Real-Time Segmentation, Tracking, and Coloring of Walls Using iOS

David Liu
Dept. of Electrical Engineering
Stanford University
davliu@stanford.edu

Jeff Piersol
Dept. of Electrical Engineering
Stanford University
jpiersol@stanford.edu

Serena Yeung
Dept. of Electrical Engineering
Stanford University
syyeung@stanford.edu

***Abstract*— In this work we implement an image processing algorithm on the iOS platform that transforms user-selected segments of a wall to different colors in real-time. Specifically, we demonstrate an algorithm that segments, tracks, and realistically modifies the color of a wall segment at video frame rates. Significant challenges were encountered in speeding up the processing time per frame to acceptable levels, and it was necessary to utilize GPU acceleration for several stages of the pipeline in order to achieve the desired performance. The results show a successful prototype of real-time wall color transformation, as well as opportunities for further work to increase the speed and accuracy of the segmentation.**

I. INTRODUCTION

Wall painting is a common home-remodeling project that typically involves choosing a paint color from hundreds of small paint cards. It can be difficult to envision what the final result will look like, and to choose between so many subtly different colors. The expensive cost of painting a wall also prevents trial-and-error, and increases the pressure to select the right color on the first try. A mobile app that can show the user what a wall will look like after painting, especially with a natural real-time experience, can be a valuable tool to help people choose the right color to paint their walls.

In this work we demonstrate a mobile app developed on the iOS platform that is suitable for such an application. Our app taps into the live camera stream from an iOS device to perform three main tasks: (1) it segments out a wall region based on a user-defined seed point; (2) it tracks the wall region as the camera is moved around; and (3) it transforms the color of the wall to a user-selected color. All of this is done at video frame rates so that as a user looks through an iOS device's camera, the user sees the wall as if it had been painted a different color.

A. Prior and Related Work

1) Segmentation: Color and edge-based segmentation is most appropriate for walls due to their typically large, flat, and single-colored surfaces. There are many established ways to perform edge detection, including the Canny, Sobel, Harris, SUSAN, and Laplacian of Gaussian edge detectors. These detectors all involve convolution with a specified kernel followed by post-processing operations such as non-maximum suppression. Alternatively, morphological operations such as erosion and dilation have also been used for edge detection [1].

2) Tracking: As with edge detection, there are various established methods for tracking, such as estimation using the wavelet transform, energy minimization via graph cuts and loxel-based visual feature organization [2] [3] [4] [5], with no universally optimal algorithm. Some methods are better suited for certain environments, and there is always a trade-off between accuracy and speed. For example, the block matching algorithm can achieve higher accuracy through exhaustive search versus local search with heuristics, but this causes it to become more computationally expensive as well [6]. Besides block matching, other well-known tracking methods include using a variety of feature descriptors, which look for one-to-one correspondences between a reference image and a target image [7] [8], and video coding algorithms, which try to find the motion vectors of each pixel based on the encoding codec [4]. All of the above methods perform tracking by determining an approximate transformation from the reference image to the target image through warping. When considering the use of tracking for wall painting in a mobile app, the highest priority is speed and simplicity of implementation.

3) Color Transformation. It is difficult to estimate the precise appearance of a specific paint color with known reflectance values to a wall, given the unknown lighting conditions of a mobile camera. Discussions with Dr. Joyce Farrell have yielded ideas for potentially strong approximations, such as attempting to first estimate the lighting adjustments of the camera, but this was determined to be outside the scope of the current project. Commercially available mobile apps that perform wall painting (all on static wall images) appear to use a semi-transparent overlay to display the paint color, but this does not work well on dark surfaces.

II. ALGORITHM

Numerous algorithms from literature were simulated in MATLAB to determine the optimal algorithm in terms of consistently accurate results and run-time performance. Algorithm prototyping was performed in close conjunction with iOS development in order to accurately characterize the speed of the algorithm on an iOS device; additionally, OpenCV was compiled as a MATLAB library to ensure accurate simulation of image processing on the mobile device. Appendix C details the other algorithms attempted prior to arriving at the final algorithm pipeline shown in Fig 1.

