Gaussian KD-Trees for Fast High-Dimensional Filtering

Andrew Adams
Stanford University

Natasha Gelfand
Nokia Research

Jennifer Dolson
Stanford University

Marc Levoy
Stanford University

Figure 1: *The Gaussian kd-tree accelerates a broad class of non-linear filters, including the bilateral (left), non-local means (middle), and a novel non-local means for geometry (right).*

Abstract

We propose a method for accelerating a broad class of non-linear filters that includes the bilateral, non-local means, and other related filters. These filters can all be expressed in a similar way: First, assign each value to be filtered a position in some vector space. Then, replace every value with a weighted linear combination of all values, with weights determined by a Gaussian function of distance between the positions. If the values are pixel colors and the positions are (x, y) coordinates, this describes a Gaussian blur. If the positions are instead (x, y, r, g, b) coordinates in a five-dimensional space-color volume, this describes a bilateral filter. If we instead set the positions to local patches of color around the associated pixel, this describes non-local means. We describe a Monte-Carlo kd-tree sampling algorithm that efficiently computes any filter that can be expressed in this way, along with a GPU implementation of this technique. We use this algorithm to implement an accelerated bilateral filter that respects full 3D color distance; accelerated non-local means on single images, volumes, and unaligned bursts of images for denoising; and a fast adaptation of non-local means to geometry. If we have n values to filter, and each is assigned a position in a d-dimensional space, then our space complexity is $O(dn)$ and our time complexity is $O(dn \log n)$, whereas existing methods are typically either exponential in d or quadratic in n.

CR Categories: I.4.3 [Image Processing and Computer Vision]: Enhancement—Filtering I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Geometric algorithms, languages, and systems

Keywords: bilateral filter, non-local means, geometry filtering, denoising

1 Introduction

In recent years, a variety of related non-linear filters have become important for various tasks in image processing, computational photography, geometry processing, and related fields. These include the bilateral filter, the non-local means filter, and various similar ad-hoc filters used for particular applications. Such filters are often highly computationally intensive. We present a framework with which to understand such filters, and an acceleration data structure and algorithm that applies broadly across all of them.

Let us begin with the simple case of a bilateral filter. Recent methods for accelerating the bilateral filter respect distance in space and in luminance. One such method, the bilateral grid [Paris and Durand 2006], does this by embedding the image as a two dimensional manifold in a coarsely voxelized three dimensional space-luminance volume, performing a 3D Gaussian blur, and then sampling again along the image manifold to construct the output. A shortcoming of this technique, as well as other recent accelerations of the bilateral filter by [Durand and Dorsey 2002] and [Weiss 2006], is that they do not respect distance in chrominance, resulting in unwanted blurring of neighbouring isoluminant regions (Figure 2).

One way to address this problem is to expand the bilateral grid to a 5D space-color volume, as described in [Paris and Durand 2009]. However, as we argue in Section 3.1, the memory required to represent the grid grows exponentially with the number of dimensions, as does the time required by each stage of the algorithm. This growth is manageable if the filter size in both space and color is large, which in turn permits the grid to be coarse. However, if the filter is small the grid must be fine, making the memory and time requirements of this approach impractical.

Bilateral filtering can alternatively be rephrased as a nearest neighbour search in five dimensions. For every (x, y, r, g, b) point in the image, we would like to gather colors from other nearby points. This suggests storing the cloud of points representing the image manifold in a kd-tree, and using approximate nearest neighbour queries (as described by [Arya et al. 1998]) to find nearby values. Unfortunately this approach scales poorly with filter size. For a large filter each pixel may be *near* to every other pixel. It would be preferable to subsample this set of neighbours in a statistically efficient manner.

Figure 2: *Bilateral filtering respecting only distance in luminance produces objectionable artifacts. On the left is a bilateral-filtered image of some roof tiles against sky respecting distance in luminance only. Note the bleeding of the blue sky into the similarly bright roof tiles (inset). On the right is the image filtered using full 3D color distance.*

To facilitate such queries, we propose a new type of kd-tree, which we term a *Gaussian* kd-tree, described in Section 2. The tree sparsely represents the high-dimensional space as values stored at points. This point cloud is derived from a reduced set of the pixels from the original image, so unlike the bilateral grid, we only ever store a 2D manifold, regardless of the size and dimensionality of the space in which it is embedded. The tree supports rapid Monte-Carlo-sampled queries to probabilistically scatter to or gather from the points, using stratified weighted importance sampling. These queries are used to implement the embedding, blurring, and sampling of the space as described in Section 2, and they do so at a computational complexity independent of the filter size and linear in the dimensionality.

Since the Gaussian kd-tree scales well with dimension, we need not constrain ourselves to three-dimensional color distances. With the ability to cheaply perform blurs weighted by higher dimensional distances, we can also accelerate non-local means [Buades et al. 2005]. Non-local means mixes pixel values with other pixels that have similar local neighborhoods, and is equivalent to a Gaussian blur on a 2D manifold embedded in a space of much higher dimensionality. Non-local means is usually made tractable by limiting it to only search for similar neighborhoods in a small local search window around each pixel. Our method is in fact slightly faster when the search is completely unbounded, as there are fewer dimensions to consider. We discuss non-local means in detail in Section 3.2.

While non-local means increases the number of *range* dimensions, we can also increase the number of *domain* dimensions to include time. In Section 3.2, we demonstrate fast non-local means for denoising space-time volumes. Non-local means is able to denoise dynamic scenes by averaging pixel values over time without requiring an explicit motion model.

Finally, the Gaussian kd-tree does not require any particular ordering or structure to the values it stores. The values need not lie on a grid, and we can mix them according to distances between any set of associated vectors we like. We therefore need not restrict ourselves to images. In Section 3.3 we apply non-local means to noisy geometry.

Our tree construction and Gaussian query algorithms are data-parallel, and so we have also implemented them on a graphics card using CUDA [Buck 2007] for a significant speedup over the CPU implementation. Implementation details are in Section 2.4.

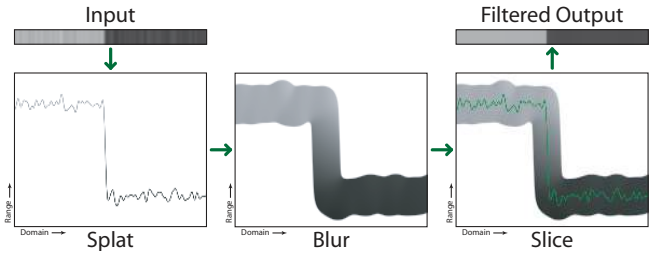

Figure 3: *Filtering that respects distance in both domain and range can be done by embedding the input signal in a higher dimensional domain-range space (splatting), performing a Gaussian filter in that space (blurring), and finally sampling the space along the original embedded manifold (slicing). Color bilateral filtering requires a five dimensional space, with range dimensions r, g, b, and domain dimensions x and y. Non-local means is conceptually the same, but requires many more range dimensions. We represent the high-dimensional space sparsely using points stored in a Gaussian kd-tree (see Figure 4).*

2 The Gaussian KD-Tree

Filtering is most generally described by replacing each value v_i in a set of size n with a linear combination of all other values v_j:

$$\hat{v}_i = \sum_{j=1}^{n} w_{ij} \cdot v_j \tag{1}$$

We assume that values are represented by homogeneous coordinates, and the homogeneous coordinate is filtered along with the others. This makes the usual division by the sum of the weights unnecessary. Weights w_{ij} are commonly computed by associating each value v_i with a position p_i in some other space, with the weight then given by a function of distance between the two positions:

$$\hat{v}_i = \sum_{j=1}^{n} f(|p_i - p_j|) \cdot v_j \tag{2}$$

For example, when performing a Gaussian blur on an image, values are pixel colors, and have (x, y) coordinates associated with them ($p_i = (x_i, y_i)^T$). The weights are given by a Gaussian function of the distance between two such positions, with standard devation σ:

$$\hat{v}_i = \sum_{j=1}^{n} e^{-|p_i - p_j|^2/2\sigma^2} \cdot v_j \tag{3}$$

When performing a bilateral filter, that weight is further reduced by a Gaussian function of distance in color space, with spatial standard devation σ_p and color space standard deviation σ_c:

$$\hat{v}_i = \sum_{j=1}^{n} e^{-|p_i - p_j|^2/2\sigma_p^2} \cdot e^{-|v_i - v_j|^2/2\sigma_c^2} \cdot v_j \tag{4}$$

