Efficient Data Structures for Piecewise-smooth Video Processing

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUN 17 2011
LIBRARIES
ARCHIVES

by

Jiawen Chen

S.B., Massachusetts Institute of Technology (2004)
M.Eng., Massachusetts Institute of Technology (2005)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2011

© Massachusetts Institute of Technology 2011. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
May 19, 2011

Certified by ....
Frédo Durand
Associate Professor
Thesis Supervisor

Accepted by............. ....
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses

Efficient Data Structures for Piecewise-smooth Video Processing

by

Jiawen Chen

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2011, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

A number of useful image and video processing techniques, ranging from low level operations such as denoising and detail enhancement to higher level methods such as object manipulation and special effects, rely on piecewise-smooth functions computed from the input data. In this thesis, we present two computationally efficient data structures for representing piecewise-smooth visual information and demonstrate how they can dramatically simplify and accelerate a variety of video processing algorithms.

We start by introducing the *bilateral grid*, an image representation that explicitly accounts for intensity edges. By interpreting brightness values as Euclidean coordinates, the bilateral grid enables simple expressions for edge-aware filters. Smooth functions defined on the bilateral grid are piecewise-smooth in image space. Within this framework, we derive efficient reinterpretations of a number of edge-aware filters commonly used in computational photography as operations on the bilateral grid, including the bilateral filter, edge-aware scattered data interpolation, and local histogram equalization. We also show how these techniques can be easily parallelized onto modern graphics hardware for real-time processing of high definition video.

The second data structure we introduce is the *video mesh*, designed as a flexible central data structure for general-purpose video editing. It represents objects in a video sequence as 2.5D "paper cutouts" and allows interactive editing of moving objects and modeling of depth, which enables 3D effects and post-exposure camera control. In our representation, we assume that motion and depth are piecewise-smooth, and encode them sparsely as a set of points tracked over time. The video mesh is a triangulation over this point set and per-pixel information is obtained by interpolation. To handle occlusions and detailed object boundaries, we rely on the user to rotoscope the scene at a sparse set of frames using spline curves. We introduce an algorithm to robustly and automatically cut the mesh into local layers with proper occlusion topology, and propagate the splines to the remaining frames. Object boundaries are refined with per-pixel alpha mattes.

At its core, the video mesh is a collection of texture-mapped triangles, which we can edit and render interactively using graphics hardware. We demonstrate the effectiveness of our representation with special effects such as 3D viewpoint changes, object insertion, depth-of-field manipulation, and 2D to 3D video conversion.

Thesis Supervisor: Frédo Durand
Title: Associate Professor

Acknowledgments

It is customary in a thesis to acknowledge everyone who has helped the author through his graduate career. But given the number of friends and family from whom I have benefited over the years, I am sure to have left some out. Please accept my apologies up front.

First and foremost, I thank the members of the MIT Computer Graphics Group, who over the years have become a second family to me. I acknowledge the following individuals in particular. My advisor Frédo, who taught me more about computer graphics and how to be a scientist more than I ever imagined. Sylvain Paris, who is the very model of a postdoc and confidant, and whose inspiration and optimism is boundless. Robert Wang, who stood by me more times than I can count. Eugene Hsu, who taught me to consider ideas that other people don't want you to express. Jonathan Ragan-Kelley, who rekindled my interest in computer systems. Ilya Baran, without whom our work on volumetric shadows (not to mention a number of tremendously nerdy jokes) would not have been possible. And of course Bryt Bradley, who has had to put up with my antics for close to 10 years now.

I extend my thank to all my colleagues and collaborators, both at MIT and elsewhere, and in particular the students of the Computer Vision Group, for all the wonderfully fruitful research discussions.

Finally, I of course thank my family and friends for their love and support.

To everyone whose name deserves to be here: live long and prosper.

Contents

Chapter 1

Introduction

Digital representations of photos and video grants artists tremendous flexibility in the way they can edit and manipulate imagery compared to traditional analog photography. However, compared to other types of digital content such as text, images and video are more difficult to edit due to the sheer amount of unstructured data involved. A single picture has tens of millions of pixels, and video contains orders of magnitude more. While it is easy to crop an image or to adjust the contrast in a video, it is much more difficult to refocus the camera or remove an unsightly lamp post.

This thesis addresses some of the problems in real-time image filtering and interactive video editing by developing new data structures to represent *piecewise-smooth* information in images and video. We show how a number of useful image and video processing algorithms are piecewise-smooth in nature, and develop efficient data representations to simplify these operators. Our data structures focus on two specific aspects of the post-production pipeline: speed and flexibility. A wide variety of software and techniques in image processing are available to dramatically improve the quality of photos and videos. However, many of these techniques operate on raw pixels and are computationally expensive [63, 91]. It is well known that interactive feedback is critical to achieving the desired result. In addition to interactive feedback, we argue that in any editing environment, the data representation should be well adapted for the wide variety of potential edits an artist might make. For example, artists working on converting 2D movies to stereoscopic 3D benefit immensely from integrated environments that manipulate a single data representation with dedicated tools [44]. In this thesis, we present two data structures. The first, the bilateral grid, is designed to simplify and accelerate *edge-aware* image processing filters, a class of filters that are smooth but respects strong edges in an image. The second, the video mesh, is designed to handle piecewise-smooth information in video, including motion, depth, and occlusion. It is intended as a central data structure for video editing and enables a number of post-exposure manipulations of both the camera and scene objects which would be otherwise difficult.

1.1 Piecewise-smooth video processing

To introduce the concept of piecewise-smooth video processing, let us consider the problem of *image colorization.* Given an old black and white photograph, we would like to estimate a color version of the same scene. While inferring the color image completely automatically can be quite difficult (if not impossible), one approach proposed by Levin et al. [48] starts by asking the user to paint a sparse set of colored scribbles over the input. These scribbles specify a set of constraints: these pixels should be this color. Their approach then smoothly interpolates these constraints onto the rest of the pixels, but taking care not to cross edges in the original image: we should "color within the lines". In Figure 1-1, the girl's face and her shirt should be assigned the color of their respective scribbles, but the colors should not cross the boundary between them. Like colorization, a number of popular image processing filters are piecewise-smooth and in particular *edge-aware* [9,13,25,27,31,46,54,59,64,90]: we want output that is smooth, but not across strong edges in the input. The bilateral grid is designed to facilitate this class of filters.

Piecewise-smooth operators are not limited to functions that respect discontinuities in image intensity. Video sequences feature a number of piecewise-smooth properties such as motion, depth, and occlusions. Consider, for example, the problem of interpolating motion in a video. Given an arbitrary video sequence, we may have reliable motion information at

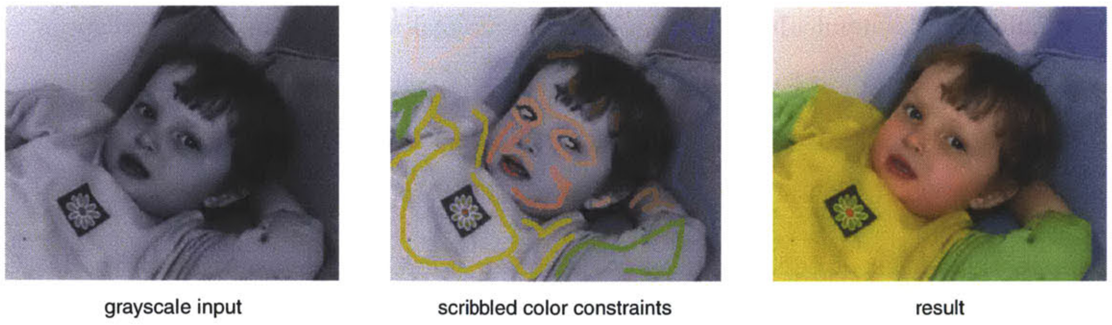


Figure 1-1: *Colorization example by Levin et al. [48].*

only a small set of locations on the image plane [80]. To propagate motion information to the remainder, we can try to interpolate the scattered data using a smooth operator, which assumes that most objects in the physical world do not deform arbitrarily—the motion field is smooth. If we perform such an interpolation, it will probably work reasonably well for most of the pixels, but will fail near object boundaries. To illustrate, what if if an actor is waving his arm, but is standing over a static background? The points on his arm move with it, but the points on the background are stationary. Any completely smooth interpolation will result in unrealistic motions near the boundary; but if we can accurately locate these occlusion boundaries, we can use a piecewise-smooth operator like the colorization technique above to generate a dense and accurate motion field. A similar argument holds for interpolating depth information. The second part of this thesis exploits the piecewise-smooth properties of video data to build an efficient, sparse representation for video editing.

1.2 The Bilateral Grid

Edge-aware filters have enabled novel image enhancement and manipulation techniques where image structure such as strong edges are taken into account. They are based on the observation that many meaningful image components and desirable image manipulations tend to be piecewise-smooth rather than purely band-limited. For example, illumination is usually smooth except at shadow boundaries [59], and tone mapping suffers from haloing artifacts when a low-pass filter is used to drive local adjustment [20], a problem which can be solved with edge-aware filters [25, 91].

In particular, the bilateral filter is a simple technique that smoothes an image except at strong edges [8,83,90]. It has been used in a variety of contexts to decompose an image into a piecewise-smooth large-scale base layer and a detail layer for tone mapping [25], flash/no-flash image fusion [27,64], and a wealth of other applications [9,13,98,99].

A common drawback of edge-aware filters is speed: a direct implementation of the bilateral filter can take several minutes for a one megapixel image. However, recent work has demonstrated acceleration and obtained good performance on CPUs, on the order of one second per megapixel [25,61,65,96]. However, these approaches still do not achieve real-time performance on high-definition content.

In this work, we dramatically accelerate and generalize the bilateral filter, enabling a variety of edge-aware image processing applications in real-time on high-resolution inputs. Building upon the technique by Paris and Durand [61], who use linear filtering in a higher-dimensional space to achieve fast bilateral filtering, we extend their high-dimensional approach and introduce the *bilateral grid*, a new compact data structure that enables a number of edge-aware manipulations. We parallelize these operations using modern graphics hardware to achieve real-time performance at HD resolutions. In particular, our GPU bilateral filter is two orders of magnitude faster than the equivalent CPU implementation. The bilateral grid is described in detail in Chapter 3.

1.3 The Video Mesh

The second data structure that we introduce is the *video mesh*, a new representation that encodes the motion, layering, and 3D structure of a video sequence in a unified data structure. The video mesh can be viewed as a 2.5D "paper cutout" model of the world. For each frame of a video sequence, the video mesh is composed of a triangle mesh together with texture and alpha (transparency). Depth information is encoded with a per-vertex z coordinate, while motion is handled by linking vertices in time (for example, based on feature tracking). The mesh can be cut along occlusion boundaries and alpha mattes enable the fine treatment of partial occlusion. It supports a more general model of visibility than traditional layer-based methods [93] and can handle self-occlusions within an object such as the actor arm's in front of his body in our companion video. The per-vertex storage of depth and the rich occlusion representation make it possible to extend image-based modeling into the time dimension. Finally, the video mesh is based on texture-mapped triangles to enable fast processing on graphics hardware. This data structure is covered in detail in Chapter 4.

Chapter 2

Background

This chapter provides some background on basic image and video processing, including a broad overview leading to the current state of the art. More detailed direct comparisons with our techniques are embedded within the relevant chapters.

We begin by describing the basic representation of images and videos on a digital computer, followed by the theory of classical linear image filtering. With some examples, we illustrate how a number of desirable properties are often nonlinear and *piecewise-smooth* in nature. We conclude the section on image processing with a discussion on a few high level image editing applications and their relation to edge-aware image filtering.

The remainder of this chapter extends image processing into the temporal domain to cover video. It covers the natural extension of image editing to image sequences, cutting and resequencing video, and the role of motion. We place particular emphasis on the difficulties in selecting and placing objects.

2.1 Image and video representation

A two-dimensional grayscale *image* is a function $\mathbb{R}^2 \to \mathbb{R}^+$, mapping each point in a subset of a plane to a positive brightness value. The natural extensions to this definition include video, which adds a time component to the domain ($\mathbb{R}^3 \rightarrow \mathbb{R}^+$), and color, where the range is a function over the wavelength.

Sampling On a digital computer, image and video data must be *sampled* and *quantized.* The domain is typically a regularly spaced two-dimensional grid of a given width and height. The range usually contains 8, 16, or 32-bit fixed or floating-point values and are often *gamma encoded* (see Section 2.2.2). Digital video also typically uses a fixed temporal sampling rate (e.g., 24 or 30 Hz). Color images usually assume the trichromatic color vision model [74] and store the response of a sensor sensitive to three particular wavelength ranges (commonly "red, green, blue", but other color spaces are also used). Ultimately, our basic image and video representation is a two- or three-dimensional array of quantized values.

2.2 Image editing

A digital representation enables a number range of creative edits on images. Some examples of simple image manipulations are *cropping*, which discards all but a rectangular subset of an image as a form of recomposing a shot after it has been taken, *cut-and-paste*, where sets of pixels (i.e., objects) are repositioned, and *compositing*, where objects from different images are layered over each other according to a expressive mathematical model to simulate effects such as transparency [69].

A large class of editing operations, with a wide range of applications including image resizing (a.k.a. resampling), denoising, and detail enhancement, are *linear shift-invariant filters*, which we cover below.

2.2.1 Linear shift-invariant filters

The family of linear shift-invariant (LSI) filters has received considerable attention in past decades due to their broad utility, elegant mathematical representation, and computational efficiency. This section provides a basic review of the theory of LSI filters, and we refer the reader to standard texts for a more comprehensive treatment [53, 60]. In particular, we omit the important discussion of the Fourier theory of linear systems.

We define a *filter* as any operation that takes an image as input, and produces another image as output. As expected from their name, linear shift-invariant filters have two properties that are particularly useful.

Linearity A filter F is *linear* if for any image x:

$$F\{ax\} = aF\{x\} \tag{2.1}$$

for some scalar a, and for any two images x and y:

$$F\{x+y\} = F\{x\} + F\{y\} \tag{2.2}$$

In other words, linearity says that if we scale an input image by a scalar, the output under a linear filter is scaled by the same factor. And if we add two images, the output of the sum under a linear filter is the sum of the individual output images.

Shift-invariance A filter F is *shift-invariant* if for any sampled image $x[i,j]$ where

$$F\{x[i,j]\} = y[i,j]$$

then,

$$F\{x[i+a, j+b]\} = y[i+a, j+b] \tag{2.3}$$

$$\text{for} \quad a, b \in \mathbb{Z} \tag{2.4}$$

The shift-invariance property simply says that if we translate an image by a fixed amount, its output is also shifted by the same amount. These two properties are particularly powerful when combined.

Impulse response Let $\delta[i,j]$, which we call the *unit sample* be defined as:

$$\delta[i,j] = \begin{cases} 1 & \text{if } i = 0 \text{ and } j = 0, \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

The unit sample is simply 1 at the origin, and 0 everywhere else. We can take any sampled image $x[i,j]$ (with width and height w and h, respectively) and redefine it as a sum of scaled and shifted unit samples:

$$x[i,j] = \sum_{b=0}^{h-1} \sum_{a=0}^{w-1} x[a,b]\, \delta[i-a, j-b] \tag{2.6}$$

Because the unit sample is itself an image, we can apply a linear shift-invariant filter to it. Let:

$$h[i,j] = F\{\delta[i,j]\} \tag{2.7}$$

which we call the *impulse response* or *kernel* of the filter F. Since F is LSI, the output of F on an arbitrary image x is simply the sum of scaled and shifted copies of h:

$$\begin{aligned}
F\{x[i,j]\} &= F\{\sum_{b=0}^{h-1}\sum_{a=0}^{w-1} x[a,b]\, \delta[i-a, j-b]\} \\
&= \sum_{b=0}^{h-1}\sum_{a=0}^{w-1} F\{x[a,b]\, \delta[i-a, j-b]\} \\
&= \sum_{b=0}^{h-1}\sum_{a=0}^{w-1} x[a,b]\, F\{\delta[i-a, j-b]\} \\
&= \sum_{b=0}^{h-1}\sum_{a=0}^{w-1} x[a,b]\, F\{\delta\}[i-a, j-b] \\
&= \sum_{b=0}^{h-1}\sum_{a=0}^{w-1} x[a,b]\, h[i-a, j-b] \qquad (2.8)
\end{aligned}$$

Hence, the impulse response completely characterizes an LSI filter F.

Convolution Equation 2.8 is known as the *convolution* of the images x and h, which we denote $x \otimes h$. Although the summation in Equation 2.8 spans the entire domain of x; in practice, it is much smaller: the product within the summation is non-zero only when both x and h are non-zero. Therefore, convolution has complexity $O(whmn)$, for a $w \times h$ image and a $m \times n$ kernel[1]. The vast majority of image processing filters have a kernel size significantly smaller than the size of the image. Since LSI filters are completely characterized by their kernel, their performance is predictable and fairly efficient in practice. Furthermore, because each output sample depends on only a small $m \times n$ neighborhood of input samples, LSI filters are trivial to implement efficiently on modern parallel processors which feature an array of arithmetic units that share a local cache.

Gaussian blur The canonical example of an LSI filter is the *Gaussian Blur*, which we discuss in the context of denoising. Consider an input image x, which, due to imperfections in sensor electronics, is corrupted by some noise n. What is recorded is the image $\hat{x}$, from

[1]The complexity is $O(whmn)$ despite having only a double summation because Equation 2.8 only evaluates the output for a particular pixel at $[i, j]$. Using a Fast Fourier Transform, it is possible to do asymptotically better, but is practical only for relatively large kernels.

which we would like to recover x.

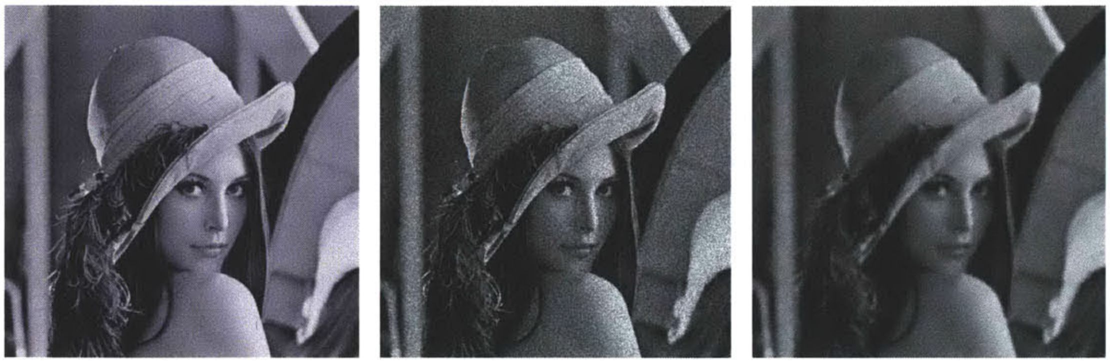


Figure 2-1: *Denoising example using Gaussian blur.*

Assuming that the noise is statistically independent over the image pixels, with zero mean, we can attempt to recover x by using the Law of Large Numbers. If the pixels are small with respect to the scene objects we want to resolve, then we can remove the noise by taking the average of neighboring pixels: the noise should "cancel itself out." The choice of how to weight the neighbors determines the filter. A popular choice is to assume that the center pixel within each region is more important and that the pattern follows a Gaussian distribution:

$$G_\sigma[dx, dy] = K\ exp(-(dx^2 + dy^2)/(2\sigma^2)) \tag{2.9}$$

where K is a normalization constant ensuring that the function integrates to unity and the parameter σ depends on the noise amplitude. The larger the noise amplitude, the larger σ should be to gather samples from a larger neighborhood. In other words, the denoised image is a convolution of the input with a Gaussian kernel:

$$x = \hat{x} \otimes G_\sigma \tag{2.10}$$

Figure 2-1 demonstrates the Gaussian blur on denoising a natural image. Note that although the noise has been removed, detail and contrast in the image are lost. This is due to the relatively low spatial resolution of the image relative to the details in the scene.

2.2.2 Nonlinear filters

As discussed above, linear shift-invariant filters are completely characterized by their impulse response. Filters that fall outside this regime are considered "nonlinear" or *shift-variant.* As we will demonstrate below, nonlinear filters have performance characteristics that are far less predictable than linear ones. Nevertheless, there are a number of popular filters which are nonlinear.

Gamma encoding Perhaps the simplest example of a nonlinear operator is *gamma encoding* or *gamma correction*, which is the map:

$$x[i,j] \rightarrow x^{\gamma}[i,j] \tag{2.11}$$

for some fixed scalar γ. Gamma encoding serves two purposes. First, the human visual system is more sensitive to ratios rather than absolute values. Given a fixed bit budget per pixel, a linear mapping would distribute information uniformly over absolute brightness values. The human visual system would interpret this distribution as having too little information in the dark regions, and too much in the bright regions. While a logarithmic distribution would be ideal, gamma encoding is an effective way of avoiding the singularity near zero. A typical value of γ for this purpose is 2.0 [70]. Gamma encoding also serves as a way to compensate for the nonlinear response curves of cathode ray tube (CRT) displays [70]. CRT displays interpret the values of an input image as voltage controlling an electron gun. However, the brightness-voltage relationship is nonlinear: the brightness E can be modeled as having a power law relation with respect to the input voltage V: $E = V^{\gamma}$, with a typical γ of 2.5. To ensure that an image is faithfully reproduced on CRT displays, many images are gamma corrected by first applying the inverse of the CRT response, with $\gamma = 1/2.5$. As a compromise between the two objectives, a common value is $\gamma = 2.2$. Figure 2-2 shows an example.