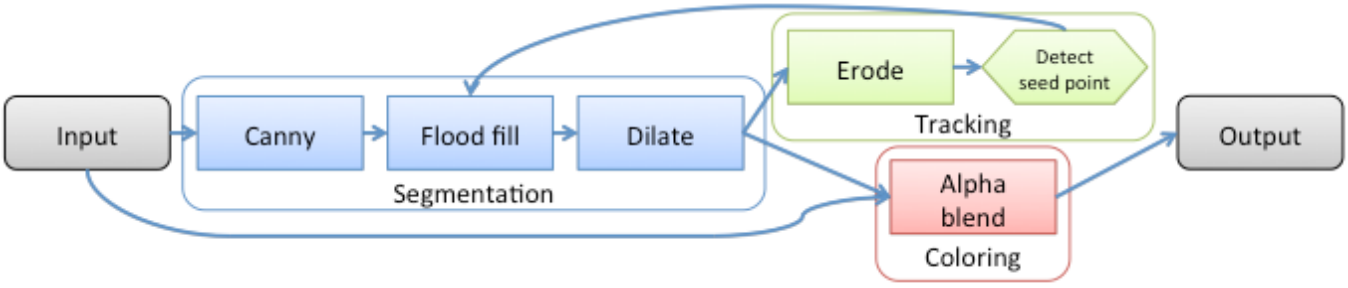

Figure 1. Image processing pipeline.

A. *Segmentation*

The first step in the image processing pipeline is Canny edge detection. Canny detection is performed on a grayscale version of the image, which was found using the formula

$$Y = 0.299R + 0.587G + 0.114B. \quad (1)$$

The threshold values used in the call to the OpenCV Canny detector were 60 and 25, which correspond to the thresholds for initial strong edge detection and the threshold for linking weak edges, respectively. A Sobel kernel of size 3 was used for computation of the gradient.

The second step in the pipeline is flood filling. The flood fill operation fills a binary mask by forming a connected component starting at a seed point, where connectivity is determined by differences in the color values between a pixel in the component and its neighbors. The threshold used is 5.0 for both upper and lower thresholds, for all three channels. If the user selected a new seed point, that pixel is used as the seed point for the flood fill; otherwise, the seed point is selected by the tracking algorithm. The result of Canny detection is passed into the flood fill operation as an input mask, so that flood filling does not cross over the Canny-detected edges. An illustration detailing the flood fill operation is shown in Fig. 2. The OpenCV implementation of the flood fill operation outputs the logical OR of the original mask and the filled in regions, so the edges are subtracted out to yield a mask selecting only the wall. The mask is then colored with the user-specified target color.

The third operation is dilation, which is the final step for segmentation. Dilation is performed with a 3x3 square structuring element and fills holes in the mask. After dilation, segmentation of the wall is complete.

B. *Tracking*

The tracking component is used to determine where the seed point should be in the next frame. To implement wall tracking, effort was made to design the simplest algorithm that would work given the real-time constraints. An erosion filter was applied to the output of the wall segmentation procedure, with a 30x30 structuring element with ones appearing every 5 rows or columns (in MATLAB notation, `SE = zeros(30); SE(1:5:end,1:5:end) = 1;`). The eroded mask then only contains points within the wall that are far away from the wall edges, so that any point in the eroded mask is likely to be a point in the wall in the next frame. Therefore, since any point in the eroded mask is likely to belong to the wall in the next frame, the first point found in the mask is used as the seed point for the next frame of input. An important assumption used by the tracking algorithm is that the user will not move the device abruptly; if the device is moved quickly, the eroded mask from the previous frame may no longer contain the wall, so the tracking component will not accurately find the wall in the current frame.

C. *Color Transformation*

The final step in the image processing pipeline is alpha blending, which effectively outputs a linear combination of the original image and the colored mask in the mask region and reproduces the original image outside of the mask region. The weighting used was 0.4 * original image + 0.6 * mask overlay. The output of alpha blending is then rendered to the display.

A video illustrating the algorithm can be found at [9]. Note that the video shows the algorithm as prototyped in MATLAB, not as implemented on iOS, and the video algorithm deviates slightly from the final implemented algorithm.

