Jump Map-Based Interactive Texture Synthesis

STEVE ZELINKA and MICHAEL GARLAND
University of Illinois at Urbana-Champaign

We present techniques for accelerated texture synthesis from example images. The key idea of our approach is to divide the task into two phases: analysis, and synthesis. During the analysis phase, which is performed once per sample texture, we generate a *jump map*. Using the jump map, the synthesis phase is capable of synthesizing texture similar to the analyzed example at interactive rates. We describe two such synthesis phase algorithms: one for creating images, and one for directly texturing manifold surfaces. We produce texture images at rates comparable to the fastest alternative algorithms, and produce textured surfaces an order of magnitude faster than current alternative approaches. We further develop a new, faster patch-based algorithm for image synthesis which improves the quality of our results on ordered textures. We show how controls used for specifying texture synthesis on surfaces may be used on images as well, allowing interesting new image-based effects, and highlight modelling applications enabled by the speed of our approach.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Color, shading, shadowing and texture*

General Terms: Algorithms

Additional Key Words and Phrases: jump maps, interactive texture synthesis, texturing surfaces

1. INTRODUCTION

The problem of texture synthesis has received much attention from the computer graphics research community in recent years. Simply stated, we are given a small sample of a texture, and wish to create "more" of it. An easy solution to this problem is to simply tile the sample texture. However, even if the sample allows seamless tiling, this is well-known to produce repetitive artifacts and patterns which are disturbingly noticeable and distracting, especially in textures depicting natural or relatively stochastic materials. Thus, we wish to produce texture such that the result is not obviously repetitive, while maintaining the statistical properties of the sample to some qualitative degree. The standard measure of success is that an average person would agree the sample and produced textures are depicting the same texture.

Texture synthesis techniques were first developed for use in the image domain, and indeed continue to have wide application for images. It is not unusual to have only a small sample of a texture available, and require more of it for applications such as replacing a textural element of an image (changing a wall from wood to a particular kind of brick, for

Author's address: S. Zelinka, Room 3238A, Department of Computer Science, 201 N Goodwin, Urbana, IL 61801-2302. Email: zelinka@uiuc.edu

Fig. 1. Typical results using jump map-based texture synthesis. Top, left to right: sample texture; image synthesized in 0.05 seconds using the pixel-based algorithm; sample texture; image synthesized in 0.02 seconds using our new patch-based algorithm. Bottom row: sample texture; surface textured with our algorithm in 0.07 seconds.

example). Another typical application [Wei and Levoy 2000; Drori et al. 2003; Jia and Tang 2003; Criminisi et al. 2003; Bertalmio et al. 2003] is to remove unwanted foreground elements of an image by synthesizing the background texture over them.

For games and other interactive applications, textures are an effective way to add significant visual detail to scenes without increasing their geometric complexity. Such texturing may pose significant modelling challenges, however, often requiring considerable time on the part of skilled artists to paint the surfaces of 3D objects. Procedural methods [Ebert et al. 1994] are useful for this problem but are generally limited to certain classes of textures such as woods or marbles, and may be difficult to control. Recent 3D scanning techniques [Hertzmann and Seitz 2003] may recover textures as well as geometry from objects, allowing this problem to be circumvented if an appropriate real-world example object may be found or created. However, in the general case, the ability to texture an object from an example texture has proven extremely useful, particularly for novice or artistically unskilled users. With the widespread availability of texture imagery due to image search engines and Internet-based collections, this capability only becomes more useful with time.

More generally, texture synthesis techniques can reduce the amount of texture that needs to be stored or initially created. Storing only the sample texture, and applying texture synthesis to recreate a much larger required texture, can be particularly useful for bandwidth-limited network applications. Texture synthesis can also reduce the artistic burden by

requiring only a small sample of a new texture to be created, with the system automatically computing the final texture of the requisite size.

Recently, a focus of texture synthesis work has been on delivering texture synthesis at interactive speeds. In this paper, we describe our techniques for accelerated texture synthesis based on the jump map [Zelinka and Garland 2002; 2003]. This framework is based on the idea of dividing the task of texture synthesis into two phases: *analysis* of the sample texture, and *synthesis* of new texture. The analysis phase, detailed in Section 4, generates a *jump map* for the texture, which records for each pixel a set of links to similar pixels. Note that this need only be done once per sample texture, and since this can generally be done in a preprocessing step, it need not be especially fast. Given the jump map, synthesizing new texture is very simple and efficient, involving copying successive pixels or patches of pixels from the input to the output, while occasionally following a jump recorded in the jump map. We describe algorithms for synthesizing images in Section 5, and our recent extension to texturing 3D surfaces in Section 6.

As we shall see in Section 7, this extension to surfaces can be directly applied to images as well. This produces two novel image synthesis results. First, we develop a fast patch-based extension to jump map-based image synthesis which can work well even on structured textures. Secondly, by treating the output image as a surface, we can apply the powerful control techniques used for texture synthesis on surfaces to image synthesis. These control techniques allow us to locally orient and scale the synthesized texture within the output image, allowing the simulation of perspective effects, surface variations, and more.

The capability to synthesize texture at interactive rates, as allowed by the jump map, itself enables a number of interesting new modelling tools, and removes some of the burden of working with automatic texture synthesis algorithms. As discussed in Section 8, texture parameters can be adjusted on the fly and decided interactively by the user. Texture paintbrushes, in which a virtual brush is used to paint a continuous, seamless texture over an object or image, are also easily accomodated by the speed of our synthesis algorithms. Imagery or scenery being dynamically generated can now be textured as well, as it is generated, without unduly slowing the system.

From a technical standpoint, the jump map framework for texture synthesis offers additional advantages. Our method for texture synthesis on surfaces is designed to output texture coordinates for the surface, which accomodates a range of texture scales without requiring any resampling of the input mesh, while also using no extra texture memory at runtime beyond that used by the sample texture. Our framework can further accomodate extensions such as progressively varying the scale of a texture across the surface.

2. PREVIOUS WORK

To put our framework for jump map-based texture synthesis in context, we begin with a survey of some of the more significant and influential recent work in the field of texture synthesis. First, we review core texture synthesis algorithms, followed by a discussion of recent generalizations to texture synthesis on surfaces. We then very briefly discuss some alternative approaches and applicable work in related fields, and conclude our review by noting some recent applications of texture synthesis algorithms to new problems.

2.1 Texture Synthesis Algorithms

In general, texture synthesis algorithms can be broadly classified into two categories: *pixel-based* methods, which perform iterative processing for every output pixel, and *patch-based methods*, which attempt to copy larger patches of texture from the input to the output.

Pioneering work in pixel-based approaches can be seen in the non-parametric sampling technique of Efros and Leung [1999], which iteratively finds and copies the best match in the input texture for the existing nearby pixels of the output texture. In other pioneering work, Heeger and Bergen [1995] and De Bonet [1997] used multi-resolution image pyramids to match texture statistics at multiple frequencies simultaneously. Wei and Levoy [2000] combined these two approaches with the use of a fixed-size neighbourhood, yielding impressive quality and an approach which could be accelerated with high-dimensional nearest neighbour or clustering techniques (they used tree-structured vector quantization). This approach was further refined for natural textures by Ashikhmin [2001], who recognized that the positions in the input around which nearby output pixels were copied are likely good matches for new output pixels. Balances between Wei and Levoy's accelerated best matching and Ashikhmin's *coherent* matching were struck by both Hertzmann *et al.* [2001], who used a parameter to weight and choose between the contributions of each, and Tong *et al.* [2002], whose *k-coherent search* matched against the candidates selected by Ashikhmin's algorithm as well as their k-nearest neighbours. As we shall see, a jump map provides a natural acceleration data structure for k-coherent search, and image synthesis with a jump map [Zelinka and Garland 2002] can be seen as a further step beyond Ashikhmin's coherent matching, in which the current output neighbourhood is removed altogether, and matches are performed only between input neighbourhoods.

Patch-based texture synthesis from an example was first shown by Xu *et al.* [2000]. Their *Chaos Mosaic* tiled the output with the sample texture, pasting and blending random overlapping blocks of texture to obscure the tiling and its boundaries. A more expensive, higher-quality approach was taken in Image Quilting [Efros and Freeman 2001], which generalizes the approach of Wei and Levoy to patches by matching patch boundaries, and finding the best cut through the error surface between overlapping patches. Liang *et al.* [2001] instead use a number of acceleration data structures to speed patch matching to real-time performance, and simply blend boundaries. Alternately, a small set of Wang Tiles for a texture have been shown to be sufficient to efficiently produce arbitrarily large amounts of the texture [Cohen et al. 2003]. For extremely high quality results, a Graph Cut-based algorithm [Kwatra et al. 2003] formulates the patch boundary problem as a max flow/min cut problem, generalizing the Image Quilting algorithm to arbitrarily shaped patches and arbitrary dimension (e.g., video).

In general, pixel-based methods are good at matching the smaller details of textures, but have trouble reproducing emergent or global features. Patch-based methods, on the other hand, capture global features of the texture well, but may suffer from artifacts from trying to fit poorly matched boundaries. Recognizing this, Nealen and Alexa [2003] propose a hybrid algorithm which initially places large patches, and refines as necessary to reduce boundary artifacts. In areas where artifacts are unavoidable, a pixel-based resynthesis algorithm is used. While slow, the approach gains the quality benefits of both classes of algorithms.

