IMPROVING FILTERING FOR COMPUTER GRAPHICS

A Dissertation

by

JOSIAH MICHAEL MANSON

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Scott Schaefer
Committee Members,	John Keyser
	Jinxiang Chai
	Guergana Petrova
Head of Department,	Nancy Amato

May 2014

Major Subject: Computer Science

ABSTRACT

When drawing images onto a computer screen, the information in the scene is typically more detailed than can be displayed. Most objects, however, will not be close to the camera, so details have to be filtered out, or anti-aliased, when the objects are drawn on the screen. I describe new methods for filtering images and shapes with high fidelity while using computational resources as efficiently as possible.

Vector graphics are everywhere, from drawing 3D polygons to 2D text and maps for navigation software. Because of its numerous applications, having a fast, high-quality rasterizer is important. I developed a method for analytically rasterizing shapes using wavelets. This approach allows me to produce accurate 2D rasterizations of images and 3D voxelizations of objects, which is the first step in 3D printing. I later improved my method to handle more filters. The resulting algorithm creates higher-quality images than commercial software such as Adobe Acrobat and is several times faster than the most highly optimized commercial products.

The quality of texture filtering also has a dramatic impact on the quality of a rendered image. Textures are images that are applied to 3D surfaces, which typically cannot be mapped to the 2D space of an image without introducing distortions. For situations in which it is impossible to change the rendering pipeline, I developed a method for pre-computing image filters over 3D surfaces. If I can also change the pipeline, I show that it is possible to improve the quality of texture sampling significantly in real-time rendering while using the same memory bandwidth as used in traditional methods.

ACKNOWLEDGEMENTS

I would like to thank my father, Michael Manson, my mother, Lily Bartoszek, and many friends for supporting me through the course of my studies. My decision to pursue a Ph.D. is in large part because of the role model that my father provided as a professor of biology and because of the love of learning that was fostered in me as a child. John Forsyth taught me algebra through college calculus when I was a teenager and his years of friendship and mentoring helped to direct my choice of careers. Without a thoughtful response from Bjarne Stroustrup when I was lost, I may never have gotten a GED and come to Texas A&M.

I would also like to thank my committee members, with all of whom I have had great conversations and taken classes. I have the longest history with Guergana Petrova, with whom I took undergraduate Differential Equations, where she for some reason nicknamed me Stallone. When I later took a class by Guergana on Fourier and wavelet transforms, my class project developed into my initial first-authored publication. I took graduate computer graphics with Jinxiang Chai and physically based modeling classes with John Keyser. I have always had close association with John and his group, and we have often shared communal lab space.

Most of all I want to thank my advisor, Scott Schaefer, for guiding and supporting me even before I started my Ph.D. studies. If it was not for Scott, I would almost certainly be in a different field than computer graphics. Scott has always done his best to help his students to succeed and make a name for themselves by sending us to international conferences to present our work. It is through Scott that I have learned to write technical papers, present in front of an audience, and conduct good research. I hold Scott in the highest esteem.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

Image filtering is the fundamental principle upon which all rendering in computer graphics is based. Although different rendering applications attempt to achieve similar filtering results, the details of how the image function is defined vary greatly, which means that algorithms used to sample the image are also very different. If the source image function is defined as a scaled grid of raster data, the algorithm used to sample a new raster image is different from the algorithm used to sample a vector image of shapes bounded by Bézier curves. Even when the source data is the same, such as a 3D scene defined by triangulated shapes, very different sampling methods, such as rasterization vs. ray-tracing, may be necessary depending on how the image is defined from those triangles. Accurate interreflections are better handled by a ray-tracer, whereas diffuse surfaces are more appropriate for a rasterizer.

Computers have traditionally been desktops with similarly sized monitors having similar resolutions and viewing distances. Recently, computing devices have diversified into tablets and mobile phones with small screens that are viewed up close. We also have set-top boxes like the Xbox and PlayStation connected to high-resolution televisions that are viewed from across the room. Displaying the same content in these different environments requires text and images to be scaled for each device, and whenever there is scaling, there is potential for aliasing artifacts. In order to prevent aliasing, images must be filtered prior to sampling.

Images often contain information that is at a higher resolution than the screen can display. An example of unrepresentable data comes from photography. Digital photographs and the phosphor grid of a computer monitor are both examples of raster graphics in which images are a uniform grid of color values. A high-end monitor will have 2 million pixels,

but digital cameras can capture images containing over 16 million pixels. This means that almost 90% of the captured data must be removed to display a photograph on a computer screen.

Removing data from a photograph is challenging, but tougher compromises arise in vector graphics, which store images as colored regions bounded by curves. Vector graphics are ubiquitous in computer images. Most text is defined by Bézier curve boundaries and can be scaled to any size. This allows the same font definition to be used to draw text at different point sizes on a monitor, to print text on a piece of paper, or to print text on a huge billboard. Vector graphics can also represent pictures such as icons and emblems. A particularly complex combination of text and images comes in the form of maps, which contain enormously detailed information. Thin lines are used for streets or elevations and text is rotated to align with features in the map. Images drawn by movies and interactive applications are a combination of raster and vector graphics, where vector graphics define the shape of an object by its triangles and raster graphics color the surface of the object as a texture. Vector graphics are called scalable because their representation is independent of screen resolution, so that a viewer can always zoom in to see more detail. Thus, vector graphics have infinite resolution because colors change instantaneously across curves.

Image filtering has a large impact on the perception of images drawn on a screen. Using a low-quality filter produces distracting aliasing artifacts that cause an image to look artificial. One prominent artifact is the appearance of jagged edges along the boundaries of polygons and in profiles of objects. Moiré patterns are rarer, but they can cover large areas of the screen when objects are drawn with high-resolution, repeating patterns, such as the banding patterns that are visible in Figure 1.1. Temporal aliasing refers to artifacts that occur when drawing animated objects, such as when a mouse cursor appears as several sequential copies rather than a blurred streak, or when a spinning wheel appears to stop or rotate in the opposite direction.

Figure 1.1: An image captured from Borderlands 2 that exhibits Moiré patterns in a billboard. Small movements create large changes to the pattern.

Movie studios and manufacturers of graphics hardware constantly improve the quality of computer-rendered images, and reduction of aliasing is one of their primary concerns. Multisampling, supersampling, mipmapping, anisotropic filtering, and morphological filtering are all methods used to reduce aliasing. Higher-quality filtering requires more computation to display each pixel, and the consumer demand for better filtering has helped drive the rapid increase in the processing power of GPUs. In addition to making images look better, improving the quality of image filters applied to text increases reading speed and reduces eye strain [122].

The mathematical foundation of image filtering has been adapted from research in signal processing that was conducted by other disciplines, such as telecommunications. The theory is sound, but difficult to implement in real-time rendering systems because of the enormous volume of data that we need to process. Filtering 3D images is also difficult compared to filtering 2D images because mapping the surface of an object to the screen can warp the image in complex ways.

In this dissertation, I highlight my contributions to the state of the art in image filtering. I describe several methods to improve the quality of filters used for voxelizing three-dimensional objects, scaling two-dimensional images, and drawing vector graphics.

I also discuss the application of filtering to surface reconstruction and the methods one can use to reverse the process of rasterization to find the source vector image.

1.1 Sampling Theory

Rendering an image onto a screen is described mathematically as a form of signal processing. Regardless of what algorithm is used to draw the image on a screen or what the internal definition of the scene to be rendered may be, the image can be thought of as a function over the 2D space of the screen. Although the image $I(x, y)$ is defined over a continuous domain, the pixels on a screen are represented as discrete samples of the image function. Unfortunately, it is not sufficient to take point samples of the image I because it is easy to introduce aliasing artifacts.

Before considering the full 2D problem, I analyze image sampling restricted to one dimension. Taking a point sample consists of evaluating our image $I(x)$ at an offset k such that the value of the sample is $I(k)$. The difficulty is that we must represent sampling as an operator applied to the image so that we can characterize the effects of the operator. To this end, I use the Dirac delta function $\delta(x)$. When taking the inner product of δ with the image I,

$$I(k) = \int_{-\infty}^{\infty} I(x)\delta(x - k)\, dx, \tag{1.1}$$

the delta function selects the value of the image at a single point k.

There are two important properties of δ that are apparent from the definition given above. First, δ has an infinitesimally small support because it selects a single point. Second, the inner product evaluates to $I(k)$, so δ has a unit integral.

$$\int_{-\infty}^{\infty} \delta(x)\, dx = 1$$

We define δ in terms of the Gaussian $e^{-\pi x^2}$, which has a unit integral. By taking the limit

of the Gaussian as its standard deviation goes to zero, the region being sampled reduces to a point, and the weight simultaneously increases so that the integral remains one.

$$\delta(x) = \lim_{\sigma \to 0^+} \sigma e^{-\pi \frac{x^2}{\sigma^2}}$$

Because we sample a single point $I(k)$ using a delta function, we can model sampling I over a regular interval as $I(x)\text{Ш}(x)$, where the comb function $\text{Ш}(x)$ is defined as the sum of translated delta functions

$$\text{Ш}(x) = \sum_{k=-\infty}^{\infty} \delta(x-k).$$

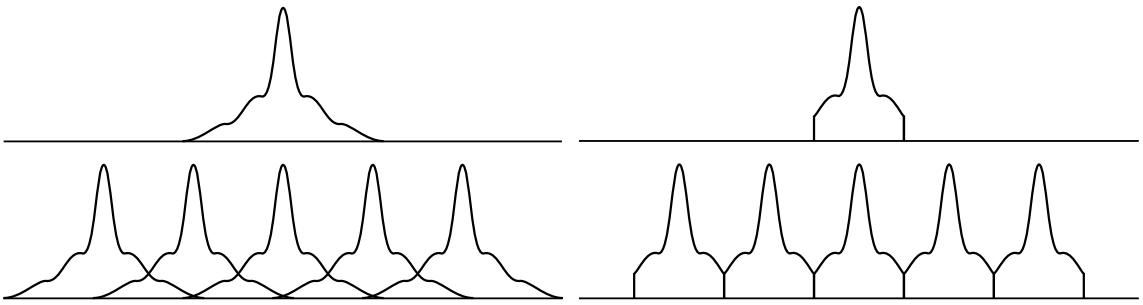
If we sample I at integer values, there is a limit to how high a frequency we can capture and reproduce. In order to determine allowable frequencies, we take the Fourier transform of the sampled image. The Fourier transform F of a function f decomposes f into its constituent frequencies. The Fourier transform operator $\mathcal{F}$ and its inverse $\mathcal{F}^{-1}$ are given by

$$F(s) = \mathcal{F}\{f\} = \int_{-\infty}^{\infty} f(x) e^{-2\pi i s x}\, dx$$

$$f(s) = \mathcal{F}^{-1}\{F\} = \int_{-\infty}^{\infty} F(s) e^{2\pi i s x}\, ds.$$

From the definition of the inverse transform, one can see that f is written as a weighted sum of the frequencies given in F. The fact that F represents a frequency decomposition comes from the relation $e^{i\theta} = \cos(\theta) + i\sin(\theta)$.

There are several other important properties of the Fourier transform that I will use. One is that the Fourier transform of the comb function $\text{Ш} = \mathcal{F}\{\text{Ш}\}$ remains the comb function. Also, convolving two functions is equivalent to multiplying their Fourier transforms so that $\mathcal{F}\{f * g\} = \mathcal{F}\{f\}\mathcal{F}\{g\}$ and $\mathcal{F}\{fg\} = \mathcal{F}\{f\} * \mathcal{F}\{g\}$, where the convolution



(a) Aliased (b) Antialiased

Figure 1.2: The Fourier transform of a signal is shown above in (a). Point sampling a signal adds duplicate copies at integer translations, shown below. If the signal contains high frequencies, the copies overlap and cause aliasing. Aliasing is removed by filtering out high frequencies from the signal (b) so that translated copies do not overlap.

operator $*$ is defined as

$$(f * g)(x) = \int_{-\infty}^{\infty} f(x-s)g(s)\,ds. \tag{1.2}$$

Using these properties of the Fourier transform, we can see that $\mathcal{F}\{I\text{Ш}\} = \mathcal{F}\{I\} * \mathcal{F}\{\text{Ш}\} = \mathcal{F}\{I\} * \text{Ш} = \sum_{k=-\infty}^{\infty} \mathcal{F}\{I\}(s-k)$. In other words, the Fourier transform of our point-sampled image is equal to a sum of integer translates of the Fourier transform of the image. Figure 1.2 shows a depiction of the Fourier transform of a point-sampled image. If the image contains frequencies that are greater in absolute value than $1/2$, then the translated Fourier transforms of the image overlap. This overlap of frequencies is called aliasing and means that frequencies that are higher than one half of the sampling rate appear as low-frequency patterns in the sampled image. These artificial low-frequency patterns misrepresent the underlying data. By discretizing the image, we are guaranteed to lose information, but we have control over what information we lose. We choose to remove frequencies that are too high to reproduce at our new sampling rate before we sample. Specifically, we want to sample the function $\mathcal{F}\{I\}(s)H(s)$, where $H(s)$ is the box

function that is equal to one for all frequencies $-1/2 < s < 1/2$ and is zero otherwise. The convolution property of the Fourier transform implies that we should sample a convolved copy of the image $I(x) * h(x)$, where the low-pass filter is

$$h(x) = \text{sinc}(x) = \mathcal{F}^{-1}\{H\} = \frac{\sin(\pi x)}{\pi x}.$$

This ideal low-pass filter is called the sinc filter and is shown as the black curve in the top panel of Figure 1.4. Other common filters are shown in different colors, and the Fourier transforms of the filters are shown in the bottom panel.

From our earlier definition in Equation 1.1 of how to take a point sample at k, we sample the convolved function $I * h$ by taking the inner product

$$\int_{-\infty}^{\infty} (I(x) * h(x))\delta(x - k)\, dx.$$

Substituting the definition of convolution from Equation 1.2, we find

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} I(s)h(x - s)\, ds\, \delta(x - k)\, dx.$$

By the definition in Equation 1.1, taking the inner product of a function $f(x)$ with $\delta(x-k)$ selects the value $f(k)$, so we simplify the expression to

$$\int_{-\infty}^{\infty} I(s)h(k - s)\, ds.$$

Finally, because h is symmetric,

$$\int_{-\infty}^{\infty} I(s)h(s - k)\, ds.$$

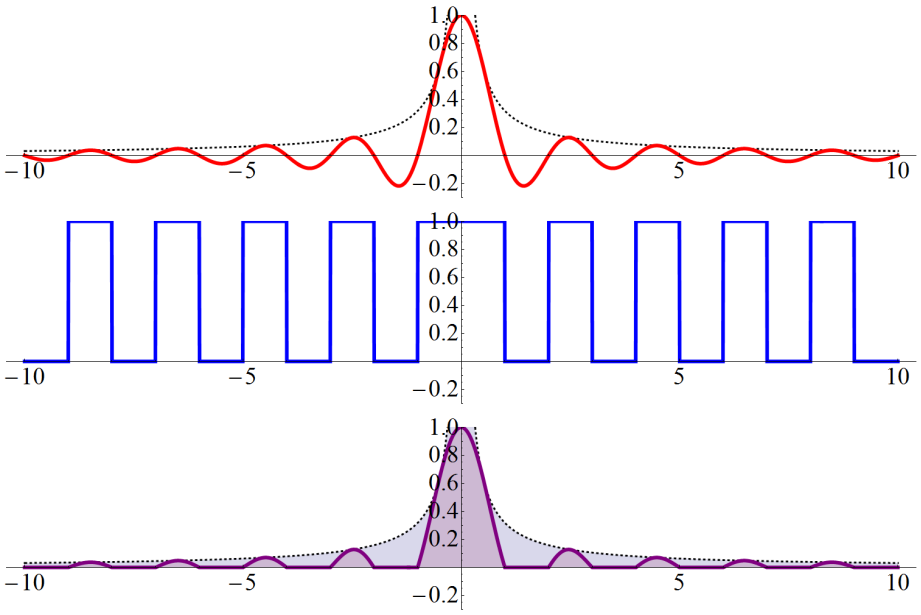


Figure 1.3: An example of how sinc can produce unbounded values from an image in the range $[0, 1]$. From top to bottom, I show sinc, a problematic image, and the product of sinc and the image.

This derivation shows that, in order to sample an image I that was convolved by a filter h at a point k, we take the inner product of the image with the filter centered at the sample.

Although taking an inner product is a relatively simple operation, the computational complexity is proportional to the support, or non-zero area, of the filter h. This poses a problem because the support of the ideal low-pass filter is infinite. Another unfortunate property of the sinc filter is that, if the range of the image is bounded such that $\forall x, I(x) \in [0, 1]$, the negative lobes of the filter can generate values that are greater than one. Thus, the sampled image may need to display pixels that are brighter than the screen can show, assuming that the range $[0, 1]$ maps linearly to the full range of displayable pixel intensities. Pixel values can also be negative, which is physically impossible, as negative light does not exist. Even more troubling is that the contribution to the sinc filter is proportional to $1/x$, and the integral $\int_a^\infty 1/x \; dx = \infty$ is divergent for all a.

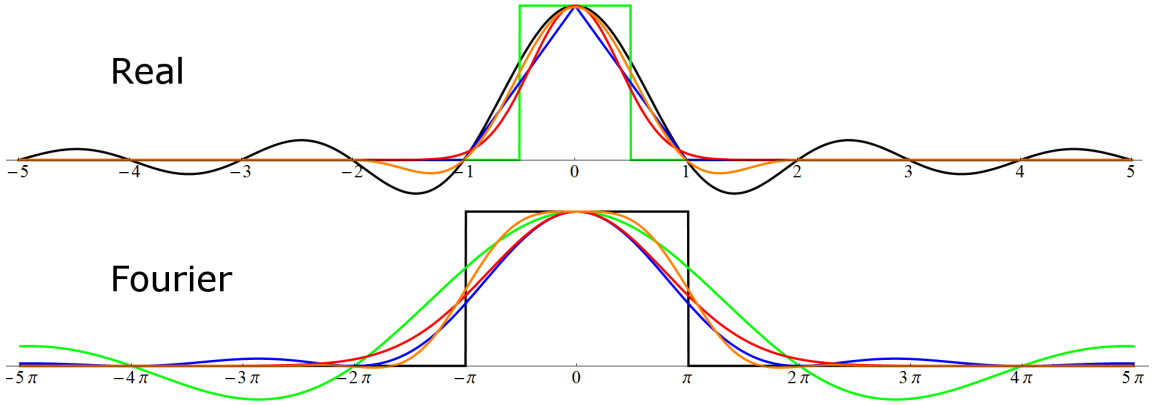


Figure 1.4: Graphs of different filters (top) and their Fourier transforms (bottom), with sinc in black, Lánczos 2 in orange, Gaussian in red, $tent$ in blue, and box in green.

One can therefore construct images where all values are in the range $[0, 1]$ but the filtered image is unbounded either positively or negatively. An example of an image that generates unbounded values is shown in Figure 1.3. In the top graph, the sinc function is drawn in red with $|1/\pi x|$ drawn with dashes. Sampling the repeating image shown in the middle graph integrates the product of sinc and the image, shown on the bottom, which has an unbounded value like the integral of $|1/\pi x|$. Thus, one cannot choose a theoretically justifiable window size a outside of which one can safely ignore the image contribution because for any finite window size a the energy of the filter is finite inside of a and infinite outside.

Because of these problems, especially the infinite computational complexity, images are sampled with either a windowed version of sinc, such as the Lánczos [47] and Mitchell-Netravali [102] filters, or filters that are similar to the main lobe of sinc. Filters in this second category are positive, which means that the filtered image is guaranteed to have values in the $[0, 1]$ range, but the filtered image can appear too blurry. Examples of positive filters are the box, tent, and Gaussian filters.

The Gaussian filter has several useful properties. One is that, although a Gaussian

technically has infinite support, Gaussians can be safely windowed because they have a very fast falloff, and nearly all of the filter energy is contained within a few standard deviations from the mean. The literature often suggests using the Gaussian e^{-ax^2} with $a = 2$ [72, 99] and provides little justification for the choice of a. We wish to approximate sinc by a Gaussian, and if one ignores normalization, the main lobe of sinc does look similar to the Gaussian with $a = 2$. More rigorously, the value $a = 2.37314$ minimizes

$$\min_a \int_{-\infty}^{\infty} (\text{sinc}(x) - e^{-ax^2})^2 \, dx.$$

However, one cannot ignore normalization because a filter must have a unit integral in order to reproduce constant colors. Rather, we should minimize

$$\min_a \int_{-\infty}^{\infty} (\text{sinc}(x) - \frac{\sqrt{a}}{\sqrt{\pi}} e^{-ax^2})^2 \, dx.$$

When accounting for normalization, we now find the minimum at $a = 2.97744$. I therefore suggest that one should use $a = \pi$ because the normalized Gaussian then has the simple form $e^{-\pi x^2}$ and has a value of 1 at $x = 0$. Furthermore, π is close in value to 2.97744. A comparison between sinc and Gaussian functions with different choices of a are shown in Figure 1.5. Gaussians also have the convenient property that the tensor product of two Gaussians is radially symmetric, which is a useful property for filtering 2D images.

Discussion of the extension of sampling a 2D image as compared to a 1D signal is brief in most texts. Samples over a regular 2D grid are modeled as the tensor product of 1D comb filters $\text{Ш}(x, y) = \text{Ш}(x)\text{Ш}(y)$. The Fourier transform is separable, so a tensor product of comb filters implies that the filter should be the tensor product of box functions in the frequency domain, and therefore that the ideal low-pass filter in 2D is the tensor product of sinc filters. Likewise, any approximation of the 2D sinc filter should be a tensor

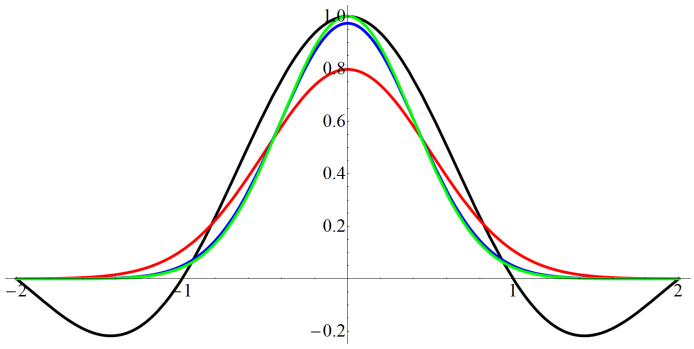


Figure 1.5: Gaussian functions with $a = 2$ (red), $a = 2.97744$ (blue), and $a = \pi$ (green) are drawn with the sinc function (black) that the Gaussians approximate.

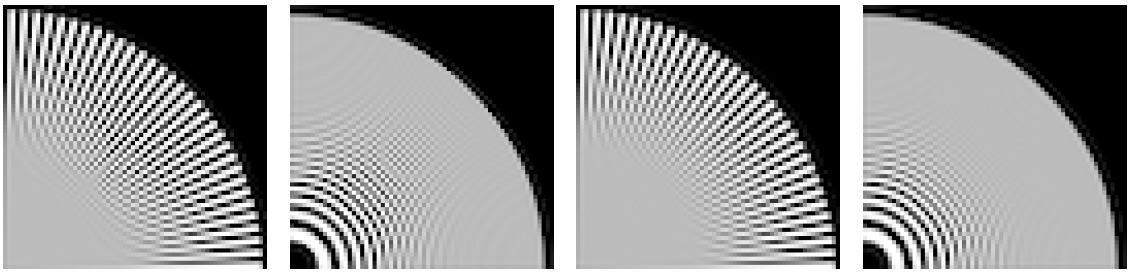
product of 1D approximations.

Tensoring everything in 2D is simple and seems logical, but radial filters may perform better, as shown in Figure 1.6. One will notice that along diagonals, the sampling rate is $1/\sqrt{2}$, but that the band pass of the tensor product filter permits frequencies up to $\sqrt{2}$, which seems like it would cause aliasing. One might imagine that, if anything, the frequency cut-off as a function of angle should be the inverse of the tensor product cut-off. However, this logic breaks down for angles other than a multiples of 45 degrees because lines at other angles do not directly pass through integer grid points, and so do not have well defined sampling rates.

Experiments show that the best frequency cut-off is a radial box filter. This filter is known as the jinc filter and is defined as

$$h(x) = \text{jinc}(x) = \frac{J_1(\pi x)}{\pi x},$$

where J_1 is the Bessel function of the first kind. The jinc function looks similar to a radial sinc function, but it is not the same. For example, jinc is not zero at integer radii. I plot the two filters in Figure 1.7 for comparison. A disadvantage of jinc compared to a tensor prod-



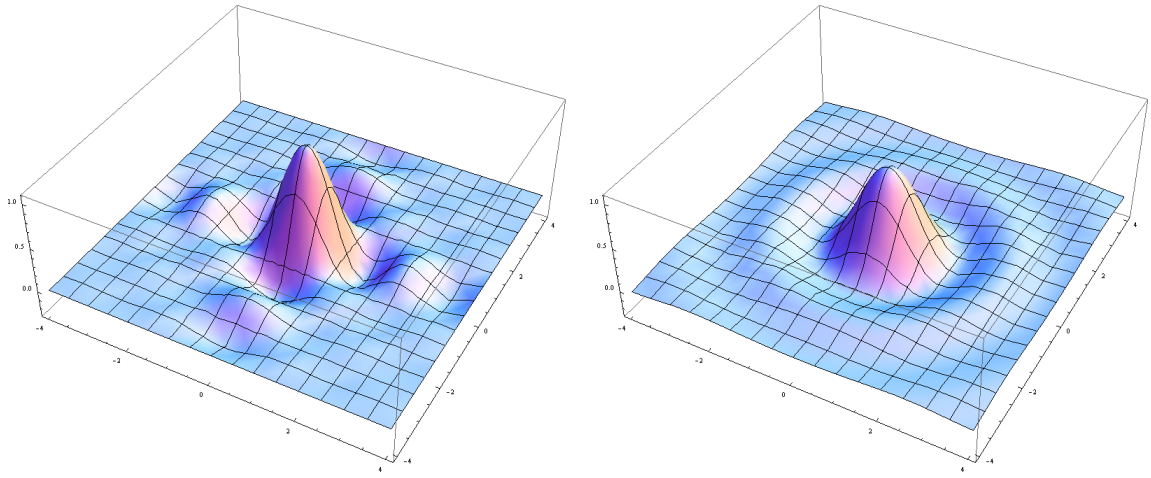
Lánczos 3 Rays | Lánczos 3 Zone Plate | Jinc 3 Rays | Jinc 3 Zone Plate

Figure 1.6: A comparison of the results from filtering with a tensor product of Lánczos 3 filters and with a radial filter jinc of radius 3 that is windowed by the first lobe of sin. The tensor product filter permits too much aliasing along the diagonal.

uct of sinc functions is that jinc does not act as a good reconstruction filter, whereas sinc has useful properties, such as interpolating sample values, reproducing constant functions, and integer translations are orthogonal.

There are several reasons to believe that sinc is not the ideal image filter. Beyond the theoretical concerns I mentioned earlier, the resulting images look bad to the human eye because halos or ripples that appear around discontinuous changes in brightness (ringing) artifacts are too strong. One might also conclude that the best window function for sinc is the box window, which minimizes the $\mathcal{L}_2$ norm between the windowed function and sinc. The box window also minimizes the $\mathcal{L}_2$ norm of the Fourier transforms of the functions according to Plancherel's theorem, but a box window is not used in practice because of excessive its ringing.

The considerations above suggest that the classic Fourier analysis of signal processing may not be appropriate for analyzing image filtering. One step in the right direction is to constrain pixel intensities to the range $[0, 1]$. We can enforce these constraints while minimizing the squared difference between I and a reconstruction of the screen pixels that



(a) Sinc (b) Jinc

Figure 1.7: The tensor product sinc filter is shown in (a), and the radial jinc filter is shown in (b). Notice that the main lobe of jinc is not as high and that the frequency of the ripples is slightly lower than in sinc.

consists of translations of the filter h weighted by the coefficients c_i.

$$\min_{c_i, 0 \leq c_i \leq 1} \int_{-\infty}^{\infty} (I(x) - \sum_i c_i h(x - i))^2 \, dx$$

Here, h denotes the reconstruction filter that is used to display the image on the screen and there is no explicit sampling filter. Solving a constrained minimization can reduce visual artifacts in some cases, as shown in Figure 1.8, but it is expensive and difficult to compute for nonorthogonal filters. I suspect that it may be necessary to minimize a different norm than $\mathcal{L}_2$ because an $\mathcal{L}_2$ minimization in the absence of constraints is equal to sampling with a sinc filter. Rather, it may be necessary to devise a perceptual norm that minimizes the difference a human sees between an image and a down-sampled version of that image. It is known that the human visual system is specialized to recognize edges [121], so edges may need to be handled differently when sampling an image.

Input Lánczos 3 Least Squares Constrained

Figure 1.8: The input image on the left is downsampled using different filtering techniques, then upsampled using bilinear interpolation. The constrained least squares projection onto the bilinear basis looks the sharpest and has the fewest ringing artifacts.

1.1.1 Supersampling

All antialiasing methods perform an integration over the image function I, but it is often impossible to provide a closed-form equation that describes the image over a region, and it is only possible to evaluate a single point in the image at a time. Ray-tracing is a classic example of this type of point sampling, where the rendering algorithm has complete freedom of where to evaluate the image function but can only evaluate at discrete points.

Estimating an integral from a set of point samples is referred to as numerical integration or quadrature, and in the context of image filtering it is referred to as supersampling. Integration can be defined as the limit

$$\int_a^b f(x)\,dx = \lim_{n\to\infty} \sum_{i=1}^{n} f\left((b-a)\frac{i-1/2}{n} + a\right)\frac{b-a}{n}.$$

When sampling a convolved image, we evaluate the function $f(x) = I(x)h(x-k)$ and $[a,b]$ is the support of the filter h. One can achieve an arbitrarily accurate approximation of the color of a pixel by using a sufficiently large number of samples n to calculate the integral.

A practical difficulty in computer graphics is that high frequency patterns often exist

in images that are beyond any reasonable sampling density. For example, a regular pattern tiled over an infinite plane is downscaled by a factor that approaches infinity on the horizon. Any regular sampling pattern and sampling rate can therefore result in aliasing artifacts. People find noise less visually disturbing than aliasing patterns, so it is possible to improve the image quality by randomizing the sampled coordinates. Several papers [43, 32, 102, 49, 54] have investigated the best randomized sampling pattern to use.

Unlike the situation with uniform sampling, it is difficult to determine what percent contribution any particular sample provides to the total color with random sample coordinates. However, as long as samples are equally spaced on average, the numerical approximation of an integral converges to the exact integral. The simplest way to add randomness is to sample the domain with a uniform random distribution. Suppose that a random value within the range $[a, b]$ is returned each time we evaluate the procedure $R(a, b)$. We can then approximate the integral as

$$\int_a^b f(x)\, dx = \lim_{n\to\infty} \sum_{i=1}^{n} f\left(R(b, a)\right) \frac{b-a}{n}.$$

The average space between samples is $\frac{b-a}{n}$ and the samples are weighted accordingly. Although the average space between samples is the same as in uniform sampling, the large variation in distance between samples reduces the convergence rate of the approximation. One way to reduce variability in sample spacing while randomizing samples is to jitter the samples, which uniformly divides the domain into n pieces and takes a random sample of f within each of the pieces

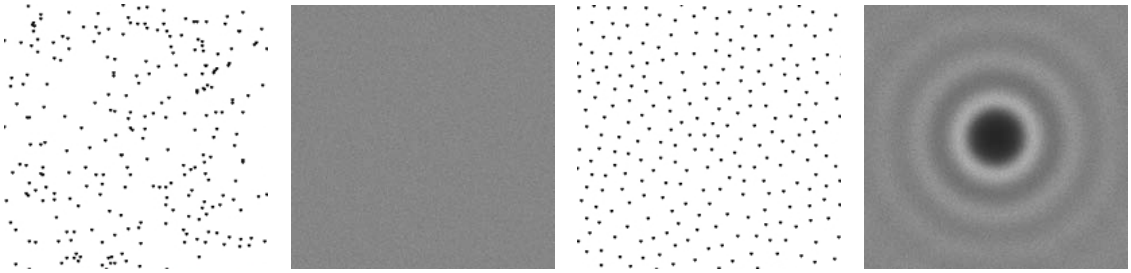
$$\int_a^b f(x)\, dx = \lim_{n\to\infty} \sum_{i=1}^{n} f\left((b-a)\frac{i - R(0,1)}{n} + a\right) \frac{b-a}{n}.$$

One can extend the numerical integration methods that I have described from 1D to

2D by taking uniform random samples within square regions and weighting each sample by the integrated area of the domain divided by the total number of samples. However, the 2D case allows a better randomized sampling strategy called Poisson disk sampling. Figure 1.9 compares uniform and Poisson disk distributions and their power spectra. Poisson disk samples are placed so that no two samples are closer than a minimal distance and the distance to neighboring points is clustered to make the closest points approximately the minimal distance apart. Thus, there will be few additional points closer than about twice the distance between the first set of points. As the distance between points increases, the power spectrum looks more like the spectrum of uniform random samples. These types of semi-regular distributions are referred to as a blue noise in computer graphics and are characterized by circular bands in their power spectra.

The simplest, but computationally inefficient, algorithm for creating such a distribution is called the dart throwing algorithm, in which positions are randomly sampled from a uniform distribution. If the new sample is within the minimum distance to any of the existing samples, the new sample is rejected. Otherwise, the new sample is added to the list of Poisson disk samples. Although it is difficult to create a Poisson disk distribution, the Poisson disk distribution is considered to be the most efficient randomized distribution for numerically approximating integrals, and several papers [57, 132, 49, 54] describe more efficient algorithms for creating such a distribution.

It is inefficient to use the same number of samples to evaluate the color of every pixel in an image. For example, if the image is constant within the support of the filter, a single color sample is sufficient to determine the color of the filtered pixel. If the support of the filter intersects a complicated part of the image function, far more samples are needed. Larger function values also require more samples than smaller function values. From a



Uniform Points | Uniform Power Spectrum | Poisson Points | Poisson Power Spectrum

Figure 1.9: Comparison of a uniform random distribution and a Poisson disk distribution. Notice that the Poisson disk distribution has regular spacing between samples but has random angles between points.

statistical perspective, the integral used to evaluate a pixel color

$$\mu = \int_{-\infty}^{\infty} h(x)I(x)\,dx = \frac{\int_{-\infty}^{\infty} h(x)I(x)\,dx}{\int_{-\infty}^{\infty} h(x)\,dx} \approx \frac{\sum_{i=1}^{n} h_i I_i}{\sum_{i=1}^{n} h_i}$$

can be viewed as the mean value of the image intensity weighted by $h(x)$. In practice the approximate mean is a weighted average of n measured image intensities I_i with associated weights h_i, where the sample positions come from a uniform distribution. How accurately we estimate the mean intensity is measured by the variance of the mean

$$\sigma_\mu^2 = \frac{\sum_{i=1}^{n} h_i}{(\sum_{i=1}^{n} h_i)^2 - \sum_{i=1}^{n} h_i^2} \sum_{i=1}^{n} h_i (I_i - \mu)^2.$$

We can stop adding samples to determine the pixel color when the variance of the mean is below a predefined tolerance. In a linear color space with intensities represented as eight bit integers, a reasonable tolerance is achieved when $\sigma_\mu^2 < 2^{-16}$.