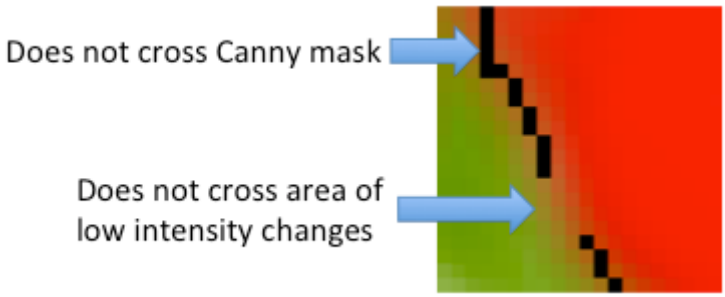

Figure 2. The two modes of operation of flood fill.

III. Performance Optimization

Perhaps the most important constraint for image processing on mobile devices is speed. In the research phase of this project, nearly all image operations were found to have a considerable negative performance impact, in effect limiting the total number of operations that the program is able to use.

One means of reducing the performance impact is to use hardware that is capable of parallel operations on data, and iOS devices have GPUs for this purpose. The software interface to the GPU is provided in the form of OpenGL ES, which is a graphics rendering API for use on mobile devices [10]. Though OpenGL is primarily intended for rendering 3D graphics, its fragment shader allows for per pixel operations to be run in parallel on data, which has proven very useful for image processing operations. Operations that apply the same algorithm to every pixel benefit greatly from parallelized GPU processing, as evidenced in the "3x3 dilation filter" column of Table 1. In this project, both morphological operations were performed using the GPU, as well as the alpha blending operation, improving the frame rate from under 1 frame per second to around 5 frames per second on a 3rd generation iPad.

In Table 1, the display frame rate for applying two different operations is shown: one row showing a typical application of the operation, the second showing the operation when executed on the GPU, and the third row showing the frame rate without applying the specified operation. To gather this data, the program was run using the algorithm described in section 3, while modifying the implementation of either the dilation filter or the alpha blending filter by implementing the filter using OpenCV or by removing the operation completely.

One opportunity for future performance enhancements is to implement all of the image processing operations with OpenGL, both to take advantage of parallel operations and to reduce the cost of data exchange between the CPU and GPU. Table 2 shows the approximate duration of all of the image processing operations for the project, of which operations 1-7 are all performed on the CPU. If these operations were performed on a GPU and achieved a speedup of factor 2, the frame rate would benefit greatly.

The GPUImage framework was used in this project to simplify communication with the GPU [11]. This framework provided the alpha blending filter that was used at the end of the pipeline and also provided a useful template for creating erosion and dilation filters.

Table 1. Frame rate for different operators (in FPS).

	3x3 dilation filter	Alpha blending
OpenCV	2.37	3.2
OpenGL (GPU-accelerated)	3.5	3.5
Without operation	3.5	6.0

IV. Results

The algorithm described in the methods section was tested on both MATLAB and iOS. The results were evaluated in terms of segmentation, tracking, and coloring. In general, the implementation appears to perform reasonably well under most circumstances, and at speeds fast enough to be used in real-time, as discussed in the Performance Optimization section. There are specific cases where the implementation was noted to show limitations, and these will also be discussed below.

A. Segmentation

Fig. 3 shows several screenshots from test runs of the mobile app in different scenarios. Fig. 3a was taken in a dining room, and shows that the algorithm was able to accurately segment out the chair and table outlines, as well as the non-standard wall boundary in the kitchen area without leaking into the kitchen. Figs. 3b and 3c show further segmentation around picture frames, chairs and windows. Fig. 3d shows a more complex scene, where the algorithm was still able to successfully segment around a window, monitor, somewhat complicated bookshelf setup, and shadows. At substantially more complex scenes, such as the one shown in Fig. 4, the algorithm begins to struggle; however such a scene would be confusing even to a human eye to distinguish what exactly should constitute a wall segment.