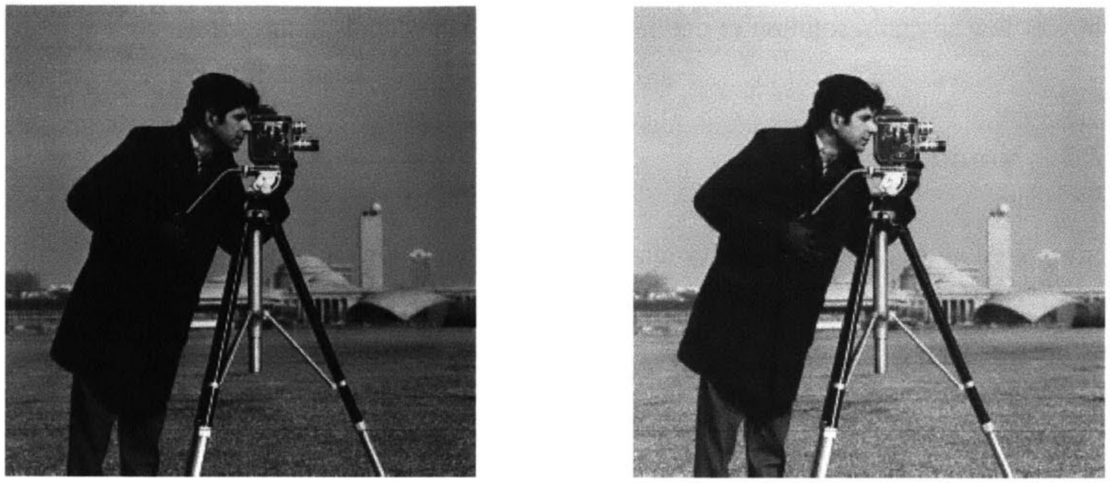


Figure 2-2: *Gamma correction example.*

It is clear from equation 2.11 that each output sample depends on exactly one input sample, and that the relationship is not linear: scaling the input by a constant factor a results in an output that is scaled by a^γ. Nevertheless, gamma correction is fairly efficient to implement on numerous hardware architectures.

General curve remapping Gamma correction is a function on the intensity values of an image. We need not limit ourselves to simple functions. Indeed, most image editing [5] tools offer interactive "brightness and contrast" adjustment options which use affine functions (adding a constant offset) or general sums of polynomials (a.k.a. *spline curves*). Curve-based remapping is used extensively [6] by photographers to enhance the overall contrast and accentuate detail in certain regions of an image (e.g., highlights and shadows).

Histogram equalization One interesting example of a nonlinear remapping curve is *histogram equalization*; where, given an input image, produces an output image that has a histogram that is uniformly distributed in intensity[2] [35]. It is a useful and automatic way to increase the overall contrast of an image. Histogram equalization is in essence a curve-based intensity mapping where the curve is computed from the image itself (the cumulative

[2]For sampled and quantized images, the distribution is only approximately uniform. It is exactly uniform only if the input image has a histogram that is continuous and positive for all values in the desired range.

distribution function (CDF) of its intensity distribution). It "stretches out" the intensity values that occur most frequently in the image. Figure 2-3 shows an example.

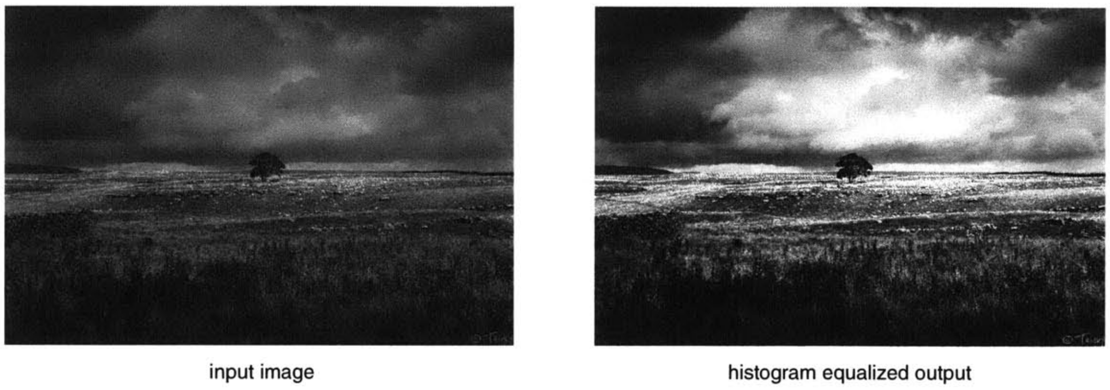


Figure 2-3: *Histogram equalization example. Tree image is © Rick Harrison / fortybelowzero.com, used with permission.*

Histogram equalization begins with first computing a discrete histogram: a count of how many pixels are at each quantized intensity level. We then turn this into a probability mass function (PMF) by normalizing it by the number of pixels. Finally, we compute a prefix sum over the PMF to produce a cumulative distribution function (CDF), which gives for each intensity level i, the fraction of pixels in the image with intensity less than i. The CDF ensures a mapping from the unit interval $[0, 1]$ to itself. Finally, to produce a histogram equalized image, each pixel looks up its value in the CDF determine the output intensity. The pseudocode is as follows:

```
Require: Input image $x$ is $w \times h$ with range $[0, 255]$.
Ensure: Output image $y$ has approximately uniform intensity distribution.
  $h \leftarrow hist(x)$ {Compute image histogram}
  $p \leftarrow h/(w \times h)$ {Normalize histogram into PMF}
  $cdf \leftarrow$ prefix-sum($p$) {Compute prefix sum of PMF to produce CDF}
  for $j = 0$ to $h - 1$ do
    for $i = 0$ to $w - 1$ do
      $y[i, j] \leftarrow 255 \times cdf[x[i, j]]$
    end for
  end for
```

Performance implications Unlike linear filters, whose performance depends purely on kernel size, nonlinear filters have more diverse performance characteristics. For example, gamma correction is a simple per-pixel floating-point operation. In contrast, histogram equalization requires the computation of a histogram, a prefix sum through the histogram, and finally a data-dependent lookup, which can be difficult to compute on data-parallel hardware such as a digital signal processor (DSP) or graphics processor (GPU) (although efficient implementations now exist for limited data sizes [67, 78]).

2.2.3 Edge-aware image filtering

This subsection studies *edge-aware* image filters, a family of piecewise-smooth filters that has received some attention in recent years due to their broad utility in photography. Edge-aware filters define some notion of an "edge" in an image, and take particular care in their vicinity to not corrupt them due to their importance within the human visual system [79]. We concentrate on a particular filter, called *anisotropic diffusion*, which smoothes an image but preserves its edges.

Anisotropic diffusion Anisotropic diffusion, introduced by Perona and Malik [63], models image smoothing as a heat diffusion process governed by a differential equation:

$$\frac{\partial I}{\partial t} = \text{div}(c(x, y, t)\nabla I) \tag{2.12}$$

where the image $I(x, y)$ is smoothed over time t, governed by a heat diffusion function c, which in this case, is allowed to vary both in space and in time. Anisotropic diffusion can be seen as a generalization of standard (isotropic) heat diffusion with constant conductivity c:

$$\frac{\partial I}{\partial t} = \text{div}(c\nabla I) = c\Delta I \tag{2.13}$$

which, at time t, is equivalent to a Gaussian blur of variance t^2 [63]. Anisotropic diffusion is simply a modification of the differential equation so that near edges, the conductivity is reduced proportional to the strength of the edge. Perona and Malik propose two possibilities:

$$c(||\nabla I||) = e^{-(||\nabla I||/K)^2} \tag{2.14}$$

or

$$c(||\nabla I||) = \frac{1}{1 + (||\nabla I||/K)^2} \tag{2.15}$$

While anisotropic diffusion produces excellent edge-preserving smoothing results, it is unfortunately a slow, iterative process that does not have an analytical solution (unlike isotropic heat diffusion, whose solution is a Gaussian). The bilateral filter [90] was introduced as a non-iterative alternative to anisotropic diffusion for edge-preserving smoothing. We defer more detailed discussion of the bilateral filter to Chapter 3.

Tone mapping One of the major applications of anisotropic diffusion is *tone mapping*, where the potentially large variation in image contrast is adapted to fit a display with lower contrast. Consider a *high dynamic range* (HDR) image, which accurately represents the brightness of a real-world scene. It can contain very small values (in the shadow regions) or very large ones (in areas directly lit by the sun). Because of this large difference in dynamic range, a simple affine scaling:

$$Y[i,j] = aX[i,j] + b \tag{2.16}$$

with constants a and b mapping the input image to the range of the output display, will destroy all the *details* in the image, which are typically small-scale variations. To reduce overall contrast while preserving details, a popular method is to use *local adaptation*: decompose the input image into the sum of a *base layer* containing the large-scale brightness, and a *detail layer* containing small-scale variations. The method is called local adaptation because, rather than using a single scaling parameter for the entire image, it uses a spatially varying base layer to modulate the scaling.

One possibility is to use Gaussian blurring to compute the base layer B and subtract it from the input to form the detail layer D:

$$B = G_\sigma \otimes X \tag{2.17}$$

$$D = X - B \tag{2.18}$$

If we scale just the base layer B and recombine the result with D, we get an image that is almost right: the only problem is visible *halos* near image edges (see Figure 2-4). Halos are the result of the contrast scaling in one region (such as the trees in) affecting the amount of scaling applied across an edge (such as the much brighter sky). To prevent these halos, it suffices to use an edge-preserving smoothing filter such as anisotropic diffusion in place of the Gaussian blur, resulting in an improved image without halos.

2.2.4 Summary

This section discussed the theory of digital image filtering and the role of linear operators, and in particular, their performance characteristics. We discussed several nonlinear filters, leading up to one example of a piecewise-smooth edge-aware operator: anisotropic diffusion, which demonstrated to be useful in applications such as tone mapping. The goal of the first data structure in this thesis, the bilateral grid, is to rewrite a number of nonlinear edge-aware operators as linear operators with better performance characteristics.

2.3 Video editing

Video editing is, in general, more difficult than the processing of single images (video frames) simply because of the extra time dimension. Not only is there more data, the time dimension has a different nature than the two spatial dimensions. First, there are usually many more video frames than there are pixels in each frame. And second, the data exhibits *temporal coherence*: most of the pixels in one frame are heavily correlated with those in the next frame; objects in the world usually move along smooth trajectories. Hence, most video processing techniques are natural extensions of image processing filters.

To illustrate, we can easily apply a Gaussian blur to every frame of a video sequence. However, consider the case of cropping: it is trivial for a user to crop an image as a one-time edit. But suppose we want to crop a video, perhaps to adapt it to a display with a different aspect ratio. If we apply the same crop to every frame in the sequence, the results will likely be poor. Important objects near the edges of the screen may be cut out in one scene, and uninteresting background left behind in another. Video cropping is typically done manually by an editor (known as pan and scan) and automatic video retargeting remains an area of active research [75].

Let us consider another common image edit: cut and paste. With a single image, cutting out an object involves carefully selecting all of its pixels and taking care at boundaries (matting [49, 95]), and carefully placing it at target location. If the goal is to move the object within the same image, we must also *inpaint* the missing pixels revealed by the displaced object [14]. Cutting and pasting objects in video is considerably more difficult. To cut, one must select all pixels over a potentially large number of frames, taking into account any motion or changes in appearance. Pasting is now also a problem: the motion of the object may not match the target scene in every frame and raw sets of pixels lack any kind of animation control. Finally, video inpainting, the hallucination of large, temporally coherent regions, is an open problem in computer graphics.

Reliable motion information is key in solving many of these problems. For instance, suppose the video features a taxi driving by and the task is to attach an image of a logo to its side. In this case, we just need to track a single point, resize the logo, and composite over the input video until the taxi leaves the scene. If there are significant perspective effects due to changes in depth, tracking several points, or inferring depth from motion, will suffice.

As another illustrative example, suppose we want to denoise a video with an edge-aware filter which preserves large-scale structure in each frame. One issue with these filters is that they may not be temporally stable: although the output at each frame may be correct, and objects in the input sequence move smoothly, the output may not be smooth due to misalignments in features such as edges, resulting in a "jumpy" video. One possible solution would be to infer, for each pixel, its velocity (known as the *optical flow* [41]), and modify the filter to incorporate temporal neighbors. The presence of these neighbors can improve temporal stability considerably [13].

The goal of our second data structure, the video mesh, is to represent a video in such a way that useful information such as motion, depth, and texture, is efficiently encoded and editable. We show in Chapter 4 how, with a video mesh, editing tasks such as the ones discussed above become simple manipulations of the data structure.

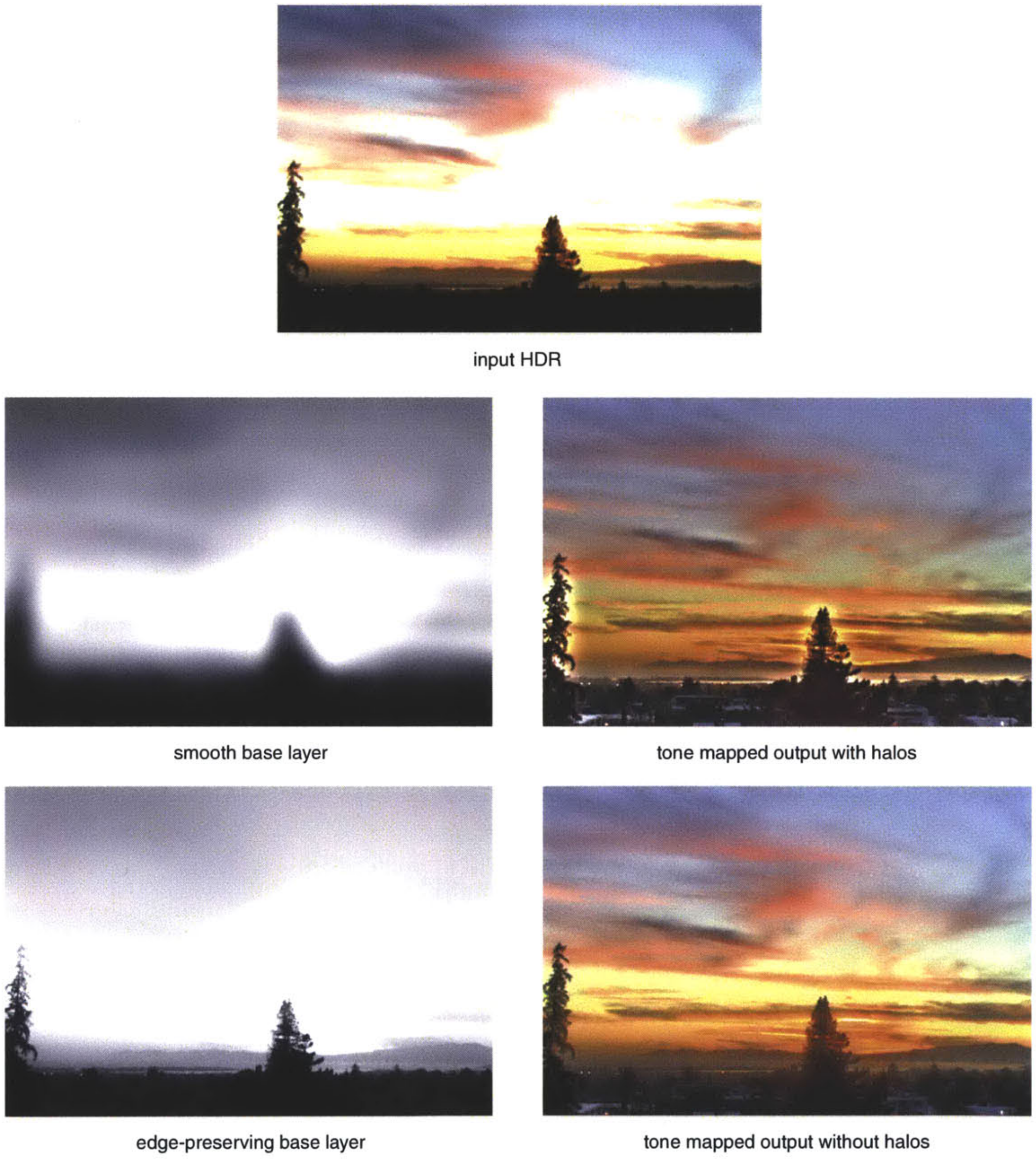

input HDR
smooth base layer
tone mapped output with halos
edge-preserving base layer
tone mapped output without halos

Figure 2-4: *Tone mapping example.* ***Top:*** *One exposure of a high dynamic range image.* ***Middle row:*** *Using a Gaussian blur to compute the base layer results in halos in the tone mapped output.* ***Bottom row:*** *Using an edge-preserving blur eliminates halo artifacts.*

Chapter 3

The Bilateral Grid

3.1 Introduction

The preceding chapter introduced the concept of edge-aware image processing and illustrated the application of edge-preserving smoothing to the tone mapping problem. The example used anisotropic diffusion as its smoothing operator, which is slow due to its iterative nature. In this chapter, we discuss an alternative edge-preserving smoothing operator—the bilateral filter, and generalize its key idea of using an intensity difference as a distance metric into a data structure, which we call the *bilateral grid*. We also consider a number of other operators which define edges by intensity differences, and formulate them as simple algorithms defined on the bilateral grid. We show how with the bilateral grid, these edge-aware algorithms are parallel and run in real-time on graphics hardware. We demonstrate novel applications with a variety of video processing applications [1].

[1]A version of this chapter appeared as a paper at SIGGRAPH 2007 [18]. This version features additional discussion regarding followup work as well as thoughts on how the bilateral grid would be implemented using 2011-era hardware.

3.1.1 Related Work

Bilateral Filter The bilateral filter is a nonlinear process that smoothes images while preserving their edges [8,83,90]. For an image I, at position $\mathbf{p}$, it is defined by:

$$bf(I)_{\mathbf{p}} = \frac{1}{W_{\mathbf{p}}} \sum_{\mathbf{q} \in N(\mathbf{p})} G_{\sigma_s}(\|\mathbf{p} - \mathbf{q}\|)\, G_{\sigma_r}(|I_{\mathbf{p}} - I_{\mathbf{q}}|)\, I_{\mathbf{q}} \tag{3.1a}$$

$$W_{\mathbf{p}} = \sum_{\mathbf{q} \in N(\mathbf{p})} G_{\sigma_s}(\|\mathbf{p} - \mathbf{q}\|)\, G_{\sigma_r}(|I_{\mathbf{p}} - I_{\mathbf{q}}|) \tag{3.1b}$$

The output is a simple weighted average over a neighborhood where the weight is the product of a Gaussian on the spatial distance (G_{σ_s}) and a Gaussian on the pixel value difference (G_{σ_r}), also called the range weight. The range weight prevents pixels on one side of a strong edge from influencing pixels on the other side since they have different values. This also makes it easy to take into account edges over a multi-channel image (such as RGB) since the Gaussian on the pixel value difference can have an arbitrary dimension.

Fast numerical schemes for approximating the bilateral filter are able to process large images in less than a second. Pham et al. [65] describe a separable approximation that works well for the small kernels used in denoising but suffers from artifacts with the larger kernels used in other applications. Weiss [96] maintains local image histograms, which unfortunately limits this approach to box spatial kernels instead of the smooth Gaussian kernel.

Paris and Durand [61] extend the fast bilateral filter introduced by Durand and Dorsey [25] and recast the computation as a higher-dimensional linear convolution followed by trilinear interpolation and a division. We generalize the ideas behind their higher-dimensional space into a data structure, the bilateral grid, that enables a number of operations, including the bilateral filter, edge-aware painting and interpolation, and local histogram equalization.

Other edge-preserving techniques include anisotropic diffusion [63,91] which is related to the bilateral filter [11,25]. Optimization [48,54] can also be used to interpolate values while respecting strong edges. Our work introduces an alternative to these approaches.

High-Dimensional Image Representation The interpretation of 2D images as higher-dimensional structures has received much attention. Sochen et al. [84] describe images as 2D manifolds embedded in a higher-dimensional space. For instance, a color image is embedded in a 5D space: 2D for x and y plus 3D for color. This enables an interpretation of PDE-based image processes in terms of geometric properties such as curvature. Our bilateral grid shares the use of higher-dimensional spaces with this approach. It is nonetheless significantly different since we consider values over the entire space and not only on the manifold. Furthermore, we store homogeneous values, which allows us to handle weighted functions and the normalization of linear filters. Our approach is largely inspired by signal processing whereas Sochen et al. follow a differential geometry perspective.

