Paint Selection

Jiangyu Liu*
University of Science and Technology of China

Jian Sun†
Microsoft Research Asia

Heung-Yeung Shum†
Microsoft Corporation

Figure 1: *Left three: the user makes a selection by painting the object of interest with a brush (black-white circle) on a 24.5 megapixel image. Instant feedback (selection boundary or image effect) can be provided to the user during mouse dragging. Rightmost: composition and effect (sepia tone). Note that the blue scribbles are invisible to the user. They are drawn in the paper for illustration only.*

Abstract. In this paper, we present Paint Selection, a progressive painting-based tool for local selection in images. Paint Selection facilitates users to progressively make a selection by roughly painting the object of interest using a brush. More importantly, Paint Selection is efficient enough that instant feedback can be provided to users as they drag the mouse. We demonstrate that high quality selections can be quickly and effectively "painted" on a variety of multi-megapixel images.

Keywords: image segmentation, user interface

1 Introduction

Selections or layers are one of the most powerful representations in image editing. Many applications, from object cut-and-paste to local color/tone adjustments, require a local selection. Recently, graph-cut-based approaches [Boykov and Jolly 2001; Li et al. 2004; Rother et al. 2004] have greatly simplified this tedious and time-consuming job. Furthermore, efficient optimization [Boykov and Kolmogorov 2001] also enables users to get instant feedback, which is critical for interactive image editing.

*This work was done when Jiangyu Liu was an intern at Microsoft Research Asia. Jiangyu Liu is with MOE-Microsoft Key Laboratory of MCC, USTC. Email: eeyu@ustc.edu

†Email: {jiansun, hshum}@microsoft.com

Gradually, however, these global optimization based approaches become incapable of providing instant feedback on multi-megapixel (10–25Mp) images produced by today's digital cameras, because the complexity of graph-cut optimization is at least proportional to the number of pixels. Even worse, it is difficult to parallelize the graph-cut optimization to take advantage of modern multi-core machines. Without instant feedback, users have to work in a dreary "act-and-wait" mode, and the total interaction time can increase significantly.

The feedback delay also makes applying local image effects or adjustments (e.g., saturation, NPR) during the selection difficult, because users may not be sure what a good selection should be before viewing the results with applied effects.

Our approach. In this work, we propose Paint Selection, a progressive painting-based tool for local selection in images which can provide instant feedback during selection, on multi-megapixel images. Users select image regions by directly painting the object of interest with a paint brush. Unlike conventional painting operations, users need not paint over the whole object. Instead, the selection can be automatically expanded from users' paint brush and aligned with the object boundary, as shown in Figure 1.

Paint Selection is very efficient and can provide instant feedback as users drag the mouse. The efficiency comes from a progressive selection algorithm and two new optimization techniques: *multi-core graph-cut* and *adaptive band upsampling*. Most importantly, by integrating progressive selection and designed optimization, the number of pixels that needs to be considered is significantly reduced, making the process much more efficient.

Paint Selection is based on a key observation: interactive local selection is a progressive process in which users create the selection step by step. Therefore, it may not be necessary to solve the global optimization problem from scratch for each user interaction. Instead, Paint Selection progressively solves a series of local optimization problems to match users' directions, without sacrificing usability and selection quality.

Paint Selection has several additional UI advantages: i) enabling *interchangeability*, users can use different selection tools (including ours) in any order to complete a job (section 2.2.3); ii) handling scribble conflict, the annoying scribble conflict issue is intelligently solved by a scribble competition method (section 2.2.4); iii) allowing local refinement of the selection based on the viewport (section 3.3).

1.1 Related works

Scribble-based selection. The selection is computed based on a number of foreground and background scribbles specified by users. According to different formulations, these approaches include graph-cut based [Boykov and Jolly 2001; Li et al. 2004; Rother et al. 2004], geodesic distance based [Bai and Sapiro 2007], matting based [Wang and Cohen 2005; Levin et al. 2008], and random walk based [Grady 2006].

Generally, for the task of binary selection, graph-cut-based methods are faster and produce higher quality results. For local image adjustments, recent scribble-based edge-preserving interpolation [Lischinski et al. 2006; Li et al. 2008; An and Pellacini 2008] may be more appropriate.

Painting-based selection. In approaches like Intelligent Paint [Reese 1999], Bilateral Grid [Chen et al. 2007], and Edge-respecting Brushes [Olsen and Harris 2008], users directly paint the object using a brush. The selection is locally created based on the brush (position and extent) and image content. In contrast to a scribble-based UI, the painting-based UI is unique in three aspects: 1) it updates and displays the selection during mouse dragging; 2) no scribbles are displayed to users; 3) users draw mainly on the foreground.