1.1.2 Gamma Correction and Color Spaces

Our perception of the brightness of a pixel is not linear in the amount of light that is emitted from the pixel. Humans are more sensitive to changes in brightness when a phos-

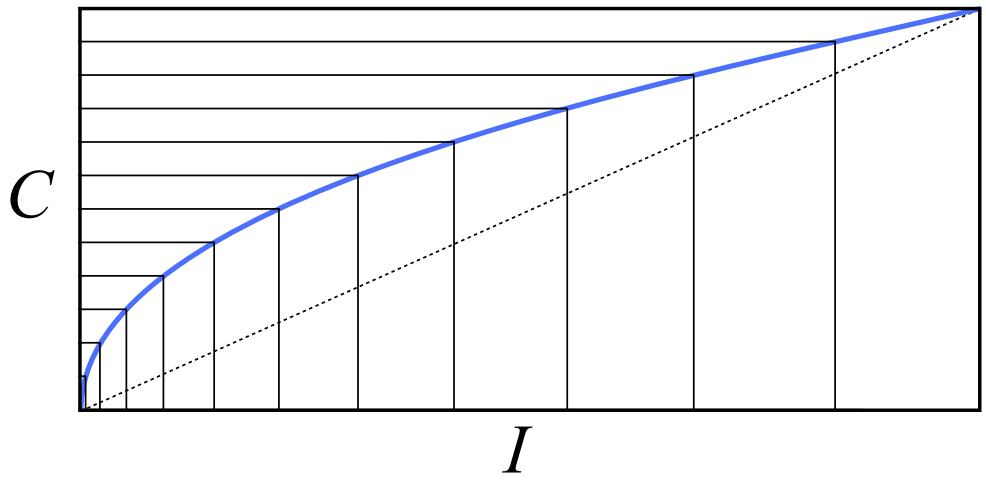


Figure 1.10: A graph of a gamma compression function with $\gamma = 2.2$. Notice that uniform discretization of the compressed space C gives a nonuniform discretization of the linear intensities I such that low intensities are sampled more densely.

phor is dim than when it is bright. Eight bits of gray levels are barely enough for people to perceive a continuous gradient in brightness. In order to represent darker intensities more accurately, pixel intensities are typically mapped into a compressed color space by a function $C(x) = I(x)^{1/\gamma}$ that is parameterized by a constant $\gamma \approx 2$. Given values in the compressed space, the compression is inverted to find the image intensities by $I(x) = C(x)^{\gamma}$. Recall that image intensities are always in the range $[0, 1]$, so $C : [0, 1] \to [0, 1]$ preserves the range. One can see in Figure 1.10 how gamma compression gives more precision to low intensities.

I consider an image to be an intensity function, but images contain color in addition to intensity. Fortunately, RGB encoded images can be filtered by treating the red, green, and blue color channels independently. The RGB color space is a discretization of the color spectrum, and each color channel encodes the amount of light that is emitted by differently colored phosphors. Most manufacturers of display devices now use the sRGB color space, which is similar to $\gamma = 2.2$. All displays take gamma-compressed inputs, and most file

formats are stored in a gamma-compressed format. In contrast, image processing operations assume a linear color space, so it is critical to perform gamma correction before and after any image processing operations. For example, when ray-tracing a scene, textures should be gamma-decompressed, image sampling and lighting calculations should be performed in linear space, and the color space should be gamma-compressed before writing to the disk or to the screen.

2. ANALYTIC RASTERIZATION*

There are two basic ways to represent images: raster graphics and vector graphics. Raster graphics are images that are stored as an array of pixel values, whereas vector graphics represent images as collections of geometric shapes, such as lines, curves, circles, and polygons. Vector graphics are used in many applications in computer graphics. In 3D, almost all geometry is represented in vector form as polygons or smooth parametric shapes. Even in two dimensions, vector images are extremely common. For example, nearly all text is drawn with vector fonts, in which letters are stored as quadratic curves that indicate the transition between the inside and outside of the letter. Point sampling pixels from a polygon results in jagged edges, which are a form of aliasing [34] because discontinuous changes in a function contain infinitely high frequencies. Vector images are also used in maps, signs, and logos to give a crisp, clear appearance.

Although many shapes are stored in vector format, displays are typically raster devices. This means that vector images must be converted to pixel intensities to be displayed. A major benefit of vector images is that they can be drawn accurately at different resolutions. For example, text appears the same when printed at 600 dpi on paper or on a 72 dpi screen, and the butterflies in Figure 2.1 are easily drawn at different sizes. Mobile devices have become another important visual medium and have generated a renewed interest in scalable graphics. In particular, websites are often viewed both on computers with screens thousands of pixels wide and on phones that have much smaller screens with higher pixel density. Therefore, text, images, and icons must scale gracefully.

There has been a trend to improve the quality of images rather than the number of

*Reprinted with permission from "Analytic Rasterization of Curves with Polynomial Filters" by Josiah Manson and Scott Schaefer, 2013. Computer Graphics Forum, 32 (2), 499–507, © 2013 The Author(s) Computer Graphics Forum © 2013 The Eurographics Association and Blackwell Publishing Ltd.

Figure 2.1: Vector graphic art of butterflies represented by cubic curves, scaled by the golden ratio. The images were analytically rasterized using our method with a jinc filter of radius three.

pixels that are drawn. In part, this trend exists because the rate at which mathematical operations are performed is increasing faster than the speed of memory. One avenue for improving pixel quality is to sample images with better antialiasing filters.

Most rasterizers draw polygons and approximate curved boundaries as many-sided polygons [73]. Thus, aliasing appears both when sampling points on the curve to form a polygonal approximation of the true image, which we call geometric aliasing, and from sampling image intensities from the polygon. We reduce both forms of aliasing simultaneously by rasterizing curved boundaries using high-order filters. We show that analytic solutions for pixel values can be found using polynomial filters to rasterize curved shapes with linear color gradients. We rasterize shapes with a variety of different curves and have closed-form solutions for Bézier curves of arbitrary degree as well as rational quadratic Bézier curves, which allow us to rasterize exact circles and ellipses. This set of shapes encompasses most primitives that make up vector images.

2.1 Related Work

The simplest form of rasterization is point sampling, which determines if a point is inside a polygon or not. Calculating which polygon, if any, contains an arbitrary sample

can be done using ray casting [5]. One can also recursively cut the image into quadrants, subdividing around polygon edges [131]. More typically, pixel values are calculated a scanline at a time [136].

The solution to reducing aliasing artifacts is to convolve the signal with a filter that removes frequencies that are higher than half the sampling rate. Supersampling approximates the convolution, but aliasing artifacts can still occur for very high-frequency images. In ray casting, there is significant freedom in sample placement, and several early papers [43, 32, 101] analyze image sampling patterns. These works show that samples should be random but have uniform density, and there is renewed interest in such sampling patterns [81, 75, 84, 57, 132, 54, 49]. There is also research on how to optimize the sampling pattern and weights simultaneously [88].

Calculating prefiltered pixel values requires an integral over areas of an image, and accurate approximations require numerous point samples, regardless of the sampling pattern. Calculating area integrals is complicated, especially when taking occlusion into account, but calculating line integrals is more tractable. Several papers [67, 80, 66] have described methods of approximating area integrals by calculating a one dimensional quadrature over line samples that are computed analytically. This approach reduces the dimensionality of the problem by one and yields a rasterization with fewer artifacts from the same number of samples.

One can also evaluate area integrals exactly. One of the first methods to do so [24] solves both the visibility and integration problems simultaneously. The method clips polygons to pixels, and then against each other, so that the remaining polygons in a pixel are all completely visible. The method then sums the areas of the polygons in the pixel times their color, which is equivalent to sampling with a box filter. Some other methods simplify clipping polygons to pixels by first cutting polygons into trapezoids aligned with scanlines [59]. These trapezoids are easier to clip to pixels than polygons, and one can easily

calculate the area of a trapezoid.

Duff uses trapezoid decomposition to evaluate polynomial filters over polygons [48]. Our method is more general, because we are able to rasterize curved boundaries in addition to polygonal boundaries. Our derivation of closed-form rasterization equations also leads to a different rasterization algorithm because Duff integrates over areas perpendicular to the scanline, whereas we integrate over the boundary along the scanline.

Some methods rasterize polygons with radial filters. An early method [25] clips polygons to pixels and adds the contribution of each edge using a radial filter. If an edge forms a triangle with the center of the filter, that triangle is split into two right triangles. Each right triangle is parameterized by two values that index a lookup table to find the contribution of that edge. A more recent algorithm [92] eliminates the need for clipping polygons to pixels by taking the modulo of final pixel values. This method also has analytic solutions for polynomial filters, but it cannot use filters with negative values because of the modulo operation, and, thus, the authors published a paper describing positive filters that are suitable for their method [91].

There are also some rasterizers that are difficult to classify. One method approximates the rasterization of self-intersecting polygons [44]. In this method, the contours within pixels that contain intersections are simplified and clipped against other contours. Another method calculates analytic rasterizations of shapes filtered by box splines [98]. Filters must be positive, and the method only rasterizes triangles. Auzinger et al. [6] analytically rasterize antialiased triangles and tetrahedra with linear data defined over the simplices. We also handle linear gradients, but we analytically rasterize complex, curved, two-dimensional shapes. A process similar to polygon rasterization has also been used to calculate surface irradiance in a raytracer by using Stokes' theorem to evaluate incoming light from polygonal light sources [28, 29].

The main contribution of this chapter is prefiltered, antialiased rasterization of shapes

with curved boundaries. There has been some work on directly rasterizing shapes with curved boundaries, but most rasterizers simply approximate curved boundaries by subdividing the curves into many-sided polygons [26]. One of the first papers to rasterize curved shapes accurately [33] determined if pixels were inside or outside of the region bounded by the curves, but it only performed point sampling for rasterization.

Most antialiased rasterization methods estimate the distance to the boundary in some way [52, 94, 110, 109, 104] and approximate a radial filter by setting pixel values based on the distance to the boundary. Using distance to approximate a radial filter is only exact for line segments when no vertices are in the support of the filter, and this method works especially poorly between curves that are within a filter diameter of one another.

2.2 Rasterizer

Rasterization is the process of sampling an image $I(x, y)$ defined by a set of closed shapes, where each shape M_i is defined by a set of boundary curves ∂M_i. We assume that the input does not contain overlapping shapes. If this is not the case, we can preprocess the input to remove overlaps [70]. We define $I(x, y) = \sum_i I_i(x, y)$ as the sum of images of each shape $I_i(x, y)$, where each shape has color $c_i(x, y)$ inside of M_i and is zero outside. To remove aliasing, we prefilter the image by convolving $I(x, y)$ with a low-pass filter $h(x, y)$ prior to point sampling, which is equivalent to taking the inner product of the image $I(x, y)$ with the filter centered at each pixel. We explain how to rasterize the image of a single shape and drop the shape index i in the remainder of this chapter because the final image is a sum of shape images. We calculate the value of a pixel located at ℓ_x, ℓ_y by the area integral

$$\iint_{\mathbb{R}^2} I(x, y)h(x - \ell_x, y - \ell_y)\, dx\, dy.$$

We can simplify the expression by changing the domain of integration to be only within M because the image is zero outside of the boundary.

$$\iint_{x,y \in M} c(x,y)h(x-\ell_x, y-\ell_y)\,dx\,dy \tag{2.1}$$

The divergence theorem relates an integral over the boundary of a domain to an integral over the domain by

$$\oint_{p(s)\in\partial M} F(p(s)) \cdot n(s)\,ds = \iint_{x,y\in M} \nabla \cdot F(x,y)\,dx\,dy, \tag{2.2}$$

where the unit normal of the shape is given by $n(s)$ and the boundary of the domain is given by the arc-length parameterized curve $p(s)$. Therefore, if we find a vector function $F(x,y)$ whose divergence is equal to $f(x,y) = c(x,y)h(x-\ell_x, y-\ell_y)$, then we can evaluate Equation 2.1 as a boundary integral.

Because divergence is a sum of differentials, we find $F(x,y)$ by integrating $f(x,y)$, and we parameterize solutions for $F(x,y)$ by α such that

$$F(x,y) = \begin{pmatrix} F_x(x,y) \\ F_y(x,y) \end{pmatrix} = \begin{pmatrix} \alpha \int_{-\infty}^{x} f(u,y)\,du \\ (1-\alpha)\int_{-\infty}^{y} f(x,u)\,du \end{pmatrix}.$$

This approach converts the rasterization problem from integration over an unknown interior to integration over a known boundary. However, if we choose $\alpha = 1$, then $F(x,y)$ has infinite support only in the direction of the scanline, which we exploit during rasterization in Section 2.2.1. I use a similar derivation in Section 3.2.1 to calculate wavelet coefficients. In the case of wavelets, it is possible to choose α so that $F(x,y)$ has compact support, but compact support is impossible for arbitrary polynomial filters.

The unit normal of the curve is defined as $n(s) = p^{\perp}(s)/\|p'(s)\|$, where the direction

perpendicular to the curve is

$$p^{\perp}(s) = \begin{pmatrix} p'_y(s) \\ -p'_x(s) \end{pmatrix}.$$

We use the notation that $p'(s)$ is the derivative of $p(s)$ with respect to s, and the x and y subscripts refer to the first and second components of a vector quantity, respectively. Changing variables from an arc-length parameterization ds to a uniform parameterization dt weights the differential by $ds = \|p'(t)\|\ dt$, which simplifies our expression because $n(s)\ ds = \frac{p^{\perp}(t)\|p'(t)\|\ dt}{\|p'(t)\|} = p^{\perp}(t)\ dt$. The dot product between vector functions also simplifies to a scalar product,

$$\oint F(p(t)) \cdot p^{\perp}(t)\ dt = \oint F_x(p(t))p'_y(t)\ dt, \tag{2.3}$$

which uses only the x-component, $F_x(x, y)$, of $F(x, y)$ because the y-component, $F_y(x, y)$, is zero when $\alpha = 0$. Note that most boundary curves $p(t)$ are defined piecewise and that it is trivial to sum the integrals of each segment of the curve to evaluate Equation 2.3. Differentiation, integration, multiplication, and function application are closed under polynomials, so if $h(x, y)$, $c(x, y)$, and $p(t)$ are polynomial, the entire expression evaluates to a polynomial. An example of $f(x, y)$ and its integral $F_x(x, y)$ are shown in Figure 2.2. In this figure, it is clear that the support of $F_x(x, y)$ extends beyond the support of $f(x, y)$ in the positive x-direction such that the value of a pixel depends only on the boundary to the right of the pixel. There are a variety of filters $h(x, y)$ to choose from. We approximate windowed jinc and Lánczos filters by piecewise cubic and bicubic polynomials, respectively, with C^0 continuity aligned to pixel boundaries. This approximation produces images that are visually indistinguishable from the images produced by the original filters.

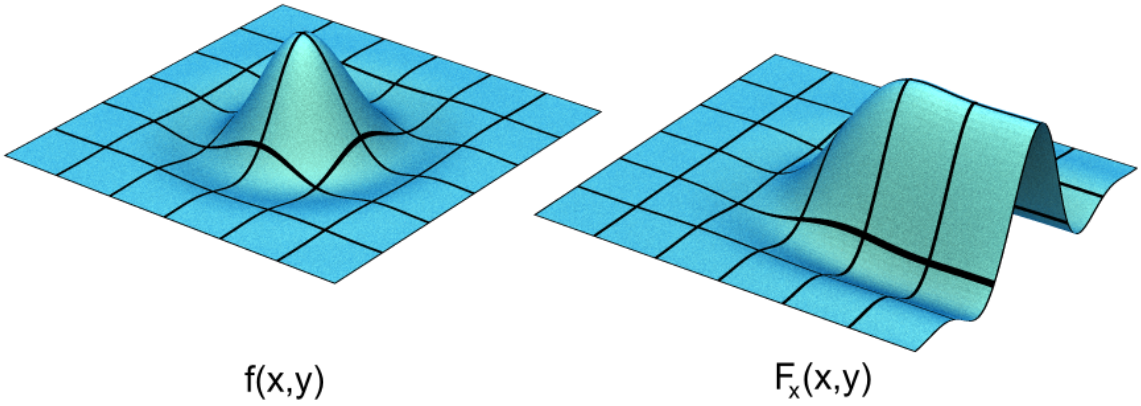


Figure 2.2: A filter $f(x, y)$ is shown on the left and its integral in the x-direction $F_x(x, y)$ is shown on the right. The graphs are plotted over the filter's support. In $F_x(x, y)$, values remain constant in the direction of integration beyond $f(x, y)$'s support.

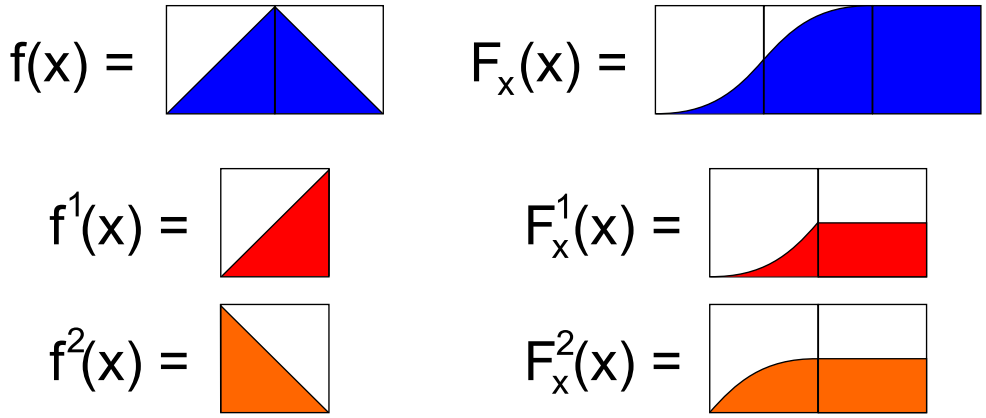


Figure 2.3: A filter $f(x)$ decomposed into pixel-sized pieces and evaluate each piece separately. The integral over $f(x)$ is equal to one, but the integral over each piece is not.

2.2.1 Scanline Algorithm

To rasterize an image, we evaluate Equation 2.3 using the curves that are in the support of $F_x(x, y)$, which is finite in y but is infinite in the positive x-direction. This directionality of infinite support leads to a scanline rasterization algorithm that evaluates pixels from right to left, although reversing the direction of integration yields a traditional left to right rasterization algorithm. We assume that the filter is represented as a piecewise polynomial in which the pieces align to the pixel grid. Because the polynomial pieces do not overlap, we can write the filter as a sum of polynomial pieces. In Figure 2.3, we show the decomposition of a one-dimensional tent filter into multiple pieces. We show the full filter and its integral in the top row of the figure, below which we show the decomposition of the filter into pixel-sized pieces and the integrals of the pieces. Each piece $h^{(i,j)}(x, y)$ is defined over the $(x, y) \in [0, 1)^2$ domain and is indexed by i, j such that

$$h(x, y) = \sum_i \sum_j h^{(i,j)}(x - i, y - j). \tag{2.4}$$

We then define the colored filter pieces over the same domain as $f^{(i,j)}(x, y) = c(x + \ell_x, y + \ell_y)h^{(i,j)}(x, y)$, with integrals $F_x^{(i,j)}(x, y) = \int_{-\infty}^{x} f^{(i,j)}(u, y)\, du$. Because our filter is piecewise polynomial with polynomial pieces that align to the pixel grid, we cut boundary curves to the pixel grid. With proper care, solving for the points to cut polynomials is robust even for degenerate curves [11, 12]. We index cut curves by the cell λ_x, λ_y such that each curve segment

$$p^{(\lambda_x, \lambda_y)}(t) = p(t) \cap \left([0, 1)^2 + (\lambda_x, \lambda_y)\right) - (\lambda_x, \lambda_y)$$

is within the same $[0, 1)^2$ domain as the filter pieces. We can evaluate each pixel-sized piece of the filter independently and sum the results to evaluate a pixel with indices ℓ_x, ℓ_y

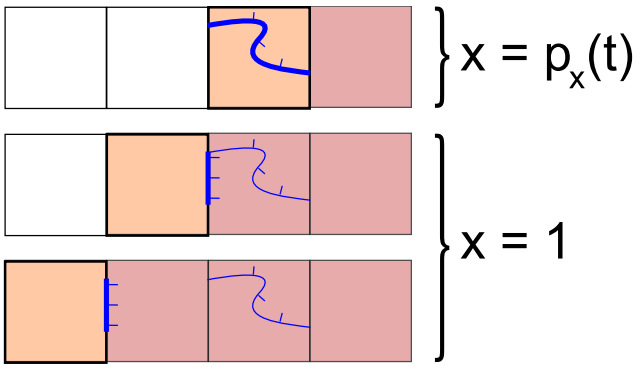


Figure 2.4: The thick box outlines show which pixel is being evaluated for a filter piece $f^{(i,j)}(x, y)$. We calculate two values for each curve: one within the filter piece and one for all pieces to the left, found by setting the curve's x-coordinate equal to one.

by

$$\sum_i \sum_j \int_0^1 F_x^{(i,j)} \left(p^{(\ell_x+i,\ell_y+j)}(t)\right) p_y'^{(\ell_x+i,\ell_y+j)}(t)\, dt. \tag{2.5}$$

We can evaluate Equation 2.5 in any order because summation is commutative, and this property allows us to evaluate each curve, scanline, and filter piece in parallel. Given a curve $p^{(\lambda_x,\lambda_y)}(t)$ and filter piece $f^{(i,j)}(x, y)$, we can see from Equation 2.5 that $p^{(\lambda_x,\lambda_y)}(t) = p^{(\ell_x+i,\ell_y+j)}(t)$, and we therefore add the contribution from curve $p^{(\lambda_x,\lambda_y)}(t)$ to the pixel $(\lambda_x - i, \lambda_y - j)$. Our only constraint is that we must evaluate each filter piece in scanline order because of the infinite support of $F_x^{(i,j)}(x, y)$.

Figure 2.4 illustrates the process of rasterizing a single piece of $f^{(i,j)}(x, y)$. We show a scanline consisting of four pixels, where each row shows how we evaluate the pixel drawn in orange with the extended support of $F_x^{(i,j)}(x, y)$ drawn in red. The figure shows evaluation of a single curve drawn in blue, with normals to indicate the orientation of the curve. When the curve is within the support of the filter piece (top row), we evaluate the curve integral and immediately update the pixel value. All pixels to the left of the curve (bottom rows) will have the same value added because $F_x^{(i,j)}(x, y)$ is constant with respect

to x in the red region. We propagate this constant contribution by evaluating the curve once to update an accumulator that we add to remaining pixels in the scanline. Furthermore, we can simplify Equation 2.3 for propagated values by treating the curve as a line segment that connects the end points of the curve and has x-coordinates equal to one. Before processing a scanline, the accumulator is initialized to zero outside of the shape. The orientation of a curve automatically adds color as the filter moves inside the boundary of the shape and subtracts color as the filter moves outside of the shape.

Rasterizing shapes with linear color gradients is almost the same as rasterizing shapes with constant color. The only difference is that, in Equation 2.5, $F_x^{(i,j)}(x, y)$ depends on the pixel coordinates ℓ_x, ℓ_y when $c(x, y)$ is linear. We define the color of the shape as $c(x, y) = C_0 + C_1 x + C_2 y$ in the reference coordinate system of the shape. This means that the difference in value when moving from a pixel (ℓ_x, ℓ_y) to $(\ell_x - 1, \ell_y)$ is

$$-C_1 \int_{-\infty}^{x} h^{(i,j)}(u, y)\, du. \tag{2.6}$$

The integral of $h^{(i,j)}(x, y)$ is constant to the right of the filter piece, so we augment the accumulation buffer with a linear term that stores the value of Equation 2.6. The modification to our algorithm is simply that, after processing a pixel, we add the linear accumulation term to the constant accumulation term. This process is easily extended to general polynomial color functions, from which a quadratic term is accumulated into a linear term, and so on.

2.2.2 Implementation

We have described a general framework for a rasterization algorithm. The key idea is that the equations we use for rasterization naturally lead to efficient evaluation by rasterizing in scanline order. Rasterization using our method is easily parallelizable. Cutting all of the curves to pixels is done as a batch process prior to rasterization, and curves

are distributed between processors. Each scanline is then evaluated in parallel, as is each polynomial piece of a filter. Note that this parallel implementation processes each curve a number of times equal to the area of support of the filter. It is also possible to write an efficient serial implementation that scans over the entire image once but maintains an array of accumulation values, one for each polynomial piece, and processes each curve once.

2.3 Results

Closed-form solutions are available for a wide variety of interesting filters and curves, and we show that high-order filters reduce aliasing artifacts in several test images. We also compare our solution to other rasterizers to show that calculating closed-form solutions of the rasterization equation is competitive in speed with approximate solutions and generates higher-quality images.

The advantages of using high-order filters are most apparent in images with high-frequency details. We show rasterizations of test patterns that are prone to aliasing in Figures 2.5 and 2.6, where we rasterize the images using different filters. Figure 2.5 shows a perspective projection of lines on a plane that are at a frequency of 1/4 lines per pixel at the bottom to 2 lines per pixel at the top. This figure illustrates the difference between point sampling, supersampling with a GPU (ATI Radeon HD 5700), and analytically rasterized images. Image (a) is point sampled and aliasing is clearly visible. Both (b) and (c) were sampled using a tent filter, but (b) is supersampled with 16 points per pixel and has obvious aliasing, whereas exact evaluation of (c) using our method suppresses most aliasing.

Figure 2.6 shows the differences between analytically evaluated filters of increasing quality. The top row shows examples using linear curves, while the bottom shapes are made of quadratic curves. As the order of the filter increases, the transition between detailed and blurred regions becomes sharper, and aliasing is reduced. The box filter (a) is clearly the worst because it does not remove high frequencies very well, which results in

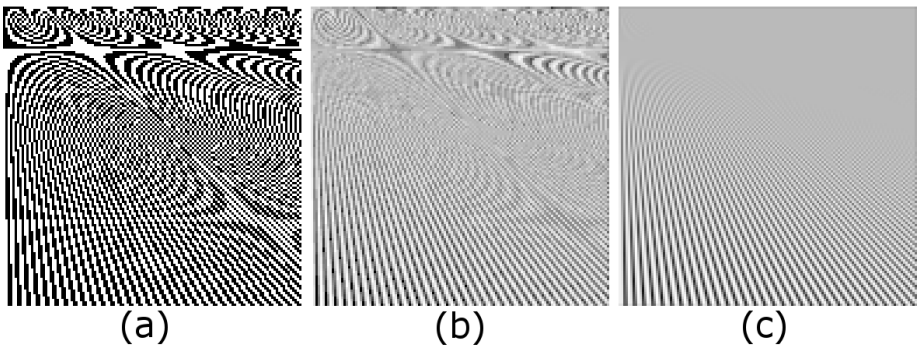


Figure 2.5: Examples of (a) point sampling, (b) 16x MSAA tent filtering using an ATI Radeon HD 5700, (c) analytic tent filtering.

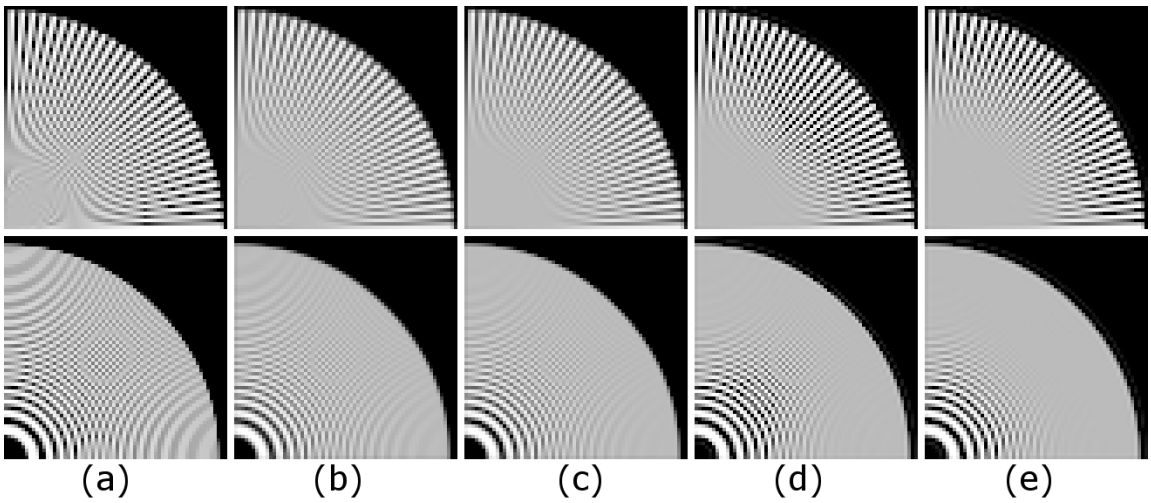


Figure 2.6: Analytic rasterizations using different filters. The first row uses radial triangles and the second row uses rings of quadratic curves. The columns show the filters: (a) box, (b) tent, (c) Mitchell-Netravali, (d) Lánczos 3, and (e) jinc 3.

	Fig. 2.5	Fig. 2.6 (top)	Fig. 2.6 (bottom)
AGG	2.73	0.25	**0.35**
Cairo	5.35	0.57	**2.34**
Wavelet	3.84	0.42	3.14
Box	1.23 *(0.60)*	0.14 *(0.07)*	0.37 *(0.17)*
Tent	2.36 *(0.83)*	0.27 *(0.12)*	0.75 *(0.26)*
Q. B-spline	9.60 *(3.29)*	1.02 *(0.31)*	4.45 *(1.26)*
Mitchell	21.2 *(6.90)*	2.26 *(0.79)*	37.0 *(11.4)*
Lánczos 2	26.3 *(7.64)*	2.71 *(0.96)*	27.6 *(7.54)*
Lánczos 3	48.7 *(16.0)*	5.37 *(1.88)*	115. *(37.7)*
Radial 2	21.2 *(5.70)*	2.31 *(0.78)*	17.1 *(5.02)*
Radial 3	42.9 *(14.1)*	4.64 *(1.68)*	45.8 *(15.6)*

Table 2.1: Time required to rasterize various images, measured in milliseconds. AGG, Cairo, and wavelet rasterization all use a box filter. Serial times are followed by the times needed using four parallel processors (in parentheses).

obvious aliasing patterns. The tent filter (b) is noticeably better than the box filter. Although some high frequencies still pass in the top image, the center of the circle is much closer to a uniform color. A disadvantage of the tent filter is that low frequencies are attenuated too much, which is visible as blurriness. The Mitchell-Netravali filter (c) appears slightly better than the tent filter. The Lánczos (d) and jinc (e) filters both have a radius of 3, but (e) is the only filter that is not a tensor product. In (d), one can see the square fall-off of aliasing in the bottom picture compared to the circular fall-off in (e).

Our method is not restricted to polynomial curves and can theoretically be applied to any boundary defined by parametric curves. Closed-form expressions are not guaranteed for all curves, but they do exist for rational quadratic Bézier curves. This represents an important class of curves that includes all circular and elliptical arcs. We show an Apollonian gasket composed of rational quadratic Bézier curves that we evaluate exactly using a box filter in Figure 2.7.

We present rasterization times for the patterns in Figure 2.5 (128^2 pixels) and Fig-

Figure 2.7: Rasterization of rational Bézier curves of varying sizes in a 128×64 pixel image.

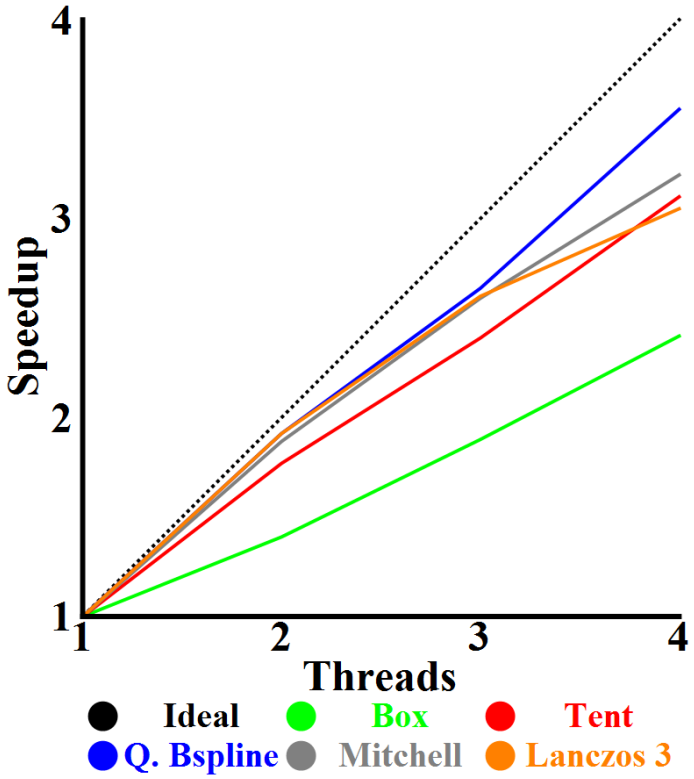


Figure 2.8: Speedup for different numbers of threads on a four core Intel Core i7 870. The ideal speedup is shown as a dashed black line, and the speedup for different filters is shown as solid lines of different colors.

ure 2.6 (64^2 pixels) in Table 2.1. Figure 2.5 contains 1024 lines in 256 quads, Figure 2.6 (top) contains 96 lines in 32 triangles, and Figure 2.6 (bottom) contains 32 concentric rings made out of a total of 128 non-rational quadratic curves and 64 radial lines. We ran tests on an Intel Core i7 870. The first row shows the times taken by Anti-Grain Geometry (AGG), which is a highly optimized, high-quality software rasterizer. The second gives times for Cairo, which is a rasterizer used in several large software projects. Both AGG and Cairo use box filtering and approximate curved boundaries by polygons, for which we used their default tolerances to subdivide curves. The third row contains times for wavelet rasterization (Chapter 3), which analytically rasterizes piecewise polynomial curves using a box filter. The remaining times were measured for our method using different filters. We use a serial implementation of our algorithm to compare with the other methods and include times for our parallel implementation running on four processors in parentheses. The Mitchell-Netravali and Lánczos 2 filters are both cubic tensor-product filters, but they have a different number of zero terms. The Radial 2 filter has the same support but has fewer coefficients compared to the cubic tensor-product filters because we use a filter of cubic total degree.

The methods that we compare against ours rasterize polygons using a box filter, so our box filter produces the same output as they do for Figure 2.5 and Figure 2.6 (top). AGG and Cairo approximate quadratic and cubic curves as polygons, so we show the time for these approximate rasterizations in bold. The times required to rasterize shapes using AGG and our method are similar, even though we compute an analytic rasterization of quadratic curves rather than a polygonal approximation. Cairo is slower than AGG and our method because it is more generic and is designed for more-complicated rendering operations.

Unlike the other three methods, our technique can analytically filter images using higher-order filters. As the order of the filter increases, so does the computation time, which is approximately linear in the area sampled by the filter. The tent filter is interesting

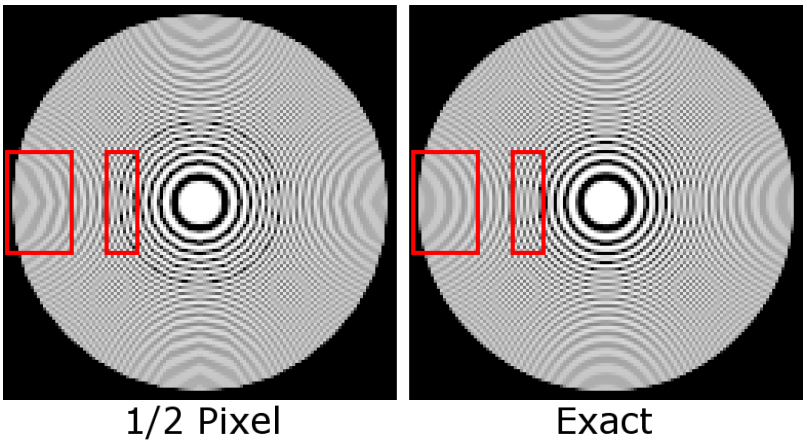


Figure 2.9: In the image on the left, curves are approximated by polygons within a tolerance of $\frac{1}{2}$ of a pixel, while the image on the right is calculated exactly.

because it provides a significant improvement in image quality but takes only twice as long to compute as a box filter and remains competitive in time with AGG, which uses a box filter. High-order filters are expensive to compute, but may be acceptably fast for final production renderings.

Figure 2.8 shows the speedups we achieve rendering Figure 2.13 with a parallel implementation of our algorithm using OpenMP to split calculations between different numbers of threads on a four-core system. The image contains 1,577 cubic curves that we rasterized into a 316×613 pixel grid. Speedups vary based on filter width. Some filters, like the box filter, are simple enough that the overhead of parallelization prevents ideal scaling. Nevertheless, with four cores, we still achieve a 2.4$\times$ speedup. As the size of the filter increases, Equation 2.3 requires more operations, and our speedup approaches the perfect scaling relationship. In the case of a quadratic B-spline filter, parallel processing achieves a 3.5$\times$ speedup over our serial implementation.

