Image Inpainter
COMP9517 Major Project

Willy Mai - wmai@cse.unsw.edu.au Xi Chen - xic@cse.unsw.edu.au

19th October 2010

Abstract

The focus of this project is to develop a flexible, user-friendly and simple open source tool for image Inpainting. A number of image Inpainting algorithms from research papers have been implemented. The variety of algorithms will give the flexibility and usability of the software, while the GUI user interface provides quick and easy access to parameter adjustments and other functions. Each algorithm will be discussed in detail along with implementation details in this report. We will conclude with some possible extensions and ways the software can be improved.

Introduction

Image inpainting is the art and science behind re-constructing parts of an image in a visually undetectable way. Inpainting has been an art form for many centuries, being carried out by a skilled image restoration artist. Research in partnership with these artists and the general increase in computing power gave rise to sophisticated algorithms for recovering the lost of corrupt portions of an image.

While image inpainting is related to noise removal, and sometimes algorithms share a majority of their ideas, image inpainting is fundamentally a different problem to noise removal. Noise often carry some sort of information about their underlying data. For example, additive noise contains the original image data, but with some amount of noise "added" to the image. Image inpainting, on the other hand, the "lost" regions contain absolutely no information relating to the original data.

The algorithms we have implemented (and will be discussed in more detail below) are:

- Gaussian Pyramid Blur
- Fast Digital Image Inpainting [1]
- Bertalmio [2]
- A. Telea - Fast Marching Method [3]

- Navier-Stokes, Fluid Dynamics, and Image and Video Inpainting [4]
- Exemplar-Based Image Inpainting [5]
- Image Inpainting Based on Local Optimisation [7]

Goal

Our goal is through research & implementation, to gain a full understanding of various inpainting algorithms. We also, along with understanding, aim to produce a versatile tool for image inpainting.

Image Separation

Our program offers 4 methods of image separation (only used for algorithms that assume greyscale, and require separation for colored images).

- Greyscale: Images are separated into RGB, then each of the colors are multiplied by a brightness factor, and the results added, to produce a greyscale image. The factors used were: $B = B * 0.0114$, $G = G * 0.587$, $R = R * 0.299$. Colored images will produce greyscale results with this method of separation.
- Desaturate: Image RGB is simply averaged, to produce a greyscale image. Colored images will produce greyscale results with this method of separation.
- RGB Separation: The R, G and B channels of the image are treated as separate greyscale images, the inpainting algorithm is run separately for all 3 channels. The reults are then combined to form the resulting color image.
- Vector Space Separation:
  The image is treated as 3 separate images, the length ρ, and the angles Ω and Ψ.

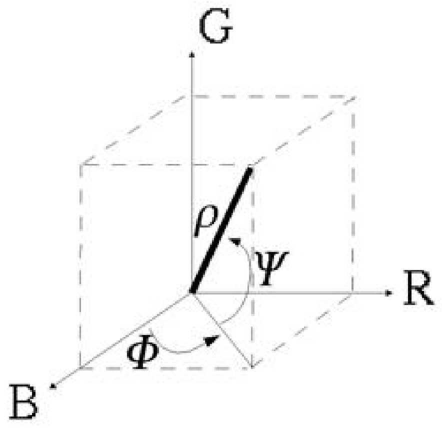


Figure 1: Relationship between vector space & RGB color space. Image Source: [2]

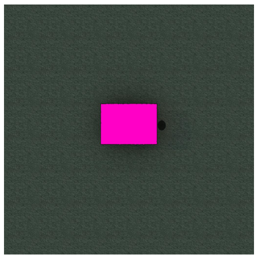
Details: Gaussian Pyramid Blur

Overview

The gaussian pyramid blur is a simple algorithm we have come up with, which aims to fill in large areas quickly.

This algorithm can be used for applications where continuity is important around the borders of the inpainting region, but not so important around the middle of the regions. One example of such an application would be in texturing for a 3D computer model.

Instead of leaving the parts of the model texture that are ”not used” in the UV map of the model to be black, we instead quickly inpaint this region. When the texture is resized, the black does not ”creep into” the image regions, and create black texturing ”seams” in the model render.


(a) Original 1024x1024 ground texture.

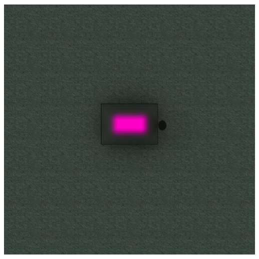
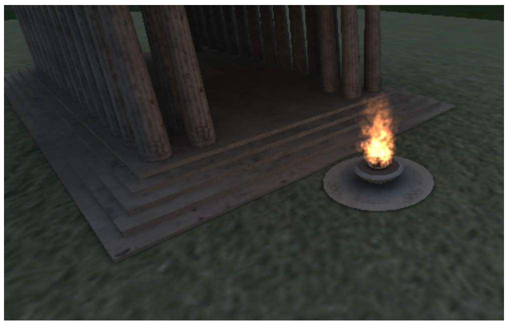
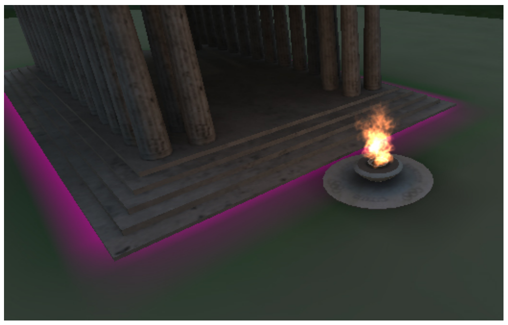

(b) Ground texture with unused region inpainted.

Figure 2: Textures.


(a) Model render using original 1024x1024 ground texture.


(b) Model render using ground texture resized to 64x64. Note the unused magenta regions ”invading” the used regions, due to downscaling errors and interpolation.

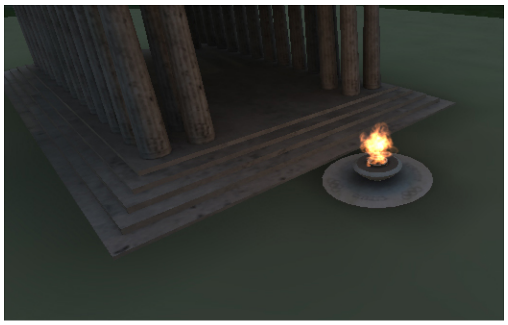

(c) Model render using inpainted ground texture resized to 64x64. Note that the ”invading” regions no longer appear visible, due to the inpainting.

Figure 3: Model render results.

