target.

3.5 Hysteresis

In pictures that are cluttered, i.e. where there are many features close to together as in Figure 1b, snapping may be difficult as the algorithm may not select the features that the user desires. Traditional snapping combats clutter with techniques that select features on criteria other than simple closeness. For example, cycling [3][12] allows the user to select among features within the snapping radius. Such a method relies on the ability to tell different features apart, which is easy in a traditional, object-oriented, snapping implementation. To use such a method in image snapping would require somehow grouping pixels into individual features, which is not possible without image understanding.

Hysteresis in snapping gives the cursor function a preference to stay snapped to the same object. This helps prevent the cursor from jumping around annoyingly. When multiple objects are found to be within the cursor radius, the last object snapped to is given preference, rather than simply selecting the closest. Hysteresis is extremely important when tracing features, which will be discussed in Section 5.2. Figure 10 provides an example of where hysteresis would avoid a problem for the user.

As described, cursor snapping does not provide any form of hysteresis. Each snap operation is independent of past and future operations. Like cycling, lack of object structures to snap to makes adding hysteresis difficult. Returning to the previously snapped point is not the same: we would like the cursor to follow smoothly along edges. To fake hysteresis, image snapping can use the heuristic that nearby feature points are likely to be on the same object.

We have created a technique that attempts to lock the cursor onto its current feature. When locking is enabled by depressing a modifier key, subsequent mouse motions pull the cursor along the locked feature. The technique is called cursor pulling because it pulls the cursor from its previous position, rather than starting searches from the pointer location.

Cursor pulling is inspired by generalized snapping that was reviewed in Section 2.1. Like generalized snapping, when the cursor is locked, we consider it “in a groove.” The positioning method switches from one that finds grooves to one that pulls the cursor along in the groove from its previous position. We displace the cursor a small amount towards the pointer and start the descent process from this point. The basic idea is that by starting at points near a point on a particular object, we are more likely to end up at another point on the same object.

The difficult decision in implementing cursor pulling is how far to displace the cursor before beginning the descent. The farther away from its initial starting point, the more likely it will end up farther away from this point, and the less likely it will return to the same object. On the other hand, if we start too close, the cursor is likely to return to the initial position. Rather than making a fixed decision, we displace the cursor back as little as possible. To implement this, we displace the cursor one pixel towards the pointer, and execute the descent process. If the cursor returns to its previous location, then we did not displace it enough, so we try again from slightly further away. We continue the process until either we find a position that descends to an acceptable location, or we give up the search because we have either gotten too far away from the original point, or have found a distance at which the cursor doesn’t snap to anything. In the event that we fail to find a new location, we return the cursor to its previous location.

We enhance locking by adding the heuristic that between two points on a feature should be only other feature pixels. We add a test that looks at the pixels along the line between the previous and current snap positions. If any of these pixels clearly do not qualify as feature pixels, we return the cursor to its previously snapped position. This technique requires a static threshold for pixels that are clearly not part of a feature. It also often causes the cursor to get stuck when there is a small break in a feature that is being traced.

As we pull the cursor along a feature, it typically moves more slowly than the pointer. The user often waits for it to "catch up" over long distances. Sometimes, the cursor will stick in places where moving towards the pointer does not lead to new positions. In such cases, the mouse can be moved to coax the cursor along.

4. FEATURE IDENTIFICATION

We now consider the problem of creating the feature map. Determining which pixels the user is more likely to want to snap to consists of two problems: determining what types of features are likely to be of interest, and determining what pixels are part of these features. The first part is often application specific. For example, we might want to snap to spots of a certain color or to bright or dark areas of the image. For some applications, we might want to manually create the feature map, attracting the cursor to features that an author can identify.

Object edges serve as a generally useful set of features to snap to. The ability to snap to edges mimics the typical snap targets of traditional snapping and is desirable for many applications. The difficulty in using edge detection stems from the fact that we are interested in *perceived* edges. Without the ability to identify objects, determining what is an object edge can be difficult. Edge detection is not always difficult. For example, if we have an image that is a line drawing or has been segmented into different objects (see Figures 7 and 8), accurately identifying edges is easy. Unfortunately, such segmentations are not always available, in fact, generating them is a motivating application (see Section 5.3).