One weakness of the algorithm is that it does not perform reproducibly at wall corners: it can flicker and sometimes exclude the ceiling (Fig. 3a) while sometimes not (Fig. 3d). This is due to the difficulty of picking out a corner among the shadows where three white walls intersect. Strong shadows can also unexpectedly cut short a wall segment (Fig 5). One potential fix is to use temporal coherence to create averaged masks using multiple frames, although this might come with the drawback of consistently non-sharp edges in the segmentation. Another potential fix is to impose stronger

Table 2. Duration of each processing step.

1. Create RGB image from RGBA image	10 ms
2. Converting color image to grayscale image for edge detection	10 ms
3. Canny edge detection	80 ms
4. Copying Canny results into RGB image for (7)	10 ms
5. Flood fill	5 ms
6. Copying target wall color to mask image	5 ms
7. Removing edges from mask	40 ms
8. Mask erosion for tracking	50 ms
9. Alpha blending	40 ms

Figure 3 a,b,c,d.

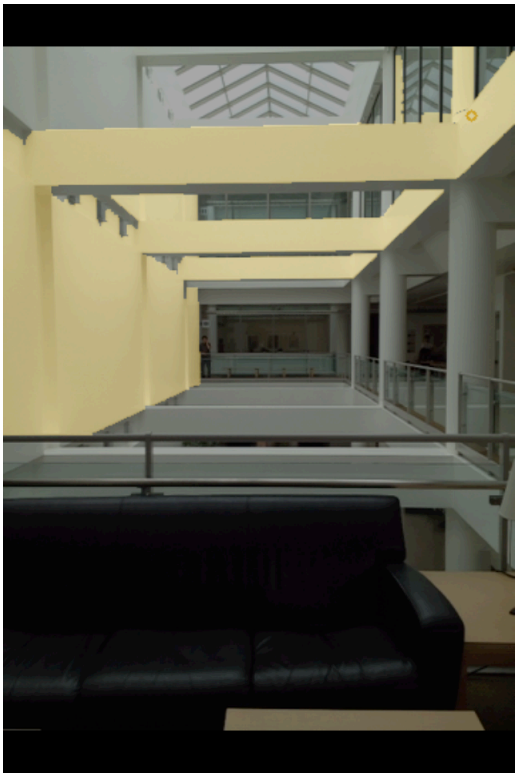

Figure 4.

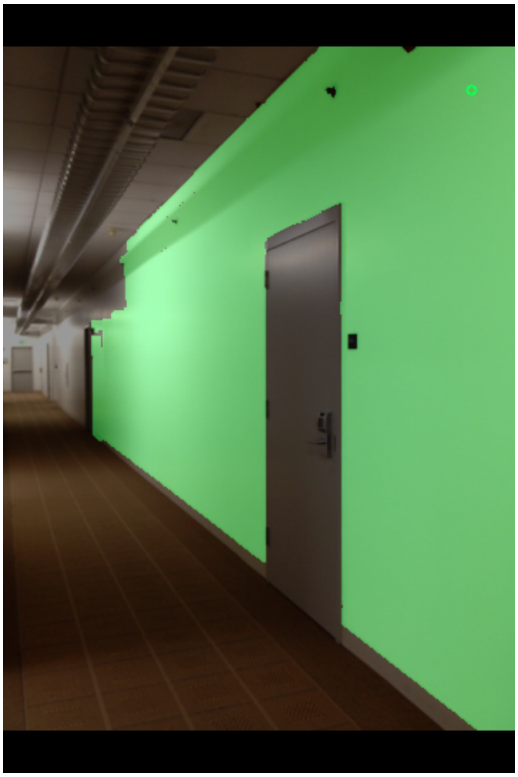

Figure 5.

thresholds on the edges so that only one plane of a wall will be segmented at a time, and then allow a user to manually select multiple seed points corresponding to the multiple wall planes the user wishes to be colored. This was considered too complicated of a user interface to implement for this prototype, so weaker thresholds were allowed to be able to color across multiple wall planes.