A common approach to handling curved boundaries is to approximate curves by line segments. However, this form of approximation produces its own aliasing artifacts, even

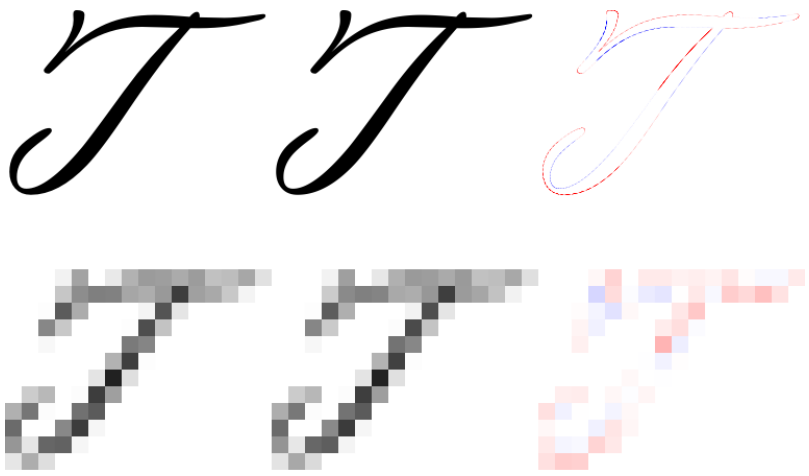


Figure 2.10: Comparison of font rendering between FreeType (left) and our Wavelet algorithm (center). In the difference image (right), red values indicate that our rendering has a higher cell coverage and blue indicates that it has a lower cell coverage. Differences are multiplied by a factor of 10 to increase visibility.

during analytic filtering. For example, the images in Figure 2.9 are composed of quadratic curves. On the left, we subdivided curves into line segments so that they are within 0.5 pixels of the actual curve. On the right, we show curves that are rasterized with exact formulas for quadratics. In both cases, we used our method for analytic rasterization so that the differences are only due to approximating curves by line segments. Notice that a black ring appears in the left image and that the aliasing patterns appear polygonal rather than round. Also, topological problems can occur when curves are subdivided into line segments because the line segments approximating those curves may intersect even if input curves they represent do not intersect. Determining the level of subdivision required to prevent line segments from intersecting can be complicated and computationally expensive.

For polynomial boundaries such as those found in fonts and vector graphics, we calculate the occupancy of pixels analytically rather than divide the curve into dense collections

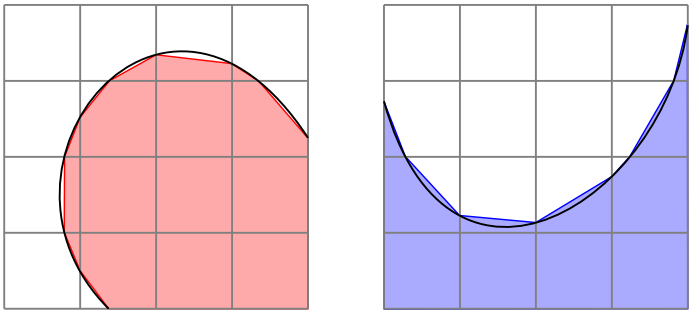


Figure 2.11: Approximation of Bézier curves by line segments. This method introduces error; even when a line segment is used for each pixel the curve intersects. The result is that approximation underestimates coverage in convex regions (left) and overestimates coverage in concave regions (right).

of line segments. Figure 2.10 shows a comparison between our output and the output of FreeType. We rasterized an upper case "T" with both methods at $256pt$ and $16pt$ sizes. Blue pixels indicate that our image had lower occupancy and red pixels indicate we had higher occupancy in the difference images. Notice that FreeType overestimates occupancies in regions of negative curvature and underestimates occupancies in regions of positive curvature. This artifact is primarily an effect of the bias introduced by linear approximation, as shown in Figure 2.11.

We show examples of SVG files that contain cubic curves in Figures 2.1, 2.12, and 2.13. Figure 2.1 shows a vector image that is scaled by irrational values. There are 60 butterflies, and each butterfly contains 452 cubic curves that form two colored shapes, so there are 27,120 cubic curves in total. Figure 2.12 shows an example of an icon, made from 108 cubic curves, that incorporates linear gradients and text. We show the input vector image on the top left and show a 64^2 pixel rasterization from Inkscape and from our implementation of box, tent, quadratic B-spline, and Mitchell-Netravali filters. Note that many renderers, including Adobe Acrobat, have problems rendering the curved boundary between the linear color gradients correctly. This problem is visible in the Inkscape image

Figure 2.12: A vector graphic image of an icon with linear gradients and cubic Bézier curves. Above, we show the vector image with rasterizations at 64^2 resolution using box, tent, quadratic B-spline, and Mitchell-Netravali filters below. Inkscape evaluates a box filter but does not properly handle blending between neighboring regions, which results in the appearance of white lines.

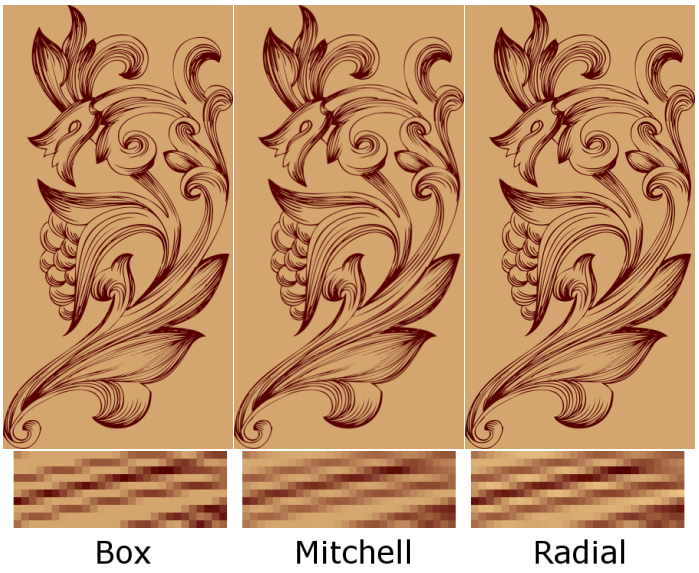


Figure 2.13: An image rasterized using a box filter (left), a Mitchell-Netravali filter (center), and a radial filter of radius 3 (right). A zoomed-in section of the image is shown below each high-resolution image to show differences in pixel values.

as a bright halo. Hence, even the vector input may appear to have artifacts depending on the viewing software. Finally, Figure 2.13 shows the effect of different filters on a detailed vector image, without Moiré artifacts, that is made of 1,577 cubic curves.

2.4 Conclusions and Future Work

With the processing power of video cards and the variety of display devices increasing in tandem, accurate rasterization of shapes is both more desirable and more achievable. We revisited the theory behind rasterization and described a new approach for finding analytic solutions for pixel colors using a variety of filters. We then used that theory to develop a method that rasterizes several types of curves that are commonly used in vector graphics.

Rasterization is a complex subject and remains an interesting topic for research. Several extensions of our method may be useful. The curves and color gradients described in this chapter are closed under affine transformations, but rasterization of three-dimensional scenes adds complicated occlusions and requires a projective transformation. For example, Gouraud shading defines linear color functions over triangles, but Gouraud-shaded triangles have color functions that are rational with respect to screen coordinates after projection. Closed-form expressions of Equation 2.1 may exist for rational functions, but the expressions are certainly not polynomial. Rasterization of textured triangles is even more difficult. Our method may also apply to higher dimensions by, for example, calculating motion blur by integrating over a three-dimensional domain.

3. WAVELET RASTERIZATION*

An alternative way to think about rasterization is to say that rasterizing an image calculates the coefficients of functions in a finite basis, where the basis functions are called the reconstruction filter. As discussed in Section 1.1, coefficients are usually calculated by an inner product of the image with a sampling function, but one can also solve for the coefficients by projecting the image onto the finite basis by a least squares minimization. By this definition, we can also rasterize an image by projecting onto a wavelet basis.

In particular, we are interested in rasterizing onto the Haar basis, which is equivalent to rasterizing using a box filter. The advantages of using an intermediate wavelet representation are that we can progressively add details to a rasterized image, the algorithm is more robust to bad input, and the method generalizes into higher dimensions. In three dimensions, rasterizing into a wavelet basis evaluates coefficients directly into an octree, which means the method is efficient in time and space.

Because we are effectively applying a box filter, we can view pixels as square regions on a display. In this interpretation, pixels that intersect a the boundary of a polygon are partially covered. Rasterizing polygons into pixels that store percent occupancy rather than Boolean inside/outside information is equivalent to sampling χ_M convolved with a box filter. Although box filters are considered to be poor approximations of the sinc filter, their interpretation of calculating the occupancy of cells is sometimes useful, especially in 3D.

We typically think of rasterization as a 2D problem, but we can extend the idea to 3D. In 3D, the equivalent operation is to calculate the occupancies of cubic cells from the trian-

*Reprinted with permission from "Wavelet Rasterization" by Josiah Manson and Scott Schaefer, 2011. Computer Graphics Forum, 30 (2), 395–404, © 2011 The Author(s) Journal compilation © 2011 The Eurographics Association and Blackwell Publishing Ltd.

Figure 3.1: Slices from a 3D rasterization of the Happy Buddha statue computed on a 64^3 grid to illustrate the anti-aliased nature of wavelet rasterization.

gles that enclose a volume. An example of such a volumetric rasterization (voxelization) is shown in Figure 3.1, where slices through the rasterized volume of a statue of Buddha are shown next to a triangle mesh of the same statue. Voxelizing an object builds an implicit representation that is useful for operations such as collision detection, constructive solid geometry (CSG), and fluid simulation, which are easier to calculate over volumes than boundaries.

Collision detection determines if objects interpenetrate and precomputed voxels can accelerate the solution by querying if points are definitely inside (one), definitely outside (zero), or near (fractional values) the boundary of an object and need further testing. The inside/outside values can be stored in an octree to accelerate queries for static objects. If the cell containing a test point has a fractional occupancy, only the part of the boundary that intersects that cell must be tested. Because the queries that require further checking are guaranteed to be within a voxel of the boundary, checks against the triangle mesh can be calculated more efficiently than if the queries are far from the mesh. In other words, voxel checks are fast when far from the mesh, and triangle checks are fast when close to the mesh, so combining the two methods is fast at all distances.

Implicit representations also provide a natural method of calculating CSG set opera-

tions. We can approximate set operations between two objects a and b by $min(a, b)$ for intersection, $max(a, b)$ for union, and $min(a, 1 - b)$ for difference. We can then convert the result back to a boundary representation as shown in Figure 3.4. For more information on how to calculate a contour from a rasterized volume, see Chapter 5. Alternatively, we can accelerate exact CSG algorithms by using implicit representations of objects to quickly classify surface elements of a mesh as being completely inside or outside each object. We can then perform exact intersection tests within the remaining indeterminate voxels [19].

In fluid simulations, it is natural to model surface tension using a surface mesh, whereas pressure and advection are best computed over a volumetric grid [126]. Thus, it might be useful to be able to convert freely between voxel and boundary representations of a fluid volume. Rasterization converts boundary meshes to voxels, whereas contouring methods like the one described in Chapter 5 convert back to a triangle mesh. Efficient methods for voxelizing surfaces can accelerate fluid simulations with air-water interfaces, and accurate cell occupancies can help remove inaccuracies in the simulation.

We calculate the exact wavelet coefficients of rasterized polygons, fonts, and volumes. To compute the coefficients efficiently, we transform integrals over the interior of an object to integrals over the boundary of the object. There are many integrals that we could use in this transform, so we choose integrals that have the smallest support and computational cost. The result is a fast algorithm for rasterizing 2D shapes and implicitizing 3D volumes. Furthermore, our algorithm is independent of surface connectivity and is robust to degenerate inputs and small gaps or overlaps in the surface.

3.1 Related Work

Many of the methods used to rasterize 2D images do not extend well to 3D volumes. One method that directly extends is ray-casting. For each point in the grid, we cast a ray

and count the number of intersections between the ray and the boundary [93, 125]. If the number of intersections is odd, we classify the point as interior. Otherwise, the point is exterior. This approach is useful because it does not depend on the orientation of the boundary, but ray-intersections are difficult to handle robustly in 3D.

One of the first analytic box-filter rasterizers [24] clips polygons in a scene to each pixel and then clips polygons against each other, ordered by depth. Areas of clipped polygons are then added to find the color of the pixel. This method does not extend well into 3D, however, because it requires that triangles in a mesh are topologically connected and cannot deal with holes or T-junctions. Duff developed a scan converter [48] that calculates analytic convolutions with cubic splines. Similar to our method, Duff clips only the edges of polygons to pixels. The main advantage of our approach over Duff's algorithm is robustness to small imperfections in input that cause errors to propagate across a line in scan-line algorithms. This property is more important in 3D than in 2D because 3D data are more likely to contain imperfections. A method of extruding box splines to filter triangles has also been developed [98], but requires simplicial decompositions of shapes.

Several algorithms have been specifically designed to voxelize boundary representations of objects. Some computationally expensive algorithms [55, 23] sample individual points. Other algorithms [53, 71, 46, 50, 139, 51] use the special-purpose hardware in a GPU to accelerate volumetric rasterization. However, these algorithms use no filtering, and therefore create obvious aliasing. Binary volumes are adequate for some applications, like collision detection, but CSG operations and other methods that extract surfaces from a volumetric representation require anti-aliasing to produce attractive surfaces. Although super-sampling approximates the anti-aliased representation of these binary volumes, super-sampling volumes is extremely time-consuming. Other methods [79] approximate the signed distance function of a surface, but these often rely on finding closest points on the surface [7], which is expensive to compute for points far from the surface.

Some researchers [123, 124] have used the GPU to accelerate the computation of signed distance functions, but those methods are still slow because of the complexity of evaluating the distance function.

Wavelets have also been applied to rendering ray-traced scenes, but in a way that is very different from our method. Overbeck et al. [107] use ray-traced color samples in the image plane to build wavelet coefficients of the image and then use the variance of the mean of wavelet coefficients to determine where more samples are required. Additionally, they use smooth basis functions with larger supports and reduce the contribution of coefficients with high variance so that noisy regions of the image (for example, around out-of-focus objects, in soft shadows, or on semi-glossy surfaces) are smoothed. In contrast, our method uses wavelets to calculate the interiors of 3D surfaces and 2D polygons.

3.2 Rasterizing into the Wavelet Basis

Wavelets provide a basis for representing functions through a hierarchy of localized refinements. They have a wide variety of applications, from solving differential equations to digital image processing, signal processing, and surface reconstruction. The main advantage of wavelets over other representations of a function is that wavelets are localized in both the spatial and frequency domains. Our derivation of the rasterization equation is similar to Section 2.2, but it is specialized to wavelets. Although the derivation is similar, the orthogonality and hierarchical nature of the wavelet basis leads to an entirely different rasterization algorithm.

First, I will briefly review some properties of wavelets necessary for our construction. Let ϕ be a compactly supported univariate scaling function, with orthogonal shifts, that satisfies the two-scale relation

$$\phi(t) = \sum_{\ell \in \mathbb{Z}} \alpha_\ell \phi(2t - \ell), \tag{3.1}$$

in which only a finite number of coefficients α_ℓ are nonzero. Let ψ be the univariate wavelet function with compact support that is obtained from ϕ by multiresolution. The formula for ψ is

$$\psi(t) = \sum_{\ell \in \mathbb{Z}} (-1)^\ell \overline{\alpha_{1-\ell}} \phi(2t - \ell), \tag{3.2}$$

where $\overline{\alpha_{1-\ell}}$ denotes the complex conjugate of $\alpha_{1-\ell}$. Examples of such wavelets and scaling functions are given by Daubechies [38]. We can use any orthogonal basis, each of which offers a trade-off among support, smoothness, symmetry, and ease of computation. Unlike more complex wavelets, Haar wavelets [68] have small support and analytic functions. Specifically, the scaling function

$$\phi(t) = \begin{cases} 1, & 0 \leq t < 1 \\ 0, & otherwise \end{cases}$$

generates the Haar basis, and ψ is given by

$$\psi(t) = \phi(2t) - \phi(2t - 1).$$

The 2D Haar wavelets shown in Figure 3.2 exactly represent piecewise-constant functions made of squares and represent a box-filtered sampling of an image.

Two-dimensional wavelets are constructed as follows. Let $\psi^0 = \phi$, $\psi^1 = \psi$, E' be the set of vertices $\{(0,0),(0,1),(1,0),(1,1)\}$, and $E = E' \backslash \{(0,0)\}$. For each $e = (e_x, e_y) \in E'$, $j \in \mathbb{N}$, and $\mathbf{k} = (k_x, k_y) \in \mathbb{Z}^2$, we define

$$\psi^e_{j,\mathbf{k}}(p) = 2^j \psi^{e_x}(2^j p_x - k_x) \psi^{e_y}(2^j p_y - k_y),$$

where $p = (p_x, p_y)$. Every function g that is locally integrable on $\mathbb{R}^2$ has the wavelet

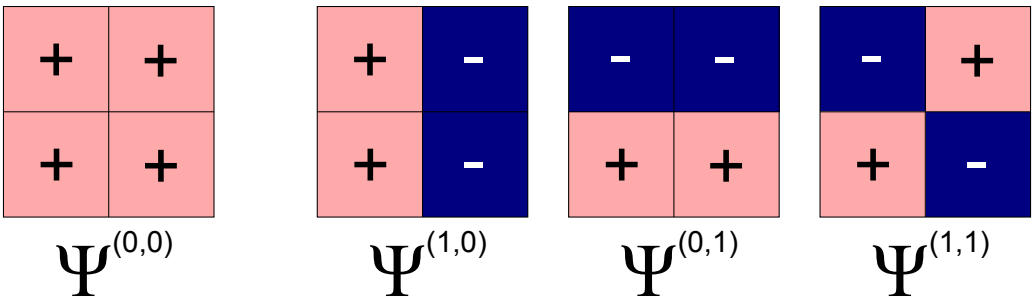


Figure 3.2: The 2D Haar basis functions. Each function is shown over the domain $[0,1)^2$ and is piecewise constant ($-1/+1$) on each quadrant. Translations of $\bar{\Psi}^{(0,0)}$ give the low-resolution representation of the function, while scalings and translations of the functions $\bar{\Psi}^{(1,0)}$, $\bar{\Psi}^{(0,1)}$, and $\bar{\Psi}^{(1,1)}$ add high-resolution details.

expansion

$$g(p) = \sum_{\mathbf{k}\in\mathbb{Z}^2} c^{(0,0)}_{0,\mathbf{k}} \psi^{(0,0)}_{0,\mathbf{k}}(p) + \sum_{j\in\mathbb{N}} \sum_{\mathbf{k}\in\mathbb{Z}^2} \sum_{e\in E} c^{e}_{j,\mathbf{k}} \psi^{e}_{j,\mathbf{k}}(p), \tag{3.3}$$

The three-dimensional construction follows the same pattern. Let E' denote the set of vertices of the cube $[0,1]^3$ and let E denote the set of vertices excluding the origin (i.e. $E = E' \setminus (0,0,0)$). For each $e = (e_x, e_y, e_z) \in E'$, $j \in \mathbb{N}$ and $\mathbf{k} = (k_x, k_y, k_z)$, we define

$$\psi^{e}_{j,\mathbf{k}}(p) = 2^{3j/2} \psi^{e_1}(2^j p_x - k_x) \psi^{e_2}(2^j p_y - k_x) \psi^{e_3}(2^j p_z - k_z),$$

where $p = (p_x, p_y, p_z)$. Each function f that is locally integrable on $\mathbb{R}^3$ has the wavelet expansion

$$f(p) = \sum_{\mathbf{k}\in\mathbb{Z}^3} c^{(0,0,0)}_{0,\mathbf{k}} \psi^{(0,0,0)}_{j,\mathbf{k}}(p) + \sum_{j\in\mathbb{N}} \sum_{\mathbf{k}\in\mathbb{Z}^3} \sum_{e\in E} c^{e}_{j,\mathbf{k}} \psi^{e}_{j,\mathbf{k}}(p), \tag{3.4}$$

where each $c^{e}_{j,\mathbf{k}}$ is given by

$$c^{e}_{j,\mathbf{k}} = \int_{\mathbb{R}^3} f(p) \psi^{e}_{j,\mathbf{k}}(p)\, dp. \tag{3.5}$$

3.2.1 Evaluating Wavelet Coefficients

We wish to rasterize objects by calculating the percent occupancy of voxels in a regular grid. If M is the set of points inside an object with boundary ∂M, represented as a set of edges, then the indicator function χ_M is defined as

$$\chi_M(p) = \begin{cases} 1, & p \in M \\ 0, & otherwise. \end{cases} \tag{3.6}$$

This function implicitly represents the shape M and we extract the boundary of the shape by finding the points at which χ_M transitions from zero to one. In particular, χ_M defines the set of points that would be drawn if the polygon ∂M was rasterized at infinite resolution. Taken to the limit, summing super-samples [45] over a pixel is equivalent to applying a box-filter or integrating χ_M over the pixel area. The value of a pixel P is therefore given by

$$\frac{\int_P \chi_M(p)\, dp}{\int_P dp}. \tag{3.7}$$

This equation shows that the value of a pixel is equal to the area of the polygon that intersects the pixel divided by the area of the pixel. Our approach to rasterizing polygons is to calculate the wavelet coefficients of χ_M to pixel resolution and then to invert the wavelet transform to convert the rasterization to a box-filtered image.

Here, the index e indicates which basis function is used, and k denotes its translation at resolution j. Because the functions have supports that are power-of-two contractions of a square, the wavelet hierarchy in 2D is naturally represented by a quadtree. Note that j controls the resolution of the wavelet expansion and that we truncate j to stop at pixel resolution.

If we consider the wavelet coefficients of χ_M and use the definition of χ_M from Equa-

tion 3.6, then Equation 3.5 reduces to

$$c^e_{j,k} = \iint_M \psi^e_{j,k}(p)\, dp. \tag{3.8}$$

We use the divergence theorem from Equation 2.2 to relate the integral over M to an integral over its boundary ∂M. By finding functions $F^e_{j,k}$ that satisfy $\nabla \cdot F^e_{j,k} = \psi^e_{j,k}$, we can calculate the wavelet coefficients of χ_M using only the polygon boundary in the line integrals

$$c^e_{j,k} = \sum_i \int_0^1 F^e_{j,k}(P_i(t)) \cdot n(P_i(t)) \| P_i'(t) \| dt, \tag{3.9}$$

where P_i represents the i^{th} polynomial segment of the boundary (line segment for polygons).

Because χ_M is constant over both the interior and exterior of the shape and because wavelets have constant precision, wavelet coefficients are non-zero only where the boundary intersects the basis functions. This property yields an adaptive quadtree that is refined only along the boundary of the polygon.

3.2.2 2D Boundaries

We make a few simplifying assumptions to facilitate analysis. First we calculate basis functions over the $[0, 1)^2$ domain (the support of the 2D Haar basis) by translating the input edges by $-k$ and scaling by 2^j. Furthermore, we clip edges to this domain because the support of the wavelet functions is only $[0, 1)^2$. Note that we only need to clip the boundary edges rather than the polygons themselves because we calculate a boundary integral. This simplification allows us to drop the j, k subscripts so that

$$\psi^e(p) = \psi^{e_x}(p_x)\psi^{e_y}(p_y).$$

There is a continuum of functions F^e in 2D that satisfy $\nabla \cdot F^e(p) = \psi^e(p)$ and that are parameterized by α such that

$$F^e(p) = \left(\alpha \bar{\Psi}^{e_x}(p_x)\psi^{e_y}(p_y), (1-\alpha)\psi^{e_x}(p_x)\bar{\Psi}^{e_y}(p_y)\right),$$

where

$$\bar{\Psi}^{\ell}(t) = \int_0^t \psi^{\ell}(s)\, ds$$

and $\ell \in \{0, 1\}$. Not all choices of α yield practical solutions or efficient calculations. In the following sections, we show how to choose α such that Equation 3.9 yields a calculation that has both small support and low computational cost.

The $c^{(0,0)}$ coefficient is special because it exists only for the root node of the quadtree and gives a value that is refined by all other wavelet coefficients. The $c^{(0,0)}$ coefficient also has the clear geometric interpretation of being the area of the polygon. Letting $\alpha = 1/2$ in Equation 3.9 yields

$$\int_0^1 F^{(0,0)}(P(t)) \cdot n(P(t))\|P'(t)\|dt = \frac{1}{2} det\left(v_0, v_1\right),$$

where v_0 and v_1 are the end-points of the edge defined by P. Adding determinants of edges is equivalent to adding the areas of the triangles formed between the edges and the origin. Adding these signed areas computes the area of a polygon.

The three coefficients other than $c^{(0,0)}$ calculate the difference between a cell and its sub-cells at the next-higher level of resolution. These three refinement coefficients, in addition to the known scale coefficient, uniquely determine values of all four sub-cells. Consider the $c^{(1,0)}$ coefficient ($c^{(0,1)}$ follows in a similar manner). Again, we could choose $\alpha = 1/2$ to give

$$F^{(1,0)}(p) = \tfrac{1}{2}(\bar{\Psi}(p_x)\phi(p_y), \psi(p_x)\bar{\Phi}(p_y)),$$

where $\bar{\Psi} = \bar{\Psi}^1$ and $\bar{\Phi} = \bar{\Psi}^0$. Although this function satisfies the divergence theorem, $F^{(1,0)}$ has infinite support because $\bar{\Phi}$ has support of $[0, \infty)$. We want to use only detail functions with finite support such that we limit the number of edges that influence a coefficient. Notice, however, that the support of $\bar{\Psi}$ is finite and is $[0, 1)$. By choosing $\alpha = 1$ for $F^{(1,0)}$ and $\alpha = 0$ for $F^{(0,1)}$, we obtain the compactly supported functions

$$\begin{aligned} F^{(1,0)}(p) &= (\bar{\Psi}(p_x), 0) \\ F^{(0,1)}(p) &= (0, \bar{\Psi}(p_y)). \end{aligned}$$

For the last function $F^{(1,1)}$, any value of α will have compact support. The trade-off is that using $\alpha = 1/2$ gives greater numerical stability such that

$$F^{(1,1)}(p) = \frac{1}{2}(\bar{\Psi}(p_x)\psi(p_y), \psi(p_x)\bar{\Psi}(p_y)),$$

whereas a value of $\alpha = 1$ is faster to compute

$$F^{(1,1)}(p) = (\bar{\Psi}(p_x)\psi(p_y), 0).$$

The functions F^e are piecewise-linear because $\bar{\Psi}$ is piecewise-linear, with each quadrant being linear, so we evaluate these integrals by splitting each polygon edge P_i into the quadrants and transforming the domains of the quadrants to be $[0, 1)^2$. We can then add the contribution of each split edge to the coefficients. Quadrants that do not contain an edge contribute nothing. Note that wavelet coefficients are most naturally calculated and stored in a quadtree. This means that if we calculate the coefficients of the root node that covers the entire image domain, we can reuse the edge splits that we calculated to calculate the coefficients of basis functions in each of the quadrants recursively. Therefore, an efficient algorithm is to process each edge of a polygon independently and to calculate all of the

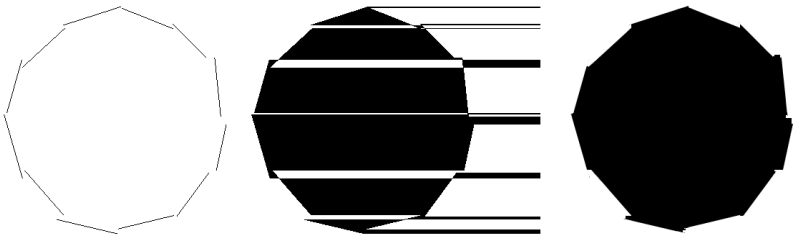


Figure 3.3: Rasterizations of a polygon made of disconnected edges (left) using a standard scanline rasterizer (center) and our wavelet rasterizer (right). Wavelets localize errors because of their local support.

wavelet coefficients that intersect that edge. In the same way that we found closed-form solutions for Bézier boundaries in Section 2.2, we can find closed-form solutions of Haar wavelet coefficients for polynomial curve boundaries.

3.2.3 3D Triangle Surfaces

In 3D, we assume the boundary ∂M of our volume M is composed of triangles. As in 2D, we translate and scale input triangles rather than modify the basis functions and then clip transformed triangles to the unit cube. We therefore restrict our discussion to the normalized basis

$$\psi^e(p) = \psi^{e_x}(p_x)\psi^{e_y}(p_y)\psi^{e_z}(p_z).$$

To satisfy Equation 2.2, we find functions F^e whose divergences are equal to ψ^e. In 3D, these functions are parameterized by α and β such that

$$F^e(p) = \begin{pmatrix} \alpha\bar{\Psi}^{e_x}(p_x)\psi^{e_y}(p_y)\psi^{e_z}(p_z) \\ \beta\psi^{e_x}(p_x)\bar{\Psi}^{e_y}(p_y)\psi^{e_z}(p_z) \\ (1-\alpha-\beta)\psi^{e_x}(p_x)\psi^{e_y}(p_y)\bar{\Psi}^{e_z}(p_z) \end{pmatrix}.$$

We choose functions that have as small a support as possible and that are as efficient to

compute as possible, yielding

$$
\begin{aligned}
F^{(0,0,0)}(p) &= \frac{1}{3}(\bar{\Phi}(p_x), \bar{\Phi}(p_y), \bar{\Phi}(p_z)) \\
F^{(1,0,0)}(p) &= (\bar{\Psi}(p_x), 0, 0) \\
F^{(0,1,0)}(p) &= (0, \bar{\Psi}(p_y), 0) \\
F^{(0,0,1)}(p) &= (0, 0, \bar{\Psi}(p_z)) \\
F^{(1,1,0)}(p) &= (\bar{\Psi}(p_x)\psi(p_y), 0, 0) \\
F^{(1,0,1)}(p) &= (\bar{\Psi}(p_x)\psi(p_z), 0, 0) \\
F^{(0,1,1)}(p) &= (0, \bar{\Psi}(p_y)\psi(p_z), 0) \\
F^{(1,1,1)}(p) &= (\bar{\Psi}(p_x)\psi(p_y)\psi(p_z), 0, 0).
\end{aligned}
$$

Again, the wavelet detail functions (functions 2-8 above) have finite support, although the scale function ($F^{(0,0,0)}$) does not, but there is only one top-level scale function corresponding to the root node of the octree. Also, as in 2D, the symmetric solution $\alpha = \beta = 1/3$ for the $c^{(0,0,0)}$ coefficient gives the determinant of the triangle. This is equivalent to the signed volume of the tetrahedron formed between the triangle and the origin, giving

$$
c^{(0,0,0)} = \int_{p \in T} F^{(0,0,0)}(p) \cdot n \; d\sigma = \frac{1}{6} det(v_0, v_1, v_2).
$$

We compute the remaining coefficients as in 2D by splitting triangles with vertices (v_0, v_1, v_2) into octants that are labeled $Q_{i,j,k}$. As in 2D, some of the functions have compact support for other values of α and β that are more robust to noise, but we choose only the most compact and computationally efficient forms.

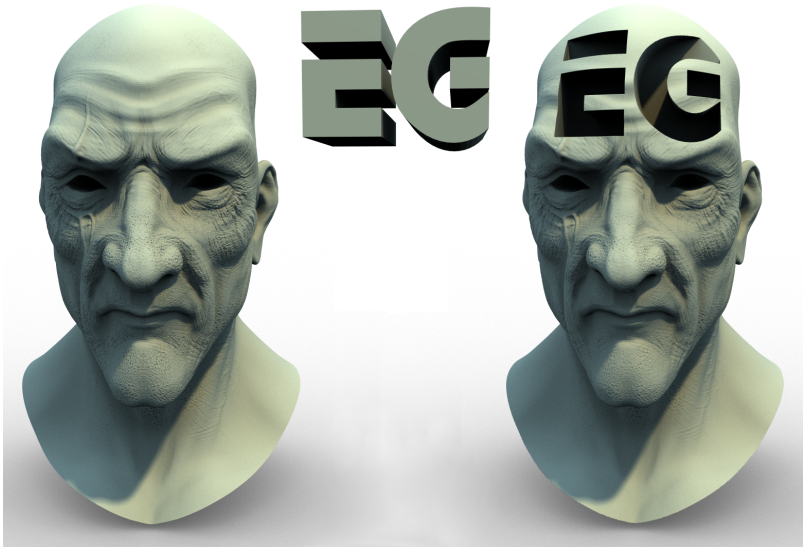


Figure 3.4: Image obtained when we used our method to calculate the CSG set difference between the head and the Eurographics logo using an anti-aliased rasterization of each model on a 1024^3 grid.

3.3 Results

Quality and robustness are strong points in favor of wavelet rasterization. Wavelets build a low-resolution image that is subsequently refined in localized areas, which means that the overall picture is retained even in the presence of degeneracies and holes. Small errors in the polygon can have a large effect, as shown in the example of the non-closed polygon in Figure 3.3, because a scanline rasterizer with an even-odd fill rule propagates information only from the current line to the right. In contrast, our wavelet rasterizer uses oriented edges in both the x and y directions to refine a coarse image locally. Although it is difficult to define the correct rasterization of a non-closed polygon, wavelet rasterization localizes rasterization errors and produces a plausible image.

Wavelet rasterization is particularly useful for rasterizing volumes of triangle meshes. Table 3.1 shows the times it took to rasterize triangle meshes of increasing complexity.

The highest resolution mesh we use is a reconstruction of Michelangelo's statue of David, which contains 7.2 million triangles that were rasterized at a resolution of 4096^3. Storing one byte per voxel consumes 64 GB of space at this resolution, but our adaptive octree stores the entire function in memory because the tree is only refined around the boundary of the surface. The majority of time is spent computing the wavelet coefficients, and the time required for this computation is proportional to the surface area of the object times tree depth. However, the time required to compute the function values from these coefficients (i.e. synthesis) over a uniform grid is proportional to the volume enclosing the object and grows quickly as the resolution increases.

We compare the 2D rasterization performance of our algorithm against other freely available, high-quality scanline rasterizers. Specifically, we compare our polygon rasterization on a Core i7 960 against Anti-Grain Geometry (AGG), which is an open-source, highly-optimized software rasterizer, and we use native GPU rasterization on an Nvidia 8800GT. For font rasterization, we compare against a high-quality, open-source font rasterizer called FreeType. Even though our algorithm is relatively efficient in terms of computation, we cannot compete with the speed of native hardware or even highly optimized software implementations with assembly tuning. For complex shapes, the speed of our algorithm is about a factor of three slower than these optimized implementations. We rasterized a circle with a million vertices at 1024^2 resolution in 50.2 ms on the GPU with 16xQ anti-aliasing, 36.8 ms with AGG, and 107 ms with our method.

It is difficult to demonstrate anti-aliasing in a volumetric image, but we have endeavored to do so in Figure 3.1. This figure shows slices through the volume of the Happy Buddha statue. Notice that the silhouette of each slice is anti-aliased. Moreover, voxels have partial occupancies at the front and back of the statue because anti-aliasing occurs in the z-dimension as well as the x,y-dimensions. This effect is most easily visible on the back of the Buddha statue.

	polys	256^3		4096^3	
		coeff	synth	coeff	synth
Armadilloman	30.0k	113	22	7,310	3,990
Head	477k	393	23	12,000	4,740
Buddha	1.09M	557	21	10,700	3,340
David 2mm	7.23M	2,250	19	14,800	1,790

Table 3.1: Time taken (in milliseconds) to rasterize volumes of increasing complexity at 256^3 and 4096^3. We show the time taken for coefficient calculation and synthesis separately.

Anti-aliasing is important for many algorithms that process rasterized volumes. For example, CSG operations can be performed by rasterizing the volumes of two meshes, performing a pairwise CSG operation on the two volumes and then extracting a surface as a level set using an algorithm like Marching Cubes (MC) [95], but are specialized to rasterized volumes (see Section 5). Figure 3.4 shows such an operation using our wavelet rasterization on a 1024^3 grid. The quality of the surface generated by contouring depends on the rasterization algorithm. Figure 3.5 shows the result of using a binary rasterization, which is typical of other methods [23, 55, 53, 71, 46, 50, 139, 51], and our anti-aliased rasterization over a 256^3 grid. Note that contouring smooths the surface from the binary voxelization because vertices lie at the midpoints of grid edges and are connected using the MC table. Even so, triangles in a contoured surface have only a small set of orientations when contoured from from a binary voxelization, which produces a poor-quality surface.

3.4 Conclusions and Future Work

We believe that 2D and 3D rasterization is a fundamental problem in Computer Graphics, and our algorithm offers a method for computing anti-aliased, box-filtered rasterizations analytically. The method we present is efficient and general in that we can rasterize arbitrary 2D polygons, shapes bounded by Bézier curves, and 3D triangle surfaces.

In wavelet rasterization, wavelet synthesis and analysis correspond to pre- and post-

Figure 3.5: The CSG operation from Figure 3.4 computed on a 256^3 grid contoured with Marching Cubes. The image to the left shows the result of using binary rasterization and the image to the right shows the result from our anti-aliased rasterization.

filtering. Direct extension to higher-order filters is trivial for filters that form wavelet bases, but most common filters do not satisfy this property. Note that, sinc forms an orthogonal wavelet basis but has infinite support. Wavelet rasterization also provides anti-aliased images at multiple resolutions that can be computed by truncating the summation of the detail coefficients in Equation 3.3 before pixel resolution. This progressive refinement of rasterized images suggests the possibility of generating a fixed-frame-rate rasterizer that continuously adds detail until a time limit for the frame is reached. Extremely detailed geometry would result in a more pixelated image rather than in dropped frames.