Edge detection is a common problem in image processing and computer vision. Most textbooks in either field, for example Pratt's image processing text [27] or Nalwa's vision text [26], contain surveys. Use of better edge detectors will improve the performance of image snapping. For image snapping, good edge detectors are characterized by locality, noise rejection, and sensitivity. While these criteria are similar to those typically desired for vision applications, the fact that we can handle grayscale and thick lines leads to some differences from the criteria defined by Canny [6]. In particular, his goal to have a single pixel wide response for each edge is less important for image snapping.

We have experimented with a variety of edge detectors for use in image snapping. Simple edge detection operators, such as the magnitude of the gradient, often work well. More sophisticated operators, such as Marr-Hildreth[25] or Canny[6] offer better locality and noise rejection, however, they have the less desirable property of "thinning" the edges. Non-local techniques that attempt to link edge pixels together to raise certainty about the edges, such as Hough transforms[27], edge relaxation[2] or morphological methods[27] may further improve edge detector performance for image snapping.

As shown in Section 3.3, image snapping can locate features to sub-pixel precision. Using this precision requires features that are localized to sub-pixel accuracy. Examples of edge detectors that provide such accuracy exist in the literature, such as [22] and [31]. We have experimented with a scheme that enlarges an image with interpolation, detects edges, then reduces the image back to its original size. However, most of our sub-pixel experiments have been performed on synthetic images.

Another type of feature detector is a template matcher [27] that computes the likelihood that a pixel is the center of a particular shape. Using a template matcher output with image snapping produces a method similar to numerical search-based tracking [21] where the matches and gradients have been pre-computed. Because we are typically interested in extreme points, not smoothly sliding along edges, we allow much higher tolerances on angle changes. Effective use of a template matcher typically requires using thresholding. Our primary experiments have used template matching to help a user point at symbols on a map, for example to identify campsites in a park. Although the example in Figure 1d is synthetically generated, we have had good results on scanned maps.

5. INTERACTION IN APPLICATIONS

In this section, we look at some common tasks in image editing applications and examine how image snapping applies. Automating image editing tasks is difficult without image understanding. Cursor snapping can provide precision for more manual approaches.

Our implementation provides continuous feedback of the cursor position and a modifier key to disable snapping at any time. Feedback always shows the cursor location and whether or not it is snapped, in addition to the pointer location. If the cursor is incorrectly snapped, holding down a modifier key causes it to return to the pointer location.

5.1 Point Positioning

Often, a user needs to indicate the position of a point to an application where precision is important, for example, identifying correspondence points for image morphs, finding coordinates of particular features, measuring distances between points, or indicating locations on a map. Drawing operations that use discrete points, such as rubber-band drawing of lines, are also enhanced by cursor snapping.

5.2 Curve Tracing

In an object oriented drawing program, the cursor position is only used at discrete events. For example, when we draw a line segment or rectangle by rubber banding only the initial and final positions of the dragging operation are important. This means that if the cursor makes an incorrect snap during the drag, the user can correct the error without it ever being recorded.

In image-based applications, an important class of operations does use the cursor path. Brushing, penciling, and lassoing are examples of such techniques. Having snapping during these continuous dragging operations can be extremely valuable for tracing feature curves, for example to circle objects. Because the path of the cursor is recorded (and is important), errors made by the snapping algorithm can be problematic. An example is shown in Figure 10.

Even with feature locking techniques, as in Section 3.5, bad transient snaps are unavoidable. Curve tracing with snapping can be made workable through the addition of some simple interaction techniques. First, it is crucial to provide feedback of the snapping state to the user so that errors can be recognized immediately. Second, methods for backing up in the midst of a drawing operation must be provided. One interaction technique for this is to have the user back up over the curve to erase previous drawn portions. Another technique uses a modifier key that switches curve drawing into "backup" mode. While the key is depressed, moving the cursor permits specifying points along the curve. This method does

