COMP4801 FINAL YEAR PROJECT

Seamless Image Editing

Interim Report

Supervisor: Prof. Yu Yizhou

Group member: Cai Lingfeng

In our project, we will implement (1) The code base for drag and drop pasting method; (2) A new algorithm to search for the optimal boundary before blending. Our new algorithm searches for the boundary that has the least difference in original colors between the source image and the target image. It has better approximation of original image colors, and it provides a better boundary for blend-in.

Table of Contents

1. Project background

Image composition, which is the process of creating a new image by imposing an object or a region from a source image to a target image, is a large component of image editing. Poisson image editing [1] has been proposed in 2003 as an effective approach for seamless image composition. By solving Poisson equations using the user-specified boundary condition, Poisson image editing blends the colors from both image without visible discontinuities around the boundary.

The effectiveness of Poisson image editing, however, largely depends on how user draws the boundary. Drag and drop pasting [2] in 2006 proposed a method to optimize the boundary condition based on a shortest closed-path algorithm in {r,g,b} color space, and it improved the composition quality by searching for the optimal boundary within the user-specified one.

In our project, we will implement (1) The code base for drag and drop pasting method; (2) A new algorithm to search for the optimal boundary before blending.

Our new algorithm searches for the boundary that has the least difference in original colors between the source image and the target image. It has better approximation of original image colors, and it provides a better boundary for blend-in.

2. Project objective

Traditional Poisson image blending process [1] will first set the boundary colors equal to the target image in the new image created, and blend in the new image region according to the gradients of the source image. To obtain better seamless composition, we need to take the target image into consideration on the boundary conditions. The less original color difference along the region boundary between the source image and the target image, the better the results will be.

Our objective of the project is to search for the optimal boundary within the user-specified one (the region of interest), and outside the real object (the object of interest) user cares about. The cumulative original color space difference should achieve the global minimum along the new boundary. We focus on the difference of the image substance and boundary reflectance, and we want to remove the lightness and shading effects of the image since these effects will influence how the original image colors are presented.

In theory, the image composition will have a better result given the boundary based on our objective. Provided the boundary with less original color difference between the source image and the target image, the blended result will be closer to the original effect of the source image. Therefore, the blending will guide the new images in the right gradient direction of the real object user

cares about, instead of pushing the effects away from the colors the objects were meant to be originally.

Drag and drop pasting method [2] has done the boundary optimization in 2006. They focused on the difference in {r,g,b} color space. We implemented the code base for the drag and drop pasting as our first milestone.

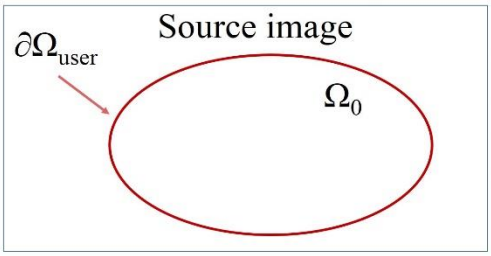
However, the drag and drop pasting method has its limitations. (1) Their energy function computes the boundary costs based on the L2-norm form and adds in a scalar factor k. Along the boundary, the color space of the source image has certain threshold k away from that of the target image, which may result in the blending results moving k away from the original source image color space; (2) k is a scalar value, and it appears in the cost function after the L2-norm of the three channels is computed. Same k differences can represent different meanings in the color space. For example, two pixels can have more difference in r channel, and the other two pixels have more difference in g channel, while these two groups may share the same color difference k because of the L2-norm.

Therefore, we want to improve on the existing approaches and focus more on the difference of the image substance and boundary reflectance. We represent the original color in {a,b} channels of {l,a,b} space, and we remove the effect of the scalar factor k.

