Edge-Respecting Brushes

Dan R. Olsen Jr., Mitchell K. Harris
Computer Science Department
Brigham Young University
olsen@cs.byu.edu

ABSTRACT

Digital paint is one of the more successful interactive applications of computing. Brushes that apply various effects to an image have been central to this success. Current painting techniques ignore the underlying image. By considering that image we can help the user paint more effectively. There are algorithms that assist in selecting regions to paint including flood fill, intelligent scissors and graph cut. Selected regions and the algorithms to create them introduce conceptual layers between the user and the painting task. We propose a series of "edge-respecting brushes" that spread paint or other effects according to the edges and texture of the image being modified. This restores the simple painting metaphor while providing assistance in working with the shapes already in the image. Our most successful fill brush algorithm uses competing least-cost-paths to identify what should be selected and what should not.

Author Keywords
Paint, flood-fill, intelligent scissors, min-graph-cut, least-cost painting

ACM Classification Keywords
H.5.2 User Interfaces

INTRODUCTION

One of the most common tasks when interacting with images is the selection of a region to be modified. This can be as simple as painting fills into areas of a hand-drawn piece of animation or as complex as trying to separate a person's face from a noisy background. This basic task goes under the names of painting (if the task is to change the color of selected pixels), selecting (if a specific region is desired) or matting (when trying to separate a foreground object from background). Fundamentally these are all the same task which is to select some set of pixels and then apply some effect to those pixels. Frequently we combine selection and effect, as when painting with a brush. The brush shape and mouse position specifies the pixel selection and the paint is the effect. At other times we select separately and then apply an effect. We might "lasso" a region (selection) and then darken it (effect).

In this paper we present "edge-respecting brushes" as a new technique for smoothly combining intelligent selection and effect. The following figures illustrate the kind of edge-respecting brushing effects that are of interest. In figures 1-3 we show the original, the brush effect and an exaggerated color to clearly show the brush. Figure 1 uses a transparent dark brown brush to paint a tan onto skin. Edge-awareness helps to avoid the dress and background. Figure 2 shows a brush being used to darken selected areas of the background for contrast without infringing on the skin and blouse. Figure 3 uses a brush whose effect is to enhance contrast. We brush on regions of the hair where we want to show highlights. The edge-respecting algorithm helps us avoid the face and eyebrows.

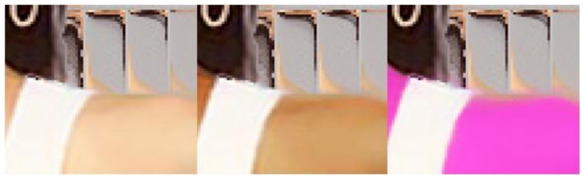
Figure 1 – Brushing on a tan on only the skin

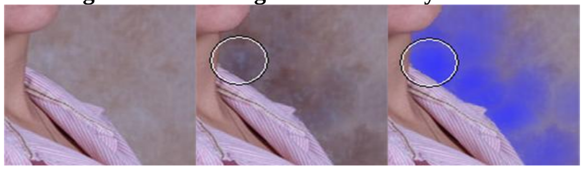
Figure 2 – Brushing in background darkening

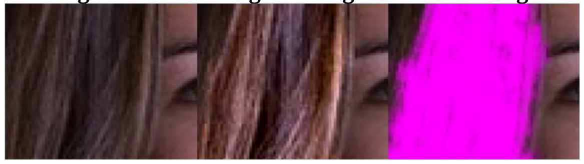
Figure 3 – Brushing in a contrast effect (hair only)

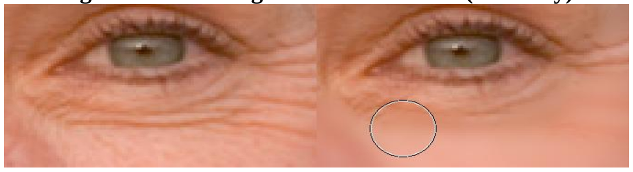
Figure 4 – Brushing away selected wrinkles with blur

Figure 4 shows a brush with a blurring effect to remove wrinkles. Too many wrinkles show age while a few key wrinkles show character. The edge-aware brush allows the artist to selectively remove wrinkles without eliminating the key details. We want smooth control over the algorithm as the artist decides which edges to remove and which to retain.

Interactive behavior

Interactively our fill brushes behave like the brushes found in most painting applications. The user presses a mouse (or pen) button and then moves the mouse around while the brush selects the pixels to be modified. The difference is that our fill brushes pay attention to the underlying image and adapt their pixel selection based on the mouse location and the texture/edges in the image.

Though in figure 2, the brush diameter is large the selection respects the edge between the blouse and the background. In figure 1 we also see texture and edges in the background. Though these edges are not as strong as the background/blouse edge, they do impact the algorithm. In figure 3 the hair provides many edges to our algorithm. The key idea in both of these examples is that the user has control of the brush position and can move over or avoid edges as desired. It is the user manipulating the brush rather than the algorithm that ultimately decides which pixels are selected. In figure 4 an algorithm would have a difficult time deciding which wrinkles to keep for character and which to blur to eliminate age. Our brush algorithm can assist the user in retaining important details while allowing them to brush over details that should be ignored.

