Project 1 : Intelligent Scissor

Assigned : February 12
Due : March 4

Quick Links

1. Grade sheet
2. Sample Solution [complete] by Yin Li (implemented by MFC and GDI) new!
3. Sample Solution [partial]
4. Example W2K skeleton UI by FLTK -- from **Impressionist** project (use your CSD username and password to logon)
5. Source for priority queue implementation
6. HelpSession: OpenGL
7. HelpSession: Image processing
8. Intelligent Scissor FAQ
9. Tool Kit Resources(FLTK, OpenGL)
10. Free digital images!! For your image scissor artifacts...
11. Use Google's image search capability to find more images.
12. When downloading images from the sources above, remember to save them in BMP format if you start with the example UI in FLTK!

Project Description

Synopsis

In this project, you will create a tool that allows a user to cut an object out of one image and paste it into another. The tool helps the user trace the object by providing a "live wire" that automatically snaps to and wraps around the object of interest. You will then use your tool to create a composite image.

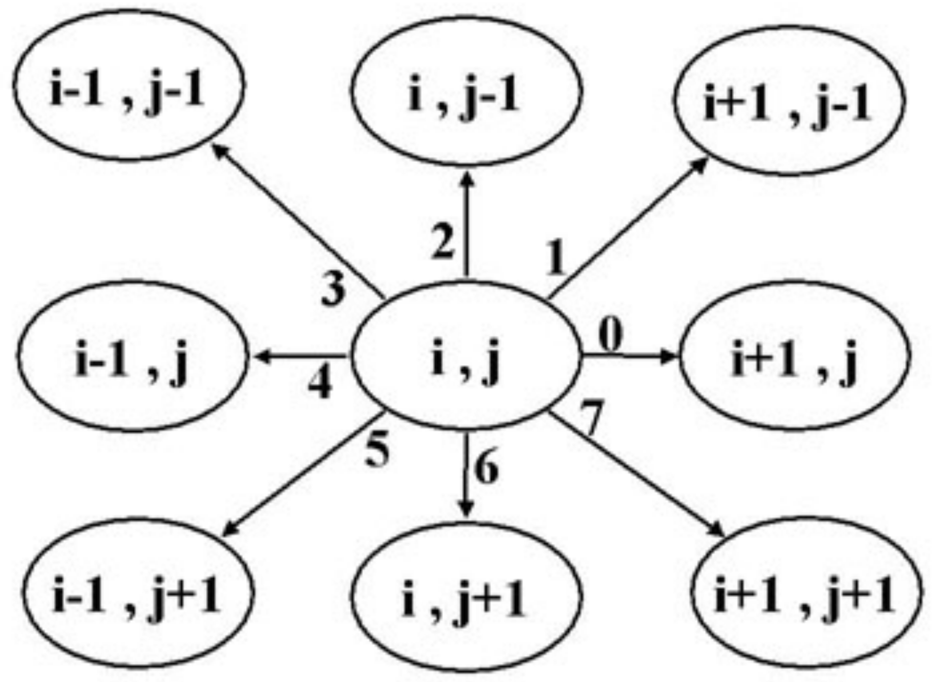
We have provided a sample solution executable and test images. Note that this sample solution may not be complete, but it contains a feature to visualize the cost graph and other related data structures that should be useful when you debug your own project. Try this out to see how your program should run.

Description

This program is based on the paper *Intelligent Scissors for Image Composition*, by Eric Mortensen and William Barrett, published in the proceedings of SIGGRAPH 1995. The way it works is that the user first clicks on a "seed point" which can be any pixel in the image. The program then computes a path from the seed point to the mouse cursor that hugs the contours of the image as closely as possible. This path, called the "live wire", is computed by converting the image into a graph where the pixels correspond to nodes. Each node is connected by links to its 8 immediate neighbors. Note that we use the term "link" instead of "edge" of a graph to avoid confusion with edges in the image. Each link has a cost relating to the derivative of the image across that link. The path is computed by finding the minimum cost path in the graph, from the seed point to the mouse position. The path will tend to follow edges in the image instead of crossing them, since the latter is more expensive. The path is represented as a sequence of links in the graph.

Next, we describe the details of the cost function and the algorithm for computing the minimum cost path. The cost function we'll use is a bit different than what's described in the paper, but closely matches what was discussed in lecture.