Felsberg et al. [32] describe an edge-preserving filter based on a stack of channels that can be seen as a volumetric structure similar in spirit to the bilateral grid. Each channel stores spline coefficients representing the encoded image. Edge-preserving filtering is achieved by smoothing the spline coefficients within each channel and then reconstructing the splines from the filtered coefficients. In comparison, the bilateral grid does not encode the data into splines and allows direct access. This enables faster computation; in particular, it makes it easier to leverage the computational power of modern parallel architectures.

3.2 Bilateral Grid

The effectiveness of the bilateral filter in respecting strong edges comes from the inclusion of the range term $G_{\sigma_r}(|I_\mathbf{p} - I_\mathbf{q}|)$ in the weighted combination of Equation 3.1: although two pixels across an edge are close in the spatial dimension, from the filter's perspective, they are distant because their values differ widely in the range dimension. We turn this principle into a data structure, the bilateral grid, which is a 3D array that combines the two-dimensional spatial domain with a one-dimensional range dimension, typically the image intensity. In this three-dimensional space, the extra dimension makes the Euclidean distance meaningful for edge-aware image manipulation.

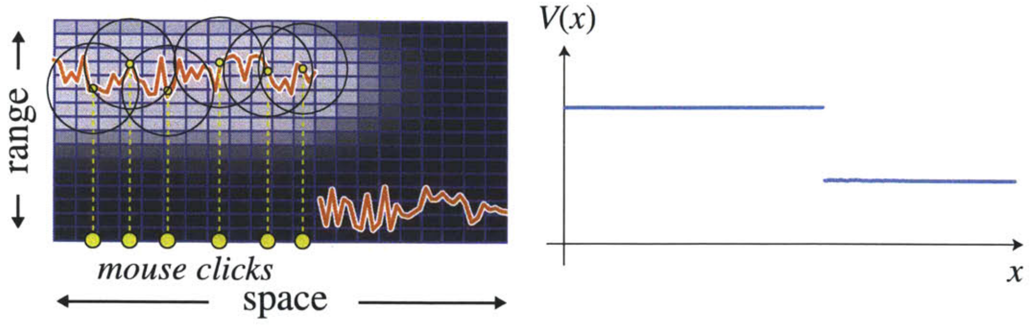
The bilateral grid first appeared as an auxiliary data structure in Paris and Durand's fast bilateral filter [61]. In that work, it was used as an algebraic re-expression of the original bilateral filter equation. Our perspective is different in that we view the bilateral grid as a primary data structure that enables a variety of edge-aware operations in real time.

3.2.1 A Simple Illustration

Before formally defining the bilateral grid, we first illustrate the concept with the simple example of an edge-aware brush. The edge-aware brush is similar to brushes offered by traditional editing packages except that when the user clicks near an edge, the brush does not paint across the edge.

When the user clicks on an image E at a location $(x_\mathrm{u}, y_\mathrm{u})$, the edge-aware brush paints directly in the three-dimensional bilateral grid at position $\big(x_\mathrm{u}, y_\mathrm{u}, E(x_\mathrm{u}, y_\mathrm{u})\big)$. The spatial coordinates are determined by the click location, while the range coordinate is the intensity of the clicked pixel. The edge-aware brush has a smooth falloff in all three dimensions. The falloff along the two spatial dimensions is the same as in classical 2D brushes, but the range falloff is specific to our brush and ensures that only a limited interval of intensity values is affected. To retrieve the painted value V in image space at location (x, y), we read the value of the bilateral grid at position $\big(x, y, E(x, y)\big)$. We use the same terminology as Paris and Durand [61] and call this operation *slicing*. In regions where the image E is nearly constant, the edge-aware brush behaves like a classical brush with a smooth spatial falloff. Since the range variations are small, the range falloff has little influence. At edges, E is discontinuous and if only one side has been painted, the range falloff ensures that the other side is unaffected, thereby creating a discontinuity in the painted values V. Although the grid values are smooth, the output value map is piecewise-smooth and respects the strong edges of E because we slice according to E. Figure 3-1 illustrates this process on a 1D image.

This simple example demonstrates the edge-aware properties of the bilateral grid. Although we manipulate only smooth values and ignore the issue of edge preservation, the resulting function generated in image space is piecewise-smooth and respects the discontinuities



(a) grid and reference function (b) reconstructed values

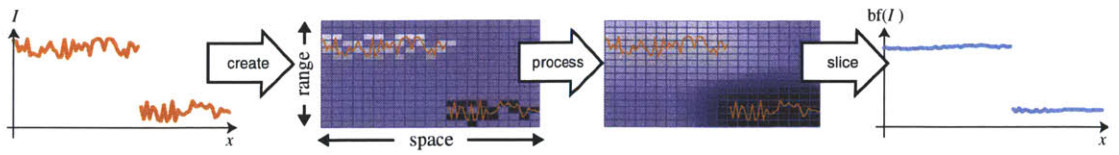
Figure 3-1: *Example of brush painting on a 1D image E. When the user clicks at location x, we add a smooth brush shape in the grid at location $\big(x, E(x)\big)$. The image space result of the brush operation at a position x is obtained by interpolating the grid values at location $\big(x, E(x)\big)$. Although the grid is smooth, we obtain an image space result that respects the image discontinuity.*

of the reference image thanks to the final slicing operation. Furthermore, computations using the bilateral grid are generally independent and require a coarser resolution than the reference image. This makes it easy to map grid operations onto parallel architectures.

3.2.2 Definition

Data Structure The bilateral grid is a three dimensional array, where the first two dimensions (x, y) correspond to 2D position in the image plane and form the spatial domain, while the third dimension z corresponds to a reference range. Typically, the range axis is image intensity.

Sampling A bilateral grid is regularly sampled in each dimension. We name s_s the sampling rate of the spatial axes and s_r the sampling rate of the range axis. Intuitively, s_s controls the amount of smoothing, while s_r controls the degree of edge preservation. The number of grid cells is inversely proportional to the sampling rate: a smaller s_s or s_r yields a larger number of grid cells and requires more memory. In practice, most operations on the grid require only a coarse resolution, where the number of grid cells is much smaller than the number of image pixels. In our experiments, we use between 2048 and 3 million grid



(a) input 1D image (b) grid created from 1D image (c) filtered grid and slicing image (d) filtered 1D image

Figure 3-2: *Bilateral filtering using a bilateral grid demonstrated on a 1D example. Blue grid cells are empty ($w = 0$).*

cells for the useful range of parameters. The required resolution depends on the operation and we will discuss it on a per-application basis.

Data Type and Homogeneous Values The bilateral grid can store in its cells any type of data such as scalars and vectors. For many operations on the bilateral grid, it is important to keep track of the number of pixels (or a weight w) that correspond to each grid cell. Hence, we store *homogeneous* quantities such as (wV, w) for a scalar V or (wR, wG, wB, w) if we deal with RGB colors. This representation makes it easy to compute weighted averages: $(w_1V_1, w_1) + (w_2V_2, w_2) = \big((w_1V_1 + w_2V_2), (w_1 + w_2)\big)$ where normalizing by the homogeneous coordinate $(w_1 + w_2)$ yields the expected result of averaging V_1 and V_2 weighted by w_1 and w_2. Intuitively, the homogeneous coordinate w encodes the importance of its associated data V. It is also similar to the homogeneous interpretation of premultiplied alpha colors [15, 97].

3.2.3 Basic Usage of a Bilateral Grid

Since the bilateral grid is inspired by the bilateral filter, we use it as the primary example of a grid operation. We describe a set of new manipulations in Section 3.4. In general, the bilateral grid is used in three steps. First, we create a grid from an image or other user input. Then, we perform processing inside the grid. Finally, we slice the grid to reconstruct the output (Fig. 3-2). Construction and slicing are symmetric operations that convert between image and grid space.

Grid Creation Given an image I normalized to $[0, 1]$, s_s and s_r, the spatial and range sampling rates, we construct the bilateral grid Γ as follows:

1. *Initialization*: For all grid nodes (i, j, k), $\Gamma(i, j, k) = (0, 0)$.

2. *Filling*: For each pixel at position (x, y):

$$\Gamma\left([x/s_\text{s}],\ [y/s_\text{s}],\ [I(x,y)/s_\text{r}]\right) \mathrel{+}= (I(x,y), 1)$$

where $[\,\cdot\,]$ is the closest-integer operator. We use the notation $\Gamma = c\,(I)$ for this construction. Note that we accumulate both the image intensity and the number of pixels into each grid cell using homogeneous coordinates.

Processing Any function f that takes a 3D function as input can be applied to a bilateral grid Γ to obtain a new bilateral grid $\tilde{\Gamma} = f(\Gamma)$. For the bilateral filter, f is a convolution by a Gaussian kernel, where the variance along the domain and range dimensions are σ_s and σ_r respectively [61].

Extracting a 2D Map by Slicing Slicing is the critical bilateral grid operation that yields a piecewise-smooth output. Given a bilateral grid Γ and a reference image E, we extract a 2D value map M by accessing the grid at $\left(x/s_\text{s}, y/s_\text{s}, E(x,y)/s_\text{r}\right)$ using trilinear interpolation. We use the notation $M = s_E(\Gamma)$. If the grid stores homogeneous values, we first interpolate the grid to retrieve the homogeneous vector; then, we divide the interpolated vector to access the actual data.

Slicing is symmetric to the creation of the grid from an image. If a grid matches the resolution and quantization of the image used for creation, slicing will result in the same image; although in practice, the grid is much coarser than the image. Processing in the grid between creation and slicing is what enables edge-aware operations. Moreover, the particular grid operation determines the required sampling rate.

Recap: Bilateral Filtering Using our notation, the algorithm by Paris and Durand [61] to approximate the bilateral filter in Equation 3.1 becomes (Fig. 3-2):

$$bf(I) \quad = \quad s_I\Big(G_{\sigma_s,\sigma_r} \otimes c(I) \Big) \tag{3.2}$$

We embed the image I in a bilateral grid, convolve this grid with a 3D Gaussian kernel with spatial parameter σ_s and range parameter σ_r, and slice it with the input image. A contribution of our work is to demonstrate that the bilateral grid extends beyond bilateral filtering and enables a variety of edge-preserving processing.

3.3 GPU Parallelization

In developing the bilateral grid, one of our major goals was to facilitate parallelization on graphics hardware. Our benchmarks showed that on a CPU, the bottleneck lies in the slicing stage, where the cost is dominated by trilinear interpolation. We take advantage of hardware texture filtering on the GPU to efficiently perform slicing. The GPU's fragment processors are also ideally suited to executing grid processing operations such as Gaussian blur.

3.3.1 Grid Creation

To create a bilateral grid from an image, we accumulate the value of each input pixel into the appropriate grid voxel. Grid creation is inherently a scatter operation since the grid position depends on a pixel's value. Since the vertex processor is the only unit that can perform a true scatter operation [38], we rasterize a vertex array of single pixel points and use a vertex shader to determine the output position. On modern hardware, the vertex processor can efficiently access texture memory. The vertex array consists of (x, y) pixel positions; the vertex shader looks up the corresponding image value $I(x, y)$ and computes the output position. On older hardware, however, vertex texture fetch is a slow operation. Instead, we store the input image as vertex color attribute: each vertex is a record (x, y, r, g, b) and we can bypass the vertex texture fetch. The disadvantage of this approach is that during slicing, where the input image needs to be accessed as a texture, we must copy the data. For this, we use the pixel buffer object extension to do a fast copy within GPU memory.

Data Layout We store bilateral grids as 2D textures by tiling the z levels across the texture plane. This layout lets us use hardware bilinear texture filtering during the slicing stage and reduce the number of texture lookups from 8 to 2. To support homogeneous coordinates, we use four-component, 32-bit floating point textures. In this format, for typical values of $s_s = 16$ and $s_r = 0.07$ (15 intensity levels), a bilateral grid requires about 1 megabyte of texture memory per megapixel. For the extreme case of $s_s = 2$, the storage cost is about 56 megabytes per megapixel. In general, a grid requires $16 \times$ number of pixels$/(s_s{}^2 \times s_r)$ bytes of texture memory. Grids that do not require homogeneous coordinates are stored as single-component floating point textures and require $1/4$ the memory. In our examples, we use between 50 kB and 40 MB.

3.3.2 Low-Pass Filtering

As described in Section 3.2.3, the grid processing stage of the bilateral filter is a convolution by a 3D Gaussian kernel. In constructing the grid, we set the the sampling rates s_s and s_r to correspond to the bandwidths of the Gaussian σ_s and σ_r, which provides a good tradeoff between accuracy and storage [61]. Since the Gaussian kernel is separable, we convolve the grid in each dimension with a 5-tap 1D kernel using a fragment shader.

3.3.3 Slicing

After processing the bilateral grid, we slice the grid using the input image I to extract the final 2D output. We slice on the GPU by rendering I as a textured quadrilateral and using a fragment shader to look up the stored grid values in a texture. To perform trilinear interpolation, we enable bilinear texture filtering, fetch the two nearest z levels, and interpolate between the results. By taking advantage of hardware bilinear interpolation, each output pixel requires only 2 indirect texture lookups.

3.3.4 Performance

We benchmark the grid-based bilateral filter on three generations of GPUs on the same workstation. Our GPUs consist of an NVIDIA GeForce 8800 GTX (G80), which features unified shaders, a GeForce 7800 GT (G70) and a GeForce 6800 GT (NV40). They were released in 2006, 2005, and 2004, respectively. Our CPU is an Intel Core 2 Duo E6600 (2.4 GHz, 4MB cache).

The first benchmark varies the image size while keeping the bilateral filter parameters constant ($\sigma_s = 16, \sigma_r = 0.1$). We use the same image subsampled at various resolutions and report the average runtime over 1000 iterations. Figure 3-3 shows that our algorithm is linear in the image size, ranging from 4.5ms for 1 megapixel to 44.7ms for 10 megapixels on the G80. We consistently measured slowdowns beyond 9 megapixels for the older GPUs, which we suspect is due to an overflow in the vertex cache. For comparison, our CPU implementation ranges from 0.2 to 1.9 seconds on the same inputs. Our GPU implementation outperforms Weiss's CPU bilateral filter [96] by a factor of 50.

The second benchmark keeps the image size at 8 megapixels and the range kernel size σ_r at 0.1 while varying the spatial kernel size σ_s. As with the first benchmark, we use the same image and report the average runtime over 1000 iterations. Figure 3-4 shows the influence of the kernel size. With the exception of a few bumps in the curve at $\sigma_s = 30$ and $\sigma_s = 60$ that we suspect to be due to hitting a framebuffer cache boundary, our algorithm is independent of spatial kernel size. This is due to the fact that the 3D convolution kernel on the bilateral grid is independent of σ_s. As we increase σ_s, we downsample more aggressively; therefore, the convolution kernel remains $5 \times 5 \times 5$. For comparison, our CPU implementation and Weiss's algorithm both consistently run in about 1.6s over the same values of σ_s.

All three generations of hardware achieve real-time performance (30 Hz) at 2 megapixels,

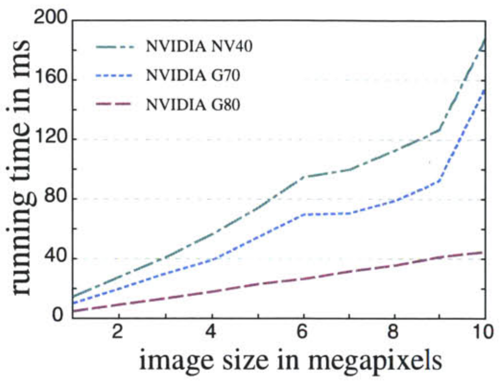


Figure 3-3: *Bilateral filter running times as a function of the image size (using $\sigma_s = 16$ and $\sigma_r = 0.1$). The memory requirements increase linearly from 625 kB at 1 megapixel to 6.25 MB at 10 megapixels.*

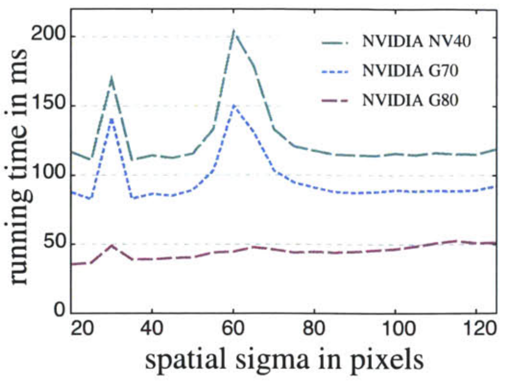


Figure 3-4: *Bilateral filter running times as a function of the spatial sigma (8 megapixels, $\sigma_r = 0.1$). The memory requirements decrease quadratically from 3.2 MB at $\sigma_s = 20$ down to 130 kB at $\sigma_s = 120$.*

the current 1080p HD resolution. The G80 attains the same framerate at 7 megapixels. On all generations of hardware, the bottleneck of our algorithm lies in the grid construction stage; the other stages are essentially free. We have verified with an OpenGL profiler that on older hardware with a fixed number of vertex processors, the algorithm is vertex limited while the rest of the pipeline is starved for data. This situation is largely improved on current hardware because it features unified shaders. The bottleneck then lies in the raster operations units.

3.3.5 Further Acceleration

On current hardware, we can run multiple bilateral filters per frame on 1080p HD video, but on older hardware, we are limited to a single filter per frame. For temporally coherent data, we propose an acceleration based on subsampling. A cell of the grid stores the weighted average of a large number of pixels and we can obtain a good estimate with only a subset of those pixels. For typical values of $\sigma_s \in [10, 50]$ and $\sigma_r \in [0.05, 0.4]$, using only 10% of the input pixels produces an output with no visual artifacts. We choose the 10% of pixels by rotating through a sequence of precomputed Poisson-disk patterns to obtain a good coverage. To combat "swimming" artifacts introduced by time-varying sampling patterns, we apply a temporal exponential filter with a decay constant of 5 frames. This produces results visually indistinguishable from the full bilateral filter except at hard scene transitions.

3.4 Image Manipulation with the Bilateral Grid

The bilateral grid has a variety of applications beyond bilateral filtering. The following sections introduce new ways of creating, processing and slicing a bilateral grid.

3.4.1 Cross-Bilateral Filtering

A direct extension to the bilateral filter is the *cross-bilateral filter* [27,64], where the notion of image data is decoupled from image edges. We define a new grid creation operator with two parameters: an image I, which defines the grid values, and an edge image E which determines the grid position.

$$\Gamma\left(\left[\frac{x}{s_\mathrm{s}}\right], \left[\frac{y}{s_\mathrm{s}}\right], \left[\frac{E(x,y)}{s_\mathrm{r}}\right]\right) \mathrel{+}= (I(x,y), 1) \tag{3.3}$$

We use the notation $\Gamma = c_E(I)$ for this operator. Analogously, the slicing operator uses the edge image E to query the grid and reconstruct the result:

$$cbf(I,E) = s_E\Big(G_{\sigma_\mathrm{s},\sigma_\mathrm{r}} \otimes c_E(I) \Big) \tag{3.4}$$

3.4.2 Grid Painting

We now elaborate on the edge-preserving brush mentioned in Section 3.2.1. Analogous to a classical 2D brush, where users locally "paint" characteristics such as brightness, the bilateral brush further ensures that modifications do not "bleed" across image contours (Figure 3-1). We use a bilateral grid Γ_brush to control a 2D influence map M that defines the strength of the applied modification. We initialize a scalar (i.e., non-homogeneous) grid Γ_{brush} to 0. When the user clicks on an image pixel $(x, y, I(x, y))$, we add to the grid values a 3D brush (e.g., a Gaussian) centered at position $\left(x/s_s, y/s_s, I(x,y)/s_r\right)$. We set s_s to the spatial bandwidth of the Gaussian brush shape, and s_r is set to the desired degree of edge preservation. We obtain M by slicing Γ_{brush} with the image I: $M = s_I(\Gamma_{\text{brush}})$.

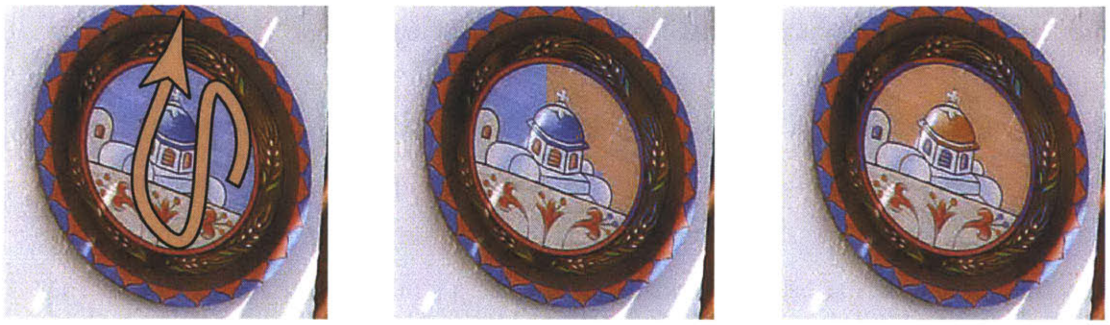


Figure 3-5: *Bilateral Grid Painting allows the user to paint without bleeding across image edges. The user clicks on the input (a) and strokes the mouse. The intermediate (b) and final (c) results are shown. The entire 2 megapixel image is updated at 60 Hz. Memory usage was about 1.5 MB for a* 20×20 *brush and* s_r *= 0.05.*

GPU Implementation We tile Γ_{brush} as a single-component 2D texture. When the user clicks the mouse, a fragment shader renders the brush shape using blending. Slicing is identical to the case of the bilateral filter. A modern GPU can support bilateral grid painting on very large images. For a 2×2 brush with $s_r = 0.05$, the grid requires 20 MB of texture memory per megapixel; a 5×5 brush consumes less than 1 MB per megapixel.