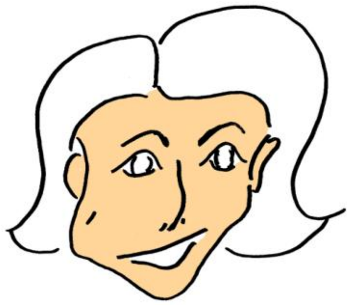
Figure 5 – Filling a line drawing

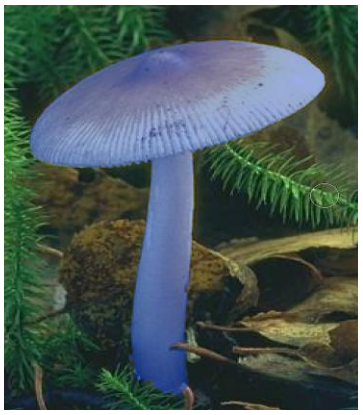
Figure 6 – Tinting a textured object

The task in figure 5 is to take a cartoon inking and then color the face flesh tone without coloring the eyes, hair or mouth. The coloring of a line drawing is a common task in animation and commercial drawing. The task is characterized by very little texture, strong edges and lots of implied connections across gaps in the sketch. The task in figure 6 is to tint a mushroom blue. This is not a very common task but it is an example of working with highly textured objects. The blue tint is to make the selection clearly visible in the paper for ease of discussion.

PRIOR WORK

Though our technique is a brush and functions with a “painting” metaphor the key prior work is actually in selection. There are five prior techniques of interest: flood fill, boundary specification, tri-maps, bilateral grids, intelligent paint and quick select.

Flood fill

The earliest automatic pixel selection tool was the flood fill. It is found in virtually every paint application. The mouse down point (MDP) acquires a base color (BC) from the pixel underneath the mouse. A recursive search is then performed to select every pixel that is similar to BC and adjacent to a pixel that is already selected. This algorithm suffers from two failings. The first is that if there are any “leaks” in the edges that enclose a region, the flood fill can select large areas of the image that are unintended. This explosive flooding is very frustrating to control. In figure 7 a flood fill, initiated at the cross, spreads through the gaps in the lines filling almost all of the drawing. Only the left eye escaped. Photoshop’s color replacement brush imposes a radius limit on such fills but still leaks through gaps in an edge.

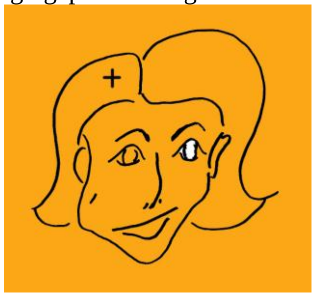
Figure 7 – Flood fill escaping through boundary gaps

A second failing of the flood fill is that it can leave speckles and islands in regions that appeared to the user to be smooth. The image on the left of figure 8 shows a drawing that appears smooth yet the flood fill on the right shows gaps and speckles. The tedious task of removing these eliminates most of the advantages of using flood fill.

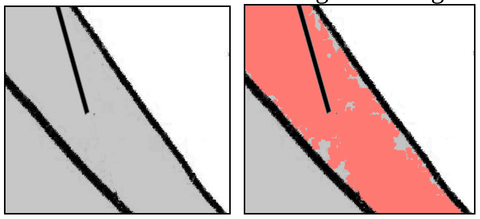
Figure 8 – Gaps and speckles from flood fill

Figure 9 shows even greater difficulties when texture is present. Additive fill selection (shift-click) was invoked 8 times using Photoshop's Magic Wand (flood fill) to get the selection shown in figure 9. The high texture frequently blocks the fill algorithm leaving regions isolated. Each isolated speckle must be individually filled or painted to complete the task. The most important interactive deficiency is the loss of control by the user. When the fill is initiated, the user is frequently surprised by the final result. This lack of predictability in the behavior is a challenge to usability.

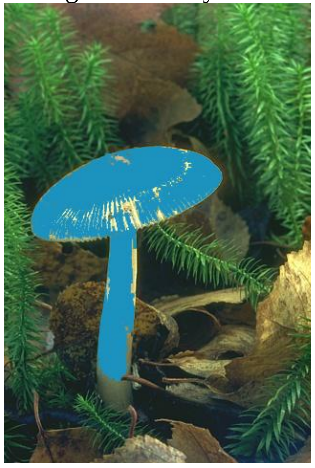
Figure 9 – Flood fill speckles

Boundary specification

Specification of a selected region by defining its boundary is also a very old technique. This technique is characterized by drawing the perimeter outline of the selected region. This "lasso" technique is a very slow and tedious process for complex regions.

Selection boundaries can be improved by algorithms that optimize selection paths to follow gradient (edge) boundaries. Snakes [5, 12] take an initial selection boundary and improve it by moving to stronger nearby edges. This suffers from a lack of user control when the desired boundary is not as strong as another nearby boundary. The "snapping" of an edge to a different one can be welcome or disconcerting depending upon the match between the snap and the desired result.

Better interactive control is provided by Intelligent Scissors [7], or the Magnetic Lasso (as it is known in Photoshop). In this technique the user draws the edge with the mouse, but the actual boundary follows the locally strongest gradient edge. This boundary is continuously echoed to the user so that they can see what will happen and attempt to correct the selected edge by moving the mouse. Techniques such as "freezing" or clicking on key points help the selected edge to remain locally stable and stick to key details. The continuous interactive echo of the edge placement is very helpful to the user and has made this a popular tool.

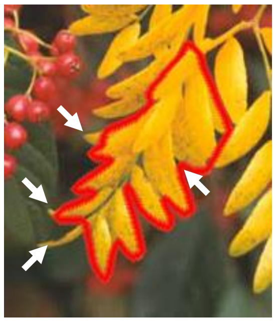
Figure 10 – Intelligent Scissors taking shortcuts

Edge optimization struggles with highly convoluted edges such as shown in figure 10 because the algorithm tries to take short cuts if there is an edge to follow. This can be corrected by manually setting anchor points in the tips and crevices, but this is more interactive work. Figure 11 shows the lower edge of our mushroom that is highly scalloped. The selection is indicated by the dotted line. This scalloping is a delicate detail that we do not want to lose, but the edge optimization of intelligent scissors smoothes it out and ignores the fine detail. On the opposite side of the stem it has again cheated on a corner.