2.2 Texture Synthesis On Surfaces

Texture synthesis directly on arbitrary surfaces has received increasing attention in recent years. Researchers have long recognized that the ability to create a larger texture image from an example is not quite adequate to this task, as dealing with the potentially challenging topology of a surface can pose difficult parameterization problems.

The first approaches to this problem were patch-based. Neyret and Cani [1999] manually create a set of triangle tiles, each of whose edges are known to match with all other tile edges; the results are limited to textures suitable for this process and relatively uniformly-sampled meshes, and somewhat akin to (though more limited than) the image-based Wang Tiles approach. Later, Lapped Textures [Praun et al. 2000] were introduced, which can be seen as a generalization of the Chaos Mosaic to surfaces, repeatedly pasting an overlapping, irregularly-shaped patch of texture on the surface and relying on blending to hide discontinuities.

High quality results for general textures were produced simultaneously by Wei and Levoy [2001] and Turk [2001], by generalizing pixel-based approaches to surfaces. These approaches involved locally flattening and resampling the mesh into a regular neighbourhood grid, followed by application of standard neighbourhood matching. Both of these approaches reproduce the texture on the mesh by assigning vertex colours, with the drawback that the apparent size of the texture relative to the mesh (the texture *scale*) is dictated by the vertex density. This approach has been extended to synthesize Bidirectional Texture Functions [Tong et al. 2002] and progressively-variant textures [Zhang et al. 2003]. Ying *et al.* [2001] instead divide the surface into a number of charts, and apply an image-based approach to synthesize texture for each chart, being careful to appropriately sample from neighbouring charts as necessary. This decouples the texture scale from the vertex density, since the chart resolutions may be easily changed, but potentially requires an excessively wasteful amount of texture memory to hold the charts.

By generating texture coordinates instead of vertex colours or charts, Soler *et al.* [2002] can accomodate a range of texture scales without remeshing, and use texture memory only for the sample texture. They attempt to position sets of hierarchical face clusters into texture space, and accelerate cluster boundary matching with a discrete Fourier transform. As necessary, patches are subdivided down to the level of individual triangles to increase match quality (this approach inspired the patch-based part of Hybrid Texture Synthesis [Nealen and Alexa 2003]). A much faster though not quite interactive approach is proposed by Magda and Kriegman [2003], who compute a set of "texton labels" [Leung and Malik 1999] for each pixel of the sample by clustering gaussian-weighted neighbourhoods. Triangles are then simply rotated and scaled into texture space, and finally placed at the position which has the highest number of labels matching the neighbours' texton labels. Because texton label matching provides a more accurate approximation to neighbourhood matching than the jump map, their results are slightly higher in quality than ours. However, larger input samples require more texton labels, which makes matching more computationally expensive, and thus slows the running time significantly. Even with small samples, their approach is still an order of magnitude slower than the method we present here. A similar approach was developed by Dischler *et al.* [2002], decomposing the texture into mostly user-identified patches of texture called "texture particles". The spatial arrangement of the instances of each kind of particle in the sample image are analyzed and simply reproduced in the output. The method is similarly fast, and allows transformations

to be applied to each particle (enlarging or rotating elements, for example), but not quite interactive in speed and texture particles may be severely distorted on surfaces.

2.3 Alternate Approaches

A few alternate methods for solving the problems which are solved by texture synthesis from examples bear mentioning here. The largest class of alternate methods are procedural in nature. Noise-based methods [Ebert et al. 1994] have been widely used to represent certain classes of textures, including marble and wood. A key advantage of these procedural methods is that they may be evaluated in any order and sampled arbitrarily, as they are based on an underlying mathematical function. By contrast, almost all texture synthesis methods have order dependencies (with the notable exceptions of the Chaos Mosaic [Xu et al. 2000] and an era-based method by Wei [2002]). Carr and Hart [2002] generate a mipmappable texture atlas for a surface, and sample 3D noise into it in real-time to texture the surface. However, these procedural methods are relatively difficult to control, and not suitable for all kinds of textures.

Other notable procedural approaches include Cellular Texturing [Fleischer et al. 1995], and a feature-based extension [Legakis et al. 2001]. Cellular texturing consists of simulating a network of interacting cells on a surface, each of which follows a user-defined "cell program" to generate some new geometry. Since cell programs may interact, the generated geometry may be consistent across the surface, but the cell programs may be difficult to write. The approach of Legakis *et al.* instead assigns feature labels to parts of a surface, and directs slightly different cell programs to each kind of feature. In this way, one may have structures which are consistent across features, such as bricks properly meeting at corners of a building. However, this requires more cell programs to be written, and the labelling of features on the surface.

Finally, the connection between texture synthesis on surfaces and parameterization bears mentioning. Patch-based texture synthesis methods, which assign texture coordinates to triangle corners, are effectively parameterizing the surface such that the texture appears continuous across the surface. A natural alternative, therefore, is to simply apply a general parameterization algorithm to the surface (e.g., [Piponi and Borshukov 2000; Sander et al. 2002; Sheffer and Hart 2002]), and simply synthesize a texture image to be mapped by the parameterization (or, indeed, use the original sample). There are several problems with this approach: similar to the chart-based methods, extra texture memory must be used as the texture scale shrinks with respect to the surface; a full parameterization must invariably introduce distortion of the surface in texture space, which can be avoided entirely by texture synthesis methods; extra effort must be expended to match texels across seams of the parameterization; and the user does not have much control over the final result. In particular, anisotropic textures generally need to be oriented on the surface by the user, and the desired orientation may not be consistent with that dictated by the parameterization algorithm. However, this may still be a useful alternative approach if the surface is already parameterized, or needs to be for some other purpose.

2.4 Applications

While much research has focused on increasing the speed and quality of texture synthesis algorithms, a number of new innovative applications of texture synthesis research have surfaced in recent years. One early application of texture synthesis was the removal of foreground elements from images [Wei and Levoy 2000] by synthesizing new background

texture over them. This approach has been alternately refined by using a confidence map-based traversal order [Drori et al. 2003; Criminisi et al. 2003], adaptive ND tensor voting and texture segmentation [Jia and Tang 2003], and image decomposition with a combination of inpainting and texture synthesis [Bertalmio et al. 2003]. Generalizing images to a third dimension, numerous researchers have worked on the problem of video synthesis [Wei and Levoy 2000; Schödl et al. 2000; Kwatra et al. 2003]. Ying *et al.* [2001] interpret the synthesized texture in novel ways, for example as a transparency or displacement map, generating stunning imagery. Similarly, Cohen *et al.* [2003] showed how Wang Tiles could be used for primitive distribution as well as texture synthesis. Zhang *et al.* [2003] have developed an innovative framework for progressive variation of texture-related signals across a surface, demonstrating varying scales of texture elements, and even smoothly varying between two textures across a surface.

An interesting class of applications uses alternate sources of information in generating the texture. A pioneering application of this type includes Ashikhmin's texture transfer technique [Ashikhmin 2001], in which the best matching neighbourhood is redefined to simultaneously match the current output neighbourhood and the corresponding neighbourhood in a target image. Harrison [2001] uses region maps over the input and output to decouple the colouring of the texture from the region identification for texture transfer. While Ashikhmin uses simple hand-painted target images, Efros and Freeman [2001] demonstrate excellent results on general target images. Image Analogies [Hertzmann et al. 2001] reproduces local transformations among images using a single example of the transformation. Their framework supports a number of filtering operations, from image super-resolution to non-photorealistic filters such as watercolour, but may require extensive parameter tuning and colour matching. Ashikhmin has produced a much faster variant of this system with only slightly lower quality by using a slightly randomized coherent search [Ashikhmin 2003]. A number of targeted techniques perform similar filtering operations with generally higher quality, including colorization of black and white images from an example [Welsh et al. 2002] and facial super-resolution [Liu et al. 2001]. All of these methods use accelerated neighbourhood matching techniques widely used for texture synthesis.

3. JUMP MAPS

The starting point for our framework for texture synthesis using jump maps is the approach of Wei and Levoy [2000], which iteratively places output pixels by finding the best match for the current output neighbourhood in the input image. The key idea behind the jump map texture synthesis algorithms is the notion that we can precompute the set of neighbourhood comparisons that will be required for any particular application of texture synthesis, and thereby avoid having to perform any costly neighbourhood matching at run-time. To perform this precomputation, we make the assumption that the neighbourhood of each output pixel to be synthesized will closely resemble the input neighbourhoods from which the nearby already-synthesized pixels were copied. Thus, the comparisons which would be performed at runtime are approximated by comparisons among input image neighbourhoods, the results of which are stored in a lookup table we call the *jump map*. Indeed, in contrast to previous algorithms, our runtime synthesis procedure need not examine pixels of the sample texture at all.