A great example of the painting-based UI is Adobe Photoshop CS3&4's Quick Selection [Adobe Photoshop]. Our approach is inspired by this tool. However, to the best of our knowledge, the technique used by this tool has not been published.

Boundary-based selection. Snake [Kass et al. 1987] and Intelligent Scissor [Mortensen and Barrett 1995] require users to trace the whole boundary of the object. When the object has a complicated boundary, or the object is in a highly-textured region, users have to put great effort into iteratively correcting the selection.

2 Paint Selection

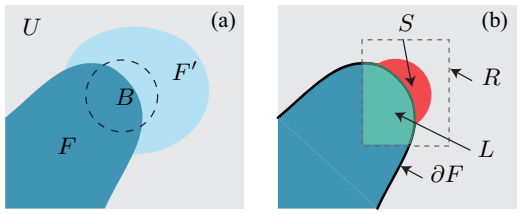
In this section, we present the UI and algorithm of Paint Selection.

2.1 User interface

To select an image region, users paint the object of interest with a brush while holding the left mouse button. Unlike previous scribble-based systems which compute results after the mouse button is released, we trigger a selection (optimization) process once users drag the mouse into the background, as illustrated by Figure 2 (a). While users drag the mouse within the existing selection, nothing happens. The scribbles are hidden to avoid distracting users. (In this paper and the accompanying video, scribbles are drawn only for illustration purpose.)

Once the selection process is triggered, we apply a progressive selection algorithm, described below, to expand the selection. The expanded selection is computed in a very short time interval (usually under 0.1 seconds) and instantly displayed to users.

By inspecting the new selection, users can continuously drag the mouse to expand the selection, until they are satisfied. Users need not paint over the entire area since the selection can be properly expanded from the brush to the nearby object boundaries.



B - user brush F - existing selection F' - new selection
U - background S - seed pixels R - dilated box
L - local foreground pixels ∂F - frontal foreground pixels

Figure 2: *Progressive selection. (a) progressive selection is triggered when users' brush B touches the background U. New selection F' is immediately computed and added into existing selection F. (b) R is a dilated bounding box of the seed pixels S. ∂F is the interior boundary of the existing selection F.*

Using the right mouse button swaps the roles of the foreground and background, so users can expand the background if necessary.

2.2 Progressive selection algorithm

Here, we introduce the progressive selection algorithm which supports the user interface above.

2.2.1 Progressive labeling

Given the existing selection F and current brush B, the progressive selection computes a new selection F' in the background U, as shown in Figure 2(a). Once the new selection F' is obtained, the existing selection is updated as $F = F \cup F'$ for the next user interaction.

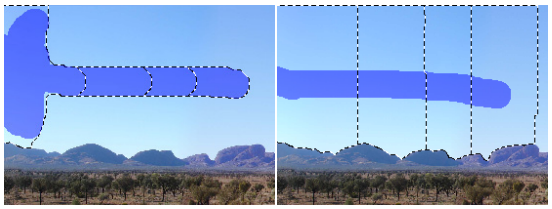
In each triggered optimization, we first estimate the foreground color information. We denote the intersection between the brush B and the background U as *seed pixels* S ($S = B \cap U$). To obtain a stable estimation, we compute a box R by dilating the bounding box of the region S by a certain width (typically 40 pixels). We denote the intersection between the dilated box R and the existing selection F as *local foreground pixels* L ($L = R \cap F$), as shown in Figure 2(b). Using both seed pixels and local foreground pixels, we build a local foreground color model $p^f(\cdot)$ by fitting a Gaussian Mixture Model (GMM) [Rother et al. 2004] with four components. Using local foreground pixels makes the estimation more stable because the brush or seed pixel region may be very small.

Then, we update the background color model. At the very beginning of the user interaction, a background color model $p^b(\cdot)$ (a GMM with eight components) is initialized by randomly sampling a number (typically 1,200) of pixels from the background. In each subsequent user interaction, we replace the samples that were labeled as foreground in the previous interaction with the same number of pixels randomly sampled from the background. The background GMM is re-estimated using the updated samples.

With the two color models, we apply a multilevel graph-cut-based optimization, described in Section 3, to obtain an expanded new selection. The data term $E_d(x_p)$ of Equation (5) in the Appendix is:

$$\begin{cases} E_d(x_p) = (1 - x_p) \cdot K & \forall p \in S \\ E_d(x_p) = x_p \cdot K & \forall p \in S^B \\ E_d(x_p) = x_p \cdot L_p^f + (1 - x_p) \cdot L_p^b & \forall p \in U \setminus (S \cup S^B) \end{cases} \quad (1)$$

where K is a sufficiently large constant, $L_p^f = -\ln p^f(I_p)$ and $L_p^b = -\ln p^b(I_p)$, S^B are "hard" background scribbles (scribbles drawn when users expand the background), and I_p is the image color at pixel p.