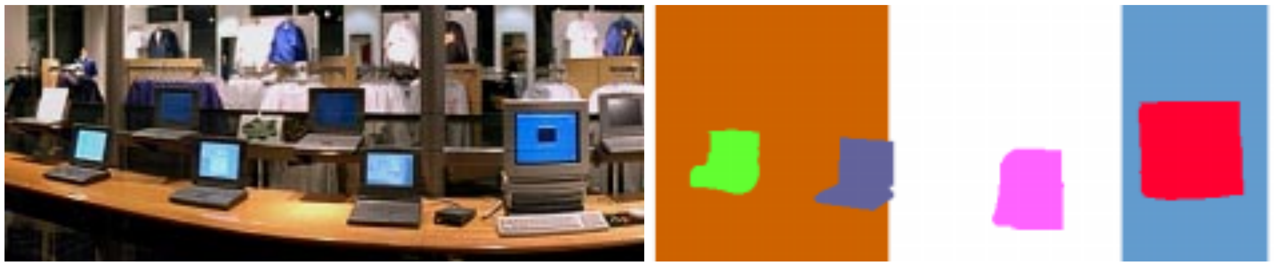

Figure 8: This piece of a panoramic image of a store (left) contains images that have particular behaviors when clicked on. To do this, an item buffer (right) stores a segmentation image identifying objects. This segmentation was not created using image snapping, but was used to create the edges in Figure 7. With the background objects (large rectangles) repainted white and the foreground objects (computers) repainted black, the segmentation can serve as a feature map for image snapping, attracting the cursor to computers. Images Copyright Apple Computer Inc, 1994.

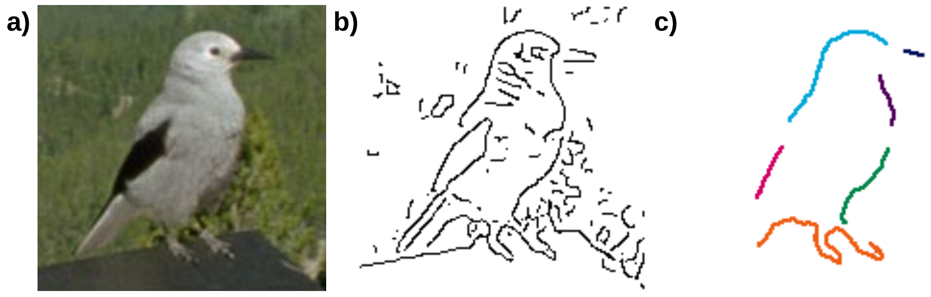

Figure 9: Lassoing a bird. The Canny edge detector (output shown (**b**)) performs well on some parts of the image, but not on others. Image snapping is used to trace the outline where meaningful snapping is easy (**c**). Other sections of the curve can be drawn manually and combined into a single lasso outline.

use a modifier key, however, it has the advantage that it allows the user to interactively find the point in the path that they want to roll back to.

5.2.1 Comparison with snakes

Curve tracing is an important enough task that techniques have been developed to help automate it. One approach is to fit deformable models to the image. A successful technique for this is the snake of Kass, Witkin and Terzopoulos [18]. The user places a snake curve near a feature and it automatically locks on to the feature. A numerical optimization moves the entire curve, both attracting it to features and keeping its shape smooth. Snake implementations allow the user to interactively pull on the snake as it progresses. Snakes have a wide variety of uses beyond assisting users in tracing curves.

A snake is an active object that seeks image features after it has been dropped on the image. Snakes provide a more highly automated process for tracing curves than image snapping, at the expense of user control. Where image snapping enhances the familiar drawing process, snakes provide a new type of active object that the user can place, watch settle, then hopefully coax into having the correct shape. Snakes address the issue of editing a curve after it has been placed, which simply drawing with image snapping does not.

Our system allows snakes and image snapping to be mixed: image snapping is used for drawing curves that can be turned into snakes for editing. This combination is useful as the snapping feedback can show the user what features the snake is likely to grab when activated. Our initial experiences show that this combination can be frustrating as the snake's internal forces rip it from a carefully placed initial curve. Improvements in the snake implementation may alleviate this problem. The issue does emphasize the difference in the use of active and passive objects as interface elements.

5.3 Image Segmentation

Another common task in image editing applications is identifying related regions of an image, known as segmenting it. Some of the many uses of segmentation that we have considered include identifying objects to cut them from their backgrounds, making mattes, and creating hit maps for interactive multimedia [8][24], as shown in Figure 8.

There is a large literature on automatically segmenting images, for example see [14] for a survey. Some of these techniques can be used to become semi-automatic interaction techniques, like Smith's tint fill [29] or the "Magic Wand" in Adobe Photoshop [1]. A more natural, but very manual, method for identifying a segment is lassoing, where a user traces around the edge of the region. Lassoing is a use of curve tracing, and is similarly enhanced by image snapping.

Lassoing objects can be a tedious task. Cursor snapping can facilitate it by freeing the user from having to precisely position the lasso. To further ease the task, our system allows the user to connect multiple curve segments to create a closed curve, eliminating the need to create the entire lasso in one motion. An example is shown in Figure 9.