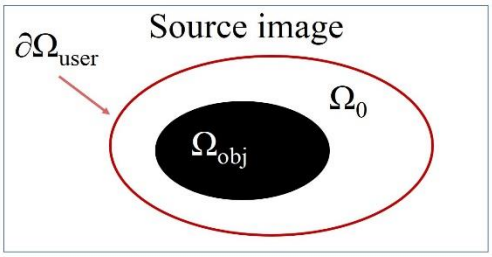
3. Project methodology

3.1. Process overview

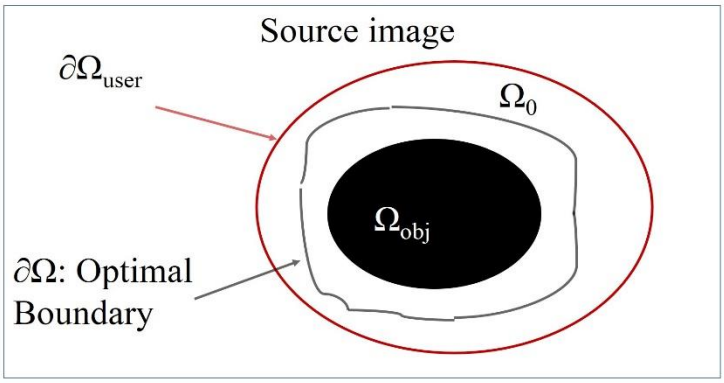
Step 1:
User selects the region Ω_0 from the source image.



Step 2:
Image segmentation:
GrabCut - find out the main object Ω_{obj}.



Step 3:
Boundary optimization: Our algorithm - find out the optimal boundary $\partial\Omega$ within $\partial\Omega_{user}$.



Step 4:
Image blending: Poisson image composition - based on optimal boundary $\partial\Omega$.

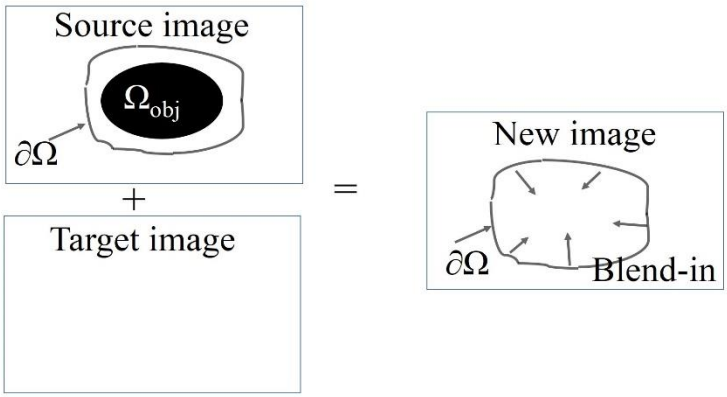


Figure 1: process overview

3.2. Image segmentation

The user selects the region Ω_0 (the region of interest), which contains the object Ω_{obj} (the object of interest) to be inserted onto the target image, from the source image. An ordinary user will not carefully trace the object boundary $\partial\Omega_{obj}$ during the selection process, but will draw a rough region Ω_0 instead.

Where is the optimal boundary to be found? Obviously it must be within the user-specified area Ω_0, and it should be outside the object Ω_{obj}. Therefore, before we proceed to the boundary optimization, we ought to find out where the object lies, and our optimized boundary should not intersect with it. The iterative segmentation techniques, such as GrabCut [3], can be used to produce Ω_{obj} when given Ω_0. For our project, all Ω_{obj}s are computed by GrabCut.

3.3. Boundary energy function

After obtaining the Ω_{obj}, we proceed to the step of boundary optimization. The optimal boundary $\partial\Omega$ should lie in between Ω_0 and Ω_{obj}. In order to minimize the original color difference along the boundary, the following energy function should be minimized:

- $E(\partial\Omega) = \sum (f_t(p) - f_s(p))^2$, s.t. $\Omega_{obj} \subset \Omega \subset \Omega_0$, $p \in \partial\Omega$

In our algorithm, $f_t(p)$ represents the value of a single pixel of the target image, and $f_s(p)$ represents the corresponding pixel value of the source image. f(p) is shown as the binary set in *{a, b}* channels in *{l, a, b}* color space, and ($f(p_1)$ -

$f(p_2)$) is computed as L2-norm form. The energy function shows the total sum of the original color difference along the boundary.