A joint bilateral filter (described by [Eisemann and Durand 2004] and [Petschnigg et al. 2004]) instead uses color distance from some other image. By extending p_i and p_j to include the color distance term, Equation 4 can be more generally expressed as:

$$\hat{v}_i = \sum_{j=1}^{n} e^{-|p_i - p_j|^2/2} \cdot v_j \tag{5}$$

In this formulation, first proposed by [D.Barash 2002], pixel values v_i are now associated with positions p_i in a five-dimensional

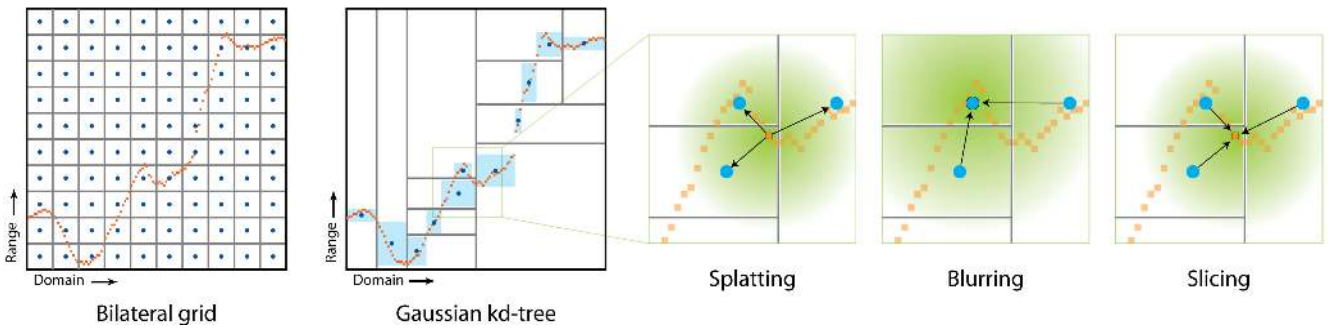

Figure 4: *We show a comparison of the bilateral grid of [Paris and Durand 2006] to our Gaussian kd-tree. Regardless of the signal (in orange), the bilateral grid stores samples on a regular grid (the blue points). The number of samples grows exponentially with dimension. The Gaussian kd-tree only stores samples along the signal, which in the case of a bilateral filter of a color image is a 2D manifold in a 5D space-color volume. We place these samples at the centroids of the bounding boxes (in light blue) of the pixels that lie within each leaf node (outlined in gray). Filtering is done by scattering pixel values onto nearby samples (splatting), gathering at each sample from nearby samples (blurring), and then gathering at each pixel from nearby samples to construct the output (slicing). Each of these stages operates using an importance-sampled query of the tree. Such a query simulates sending a number of samples, distributed in a Gaussian cloud (shown in green) around the query point down to the leaves of the tree, so that the probability of a sample arriving at a particular leaf is proportional to the integral of the Gaussian over that leaf. The effective standard deviation of the entire blur is the square root of the sum of the squares of the standard deviations of the Gaussian clouds associated with each stage.*

space whose axes are (x, y, r, g, b), scaled by the inverse of the standard deviations of the filter in the respective dimensions. We are free to scale the positions arbitrarily, so without loss of generality our Gaussian kernel always has standard deviation of one (σ is absent in Equation 5). We can transform it to an arbitrary elliptical ball with a linear transform of the position vectors. Non-local means (described by [Buades et al. 2005]), which averages pixels with others that have a similar local neighborhood, can also be expressed as Equation 5 with p_i equal to a neighborhood around pixel i. Non-local means can also be adapted to geometry using a similar formulation (Section 3.3).

Given an arbitrary set of (v_i, p_i) of size n, with p_i of dimension d, and v_i of lesser dimension, a naive computation of Equation 5 would take $O(n^2 d)$ time, as every value interacts with every other value (for example when blurring an image with a filter as large as the image). Equation 5 is a type of discrete Gauss transform, which can be accelerated using the Improved Fast Gauss Transform of [Yang et al. 2003]. The Improved Fast Gauss Transform groups vectors p_i into clusters of radius proportional to the standard deviation of the desired Gaussian, and computes Taylor series approximations of the result at each cluster center. It is an effective tool for very large radius blurs, as few clusters are needed. Unfortunately the standard deviations commonly used in filtering are small enough that there are few data points per cluster, and little benefit is derived from the clustering. We find that when applied to bilateral filtering, the Improved Fast Gauss Transform is in fact slower than a naive filter implementation for typical parameter settings.

We instead accelerate computation of Equation 5 in three ways. Firstly, interactions further than three standard deviations apart can be safely ignored (as the weights become very small) making this a collision detection problem for spheres in d-dimensional space. This suggests placing the points in a kd-tree (or a grid if d is small). Details of our tree construction are in Section 2.1. Secondly, Equation 5 replaces each value with a sum over many values. This sum can be importance sampled to avoid having to consider every interaction between $\hat{v}_i$ and some v_j. Details of this are in Section 2.2.

Thirdly, Gaussian filtering can be accelerated by computing the filter at a lower resolution and then interpolating the result. We construct a reduced set with only m positions and values, downsample to it using a Gaussian kernel of size σ_s, blur the smaller set with a Gaussian filter of size σ_b, then upsample to the original positions with a Gaussian kernel of size σ_s. As long as our m points sample the space densely enough, this will be equivalent to a single Gaussian blur of size $\sqrt{2\sigma_s^2 + \sigma_b^2}$. We consider the sampling dense enough when the maximum spacing between data points in the reduced space is σ_s. In the work of [Chen et al. 2007], these three stages are termed grid construction, low pass filtering, and slicing. We refer to them as splatting, blurring, and slicing (Figure 3). We typically set $\sigma_b = 3\sigma_s$, and scale σ_b and σ_s so that their combined effect is equivalent to a Gaussian blur of standard deviation one. If memory use is a concern, we can omit the blurring stage and achieve the same effective filter by increasing σ_s. This allows for more coarsely spaced points, but increases the number of samples required during the Monte-Carlo splatting and slicing. We derive our reduced set of size m during tree building, described below.

2.1 Building the tree

Our Gaussian kd-tree stores a cloud of m points in d dimensions, one point per leaf, and is designed to allow for fast importance-sampled queries of these points (Section 2.2). Each inner node of the tree η represents a d-dimensional rectangular cell, which may extend to infinity in one or more dimensions. An inner node stores a dimension η_d along which it cuts, and value η_{cut} on that dimension to cut along, the bounds of the node in that dimension η_{min} and η_{max}, and pointers to its children η_{left} and η_{right}. Leaf nodes contain only a d dimensional point, which lies somewhere within the cell they represent. The key difference between this tree and a conventional kd-tree is that we store η_{max} and η_{min} as well as η_{cut}. The maximum bound is computed as the minimum cut value of all ancestors which cut along the same dimension and have a larger cut value. The minimum bound is similarly the maximum cut value of all ancestors which cut along the same dimension and have a smaller cut value. See Figure 4 for a comparison of the tree to a bilateral grid.

We are now faced with the task of building a Gaussian kd-tree containing a point cloud (the blue points in Figure 4) with adequate

density around the regions where we intend to sample. Fortunately we know ahead of time that we will only ever sample at the positions used to construct the tree. For example when bilateral filtering, we will construct the tree using the (x, y, r, g, b) values of every pixel, and then scatter from and gather to those locations in five-dimensional space. Therefore, we can ensure adequate density by guarantee that every position p_i is within $\sigma_s/2$ of a point stored at a leaf node.

