Faster Fragment-Based Image Completion

Benjamin Lubin and Geetika Tewari
Computer Science 278
Harvard University
blubin@fas.harvard.edu
gtewari@eecs.harvard.edu

Abstract

The fragment-based image reconstruction algorithm presented at SIGGRAPH 2003 by Drori et al. provides a way to complete missing parts of an image by iteratively approximating the unknown regions and compositing image fragments into the original image. We modified this algorithm in several ways, including the addition of a KD-Tree to speed up the fragment search process, and implemented the algorithm in Java using the Java Advanced Imaging APIs. We also used images of significantly higher resolution than the images used in the paper by Drori et al., therefore providing a challenging test of the algorithm’s effectiveness. Our experiments demonstrate that this modified algorithm does an effective job of reconstructing higher resolution images and improves upon the speed of the original. Empirically, we find that our implementation with a KD-Tree can perform at least 16 times faster than one without it.

1 Introduction

While many tools allow removal and patch up of image regions by hand, it requires a considerable amount of skill and time to do so successfully. Consequently, there have been a number of attempts to complete missing regions in an image automatically. However, these techniques have not been generally effective when large-scale regions need to be reconstructed and only intermediate scale fragments are available. The image reconstruction technique proposed by Drori et al. in 2003 addresses this problem. Their method produces great results when the missing parts of an image match the remaining visible regions, since it is based on selecting fragments from visible regions of the image and fitting them into the missing regions.

Related Work

The algorithm we have implemented is a modified version of the one presented by Drori et al. [3]. Their algorithm is given a source image and a matte that specifies regions that have been removed. It then attempts to fill in the unknown regions, first using a smooth approximation, and subsequently using data sampled from the known regions in the image.

There has been a large body of work prior to the Drori paper that relates to image completion, from the fields of both computer graphics and computer vision. One method employed by Hertzmann et al. [8] is Image Analogies. This technique applies the relation between a pair of stock images to a target image, creating a novel destination image. Freeman et al. [5] developed a technique for performing super-resolution, by using a database of training images to supply additional detail.

The large body of work in texture synthesis is also relevant to attempts to perform effective image completion. Wei and Levoy presented a form of texture synthesis in which a new texture is created by considering similar neighborhoods in a source texture. Welsh et al. [9] extended this idea, creating a form of image analogies based on matching local image statistics. Lastly, Efros and Freeman [4] presented a synthesis technique that creates additional texture by putting together blocks of existing simple textures.

Another recently developed image completion technique that seems quite relevant to the algorithm implemented here is the image inpainting algorithm developed by Bertalmio et al. in 2000 [1]. Image inpainting is targeted at completing relatively small scale regions of an image, that are smooth and non-textured. In 2003, Bertalmio et al. [2] combined their inpainting algorithm with texture synthesis by representing an an image as the sum of two components.

Our Approach

Our reason for implementing the algorithm by Drori et al., as opposed to the aforementioned methods, lies in the nature of the input images we want to complete. In our images, a large chunk of the image is missing, and the content of the visible part of the images seems to closely match those areas that have been removed, which implies that algorithms like Drori’s should perform well.

The specific contributions of this paper are:
• An implementation of the Fragment-Based Image Completion algorithm by Drori et al. [3].
• Incorporation of a KD-Tree into the fragment search process.
• Several variations on Drori’s image reconstruction process, particularly in the search and compositing stages.

The remainder of this paper is organized as follows. Section 2 introduces the fragment-based image completion algorithm by Drori et al. and points out our main modifications we introduced. Section 3 discusses the details of our implementation and highlights the precise design decisions and challenges. It also provides an overview of our code structure and system design. We present visual examples of images completed by our algorithm in section 4, and we conclude with ideas of ways to extend this work in section 5.

2 Completion Algorithm

Fragment-Based Algorithm

Dori et al.’s algorithm requires that the missing region of the image be designated by a matte (which we’ll call the completion matte) where the missing and visible regions of the image are specified by a value in [0, 1]. This inverse of the matte is used to define a confidence value for each pixel, a value that increases as the image completion process proceeds.

Dori et al. define a fragment as a circular neighborhood around a pixel. Image completion proceeds hierarchically from coarse to fine, where initially a low resolution image is generated and then the results serve as an approximation for the next higher-resolution (finer) level. For each image scale target fragments that need to be filled in are chosen, concentrating on targets that have confidence near the mean.