(a) without adding frontal pixels (b) with adding frontal pixels

Figure 3: *Adding frontal foreground pixels makes selection expansion faster in smooth regions.*

Figure 4: *Left: the selection of the rear-wheels is created by a lasso tool since there is no edge information to be used. Middle: when the user intends to expand the selection upwards, the fluctuation effect appears (undesired expansion at the rear of the vehicle). Right: with fluctuation removal, the existing selection far from the brush can be preserved.*

Progressive selection is more efficient for three reasons. First, only background pixels participate in the optimization. Second, the data term we constructed in Equation (1) is less ambiguous in most areas, since our foreground color model is compact. It is not related to other foreground scribbles outside of the dilated box R, which usually makes the optimization problem easier. Finally and most importantly, the boundary of the expanded new selection in each user interaction is usually a small fraction of the whole object boundary, so the multilevel optimization described in Section 3 runs very quickly.

Next, we solve two complications with our progressive algorithm: slow propagation in smooth regions (Figure 3(a)), and fluctuation effects (Figure 4).

2.2.2 Adding frontal foreground pixels.

Propagation of the selection is slow in smooth regions due to the "shrinking bias" [Boykov and Jolly 2001] – a bias towards shorter boundaries because the contrast term in the graph-cut is the length of the boundary modulated with the inverse of image contrast. In smooth regions, the boundary of the new selection tends to snap to the existing selection.

We mitigate the shrinking bias by adding a number of *frontal foreground pixels* as hard constraints. Frontal foreground pixels ∂F are interior boundary pixels of the existing selection, as shown in Figure 2(b). Accordingly, we change the first row in Equation (1) to:

$$E_d(x_p) = (1 - x_p) \cdot K \qquad \forall p \in S \cup \partial F. \tag{2}$$

Using ∂F as hard constraints, the selection boundary can be more effectively expanded and the resulting propagation is faster in smooth regions, as shown in Figure 3(b).

2.2.3 Fluctuation removal

The fluctuation effect is: when users intend to change the selection locally, some parts of the selection far from the region of interest may also change. The effect is distracting and may conflict with users' intentions. This is mainly caused by unavoidable color ambiguities.

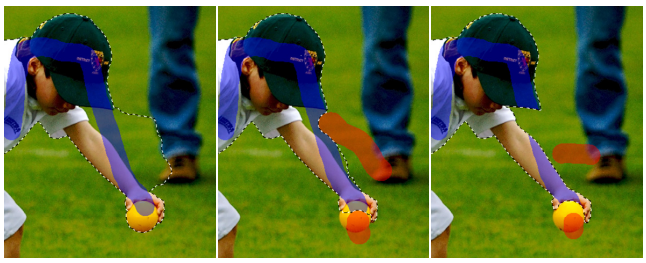


Figure 5: *Left: part of foreground scribble (blue) is drawn on the background. Middle: conflicting scribble prevents the background selection and the deselection of the ball. Right: scribble competition segments the conflicting scribble based on the image content.*

We eliminate the fluctuation effect by assuming that users only want to make a new selection adjacent to the brush. After the progressive labeling, the new selection may consist of several disconnected regions. We reject those regions that are not connected to the seed pixels. In other words, we only allow the local change. Figure 4 shows a comparison with/without fluctuation removal.

Interchangeability. Since progressive labeling prohibits existing foreground regions from being turned to background, and fluctuation removal only keeps local, user-intended new foreground, using them together enables a nice property of our tool – *interchangeability*: users can combine our tool with any other selection tool (e.g., marquee, lasso) in any order to complete a selection task. In Figure 4, for example, the selection is jointly created by a lasso tool and our tool.

Interchangeability gives users more flexibility to maximize their productivity. Without interchangeability, users can only apply other selection tools and Boolean operations after closing our tool.

2.2.4 Scribble competition

There are two situations in which users want to override previous scribbles: deselecting an unwanted object or a part of the object, and mistakenly dragging the mouse into the background as the scribbles are invisible. With previous tools, users have to paint over all conflicting scribbles manually because they are hard constraints. This process is tedious and even confuses a novice user, as shown in Figure 5.

We address this issue with a scribble competition method. Suppose C is a scribble which conflicts with a new scribble. We segment the conflicting scribble using graph-cut-based segmentation:

$$\begin{cases} E_d(x_p) = (1 - x_p) \cdot K & \forall p \in S \\ E_d(x_p) = x_p \cdot L_p^f + (1 - x_p) \cdot L_p^b & \forall p \in C \setminus S \end{cases} \tag{3}$$

where the color model $p^b(I_p)$ is estimated using all pixels within the scribble C. As shown in Figure 5, only a coherent part of the scribble is removed, based on both color and edge information. With scribble competition, users can effectively override conflicting scribbles and freely select/deselect objects.