The goal when building a kd-tree is usually to minimize the expected time taken by a query. In raytracing, for example, this means it can be advantageous to have a highly unbalanced tree which carves off empty space and commonly hit areas early. However, we never sample in unpopulated areas, so how we deal with empty space is irrelevant, and for typical data each of our leaf nodes is as likely to be reached as any other, so the tree should be balanced.

To recursively turn a list of positions p_i into a tree, we first compute their bounding box. If the bounding box has diagonal length less than σ_s we create a leaf node, and an associated point at the center of the bounding box. Otherwise, we split halfway along the longest bounding box dimension, divide the input list into two, and continue recursively. This scheme descends to cells that have a small diagonal as quickly as possible. Another common scheme for generating balanced trees is to split on the median value along the longest dimension. In our case, an uneven distribution of points, for example those produced from an image which is mostly a single color, can in fact cause this to produce an unbalanced tree. While this has the advantage of placing the most commonly accessed leaves closer to the root of the tree, in practice we found that it did not improve performance.

2.2 Querying the tree

A query into our Gaussian tree is designed to facilitate gathers from (or scatters to) values around a given query position, for the purpose of computing an importance-sampled approximation of Equation 5. Figure 4 illustrates the process. A query takes as input a query position q in the space, a standard deviation σ around that position, and a number of samples s, and returns a list of at most s points p_i and corresponding weights w_i. If the number of samples is set to infinity, the list returned will include all points within three standard deviations of the query, with weights proportional to a Gaussian kernel of the given standard deviation ($w_i = e^{-|q-p_i|^2/2\sigma}$). If the number of samples is set to one, the list will contain a single leaf node, probabilistically chosen from all leaf nodes within three standard deviations of the query, such that repeatedly asking for a single sample and merging the resulting lists will produce the same result in the limit as asking for an infinite number of samples from a single query.

We can think of our samples as a cloud of points normally distributed around the query with the given standard deviation, although we do not explicitly represent them as such. At each inner node η we compute the expected number of samples that lie within the left and right child by computing the area of the Gaussian, truncated to with η_{min} and η_{max}, that lies on either side of η_{cut}. The Gaussian is separable, so decisions already made by nodes that split in other dimensions are irrelevant. The expected number of samples that split each way are rounded down to the nearest integer, and that many samples are assigned to the left or right child respectively. The final sample omitted by the rounding, if there is one, is probabilistically assigned to either the left or the right child.

This splitting scheme saves work compared to individually simulating every sample, resulting in a runtime which is sublinear in the number of samples, and bounded by the number of cells overlapping a query. It also stratifies the sampling, resulting in less noise in the output for a fixed number of samples.

Algorithm 1 Requesting multiple samples from a multi-dimensional Gaussian kd-tree.

```
// A quadratic approximation to the integral of a
// Gaussian of standard deviation one.
float cdfApprox(float x);

// A uniform random float between zero and one.
float urand();

class InnerNode :  public Node {
  int d;
  float min, max, cut;
  Node *left, *right;

  void Query(vector<float> q, float sigma, int samples,
             vector<Result> &results, float p=1) {
    float cdfMin = cdfApprox((min - q[d])/sigma);
    float cdfMax = cdfApprox((max - q[d])/sigma);
    float cdfCut = cdfApprox((cut - q[d])/sigma);
    float pLeft = (cdfCut - cdfMin)/(cdfMax - cdfMin);
    float expectedLeft = pLeft*samples;
    int samplesLeft = floor(expectedLeft);
    int samplesRight = floor(samples - expectedLeft);
    if (samplesLeft + samplesRight < samples) {
      if (urand() < expectedLeft - samplesLeft)
        samplesLeft++;
      else
        samplesRight++;
    }
    if (samplesLeft > 0)
      left->Query(q, sigma, samplesLeft,
                  results, p*pLeft);
    if (samplesRight > 0)
      right->Query(q, sigma, samplesRight,
                   results, p*(1-pLeft));
  }
};

class LeafNode :  public Node {
  vector<float> position;

  void Query(vector<float> q, float sigma, int samples,
             vector<Result> &results, float p) {
    float distance = Distance(q, position);
    float correctP = exp(-distance*distance/(2*sigma));
    results.push_back(Result(this, samples*correctP/p));
  }
};
```

We arrive at a given leaf node with a probability proportional to the integral of the Gaussian over the corresponding cell. This is not the correct weight, however, as our tree stores values at points, not cells. To correct for this, we keep track of the (unrounded) expected number of samples to reach this leaf, compute the probability with which we should have reached this point by evaluating the Gaussian at it, and return a weight which is the latter divided by the former. This is *weighted* importance sampling, as described by [Bekaert et al. 2000] in the context of radiosity. This correction allows us to use a piecewise quadratic approximation to the Gaussian (given by the convolution of three identical rect filters) while descending the tree, as its integral is easier to compute than that of a Gaussian. See Algorithm 1 for the relevant snippets of C++ code.

2.3 Complexity Analysis

Recall that we start with n d-dimensional data points, reduced to m during tree building, and that we use s samples when querying the tree. We filter the data set by first constructing our Gaussian kd-tree. Tree construction must process $O(n)$ nodes at each level of the tree, doing $O(d)$ work per node. Our splitting scheme balances

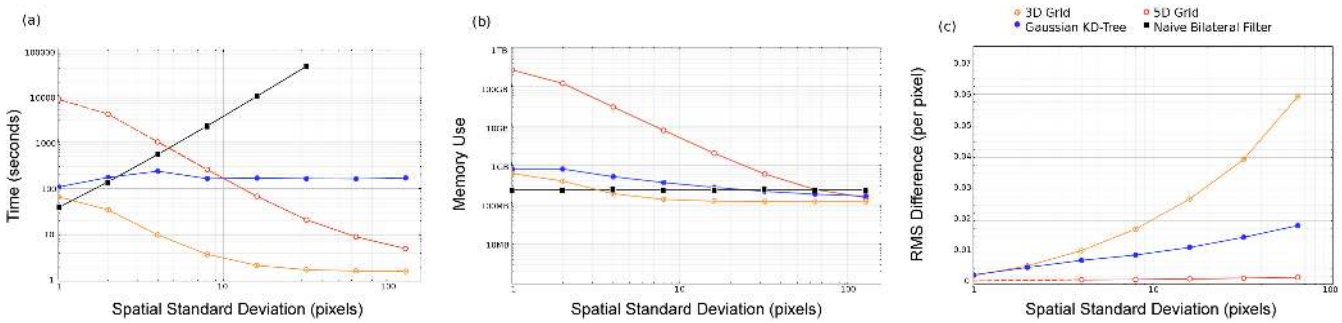

Figure 5: *Timing, memory use, and difference from a naive implementation respectively for the various implementations of the bilateral filter. All filters were run on a 10 megapixel image using a color space standard deviation of $\frac{1}{8}$. The first graph shows that the running time of the naive implementation grows large as the filter size grows, as its run time is proportional to the filter size squared. The running time of the bilateral grid (and its memory use) grow large as the filter size shrinks, as both are* inversely *proportional to the filter size squared. The Gaussian kd-tree has running time independent of filter size. The third graph shows that the 3D bilateral grid does not compute the color bilateral filter, the Gaussian kd-tree computes something similar but not exactly the same (described below), and the 5D bilateral grid almost exactly computes the color bilateral filter.*

the tree, so we can expect a depth of $O(\log m)$. Tree construction therefore takes $O(nd \log m)$ time. We then initialize the leaf nodes to have a value of zero, and do a Gaussian query with s samples for each of the n input data points to scatter values into the tree. A Gaussian query has a runtime bounded by $O(s(\log m + d))$, so this stage takes $O(sn(\log m + d))$ time. Next we blur with a Gaussian query at each leaf node which gathers nearby values, and costs $O(sm(\log m + d))$, and finally we slice with a Gaussian query at each input position for a cost of $O(sn(\log m + d))$.

This all results in a total complexity of $O(n((s + d) \log m + sd))$. Recall that $m < n$ and s is a sampling constant (typically $4 \leq s \leq 256$). This results in the simplified expression $O(dn \log n)$ given earlier. The important two features of this bound are that it is neither exponential in d (as are grid techniques) nor is it quadratic in n (as is the naive technique).