For each pixel of the input image, the jump map stores a list of *jumps* to other pixels whose neighbourhoods are similar, as pictured in Figure 2. Note that, in addition to the

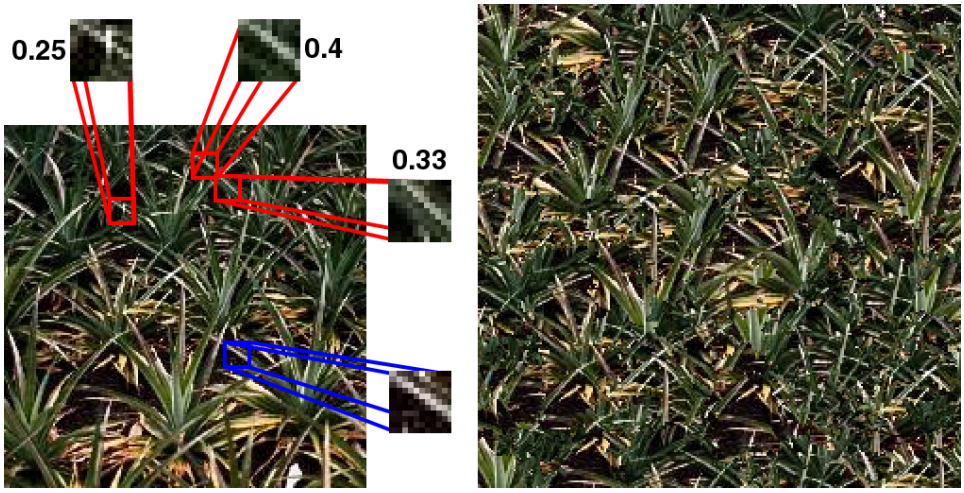


Fig. 2. Left: At each pixel, such as the one centered on the blue box, the jump map stores a set of links to pixels with similar neighbourhoods, outlined in red. Each jump is weighted according to the similarity of the neighbourhoods. Right: Synthesized result.

destination address of each jump, we also store a similarity value for each jump. Jumps between pixels which match extremely well have higher similarity values than jumps between pixels which do not match so well. We typically store the best 2–4 jumps per pixel, subject to some diversity constraints. Note that more jumps could easily be stored per pixel, but we keep the number low to minimize storage requirements and runtime memory requirements. The details of how we establish the set of jumps to be stored at each pixel, and how we compute and normalize similarities, are given in Section 4.

The process of synthesizing new texture using a jump map (see Section 5) is similar to the process used in Video Textures [Schödl et al. 2000] to generate endless amounts of video from an example video. It is interesting to note that this paradigm has also recently been successfully applied to motion capture data [Kovar et al. 2002; Arikan and Forsyth 2002; Lee et al. 2002], allowing unlimited new motion to be generated from a limited set of motion capture data. We traverse the output image following a particular pixel ordering, and the usual case is that pixels are simply copied one after the next from the input to the output. Occasionally, and especially as an input image boundary is approached, a decision is made to perform a jump through the jump map. To do this, we look up the jumps stored in the jump map for the current position in the input image, and randomly select a destination from among them. We then simply resume copying pixels from this new region of the input image. This ensures that we do not run out of texture to copy (and simply tile the input over the output, or create other noticeable image boundary artifacts) and also encourages a certain amount of randomness in the output, as the decision of when to jump and the choice of destination are both probabilistically determined. Note that this entire synthesis procedure operates only on texture addresses, and need not look at pixels of the sample image at all. As we shall see, the key difference between our problem and that of Video Textures is that our output must be coherent over two dimensions, rather than one. We overcome this by carefully choosing the order in which pixels are synthesized.

Generalizing the image-based algorithm to create texture on surfaces is quite straightforward, and mainly involves tackling the same problems facing other such generalizations. In particular, we must adapt the algorithm to deal with non-regular surface topologies and non-uniform distances between neighbours. Details of these issues and our solutions are given in Section 6.

Our application of jump maps to surfaces in fact leads directly to a new patch-based algorithm for synthesizing images with jump maps. This results in an even faster image synthesis algorithm, which we describe in Section 7.1.

4. TEXTURE ANALYSIS: GENERATING JUMP MAPS

Each texture to be synthesized must first go through an analysis phase, in which we generate a jump map for it. For each pixel of the input texture to be analyzed, we compute a set of similar pixels to be stored in the pixel's corresponding jump map entry (jump list). Like previous approaches, we use simple fixed-size neighbourhood comparisons to determine the similarity between pixels (§4.1). We apply relatively standard acceleration techniques to speed the search for good matching neighbourhoods (§4.2), and further acceleration can be achieved by using multi-resolution neighbourhoods (§4.3). In Section 4.4, we detail a few minor issues helpful to ensure sufficient randomness in generated jump maps. Finally, we discuss some normalization issues in Section 4.5.

4.1 Neighbourhood Similarity

We define the neighbourhood of a pixel as an ordered set of pixels around the target pixel (we typically simply use square neighbourhoods centered on the target pixel). To compare two neighbourhoods, we use the L_2 norm: we first construct neighbourhood vectors, which consist of the colour values at each pixel in the neighbourhood concatenated into one long vector. The difference between two neighbourhoods is then quantified as the length of the difference between their neighbourhood vectors. While the L_2 norm is well-known to be a poor choice for perceptual similarity between images, it is very fast to compute and performs adequately for texture synthesis. Indeed, Nealen and Alexa [2003] suggest the use of metrics based on human perception do not noticeably improve their results.

Computing a set of similar pixels to a target pixel then amounts to a high-dimensional all-nearest-neighbours problem: we wish to find the nearest neighbours of vectors of dimension cn^2 (for a neighbourhood of $n \times n$ pixels and image with c colour channels). Since we only need generate a jump map once for any particular texture, the speed with which we solve this problem is not of utmost importance; indeed, one could simply use brute force, comparing against all other neighbourhoods in the image directly.

4.2 Accelerated Matching

Instead, following previous researchers [Hertzmann et al. 2001; Liang et al. 2001], we insert all the neighbourhood vectors into an approximate nearest neighbours (ANN) search data structure for accelerated searching [Mount 1998]. In our experiments, we have found that we do not need the *actual* best matches of a particular target neighbourhood, and indeed typically use error tolerances of 10%. However, since ANN search time expotentially increases with the vector dimension, we first apply principal components analysis (PCA) [Jollife 1986] to reduce the dimension of the neighbourhood vectors. We typically randomly sample 10% of the set of image neighbourhoods to use to compute the principal components, and use enough components to retain 97% of the variation in this sampling.

We have found these techniques to work exceedingly well in practice. Whereas brute force analysis for a texture can take from half an hour to several hours, depending on the input image size and neighbourhood size, applying these acceleration techniques brings analysis time down to anywhere from a few seconds to a few minutes. PCA is critical to achieve these efficiency gains, as ANN works best at relatively modest vector dimensions (under 30 or so). Only rarely do we observe sample textures requiring more components than this.

It should be noted that at image boundaries, we do not have full neighbourhoods available. Our choice of dealing with this problem depends on the sample's tileability: if the sample image is seamlessly tileable, we simply use toroidal neighbourhoods, wrapping around the image when we encounter a border; if the image is not seamlessly tileable, then we simply inset the sample image, and only consider the region where full neighbourhoods are available. Note that this can cause problems for particularly small sample textures that require large neighbourhood sizes, as there could be little left of the sample image after insetting. There are a number of alternate possibilities which we have not explored in detail, for example: using truncated neighbourhoods (requiring brute force searches for those pixels with truncated neighbourhoods); reflecting the image over the image boundaries, effectively creating a seamless image; or filling the unavailable entries of a neighbourhood with some sort of reasonable data (such as an average or median pixel). In practice, very few textures are small enough to present a problem for insetting in any case.

4.3 Multi-Resolution Neighbourhoods

As just discussed, limiting the dimension of the neighbourhood vectors is critical to achieving large efficiency gains in constructing jump maps. Perhaps unintuitively, one way to further reduce neighbourhood dimension is to use multi-resolution neighbourhoods. Here, we downsample the sample texture a number of times to create a Gaussian image pyramid, and concatenate corresponding neighbourhoods from each level of the pyramid onto the existing neighbourhood vectors. The entries in the multi-resolution neighbourhood vector from each level of the pyramid are weighted such that the total weight from each level of the pyramid is equal (for example, if a coarser level has half as many pixels in its neighbourhood, those pixels are each given twice the weight given to each finer level pixel). Typically, we use three levels for our multi-resolution neighbourhoods.

In our experience, PCA does a much better job on these multi-resolution neighbourhood vectors than it does on flat neighbourhood vectors. Thus, while the uncompressed vectors are longer than single resolution vectors, the PCA-compressed vectors are smaller when including multi-resolution entries. We theorize that this is because the multi-resolution neighbourhoods effectively capture the spatial nature of the data. The entries from each coarser level link the otherwise unrelated pixels in the corresponding finer level according to their spatial positions within the neighbourhood. It is then this spatial linking which leads to greater dimensional reductions from PCA. In general, the use of multi-resolution neighbourhoods also supports high quality matching with smaller finest-level neighbourhoods than would be required for flat neighbourhoods.

4.4 Avoiding Repetition

The primary deficiency of the L_2 norm for our purposes is its tendency to cluster around good matches. In particular, if one neighbourhood is a particularly good match, it often remains a relatively good match when shifting it over by a pixel. Indeed, an early version

of our software gave best matches for a pixel that were often right next to the pixel itself. Alternately, all of the matches would be clustered right next to one another in one particular portion of the image. This produces problems for texture synthesis since it is likely to induce an undesirable amount of repetition. For example, all of the jumps from one position may lead to one other particular position, and all of the jumps from that position may lead back to the original position.

