PatchTable: Efficient Patch Queries for Large Datasets and Applications

Connelly Barnes[1] Fang-Lue Zhang[2] Liming Lou[1,3] Xian Wu[2] Shi-Min Hu[2]
[1]University of Virginia [2]TNList, Tsinghua University [3]Shandong University

(a) Input (b) Engraving (c) Oil (d) Watercolor

Figure 1: *Our artistic video stylization, which generalizes image analogies [Hertzmann et al. 2001] to video. An input video (a) is stylized by example in several different styles (b-d). We show a detailed crop region of a single frame of the full video, which is shown in the supplemental video. In the second row we show the exemplar pair that is used to drive the stylization, which consists of an image "before" and "after" the effect is applied. Our data structure enables this effect to be rendered on the CPU at 1024x576 resolution at 1 frame/second, which is significantly faster than previous work.* *Credits: Video in top row © Renars Vilnis; (b) Thomas Nast; (c) Vincent Van Gogh; (d) © Hashimoto et al. [2003].*

Abstract

This paper presents a data structure that reduces approximate nearest neighbor query times for image patches in large datasets. Previous work in texture synthesis has demonstrated real-time synthesis from small exemplar textures. However, high performance has proved elusive for modern patch-based optimization techniques which frequently use many exemplar images in the tens of megapixels or above. Our new algorithm, PatchTable, offloads as much of the computation as possible to a pre-computation stage that takes modest time, so patch queries can be as efficient as possible. There are three key insights behind our algorithm: (1) a lookup table similar to locality sensitive hashing can be precomputed, and used to seed sufficiently good initial patch correspondences during querying, (2) missing entries in the table can be filled during precomputation with our fast Voronoi transform, and (3) the initially seeded correspondences can be improved with a precomputed k-nearest neighbors mapping. We show experimentally that this accelerates the patch query operation by up to 9× over k-coherence, up to 12× over TreeCANN, and up to 200× over PatchMatch. Our fast algorithm allows us to explore efficient and practical imaging and computational photography applications. We show results for artistic video stylization, light field super-resolution, and multi-image editing.

CR Categories: I.3.6 [Computing Methodologies]: Computer Graphics—Methodology and Techniques; I.4.9 [Computing Methodologies]: Image Processing and Computer Vision—Applications;

Keywords: Approximate nearest neighbor, patch-based synthesis

1 Introduction

Digital photography has recently shown clear trends towards higher resolutions and large collections of photos. Users frequently have personal collections of thousands of pictures, and billions of photographs are available from online photo-sharing websites. Resolutions range from tens of megapixels on consumer cameras up to gigapixels on specialized capture devices. There is a need for algorithms that scale to large image datasets.

Modern patch-based methods have delivered high quality results for many applications, by synthesizing or analyzing images in terms of small regions called patches (or neighborhoods). However, these methods have had a difficult time scaling to large quantities of data. For instance, the PatchMatch [Barnes et al. 2009] method is inefficient for large dataset sizes. This is because finding a correspondence in the worst case reduces to random sampling, which takes time linear in the database size. In contrast, earlier methods for texture synthesis could operate in real time [Lefebvre and Hoppe 2005], due to using low resolution inputs and the use of precomputation in the design of their data structures.

Our aim is to accelerate modern patch-based methods so they efficiently scale to high-resolution photographs and collections of photos. In particular, we aim to accelerate the approximate nearest neighbor search problem over image patches. This problem has attracted much research interest recently [Barnes et al. 2009; Korman and Avidan 2011; Olonetsky and Avidan 2012]. To accelerate this search problem, we use an approach similar to Web search engines. We assume that modest time resources can be devoted to precomputing an *inverted index* data structure that consequently will permit very fast queries. As we demonstrate, for many applications, this index can be computed once when a user's photographs are first loaded, and re-used for many queries. This improves interactivity and efficiency.

The main ideas behind our method are to push as much as possible of the work to a precomputation stage when the index is built, and to use array data structures to represent patch appearance space. This allows us to avoid the complex traversals that hierarchical data structures like kd-trees must make during patch queries. After the

index is constructed, the time to query the patches of an image against the index is quite fast. We show improvements in query time of up to 9× over k-coherence [Tong et al. 2002], up to 12× over TreeCANN [Olonetsky and Avidan 2012], and up to 200× over PatchMatch [Barnes et al. 2009], with relative speed-ups generally increasing for larger datasets (see Section 7 for more details).

To create this data structure, we make three technical contributions. First, we precompute a lookup table, and use this during querying to seed good initial patch correspondences. This table is constructed similarly to locality sensitive hashing (LSH) tables [Datar et al. 2004; Korman and Avidan 2011]. Second, we fill missing entries in the table using a novel fast Voronoi transform. Third, we improve initially seeded correspondences with a precomputed k-coherence mapping, which uses a novel early termination test for efficiency. We call our combined inverted index data structure a *PatchTable*. We present our method in Sections 3-5.

In the context of artistic and photographic manipulations, efficient and interactive methods are critical. We demonstrate this with our three applications: artistic video stylization, light field super-resolution, and multi-image editing. Our artistic video stylization method (presented in Section 8.1) generalizes the image analogies framework [Hertzmann et al. 2001] to video. Here efficiency is important to render the output video at a reasonable rate. For light field super-resolution (shown in Section 8.2), we show that our method permits a practical running time. We believe this will encourage the adoption of light field technologies. For multi-image editing (presented in Section 8.3), we show that a user can compute high-quality results from image collections quickly. If the user is not happy with the result, then because the index has already been created, a new result can be recreated nearly instantly. Our new data structure improves the running time and interactivity of these applications, due to its better scalability properties.

2 Related work

We first discuss approximate nearest neighbor techniques used for image patch searches. Then we discuss how these techniques have been incorporated into applications in traditional texture synthesis and inpainting, artistic video stylization, and modern patch-based methods that use large datasets.

Approximate nearest neighbors. Exact nearest neighbor searches in high-dimensional spaces generally have high run-times, due to the necessity of either performing linear scans over the database [Xiao et al. 2011] or due to the curse of dimensionality when using data structures [Borodin et al. 1999]. This has motivated the development of approximate nearest neighbor methods. Different algorithms have been explored for approximate nearest neighbor queries in metric spaces, such as kd-trees, locality sensitive hashing (LSH) [Datar et al. 2004], and others. These have been incorporated in libraries such as FLANN [Muja and Lowe 2009]. Image patches can be indexed and searched by such general methods. However, more efficient techniques have been developed that rely on *coherence*, that is, correspondences for adjacent patches tend to be piecewise smooth. This observation led to a fast patch querying method called PatchMatch [Barnes et al. 2009; Barnes et al. 2010], which used randomized sampling and spatial propagation of good matches. Korman and Avidan [2011] combined spatial coherence with locality sensitive hashing to improve query times and accuracy over PatchMatch. Similarly, Arietta and Lawrence [2011] explored locality sensitive hashing for patches. TreeCANN [Olonetsky and Avidan 2012] and propagation-assisted kd-trees [He and Sun 2012] used spatial coherence to accelerate kd-tree searches using patches, and therefore improved query times and accuracy over the method of Korman and Avidan. Our work uses techniques from these previous methods, including a technique similar to locality sensitive hashing, and spatial propagation. However our goal is to minimize query time even at the expense of higher pre-computation time, so we develop different data structures which permit fast lookups and few patch comparisons at query time.