2.4 GPU Implementation

Once the tree is built, all stages of our algorithm are data-parallel across queries. With this in mind we implemented the algorithm in CUDA [Buck 2007] and ran it on an NVIDIA GeForce GTX 260. We observed a typical speedup of 10x over our single-threaded CPU implementation running on an Intel Core 2 Duo E6400 at 2.13 GHz.

There are a few interesting issues related to running the algorithm on the GPU. Firstly, the recursion of the query method in Algorithm 1 is not possible on the GPU, which has no function call stack. We convert the recursive code to iterative code by storing the arguments to pending calls to the query method in shared memory. Each thread in a block takes work from this structure when idle. If the work represents a leaf node, the thread either scatters to memory (for splatting), or gathers (for blurring or slicing), using atomic floating point adds to memory in either case. If the work represents an inner node the thread walks the samples down the tree until they reach a leaf node or diverge over a split. In the latter case, the thread continues working on the smaller of the two resulting tasks, and places the other back into the pending work structure. Although each thread is initially responsible for its own query, the sharing of pending work allows for load balancing between the threads in a block. If the pending work structure fills, threads revert to independently simulating each sample. For the case of a single sample, Algorithm 1 becomes tail-recursive, and can be converted to iteration without using extra space.

Secondly, building a kd-tree on the GPU is difficult, and has been the subject of recent research (such as [Zhou et al. 2008]). For this stage we again mimic the recursive structure of the CPU algorithm, using explicit pending work queues stored in global graphics memory. Our algorithm builds the tree in stages in a breadth-first manner using a pair of queues containing build jobs. A single build job is an array of points and a pointer to a parent node to which the resulting subtree should be attached. For the initial few large build jobs, the CPU runs the algorithm recursively, using GPU kernels to accelerate the tasks of bounding box computation and sorting data over a pivot. Once there are enough build jobs to parallelize across them effectively, the GPU takes over. In each stage all the jobs from the first queue are processed, creating the same number of nodes, and the new jobs created to build any children are placed on the second queue. In between stages the queues are swapped. We parallelize build jobs over thread blocks rather than threads, treating each thread block as a SIMD unit, in which each thread concerns itself with a single dimension. In a final phase after construction, each node η in parallel walks up the tree to the root to calculate η_{min} and η_{max}.

The graphics card typically has less memory than the host system, so it may not be able to fit all n vectors in memory for tree building, even if the final tree only uses $O(md)$ memory. To overcome this, we build the tree using a large random subset of the data. We then perform a two-phase query to include the vectors that were not initially selected. First we parallelize across input vectors and send each to the leaf node that contains it. Then we parallelize across leaf nodes, and process the vectors that arrived at each to locally extend the tree if necessary. If the initial random subset selected covers the space well, we will typically see only a small growth of the tree.

If the data set is too large to fit into host memory, we can pick some of the position dimensions with large extent and subdivide the data into overlapping blocks, processing each block individually. Typically the dimensions with largest extent will be those representing spatial coordinates, making blocking easy.

3 Applications

We have described a high-speed, low-memory way to compute a filtering of a set of values (Equation 5), such that every value is replaced with a weighted linear combination of all other values, with

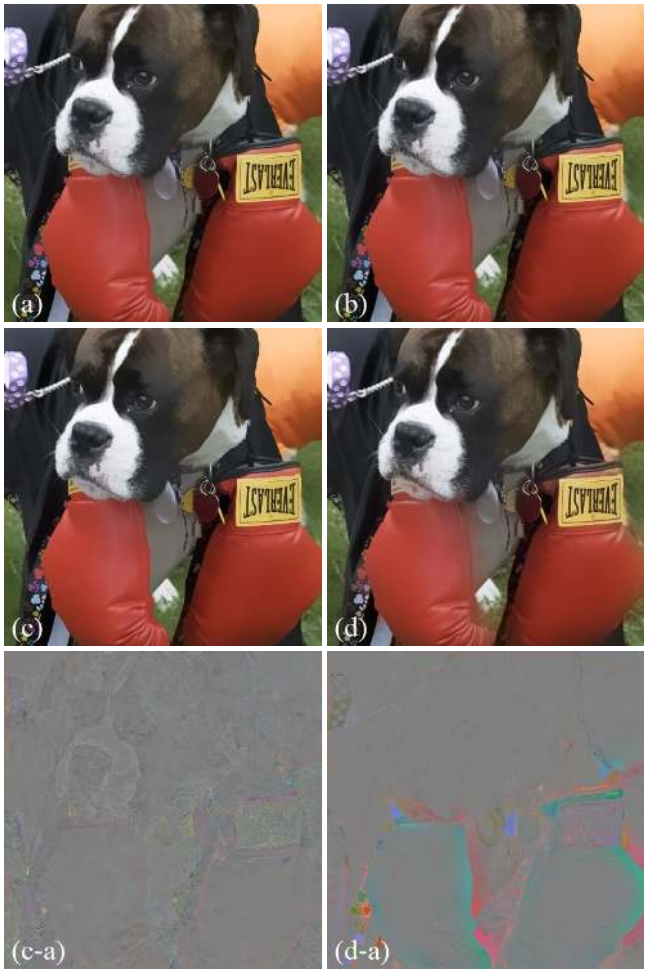

Figure 6: *Different methods of computing the bilateral filter produce different results. This filter had a spatial standard deviation of* 16 *pixels and a color space standard deviation of* $\frac{1}{8}$*. (a) Computed using the naive algorithm. (b) Computed using a 5D bilateral grid. The result is nearly identical. (c) Computed using a Gaussian kd-tree. Artifacts from the random sampling are visible in some places in the image, and the result is very slightly more aggressive in preserving edges than the naive, as can be seen in the amplified difference image at bottom left. (d) Computed using the 3D bilateral grid. It works perfectly on the uniformly brown head, but displays unwanted blurring around the boxing gloves where there is a strong chrominance boundary. The unwanted color transfer is shown in the amplified difference image on the bottom right.*

the weights given by a Gaussian function of the distance between arbitrary vectors associated with each value. This is a very general method, which we will now apply to three particular problems, each of which has been solved in its own separate way in the past.

3.1 Bilateral Image Filtering

The bilateral filter, first proposed in the work of [Aurich and Weule 1995], [Tomasi and Manduchi 1998], and [Smith and Brady 1997], is a non-linear filter that replaces each pixel value with a weighted average of all pixel values, with weights respecting distance in both position and color. With small spatial extent it is an effective way to denoise, and with large spatial extent it is used for decomposition into base and detail layers. [Durand and Dorsey 2002] accelerated the filter by using subsampling in conjunction with piecewise linear approximation in the spatial domain. [Paris and Durand 2006] then introduced the idea of expressing the filter as a linear filter in a higher dimensional space, by explicitly representing the filter in a higher dimensional data structure. [Chen et al. 2007] accelerates the filter in the same way by treating it as a three-dimensional bilateral grid, and also applies that grid to related problems. [Weiss 2006] takes a different approach, and accelerates the filter by maintaining partial histograms during a scan of the image, which makes it cheap to compute a local histogram at any one pixel on the fly, from which a bilateral filter can be approximated. [Eisemann and Durand 2004] and [Petschnigg et al. 2004] introduced the idea of the cross or joint bilateral filter, where an image can be filtered with respect to color distances in a different image. In the bilateral grid, this is equivalent to decoupling the position of the image manifold in the volume from the values stored along it.

The most common implementation of a bilateral filter directly evaluates the appropriate weighted sum at each pixel. It runs faster than the $O(n^2)$ implied by Equation 5 by only considering neighbouring pixels within some small number of spatial standard deviations. This approach is fine for small spatial standard deviations, but running time scales with the square of the spatial standard deviation; thus, processing a 10 megapixel image using a spatial standard deviation of more than 16 pixels takes hours (Figure 5(a)).

While the grid-based accelerations have superior scaling with filter size, they have the disadvantage of only respecting distance in luminance, rather than full color distance (Figure 2). Fortunately, human eyes are more sensitive to luminance variation than chrominance variation, and demosaicing algorithms exploit this, so that most photographs at full resolution have locally constant chrominance. When the spatial standard deviation is small enough for this condition to hold, but large enough for a naive algorithm to run slowly, a 3D bilateral grid performs well (see Figure 5).