A simple solution to avoid this problem is Poisson Disc Sampling. Instead of gathering the k nearest neighbours for each pixel, we instead gather the ks nearest neighbours, and filter the results such that placing a disc at each accepted result does not create any overlapping discs. We simply sort the ks nearest neighbours by similarity, and iteratively accept the first k neighbours which are not within some distance d of any other accepted neighbour (or the pixel for which we are finding nearest neighbours).

Typically, we retrieve 5 to 10 times as many neighbours as we want to keep (i.e., $s = 5$ to 10), and use Poisson discs with diameter (d) equal to the neighbourhood width. In general, s trades off between computation time and fidelity to the Poisson disc sampling criterion; a higher s means retrieving more neighbours, increasing computation time, while if s is too low, there may not be a k-subset of ks neighbours retrieved that satisfies the Poisson disc criterion. In this case, we iteratively halve d until we accept enough neighbours. In choosing d, the diameter of the Poisson discs, we would generally like at most one candidate from any particular identifiable feature of the sample texture. Since the neighbourhood width is assumed to be a rough measure of feature size, the neighbourhood width is a natural choice for d.

4.5 Jump Map Normalization

The final step in generating a jump map is to normalize the entries. In addition to the destination for each jump, we also store a normalized similarity value, which, as we shall see in Section 5, is used as a weighting term in the probabilistic selection of jumps. To simplify texture synthesis at runtime, we normalize the weights of all jumps such that the summed weight in any given pixel's jump list is at most 1.0. In doing this, we wish to recognize that not all jumps are equal. Some pixels may have high-quality jumps, while other pixels may only have mediocre jumps, and we would like to discourage jumping when only mediocre jumps are available. Thus, we globally filter the *sum* of the neighbourhood differences of the jumps at each pixel to lie in the range $[\alpha, 1], 0 \leq \alpha \leq 1$. Note that higher neighbourhood differences imply worse similarity, so high sums are mapped near to α, and lower sums are mapped closer to 1. Given this globally-assigned sum of weights at some pixel, the jumps at that pixel are assigned weights by filtering their neighbourhood differences relative to this sum. We typically use $\alpha = 0.5$, meaning that the pixel with the best jumps is roughly twice as likely to be jumped from as the pixel with the worst jumps. We have found simple linear filtering to be adequate for both the global normalization and the local weight distribution at each pixel.

4.6 Performance

Given our optimizations and constraints, analysis time varies from a few seconds to a few minutes for most textures on an Athlon 1800+ PC. There are a number of sources of variation here. First, the requisite neighbourhood size, directly influencing the initial dimension of neighbourhood vectors, has a large impact on the analysis time. Generally, the neighbourhood should be large enough to capture significant texture features. For most textures

in this paper, we have used $9\times9, 5\times5, 3\times3$ multi-resolution neighbourhoods. Larger neighbourhood sizes rarely result in dramatic quality improvement, but can increase analysis times significantly (e.g., on the order of half an hour for $17\times17, 9\times9, 5\times5$ neighbourhoods). Related to this, colour images, having three (or more) channels per pixel require neighbourhood vectors three times as long as greyscale images. Thus, a standard optimization in texture synthesis is to perform comparisons in the luminance domain, reducing neighbourhood dimension by a factor of three. While this would improve jump map construction time, it could reduce match quality for textures having colour differences not captured by luminance comparisons, and analysis is performed offline in any case. Another major influence on analysis time is the nature of the texture itself, and in particular how effective PCA is at reducing its neighbourhood dimension. A key area for future research is to attempt to understand how some textures are much more amenable to PCA reduction than others. Finally, analysis time is also influenced by the size of of the sample texture. Larger textures require larger jump maps, which means more ANN searches are required. Further, larger textures also have more possibilities for matches, so each ANN search itself takes more time.

5. IMAGE SYNTHESIS WITH JUMP MAPS

Now that we know how jump maps are created, we turn to the problem of synthesizing new images given a texture and jump map for the texture. As alluded to earlier, our approach is quite similar in nature to Video Textures, with the additional problem that we must attempt to ensure coherent synthesis across two dimensions instead of one. We simply choose a random start position in the sample image, and iteratively copy pixels from the input to the output. Based on a random choice, we also periodically follow a jump recorded in the jump map, and continue copying pixels from the destination of the jump. It turns out that a critical component of generalizing this approach to work well over images is the *order* in which pixels are synthesized. We begin our discussion of this algorithm in Section 5.1 by detailing the processing that occurs at each pixel, and continuing in Section 5.2 with our boundary avoidance scheme. A good understanding of this per-pixel processing develops the intuition necessary to discern the merits of various schemes for pixel orderings, which we discuss in Section 5.3.

5.1 Per-Pixel Processing

To each pixel of the output image, we must assign the address of a pixel from the input image. Given a jump map for the input texture, this assignment is very simple and especially efficient to evaluate. We simply examine the immediate neighbours of the pixel to be synthesized, and pick at random one which has already been assigned an address to use as a source. In the usual case, we simply copy the next pixel from the source's address. For example, if we chose the left neighbour, we would assign the pixel to the right of the source's address. Occasionally, however, we instead choose one of the jumps in the jump list stored in the jump map at the source's address, and use the jump destination as a "virtual source address". If, again, we chose the left neighbour, we would then copy the pixel to the right of this virtual source. Note that in all cases we use uniform random distributions.

The choice of which neighbour to use as a source is a simple random choice from the available set of already-synthesized neighbours (which itself depends on the order in which pixels are synthesized, described below). The choices of whether to jump, and which jump to use, are determined simultaneously by generating a single random number. The random

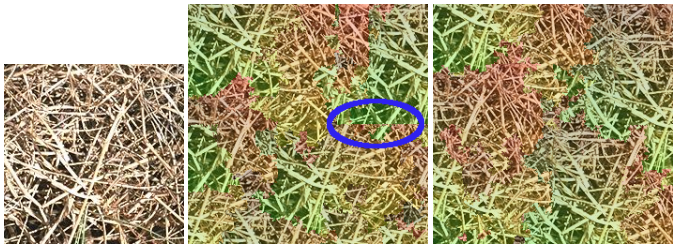


Fig. 3. Left: Sample texture. Middle: Synthesized texture without boundary avoidance. We have emphasized the patch boundaries by modulating each output pixel according to the position from which it was copied in the input image (thus, colour discontinuities correspond to patch boundaries). Notice the artifact in the middle right of the image where the input image boundary was hit on a line of successive pixels. Right: Synthesized texture with boundary avoidance.

number acts as an index into the jump list, where the jumps are indexed by cumulative probability. Thus, if we have two jumps listed, one with weight 0.4 and one with 0.5, the first would be selected if the random number is in $[0, 0.4)$, while the second would be selected if the random number is in $[0.4, 0.9)$. If the generated random is 0.9 or higher, then no jump is performed. Note that since we have normalized the jump lists to sum to at most 1.0 (§4.5), any random number generated above 1.0 means there is no chance of jumping. This implies a simple frequency-based user-specified control for how often jumps should occur: we simply scale the range over which the random numbers are generated in order to control the frequency of jumps. In our experiments, we have found that having patches on the order of one-third to three-quarters of the sample image size works best for most textures. We thus generate uniform random numbers in the range of 0 to twice this desired patch size, as the expected number of pixels between jumps is one half of the range.

5.2 Boundary Avoidance

One remaining problem is that, as given, our synthesis algorithm will tend to continue copying pixels right up to the boundary of the sample texture. This problem manifests itself particularly severely if a number of neighbouring pixels also come up against the boundary. While a one-pixel artifact may not be noticeable, an entire line of boundary may be especially noticeable, as shown in Figure 3. Note, of course, that this is not an issue for sample textures which are seamlessly tileable; also, recall that for non-tileable textures we inset the boundaries to get full neighbourhoods during analysis, and thus we refer to the inset boundaries below.

Our solution to this is to simply increase the probability of jumping as a boundary is approached. Thus, whenever a source address is within a fraction of the image size of the image boundary (we typically use one-fifth), we linearly decrease the random generation range down to sum of the jump weights of the corresponding jump list (guaranteeing a jump at the boundary). Thus, different neighbouring pixels are likely to jump at different distances from the image boundary, making line-type artifacts highly unlikely.

Note that there is an inverse problem here as well: if a jump lands near an image boundary, another jump will undoubtedly soon be required. This could potentially cause artifacts due to smaller patches being generated, as smaller patches make it more unlikely that our method's underlying assumption holds (that the current output neighbourhood resem-

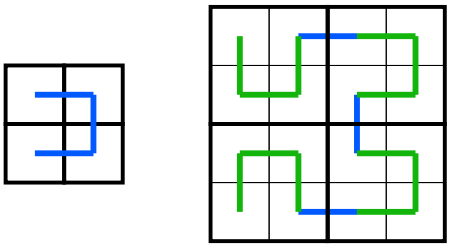


Fig. 4. Hilbert curves. A set of four cells is traversed in a U-shaped pattern (left). The curve may be subdivided (right) along with the cells while respecting the coarse-level traversal order with a simple L-system.