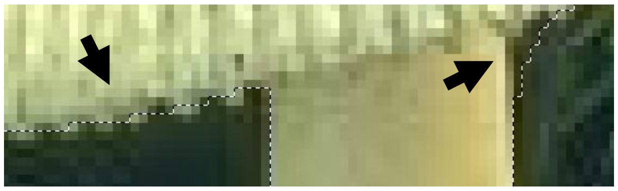
Figure 11 – Detail smoothing from optimization

Intelligent scissors also has problems with the cartoon face. It is easy to drag the control around the edges of the face, but the resulting fill covers all of the facial detail. To achieve the desired result we must use the tool to unselect around all the edges of each eyebrow, mouth or eye line so that the line is not covered. This is very time consuming. In this example, intelligent scissors may also skip over to the opposite side of a line we are trying to avoid if the other side offers a more economical path. This is very unsatisfactory.

The background contrast problem in figure 3 is also not well suited to boundary techniques. The width and intensity of the contrast region are an artistic choice that must be made continuously as the user views the resulting effect. The problem is that we have a painting task to which we are trying to apply an edge-defining tool. They are not well matched. All boundary specification techniques separate the selection from the effect.

Tri-map techniques

A number of interactive image segmentation systems have been developed around tri-map specification. Many of these use the graph-cut algorithm [3]. The basic idea is that the user designates some pixels as inside and some as outside. This forms a tri-map of foreground, background and undecided pixels. The image is viewed as a connected graph with pixels forming vertices and pixel adjacency forming the edges. Generally an edge cost function is computed by comparing the foreground and background training pixels. The selection boundary is then determined as the minimal cut of this graph that separates inside and outside training pixels. Prominent examples of this technique are Lazy Snapping [6] and GrabCut [10]. Graph-cut algorithms vary based on the energy function that they attempt to minimize and various ways of speeding up the calculation of the minimal cut of the graph. The graph-cut algorithm itself is quite complex.

Figure 12 shows the training strokes necessary to produce a reasonable selection of the face without including the eyes, hair or mouth using an implementation of Lazy Snapping. Green strokes indicate pixels that are outside and red strokes indicate inside. The number of strokes is quite modest and little accuracy is required of the user. Problems occur at the corners of the eyes where the automatic algorithm fights with the desires of the user as to where the edge should be. The implementation that we used also steals 1-2 pixels out of the lines making them thinner.

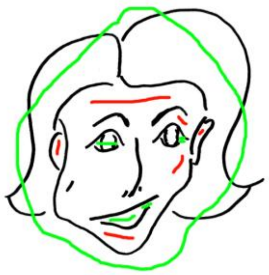

Figure 12 – Lazy Snapping of the face

The mushroom task was more difficult using Lazy Snapping. The leaf on the right of figure 6 was initially selected as part of the mushroom though it is quite distant from it. The reason is its similar color. Marking the leaf as unselected then caused regions of the mushroom to be unselected. Correcting the mushroom regions caused problems again with the leaf. Several back and forth operations finally corrected the problem. When a user makes a mark in one region of the screen and the selection changes in another, there is a serious usability problem. The user is now required to continuously scan the entire image for new errors after each correction.

Figure 13 shows a close-up of the mushroom stem that the algorithm refused to correct. Note that the dotted selection line cuts off before the bottom of the stem where the algorithm gets confused by the branch protruding across the stem. There are training strokes in the lower area but the algorithm has chosen to ignore them so as to optimize its function. Automatically computed selections can save users significant amounts of time.

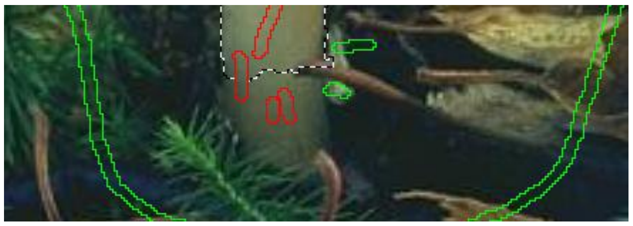

Figure 13 – Lazy Snapping selection failure.

There are a variety of formulations used to produce an appropriate selection from a tri-map (inside, outside, unknown). Each of these optimizes in various ways to eliminate deficiencies of various algorithms. Many of the problems demonstrated above are solved in other formulations. There is still a fundamental problem with the tri-map technique's interactive style. The user specifies inside/outside with some strokes and the algorithm then makes its best choice. If the image, task and algorithm are well matched then the result is highly predictable and highly satisfactory. If they are not well matched then the result is frequently not predictable. This leads to a user/algorithm negotiation through additional input as the user tries to convince the algorithm to adopt a more acceptable selection. This is particularly problematic in cases like figures 3 and 4 where the user is not sure what the right choice should be until the results are seen.

The fault is not so much in the matting algorithm as in the tri-map approach to interaction. We need an approach that is interactively more fluid than the tri-map offers. The problem is that the user is not painting or selecting an image, they are training an algorithm to paint or select an image.

Early tri-map approaches created binary selections which are not good in figures 2 or 4. Improvements to provide alpha matte selections can produce nice results but are generally not interactive at interactive speeds. Some algorithms take almost 2 minutes to solve. We have not referenced these numerous algorithms because they are not appropriate to our task.