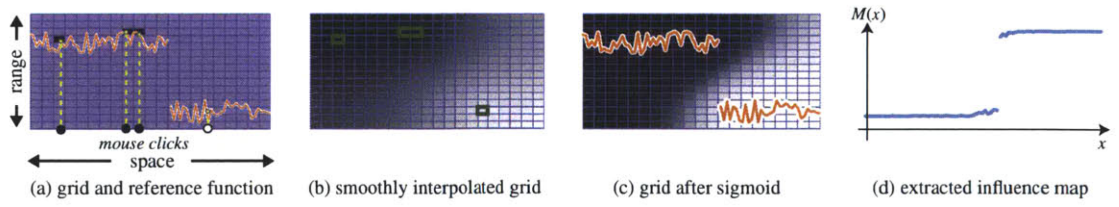

range
mouse clicks
space
(a) grid and reference function
(b) smoothly interpolated grid
(c) grid after sigmoid
M(x)
x
(d) extracted influence map

Figure 3-6: *Edge-aware interpolation with a bilateral grid demonstrated on a 1D example. The user clicks on the image to indicate sparse constraints (a) (unconstrained cells are shown in blue). These values are interpolated into a smooth function (b) (constraints are shown in green). We filter the grid using a sigmoid to favor consistent regions (c). The resulting is sliced with the input image to obtain the influence map* M *(d).*

Results In Figure 3-5, the user manipulates the hue channel of an image without creating a mask. An initial mouse click determines (x_0, y_0, z_0) in the grid. Subsequent mouse strokes vary in x and y, but z_0 is fixed. Hence, the brush affects only the selected intensity layer and does not cross image edges.

3.4.3 Edge-Aware Scattered Data Interpolation

Inspired by Levin et al. [48], Lischinski et al. [54] introduced a scribble interface to create an influence map M over an image I. The 2D map M interpolates a set of user-provided constraints $\{M(x_i, y_i) = m_i\}$ (the scribbles) while respecting the edges of the underlying image I. We use a scalar bilateral grid Γ_{int} to achieve a similar result: instead of solving a *piecewise-smooth* interpolation in the image domain, we solve a *smooth* interpolation in the grid domain and then slice.

We lift the user-provided constraints into the 3D domain:

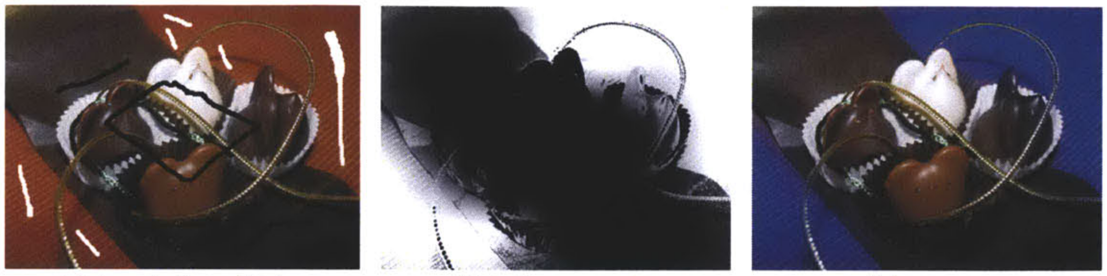
$$\{\Gamma_{\text{int}}\left([x/s_{\text{s}}], [y/s_{\text{s}}], [I(x,y)/s_{\text{r}}]\right) = m_i\}, \tag{3.5}$$

and minimize the variations of the grid values:

$$\begin{aligned} &\text{argmin} \int \|\text{grad}(\Gamma_{\text{int}})\|^2 \\ &\text{under the constraints:} \quad \left\{\Gamma_{\text{int}}\left(\left[\tfrac{x}{s_{\text{s}}}\right], \left[\tfrac{y}{s_{\text{s}}}\right], \left[\tfrac{I(x,y)}{s_{\text{r}}}\right]\right) = m_i\right\} \end{aligned} \tag{3.6}$$

The 2D influence map is obtained by slicing: $M = s_I(\Gamma_{\text{int}})$. Finally, we bias the influence map toward 0 and 1 akin to Levin et al. [50]. We achieve this by applying a sigmoid function to the grid values. Figure 3-6 summarizes this process and Figure 3-7 shows a sample result.

Discussion Compared to image-based approaches [48, 54], our method does not work at the pixel level which may limit accuracy in some cases; although our experiments did not reveal any major problems. On the other hand, the bilateral grid transforms a diffi-



(a) input & scribbles (b) influence map (c) output

Figure 3-7: *Fast scribble interpolation using the Bilateral Grid. The user paints scribbles over the input (a). Our algorithm extracts an influence map (b), which is used to adjust the input hue and produce the output (c). The entire 2 megapixel image is updated at 20 Hz. Memory usage was about 62 kB for* $s_s = 256$ *and* $s_r = 0.05$.

cult image-dependent and non-homogeneous 2D optimization into a simpler smooth and homogeneous interpolation in 3D. Furthermore, the grid resolution is decoupled from the resolution of the image, which prevents the complexity from growing with image resolution.

Another difference is that image-based techniques use the notion of "geodesic distance" over the image manifold, while we consider the Euclidean distance in a higher-dimensional space. The comparison of those two measures deserves further investigation and we believe that which method is most appropriate is an application-dependent choice.

GPU Implementation Analogous to grid painting, we rasterize scribble constraints into the bilateral grid using a fragment shader. To obtain a globally smooth bilateral grid that respects the constraints, we solve Laplace's equation by extending a GPU multigrid algorithm [36] to handle irregular 3D domains. The domain has holes because the user-specified hard constraints create additional boundaries.

Results We demonstrate real-time scribble interpolation with a simple color adjustment application. The user paints scribbles over an image; white scribbles denote regions where the hue should be adjusted, while black scribbles protect the image region. In practice, the sampling rate of Γ_{int} can be very coarse and still yield good influence maps. The example in

Figure 3-7 was generated using a coarse grid containing about 600 variables, allowing our GPU solver for generating the influence map in real time (40 Hz). When finer adjustments are required, we can still achieve interactive rates (1 Hz) on finely sampled grids with over 500,000 variables. Refer to the supplemental video for a screen capture of an editing session.

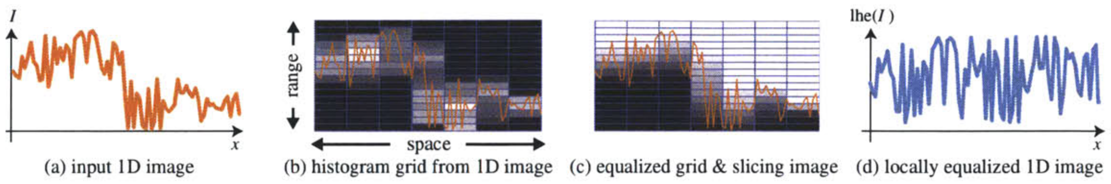

I
x
(a) input 1D image
range
space
(b) histogram grid from 1D image
(c) equalized grid & slicing image
lhe(I)
x
(d) locally equalized 1D image

Figure 3-8: *Local histogram equalization demonstrated on a 1D image. We build a grid that counts the number of pixels in each bin (b). Each grid column corresponds to the histogram of the image region it covers. By equalizing each column, we obtain a grid (c) which leads to an image-space signal with an enhanced contrast that exploits the whole intensity range (d).*

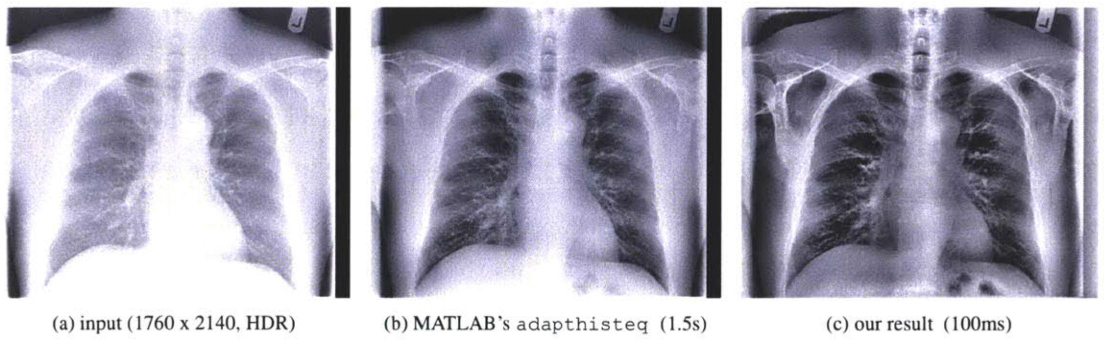


Figure 3-9: *Local histogram equalization reveals low-contrast details by locally remapping the intensity values. The input (a) is an HDR chest X-Ray (tone mapped for display). Our algorithm (c) based on the bilateral grid has a similar visual appearance to MATLAB's* `adapthisteq` *(b) while achieving a speedup of an order of magnitude. For this example, we used $s_s = 243.75$, $s_r = 0.0039$; memory usage was 500 kB total for the two grids.*

3.4.4 Local Histogram Equalization

Histogram equalization is a standard technique for enhancing the contrast of images [35]. However, for some inputs, such as X-Ray and CT medical images that have high dynamic range, histogram equalization can obscure small details that span only a limited intensity range. For these cases, it is more useful to perform histogram equalization locally over image windows. We perform *local histogram equalization* efficiently using a bilateral grid, and achieve real-time performance using the GPU.

Given an input image I, we construct a scalar bilateral grid Γ_{hist}. We initialize $\Gamma_{\text{hist}} = 0$ and fill it with:

$$\Gamma_{\text{hist}}\left(\left[\frac{x}{s_s}\right], \left[\frac{y}{s_s}\right], \left[\frac{I(x,y)}{s_r}\right]\right) \mathrel{+}= 1 \tag{3.7}$$

We denote this operator $\Gamma_{\text{hist}} = c_{\text{hist}}(I)$. Γ_{hist} stores the number of pixels in a grid cell and can be considered a set of local histograms. For each (x, y), the corresponding column splits the $s_s \times s_s$ covered pixels into intensity intervals of size s_r. By using the closest-integer operator when constructing Γ_{hist}, we perform a box filter in space. If a smoother spatial kernel is desired, we blur each z level of the grid by a spatial kernel (e.g., Gaussian). We perform local histogram equalization by applying a standard histogram equalization to each column and slicing the resulting grid with the input image I.

GPU Implementation We construct the bilateral grid the same way as in bilateral filtering, except we can ignore the image data. Next, we execute a fragment shader that accumulates over each (x, y) column of the bilateral grid. This yields a new grid where each (x, y) column is an unnormalized cumulative distribution function. We run another pass to normalize the grid to between 0 and 1 by dividing out the final bin value. Finally, we slice the grid using the input image.

Results Our algorithm achieves results visually similar to MATLAB's `adapthisteq` (Figure 3-9). In both cases, low-contrast details are revealed while the organ shapes are preserved. Our method based on the bilateral grid achieves a speed-up of one order of magnitude: 100ms compared to 1.5s on a 3.7 megapixel HDR image.

3.5 Applications and Results

In this section, we describe a variety of applications which take advantage of image processing using the bilateral grid. Refer to the supplemental video for a demonstration of our results. For the video applications, decoding is performed on the CPU in a separate thread that runs in parallel with the GPU. The timings measure the average throughput for the entire pipeline. On our CPU, the largest input video (1080p resolution using the H.264 codec) takes about 25ms to decode each frame; which means that in many cases, decoding is more expensive than our edge-aware operators and becomes the pipeline bottleneck.

3.5.1 High Resolution Video Abstraction

Winnemöller et al. [98] demonstrated a technique for stylizing and abstracting video in real time. A major bottleneck in their approach was the bilateral filter, which limited the video to DVD resolution (0.3 megapixels) and the framerate to 9 to 15 Hz. To attain this framerate, they used a separable approximation to the bilateral filter with a small kernel size and iterated the approximation to obtain a sufficiently large spatial support [65]. Using the bilateral grid with our GPU acceleration technique (without the additional acceleration described in Section 3.3.5), we are able to perform video abstraction at 42 Hz on 1080p HD video (1.5 megapixels).

Progressive Abstraction To further enhance the technique, we incorporated a progressive abstraction component in the spirit of the stylization technique by DeCarlo et al. [24] where details near a focus point are less abstracted than those far away. In our method, we build a 4-level *bilateral pyramid*—bilateral grids computed at multiples of a base s_s and s_r. To abstract an input frame, we first compute a distance map that falls off from the user's focus point. We ensure that this map respects the image edges by cross-bilateral filtering it with the image as reference to get an *importance map*. We use the importance map to linearly interpolate between the input frame (at the focus point) and the levels of the multiscale bilateral grid. We use the result as input to the rest of the video abstraction pipeline.

We found that extracting the lines from the first pyramid level yields more coherent outputs. With a 4-level pyramid, we are still able to maintain 20 Hz on 1080p video.

3.5.2 Transfer of Photographic Look

Bae et al. [9] introduced a method to transfer the "look" of a model photograph to an input photograph. We adapt their work to operate on videos in real time. We describe two modifications to handle the constraints inherent to video. We use a simplified pipeline that yields real-time performance while retaining most of the effects from the original process. We also specifically deal with noise and compression artifacts to handle sources such as DVDs and movies recorded using consumer-grade cameras.

Processing Pipeline Akin to Bae et al., we use the bilateral filter on each input frame I and name the result the *base layer* $B_\mathrm{i} = bf(I)$ and its residual the *detail layer* $D_\mathrm{i} = I - B_\mathrm{i}$. We perform the same decomposition on the model image M to get B_m and D_m. We use histogram transfer to transform the input base B_i so that it matches the histogram of B_m. We denote by ht the histogram transfer operator and $B_\mathrm{o} = ht_{B_\mathrm{m}}(B_\mathrm{i})$ the base output. For the detail layer, we match the histogram of the amplitudes: $|D_\mathrm{o}| = ht_{|D_\mathrm{m}|}(|D_\mathrm{i}|)$. We obtain the detail layer of the output by using the sign of the input detail: $D_\mathrm{o} = \mathrm{sign}(D_\mathrm{i})\,|D_\mathrm{o}|$. The output frame O is reconstructed by adding the base and detail layers: $O = B_\mathrm{o} + D_\mathrm{o}$.

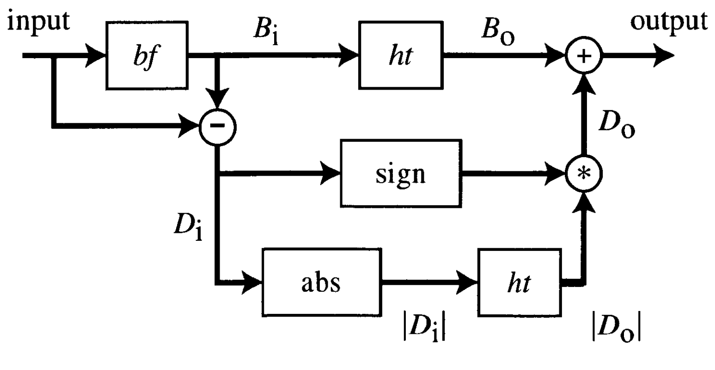


Figure 3-10: *Tone management pipeline.*

Denoising A number of videos are noisy or use a compression algorithm that introduces artifacts. These defects are often not noticeable in the original video but may be revealed as we increase the contrast or level of detail. A naïve solution would be to denoise the input frames before processing them but this produces "too clean" images that look unrealistic. We found that adding back the denoising residual after processing yields superior results with a more realistic appearance. In practice, since noise amplitude is low compared to scene edges, we use a bilateral filter with small sigmas.

Discussion Compared to the process described by Bae et al., our method directly relies on the detail amplitudes to estimate the level of texture of a frame. Although the *textureness* measure proposed by Bae et al. capture more sophisticated effects, it induces three additional bilateral filtering steps whose computational cost would prevent our algorithm to run in real time on HD sequences. Our results show that detail amplitude is a satisfying approximation. Furthermore, it provides sufficient leeway to include a denoising step that broadens the range of possible inputs.

3.5.3 Local Tone Mapping

We describe a user-driven method to locally tone map HDR images based on grid painting. We build upon Durand and Dorsey's tone mapping algorithm [25], where the log luminance L of an image is decomposed into a base layer $B = bf(L)$ and a detail layer $D = L - B$. The contrast of the base is reduced using a simple linear remapping $B' = \alpha B + \beta$ while the detail layer D is unaffected. This reduces the overall dynamic range without losing local detail. The final output is obtained by taking the exponential of $B' + D$ and preserving color ratios.

Our method extends this global remapping of the base layer and lets users locally modify the remapping function using an edge-aware brush. We represent the remapping function with a grid Γ_{TM} initialized with a linear ramp:

$$\Gamma_{\text{TM}}(x, y, z) = \alpha z + \beta \tag{3.8}$$

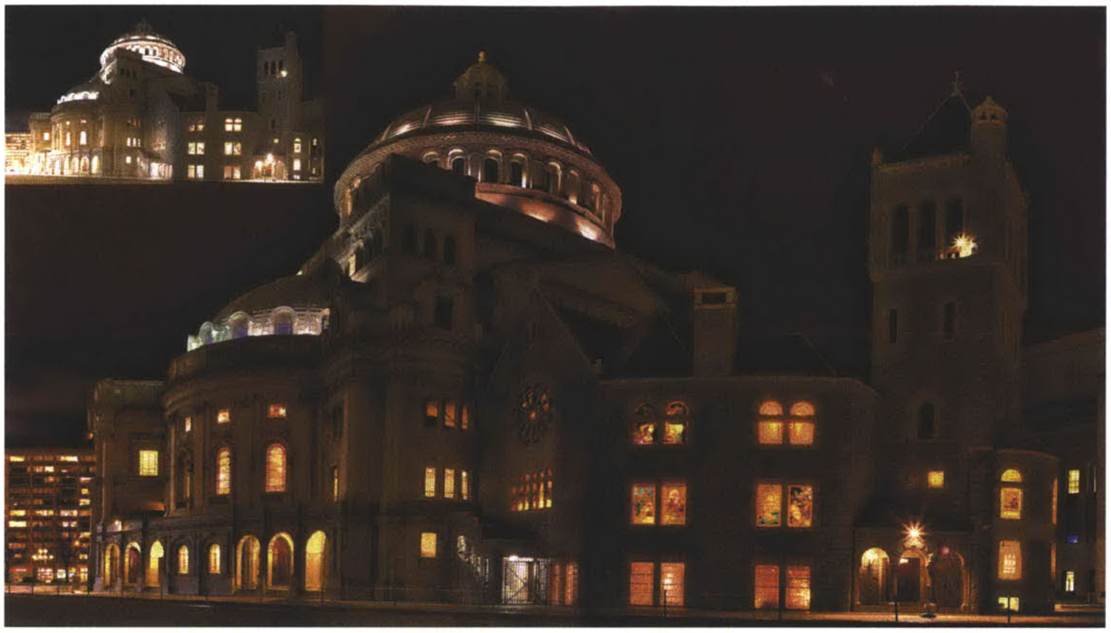


Figure 3-11: *The bilateral grid enables edge-aware image manipulations such as local tone mapping on high resolution images in real time. This 15 megapixel HDR panorama was tone mapped and locally refined using an edge-aware brush at 50 Hz. The inset shows the original input. The process used about 1 MB of texture memory.*

If Γ_{TM} is unedited, slicing Γ_{TM} with B yields the same remapped base layer as Durand and Dorsey's operator: $B' = s_B(\Gamma_{\text{TM}})$.

Users edit Γ_{TM} with an edge-aware brush to locally modify the grid values. The modified base layer is still obtained by slicing according to B. Clicking with the left button on the pixel at location (x, y) adds a 3D Gaussian centered at $(x/s_{\text{s}}, y/s_{\text{s}}, L(x, y)/s_{\text{r}})$ to the grid values. A right click subtracts a 3D Gaussian. In practice, we use Gaussian kernels with a user-specified amplitude A and parameters $\sigma_{\text{s}} = s_{\text{s}}$ and $\sigma_{\text{r}} = s_{\text{r}}$. The spatial sampling s_{s} controls the size of the brush and the range sampling s_{r} controls its sensitivity to edges. If users hold the mouse button down, we lock the z coordinate to the value of the first click $L(x_0, y_0)/s_{\text{r}}$, thereby enabling users to paint without affecting features at different intensities.

Using our GPU algorithm for the bilateral filter that creates the base layer and for grid painting, we tone map a 15 megapixel image at 50 Hz (Figure 3-11). Refer to the video for a screen capture of a local tone mapping session.