We eliminate *l* channel in *{l, a, b}* color space to remove the lightness effect of the images. We compute the cost function fully based on color channels *{a, b}* to better approximate the true color difference between the source image and the target image.

3.4. A Shortest closed-path problem

Given the boundary energy representation and the objective to minimize the function, we transform our boundary optimization problem into a shortest closed-path problem.

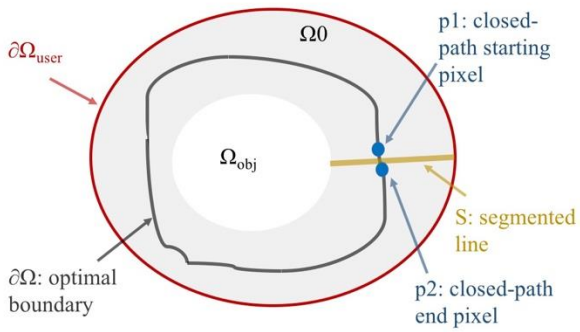


Figure 2: Closed-path problem illustration

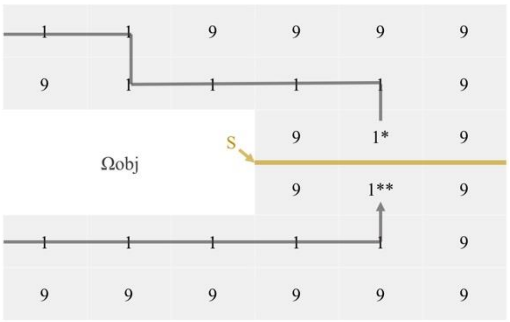


Figure 3: Pixel-look of the segmented line

As illustrated in the *Figure 2*, we draw a segmented line S (shown in yellow) from the object boundary $\partial\Omega_{obj}$ to $\partial\Omega_{user}$, on the right side of the Ω_0. To compute the optimal boundary, we start from one of the pixels p1 right above the S, and end with the neighboring pixel p2 right below p1. Among all paths starting from p1 to p2, the optimal path is the one with the least cumulative costs and

therefore the shortest closed-path. In order to find out the optimal boundary, we iterate through every pixel above S and select the path with the global minimum cumulative costs among all shortest paths.

Since the graph is a 2D grid, the boundary with minimized cumulative costs can be computed by 2D dynamic programming. In our project, we used Dijkstra algorithm to achieve the shortest path computation.

3.5. Image blending – Poisson image composition

Given the boundary, we could use Poisson image composition technique [1] to easily blend the images. For images with discrete pixels, in the new image, the composition first sets the boundary pixels (and pixels outside the boundary) equal to the target image, and subsequently constructs the new region based on the source image selected-region gradients.

4. What has been accomplished

In the first semester, we mainly implemented the code base for drag and drop pasting method. The implementation of our algorithm will be accomplished in the second semester.

4.1. Image segmentation

We used the GUI tool provided by Grab Cut Weebly website [4] implemented by Itay Blumenthal. GC_GUI is a useful tool to separate the object from the background image, and it is the preliminary step for us to block out the Ω_{obj} area.

Image segmentation details

Run GC_GUI in matlab, choose the source image (upper left image) and select the area Ω_0 (region of interest, area with red line in the upper right image) to be inserted onto the target. The tool will selects out the real object Ω_{obj} (object of interest, the boat in this example).