B. *Tracking*

The algorithm was able to track wall regions well under most circumstances; however, the seed point will occasionally jump from one wall surface to another (e.g. from side wall to ceiling) when the seed point gets too near a border, and this can cause some flickering. This is likely due to the same cause as was described in the segmentation section, that the wall edge borders are weak enough allow the seed point to jump when two regions become momentarily detected as one region. This should be addressed in the next iteration of the app.

A second means of improving tracking is by increasing the frame rate, as this increases the validity of the assumption that the wall will remain in approximately the same location from frame to frame. This is because reducing the time between frames will reduce the displacement of the wall between frames, so that a point in the eroded mask is more likely to be a part of the wall in the next frame. Hence, tracking would be improved.

C. *Color Transformation*

While this was not the focus of the work presented in this paper, the alpha-blending method used for color transformation of the segmented wall nevertheless proved to be pleasing and realistic, accurately depicting shadows. Fig. 6 shows examples of a wall segment under different color choices.

V. Conclusion

In this work we demonstrated a successful prototype of a mobile app developed for iOS that can segment, track, and color walls at video frame rates. Test runs showed the algorithm to be able to perform well under most circumstances, with robustness decreasing at scenes of multiple non-simple wall shapes or intersecting walls. Fine-tuning parameters in pipeline stages such as the Canny edge detection stage (to increase the strength of wall borders) can improve the performance, and we plan to address this in future work for the app. An improved user interface that allows a user to select multiple wall regions will also be useful once these stronger wall borders limit the colored area in multi-plane wall regions. The speed of the app can be increased even further by implementing more of the algorithmic pipeline on the GPU instead of the CPU. Finally, we can look into color transformation methods that estimate lighting conditions to present an even more accurate preview of a desired paint color.

References

[1] Lee, James, R Haralick, and Linda Shapiro. "Morphologic edge detection." *Robotics and Automation, IEEE Journal of* 3.2 (1987): 142-156.

[2] Magarey, Julian, and Nick Kingsbury. "Motion estimation using a complex-valued wavelet transform." *Signal Processing, IEEE Transactions on* 46.4 (1998): 1069-1084.

[3] Boykov, Yuri, Olga Veksler, and Ramin Zabih. "Fast approximate energy minimization via graph cuts." *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 23.11 (2001): 1222-1239.

[4] Takacs, Gabriel et al. "Outdoors augmented reality on mobile phone using loxel-based visual feature organization." *Proceeding of the 1st ACM international conference on Multimedia information retrieval* 30 Oct. 2008: 427-434.

[5] Friedman, Nir, and Stuart Russell. "Image segmentation in video sequences: A probabilistic approach." *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence* 1 Aug. 1997: 175-181.

[6] Zeng, Bing, Renxiang Li, and Ming L Liou. "Optimization of fast block motion estimation algorithms." *Circuits and Systems for Video Technology, IEEE Transactions on* 7.6 (1997): 833-844.

[7] Takacs, Gabriel et al. "Unified real-time tracking and recognition with rotation-invariant fast features." *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on* 13 Jun. 2010: 934-941.

[8] Klein, Georg, and David Murray. "Parallel tracking and mapping on a camera phone." *Mixed and Augmented Reality, 2009. ISMAR 2009. 8th IEEE International Symposium on* 19 Oct. 2009: 83-86.

[9] https://vimeo.com/43400549, password 'nana'

[10] http://www.opengl.org/wiki/FAQ

[11] https://github.com/BradLarson/GPUImage

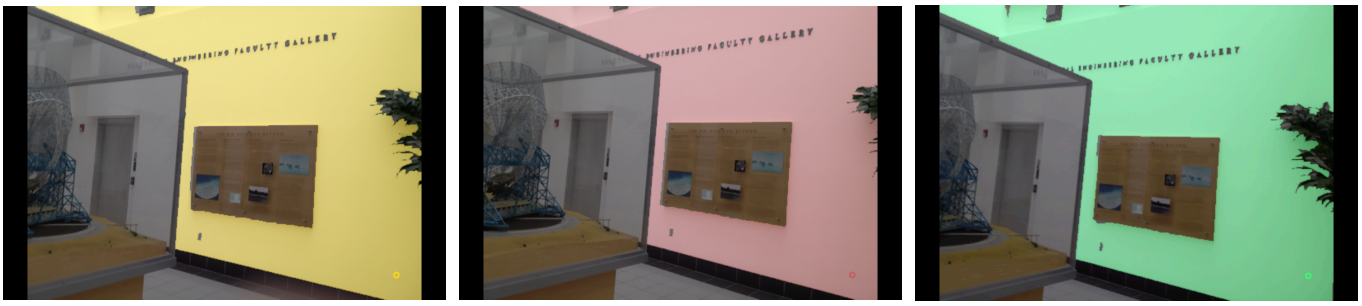

Figure 6. Wall under different applied colors.