A variation on the tri-map strategy is Soft Scissors[11]. In this approach the user is given a wide brush with which to trace out the boundary of a region. At each movement of the brush some pixels on the brush edge are identified as background and others foreground with the interior pixels identified as unknown. The tri-map selection is then solved locally, providing a more interactive selection process. However, if the selection is not what the user desired, the user has less control than before in correcting the choice. Interactivity is increased by the real-time nature of the algorithm but control is not. Soft Scissors has the nice property of adapting to fuzzy edges such as fur where the selection may be indistinct.

Another property of tri-map algorithms that will become important in our brush implementation is the tri-map model for inside/outside. All of these algorithms take the pixels identified as inside and those identified as outside and develop a model for evaluating whether a given pixel is likely to be inside or outside. The algorithms used to develop such models are quite diverse. They might be as simple as an average color for inside and outside or more powerful machine learning models. The key point is that they all assume that a sample of outside pixels is known and a sample of inside pixels is also known. In our brush strategy, these facts are not known, primarily because the users themselves may not know them until they perceive the resulting effects of their brush.

One tri-map technique proposed by Bai and Sapiro [1] is of interest because instead of graph-cut it uses a *geodesic distance*. A pixel is labeled foreground or background depending upon whether it is closest to a foreground or background pixel using a geodesic distance. We can compute the distance between two pixels by summing the pixel transitions along the least cost path. If the pixel transition costs are weighted by the change in pixel color across the edge or by the similarity of the new pixel to the original source pixel, we have a geodesic distance. Paths that must go through dissimilar pixels acquire a greater cost and thus a greater distance. If we seed Dijkstra's algorithm with foreground and background pixels it will compute the least cost path to each pixel being considered and thus produce a labeling. We will use a similar approach to this in our brush formulation.

The indirection problem

All of the tri-map techniques share the indirection problem which is that the user is not painting but rather training an algorithm that then does the painting. The user is not applying a desired effect to an image but rather trying to coax an algorithm into generating the desired effect. One advantage of tri-map interaction is its brevity of expression for many situations. A few strokes of inside and outside scribbles can quickly generate a good selection region. However, in figures 2, 3 and 4 this is much more difficult because of the user judgment required to get it right. The indirect UI offered by tri-maps in problematic in many situations.

Painting methods

Some prior methods do not use tri-maps but instead provide a painting metaphor over an underlying algorithm and data structure. Intelligent paint [8] works by finding "watershed" regions of similar color and filling those regions as the user moves a brush through them. This algorithm has the same automation properties as graph-cut in that the user intent may not match the algorithm's assumptions. However, it does have a more paint-oriented user interface than the graph-cut techniques. It also behaves locally rather than having input in one region causing changes in a totally different region. It does, however, have a "surging" problem. That is as the brush moves from region to region the paint appears to surge into new regions. This discontinuous flow of paint is problematic for users. Photoshop's color replacement brush exhibits this surging behavior. Some of our early brushes had similar problems as will be discussed later.

The bilateral grids of Chen et. al.[4] also support a painting style of interaction. A brush's transparency can sometimes be modeled as a Gaussian function of the distance between a brush's center and a pixel to be painted. This formulation found in many paint programs ignores the strong edges found in many images and paints or blurs across them. The bilateral filter adds a second term which is a Gaussian of the distance in color space rather than pixel space. This then sharply reduces a brush's effect on pixels that are very different in color. This is described by Chen et. al. as an "edge-aware" brush. However, unlike the graph-cut algorithms that truly find edges, this algorithm is color difference aware rather than edge-aware, which is not quite the same thing. If one is painting flesh tone onto the cartoon face in figure 12 the bilateral filter will not paint the lines (a good thing) but if the brush got too close to the hair line, for example, the brush would jump over the line (a bad thing) because the white pixels in the hair area are not very different from the brush pixel. A similar behavior occurs in Photoshop's color replacement brush in "discontiguous" mode.

The bilateral filter on its own is too expensive for interactive work. The bilateral grid is a data structure that renders bilateral filters fast enough to be used in a painting interaction if the GPU of a graphics card is available for the computation.

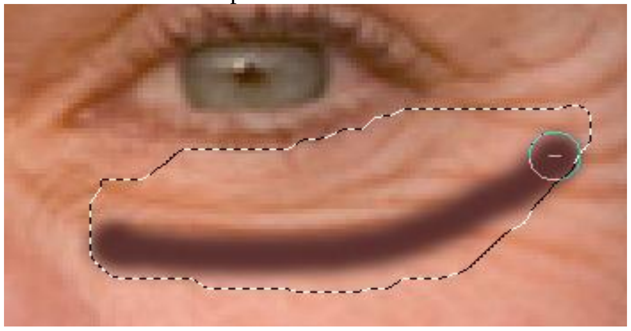

Figure 14 – Quick Select moving beyond the brush track

Quick Select

Photoshop CS3 has a Quick Select tool that allows a user to "paint" a selection using a brush. From the pixels that the brush passes over the extent of the selection is inferred. The algorithm for this is not public. This technique has a nice interactive feel. This technique is an "eager" technique in that it tries to reach out as far as indicated by the pixels that have been brushed over. In figure 14 the dark streak shows the pixels that the user passed the brush over. The dotted line indicates the inferred selected region. This algorithm has the same surging property as intelligent paint. As the user moves the brush the region tends to jump out in ways that are reasonable but not always predictable. It does have a “subtract selection” mode where the user can push back to unselect regions that were too aggressively selected.

As with the tri-map algorithms there is still an indirection issue in its interactive style. The user pushes around the selection edge until an appropriate edge is produced and then the effect is applied to the selected region. We want a method with more direct control and simultaneous with the desired effect.