At each algorithm step, a target fragment is completed by adding more detail to it from a source fragment with higher confidence. The source and target fragments are composited together using a Laplacian pyramid.

2.1 Our adaptation of the Image Completion Algorithm

Fig. 1 summarizes the main steps of our modified version of the Image Completion algorithm. While the basic structure of our algorithm is the same as in Drori et al., we have made important changes to both the overall algorithm and several of the individual steps it performs. We now outline the details of how we handled each stage of the algorithm:

Initial Approximation via Push-Pull

We have implemented the iterative filtering method by Gortler et al. known as Push-Pull [7]. This method is capable of quickly generating a smooth completion by repeatedly down-sampling and up-sampling the image hierarchically (as illustrated in Fig. 2): The algorithm proceeds in levels from \( n \in \{N, ..., 1\} \), and within each level, it proceeds in rounds. At a given level \( n \) a round consists of the following: first the algorithm caches the completion obtained from the last round (or the raw resampled image for this level, if it is the first level). Next, the algorithm downsamples the image \( n \) times by some factor via a specific kernel, and then upsamples the image again. This results in a blurring of the image. The known pixel data from the original image is then superimposed upon this blurry image, letting the blurred data from the resampled data show through in the unknown region(s). Finally, the algorithm compares the results of this round with the cached results from the last round. If they are similar enough, the current level is considered complete; otherwise, the algorithm loops to begin a new round.

When all of the rounds for a given level are complete, the algorithm begins a new level, provided that the current level is greater than 1. Each new level downsamples one fewer time. The net effect is that the blurring that occurs in early levels is coarse, permitting large changes in color in the fill-area. As the level numbers drop, though, this anneals and the smoothing becomes more fine grained and gradual.

In our implementation of Push-Pull, when we downsample/upsample, we do so by a factor of 2 each time. We allow the downsampling to proceed down to a small group of pixels. The downsampling and upsampling is performed by bilinear interpolation, thus using a very small kernel.

The Push-Pull algorithm is capable of very quickly and consistently producing a convincing and visually appealing
smooth blurring over the unknown region, and provides an excellent starting point for further processing.

Creating the Confidence Map

Like Drori et al., we create a confidence map, where each pixel $i$ is assigned a confidence value $\beta$ as follows:

$\beta_i = 1$ if the inverse completion matte $= 1$ and otherwise $\beta_i = \sum_{j \in N(i)} g_j \cdot (\text{inverse completion matte})^2_i$ where $N(\cdot)$ is a neighborhood around a pixel, $g$ is a Gaussian falloff factor. It is worth noting that these $\beta$ values must be maintained throughout our algorithm, which the composite process (described below) ensures when it copies data from one part of the image to another.

Building the KD-Tree

While the Drori et al. paper does a brute force search through the image when looking for a match for a given destination fragment, our approach precomputes a data structure to make this search fast and efficient. The details of this search are given in a section below; here we specify how we construct the data structure.

First, the original image is border-expanded by reflection in order to make the neighborhoods of the edge pixels well defined. Now, because the search is to include all of the fragments in the image, what we really need to do is index these fragments in such a way that we can quickly find fragments similar to a given query fragment. To do this, we treat each fragment as a point in $n$-dimensional space, 3 dimensions for each pixel in the fragment (one for each of the bands). Along with the dimensional data, each point also maintains its $x, y$ location in the original image, although these coordinates are not considered to be among the dimensions of the point for indexing purposes. We then insert these points into a KD-Tree.

The KD-Tree is a generalization of a standard binary tree used for sorting and searching that is optimized for dealing with high dimensional data points. Our implementation of the KD-Tree is based on the algorithm outlined by Friedman et al. [6]. At each level of a KD-Tree, a different key is used to index which subtree the query point should belong to (in our case, the keys are the value for a specific dimension of the points). In a standard KD-Tree, these keys are chosen sequentially. We have implemented an optimization that stores which key to use as for the index explicitly at each node, permitting each node to use an arbitrary key. At each node, we choose the key whose values have the widest spread in values which should, in general, lead to a well balanced tree. Having a balanced tree means that we expect build time for the tree to be $kn(log(n))$ (where $k$ is the number of dimensions and $n$ is the number of elements added to the tree) and for search time to be $(log(n))$.