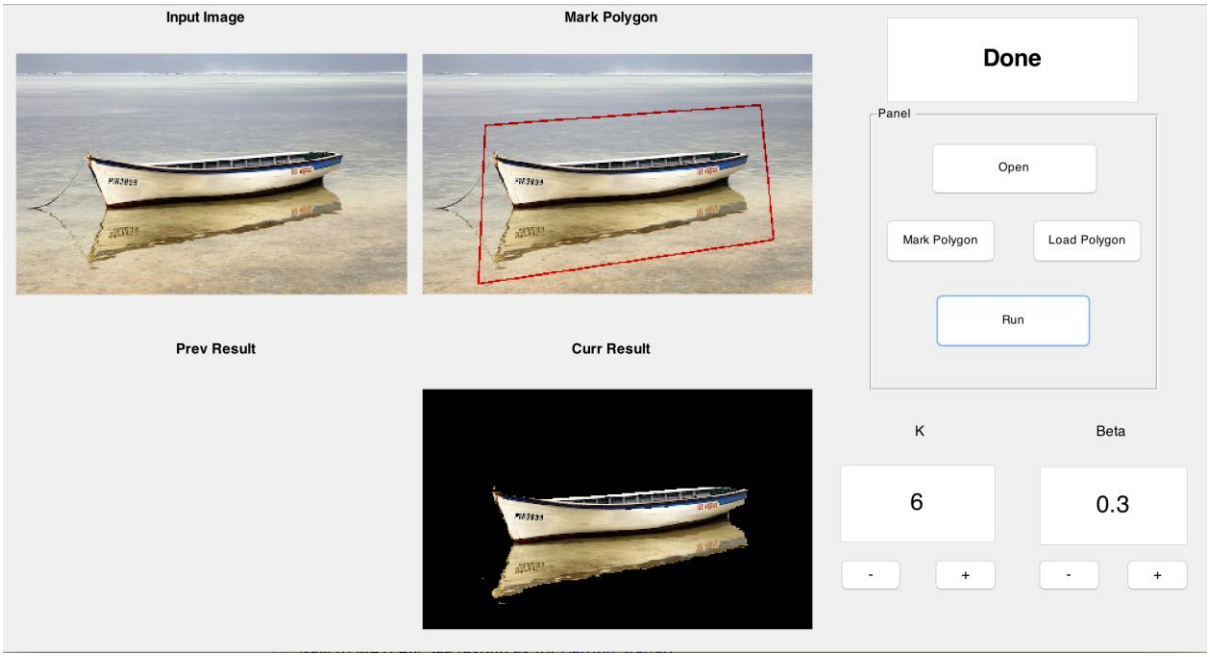


Table 1: Image segmentation details

We modified on top of the tool results. The output is the maskMatrix, which is a 2D matrix with exactly the same size as the source image. It will highlight the

area ($\Omega_0 \backslash \Omega_{obj}$), and block out the remaining area (source image$\backslash\Omega_0 + \Omega_{obj}$). The maskMatrix will be further used to calculate the cost matrix in the next step. Only the highlighted area will be computed.

4.2. Cost matrix calculation

- **Drag and drop pasting cost matrix implementation details**

In drag and drop pasting method, the whole process is an iterative algorithm (Details are disclosed in 4.4. Shortest path search). The cost matrix will be computed for multiple times and it will depend on the renewal of the scalar factor k.

Drag and drop pasting cost matrix implementation details [1]	
1:	**Procedure** (maskMatrix(sh*sw), simg, timg, offsetX, offsetY, k) [2]
2:	**Initialize**: costMatrix = maskMatrix (sh*sw)
3:	decompose simg, timg into {r,g,b} channels.
4:	**for** pixels p within the highlighted area of maskMatrix(i,j) [3]
5:	compute costs c between simg(i,j) & timg(offsetY+i, offsetX + j), k is involved
6:	costMatrix(i,j) = c
7:	**if** c = 0
8:	costMatrix(i,j) = 1e-10 [4]
9:	**return** costMatrix