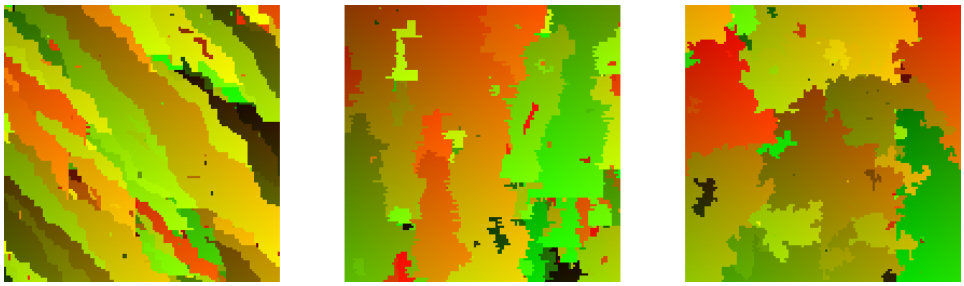


Fig. 5. Comparison of patches formed by different pixel orderings. Instead of copying the input pixels to the output, we directly visualize the pixel addresses from which we would have copied pixels, so colour discontinuities correspond to separate patches. Orderings are, left to right: scanline, serpentine, Hilbert path-based. This general behaviour of each ordering is roughly independent of the texture being synthesized.

bles the input neighbourhoods of the neighbouring output pixels). One way to deal with this would be to bias the jumps listed in the jump map, such that destinations near image boundaries receive lower weights. In our experience, however, this is generally not required, especially when following the Hilbert path-based pixel ordering, which changes direction very frequently.

5.3 Pixel Orderings

We have examined three different pixel orderings: scanline ordering, which simply goes left to right, top to bottom through the image; serpentine ordering, which alternates left to right, right to left, from top to bottom; and Hilbert path-based ordering, which follows the well-known Hilbert path [Sagan 1994] through an image, illustrated in Figure 4. Starting with the string "L", a Hilbert path may be generated by the L-system with rewrite rules: $L \rightarrow +RF - LFL - FR+$, $R \rightarrow -LF + RFR + FL-$; where "F" means move forward one unit, and "+" and "-" indicate opposite 90 degree rotations of the direction of subsequent movement. Note that Hilbert paths require the image dimensions to be equal and a power of two, as they are defined recursively on a quad grid. We accommodate other dimensions by simply using the smallest Hilbert curve which covers the entire output, and skipping locations on the curve outside of the output image.

Quality-wise, scanline ordering gives the worst results, and Hilbert ordering gives the

best. The reason for this is clear from examining the produced patch layouts (Figure 5). In particular, scanline ordering encourages the formation of patches which extend diagonally from top-left to bottom-right. This is in fact to be expected based on our per-pixel processing. With scanline ordering, already-synthesized neighbours appear only above or to the left of the current output pixel. This makes it extremely unlikely that a patch boundary would form along the alternate direction (from bottom-left to top-right), since it would require the same jump decision to be made *late* in one scanline, and *earlier* in the next. However, patch formation is based on extending existing patches, not on independent pixels jumping to the same area of the sample texture. Thus, it should come as no surprise that following a Hilbert path, which changes its direction on almost every pixel, leads to the highest quality results. Such frequent direction changes allow a very balanced set of available neighbours, which in turn allows patches to be extended in any direction, effectively. Also not surprisingly, serpentine paths lead to patches relatively balanced horizontally, but which primarily extend downward.

Previously [Zelinka and Garland 2002], we noted a higher cost for following a Hilbert path over simple iteration along scanline or serpentine paths: synthesis time was about three times slower. This turned out to be due to a particularly inefficient (but stateless) Hilbert path calculation. We subsequently implemented a stateful, recursive iterator for the Hilbert traversal, which all but removes the speed penalty for using the Hilbert-based pixel ordering, being about 10% slower than that of the simpler scanline or serpentine iterators. Thus, we now use the Hilbert path-based pixel ordering for all image synthesis tasks, as the slight loss in speed is more than offset by the higher quality of results.

5.4 Image Synthesis Results

Figure 6 shows a number of results on a variety of textures for the image synthesis algorithm we have presented. Output images are 256×256, and each was generated in about 0.060 seconds[1] on an Athlon 1800+ PC. Note that the speed of the synthesis phase of our algorithm is independent of the texture being synthesized, and depends only on the output size.

Quality-wise, our results are clearly best on relatively stochastic or "natural" textures, but as can be seen from Figure 6, are often quite acceptable on more structured textures. Compared to Ashikhmin's algorithm [2001], which our algorithm can be seen as approximating, the results are quite favourable (see Figure 7). The quality is roughly similar for most textures, indicating our approximation is quite good, but our approach is two orders of magnitude faster. On the other hand, there does not appear to be an effective way to direct synthesis to produce something resembling a particular target image, as is possible with Ashikhmin's algorithm. Figure 8 compares our results with those of Graphcut Textures [Kwatra et al. 2003]. While our algorithm performs worse on the structured texture, it performs comparably well on the more natural texture, and produces results over two orders of magnitude faster. Boundary mismatches are less of a problem with Graphcut Textures, though as it is difficult to inject randomness into their results, they can suffer from a high level of repetition; if a good but repeatable seam is found by their algorithm, it will use it continually. Note, however, that Graphcut Textures generalize easily to higher

[1]This corresponds to about 1.1 million pixels per second, about half the speed reported in our previous paper [Zelinka and Garland 2002]. The difference is due to image-domain optimizations (integer arithmetic, known neighbour layouts) we have since removed so the same code works for synthesizing texture on surfaces as well.

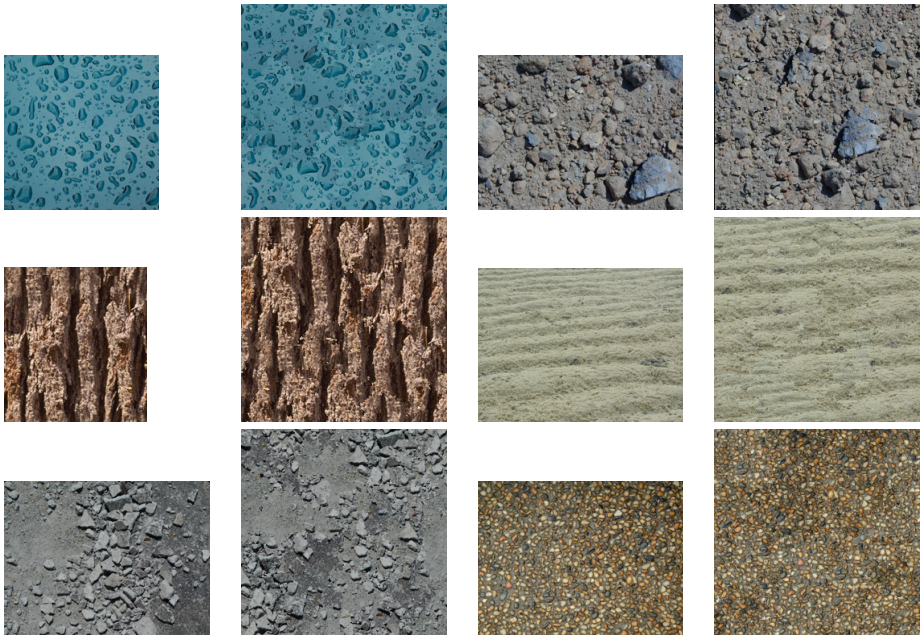


Fig. 6. Image synthesis results. For each pair of images, the sample is on the left, and the synthesized result on the right.

dimensions (such as video synthesis), while jump maps may not be feasible in such higher dimensions. Finally, we show direct comparisons with the Chaos Mosaic in Figure 9. We did not have access to very natural or stochastic examples of Chaos Mosaic results, but strong vertical and horizontal artifacts can be seen in the textures shown, which would tend to be more objectionable on natural or stochastic textures. In contrast, artifacts from our algorithm tend to be texture mismatches on a more fine-grained level, often perceived simply as noise in the image. Chaos Mosaic produces results as fast if not faster than jump maps, however, and allows procedural evaluation of the output, which is not possible with our algorithm due to its inherent serial nature.

6. TEXTURE SYNTHESIS ON SURFACES

We now turn to the related problem of generating textures directly on surfaces. It is interesting to note that with images, in the absence of a texture synthesis algorithm, one could always simply tile the input to produce more of it (possibly reflecting and mirroring it if necessary to create a tileable texture). While this often produces very objectionable, repetitive artifacts, it is an easy, viable option, especially for regular patterns or structures. With surfaces, however, tiling is not so simple. In order to be able to tile a texture over the surface, some sort of parameterization is needed, which can be very expensive and complex to compute.