3.6 Discussion and Limitations

Memory Requirements Our approach draws its efficiency from the coarser resolution of the bilateral grid compared to the 2D image. However, operations such as a bilateral filter with a small spatial kernel require fine sampling, which results in large memory and computation costs for our technique. In this case, Weiss's approach is more appropriate [96]. Nonetheless, for large kernels used in computational photography applications, our method is significantly faster than previous work.

Due to memory constraints, the bilateral grid is limited to a one-dimensional range that stores image intensities and can cause problems at isoluminant edges. We found that in most cases, we achieve good results with 7 to 20 levels in z. A future direction of research is to consider how to efficiently store higher dimensional bilateral grids: 5D grids that can handle color edges and even 6D grids for video. Another possibility is to look at fast dimensionality reduction techniques to reduce the memory limitations.

Interpolation We rely on trilinear interpolation during slicing for optimal performance. Higher-order approaches can potentially yield higher-quality reconstruction. We would like to investigate the tradeoff between quality and cost in using these filters.

Thin Features Techniques based on the bilateral grid have the same properties as the bilateral filter at thin image features. For example, in an image with a sky seen through a window frame, the edge-aware brush affects the sky independently of the frame; that is, the brush paints across the frame without altering it. Whether or not a filter stops at thin features is a fundamental difference between bilateral filtering and diffusion-based techniques. We believe that both behaviors can be useful, depending on the application.

3.7 Conclusion and Hindsights

We have presented a new data structure, the bilateral grid, that enables real-time edge-preserving image manipulation. By lifting image processing into a higher dimensional space, we are able to design algorithms that naturally respect strong edges in an image. We demonstrated the practicality of our technique with a number of sophisticated video processing pipelines applied to high-definition video. Our examples showed that edge-aware filters can be composably mapped onto GPUs in conjunction with other filters in a full shade graph. We conclude this chapter with some discussion of the key ideas that made our approach work, how it could be done differently looking back after four years, and comparison with followup work after the original publication.

The bilateral grid was inspired by the fast bilateral filter technique by Paris and Durand [61]. It represented a grayscale image not as a mapping $\mathbb{R}^2 \rightarrow \mathbb{R}$, but as a sparse collection of delta functions in $\mathbb{R}^3$. Our key insight is that, in this representation, which we eventually quantized into a 3D grid, computations that generate smooth functions are piecewise-smooth once sliced with the input image. This paper expands on this idea by formally calling the representation a data structure, which supports a number of other edge-aware operators.

3.7.1 Implementation on 2011 hardware

The GPU algorithms in this chapter were developed when GPU computing was in its infancy. We relied purely on OpenGL primitives for both grid construction (vertex scatter) and computations (Gaussian blur, multi-grid solver, prefix-sum). These operations are substantially simpler and more efficient with modern GPU computing tools. In particular, there are fast algorithms for histogram generation / scatter [67] and a linear-time algorithm for prefix scan [78] (we used a brute force quadratic time algorithm to build the CDFs for local histogram equalization).

Memory layout is another factor which we did not consider because we were limited to using four-component floating point 2D textures (at the time, we could not render into either two-component or 3D textures). Memory access coherence can significantly influence performance on modern hardware. If we were to implement it today, we would definitely consider other possible layouts to maximize memory bandwidth.

Related to memory layout, when we first developed the bilateral grid, we did not consider how much space we were actually wasting. This point was first brought up when the paper was under review, and we realized that the representation uses memory exponential in the number of dimensions. Followup work showed that most of the speed advantages of the bilateral grid comes from downsampling: larger filters actually make smaller grids.

3.7.2 Discussion of followup work

Subsequent to the publication of our work, the area of edge-aware filtering and interpolation continued to receive considerable attention. We survey a selection of followup works and discuss how they relate to our approach.

Fast bilateral filtering A number of researchers have looked at alternative formulations of the bilateral filter. Porikli [68] extends the work by Weiss [96] to compute exact and approximate bilateral filters on grayscale images in "constant time": the computation time is independent of the kernel size (we would call this *linear time* since it is still linear in the size of the image!). The algorithm relies on spatial integral histograms, which essentially discretizes the intensity range to a fixed number of bins and makes a different memory-computation tradeoff than the one we make. It also has some interesting tradeoffs in the kinds of kernels it can support: either a box domain kernel with an arbitrary range kernel, or an arbitrary spatial kernel with a polynomial range kernel (which can be used to approximate a Gaussian using Taylor series expansion). Because it relies on integral histograms, it is not as parallelizable as our method.

Yang et al. [100] improved on the Porikli's algorithm by taking inspiration from the fast bilateral filtering technique by Durand and Dorsey [25]. Like the other two methods, this algorithm discretizes the intensity range, computes a linear filter at each level, and interpolates between them. Their key observation is that if the spatial filter can be computed in "constant time", such as in the case of the box (using integral histograms), Gaussian (with an IIR implementation), or polynomial (with a set of integral histograms), then the methods can be directly combined. It also has the advantage of being parallelizable both across and between levels, achieving excellent performance on GPUs at the cost of not being able to handle HDR content.

Higher-dimensional filtering A key limitation of the bilateral grid is the classic *curse of dimensionality*. By lifting the our data into higher-dimensional space and representing the entire space with a uniform grid, the data structure consumes memory exponential in the dimension. Although this memory-speed tradeoff is works for grayscale images (a 3D grid), it our technique is infeasible with a 5D grid for color bilateral filtering or 6D grid for video.

Adams et al. [3] showed that in addition to bilateral filtering, a number of related filters including the color and space-time bilateral filter, and the non-local means filter [16], can be simply expressed as a high-dimensional Gaussian filter. They then used a kd-tree rather than a uniform grid to achieve an $O(dn\ log\ n)$ algorithm for d-dimensional filtering on an image with n pixels. Adams et al. [2] extended their earlier work to scale to even higher dimensions using a clever geometric hashing algorithm, achieving a $O(d^2(n + l))$ time algorithm (where $l \approx n$). The paper also features an excellent empirical analysis of the parameter space, showing which data structure performs well in each regime.

Multi-scale representations Our paper demonstrated an edge-aware extension to the classic Gaussian pyramid in the form of the bilateral pyramid. Following our work, Farbman et al. [29] showed that the bilateral filter is not a good primitive for an edge-preserving multi-scale decomposition. Increasing the spatial variance of the bilateral filter does not always increase its spatial smoothing power and can cause textures to "leak" through. Furthermore, increasing the range variance is not a good strategy either: the filter simply approaches a Gaussian. They instead proposed an alternative filter called *weighted least squares filter*. Rather than using an explicitly nonlinear filtering kernel, they express the same goal as an optimization where the objective is to compute the smoothest possible image but where the weights vary depending on the local image gradient. This strategy lets them compute an edge-preserving Laplacian pyramid that exhibits fewer artifacts at edges. Fattal [30] improves the performance of the weighted least squares method by casting it into a wavelet framework. By using the same weights but expressed as wavelet coefficients, he can avoid solving a poorly-conditioned sparse linear system over a large image and compute a wavelet transform in linear time.

Concurrent with our work is the joint bilateral upsampling algorithm by Kopf et al. [46]. Inspired by the cross-bilateral filter [27, 64], joint bilateral upsampling uses the edges defined in the input image to upsample related data from a lower resolution (for instance, a less accurate depth map, or an expensive to compute scribble interpolation). The bilateral grid can easily be adapted for fast joint-bilateral upsampling.

Alternative edge-aware representations The bilateral grid uses a simple model of edges: differences in image intensity. However, depending on the application, other definitions may be desirable. As we already discussed, for some types of images, intensity differences are insufficient and a color bilateral filter is needed. For multi-scale tonal adjustment, the weighted least squares filter [29] appears to be the right choice. Subr et al. [85] propose an interesting alternative targeted at enhancing image detail based on modeling textures as *local oscillations*. They compute an envelope of local minima and maxima, and defines detail as small-scale oscillations within the envelope. Edges are regions that exhibit high variance in their local extrema. This model allows them to capture and enhance fine details that other models typically miss.

Farbman et al. [28] formulates a very different notion of edges and pixel similarity. Unlike most popular definitions that transform the definition into a Euclidean distance in some feature space, *diffusion maps* casts the affinity between pixels as the probability that a random walker would travel from one pixel to another under a Gaussian distribution over their color difference. Instead of capturing local edges, fine-scale details, or preserving edges over multiple spatial scales, diffusion maps features a "time" parameter that lets the user select progressively larger sets of pixels independent of pixel position. This metric captures a semantically different notion of similarity (where one would define edges as "not similar") and proves useful in selecting similar objects spread across an image.

3.7.3 Future work

Given the amount of followup work in this area, edge-aware image processing remains an active area of research. One direction we would like to explore is using the bilateral grid for edge-aware interpolation of other kinds of data. For instance, we can use the grid for edge-preserving optical flow interpolation with explicit temporal coherence, using the homogeneous weight as a confidence. The dense flow field could, for instance, be combined with scribbles to yield a time-varying edge-preserving influence map.

Chapter 4

The Video Mesh

4.1 Motivation

A variety of video editing tasks require dense structured information to achieve high quality results. For instance, consider the task of inserting a logo onto the shirt of an actor. The artist begins by painting the logo onto one frame of the video. In order to propagate the logo to other frames, we need dense motion information: where does each pixel go in the next frame. We also need depth information: if the actor waves his arms, the logo should be appropriately occluded. In general, raw video is too unstructured for complex video editing operations. For these tasks, we need a higher-level representation for motion, depth, and occlusions.

We leverage a number of existing computational photography techniques to provide user-assisted tools for the creation of a video mesh from an input video. Feature tracking provides motion information. Rotoscoping [7] and matting (e.g., [21,49,95]) enable fine handling of occlusion. A combination of structure-from-motion [39] and interactive image-based modeling [42,59] permit a semi-automatic method for estimating depth. The video mesh enables a variety of video editing tasks such as changing the 3D viewpoint, occlusion-aware compositing, 3D object manipulation, depth-of-field manipulation, conversion of video from 2D to 3D, and relighting.

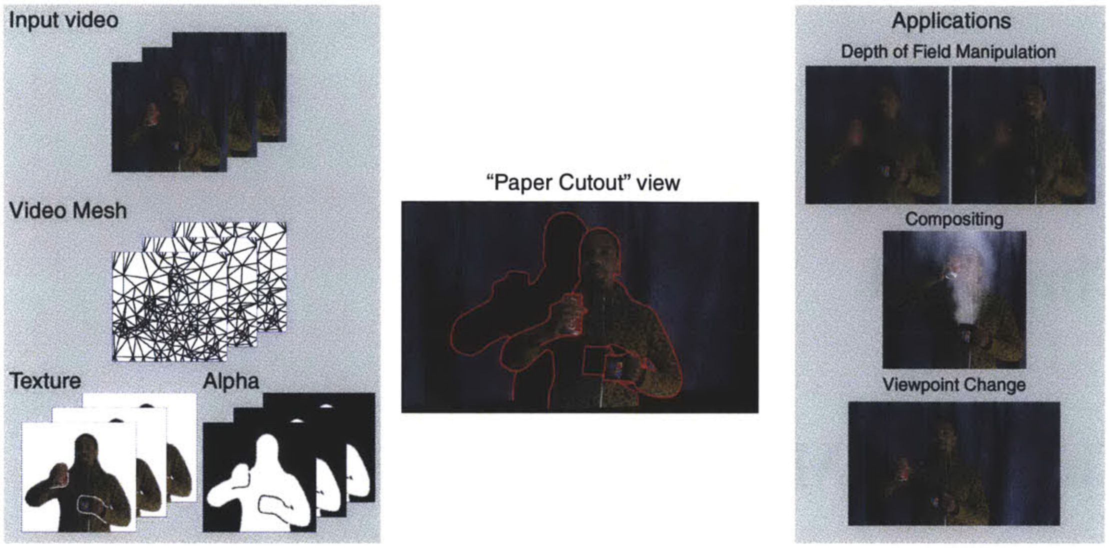


Figure 4-1: *The video mesh data structure represents the structural information in an input video as a set of deforming texture mapped triangles augmented with mattes. The mesh has a topology that resembles a "paper cutout". This representation enables a number of special effects applications such as depth of field manipulation, object insertion, and change of 3D viewpoint.*

We make the following contributions [1]:

- The video mesh, a sparse data structure for representing motion and depth in video that models the world as "paper cutouts."
- Algorithms for constructing video meshes and manipulating their topology. In particular, we introduce a robust mesh cutting algorithm that can handle arbitrarily complex occlusions in general video sequences.
- Video-based modeling tools for augmenting the structure of a video mesh, enabling a variety of novel special effects.

[1]A version of this chapter appeared as a paper at ICCP 2011 [19]. This version features details left our due to space limitations and hindsights on how to design a video editing system around the video mesh data structure.

4.1.1 Related work

Mesh-based video processing Meshes have long been used in video processing for tracking, motion compensation, animation, and compression. The Particle Video system [76], uses a triangle mesh to regularize the motion of tracked features. Video compression algorithms [17] use meshes to sparsely encode motion. These methods are designed for motion compensation and handle visibility by resampling and remeshing along occlusion boundaries. They typically do not support self-occlusions. In contrast, our work focuses on using meshes as the central data structure used for editing. In order to handle arbitrary video sequences, we need a general representation that can encode the complex occlusion relationships in a video. The video mesh decouples the complexity of visibility from that of the mesh by encoding it with a locally dense alpha map. It has the added benefit of handling partial coverage and sub-pixel effects.

Motion description Motion in video can be described by its dense optical flow, e.g. [41]. We have opted for a sparser treatment of motion based on feature tracking, e.g. [56, 80]. We find feature tracking more robust and easier to correct by a user. Feature tracking is also much cheaper to compute and per-vertex data is easier to process on GPUs.

Video representations The video mesh builds upon and extends layer-based video representations [4, 93], video cube segmentation [94], and video cutouts [52]. Commercial packages use stacks of layers to represent and composite objects. However, these layers remain flat and cannot handle self-occlusions within a layer such as when an actor's arm occludes his body. Similarly, although the video cube and video cutout systems provide a simple method for extracting objects in space-time, to handle self-occlusions, they must cut the object at an arbitrary location. The video mesh leverages user-assisted rotoscoping [7] and matting [21, 49, 95] to extract general scene components without arbitrary cuts.

Background collection and mosaicking can be used to create compound representations, e.g.,, [43, 87]. Recently, Rav-Acha et al. [72] introduced Unwrap Mosaics to represent object texture and occlusions without 3D geometry. High accuracy is achieved through a sophisticated optimization scheme that runs for several hours. In comparison, the video mesh outputs coarse results with little precomputation and provides tools that let the user interactively refine the result. Unwrap Mosaics are also limited to objects with a disc topology whereas the video mesh handles more general scenes.

Image-based modeling and rendering We take advantage of existing image-based modeling techniques to specify depth information at vertices of the video mesh. In particular, we adapt a number of single-view modeling tools to video [42,59,102]. We are also inspired by the Video Trace technique [92] which uses video as an input to interactively model static objects. We show how structure-from-motion [39] can be applied selectively to sub-parts of the video to handle piecewise-rigid motion which are common with everyday objects. We also present a simple method that propagates depth constraints in space.

Stereo video Recent multi-view algorithms are able to automatically recover depth in complex scenes from video sequences [77]. However, these techniques require camera motion and may have difficulties with non-Lambertian materials and moving objects. Zhang et al. demonstrate how to perform a number of video special effects [101] using depth maps estimated using multi-view stereo. Recent work by Guttman et al. [37] provides an interface to recovering video depth maps from user scribbles. The video mesh is complementary to these methods. We can use depth maps to initialize the 3D geometry and our modeling tools to address challenging cases such as scenes with moving objects.

By representing the scene as 2.5D paper cutouts, video meshes enable the conversion of video into stereoscopic 3D by re-rendering the mesh from two viewpoints. A number of commercial packages are available for processing content filmed in with a stereo setup [71,89]. These products extend traditional digital post-processing to handle 3D video with features such as correcting small misalignments in the stereo rig, disparity map estimation, and inpainting. The video mesh representation would enable a broader range of effects while relying mostly on the same user input for its construction. Recent work by Koppal et al. [47], describes a pre-visualization system for 3D movies that helps cinematographers plan their final shot from draft footage. In comparison, our approach aims to edit the video directly.

4.2 The video mesh data structure

We begin by describing the properties of the video mesh data structure and illustrate how it represents motion and depth in the simple case of a smoothly moving scene with no occlusions. In this simplest form, it is similar to morphing techniques that rely on triangular meshes and texture mapping [34]. We then augment the structure to handle occlusions, and in particular self-occlusions that cannot be represented by layers without artificial cuts. Our general occlusion representation simplifies a number of editing tasks. For efficient image data storage and management, we describe a tile-based representation for texture and transparency. Finally, we show how a video mesh is rendered.

4.2.1 A triangular mesh

Vertices The video mesh encodes depth and motion information at a sparse set of vertices, which are typically obtained from feature tracking. Vertices are linked through time to form tracks. A vertex stores its position in the original video, which is used to reference textures that store the pixel values and alpha. The current position of a vertex can be modified for editing purposes (e.g. to perform motion magnification [56]), and we store it in a separate field. Vertices also have a continuous depth value which can be edited using a number of tools, described in Section 4.3.2. Depth information is encoded with respect to a camera matrix that is specified per frame.

Faces We use a Delaunay triangulation over each frame to define the faces of the video mesh. Each triangle is texture-mapped using the pixel values from the original video, with texture coordinates defined by the original position of its vertices. The textures can be edited to enable various video painting and compositing effects. Each face references a list of texture tiles to enable the treatment of multiple layers.

The triangulations of consecutive frames are mostly independent. While it is desirable that the topology be as similar as possible between frames to generate a continuous motion field, this is not a strict requirement. We only require vertices, not faces, to be linked in time. The user can force edges to appear in the triangulation by adding *line constraints*. For instance, we can ensure that a building is accurately represented by the video mesh by aligning the triangulation with its contours.

Motion For illustration, consider a simple manipulation such as motion magnification [56]. One starts by tracking features over time. For this example, we assume that all tracks last the entire video sequence and that there is no occlusion. Each frame is then triangulated to create faces. The velocity of a vertex can be accessed by querying its successor and predecessor and taking the difference. A simple scaling of displacement [56] yields the new position of each vertex. The final image for a given frame is obtained by rendering each triangle with the vertices at the new location but with texture coordinates at the original position, indexing the original frames. This is essentially equivalent to triangulation-based morphing [34].

4.2.2 Occlusion

Real-world scenes have occlusions, which are always the most challenging aspect of motion treatment. Furthermore, vertex tracks can appear or disappear over time because of, for instance, occlusion or loss of contrast. The video mesh handles these cases by introducing *virtual vertices* and duplicating triangles to store information for both foreground and background parts.

Consider first the case of vertices that appear or disappear over time. Since we rely on the predecessor and successor to extract motion information, we introduce *temporal* virtual vertices at both ends of a vertex track. Like normal vertices, they store a position, which is usually extrapolated from adjacent frames but can also be fine-tuned by the user.

Real scenes also contain spatial occlusion boundaries. In mesh-based interpolation approaches, a triangle that overlaps two scene objects with different motions yields artifacts when motion is interpolated. While these artifacts can be reduced by refining the triangulation to closely follow edges, e.g.,, [17], this solution can significantly increase geometric complexity and does not handle soft boundaries. Instead, we take an approach inspired by work in mesh-based physical simulation [58]. At occlusion boundaries, where a triangle partially overlaps both foreground and background layers, we *duplicate* the face into foreground and background copies, and add *spatial* virtual vertices to complete the topology. To resolve per-pixel coverage, we compute a local alpha matte to disambiguate the texture (see Figure 4-2). Similar to temporal virtual vertices, their spatial counterparts store position information that is extrapolated from their neighbors. We extrapolate a motion vector at these points and create temporal virtual vertices in the adjacent past and future frames to represent this motion. Topologically, the foreground and background copies of the video mesh are locally disconnected: information cannot directly propagate across the boundary.

When an occlusion boundary does not form a closed loop, it ends at a singularity called a *cusp*. The triangle at the cusp is duplicated like any other boundary triangle and the alpha handles fine-scale occlusion. We describe the topological construction of cuts and cusps in Section 4.3.1.

The notion of occlusion in the video mesh is purely local and enables self-occlusion within a layer, just like how a 3D polygonal mesh can exhibit self-occlusion. Occlusion boundaries do not need to form closed contours. We cut the mesh and duplicate triangles locally. Depth ordering is specified through the per-vertex z information.