Texture synthesis. Texture synthesis aims to create tileable textures from small exemplars [Efros and Leung 1999]. Real-time and GPU implementations have been developed [Lefebvre and Hoppe 2005; Lefebvre and Hoppe 2006]. Two ideas were important for efficiency: spatial propagation, which was explored by Ashikhmin et al. [2001], and k-coherence [Tong et al. 2002], which involved the pre-computation of k most similar patches within the exemplar. Applications include texture transfer [Efros and Freeman 2001] and super-resolution [Freeman et al. 2002]. Our work incorporates the k-coherence idea, and the idea that run-time efficiency can be improved by performing a more extensive pre-computation. Our goal and approach are different because we aim to have low matching error even in high-resolution image collections, which requires a more sophisticated matching algorithm than for small textures.

Image inpainting. Image inpainting [Bertalmio et al. 2000] removes part of an image by replacing it with a synthesized region. High-quality inpainting results have been demonstrated by greedy [Criminisi et al. 2004] and iterative patch-based methods [Wexler et al. 2007]. Image melding can perform inpainting across multiple photographs, and geometric or lighting transformations [Darabi et al. 2012]. Our multi-image inpainting application is similar to image melding in its use of multiple photographs and geometric transformations, however our technique is interactive whereas image melding took many minutes.

Artistic video stylization. Our *artistic video stylization* application extends the image analogies framework [Hertzmann et al. 2001]. Image analogies permit many effects including transfer of painterly styles by example. By-example video stylization was explored by Hashimoto et al. [2003], Bousseau et al. [2007], and Bénard et al. [2013]. High-resolution was precluded in those works by computational cost. For example, Hashimoto et al. reports 2 minutes rendering time per frame on a 352x240 output video, and Bénard et al. requires 10 minutes per frame on the GPU. Our novel data structure increases the rendering speed to 1 frame/second on the CPU, for videos of medium resolution (1024x576).

Patch-based methods for large datasets. One of the motivations for our work is the trend towards larger datasets at higher resolution. Such datasets are seen in volumetric texture synthesis [Kopf et al. 2007], image melding [Darabi et al. 2012], and patch-based HDR video [Kalantari et al. 2013] which takes several minutes per frame. Recently, AverageExplorer has shown interactive patch-based alignment [Zhu et al. 2014], but required an expensive pre-computation for real-time matching. High-quality BTF interpolation has been demonstrated with patch-based synthesis [Ruiters et al. 2013], however this takes 1 hour to generate a 512x512 image. Matching graphs have been computed between and within photos in image collections, for photo collection editing [HaCohen et al. 2013; Hu et al. 2013], and image extrapolation [Wang et al. 2014]. We instead focus on the problem of finding patch correspondences. Image super resolution for light field cameras has recently been demonstrated by Boominathan et al. [2014]. We show how our method can accelerate this super resolution problem. Our approach takes a first step towards making such large dataset patch-based methods more efficient, interactive, and practical.

3 Overview of Method

Our method works by first precomputing a data structure that indexes the patches within the *database image*. Subsequently, the data structure can be queried with a *query image*. Note that we can easily index image collections by simply concatenating them into a single large database image[1]. We now proceed to define our terminology, and then explain at a high level how the pre-computation and query work for our data structure.

The terminology we use is as follows. We define at every pixel of both the query and database image a *feature descriptor* which has dimensionality d_0. Our goal is to find approximate nearest neighbors for feature descriptors, given a distance function between them. Unless otherwise specified, we use Euclidean distance between descriptors. The feature descriptor can be the output of any function of an image region surrounding the target pixel. We refer to this image region as the *patch*, and its position coordinate is simply the pixel that generated the patch. For example, if color patch descriptors are used, for square patches with size $p \times p$, then the feature descriptor simply collects RGB colors for the neighboring region centered at the given target pixel.[2] This results in a feature descriptor with dimension $d_0 = 3p^2$. Note, however, that we can actually use any descriptor, so our technique is fairly general. As another example, our light field super-resolution application in Section 8.2 uses features based on edge responses that also have PCA dimension reduction applied. The final output of our method is a mapping from each query patch to an approximate nearest neighbor patch in the database image. We call this a *Nearest Neighbor Field* (NNF). This stores for every query patch (x, y) coordinate, the coordinate of an approximate nearest neighbor patch in the database, as well as the patch distance. Our goal is for the lookup stage to produce a NNF efficiently, even for large database sizes.

We now explain at a high level how the pre-computation for our data structure proceeds. An overview of the precomputation process is shown in Figure 2. Our data structure has two components: one or more tables and a k-coherence mapping. The tables serve to give a good initial correspondence for a patch, whereas the k-coherence mapping helps improve existing correspondences. We discuss in Section 4.1 how we initially populate the tables with patches drawn from the database image. Next, we explain in Section 4.2 how we fill holes that remain in the tables using our fast Voronoi transform. Finally, we discuss in Section 4.3 how we construct the k-coherence mapping. Once the precomputation is finished, the tables and k-coherence mapping can be stored in memory or on disk until a query operation is desired.

The steps of our query method are shown graphically in Figure 3. The query stage works by looping through patches on a sparse grid with grid spacing s. We first discuss in Section 5.1 how we perform a lookup against each table, which gives a good initial correspondence for each patch. Then we discuss in Section 5.2 how we use k-coherence to improve correspondences, including a novel triangle inequality test which allows for higher efficiency. In Section 5.3, we explain how any slight misalignments in correspondences can be improved with a spatial search. Finally, we discuss in Section 5.4 how we propagate good matches to a dense grid. After this we can proceed to the next iteration of the sparse grid or output the final correspondences.

[1] This requires a mask to forbid patches from crossing image boundaries.

[2] If color patch descriptors are used, for efficiency, we actually compute descriptors and distances on demand instead of extracting the descriptor.

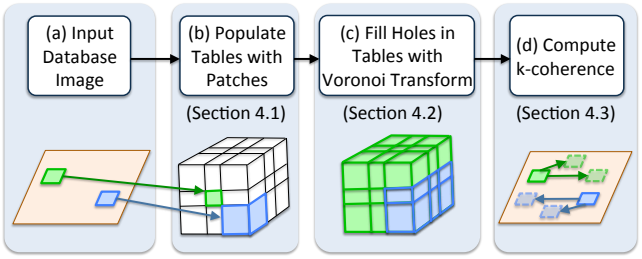


Figure 2: *Precomputation stages of our method. Tables are built from an input database image (a) that will be queried later. In (b) all patches from the database image are added to the tables. The irregular drawn spacing is intentional, because the table grid spacings are irregular. In (c) table cells with no patch are filled using our Voronoi transform. Finally in (d), k-coherence is computed for the patches in the database image.*

4 Precomputing the Data Structure

4.1 Populate Tables with Patches

In this section, we briefly review the method of Korman and Avidan [2011], which our method builds upon. We then discuss how the table dimensions are chosen, how patch descriptors are computed, and how the table is populated with patches. This process is shown iconically in Figure 2(a,b).

As a review, the method of Korman and Avidan relies on locality sensitive hashing (LSH). Locality sensitive hashing constructs T multidimensional hash tables. Database and query feature descriptors are both mapped into the hash table cells by T independent hash functions, one per table. In our case, a feature descriptor of dimension d_0 has been defined for every patch in the database and query images. This dimension may be large, however, so a table dimension parameter d must also be chosen, which is typically lower (that is, $d \leq d_0$).