As described in the lecture notes, the image is represented as a graph. Each pixel (i,j) is represented as a node in the graph, and is connected to its 8 neighbors in the image by graph links (labeled from 0 to 7), as shown in the following figure.



Cost Function

To simplify the explanation, let's first assume that the image is grayscale instead of color (each pixel has only a scalar intensity, instead of a RGB triple) as a start. The same approach is easily generalized to color images.

- Computing cost for grayscale images

  Among the 8 links, two are horizontal (links 0 and 4), two are vertical (links 2 and 6), and the rest are diagonal. The magnitude of the intensity derivative across the diagonal links, e.g. link1, is approximated as:

  ```
  D(link1)=|img(i+1,j)-img(i,j-1)|/sqrt(2)
  ```

  The magnitude of the intensity derivative across the horizontal links, e.g. link 0, is approximated as:

  ```
  D(link 0)=|(img(i,j-1) + img(i+1,j-1))/2 - (img(i,j+1) + img(i+1,j+1))/2|/2
  ```

  Similarly, the magnitude of the intensity derivative across the horizontal links, e.g. ln2, is approximated as:

  ```
  D(link2)=|(img(i-1,j)+img(i-1,j-1))/2-(img(i+1,j)+img(i+1,j-1))/2|/2.
  ```

  We compute the cost for each link, cost(link), by the following equation:

  ```
  cost(link)=(maxD-D(link))*length(link)
  ```

  where maxD is the maximum magnitude of derivatives across links over in the image, e.g., maxD = max{D(link) | forall link in the image}, length(link) is the length of the link. For example, length(link 0) = 1, length(link 1) = sqrt(2) and length(link 2) = 1. If a link lies along an edge in an image, we expect that the intensity derivative across that link is large and accordingly, the cost of link is small.

- Cost for an RGB image

  As in the grayscale case, each pixel has eight links. We first compute the magnitude of the intensity derivative across a link, in each color channel independently, denoted as

  ```
  ( DR(link),DG(link),DB(link) ).
  ```

  Then the magnitude of the color derivative across link is defined as

  ```
  D(link) = sqrt( (DR(link)*DR(link)+DG(link)*DG(link)+DB(link)*DB(link))/3 ).
  ```

  Then we compute the cost for link link in the same way as we do for a gray scale image:

  ```
  cost(link)=(maxD-D(link))*length(link).
  ```

  Notice that cost(link 0) for pixel (i,j) is the same as cost(link 4) for pixel (i+1,j). Similar symmetry property also applies to vertical and diagonal links.

  For debugging perpose, you may want to scale down each link cost by a factor of 1.5 or 2 so that they can be converted to byte format without clamping to[0,255]

Computing the Minimum Cost Path

The pseudo code for the shortest path algorithm in the paper is a variant of Dijkstra's shortest path algorithm, which is described in any of the classic algorithm books (including text books used in data structures courses like 271). You could also refer to any of the classic algorithm text books (e.g., Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein, published by MIT Press). Here is some pseudo code which is equivalent to the algorithm in the SIGGRAPH paper, but we feel is easier to understand.

```
procedure LiveWireDP

    input: seed, graph

    output: a minimum path tree in the input graph with each node pointing to its predecessor along the minimum cost
    path to that node from the seed.  Each node will also be assigned a total cost, corresponding to the cost of the the
    minimum cost path from that node to the seed.

    comment: each node will experience three states: INITIAL, ACTIVE, EXPANDED sequentially. the algorithm
    terminates when all nodes are EXPANDED. All nodes in graph are initialized as INITIAL. When the algorithm runs,
    all ACTIVE nodes are kept in a priority queue, pq, ordered by the current total cost from the node to the seed.

Begin:

    initialize the priority queue pq to be empty;

    initialize each node to the INITIAL state;

    set the total cost of seed to be zero and make seed the root of the minimum path tree ( pointing to NULL ) ;

    insert seed into pq;

    while pq is not empty

        extract the node q with the minimum total cost in pq;

        mark q as EXPANDED;

        for each neighbor node r of q

            if  r has not been EXPANDED

                if  r is still INITIAL

                    make q be the predecessor of r ( for the the minimum path tree );

                    set the total cost of r to be the sum of the total cost of q and link cost from q to r
                    as its total cost;

                    insert r in pq and mark it as ACTIVE;

                else if  r is ACTIVE, e.g., in already in the pq

                    if the sum of the total cost of q and link cost between q and r is less than the total cost
                    of r

                        update q to be the predecessor of r ( for the minimum path tree );

                        update the total cost of r in pq;

End
```

We provide the priority queue functions that you will need in your project (implemented as a Fibonacci Heap your learned in 271). These are: **ExtractMin, Insert, and Update (DecreaseKey)**.