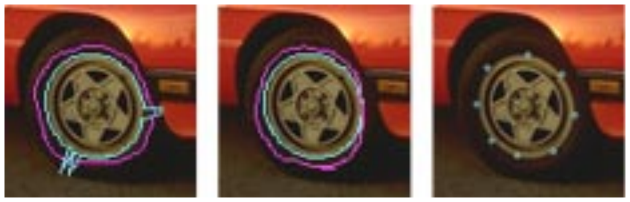

Figure 10: Circling the wheel. The counter-clockwise path of the pointer is shown in magenta, the path of the cursor in cyan. If the user is not careful, lack of hysteresis can cause bad snaps during circling, as seen on the left. The center shows a more careful tracing. The right shows an alternate interaction technique: the user specifies discrete points through which a curve will be fitted.

6. CONCLUSIONS

Creating interaction techniques that help automate image editing tasks is difficult because of the challenges in understanding the content of images. Even operations that are trivial for users, such as seeing edges, can be difficult to perform in a robust and accurate fashion. This can lead to frustrating interfaces as the system fails to see things that are obvious to a user. It also suggests a partnership where visual processing is performed by the user, rather than attempting to develop fully automated solutions. Avoiding tedium and providing precision are issues in such manual interfaces. Image snapping attempts to address these issues by facilitating precise pointing at features in the image.

The difficulty of identifying features means that image snapping is not perfect. Uncertainties in edge detection lead to noise in the feature maps and "blotching" along feature edges. We are exploring the use of better feature detectors and more sophisticated image processing techniques to reduce these problems, enhance the quality of interactions, and improve the range of images on which the techniques can be applied. The methods are unlikely to ever be perfect, therefore it is important to design interaction that accommodates this unreliability. Dealing with cluttered images is particularly challenging.

Image snapping brings a familiar feature of drawing programs to image editing. However, the interactions in image editing more often use the paths drawn by the cursor than discrete positions at snap points. As discussed in Section 5.2, cursor snapping does not apply as well to paths as to discrete points. Deficiencies in image snapping, such as the lack of hysteresis and imperfect feature identification, become more serious when tracing paths. This suggests the importance of developing new interaction techniques that will work better with image snapping. Figure 10 shows an example where an alternative interface avoids path tracing. This particular example exploits the fact that the system knows the user is identifying an ellipse. The multiple click interface is also useful when the positioning must be completely manual because the edge detector fails completely (for example trying to circle the tire).

Presently, we are exploring image snapping inside of a prototype image editing application. However, since the inputs and outputs of the methods are standard, it is possible to construct a general purpose implementation. For example, an implementation at the window system level could intercept the mouse positions and look at the frame buffer directly. Such an implementation might construct and blur feature maps on the fly.

Our prototype implementation provides a testbed for exploring image snapping. Performing gradient descent on blurred feature maps creates precise pointing in images. We are exploring a variety of sources of features, including a number of edge detectors. Standard image editing interaction techniques are enhanced with a familiar method for providing precision, although new interaction techniques should better address image snapping's shortcomings.

ACKNOWLEDGMENTS

Lance Williams' encouragement and help with image processing were instrumental in the development of Image Snapping. Bruce Horn and Steve Rubin helped with Macintosh hacking. Heung-Yeung Shum, Pete Litwinowicz and Demetri Terzopoulos donated example code. Eric Chen and Ted Casey provided panoramic test images and scanning help. Bruce Horn, Frank Crow, Gavin Miller, Ken Turkowski, Heung-Yeung Shum, and Mike Kelley proofread drafts of this paper.

REFERENCES

[1] Adobe Systems, Inc. Photoshop™ 3.0. Computer Program, 1994.

[2] Ballard, D and Brown, C. *Computer Vision.* Prentice-Hall, 1982.

[3] Bier, E. Snap-Dragging: Interactive Geometric Design in Two and Three Dimensions. Ph.D. Thesis, University of California, Berkeley, 1989. Also as Xerox PARC report EDL-89-2.

[4] Bier, E and Stone, M. Snap-dragging. Proceedings of SIGGRAPH 86.Computer Graphics (20) 4: 233-240, 1986.

[5] Burt, P and Adelson, E. A Multiresolution Spline With Application to Image Mosaics. *ACM Transactions on Graphics* (2) 4:217-236, October 1983.