In the precomputation stage, we fill T tables with patches such that a fast initial query can be made by lookups against the tables. The ith table has a targeted number of cells $c_i = c_0/2^i$. All tables have identical dimension d but different sizes along each dimension so as to achieve the targeted number of cells. The parameters introduced are the number of tables T and number of cells of the first table c_0.

To construct the ith table, we follow Korman and Avidan [2011]. If an arbitrary patch descriptor has been used, then we assume that the user has previously applied dimensionality reduction, and we thus retain the first d dimensions to map into the table. If color patch descriptors are used, then like Korman and Avidan, we reduce dimensionality using the Walsh-Hadamard basis in YUV color space by means of gray-code filter kernels [Ben-Artzi et al. 2007]. Only the first Walsh-Hadamard basis vector is used for the chroma channels, whereas the full basis is used for the luminance channel. These choices were made mainly for efficiency. Next, we create hash functions for each table in the same manner. Our hash function must assign an integer index for each of the d dimensions of the table. Like Korman and Avidan, we do this by partitioning each coordinate into variable size bins along the given dimension. The bin spacing is determined by dividing a sampled subset of database patches up into partitions with roughly equal number of samples in each. For each dimension we maintain a 1D array that maps from floating point patch descriptor to bin index.

We choose table sizes that are proportional to the range along each dimension. This is done in practice by scaling the ranges by the largest real constant ρ such that after rounding to the nearest integer, the product of all the table sizes does not exceed the target cell count

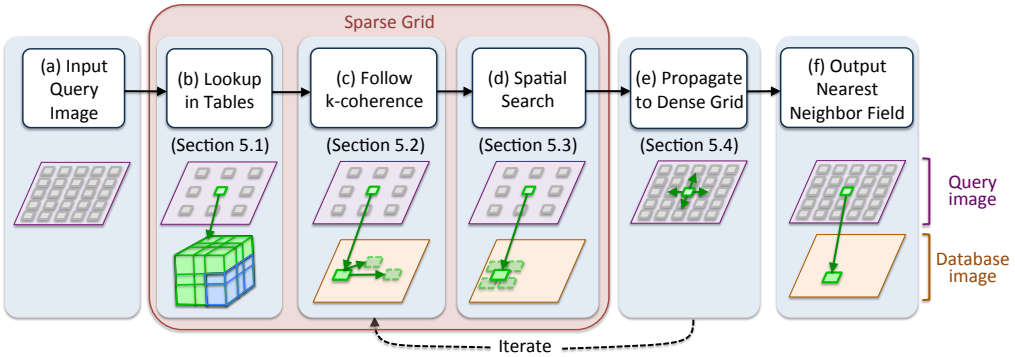


Figure 3: *Query stages of our method. Every patch in input query image (a) finds a nearest neighbor correspondence to a patch in the database image. Patch center positions are denoted by grey squares. In (b-d) the algorithm improves correspondences for query patches only on a sparse grid. In (b) a query patch determines an initial correspondence by lookups in the tables. In (c) the current correspondence is improved by trying instead k nearest neighbors that were previously pre-computed for k-coherence. In (d) small misalignments are improved by a local spatial search. In step (e) correspondences on the sparse grid are propagated to the dense grid of patches defined around every pixel, and the algorithm can iterate by returning to step (c). Finally the output nearest neighbor field (NNF) is returned.*

c_i. We experimentally determined that the table sizes selected by Korman and Avidan produce poor results for our data structure.

We also found little benefit to storing multiple patches in each table cell, so we simply traverse the database patches in random order and insert only the first patch into each table cell.

4.2 Fill Holes in Table with Voronoi Transform

We wish to push as much of the work as possible into the precomputation so that lookups are as efficient as possible. To this end, we would like to avoid the situation of "hash misses" where the query algorithm looks up a patch in the table, but the table has no entry. This scenario can be avoided by filling table cells that have no patch by using our fast Voronoi transform algorithm. This step is shown graphically in Figure 2(c).

Grid-based distance or Voronoi transform algorithms can be divided into two categories. The first category consists of raster-style algorithms like the classic Rosenfeld and Pfaltz [1966] method, which use a forward and backwards pass through the dataset. The second category consists of ordered propagation methods such as Ragnemalm [1992] which maintain a propagation front in memory, where the front fills in grid cells by increasing distance. In our case, we experimentally determined that the Manhattan distance metric where the table cell coordinates are denoted by integers resulted in equally good query performance as more costly metrics such as Euclidean distances [Maurer et al. 2003; Felzenszwalb and Huttenlocher 2012] or distances computed over the table's irregular grid center positions. Also, we found that ordered propagation methods are more efficient in practice than raster methods.

We now present our ordered propagation Voronoi transform which fills each empty cell of our table with the patch index stored in the closest cell, under the Manhattan distance metric. This algorithm assumes we start with an *originating set* of cells that were initially filled with patches. At each iteration i, it then determines a *propagation front*, which is the set of cells with Manhattan distance i to the originating set. This set can be determined by finding cells that are adjacent to the previous propagation front. This process is repeated until the cells at every Manhattan distance have been found. The challenge of this approach is that we must enumerate cells in increasing Manhattan distance from the origin, without backtracking or duplicating points. We do this by using a specially constructed *neighborhood graph list*, which allows us to use graph labels to track state information about the propagation direction. The pseudocode for the full algorithm is shown in Alg. 1.

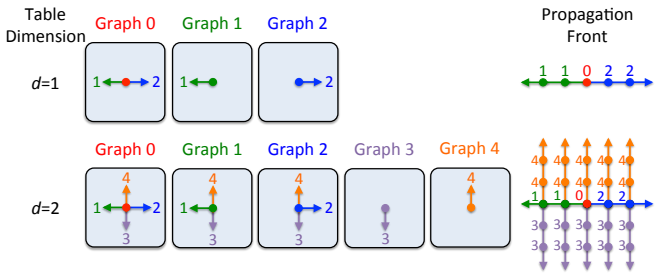


Figure 4: *To fill holes in the table we use our ordered propagation Voronoi algorithm (see Algorithm 1). These are examples of the neighborhood graph list needed for that algorithm, shown here for dimensions $d = 1$ and 2. The Voronoi algorithm creates a propagation front (shown at right), which fills the d-dimensional lattice in order of increasing Manhattan distance from label 0. The propagation front starts from a position that contains a patch (labelled with 0), and expands by following all edges of the graph at left that corresponds to the current label. For this visualization, each graph number is also given a unique color. See Section 4.2 for details and a construction of these graphs for all dimensions d.*

The neighborhood graph list required for the Voronoi transform must have two key properties: (1) it fills the integer lattice in d dimensions in order of increasing Manhattan distance from where graph 0 is expanded, and (2) it visits each cell exactly once. In Figure 4 we give neighborhood graph lists that have these desirable properties, for dimensions $d = 1$ and 2. For arbitrary dimension d we now give a construction for a list of $2d + 1$ neighborhood graphs that has these desired properties. We index the neighborhood graph list from $0, 1, ..., 2d$, and add a center vertex to each graph. For neighborhood graph 0 the center vertex has outgoing edges labelled $\{1, 2, ..., 2d\}$. Each neighborhood graph $i \geq 1$ from its center vertex has outgoing edges labelled i and $\{j, j+1, ..., 2d\}$, where $j = 2\lceil i/2 \rceil + 1$. For all graphs, the outgoing edges labelled $i = 1, ..., 2d$ have corresponding edge vectors -$\mathbf{e}_1$, +$\mathbf{e}_1$, -$\mathbf{e}_2$, +$\mathbf{e}_2$, -$\mathbf{e}_3$, +$\mathbf{e}_3$, etc. Here $\mathbf{e}_i$ is the Euclidean basis vector i. This construction generates the neighborhood graphs in Figure 4.