3 Optimization

Our optimization uses the multilevel banded graph-cut [Lombaert et al. 2005], which first computes a coarse result on a low resolution image (grid graph), then generates a narrow band (usually ± 2 pixels) and upsamples the narrow band, and finally computes the result within the upsampled narrow band (band graph) in the high

resolution image. This banded optimization process is performed in multiple levels.

In this section, we introduce two techniques to improve the multilevel banded optimization: 1) multi-core graph-cut for both grid graphs and band graphs; 2) adaptive band upsampling for effectively reducing the size of band graphs.

3.1 Multi-core graph-cut

In general, parallelization can improve performance. However, Boykov's sequential algorithm based on augmenting path with "tree-reuse" [Boykov and Kolmogorov 2001] is still the fastest for typical 2D graphs used in vision and graphics, even compared with the leading parallel push-relabel algorithm [Delong and Boykov 2008] on a dual-core or quad-core machine. Unfortunately, it is hard to parallelize Boykov's path augmentation without introducing expensive synchronization. Recently, Vineet et al. [2008] introduced a GPU-based push-relabel algorithm that runs on a subset of NVIDIA graphics cards and outperforms most CPU-based graph-cut algorithms; in contrast, our goal is to design a general algorithm that exploits the power of modern multi-core processors.

We propose a parallel version of Boykov's algorithm using an alternative graph partitioning method. Boykov's algorithm performs a breadth-first search over the graph to find paths from the source to the sink using two dynamic trees. If an augmenting path is found, the capacities of all edges along the path are decremented appropriately. Note that it is not necessary to find an optimal path in every search. Any path found can make progress on the optimization.

Based on this fact, taking a dual core case as an example, we first partition the graph into two disjoint subgraphs and find augmenting paths in both subgraphs concurrently. Because no crossing path between the two subgraphs can be found, the result may not be optimal. Once we cannot find an augmenting path in one of the subgraphs, we partition the whole graph into two different disjoint subgraphs and continuously search for augmenting paths in parallel. The new partition will give a chance to find paths that cannot be found in the previous partition. The dynamic trees in the two subgraphs can be reused after a simple "orphan adoption" [Boykov and Kolmogorov 2001]. We alternatively partition the graph and perform path finding. Until we cannot find any augmenting path in two successive iterations (usually after 6–10 iterations), we perform a sequential path finding on the whole graph to guarantee the optimality. Since most flows have been sent from the source to the sink in the parallel iterations, the final sequential path finding takes only a very small fraction of the execution time (3–5%). The iteration process is illustrated in Figure 6.

The graph partition impacts the parallelism performance. We found that alternatively dividing the graph horizontally and vertically works well on the grid and band graphs we used. To allocate balanced workloads, we dynamically determine the dividing line by equally bi-partitioning "active nodes" [Boykov and Kolmogorov 2001], which can be used as a rough workload estimation.

To apply the above algorithm on a quad-core or eight-core processor, we simply recursively use the dual-core algorithm. Our graph partitioning method is extremely suitable for grid and band graphs. On the band graph, the speedup ratio is nearly 2.0 on a dual-core processor. On average, applying multi-core graph-cut in all levels can reduce 35–45% total runtime on a dual-core processor and 55–65% total runtime on a quad-core processor.

3.2 Adaptive band upsampling

As we introduced before, the banded graph-cut [Lombaert et al. 2005] computes the result in a fixed width narrow band in each level. If the band width is too small, full details may not be discovered; otherwise, the computational cost increases. We use Joint Bilateral Upsampling (JBU) [Kopf et al. 2007] to create an adaptive band in each level.

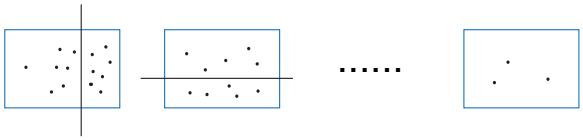


Figure 6: *Alternative graph partition. The dividing line (grey) is dynamically set in each iteration based on active nodes (black dots).*

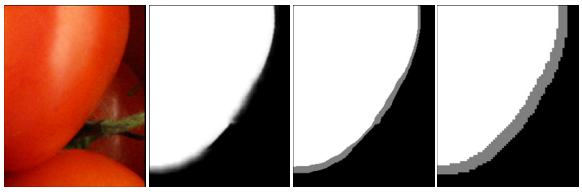


Figure 7: *Adaptive band upsampling. From left to right: input image, upsampled solution, adaptive band, fixed width band.*

Given the binary result in one level, we first create a narrow band by dilating (± 2 pixels) the boundary. Then, we upsample the narrow band to the upper level using JBU. For each pixel p in the upper narrow band, its upsampled value x_p is:

$$x_p = \frac{1}{k_p} \sum_{q_\downarrow \in \Omega} x_{q_\downarrow} f(||p_\downarrow - q_\downarrow||) g(||I_p - I_q||) \qquad (4)$$

where $f(\cdot)$ and $g(\cdot)$ are spatial and range Gaussian kernels, $p_\downarrow$ and $q_\downarrow$ are coarse level coordinates, $\{x_{q_\downarrow}\}$ is the coarse result, Ω is a 5×5 spatial support centered at $p_\downarrow$, and k_p is a normalization factor.