Algorithm

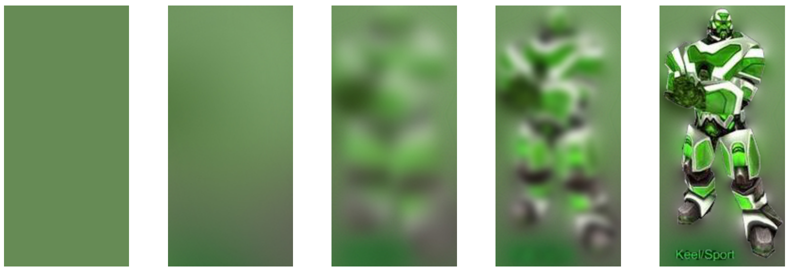
Conceptually, the algorithm treats the image mask Ω as the alpha channel to image I, and builds an image pyramid. It then overlays and combines this image pyramid according to the following formula:

$$R_{n+1} = (R_n(1-\alpha_n) + I_n\alpha_n), \forall n \in N$$

where N is the list of images in the pyramid, I_n is the n^{th} smallest image octave in the pyramid, R_n is the intermediate result image at stage n, α_n is the alpha channel image for n.

Thus, we start with the smallest image I_0 (1x1 pixels) in the pyramid, up-size it back to the original size of the image. To this, we add the next image in the pyramid (2x2), I_1. If a pixel in this next image which has an alpha value of 1, it will "replace" the underlying pixel. This is the common color "transparency" formula. Then, we add the next image, and so on, until we finally add the original image.

This process will produce less blurred pixels are the mask region borders. It will use more blurred versions of the image for the pixels further away from the region borders.



(a) R_0 (b) R_1 (c) R_2 (d) R_3 (e) R_4

Figure 4: Intermediate stages of R_n.

Implementation

The algorithm pseudocode for our implementation:

```
For each octave $o = 1$ to $O$
    Apply simple blur kernel to the original source image with a radius r = k + (O-o)j
    Where
        $k$ = the base radius.
        $o$ = the octave number.
        $j$ = is the factor by which the radius increases by.
    Use "transparency" formula to alpha-blend blurred image onto current result.
```

Where the final octave $o = O$ is the original image, unblurred. The image mask is treated as the alpha channel, and is thus, blurred with the image.

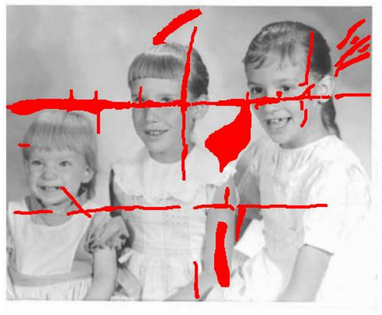
Since OpenCV doesnt have a blur function that takes in a mask, to achieve a estimated alpha channel blur effect, this following algorithm is used for blurring:

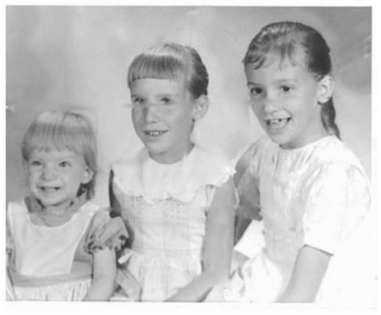
```
Set the regions to be inpainted in the original image to zero.
Blur the 3-color image.
Blur the image alpha channel (the mask image).
Normalize the color vector using the alpha value:
```

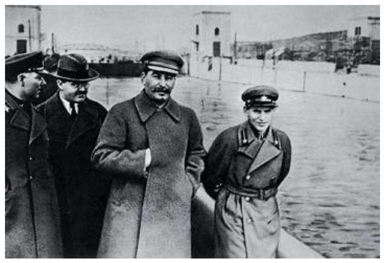
$$I(x,y) = \begin{cases} \frac{I(x,y)}{\alpha(x,y)} \text{if } \alpha(x,y) > 0 \\ 0 \text{ otherwise} \end{cases}$$

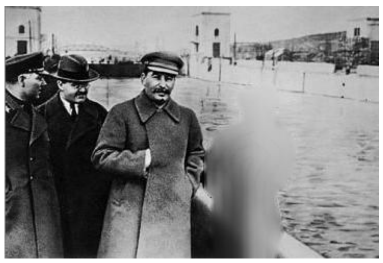
For our implementation, we used the basic square blurring kernel, which is just an average over a pixel of the pixels in the square window around it. This has been chosen over gaussian for two reasons. First, it is faster than gaussian. Secondly, it reduces the error introduced by our alpha channel blur estimation above.

Results


(a) image 1


(b) result 1


(a) image 2


(b) result 2

Strengths & Limitations

This algorithm is very fast, even for large areas to be filled in. It also preserves good visual continuity near the borders of the inpainting region; the images seem to "continue" visually into the inpainted region border. However, the algorithm does not preserve strong structure information, nor will reproduce any texturing of the target region. The middle pixels of the inpainting regions are very blurred, and only have general color information.

Conclusion

The algorithm successfully fills in large areas very quickly. Although the filled in region is badly blurred, the general color information is retained. This could be useful for applications where one needs to fill in "useless" areas of the image quickly, and is only concerned about the visual continuity of the border of the area, and not too fussed about the contents of the inpainted region, such as the texturing example given.

Although our implementation could be improved with a more proper implementation of masked blur, the estimation algorithm used is quite sufficient.

While a more sophisticated fast inpainting algorithm such as navier-stokes or fast marching method could quite possibly do this job better and faster, this algorithm provides a very solid starting point for a naive algorithm.

Details: Fast Digital Image Inpainting

Overview

This paper provides a very simple kernel based inpainting algorithm designed to be faster than [2], while being much easier to implement. It is an iterative algorithm which repeatedly convolves the area to be inpainted by a filter with a zero centre element to propagate information from outside of the inpainting area to the inside. Its disadvantage lies in its poor transmission of both structural and textural information, providing results that appear similar to blurring. The algorithm is intended for the inpainting of narrow strips for removing thin lines and is stated as such by the authors.