Table 2: Drag and drop pasting cost matrix implementation details
1. The parameter details are disclosed in Appendix 7.1 Algorithm parameter details.
*2. sh*sw is the same size (source height * source width) as the source image (simg); offsetX and offsetY are offset position in the target image (timg), in X and Y axis respectively; k is a floating number, initially set as the average {r,g,b} L2-norm costs along the initial boundary, and will be substituted with the new average costs along the new boundary.*
3. Highlighted area: $\Omega_0 \backslash \Omega_{obj}$
4. We use a small number (1e-10) instead when the costs are zero. Details are disclosed in 4.3. Graph construction.

The *Figure 4* is an example of the returned cost matrix with highlighted area ($\Omega_0 \backslash \Omega_{obj}$, the dark grey area) set as costs, and non-highlighted area (source image$\backslash\Omega_0$ & Ω_{obj}) set as -1.

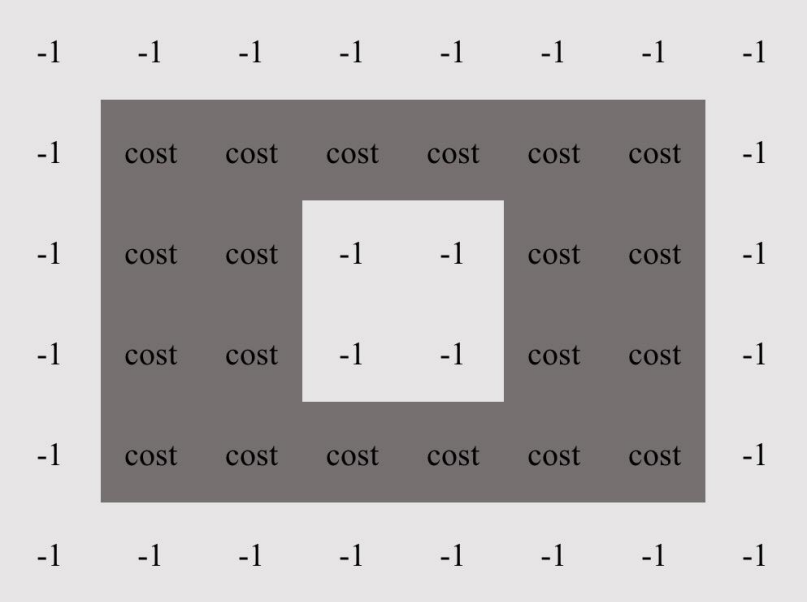


*Figure 4: Cost matrix (6*8) example*
**Highlighted area: dark grey area ($\Omega_0 \backslash \Omega_{obj}$, 4*6 excluding Ω_{obj})*
***Ω_{obj} area: 2*2 in the middle*

When the pixel cost is as zero (pixel color space between source image and target image is the same), we set a small number (1e-10) in the pixel location of the cost matrix. The reasons are when we later construct the graph based on the cost matrix for the shortest path problem, we still want to create the graph edge between zero-cost pixels. Details will be illustrated in 4.3. Graph construction.

4.3. Graph construction

In our theoretical analysis, the optimal boundary can be transformed to a shortest closed-path search problem in the graph. In this step, we will construct the graph for the path search, based upon the cost matrix computed.

Graph construction algorithm [1]

1: **Procedure** (costMatrix(sh*sw), segmentedLineRow, segmentedLineCol, segmentedLineEndCol) [2]
2: **Initialize**: totalPts = sw*sh; graphMatrix = sparse (totalPts*totalPts) [3]
3: **for** pixels p within the highlighted area of costMatrix(i,j) [4]
4: **if** p is *not* right above the segmented line S

5:	**for** all the neighbor pixels (s,t) of p within the highlighted area
6:	graphMatrix(sw*(i-1)+j, sw*(s-1)+t) = costMatrix(s,t)
7:	**else if** p is right above S
8:	same steps of 5&6 except neighbor pixels right below S
9:	**return** graphMatrix