BRUSH DEVELOPMENT

In this paper we describe a brush technique for filling, tinting, selecting, blurring or any other effect that the user might choose. The role of the brush is to select pixels to be modified and the apply the effect. What we want is a technique that is very easy to learn and very easy for the user to control. We see this issue of user control as being problematic in the more automated techniques.

User interface

Our user interface is quite simple. When the left mouse button (or pen tip button) is down the brush is engaged. Starting from the mouse position the brush pushes outwards to select the pixels that are to be modified. This is much like water color flowing out from a brush onto damp paper. As pixels are selected, the desired effect (paint, tint, blur, select, etc) is applied and feedback supplied to the user. Each time the mouse moves, the brush pixels are reselected using the brush’s algorithm. The pixel modification effect is not confirmed into the image until explicitly requested later by the user. The modified pixels are stored in a separate display layer.

Using the right mouse button applies the brush selection in exactly the same way except that the pixels are unselected rather than selected. This provides a natural mechanism for undoing any mistakes. This symmetric brush behavior has been very effective for our users because mistakes do regularly occur. Reversing the mistake is conceptually simpler than undoing the last stroke because it is pixels that are of interest to the user, not the paint strokes.

Lastly our brushes have a “radius,” which defines the extent of the brush’s selection behavior. This is analogous to the radius of a traditional round brush but not exactly so. We have attached the radius adjustment to the mouse wheel for easy manipulation. The radius can also be associated with pen pressure or adjusted by other devices with the non-dominant hand. This user interface is not particularly novel but it is the foundation for all of the techniques that follow.

Simple cost-threshold-fill

Our first fill brush technique uses Dijkstra’s algorithm to find the least cost path between the mouse position and each of the surrounding pixels. This is very similar to the traditional flood fill algorithm except for the termination condition. The traditional queue-based flood fill ignores a pixel *P* when the difference between its color and the color of the mouse position pixel *M* is greater than some threshold. Our cost-threshold-fill brush modifies this flood fill algorithm slightly. It ignores a pixel *P* when the least cost path between *M* and *P* exceeds a threshold. This “fill until threshold” technique imposes the limits on the brush while retaining many of the characteristics of a flood fill that stops at strong edges. By moving the mouse point *M* around, the brush moves naturally into new areas under user control. This modified algorithm is shown in figure 15.

```
mark all pixel states as “unknown”
M = info about the pixel under the mouse
T = cost threshold to limit the fill
Q = a new least cost priority queue
Q.add(M,0);
while (!Q.empty())
{   [P,cost]=Q.nextLeastCost();
    if (P.state==unknown && cost<=T)
    {   P.state=selected;
        foreach N in P.neighbors()
        {   Q.add(N, cost+C(M,P,N) );
        }
    }
}
```

Figure 15 – Fill brush with cost

This accumulated cost is very close to the geodesic cost of Bai and Sapiro. The heart of this brush algorithm is the definition of the transition cost *C* of moving from pixel *P* to a neighboring pixel *N*. The sum of these costs defines the cost of some path. More formally we define *minPathCost(U,V,C)* to be the cost of the minimal path from pixel *U* to pixel *V* using the transition cost function *C*. A pixel *P* is said to be selected by a brush if *minPathCost(M,P,C)<threshold* where *M* is the mouse position. The variation of Dijkstra’s algorithm shown in figure 15 computes *minPathCost* by adding the result of the transition cost function *C* at each step. This algorithm actually defines a family of possible brushes depending upon our choice for the function *C*.

We can start with a simple transition function. The function *baseCost(M, P, N)=diff(M.color,N.color)* defines the cost of moving from pixel *P* to pixel *N* as being the difference between the color of *M* and the color of *N*. Using *baseCost* by itself as the transition function, the result is very similar to the traditional flood fill. The difference is that small transition costs can accumulate to terminate the fill rather than a single dominant difference being the terminator as in flood fill. The problem is that small transition costs may not accumulate fast enough to provide good user control. On a smooth region of uniform color the brush behaves like unconstrained flood fill because the costs do not accumulate rapidly enough. Very low transition costs in smooth regions allow the fill to cover many pixels before exceeding the threshold. The presence of texture will slow down the spread of the brush and sharp edges will stop it. Figure 16 shows how the brush flows wildly in smooth regions. The mouse point is at the cross hair. This is similar to the frustration of wet-on-wet watercolor where the flow of paint is difficult to control. We refer to this unexpected flow of paint as "surging". It is very disconcerting to the user.

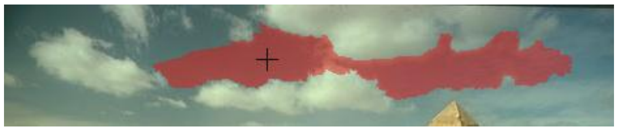
Figure 16 – Flowing problems with *baseCost()* brush

We can improve the brush's behavior by modifying *baseCost*() to include a term for the distance moved in the pixel-to-pixel transition. The prevents wild surging but still produces paint regions that vary widely depending on the amount of texture accumulated in the cost. Mouse point A in figure 17 is the origin of a brush using a threshold of 44. The blend between transition cost and distance is 0.5. Because the sky is relatively uniform the brush spreads quite far up to the edge of the cupola. Mouse point B is exactly the same brush in a more open area that has little transition cost. Mouse point C is the identical brush in a textured region.

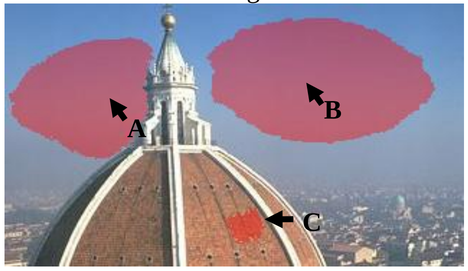

Figure 17 – Variants in behavior with *B*=0.5 and *T*=44

Figure 18 shows another difficulty with this flooding brush algorithm, which is that it tends to leave holes just as the original flood fill did in figure 9. This is not very satisfactory.