4.3 Compute k-coherence

The final stage of our precomputation is to find k-coherence [Tong et al. 2002]. As a review, k-coherence finds for each patch p, k-nearest neighbors that are each some minimum spatial distance δ_0 away from p. We compute the k-coherence approximately using

FLANN [Muja and Lowe 2009] with only one kd-tree, high error tolerance, exactly one tree traversal, and $\delta_0 = 5$ pixels. We sort the resulting k-nearest neighbors by their exact patch distance. For most applications it is also beneficial to store the exact patch distance alongside the k-nearest neighbors, because this allows for more efficient queries at the cost of more memory usage. See Section 5.2 for more details on how this is used during querying. The resulting final index stores both the table and the k-coherence mapping together.

Algorithm 1 Voronoi transform by ordered propagation

Input: Table T containing patch positions or empty cells.
Output: Table T where empty cells are filled with nearest patch.
Initialize propagation front (J, P, L) to the originating set:
- Table index list J with indices of non-empty cells.
- Source patch list P with corresponding patch positions.
- Label list L with all 0s.

while J is nonempty **do**
  Initialize next propagation front (J', P', L') as empty lists.
  for j = 1...$|J|$ **do**
    Find neighborhood graph for current label L_j.
    Get edge vectors E and labels ϵ from neighborhood graph (see Figure 4 for examples of the neighborhood graphs).
    for i = 1...$|E|$ **do**
      Let new table index j' be table index J_j plus edge E_i.
      if $T_{j'}$ is empty **then**
        Fill table $T_{j'}$ with source patch position S_j.
        Append (j', P_j, ϵ_i) to (J', P', L'), respectively.
      end if
    end for
  end for
  Update propagation front: $(J, P, L) \Leftarrow (J', P', L')$.
end while

5 Querying the Data Structure

Before we explain our query stage, we briefly review TreeCANN [Olonetsky and Avidan 2012], which our query method builds upon. TreeCANN works by inserting database patches into a kd-tree. To make the query efficient, only query patches on a sparse grid with spacing s are queried against the kd-tree. Other query patches have their correspondences filled in using a coherency assumption. Unlike TreeCANN, we perform a lookup on our tables instead of a kd-tree, and we query a k-coherence mapping.

Note that throughout the query section, we compute exact patch distances so we attain maximum accuracy. We do this by comparing patch distances in the original dimensionality d_0 of the descriptor. The nearest neighbor field is initialized with either infinite patch distances or else with a prior, where the patch distances for the prior are scaled by $1/(1+\kappa_t)$, where κ_t is a temporal coherence parameter (by default, $\kappa_t = 0$). This prior is used to improve temporal continuity for artistic video stylization (see Section 8.1), as well as to ensure that multi-image editing takes only downhill steps during optimization (see Section 8.3).

5.1 Lookup in Tables

The lookup operation for the tables is shown in Figure 3(b). For each table, we determine the table cell for the current query patch on the sparse grid in the same manner as in Section 4.1. Each table thus proposes a candidate correspondence. For each candidate correspondence, we initialize D_{best} as the patch distance recorded at the current NNF position times $1/(1+\kappa_s)$. Here κ_s is a spatial coherence parameter (by default, $\kappa_s = 0$). If the candidate has lower patch distance than D_{best}, then the current NNF correspondence is replaced with the candidate, and D_{best} is replaced with the proposed patch distance. Subsequently we refer to this process as *improving* the current correspondence.

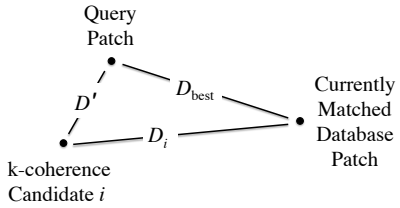


Figure 5: *The triangle inequality allows for an early termination of the k-coherence search. Here we visualize in 2D both patches and distances between patches (in reality this would be a higher dimensional vector space). See Section 5.2 for the derivation.*

5.2 Follow k-coherence

For the current query patch on the sparse grid, we next follow the k-coherence mapping to obtain potentially better candidate correspondences. This is depicted graphically in Figure 3(c). We look up in the k-coherence mapping the k-nearest neighbors of the current patch that is pointed to by the NNF. We attempt to improve the current correspondence with each of these k candidates.

For efficiency, we also apply *early termination* of the search of the k nearest neighbors list, based on triangle inequality tests. This assumes that when the k-coherence was computed (in Section 4.3), the exact patch distances between database patches were stored alongside the k-nearest neighbors. Suppose that we are examining a query patch, which has a currently matched database patch, and associated distance D_{best} (these are stored in the NNF). Suppose that a k-coherence candidate i is being examined, which has a distance D_i from the currently matched database patch. We want to know the distance D', which is the distance between the query and k-coherence candidate i. This is shown graphically in Figure 5. By the triangle inequality, we know $D' \geq D_i - D_{\text{best}}$, but if $D' \geq D_{\text{best}}$ then with certainty we can early terminate. Thus if $D_i \geq 2D_{\text{best}}$ then we can early terminate. However, the only scenario where the bound is exactly achieved is when $D_i = 2D_{\text{best}}$, which requires that the three points (representing patch descriptors) in Figure 5 be collinear. This is unlikely in high dimensional spaces, so in practice we test whether $D_i \geq \tau D_{\text{best}}$ for a constant parameter $\tau \leq 2$ instead. For efficiency it is convenient to compute squared patch distances, so the early termination test becomes $D_i^2 \geq \tau^2 D_{\text{best}}^2$.

5.3 Spatial Search

Similar to TreeCANN [Olonetsky and Avidan 2012], we found that the resulting correspondences can still be slightly spatially misaligned. Therefore, we improve the correspondence for the query patch, by searching exhaustively on a $[-r, r] \times [-r, r]$ spatial grid in the database, which is centered at the current best corresponding patch position (i.e. the position stored in the NNF).

5.4 Propagate to Dense Grid

Finally we propagate the sparse query correspondences so that the dense grid is filled in. This step is shown in Figure 3(e). We propagate correspondences from the current query patch to a surrounding $[-s, s] \times [-s, s]$ spatial grid in the query image. Therefore this propagates good correspondences not only to the surrounding sparse grid cells but also to the dense correspondences in between. If color patch descriptors are used with an L_p distance metric then

we do the same as TreeCANN [Olonetsky and Avidan 2012] and first compute a summed area table [Crow 1984] from the differences between corresponding pixels, and then for efficiency compute patch distances from this summed area table. If a custom patch descriptor is used then we do the same as PatchMatch [Barnes et al. 2009] and compute propagated patch distances directly between appropriately shifted versions of the center correspondence. We improve the current correspondences stored in the NNF with the propagated correspondences.