Algorithm

The algorithm works by repeatedly convolving the area to be inpainted Ω by a weighted average kernel which only considers contribution from neighbouring pixels, that is, the centre pixel has zero weight. The paper presents two example kernels:

$$\begin{bmatrix} a & b & a \\ b & 0 & b \\ a & b & a \end{bmatrix} \begin{bmatrix} c & c & c \\ c & 0 & c \\ c & c & c \end{bmatrix}$$

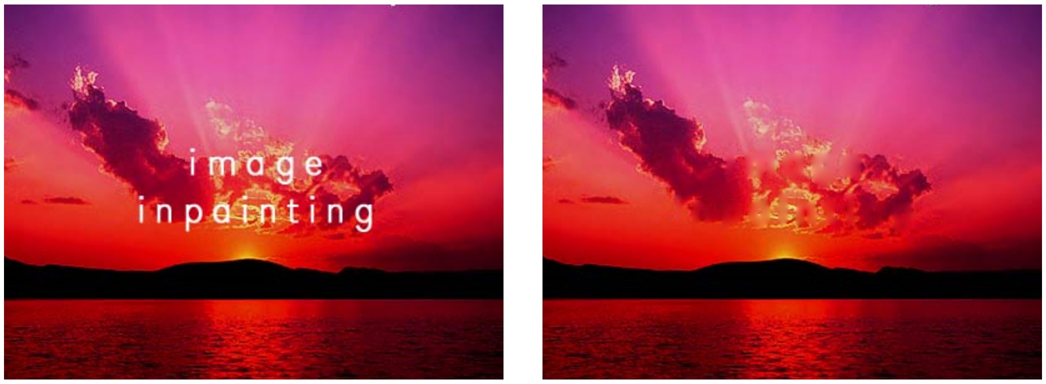
where $a = 0.073235, b = 0.176765, c = 0.125$. Eventually, the algorithm will converge and inpainted pixels will stabilise to the weighted average of their neighbour pixels.

Implementation

The algorithm is very simple to implement. The following is our implementation of the algorithm in OpenCV:

```
// Constants
#define FDII_A 0.073235
#define FDII_B 0.176765
#define FDII_C 0.125

// Kernel
double FDIIKernel[][9] = {

    { FDII_A, FDII_B, FDII_A,
      FDII_B,    0.0, FDII_B,
      FDII_A, FDII_B, FDII_A },

    { FDII_C, FDII_C, FDII_C,
      FDII_C,    0.0, FDII_C,
      FDII_C, FDII_C, FDII_C },

};

// Main algorithm:
// Repeatedly apply kernel
for(int i = 0; i < iter; i++) {
    cvFilter2D(out, out, kern);
    cvCopy(image, out, mask2);
}
```

Results



(a) image (b) result

Strengths & Limitations

The algorithm's strength lies in its simplicity, being very easy to understand and implement. Although the algorithm is simple to implement, it gives decent results. This algorithm is very fast, and is able to preserve general color information and some low-frequency texture.

The algorithm however, produces results which are quite blurry. More importantly, the visual continuity at the image edges seem to be pretty bad; if one looks enough,the visual discontinuity at the region boundaries can be quite easily seen.

Conclusion

This algorithm is the simplest inpainting algorithm we have come across in our research, requiring only repeated image convolution. While it probably isn't the best at producing visually hard to detect inpainting results, it does give a solid starting point for such a simple algorithm.

Details: Image Inpainting (SIGGRAPH 2000)

Overview

This algorithm is based on the paper [2]. Originally published in SIGGRAPH 2000, it is referenced very extensively, and inspired a whole class of algorithms with its ideas. The algorithm is based on research into inpainting methods used by skilled artists.

The algorithm iteratively propagates image information along the image isophotes (the direction of contours of the image with similar intensity). This is done by numerically solving the partial differential equation:

$$\frac{\partial I}{\partial t} = \nabla^{\perp} I \cdot \nabla \Delta I$$

for image intensity I inside the region to be filled. The goal is to solve this equation to a steady state, such that

$$\nabla^{\perp} I \cdot \nabla \Delta I = 0$$

which means the image intensities are constant in the direction of image isophotes.

Algorithm

The algorithm is divided into two parts: the inpainting steps and the anisotropic diffusion steps. During the inpainting steps, the image is updated iteratively using the formula:

$$I^{n+1}(i,j) = I^{n}(i,j) + \Delta t I_t^n(i,j), \forall (i,j) \in \Omega$$

Where n is the inpainting step "time", (i,j) are pixel co-ordinates, δt is the "climb rate" or "rate of improvement", and $I_n^t(i,j)$ stands for the update of the image at time n. This is only run for pixels inside the region Ω, the region to be inpainted.

The update of the image is given by:

$$I_n^t(i,j) = \delta \vec{L}^n(i,j) \cdot \vec{N}^n(i,j)$$

Where $\delta \vec{L}^n(i,j)$ represents the change in the image laplacian, $L^n(i,j)$, and $\vec{N}^n(i,j)$ is the direction of the isophote. This effectively projects the gradient of the laplacian onto the normal of the isophote, which is a calculation of the change of information along the isophote.

The anisotropic diffusion equation is described in the paper [2] as follows:

$$\frac{\partial I}{\partial t}(x,y,t) = g_{\epsilon}(x,y)\kappa(x,y,t)|\nabla I(x,y,t)|, \forall (x,y) \in \Omega^{\epsilon}$$

where Ω^{ϵ} is a dilation of the inpainting region Ω with radius ϵ, κ is the Euclidean curvature of the isophotes of I, and $g_{\epsilon}(x,y)$ in Ω^{ϵ} such that $g_{\epsilon}(x,y) = 0$ in $\partial\Omega^{\epsilon}$, and $g_{\epsilon}(x,y) = 1$ in Ω.

Implementation

The discretization of the above image update formula is given as follows (from the paper [2]):

$$I_t^n(i,j) = \left(\delta \vec{L}^n(i,j) \cdot \hat{\vec{N}}(i,j,n)\right)|\nabla I^n(i,j)|,$$

$$\delta \vec{L}^n(i,j) := (L^n(i+1,j) - L^n(i-1,j), L^n(i,j+1) - L^n(i,j-1)),$$

$$L^n(i,j) = I_{xx}^n(i,j) + I_{yy}^n(i,j),$$

$$\hat{\vec{N}}(i,j,n) = \frac{\vec{N}(i,j,n)}{|\vec{N}(i,j,n)|} := \frac{(-I_y^n(i,j), I_x^n(i,j))}{\sqrt{(I_x^n(i,j))^2 + (I_y^n(i,j))^2}},$$

$$\beta^n(i,j) = \delta\vec{L}^n(i,j) \cdot \hat{\vec{N}}(i,j,n)$$

and

$$|\nabla I^n(i,j)| = \begin{cases} \sqrt{(I_{xbm}^n)^2 + (I_{xfM}^n)^2 + (I_{ybm}^n)^2 + (I_{yfM}^n)^2}, & \text{when } \beta^n > 0 \\ \sqrt{(I_{xbM}^n)^2 + (I_{xfm}^n)^2 + (I_{ybM}^n)^2 + (I_{yfm}^n)^2}, & \text{when } \beta^n < 0 \end{cases}$$

Where the subindex b denotes backwards difference (this pixel minus the previous pixel), and f (the next pixel minus this pixel). The subindexes m and M mean minimum or maximum between the derivative and zero respectively.