Table 3: Graph construction algorithm
1. The parameter details are disclosed in Appendix 7.1 Algorithm parameter details.
2. costMatrix is taken from the previous step with sh & sw being the source image height & width respectively. The row and columns (start column & end column) of the segmentedLine S are also the parameter inputs.
3. graphMatrix is the sparse matrix, with the width and height being the total number of entries of the cost matrix.
4. Highlighted area: $\Omega_0 \backslash \Omega_{obj}$

graphMatrix is a N-by-N sparse matrix representing the graph, with N being the total number of entries of the cost matrix. The nonzero entries in graphMatrix represent the weights of the edges, for example, graphMatrix(i, j) is the edge weight from node i to j in the graph constructed.

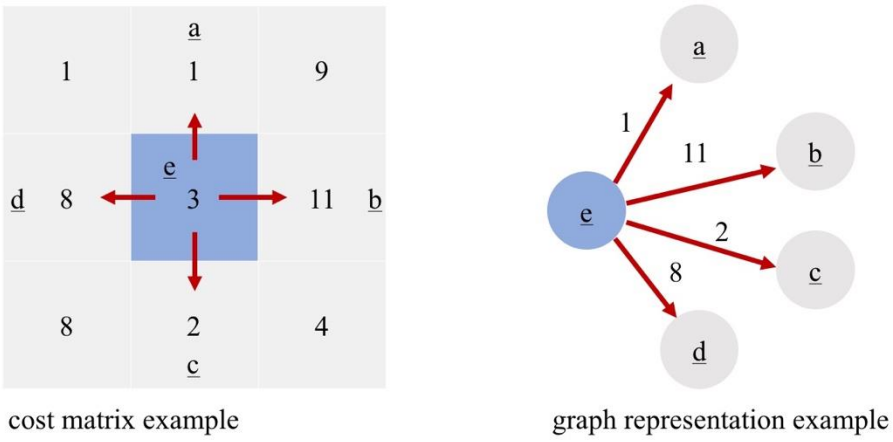


Figure 5: Graph construction example of the entry 3 (blue node e)

Substitution of zero cost with 1e-10 in cost matrix: Because graphMatrix is a sparse matrix, unless we change the graphMatrix(i,j) to a nonzero number, there will be no edge between node i and j. In the previous cost matrix computation, we substitute the cost zero with a small number 1e-10 for two reasons: (1) The shortest path will be in favor of the zero cost entry in the cost matrix, and we have to create the edge for this cost entry in the graph, but with small weight; (2)

Based on our calculation, 1e-10 is a comparatively small number compared to those cost matrix entries, therefore, it will not affect the results of the optimal path search.

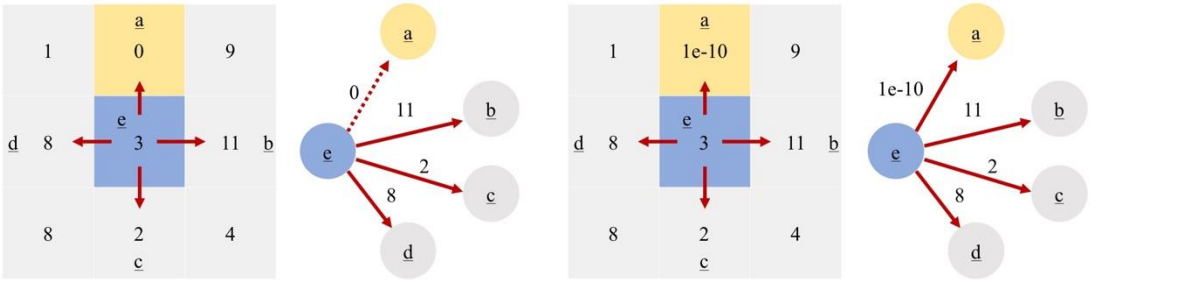


Figure 6: Example: substitution of zero to 1e-10

The effects of the segmented line S: The segmented line S is drawn on the right side of the cost matrix. In order to construct the graph for the paths surrounding the object Ω_{obj}, the paths can only go up across S, but *not* go down across.