Another benefit of wavelet rasterization is that it is extremely easy to parallelize because the contribution of every line segment, curve, or triangle can be computed independently. Each depth can also be computed independently, although it is probably more efficient to reuse the clipping operations from parent cells. Conversion of coefficients to function values is even more easily parallelized, because memory accesses are disjoint.

4. STREAMING SURFACE RECONSTRUCTION USING WAVELETS*

Creating digital models of real-world objects has many applications. Digital models are used in industry to perform physical simulations and to visualize shapes in ways not possible in real life. In entertainment, digitized clay models are animated for games and movies. In archeology and art, the same techniques are used to create digital repositories of artistic works like Michelangelo's Schiavo Barbuto statue shown in Figure 4.1 that was scanned by the Digital Michelangelo Project [90]. Typically, shapes are acquired by measuring the distance of a surface to a scanning device such as a laser range scanner or a Microsoft Kinect that provides point samples on the surface of the object. The point samples are then used to build a three-dimensional polygonal model that approximates the shape of the object being scanned.

Reconstructing surfaces from the data produced by these scanning devices is difficult. The surface may be oversampled in some regions because of multiple overlapping scans that are intended to cover the entire shape. On the other hand, cracks and crevices usually cannot be scanned, and physical size limitations may prevent the scanner from accessing every portion of the shape, so the data may contain gaps and holes. With the advent of better scanners, the amount of data collected has grown dramatically. For large statues such as Michelangelo's 14 foot tall David, the number of samples can easily be in the hundreds of millions to billions of points. Such large data sets necessitate efficient algorithms, in terms of both time and memory, to process the collected samples. In addition, real-world data always contain noise due to sensor inaccuracies, which creates the need for a robust algorithm.

*Reprinted with permission from "Streaming Surface Reconstruction Using Wavelets" by Josiah Manson, Guergana Petrova, and Scott Schaefer, 2008. Computer Graphics Forum, 27 (5), 1411–1420, © 2008 The Author(s) Journal compilation © 2008 The Eurographics Association and Blackwell Publishing Ltd.

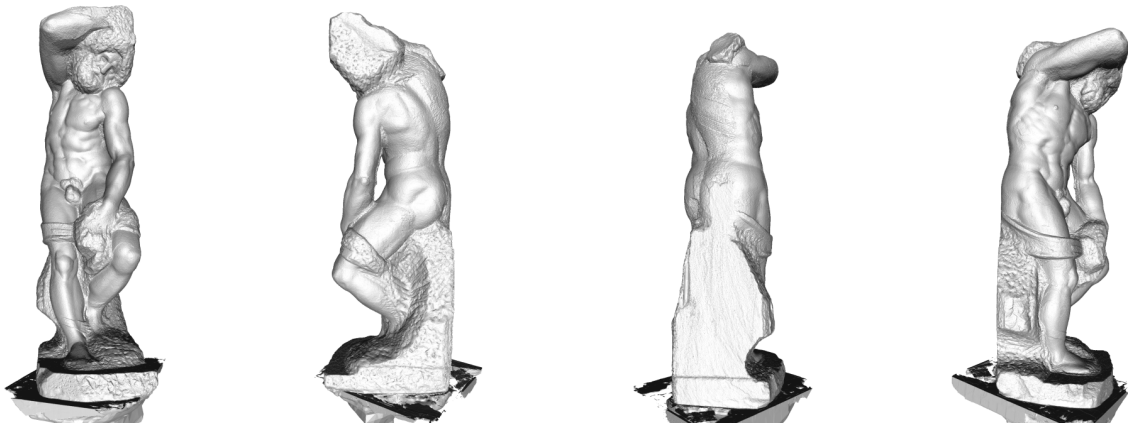


Figure 4.1: Surface reconstruction of "Barbuto" from laser range scans containing a total of 329 million points (7.34GB of data). Our wavelet surface-reconstruction method completed the reconstruction in 112 minutes with 329 MB of memory.

Recently, implicit methods, such as the level-set methods pioneered by Osher [106] and Sethian [120], have gained popularity for surface reconstruction. Solving for the implicit representation of an object from samples on the boundary of a shape is conceptually similar to rasterizing voxels from a known triangle mesh, as described in Chapter 3. The difficulty of surface reconstruction is that, unlike rasterization, the exact boundary of the shape is not known. Instead, the scanned data have noise, inconsistent sampling density, occlusion, and miscalibration. We use wavelets to calculate the implicit function representing an object because wavelets are robust to noisy data. Computing wavelet coefficients is fast and memory efficient because wavelets form an orthogonal basis with compact support. We can also use the small support of wavelet basis functions to evaluate only part of the surface at a time. We keep an octree of some small depth d_m in memory and encode subtrees corresponding to high-resolution details in a streaming fashion. This streaming technique allows us to process massive data sets that exceed the available computer memory.

In some applications, quality of the reconstructed surface is more important than the speed of reconstruction. In others, such as the navigation of unmanned vehicles, recon-

struction speed is more important than quality. Our technique provides a framework for handling any of these applications by selecting an appropriate wavelet basis. This selection depends on the smoothness of the wavelet, which is intrinsically connected to its support. As the support of the wavelet decreases, the smoothness of the wavelet and the smoothness of the reconstructed surface decreases. Smoother wavelets produce smoother surfaces but require more computation.

4.1 Related Work

Surface reconstruction is a well-studied problem. We discuss only some of the existing methods and refer the reader to Shall and Samozino [116] for a survey on recent developments in this field and to Kazhdan et al. [86], which compares several reconstruction techniques. Surface reconstruction methods fall into two main categories: explicit and implicit. Explicit methods connect scanned points with triangles. The Power Crust algorithm [2], Robust Cocone [40], and Super Cocone [39] are among the well-known examples of such methods. Streaming triangulation algorithms for surface reconstruction [9, 1] have also been developed to handle large data sets. However, these algorithms do not perform well in the presence of noise because they interpolate the input data. Furthermore, they typically need to look up neighboring information to create the triangulations, which makes them many times slower than most implicit methods and thus unsuitable for reconstructing extremely large data sets.

Implicit algorithms reconstruct a surface by extracting the level-set of an implicit function that is calculated from sensor data. The advantage is that it is unnecessary to parameterize the surface, and operations such as shape-blending, offsets, and deformations are easy to calculate. Implicit methods also approximate the input data, which means that they are more robust to noise in the input data than interpolatory methods. One such method uses radial basis functions (RBF) [22] to represent the implicit function. However, fitting

and evaluating RBFs is too slow to reconstruct implicit surfaces from point sets consisting of more than a few thousand points. Multiple partition of unity (MPU) implicits [105] is an octree subdivison method that fits local, piecewise quadratic functions to the data and uses weighting functions (partitions of unity) to blend these functions together. Similar to the FastRBF method, MPU implicits can produce noisy surfaces with extraneous parts. However, MPU implicits is simple and fast. Other implicit methods include Hoppe et al. [74] and VRIP [36], which are more robust to noise than the other algorithms. Although slower than MPU implicits, both methods can be used on medium-sized data sets. Note that, despite run-length encoding tricks used in VRIP, both methods have difficulty processing extremely large data sets because the entire representation of the implicit function must reside in memory.

More recently, an implicit surface-reconstruction method based on Fourier series [85] was developed to reconstruct a smooth surface and robustly handle noise and gaps in the data. However, computing a single Fourier coefficient must sum over all input samples because the basis functions are globally supported. The method also requires a huge amount of memory because of the use of a uniform grid, which limits its application to relatively modestly sized data sets. A solution to this problem was recently proposed [117]. The authors suggested combining an FFT method with adaptive subdivision and partition of unity blending techniques of MPU implicits. The FFT approach was later modified [86] to use an octree and find the implicit function by solving a Poisson equation. The method is further improved by using a streaming approach [13] to process data sets out of core. This approach allows the algorithm to handle data sets on the order of hundreds of millions of point samples. Despite its relative speed, processing can still take days for large number of points, even with a parallel implementation [14].

4.2 Wavelet Approximation of the Indicator Function

We use the input points p_i on the surface ∂M and their outward normals $\vec{n}_i$ to approximate the indicator function χ_M (defined in Equation 3.6) of the solid M with boundary ∂M by approximating the wavelet coefficients of χ_M. The surface $\partial\tilde{M}$ of the level set $\tilde{M}$ is an approximation to the original surface ∂M. Without loss of generality, we assume that M lies in the cube $[0,1)^3$. The coefficients $c_{0,\mathbf{k}}$ and $c^e_{j,\mathbf{k}}$ in the wavelet representation of χ_M are determined by Equation 3.8.

Kazhdan wrote that the Fourier coefficients of χ_M can be calculated using the divergence theorem [85], and we derive the corresponding formulas to evaluate wavelet coefficients in Section 3.2.1. When rasterizing triangle meshes into voxels, numerical robustness was less important than the ability to evaluate wavelet coefficients quickly, because we assume the mesh is well defined and closed. In surface reconstruction, the boundary is poorly defined and captured with noisy sensors, so we choose values for the parameters α and β that are most tolerant to noise.

$$
\begin{aligned}
F^{(0,0,0)}(p) &= \frac{1}{3}(\bar{\Phi}(p_x)\phi(p_y)\phi(p_z), \phi(p_x)\bar{\Phi}(p_y)\phi(p_z), \phi(p_x)\phi(p_y)\bar{\Phi}(p_z)) \\
F^{(1,0,0)}(p) &= (\bar{\Psi}(p_x)\phi(p_y)\phi(p_z), 0, 0) \\
F^{(0,1,0)}(p) &= (0, \phi(p_x)\bar{\Psi}(p_y)\phi(p_z), 0) \\
F^{(0,0,1)}(p) &= (0, 0, \phi(p_x)\phi(p_y)\bar{\Psi}(p_z)) \\
F^{(1,1,0)}(p) &= \frac{1}{2}(\bar{\Psi}(p_x)\psi(p_y)\phi(p_z), \psi(p_x)\bar{\Psi}(p_y)\phi(p_z), 0) \\
F^{(1,0,1)}(p) &= \frac{1}{2}(\psi(p_x)\phi(p_y)\psi(p_z), 0, \psi(p_x)\phi(p_y)\bar{\Psi}(p_z)) \\
F^{(0,1,1)}(p) &= \frac{1}{2}(0, \phi(p_x)\bar{\Psi}(p_y)\psi(p_z), \phi(p_x)\psi(p_y)\bar{\Psi}(p_z)) \\
F^{(1,1,1)}(p) &= \frac{1}{3}(\bar{\Psi}(p_x)\psi(p_y)\psi(p_z), \psi(p_x)\bar{\Psi}(p_y)\psi(p_z), \psi(p_x)\psi(p_y)\bar{\Psi}(p_z)).
\end{aligned}
$$

Given a function $\vec{F}^e_{j,\mathbf{k}}$, we can discretize the surface integral by summing over the point samples $(p_i, \vec{n}_i)$

$$c^e_{j,\mathbf{k}} = 2^{3j/2} \int_M \vec{F}^e_{j,\mathbf{k}}(p(\sigma)) \cdot \vec{n}(\sigma) d\sigma \tag{4.1}$$

$$\approx 2^{3j/2} \sum_i \vec{F}^e_{j,\mathbf{k}}(p_i) \cdot \vec{n}_i \Delta\sigma_i, \tag{4.2}$$

where $\Delta\sigma_i$ is an estimate of the differential surface area associated with the sample point p_i. Notice that, unlike globally supported functions such as the Fourier basis, this summation does not involve all point samples but only those within the support of $\psi^e_{j,\mathbf{k}}$. The only coefficients that sum over all points p_i are at the top level of the tree and are designated as $c^{(0,0,0)}_{0,\mathbf{k}}$. In this case, we only compute those coefficients $c^{(0,0,0)}_{0,\mathbf{k}}$ that correspond to basis functions $\phi(x_1 - k_1)\phi(x_2 - k_2)\phi(x_3 - k_3)$, whose support overlaps the region of interest $[0, 1)^3$ containing M. Fortunately, this is a small, constant number of coefficients that depends on the support of the scaling function ϕ.

Finally, we need to approximate the surface area $\Delta\sigma_i$ associated with each point p_i. There are many ways of estimating $\Delta\sigma_i$, such as weighting by a Gaussian [85]. We use a simple, octree-based method to compute $\Delta\sigma_i$ that can handle non-uniformly sampled data points. We refine all octree cells containing sample points until we reach the maximum depth d specified by the user. Once all points are inserted into the octree, we prune leaves of the tree until each leaf is adjacent to at least 3 occupied leaves of the same depth. This pruning guarantees that the surface is adequately sampled for the resolution of cells in the tree. The area associated with a point p_i is then the area of the side of the leaf divided by the number of points in that leaf $\Delta\sigma_i = 2^{-2d_i}/m$, where m is the total number of points in the leaf containing p_i and d_i is the depth of the leaf.

4.3 Surface Extraction

As described in Section 3.2, we compute an approximation $\tilde{\chi}_M$ to χ_M and recover a solid $\widetilde{M}$ that is a level set of the $\tilde{\chi}_M$. Because $\tilde{\chi}_M \approx \chi_M$, which is 0 outside M and 1 inside of M, one reasonable option is to extract the surface that is an iso-contour of the function at $1/2$ using Marching Cubes [95]. However, the Marching Cubes surface is not smooth because Marching Cubes assumes that the sampled function is smooth. We describe a contouring method specific to indicator functions in Section 5. When a wavelet basis other than Haar is used, or if the function is altered after evaluating the coefficients, then the contouring method described in Section 5 does not apply, and we use Marching Cubes instead. For poorly scanned surfaces or point sets with high amounts of noise, we can choose a data-dependent iso-value using the average value of $\tilde{\chi}_M$ over the sample points.

The ability of wavelets to detect discontinuities creates an octree that is adaptively refined along the boundary ∂M of M. We use this octree to construct a polygonal model of the boundary $\partial\tilde{M}$ by applying an octree-contouring method [82]. This algorithm computes the dual-cell structure of the octree through a recursive octree walk and uses the values of $\tilde{\chi}_M$ at the vertices of the dual cells, which are located at the centers of the corresponding octree cells. The surface is then contoured over the dual grid of the octree to create a surface that produces a topological and geometrical manifold. Because the number of dual cells is proportional to the size of the octree, the running time of contouring is proportional to the size of the octree.

4.3.1 Post-processing of the Indicator Function

The smoothness of the wavelet basis determines the smoothness of $\tilde{\chi}_M$ and, therefore, the smoothness of the level set $\tilde{M}$. Wavelets with small support yield higher performance algorithms because fewer wavelet coefficients are calculated, and each coefficient is influ-

Figure 4.2: Surface reconstruction using Haar wavelets (left) results in a noisy surface because the basis functions are discontinuous. Smoothing the indicator function results in a substantially smoother surface at a small cost to speed and approximation quality (right).

enced by a smaller number of sample points. However, smaller support reduces the quality of the resulting surface. Instead of increasing the support of the wavelet to improve the quality of the reconstructed surface, an alternative is to perform a post-processing smoothing step on the indicator function $\tilde{\chi}_M$. With this method, we can retain the time efficiency associated with wavelets of small support and get a more visually appealing surface. We compare the Hausdorff error of our method to the error of other methods in Table 4.3. Our method outperforms other techniques in terms of accuracy, both with and without post-processing.

Smoothing a function is typically performed by convolving the function with a small smoothing kernel over a uniform grid. We have an adaptive octree, so we modify a small convolution kernel to operate over octrees. The kernel evaluates adjacent cells and is the tensor product of the mask $(1/4, 1/2, 1/4)$ in $\mathbb{R}^3$. We smooth the function $\tilde{\chi}_M$, given by Equation 3.4, by summing over the dyadic levels j up to a user specified depth d_{max}. Given a cell at depth d, we compute the value of $\tilde{\chi}_M$ at the center of the cell and its 26

neighbors at the same level using Equation 3.4. If a neighbor does not exist, then the wavelets indexed by this neighbor do not contribute to the sum. We then perform uniform convolution over this locally uniform grid and treat the obtained value as the value of the smoothed function at the center of that cell. Figure 4.2 shows an example of a surface reconstructed using Haar wavelets (see Section 4.4) without (left) and with (right) this smoothing step.

4.3.2 Streaming Reconstruction

The storage space for $\tilde{\chi}_M$ is proportional to the surface area of the reconstructed surface because refinement is only performed near point samples. However, extremely large data sets may require more space than can fit into the memory of most desktop machines. To process these data sets, we developed a streaming version of our algorithm. Like most streaming algorithms, we require that the input points are sorted in one of the Euclidean directions. If the points are not sorted, we preprocess them by sorting along the longest Euclidean direction of their bounding box using an out-of-core merge sort. We assume, without loss of generality, that the sort is in the z-direction. For a data set of 205 million points, the sort takes 20 minutes, and the sorting time is small compared to the time for surface reconstruction.

Our streaming algorithm builds a low resolution, in-core approximation of χ_M down to some depth $d_m < d_{max}$, where d_{max} is the maximal depth of the octee and encodes subtrees corresponding to high-resolution details in a streaming fashion. Note that the coefficients $c_{0,\mathbf{k}}^{(0,0,0)}$ depend on all sample points. Therefore, $\tilde{\chi}_M$ cannot be evaluated until all points are processed at least once. Our solution is to perform two passes over the data. The first pass constructs all coefficients down to depth d_m. The second, streaming pass builds the non-zero wavelet coefficients $c_{j,\mathbf{k}}^{e}$ of $\tilde{\chi}_M$ for $d_m < j \leq d_{max}$ and $\ell 2^{-d_m} \leq k_3 < (\ell+1)2^{-d_m}$ for $0 \leq \ell < 2^{d_m}$. For each slice ℓ, we build the corresponding wavelet coefficients, smooth

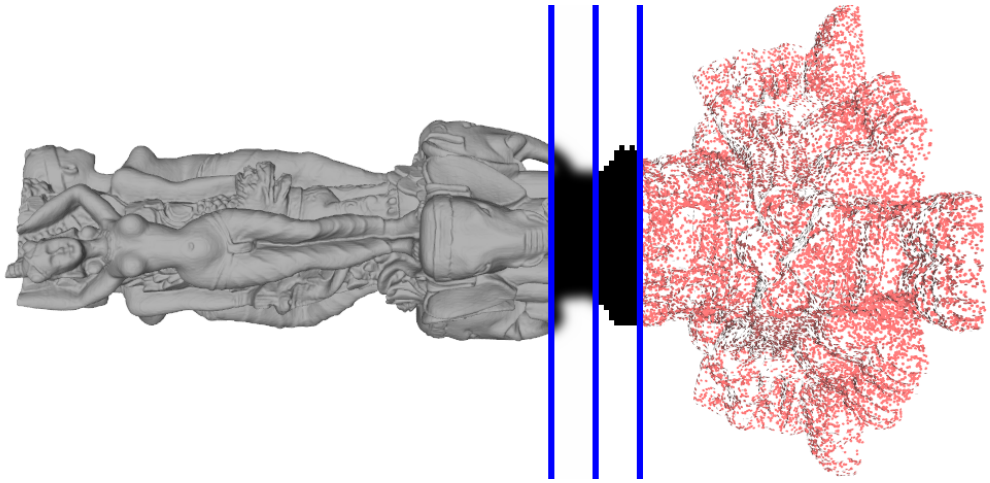


Figure 4.3: Depiction of our streaming implementation. We first construct the wavelet coefficients of the indicator function, smooth the function, and then extract the iso-surface.

the function values, and create polygons before deleting the subtrees corresponding to the slice from memory. Figure 4.3 depicts this streaming process.

Any choice of d_m works with our streaming algorithm, but its value will affect the amount of memory required to reconstruct the surface. Choosing $d_m = 0$ or d_{max} requires that the entire tree fits into memory, so we choose the d_m that minimizes memory usage. Assume that there are L leaves of the octree at depth d_{max}. Because the size of the octree at each level is proportional to the surface area of ∂M, the number of cells at depth j is approximately $L4^{-(d_{max}-j)}$. Therefore, our streaming algorithm has a working set of approximately

$$\sum_{j=0}^{d_m} \frac{L}{4^{d_{max}-j}} + \frac{\beta}{2^{d_m}} \sum_{j=d_m+1}^{d} \frac{L}{4^{d_{max}-j}}$$

cells in memory at any one time. We keep β slices in memory for the different stages of our algorithm, where β is dependent on the support of the wavelet used. For the Daubechies wavelets, which we consider in Section 4.4, $\beta = 4$. This sum is minimized with respect to d_m to yield an optimal value of $d_m \approx 0.69 d_{max}$. This technique does not allow us to

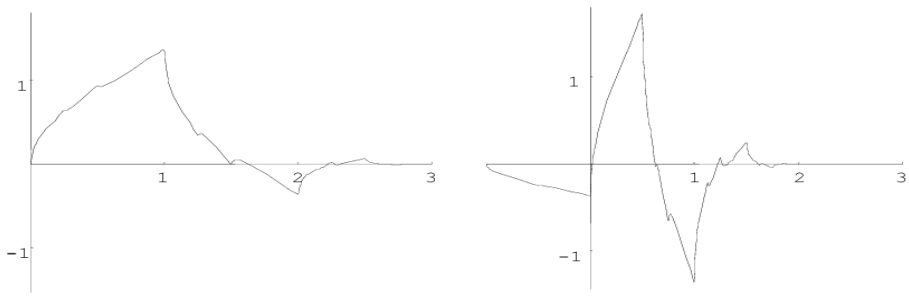


Figure 4.4: The D4 scaling function ϕ (left) and the corresponding wavelet ψ (right).

process arbitrarily deep octrees (the octree to depth d_m must still be stored), but it allows us to process much deeper trees than a strictly in-core algorithm. We have been able to reconstruct surfaces down to depth 14 in memory (see Section 4.5). At that resolution, a single cross-section of the grid at the maximal depth has over 250 million cells, and we are able to process the largest data sets that can be obtained.

4.4 Implementation

Computational time is a major concern, which motivates us to explore wavelets with small support, such as the Haar and 4^{th} order Daubechies (D4) wavelets. Despite the lack of smoothness of these wavelets, the reconstructed surface is a good approximation to the original surface (see Section 4.5). In Section 3.2 we defined Haar wavelets by Equations 3.2 and 3.2 and show a depiction of the 2D tensor product functions in Figure 3.2. In this chapter, we also rasterize into the D4 basis and Figure 4.4 depicts plots of the scaling and wavelet functions of the D4 basis. The D4 wavelet has the scaling relationship

$$\phi(t) = \begin{pmatrix} \frac{1+\sqrt{3}}{4} & \frac{3+\sqrt{3}}{4} & \frac{3-\sqrt{3}}{4} & \frac{1-\sqrt{3}}{4} \end{pmatrix} \begin{pmatrix} \phi(2t) \\ \phi(2t-1) \\ \phi(2t-2) \\ \phi(2t-3) \end{pmatrix}.$$

Given that the D4 wavelet ψ is not analytic, we evaluate the functions $\vec{F}^{e}_{j,\mathbf{k}}$ at the sample points p_i using a piecewise linear interpolant of the ϕ, ψ, Φ and Ψ functions sampled over a uniform grid. The exact values on the uniform grid are found using a standard technique for evaluating functions that satisfy a scaling relationship with a finite number of non-zero scaling coefficients [37, 10]. Note that the functions Φ and Ψ are in this category as well. For example, Φ satisfies the scaling relation

$$\Phi(t) = \begin{pmatrix} \frac{1+\sqrt{3}}{8} & \frac{3+\sqrt{3}}{8} & \frac{3-\sqrt{3}}{8} & \frac{1-\sqrt{3}}{8} \end{pmatrix} \begin{pmatrix} \Phi(2t) \\ \Phi(2t-1) \\ \Phi(2t-2) \\ \Phi(2t-3) \end{pmatrix},$$

derived by integrating the scaling relation for ϕ. In our implementation, we use a uniform rational grid with spacing $\frac{1}{64}$ to represent these functions, but grids with other spacings can also be used.

4.5 Results

We compare the speed, memory usage, and accuracy of Haar and D4 wavelets used to reconstruct a surface with other surface-reconstruction methods. Our method is an order of magnitude faster than other methods (see Table 4.2) and the surfaces from our method are more accurate than the surfaces obtained by other methods (see Table 4.3).

We apply our algorithm to point clouds for which the separate point scans are already aligned in 3D. We also assume that the point samples are oriented, i.e., we use the coordinates of the points p_i on the surface and the coordinates of the outwards unit normals n_i to the surface at p_i. If the data for the normals are absent, we estimate the normals using a local PCA method or polynomial fitting. However, many scanners produce not only point samples but also partially triangulated scans that estimate how samples are connected from

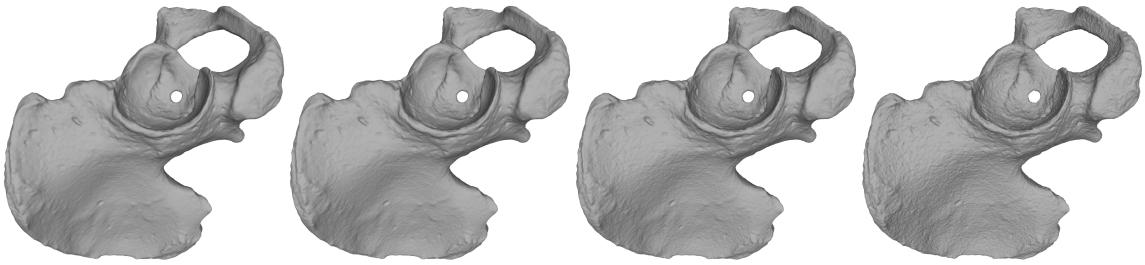


Figure 4.5: Reconstruction of a hip joint using Haar wavelets with varying amounts of noise in the normals. From left to right: 0 degrees, 30 degrees, 60 degrees, 90 degrees uniform random rotational deviation in the normal direction.

a single scanning direction (all of the data in the Digital Michelangelo Project is of this form). Although these triangles are not sufficient to produce a closed, triangulated model, they do allow the efficient estimation of normals. Furthermore, the orientation of the normal (inward vs. outward) can also be constructed robustly from these scans because the surface must be visible to the scanner and the normals must, therefore, be oriented in the direction of the scanner.

Normals of point samples estimated from the range-scanned data are especially susceptible to noise because normals depend on the derivative of measured surface positions, and derivatives magnify measurement errors. However, we show that our technique is robust with respect to errors in the normal directions. Figure 4.5 shows several reconstructions in which we incrementally added more noise to the input normals. Even with errors up to 90 degrees, the surface was faithfully reconstructed. With more noise, the reconstruction quality begins to suffer noticeably, but at this noise level (> 90 degrees deviation) the normals point inward and become meaningless.

Surface reconstruction using Haar wavelets is extremely fast. Haar wavelets create a minimal number of coefficients in the octree, and the coefficients can be computed quickly because the scaling and wavelet functions, as well as their integrals, have an analytical form and because the support of the wavelet is small. Figure 4.6 shows a depth 14 recon-

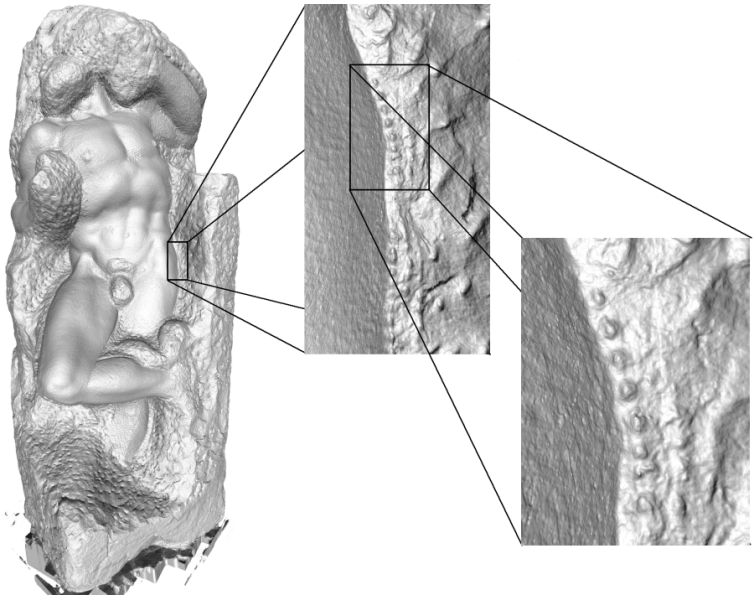


Figure 4.6: "Awakening" with 381 million points (8.51 GB of data) reconstructed using Haar wavelets at depth 14 took about 81 minutes and produced over 590 million polygons. The two zooms show that even small chisel marks are reconstructed with a high degree of accuracy.

struction of Michelangelo's Awakening statue using Haar wavelets. This data set is one of the largest we obtained and contains 381 million points. Despite its size, our method was able to produce a faithful reconstruction in about 81 minutes.

For many real-world data sets, Haar wavelets create pleasing surface reconstructions when the scans are well-aligned and the noise is relatively low. However, in some cases, the reconstruction begins to fit noise in the data due to the small support of the Haar wavelet. Surfaces are higher quality when reconstructed using D4 wavelets rather than with Haar wavelets because the support of the basis functions is larger. The D4 wavelets sum over more point samples and make the method more resilient to noise in the input data. The larger support of this basis also increases the number of coefficients that must be stored and increases the computation time. Using D4 wavelets roughly triples the number of octree cells compared to Haar wavelets, but reconstruction times are still fast compared

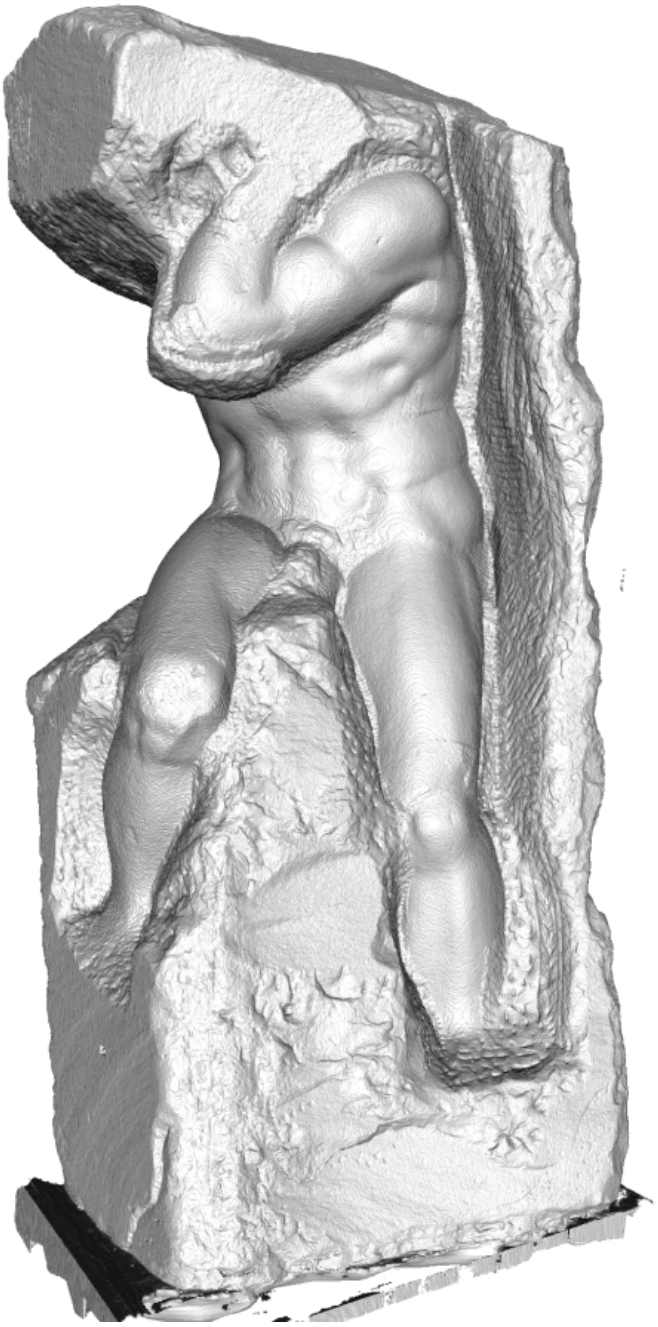


Figure 4.7: Reconstruction of Michelangelo's Atlas with 410 million points (9.15 GB of data) at depth 12 with D4 wavelets took less than 2.5 hours and produced 42.7 million polygons.

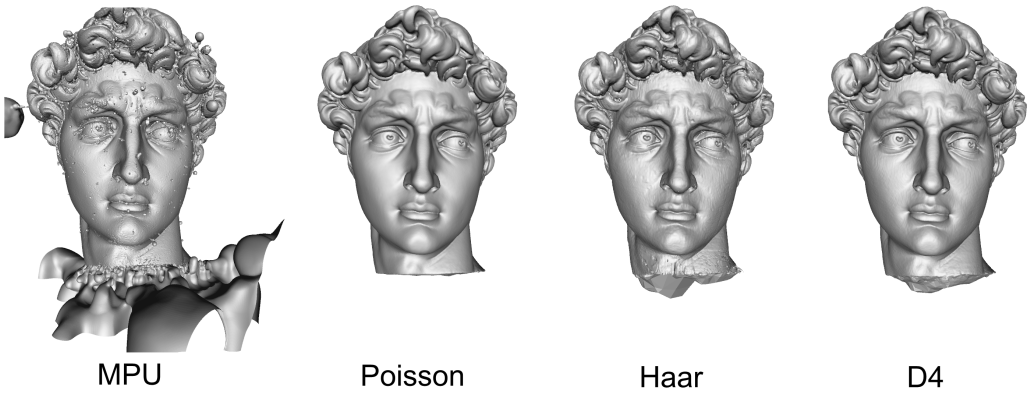


Figure 4.8: Comparison of reconstructions of David's head with 4.5 million points (103 MB of data) at depth 9 with MPU implicits, Poisson reconstruction, Haar wavelets, and D4 wavelets.

Method	Time(s)	Memory(MB)	Polygons
MPU	551	750	1582380
Poisson	289	57	1257980
Haar Wavelet	17	13	1357872
D4 Wavelet	82	43	1377858

Table 4.1: Reconstruction of David's head consisting of 4.5 million points at depth 9 with various methods.

to other methods. Figures 4.1, 4.7 and 4.8 (bottom right) were all reconstructed using D4 wavelets.

Table 4.1 shows running times for several popular surface-reconstruction methods for which implementations are freely available. All operated on the same data set of 4.5 million points from David's head and at a maximal octree depth of 9. All tests were run on an Intel 6700 with 2GB of RAM. Poisson surface reconstruction [13] is a fast surface-reconstruction algorithm, yet our method using D4 wavelets is 3.5 times faster and Haar wavelets are 17 times faster. Our memory requirements are also low because of our streaming implementation. Poisson surface reconstruction also uses a streaming

Model	Points	Haar,12	Haar,13	Haar,14	D4,12
Barbuto	329M	38.7/100	58.3/252	81.6/777	111.9/329
Awakening	381M	45.5/100	62.4/187	80.8/573	133.3/339
Atlas	410M	51.7/133	59.0/351	97.6/1188	148.4/448

Table 4.2: Reconstruction of various models using Haar and D4 wavelets at various depths. Data is of the form time/memory where time is measured in minutes and memory in MB.

Model	MPU	Poisson	Haar	Haar Smoothed	D4	D4 Smoothed
armadilloman	1.000000	0.259120	0.153069	0.212276	0.151215	0.224511
happy buddha	1.000000	0.364243	0.223781	0.339264	0.346450	0.356715
cow	1.000000	0.086590	0.036725	0.046528	0.071015	0.074601
dragon	1.000000	0.790500	0.602828	0.536930	0.617106	0.636463
elephant	1.000000	0.507040	0.363651	0.221071	0.320602	0.372229
hand	0.332169	1.000000	0.380563	0.565324	0.335825	0.589883
hip	1.000000	0.110895	0.064997	0.093744	0.061002	0.091113
malaysia	1.000000	0.217397	0.151879	0.189831	0.144758	0.200062
teeth	0.835987	1.000000	0.418790	0.503185	0.471338	0.702229
venus	1.000000	0.371843	0.184752	0.260992	0.198316	0.285313

Table 4.3: Hausdorff distance between real surfaces and reconstructed surfaces from sampled data. Each row is normalized by the worst geometric error (lower is better).

implementation, but the smaller support and orthogonality of wavelet basis functions gives our method a smaller memory footprint.