4.2.3 Tile-based texture storage

At occlusion boundaries, the video mesh is composed of several overlapping triangles and a position in the image plane can be assigned several color and depth values, typically one for the foreground and one for the background. While simple solutions such as the

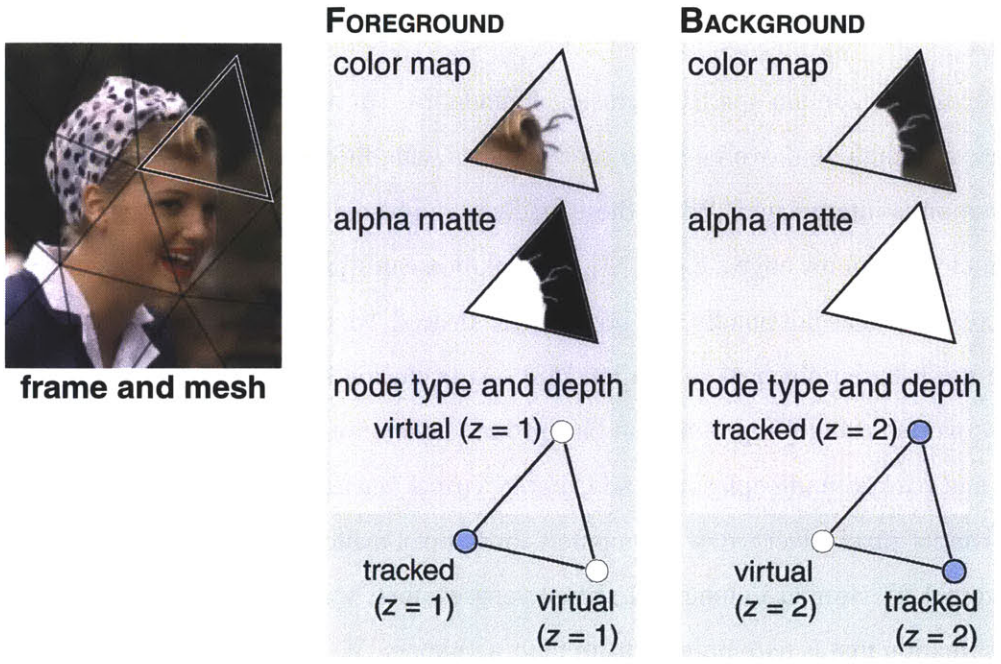


Figure 4-2: *Occlusion boundaries are handled by duplicating faces. Each boundary triangle stores a matte and color map. Duplicated vertices are either* tracked, *i.e.,, they follow scene points, or* virtual *if their position is inferred from their neighbors.*

replication of the entire frame are possible, we present a tile-based approach that strikes a balance between storage overhead and flexibility.

Replicating the entire video frame for each layer would be wasteful since few faces are duplicated and in practice, we would run out of memory for all but the shortest video sequences. Another possibility would be generic mesh parameterization [40], but the generated atlas would likely introduce distortions since these methods have no knowledge of the characteristics of the video mesh, such as its rectangular domain and preferred viewpoint.

Tiled texture We describe a tile-based storage scheme which trades off memory for rendering efficiency—in particular, it does not require any mesh reparameterization. The image plane is divided into large blocks (e.g.,, 128×128). Each block contains a list of texture tiles that form a stack. Each face is assigned its natural texture coordinates; that is, with (u, v) coordinates equal to the (x, y) image position in the input video. If there is already data stored at this location (for instance, when adding a foreground triangle and its back-

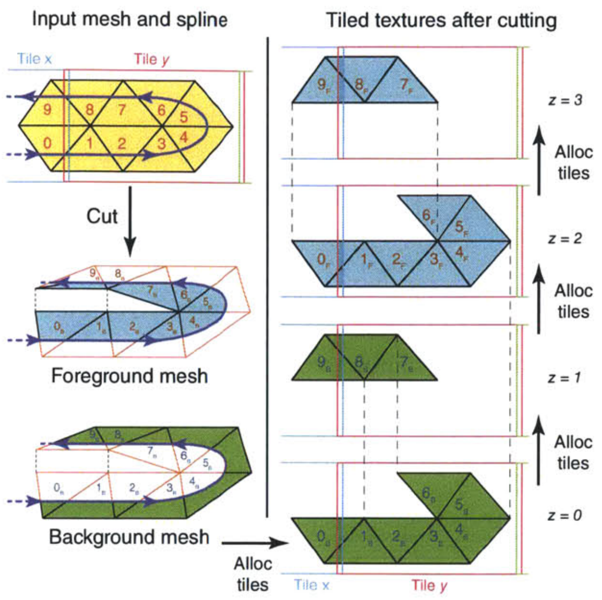


Figure 4-3: ***Left:*** *An initially flat video mesh falls into two overlapping texture tiles x and y. Cutting the mesh with the spline results in a foreground and background mesh. Note how vertices in the interior are disconnected.* ***Right:*** *Texture allocation after cutting. Our implementation allocates space for the background mesh first, in the direction of the spline. Triangles 0_B and 1_B overlap the tile boundary and are assigned to both tiles. After allocating $0_B \ldots 6_B$, because the interior edges are disconnected, 7_B through 9_B are assigned to a new tile. We then allocate 0_F, which also needs a new tile since they do not share any edges with the triangles at levels $z = 0$ and $z = 1$.*

ground copy already occupies the space in the tile), we move up in the stack until we find a tile with free space. If a face spans multiple blocks, we push onto each stack using the same strategy: a new tile is created within a stack if there is no space in the existing tiles.

To guarantee correct texture filtering, each face is allocated a one-pixel-wide margin so that bilinear filtering can be used. If a face is stored next to its neighbor, then this margin is already present. Boundary pixels are only necessary when two adjacent faces are stored in different tiles. Finally, tile borders overlap by two-pixels in screen space to ensure correct bilinear filtering for faces that span multiple tiles.

The advantages of a tile-based approach is that overlapping faces require only a new tile instead of duplicating the entire frame. Similarly, local modifications of the video mesh such as adding a new boundary impact only a few tiles, not the whole texture. Finally, the use of canonical coordinates also enable data to be stored without distortion relative to the input video.

4.2.4 Rendering

The video mesh is, at its core, a collection of texture-mapped triangles and is easy to render using modern graphics hardware. We handle transparency by rendering the scene back-to-front using alpha blending, which is sufficient when faces do not intersect. We handle faces that span several tiles with a dedicated shader that renders them once per tile, clipping the face at the tile boundary. To achieve interactive rendering performance, tiles are cached in texture memory as large atlases (e.g.,, 4096×4096), with tiles stored as subregions. Caching also enables efficient rendering when we access data across multiple frames, such as when we perform space-time copy-paste operations. Finally, when the user is idle, we prefetch nearby frames in the background into the cache to enable playback after seeking to a random frame.

4.3 Video mesh operations

The video mesh supports a number of creation and editing operators. This section presents the operations common to most applications, while we defer application-specific algorithms to Section 4.4.

4.3.1 Cutting the mesh along occlusions

The video mesh data structure supports a rich model of occlusion as well as interactive creation and manipulation. For this, we need the ability to cut the mesh along user-provided occlusion boundaries. We use splines to specify occlusions [7], and once cut, the boundary can be refined using image matting [21,49,95]. In this section, we focus on the topological cutting operation of a video mesh given a set of splines. A boundary spline has the following properties:

1. It specifies an occlusion boundary and intersects another spline only at T-junctions.
2. It is **directed**, which *locally* separates the image plane into foreground and background.
3. It can be **open** or **closed**. A closed spline forms a loop that defines an object detached from its background. An open spline indicates that two layers merge at an endpoint called a *cusp*.

Ordering constraints In order to create a video mesh whose topology reflects the occlusion relations in the scene, the initial flat mesh is cut front-to-back. We organize the boundary splines into a directed graph where nodes correspond to splines and a directed edge between two splines indicates that one is in front of another. We need this ordering only at T-junctions, where a spline a ends in contact with another spline b. If a terminates on the foreground side of b, we add an edge $a \rightarrow b$, otherwise we add $b \rightarrow a$. Since the splines represent the occlusions in the underlying scene geometry, the graph is guaranteed to be acyclic. Hence, a topological sort on the graph produces a front-to-back partial ordering from which we can create layers in order of increasing depth. For each spline, we walk from one end to the other and cut each crossed face according to how it is traversed by the spline. If a spline forms a T-junction with itself, we start with the open end; and if the two ends form T-junctions, we start at the middle of the spline (Fig. 4-7). This ensures that self T-junctions are processed top-down.

Four configurations To cut a mesh along splines, we distinguish the four possible configurations:

1. If a face is fully cut by a spline, that is, the spline does not end inside, we duplicate the face into foreground and background copies. Foreground vertices on the background side of the spline are declared virtual. We attach the foreground face to the uncut and previously duplicated faces on the foreground side. We do the same for the

background copy (Fig. 4-4).

2. If a face contains a T-junction, we first cut the mesh using the spline in front as in case 1. Then we process the back spline in the same way, but ensure that at the T-junction, we duplicate the background copy (Fig. 4-5). Since T-junctions are formed by an object in front of an occlusion boundary, the back spline is always on the background side and this strategy ensures that the topology is compatible with the underlying scene.

3. If a face is cut by a cusp (i.e.,, by a spline ending inside it), we cut the face like in case 1. However, the vertex opposite the cut edge is not duplicated; instead, it is shared between the two copies (Fig. 4-6).

4. In all the other cases where the face is cut by two splines that do not form a T-junction or by more than two splines, we subdivide the face until we reach one of the three cases above.

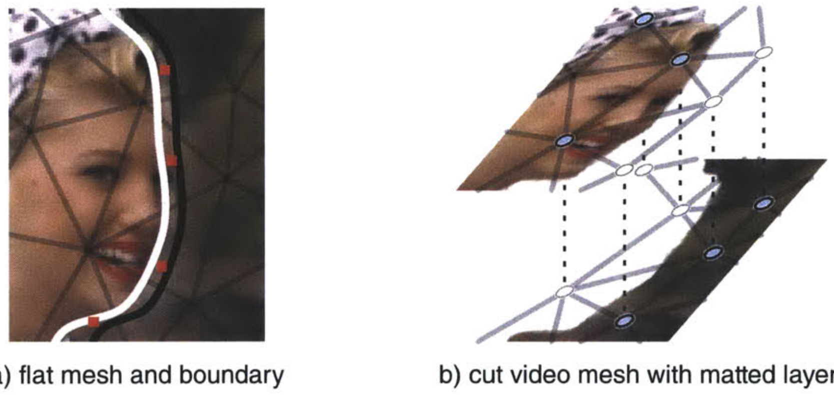

a) flat mesh and boundary
b) cut video mesh with matted layers

Figure 4-4: *Cutting the mesh with a boundary spline. The cut faces are duplicated. The foreground copies are attached to the adjacent foreground copies and uncut faces. A similar rule applies to the background copies. Blue vertices are* real *(tracked), white vertices are* virtual.

Motion estimation Cutting the mesh generates spatial virtual vertices without successors or predecessors in time. We estimate their motion by diffusion from their neighbors. For each triangle with two tracked vertices and a virtual vertex, we compute the translation, rotation, and scaling of the edges with the two tracked vertices. We apply the same

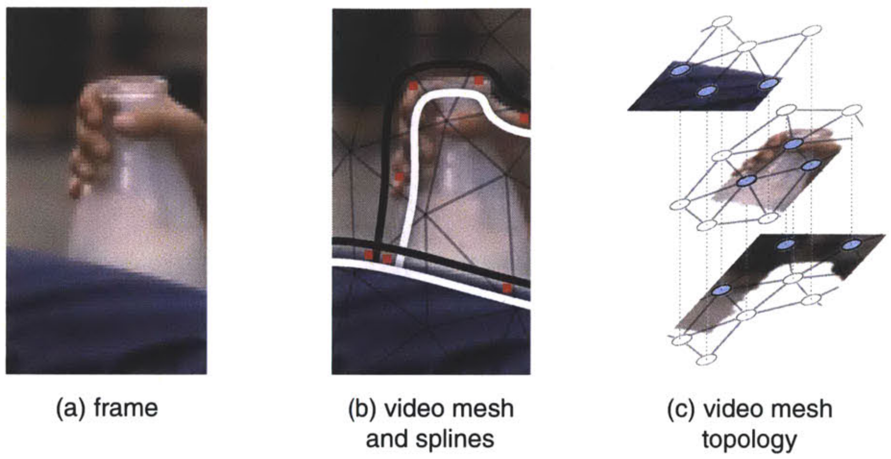


Figure 4-5: *Cutting the mesh with two splines forming a T-junction. We first cut according to the non-ending spline, then according to the ending spline.*

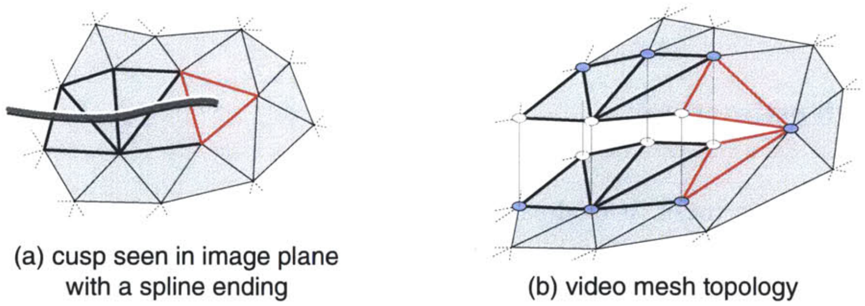


Figure 4-6: *Cutting the mesh with a cusp. This case is similar to the normal cut (Fig. 4-4) except that the vertex opposite to the cut edge is shared between the two copies.*

transformation to the virtual vertex to obtain its motion estimate. If the motion of a virtual vertex can be evaluated from several faces, we find a least-squares approximation to its motion vector. We use this motion vector to create temporal virtual vertices in the previous and next frame. This process is iterated as a breadth-first search until the motion of all virtual vertices are computed.

Boundary propagation Once we have motion estimates for all spatial virtual vertices in a frame, we can use the video mesh to advect data. In particular, we can advect the control points of the boundary spline to the next (or previous) frame. Hence, once the user specifies the occlusion boundaries at a single keyframe, as long as the topology of occlusion boundaries does not change, we have enough information to build a video mesh over all

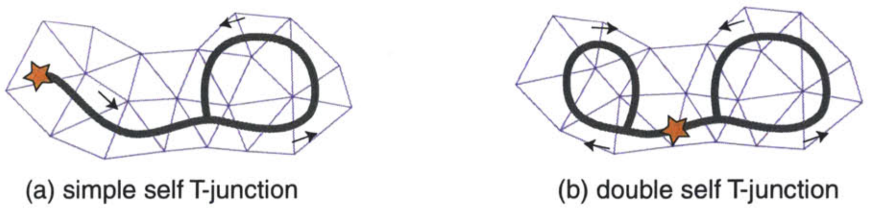


Figure 4-7: *If a spline forms a T-junction with itself (a), we start from the open end (shown with a star) and process the faces in order toward the T-junction. If a spline forms two T-junctions with itself (b), we start in between the two T-junctions and process the faces bidirectionally.*

frames. We detect topology changes when two splines cross and ask the user to adjust the splines accordingly. In practice, the user needs to edit 5 to 10% of the frames, which is comparable to the technique of Agarwala et al. [7].

4.3.2 Depth estimation

After cutting the video mesh, it is already possible to infer a pseudo-depth value based on the foreground/background labeling of the splines. However, for a number of video processing tasks, continuous depth values enable more sophisticated effects. As a proof of concept, we provide simple depth-modeling tools that work well for two common scenarios. For more challenging scenes, the video mesh can support the dense depth maps generated from more advanced techniques such as multi-view stereo.

Static camera: image-based modeling For scenes that feature a static camera with moving foreground objects, we provide tools inspired from the still photograph case [42, 59] to model a coarse geometric model of the background. The *ground tool* lets the user define the ground plane from the horizon line. The *vertical object tool* enables the creation of vertical walls and standing characters by indicating their contact point with the ground. The *focal length tool* retrieves the camera field of view from two parallel or orthogonal lines on the ground. This proxy geometry is sufficient to handle complex architectural scenes as demonstrated in the supplemental video.

Moving camera: user-assisted structure-from-motion For scenes with a moving camera, we build on structure-from-motion [39] to simultaneously recover a camera path as well as the 3D position of scene points. In general, there might be several objects moving independently. The user can indicate rigid objects by selecting regions delineated by the splines. We recover their depth and motion independently using structure-from-motion and register them in a global coordinate system by aligning to the camera path which does not change. We let the user correct misalignments by specifying constraints, typically by pinning a vertex to a given position.

Even with a coarse video mesh, these tools allow a user to create a model that is reasonably close to the true 3D geometry. In addition, after recovering the camera path, adding vertices is easy by clicking on the same point in only 2 frames. The structure-from-motion solver recovers its 3D position by minimizing reprojection error over all the cameras.

4.3.3 Inpainting

We triangulate the geometry and inpaint the texture in hidden parts of the scene in order to render 3D effects such as changing the viewpoint without revealing holes.

Geometry For closed holes that typically occur when an object occludes the background, we list the mesh edges at the border of the hole and fill in the mesh using constrained Delaunay triangulation with the border edges as constraints.

When a boundary is occluded, which happens when an object partially occludes another, we observe the splines delineating the object. An occluded border generates two T-junctions which we detect. We add an edge between the corresponding triangles and use the same strategy as above with Delaunay triangulation.

Texture For large holes that are typical of missing static backgrounds, we use *background collection* [86]. After infilling the geometry of the hole, we use the motion defined by the mesh to search forward and backward in the video for unoccluded pixels. Background collection is effective when there is moderate camera or object motion and can significantly reduce the number of missing pixels. We fill the remaining pixels by isotropically diffusing data from the edge of the hole.

When the missing region is textured and temporally stable, such as on the shirt of an actor in Soda sequence of our video, we modify Efros and Leung texture synthesis [26] to search only in the same connected component as the hole within the same frame. This strategy ensures that only semantically similar patches are copied and works well for smoothly varying dynamic objects. Finally, for architectural scenes where textures are more regular and boundaries are straight lines (Figure 4-8), we proceed as Khan et al. [45] and mirror the neighboring data to fill in the missing regions. Although these tools are simple, they achieve satisfying results in our examples since the regions where they are applied are not the main focus of the scene. If more accuracy is needed, one can use dedicated mesh repair [51] and inpainting [12, 14] algorithms.

4.4 Results

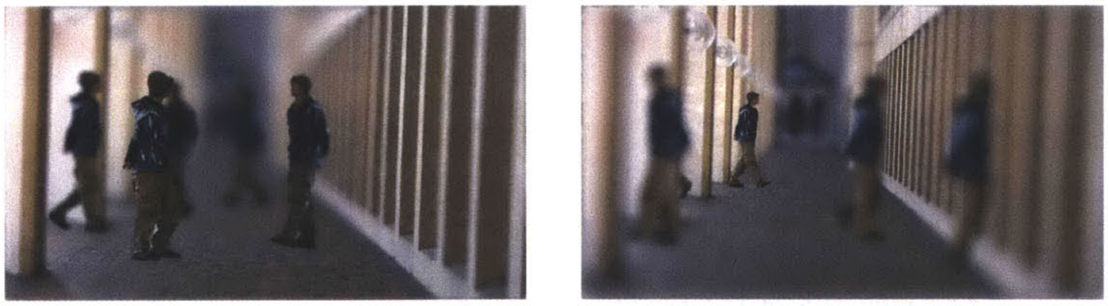

(a) focus on foreground (b) focus on background

Figure 4-8: *Compositing and depth of field manipulation. We replicated the character from the original video, composited multiple copies with perspective, and added defocus blur.*

We illustrate the use of the video mesh on a few practical applications. These examples exploit the video mesh's accurate scene topology and associated depth information to create a variety of 3D effects. The results are available in the companion video.

Depth of field manipulation We can apply effects that depend on depth such as enhancing a camera's depth of field. To approximate a large aperture camera with a shallow depth of field, we construct a video mesh with 3D information and render it from different viewpoints uniformly sampled over a synthetic aperture, keeping a single plane in focus. Since the new viewpoints may reveal holes, we use our inpainting operator to fill both the geometry and texture. For manipulating defocus blur, inpainting does not need to be accurate. This approach supports an arbitrary location for the focal plane and an arbitrary aperture. In the Soda and Colonnade sequences, we demonstrate the *rack focus* effect which is commonly used in movies: the focus plane sweeps the scene to draw the viewer's attention to subjects at various distances (Fig. 4-8). This effect can be previewed in real time by sampling 128 points over the aperture. A high-quality version with 1024 samples renders at about 2 Hz.

Object insertion and manipulation The video mesh supports an intuitive copy-and-paste operation for object insertion and manipulation. The user delineates a target object with splines, which is cut out to form its own connected component. The object can then be replicated or moved anywhere in space and time by copying the corresponding faces and applying a transformation. The depth structure of the video mesh enables occlusions between the newly added objects and the existing scene while per-pixel transparency makes it possible to render antialiased edges. This is shown in the Colonnade sequence where the copied characters are occluded by the pillars and each other. The user can also specify that the new object should be in contact with the scene geometry. In this case, the depth of the object is automatically provided according to the location in the image. We further develop this idea by exploiting the motion description provided by the video mesh to ensure that the copied objects consistently move as the camera viewpoint changes. This feature is shown in the Copier sequence of the companion video. When we duplicate an animated object several times, we offset the copies in time to prevent unrealistically synchronized movements.