Determining Target Positions

In each scale from coarse to fine, we traverse the image from high to low confidence regions. We compute the next target position exactly as outlined in the Drori et al. paper. Pixels that are greater than the mean confidence $\mu(\beta)$ are never choosen. We choose among the remaining pixels according to their confidences, except that we add a random noise value between 0 and the standard deviation $\sigma(\beta)$ of the $\beta$s. This provides a little bit of randomness as to which point is chosen and corresponds to us being somewhat uncertain in our confidence assignments. It should also be noted that we may subsequently reject certain target positions based on the search process they engender (described in the next section). We track these points and make sure that we never choose them again.

In addition to searching for points to copy unmodified, we can also search over transforms of the source points. We have implemented the following fragment transforms: rotation by 90, 180, and 270 degrees and vertical, horizontal, diagonal and anti-diagonal transpositions. These transforms can be incorporated into our KD-Tree-based search method simply by adding points corresponding to the post-transform neighborhoods around the source pixels to the KD-Tree when it is built.

Searching for Similar Fragments

An important difference between our implementation of the search technique and that outlined in the paper by Drori et al. is that we set a threshold for the maximum distance $max_d$ between a source and target pixel. If we cannot find a source pixel that is less than $max_d$, then we abandon the search on that target fragment, and choose a new target point. Thus, even if we are never able to find an appropriate source match for the target neighborhood, we prefer the the blurry version from the coarser approximation instead of copying...
in noise.

Since we only search for source fragments that are amongst the visible regions of the image, we have not implemented the $\beta$-based selection function $r^* = \text{argmin} \sum_{s = D,(i,t),i \in N(d(s,t)\beta_3 + (\beta_1 - \beta_3))\beta_4}$ used by Drori et al. to discriminate amongst potentially close source fragments.

KD-Trees are often used to obtain the k-nearest neighbors to the query. Despite having implemented Friedman et al. [6]’s complex k-nearest search method, we are only searching for the single nearest neighbor to our query. It would be interesting to instead query for the k-nearest neighbors and then use some complex metric to choose between them instead of the L2 metric used by the tree. This metric could be the $\beta$-based one listed above, but it could also be an analysis of the structure of the fragments themselves and perhaps even their surrounding contexts.

**Compositing Fragments**

Our fragment-compositing method is illustrated in Fig. 3 and is one of the areas of the paper where our implementation differs significantly from that of Drori et al. We did not implement a Laplacian over operator to perform the image compositing step as they do. Rather, an $\alpha$-channel for the source fragment is created by multiplying the source fragments confidence at each pixel with a Gaussian falloff. The $\alpha$-channel for the destination fragment is simply the $\beta$ values located at the destination fragment. We then run the standard $\alpha$-over operator, compositing the destination over the source. This means the destination pixels get precedence according to their confidence. It is also worth mentioning that it is important that this composite operation not be pre-multiplied. The is because we don’t want the color bands $\alpha$-scaled (in the absolute sense) in the output – only scaled with respect to each other. The result of this compositing is an image with four channels, from which the color bands are extracted for use at the destination and the $\alpha$ channel is extracted for use as the new destination $\beta$ values.

The extra gaussian falloff term specified here provides some extra $\alpha$-blending near the edge of the fragments. This helps ensure that the edges of the copied fragments are not visible and helps ensure that the image looks consistent, at the price of some additional blurring. We attempt to compensate for this blurring by running a Laplacian sharpening filter over the fragments when upscaling, as described in the next section.

**Integrating Images at Different Scales**

We must be able to integrate course and fine images when the completion of the course version of the image is finished and we need to start working on the next finer version. Our procedure for doing this also involves alpha compositing and is illustrated in Fig. 4. First we down sample to the course level and recursively complete the image at that lower level.

Once that completion has been performed, we need to integrate the resultant image into the initial approximation at the finer level before beginning completion at that level.

Beginning with the result from the course level, we first upsample the image using a reasonable kernel, such as bilinear. Next we apply a Laplacian sharpening filter to compensate for the several sources of blurring introduced into the image. In our method, there are three sources of blurring: (1)the blurring inherent in the $\alpha$-composite used when combining fragments (2)the Gaussian falloff multiplied into the source OS to reduce edge effects at fragment borders, and (3)blur introduced by upsampling the image with most kernels. Due to this blurring, the colors tend to become more indistinct. This can be problematic during the search process, and adversely impacts the results. The Laplacian sharpening filter attempts to compensate for this blurring. We believe that this produces results similar to the more complex Laplacian pyramid approach used by Drori et al.