In JBU, the upsampled solution is converted to a binary result by thresholding. We found that the upsampled value itself is a good approximation of the alpha matte, as shown in Figure 7. We exploit this useful information to generate an adaptive narrow band: we directly label a pixel as foreground or background if its upsampled value x_p is out of the range $[0.25, 0.75]$. The resulting band is narrow around the sharp edges and wide in low contrast regions, as compared in Figure 7. Adaptive band upsampling can effectively reduce the size of graph without sacrificing image details.

It is important to note that combining progressive selection and adaptive band upsampling can substantially reduce the number of pixels we need to consider. The optimization is mainly performed in a very narrow band of the new selection, whose length is usually much shorter than the whole object boundary. To get a seamless connection, we add those frontal foreground pixels neighbored to the narrow band as foreground hard constraints.

In the multilevel optimization, we need to determine the size of the coarsest image as well as the number of levels. For speed considerations, in our default setting, the coarsest image is obtained by downsampling (keeping the aspect ratio) the input image so that $\sqrt{w \times h} = 400$, where w and h are the width and height of the coarsest image. Then, the number of levels is automatically set so that the downsampling ratio between two successive levels is about 3.0. For example, the number of levels will be four for a 20Mp image.

3.3 Viewport-based local selection

The multilevel banded optimization usually produces nearly the same segmentation on the full resolution as the conventional graph-cut [Lombaert et al. 2005]. However, if the downsampling ratio

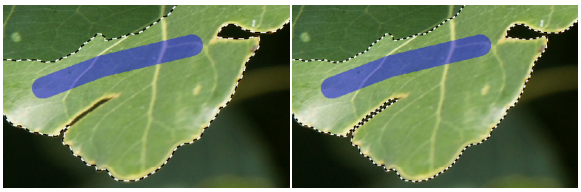


Figure 8: *Zoomed-in views of selections on a 24.5Mp image. Left: global selection. Right: viewport-based local selection.*

of the input image to the coarsest image is too large, the segmentation accuracy in the coarsest image decreases, which may make selection difficult for thin structures.

We observed that users often zoom in when they are working on a small object or region. Based on this observation, we introduce a viewport-based local selection in a dynamic local window around the area that users are focusing on.

Dynamic local window. We calculate a dynamic local window based on the current zoom ratio (displayed image size over the actual image size). In the image coordinates, we first construct a brush window that is centered at users' brush. The extent of the brush window is equal to the viewport size. Then, we define the *dynamic local window* as the minimal area window containing both brush window and screen region.

We perform the multilevel optimization in the dynamic local window. The downsampling ratio of the coarsest image is decreased because the local window size is usually smaller than the input image. The resulting benefit is demonstrated in Figure 8.

In the local selection, we should prevent the selection from partially aligning with the rectangular boundaries of the local window, because it seems like an artifact from users' point of view. We do this by adding background pixels that are adjacent (outside) to the local window as background hard constraints.

Because local selection is more useful when the zoom ratio is large, we automatically switch to viewport-based local selection if the zoom ratio is larger than 100%. For a typical $1K \times 1K$ screen, viewport-based local selection guarantees instant feedback independent of the image size.

4 Results

Performance. We compared Paint Selection with our implementation of Lazy Snapping using a banded graph-cut. (We found that the multilevel optimization is more efficient and accurate than the watershed-based optimization used in Lazy Snapping [Li et al. 2004].) The two tools are the same, except our tool uses progressive selection and the two proposed optimization techniques. Both tools were used to select the same objects on a dual-core 2.8G machine with 2G memory.

Figure 9 shows a detailed comparison for a 20Mp image. On average, Paint Selection is 15 times faster. The maximum response time of Paint Selection is less than 0.13s, while Lazy Snapping fails to provide instant feedback. Although the number of triggered optimizations in Paint Selection is larger, the total optimization time of Paint Selection is shorter (3.05s vs. 9.56s).

Figure 10 compares the average response time and maximum response time among four systems. As can be seen, all techniques we proposed play important roles: progressive selection significantly reduces the average response time, and multi-core graph-cut and adaptive band upsampling are especially helpful for reducing the maximum response time.

