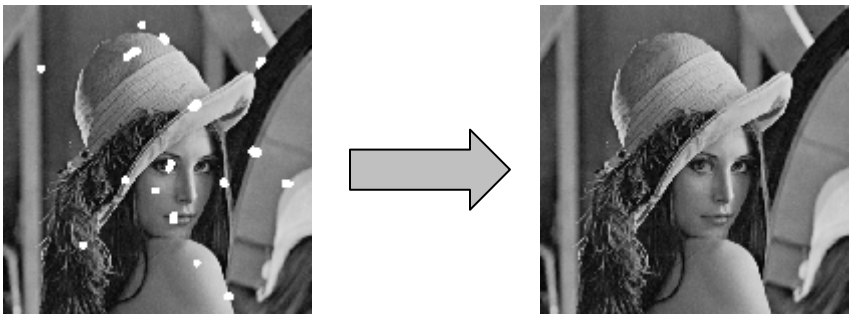
Ooooh! Is that your face?!

Digital Communications and Signal Processing
18-551 – Fall 2005



Group 6

Charlette Betts (cbetts@andrew.cmu.edu)
Marcel Davey (mdavey@andrew.cmu.edu)
John Gu (jxg@andrew.cmu.edu)
Neema Jain (nvjain@andrew.cmu.edu)

TABLE OF CONTENTS

INTRODUCTION

Problem Description:

The restoration and modification of images in a way that is not detectable by an observer who has not seen the original picture is a very old practice – dating back to the manual restoration of medieval art by filling in any gaps that may have distorted the artwork over the years. In modern times, the digitizing of analog images to 'live forever' and preventing their decay, often results in defects like scratches, etc. that have to be removed. This practice is known as inpainting. The aim of inpainting is to reconstruct the missing or damaged portions of the picture, in order to restore its unity. The obvious need to restore images extends from paintings to photographs and films. The purpose, however, remains the same – to recondition any deterioration (e.g., cracks or scratches in paintings and photographs), or to modify (e.g., add or remove elements like red eyes), the goal being to produce a modified image in which the inpainted region is merged into the original image so well that an observer is not aware of the modification. The filling-in of missing information is an important aspect of image processing, with applications that range from image restoration, to image coding and transmission (e.g., recovering lost packets), and special effects (e.g., removal of objects).

Traditionally, artists performed image inpainting manually which was a very cumbersome and tedious process. Motivated in part by the work of Nitzberg-Mumford and Masnou-Morel, Bertalmio et al have developed algorithms for digital inpainting of still images that produce very impressive results **[7]**. However, the algorithm usually takes several minutes for the inpainting of a relatively small area, prompting our group to research other faster algorithms that can produce similar results in lesser time, with added features.

Solution and Project Goal:

Our goal in this project was to find ways to remedy the primary defects that afflict digital and scanned photographs, using a combination of algorithms which would make the inpainting process faster, and would also require less user-input.

PRIOR WORK

Previous Project:

As discussed in class, and in the presentation, no prior project has covered this topic

Other Work:

Image restoration and repair can be more efficiently tackled using a system that automatically fills in damaged areas. This technique, better known as digital image inpainting, was pioneered by Marcelo Bertalmio and Guillermo Sapiro **[1]** in which they used partial differential equations (PDEs) to model the way pigments of various shades of gray might seep into a central pool from the shoreline. In our case, this would be the defective areas in the image. These equations specify the directions and rates at which the shade changes throughout the pool. After a number of iterations, this procedure fills in the blank area. For a color image, the technique is applied independently to each of the three grayscale images, which can then be combined to generate a color rendering.

In Bertalmio's paper **[1]**, the image smoothness information, estimated by a Laplacian (which can be approximated by a convolution sum), is propagated along the isophotes directions (bands equal on the grey-scale), estimated by the image gradient rotated 90 degrees so that the propagation occurs perpendicular to the edge of the boundary, growing the correction inwards into the corrupted region.

The Total Variational model proposed by Chan and Shen **[2]** uses an Euler-Lagrange equation together with direction dependent diffusion to maintain the isophotes directions. This method has the benefit of preserving edges and lines better. A further improvement to the Total Variational technique is made in the Curvature Driven Diffusion technique, also pioneered by Chan and Shen **[3]**. In this method, diffusion is driven along the (bands of equi-intensity and color) isophotes, allowing inpainting of larger areas. The premise behind variational approaches is to postulate an energy, which is minimized by the extension of the image to the corrupted region. Intensity of the pixels and their positions are all weighted and incorporated in the algorithm.

Most algorithms employ partial differential equations of one sort or another to describe the flow of the inpainting, which ought to follow the strength and direction of the surrounding isophotes. However, probabilistic methods **[4]** can also be employed, though to produce the best inpainting results, the probabilistic and variational approaches are used in tandem. Probabilistic methods propose that a probability distribution can approximate the essential features and interactions of different structures relevant in the photograph. Bayesian statistics and Markov models are extensively employed in these methods.

There are a third group of solution methods that are far simpler, produce similar results in most cases, and are far faster. These methods are those such as normalized convolution and other convolution based techniques. The premise behind normalized convolution is the interpolation of signal we do have, to reconstruct the lost segment. The missing values of the signal are calculated by interpolation, usually done by convolution. The convolution can be made more effective by a normalization operation that takes into account the possibility of missing samples.

In Knutsson and Westin **[5]** the problem of the image analysis when performed on irregularly sampled image data is considered under the theory of signal and its certainty. This is to consider the separation of both data and operator applied to the data in a signal part and a ‘certainty’ part. Missing data in irregularly sampled series is handled by setting the certainty of the data equal to zero. In the case of uncertain data, an estimate of certainty accompanies the data, and this can be used in a probabilistic framework. The theory that they developed following these ideas is called Normalized Convolution.

Another convolution based technique by Oliveira **[6]**, repeatedly convolves a filter over the missing regions so that the information from the edges is diffused inwards to the corrupted region. Both these convolution processes are reasonable in reconstructing images but falter in terms of resolution. The corrupted segments get populated, but everything tends to get blurred.

ALGORITHMS

The different types of algorithms, in the past, have broadly been classified as: (i) ***Texture Synthesis*** algorithms that generate image regions from sample textures, and (ii) ***Inpainting techniques*** that fill in small gaps and holes in the pictures. Texture Synthesis algorithms work better with ‘textures’, two-dimensional patterns that repeat; Inpainting techniques focus on linear structures such as lines and contours that can be thought of as one-dimensional patterns. The most efficient algorithms included as part of the project are – the Convolution Method, the Total Variational (T.V.) method, and the Exemplar Method.

Why did we choose these?

The convolution method exhibited compelling reasons to select it. It was the fastest algorithm on thin defects (less than 9 pixels across) producing results comparable to the other candidates.