L^n is the lagragian of the image, an estimation of the "smoothness", and is in our implementation estimated using the simple lagragian kernel:

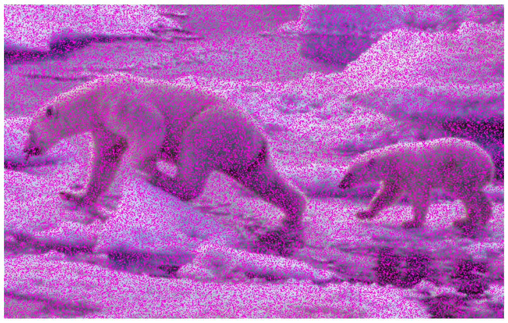
$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

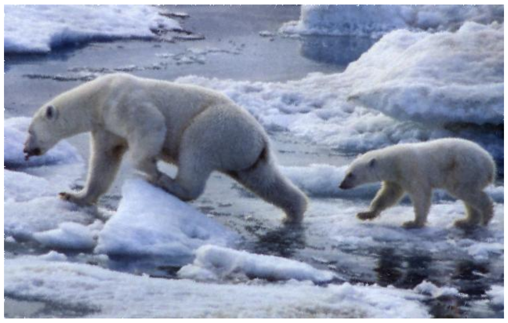
The isophote norm is estimated by first getting the image gradient, then rotating the gradient vector by 90 degrees. The image gradient is implemented in our algorithm using simple central differences. If the central differences yield a 0-lengthed vector, then the radius of the central differences is increased until the algorithm finds a suitable gradient, pr hits a maximum radius, in which case it just returns the 0-lengthed vector as the gradient.

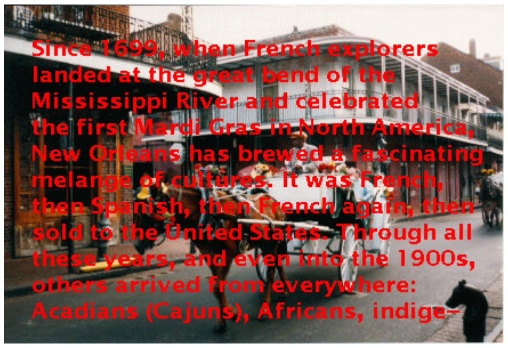
The algorithm estimates the dot product using a slope-limited isophote norm, in order to drastically improve the numerical stability of the algorithm. It breaks down the isophote vector into two components: the direction and the length. The length is then estimated using the mentioned slope-limited equation above.

Our implementation of anisotropic diffusion uses a refactored version of the code in from [6].

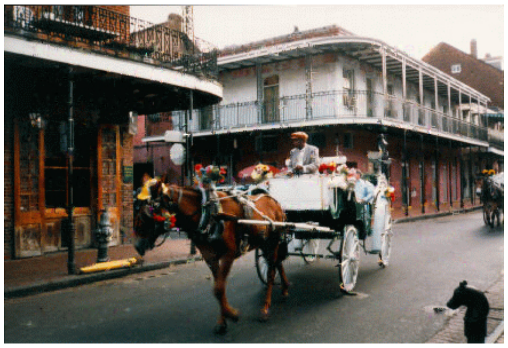
Results

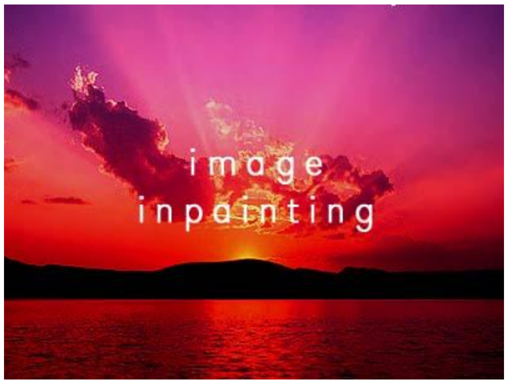

(a) image 1


(b) result 1

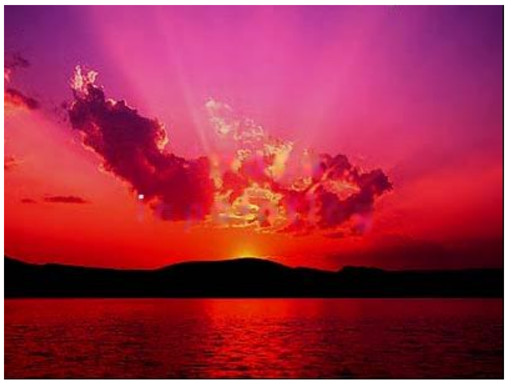

Since 1699, when French explorers landed at the great bend of the Mississippi River and celebrated the first Mardi Gras in North America, New Orleans has brewed a fascinating melange of cultures. It was French, then Spanish, then French again, then sold to the United States. Through all these years, and even into the 1900s, others arrived from everywhere: Acadians (Cajuns), Africans, indige-

(a) image 2


(b) result 2



(a) image 3


(b) result 3

Strengths & Limitations

This algorithm produces very good structural results. The image structure information (lines, curvesetc) are very well interpolated / extrapolated inside the region, better than any other algorithm we have came upon in our research so far.

However, this algorithm is numerically unstable, and very slow to run. The improvement rate parameter must be carefully tweaked; too small and the algorithm takes forever to run, too large and the algorithm numerically explodes. The algorithm can easily take many hours to run on large images or large inpainting regions.

The algorithm also does not produce texture information. So when applied to large regions, it will preserve the structural information, but still suffer from inevitable blurring that all algorithms of the PDE class (that solve to some extent the above partial differential equation in order to inpaint the image) suffer from.

However, this algorithm does not make use of any image patch information, so is able to recover image data from very limited border region information (such as the polar bear noise example above).

This algorithm would be most suited to images with a lot of structure, but not a lot of stochastic texture. The inpainting region is most suited to this algorithm when it is disconnected and regions have small radii.

Conclusion

While this algorithm is only able to provide results for areas with small radii, and takes a very long time to run, and also requires careful adjustments to parameters, the results it produces can be absolutely fantastic.

This algorithm is the building block basis of almost every modern image inpianting algorithm, and is referenced by nearly every image inpainting paper weve come across in our research. By understanding and implementing this paper, we have gained a very strong understanding of the very basic starting ideas, and we are able to see how these ideas have progressed through the last decade from this starting point.

Details: Image Inpainting Technique Based on the Fast Marching Method

Overview

This algorithm developed by Alexandru Telea is based on the PDE-based inpainting method in [2]. The algorithm attempts to improve on the speed of PDE-based inpainting, which requires iterative numerical methods and complicated anisotropic blurring. The improvement in speed relies on estimating the value of a pixel to be inpainted based on a neighbourhood of known pixels around it. In this method, each pixel is only inpainted once, which is a large improvement over iterative methods, which may run for many iterations before converging and are known for numerical stability issues.