One way to respect full color distance is to extend the bilateral grid to five dimensions, representing the two spatial and three color dimensions in an image, as described by [Paris and Durand 2009]. We implemented such a grid, using tent filters for splatting and slicing, and a Gaussian for blurring, with filter widths designed so that the combined effect of the three is an approximate Gaussian blur of standard deviation one. While the results are very close to the naive bilateral filter (Figure 5(c)), the memory usage is prohibitive for small filter sizes (Figure 5(b)), as samples in the grid are placed proportionally to the filter size. Runtime is proportional to the total size of the grid, as the blur stage must process every grid point, and so the computational cost is also prohibitive for small filter sizes (Figure 5(a)). Furthermore, running time and memory use both scale exponentially with d, so the grid generalizes poorly to higher dimensional filters.

The Gaussian kd-tree We now apply the Gaussian kd-tree described in Section 2 to this task. The value vectors v_i are the (homogeneous) pixel colors, and the position vectors p_i are their locations in (x, y, r, g, b) space, scaled by the inverse of the respective standard deviations. After performing some exploratory experiments, we settled on 8, 32, and 16 samples with standard devations of $1/\sqrt{11}$, $3/\sqrt{11}$, and $1/\sqrt{11}$, for splatting, blurring, and slicing respectively. We did not use our faster GPU implementation for these experiments, so that we can provide a fair comparison against the other methods.

We can see from Figure 5(b) that the memory use is initially bounded, and then drops gradually with higher spatial standard deviations as the space is more coarsely sampled. The timing results in Figure 5(a) show that, of the methods that respect color distance,

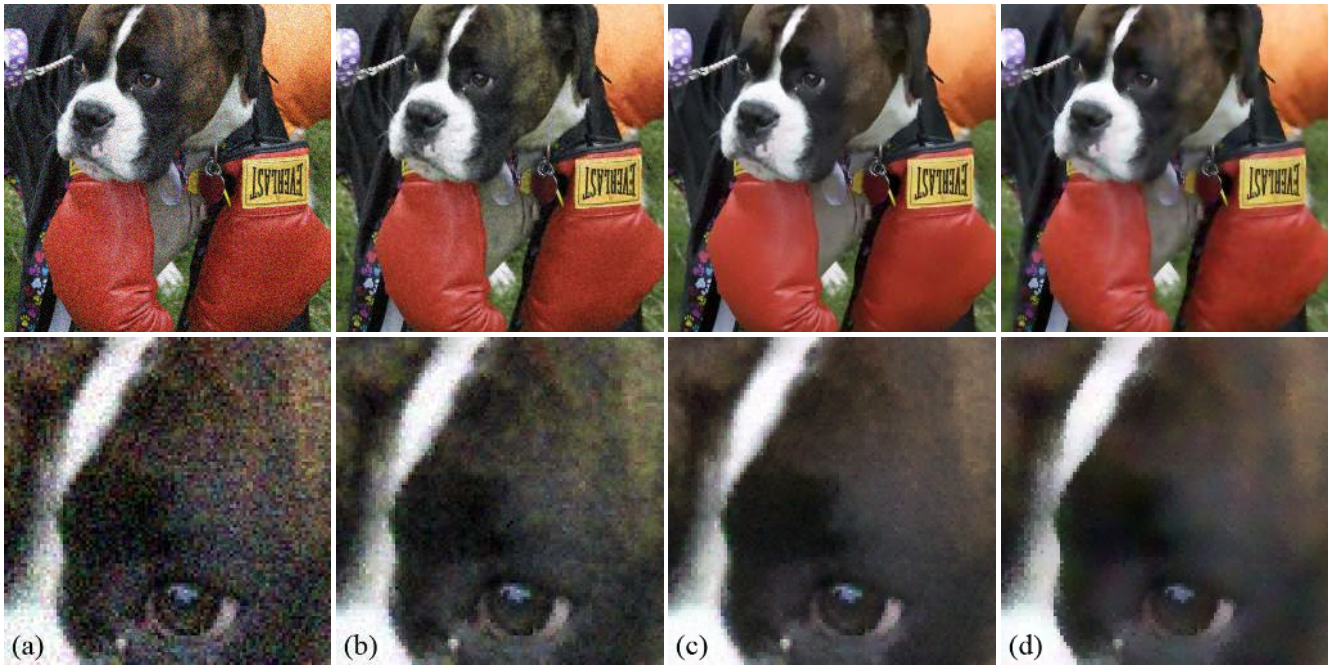

Figure 7: *Non-local means denoises without removing detail. (a) At the top we show a crop of the noisy input. Below it is a further crop of a portion of the dog's head. (b) The same crops of the output of non-local means, implemented using our Gaussian kd-tree, with* 9×9 *patches, reduced with PCA to* 25 *dimensions, using a filter with standard deviation in patch space of* 0.2*, and no spatial term. It mixes the dog's head with far away grass, turning it greenish. (c) Next we add a spatial term with standard deviation of* 10 *pixels to prevent such mixing, and expand the patch space standard deviation to* 0.3*, producing a better result. (d) Finally, we show the result of a bilateral filter that removes an equivalent amount of noise. The bilateral filter produces an inferior result to non-local means. Some detail has been lost, yet chrominance noise remains. Furthermore, as our algorithm scales linearly with dimension, non-local means is not significantly more expensive to compute.*

we perform the best at moderate standard deviations. This effect is further illustrated in Figure 6, which shows the outputs produced by the various techniques.

Figure 5(c) shows us that what we compute is not quite the bilateral filter. The difference is related to the sparsity of our sampling. Consider a bilateral filter of a hard edge between a black region and a white region. All the samples in our tree are either of black or white pixels. A single large Gaussian blur in range-domain space may allow for some energy transfer between the two, slightly graying either side of the boundary. However, each stage of our algorithm represents a smaller blur, and it is possible for no energy to cross the boundary during any stage, leaving the input unchanged. If instead there were a line of gray pixels along the boundary to serve as a stepping stone, then the combined effect of the three stages could transfer energy between black and white pixels via those gray pixels.

Our version of the bilateral filter therefore respects hard boundaries slightly more than soft ones, which may in fact be a benefit in most applications. If this behaviour is undesirable, one can set $\sigma_s = 0$, which forces $n = m$ (i.e. we allocate one leaf node per input pixel), and then set $\sigma_b = 1$, so that the full blur happens during the blur stage only. With these parameters, the typical RMS difference between our output and the naive output drops to 0.002, or half of the quantization limit. However, under these settings more samples are required for splatting or slicing, reducing performance.

Conclusion The graphs tell a mixed story. We recommend using the naive approach for small spatial standard deviations when accuracy is important. When a locally-constant chrominance assumption holds across the filter size desired, the three-dimensional bilateral grid is the best option. For very large filters, the five-dimensional grid is superior. For moderate filter sizes, with spatial standard deviations between two and ten pixels, the Gaussian kd-tree performs the best. $d = 5$ appears to be a tipping point, at which grid methods are comparable to the tree. As we scale d higher in the following sections, we begin to see results much more difficult to obtain with existing methods.

3.2 Image Denoising with Nonlocal Means

Now that we can use three-dimensional color distances in an accelerated bilateral filter, it is natural to ask what other dimensions we could add to the position vectors. One could include local gradients as well, or the output of any set of local filters. As one adds dimensions to the position, and in this way becomes more specific about what constitutes a good match between two pixels, it is desirable to simultaneously extend the spatial extent of the filter, to search for similar pixels over a wider area. The limit of this expansion is the non-local means filter, though an effective filter for a given purpose may lie anywhere along the continuum between the bilateral and non-local means.