Traditional texture synthesis models, as well as variational models were considered. They both inpainted large sized defections. However, the variational models respected geometrical properties of the picture. This was done by calculating a set of partial differential equations to determine how to grow the isophotes (bands of equal color and intensity). However, as diffusion was part of the process, blurring would occur as the inpainting proceeded inwards. Research has shown that it is very computationally expensive with the inpainting time proportional to the area of the defect.

Traditional methods of texture synthesis would grow inward from the boundary of the corrupted region. Resolution is maintained as there is no diffusion involved – textures sampled from the rest of the image are plugged into the inpainted block.

However, both traditional texture synthesis and the variational models were discarded as most images are a combination of textures and geometry. The chosen algorithm, the exemplar based algorithm, was chosen. It maintains the resolution of a picture, using texture synthesis methods. However, the synthesis is guided by geometrical attributes in an image (the colors representing different objects in a picture). It finds and linearly extends the isophotes of the image through texture synthesis, growing these sharp color gradients before attacking the rest of the boundary, maintaining the lines and edges that define the geometrical attributes of an image. It thus combines the advantages of the variational and texture synthesis methods, leaving behind their shortcomings. However, it is more computationally expensive than the convolution method. As traditional texture synthesis methods don't find color gradients to grow on, this method is more expensive than traditional texture synthesis algorithms.

Convolution Method:

The technique of blurring the colors out into the missing areas is called Convolution. Mathematically, repeated blurring and diffusion are identical. Isotropic diffusion is the idea behind these methods. When an image is blurred, the colors of each pixel are averaged with a small portion of the color from neighboring pixels. That pixel, in turn, contributes a small part of its color to each of its neighbors, that is, colors from the untainted areas are spread into the corrupted zones in an even way. Directions and isotopes are not given a weighting and as the zones of corruption are narrow, we don't have to diffuse very far and the deformations resulting from such diffusion are not very great.

This method works well for deformations that are not big. A large amount of surface area may be corrupted, but as long as the corruptions themselves are 'thin' in nature, the algorithm works well. This is good as most deformations are pen marks and scratches which are thin in nature. It is also good at filling in the picture when we only have a few components of the original signal. This happens if you're receiving an image over a noisy or faulty line.

The general idea is to create a kernel of some sort (an N x N matrix) that is repeatedly convolved over areas of picture, which fills in a pixel based on surrounding pixel values. The values in the matrix determine the way it will spread colors from the surrounding scene into the corrupted zone. This process is repeated over and over until the image is restored. Such a method has the benefit of being very fast as only multiplication is used. An example of such a kernel is given below:

a	b	a
b	0	b
a	b	a

c	c	c
c	0	c
c	c	c

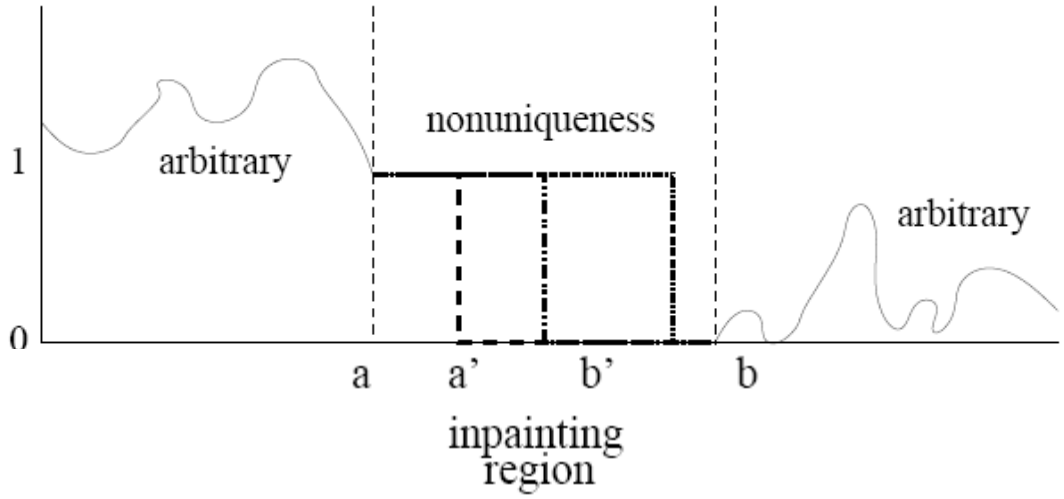
a = 0.073235

b = 0.176765

c = 0.125

Total Variation Method:

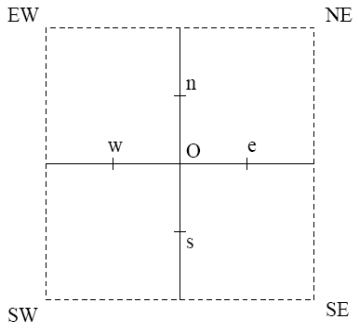
The Total Variation (TV) method minimizes the absolute value of the gradient of the image. The reason for implementing the TV method is because it is much better at dealing with sharp edges than other methods are. For TV there is a high degree of nonuniqueness which holds true even when we minimize the TV over an appropriate space. This lack of uniqueness is because of the high degree of symmetry or the fact that all monotone jumps are treated equally. The numerical solution propagates the boundary data inwards at equal speeds, so a minimizer that is symmetric at the mid point will be chosen.



In this method we solve the following ODE:

$$\frac{d}{dx}\left(\frac{u'}{|u'|}\right) = 0 \quad for x \in [a,b] \qquad u(a) = 1,\ u(b) = 0$$

The following graph shows O as the inpainted region and the directions as the computed neighbors. This method uses the neighbors of the corrupted region to calculate the new inpainted region. Each direction (North, South, East, and West) is equated using a circular shift and set as neighbor values (u_E, u_N, etc)



After these are computed, the weights are computed. This computation is shown in the following equation, A. This equation is the gut of the numerical implementation:

$$|\nabla u_\varepsilon| \approx \frac{1}{h}\sqrt{(u_E - u_O)^2 + [(u_{NE} + u_N - u_S - u_{SE})/4]}$$

The key to this method is in this approximation and the degeneracy of the PDE. After computing the weights, the method calls for approximating the new inpainted region. The equation becomes

$$0 = \sum_{P \in \Lambda_O} \frac{1}{|\nabla u_p|}(u_O - u_P)$$

by using

$$w_P = \frac{1}{|\nabla u_p|} \quad P \in \Lambda_O$$

$$h_P = \frac{w_P}{\sum_{Q \in \Lambda_O} w_Q}$$

We can solve this using a Gauss-Jacobi iteration scheme for linear systems, which gives the following update of $u^{(n-1)}$ to $u^{(n)}$ by the following equation:

$$u_O^{(n)} = \sum_{P \in \Lambda_O} h_P^{(n-1)} u_P^{(n-1)}.$$

Due to h being a low-pass filter, the iterative algorithm is stable and satisfies the maximum principle. The final process is multiple iterations. This is done to achieve the best result.