In the next section, we note some important difficulties in specifying the problem of texture synthesis on surfaces, and present our algorithm for texture synthesis on surfaces in the following sections. Our approach is to simply draw an equivalence between pixels

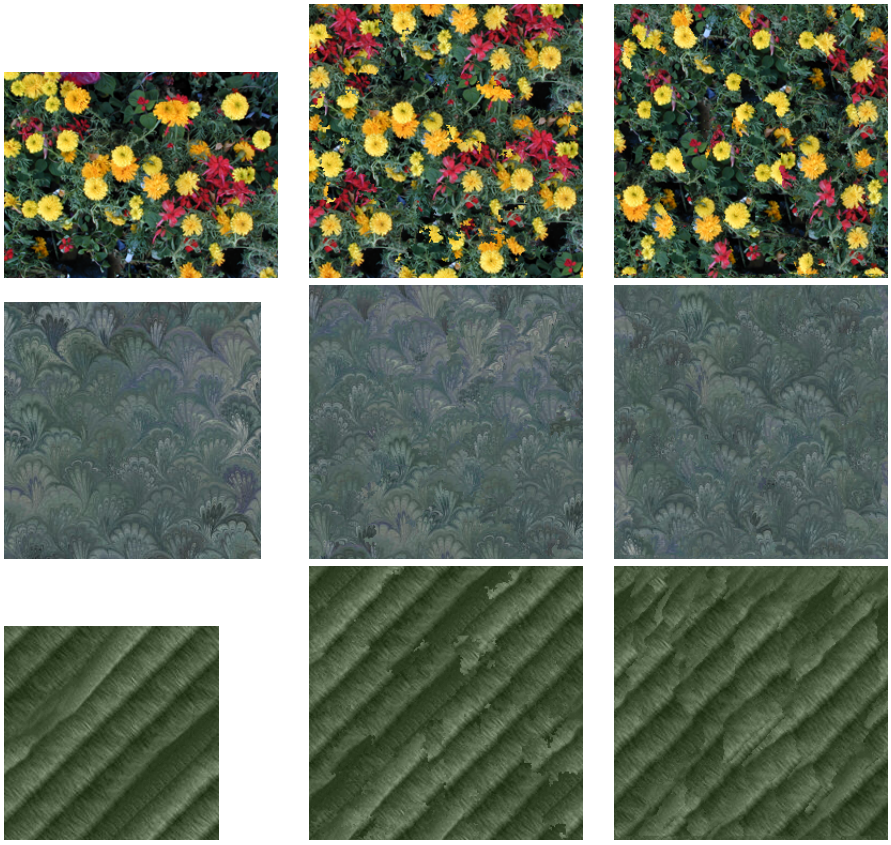


Fig. 7. Comparison with coherent synthesis. Left to right in each row: sample image; image synthesized using our algorithm; image synthesized using Ashikhmin's algorithm. The quality of our results is quite comparable for most textures.

of an image and vertices of a surface. We thus perform a single pass over the vertices of a mesh, processing each vertex iteratively to assign its texture coordinates. Drawing this equivalence poses a number of problems. First, surface topologies are much more complex than that of images, so we must be careful in choosing a vertex ordering appropriate for a surface (§6.2). Further, vertices generally have non-uniform spacing between them, so we must define texture space distances across the surface, being careful to respect the user-provided information (§6.3). Since these inter-vertex distances are invariably no longer integer offsets, in Section 6.4 we define a method to perform floating point jump map lookups. Also, anytime a jump is performed between two vertices, triangles incident to both vertices will be distorted in texture space. The final step of our approach is therefore to assign texture coordinates to each triangle's corners, as detailed in Section 6.5. In dealing with this, we encounter the possibility of invalid jumps, and thus revise our synthesis procedure slightly. This revision, as discussed in Section 6.6, does not change the behaviour of the algorithm, but does make it clear that our method effectively performs a kind of probabilistic matching.

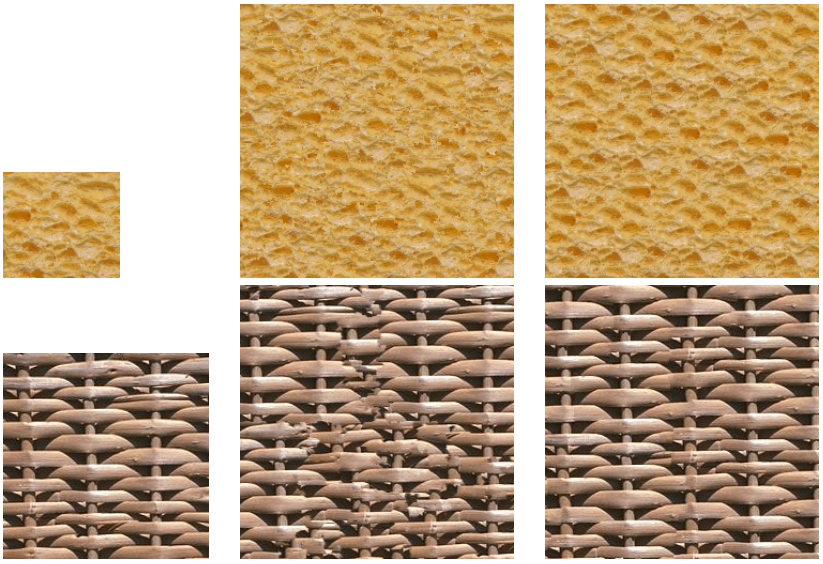


Fig. 8. Comparison with GraphCut synthesis. Left to right in each row: sample image; image synthesized using our algorithm; image synthesized using GraphCut textures. While GraphCut textures are better for ordered textures, our method produces results in a fraction of the time.

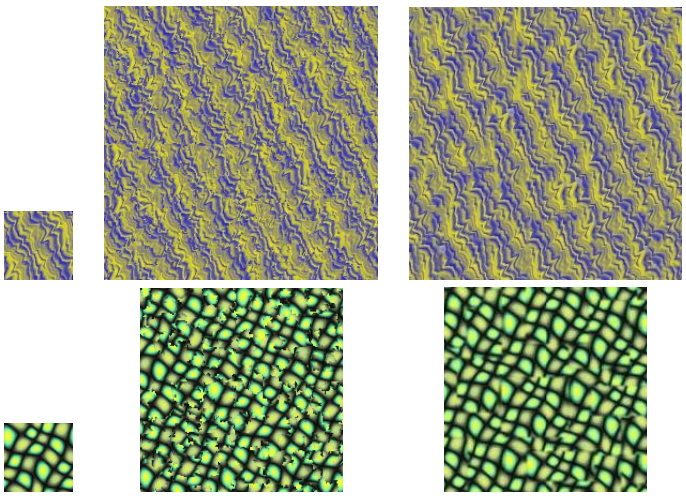


Fig. 9. Comparison with Chaos Mosaic synthesis. Left to right in each row: sample image; image synthesized using our algorithm; image synthesized using the Chaos Mosaic. The Chaos Mosaic generally produces sharp boundary artifacts, while our method produces noiser boundaries. The former is good for structured textures, while the latter is less noticeable in stochastic or natural textures.

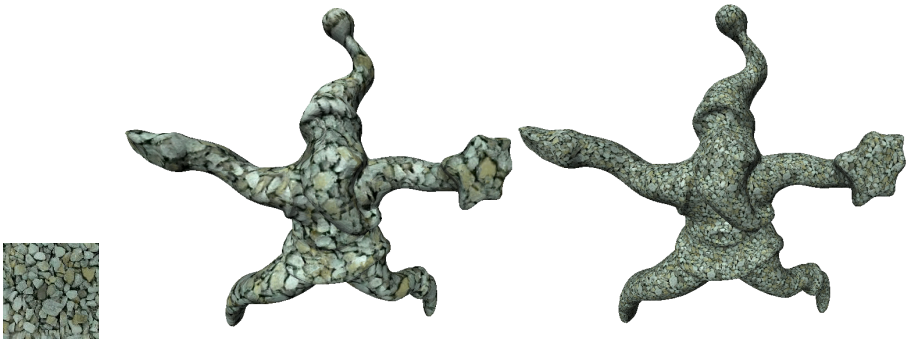


Fig. 10. Using the sample on the left, the same mesh is textured twice using two very different scales for the texture.

One key difference between synthesis on surfaces and synthesis of images is the *output* of the process. With image synthesis, we simply output the final colour values at each position in the image; on surfaces, we instead output texture coordinates, in order to permit easy hardware-accelerated texture mapping and filtering. Note that this is actually easier than outputting colours in our case, since our method is driven by the texture coordinates assigned at each position, not their particular colours (recall that we only examine the texture's colours during the analysis phase).

6.1 Specifying Textures on Surfaces

In terms of user effort, texture synthesis on surfaces is a far more difficult problem to properly specify than simple image synthesis. In particular, the scale and rotation of the texture on the surface must be specified by the user. With images, both of these parameters are usually implicit. One pixel of an output image should contain as much texture detail as one pixel of the input image, and directions in texture space map directly onto each other (e.g., "up" in the output image is the same direction as "up" in the input image). In contrast, one cannot automatically assign an object-space distance on a surface to be covered by one pixel, nor which direction on a surface should correspond to "up" in texture space. These parameters are inherent user input to the problem of synthesizing textures directly on 3D surfaces, and correspondingly complicate the user interface of any system solving this problem. Note that one may, of course, make heuristic guesses for these parameters [Wei and Levoy 2001; Turk 2001], especially for isotropic textures or textures exhibiting various symmetries, but direct user control is required in the general case.

Specifying the scale of the texture on the surface is generally quite easy; we simply show a tile next to (or even mapped on) the surface, and allow the user to rescale the tile until the desired scale is achieved. We can then simply infer the texture scale in terms of number of pixels per world space unit. An example of the range of variation easily accomodated by our algorithm is shown in Figure 10. It is important to note that, unlike vertex colouring methods [Wei and Levoy 2001; Turk 2001], no remeshing is required to achieve this range of variation, and unlike atlas-based methods [Ying et al. 2001], each result uses the same amount of texture memory (simply that required to hold the sample texture). This scale independence and low memory usage is a direct consequence of the generation of texture

Fig. 11. Orienting textures on surfaces. Left to right: orientation field visualized with blue vectors pointing in the texture space "up" direction; sample texture; textured results with given orientation field; textured results with a global 90^o rotation of the orientation field.

coordinates rather than vertex colours or image data, as noted earlier by Soler *et al.* [2002].