6 Theoretical Time and Memory Bounds

In this section, we briefly derive theoretical bounds for the time and memory usage. The query time is $O(Nd_0(T + km))$, where N is the query image pixel count. Thus, for fixed parameter settings, the query time is linear in pixel count. The precomputation takes time $O(cd + M\eta(k))$, where c is cell count in all tables, M is database image pixel count, and $\eta(k)$ is time per kd-tree query and insertion. The memory consumption is roughly $B(c + kM)$, where B is the number of bytes to store an index to a patch. This is generally higher than previous work, such as k-coherence [Tong et al. 2002] which uses BkM bytes instead. For a concrete example, at the highest accuracy setting trained in Section 7.1, PatchTable uses 1.1 GB of memory as compared to 300 MB for k-coherence. These expressions give rough indications about memory and performance. However, it is necessary to perform empirical analysis to gain greater insight. We conduct such an analysis next.

7 Experiments Demonstrating Efficiency

In this section we discuss the performance of our method on some benchmark datasets. We first discuss training and testing of the query algorithm in Section 7.1 and Section 7.2. We then examine the contribution of different system components to the query performance in Section 7.3. We finally report the time used by the precomputation in Section 7.4.

7.1 Training

Because there are a large number of parameters for our method, we first need to train it on example pairs of images to find a list of reasonable parameter settings. Note that the goal of our training is not to select a single setting of the parameters, but rather, a list of parameter settings which gives the best tradeoff between time and error. This training needs to be performed only once on a representative dataset. Subsequently, users of the data structure can simply select from the pre-trained parameter settings, to achieve any targeted time or error trade-off.

We constructed a challenging training and testing dataset that we call *one vs many*. Because a major goal of this project is to accelerate queries against databases containing high resolution or numerous images, this dataset is constructed to stress test that aspect of the system. We started by selecting the images from the *vidpairs* benchmark by Korman and Avidan [2011]. This contains pairs of similar images taken from movie trailers. We resized these images to 0.4 megapixels. We then constructed a dataset containing a sequence of query and database image pairs. The query image is the first of the pair of similar images. The database is the concatenation of a number F of frames, including the matched pair, and $F-1$ images randomly sampled from *vidpairs*. Our training set consists of the first pair of images constructed in this way, for only the lowest number of frames in the database ($F = 10$). Our test set consists of the next 10 such pairs of images, for three different numbers of frames in the database ($F = 10, 30, 60$).

Parameter	Range
Table cells c_0	$\{10^8\}$
Table dimension d	$\{6, 7, 8, 9, 10\}$
Number of tables T	$\{1, 2, 3, 4, 5, 6\}$
k-coherence k	$\{0, 1, 2, 3, 4, 5, 10, 20\}$
Query iterations m	$\{1, 2\}$
Sparse grid spacing s	$\{1, 2, 3, 4, 5, 6, 10\}$
Spatial search radius r	$\{0, 1, 2\}$
Early termination threshold τ^2	$\{1, 1.5, 2, 4\}$

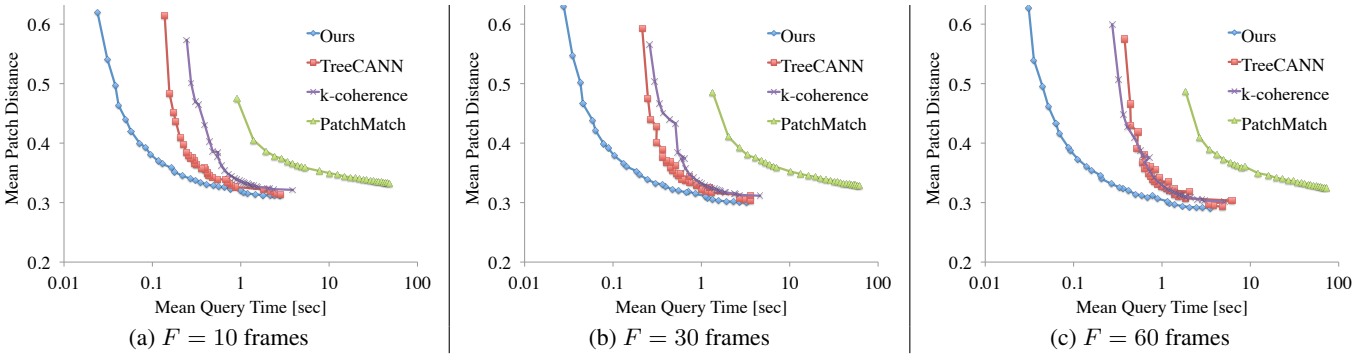
Table 1: *The range of each parameter used in our training. Note that we tried searching over a range of table cell parameters c_0. However, the search simply converges to the largest permissible table size, because higher resolution tables are more accurate.*

Our training proceeds by randomly sampling the parameters. For each parameter we chose a set of values that are reasonable and then independently sample at random several hundred parameter settings from the parameter space. We show in Table 1 the range of values for each parameter. For each setting of training parameters, we collect the mean query time t_i and mean patch error e_i, where the mean has been taken over the training images. Therefore, over all parameter settings, the collected time and error pairs are $S = \{(t_i, e_i)\}$. We compute the Pareto frontier of these: informally, this is the band of points that trades off between minimum time and error. Formally, the Pareto frontier is the set $P = \{(t, e) \in S : \forall (t', e') \in S, t < t' \text{ or } e < e' \text{ or } (t, e) = (t', e')\}$. In our case, it suffices to compute the Pareto frontier from its definition, however, it can be computed more efficiently by sorting [Kung et al. 1975].

We next refine the Pareto frontier additionally by a *crossover* operation. This can be viewed as a highly simplified genetic algorithm with only one operation. We select several hundred new samples of parameters. Each new parameter setting is constructed from two *parent* parameter settings that are adjacent on the Pareto frontier, with the selection between parent parameter values being made by a coin flip. We sort the final refined Pareto frontier by time and map the logarithm of time to a normalized speed interval $[0, 1]$. This allows a user of our code to simply select an intuitive speed setting, and not need to know anything about the training process. For our comparison we discretize the speed interval into 30 points.

We use a simpler training process to select parameters for the methods we compare against. We compare our method against TreeCANN [Olonetsky and Avidan 2012], k-coherence [Tong et al. 2002], and PatchMatch [Barnes et al. 2009]. For PatchMatch, there is only one parameter, the number of iterations, so we sample 30 settings of the iterations parameter from 1 to 100. For TreeCANN, we densely the sample query grid spacing from 1 to 10 and the database grid spacing from 1 to 100. We then extract a Pareto frontier from the training set, and sample 30 parameters in the same manner from this frontier as for our method.

For the k-coherence comparison, we must first note that Tong et al. [2002] was published in the context of a particular greedy synthesis algorithm. Therefore, the manner in which a general query procedure should be built from k-coherence is somewhat subjective. We chose to build a k-coherence query algorithm that is similar to PatchMatch [Barnes et al. 2009]. It is initialized with a uniformly randomly sampled NNF, which is then improved by one or more rounds of propagation (in alternating forward and reverse scan order), and k-coherence lookups. This is equivalent to restricting our algorithm to just the k-coherence and propagate steps, and sparse grid spacing $s = 1$. For this comparison, we also disable the use of the summed area propagation of TreeCANN because that was researched only many years after Tong et al. [2002]. We trained the k-coherence method in the same manner as TreeCANN, by sampling k from 1 to 20 and the number of iterations from 1 to 5.