We also use transparency to render volumetric effects. In the Soda sequence, we insert a simulation of volumetric smoke. To approximate the proper attenuation and occlusion that depends on the geometry, we render 10 offset layers of 2D semi-transparent animated smoke.

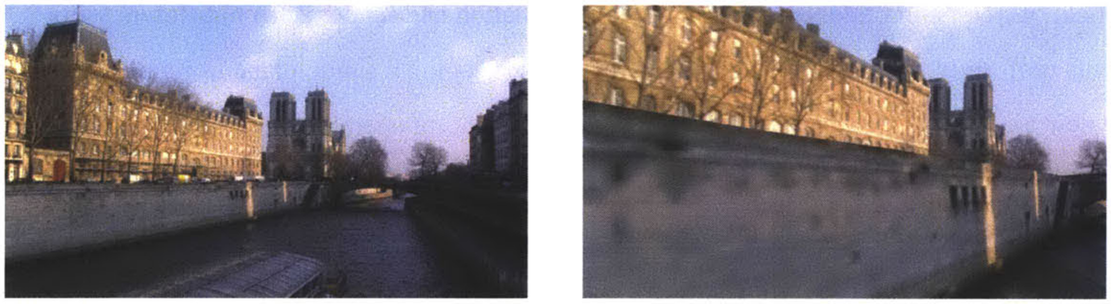


Figure 4-9: ***Left:*** *original frame.* ***Right:*** *camera moved forward and left toward the riverbank.*

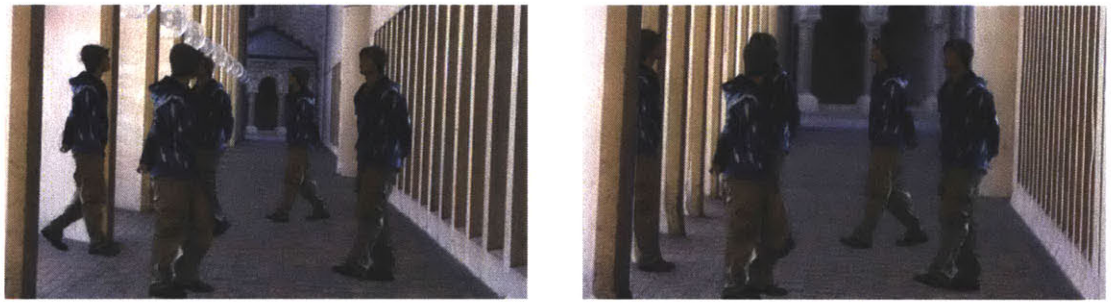

(a) wide field of view, camera close to the subject
(b) narrow field of view, camera far from the subject

Figure 4-10: *Vertigo effect enabled by the 3D information in the video mesh. We zoom in and at the same time pull the camera back.*

Change of 3D viewpoint With our modeling tools (Sec. 4.3.2) we can generate proxy geometry that enables 3D viewpoint changes. We demonstrate this effect in the companion video and in Figure 4-9. In the Colonnade and Notre-Dame sequences, we can fly the camera through the scene even though the input viewpoint was static. In the Copier sequence, we apply a large modification to the camera path to get a better look at the copier glass. Compared to existing techniques such as Video Trace [92], the video mesh can handle moving scenes as shown with the copier. The scene geometry also allows for change of focal length, which in combination with change of position, enables the *vertigo effect*, a.k.a. *dolly zoom*, in which the focal length increases while the camera moves backward so that the object of interest keeps a constant size (Fig. 4-10).

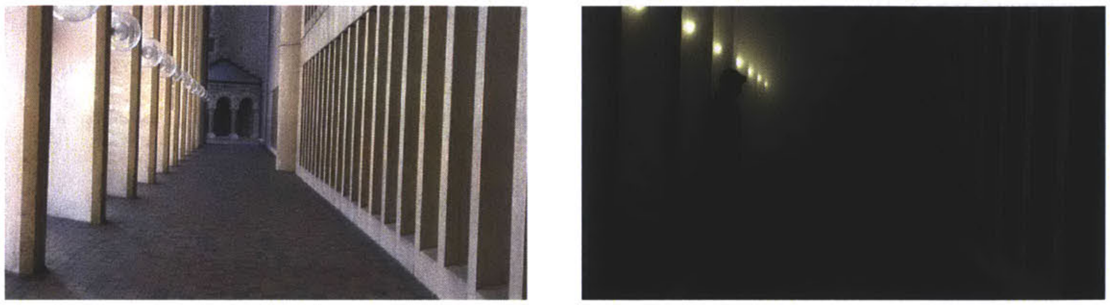


Figure 4-11: *Relighting a daytime scene to depict night.* ***Left:*** *original video frame.* ***Right:*** *Scene relit with simulated lights, fog, and shadows cast by composited characters.*

Relighting and participating media We use the 3D geometry encoded in the video mesh for relighting. In the companion video, we transform the daylight Colonnade sequence into a night scene (Fig. 4-11). We use the original pixel value as the diffuse material color, and let the user click to position light sources and add volumetric fog. We render the scene using raytracing to simulate shadows and participating media [66].

Stereo 3D With a complete video mesh, we can output stereo 3D by rendering the video mesh twice from different viewpoints. We rendered a subset of the results described above in red/cyan anaglyphic stereo; the red channel contains the red channel from the "left eye" image, and the green and blue channels contain the green and blue channels from the "right eye" image. The results were rendered with both cameras parallel to the original view direction, displaced by half the average human interocular distance. Our technique supports any stereo technology and is not limited to anaglyphic stereoscopy. We chose anaglyphic red/cyan since anyone can view the results with a low-cost pair of glasses.

Performance and user effort We implemented our prototype in DirectX 10 and ran our experiments on an Intel Core i7 920 at 2.66 GHz with 8 MB of cache and a NVIDIA

GeForce GTX 280 with 1 GB of video memory. At a resolution of 640×360, total memory usage varies between 1 GB to 4 GB depending on the sequence, which is typical for video editing applications. To handle long video sequences, we use a multi-level virtual memory hierarchy over the GPU, main memory, and disk with background prefetching to seamlessly access and edit the data. With the exception of offline raytracing for the fog simulation, and high-quality depth of field effects that require rendering each frame 1024 times, all editing and rendering operations are interactive ($>$ 15 Hz).

4.5 Discussion

We discuss the design of our video editing system; in particular, the motivation behind designing the video mesh data structure, and why we chose particular components. We then describe the user interactions needed to create each of the examples.

4.5.1 System design

The primary motivation for the design of the video mesh is the desire for a flexible, user-assisted system for manipulating video in 3D. Our system is designed to let users work on a single frame at a time (c.f., Video Cubes [94]) and have their edits automatically propagate to nearby frames while maintaining control over the process. Keeping frames independent also dramatically simplifies our implementation of undo, which is critical in an interactive editing setting. At places, our system relies on a number of existing algorithms. Our choice of these off-the-shelf components is driven primarily by the ability to provide a simple user interface for controlling their output.

For motion analysis, we prefer point tracking to optical flow because dense motion fields are difficult to interpret and edit. In contrast, point tracking gives users only a handful of controls to manipulate. Our current implementation is based on RealVIZ MatchMover [73], which provides an automatic point tracking algorithm which is also user-controllable: users can place hard constraints and the algorithm optimizes for a smooth path satisfying the constraints. For depth estimation, our system uses structure-from-motion [1] to recover sparse depth estimates instead of stereo disparity maps, which, like optical flow, can be difficult to edit. In our system, depth inference is weakest on scenes with independently moving articulated bodies (for example, the Copier sequence). We let the user correct any misregistrations by constraining the shared vertices and camera path. Finally, for decomposing textures, we find that the default settings for published matting algorithms [49, 95] have difficulties with the large number of hard edges found in real scenes (for instance, the Colonnade and Copier examples). We address this issue by providing a spatially varying hardness weight along each spline, going from hard binary masking to soft matting.

4.5.2 User interaction

This section discusses the user interactions used to create each of the examples in the companion video. The workflow typically begins with first estimating motion and topology with the tracking and spline tools. The next step is computing the depth using one of our depth recovery operators, which yields a rough but complete video mesh upon which the various effects can be directly applied. We stress that the editing process is iterative: the user is free at any point to go back and correct any mistakes. For instance, if we discover, after drawing splines and computing depth, that a vertex has drifted, the user can simply drag the vertex to the correct location. Our system automatically updates the mesh in the background. The ratio of time spent between the various tools is scene dependent. We found that the majority of user time was spent tracking points and manipulating splines. Once the basic topology is complete, estimating depth and creating the various effects took only a small fraction of the time.

The Soda sequence in the companion video demonstrates the basic interactions used to create a video mesh. An expert user first spent roughly 5 minutes tracking points. With the actor's highly textured shirt, the process was mostly automatic and the majority of the user's time was spent correcting tracks near occlusion boundaries and adding a few extra vertices to ensure a uniform density of points. In the next step, the user spent approximately 10 minutes labeling the occlusion boundaries with splines. Although temporally propagating splines is mostly automatic in this sequence with smooth motion, the video does contain a number of changes in topology which necessitated user intervention. As shown in the video, in the next step, the user pulls on a few vertices in the mesh to turn the actor into a paper cutout. Finally, to complete the video mesh, the user spent another 5 minutes adjusting various matting parameters to ensure a consistent boundary in the video. Once the video mesh is complete, we could change the camera's position, aperture, and focus plane as shown in the video by mapping them to various mouse gestures. To create the smoke effect, we imported 10 copies of a precomputed smoke simulation video and scaled and positioned the layers in 3D.

The Colonnade sequence turned out to be our most challenging example. Even though the actor's motion was simple, it proved problematic for the point tracker because his pants and face had virtually no texture. Moreover, due to the change in mesh topology each time the actor's legs crossed during his walk cycle, the automatic spline propagation had to be restarted approximately every 20 frames. Overall, the user spent about 20 minutes tracking 42 points on the actor and 25 minutes adjusting splines. Like the previous example, once the mesh is complete, creating the effects was straightforward and demonstrated in real time in the companion video. Selecting the actor in space-time is trivial because he forms a single connected component in the video mesh. In our interface, once a component is selected for copying, the user can interactively drag and drop to choose the destination, optionally applying a temporal offset to decorrelate the movement, or a flip to let the video play backward. We also found that snapping the lowest pixel of object to the triangle under the cursor to be a useful tool. Similar to copy/paste, our interface lets the user point and click to place a light source for relighting. The companion video demonstrates space-time copy-paste, interactive post-exposure camera control, and relighting.

The Copier sequence was our most fully automated sequence. We first automatically tracked points in the sequence, manually adding only a handful of points. Next, the user lassoed the points on the lid of the copier on one frame, exporting the set of tracks to Boujou for structure-from-motion estimation. The process was repeated for the chassis of the copier. After recovering two sets of 3D points and camera paths, we register them together by enforcing the camera paths to be the same. Tracking and 3D estimation only took about 5 minutes. Since the topology is relatively simple, the user only spent 5 minutes cutting the copier out of the background. Most of the time was spent converting spline control points from "smooth" to "corner" to respect the sharp geometry. Finally, less than one minute was spent enforcing lines in the mesh which correspond to straight segments in the scene: the user clicks on two vertices to create a segment that lasts through the entire sequence. Like the Colonnade sequence, once we had a complete video mesh, we composited multiple copies of the actor onto the glass of the copier by transforming a component of another video mesh in space and time. In this case, we keyframed the character's transparency so he smoothly comes into view. The companion video demonstrates interactive viewpoint control while the video plays.

In contrast to the Colonnade sequence where the camera was also stationary, our system worked well on the Notre Dame sequence despite its complex geometry. Automatic tracking was accurate on the boat since it had a smooth trajectory and a moderately textured surface. Modeling the geometry was straightforward: splines separated the scene components over which the facades were applied. The only difficulty we encountered was due to the relatively large z-range compared to image resolution, where a small error in the placement of a modeling facade may result in a large distortion in depth. Overall, the video mesh took the authors roughly 20 minutes to create: about 10 minutes for tracking and rotoscoping the boat and 10 minutes for depth modeling. The companion video shows the quality of the recovered depth map and shows a large viewpoint change by moving the camera onto the boat as it sails down river.

4.5.3 Limitations and future work

Although our approach gives users a flexible way of editing a large class of videos, it is not without limitations. The primary limitation stems from the fact that the video mesh is a coarse model of the scene: high-frequency motion, complex geometry, and thin features would be difficult to accurately represent without excessive tessellation. For instance, the video mesh has trouble representing a field of grass blowing in the wind, although we believe that other techniques would also have difficulties. For the same reason, the video mesh cannot represent finely detailed geometry such as a bas-relief on a wall. In this case, the bas-relief would appear as a texture on a smooth surface, which may be sufficient in a number of cases, but not if the bas-relief is the main object of interest. A natural extension would be to augment the video mesh with a displacement map to handle high-frequency geometry. Other possibilities for handling complex geometry are to use an alternative representation, such as billboards, imposters, or consider a unified representation of geometry and matting [88]. To edit these representations, we would like to investigate more advanced interactive modeling techniques, in the spirit of those used to model architecture from photographs [23, 81]. Integrating these approaches into our system is a promising direction for future research.

Summary We have presented the video mesh, a data structure to represent video sequences and whose creation is assisted by the user. The required effort to build a video mesh is comparable to rotoscoping but the benefits are higher since the video mesh offers a rich model of occlusion and enables complex effects such as depth-aware compositing and relighting. Furthermore, the video mesh naturally exploits graphics hardware capabilities to provide interactive feedback to users. We believe that video meshes can be broadly used as a data structure for video editing.

4.6 Hindsights

Our prototype demonstrates that with the right data structure, a large number of techniques from computer graphics and computer vision can be combined to effectively manipulate video. However, our implementation lacks a number of features necessary for a complete video editing system. A production video editing system will need to be holistically designed with substantial amounts of user feedback to ensure usability. In this section, we discuss some of the lessons learned while developing the video mesh.

Undo The ability to undo an arbitrary operation is critical in a user-interactive application. Our prototype implements undo in an ad hoc fashion where not every operation is reversible. In particular, vertex tracking and mesh cutting operations can be undone, but matting cannot—it can only be reverted to the original video frame. The underlying implementation of the video mesh is a table with indices and pointers into other data structures holding the actual motion and texture information. Edits on the data structure involve mutating pointers and keeping them consistent. In hindsight, a much simpler strategy would have been to use a persistent data structure with nondestructive copies for each edit, in the spirit of functional programming. It would not only reduce the amount of bookkeeping involved, but also dramatically simplify undo (although at the cost of some memory overhead).

Visualization Our prototype features a traditional "3D trackball" interface for manipulating the 2.5D video mesh. It features only a rudimentary visualization of depth maps and lacks an effective motion visualization tool. Intuitive visualizations can provide valuable feedback to artists using the system. For instance, a motion viewer like the one used by Liu et al. [55] in their user-assisted motion annotation system, displaying the piecewise-linear interpolated motion, can help the user both adjust vertex tracking and place boundary splines.

The visualization of the video mesh topology is another interesting direction to explore. Because the mesh has the structure of overlapping triangles, we discovered during the debugging of our cutting algorithm that it can be difficult to tell if the topology is correct. Without assigning depth, the triangles overlap exactly and can only be represented by a sorted list (because we can order them from top to bottom). We cannot directly assign colors to connected components because the layers are only locally separated. One promising idea with which we experimented was to assign temporary integer depth values based on the front-to-back ordering of the cuts.

Finally, a visualization mode that isolates various connected components in the space-time mesh would be very useful for editing since they correspond to semantically meaningful objects. The artist can use this mode to debug cuts, inspect and paint objects individually, or visualize the trajectory of the object in a video cube without the clutter of the background.

Motion effects The video mesh interpolated a dense motion field from sparse point tracking. However, our examples did not fully take advantage of the information: we used the motion only for automatically advecting boundary splines. It would be useful to see what kind of higher order motion "filters" may be applied on top of the basic interpolation, and implement additional effects that take advantage of motion such as slow motion retiming, temporal edit propagation, temporally coherent video filtering, and character animation.

Transparency rendering Our system used traditional rasterization to interactively render the video mesh. However, because the video mesh is a collection of textured triangles with alpha, these triangles tend to intersect in 3D after most nontrivial editing operations involving depth assignment or object insertion. Examples include the Soda sequence when the actor's hand intersects the added volumetric smoke and the Colonnade sequence where the inserted actor's feet are close to the ground. Our implementation used a simple centroid-based depth sort and rendered the triangles back to front, which resulted in compositing artifacts which appear as translucent triangle shapes. In hindsight, we could have eliminated all these artifacts by using a real-time raytracing engine such as OptiX [62] with order-independent transparency.

Video matting Our implementation neglected to enforce any temporal smoothness into the matting component of our system. We thought that per-frame natural image matting would perform well with sufficiently tight trimaps. Unfortunately, in our experience, the matting algorithms we used [48, 95] still suffered from a number of failures when backgrounds are textured or when the sequence has more than a few dozen frames. Leveraging new video matting and segmentation algorithms such as Video SnapCut [10] seems to be a natural solution, although integrating them into a consistent user interface can pose a significant challenge.

In retrospect, these features would probably have significantly reduced the amount of manual labor involved in creating our example sequences, and eliminated some of the most noticeable artifacts.

Chapter 5

Conclusion

We conclude this thesis with a summary of results and a discussion of open problems in video processing.

5.1 Edge-aware image processing

The first part of the thesis introduced the theory of digital image filtering in the context of classical linear shift-invariant signal processing. We demonstrated the utility of nonlinear image filtering through a series of examples leading up to piecewise-smooth operators and the concept of edge-aware image processing.

Our contribution to edge-aware image processing is the bilateral grid data structure which implicitly includes a simple model of edge awareness. The bilateral grid augments the standard two-dimensional image with an additional intensity dimension that accounts for edges. Near strong edges in an image, pixels that are close together in image space are far apart in the grid: the Euclidean distance in the bilateral grid is edge-aware.

In Chapter 3, we demonstrated a number of edge-aware image filtering and editing operations that exploit the properties of the bilateral grid including bilateral filtering, scribble interpolation, painting, interactive tone mapping, and local histogram equalization. Although these algorithms can certainly be performed directly in image space, the bilateral grid provides the advantages of simplicity and speed. Because the bilateral grid encodes intensity differences as part of its Euclidean distance metric, many edge-aware algorithms can be expressed as relatively simple operations on the grid. Moreover, we demonstrated that these simpler formulations are often parallel, granting them massive speedups on modern parallel hardware.

5.2 Image-based three-dimensional video editing

The other major contribution of this thesis is the video mesh data structure for user-driven video editing. We followed the same philosophy in designing the video mesh as we did for the bilateral grid: take a number of previously disparate algorithms on raw pixels and express them as simple operations on a unified data structure. The video mesh represents the available scene content as "pieces of paper" whose connectivity reflects that of the underlying scene. Compared to other video representations, the main advantage of the video mesh is its ability to remain relatively sparse, making it user-editable, while being able to produce dense data through interpolation. Depending on the desired level of detail, users are able to augment the data structure in regions which are difficult to interpolate with dense information such as per-pixel alpha, motion, and depth. Early in the design process, a key decision was made to ensure that our data structure remained a manifold-connected triangle mesh. This property enabled us to not only use standard meshing algorithms to reason about the underlying geometry during cutting and inpainting, but also to use standard graphics hardware for interactive rendering.

We demonstrated the flexibility of the video mesh in handling a number of traditional and novel special effects as simple manipulations of the data structure. Similar to the relation between edge-aware image processing algorithms versus their implementation as operations on the bilateral grid, these special effects can be written as direct edits on the raw video pixels as opposed to manipulations of the video mesh. We showed that once a video mesh is created, a large variety of effects, such as the adjustment of camera parameters, object cut-and-paste, and relighting, can be achieved easily by performing interactive edits on the resulting data structure. It can be thought of as "paying up front" (through a potentially labor-intensive process) for a useful representation to gain additional flexibility later in the editing process.

5.3 Open Problems

The general problem of efficient video processing remains open. Indeed, being ultimately a creative activity, we do not believe video processing can truly be considered "solved".

The bilateral grid and its successors tackled the problem of edge-aware filtering. However, they all use a fairly simple model of what determines an edge (i.e., intensity differences). This simple model does not always conform to the *semantic* notion of an edge. The question of how different edge definitions affects edge-aware filters, and which definition is relevant to which situation, deserves further examination. Finally, edge-aware filters form only a small subset within the space of useful piecewise-smooth operators. Discovering common themes among other classes of filters and designing efficient data structures for their application should be a fruitful direction for further study.

While the video mesh provides the ability to effectively represent the information necessary for video editing, it does not do so automatically. In fact, as we noted in Section 4.5.2, constructing a video mesh can be quite labor intensive for scenes of even moderate complexity. Automated video processing is difficult due to a number of reasons. We identify the most significant challenges to be the massive amounts of data involved, the lack of truly robust inference algorithms whose output is needed for most nontrivial video editing tasks, and the remarkable ability of the human visual system to detect even the smallest artifacts. The combination of these factors is what makes video processing particularly difficult to automate.