Specifying the texture rotation is much more difficult, due to the fact that most surfaces are not developable, and thus must necessarily have some distortion introduced when mapped to the plane. We follow what has become the standard approach for this problem, and have the user specify a vector field over the surface (in our case, defined at vertices). The *orientation* vector at each vertex denotes the "up" direction in texture space. The problem of specifying the rotation then becomes primarily a user-interface issue, typically handled by having the user specify a few anchor points with known orientation, and interpolating these anchors across the mesh.

We have found it quite useful to allow the specification of singularity points in the vector field (sinks and sources), especially as the genus of the surface increases (higher genus surfaces inherently require more singularities). For example, the orientation field for the rabbit model shown in Figure 11 was created by specifying only one source at its nose and one sink at its tail. The different results in this Figure were produced by globally rotating the vector field interactively. More sophisticated manipulations, such as locally grabbing and twisting the field with a distance-based fall-off, are certainly possible with our method.

6.2 Vertex Orderings

The first issue we must deal with is the order in which vertices are processed on a surface. Since following a Hilbert path gives good results for images, it would seem natural that some sort of analogue to Hilbert paths for surfaces would give good results as well. Computing such an analogue is a fairly difficult problem for arbitrarily connected graphs, adequately solved by Bagomjakov and Gotsman [2002] in the context of attempting to maximize locality of vertex access for a mesh to take full advantage of vertex caching graphics hardware. Their *Universal Rendering Sequences* (URS) take a few seconds to minutes to compute for typical meshes, and behave quite well for our purposes. Note that computing a URS for a surface has the added benefit of improving the rendering performance on vertex-caching graphics hardware, simply by reordering the faces of the mesh.

However, the compute time for Universal Rendering Sequences may be prohibitive for interactive applications or dynamically-generated geometry, and we have therefore developed a much faster approach which works just as well for most surfaces. This approach, which we refer to as the *emerging front-based ordering*, is based on the idea that each available already-synthesized neighbour can be seen as placing additional constraints on

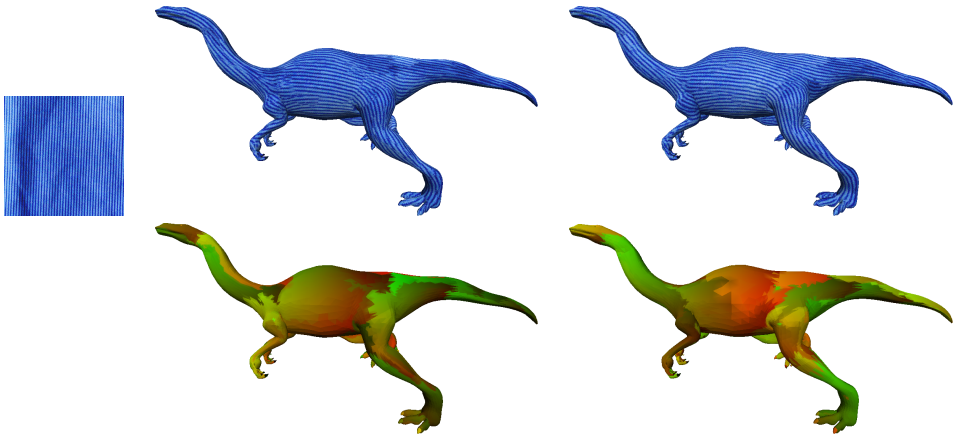


Fig. 12. Comparison of vertex orderings. Left: Sample texture used. Middle: Emerging front-based ordering. Right: Universal Rendering Sequence-based result. The top row shows the textured result, while the bottom row shows the corresponding patch layouts by texturing using a colour ramp. Both orderings give comparable results, and differ primarily in the precomputation time required. Note that unlike a texturing approach based on parameterizing the surface, there is no noticeable distortion of the texture even in regions of high curvature.

the texture positions available to a new vertex (i.e., it has to match well with all of them). However, in the end, a vertex has to match well with *all* of its neighbours. Thus, the more information we know about what those neighbours actually are, the better equipped we are to make a good match. Thus, we wish to maximize the amount of already-synthesized area in the neighbourhood of each vertex. We choose a simple greedy approach, randomly choosing the first vertex, and iteratively choosing the next vertex to be the vertex with the highest proportion of already-synthesized neighbours. This ordering depends solely on the surface connectivity, and is extremely fast to compute.

We observe both orderings to be quite comparable in quality, with typical results shown in Figure 12. Note that regardless of which ordering is used, it may be precomputed on a per-mesh basis. Thus, the emerging front-based ordering is best for applications which do not have significant per-mesh precomputation time available. Applications for which the higher precomputation time is tolerable may instead use a URS-based ordering, and receive the benefit of (potentially) faster rendering times on vertex-caching graphics hardware.

6.3 Texture Space Distances on Surfaces

A key component of our image-based synthesis algorithm is extending a patch of texture from one pixel to the next. This is straightforward since there is a natural mapping between pixel offsets in the output image and pixel offsets in the input image: they are the same (extending one pixel to the right in the output may be done by simply moving one pixel to the right in the input). With surfaces, there is no such natural mapping. When we move some distance in world space across an edge of the mesh, we must define how far we have moved in texture space. Locally, we would like texture space to correspond to the tangent plane, while avoiding unnecessary distortion. Thus, our goal is to define basis vectors for the tangent plane for each edge of the mesh. Given these basis vectors, we can then project each edge into its tangent plane, and measure the 2D distance between the edge's endpoints

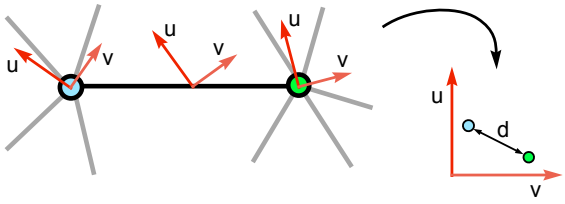


Fig. 13. Edge offset calculation. The orientation vector (u) and tangent vector (v) at each vertex span the vertex's tangent plane. The endpoint basis vectors are averaged to form the edge's tangent plane basis vectors. The edge is projected into the plane, and scaled according to the desired texture scale, determining the offset made in texture space for traversing the edge (d).

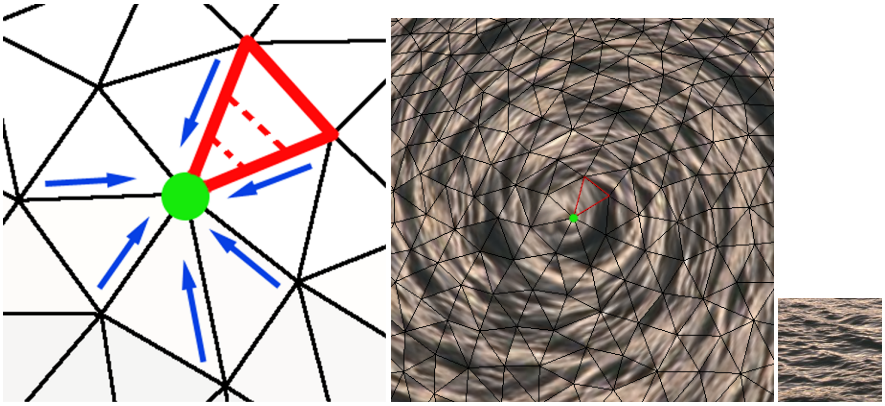
to determine the distance in texture space traversed by crossing the edge.

To form a basis for the edge's tangent plane, we interpolate and orthogonalize basis vectors defined at each of its endpoints. At each vertex, then, we must form a basis for its tangent plane which respects the user parameters. These user parameters consist of a local orientation vector at each vertex, and a global scale factor. We use the component of the orientation vector which is orthogonal to the vertex normal as one of the basis vectors, $\mathbf{v}$, and take the cross product of $\mathbf{v}$ and the vertex normal to form the other basis vector, $\mathbf{u}$.

For a given edge e with endpoints a and b, we define $\mathbf{v}_e$ as the average of $\mathbf{v}_a$ and $\mathbf{v}_b$ (the basis vectors at a and b), and $\mathbf{u}_e$ as the component of the average of $\mathbf{u}_a$ and $\mathbf{u}_b$ which is perpendicular to $\mathbf{v}_e$. We then scale $\mathbf{u}_e$ and $\mathbf{v}_e$ so their 3D length corresponds to one pixel of the texture, according to the user-specified scale. Now, given 3D positions of a and b as $\mathbf{p}_a$ and $\mathbf{p}_b$, we can project the edge onto our tangent space basis vectors to get projected endpoints $\mathbf{q}_a = (\mathbf{p}_a \cdot \mathbf{u}_e, \mathbf{p}_a \cdot \mathbf{v}_e)$ and $\mathbf{q}_b = (\mathbf{p}_b \cdot \mathbf{u}_e, \mathbf{p}_b \cdot \mathbf{v}_e)$. The offset from a to b in texture space is then simply $(\mathbf{q}_b - \mathbf{q}_a)$. This process is illustrated in Figure 13.