Table 4.2 includes reconstruction times for the largest data sets that we could find. Many of these data sets contain well-aligned scans, and Haar wavelet reconstruction performs well without many errors due to noise or misaligned scans. Our method using Haar wavelets was able to process each of these data sets in under an hour at depth 12 and under 2 hours at depth 14. Using D4 wavelets was slower, but we were still able to complete the 410 million point Atlas data set at depth 12 in under 2.5 hours.

It is difficult to assess the accuracy of surface reconstruction if only point scans are available because fitting the points exactly may yield an undesirable surface. More criti-

cally, it is impossible to know the ground truth of measured data, so we generate synthetic test data in addition to using real-world scans. We sample points and normals densely from known polygon models and then reconstruct surfaces with each method from these points sets. We show the Hausdorff distance between each reconstructed surface and the original shape in Table 4.3 as computed by Metro [31]. The values in each row are normalized by the maximum error among all methods to provide a relative comparison among the different techniques. MPU implicits is on average the worst of the tested models because noisy data sets can cause MPU implicits to create extraneous sheets such as the ones shown in Figure 4.8. Poisson reconstruction typically performs much better. However, in all cases, Haar and D4 wavelets with and without smoothing recover the surfaces with higher degree of accuracy than the Poisson reconstruction. In all but one case (the hand), the wavelet methods also have lower error than MPU implicits.

4.6 Future Work

We would like to implement smoother basis functions whose support size is relatively small. Controlling support is important because the size of the support of the basis functions is closely related to the computational cost and time efficiency of the algorithm. Thus, a smaller support leads to a faster algorithm. On the other hand, smoother basis functions are needed for smoother reconstructions. Some types of basis functions we would like to explore include smoother wavelets, biorthogonal wavelets, and quasi-interpolants. Quasi-interpolant decompositions based on multivariate splines are an especially appealing choice because the support of spline basis functions is smaller than the support of a smoother wavelet or biorthogonal wavelet, and quasi-interpolants still allow a multi-scale decomposition.

5. CONTOURING DISCRETE INDICATOR FUNCTIONS*

An indicator function of a closed object has a value of one on the interior of the object and zero on the exterior. In practice, we use a Discrete Indicator Function (DIF) that represents the indicator function using a uniform grid of function values. Cells that are entirely exterior or interior to the object will have values of either zero or one respectively, but cells on the boundary of the shape will have values between zero and one. DIFs arise in several areas of Computer Graphics when a function is sampled with a box filter. These DIFs form an implicit representation of an object, and we can reconstruct an approximation of the underlying object by taking a level set of the function at value $1/2$.

A common example of a DIF is the pixels used to display an anti-aliased font on a computer screen, where values of gray represent how much of a glyph is in each pixel. Beyond fonts, there are applications in which arbitrary images must be converted from a pixel to a vector format. For example, it may be desirable to magnify a low-resolution image that contains hard boundaries. Although typical pixel-based magnification results in a blurred image, finding a contour first produces a far crisper and more aesthetic result, as demonstrated by Valve Software [64] in their newest games. Once a discrete contour has been calculated, it is possible to draw the contour with hardware acceleration [89].

DIFs are also used as alpha masks when compositing multiple images. For example, it is common to film objects in front of a blue or green screen and to remove the background by calculating an anti-aliased mask at each pixel. A pixel-based composition of many overlaid objects may produce an undesirable, aliased result in which multiple boundaries overlap unless an accurate vector-based composition is calculated using the contours of

Figure 5.1: A DIF representing a complex shape sampled over a 512^3 grid. From left to right: Marching Cubes, Marching Cubes after a Gaussian blur of size 3, Marching Cubes after a Gaussian blur of size 7, and our method without blurring. Blurring increases the smoothness of the Marching Cubes surface but sacrifices details in the shape.

the objects.

Analogs of DIFs are also encountered in medical imaging. In a CAT (Computerized Axial Tomography) scan or an MRI (Magnetic Resonance Image) there are structures such as bones or tumors that stand out from the background and have clear boundaries that must be identified. In the case of a 3D MRI, these boundaries define a 3D surface. Surface boundaries are also found in other areas of computer graphics. For example, Eulerian simulations of water physics define occupancies over a 3D grid. Any inaccuracies in calculating the air/water interface from the DIF result in high-frequency noise or ripples that do not exist in the simulation. Another important occurrence of a DIF is during reconstruction of the surface of an object from a point cloud. For more information on surface reconstruction using DIFs, see Chapter 4.

In the past, contouring methods have focused on smooth functions rather than on DIFs. We believe that DIFs are an important subclass of functions for which people typically calculate suboptimal contours. The inaccuracies in calculated surfaces are often small in scale, but they can have large visual effects on the texture, lighting, refraction, and

reflection of an object. In this chapter, I describe a simple algorithm that is tailored to find accurate contours for the specific case of DIFs.

We describe a simple modification to standard contouring algorithms, such as Marching Cubes (MC), that enhances the quality of contours calculated from DIFs. Our analysis shows that the linear interpolation in these methods can be replaced by our new interpolant to estimate the position of the contour more accurately. Our interpolant has four cases, and the two most common cases are extremely simple to calculate. We also provide qualitative and quantitative comparisons between the interpolants that show that our method clearly improves the contours of DIFs.

5.1 Related work

The simplest and most common strategy for calculating implicit contours of functions is to contour cells of a regular grid independently from one another using algorithms like MC [137, 95]. In our examples, we assume that samples are taken at the centers of cells and therefore run MC over the dual grid. Because the possible values of the DIF are between zero and one, it is natural to define the contour at the position where the DIF has a value of $1/2$. MC determines the topology of a cell from a lookup table indexed by an eight-bit integer, where each bit corresponds to one of the eight corners of the cell and is set to 0 if the corner is outside the solid or 1 if it is inside. In MC, the function is assumed to be smooth so that over short distances, like the span of a cell, the function can be approximated as a linear function. Vertices of the surface passing through each cell are therefore placed along the edges of the cells where the line interpolating the connected grid values has a value of $1/2$.

When a function is smooth, it can be approximated by linear functions at small scales, but indicator functions are discontinuous, which means that a linear interpolant is not a suitable approximation of a DIF. Therefore, using a linear interpolant on a DIF produces

a contour that deviates from the ideal contour. Although these deviations are small on an absolute scale, they are easily visible on an illuminated surface because angular deviation of surface normals is independent of sampling resolution. Inaccuracies in the surface follow a regular pattern determined by the grid and create noticeable aliasing. This results in an easily visible, high-frequency pattern rather than uniform noise (see Figure 5.1).

Because MC only works well with smooth functions, a common solution to the aliasing problem is to generate a smooth function by applying a low-pass filter to the input function. Although applying a Gaussian filter [103] is the simplest solution, more sophisticated methods have been used [129, 130]. Unfortunately, the contour extracted from the smoothed function loses high-resolution details that are present in the input function.

Although many researchers have worked on modifying and extending MC, they have focused on areas other than the interpolation function. For example, there has been much interest in calculating closed contours over adaptive grids. Many of these methods, such as Dual Marching Cubes [115] and Unconstrained Isosurface Extraction on Arbitrary Octrees [87], reduce to MC in regions that have uniform sampling.

In contrast, other methods [30, 58, 113, 135] assume that the function provides no information besides classifying the vertices as belonging to a particular solid. In the simplest case of binary segmented data (inside/outside), these methods have a goal similar to that of our method in that they attempt to calculate a smooth boundary to an indicator function. The difference is that the grids on which they operate do not store fractional occupancies like DIFs. This means that any initial guess made by these methods is not smooth, and a filtering pass is required to fix the poor initial guess.

The Surface Nets algorithm [58] segments a DIF with a contour that is both smooth and guaranteed to be within one pixel of the true contour, but it does so at a high computational cost. Surface Nets first creates an initial guess of the contour by creating a contour that is dual to the input grid. The method then performs an iterative relaxation of the surface while

enforcing constraints that prevent surface vertices from exiting their dual cells. Similarly, Pressing [30] calculates an initial guess and constrains planar patches to remain planar while iteratively relaxing vertices on cell edges. In these methods, a constrained, iterative smoothing operation is required to remove the obvious aliasing that is present in the initial guess. Our method is much more efficient because we require no iteration or optimization to produce a contour that is free of aliasing.

Several methods have been proposed for contouring multi-material volume fractions stored in a grid. Contouring a DIF can be viewed as the two-material case of this more general problem. The multi-material contouring method discussed by Bonnell et al. [15] extends Marching Tetrahedra (MT) to the multi-material setting. However, this method degenerates to linear interpolation with two materials and generates surfaces with the same oscillations as those introduced by MC on DIFs.

Another approach to solving the multi-material problem is to use particles that repel each other and are attracted to the interface between materials [100]. A computationally costly Delaunay triangulation of the particles can then be used to produce triangles with good aspect ratios. However, the restorative forces that place particles on the boundary require a smooth function, which the authors achieve by blurring the material composition functions. This approach results in the same loss of quality and volume preservation that affects the surface reconstructed from a blurred function in MC.

Anderson et al. propose two methods of contouring multi-material volume fractions. Their first method [3] finely subdivides each cell and randomly assigns subcells to be different materials so that the ratio of materials in the subcells approximates the volume fractions in the original cells. Subcells are then randomly swapped in a simulated annealing minimization of the surface area between material types. Because each subcell can be of only one material type, the resulting surface contoured over the subcells is blocky. In their second paper [4], the authors address this problem by solving for new vertex positions

that minimize surface curvature. Unfortunately, the accuracy of volume preservation is determined by the level of subdivision and the proposed energy function does not prevent oscillations in the contoured surface, even for a highly subdivided grid.

There have also been several recent techniques designed to improve the quality of the triangles in surfaces produced by MC [69, 119, 111, 41, 42]. These methods modify the topology of the surface produced by MC to avoid long, skinny triangles. Unfortunately, visual artifacts arise from using MC on DIFs because of the incorrect assumption that the underlying function is smooth. Our technique improves surface geometry by modifying that assumption. Methods that improve the triangle quality of MC surfaces solve an orthogonal problem and can (perhaps even should) be used in conjunction with our method.

5.2 Calculating contours from DIFs

Our input is a DIF in which we are given the fractional occupancies of cells in a regular grid. Every sampled value corresponds to a cell volume, so we consider samples to be located at the centers of cells and we connect the samples with a dual grid. Our contouring method is a simple modification of MC in which we replace the function that calculates vertex positions along cube edges. For simplicity, we first analyze the 2D case in which areas are stored for each cell and contours are lines rather than surfaces. We then apply the technique to 3D data in Section 5.3. Instead of assuming that the contoured function is smooth along a dual edge and can be approximated by linear interpolation, we solve for a contour line that separates the inside/outside of the object so that the inside area matches the occupancy values of the cells. We then place the contour vertex at the intersection of the contour line and the dual edge.

There are only a few ways in which a linear surface can pass through a pair of cells in a 2D grid. Without loss of generality, we consider a pair of unit cells whose bottoms are on the $y = 0$ axis and that join along $x = 0$, as shown at the top of Figure 5.2. In this figure,

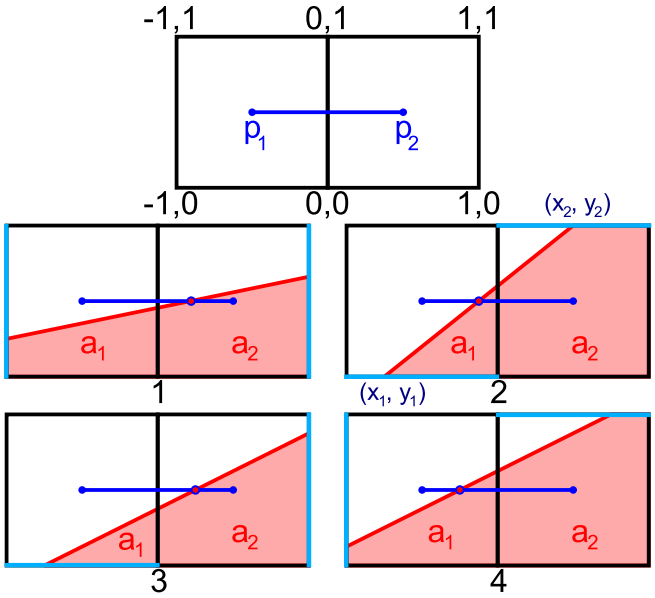


Figure 5.2: We show the coordinate systems of adjacent cells and their dual edge on top. Below, we show the four configurations in which a line can intersect the cell edges. Intersected edges are highlighted in light blue. The end points of the contour line are labeled in case 2.

the black lines are the primal cells, the blue line is a dual edge between function samples, and the red line is the estimated line separating exterior regions (white) and interior regions (pink). We also assume that the occupancy of the left cell a_1 is less than that of the right cell a_2 so that a contour intersects the dual edge when $a_1 \in [0, 1/2), a_2 \in [1/2, 1]$. To remove symmetric cases, we also assume that the contour line we estimate is oriented upward (i.e. the y-component of the normal is positive). In each of the two cells, the contour can pass through either a horizontal or vertical border, excluding the shared border, so that there are a total of four possible configurations that we enumerate in Figure 5.2: Case 1 is side-side, Case 2 is bottom-top, Case 3 is bottom-side, and Case 4 is side-top.

There are multiple ways a line can intersect the pair of cells, so we find an intersection with the contour in two steps. First, we determine which case to use based on the values

of a_1, a_2. Second, we locate the intersection between the dual edge and the estimated contour. Our strategy for all cases is to first calculate the equation for the contour line. Our lines are always of the form $y = mx + b$, with $m = \frac{y_2 - y_1}{x_2 - x_1}$ and $b = \frac{x_2 y_1 - x_1 y_2}{x_2 - x_1}$, and where (x_1, y_1) and (x_2, y_2) are the points at which the line intersects the boundary of the two cells. Notice that one component of each of the endpoints is constrained for each of the four cases. Once we determine the equation of the line, we intersect the line with the dual edge $p(t)$ to find the parametric value t, where $p(t) = p_1(1 - t) + p_2 t$ and p_1, p_2 are the centers of the corresponding cells. We can apply this procedure to all edges in the grid because t is independent of translation, scale, and orientation.

5.2.1 Case 1

In this case, x_1 and x_2 are constrained to the left and right sides of the cells, respectively. Therefore, $x_1 = -1$ and $x_2 = 1$. By integrating the line equations restricted to the cells, we can also write the area enclosed by this line for each cell as

$$a_1 = \int_{x_1}^{0} (mx + b)dx = \int_{-1}^{0} (mx + b)dx = b - \frac{m}{2},$$

$$a_2 = \int_{0}^{x_2} (mx + b)dx = \int_{0}^{1} (mx + b)dx = b + \frac{m}{2}.$$

Notice that y_1 and y_2 are the only free variables because a_1 and a_2 are provided by the DIF, x_1 and x_2 are constrained, and m, b are in terms of (x_1, y_1), (x_2, y_2). Furthermore, $0 \leq y_1 \leq 1$ and $0 \leq y_2 \leq 1$. If y_1 and y_2 are outside of this range, then the values of a_1 and a_2 are incompatible with the way the line intersects the edges of the cell and must correspond to one of the other three remaining cases. Hence, we test for this case by

checking if the solutions for y_1 and y_2 are within their valid ranges, which yields

$$\begin{aligned} 0 \leq \tfrac{3a_1 - a_2}{2} \leq 1, \\ 0 \leq \tfrac{3a_2 - a_1}{2} \leq 1. \end{aligned} \tag{5.1}$$

If Equation 5.1 is true, then the intersection of our estimated line with the dual edge has the parameter value

$$t = \frac{a_1 - \frac{1}{2}}{a_1 - a_2}. \tag{5.2}$$

This case is identical to linear interpolation (i.e. $(1-t)a_1 + ta_2 = 1/2$).

5.2.2 Case 2

In this case, intersection points are constrained to the bottom of the left cell, $y_1 = 0$, and the top of the right cell, $y_2 = 1$. The area bounded by the line in this configuration is given by the integrals

$$a_1 = \int_{x_1}^{0} (mx + b)dx = \frac{b^2}{2m},$$

$$a_2 = 1 - \int_{0}^{x_2} (1 - mx - b)dx = 1 - \frac{(1-b)^2}{2m}.$$

Again, x_1 and x_2 are the only free variables in these equations. Furthermore, $-1 \leq x_1 \leq 0$ and $0 \leq x_2 \leq 1$ yield the conditions

$$\begin{aligned} 0 \leq 2a_1 + 2\sqrt{a_1 - a_1 a_2} \leq 1, \\ 0 \leq 2 - 2a_2 + 2\sqrt{a_1 - a_1 a_2} \leq 1. \end{aligned} \tag{5.3}$$

Intersecting this line with the dual edge provides an extremely simple solution for the parameter t.

$$t = \frac{3}{2} - a_1 - a_2 \tag{5.4}$$

Notice that a_1 and a_2 are quadratic in b, so that m and b have two solutions. In Case 2,

the solution for t is the same regardless of which solution we use. Cases 3 and 4 produce different solutions depending on the choice of m and b, but only one of these solutions is valid; that is, the line formed by m, b intersects the boundary between adjacent cells in only one of the two solutions. We show only the valid solutions to Cases 3 and 4, and these solutions are valid over the entire domain.

5.2.3 Case 3

In this case, our estimated line intersects the bottom of the left cell, $y_1 = 0$, and the right of the right cell, $x_2 = 1$. Expressing a_1 and a_2 in terms of integrals gives

$$a_1 = \int_{x_1}^{0} (mx + b)dx = \frac{b^2}{2m},$$

$$a_2 = \int_{0}^{1} (mx + b)dx = b + \frac{m}{2}.$$

In this situation, x_1 and y_2 are the free variables and must satisfy $-1 \leq x_1 \leq 0$ and $0 \leq y_2 \leq 1$. Solving for these variables gives the conditions

$$\begin{aligned} 0 \leq \tfrac{a_1 + \sqrt{a_1(a_1+a_2)}}{a_2} \leq 1, \\ 0 \leq 2a_1 + 2a_2 - 2\sqrt{a_1(a_1+a_2)} \leq 1. \end{aligned} \tag{5.5}$$

Finally, intersecting this line with the dual edge gives the intersection parameter

$$t = 1 - \frac{2a_2 - 1}{8a_1 + 4a_2 - 8\sqrt{a_1(a_1+a_2)}}. \tag{5.6}$$

5.2.4 Case 4

The final case we consider is the situation in which the estimated line intersects the left side of the left cell, $x_1 = -1$, and the top of the right cell, $y_2 = 1$. Again, we express a_1

and a_2 in terms of integrals, which yields

$$a_1 = \int_{-1}^{0} (mx + b)dx = b - \frac{m}{2},$$

$$a_2 = 1 - \int_{0}^{x_2} (1 - mx - b)dx = 1 - \frac{(1-b)^2}{2m}.$$

For this case to be valid, $0 \leq y_1 \leq 1$ and $0 \leq x_2 \leq 1$. To simplify the equations and show the symmetry with Case 3, we substitute $\bar{a}_1 = 1 - a_2$ and $\bar{a}_2 = 1 - a_1$. Solving for y_1, x_2 gives the conditions

$$\begin{aligned} 0 \leq \tfrac{\bar{a}_1 + \sqrt{\bar{a}_1(\bar{a}_1 + \bar{a}_2)}}{\bar{a}_2} \leq 1, \\ 0 \leq 2\bar{a}_1 + 2\bar{a}_2 - 2\sqrt{\bar{a}_1(\bar{a}_1 + \bar{a}_2)} \leq 1. \end{aligned} \tag{5.7}$$

When these conditions are met, we intersect our estimated line with the dual edge and find that

$$t = \frac{2\bar{a}_2 - 1}{8\bar{a}_1 + 4\bar{a}_2 - 8\sqrt{\bar{a}_1(\bar{a}_1 + \bar{a}_2)}}. \tag{5.8}$$

5.2.5 Efficient Case Selection

From the previous description, it appears that 16 checks are necessary to choose the correct case (4 inequalities are shown for 4 cases). However, we can reduce this number to 2 simple checks against lines with one additional check that we perform $1/3$ of the time.

The two boundary half-spaces ($a_2 \leq 3a_1$ and $3a_2 \leq a_1 + 2$) of Case 1 segment the domain into four regions (shown in Figure 5.3). The region in which the half-spaces intersect is exactly Case 1. Notice also that the region that intersects neither half-space contains only Case 2. Each of these large regions occupies $1/3$ of the domain. The remaining regions contain Case 2 and either Case 3 or Case 4. We differentiate these smaller regions by a single additional test against a quadratic. Specifically, we select Case 3 if

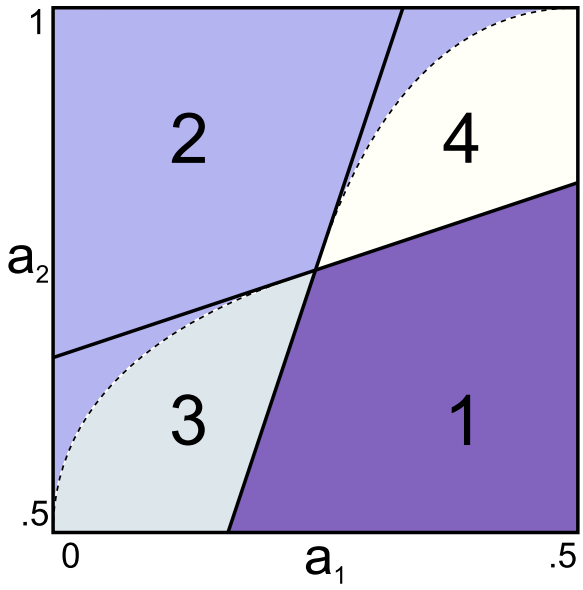


Figure 5.3: The half-spaces (solid lines) used in the check for Case 1 segment the domain into 4 regions. The two smaller regions are partitioned by quadratics (dashed lines).

$(2a_1 + 2a_2 - 1)^2 < 4a_1(a_1 + a_2)$, and we select Case 4 if $(2\bar{a}_1 + 2\bar{a}_2 - 1)^2 < 4\bar{a}_1(\bar{a}_1 + \bar{a}_2)$.

5.2.6 2D Summary

To summarize our algorithm in 2D, we check to see if there is an intersection along each dual edge by determining whether the values of the cells span $1/2$. If so, we use the method described in Section 5.2.5 to determine which case the cell values correspond to. We then compute the parameter value t along the edge using the corresponding formula for t in Equation 5.2, 5.4, 5.6, or 5.8 to find the intersection point and to generate the topology of the surface in each dual cell using the MC algorithm.

Figure 5.3 shows the domains of the four cases according to Equations 5.1, 5.3, 5.5, and 5.7. From the figure, one can see that the domains are disjoint except along the shared boundary of the cases and that the domains cover the entire valid parameter range for a_1, a_2. Hence, for a given set of values a_1, a_2 in a DIF, only one of the four cases will apply.

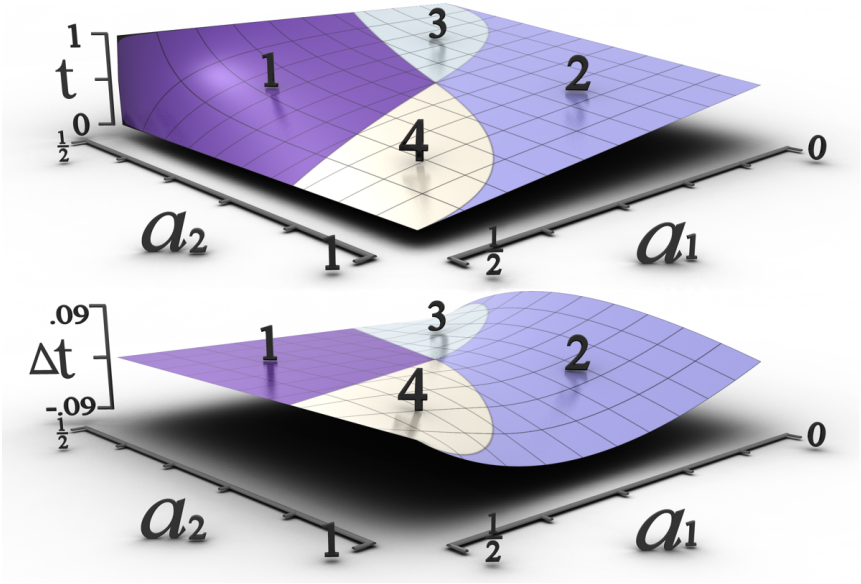


Figure 5.4: For each of the four cases, we color the values of a_1 and a_2 that correspond to that case and plot the function value of t provided by that case (top). The bottom graph shows the difference between our function and the values provided by linear interpolation.

The top of Figure 5.4 also shows a plot of the value of t given by the corresponding cases and shows that the function is smooth when transitioning between cases. Notice also that Case 1 corresponds exactly to linear interpolation, which produces the discontinuity as a_1, a_2 both approach $1/2$.

Figure 5.4 (bottom) shows the difference between parameter values found using our intersection function and standard linear interpolation. Since Case 1 is identical to linear interpolation, the difference is 0. However, the other three cases are different from linear interpolation, and our function diverges smoothly from linear interpolation up to nearly a tenth of a grid cell. Our method reproduces straight lines exactly because we calculate the intersection of lines with cells. Therefore, linear interpolation underestimates the distance to linear contours when a_1 is small. Likewise, linear interpolation overestimates the distance to linear contours when a_2 is large.

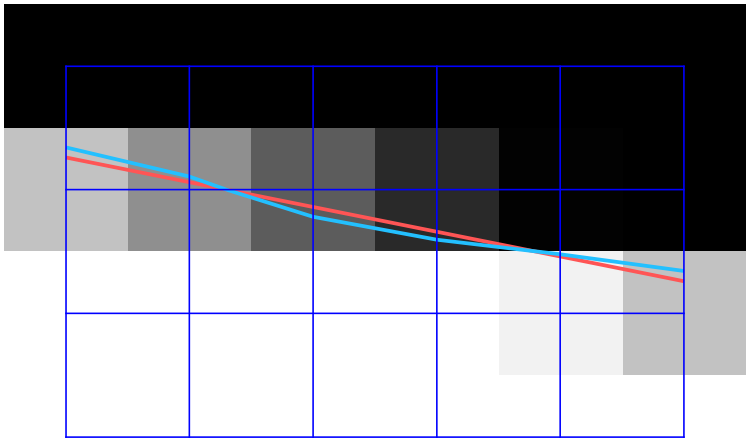


Figure 5.5: Values of the DIF calculated from a linear function are shown in gray-scale. From the DIF, we calculate contours using our method (red) and MC (cyan) where vertices are placed along the blue lines of the dual grid.

Figure 5.5 shows a 2D example in which we reconstruct a straight line from a DIF. The red line represents our method and the light blue line represents the contour created by MC using linear interpolation. Our method exactly reproduces the original linear function represented by the DIF. The oscillation effect that is apparent in the MC reconstruction results in normals that vary significantly from those of the original surface.

Although we reproduce lines exactly, we do not preserve the area of objects along curved boundaries. The linear functions between pairs of cells that we show in Figure 5.2 preserve the areas a_1 and a_2, and we place our vertex at the point where these lines intersect the dual edges. However, when we connect vertices using the MC table, we do not reproduce the areas in the cells if the resulting contour is not a straight line. In general, our method generates shapes that underestimate areas in regions with positive curvature and overestimate areas in regions with negative curvature.

5.3 Extension to 3D

In 2D we were able to derive the intersection function defined in Figure 5.4 because the equation for a line has two degrees of freedom. The two areas in adjacent cells, a_1 and a_2, therefore determine a unique line. However, a 3D solution is more complicated because a plane has three degrees of freedom and is not fully constrained by two samples. Although only one more grid sample is required to determine a plane, there is no symmetric choice of a single additional sample. Preserving symmetry therefore requires all neighbor cells to be considered and overly constrains the problem. We could find a separating plane using a non-linear least squares minimization, but we propose a simpler solution.

Given an edge of the 3D dual grid, let a_1 and a_2 represent the occupancies of the cubes that the edge connects. If $a_1 < 1/2 \leq a_2$ or $a_2 < 1/2 \leq a_1$, the surface intersects that edge, and we place a vertex on the edge using the 2D method described in Section 5.2. Specifically, we choose between the four cases using a_1, a_2 as described in Section 5.2.5 and find t using Equations 5.2, 5.4, 5.6, and 5.8.

The reason this works is that one can imagine extruding Figure 5.2 out of the page into 3D. When the separating plane aligns with a Cartesian axis, the 3D case reduces to 2D. This 2D reduction is imperfect for unaligned planes, but still provides a reasonable approximation. Although we cannot reproduce all 3D linear functions exactly from just a_1 and a_2, we find that this simple, efficient technique works remarkably well in practice.

5.4 Analysis

Our algorithm exactly reproduces two dimensional lines. This property is important because it is in flat regions where aliasing patterns are most noticeable. Moreover, accurate reproduction of normals is especially important in 3D because we perceive surfaces mainly through the way light interacts with the surface, which depends on the normals of the object.

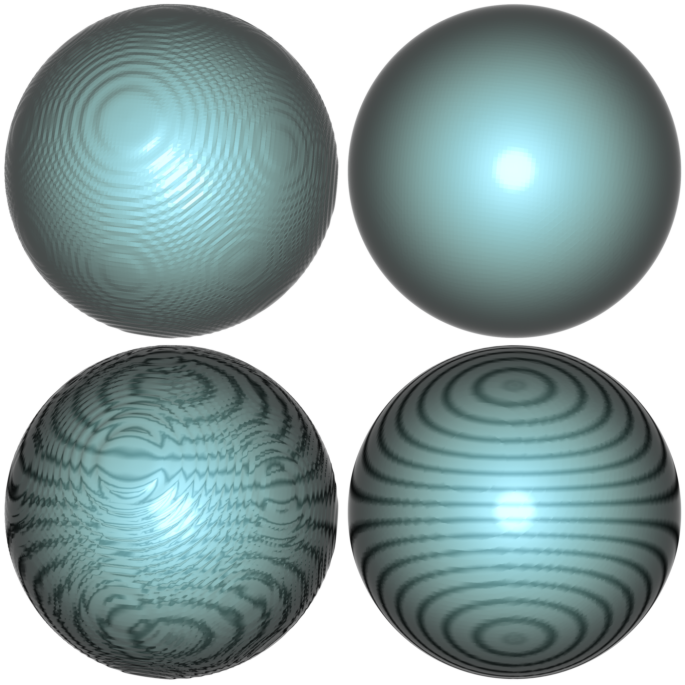


Figure 5.6: When lighting and reflection are applied to 3D surfaces, the ripple patterns produced by Marching Cubes (left) are much more obvious than with our method (right). Top: surfaces using diffuse lighting and specular highlights. Bottom: reflection lines.

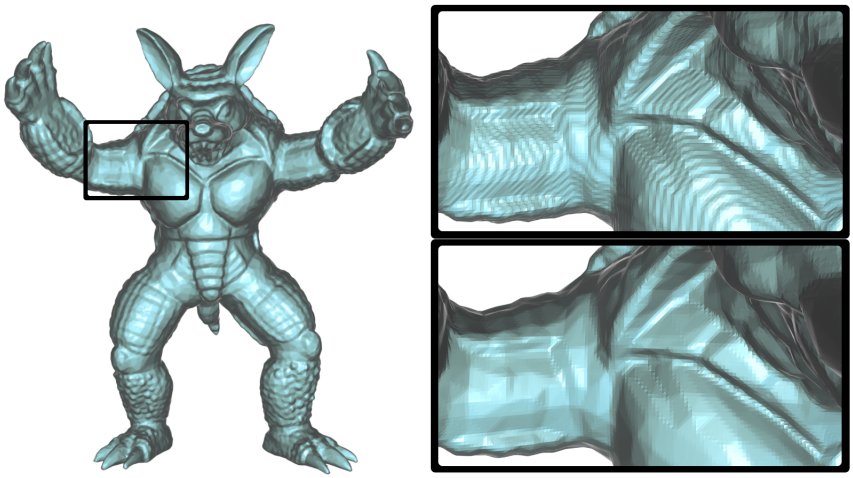


Figure 5.7: Our interpolant can find accurate surface intersections for complicated discrete indicator functions like an armadillo man sampled over a 512^3 grid. For the highlighted region, we show a zoomed picture of the surface found using linear interpolation (top) and the surface found using our interpolant (bottom).

A visual comparison of the MC contours and our contours shown in Figures 5.1, 5.6, and 5.7 indicates that applying the 2D equations to 3D works well. For example, Figure 5.7 shows an example of reconstructing an armadillo man from a DIF using MC (top) and our method (bottom). The MC surface is extremely aliased and obscures small details in the model, whereas our reconstruction produces a much higher quality surface that does not obscure small details in the reconstructed surface. It is even possible to see the edges from the polygons of the original surface used to create the DIF in our reconstruction.

Beyond providing visual comparisons between methods, we quantitatively compare surfaces with different curvatures. We compare MC and our method on surfaces of constant curvature by looking at spheres that vary in radius from one to thirty cell widths. We calculate the percent occupancy of each boundary cell over a million samples and then compare the difference between the reconstructed surfaces and the true sphere by firing ten million rays from the center of the sphere in different directions. For each ray, we measure the distance between intersections of the reconstructed surfaces and the ideal sphere as well as differences in their normals. We eliminate bias introduced from the position of the sphere with respect to the grid by adding random offsets to the spheres averaged over one hundred trials.

Figure 5.8 shows the difference in the error of surface normals between the surfaces created using our intersection function and MC as curvature decreases. This figure shows both the maximum and average differences between normals of the extracted surfaces and the analytic spheres. The error for the MC surface is shown in blue and the error for our surface is shown in red. As the radius of the spheres increases, the surface becomes locally planar and the average error in the normals of our surface almost vanishes, whereas the error for the MC surface remains constant. In fact, for a sphere with a radius of fifteen cells or more, even the maximum error from our method is as good as or better than the average error for surfaces produced by MC.

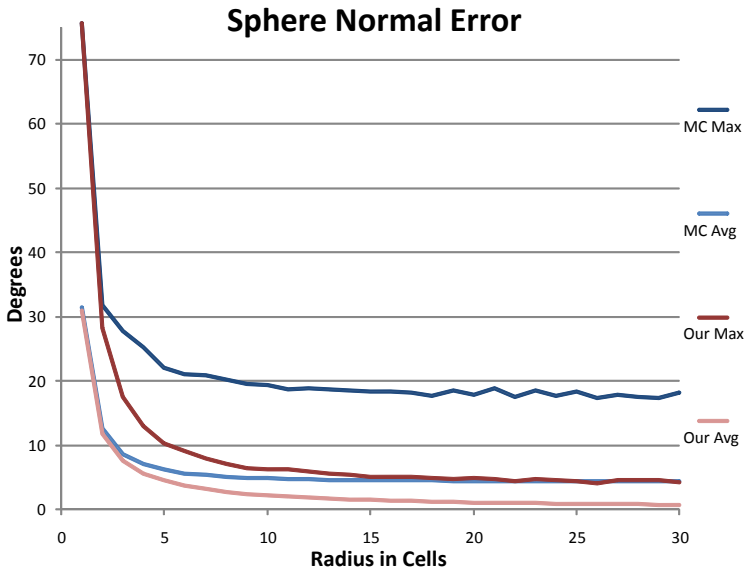


Figure 5.8: The maximum and average errors in normal direction are plotted for spheres with radius varying from one to thirty cells. Once the radius is approximately fifteen, the maximum normal error from our method is equal to the average error of Marching Cubes.