Video contains an order of magnitude more data than images. The ability to produce any sophisticated special effects beyond basic cropping and filtering requires the inference of large amounts of temporally coherent information. For instance, to convert a standard video from 2D to 3D, we require temporally coherent depth to produce the appropriate parallax, transparency to separate objects, and geometry and texture in the revealed background. Although recent advances in multi-view stereo [101], video matting [10, 21], and image inpainting [12] have made significant progress in estimating all of these (and in fact we use many of them in the video mesh), the quality is still insufficient in a number of scenarios.

We identify temporal coherence as a key issue in improving the robustness of all of these algorithms and the most promising direction for future research. The classical approach to incorporating temporal coherence is to first compute scene motion using an optical flow algorithm, and applying the appropriate motion vectors to the filter or editing operation. However, like the other inference algorithms we highlighted, optical flow algorithms are fragile and suffer from a number of known problems [41]. We highlight optical flow as a crucial open problem: to handle video, future vision algorithms will need to incorporate motion estimation as a first-class component.

In our experience, we were quite disappointed by performance of matting and segmentation algorithms on real-world imagery. Although they perform well on standard data sets, they often fail on textured backgrounds and are unstable in the presence of noise. We also discovered that current matting algorithms are intended to extract foreground objects for composition over new backgrounds, and do not perform a full image *decomposition.* A true decomposition into a foreground (with alpha) and an opaque background will re-composite back into the original input. A robust matting solution would dramatically broaden the space of semantically meaningful video manipulations.

Another direction that deserves further investigation is exploiting additional sensor modalities. Perhaps the problem of simultaneously inferring motion and depth is fundamentally too difficult given the complexity of real-world scenes. In fact, for production-quality video editing, artists may want even more information than we can hope to gather from a traditional digital camera, such as precise lighting conditions, material properties, and sound sources. We encourage "cheating": if the data cannot be reliably inferred using computer vision, why not simply make it easier to acquire? Traditional film studios, to a degree, already do this. For example, compositing is used heavily in modern movie production. Although natural image matting is gaining some traction, blue-screen matting [82], or even front or rear projection [22] is simply cheaper and fails more gracefully.

We argue that the basic assumptions on the available data are already changing with readily available depth [57] and stereo cameras [33]. We envision novel devices in the near future which can provide reliable estimates on scene motion (perhaps from tracking high-speed photography, radar, or sonar), or materials (from structured lighting). The changing hardware landscape is ripe with opportunities to revisit old problems. It is our hope that with the right combination of hardware, and algorithms to exploit it, computational photography can cheat its way to robust video processing systems.

Bibliography

[1] 2d3. boujou 4.1, 2008. http://www.2d3.com/. 81

[2] Andrew Adams, Jongmin Baek, and Myers Abraham Davis. Fast high-dimensional filtering using the permutohedral lattice. *Computer Graphics Forum (Proceedings of Eurographics 2010)*, 29(2):753–762, 2010. 57

[3] Andrew Adams, Natasha Gelfand, Jennifer Dolson, and Marc Levoy. Gaussian kd-trees for fast high-dimensional filtering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2009)*, 28(3):21:1–21:12, 2009. 57

[4] Adobe Systems, Inc. After Effects CS4, 2008. 63

[5] Adobe Systems, Inc. After Photoshop CS4, 2008. 24

[6] Adobe Systems, Inc. After Lightroom 3, 2009. 24

[7] Aseem Agarwala, Aaron Hertzmann, David H. Salesin, and Steven M. Seitz. Keyframe-based tracking for rotoscoping and animation. *ACM Transactions on Graphics*, 23(3):584–591, 2004. 61, 63, 70, 74

[8] Volker Aurich and Jörg Weule. Non-linear gaussian filters performing edge preserving diffusion. In *Proceedings of the DAGM Symposium*, 1995. 14, 34

[9] Soonmin Bae, Sylvain Paris, and Frédo Durand. Two-scale tone management for photographic look. *ACM Transactions on Graphics*, 25(3):637 – 645, 2006. Proceedings of the ACM SIGGRAPH conference. 12, 14, 51

[10] Xue Bai, Jue Wang, David Simons, and Guillermo Sapiro. Video SnapCut: robust video object cutout using localized classifiers. *ACM Transactions on Graphics*, 28:70:1–70:11, July 2009. 86, 92

[11] Danny Barash. A fundamental relationship between bilateral filtering, adaptive smoothing and the nonlinear diffusion equation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(6):844, 2002. 34

[12] Connelly Barnes, Eli Shechtman, Adam Finkelstein, and Dan B Goldman. PatchMatch: A randomized correspondence algorithm for structural image editing. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 28(3), Aug 2009. 76, 92

[13] Eric P. Bennett and Leonard McMillan. Video enhancement using per-pixel virtual exposures. *ACM Transactions on Graphics*, 24(3):845 – 852, July 2005. Proceedings of the ACM SIGGRAPH conference. 12, 14, 30

[14] Marcelo Bertalmio, Guillermo Sapiro, Vincent Caselles, and Coloma Ballester. Image inpainting. In *Proceedings of the ACM SIGGRAPH conference*, pages 417–424. ACM, 2000. 29, 76

[15] James F. Blinn. Fun with premultiplied alpha. *IEEE Computer Graphics and Applications*, 16(5):86–89, 1996. 38

[16] Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. A non-local algorithm for image denoising. *International Journal of Computer Vision*, 76(2):123–139, 2008. 57

[17] Nathalie Cammas, Stéphane Pateux, and Luce Morin. Video coding using non-manifold mesh. In *Proceedings of the 13th European Signal Processing Conference*, Antalya, Turkey, September 2005. 63, 67

[18] Jiawen Chen, Sylvain Paris, and Frédo Durand. Real-time Edge-Aware Image Processing with the Bilateral Grid. *ACM Transactions on Graphics*, 26(3):103:1–103:9, July 2007. 33

[19] Jiawen Chen, Sylvain Paris, Jue Wang, Wojciech Matusik, Michael Cohen, and Frédo Durand. The video mesh: A data structure for image-based three-dimensional video editing. In *IEEE International Conference in Computational Photography*, 2011. 62

[20] K. Chiu, M. Herf, P. Shirley, S. Swamy, C. Wang, and K. Zimmerman. Spatially nonuniform scaling functions for high contrast images. In *Proceedings of the conference on Graphics Interface*, pages 245–253, May 1993. 13

[21] Yung-Yu Chuang, Aseem Agarwala, Brian Curless, David Salesin, and Richard Szeliski. Video matting of complex scenes. *ACM Transactions on Graphics*, 21(3):243–248, 2002. 61, 63, 70, 92

[22] Charles G. Clarke. *Professional Cinematography*. ASC Holding Corp., 1964. ISBN 093557820X. 93

[23] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs. In *Proceedings of the ACM SIGGRAPH conference*. ACM, August 1996. 84

[24] Doug DeCarlo and Anthony Santella. Stylization and abstraction of photographs. *ACM Transactions on Graphics*, 21(3), 2002. Proceedings of the ACM SIGGRAPH conference. 50

[25] Frédo Durand and Julie Dorsey. Fast bilateral filtering for the display of high-dynamic-range images. *ACM Transactions on Graphics*, 21(3), 2002. Proceedings of the ACM SIGGRAPH conference. 12, 13, 14, 34, 52, 56

[26] Alexei A. Efros and Thomas K. Leung. Texture synthesis by non-parametric sampling. In *IEEE International Conference on Computer Vision*, pages 1033–1038, Corfu, Greece, September 1999. 76

[27] Elmar Eisemann and Frédo Durand. Flash photography enhancement via intrinsic relighting. *ACM Transactions on Graphics*, 23(3), July 2004. Proceedings of the ACM SIGGRAPH conference. 12, 14, 44, 58

[28] Zeev Farbman, Raanan Fattal, and Dani Lischinski. Diffusion maps for edge-aware image editing. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2010)*, 29(6):145:1–145:10, December 2010. 58

[29] Zeev Farbman, Raanan Fattal, Dani Lischinski, and Richard Szeliski. Edge-preserving decompositions for multi-scale tone and detail manipulation. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2008)*, 27(3):67:1–67:10, August 2008. 57, 58

[30] Raanan Fattal. Edge-avoiding wavelets and their applications. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2009)*, 28(3):22:1–22:10, July 2008. 58

[31] Raanan Fattal, Maneesh Agrawala, and Szymon Rusinkiewicz. Multiscale shape and detail enhancement from multi-light image collections. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007)*, 26(3):51:1–51:9, July 2007. 12

[32] Michael Felsberg, Per-Erik Forssén, and Hanno Scharr. Channel smoothing: Efficient robust smoothing of low-level signal features. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(2):209–222, February 2006. 35

[33] Fujifilm. FinePix REAL 3D W1, 2009. 93

[34] Jonas Gomes, Lucia Darsa, Bruno Costa, and Luiz Velho. *Warping and morphing of graphical objects*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998. 65, 66

[35] Rafael C. Gonzales and Richard E. Woods. *Digital Image Processing*. Prentice Hall, 2002. ISBN 0201180758. 24, 48

[36] Nolan Goodnight, Cliff Woolley, Gregory Lewin, David Luebke, and Greg Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH / Eurographics conference on Graphics Hardware*, 2003. 47

[37] Moshe Guttman, Lior Wolf, and Daniel Cohen-Or. Semi-automatic stereo extraction from video footage. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 136–142, 2009. 64

[38] Mark Harris and David Luebke. GPGPU. In *Course notes of the ACM SIGGRAPH conference*, 2004. 40

[39] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision.* Cambridge University Press, June 2000. 61, 64, 75

[40] Kai Hormann, Bruno Lévy, and Alla Sheffer. Mesh parameterization: Theory and practice. In *ACM SIGGRAPH Course Notes.* ACM, 2007. 68

[41] Berthold K. P. Horn and Brian G. Schunck. Determining optical flow. *Artificial Intelligence*, 17(1-3):185–203, 1981. 30, 63, 92

[42] Youichi Horry, Ken-ichi Anjyo, and Kiyoshi Arai. Tour into the picture: Using a spidery mesh interface to make animation from a single image. In *Proceedings of SIGGRAPH 1997*, Computer Graphics Proceedings, Annual Conference Series, pages 225–232. ACM Press / Addison-Wesley Publishing Co., August 1997. 61, 64, 74

[43] Michal Irani, P. Anandan, and Steve Hsu. Mosaic based representations of video sequences and their applications. In *Proceedings of the International Conference on Computer Vision*, pages 605–611. IEEE, 1995. 63

[44] Michael C. Kaye. System and method for dimensionalization processing of images in consideration of a predetermined image projection format. U.S. Patent 6208348, March 2001. 12

[45] Erum Arif Khan, Erik Reinhard, Roland Fleming, and Heinrich Buelthoff. Image-based material editing. *ACM Transactions on Graphics*, 25(3), July 2006. Proceedings of the ACM SIGGRAPH conference. 76

[46] Johannes Kopf, Michael F. Cohen, Dani Lischinski, and Matt Uyttendaele. Joint bilateral upsampling. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2007)*, 26(3):96:1–96:5, July 2007. 12, 58

[47] Sanjeev Koppal, Charles Lawrence Zitnick, Michael Cohen, Sing Bing Kang, Bryan Ressler, and Alex Colburn. A viewer-centric editor for stereoscopic cinema. *IEEE Computer Graphics and Applications*, 99, 2010. 64

[48] Anat Levin, Dani Lischinski, and Yair Weiss. Colorization using optimization. *ACM Transactions on Graphics*, 23(3), July 2004. Proceedings of the ACM SIGGRAPH conference. 12, 13, 34, 46, 86

[49] Anat Levin, Dani Lischinski, and Yair Weiss. A closed form solution to natural image matting. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 61–68, 2006. 29, 61, 63, 70, 81

[50] Anat Levin, Alex Rav-Acha, and Dani Lischinski. Spectral matting. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2007. 46

[51] Bruno Lévy. Dual domain extrapolation. *ACM Transactions on Graphics*, 22(3), July 2003. Proceedings of the ACM SIGGRAPH conference. 76

[52] Yin Li, Jian Sun, and Heung-Yeung Shum. Video object cut and paste. *ACM Transactions on Graphics*, 24(3):595–600, 2005. 63

[53] Jae S. Lim. *Two-Dimensional Signal and Image Processing.* Prentice Hall, 1990. ISBN 0139353224. 19

[54] Dani Lischinski, Zeev Farbman, Matt Uyttendaele, and Richard Szeliski. Interactive local adjustment of tonal values. *ACM Transactions on Graphics*, 25(3):646 – 653, 2006. Proceedings of the ACM SIGGRAPH conference. 12, 34, 46

[55] Ce Liu, William T. Freeman, Edward H. Adelson, and Yair Weiss. Human-assisted motion annotation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Conference.* IEEE, 2008. 85

[56] Ce Liu, Antonio Torralba, William T. Freeman, Frédo Durand, and Edward H. Adelson. Motion magnification. *ACM Transactions on Graphics*, 24(3):519–526, 2005. 63, 65, 66

[57] Microsoft Corporation. Kinect, 2010. 93

[58] Neil Molino, Zhaosheng Bao, and Ron Fedkiw. A virtual node algorithm for changing mesh topology during simulation. *ACM Transactions on Graphics*, 23(3):385–392, 2004. 67

[59] Byong Mok Oh, Max Chen, Julie Dorsey, and Frédo Durand. Image-based modeling and photo editing. In *Proceedings of the ACM SIGGRAPH conference*, 2001. 12, 13, 61, 64, 74

[60] Alan V. Oppenheim, Alan S. Willsky, and S. Hamid. *Signals and Systems.* Prentice Hall, 2nd edition, 1996. ISBN 9780138147570. 19

[61] Sylvain Paris and Frédo Durand. A fast approximation of the bilateral filter using a signal processing approach. In *Proceedings of the European Conference on Computer Vision*, 2006. 14, 34, 36, 39, 40, 41, 55

[62] Steven G. Parker, James Bigler, Andreas Dietrich, Friedrich Heiko, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. OptiX: a general purpose ray tracing engine. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 29(4):66:1–66:13, July 2010. 86

[63] Pietro Perona and Jitendra Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions Pattern Analysis Machine Intelligence*, 12(7):629–639, July 1990. 11, 26, 27, 34

[64] Georg Petschnigg, Maneesh Agrawala, Hugues Hoppe, Richard Szeliski, Michael Cohen, and Kentaro Toyama. Digital photography with flash and no-flash image pairs. *ACM Transactions on Graphics*, 23(3), July 2004. Proceedings of the ACM SIGGRAPH conference. 12, 14, 44, 58

[65] Tuan Q. Pham and Lucas J. van Vliet. Separable bilateral filtering for fast video preprocessing. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, 2005. 14, 34, 50

[66] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kauffmann, August 2004. 79

[67] Victor Podlozhnyuk. Histogram calculation in CUDA. http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/histogram256/doc/histogram.pdf, 2007. 26, 55

[68] Fatih Porikli. Constantt time O(1) bilateral filtering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, page 108. IEEE, 2008. 56

[69] Thomas Porter and Tom Duff. Compositing digital images. *Computer Graphics*, 18(3):253–259, 1984. 18

[70] Charles Poynton. *Digital Video and HDTV: Algorithms and Interfaces*. Morgan Kaufmann, 2002. ISBN 1558607927. 23

[71] Quantel Ltd. Pablo. http://goo.gl/M7d4, 2010. 64

[72] Alex Rav-Acha, Pushmeet Kohli, Carsten Rother, and Andrew Fitzgibbon. Unwrap mosaics: a new representation for video editing. *ACM Transactions on Graphics*, 27(3):17:1–17:11, 2008. 63

[73] RealVIZ Corporation. RealVIZ MatchMover, 2007. 80

[74] Erik Reinhard, Greg Ward, Sumanta Pattanaik, and Paul Debevec. *High Dynamic Range Imaging: Acquisition, Display, and Image-Based Lighting*. Morgan Kaufmann, 2005. ISBN 0125852630. 18

[75] Michael Rubinstein, Ariel Shamir, and Shai Avidan. Improved seam carving for video retargeting. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008)*, 27(3):1–9, 2008. 29

[76] Peter Sand and Seth Teller. Particle video: Long-range motion estimation using point trajectories. In *Proceedings of the IEEE Computer Vision and Pattern Recognition Conference*. IEEE, 2006. 63

[77] Steven M. Seitz, Brian Curless, James Diebel, Daniel Scharstein, and Richard Szeliski. A comparison and evaluation of multi-view stereo reconstruction algorithms. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 519–528. IEEE, June 2006. 64

[78] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106. ACM, August 2007. 26, 55

[79] Robert M. Shapley and David J. Tolhurst. Edge detectors in human vision. *The Journal of Physiology*, 229(1):165–183, 1973. 26

[80] Jianbo Shi and Carlo Tomasi. Good features to track. *Proceedings of the IEEE Computer Vision and Pattern Recognition*, pages 593–600, 1994. 13, 63

[81] Sudipta N. Sinha, Drew Steedly, Richard Szeliski, Maneesh Agrawala, and Marc Pollefeys. Interactive 3D architectural modeling from unordered photo collections. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2008)*, 27(5):1–10, 2008. 84

[82] Alvy Ray Smith and James F. Blinn. Blue screen matting. In *Proceedings of the 23rd annual conference on computer graphics and interactive techniques*, SIGGRAPH '96, pages 259–268, New York, NY, USA, 1996. ACM. 93

[83] Stephen M. Smith and J. Michael Brady. SUSAN – a new approach to low level image processing. *International Journal of Computer Vision*, 23(1):45–78, May 1997. 14, 34

[84] Nir Sochen, Ron Kimmel, and Ravi Malladi. A general framework for low level vision. *IEEE Transactions in Image Processing*, 7:310–318, 1998. 35

[85] Kartic Subr, Cyril Soler, and Frédo Durand. Edge-preserving multiscale image decomposition based on local extrema. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2009)*, 28(5):147:1–147:9, December 2009. 58

[86] Richard Szeliski. Video mosaics for virtual environments. *IEEE Computer Graphics and Applications*, 16(2):22–30, March 1996. 76

[87] Richard Szeliski. Video-based rendering. In *Proceeedings of the Vision, Modeling, and Visualization Workshop*, page 447, 2004. 63

[88] Richard Szeliski and Polina Golland. Stereo matching with transparency and matting. *International Journal of Computer Vision*, 32(1):45–61, 1999. 84

[89] The Foundry Visionmongers Ltd. Ocula. `http://www.thefoundry.co.uk/products/ocula/`, 2009. 64

[90] Carlo Tomasi and Roberto Manduchi. Bilateral filtering for gray and color images. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 839–846, 1998. 12, 14, 27, 34

[91] Jack Tumblin and Greg Turk. LCIS: A boundary hierarchy for detail-preserving contrast reduction. In *Proceedings of the ACM SIGGRAPH conference*, pages 83–90, August 1999. 11, 13, 34

[92] Anton van den Hengel, Anthony Dick, Thorsten Thormählen, Ben Ward, and Philip H. S. Torr. VideoTrace: rapid interactive scene modelling from video. *ACM Transactions on Graphics*, 26(3):86:1–86:5, 2007. Proceedings of the ACM SIGGRAPH conference. 64, 78

[93] John Y. A. Wang and Edward H. Adelson. Representing moving images with layers. *IEEE Transactions on Image Processing*, 3(5):625–638, Sep 1994. 15, 63

[94] Jue Wang, Pravin Bhat, R. Alex Colburn, Maneesh Agrawala, and Michael F. Cohen. Interactive video cutout. *ACM Transactions on Graphics*, 24(3):585–594, 2005. 63, 80

[95] Jue Wang and Michael Cohen. Optimized color sampling for robust matting. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, June 2007. 29, 61, 63, 70, 81, 86

[96] Ben Weiss. Fast median and bilateral filtering. *ACM Transactions on Graphics*, 25(3):519 – 526, 2006. Proceedings of the ACM SIGGRAPH conference. 14, 34, 42, 54, 56

[97] Philip J. Willis. Projective alpha colour. *Computer Graphics Forum*, 25(3):557–566, Sep 2006. 38

[98] Holger Winnemöller, Sven C. Olsen, and Bruce Gooch. Real-time video abstraction. *ACM Transactions on Graphics*, 25(3):1221 – 1226, 2006. Proceedings of the ACM SIGGRAPH conference. 14, 50

[99] Jiangjian Xiao, Hui Cheng, Harpreet Sawhney, Cen Rao, and Michael Isnardi. Bilateral filtering-based optical flow estimation with occlusion detection. In *Proceedings of the European Conference on Computer Vision*, 2006. 14

[100] Qingxiong Yang, Kar-Han Tan, and Narendra Ahuja. Real-time O(1) bilateral filtering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 557–564. IEEE, 2009. 56

[101] Guofeng Zhang, Zilong Dong, Jiaya Jia, Liang Wan, Tien-Tsin Wong, and Hujun Bao. Refilming with depth-inferred videos. *IEEE Transactions on Visualization and Computer Graphics*, 15(5):828–840, 2009. 64, 92

[102] Li Zhang, Guillaume Dugas-Phocion, Jean-Sebastien Samson, and Steven M. Seitz. Single view modeling of free-form scenes. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2001. 64