Exemplar Method:

The Exemplar method can essentially replicate both texture and structure. However, the success of the method depends highly on the order in which the filling proceeds. The procedure involves selecting a target region to be filled. The size of the template window is specified as a 9x9 pixel window. After the target region and window size have been decided on, we calculate the *confidence term* and the *data term*.

Each pixel, in the algorithm, has a specific color value, and a confidence value that represents the confidence in the color value. Furthermore, patches along the fill front have a priority value which establishes the order in which the patches are filled. The general structure of the algorithm is as follows –

Say we start with a patch, $\Psi_{\mathbf{p}}$, centered at point 'p'. The priority of this patch is calculated as:

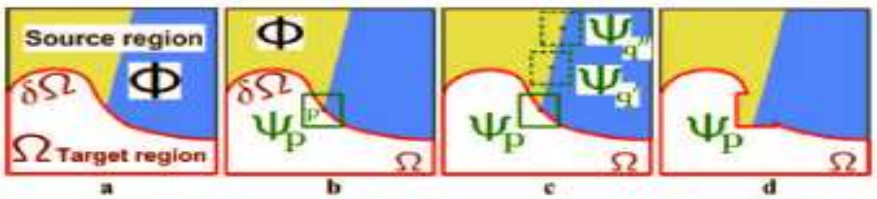
$$P(\mathbf{p}) = C(\mathbf{p})D(\mathbf{p}).$$

Where C(p) is the confidence term and D(p) is the data term, calculated as:

$$C(\mathbf{p}) = \frac{\sum_{\mathbf{q} \in \Psi_{\mathbf{p}} \cap \bar{\Omega}} C(\mathbf{q})}{|\Psi_{\mathbf{p}}|}, \qquad D(\mathbf{p}) = \frac{|\nabla I_{\mathbf{p}}^{\perp} \cdot \mathbf{n}_{\mathbf{p}}|}{\alpha}$$

Here, $|\Psi_{\mathbf{p}}|$ defined as the area of the patch, and α is the normalization factor. We calculate the priorities for all patches along the boundary of the fill region. The confidence term, C(p), helps in filling those patches first which have more of their pixels filled in already, either because they were never part of the target region, or because they were filled in earlier. The data term D(p), gives patches that have higher color gradients a higher priority. Thus the priority with which we fill a patch depends on how many pixels are already filled in within that region, and how high a color gradient exists within that patch.

A representation of the process is explained as –



As the filling process goes on, the pixels towards the target boundary or contour will have higher confidence values, and will be filled before the pixels at the center of the target region which have lower confidence values. The Data term D(p) depends on the isophotes hitting the contour on each iteration.

A clearer way to understand this procedure is to look at the calculations in the MATLAB code.

The gradient, Ix Iy is calculated for the image coordinates in each of the colors – R G B (Red, blue and green). The gradient measures the change in the color components, thus measuring the change in amount of Red, Blue or Green in each pixel. This means that there would be a higher gradient value at color boundaries. (For example, for the pixels at the boundary of the sky and grass, the gradient value is higher than the gradient values for pixels in the sky region.)

This is for Red. The same line is repeated for img(:,:,2) and img(:,:,3) (blue and green)

```
[Ix(:,:,1) Iy(:,:,1)] = gradient(img(:,:,1));
                  % add Red blue green components, divide by the size
Ix = sum(Ix,3)/(3*255); Iy = sum(Iy,3)/(3*255);
```

Convole the flll region with the patch and for all values>0, store them in dR

```
dR = find(conv2(fillRegion,[1,1,1;1,-8,1 ;1,1,1] , 'same')>0);
```

N stores the gradient of the not fill region. We get the corresponding location in dR from this gradient, and normalize it.

```
[Nx,Ny] = gradient(~fillRegion);
N = [Nx(dR(:)) Ny(dR(:))];
N = normr(N);
```

The calculated dR is used to get the patch with the highest priority, say best_patch and fill it with data extracted from the source region.

The priorities and confidence values are calculated as –

```
q = best_patch(~(fillRegion(best_patch)));
                  C(k) = sum(C(q))/numel(best_patch);
                                        ↑_____Number of elements in best_patch
```

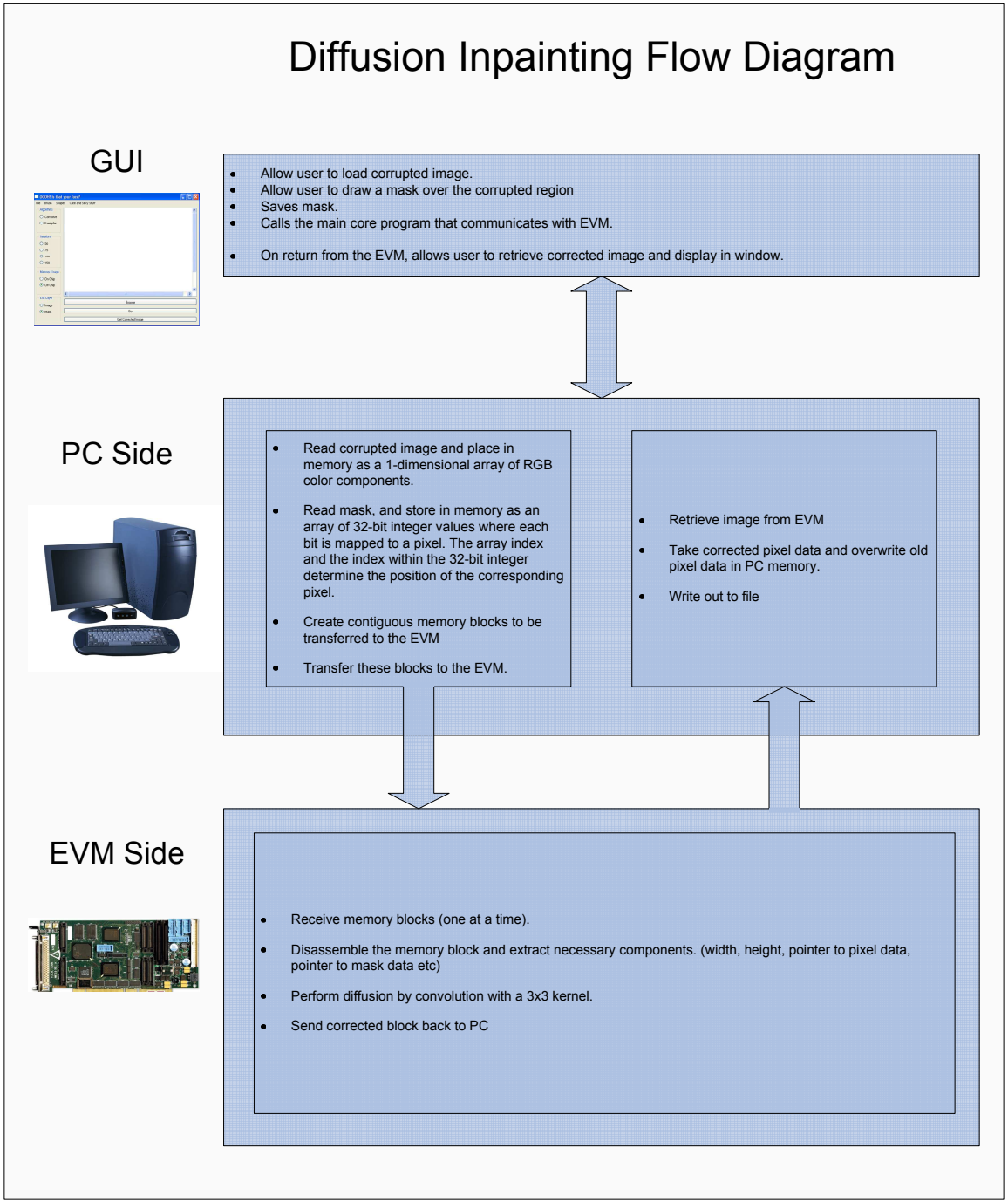
(This code follows the definition for confidence terms as stated earlier)