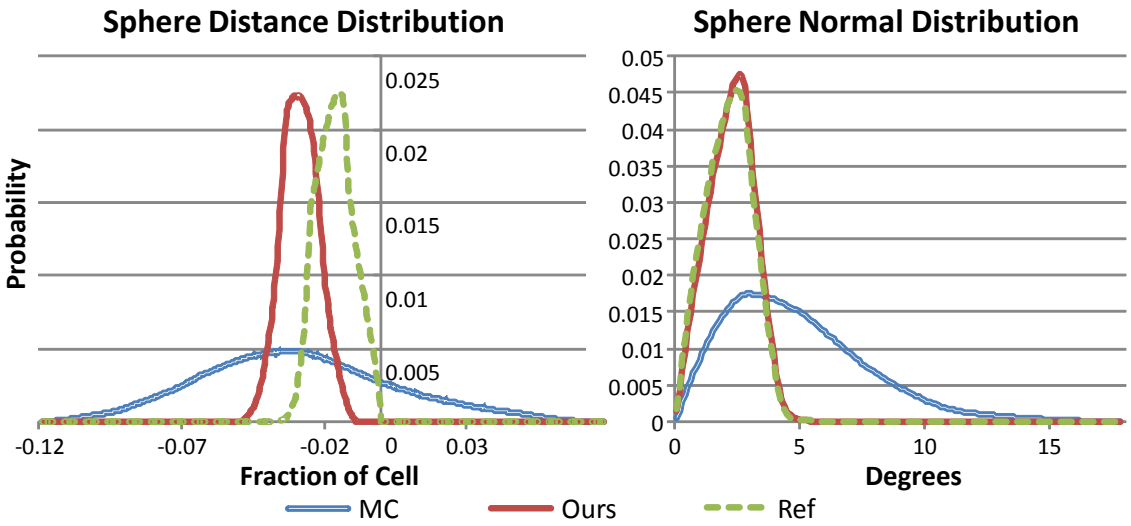


Figure 5.9: For a sphere with radius of ten cells, we calculated contours that positioned vertices using linear interpolation (MC), using our intersection function (Ours). We also measure the error from placing vertices directly on the sphere's surface (Ref). The distribution of distances from an exact sphere is shown on the left, and the distribution of normal errors is shown on the right.

Figure 5.9 shows the distribution of error in positions and normals for spheres with a radius of ten cells. We compared the MC surface (blue), our surface (red), and a reference surface with vertices directly on the surface of the sphere (dashed green). Even the reference surface underestimates the radius of the sphere by an average of about 0.02 cells because vertices on the surface are connected with planar polygons. Our method and MC both underestimate the radius of the sphere by approximately 0.03 cells, but our method has a much smaller standard deviation like that of the reference surface. Our contour is very accurate for nearly planar surfaces (spheres with large radii), but linear interpolation is up to ten percent different from our interpolant, which gives MC a larger error distribution. The surface produced by our method reproduces normals of the sphere nearly as well as the reference surface does. In contrast, the normals of MC surfaces deviate significantly from those of the sphere.

In 2D, we exactly preserve areas partitioned by lines, but we do not preserve volumes partitioned by planes in 3D. To determine how accurately we preserve volumes in 3D, we calculate the occupancies of cells from planes with random orientations and positions. We contour these functions with both MC and our method and compute the absolute difference in volume between the reconstructed surfaces and the exact plane on a per-cell basis. The standard deviation of the difference in reconstructed volumes sampled over 5000 planes was 2.76% for MC, whereas the standard deviation was only 0.07% for our method. The average absolute difference in volume was 8.36% for MC and only 0.46% for our method. These results demonstrate that our method provides a good approximation of arbitrary 3D planes.

Although there is a clear statistical difference between contouring methods, these numbers do not adequately convey the displeasing banding patterns present in MC surfaces. Figure 5.6 shows a sphere generated by MC (left) and our method (right) under typical lighting conditions (top) and with a reflection map (bottom). The error in vertex positions

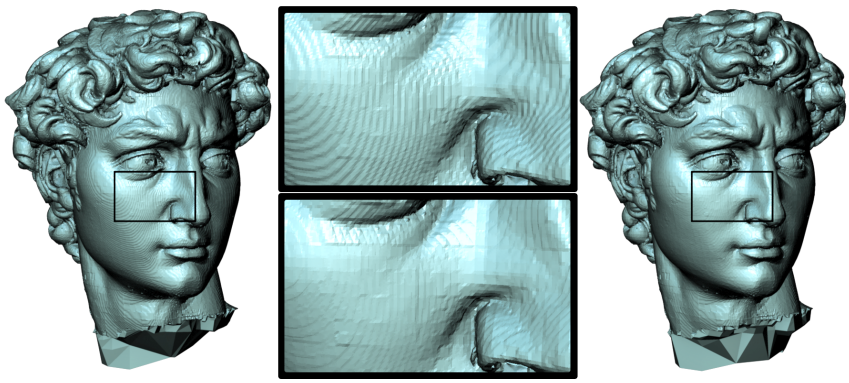


Figure 5.10: Reconstruction of the head of Michelangelo's David using MC (left and top) and our interpolant (right and bottom).

depends on the size of the grid used, but the normal error is independent of grid resolution and is clearly visible.

We have focused our analysis on spheres of various curvature, but our method works equally well for complex surfaces that have varying curvature. For example, the result of contouring a DIF of the armadillo man with our method and MC, shown in Figure 5.7, is striking. Figure 5.1 shows another complex surface that appears smoother without removing details. In both of these examples, the DIF is calculated to very high precision so that errors in contouring dominate errors in sampling. MC produces a surface with noticeable ridges and, even after applying a blur with a Gaussian kernel of sizes 3 and 7 to the DIF, the ridges in the MC surface are still faintly visible on his chest. Although blurring makes the surface smoother, it also destroys details of the original shape. Moreover, blurring the DIF [103] or performing an iterative, constrained surface fairing [58] takes a significant amount of time in addition to running MC. In contrast, our method produces a high quality surface without any iteration. We were unable to determine any significant difference in the time taken to run MC with linear interpolation and to run our method.

Although the formulation of our method assumes that the input is a perfect DIF, perfect

DIFs are hard to come by in practice. We tested our contouring method in the presence of noise by calculating an approximate DIF from laser range scans of the head of Michelangelo's David. We computed the DIF using the wavelet reconstruction method described in Chapter 4. The DIF is imperfect, both because of noise in the scanned data and from structured noise resulting from the scanned surface being open at David's neck. Notice in Figure 5.10 that ridges in the surface are greatly reduced in our surface compared to MC, but the improvement in quality is less than with perfect DIFs. Block artifacts from using a Haar basis are visible in both contours but are partially obscured by the ridges from MC.

5.5 Conclusion and Future Work

Discrete indicator functions are an important class of implicit functions that require special consideration to produce accurate contours. Replacing the interpolation function used in MC is both simple and effective. The apparent visual improvement in the surfaces generated by our method over MC is confirmed by our statistical analysis. We note that our method is specific to DIFs and should not be applied to arbitrary functions, and linear interpolation typically suffices in these situations. However, DIFs often arise in practice and, for these functions, our method greatly improves the quality of the extracted surface with essentially zero cost.

One drawback of our method is that we do not preserve the volume of cells that intersect curved surfaces. Most methods that preserve volume do so by performing a global optimization. However, we may be able to extend our method to estimate the local curvature of a surface from the DIF or locally fit a curved object like an ellipsoid to calculate a volume preserving surface. Furthermore, our method is restrictive in that we handle only two materials as opposed to the multiple materials considered in volume fraction methods. We plan to explore the possibility of extending our approach to multiple materials and preserving volume in the future.

We have only considered situations in which cells that intersect the contour are uniform in size, but it may be possible to apply our method to adaptive grids such as octrees. This application is more complicated because a plane passing through a large cell may pass through any of the smaller cells bordering it. Thus, more cells must be considered. Additionally, it is not clear that the problem is always well defined, even in the 2D case.

6. CARDINALITY-CONSTRAINED TEXTURE FILTERING*

Artists often apply images, called textures, to the surface of three-dimensional models to add visual interest. However, care must be taken when displaying images on a model, because there is not a one-to-one correspondence between the texels (texture elements) and pixels of the display. When a model is in the distance and several texels correspond to each pixel, poor sampling can cause the image to be aliased. Filters that are effective at removing aliasing without overblurring sum over a greater number of texels, which makes them expensive to compute. Directly adding all samples that fall under the filter support becomes impractical for distant objects because we must sum over a number of texels proportional to the squared distance.

Rendering algorithms typically use image pyramids called mipmaps [134] to accelerate image filtering. Mipmaps consist of precalculated images downsampled at power-of-two resolutions and can be used to compute filters in constant time, regardless of the scaling factor. We present a method that combines texels in a mipmap to reproduce the results of low-pass filters while only reading a few texels per sample. Our insight is twofold. Rather than interpolating colors between single points so that colors are exact at those points but poor everywhere else, we find weights that give good results over all possible sample points. Our second insight is that we can combine texels from any mipmap resolution. Given a sampling filter, the prefilter used to construct the mipmap, and a texel budget, we can solve for which texels to use and which weights best reproduce the sampling filter.

Memory bandwidth is often a bottleneck in graphics applications, so we attempt to use the bandwidth as efficiently as possible. Our method can also scale the number of texel

*Reprinted with permission from "Cardinality-Constrained Texture Filtering" by Josiah Manson and Scott Schaefer, 2013. ACM Transactions on Graphics, 32 (4), 140:1–140:8, © 2013 The Association of Computing Machinery.

(a) Input (b) Exact (c) 8 Texels (d) Trilinear

Figure 6.1: A comparison of Lánczos 2 filter approximations showing (a) the 1024^2 input image downsampled to a resolution of 89^2 pixels using (b) an exact Lánczos filter, (c) an eight texel approximation using our method, and (d) trilinear interpolation applied to a Lánczos filtered mipmap. Our approximation produces an image that is nearly the same as the exact filtered image while using the same number of texels as trilinear interpolation.

reads per sample to match the available bandwidth. By carefully choosing which texels to use, we accurately reproduce image filters that are sharp and free of aliasing for all scales, translations, and rotations of an image. Furthermore, we can approximate high-quality filters such as the Lánczos 2 filter in real-time, because the size and complexity of a filter only affects the preprocessing time to calculate coefficient tables of filters and to generate mipmaps. We show an example in Figure 6.1 in which we approximate a Lánczos 2 filter and compare the resulting image to those obtained by exact evaluation of the filter or trilinear interpolation of the mipmap.

When sampling a texture, we measure the distortion of each pixel into texture space. Isotropic filtering assumes that distortions scale the pixel, whereas anisotropic filtering allows pixels to stretch. When viewing three-dimensional surfaces at oblique angles, anisotropic filtering improves image quality but reduces to isotropic filtering in perpendicular views. We focus our attention on improving the quality of isotropic filtering, and we describe how our method applies to anisotropic image filtering at the end of the chapter.

6.1 Related Work

Most real-time rendering algorithms use mipmapping [134, 18] to sample textures. Mipmapping reduces aliasing by precalculating downsampled images at several resolutions with a low-pass filter. Because sample positions do not typically coincide with texel centers, GPUs use trilinear interpolation to calculate colors between texels. Mipmapping allows sampling algorithms to be independent of scale while using only 33% more memory than the input image.

There is surprisingly little literature on how to improve upon mipmapping for isotropic filtering. The attention of researchers has instead focused on how to improve anisotropic texture filtering [35, 60, 65, 72, 118, 20, 77, 99, 21, 27, 141, 97]. Although these methods are designed to improve anisotropic filtering, some of the methods also improve isotropic filtering. Summed area tables [35] accurately calculate axis-aligned box filters, but at the cost of significantly more memory usage. For example, 28 bits instead of 8 bits are required per color channel for a 1024^2 image to avoid loss of precision, and storage increases by 250% compared to 33% for a mipmap. Elliptical weighted averaging (EWA) [65] samples with a Gaussian filter, and Heckbert uses a mipmap to accelerate EWA [72] by reading between 9 and 36 texels from one resolution. Another method stores tables of texel weights for box filters when sampling from a single mipmap level [77].

In contrast, our approach combines a fixed number of prefiltered texels at different resolutions to generate a filter at arbitrary scales. Researchers have explored the idea of reproducing filters by combining texels from images at different resolutions [17], but for the purpose of feature detection rather than fast image sampling. Wavelet theory [96] also combines multiresolution basis functions into arbitrary sampling filters, but building a filter from wavelets requires summing a number of basis functions that are logarithmic in the scale of the filter. In order to reproduce sampling filters with a constant number of

basis functions, we use scales and translates of the filter functions as our basis.

NIL mapping [56] computes filters through adaptive quadrature, a recursive process of refining a filter in areas of high approximation error. Because NIL mapping stops recursion once a sampling limit is reached, the filter is computed in constant time. Although NIL mapping uses multiple resolutions over the support of the filter, the color of a texture sample depends only on texels from one resolution at any point. Also, NIL mapping computes texel weights directly from the filter function rather than choosing values to minimize approximation error. The resulting algorithm is somewhat slow, difficult to implement on a GPU, and has higher error than necessary.

An alternative approach for constant time filtering is to optimize for the best set of basis functions to reconstruct a filter [63] rather than optimizing for the coefficients of a fixed basis. The authors optimized a set of basis functions to represent rotations and non-uniform scales of a Gaussian filter around a point. They do not include translations of the filter in their optimization and do not discuss how they could use a mipmap-like hierarchy of resolutions, which limits the scalability of their method.

6.2 Multi-resolution Sampling

We wish to sample an image $\hat{I}$ defined over the $[0,1]^2$ domain using a low-pass filter h. To compute the color of a pixel with scale $\hat{s}$ and translation $\hat{t} = (\hat{t}_0, \hat{t}_1)$ relative to $\hat{I}$, we transform h to match the position and scale of the sample by $h_{\hat{s},\hat{t}}(x) = 2^{\hat{s}} h(2^{\hat{s}}(x - \hat{t}))$ so that the color of the sample $v_{\hat{s},\hat{t}}$ integrated over points $x = (x_0, x_1)$ is

$$v_{\hat{s},\hat{t}} = \iint_{\mathbb{R}^2} \hat{I}(x) h_{\hat{s},\hat{t}}(x)\, dx. \tag{6.1}$$

Directly computing $v_{\hat{s},\hat{t}}$ is costly when the support of $h_{\hat{s},\hat{t}}$ is large, so we approximate this integral for real-time rendering. In particular, the time taken to sample an image should be independent of the position and scale of $h_{\hat{s},\hat{t}}$ so that we always read a constant number of

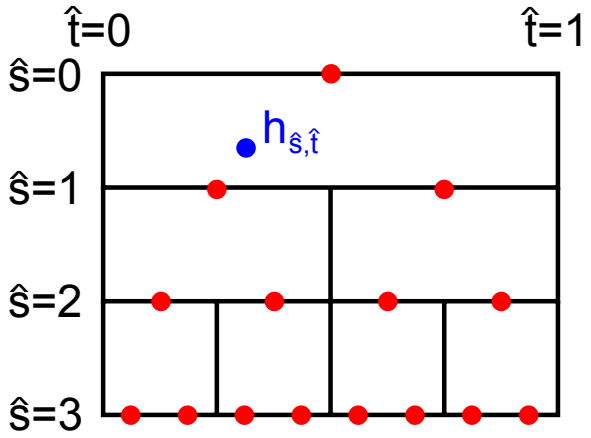


Figure 6.2: A two-dimensional depiction of the reference coordinate system. Texels in the mipmap are shown as red dots, and the coordinate of a possible filter $h_{\hat{s},\hat{t}}$ is shown in blue.

texels. For scale independence, we store downsampled images in a mipmap image stack I and compute the texels $I_{\mathcal{S},\mathcal{T}} = v_{\hat{s},\hat{t}}$ using the same filter $h_{\hat{s},\hat{t}}$ with which we want to sample. Although we could generate the texels $I_{\mathcal{S},\mathcal{T}}$ with a filter other than $h_{\hat{s},\hat{t}}$, using $h_{\hat{s},\hat{t}}$ ensures that we can exactly compute $v_{\hat{s},\hat{t}}$ at texel samples. The set of coordinates E of mipmap samples are the standard cell-centered positions, which have integer coordinates $\mathcal{S}, \mathcal{T}$ that we relate to positions in the mip-volume by $\hat{s} = \mathcal{S}$ and $\hat{t} = 2^{-\mathcal{S}}(\mathcal{T}_0 + \frac{1}{2}, \mathcal{T}_1 + \frac{1}{2})$. We visualize the $\hat{s}, \hat{t}$ coordinate system in Figure 6.2, with the positions of texels shown as red dots and a hypothetical sampling query for $h_{\hat{s},\hat{t}}$ shown in blue.

We perform an optimization to approximate $v_{\hat{s},\hat{t}}$ by reading a subset of texels $e \subset E$. Our optimization for the coefficients c_i of the texels $e_i \in e$ has the cardinality constraint that $|e| = n$, where n is a fixed sample budget. Solving a cardinality-constrained optimization is proven to be NP-hard [133] because we must test all possible solutions to find the minimal solution. In higher-dimensional problems, the number of basis functions, and therefore the number of combinations of basis functions, becomes too large to check exhaustively. However, we show how we can efficiently approximate this solution

(a) Input (b) Exact (c) Constant Precision (d) No Constant Precision

Figure 6.3: The difference between enforcing constant precision when downsampling an image and not enforcing constant precision with Lánczos 2 filtering. Public domain US Air Force photo with ID 050119-F-7709A-023, by Master Sgt. Michael Ammons.

in Section 6.2.2.

Another constraint is that our filter should reproduce constant functions (i.e., have constant precision) to prevent distracting patterns from appearing in constant and nearly constant regions of an image. A filter has constant precision when $\sum c_i = 1$. We demonstrate the importance of constant precision in Figure 6.3. Compared to the image downsampled using an exact Lánczos 2 filter, our approximation without constant precision does not reproduce the brightness of the input image. Lack of constant precision also introduces a pattern in the sky where the color should be nearly uniform.

We can write the constrained optimization for the best set of coefficients c and texels e to approximate $v_{\hat{s},\hat{t}}$ as

$$\underset{\substack{c,e \subset E \\ \sum c_i = 1, |e| = n}}{\operatorname{argmin}} \iint_{\mathbb{R}^2} \left(\hat{I}(x) h_{\hat{s},\hat{t}}(x) - \sum_{i=1}^{n} \hat{I}(x) h_{e_i}(x) c_i \right)^2 dx. \tag{6.2}$$

This optimization depends on the values of $\hat{I}$, but we wish to precalculate coefficients that are independent of the input image so that we can quickly compute the filter later. Notice that $\hat{I}$ weights the importance of reproducing the shape of the filter at point x. To give the

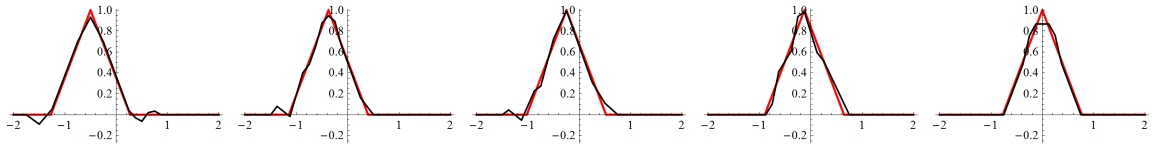
best result when $\hat{I}$ is unknown, we give all values of x equal weight, which simplifies the minimization to

$$\underset{\substack{c,e\subset E\\ \sum c_i=1,|e|=n}}{\operatorname{argmin}} \iint_{\mathbb{R}^2} \left(h_{\hat{s},\hat{t}}(x) - \sum_{i=1}^{n} h_{e_i}(x)c_i\right)^2 dx. \tag{6.3}$$

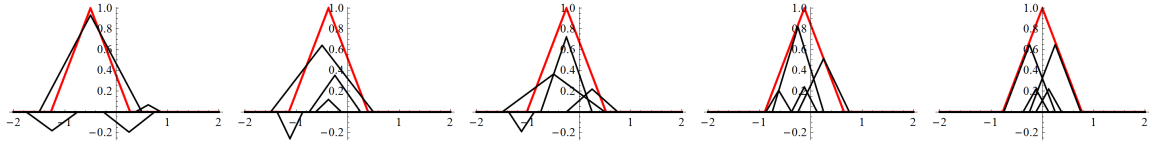
To understand the properties of two-dimensional filters, we analyze the optimization in Equation 6.3 for one-dimensional filters, which are easier to visualize. For a two-dimensional image, trilinear interpolation interpolates over $\hat{t}_0$, $\hat{t}_1$, and $\hat{s}$ to approximate arbitrary filters, and the equivalent one-dimensional process interpolates over $\hat{t}_0$, and $\hat{s}$.

We illustrate how we can approximate filters accurately in Figure 6.4. We approximate translations of the one-dimensional tent filter shown in red using weighted combinations of the black basis functions. We show h halfway between mipmap levels at translates of $0/8$, $1/8$, $2/8$, $3/8$, and $4/8$ texels. Both our method and bilinear interpolation over $\hat{t}_0$ and $\hat{s}$ use four basis functions. We show the result of our method in (a) and show the result of bilinear interpolation in (c). Our method reproduces the filter better than bilinear interpolation. We show the basis functions multiplied by the coefficients used to approximate the filter beneath the approximation in (b) and (d). Our method maintains image sharpness by sampling from higher-resolution basis functions to shape the filter. The different translations in (b) also show how the optimal strategy for approximating a filter depends strongly on the parameters of $h_{\hat{s},\hat{t}}$. On the far left, the best solution is to subtract the sides from a basis function that is wider than $h_{\frac{1}{2},0}$, whereas on the right, the best solution is to add high-resolution basis functions to approximate $h_{\frac{1}{2},\frac{1}{2}}$. For intermediate translations, a combination of both approaches works best.

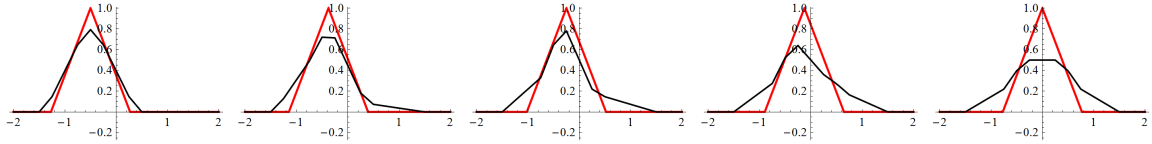
In Figure 6.5, we show the approximation error of our method in blue when h is a one-dimensional tent filter, compared to the error of bilinear interpolation, which is shown in black. We evaluated the errors in the graph for translations over the width of a texel $[0, 1]$ at an integer mipmap resolution and plotted the error for unique subsets of four texels



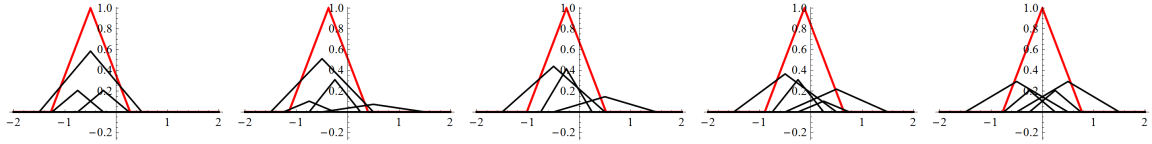
(a) Optimal approximation



(b) Optimal approximation functions used



(c) Bilinear interpolation



(d) Bilinear interpolation functions used

Figure 6.4: A one-dimensional example of how our optimization improves over bilinear and trilinear interpolation using the same number of basis functions. The filter we approximate is shown in red and the four basis functions or their sums are shown in black. Filters are sampled halfway between integer mipmap resolutions at translates of $0/8$, $1/8$, $2/8$, $3/8$, and $4/8$ of a texel.

in green. None of these subsets has the lowest error over the entire domain, so finding the optimal solution shown in blue required minimizing the error for all of the possible subsets at each point and choosing the subset with the least error. In Section 6.2.1, we show that we can minimize the error over regions rather than for every point. Although the problem is NP-hard, we can exhaustively check all possible combinations for this low-dimensional problem, but we have to use a heuristic method described in Section 6.2.2 for higher dimensions. Bilinear interpolation and the optimal solution both have zero error

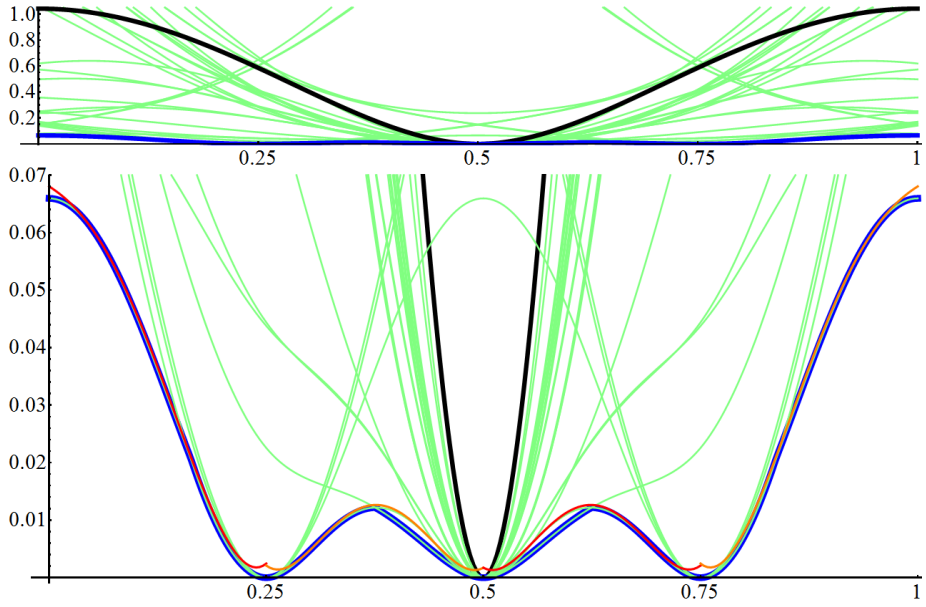


Figure 6.5: The error of bilinear interpolation is shown in black, different subsets of texels are shown in light green, optimal error is shown in blue, and error of our piecewise polynomial is shown in alternating red and orange. The bottom part of the the image shows small error values in greater detail.

at $1/2$, which is when $h_{\hat{s},\hat{t}}$ aligns with a texel center. Thus, both methods interpolate the texel values. An interesting property of our method is that, because h is a tent function in this example, we also have zero error at $1/4$ and $3/4$. This happens because tent functions have the recurrence relation that a tent function can be built from three tent functions of twice the resolution. Several of the sets shown in green have zero error at $1/4$, $1/2$, and $3/4$, because fewer than the maximum ($n = 4$) texels are required to give an solution with zero error.

6.2.1 Polynomial Fitting

For sampling to be practical in a real-time system, it is not possible to use the optimal solution for all possible sampling filters because the best set of texels to use depends strongly on the parameters $\hat{s}$ and $\hat{t}$ of the filter $h_{\hat{s},\hat{t}}$. In Figure 6.5, one can see that there is

no single best subset to use because the green lines cross at the bottom of the graph. After dividing the domain into a few pieces, we can choose a subset to fit each piece accurately. Also, texel coefficients have no closed-form solution, and I describe how to fit polynomials to the coefficients of a set of texels in this section. The error when fitting linear coefficients over four pieces in Figure 6.5 is depicted as alternating red and orange curves.

We can parameterize cells in Figure 6.2 by $(\mathbf{t}_0, \mathbf{t}_1, \mathbf{s}) \in [0,1]^3$ so that $\mathbf{s} = \hat{s} - \mathcal{S}$ and $\mathbf{t} = 2^{\mathcal{S}}\hat{t} - \mathcal{T}$, where the integer coordinates of a cell are given by $\mathcal{S} = \lfloor \hat{s} \rfloor$ and $\mathcal{T} = \lfloor 2^{\mathcal{S}}\hat{t} \rfloor$. This domain can be cut into $\mathcal{J} \times \mathcal{J} \times \mathcal{K}$ smaller subdomains D that are parameterized by $s = \mathcal{K}\mathbf{s} - \lfloor \mathcal{K}\mathbf{s} \rfloor$ and $t = \mathcal{J}\mathbf{t} - \lfloor \mathcal{J}\mathbf{t} \rfloor$. We fit sets of polynomial coefficients c_{ij} for the power basis $p(s,t)$ to the texel weights for each of the subdomains, where $j = 1 \ldots m$ indexes the power basis function. We have tested using a linear basis $p(s,t) = (1, t_0, t_1, s)$ and a quadratic basis $p(s,t) = (1, t_0, t_1, s, t_0^2, t_1^2, s^2, t_0 t_1, t_0 s, t_1 s)$. Holding the set of texels $e \subset E$ fixed and defining c_i by its polynomial expansion $c_i(s,t) = \sum_j p_j(s,t) c_{ij}$ gives the optimization

$$\underset{\substack{c_{ij} \\ \sum c_i(s,t)=1}}{\operatorname{argmin}} \iint_{\mathbb{R}^2} \iint_D \left(h_{s,t}(x) - \sum_{i=1}^{n} h_{e_i}(x) c_i(s,t) \right)^2 dt\, ds\, dx. \tag{6.4}$$

The constant precision constraint creates a linear dependence between coefficients, which allows us to replace one of the coefficients and simplify the minimization. Written in the power basis,

$$\begin{pmatrix} c_{11} \\ c_{12} \\ \vdots \\ c_{1m} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} - \sum_{i=2}^{m} \begin{pmatrix} c_{i1} \\ c_{i2} \\ \vdots \\ c_{im} \end{pmatrix}. \tag{6.5}$$

Equation 6.4 is quadratic in c_{ij}, which can be solved as a linear system. Our optimization allows freedom to choose how many texels to use, how to subdivide the domain, and what

order polynomial to use. Each option provides a tradeoff in terms of speed, memory usage, and quality. We discuss the tradeoffs and our choices in Section 6.3.

6.2.2 Combinatorics and Heuristics

Solving linear systems to find texel weights is reasonably fast, but there are many possible sets of n texels. For each subdomain, we need to find the set of texels $e \subset E$ that has the lowest error when evaluating Equation 6.4. If we choose n texels out of a pool of $N = |E|$ possible texels, then we need to check the error of $\frac{N!}{(N-n)!n!}$ combinations of texels. Clearly, we need to limit N as much as possible for the problem to be tractable. Our first observation is that we can exclude texels that are not in the support of $h_{s,t}$. Although including texels outside of the support of $h_{s,t}$ could theoretically be beneficial, the fact that we use relatively few texels makes it unlikely that they would reduce the approximation error. A second observation is that low-resolution texels are used to approximate the filter when n is small. We only use the texels from relative mipmap levels 0, 1, and 2 for a tent filter with $n = 8$, so we can exclude other resolutions from our optimization.

Even after restricting E to fewer texels, a tent filter has 189 texels from which to choose. Checking all combinations is not practical because there are 34 trillion combinations of eight texels. Exhaustively checking all combinations would take 33 years because we check approximately two million combinations per minute. We therefore developed a heuristic for determining which sets are most likely to have low error. We define the error of a texel to be the minimal error of the texel by itself in Equation 6.4. Our heuristic is that a texel basis function that matches $h_{s,t}$ with low error is likely to be in the set of functions that approximates $h_{s,t}$ with minimal error. By extension, sets of basis functions in which each function is a good approximation of $h_{s,t}$ are more likely to approximate $h_{s,t}$ well. We therefore check combinations of low-error texels before checking high-error texels.

The single-texel error defines a priority by which we order texels in a list. We select the

best n-texel subset from among the highest priority texels before progressively widening the search space to include lower-priority texels. We terminate our search after we check a desired number of combinations, and, although we can only test a small fraction of the total space for $n = 8$, we often find good solutions quickly. We checked 100 million sets for each of the six unique subdomains (using symmetry) in a $4\times4\times2$ discretization of an eight sample tent filter. In this test, we found the best of the sets checked after 15, 513, 518, 12991, 35960, 534979 trials, and we found several other sets with low errors prior to that. All of our best solutions were within the first 1% of the subsets that were checked. Therefore, our heuristic works well and we find nearly optimal sets.

6.2.3 Implementation

To implement sampling, we use two tables: an index table and a coefficient table. The index table stores the relative offsets of the n texel indices for each subdomain, and the coefficient table stores the coefficients for the texel weights. Index offsets are vectors of three integers that indicate the coordinate of the texel $(\mathcal{T}_0, \mathcal{T}_1, \mathcal{S})$ relative to the sample. Coefficients of a linear function consist of four component vectors (one constant coefficient and three linear coefficients).

Sampling $v_{\hat{s},\hat{t}}$ using the filter $h_{\hat{s},\hat{t}}$ consists of the steps:

1. Find subdomain $D \in \mathbb{Z}^3$, texel index $(\mathcal{T}_0, \mathcal{T}_1, \mathcal{S}) \in \mathbb{Z}^3$, and remainder $(t_0, t_1, s) \in [0, 1]^3$.

2. Calculate the offset into the index table and the coefficient table from the subdomain index D.

3. For all n texels,

   (a) compute the polynomial texel coefficient $c_i(s, t)$ and

   (b) add $c_i(s, t)$ times the texel color I_{e_i} into $v_{s,t}$.

A slight complication is that higher-resolution mipmaps are not available for all scales of $h_{\hat{s},\hat{t}}$, so we generate additional tables for low mipmap levels. This is similar to the difference between minification and magnification in GPUs. In our case, we use three mipmap levels when $1 < \hat{s}$, but we must optimize for two mipmap levels when $0 < \hat{s} \leq 1$ and for one level when $\hat{s} \leq 0$. In practice, there is little benefit to optimizing a single level, and we therefore revert to the reconstruction filter for $\hat{I}$ when $\hat{s} \leq 0$.

We significantly reduce the number of stored tables by taking symmetry into account. Tensor-product filters have four-fold rotational symmetry and are symmetric across the diagonal, which means that texel coefficients are uniquely defined over an eighth of the parametric space. If we subdivide the domain into $\mathcal{J} \times \mathcal{J} \times \mathcal{K}$ pieces, symmetries reduce the number of subdomains from $\mathcal{J}^2\mathcal{K}$ to $\mathcal{J}(\mathcal{J}+2)\mathcal{K}/8$. This space optimization allows us to fit precomputed tables into constant memory on a GPU.

Evaluating the color of a sample consists of a table lookup and n multiply-add operations. The overhead from finding table and texel entries based on symmetry requires $3n + 3$ *if* statements. If our method was implemented in hardware, we could handle the *if* statements more efficiently than is possible in a shader by computing the relatively simple symmetry corrections in parallel and selecting the correct symmetry with a multiplexer. We could also compress the index table significantly by using three bits per index. With this in mind, we anticipate that it would be more efficient to implement our method in hardware than in software.

6.3 Results

We graph the approximation error of our method using 2 to 10 samples compared to a directly convolved tent filter in Figure 6.6. Errors are measured by Equation 6.3 and normalized by the error of trilinear interpolation. The cost of our method depends on the number of subdomains and the order of the polynomials we fit, so we compare the

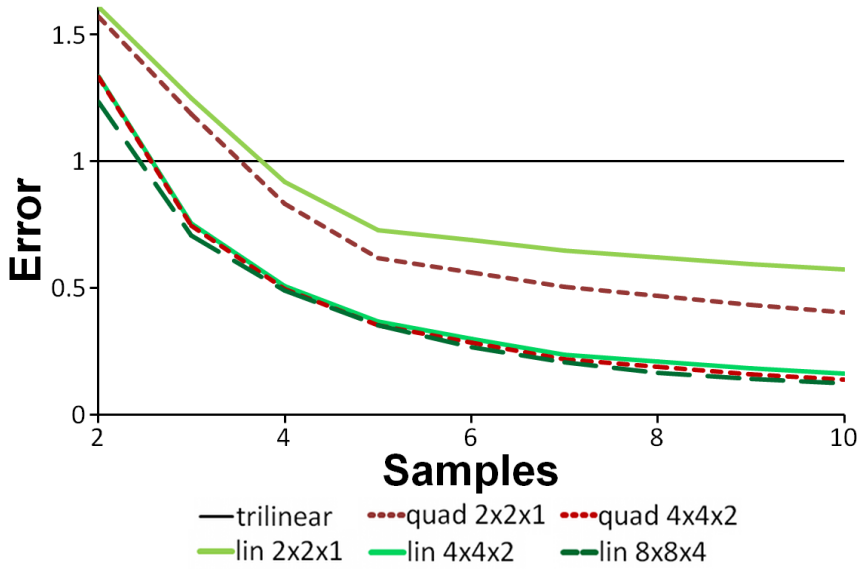


Figure 6.6: The error of approximating a tent filter using varying numbers of texels with different optimization choices is compared against trilinear interpolation. The errors are normalized so that trilinear interpolation has an error of one.

error of linear polynomials for the coefficients c_i of texels e_i over $2\times2\times1$, $4\times4\times2$, and $8\times8\times4$ subdivided domains, and quadratic polynomials over $2\times2\times1$ and $4\times4\times2$ subdivided domains. The data show that using more than $4\times4\times2$ subdomains and fitting quadratic polynomials does not significantly reduce error, so we use linear polynomials and a $4\times4\times2$ discretization of subdomains for all of our examples. Our method can approximate a variety of filters, and we compare the error of our method versus trilinear interpolation of mipmaps sampled with different filters. The errors of our method using eight texels relative to trilinear interpolation of box, tent, Gaussian, and Lánczos 2 filtered mipmaps is 0.569, 0.209, 0.142, and 0.232, respectively.