APPENDIX A: CONTRIBUTIONS

This section describes the contributions of each individual of the team.

Serena Yeung

- iOS Developer - focused on CPU/GPU integration and color adjustment
- Took the RIFF tracker C++ code from the TA and debug/compiled it to run on PC. It was ultimately judged unnecessary for the app, which does not need feature recognition.
- Integrated GPUImage framework into the project and implemented communication mechanism between CPU and GPU
- Integrated OpenCV framework into the project
- Implemented the Color picker UI
- Report writing

Jeff Piersol

- iOS Developer - focused on real-time segmentation and tracking development
- Debugged memory leakage issues on iOS
- Modularize the Objective C code for readability and performance
- Implemented seed point tracking algorithm on iOS
- Report writing: algorithm and performance optimization

David Liu

- Algorithm design - focused on designing a working algorithm
- Contacted and interacted with TAs and Professor Girod for possible algorithm ideas to experiment
- Researched relevant IEEE papers
- Created the pipeline video in the poster
- Created the poster

APPENDIX B: ALGORITHMS CONSIDERED

The following algorithms are some key milestone algorithms that were attempted but were not included in the final pipeline of the project. Although these algorithms were not used, they aided in development of the implemented algorithm.

A. Initial Video Image Analysis

- Grayscale histogram
- Global Otsu thresholding
- Adaptive thresholding from seed point

These algorithms gave insight to characteristics of video frames. One key finding was that the camera does automatic brightness adjustment. Thus, it is important to not just rely only on the illuminance value of the image from one frame to next frame. This was evident through grayscale histogram video.

B. Feature based Tracking method

The feature-based method was exploited further to see if there exists a feature that can be computed in video frame rate at a 480 x 360 pixel resolution. The rotation-invariant fast feature detector promises to achieve such criteria [1]. Thus, experimental RIFF tracking C++ code was obtained from the RIFF author and testing is conducted to test the feasibility of the algorithm. However, it was not fast enough for iOS implementation.

C. Simple Tracking methods

Various feature based tracking methods were attempted: tracking by centroid, tracking the convex hull centers, and simple geometric quantities. However, the segments were not necessarily convex and it do not guarantee that next frame's seed point is in the same segment.

D. Edge detection:

- Laplacian of Gaussian edge detection
- Horizontal/vertical Sobel filter
- Canny edge detection

The above are well known edge detection algorithms from literature that were tested with various parameters. Canny edge detection has proven to be best result in this situation. Further enhancement was done through hit-miss filtering, then dilation to close the gaps.

- Hit-miss filtering:
    - Remove one pixel noise
    - Remove two pixel noise (vertical and horizontal)
    - Remove any pixel that is inside a 3x3 bounding box

These operations removed significant noise and closed some edge gaps. However, they are relatively computationally expensive and were not included in the final algorithm.

E. Hough Transform

Because there are missing gaps in the edge detection algorithm, Hough lines were found to attempt to accurately fill the gaps. However, the result was inaccurate and too computationally expensive.

F. Illumination and Color Adjustment Algorithm

The entire wall is assumed to have same spectral reflectance. Further, it is assumed that the image is lit with a single light source; the illuminance spectrum at each pixel can only be different by a scalar. Thus, using the measured RGB values, attempts were made to derive an accurate estimation of the illuminant spectrum. However, implementation of such estimation did not produce a useful result in MATLAB simulation. One major problem was that the camera applies automatic color balancing on the raw RGB values based on illuminance. Further work can be conducted in this area to refine the results.