Patch priorities = confidence term * data term

```
D(dR) = abs(Ix(dR).*N(:,1)+Iy(dR).*N(:,2)) + 0.001;
priorities = C(dR).* D(dR);
```

After this, we find the best exemplar patch, such that it has a minimum difference with respect to the patch being filled in. The data from the best exemplar patch is copied to the patch being filled in, and the confidence values are updated. This procedure is repeated till all the pixels are filled.

IMPLEMENTATION

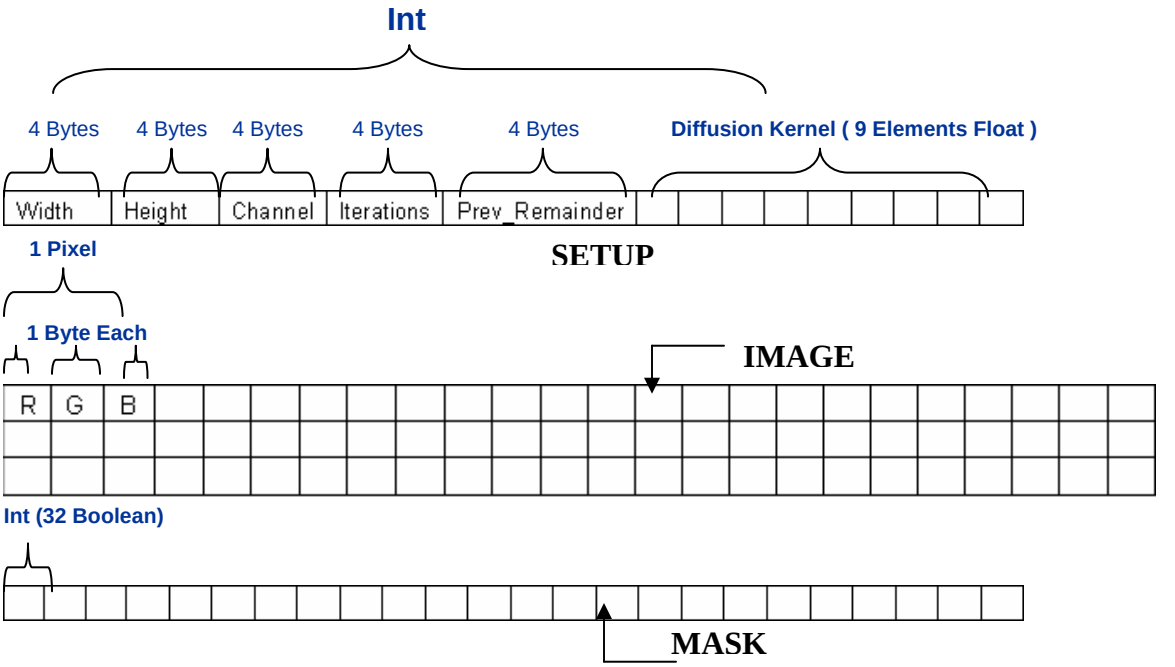


Project Data Flow

At the onset, the user is presented with a friendly graphical user interface, which allows the user to load the corrupted image into the GUI. The user can then draw on a mask of variable user-selected width over the corrupted regions. The user is then allowed to select the algorithm they wish to use to fix the corrupted region, the number of iterations they wish the algorithm to run for, as well as on-chip or external memory on the EVM.

If exemplar is chosen, the latter 2 choices are ignored, and the mask and the corrupted image are saved. The user is then directed to MATLAB, where they can use the exemplar algorithm to correct the image. After the exemplar algorithm is complete, the user can compare the original image, side by side, with the corrected image.

If diffusion is selected, a separate program (for which the source code is provided at the end of this report) is called with the appropriate arguments. Within this program, the corrupted image is read in as an unsigned char 1-dimensional contiguous array of RGB components for each pixel in the image. The mask is also read in and converted to an integer array of Boolean values where each 32-bit integer represents 32 pixels. Thus for example, an image of size 200x160, would have a pixel array containing $200*160*3=96000$ elements, and a mask array of $\frac{200*160}{32}=1000$ elements. These arrays along with other pertinent and useful information such as the dimensions of the image, and the number of iterations to be done on the image, are then collectively arranged into one massive chunk of contiguous memory, and then sent to the EVM. If it is the case that this chunk of memory is too large to fit in a single bank of EVM SDRAM, the chunk is split up into *n* components, where *n* is determined by dividing the total size of the memory chunk by the size of a single bank of SDRAM memory (4 megabytes).



Once a chunk is received by the EVM, the EVM disassembles the chunk. Variables such as width, height, number of iterations, pointers to the pixel and mask arrays are stored in global variables for use throughout the program. Once all these values have been obtained, the process of diffusion is begun.

The diffusion effect is obtained by a convolution of a 3x3 kernel $\begin{bmatrix} 1/8 & 1/8 & 1/8 \\ 1/8 & 0 & 1/8 \\ 1/8 & 1/8 & 1/8 \end{bmatrix}$ that averages the value of the surrounding pixels into the middle pixel. Depending on the user's memory selection in the GUI (on-chip or external), one of the 2 types of memory schemes is used.

If off-chip was selected, pixels that were being used for calculations were retrieved from external memory every time they were needed. The calculated pixel, was stored in a local variable in on-chip memory until the entire kernel had been traversed. This newly calculated pixel was then deposited back into external memory.