Finally we need to integrate our upsampled and sharpened image with the finer resolution image for this level, which we do by alpha compositing. We composite the finer resolution version over the upsampled image to ensure that pixels outside of the completion region are not compromised. For the upsampled image we simply use an opaque $\alpha$-channel. For the finer image we use an alpha channel created by performing a Gaussian blur of the original completion matte. This blur reduces the edge effects that would otherwise occur by allowing some of the initial approximation color to influence the edges of the fill area at the new level, and by letting some of the lower level completion data influence the color of the data outside the completion area of the higher level. One consequence of doing this is that some pixels near the completion area may be subtly changed to better blend with the completed region.

**2.2 Additional Algorithmic Considerations**

**Color Coding**

We allow the representation of image color in two possible formats, Hue-Intensity-Saturation (HIS) or RGB. Start-
ing with RGB, we optionally convert to HIS before copying fragments and then switch back afterwards. Representing the data in a different color space affects the properties of the L2 metric used in our KD tree by weighting certain aspects of the fragments more highly than others. One could certainly envision trying other color spaces, tuning them for optimal results when searching for matching fragments.

2.3 Impact of KD-Tree on Time Complexity

Let \( k \) be the number of pixels in the neighborhood of each pixel, \( h \) be the number of pixels in the missing region of the image, and \( n \) be the total number of pixels in the image. Using this notation, the time taken to build our KD-Tree is: \( k(n - h)\log(n - h) \) and the time taken to search enough to fill the whole is \( \frac{c}{k} \log(n - h) \), where \( c \) is some constant representing the how much the fragments must overlap due to the \( \alpha \)-composing we do. The total time to work on our tree is: \( O(k(n - h)\log(n - h)) \). Assuming \( n > h \), this implies that the total time to build the KD-Tree and search for closest source fragments on it is: \( O(kn\log(n)) \). In contrast, the implementation by Drori et al. would take \( O(hnk) \), according to the method outlined in their paper. If the missing region in the image is even moderately large in comparison to the image as a whole, then there will thus be significant gains by using our KD-Tree based approach.

3 Implementation

We have implemented this algorithm just described in Java using the Java Advanced Imaging library. In this section, we give precise details of our implementation, an overview of the accompanying design decisions and challenges, and an outline of our code structure.

Code Structure

Fig. 5 shows the global structure of our code. The top level \texttt{ImageCompletion} class initializes the other classes and is the starting point for execution. The main function reads in a \texttt{properties} file that specifies the parameters used by the program, allowing the user to define parameters without having to recompile each time. Our code defines an abstract class \texttt{ImageChanger} which specifies an Object that is capable of altering an image to produce a new image. The \texttt{ImageCompletion} class instantiates an array of subclasses of \texttt{ImageChanger}, then calls each in turn. The calls on these sub objects cause the image to be loaded from disk, altered in various ways, and then written back to disk (optionally recording intermediary results to disk as well).

In addition, we have a Java GUI that can be turned on to visualize the image as the algorithm proceeds. The GUI implements the \texttt{ImageChangeListener} interface, which enables it to be told about changes to an image. The top level class ensures that the GUI is regestered as a listener such that it hears about the image alterations that occur in various subclasses of \texttt{ImageChanger}, causing the GUI to be updated appropriately.

We have implemented the following separate classes for main steps of the image completion process (as subclasses of \texttt{ImageChanger}):

- \texttt{FindMatte}: Computes the completion matte from the original image where it is specified as black pixels for our input files. This matte partitions the image into two regions: the known region (where it is 1 and the unknown region where it is 0).
- \texttt{InverseMatte}: generates an inverse completion matte from the completion matte.
- \texttt{PushPull}: A fast estimate of the colors of in the unknown region generated by an iterative resampling and overlay process.
- \texttt{TreeFragmentCopy}: Carries out hierarchical fragment based reconstruction of the unknown region, optimized by using a KD-Tree.
Properties File

As mentioned previously, our program employs a user specified properties file to set the parameters that control functionality of our algorithm (as well as program specific parameters not listed here, such as whether or not to display a GUI):