The algorithm describes a method of estimating the intensity of a pixel on the inpainting boundary $\delta\Omega$ based on the known neighbourhood of that pixel. Given this method, the Fast Marching algorithm is used to determine the order in which border pixels are inpainted - that is, from the least distance to known pixels to greatest. By inpainting in this way, the inpainting area Ω reduces in size by $\delta\Omega$ as the algorithm progresses.

Algorithm

The algorithm uses the Fast Marching algorithm to solve the Eikonal equation:

$$|\nabla T| = 1 \text{ on } \Omega, T = 0 \text{ on } \delta\Omega$$

to ensure that always select the pixels closest to the known area of the image first. The level sets of T correspond to boundary $\delta\Omega$ as Ω reduces in size. The Fast Marching algorithm is described in more detail in original paper.

Given that we have selected a pixel p to inpaint, the intensity of p is given by a function of all points $q \in B(p)$, where $B(p)$ are all pixels within a fixed radius ϵ to p which have a known intensity (the 'neighbourhood' of p):

$$I(p) = \frac{\sum_{q \in B(p)} w(p,q)[I(q) + \nabla I(q)(p-q)]}{\sum_{q \in B(p)} w(p,q)}$$

where the image gradient $\nabla I(q)$ is calculated using central differences and $w(p,q)$ is the following normalised weighting function:

$$w(p,q) = \text{dir}(p,q) \cdot \text{dst}(p,q) \cdot \text{lev}(p,q)$$

$$\begin{aligned} \text{dir}(p,q) &= \frac{p-q}{||p-q||} \cdot \nabla T(p) \\ \text{dst}(p,q) &= \frac{1}{||p-q||^2} \\ \text{lev}(p,q) &= \frac{1}{1 + |T(p) - T(q)|} \end{aligned}$$

with a directional component 'dir', geometric distance component 'dst' and level set component 'lev'. The weighting function gives more weighting to pixels along the image gradient normal, pixels closer to p and pixels closer to the isoline of p.

Implementation

Implementation details and optimisation are fully discussed in the original paper. Our implementation is provided by OpenCV in the function 'cvInpaint'.

Navier-Stokes, Fluid Dynamics, and Image and Video Inpainting

Overview

This paper reveals that the PDE presented in [2] is analogous to the problem of convection in an incompressible fluid from the field of fluid dynamics. They draw expertise from the field of fluid dynamics to yield a solution to the problem of image inpainting based on the Navier-Stokes equations.

Algorithm

The PDE in [2] can be solved with the same techniques used to solve fluid flow in 2 dimensions.

There exists known equations [6] for solving fluid flow for a rectangular region, given boundary conditions, that is, given an empty rectangular region with flow information on the boundary, we can extrapolate the eventual convergent state of the fluid flow. The paper provides ways to generate the boundary information required by the fluid equations from the context of image inpainting and the information at the boundary of the inpainting region.

From the solution for a rectangular inpainting region, the paper presents a solution for arbitrary mask shapes by first obtaining the smallest rectangular region around the inpainting area Ω and using that as the new inpainting area $\bar{\Omega}$. The pixels in the difference of the areas, $\Omega/\bar{\Omega}$ is assigned back to $\bar{\Omega}$ at each iteration, forcing the values of $\Omega/\bar{\Omega}$.

Implementation

Our implementation is provided by OpenCV in the function 'cvInpaint'.

Conclusion

The paper formulates the solution to the PDE in [2] as a physics computation of the solution to a system of fluid flow in 2D. From testing of the OpenCV implementation of the algorithm, we deduce that it is very fast compared to the original algorithm presented in [2]. The algorithm itself, however, is founded on the mathematics of fluid dynamics and this poses a significant problem to anyone attempting to implement it.

Details: Exemplar Based Image Inpainting

Overview

This algorithm is based on the paper [3]. Previous texture synthesis image inpainting algorithms require large amounts of user interaction. The key observations behind this algorithm are that:

1. Exemplar-based synthesis suffices. This means that exemplar-based texture synthesis does very well to preserve, propagate and extend linear image structure. Exemplar-based texture synthesis is based around finding patches in the rest of the image which are similar to the known parts of the image patch to be filled, and then copying the missing information over. This mechanism will do well to preserve and extend and image isophotes we had in the known parts of our target patch.

2. Filling order is critical. This means that the order in which exemplar-based synthesis is carried out has a large impact on the resulting quality.

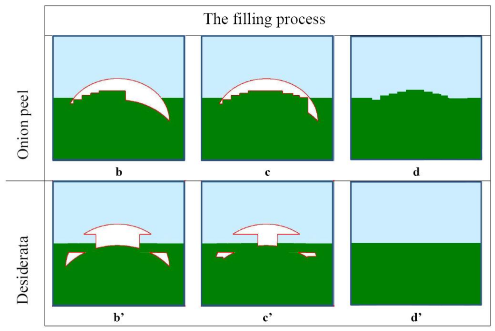


Figure 5: Example showing the importance of filling order. Image source: [3]

This leads to the exemplar-based inpainting algorithm. The algorithm first estimates the importance, or priority of our current boundary pixels of the region to be inpainted, which will respect image isophotes. It then finds the border pixel with the largest priority. It then finds a similar patch from the source regions (the regions not covered by the mask), and copies the missing information over.

This results in an algorithm that has the advantages of both texture-synthesis based methods (preserves and extends texture) and Partial Differential Equation (PDE) based methods (which preserve and extend structure).

Algorithm

The algorithm first needs to be able to estimate border pixel priorities. Given a patch $\Psi_{\vec{p}}$ centered at the point $\vec{p}$ for some $\vec{p} \in \delta\Omega$, the algorithm defines its priority $P(\vec{p})$ as:

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

The terms are defined as follows:

$$C(\vec{p}) = \frac{\sum_{\vec{q} \in \Psi_{\vec{p}} \cap (I - \Omega)} C(\vec{q})}{|\Psi_{\vec{p}}|},$$

$$D(\vec{p}) = \frac{|\nabla I_{\vec{p}}^{\perp} \cdot \vec{n}_{\vec{p}}|}{\alpha}$$

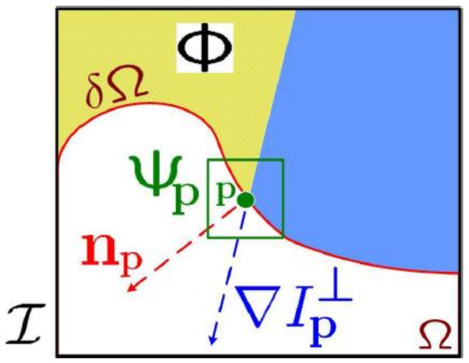


Figure 6: Notation diagram. Given patch $\Psi_{\vec{p}}$, $\vec{n}_{\vec{p}}$ is the normal to the contour $\delta\Omega$ of the target region Ω and $\nabla I_{\vec{p}}^{\perp}$ is the isophote at point $\vec{p}$. The entire image is denoted with I. Image and caption source: [3].