If on-chip was selected, a 57,600 bytes of the image was copied into on-chip memory, with an additional 600 bytes copied in for the Boolean mask data. Calculations were then performed on the data in on-chip memory and the results were all stored on-chip until *all* calculations on the pixels in on-chip memory were complete. Once completed, the results were transferred back to external memory, and new data was once again transferred in. This process of course was repeated for as many iterations as specified by the user.

Once diffusion is complete for that chunk, it is sent back to the PC. On the PC side, the corrected pixels are extracted from the returned memory chunk and are used to overwrite the old pixel data in the PC memory. If there existed more than one chunk, the exact process would be repeated on all the chunks until all the chunks have been processed. At this point, the new corrected pixels are saved to a file on the PC.

The user can then retrieve the corrected image and display it in the GUI.

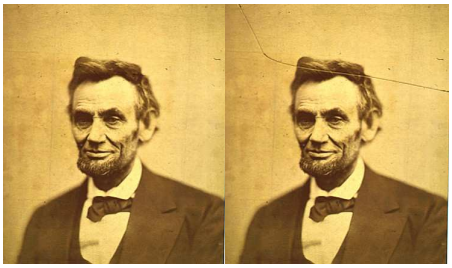
Memory Performance and Analysis:

As mentioned, 2 types of memory usage schemes were used. The speed of the overall algorithm differed based on the type of memory use, but also on the percentage of mask coverage of the image.

It was found that corrupted images with small mask coverage were actually inpainted quicker using external memory over on-chip memory than images with larger mask coverage. This stems from the fact that in the external memory scheme, no unnecessary paging from external memory to internal memory is done. Thus the additional overhead that comes with paging in data that does NOT need to operated on, is not present. However, with this method each pixel must be accessed in external memory 9 times since they are not cached in on-chip memory at all. This is true even if those pixels were used in an immediately previous calculation. Thus for larger mask coverage, more pixels have to be accessed 9 times without caching.

Conversely, for images with a large percentage of mask coverage, our on-chip scheme with DMA paging performs better than the external memory scheme. This is due to the fact that larger mask coverage implies more calculations. However, unlike with the external memory scheme where each needed pixel is accessed from external memory each time it is needed, our on-chip scheme guarantees that pixels from a prior calculation are already in fast on-chip memory, thus the time needed to access these pixels is 15 times less than if they were off-chip. Unfortunately, since our implementation sends the entire image to the EVM, including the non-corrupted regions, the additional overhead of paging in the non-corrupted regions significantly increases the time needed to process the image.

The idea that the additional time used by our DMA scheme is explicitly due to the additional unnecessary memory transfers that must be done on uncorrupted portions of the image is supported by the test case where the ever so popular Lincoln image is reduced to only the corrupted portion.

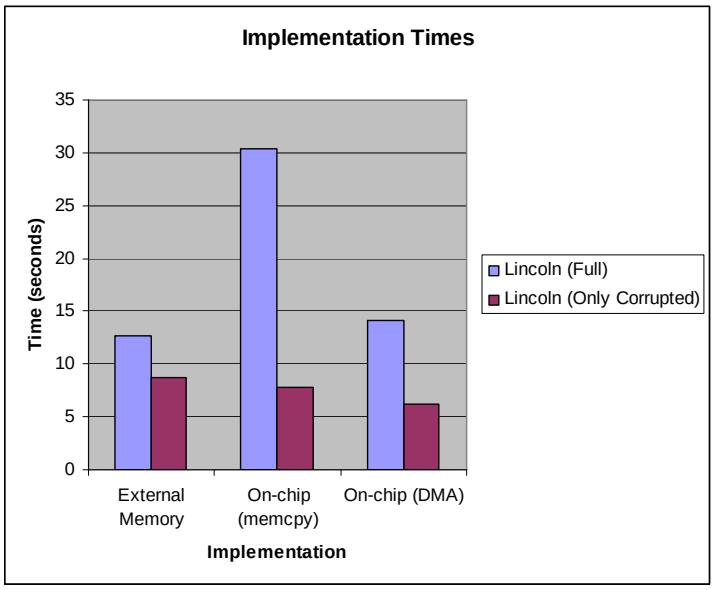


Full Lincoln Image (372x200)

Corrupted Region of Lincoln (372x160)

Image	Mask	External Memory	On-chip (memcpy)	On-chip (DMA)

	Coverage (%)			
Lincoln (Full Image)	1.19	12.67 seconds	30.34 seconds	14.19 seconds
Lincoln (Only Corrupted region)	3.476	8.775 seconds	7.812 seconds	6.25 seconds



Notice that when the image was reduced to only the corrupted region, DMA outperformed external memory by almost 3 seconds - an almost 30% increase.

	Function Name and Total Cycles for 100 iterations	
	diffusion	mem_transfer
External Memory	812816000	*Not used*
On-chip (memcpy)	758508451	220506049
On-chip (DMA)	538277980	274000

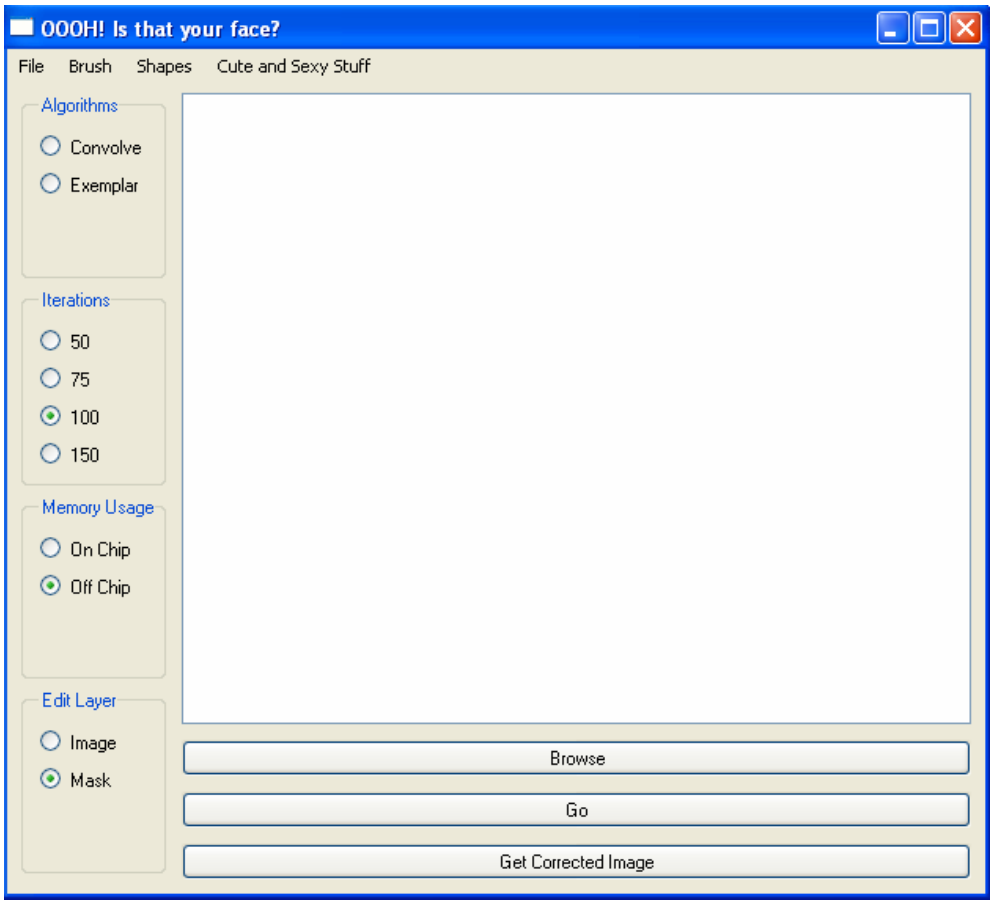
In cycles, even though the external memory implementation does not use the function mem_transfer at all, it still takes more cycles to perform the diffusion 100 times. We can also note that the additional time spent by the on-chip implementation that uses memcpy as opposed to the on-chip implementation that uses DMA, is solely a result of the additional cycles needed by the memcpy function. This is probably due to the fact that unlike the C library function memcpy, the DMA functionality on the EVM is a separate hardware entity and can execute efficiently without using the EVM CPU. Notice that if in both the case of the memcpy and the DMA method, if we subtract the total number of cycles needed for the memory transfer, we end up with approximately the same number. Thus indicating that the actual time needed to perform calculations once the memory is in on-chip memory, is practically equal.