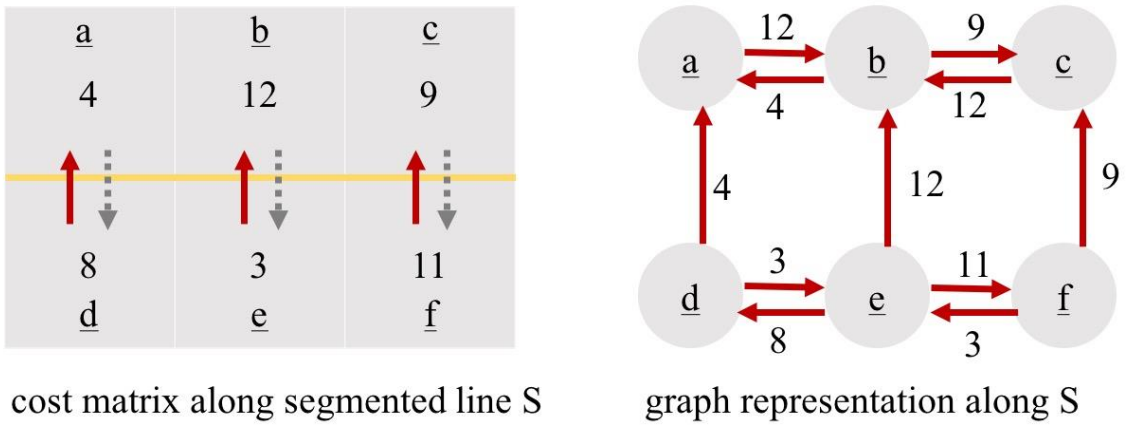


Figure 7: Graph representation along segmented line S

4.4. Shortest path search

We implemented the drag and drop pasting shortest path search. This is an iterative algorithm. We will search for the shortest path in the graph constructed, starting from nodes n right above S, and end with nodes right below n.

- **Drag and drop iterative path search**

This is an iterative algorithm which combines cost matrix computation, graph construction as well as the shortest path search steps. We implemented the automation script rgbScript.m for the whole process.

Drag and drop iterative path search	
1:	**Procedure** (k, maxItr) [1]
2:	**while** (maxItr >0)
3:	compute costMatrix -> cost matrix calculation based on k
4:	compute graphMatrix -> graph construction based on costMatrix
5:	compute global minimum path -> search for the shortest path in graphMatrix
6:	compute k -> calculate the new k based on the new path
7:	**Procedure** (k, maxItr-1)
8:	**return** path *l* with global minimum distance; mask of *l* [2]

Table 4: Drag and drop iterative path search
1. k is the floating number used in cost matrix calculation; maxItr represents the maximum iterations users want to impose.
2. mask based on the global shortest path l is used to blend the image in the 4.5. Image blending.

k calculation: The number k is a floating number, initially set as the average {r,g,b} L2-norm costs along the initial boundary, and will be substituted with the new average costs along the new boundary.

4.5. Image blending

We modified the image blending implementation details on top of the Yijia's code on Google archive. [5]

Image blending implementation [1]
Procedure (simg, timg, mask, offsetX, offsetY)
decompose simg and timg into {r,g,b} channels
for each channel
solve for Poisson equation based on the offset location (in the target image) and the mask
compose the new image nimg on new computed {r,g,b} values
return nimg

Table 5: Drag and drop iterative path search
1. The parameter details are disclosed in Appendix 7.1 Algorithm parameter details.