Where $|\Psi_{\vec{p}}|$ is the area of $\Psi_{\vec{p}}$, α is a normalization factor (e.g. 255 for a greyscale image), $\vec{n}_{\vec{p}}$ is a unit vector orthogonal to the front of $\delta\Omega$ in the point $\vec{p}$, and $\perp$ denotes the orthogonal operator. The priority $P(\vec{p})$ is computed for every border patch, with distinct patches for each pixel on the boundary of the target region. Paragraph source: [3].

The term $C(\vec{p})$ is referred to as the "confidence" term, and will model how confident the algorithm is about a patch. This confidence value will be designed such that it gives high confidence values to patches which are almost filled; as the exemplar based patch matching has a high chance of finding the correct patch. As the image progresses from the border of the region into the center areas of the region, the confidence will decay, reflecting that the algorithm is less sure about the center areas of the inpainting region than the areas close to the border.

The term $D(\vec{p})$ is referred to as the "data" term. This will model the amount of structure information that this patch contains, and how relevant is this structure to our patch. This term will boost the priority of a patch that an isophote "flows" into. This will encourage strong linear image structure to be inpainted first, thus not be overridden by patches with less structure.

After the priorities for all border pixel have been detemined, the border patch centered at the border pixel with the highest priority will be chosen to be filled. Call this chosen border patch $\Psi_{\hat{\vec{p}}}$.

The algorithm then searches in the source region the patch which is the most similar to $\Psi_{\hat{\vec{p}}}$, or in other words, searches in the source region for the exemplar patch $\Psi_{\hat{\vec{q}}}$:

$$\Psi_{\hat{\vec{q}}} = \arg \min_{\Psi_{\vec{q}} \in \Phi} \left(d(\Psi_{\hat{\vec{p}}}, \Psi_{\vec{q}}) \right)$$

Distance $d(\Psi_{\hat{\vec{a}}}, \Psi_{\vec{b}})$ is measured using simple sum of squared differences (SSD) over each patch pixel. Once we have found the exemplar $\Psi_{\hat{\vec{q}}}$, we simple copy the missing information over from $\Psi_{\hat{\vec{q}}}$ to $\Psi_{\hat{\vec{p}}}$:

$$\Psi_{\hat{\vec{p}}}(\vec{x}) = \Psi_{\hat{\vec{q}}}(\vec{x}), \forall \vec{x} \in \Omega$$

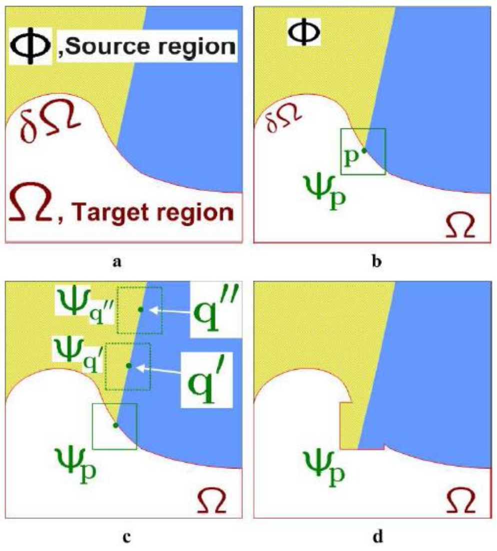


Figure 7: Visualisation of the process. Image source: [3].

After this, the algorithm then needs to update the confidence values. The confidence values are updated only for the pixels in our border patch $\Psi_{\hat{\vec{p}}}$ which were missing and has just been filled in:

$$C(\vec{p})' = C(\vec{p}), \forall \vec{p} \in \Psi_{\hat{\vec{p}}} \cap \Omega$$

The boundary pixels and their priorities are then re-computed, and this whole process is then repeated, until there are no more boundary pixels left and the whole region has been filled.

Implementation

The implementation of the above priority formulae is quire straightforward. While the paper [3] suggests a different method to calculate the normal to the region in the data term $D(\vec{p})$, we used a Sobel gradient operator on the black-and-white mask image. This gave sufficient results. Image isophotes are determined by turning the image gradient at the pixel by 90 degrees. To estimate these image gradients, a similar method is used to our implementation of Bertlamio Image Inpainting [2], where a simple central difference is used with increasing radius until a valid non-zero length gradient vector is achieved.

Implementation gets slightly more complicated when it comes to the exemplar matching. A naive brute force solution will work, but will take about 1 full second on a 2.53 mhz machine, to find the matching patch from the source region.

So to speed this up, we have implemented 3 optimisations.

First, we implemented early exit. As the algorithm loops through every pixel in the patch to compute the distance of the boundary patch to the proposed source exemplar patch, it continually compares the current distance to the distance to the best source patch found so far. Since further pixels can only ever add to the current distance, if our current distance is already bigger than the best distance, then there is no way this patch is going to end up with smaller distance than the best patch we have found so far. Thus, we can stop computing adding the distances for the remaining pixels and just discard this proposed patch. This approximately doubles the speed of the algorithm when running on the CPU.

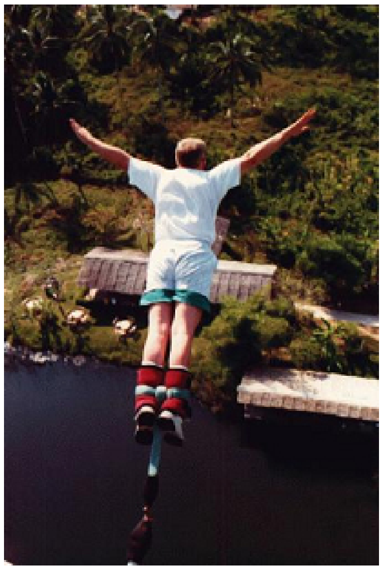
Secondly, we have implemented extraction of source patches only from within a certain radius around the inpainting mask region. This both dramatically decreases the number of source patches we have to search through, and stops the algorithm from erroneously choosing far away patches are most likely have nothing to do with the information being filled.