In theory, we can expect a sub-linear growth of the response time

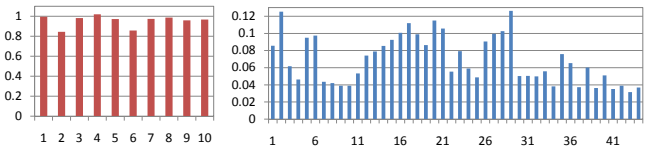


Figure 9: *Performance comparisons for a 20Mp image. X-axis denotes the number of user interactions and Y-axis denotes the system response time (in seconds). In Lazy Snapping, the response time is measured from mouse button release to result display; in Paint Selection, the response time is measured from touching the background pixels to result display.*

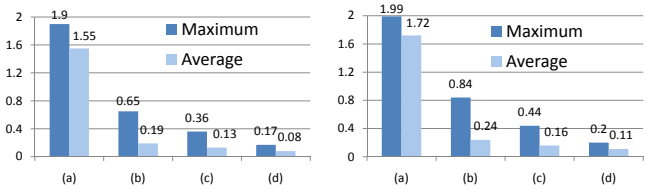


Figure 10: *Performance comparisons for a 30Mp image (left) and a 40Mp image (right). (a) Lazy Snapping. (b) Paint Selection using progressive selection only. (c) progressive selection + dual-core graph-cut. (d) progressive selection + dual-core graph-cut + adaptive band upsampling. All methods use banded graph-cut.*

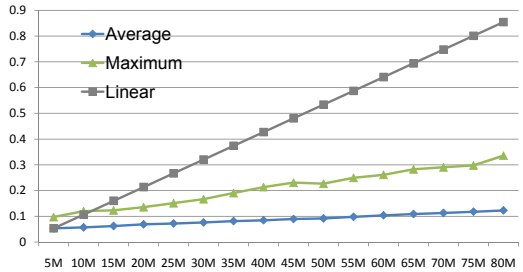


Figure 11: *Scalability. "Linear" represents a system in which the increasing rate of the response time is equal to that of the number of pixels.*

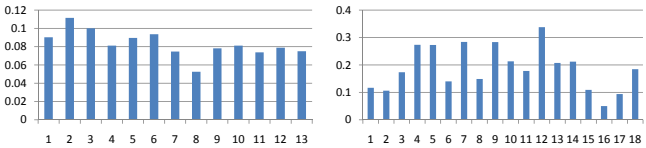


Figure 12: *Response time (in seconds) on a 110Mp panorama. Left: selecting a small vehicle and building. Right: selecting a large sky.*

with respect to the number of pixels because the complexity of the banded graph-cut is proportional to the length of the object boundary. To verify this, we first made selections on a 20Mp image using Paint Selection and resized the image and scribbles to multiple versions of varying sizes. Then, we replayed the scribbles and recorded the average and maximum response time. As shown in Figure 11, the increasing rate of the average response time is roughly $\sqrt{M}$, where M is the increasing rate of the number of pixels.

Figure 12 shows the performance of Paint Selection on a 110Mp panorama. If we select a very large region, e.g., the sky, the average response time is about 0.2s; if we select a small or textured object, the average response time is less than 0.1s. Note that the memory consumption of the multilevel optimization is small. In this example, the peak memory usage in optimization is about 30MB. The memory consumption is due mostly to keeping the input image in the memory.

Figure 13: *Selection results during mouse dragging on a 12.7Mp image (top row) and a 24.5Mp image (bottom row). Left: input image. Middle: Photoshop Quick Selection result. Right: our result (viewport-based local selection turned off).*

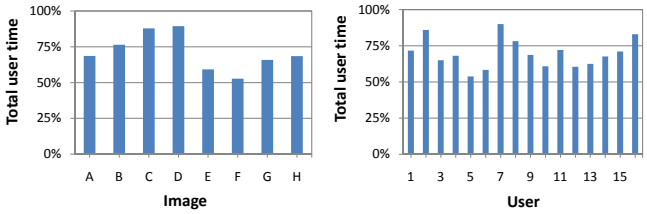


Figure 14: *User time comparison between Paint Selection and Lazy Snapping (100%) across eight images and sixteen users.*

In the performance comparisons above, we turn off the viewport-based local selection. All images, scribbles, and selection results in the above experiments are included in the supplementary material.

Comparison with Photoshop Quick Selection. Adobe Photoshop CS4 Quick Selection is the commercial tool most similar to ours, although its algorithm has not been made public. Compared with Photoshop Quick Selection, Paint Selection has three advantages: 1) more accurate results during mouse dragging; 2) greater speed for large images (e.g., larger than 100Mp); 3) no extra refinement after mouse release.