(a) $F = 10$ frames (b) $F = 30$ frames (c) $F = 60$ frames

Figure 6: *Results for our* one vs many *test sets, which query a 0.4 megapixel image against a large image formed by concatenating various numbers of frames from movie trailers (10, 30, and 60 frames in plots a-c, respectively). The concatenated database image sizes are 4, 11, and 23 megapixels. We specifically exclude the precomputation time required to index the database image. The query operation in our method is up to 12× faster than TreeCANN [Olonetsky and Avidan 2012], up to 9× faster than k-coherence [Tong et al. 2002], and up to 200× faster than PatchMatch [Barnes et al. 2009]. The relative speed-ups generally increase with increasing resolution. Note that the time axis was chosen to be logarithmic because this was necessary for all competing methods to fit on the same plot. Thus, this plot measures* order of magnitude *improvements. For details, see the descriptions in Sections 7.1-7.2.*

7.2 Testing of the Query Algorithm

In this section, we present the results on the testing set and discuss our experiences with generalization and overfitting.

We show experimental results in Figure 6 for our *one vs many* dataset. In Figure 6(a,b,c) we show the test results for $F = 10, 30$ and 60 frames, respectively. The horizontal axis is logarithmic query time, averaged over the test set. The query operation in our method is up to 12× faster than TreeCANN [Olonetsky and Avidan 2012], up to 9× faster than k-coherence [Tong et al. 2002], and up to 200× faster than PatchMatch [Barnes et al. 2009]. The relative speed-ups generally increase with increasing resolution. Note that when the error tolerance is very small, the performance of our algorithm ends up being close to both TreeCANN and k-coherence. This is because for sufficiently low error tolerance, all methods end up sampling all nearby patches in "patch space." We omitted comparisons against Korman and Avidan [2011], because we found that the indexing and querying steps for that method cannot easily be separated without changing the algorithm. Note also that the precomputation time required to index the data structure has been excluded. This is because our data structure is designed specifically for applications where this cost can be paid upfront once to later enable a highly efficient query operation.

In practice we found that the large number and variety of patches in our training set prevent overfitting. For example, the performance curves in Figure 6 represent test datasets that were held out from training: they are very nearly monotonic (indicating an absence of overfitting). They are also very close to the curves produced in the testing phase if other training sets are used from the same dataset. Thus, we trained on only the single training set described previously, and used these learned parameters for our applications.

7.3 Contribution of System Components

In Figure 7, we investigate the contribution of each component to the query performance. We do this by starting from the full system and selectively disabling a number of components. Clearly, the table lookup, sparse grid, and distance transform components are critical for performance, because disabling any of them increases the patch distance significantly. The spatial search appears to improve the numerical patch distance only slightly when it is enabled. However, in practice, we found the spatial search to be important for fine-scale alignment, which is perceptually important. The k-coherence search decreases the patch distance slightly when it is

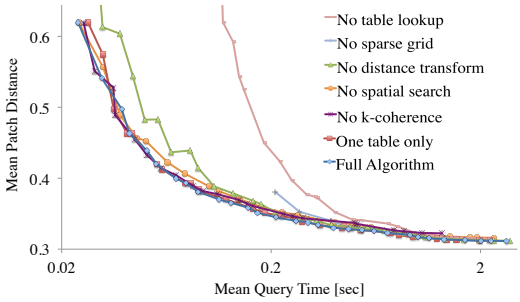


Figure 7: *We investigate the contribution of each component to the system by selectively disabling components. These results were computed over the* one vs many *dataset, for $F = 10$ frames.*

enabled, particularly in the higher running time regime. Finally, changing from one table to an arbitrary number of tables improves performance by only a small amount.

7.4 Time Used by Precomputation

We report here the time used by the precomputation, as well as a comparison between our Manhattan Voronoi transform and other alternatives. Note that we have not particularly focused on efficient precomputation in this paper.

For the *one vs many* dataset, the precomputation takes the following times for the fastest speed setting: 7 seconds for F=10 frames, 9 seconds for F=30 frames, and 11 seconds for F=60 frames. For the slowest speed setting, it takes 53 seconds for F=10, 143 seconds for F=30, and 306 seconds for F=60. For the slower speed settings, the majority of the time is taken by the k-coherence computation.

We also compared our Manhattan Voronoi transform with two alternative approaches. First, we compared with an exact Euclidean transform [Felzenszwalb and Huttenlocher 2012]. We evaluated both approaches on the *one vs many* test set. We found ours is approximately twice as fast, while incurring nearly identical error. Secondly, we compared with a Voronoi transform that used the floating-point center positions of the grid cells, instead of their integer indices. We found that using floating-point centers was significantly slower, without providing additional accuracy.

Figure 8: *We increase the resolution of 25 views of a light field by 9x, using a high-resolution 12 megapixel reference photo as a training set. We show the result of super-resolution for one view. A bicubic upsampling result is shown at left. The result of the super-resolution of Boominathan et al. [2014] is shown at center and right. At center is the result with TreeCANN [Olonetsky and Avidan 2012], which takes 1 minute. At right is the result computed with our data structure, which takes 20 seconds. This has similar error but is 3 times faster.*

8 Applications

8.1 Artistic Video Stylization

Our artistic video stylization application is shown in Figure 1. Our algorithm follows a similar by-example texture synthesis approach as Hertzmann et al. [2001]. Like Hashimoto et al. [2003], we increase temporal coherency by guiding the synthesized texture using optical flow.

Our algorithm works as follows. We first remap the luminance of the exemplar pair to match the input video luminance, in the same manner as Hertzmann et al. [2001]. Suppose our exemplar pair of images "before" and "after" the effect is applied are a and a'. We select patch descriptor dimension $d = d_0 = 8$, and produce patch descriptors using the lowest frequency gray code kernels [Ben-Artzi et al. 2007] where the first $d/2$ dimensions are from a and the second $d/2$ dimensions are from a'.

We synthesize using coarse-to-fine synthesis, in the same framework as Wexler et al. [2007] and Barnes et al. [2009]. This performs an alternating optimization between two stages: patch search and "voting" of colors between overlapping patches to establish the best estimate for colors at the next iteration. For our application, we are given the luminance of the input frame b_i and wish to synthesize the luminance of the stylized frame b'_i. For all frames we do this by iterative coarse-to-fine synthesis, matching patch descriptors in images (b_i, b'_i) to patches (a, a') that have been indexed in the table (for the coarsest level of the first frame, no b'_i image exists yet so we match to a separate table that only indexes a patches). When there is a preceding frame, at each pyramid level, we first advect the nearest neighbor field found at the previous frame according to optical flow, via inverse warping. We use the optical flow algorithm of Farneback et al. [2003] to match the current frame to the previous frame. For good results, it is important to enable both the temporal and spatial coherence parameters. These constrain synthesis along the time dimension and encourage spatially coherent regions.

We show in Figure 1 the result of our artistic video stylization application using our data structure. See the supplemental video for the full results, which shows the result of stylization with several different styles. The video also includes a comparison with Hashimoto et al. [2003], which takes 2 minutes per frame with a 352x240 video. Our method in contrast renders 1024x576 resolution videos at 1 frame/second.

We do not claim to have the highest possible quality results. Rather, this application demonstrates the efficiency of our data structure. In particular, if slower run-time is acceptable, one could improve quality by using the backwards advection of Bousseau et al. [2007] or the video cube optimization of Bénard et al. [2013].

8.2 Light Field Super-resolution

Light field cameras must make tradeoffs between spatial and angular resolution as part of their design. For example, a light field camera could capture more views, but this would require lower spatial resolution. To overcome these limitations, algorithms for *light field super-resolution* have been developed. One such method is Boominathan et al. [2014], which requires the photographer to capture a high-resolution photograph alongside a light field. The resolution of the light field is then increased by using the photograph as a training set. Specifically, the algorithm queries each patch of the light field against a downsampled version of the photograph, using simple gradient-based patch descriptors. High resolution light-field detail is then synthesized from the high resolution photograph.