One variation is the maximum radius brush that uses the modified cost function but also imposes a maximum distance around the mouse point that restrains uncontrolled surging. The problem with holes still remains and a tendency when pushed close to a strong edge to suddenly surge across in a way that feels very uncontrolled to the user. This surging is similar in behavior to Photoshop's color replacement brush. The color replacement brush attempts to resolve this problem by giving the user indirect control of a threshold parameter rather than as a natural part of the painting process and is awkward to control.

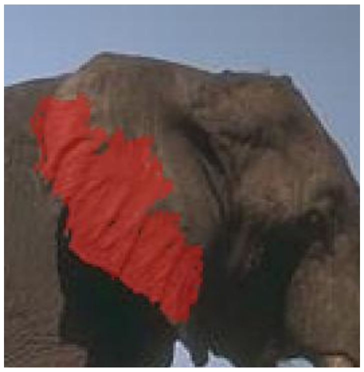
Figure 18 – Flood brush blend effects

Perimeter competition

Our most effective brush technique is based on a competition between paint and not-paint pixels. We use an approach that is reminiscent of the tri-map, but as will be discussed, there are important differences. In this technique the mouse point is seeded as a paint (to be selected) pixel, all of the points on the perimeter of the brush are seeded as not-paint (not selected) pixels and all other pixels are unknown. The algorithm tries to find the best labeling (paint/not-paint) for all pixels inside the brush radius by choosing paint or not-paint based on which seed point is "nearest" the pixel using our cost function described earlier. This is very much like the geodesic distance approach. The algorithm is shown in figure 19. Figure 20 shows this competing brush in action and how it respects the scalloped edges of the mushroom while providing user control of where to paint.

```
mark all pixel states as "unknown"
M = info about the pixel under the mouse
Q = a new least cost priority queue
R = radius of the brush
Q.add(M,0,paint);
foreach pixel C on circumference of the brush
{    Q.add(C,0,not_paint);
}
while (!Q.empty())
{    [P, cost, source]=Q.nextLeastCost();
     if (P.state==unknown)
     {    P.state=source;
          foreach N in P.neighbors()
          {    if (dist(N,M)<=radius)
               {    Q.add(N,
                              cost
                                   +diff(N.color,P.color)
                                   +dist(N,P),
                              source);
               }
          }
     }
}
```
Figure 19 – Competing fill algorithm

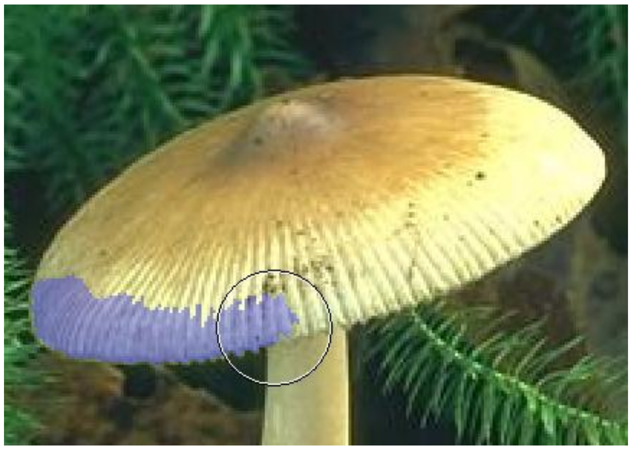
Figure 20 – Competing fill brush

The final state of a pixel P using the algorithm in figure 19 is.

```
if (minPathCost(M,P,colorDif())<minPathCost(C,P,colorDiff())
        paint
else
        not-paint
```

This competition approach solves many of the problems with our other fill brush techniques. The isolated speckling problem does not occur because if an island is surrounded by paint pixels the least cost path to all pixels in the island must come through one of those paint pixels. The island therefore becomes paint rather than remaining speckled. The surging problem when crossing an edge is also mitigated. When the closeness of the mouse to an edge causes selection to overrun the edge into a smooth area, the competing paths from the not-paint perimeter are also moving across that same smooth area. The effect of pushing across an edge is much more subtle and controlled than with the thresholded brush. The competing technique mitigates many of the texture-related problems because both paint and not-paint are working through the same textures. The result is a brush that respects edges but has a smoother more controlled feel than Photoshop's color replacement brush.

The brush radius in figure 20 shows why our algorithm is not exactly a tri-map. The mouse point is in the center of the brush circle and is inside the mushroom cap. This point should definitely be painted if we are trying to tint the cap. However, a majority of the pixels on the perimeter of the brush are also on the mushroom cap. They therefore are not background pixels as in the tri-map formulation. They are labeled is pixels not to be painted for this particular brush point. As the brush moves and successive new brush points are established, the perimeter also moves as does the paint. The algorithm in figure 19 is to calculate a brush shape for a given mouse point, not for the entire task. At each mouse movement event this algorithm is rerun to establish a new brush shape. This does not behave like a tri-map it behaves like a brush. The total task is controlled by the user's brush movement, not the algorithm.

A second issue is that the pixels of the perimeter are very similar to the mouse point pixel. In addition most of the pixels outside of the mushroom cap are in the stem. The color and texture of the stem is very similar to that of the cap. This pixel similarity between paint and not-paint pixels eliminates the possibility of training a foreground/background model as is done in most tri-map implementations. The brush model of interaction precludes such model development. In figure 3 where we are tinting hair with contrast, the entire brush region usually involves hair and thus there is no intrinsic difference in the paint/not-paint pixels. With the face in figure 4 the majority of the pixels are flesh color. The algorithm is just respecting the wrinkles with no true foreground or background differences to model.