As compared in Figure 13, Photoshop Quick Selection (in CS4) displays a coarse and jagged boundary during mouse dragging, while Paint Selection produces a more accurate boundary. Photoshop Quick Selection often takes additional time to refine the result after users release the mouse button. This "preview-refine" UI may give rise to a UI issue: users have to frequently stop the selection to inspect the refined result since they cannot predict it from a coarse one, especially when making precise selections. Furthermore, computing the coarse result only also prevents applying local image effects or adjustments during mouse dragging.

As demonstrated in the accompanying video, Photoshop Quick Selection (in CS4) fails to instantly provide even coarse feedback on a 110Mp panorama, and its refinement step may take up to several seconds. In contrast, Paint Selection still provides quick responses and accurate results.

Usability study. To compare a scribble-based UI (Lazy Snapping) and painting-based UI (Paint Selection), we conducted a usability study. To isolate the speed issue, we downsampled all test images to 800×600 so that Lazy Snapping can instantly output a result.

Figure 15: *Instant image editing on a 10Mp image. Left and middle: The user adjusts color balance at an intermediate selection. Here we use simple feathering within a 4-pixel-width band. Right: As the user continues the selection, the desired effect is instantly applied to the newly selected region. The user is often satisfied with the result even if the actual selection contains minor errors.*

We invited sixteen volunteers and gave them short training on both tools. The users were allowed to practice until they felt competent with both tools. Eight images of various complexities were used in the testing phase. In each image, a pre-defined object was asked to be selected. We recorded the interaction time and collected subjective feedback from each user.

On average, Paint Selection required 30% less user time compared with Lazy Snapping, as shown in Figure 14. We concluded several reasons from our conversations with the users: Paint Selection is more intuitive and simple. Some users said they were happy to be spared from drawing scribbles on the background. Paint Selection is also more direct. Users can quickly move the brush to the region that needs hints since Paint Selection provides instant feedback during mouse dragging. Finally, Paint Selection does not introduce any annoying fluctuation effects.

Instant image effects. Paint Selection is fast enough for directly applying image effects or adjustments during mouse dragging. For instance, users can pause at any intermediate selection, choose an effect, then continue to instantly "paint" that effect over the rest of the object. The advantage over editing after complete selection is that users do not have to pursue a perfect selection, as long as they are satisfied with the adjusted result.

Using the obtained selection, users can either apply the effect on the fly, as shown in Figure 15, or blend a pre-computed image containing the desired effect with the original image (shown in the accompanying video). For a seamless composition, we can perform simple feathering within a narrow band around the object, which in many cases produces few noticeable artifacts since the binary selection tightly snaps to the object boundary. We can also use the upsampled solution in the finest level of the optimization as a soft selection.

Limitations. Like other graph-cut-based selection tools, Paint Selections suffers several drawbacks due to the nature of the graph-cut optimization. One problem is that foreground expansion is impeded in highly-textured regions. The optimization-guided contour tends to snap to strong edges, demanding extensive user interaction, as shown in Figure 16. Another problem is that around low contrast edges, users may have to specify both foreground and background scribbles to constrain the boundary. Finally, for objects with complex topologies (e.g., tree branches with small holes), obtaining a fine selection may involve too much background scribbling, which can be regarded as a typical failure case for many scribble/painting-based selection tools.

These drawbacks can be alleviated to a large extent by utilizing the

Figure 16: *Foreground expansion is impeded by strong edges in the highly-textured region.*

interchangeability feature of our tool. For highly-textured objects, users can first roughly encircle the whole object with a lasso and then perform a background expansion, making the contour quickly snap to the object boundary. For low contrast edges, users may use a simple brush to override ambiguous pixels, or a lasso to directly modify the selection contour. For objects with complex topologies, users can choose a color-based tool such as AppProp[An and Pellacini 2008] or Magic Wand[Adobe Photoshop] to accomplish the selection task.

5 Conclusions

We have presented a progressive painting-based user interface and algorithm for local selection in images. Our system exploits the progressive nature of interactive selection to provide instant feedback so that users are able to quickly and effectively make a high quality selection on multi-megapixel images. In the future, we plan to apply this methodology to other image, video, and 3D volume editing tasks.

Paint Selection does not support simultaneously selecting two or more connected regions, which may be required for local image adjustment applications. Applying region competition could be a possible solution. For high quality cut-and-paste applications, we still need a matting operation [Wang et al. 2007] after the selection. Integrating progressive selection and matting may be useful for realtime composition.

Acknowledgements We would like to thank anonymous reviewers for their constructive comments and suggestions. Many thanks to Yin Li for helping us to refine the code and paper, and to Bennett Wilburn and Matthew Callcut for proofreading, Bennett also kindly narrated the video.