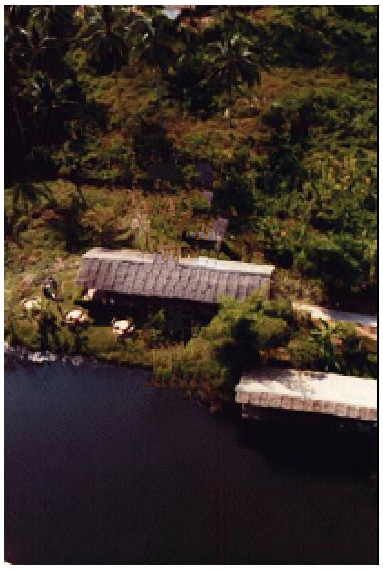
Finally, we have managed to implement exemplar searching using the systems Graphics Processor Unit, or GPU. This is done using the nVidia CUDA system. Patch data, image data and mask data, along with parameters, are copied into GPU memory, where the massively parallel architecture of todays extremely powerful graphics cards can be utilized.

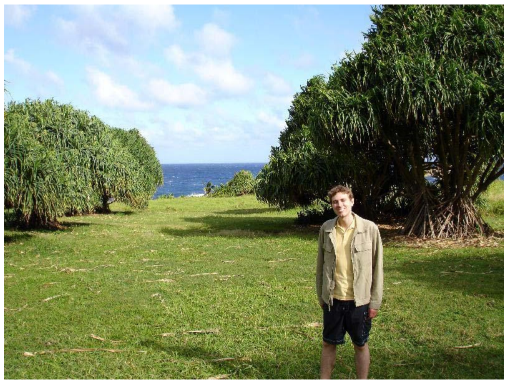
This optimization decreases brute force exemplar searching that usually takes 500ms to taking a negligible amount of time, and the actual boundary priority calculations become the major bottleneck of the algorithms speed.

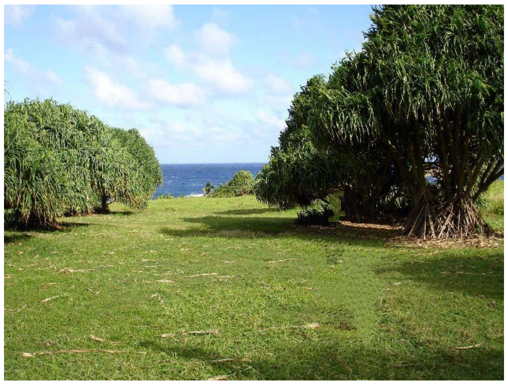
Implementing the boundary priority calculation in the GPU is also possible, but will take a much longer time, since basically the whole algorithm must be coded for the GPU to run in parallel.

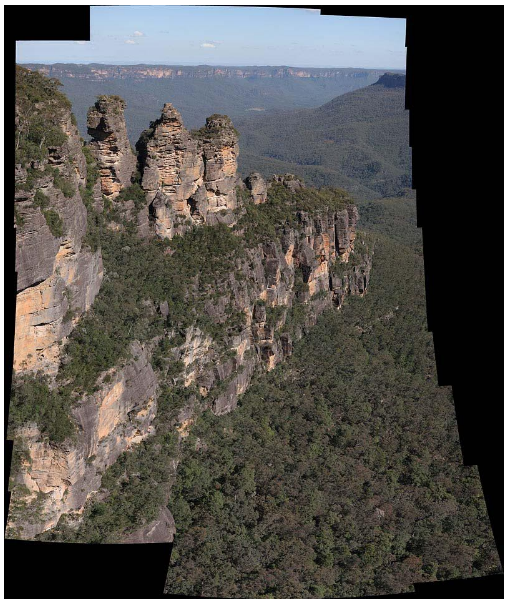
Results


(a) image 1

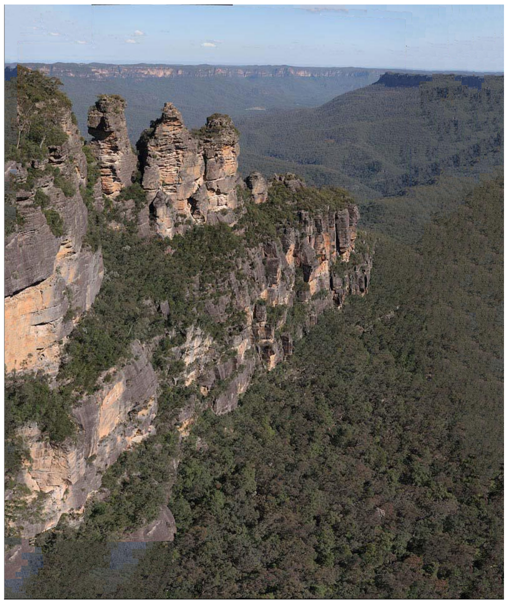

(b) result 1


(a) image 2


(b) result 2



(a) image 3



(b) result 3

Strengths & Limitations

The results from this algorithm are quite astonishing. Unlike PDE methods, which result in inevitable blurring of the inpainted region, this method uses texture synthesis, and therefore preserves texture information. Furthermore, it is also sensitive to image structure, and will propagate along strong image isophotes to preserve and extend the structure.

However, this algorithm is not without its disadvantages. Since it is a texture synthesis based method, it will perform badly when the inpainting region is spread out along most of the image, and thus there is not many valid source patches to choose from. One example of this is the polar bear example in the results section of our Bertalmio algorithm, where the algorithm has a very hard time finding a list source patches to use.

The algorithm also requires parameter tweaking in order to get the most effective results. Patch sizes must be manually chosen, with regards to the size of features in the image. With too large a patch size, the algorithm will have a high chance of picking a "bad" patch. With too small a patch size, textures will be repeated in a noticeable pattern, and the algorithm will also take much longer to run.

Conclusion

Understanding and implementing this algorithm allowed us to understand the ideas behind modern inpainting techniques, which aim to synthesize both texture and structure. It also provided us with some insight into how the parallel processing power of modern GPUs can be relevant to modern texture synthesis based inpainting techniques.

Details: Image Inpainting Based On Local Optimisation (ICPR 2010)

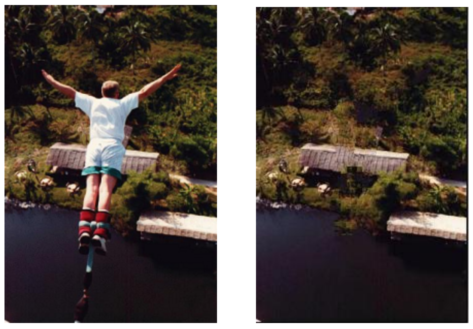
Overview

This algorithm was originally resented in [7]. It is by far the most recently published paper we have studied in our research into the field. While this algorithm is very slow, and the results are comparable to the algorithm presented in [3], it does show some margin of improvement. This algorithm's ideas represent, to us, the flagship of image inpainting methods; the most recent ideas. The algorithm is based upon exemplar-based image inpainting [3]. The most important border patch to fill uses exactly the same method as [3]. However, instead of simply selecting the most similar patch, this algorithm looks at neighbouring pixels, and finds a local optimisation of the current patch and its neighbours.