For our brush technique to work, we cannot directly modify the image on every mouse movement. The reason is that the modified pixels become more uniform in color from preceding iterations of the brush technique. This causes everything to behave badly. The selection (represented by pixel State in the algorithms) must be separate from the image and the desired effect, paint, tint, blend, lighten, darken, etc. must be performed on a transparent overlay plane.

As can be seen in figure 20, the perimeter of the brush is about twice as far out from the mouse point as the actual extent of the paint. This can be disconcerting to users because the perimeter circle gives little indication of the extent of the brush. This is easily resolved by initializing all perimeter seed points with a cost equal to the radius of the brush. By "handicapping" the perimeter pixels the paint region will push out closer to the perimeter. On a completely smooth region the brush pushes out to the perimeter. This cost seeding does not change the behavior of the brush but makes the perimeter more understandable to the user.

Alpha brush values

In many cases a hard edged brush is not desired. In figure 3 the contrast should fade smoothly into the untinted hair. In figure 2 the darkening of the background should fade smoothly into the unmodified portions of the background. In figure 4 the blurring effect should fade smoothly into the unmodified skin. Our alpha blending approach is modeled on the soft brushes of paint programs such as Photoshop. The alpha matting problem of selecting regions with fuzzy edges is an important but different problem from our alpha brushes. For that problem we would recommend Soft Scissors. Our alpha blending goal is to remove harsh edges from our brushes. What we want is an alpha value of 1.0 at the mouse point that drops off slowly across smooth areas and quite suddenly against sharp edges. We want our brush edges to be crisp against image edges and a smooth gradient in smooth regions.

Many alpha matting approaches use the foreground/ background model to guide the alpha interpolation. However, as described earlier we have no such model. Once a brush is formed we do have a set of pixels defined as inside and outside of the brush. We can use a neighborhood of these pixels to define the alpha value. For each pixel *C* that is inside the brush we define a *voting window* around that pixel. Controlling the size *W* of this window controls the width of the blurred region at the edges of the brush. A *W* of 0.0 will create a hard-edged brush. *W* is never larger than the distance between *C* and the mouse point *M*.

Figure 21 shows the algorithm to compute alpha for each pixel in the brush. The general idea is to sum weighted votes for all inside pixels around some pixel *C* and a separate vote for all outside pixels in the window. The votes are weighted by a combination of the geometric distance between the pixels (farther away have less influence) and color difference (pixels that look different have less influence). It is the color difference term that causes a sharp falloff of alpha against strong edges. The alpha value is determined by the percentage of the vote belonging to inside pixels. However, because only inside pixels are considered for alpha values, they never have all outside neighbors and thus alpha cannot reach zero. Experience shows that inside pixels usually have at least 33% inside pixel neighbors. The inside neighbors (*iN*) term accounts for this and stretches alpha down to an appropriate range.

```
M = The mouse point for the current brush instance
I = The image being painted
B = computeBrush(M, I) // pixels in the brush
W = The maximum size of the voting window
foreach pixel C in B
{   w=min(W, dist(C,M)-1 );
    iVote=0; oVote=0;
    foreach pixel F such that dist(C,F)<=w
    {   vote=(1/(1-dist(C,F))*(1/(1+colorDiff(C,F)));
        if (F is in B)
            iVote+=vote
        else
            oVote+=vote;
    }
    iN=.33;
    C.alpha=1/(1-iN)*(iVote/(iVote+oVote)-iN);
}
```

Figure 21 – Computing Alpha for a brush

Paint compositing

The brush algorithm described above gives us a map of alpha values in the region surrounding the mouse point. This is computed on each mouse-move event, generating a new brush map (***B***) each time. All of the brush maps are accumulated to form the paint map (***P***) which is a set of alpha values for the entire image. Given the paint map, an image (***I***) and an effect to be painted (***E***) we compute a displayed image (***D***) to be shown to the user. In addition, there is an opacity value (*O*) that controls the maximum opacity value of the effect being painted. This is equivalent to brush or layer transparency in many painting programs. For example in figure 2 we want to darken the background not obscure it. Using a value of *O* that is closer to 0.2 will accomplish this. Given these values the compositing function[9] to produce ***D*** is:

$$\boldsymbol{D}_{x,y}=(1-O*\boldsymbol{P}_{x,y})*\boldsymbol{I}_{x,y}+O*\boldsymbol{P}_{x,y}*\boldsymbol{E}_{x,y}$$

The effect image ***E*** can be any image. For traditional painting it can be a solid color. For smoothing tasks such as figure 4 it can be a Gaussian blur of image ***I***. It can also be an image or texture to be painted into a region. At each mouse movement the brush map ***B*** is recomputed from ***I*** and the current mouse position. Note that image ***I*** is not modified during the process because the construction of the brush map ***B*** on each mouse move is dependent upon ***I***. Successive modifications to ***I*** would seriously distort the brush behavior. Each brush map instance $\boldsymbol{B_i}$ is composited with the previous paint map $\boldsymbol{P}_{i-1}$ to produce a new paint map $\boldsymbol{P}_i$. The function is:

$$\boldsymbol{P}_i=\max(\boldsymbol{B}_i, \boldsymbol{P}_{i-1})$$

Thus the paint map ***P*** accumulates alpha values that are used to blend the effect with the image. At some point the image ***I*** is replaced by ***D***. This can be at mouse-up but we have found it best to defer this longer until an entire paint task is complete.