[6] Canny, J. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (PAMI-8) 6:679-698, 1986.

[7] Cayley, A. On Contour and Slope Lines. *London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science,* (18) 4:264-269, 1859.

[8] Chen, SE. An Image-Based Approach to Virtual Reality. Proceedings of SIGGRAPH 95. In *Computer Graphics* Proceedings, August 1995. This volume.

[9] Danielson, P. Euclidean Distance Mapping. *Computer Graphics Image Processing* (14) 3:227-248, November, 1980.

[10] Fletcher, R. *Practical Methods of Optimization.* John Wiley and Sons, 1987.

[11] Gleicher, M. A Differential Approach to Graphical Manipulation. Ph.D. Thesis, Carnegie Mellon University, 1994.

[12] Gleicher, M and Witkin, A. Drawing with constraints. *The Visual Computer* (11) 1, 1995.

[13] Gonzales, R and Wintz, P. *Digital Image Processing,* second edition. Addison-Wesley, 1987.

[14] Haralick, R and Shapiro, L. Survey: Image Segmentation Techniques. *Computer Vision, Graphics, and Image Proc.* (29) 100-132, 1985.

[15] Heckbert, P. Filtering by Repeated Integration. Proceedings of SIGGRAPH 86.*Computer Graphics* (20) 4:315-321.

[16] Hou, H, and Andrews, H. Cubic Splines for Image Interpolation and Digital Filtering. *IEEE Transactions on Acoustics, Speech and Signal Processing* (ASSP-26) 6:508-517, 1978.

[17] Hudson, S. Adaptive semantic snapping - a technique for feedback at the lexical level. In Proceedings CHI '90, pages 65-70, 1990.

[18] Kass, M, Witkin A, and Terzopoulos, D. Snakes: Active Contour Models. *Intern Journal of Computer Vision* (1) 4:321-331, 1988.

[19] Koenderink, J. The Structure of Images. *Biological Cybernetics* (50): 363-370, 1984.

[20] Leymarie, F and Levine, M. Fast Raster Scan Propagation on the Discrete Rectangular Lattice. *Computer Graphics, Vision, Image Processing: Image Understanding* (55) 1: 84-94, 1992.

[21] Lucas, B and Kanade, T. An Iterative Image Registration Technique with an Application to Stereo Vision. *Proceedings 7th IJCAI 1981,* pages 674-679, August 1981.

[22] MacViar-Whelan, P and Binford T. Intensity Discontinuity Location to Subpixel Precision. *Proceedings 7th IJCAI 1981,* pages 752-754, August 1981.

[23] Maxwell, J. On Hills and Dales. *London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science,* (40) 269:421-427, 1870.

[24] Miller, G and et. al. The Virtual Museum: Interactive 3D Exploration of a Multimedia Database. *Journal of Visualization and Computer Animation* (3) 183-197, 1992.

[25] Marr, D and Hildreth, E. Theory of Edge Detection. *Proc. Royal Society of London* (207) 187-217, 1980.

[26] Nalwa, V. *A Guided Tour of Computer Vision.* Addison-Wesley, 1993.

[27] Pratt, W. *Digital Image Processing,* 2nd ed. J Wiley and Sons, 1990.

[28] Press, W, Flannery, B, Teukolsky, S, and Vettering, W. *Numerical Recipes in C.* Cambridge University Press, 1986.

[29] Smith, AR. Tint Fill. Proceedings of SIGGRAPH 79. *Computer Graphics* (13) 2:276-283.

[30] Sutherland, I. Sketchpad: A Man Machine Graphical Communication System. Ph.D. Thesis, Massachusetts Institute of Technology, 1963.

[31] Tabatabai, A and Mitchell, O. Edge Location to Subpixel Values in Digital Imagery. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (PAMI-6) 2:188-201, March 1984.

[32] Torre, V and Poggio, T. On Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (PAMI-8) 2:147-163, 1986.

[33] Venolia., D. Facile 3D Manipulation. In Proceedings INTERCHI '93, pages 31-36, 1993

[34] Williams, L. Pyramidal Parametrics. Proceedings of SIGGRAPH 83. *Computer Graphics* (17) 3:1-11.

[35] Witkin, A. Scale Space Filtering. In Alex Pentland, ed., *From Pixels to Predicates*. Ablex, 1984. Reprinted from Proceedings of IJCAI '83.