The blending process will depend on the mask and the offset location in the target image. Mask is the black-and-white image with the same size as the source image, with the white area showing the region to blend in. The boundary of the white area is the blend-in boundary. Offset location is the position to insert relative to the upper left pixel (1.1) of the target image. It is the same offset location when we compute the cost matrix in 4.2. Cost matrix calculation.

5. What will be done

In the second semester, we will better implement our boundary optimization algorithm, and we will finish massive testing to perfect our approach.

	Deadline	**Milestone**	**Description**
1.	February 29, 2016	Implement the new boundary optimization algorithm.	Boundary optimization related code implementation. There will be some overlapping steps between the existing implementations and our new one.
2.	March 31, 2016	Finish experiments and testing.	Massive trials & tests on our implementation at this stage. Comparisons should be done between 1. Our results vs. user-specified boundary 2. Our results vs. drag and drop pasting Also, we should include failed trials to discuss about the future works.
3.	Mid-April, 2016	Wrap up the project and finish the reports and related course materials.	Related materials include reports, website, presentation, posters, etc.

Table 6: Project schedule and milestones

6. References

[1] P. Perez, M. Gangnet and A. Blake, "Poisson Image Editing," 2003.

[2] J. Jia, J. Sun, C.-K. Tang and H.-Y. Shum, "Drag and Drop Pasting," 2006.

[3] C. Rother, V. Kolmogorov and A. Blake, "“GrabCut” - Interactive Foreground Extraction using Iterated Graph Cuts," 2004.

[4] I. Blumenthal, "CODE," [Online]. Available: http://grabcut.weebly.com/index.html.

[5] Y. Xu, H. Zhou and T. Leyvand, "Image Blending," [Online]. Available: https://code.google.com/archive/p/imageblending/.

7. Appendix

7.1. Algorithm parameter details

Cost matrix implementation parameters		
Para. name	**Explanation**	**Type**
maskMatrix	The matrix with the non-highlighted area (source image\Ω_0 & Ω_{obj}) set as -1.	source image height * width
simg	Source image	image height * width * 3
timg	Target image	image height * width * 3
offsetX	X-axis distance in target image between insert pixel and upper left pixel (timg(1,1)).	integer number
offsetY	Y-axis distance in target image between insert pixel and upper left pixel (timg(1,1)).	integer number
costMatrix	Returned cost matrix with highlighted area (Ω_0\$\Omega_{obj}$) set as costs, and non-highlighted area (source image\$\Omega_0$ & Ω_{obj}) set as -1.	source image height * width
k	Initially set as the average {r,g,b} L2-norm costs along the initial boundary, and will be updated with the new average costs along the new boundary.	floating number

Table 7: Cost matrix implementation parameters

Graph construction algorithm parameters		
Para. name	**Explanation**	**Type**
costMatrix	Cost matrix from the cost matrix calculation step.	source image height * width
segmentedLineRow	The row number of the segmented line S. Starting from 1.	integer number
segmentedLineCol	The starting column number of the segmented line S. Starting from 1.	integer number
segmentedLineEndCol	The ending column number of the segmented line S.	integer number
totalPts	The total number of entries of the cost matrix. Computed as source image height * width.	integer number
graphMatrix	An N-by-N sparse matrix representing the graph. Nonzero entries in graphMatrix represent the weights of the edges. For example, graphMatrix(i,j) represents edge weight from pixel(i) to pixel(j).	sparse matrix, with N being the total number of entries of the cost matrix.

Table 8: Graph construction algorithm parameters

Image blending implementation parameters		
Para. name	**Explanation**	**Type**
simg	Source image	source image height * width * 3
timg	Target image	target image height * width * 3
mask	The black-and-white mask, with the white area showing the region to blend in.	source image height * width
offsetX	X-axis distance in target image between insert pixel and upper left pixel (timg(1,1)).	integer number
offsetY	Y-axis distance in target image between insert pixel and upper left pixel (timg(1,1)).	integer number

Table 9: Image blending implementation parameters