Non-local means, first proposed by [Buades et al. 2005], averages pixels with other pixels whose local neighborhoods contain similar image features. That is, non-local means evaluates Equation 5 with v_i set to the (homogeneous) color of pixel i and p_i set to a window of pixel values around pixel i. Non-local means is thus very effective for self-similar images. An image need not contain explicitly repeated elements to be self-similar. For example, every pixel along a straight edge between two flat regions has a similar local neighborhood to every other pixel along that edge. Non-local means is particularly effective at denoising without removing detail because it makes no smoothness assumptions in its image model.

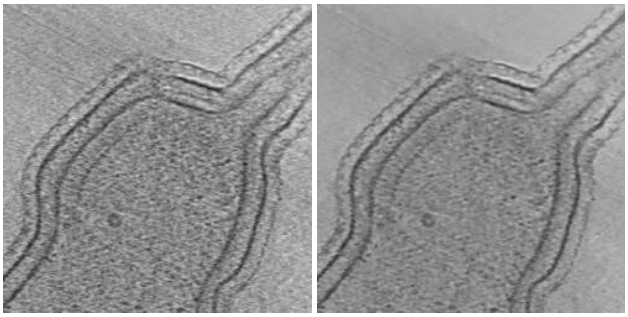

Figure 8: *One slice of the input (left) and output (right) of non-local means applied to a* $500 \times 500 \times 240$ *volume data set gathered using cryo-electron tomography [Amat et al. 2008]. Patches were* $5 \times 5 \times 5$ *voxel subvolumes reduced to the most important* 16 *dimensions with PCA. The spatial standard deviation was* 10 *pixels, and the patch space standard deviation was* 0.1*. The resulting volume is easier for biologists to analyze than the input.*

Non-local means, however, is intractably slow in its basic form, as every image patch must be compared with every other patch, resulting in a complexity of $O(f^2n^2)$ for n pixels and $f \times f$ patches. The simplest way to ameliorate this is to reduce the search to a small local search window. More sophisticated approaches, such as [Brox et al. 2008] have focused on accelerating the patch search over the entire domain, by clustering the patches in a tree structures of various kinds. When applied to non-local means, the Gaussian kd-tree can be viewed as a member of this family of techniques. As discussed above, gridded approaches will not work here, due to the exponential memory use and computational complexity with respect to dimension.

The Gaussian kd-tree can be used to accelerate non-local means in exactly the same way it accelerates bilateral filtering, using the same kd-tree implementation. To do this, we construct the position vectors p_i out of patches around each pixel in the input, rather than the (r, g, b, x, y) vectors used for bilateral filtering. If the patches are large and memory is limited it may be difficult to explicitly construct and store all of them, and they can instead be gathered from the input image as needed during splatting and slicing.

At dimensionalities above around 50, for example when using large patches, the Gaussian kd-tree begins to exhibit poor sampling behavior. By the time any given sample has reached a leaf node, it has been split over $\log m$ different partitions. If $d \gg \log m$ there are many dimensions over which no splitting was done, and the chance of the point stored in the cell being close to the query point becomes low. To ameliorate this, as a preprocess we perform PCA over the set of patches to compute a set of filters that best capture the variance in a patch. PCA helps even if we do not use it to reduce dimensionality, as the transformation decorrelates the dimensions, and orders them from most to least variance across the data set. This allows kd-tree to split on dimensions with the largest variance first, which are now axis-aligned. Once a query makes it down to a leaf node, the dimensions that have not yet been split over are those with the lowest variation over the data set, so it is much more likely that the query point is close to the point stored in the leaf node. In practice, it is also advantageous to simply drop the dimensions with small eigenvalues. This speeds up the algorithm without noticeably changing the results. For very noisy scenes (such as the top of Figure 9) it in fact improved the results, as it slightly denoises the position vectors.

Figure 9: *Denoising using non-local means on a burst of images is able to average information across frames without computing any global alignment or optical flow. On the top is a burst of 16 shots of a man looking around and waving a toy. This is denoised using non-local means with* 21×21 *patches reduced to* 16 *dimensions with PCA. The spatial standard deviation was* 10 *and the patch standard deviation was* 0.23*. The eighth frame from the burst, before and after denoising, is shown. At the bottom is a noisy burst of 10 shots of a dog walking through foliage, with the fifth frame shown enlarged. This is a deforming textured scene with the typical amount of noise from a point-and-shoot camera. Non-local means performs well at removing the noise (see for example the dog's nose), although it removes some fine details in the fur. The patch size was* 11×11 *again reduced to* 16 *dimensions, the spatial standard deviation was* 30 *and the patch standard deviation was* 0.1*.*

Results We use our algorithm to apply non-local means to several types of data. Figure 7 shows our algorithm used to generate a comparison between non-local means, non-local means with a spatial term added, and bilateral filtering.

In Figure 8 we show results on a volume data set of a bacteria produced by cryo-electron tomography by [Amat et al. 2008]. Such volumes are typically very noisy, because bombarding specimens with large numbers of electrons tends to alter them, meaning few electrons must be used, limiting the signal-to-noise ratio obtainable.

Non-local means is able to robustly use nearby similar information to improve an image. The easiest way to acquire similar information on a digital camera is to take a second noisy photograph of the same scene, or an entire burst of shots. This property makes non-local means excellent for denoising from a burst of unaligned shots, which may contain objects that deform or change their appearance. Existing methods for denoising from multiple shots or videos either globally align and average (such as the work of [Telleen et al. 2007] and [Adams et al. 2008]) or search for explicit block matches (such as [Avanaki 2006]). These methods are fairly brittle. A more robust approach is the work of [Bennett and McMillan 2005] which averages in either space or time, as appropriate, but does not denoise moving textured objects. [Buades et al. 2008] finds that applying non-local means to the volume produces better output than explicit searches for matching blocks or pixel trajectories. Figure 9 shows results from applying non-local means to such two such bursts.

Figure 10: *Non-local means for geometry smoothing. Left: the dragon model corrupted by Gaussian noise with $\sigma = 1/2$ mean edge length Middle: smooth base layer produced by Laplacian smoothing of the noisy mesh; Right: non-local means result. This mesh contains 300K vertices and it took under a minute to perform the denoising.*

The running time for our implementation of non-local means is typically spent half performing the patch PCA (which is implemented as a stack of convolutions accelerated on the GPU), and half computing the denoising. The time for each portion is typically under one minute per megapixel at 16 dimensions, regardless of the size of the search.

3.3 Geometry Filtering

Due to the success of 3D range acquisition techniques, denoising of meshes and point clouds is an active research area in geometry processing. The goal is the same as in image denoising, to remove noise while best preserving the underlying signal, which in this case comes as a set of 3d points (potentially with mesh connectivity) sampled from some surface.

Isotropic geometry denoising methods, such as [Taubin 1995], perform the same amount of smoothing irrespective of whether sharp features such as edges and corners are present in the input, which results in these features appearing rounded-off in the result. Recently, several approaches for feature-preserving denoising have been proposed based on geometry diffusion [Desbrun et al. 1999], projections [Fleischman et al. 2005], the bilateral filter [Jones et al. 2003] [Fleishman et al. 2003], and its extension to non-local means [Yoshizawa et al. 2006]. In this section, we will show that, since the Gaussian kd-tree does not place any structural constraints on input data, it can be used for filtering of geometry.

Refer once again to Equation 5, which states that to produce an output value $\hat{v_i}$ at a point p_i, a generalized bilateral filter averages together a set of values v_j weighted by a Gaussian function of the distance between p_i and each point p_j. The main difficulty in adapting this bilateral filtering framework from the image domain to 3d geometry is that, in general, our input is a set of 3d point coordinates (x_i, y_i, z_i) describing a surface, which does not come parameterized over some regularly sampled domain. Therefore, unlike for images, there is no natural decomposition of the input into positions (p_i's) and values (v_i's), as required for Equation 5.

Two approaches to decompose a set of 3d points into the spatial and signal domains for bilateral filtering of meshes have been proposed recently. [Jones et al. 2003] computes the filtered coordinates of each mesh vertex as a weighted sum of its projections onto the mesh faces within the point's neighborhood. In this case, the positions in Equation 5 are the centroids of the neighboring triangles, and the values are the projections. And alternative method was proposed by [Fleishman et al. 2003], which uses tangent planes at each vertex to build a local parametrization of the geometry. For each neighboring vertex, the projection onto the tangent plane becomes the position, and the height above the tangent plane becomes the value. The filtered vertex is then moved along its normal by the averaged height computed by the bilateral filter. The approach of [Fleishman et al. 2003] has been extended to non-local means in [Yoshizawa et al. 2006] by adding a geometric descriptor to each vertex in the mesh.