Implementation

The impressionist project provides a sample skeleton code for you to open and save an image. However, it contains some unrelated features (e.g. the paint brushes). But the portion of the codes that open, display, and save an image file, and the codes that track the mouse movement, should be useful. However, you may opt not to use it a as your starter codes (e.g. you may use MFC). The following are some high-level suggestions on data structures. Again, you can have your own choice. To make your life easier, we provide an implementation for priority queues. But you have to make up your own data structures for the cost graph, etc.

Data structures

The data structure that you will use in this project is the Pixel Node.

Pixel Node

We suggest you to use the following Node structure when computing the minimum path tree. You may have your own choice.

```
struct Node{
   double linkCost[8];
   int state;
   double totalCost;
   Node *prevNode;
   int column, row;
   //other unrelated fields;}
```

linkCost contains the costs of each link, as described above.
state is used to tag the node as being INITIAL, ACTIVE, or EXPANDED during the min-cost tree computation.
totalCost is the minimum total cost from this node to the seed node.
prevNode points to its predecessor along the minimum cost path from the seed to that node.
column and row remember the position of the node in original image so that its neighboring nodes can be located.

For visualization and debugging purposes, you should convert a pixel node array into an image that displays the computed cost values in the user interface. This image buffer, called Cost Graph, has the structure shown below. Cost Graph has 3W columns and 3H rows and is obtained by expanding the original image by a factor of 3 in both the horizontal and vertical directions. For each 3 by 3 unit, the **RGB(i,j)** color is saved at the center and the eight link costs, as described in the Cost Function section, are saved in the 8 corresponding neighbor pixels. The link costs shown are the average of the costs over the RGB channels, as described above (NOT the per-channel costs). The Cost Graph may be viewed as an RGB image in the interface, (dark = low cost, light = high cost). This feature is also present in the partial sample solution.

Cost(link3)	Cost(link2)	Cost(link1)	Cost(link3)	Cost(link2)	Cost(link1)	Cost(link3)	Cost(link2)	Cost(link1)
Cost(link4)	**RGB(i-1,j-1)**	Cost(link0)	Cost(link4)	**RGB(i,j-1)**	Cost(link0)	Cost(link4)	**RGB(i+1,j-1)**	Cost(link0)
Cost(link5)	Cost(link6)	Cost(link7)	Cost(link5)	Cost(link6)	Cost(link7)	Cost(link5)	Cost(link6)	Cost(link7)
Cost(link3)	Cost(link2)	Cost(link1)	Cost(link3)	Cost(link2)	Cost(link1)	Cost(link3)	Cost(link2)	Cost(link1)
Cost(link4)	**RGB(i-1,j)**	Cost(link0)	Cost(link4)	**RGB(i,j)**	Cost(link0)	Cost(link4)	**RGB(i+1,j)**	Cost(link0)
Cost(link5)	Cost(link6)	Cost(link7)	Cost(link5)	Cost(link6)	Cost(link7)	Cost(link5)	Cost(link6)	Cost(link7)
Cost(link3)	Cost(link2)	Cost(link1)	Cost(link3)	Cost(link2)	Cost(link1)	Cost(link3)	Cost(link2)	Cost(link1)
Cost(link4)	**RGB(i-1,j+1)**	Cost(link0)	Cost(link4)	**RGB(i,j+1)**	Cost(link0)	Cost(link4)	**RGB(i+1,j+1)**	Cost(link0)
Cost(link5)	Cost(link6)	Cost(link7)	Cost(link5)	Cost(link6)	Cost(link7)	Cost(link5)	Cost(link6)	Cost(link7)

Pixel layout in Cost Graph (3W*3H)

User Interface

In the partial sample solution, it contains some of the following features.