At the beginning of this paper we indicated that the right mouse button would perform “unpainting” to correct user errors. For unpainting we use the same brush map, but we composite it with the paint map using the function:

$$\boldsymbol{P}_i=\min(1-\boldsymbol{B}_i, \boldsymbol{P}_{i-1})$$

Unlike the tri-map techniques, it is not the algorithm but the successive mouse movements of the user that ultimately define where an effect is to be painted. Thus there never is a foreground/background classification, only successive brush maps that alter their shape based on the edges in the underlying image. This provides users with *assisted control* of their task.

EVALUATION

To evaluate our fill brush algorithm we look both at its algorithmic complexity and a user study that compares the brush to other selection techniques. The space complexity of the brush is dependent upon the maximum size of the queue. The queue can never have more entries than there are pixels inside of the radius of the brush. This makes the maximum space complexity O(πR^2) The actual behavior is much better than that. The pixels in the queue are on the frontiers of the region moving in from the perimeter and out from the mouse point. The number of pixels on these frontiers at any point in time rarely exceeds the circumference of the brush. This makes the average space complexity less than O($2\pi R$).

The algorithmic complexity involves the number of pixels visited and the complexity of the queue's add() and next() methods. The algorithm visits every pixel at most 4 or 8 times depending on whether diagonal pixels are considered neighbors. Every pixel will be visited at least once. The complexity of the queue is the log of the number of items in the queue. The maximum algorithmic complexity therefore is O($\pi R^2 * \log(\pi R^2)$).

User experience

For our user study we compared three techniques.

- Fill Brushes using the perimeter competition algorithm.
- Photoshop's Magnetic Lasso, which is an implementation of Intelligent Scissors
- Photo Crop Editor, which is an implementation of Lazy Snapping.

We did not have readily available implementations of the other graph-cut algorithms. The problems that we described earlier and the behavior of the algorithms will be similar among all graph-cut techniques. They vary in the energy function that they minimize and in how they achieve interactive speeds. The problems of these algorithms lie in their doing different things than the user intends.

We used six images from the Berkeley segmentation data set [2] with each of 9 subjects. The subjects were all university students with no more than casual experience in segmenting images. Each subject received the images in the same order. For a given subject/image the algorithms were presented in random order. Because we had no mechanism for evaluating selection accuracy over time we give the users a specific amount of time and instructed them to achieve as much accuracy as possible within that time. We then compared the user's results to the average Berkeley annotation. Figure 22 shows the average edge agreement of each technique at accuracy distances of 3 through 6 pixels. The various segmentations provided in the Berkeley data did not themselves agree at accuracy less than 3 pixel which would make more accurate comparisons spurious. Our brush technique shows a good match to the gold standard. All of these differences are statistically significant ($p<0.01$). When asked, eight of the users preferred the brush to the other two techniques. One subject thought the automatic snapping to the image was more fun. Our conclusion is that the brush technique is as accurate or better than others but that its key contribution is the interactive control that supports.

D	**Brush**	**Lasso**	diff		**Snap**	diff
3	98%	95%	3%		93%	5%
4	99%	96%	3%		96%	3%
5	99%	96%	3%		97%	2%
6	99%	96%	3%		97%	2%

Figure 22 – Edge agreement in neighborhoods 0-6 pixels

REFERENCES

1. Bai, X., and Sapiro, G. "A Geodesic Framework for Fast Interactive Image and Video Segmentation and Matting," *IEEE International Conference on Computer Vision, ICCV 2007,* (2007), pp. 1-8.
2. Berkeley Segmentation Dataset and Benchmark 2007. http://www.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/segbench/
3. Boykov, Y., and Jolly, M.P. "Interactive Graph Cuts for Optimal Boundary & Region Segmentation of Objects in N-d Images," In *Proceedings of ICCV 2001,* (2001).
4. Chen, J., Paris, S., Durand, F., "Real-Time Edge-Aware Image Processing with the Bilateral Grid," *ACM Trans. on Graphics*, 26, 3 (July 2007).
5. Kass, M., Witkin, A., and Terzopoulos, D., "Snakes: Active Contour Models," in *Proceedings of the First International Conference on Computer Vision,* (1987) London, England, pp. 259-68.
6. Li, Y., Sun, J., Tang, C. K., Shum, H. Y. Lazy Snapping. In *Proceedings of the International Conference on Computer Graphics and Interactive Techniques*, (2004) ACM, pp. 303-308.
7. Mortensen, E. N. and Barrett, W. A. Intelligent scissors for image composition. In *Proceedings of the 22nd Annual Conference on Computer Graphics and interactive Techniques* S. G. Mair and R. Cook, Eds. SIGGRAPH '95. (1995) ACM Press, New York, NY, 191-198.
8. Mortensen, E. N., Reese, L. J. and Barrett, W. A. "Intelligent Selection Tools," *Computer Vision and Pattern Recognition (CVPR 2000)*, IEEE, pp776-777 vol 2.
9. Porter, T. and Duff, T. "Compositing Digital Images," *Computer Graphics* 18(3), (July 1984).
10. Rother, C., Kolmogorov, V. and Blake, A. GrabCut: Interactive Foreground Extraction Using Iterated Graph Cuts," In *Proceedings of the International Conference on Computer Graphics and Interactive Techniques*, (2004) ACM, 309-314.
11. Wang, J., Agrawala, M, and Cohen, M. F. "Soft Scissors: an Interactive Tool for Realtime High Quality Matting," *SIGGRAPH '07*, (2007).
12. Williams, D. J. and Shah, M. "A Fast Algorithm for Active Contours and Curvature Estimation," *CVGIP: Image Understanding*, 55,1(1992) , pp. 14-26.