In this section we show how our data structure can be used to accelerate the light field image super-resolution algorithm of Boominathan et al. [2014]. For efficiency, we modify their algorithm by reducing patch feature descriptors to 20 dimensions with PCA.

To provide ground truth data for super resolution we captured two light fields ourselves. These have 25 and 173 views. We captured these light fields at high resolution with a camera and then downsampled the inputs to our algorithm. We can therefore compare any super-resolved image with the original high-resolution photograph, to determine the error introduced by the super-resolution process. These datasets are available from our project page.

In Figure 8, we show a first result of increasing resolution by 9x on the 25 view data set we captured. With TreeCANN [Olonetsky and Avidan 2012], the super-resolution of all 25 views takes 1 minute. For equal error, PatchTable takes only 20 seconds, which is 3x faster. We measure error by taking mean L^2 distance in CIE L*a*b* color space. We explored the parameters for both methods, starting from the parameters found in the experiments in Section 7, to give optimal performance vs quality tradeoffs for both methods.

The per-view super-resolution time with our method is 11x faster than with TreeCANN, with the remainder of the time accounted for by the overhead of building the table. Thus, for the 173 view light field, the speedup over TreeCANN increases to 8x. In order to accelerate our table building time, for this application, we subsampled the k-coherence mapping with a spacing of 4. Note that in this paper we have not focused on optimizing the table building time.

8.3 Multi-image editing

By using PatchTable to store and query patches in an image collection, patch-based image editing can be performed much faster than previous methods. Although patch-based image edits like reshuffling [Simakov et al. 2008] and inpainting in a single image can be performed interactively, the process becomes more time-consuming when using an image collection. For example, when using the PatchMatch algorithm [Barnes et al. 2009], for large database image sizes, the algorithm resorts to random sampling, which causes the query time to become very inefficient.

We now explain how to use PatchTable in multi-source editing, by illustrating the process for image inpainting. Other applications work similarly. We first concatenate all the images in the collection to form a large source image L. We are given an input image I and a region R to complete. We proceed with the multi-scale approach of Wexler et al. [2007]. We build Gaussian pyramids for the input and source images. We next build a PatchTable for each pyramid level for L, which excludes patches in the hole region. The pixels surrounding the hole in the coarsest level are used to find a good initialization region in the source image, by minimizing boundary SSD color difference. Starting from the coarsest level, we look up matching patches in the corresponding PatchTable. We then generate the target image by the "voting" scheme of Wexler et al. [2007] and look up again. As in [Barnes et al. 2009], we only need to perform such EM iterations for 1–2 times in each pyramid. We upsample and repeat until we terminate at the finest level.

Using this framework, we can accelerate image completion methods like image melding [Darabi et al. 2012]. In one image melding application, patches can undergo geometric transformations like rotation and scaling. Using our method, we can directly take multiple transformed versions of the input image as the library image, to proceed with the above editing process. An example is shown in Figure 9, which only takes 3 seconds to query patches and generate results from a built PatchTable. We can also stitch images by inpainting between two images of the same scene, when given an image collection taken at the same location. To generate the best stitching results, when initializing, we check a range of possible scalings of the two input images, and retain the initialization with minimal boundary color difference. A result is shown in Figure 9.

9 Discussion, Limitations, and Future Work

In this paper we show that *an inverted index* that we call a PatchTable can be constructed, which permits efficient queries for large visual datasets. Our data structure enables applications in artistic video stylization, light field super-resolution, and multi-image editing. Our work also opens up additional potential research. On the application side, it would be interesting to explore RGBD, stereo imaging, and larger image collections. For greater scalability, it would be interesting to develop distributed implementations, cluster patches, subsample or compress the k-coherence mapping, or accelerate the k-coherence precomputation.

Sophisticated binary quantizations and embeddings have been recently developed for nearest neighbor search [Jegou et al. 2011; Hwang et al. 2012]. The application of these to image patch searches would be an interesting area of future research.

One intellectual question raised by our work is: what is the most efficient inverted index that can be constructed for patches, within reasonable precomputation time and memory constraints? We have taken a first step towards this goal by demonstrating that an efficient index can be created. However, many research questions remain open. Are table or tree data structures more efficient, or some hybrid thereof? What are the best schemes for hashing, quantization, and clustering? What are theoretical bounds on the trade-offs between memory, pre-computation time, and query time?

10 Acknowledgements

We thank the SIGGRAPH reviewers. For supporting Liming Lou, we thank Meng Xiangxu and the Chinese Scholarship Council. We thank ShanShan He for the paper video. This work was supported by the National Basic Research Project of China (project number 2011CB302205), and the Natural Science Foundation of China (project number 61120106007). Figures 1 (top) and 9(b) are licensed under Creative Commons, and have been modified.

References

ARIETTA, S., AND LAWRENCE, J. 2011. Building and using a database of one trillion natural-image patches. *IEEE computer graphics and applications 31*, 1, 9–19.

ASHIKHMIN, M. 2001. Synthesizing natural textures. In *Proceedings of symposium on Interactive 3D graphics*, ACM, 217–226.

BARNES, C., SHECHTMAN, E., FINKELSTEIN, A., AND GOLDMAN, D. 2009. PatchMatch: A randomized correspondence algorithm for structural image editing. *ACM Transactions on Graphics 28*, 3, 24:1–24:10.

BARNES, C., SHECHTMAN, E., GOLDMAN, D. B., AND FINKELSTEIN, A. 2010. The generalized PatchMatch correspondence algorithm. In *ECCV*. Springer, 29–43.

BEN-ARTZI, G., HEL-OR, H., AND HEL-OR, Y. 2007. The gray-code filter kernels. *IEEE TPAMI 29*, 3, 382–393.

BÉNARD, P., COLE, F., KASS, M., MORDATCH, I., HEGARTY, J., SENN, M. S., FLEISCHER, K., PESARE, D., AND BREEDEN. 2013. Stylizing animation by example. *ACM Transactions on Graphics 32*, 119:1–119:9.

BERTALMIO, M., SAPIRO, G., CASELLES, V., AND BALLESTER, C. 2000. Image inpainting. In *Computer graphics and interactive techniques*, ACM, 417–424.

BOOMINATHAN, V., MITRA, K., AND VEERARAGHAVAN, A. 2014. Improving resolution and depth-of-field of light field cameras using a hybrid imaging system. In *IEEE ICCP*, 1–10.

BORODIN, A., OSTROVSKY, R., AND RABANI, Y. 1999. Lower bounds for high dimensional nearest neighbor search and related problems. In *ACM symposium on Theory of computing*, ACM.

BOUSSEAU, A., NEYRET, F., THOLLOT, J., AND SALESIN, D. 2007. Video watercolorization using bidirectional texture advection. In *ACM Transactions on Graphics*, vol. 26, 104:1–104:10.

CRIMINISI, A., PÉREZ, P., AND TOYAMA, K. 2004. Region filling and object removal by exemplar-based image inpainting. *IEEE Transactions on Image Processing 13*, 9, 1200–1212.

CROW, F. C. 1984. Summed-area tables for texture mapping. *ACM SIGGRAPH 18*, 3, 207–212.

DARABI, S., SHECHTMAN, E., BARNES, C., GOLDMAN, D. B., AND SEN, P. 2012. Image melding: combining inconsistent images using patch-based synthesis. *ACM Transactions on Graphics 31*, 4, 82:1–82:12.