Appendix – Graph-cut-based segmentation

The binary labels $X = \{x_p\}$ of the image are obtained by minimizing an energy $E(x)$ [Boykov and Jolly 2001]:

$$E(X) = \sum\nolimits_p \; E_d(x_p) + \lambda \sum\nolimits_{p,q} \; E_c(x_p, x_q) \qquad (5)$$

where λ is the weight (set to 60 in all experiments), $E_d(x_p)$ is the data term, encoding the cost when the label of pixel p is x_p (1 - foreground, 0 - background), and $E_c(x_p, x_q)$ is the contrast term, denoting the labeling cost of two adjacent pixels p and q. We use the following contrast term: $E_c(x_p, x_q) = |x_p - x_q| \cdot (\beta \cdot ||I_p - I_q|| + \epsilon)^{-1}$ where $\epsilon = 0.05$ and $\beta = \left(\langle \|I_p - I_q\|^2 \rangle\right)^{-1}$ [Blake et al. 2004]. Here $\langle \cdot \rangle$ is the expectation operator over the whole image.

References

ADOBE PHOTOSHOP. http://www.adobe.com/support/photoshop/.

AN, X., AND PELLACINI, F. 2008. Appprop: all-pairs appearance-space edit propagation. *ACM Trans. Graph. 27*, 3, 1–9.

BAI, X., AND SAPIRO, G. 2007. A geodesic framework for fast interactive image and video segmentation and matting. In *Proceedings of ICCV*, 1–8.

BLAKE, A., ROTHER, C., BROWN, M., PEREZ, P., AND TORR, P. 2004. Interactive image segmentation using an adaptive gmmrf model. In *Proceedings of ECCV*.

BOYKOV, Y., AND JOLLY, M. P. 2001. Interactive graph cuts for optimal boundary & region segmentation of objects in n-d images. In *Proceedings of ICCV*, 105–112.

BOYKOV, Y., AND KOLMOGOROV, V. 2001. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. In *Energy Minimization Methods in CVPR*.

CHEN, J., PARIS, S., AND DURAND, F. 2007. Real-time edge-aware image processing with the bilateral grid. *ACM Trans. Graph. 26*, 3, 103.

DELONG, A., AND BOYKOV, Y. 2008. A scalable graph-cut algorithm for n-d grids. In *Proceedings of CVPR*, 1–8.

GRADY, L. 2006. Random walks for image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell. 28*, 11, 1768–1783.

KASS, M., WITKIN, A., AND TERZOPOULOS, D. 1987. Snakes: Active contour models. *IJCV 1*, 4, 321–331.

KOPF, J., COHEN, M. F., LISCHINSKI, D., AND UYTTENDAELE, M. 2007. Joint bilateral upsampling. *ACM Trans. Graph. 26*, 3, 96.

LEVIN, A., LISCHINSKI, D., AND WEISS, Y. 2008. A closed-form solution to natural image matting. *IEEE Trans. Pattern Anal. Mach. Intell. 30*, 2, 228–242.

LI, Y., SUN, J., TANG, C. K., AND SHUM, H. Y. 2004. Lazy snapping. *ACM Trans. Graph. 24*, 3, 303–308.

LI, Y., ADELSON, E. H., AND AGARWALA, A. 2008. Scribbleboost: Adding classification to edge-aware interpolation of local image and video adjustments. In *EGSR*.

LISCHINSKI, D., FARBMAN, Z., UYTTENDAELE, M., AND SZELISKI, R. 2006. Interactive local adjustment of tonal values. *ACM Trans. Graph. 25*, 3, 646–653.

LOMBAERT, H., SUN, Y., GRADY, L., AND XU, C. 2005. A multilevel banded graph cuts method for fast image segmentation. In *ICCV 2005*, 259–265.

MORTENSEN, E. N., AND BARRETT, W. A. 1995. Intelligent scissors for image composition. In *Proceedings of ACM SIGGRAPH*.

OLSEN, JR., D. R., AND HARRIS, M. K. 2008. Edge-respecting brushes. In *UIST*, 171–180.

REESE, L. J. 1999. Intelligent paint: Region-based interactive image segmentation. In *Masters Thesis, Department of CS, Brigham Young University, Provo, UT*.

ROTHER, C., BLAKE, A., AND KOLMOGOROV, V. 2004. Grabcut - interactive foreground extraction using iterated graph cuts. *ACM Trans. Graph. 24*, 3, 309–314.

VINEET, V., AND NARAYANAN, P. 2008. Cuda cuts: Fast graph cuts on the gpu. In *Proceedings of CVPR Workshops*.

WANG, J., AND COHEN, M. F. 2005. An iterative optimization approach for unified image segmentation and matting. In *Proceedings of ICCV*, 936–943.

WANG, J., AGRAWALA, M., AND COHEN, M. F. 2007. Soft scissors: an interactive tool for realtime high quality matting. *ACM Trans. Graph. 27*, 3, 9.