The above approaches solve the problem of separating 3d point coordinates into the spatial and data components by representing the neighbors of each vertex in its own local coordinate system. However, there are two problems in using the decompositions of [Jones et al. 2003] and [Fleishman et al. 2003] in Equation 5. First, since local projections are used, each value v_j in the sum to produce the output value v_i depends both on coordinates of vertex i and vertex j, which does not give us a one-to-one mapping between values and positions required for efficiently computing the blur. The second problem is that the parameterizations only make sense locally around each vertex since they use tangent plane approximations of the geometry. For non-local means denoising, we need to average values that are potentially far apart in space, as long as the local geometry looks similar, for example the scales on the front and back of the dragon in Figure 10. However, the projection of a point j on the back of the mesh onto the local frame of a point i on the front can be very far away from i, especially if surface orientation at i and j are different, even if local geometry is similar. We would prefer a global notion of value that makes sense across the whole mesh.

Computing Global Positions and Values To produce a globally meaningful value that we can average across all points on the mesh, we will treat feature-preserving mesh smoothing as a problem of adding back lost detail to a smooth base layer. This approach is often used in mesh editing [Sorkine et al. 2004], where a smooth base layer is used to produce large-scale geometric deformations, and fine detail is then added back as offsets from the deformed geometry.

Let $\mathcal{M}$ be the input mesh, let $\mathbf{x}_i, i = 1 \ldots n$ be the vertex positions, and $\mathbf{n}_i$ be the vertex normals computed by averaging the face normals for faces incident on vertex i. We apply Laplacian smoothing to $\mathcal{M}$ to produce the smooth base layer $\tilde{\mathcal{M}}$ with vertex positions $\tilde{\mathbf{x}}_i$ and normals $\tilde{\mathbf{n}}_i$. Laplacian smoothing successfully removes the noise from the mesh, but will smooth across sharp features. The difference between the vertex coordinates of $\mathcal{M}$ and $\tilde{\mathcal{M}}$ gives us the noisy detail layer, $\mathbf{d}_i = \mathbf{x}_i - \tilde{\mathbf{x}}_i$. The resulting detail vectors are translation invariant, however they are still dependent on the surface orientation at $\mathbf{x}_i$. To achieve invariance to rigid transformations, we express each detail vector $\mathbf{d}_i$ in the principal coordinate frame of

Figure 11: *On the left is a carved box model corrupted by Gaussian noise with $\sigma = 1/4$ mean edge length. Next is the smooth base layer produced by Laplacian smoothing. Next is the output of non-local means with standard deviation in spin image space on 0.05. It took under a minute to denoise this 700K vertex mesh with our algorithm. Finally a closeup of the detail produced by non-local means (top) and a bilateral filter that produces equivalent smoothness (bottom). While the bilateral filter still keeps some edge information, more detail is visible in the non-local means result.*

the vertex i of $\tilde{\mathcal{M}}$. That is, for each vertex on the smoothed base layer, we compute the coordinate frame $(\tilde{\mathbf{n}}_i, \tilde{\mathbf{k}}_i^1, \tilde{\mathbf{k}}_i^2)$, where $\tilde{\mathbf{k}}_i^1, \tilde{\mathbf{k}}_i^2$ are the directions of minimum and maximum curvature. These vectors are computed on the smoothed base layer, so they are not corrupted by noise. The final offset vector for each vertex is computed as the projection of $\mathbf{d}_i$ into the principal curvature frame, which is then expressed in homogeneous coordinates to give the value vector for each vertex:

$$v_i = (\langle \mathbf{d}_i, \tilde{\mathbf{n}}_i \rangle, \left\langle \mathbf{d}_i, \tilde{\mathbf{k}}_i^1 \right\rangle, \left\langle \mathbf{d}_i, \tilde{\mathbf{k}}_i^2 \right\rangle, 1) \quad (6)$$

This gives us the set of n values v_i, which are meaningful globally and can be averaged across the entire input using the regular vector addition. The second component of the non-local means denoising is the position values in Equation 5, which should be related to some measure of neighborhood similarity. Our neighborhood descriptor should be robust to noise, invariant to rigid transformations, and represented as a vector in R^n. We use the well-known spin image descriptors [Johnson and Hebert 1999], which are orientation-invariant histograms of cylindrical coordinates of points within a given neighborhood. For a point $\mathbf{x}_i$ with normal $\mathbf{n}_i$, the spin value (α, β) of a point $\mathbf{x}_j$ in the neighborhood of $\mathbf{x}_i$ is defined as:

$$(\alpha, \beta) = (\sqrt{||\mathbf{x}_j - \mathbf{x}_i||^2 - \langle \mathbf{n}_i, \mathbf{x}_j - \mathbf{x}_i \rangle^2}, \langle \mathbf{n}_i, \mathbf{x}_j - \mathbf{x}_i \rangle) \quad (7)$$

To build a spin image of a surface patch around $\mathbf{x}_i$, we quantize the pairs (α, β) into a set of bins. Since the spin images are most sensitive to the orientation of the surface normal, we use the normals $\tilde{\mathbf{n}}_i$ from the base layer and the point coordinates from the original mesh to form the spin images. If the normals are robust, the rest of the computation is relatively robust to noise due to the binning that is performed to compute the spin image. We use 5 bins for the values of α and 10 bins for the values of β with the bin size equal to the sample spacing of the mesh as recommended in [Johnson and Hebert 1999]. This gives us 50-dimensional position vectors p_i. Once the values are positions are computed, we blur using the Gaussian kd-tree to produce the smoothed detail vectors, which are then added back as offsets to the base layer to produce the final denoised result.

Results We apply our non-local means denoising algorithm to several examples of meshes corrupted by Gaussian noise. In all examples, we use 20 iterations of Laplacian smoothing to produce the base mesh, and smooth the detail layer with the filter of standard deviation 0.05 in spin image space.

Figure 10 shows the results of applying non-local means smoothing to a dragon model corrupted by Gaussian noise. The denoising is particularly effective at recovering self-similar areas of the mesh such as the scales and the back ridge of the dragon. In Figure 11 we apply non-local means to a noisy model with many sharp features and fine detail. Notice that we are able to maintain sharp edges of the carvings on the box, as well as recover the fine detail in the petals. This model has many planar areas, so it is also particularly suitable for the algorithm of [Jones et al. 2003], which uses local planar approximations. On the right of Figure 11 we show the results of applying bilateral smoothing to produce equivalent amount of noise reduction in the flat areas. While bilateral smoothing preserves edges better than Laplacian smoothing used to produce the base layer, non-local means is able to recover more detail in the petals.

In this section, we demonstrated that the Gaussian kd-tree can be used for non-local means smoothing of geometry. Our method relies on decomposing the input into a base and a detail layer. Such decompositions have also been addressed in the context of mesh parameterizations [Sheffer et al. 2006], and we expect that a variety of parametrization and decomposition approaches can be used in our framework. In addition, investigating different geometry descriptors in the context of non-local means denoising is a promising area of future work. Finally, we expect that a similar method can be applied to point cloud denoising.

4 Conclusion and Future Work

We have described a novel method for computing the broad class of non-linear filters which can be described by Equation 5, based on weighted importance sampling of a modified kd-tree. This class of filters includes bilateral filters, joint bilateral filters, non-local means filters, and related filters in which values are averaged with other values that are considered nearby in some high-dimensional space. For bilateral filtering, we compare this method to a 5D extension of the bilateral grid of [Paris and Durand 2006], and find that which method is superior depends on the filter size used. For higher dimensional filters, such as non-local means, our tree-based filter exhibits excellent performance, as its runtime and memory use both scale linearly with dimension. Our method requires no particular structure to the input, so we also apply it to the task of denoising

geometry to produce a novel non-local means filter for meshes.