DATAR, M., IMMORLICA, N., INDYK, P., AND MIRROKNI, V. S. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. symp. on Computational geometry*, ACM.

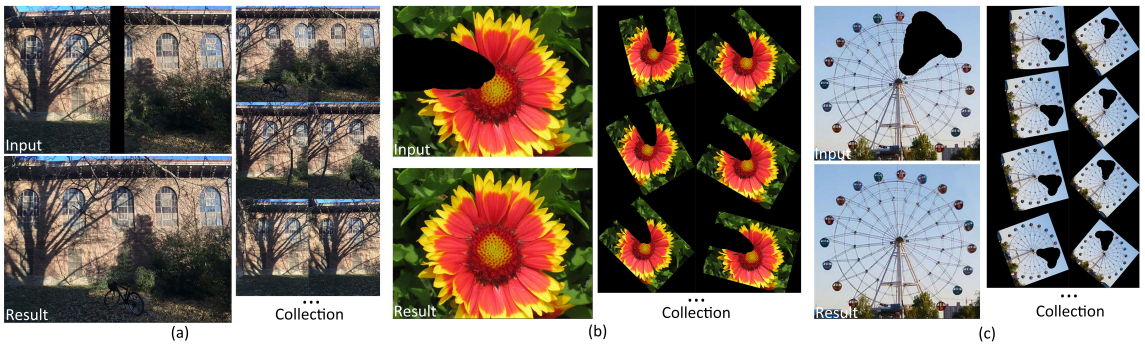


Figure 9: *Multi-image editing. (a) Image stitching using an image collection. The collection contains 10 images. (b)(c) Image completion results similar to image melding [Darabi et al. 2012].* *Credits: (b) © Fujii Hitomi; (c) Fujii Hitomi, public domain.*

EFROS, A. A., AND FREEMAN, W. T. 2001. Image quilting for texture synthesis and transfer. In *Proc. ACM SIGGRAPH*, ACM, 341–346.

EFROS, A. A., AND LEUNG, T. K. 1999. Texture synthesis by non-parametric sampling. In *IEEE ICCV*, vol. 2.

FARNEBÄCK, G. 2003. Two-frame motion estimation based on polynomial expansion. In *Image Analysis*. Springer, 363–370.

FELZENSZWALB, P. F., AND HUTTENLOCHER, D. P. 2012. Distance transforms of sampled functions. *Theory of computing 8*, 1, 415–428.

FREEMAN, W. T., JONES, T. R., AND PASZTOR, E. C. 2002. Example-based super-resolution. *IEEE Computer Graphics and Applications 22*, 2, 56–65.

HACOHEN, Y., SHECHTMAN, E., GOLDMAN, D. B., AND LISCHINSKI, D. 2013. Optimizing color consistency in photo collections. *ACM Transactions on Graphics 32*, 4, 38:1–38:10.

HASHIMOTO, R., JOHAN, H., AND NISHITA, T. 2003. Creating various styles of animations using example-based filtering. In *Computer Graphics International*, IEEE, 312–317.

HE, K., AND SUN, J. 2012. Computing nearest-neighbor fields via propagation-assisted kd-trees. In *IEEE CVPR*, 111–118.

HERTZMANN, A., JACOBS, C. E., OLIVER, N., CURLESS, B., AND SALESIN, D. H. 2001. Image analogies. In *Proc. ACM SIGGRAPH*.

HU, S.-M., ZHANG, F.-L., WANG, M., MARTIN, R. R., AND WANG, J. 2013. PatchNet: a patch-based image representation for interactive library-driven image editing. *ACM Transactions on Graphics 32*, 6, 196:1–196:11.

HWANG, Y., HAN, B., AND AHN, H.-K. 2012. A fast nearest neighbor search algorithm by nonlinear embedding. In *IEEE CVPR*, 3053–3060.

JEGOU, H., DOUZE, M., AND SCHMID, C. 2011. Product quantization for nearest neighbor search. *IEEE TPAMI 33*, 1.

KALANTARI, N. K., SHECHTMAN, E., BARNES, C., DARABI, S., GOLDMAN, D. B., AND SEN, P. 2013. Patch-based high dynamic range video. *ACM Transactions on Graphics 32*, 6, 202:1–202:11.

KOPF, J., FU, C.-W., COHEN-OR, D., DEUSSEN, O., LISCHINSKI, D., AND WONG, T.-T. 2007. Solid texture synthesis from 2d exemplars. *ACM Transactions on Graphics 26*, 3, 2:1–2:9.

KORMAN, S., AND AVIDAN, S. 2011. Coherency sensitive hashing. In *IEEE ICCV*.

KUNG, H.-T., LUCCIO, F., AND PREPARATA, F. P. 1975. On finding the maxima of a set of vectors. *Journal of the ACM 22*.

LEFEBVRE, S., AND HOPPE, H. 2005. Parallel controllable texture synthesis. *ACM Transactions on Graphics 24*, 3, 777–786.

LEFEBVRE, S., AND HOPPE, H. 2006. Appearance-space texture synthesis. 541–548.

MAURER, C. R., QI, R., AND RAGHAVAN, V. 2003. A linear time algorithm for computing exact euclidean distance transforms of binary images in arbitrary dimensions. *Pattern Analysis and Machine Intelligence, IEEE Transactions on 25*, 2, 265–270.

MUJA, M., AND LOWE, D. G. 2009. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP(1)*, 331–340.

OLONETSKY, I., AND AVIDAN, S. 2012. TreeCANN-kd tree coherence approximate nearest neighbor algorithm. In *ECCV*. Springer, 602–615.

RAGNEMALM, I. 1992. Neighborhoods for distance transformations using ordered propagation. *CVGIP 56*, 3, 399–409.

ROSENFELD, A., AND PFALTZ, J. L. 1966. Sequential operations in digital picture processing. *Journal of the ACM 13*, 4, 471–494.

RUITERS, R., SCHWARTZ, C., AND KLEIN, R. 2013. Example-based interpolation and synthesis of bidirectional texture functions. In *Computer Graphics Forum*, vol. 32, Wiley, 361–370.

SIMAKOV, D., CASPI, Y., SHECHTMAN, E., AND IRANI, M. 2008. Summarizing visual data using bidirectional similarity. In *IEEE CVPR*, 1–8.

TONG, X., ZHANG, J., LIU, L., WANG, X., GUO, B., AND SHUM, H.-Y. 2002. Synthesis of bidirectional texture functions on arbitrary surfaces. *ACM Transactions on Graphics (TOG) 21*, 3, 665–672.

WANG, M., LIANG, Y.-K. L. Y., MARTIN, R. R., AND HU, S.-M. 2014. BiggerPicture: data-driven image extrapolation using graph matching. *ACM Transactions on Graphics 33*, 6, 173:1–173:12.

WEXLER, Y., SHECHTMAN, E., AND IRANI, M. 2007. Space-time completion of video. *IEEE TPAMI 29*, 3, 463–476.

XIAO, C., LIU, M., YONGWEI, N., AND DONG, Z. 2011. Fast exact nearest patch matching for patch-based image editing and processing. *IEEE TVCG 17*, 8.

ZHU, J.-Y., LEE, Y. J., AND EFROS, A. A. 2014. AverageExplorer: Interactive exploration and alignment of visual data collections. *ACM Transactions on Graphics 33*, 4, 160:1–160:11.