File-->Save Contour, save image with contour marked;

File-->Save Mask, save compositing mask for PhotoShop;

Tool-->Scissor, open a panel to choose what to draw in the window

Work Mode:

Image Only: show original image without contour superimposed on it;

Image with Contour: show original image with contours superimposed on it;

Debug Mode:

Pixel Node: Draw a cost graph with original image pixel colors at the center of each 3by3 window, and black everywhere else;

Cost Graph: Draw a cost graph with both pixel colors and link costs, where you can see whether your cost computation is reasonable or not, e.g., low cost (dark intensity) for links along image edges.

Path Tree: show minimum path tree in the cost graph for the current seed; You can use the counter widget to simulate how the tree is computed by specifying the number of expanded nodes. The tree consists of links with yellow color. The back track direction (towards the seed) goes from light yellow to dark yellow.

Min Path: show the minimum path between the current seed and the mouse position;

To use the "path tree" and "min path" in debug mode, you need to have an *active seed point*.

the active seed point is the last "left click / ctrl + left click" point on the contour you are working with, e.g., the contour that has not yet be committed by "enter / ctrl + enter". (See the short cut keys below)

Ctrl+"+", zoom in;

Ctrl+"-", zoom out;

Ctrl+Left click first seed;

Left click, following seeds;

Enter, finish the current contour;

Ctrl+Enter, finish the current contour as closed;

Backspace, when scissoring, delete the last seed; otherwise, delete selected contour.

Select a contour by moving onto it. Selected contour is red, un-selected ones are green.

The Artifact

For this assignment, you will turn in a final image (the *artifact*) which is a composite created using your program. Your composite can be derived from as many different images as you'd like. Make it interesting in some way--be it humorous, thought provoking, or artistic! You should use your own scissoring tool to cut the objects out and save them to matte files, but then can use Photoshop or any other image editing program to process the resulting mattes (move, rotate, adjust colors, warp, etc.) and combine them into your composite. Instructions on how to do this in Photoshop are provided here. You should still turn in an artifact even if you don't get the program working fully, using the scissoring tool in the sample solution or in Photoshop.

What to Turn In

Upload your code and executable as a zip file, **iscissor.zip**, to CASS before 11:59pm on the due date. In your zip file, please include the following items:

- Your source and executable, as a zip file called **iscissor.zip**.
- A README (**README.html**) describing (1) any extra credit features you implemented, and (2) any aspects of your code that require explanation. If you did exactly the basic requirements as described in this handout, you **don't** need this README.
- Your artifact, a zip file called **iscissor_artifact.zip**

Make a webpage **http://www.cse.ust.hk/~yourlogin/5421/iscissor/index.html** showing both your artifact and the original images and masks. If you'd like, you can also provide more detail in this web page (e.g., "the making of..."). You can include more than one composite result if you'd like. We'll link your pages to the course web page online.

Demos

We are scheduling a demo session on the due date, where each student will give a 10 minute live demo of their project executable and answer questions about the source code. We'll pass around a signup sheet in class (or post it online) soon.

Bells and Whistles

Here is a list of suggestions for extending the program for extra credit. You are encouraged to come up with your own extensions. We're always interested in seeing new, unanticipated ways to use this program!

One problem with the live wire is that it prefers shorter paths so will tend to cut through large object rather than wrap around them. One way to fix this is specify a specific region in which the path must stay. As long as this region contains the object boundary but excludes most of the interior, the path will be forced to follow the boundary. One way of specifying such a region is to use a thick (e.g., 50 pixel wide) paint brush. Implement this feature. Note: we already provide support for brushing a region using a selection buffer, in the same way as for project0.

Modify the interface and program to allow blurring the image by different amounts before computing link costs. Describe your observations on how this changes the results.

Try different costs functions, for example the method described in *Intelligent Scissors for Image Composition*, and modify the user interface to allow the user to select different functions. Describe your observations on how this changes the results.

The only point that doesn't snap to edges is the seed. Implement a seed snapping feature, where the seed is automatically moved to the closest edge.

Implement path cooling, as described in *Intelligent Scissors for Image Composition*.

🔔 Implement dynamic training, as described in *Intelligent Scissors for Image Composition*.

🔔 🔔 Implement a live wire with *sub-pixel* precision. You can find the position of an edge to sub-pixel precision by fitting a curve (e.g., a parabola) to the gradient magnitude values across an edge, and finding the maximum. Another way (more complex but potentially better) of doing this is given in a follow on to Mortensen's scissoring paper. It is probably easiest to first compute the standard (pixel-precision) live wire and then use one of these curve fitting techniques to refine it.