- Convergence threshold for the push-pull step
- Maximum number of down-sampled levels for the push-pull step
- If the image should be shifted into the HIS color space and back
- Size of the neighborhood used for fragment Completion search
- Convergence threshold for a given level of the fragment completion algorithm to stop.
- Amount to blur the edges of fragments by to remove edge effects
- Amount to sharpen between levels to avoid washed out colors.
- How much to blur the edges of levels together to avoid edge effects around the completion region.
- How far apart source and target fragments can be before that destination is rejected ($\delta$)
- Which transformations of the source fragments should be included in the tree for search

4 Results

As mentioned before, the image set we tested our implementation on contains higher resolution images than those used in the Drori et al. paper. These initial test images are shown on the left side of Figs. 6-9. The results of our algorithm on these test images are illustrated on the right hand side of these figures. Our implementation performs best on the Lake scene (Fig. 7). The reconstruction of the Mountain scene is reasonably convincing given the size of the patch. Spacial information is more prominent in the Tree and Shadows image. Because of this you can see in Fig. 8 that it is not as successful as the others. However, we found that this reconstruction is improved when we perform the completion in the HIS color space, as seen in Fig. 9.

A comprehensive summary of our results can be found at: http://www.fas.harvard.edu/blubin/cs278/ImageData/. This site also includes intermediate reconstruction frames, as well as animated gifs of our algorithms in action. The running times of our implementation are displayed in Table 1. All experiments were run on a Pentium 4, 2.8 Ghz with 1GB of main memory, java 1.4.2_03 and JAI 1.1.2.

<table>
<thead>
<tr>
<th>Image</th>
<th>Color Space</th>
<th>Algorithm</th>
<th>Run Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pic1</td>
<td>RGB</td>
<td>Brute Force</td>
<td>&gt; 16hr</td>
</tr>
<tr>
<td>Pic1</td>
<td>HIS</td>
<td>Tree</td>
<td>35m16s</td>
</tr>
<tr>
<td>Pic2</td>
<td>RGB</td>
<td>Tree</td>
<td>65m26s</td>
</tr>
<tr>
<td>Pic3</td>
<td>RGB</td>
<td>Tree</td>
<td>32m34s</td>
</tr>
<tr>
<td>Pic3</td>
<td>HIS</td>
<td>Tree</td>
<td>37m15s</td>
</tr>
<tr>
<td>Pic3</td>
<td>HIS</td>
<td>Tree</td>
<td>10m27s</td>
</tr>
<tr>
<td>Pic3</td>
<td>HIS</td>
<td>Tree</td>
<td>18m</td>
</tr>
</tbody>
</table>

Table 1. Running Times of our Implementation
5 Conclusions and Future Work

There are also several extensions to this work that would be interesting to pursue:

- Our results indicate that some of the time fragment search should be performed with the image represented in the Hue-Intensity-Saturation (HIS) color space. Firstly, perhaps other color spaces, or a custom one, would work better. Further, in our implementation fragment search and image construction are done on the same image and thus in the same color space. However, in practice we find that converting between RGB and HIS results some loss of color fidelity. Therefore, one improvement would be to maintain two images to permit searching in one color space while simultaneously performing all image operations in the original color space to avoid conversion loss.

- Image quality could probably be improved by performing additional transformations when building the KD-Tree (e.g., arbitrary rotations, and source fragment scaling).

- It would like increase efficiency to prune fragments before adding them to the KD-Tree. Filtering extremely similar fragments would avoid bloating the tree with redundant information.

- Searching over multiple destination neighborhood sizes would likely make the images look better. Drori et al. search for only one size at each destination, but vary this size based on the contrast at that location. With our faster Tree-based technique, it should be possible to maintain multiple trees containing fragments of different sizes. The algorithm could then either choose a specific size to use by some heuristic (such as contrast), or it could search for multiple sizes (each in its own tree) and then pick one to use based on some quality metric.

- As mentioned early, if we took advantage of the k-closest neighbors search provided by the KD-Tree, we could then pick among these candidate source points by a metric that is more complex and accurate than the simple L2 metric used by the tree. Depending on the quality of metric used, this could result in a significant improvement in image quality.

This paper demonstrates that our modifications of the fragment completion algorithm presented by Drori et al. can be used successfully on higher resolution images, although there are still limitations when higher-order spatial relations are key to the image gistalt. Our addition of the KD-tree further enhances the Drori et al. algorithm by significantly reducing runtime.

References