The times to calculate our filtering method compared to trilinear interpolation on an NVidia GeForce GTX 580 are shown in Figure 6.7. We give two times for trilinear interpolation; one measurement is for the native hardware trilinear interpolation exposed by

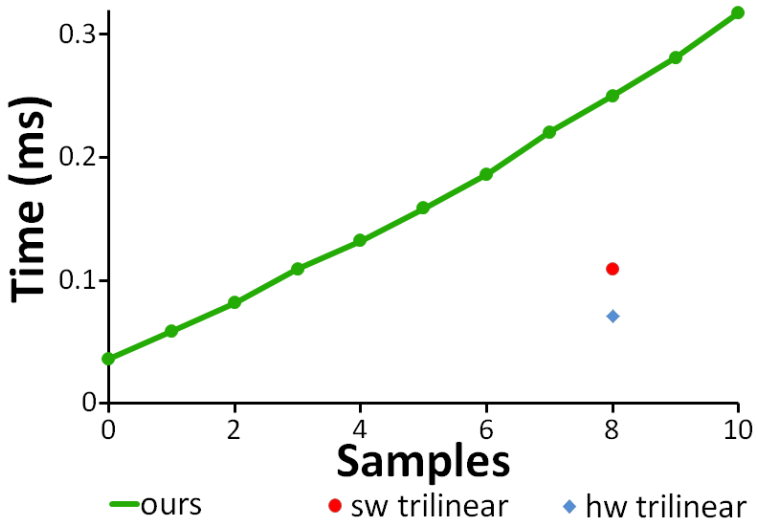


Figure 6.7: The times required to draw Figure 6.10 at 512^2 resolution using our method are compared to trilinear interpolation as implemented by the hardware (HW Trilinear) and in a GPU shader (SW Trilinear). The number of texels that the GPU reads per sample is shown on the horizontal axis.

the shading language, and the second is our shader implementation of trilinear interpolation in which we explicitly perform eight texel reads. Our timing results do not match the prediction, based on the number of mathematical operations performed, that our method should be only slightly slower per texel read than trilinear interpolation. The most plausible explanation is that we have lower throughput because trilinear interpolation has a more structured and cache-friendly pattern for accessing memory.

An example of the access pattern of our method for a tent filter using eight texels is shown in Figure 6.8. We read from three mipmap levels, whereas trilinear interpolation reads from only two levels. It is likely that GPUs optimize for trilinear accesses by using two caches for alternate mipmap levels [78] and that reading from three levels causes cache conflicts. GPUs are also likely to optimize for the 2×2 quads of texels accessed by a trilinear interpolant, whereas our reads are less regular. Our method will also issue

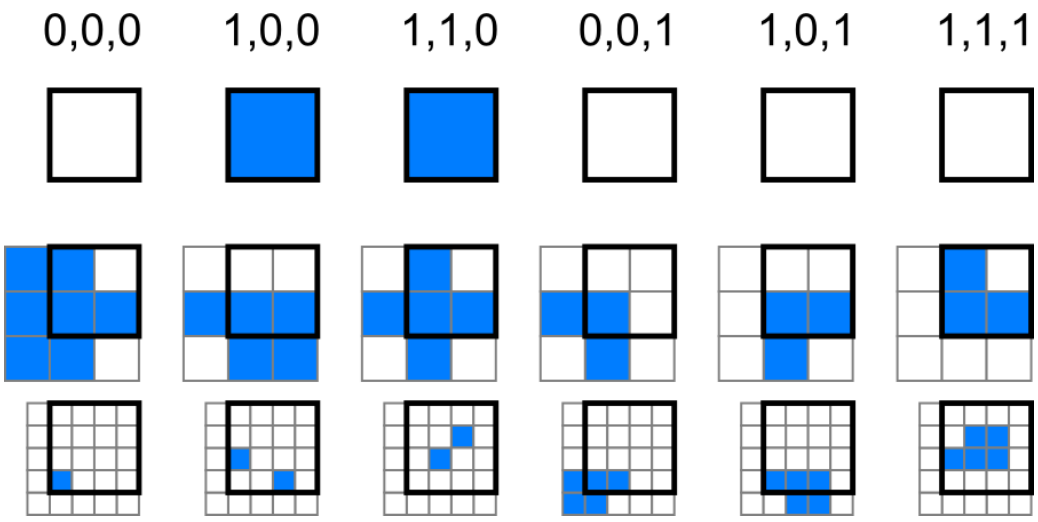


Figure 6.8: The eight texel access pattern of a 4x4x2 discretization of a tent filter is shown. The unit domain is outlined in black, and each column of images shows the texels used in a subdomain, where texels with nonzero coefficients are blue. There are only six subdomains because of symmetry, and the index of the subdomain is ordered (left to right, bottom to top, low to high resolution).

irregular reads for adjacent pixels because neighboring pixels in a $4 \times 4 \times 2$ discretization will have a stride of at least one subdomain. GPU profiling tools show that our method reads more texels than we expect and that the time taken is almost directly proportional to the number of texels read in our method, our trilinear implementation, and the hardware trilinear interpolant. Our tests are consistent between ATI and NVidia GPUs and show that a native hardware implementation significantly improves the performance of trilinear interpolation. Even our shader implementation of trilinear interpolation takes $1.5\times$ more time and bandwidth than the native hardware implementation, although the same texels are read. It is possible that hardware designed for our sampling pattern would achieve a similar result.

Figure 6.1 demonstrates that generating mipmaps with a high-quality filter is insufficient to produce sharp images at arbitrary scales when trilinear interpolation is used to

(a) Trilinear (b) 4 Texels (c) 8 Texels (d) Exact

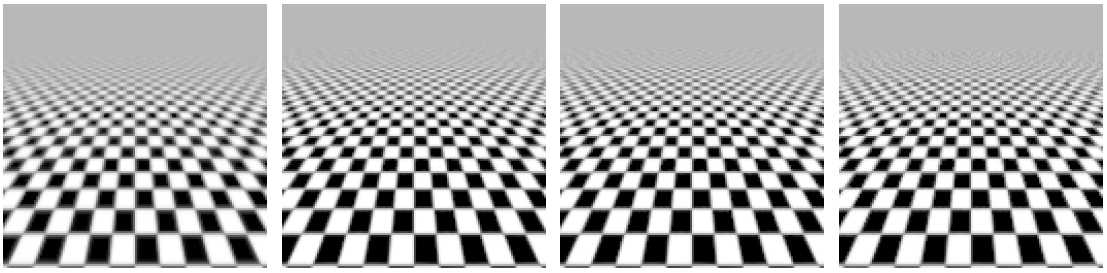
Figure 6.9: The images shown were downsampled using (a) trilinear interpolation, (b) and (c) our approximation of the Lánczos 2 filter using 4 and 8 texels, and (d) exact evaluation of the Lánczos 2 filter.

sample the mipmaps. Trilinear interpolation gives the correct filtered values when evaluated at a texel, but it does a poor job between texels, even at the same scale as the mipmap images. In contrast, our method minimizes the error over all points. We show an example of an image that we sampled between mipmap levels in Figure 6.1 using a direct convolution of a Lánczos 2 filter as the ground truth, our approximation of the Lánczos 2 filter using eight texels, and trilinear interpolation on mipmaps that were created using a Lánczos 2 filter. The Lánczos filter and our approximation of the Lánczos filter look nearly the same, but trilinear interpolation produces an image that is blurry.

Our method can be tuned to use different numbers of texels for fine-grained control over the memory bandwidth and quality of texture sampling. Figure 6.9 shows an example in which trilinear interpolation, our method with four texels, our method with eight texels, and exact evaluation of the Lánczos 2 filter on a two-dimensional image are compared. This image contains high-frequency details aligned in all directions: horizontally, vertically, and diagonally. With either four or eight texels, our method produces results similar to those obtained with the exact filter, whereas trilinear interpolation of the Lánczos 2 filtered mipmap produces a blurry image. Figures 6.6, 6.9, and 6.11 provide quantitative and

(a) Trilinear (b) Our Method

Figure 6.10: A trilinear interpolation of a Lánczos 2 filtered mipmap (a) compared against our approximation of the Lánczos 2 filter using 8 texels (b).



(a) Trilinear (b) 8 Texels (c) 6 Texels (d) 4 Texels

Figure 6.11: Aliasing of a checker pattern with ten squares on a side repeated over an infinite plane. The results of (a) trilinear interpolation and our approximation of the tent filter using (b) 8, (c) 6, and (d) 4 texels.

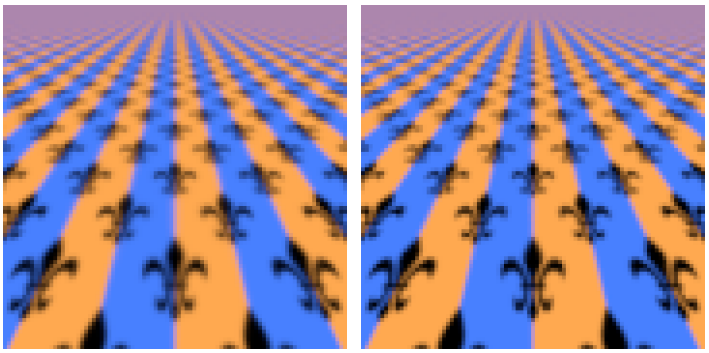
qualitative evidence that our method adapts image quality smoothly to available memory bandwidth.

Figure 6.10 compares the visual quality of trilinear interpolation with our method for a three-dimensional scene using a Lánczos 2 filter. Again, trilinear interpolation produces an image that is blurry, whereas our approximation is sharper. Figure 6.11 shows a checkerboard pattern on an infinite plane using a tent filter while reading four, six, or eight texels

to demonstrate aliasing. The texture has ten checkers on a side so that there is not an even power of two checkers to texels. Therefore, a poor filter cannot easily hide aliasing patterns. When using eight texels, the same number of texels that are read in trilinear interpolation, our filter creates sharp and clear image. The results when using six texels are almost indistinguishable from those obtained with eight texels, despite using 75% of the bandwidth. When using only four texels, the image in the distance appears slightly noisier, and the edges in the foreground appear somewhat rougher.

A possible concern is that flickering or popping artifacts will occur in animated scenes because of the piecewise nature of our method. In our tests, we have seen no obvious flickering. Although coefficients change discontinuously across subdomain boundaries, the filter we approximate changes continuously, and our approximation error is typically low enough that no artifacts are detectable. For the particularly challenging task of rotating the checker pattern in Figure 6.11, we could see a single transition line in the distance when the method samples from a very coarse resolution mipmap for a Lánczos 2 filter with $n = 4$. However, this artifact was data dependent, and we did not see the problem at $n = 4$ for other images. When $n > 5$, we did not see any artifacts in any images under animation for the Lánczos 2 filter, and when we used a tent filter, we found that we could not see the transition line with $n = 4$ because the tent filter is blurrier than a Lánczos 2 filter.

Our optimized texture samples can also be used to improve the results of anisotropic texture filters that combine isotropic samples. Hardware anisotropic filtering uses the model of Feline [99], where anisotropic filters are approximated by summing smaller isotropic filters. Feline approximates stretched Gaussians and uses trilinear interpolation to approximate isotropic samples at low computational cost. By replacing trilinear interpolation with our approximation of the isotropic Gaussians, we can generate higher-quality anisotropic filters while still using the same texture bandwidth. We compare the results



(a) Feline Trilinear (b) Feline 8 Texels

Figure 6.12: The Feline algorithm (a) using trilinear probes, and (b) using our method, which reproduces isotropic Gaussian probes more accurately.

of Feline and our improved anisotropic sampling in Figure 6.12. Using more isotropic samples in Feline improves filtering in the direction of stretch, but increasing the quality in the perpendicular direction requires better isotropic samples.

6.4 Conclusions and Future Work

Our method is valuable because memory bandwidth is often a bottleneck in graphics applications. A limitation of our method is that GPUs have been designed to optimize for trilinear interpolation and do not perform well on less structured reads. However, our method may be advantageous in certain situations. One possibility is that our method will be more suitable for offline rasterizers and ray-tracers with more flexible pipelines. Another possibility is that hardware designs will change to provide better support for random memory accesses or the access pattern of our method. Our approach can also be viewed as a stepping stone. We have shown that better filtering is possible by optimizing which texels and coefficients are used, operating under the simple assumption that cost is proportional to number of texels read. It may be possible to incorporate the current texel reading behavior of GPUs in our optimization. For example, we could optimize for reading quads

of texels using bilinear interpolation.

This chapter focuses on improving the quality of isotropic texture filtering. When displaying two-dimensional images such as in Figures 6.1, 6.3, and 6.9, or when viewing a surface straight-on, anisotropic filtering does not apply. It is possible to improve anisotropic filtering by replacing isotropic probes used in current hardware with our method as in Figure 6.12, but we could also apply the principle of optimizing for the best set of texels and their coefficients to anisotropic texture filtering. This strategy could improve sampling quality relative to the number of texels used by reducing the number of redundant texel reads. The challenge of extending our method to anisotropic filters is that the dimensionality of the optimization increases from three to five dimensions because we must include stretch and orientation of the filter, which, in turn, increases the complexity of the optimization. The idea of simultaneously optimizing basis functions and their coefficients for filter reproduction [63] has the potential to produce even better results when combined with our idea of optimizing which texels are use from different resolutions; although the simultaneous optimization may be complex to solve.

7. PARAMETERIZATION-AWARE TEXTURE FILTERING*

Movies and games are filled with three-dimensional objects represented as triangle meshes. The geometry of these meshes is important, but much of the detail and interest of an object comes from the variation of colors on its surface. Texture mapping provides a way of annotating surfaces with information such as color. Typically, the color at any point on a model is calculated from an image, called a texture, that is applied to the object. Textures are stored as two-dimensional grids of color samples (texels), but there is no obvious way of automatically mapping a point $p \in \mathbb{R}^3$ on a three-dimensional surface to a point $t \in \mathbb{R}^2$ in a texture. Instead, a parameterization $\Theta : \mathbb{R}^3 \to \mathbb{R}^2$ of the surface of the object is usually supplied by an artist, and the surface of the object is cut into separate charts that are flattened into the plane. For triangle meshes, Θ is typically encoded as texture coordinates associated with each vertex of the triangles.

The projection of a point p on a surface to a pixel s on the screen is given by $\Phi : \mathbb{R}^3 \to \mathbb{R}^2$. To draw a textured surface, the graphics card (GPU) samples the value in the texture associated with a triangle at the coordinate $\Theta \circ \Phi^{-1}(s)$ in order to determine the color of the pixel. However, a pixel on the screen may correspond to a large area of the texture for distant objects so that filtering must be applied to avoid aliasing. Depending on the distance to an object, a filter h may integrate over many texels. Rather than compute exact filter integrals, GPUs use precalculated downsamplings of the texture that are stored in a mipmap [134].

Filters can be expensive to evaluate even without distortion, but evaluation is more complicated over the distorted support of $h(\Theta \circ \Phi^{-1}(s))$. Several algorithms have been

*Reprinted with permission from "Parameterization-Aware MIP-Mapping" by Josiah Manson and Scott Schaefer, 2012. Computer Graphics Forum, 31 (4), 1455–1463, © 2012 The Author(s) Computer Graphics Forum © 2012 The Eurographics Association and Blackwell Publishing Ltd.

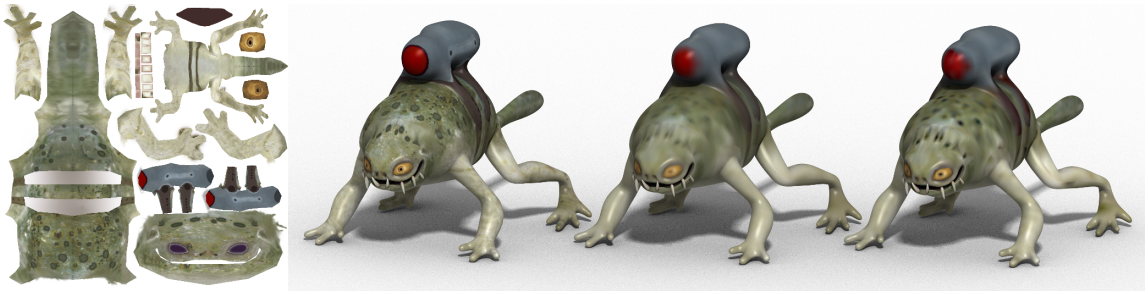


Figure 7.1: From left to right, we show the input texture (1024^2), the monster frog model drawn with the full-resolution texture, the fourth mip-levels (64^2) downsampled with a guttered box filter, and our parameterization-aware bilinear filter. The tent filter does not preserve details as well as our method, and it allows the background color of the texture to bleed in at texture seams.

designed to improve the quality of texture filtering since the invention of mipmaps. The simplest improvement is to generate mipmaps using more advanced filters derived from signal processing. More recently, research has focused on designing anisotropic filters that can be evaluated in real-time [141, 99, 97, 35, 21, 20, 61].

We solve a problem that is related to, but separate from, anisotropic filtering. Anisotropic filtering calculates texture samples by transforming the filter using a first-order approximation of $\Theta \circ \Phi^{-1}$, which is the affine transform defined by the Jacobian of $\Theta \circ \Phi^{-1}$. Describing the local distortion by an affine transformation is correct for a single point, but is only approximate when integrating over the support of h, unless the distortion from screen to texture space, $\Theta \circ \Phi^{-1}$, is uniform. GPUs typically approximate integration of h by sampling from a mipmap. However, at higher mip-levels, texels contain many triangles, and the assumption of uniformity of $\Theta \circ \Phi^{-1}$ is violated.

We assume that the parameterization of the model is fixed and improve the texture quality without modifying the model or the base texture. We can correct for nonuniform distortion of Θ during mipmap generation and improve texturing, with or without

anisotropic sampling. Our method uses the fact that Θ is view-independent to precompute mipmaps that prefilter the texture to correct for the nonuniformity of the parameterization of a surface in the support of h. Our method corrects for parametric distortions introduced by Θ and shows how to optimize texture reproduction when using a trilinear postfilter. We solve the large-scale problem in which parameterization changes between triangles, whereas anisotropic filtering solves the small-scale problem of sampling points on the screen. Figure 7.1 shows an example of the improvement obtained with our method compared to tent filtering. The model on the left is drawn using the 1024^2 input texture, whereas the images in the center and to the right are drawn with 64^2 tent and parameterization-aware mipmapped (PAM) bilinear filters, respectively.

Anisotropic filtering on a GPU still provides benefits with our method because anisotropic filtering adapts to changes of Φ, which are only known at run-time. Thus, anisotropic filtering and our method complement each other. We generate optimized mipmaps as a preprocessing step and improve image quality with no change to art assets or rasterization algorithms and at no cost to run-time performance. As an added benefit, our method automatically ignores the unused portion of the texture that forms the background color, which prevents color bleeding at higher mip-levels.

7.1 Related Work

In the original description of mipmapping [134], downsampled images were generated using a box filter, but one can easily imagine using higher-order filters at each level, such as in a Gaussian pyramid [17]. An obvious way to improve the quality of mipmapping is to use high-quality antialiasing filters to generate the images at each mip-level. Hummel [76] described optimal prefilters for linearly dependent postfiltering bases such as tent and cubic B-splines. However, Hummel only considers functions over an infinite, uniform grid. Kajiya and Ullner [83] solve a least-squares problem in which intensities were

constrained to be in the range $[0, 1]$ to render fonts on CRT displays in which the pixel response is approximately Gaussian. Least-squares downsampling [140] is a method for finding a downsampling that optimizes over point samples. In contrast, our work finds a downsampling that is optimized over all mip-levels simultaneously to match the postfiltering performed by GPUs, to handle boundary effects of the image and chart boundaries, and to account for non-uniform parameterization of the surface during filtering.

Since the initial description of mipmapping, most subsequent work has since sampled images with affine transformations of filters [65, 141, 99, 97, 35, 21, 20, 72, 61]. Although anisotropic filtering methods correct for the asymmetric stretch induced by mapping a texture to screen space, they do not account for different transformations within the support of a filter. The support of texels in the mipmap from trilinear interpolation also influences the shape of the filter. The large support of texels can cause colors from unused portions of the texture to bleed through texture seams onto the surface of the object. Guttering [128] can reduce this problem by creating an extended border of similar color around the chart boundary. The process of guttering can be automated through a push-pull algorithm [62, 114] in which texels outside of texture charts are ignored during mipmap creation. The image is successively upsampled by factors of two from the lowest resolution, overwriting only the unused texels.

One can also circumvent parameterization by storing textures on the surface itself, such as in Ptex [16] and Mesh Colors [138]. Both of these methods implement mipmapping and anisotropic filtering calculated directly on a surface. Alternatively, textures can be stored in three-dimensional space around a surface [8, 89] to avoid surface parameterization. In general, these methods tend to be slower than typical texturing, either because they are more complicated or because they have no hardware support. In contrast, our method improves texture sampling when using the native trilinear sampling implemented by GPUs. Some methods [108, 112] also remove seams that result from trilinear filtering by chang-

ing the surface parameterization Θ, but they do not fix other parameterization-induced filtering artifacts. We improve texture filtering without modifying Θ.

7.2 Parameterization-aware Filtering

The mapping Θ between the surface of an object and a texture is often nonuniform, meaning that each triangle can have a different texel density. Additionally, triangles may overlap in texture space, and some of the texture space may not be used. Most methods for generating mipmaps filter the texture without regard to how the texture is applied to the surface of an object. We show that image filtering operations can be performed directly on the surface of an object rather than in texture space.

By filtering over the surface of the object, we can ensure that the weight of a texel is proportional to the surface area that it covers on an object. Weighting by surface area means that, if a texel is used in two overlapping charts, then the texel will be given twice the weight of a texel used in a single chart. Conversely, if a texel is not used on the surface of an object, it will be given zero weight. All of the filtering is performed in a preprocessing step, which means that the GPU can draw textured objects using standard texture sampling operations. Parameterization-aware filtering incurs absolutely no run-time performance penalty and is robust to degenerate inputs such as triangles that have zero area in either object space or texture space.

The input image $\hat{I}$ is the the sum $\sum_i \hat{c}_i \hat{b}_i(u, v)$ of basis functions time coefficients. In typical texture sampling, the coefficient c_h of the sampling filter h is calculated by integrating

$$c_h = \iint_{\mathbb{R}^2} h(u, v) \sum_i \hat{c}_i \hat{b}_i(u, v)\, du\, dv$$

over the texture, where $\hat{c}_i$ are the coefficients (i.e. colors) of the texels. Each coefficient $\hat{c}_i$ is associated with a basis function $\hat{b}_i$, where i denotes the translation of the basis functions. Although $\hat{b}_i$ can be arbitrary, GPUs typically multiply texture samples by bilinear basis

functions for texture magnification. We therefore use a bilinear basis. In parameterization-aware filtering, we evaluate filters by integrating over the surface Ω of the object

$$c_h = \frac{\iint_\Omega h(\Theta(p)) \sum_i \hat{c}_i \hat{b}_i(\Theta(p))\, dp}{\iint_\Omega h(\Theta(p))\, dp},$$

where p are three-dimensional points on the surface and the functions h and $\hat{b}_i$ are defined in texture space, as before. In typical texture filtering, no normalization is required because the integral of h is assumed to be unitary, but we need a normalization term because the domain of h may be distorted when integrating over a surface. In practice, we evaluate c_h by summing over triangles in a mesh. Each triangle T_k of Ω has a barycentric basis with the triangle coordinates $(0,0), (1,0), (1,1)$. Let $\Gamma_k : \mathbb{R}^2 \to \mathbb{R}^3$ be the map from this barycentric basis to the triangle T_k. We can then write that

$$c_h = \frac{\sum_k \int_0^1 \int_0^x h(\Theta(\Gamma_k(x,y))) \sum_i \hat{c}_i \hat{b}_i(\Theta(\Gamma_k(x,y)))\Delta_k\, dy\, dx}{\sum_k \int_0^1 \int_0^x h(\Theta(\Gamma_k(x,y)))\Delta_k\, dy\, dx}, \tag{7.1}$$

where Δ_k is the absolute value of the determinant of the Jacobian of Γ_k and is equal to the area of triangle T_k.

Because h and b_i are typically piecewise polynomial functions that align with the texel grid, we cut the 3D polygons T_k so that their images $\Theta(T_k)$ in texture space intersect only one texel. This operation can be done quickly and robustly. Because both b_i and h have a single polynomial, Equation 7.1 has a closed-form expression.

Our method integrates directly over 3D triangles by using their image in a barycentric space. Equivalently, we can integrate over the image of triangles in texture space $\Theta(T_k)$, which requires multiplying by $|J(\Theta^{-1})|$, where J is the Jacobian of a transform, instead of by Δ_k to account for the change in variables. We say that our mipmaps are parameterization-aware because we weight texels by their parametric distortion $|J(\Theta^{-1})|$, whereas

standard filtering gives all texels equal weight. By integrating over the 3D triangles T_k rather than their image in texture space $\Theta(T_k)$, we avoid problems in which 3D triangles in Ω map to degenerate triangles in the texture, which would make $J(\Theta^{-1})$ undefined.

A beneficial property of our optimization is that our method does not interfere with the anisotropic filtering that is performed by GPUs. This is achieved because we solve a different problem from anisotropic filtering. First, consider the case in which anisotropic filtering produces the correct result. If a square texture is mapped to a rectangle, anisotropic sampling will appropriately stretch the sampling filter by the aspect ratio. In this case, our method is unaffected by the change in parameterization and produces the same result as traditional image filtering because the Jacobian of the parameterization is uniform and all texels have equal weights. Therefore, the result from anisotropic filtering is unaltered by our method and produces the correct result.

Our method produces different images only when the parameterization is non-uniform, which occurs when the assumptions of anisotropic filtering are violated. Excluding developable surfaces, flattening a surface always introduces distortions. Severe distortions within a texel are common at lower-resolution mip-levels, when texels cover many triangles of the surface. We show an example of non-uniform parameterization in Figure 7.2. In the top row of the figure, we show a flat, triangulated object on the left and its texture on the right. Although blue and green triangles have equal areas in object space, blue triangles occupy the majority of the texture. This means that blue dominates higher mip-levels with standard mipmap generation. In contrast, our method gives blue and green colors equal weight in the distance while preserving the sharp transition between colors in the foreground. This effect can be seen in the middle row of Figure 7.2, where the plane turns blue in the distance instead of blue-green. Our method draws the correct color at high mip-levels in the distance. Comparing PAM box filtering in the middle and PAM constrained trilinear filtering on the right, the images are similar, but PAM trilinear filtering decreases

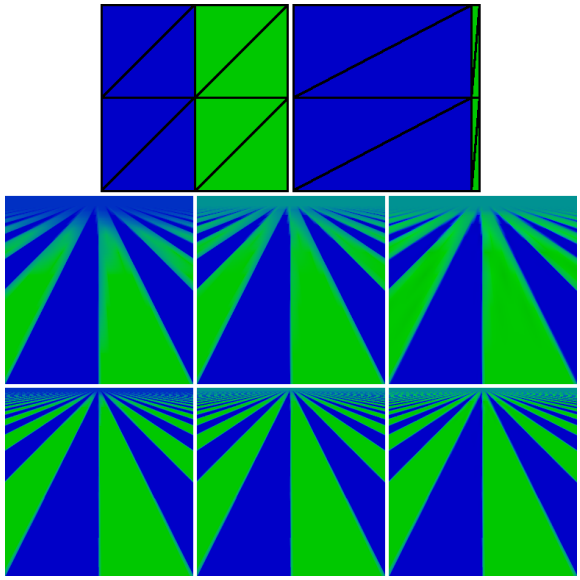


Figure 7.2: The top images show the parametric distortion with object space on the left left and texture space on the right. The middle row shows trilinear mipmapping, and the bottom row shows 16x anisotropic mipmap sampling. From left to right: box, PAM box, and PAM constrained trilinear mipmaps.

the bleeding of blue into the green bands.

In fact, anisotropic filtering and our method are complementary. Because the projection Φ is known only at run-time, anisotropic filtering is required to filter surfaces accurately with respect to their orientation to the camera. On the other hand, anisotropic filtering cannot filter correctly when the Jacobian of Θ changes over multiple triangles. Combining our mipmaps with anisotropic filtering provides a good approximation of $\Theta \circ \Phi^{-1}$ integrated over the entire support of the filter. The bottom row of Figure 7.2 shows how texture filtering is improved by a combination of anisotropic filtering and parameterization-aware mipmapping. Anisotropic filtering samples from higher-resolution mip-levels, which improves image quality, but the plane still turns blue in the distance in the left image. Anisotropic filtering combined with our method draws crisp lines that correctly appear blue-green in the distance.

7.3 Optimal Trilinear Approximation

GPUs store downsampled images as a small set of precalculated images, called a mip-map, in order to accelerate downsampling at arbitrary resolutions. The GPU approximates colors at intermediate resolutions by trilinear interpolation between the color samples of the two closest resolutions. Our goal is to make the color samples calculated by trilinear filtering match the color of the input image $\hat{I}$ as accurately as possible. We achieve this by solving a least-squares optimization, as suggested by Kajiya and Ullner [83]. Our contribution is to simultaneously optimize the entire mip-stack so that we find the optimal representation over all resolutions rather than within a single two-dimensional image. Thus, instead of filtering each image in the mip-stack separately, the filtered images are all interdependent. We also incorporate corrections for parametric distortions into our optimization and explicitly handle the effects that chart boundaries have on the optimization, whereas Kajiya and Ullner [83] assume that pixels reside on an infinite plane and can all be treated identically.

Performing a least-squares optimization significantly improves downsampled image quality but can introduce ringing. In Figure 7.3, we show an example of an image that is downsampled followed by upsampling with a bilinear filter. Comparing the box filter (top right) and least-squares optimization (bottom left) shows that the optimized filter reproduces the original image more accurately and appears less blurry, but it suffers from some ringing artifacts. As described by Kajiya and Ullner [83], some of the ringing results in values that are outside of a displayable range of the monitor. Finding the best values to reproduce an image on a physical device requires optimizing for values constrained to [0,1]. We show the results of this constrained optimization on the bottom right of Figure 7.3. The image appears sharp and has fewer artifacts than the unconstrained optimization.

The trilinear interpolant is best understood by visualizing the mipmap as a stack of

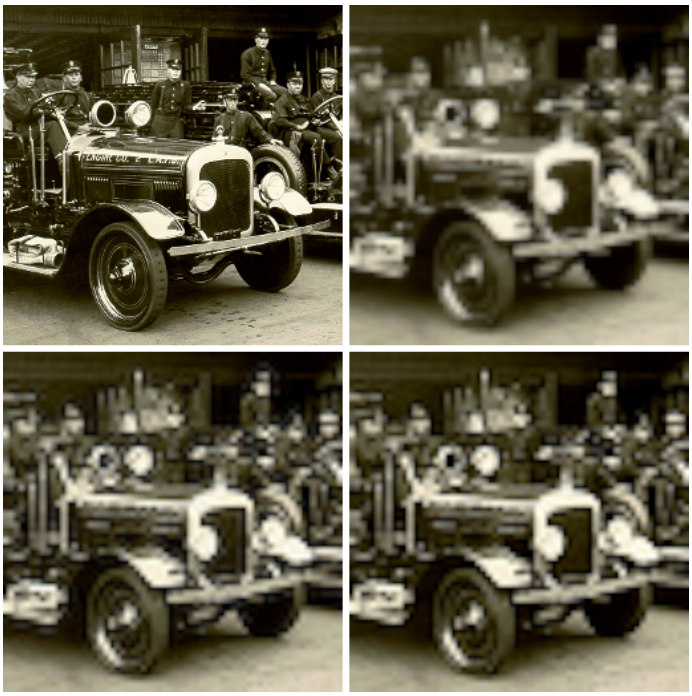


Figure 7.3: An example of a high-resolution image (top left) downsampled using a box filter (top right), optimized for bilinear reconstruction (bottom left), and optimized for bilinear reconstruction constraining values to [0,1] (bottom right).

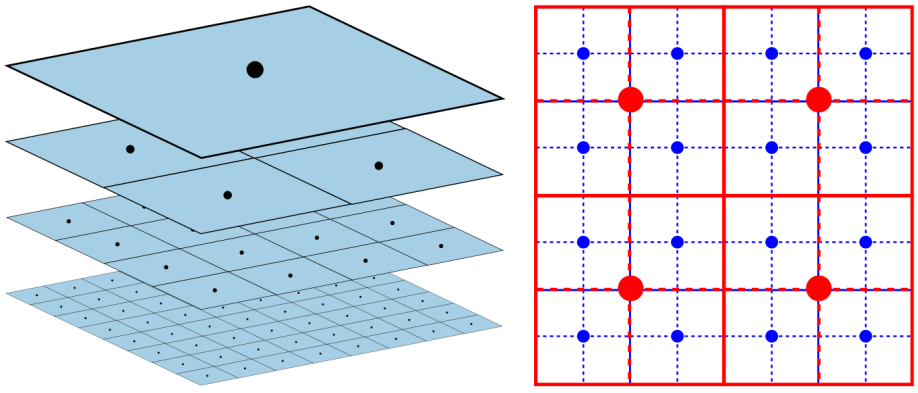


Figure 7.4: A mipmap can be visualized as a stack of overlaid images, as shown on the left. The alignment between neighboring resolutions is shown on the right.

images, as shown on the left of Figure 7.4. The basis functions are centered on texels and are shown as black dots. The stack defines a rectangular solid parameterized by u, v, w, and the images are evenly distributed in w from 0 to n. Given an input image $\hat{I}$ at resolution 2^n, the mipmap of $\hat{I}$ is a stack of $n+1$ images $I = (I_0, I_1, ..., I_n)$. The images are indexed in order of the distance at which they are displayed so that I_0 has same resolution as $\hat{I}$, and the resolution of I_w is 2^{n-w}. The canonical trilinear basis function is the tensor product of unit tent functions, and the trilinear basis function $b_j(u, v, w)$ of image w is scaled in u and v by 2^w but is not scaled in w. The index j is a triplet of integers that indicates the translation and scaling of b_j.

We minimize the error of a sample by minimizing the difference between $\hat{I}$ and I over all distances, such that

$$\min_{c_j} \quad \sum_k \int_{-\infty}^{\infty} \int_0^1 \int_0^x \left| \sum_j b_j(\Theta(\Gamma_k(x,y)), w) c_j - \sum_i \hat{b}_i(\Theta(\Gamma_k(x,y)))\hat{c}_i \right|^2 \Delta_k \, dy \, dx \, dw.$$

The system is large, but sparse, so we solve the system using conjugate gradients implemented in TAUCS [127]. When w is outside of the range $[0, n]$, we reproduce the behavior of GPUs by sampling from the closest available mip-level. For $w < 0$, there is more than one pixel per texel, and the image is magnified using bilinear interpolation. The weight from the negative values of w is infinite and adds only to the basis functions of I_0, which constrains $\hat{I} = I_0$. Likewise, for $w > n$, more than one copy of the texture may exist in a pixel, and we constrain I_n to be the area-weighted average of $\hat{I}$. Constraining I_0 and I_n dramatically reduces the size of the linear system we must solve. At a small cost to image quality, we can instead optimize color reproduction for each mip-image individually by using a bilinear basis instead of a trilinear basis. If we use box basis functions instead of bilinear or trilinear functions, the linear system is diagonal, and each texel is indepen-

dent. This means that, for box filtering, the least-squares solution reduces to the filtering described in Section 7.2 and is quite fast.

We show how the trilinear basis functions of neighboring resolutions overlap in Figure 7.4. We use blue for one resolution and red for half the blue resolution. Solid lines show the primal grid, dashed lines show the dual grid, and dots show the centers of basis functions. Because the dual grids over which trilinear basis functions are defined do not nest, we must cut triangles by a grid that is twice the resolution of the high-resolution image (the half-texel grid).

Conceptually, basis functions of color samples from outside of the domain of an image can intersect the image. GPUs define multiple methods for dealing with this problem. In OpenGL, borders are treated based on the *wrap mode* of the image, of which three modes are commonly used. One approach is to add a border that is one texel wide around the image so that the color values are defined for all basis functions that intersect the image. The second and third approaches require no extra storage because they define the border values to be equal to values within the image. If the image repeats, border colors are defined by taking the modulo of the index of a texel, whereas non-repeating images clamp border colors to be equal to the nearest color in the image. Fortunately, it is simple to match our optimization to the *wrap mode* used to sample textures. All of the examples shown here use the clamp functionality in the optimization, except for Figure 7.2, which we optimize for a repeating texture.