On-chip (memcpy): 758508451 – 220506049 = 538002402 cycles
On-chip (DMA): 538277980 – 274000 = 538003980 cycles

LAB DEMO

Graphical User Interface (GUI):

The final version of the Graphical User Interface allows for the user to choose the algorithm, the number of iterations, whether or not to do the algorithm on or off the EVM. This GUI allows the user to implement everything within the same window.



Our graphical user interface allows the user to load the desired image into the program. The user can then select the corrupted regions manually using a brush. The user defines the brush size as well as the color and selects an area using it. The user may also corrupt the image further by choosing to edit the image layer. When the user has completed loading the corrupt image into the GUI, the user may edit the mask for the program to correct the image. The user also may chooses a desired number of iterations he feels he needs to complete this restoration. The user may also select whether the program is to be implemented on or off of the EVM. If the user doesn't know what any of the options mean, he can choose the algorithm only, as there are default presets in the GUI. The user selects an algorithm which will be the algorithm that performs the given task.

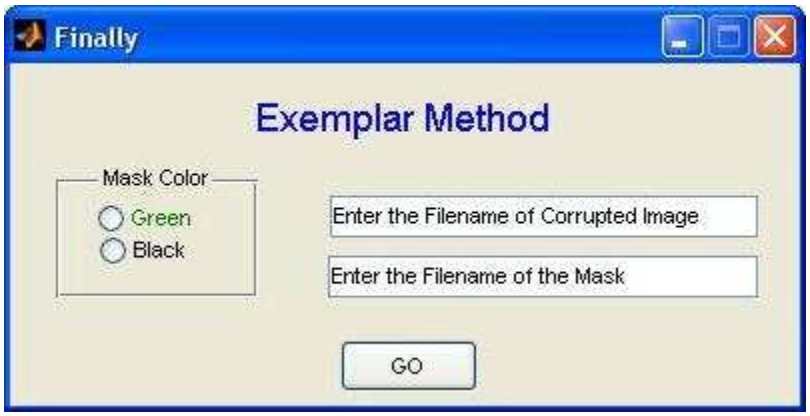
Once the region, number of iterations, and algorithm is selected, the selected region is sent from our GUI to the EVM.

If the corrupted region is too large, we will page portions of the image into the EVM, where the inpainting will similarly be performed. If paging occurs, then we will be careful to recognize that the inpainting done on each page must be consistent with the inpainting done on another page. For that reason, we will page overlapping segments of the image when we cannot bring the whole region onto the EVM. The inpainting process is iterated the number of

user selected times, but the user must remember that too much inpainting may be worse than just enough. Thus our inpainting process will proceed in steps, guided by the user.

After the program has finished correcting the image, the user presses the button labeled “Get Corrupted Image” to view the restored image in the same GUI window. If the user is not satisfied with the restored image, the user can use that image as the input to the program, select the corrupted area, and complete the algorithm using a user defined number of iterations. The user will need to decide whether more inpainting needs be done or whether the restoration is to their liking.

For the lab demo the GUI shown above was used for the Convolution method as well as beginning the Exemplar method. Due to drawbacks in time the GUI shown above did not include the implementation of the Exemplar method. The GUI for the Exemplar method, done in MATLAB is shown below.



The user will select the files that he wishes to input into the program. From the original GUI, the Exemplar method saves the image and the mask into specific files that are automatically inputted into the MATLAB GUI. If there is a change to be made, there is an option for the user to input a different file name in both the corrupted image and mask fields. After making sure the input files are correct, the user will select the color of the mask (either green or black). Finally, when the user presses the “GO” button, the exemplar method is implemented. After finishing this method, the user will have the chance of redoing the new inpainted image, by inputting it as the corrupted image and pressing “GO” again.

Actual Demo:

The demo showcased the capabilities and limitations of the convolution and exemplar-based method of inpainting. Performance was based on plausibility of correction, and processing time. Two algorithms were implemented to repair all nature of defects. Thin defects were repaired primarily using the convolution method while larger defects were repaired using the exemplar-based algorithm. This was demonstrated in the demo. The GUI was made using QT, a multiplatform GUI developer. The GUI linked all the components of the system together and allowed the user to manipulate that system. It allowed the user to load images, place masks over the corrupted regions, and run the repair process using either algorithm. Our GUI allowed the user to select not only the image and mask, but specify the algorithms, and the conditions the algorithms would be run under. This included the number of iterations on the convolution method, and whether the code was to be run off the chip or on the chip (though on-chip performance in most cases was faster).

When the corruption had a thickness of less than 9 pixels, the convolution method was faster, and performed just as well to the exemplar method. The convolution method worked by effectively bleeding the good region over the bad. As it is a diffusion of the non-corrupted region over the corrupted region, blurring would occur. This was not noticeable when the width of the corruption was less than 9 pixels. It became obvious with a 10-pixel mask. The

process took longer when we increased the number of diffusion iterations, increased the size of the image, and increased the size of the mask.