Several issues remain to be addressed in future work. Firstly, our tree building takes a significant fraction of our total runtime, and so we use a very simple splitting scheme. It is possible that a more sophisticated building algorithm could improve the runtime of later stages enough to justify its cost.

Secondly, in cases with n values and many more than $log(n)$ dimensions, the splitting that takes place in our tree does not adequately constrain a sample's location before it reaches a leaf, and many samples are returned with very small weights attached. In this work, we solved this by throwing away the least important dimensions with PCA, but it may be that other tree structures are still amenable to weighted importance sampling while more strongly constraining sample locations. It may also be beneficial to store values at leaf cells, rather than at a point somewhere within them. This would improve the complexity of the algorithm by removing the distance evaluation currently required to compute the correct probabilities at the leaf nodes, and making the importance sampling exact rather than weighted, but it would compute a different function of the values - one far more dependent on the specific way in which the tree was built.

Finally, tree traversal is an extremely irregular algorithm, and the speedup we observed from our GPU implementation is significantly less than theoretically possible. More intelligent software caching of portions of the tree and other data structures may speed this up further.

Acknowledgments

This work was supported by a Reed-Hodgson Stanford Graduate Fellowship, an NDSEG Graduate Fellowship from the United States Department of Defense, and an NSF Graduate Fellowship from the National Science Foundation. Thanks also to Justin Talbot, Leonidas Guibas, and Jeremy Sugerman for fruitful discussion and advice, to Hao Li for providing us with mesh data, and also to our human and canine figure subjects.

References

ADAMS, A., GELFAND, N., AND PULLI, K. 2008. Viewfinder alignment. *Computer Graphics Forum (Proc. Eurographics) 27*, 2, 597–606.

AMAT, F., MOUSSAVI, F., COMOLLI, L., ELIDAN, G., DOWNING, K., AND HOROWITZ, M. 2008. Markov random field based automatic image alignment for electron tomography. *Journal of Structural Biology* (March), 260–275.

ARYA, S., MOUNT, D., NETANYAHU, N. S., SILVERMAN, R., AND WU, A. 1998. An optimal algorithm for approximate nearest neighbor searching. *Journal of the ACM 45*, 6.

AURICH, V., AND WEULE, J. 1995. Non-linear gaussian filters performing edge preserving diffusion. In *Mustererkennung 1995, 17. DAGM-Symposium*, Springer-Verlag, London, UK, 538–545.

AVANAKI, A. N. 2006. A spatiotemporal edge-preserving denoising method for high-quality video. *Signal Processing and Information Technology, 2006 IEEE International Symposium on* (Aug.), 157–161.

BEKAERT, P., SBERT, M., AND WILLEMS, Y. D. 2000. Weighted importance sampling techniques for monte carlo radiosity. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, Springer-Verlag, London, UK, 35–46.

BENNETT, E. P., AND MCMILLAN, L. 2005. Video enhancement using per-pixel virtual exposures. *ACM Transactions on Graphics (Proc. SIGGRAPH 05)*.

BROX, T., KLEINSCHMIDT, O., AND CREMERS, D. 2008. Efficient nonlocal means for denoising of textural patterns. *Image Processing, IEEE Transactions on 17*, 7 (July), 1083–1092.

BUADES, A., COLL, B., AND MOREL, J.-M. 2005. A non-local algorithm for image denoising. In *Proc. CVPR 05.*, vol. 2, 60–65 vol. 2.

BUADES, A., COLL, B., AND MOREL, J.-M. 2008. Nonlocal image and movie denoising. *International Journal of Computer Vision 76*, 2, 123–139.

BUCK, I. 2007. Gpu computing: Programming a massively parallel processor. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, IEEE Computer Society, Washington, DC, USA, 17.

CHEN, J., PARIS, S., AND DURAND, F. 2007. Real-time edge-aware image processing with the bilateral grid. *ACM Transactions on Graphics (Proc. SIGGRAPH 07)*.

D.BARASH. 2002. A fundamental relationship between bilateral filtering, adaptive smoothing and the nonlinear diffusion equation. *IEEE Transactions on Pattern Analysis and Machine Intelligence 24*, 6, 844–847.

DESBRUN, M., MEYER, M., SCHRODER, P., AND BARR, A. 1999. Implicit fairing of irregular meshes using diffusion and curvature flow. *Proc. SIGGRAPH 99*.

DURAND, F., AND DORSEY, J. 2002. Fast bilateral filtering for the display of high-dynamic-range images. *Proc. SIGGRAPH 02*.

EISEMANN, E., AND DURAND, F. 2004. Flash photography enhancement via intrinsic relighting. *ACM Transactions on Graphics (Proc. SIGGRAPH 04)*.

FLEISCHMAN, S., COHEN-OR, D., AND SILVA, C. 2005. Robust moving least-squares fitting with sharp features. *ACM Transactions on Graphics (Proc. SIGGRAPH 05)*.

FLEISHMAN, S., DRORI, I., AND COHEN-OR, D. 2003. Bilateral mesh denoising. *ACM Transactions on Graphics (Proc. SIGGRAPH 03)*.

JOHNSON, A., AND HEBERT, M. 1999. Using spin images for efficient object recognition in cluttered 3D scenes. *PAMI 21*.

JONES, T., DURAND, F., AND DESBRUN, M. 2003. Non-itertive, feature-preserving mesh smoothing. *ACM Transactions on Graphics (Proc. SIGGRAPH 03) 24*, 3.

KUMAR, N., ZHANG, L., AND NAYAR, S. K. 2008. What is a good nearest neighbors algorithm for finding similar patches in images? *Proc. ECCV 08*, 364–378.

PARIS, S., AND DURAND, F. 2006. A fast approximation of the bilateral filter using a signal processing approach. In *Proc. ECCV 06*.

PARIS, S., AND DURAND, F. 2009. A fast approximation of the bilateral filter using a signal processing approach. *International Journal of Computer Vision 81*, 24–52.

PARK, S. W., LINSEN, L., KREYLOS, O., OWENS, J. D., AND HAMANN, B. 2006. Discrete sibson interpolation. *IEEE Transactions on Visualization and Computer Graphics 12*, 2 (Mar./Apr.), 243–253.

PETSCHNIGG, G., SZELISKI, R., AGRAWALA, M., COHEN, M., HOPPE, H., AND TOYAMA, K. 2004. Digital photography with flash and no-flash image pairs. *ACM Transactions on Graphics (Proc. SIGGRAPH 04)*.

SHEFFER, A., PRAUN, E., AND ROSE, K. 2006. *Mesh Parameterization Methods and Their Applications*. Now Publishers.

SMITH, S., AND BRADY, J. M. 1997. Susan:a new approach to low level image processing. *International Journal of Computer Vision 23*, 45–78.

SORKINE, O., COHEN-OR, D., LIPMAN, Y., RSSL, C., PETER SEIDEL, H., AND ALEXA, M. 2004. Laplacian surface editing. *Proc. SGP*.

TAUBIN, G. 1995. A signal processing approach to fair surface design. *Proc. SIGGRAPH 95*.

TELLEEN, J., SULLIVAN, A., YEE, J., WANG, O., GUNAWARDANE, P., COLLINS, I., AND DAVIS, J. 2007. Synthetic shutter speed imaging. *Computer Graphics Forum (Proc. Eurographics) 26*, 3, 591–598.

TOMASI, C., AND MANDUCHI, R. 1998. Bilateral filtering for gray and color images. *Proc. ICCV 98 0*, 839.

WEISS, B. 2006. Fast median and bilateral filtering. *ACM Transactions on Graphics (Proc. SIGGRAPH 06)*.

YANG, C., DURAISWAMI, R., GUMEROV, N. A., AND DAVIS, L. 2003. Improved fast gauss transform and efficient kernel density estimation. *Proc. ICCV 03*, 664–671 vol.1.

YOSHIZAWA, S., BELYAEV, A., AND SEIDEL, H.-P. 2006. Smoothing by example: mesh denoising by averaging with similarity-based weights. *IEEE Shape Modeling International*.

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia 08)*.