7.4 Results and Discussion

We compare our parameterization-aware filtering using a box filter versus naïve box filtering in Figure 7.5. The input texture is shown at the top left. In this example, there is relatively little unused texture space, so the background color has less of an effect on the filtered images. However, this model exploits symmetry to reuse parts of the texture

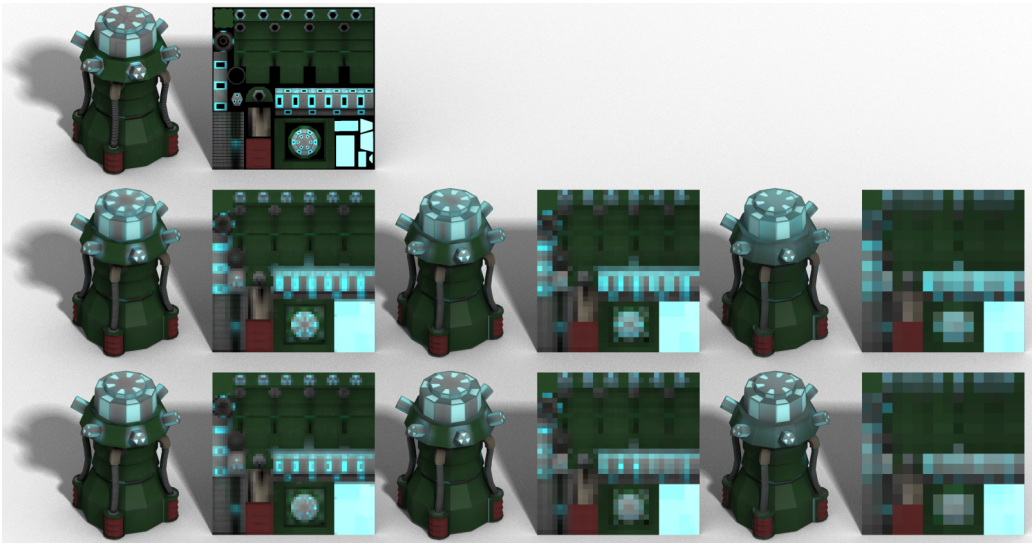


Figure 7.5: The model at the top right is drawn with the full 1024^2 input texture. The models in the row directly below that are drawn with 64^2, 32^2, and 16^2 box-filtered textures that ignore texels that do not intersect triangles. The models in the bottom row are drawn using PAM box filtered textures at the same resolutions, and this method better preserves the original texture.

and contains overlapping charts. For example, there are four gray pipes and red grills in the model, but only one instance of these objects in the texture. Moreover, some of the triangles in the three-dimensional surface map degenerately to lines in the texture.

Our method handles all of these issues properly. The top row shows the model drawn with its 1024^2 input texture. The middle row shows the results from using a standard box filter, and the bottom row shows the results from using our PAM (parameterization-aware mipmapping) box filter. From left to right, the resolutions of the textures in the middle and bottom rows are 64^2, 32^2, and 16^2. One can see that parameterization-aware filtering significantly reduces the amount of blue that bleeds into the gray at the top of the models. Each of the blue lights is inset slightly, and the wall of the inset is given a disproportionately large fraction of the texture space. Our PAM filter weights these insets proportionally to their surface area rather than to their texture area to reproduce the

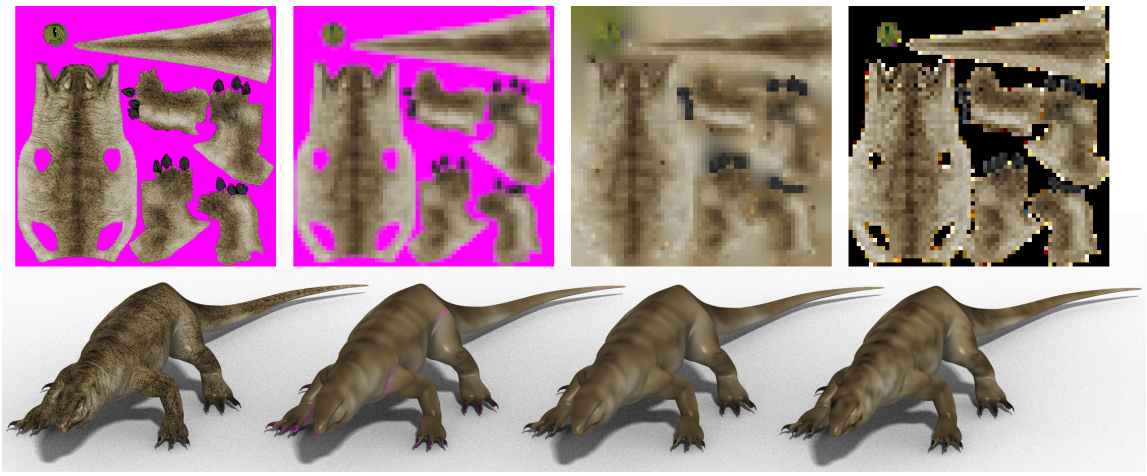


Figure 7.6: An example of a model with a 1024^2 input texture is shown on the left. The subsequent models are drawn with downsampled textures at 64^2 resolution calculated using a box filter, a box filter that ignores texels that do not intersect triangles and uses guttering, and our parameterization-aware bilinear filter. Unused parts of the input texture are colored magenta to illustrate the color bleeding that occurs with image filtering that does not use information about the model.

appearance of the original model more faithfully. We should note that our method even handles the case when two triangles with different Jacobians overlap, a situation in which a texel will be weighted by the sum of the surface areas that intersect that texel.

A consequence of using Θ to weight downsampled colors based on how often they appear on a surface is that unused parts of a texture are given no weight, so that the colors between charts have no effect on downsampled images. We show an example of the effect from using different filters on a lizard model in Figure 7.6, where we modified the input texture to be magenta between charts in order to emphasize how the background color affects filtering. From left to right, we show the model drawn with its 1024^2 input texture and models with textures downsampled to 64^2 using a box filter, a box filter that ignores unused parts of the texture, and our PAM bilinear filter. Standard box filtering does not take into account which texels are visible on the model and allows the unused texels, colored

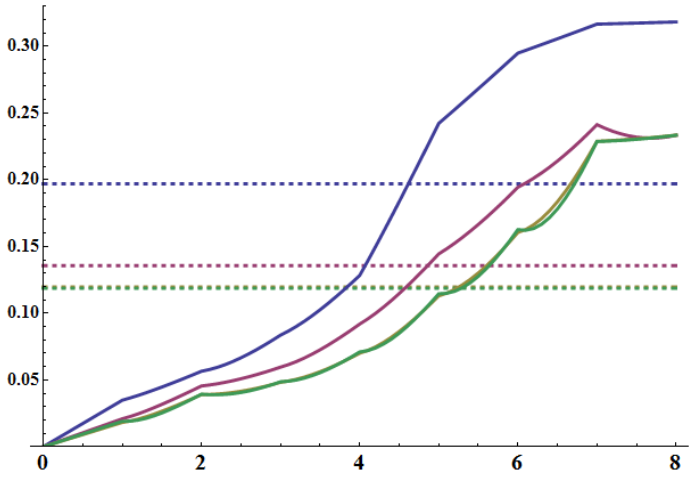


Figure 7.7: Graphs of the errors of textures in Figure 7.2 measured at different mip-values from zero through eight. The filters used are box (blue), PAM box (red), PAM constrained bilinear (yellow), and PAM constrained trilinear (green).

in magenta, to influence the color of texels at seams. Ignoring unused texels in a box filter approximates the effect of our parameterization-aware box filter, but is insufficient by itself. The support of the box downsampling filter is smaller than the support of the trilinear filter used by hardware to sample the texture, which means that at least a one texel border of similar colors must be added around charts. Our method can directly optimize for trilinear filtering, which takes into account the full support of the sampling filter, fractional texel coverage, and overlapping/degenerate charts.

Methods exist to remove texture seams [108, 112] that prevent the background from affecting the surface color, but these methods do not correct for other artifacts due to parameterization. We show a similar result in Figure 7.1, where it is clear that our parameterization-aware method preserves details better than conventional texture filtering and eliminates color bleeding across chart boundaries.

In Figure 7.7, we show a graph of the errors from using a box filter compared to the parameterization-aware box, bilinear, and trilinear filters of the image in Figure 7.2.

This example utilizes the entire texture, so differences in the graph result entirely from differences in parameterization and not from background color bleeding. The root mean square error (RMSE) of the approximation measured at different mip-values is shown by solid lines, and the RMSE over all mip-levels is shown by dashed lines. The PAM bilinear and trilinear filters have noticeably lower error than the box filter. The bilinear filter has lower errors at integer mip-levels than the trilinear filter, but has a slightly higher overall error. One can see that correcting for parametric distortion significantly reduces the RMSE, especially at low resolutions.

The times taken to calculate mipmaps using a variety of filters at different input sizes are shown in Table 7.1. All of the times were calculated for the lizard model on an Intel Core i7-2600k. Calculation time depends mainly on the number of texels in the charts rather than triangles in the mesh, as long as there are fewer triangles than texels, because triangles are cut to a fine grid. We should emphasize that mipmap generation only has to be performed once, after which results can be stored. Hence, mipmap generation time should not be an issue for most applications. Nevertheless, in the worst case, solving for mipmaps that are optimized for trilinear filtering from a 1024^2 source image takes slightly more than one minute. Error graphs suggest that optimizing for bilinear reconstruction is almost as good as optimizing for trilinear reconstruction, but it takes only a fifth of the time. Therefore, we expect most users will prefer PAM bilinear filtering to PAM trilinear filtering. Parameterization-aware box filtering is very fast because no linear system has to be built or solved. In models with parametric distortion, parameterization-aware box filtering provides a significant reduction in error and costs only a second of preprocessing time.

	64	128	256	512	1024
box	0.004	0.004	0.011	0.034	0.134
tent	0.004	0.008	0.037	0.164	0.697
Lanczos 3	0.033	0.151	0.677	3.064	13.58
PAM box	0.030	0.051	0.124	0.338	1.061
PAM bi.	0.166	0.344	0.993	3.703	15.48
PAM tri.	0.411	1.291	4.699	18.606	75.68

Table 7.1: Times measured in seconds to construct mipmaps for the lizard from different input resolutions. Traditional filters that ignore parameterization are shown on top, while our filters are shown on the bottom.

7.5 Conclusions and Future Work

We present a method for calculating downsampled images for display as textures in three-dimensional applications in which we minimize the difference between a source image and bilinearly or trilinearly interpolated textures and correct for distortions introduced by mesh parameterization. We perform all calculations as a preprocessing step to improve image quality without changing the renderer or reducing run-time performance.

There are several extensions to our method that we would like to investigate. One of the assumptions of our method is that mesh geometry is static. In practice, meshes are often animated, which means that the optimal texture is different for each frame of animation. It is trivial to modify Equation 7.1 to use the average Jacobian over all frames, but how often a frame is displayed, as in a game, may not be known in advance. In practice, most realistic objects undergo nearly isometric deformations, and our method will perform well even when optimized for a single, static pose.

8. CONCLUSION

This dissertation considers rasterization as a specific application of image filtering, and I have analyzed the two principle aspects of rasterization in a modern graphics pipeline: how to calculate coverage values from a boundary representation and how to sample textures. Rendering generally has simple known solutions that can be evaluated through brute-force computation, and the role of research is to design algorithms that are able to evaluate the solutions more efficiently. The methods described in Chapters 2 and 3 evaluate coverage values and vector graphics for regions bounded by Bézier curves by evaluating closed-form solutions for pixel colors using high-quality filters. For texture sampling, I present a method in Chapters 6 and 7 for approximating filters using far fewer texture fetches than would otherwise be necessary in existing methods.

There is still room for further improvement of the efficiency of rendering and image sampling algorithms, but there is another direction of research I want to investigate. It intrigues me that there is no general consensus about what is the best image filter. It is of particular note that the theoretically ideal sinc filter is unanimously considered not to be the best filter. The fact that there is such a discrepancy between theory and perception requires further investigation. If we better understand how filtering is perceived, there will be opportunities to improve the maximum quality of images and to spend less time computing details that are not perceptually important.

Filtering is also important in many fields outside of computer graphics and rendering. In this report, I demonstrated that similar mathematical derivations of filtering with wavelets can be used for the dissimilar purposes of rasterizing images and reconstructing surfaces from scanned data. Computer graphics is often concerned with designing algorithms that are computationally efficient rather than perfectly accurate because of the

demand for real-time performance. Other fields that sample signals may also benefit from recent developments in computer graphics. When accuracy is paramount, it can still be useful to have fast approximations to preview results or to process larger data sets than would otherwise be possible. I am interested to find out if the methods I have developed can be applied across disciplines.

REFERENCES

[1] Rémi Allègre, Raphaëlle Chaine, and Samir Akkouche. A streaming algorithm for surface reconstruction. In *Proceedings of the Eurographics Symposium on Geometry Processing*, pages 79–88, 2007.

[2] Nina Amenta, Sunghee Choi, and Ravi Krishna Kolluri. The power crust. In *Proceedings of the ACM Symposium on Solid Modeling and Applications*, pages 249–266, 2001.

[3] John C. Anderson, Christoph Garth, Mark A. Duchaineau, and Ken Joy. Discrete multi-material interface reconstruction for volume fraction data. *Computer Graphics Forum (Proceedings of Eurographics)*, 27(3):1015–1022, 2008.

[4] John C. Anderson, Christoph Garth, Mark A. Duchaineau, and Ken Joy. Smooth, volume-accurate material interface reconstruction. *IEEE Trans. Vis. Comp. Grap.*, 16(5):802–814, 2010.

[5] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of AFIPS*, pages 37–45, 1968.

[6] Thomas Auzinger, Michael Guthe, and Stefan Jeschke. Analytic anti-aliasing of linear functions on polytopes. *Computer Graphics Forum (Proceedings of Eurographics)*, 31(2):335–344, 2012.

[7] J. Andreas Bærentzen and Henrik Aanæs. Signed distance computation using the angle weighted pseudonormal. *IEEE Transactions on Visualization and Computer Graphics*, 11:243–253, 2005.

[8] David Benson and Joel Davis. Octree textures. In *Proceedings of ACM SIGGRAPH*, pages 785–790, 2002.

[9] Fausto Bernardini, Joshua Mittleman, Holly Rushmeier, Cláudio Silva, and Gabriel Taubin. The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):349–359, 1999.

[10] Silvia Bertoluzza, Claudio Canuto, and Karsten Urban. On the adaptive computation of integrals of wavelets. *Applied Numerical Mathematics*, 34(1):13–38, 2000.

[11] James F. Blinn. How to solve a quadratic equation? *IEEE Comput. Graph. Appl.*, 25(6):76–79, 2005.

[12] James F. Blinn. How to solve a cubic equation, part 5: Back to numerics. *IEEE Comput. Graph. Appl.*, 27(3):78–89, 2007.

[13] Matthew Bolitho, Michael Kazhdan, Randal Burns, and Hugues Hoppe. Multilevel streaming for out-of-core surface reconstruction. In *Proceedings of the Eurographics Symposium on Geometry Processing*, pages 69–78, 2007.

[14] Matthew Bolitho, Michael Kazhdan, Randal Burns, and Hugues Hoppe. Parallel poisson surface reconstruction. In *Proceedings of the International Symposium on Advances in Visual Computing*, pages 678–689, 2009.

[15] Kathleen S. Bonnell, Mark A. Duchaineau, Daniel R. Schikore, Bernd Hamann, and Kenneth I. Joy. Material interface reconstruction. *IEEE Trans. Vis. Comp. Grap.*, 9(4):500–511, 2003.

[16] Brent Burley and Dylan Lacewell. Ptex: Per-face texture mapping for production rendering. In *Proceedings of the Eurographics Symposium on Rendering*, pages 1155–1164, 2008.

[17] Peter Burt. Fast filter transform for image processing. *Computer Graphics and Image Processing*, 16(1):20–51, 1981.

[18] P.J. Burt and E.H. Adelson. The laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, 31(4):532–540, 1983.

[19] Marcel Campen and Leif Kobbelt. Exact and robust (self-) intersections for polygonal meshes. *Computer Graphics Forum (Proceedings of Eurographics)*, 29(2):397–406, 2010.

[20] R Cant and P Shrubsole. Texture potential mapping: A way to provide antialiased texture without blurring. In *Visualization and Modelling*, pages 223–240, 1997.

[21] R Cant and P Shrubsole. Texture potential mip mapping, a new high-quality texture antialiasing algorithm. *ACM Transactions on Graphics*, 19(3):164–184, 2000.

[22] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans. Reconstruction and representation of 3d objects with radial basis functions. In *Proceedings of ACM SIGGRAPH*, pages 67–76, 2001.

[23] Paulo Cezar Pinto Carvalho and Paulo Roma Cavalcanti. *Graphics Gems V*, chapter Point in Polyhedron Testing Using Spherical Polygons, pages 42–49. AP Professional, Chestnut Hill, MA, 1995.

[24] Edwin Catmull. A hidden-surface algorithm with anti-aliasing. In *Proceedings of ACM SIGGRAPH*, pages 6–11, 1978.

[25] Edwin Catmull. An analytic visible surface algorithm for independent pixel processing. In *Proceedings of ACM SIGGRAPH*, pages 109–115, 1984.

[26] Edwin Earl Catmull. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, 1974. The University of Utah.

[27] Baoquan Chen, Frank Dachille, and Arie E. Kaufman. Footprint area sampled texturing. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):230–240, 2004.

[28] M. Chen and J. Arvo. A closed-form solution for the irradiance due to linearly-varying luminaries. In *Proceedings of the Eurographics Workshop on Rendering Techniques*, pages 137–148, 2000.

[29] M. Chen and J. Arvo. Simulating non-lambertian phenomena involving linearly-varying luminaires. In *Proceedings of the Eurographics Workshop on Rendering Techniques*, pages 25–38, 2001.

[30] Antonio Chica, Jason Williams, Carlos Andujar, Pere Brunet, Isabel Navazo, Jaroslaw R. Rossignac, and Alvar Vinacua. Pressing: Smooth isosurfaces with flats from binary grids. *Computer Graphics Forum*, 27(1):36–46, 2007.

[31] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: Measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, 1998.

[32] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, 1986.

[33] Marc Corthout and Hans Jonkers. A new point containment algorithm for B-regions in the discrete plane. In *Theoretical Foundations of Computer Graphics and CAD*, pages 297–306, 1988.

[34] Franklin C. Crow. The aliasing problem in computer-generated shaded images. *Communications of the ACM*, 20(11):799–805, 1977.

[35] Franklin C. Crow. Summed-area tables for texture mapping. In *Proceedings of ACM SIGGRAPH*, pages 207–212, 1984.

[36] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of ACM SIGGRAPH*, pages 303–312, 1996.

[37] Wolfgang Dahmen and Charles A. Micchelli. Using the refinement equation for evaluating integrals of wavelets. *SIAM Journal on Numerical Analysis*, 30(2):507–537, 1993.

[38] Ingrid Daubechies. *Ten lectures on wavelets*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[39] Tamal K. Dey, Joachim Giesen, and James Hudson. Delaunay based shape reconstruction from large data. In *Proceedings of the Symposium on Parallel and Large-data Visualization and Graphics*, pages 19–27, 2001.

[40] Tamal K. Dey and Samrat Goswami. Provable surface reconstruction from noisy samples. *Comput. Geom. Theory Appl.*, 35(1):124–141, 2006.

[41] Carlos Dietrich, Carlos Scheidegger, Joao Comba, Luciana Nedel, and Claudio Silva. Edge groups: An approach to understanding the mesh quality of marching methods. *IEEE Trans. Vis. Comp. Grap.*, 14:1651–1666, 2008.

[42] Carlos A. Dietrich, Carlos E. Scheidegger, John Schreiner, Joao L.D. Comba, Luciana P. Nedel, and Claudio T. Silva. Edge transformations for improving mesh quality of marching cubes. *IEEE Trans. Vis. Comp. Grap.*, 15:150–159, 2009.

[43] Mark A. Z. Dippé and Erling Henry Wold. Antialiasing through stochastic sampling. In *Proceedings of ACM SIGGRAPH*, pages 69–78, 1985.

[44] Khanh Doan. Antialiased rendering of self-intersecting polygons using polygon decomposition. In *Proceedings of Pacific Graphics*, pages 383–391, 2004.

[45] David P. Dobkin, David Eppstein, and Don P. Mitchell. Computing the discrepancy with applications to supersampling patterns. *ACM Transactions on Graphics*, 15(4):354–376, 1996.

[46] Zhao Dong, Wei Chen, Hujun Bao, Hongxin Zhang, and Qunsheng Peng. Real-time voxelization for complex polygonal models. In *Proceedings of Pacific Graphics*, pages 43–50, 2004.

[47] Claude Duchon. Lanczos filtering in one and two dimensions. *Journal of Applied Meteorology*, 18(8):1016–1022, 1979.

[48] Tom Duff. Polygon scan conversion by exact convolution. In *Proceedings of the International Conference on Raster Imaging and Digital Typography*, pages 154–168, 1989.

[49] Mohamed S. Ebeida, Andrew A. Davidson, Anjul Patney, Patrick M. Knupp, Scott A. Mitchell, and John D. Owens. Efficient maximal poisson-disk sampling. *ACM Transactions on Graphics*, 30(4):49:1–49:12, 2011.

[50] Elmar Eisemann and Xavier Décoret. Fast scene voxelization and applications. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, pages 71–78, 2006.

[51] Elmar Eisemann and Xavier Décoret. Single-pass GPU solid voxelization for real-time applications. In *Proceedings of Graphics Interface*, pages 73–80, 2008.

[52] A. E. Fabris and A. R. Forrest. Antialiasing of curves by discrete pre-filtering. In *Proceedings of ACM SIGGRAPH*, pages 317–326, 1997.

[53] Shiaofen Fang and Duoduo Liao. Fast CSG voxelization by frame buffer pixel mapping. In *Proceedings of the Symposium on Volume Visualization*, pages 43–48, 2000.

[54] Raanan Fattal. Blue-noise point sampling using kernel density model. *ACM Transactions on Graphics*, 28(3):48:1–48:10, 2011.

[55] F. R. Feito and J. C. Torres. Inclusion test for general polyhedra. *Computers and Graphics*, 21(1):23–30, 1997.

[56] Alain Fournier, Eugene Fiume, and Sandford Fleming Building. Constant-time filtering with space-variant kernels. In *Proceedings of ACM SIGGRAPH*, pages 229–238, 1988.

[57] Manuel N. Gamito and Steve C. Maddock. Accurate multidimensional poisson-disk sampling. *ACM Transactions on Graphics*, 29(1):8:1–8:19, 2009.

[58] Sarah Gibson and F. Frisken. Constrained elastic surface nets: Generating smooth surfaces from binary segmented data. In *Proceedings of Medical Image Computing and Computer-Assisted Intervention*, pages 888–898, 1998.

[59] Walter Gish and Allen Tanner. Hardware antialiasing of lines and polygons. In *Proceedings of the Symposium on Interactive 3D graphics*, pages 75–86, 1992.

[60] Andrew Glassner. Adaptive precision in texture mapping. In *Proceedings of ACM SIGGRAPH*, pages 297–306, 1986.

[61] Alexander Goldberg, Matthias Zwicker, and Frédo Durand. Anisotropic noise. In *Proceedings of ACM SIGGRAPH*, pages 54:1–54:8, 2008.

[62] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In *Proceedings of ACM SIGGRAPH*, pages 43–54, 1996.

[63] Craig Gotsman. Constant-time filtering by singular value decomposition. *Computer Graphics Forum*, 13(2):153–163, 1994.

[64] Chris Green. Improved alpha-tested magnification for vector textures and special effects. In *Proceedings of SIGGRAPH Courses*, pages 9–18, 2007.

[65] Ned Greene and Paul Heckbert. Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter. *IEEE Computer Graphics and Applications*, 6(6):21–27, 1986.

[66] Carl Johan Gribel, Rasmus Barringer, and Tomas Akenine-Möller. High-quality spatio-temporal rendering using semi-analytical visibility. In *Proceedings of ACM SIGGRAPH*, pages 54:1–54:12, 2011.

[67] Brian Guenter and Jack Tumblin. Quadrature prefiltering for high quality antialiasing. *ACM Transactions on Graphics*, 15(4):332–353, 1996.

[68] Alfred Haar. Zur Theorie der orthogonalen Funktionensysteme. *Mathematische Annalen*, 69:331–371, 1910.

[69] Mark Hall and Joe Warren. Adaptive polygonalization of implicitly defined surfaces. *IEEE Comput. Graph. Appl.*, 10:33–42, 1990.

[70] Iddo Hanniel and Ron Wein. An exact, complete and efficient computation of arrangements of bézier curves. In *Proceedings of the Symposium on Solid and Physical Modeling*, pages 253–263, 2007.

[71] Denis Haumont and Nadine Warzée. Complete polygonal scene voxelization. *Journal of Graphics, GPU, and Game Tools*, 7(3):27–41, 2002.

[72] Paul Heckbert. Fundamentals of texture mapping and image warping. Master's thesis, University of California, Berkeley, 1989.

[73] R. D. Hersch and (Ed.). Font rasterization, the state of art. In *Visual and Technical Aspects of Type*, pages 78–109. Cambridge University Press, 1993.

[74] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Surface reconstruction from unorganized points. In *Proceedings of ACM SIGGRAPH*, pages 71–78, 1992.

[75] Ren Huang and Soo-Ik Chae. Implementation of an openvg rasterizer with configurable anti-aliasing and multi-window scissoring. In *Proceedings of the International Conference on Computer and Information Technology*, pages 179–185, 2006.

[76] Robert Hummel. Sampling for spline reconstruction. *SIAM Journal on Applied Mathematics*, 43(2):278–288, 1983.

[77] Tobias Hüttner and Wolfgang Straßer. Fast footprint mipmapping. In *Proceedings of the ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 35–44, 1999.

[78] Homan Igehy, Matthew Eldridge, and Kekoa Proudfoot. Prefetching in a texture cache architecture. In *Proceedings of the ACM SIGGRAPH/Eurographics workshop on Graphics Hardware*, pages 133–143, 1998.

[79] Frank Dachille Ix and Arie Kaufman. Incremental triangle voxelization. In *Proceedings of Graphics Interface*, pages 205–212, 2000.

[80] Thouis R. Jones and Ronald N. Perry. Antialiasing with line samples. In *Proceedings of the Eurographics Workshop on Rendering Techniques*, pages 197–206, 2000.

[81] Norman P. Jouppi and Chun-Fa Chang. Z3: An economical hardware technique for high-quality antialiasing and transparency. In *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 85–93, 1999.

[82] T. Ju, F. Losasso, S. Schaefer, and J. Warren. Dual contouring of Hermite data. *ACM Transactions on Graphics*, 21(3):339–346, July 2002.

[83] J. Kajiya and M. Ullner. Filtering high quality text for display on raster scan devices. In *Proceedings of ACM SIGGRAPH*, pages 7–15, 1981.

[84] Kiia Kallio. Scanline edge-flag algorithm for antialiasing. In *Proceedings of Theory and Practice of Computer Graphics*, pages 81–88, 2007.

[85] Michael Kazhdan. Reconstruction of solid models from oriented point sets. In *Proceedings of the Eurographics Symposium on Geometry Processing*, pages 73–84, 2005.

[86] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Proceedings of the Eurographics Symposium on Geometry Processing*, pages 61–70, 2006.

[87] Michael Kazhdan, Allison Klein, Ketan Dalal, and Hugues Hoppe. Unconstrained isosurface extraction on arbitrary octrees. In *Proceedings of the Eurographics Symposium on Geometry Processing*, pages 125–133, 2007.

[88] Samuli Laine and Timo Aila. A weighted error metric and optimization method for antialiasing patterns. *Computer Graphics Forum*, 25(1):83–94, 2006.

[89] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. In *Proceedings of ACM SIGGRAPH*, pages 579–588, 2006.

[90] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3d scanning of large statues. In *Proceedings of ACM SIGGRAPH*, pages 131–144, 2000.

[91] Zhouchen Lin, Hai-Tao Chen, Heung-Yeung Shum, and Jian Wang. Optimal polynomial filters. *Journal of Graphics Tools*, 10(1):27–38, 2005.

[92] Zhouchen Lin, Hai-Tao Chen, Heung-Yeung Shum, and Jian Wang. Prefiltering two-dimensional polygons without clipping. *Journal of graphics, GPU, and game tools*, 10(1):17–26, 2005.

[93] Johann Linhart. A quick point-in-polyhedron test. *Computers and Graphics*, 14(3-4):445–447, 1990.

[94] Charles Loop and Jim Blinn. Resolution independent curve rendering using programmable graphics hardware. *ACM Transactions on Graphics*, 24(3):1000–1009, 2005.

[95] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 21(4):163–169, 1987.

[96] S.G. Mallat. A theory for multiresolution signal decomposition: the wavelet representation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 11(7):674–693, 1989.

[97] Pavlos Mavridis and Georgios Papaioannou. High quality elliptical texture filtering on gpu. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, pages 23–30, 2011.

[98] Michael D. McCool. Analytic antialiasing with prism splines. In *Proceedings of ACM SIGGRAPH*, pages 429–436, 1995.

[99] Joel McCormack, Ronald N. Perry, Keith I. Farkas, and Norman P. Jouppi. Feline: Fast elliptical lines for anisotropic texture mapping. In *Proceedings of ACM SIGGRAPH*, pages 243–250, 1999.

[100] Miriah Meyer, Ross Whitaker, Robert M. Kirby, Christian Ledergerber, and Hanspeter Pfister. Particle-based sampling and meshing of surfaces in multimaterial volumes. *IEEE Trans. Vis. Comp. Grap.*, 14(6):1539–1546, 2008.

[101] Don P. Mitchell. Generating antialiased images at low sampling densities. In *Proceedings of ACM SIGGRAPH*, pages 65–72, 1987.

[102] Don P. Mitchell and Arun N. Netravali. Reconstruction filters in computer-graphics. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 22:221–228, 1988.

[103] Andrew Mor, Sarah Gibson, and Joseph Samosky. Interacting with 3-dimensional medical data: Haptic feedback for surgical simulation. In *Proceedings of the Phantom User Group Workshop*, 1996.

[104] Diego Nehab and Hugues Hoppe. Random-access rendering of general vector graphics. In *Proceedings of ACM SIGGRAPH Asia*, pages 135:1–135:10, 2008.

[105] Yutaka Ohtake, Alexander Belyaev, Marc Alexa, Greg Turk, and Hans-Peter Seidel. Multi-level partition of unity implicits. *ACM Transactions on Graphics*, 22(3):463–470, 2003.

[106] Stanley J. Osher and Ronald P. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer, 2002.

[107] Ryan S. Overbeck, Craig Donner, and Ravi Ramamoorthi. Adaptive wavelet rendering. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia)*, 28(5):140:1–12, 2009.

[108] Budirijanto Purnomo, Jonathan D. Cohen, and Subodh Kumar. Seamless texture atlases. In *Proceedings of the Eurographics Symposium on Geometry Processing*, pages 65–74, 2004.

[109] Zheng Qin, Michael D. McCool, and Craig Kaplan. Precise vector textures for real-time 3D rendering. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, pages 199–206, 2008.

[110] Zheng Qin, Michael D. Mccool, and Craig S. Kaplan. Real-time texturemapped vector glyphs. In *Proceedings of the Symposium on Interactive 3D Graphics and*

Games, pages 125–132, 2006.

[111] Sundaresan Raman and Rephael Wenger. Quality isosurface mesh generation using an extended marching cubes lookup table. *Computer Graphics Forum*, 27(3):791–798, 2008.

[112] Nicolas Ray, Vincent Nivoliers, Sylvain Lefebvre, and Bruno Lévy. Invisible seams. In *Proceedings of the Eurographics Symposium on Rendering*, 2010.

[113] Bernhard Reitinger, Alexander Bornik, and Reinhard Beichel. Constructing smooth non-manifold meshes of multi-labeled volumetric datasets. In *Proceedings of Computer Graphics, Visualization and Vision*, pages 227–234, 2005.

[114] Pedro V. Sander, John Snyder, Steven J. Gortler, and Hugues Hoppe. Texture mapping progressive meshes. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, pages 409–416, August 2001.

[115] Scott Schaefer and Joe Warren. Dual marching cubes: Primal contouring of dual grids. In *Proceedings of Pacific Graphics*, pages 70–76, 2004.

[116] O. Schall and M. Samozino. Surface from scattered points: a brief survey of recent developments. In *Proceedings of the International Workshop on Semantic Virtual Environments*, pages 138–147, 2005.

[117] Oliver Schall, Alexander Belyaev, and Hans-Peter Seidel. Error-guided adaptive fourier-based surface reconstruction. *Computer Aided Design*, 39(5):421–426, 2007.

[118] A. Schilling, G. Knittel, and W. Strasser. Texram: a smart memory for texturing. *Computer Graphics and Applications, IEEE*, 16(3):32–41, 1996.

[119] John Schreiner, Carlos Scheidegger, and Claudio Silva. High-quality extraction of isosurfaces from regular and irregular grids. *IEEE Trans. Vis. Comp. Grap.*, 12(5):1205–1212, 2006.

[120] J. Sethian. *Level set methods and fast marching methods. Evolving interfaces in computational geometry, fluid mechanics, computer vision, and materials science*. Cambridge University Press, 2 edition, 1999.

[121] R. M. Shapley and D. J. Tolhurst. Edge detectors in human vision. *Journal of Physiology*, 229(1):165–183, 1973.

[122] James E. Sheedy and Molly McCarthy. Reading performance and visual comfort with scale to grey compared with black-and-white scanned print. *Displays*, 15(1):27–30, 1994.

[123] Christian Sigg, Ronald Peikert, and Markus Gross. Signed distance transform using graphics hardware. In *Proceedings of IEEE Visualization*, pages 83–90, 2003.

[124] Avneesh Sud, Miguel A. Otaduy, and Dinesh Manocha. Difi: Fast 3D distance field computation using graphics hardware. *Computer Graphics Forum*, 23(3):557–566, 2004.

[125] Sébastien Thon, Gilles Gesquière, and Romain Raffin. A low cost antialiased space filled voxelization of polygonal objects. In *Proceedings of GraphiCon*, pages 71–78, 2004.

[126] Nils Thürey, Chris Wojtan, Markus Gross, and Greg Turk. A multiscale approach to mesh-based surface tension flows. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 29(3):48:1–10, 2010.

[127] Sivan Toledo, Doron Chen, and Vladimir Rotkin. TAUCS 2.2. http://www.tau.ac.il/ stoledo/taucs/, 2003.

[128] Huamin Wang, Yonatan Wexler, Eyal Ofek, and Hugues Hoppe. Factoring repeated content within and among images. In *Proceedings of ACM SIGGRAPH*, pages 14:1–14:10, 2008.

[129] Sidney W. Wang and Arie E. Kaufman. Volume sampled voxelization of geometric primitives. In *Proceedings of IEEE Visualization*, pages 78–84, 1993.

[130] Sidney W. Wang and Arie E. Kaufman. Volume-sampled 3d modeling. *IEEE Comp. Graph. and Appl.*, 14(5):26–32, 1994.

[131] John Edward Warnock. *A hidden surface algorithm for computer generated halftone pictures*. PhD thesis, 1969. The University of Utah.

[132] Li-Yi Wei. Multi-class blue noise sampling. *ACM Transactions on Graphics*, 29(4):79:1–79:8, 2010.

[133] William J. Welch. Algorithmic complexity: three NP-hard problems in computational statistics. *Journal of Statistical Computation and Simulation*, 15(1):17–25, 1982.

[134] Lance Williams. Pyramidal parametrics. In *Proceedings of ACM SIGGRAPH*, pages 1–11, 1983.

[135] Ziji Wu and John M. Sullivan. Multiple material marching cubes algorithm. *International Journal for Numerical Methods in Engineering*, 58(2):189–207, 2003.

[136] Chris Wylie, Gordon Romney, David Evans, and Alan Erdahl. Half-tone perspective drawings by computer. In *Proceedings of the American Federation of Information Processing Societies*, pages 49–58, 1967.

[137] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, 1986.

[138] Cem Yuksel, John Keyser, and Donald H. House. Mesh colors. *ACM Transactions on Graphics*, 29(2):1–11, 2010.

[139] Long Zhang, Wei Chen, David Ebert, and Qunsheng Peng. Conservative voxelization. *Visual Computer*, 23(9):783–792, 2007.

[140] Y. Zhang, D. Zhao, J. Zhang, R. Xiong, and W. Gao. Interpolation-dependent image downsampling. *IEEE Transactions on Image Processing*, 20(11):3291–3296, 2011.

[141] Lifeng Wang Zhouchen Lin and Liang Wan. First order approximation for texture filtering. In *Pacific Graphics Poster*, 2006.