The exemplar method was showcased in MATLAB. It was able to perform just as well as the convolution method for thin (less than 9 pixels in width). Furthermore, it surpassed the convolution method, being able to deal with defects covering a wider area. What's more, it did not introduce blurring of any sort, and managed to linearly maintain geometrical properties of the image. However, its ability to generate plausible recoveries had limits.

With the exemplar method, performance was determined by quite a few factors. With regards to speed, the size of the image was the most important factor, and the area of the mask has the second greatest impact. Plausibility of repair was affected by the nature of the image surrounding the corrupted region. Like the convolution method, the inpainting starts somewhere on the border of the corruption and migrates inward. To inpaint block 'A', we select a block in the uncorrupted region that is surrounded by pixels most closely matching that of the surrounding pixels of block 'A'. We do this assuming a Markov Random Field distribution in the image.

The texture surrounding the corrupted region is perpetuated inwards. If the sample space (the non-corrupted region) in the image was small, then a reasonable match might not exist; the inpainted block may not be a plausible match. This would also occur if the texture to be inpainted (the texture bounding the corrupted region) did not exist in much of the rest of the picture; in general, the larger the sample size of a texture similar to that being inpainted, the better the result. Additionally, if there are different textures on different sides of the boundary, they will all be grown in to some degree. Defect removal happens best when the defection is bounded by a uniform texture. Then it will be covered by in a uniform manner. Otherwise, different textures may be grown in, though they will be inpainted in a plausible manner.

Thus when the corruption was bounded by a more or less uniform texture, the inpainting was excellent. The inpainting fit plausibly. However, when we had the boundary cut across more than one texture, the inpainting would grow all of the various textures in to some degree.

Demo Comment:

"How is this different from lab 3?"

Though very similar to the 18-551 Lab 3, the EVM portion of the diffusion algorithm differs slightly. First off, the diffusion algorithm uses convolution not correlation to create the diffusion effect, thus requiring a reversal of the diffusion kernel. In addition, because the quality of the resultant image is dependent on the number of iterations done successively on an image, it is important that the entire image be processed during each iteration, as opposed to only a portion of the image being processed 100 times, then another portion. This is to ensure uniform results, as well as ensure that portions that depend on other portions of the image are diffused uniformly across the chunk boundaries being paged into on-chip memory. Memory-wise, lab 3 used a fixed image size, and thus there was no additional overhead needed in each cycle to calculate how many lines of the image the on-chip memory could hold. In our project, we take into account that image size is variable, and thus before new information is transferred from external to internal memory, calculations must be done to determine how many lines of the image can be brought in based on the image's width.

Our first implementation of the diffusion algorithm was a lot like lab 3 part 1 in which pixels that were being used for calculations are fetched from external memory every time they were needed. The resultant pixel, was stored in a local variable in on-chip memory until the entire kernel had been traversed. The resultant pixel was then deposited back into external memory.

The implementations of our project where on-chip memory was used, were most similar to parts 3 and 4 of Lab 3. In lab 3 parts 3 and 4, a pre-calculated number of lines from a fixed size image was brought on chip using either a for-loop or a DMA transfer. Similarly, in the our implementations that used on-chip memory, a portion of the image was copied into on-chip memory using the standard C library function *memcpy* or the EVM's direct memory access (DMA) functions. Calculations were then performed on the data in on-chip memory and the results were all stored

on-chip until *all* calculations on the pixels in on-chip memory were complete. Once completed, the results were transferred back to external memory, and new data was once again transferred in.
Each inward and outward transfer was approximately 58,200 bytes, where 57,600 bytes (56.25 KB) is for the actual pixel data and 600 bytes is for the Boolean mask.

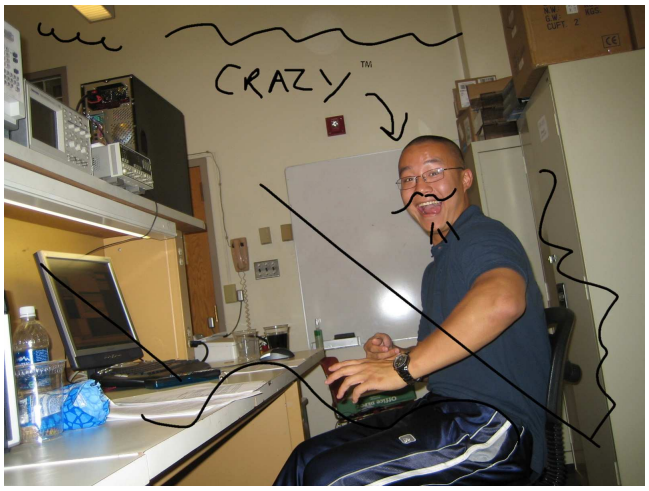
Finally, on the PC side, the EVM communication method was modified slightly to load multiple program files on the EVM depending on user input.

ANALYSIS AND CONCLUSION

The project showed much promise, though it was regrettably not explored to its full potential. The convolution method worked within reasonable time bounds for defections of a thin nature. However, the exemplar-based method was never implemented on the EVM. Its results in Matlab were quite impressive though. The results were dependent on the textures bordering the corruption and the nature of the sample space around the corrupted region. But in many cases, it was able to plausibly inpaint a large region of corruption.

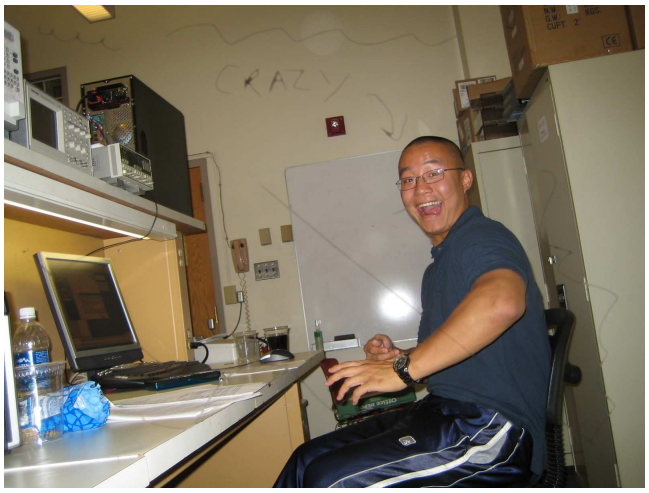
The project would have been more successful if the exemplar algorithm was implemented on the EVM. For pictures of 1200x 1600 dimensions, the lab computers actually ran out of virtual memory. For pictures of size 480 x 640, the algorithm ran for a few minutes. This speed would most likely be reduced if the exemplar-based algorithm was based in C. This is a topic for further work.

Hence, we managed to find ways to effectively inpaint thin and thick regions in various photographs. The finished products repaired the corrupted image so that the inpainting was not noticeable. The primary task was accomplished. However, although we explored the convolution method on the EVM, these tests have yet to be fully implemented on the EVM. This remains for further study.