Algorithm

1. Select the border pixel with highest priority, using the method in [3].
2. For every neighbour, find a list of best k matching (we used $k = 10$) patches from the source region.
3. Find the optimum patch for this region by maximizing the following joint distribution probability formula:
$$P(\Psi_p, x_p, x_{\tilde{p}}) = P(\Psi_p|x_p)P(x_p|x_{\tilde{p}})P(x_{\tilde{p}})$$
where Ψ_p is the image, x_p is a target patch, and $x_{\tilde{p}}$ is a neighbour patch.
The term $P(\Psi_p|x_p)$ represents the similarity of the patch to the target region, $P(x_p|x_{\tilde{p}})$ represents the similarity of the patch to its neighbours, and $P(x_{\tilde{p}})$ represents the "importance" of this neighbour.
$$P(\Psi_p|x_p) = exp\Big(-d(x_p, \Psi_p)^2\Big)$$
$$P(x_p|x_{\tilde{p}}) = \arg\min_{x_{\tilde{p}}}\Bigg(exp\Big(-d(x_{\tilde{p}}, x_p)^2\Big)\Bigg)$$
$P(x_{\tilde{p}})$ is the border pixel priority value for $x_{\tilde{p}}$.
4. Fill in the missing information from the chosen source patch with maxim probability.
5. Repeat from step 1 until whole image region has been filled.

Results



(a) image (b) result

Implementation

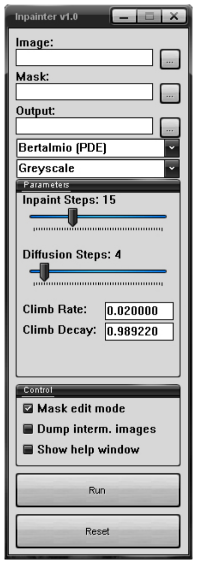
Much of the code has is shared with our code for exemplar-based image inpainting [3]. The new distance patch matching functions are also been implemented for the GPU.

Graphical User Interface

As we stated in our project goals, we aim not only to produce a simple working demo, but also to produce a workable program. Since there are a lot of methods for image inpainting with different advantages and disadvantages each with many parameters to adjust, we have gone with the decision to build a graphical user interface.

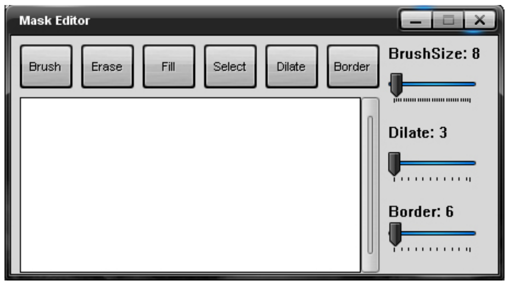
The graphical user interface would allow switching between algorithms, easy adjustments of algorithm parameters, easy image and mask loading/saving. It would also provide the user with a simple mask editing interface.

The interface was developed in Win32 Common Controls API. The image displaying and mask editing mouse events are handled by OpenCV GUI. The two GUI APIs seamlessly integrate for the purposes of this project.



Bertalmio (PDE)
Greyscale
Inpaint Steps: 15
Diffusion Steps: 4
Climb Rate: 0.020000
Climb Decay: 0.989220
☑ Mask edit mode
Dump interm. images
Show help window

(a) Main window interface



BrushSize: 8
Dilate: 3
Border: 6

(b) Mask editor interface

Figure 8

Scope & Specification

The project, from our point of view, has been quite challenging. Every algorithm has to be researched, understood, and among them, 4 of them implemented from bare scratch, with 2 of them having parts implemented on the GPU. On top of that, the user interface had to be designed, then built in Win32 GUI from scratch. We believe that native Win32 GUI code will be the most versatile, elegant, portable and compatible for the end user (no additional GUI libraries have to be installed / included with package, simply download the executable then run).

Project Name: Image Inpainter
Version: 1.0
Supported image separation methods: 4
Supported algorithms: 7
Supported file formats: BMP, JPEG, PNG
Memory requirements: 512MB+ ram, 1024MB recommended for large images
CPU requirements: Any x86 / x64 processor
Operating system: ©Microsoft ©Windows 2000 or newer, x86 or x64
GPU requirements: Any CUDA-enabled ©Nvidia graphics card; ©GeForce 8600 or newer

Group

- Xi Chen: Algorithm development, evaluation and report, general research, user interface developing, user interface designing.
- Willy Mai: Testing, algorithm research, algorithm development, evaluation and report.

Project Conclusion

We have learnt a lot from this project. Our outcomes were very successful; we have gained the understanding we wanted, including reading and understanding some recent cutting edge technology papers; and we have also managed to develop a simple, powerful, versatile tool for image inpainting with some great results.

Future Expansions

As suggested by a tutor, one possible direction of expansion could be porting the implementations to a handheld device, which would give the project more commercial marketability.

The more contemporary image inpainting algorithms could be implemented.

Making more use of the GPU is also a possible extension, as graphics processing units were designed for the very task of image processing, more or less. Image processing required by image inpainting often will run very well on the GPU; one such example being image patch template matching.

References

[1] Manuel M. Oliveira, Brian Bowen, Richard McKenna and Yu-Sung Chang, Fast Digital Image Inpainting, Imaging and Image Processing (VIIP 2001), Marbella, Spain, September 3-5, 2001. Available at `http://www.inf.ufrgs.br/~oliveira/pubs_files/inpainting.pdf`

[2] Marcelo Bertalmio, Guillermo Sapiro, Vicent Caselles and Coloma Ballester, Image Inpainting, SIGGRAPH 2000, available at `http://www.dtic.upf.edu/~mbertalmio/bertalmi.pdf`

[3] Alexandru Telea An Image Inpainting Technique Based on the Fast Marching Method, Eindhoven University of Technology

[4] Marcelo Bertalmio, A. L. Bertozzi, Guillermo Sapiro, Navier-Stokes, Fluid Dynamics, and Image and Video Inpainting, available at `http://www.dtic.upf.edu/~mbertalmio/final-cvpr.pdf`

[5] A. Criminisi, P. Perez and K. Toyama, Region Filling and Object Removal by Exemplar-Based Image Inpainting, IEEE TRANSACTIONS ON IMAGE PROCESSING, VOL. 13, NO. 9, SEP 2004

[6] Wilson Au and Ryo Takei, Image Inpainting with the Navier-Stokes Equations, available at `http://www.math.ucla.edu/~rrtakei/gradProj/930project.pdf`

[7] Jun Zhou and Antonio Robles-Kelly, Image Inpainting Based on Local Optimisation, ICPR 2010, available at `http://users.cecs.anu.edu.au/~junzhou/papers/C_ICPR_2010.pdf`