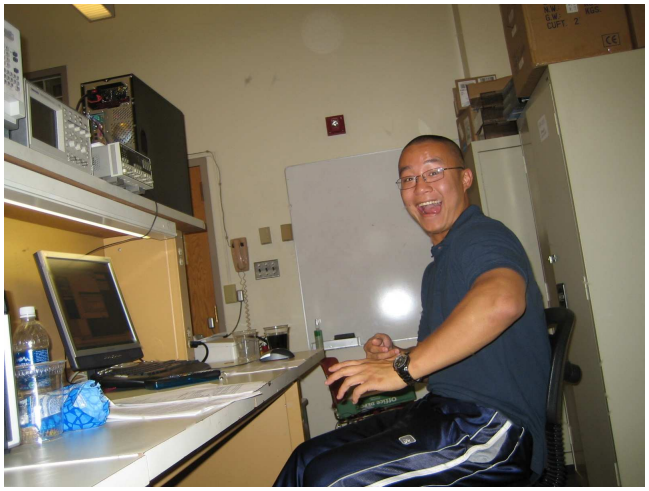
1 iteration


1 iteration



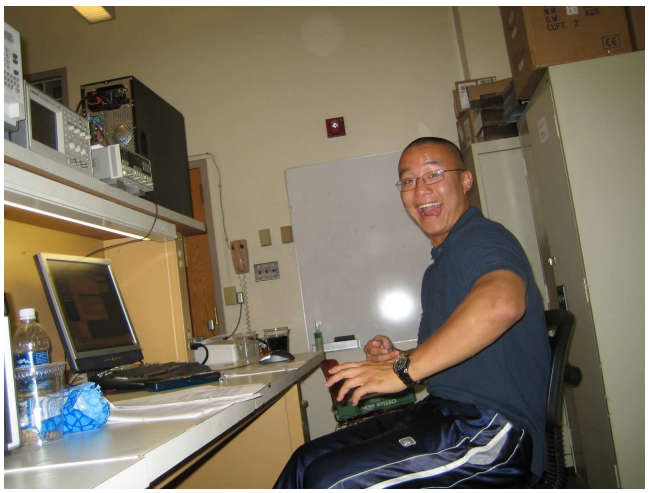
25 iterations

25 iterations



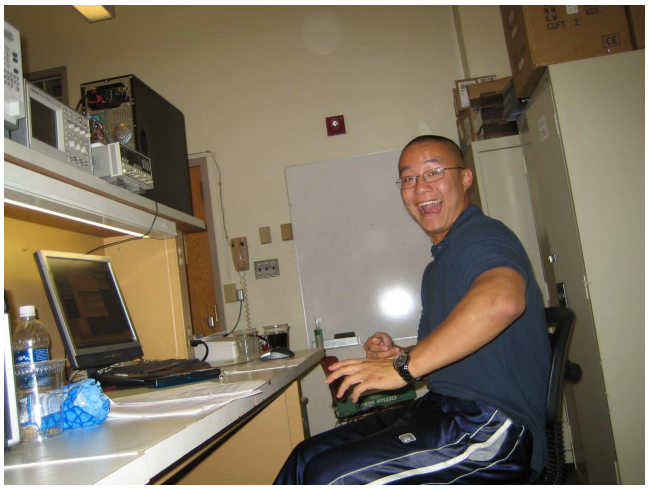
50 iterations

50 iterations



75 iterations

75 iterations



100 iterations

100 iterations

Further Work:

There are a few possibilities for further work; implementing the exemplar method on the EVM, optimizing the algorithm for better speed performance, and implementing the exemplar method so that it can inpaint isotopes not just linearly, but in a curvature driven fashion.

The exemplar method is quite powerful. However, it should be adapted to allow more flexible masks; the system should allow specification of what textures grow in where, instead of having all textures grow inward. Thus the algorithm should be adapted so that we have more control over what kinds of textures are grown. The current implementation also looks for textures using a sampling window size of 9x9 (allowing capture of most sized patterns). However, some patterns are larger than 9 pixels across. The algorithm should allow the size of the sampling window to be adjusted, in order to capture these larger patterns. The results generated using the current implementation is quite reasonable.

Time is a big issue. The implementation on Matlab took a few minutes to process a 480x640 picture with a mask of approximately 30% of the image. Optimizations should be explored both in the implementation of the algorithms both on the pc and the EVM side. The EVM should be maximally utilized so that as many of the adders, multipliers and registers are occupied at any running time.

The exemplar method has the advantage of finding edges (differing bands of color) and growing these areas first, preserving the edges linearly, respecting the geometry of the objects in the picture. However, it only grows the edges linearly. There are variational approaches that are curvature driven, meaning the diffusion process looks at the way edges are curving and inpainting appropriately. Such a method should be explored for the exemplar case so that it grows much more accurately.

REFERENCES

[1] [Bertalmio 1] M. Bertalmio, A.L. Bertozzi, and G. Sapiro. “Navier-Stokes, Fluid Dynamics, and Image and Video Inpainting” In *Proc ICCV 2001,* pp. 1335-1362, IEEe CS Press

[2] [Chan 1] T. Chan and J. Shen. “Mathematical Models for Local Deterministic Inpaintings” Technical Report CAM 00-11, Image Processing Research Group, UCLA 2000

[3] [Chan 2] T. Chan and J.Shen “Non-Texture Inpainting by Curvature-Driven Diffusions (CCD).” Technial Report CAM 00-35, Image Processing Research Group, UCLA 2000

[4] [Geman 1] S.Geman and D.Geman, Stochastic Relaxation, Gibbs Distributions and the Bayseian Restoration of Images, IEEE Trans. Pattern Anal. Machine Intell., 6, 721-741 (1984)

[5] [Knutsson 1] Knutsson, H. and Westin, C-F.: Normalized and differential convolution: Methods for Interpolation and Filtering of incomplete and uncertain data.*Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition*, 1993, 515-523.

[6] [Oliveira 1] M. Oliveira, B. Bowen, R. McKenna, and Y. S. Chang. “Fast Digital Image Inpainting” inn Proce. VIIP 2001, pp. 261-266, [CITY]:[PUB],2001

[7] [Sapiro 1] Guillermo Sapiro, “Image Inpainting”, SCIAM News, Volume 35, #4

[8] Alexei A. Efros and Thomas K. Leung “Texture Synthesis by Non-parametric Sampling” IEEE International Conference on Computer Vision, Corfu, Greece, September 1999

[9] Tony F. Chan and JianHong Shen “Mathematical Models for Local Non-texture Inpaintings” SIAM J. Appl. Math Vol. 62 No.3 pp 1019-1043 2002 Society for Industrial and Applied Mathematics

[10] Original Oliveira Code: /afs/ece/usr/mdavey/Public/Fast_Inpainting_demo_files.zip