There are a few facts to note about our method. For a given triangle, the distance along one edge is not nessarily equivalent to the sum of the distances along the other two edges. This will naturally introduce distortion to the texture mapping, but this is generally a *desired* distortion implied by the user-supplied orientation field. For example, consider a triangle incident to a singularity in the orientation field, as shown in Figure 14. The horizontal distance in texture space between the ends of the dashed lines must be roughly equal, according to the orientation field. While this results in triangles which are clearly distorted in texture space, the final texture on the surface looks as it should for the given orientation field. What is key to this result is that the distance along a particular edge is the same for both triangles incident to that edge. Thus, even though the texture within each triangle is distorted in various ways, the texture still matches along the common edge and appears smooth over the surface.

Note that there is no particular reason that the assigned scale must be a global parameter. Indeed, as shown in Figure 15, it is trivial to vary the scale of the texture over the surface. Here, we simply vary the scale used on each edge according to the object-space heights of its endpoints. Note that we are effectively introducing even more distortion into the mapping of each triangle into texture space, but because the mapping is on a per-edge basis, and similar for each triangle incident to an edge, the result still appears smooth. Of course, more complicated schemes are possible, and we in fact allow scale to vary on a per-vertex basis. As long as the scale field is relatively smooth, the results are seamless



(a) (b) (c)

Fig. 14. User-induced distortion near a singularity in the orientation field. (a) At a singularity (green) in the orientation field (blue), triangles must be distorted in texture space. For example, the dashed lines in the red triangle should have equal lengths in texture space, according to the orientation field. (b) Textured result for the singularity shown in (a), and surrounding region on the mesh. (c) Sample texture used in (b).

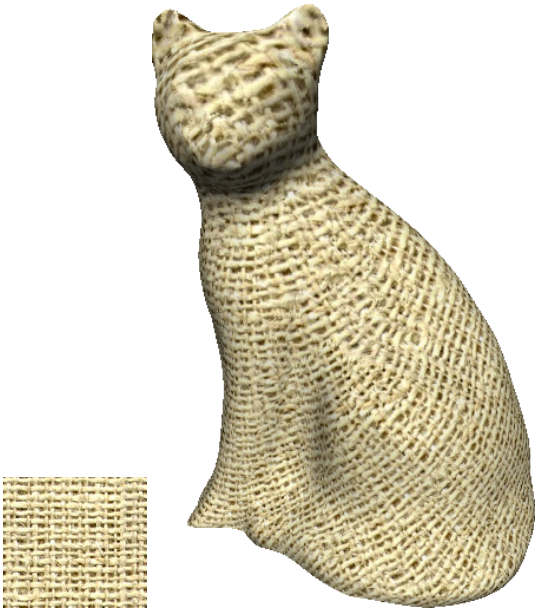


Fig. 15. Progressively varying the scale of texture across a surface.

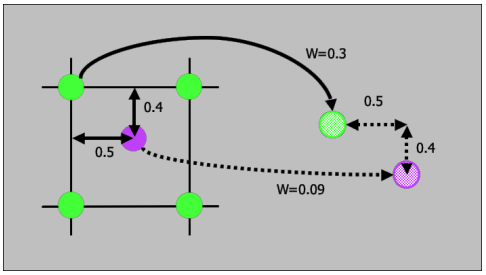


Fig. 16. Floating point jump map lookups. Jumps at neighbouring pixels are appropriately bilinearly weighted and offset. Here, a jump at the upper left neighbour is offset and reweighted according to the purple address's subpixel offset to form one of the jumps returned in the floating point jump map lookup.

and quite natural-looking.

It is worth noting that our first approach to assigning texture space distances to mesh edges was based on conformal maps, locally flattening each vertex's neighbourhood into 2D while preserving angles and lengths. The key problem with this approach was that the distance in texture space traversed along a given edge varied depending on which way the edge was traversed, causing straight lines in the texture to be mapped to broken or wavy lines on the surface. This occurred as a result of the conformal maps at each vertex being improperly aligned, which would have needed a global optimization procedure to fix. An alternative would be to flatten larger areas at a time, as previous researchers have done [Praun et al. 2000; Wei and Levoy 2001; Turk 2001; Ying et al. 2001]. Our projection scheme detailed above, however, solves the problem very simply, using only local operations, at a much lower cost.

6.4 Floating Point Jump Map Lookups

A natural consequence of our method for determining texture space distances over surfaces is that these distances invariably require floating point precision. In particular, this means that the texture coordinates assigned to a vertex must be floating point addresses. This introduces a new problem: jump lists are only recorded at integer positions within the jump map. Thus, we must define a filtering procedure for performing jump map lookups at floating point addresses.

Our approach is to effectively define a bilinear filtering operation for jump maps. This process is outlined for one particular jump in Figure 16. We simply coalesce the jump lists at each of the four neighbouring integer addresses, and appropriately modify them: the weights are scaled bilinearly according to the subpixel offset of the floating point address, and the destinations are offset by the subpixel address. This adequately controls the complexity of the algorithm while allowing us to use the same jump maps for synthesizing textures on surfaces as we used for synthesizing texture images.

6.5 Corner Texture Coordinates

One undesirable form of distortion produced by our algorithm so far occurs on any edge over which a jump was taken. Here, as shown in Figure 17, the triangles incident to the edge become distorted in texture space, since their vertices were given texture coordinates from different regions of the texture. Our solution to this is to assign texture coordinates

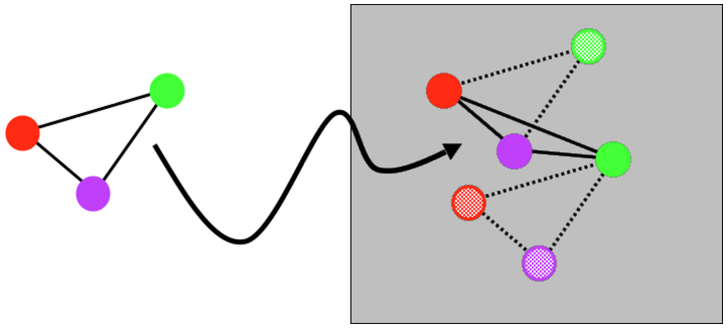


Fig. 17. A triangle on the surface (left) is distorted in texture space (right) whenever a jump was taken between any of its vertices. Selecting one vertex to use as a base vertex, and directly using the edge offsets to map the other corners of the triangle (dashed lines) removes the undesirable distortion.

to each corner of each triangle. This ensures that any particular triangle will have consistent texture coordinates, while respecting the vertex texture coordinates assigned by our algorithm. For each triangle, we first choose a *base vertex* for the triangle; the corner corresponding to the base vertex simply inherits the base vertex's texture coordinates. The other corners of the triangle, however, are assigned texture coordinates from the base vertex, simply adding the corresponding texture space offsets along the edges of the triangle. Thus, the texture assigned to a particular triangle is distorted only according to the distortion implied by the edge offsets, which as we showed in Section 6.3 is a desirable form of distortion.

The remaining question is how to choose a base vertex for each triangle. We have not examined this question deeply, but one can imagine using a perceptual measure for ranking vertices. For example, vertices that are part of high-frequency areas are more likely to have texture discontinuities masked than vertices which are part of smoother areas. Instead, we typically assume the availability of texture blending facilities. For each of the three possible choices of base vertex, we get a slightly different textured result (while these results may come from different areas of the input texture, their content is usually very similar). We then blend these results together, using simple linear alpha ramps (alpha of one at the base vertex, and zero at the other vertices) to get the final results. While this may introduce some blending artifacts for some textures (for example, a checkerboard), this is generally outweighed by the decreased visibility of any texture mismatches.

In doing this, we are effectively using each vertex of a triangle as the base vertex simultaneously. This implies an additional constraint on our synthesis procedure: each triangle incident to a vertex must be texturable from that vertex. In other words, the "flattened" neighbourhood of the vertex must remain within the texture when centered at the vertex's texture coordinates (see Figure 18). Note, however, that this is not a typical neighbourhood flattening operation [Praun et al. 2000; Wei and Levoy 2000], but simply adding, in turn, the texture space offset of each incident edge to the proposed texture coordinates for the vertex, and ensuring the result stays within the texture. Thus, to fulfill this constraint, we can simply construct a 2D bounding box for these incident edge offsets, and test against two opposing corners.

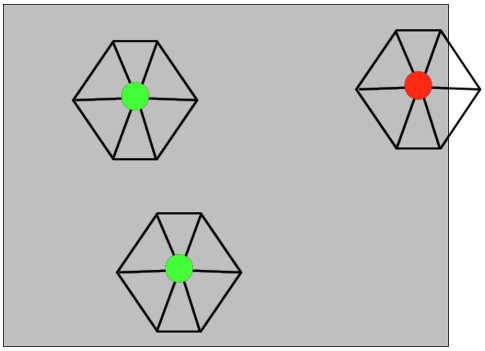


Fig. 18. Invalid texture coordinates. The "flattened" neighbourhood of a vertex must remain within the texture in order to allow texture blending to occur. Thus, the green positions are valid positions for a vertex with the given set of incident edges, but the red position is not; the two triangles on the right could not be assigned texture coordinates when using the middle vertex as a base vertex.

6.6 Probabilistic Matching

With images, we knew *a priori* the set of pixel offsets that would be used during the synthesis phase. Thus, we could easily ensure at jump map construction time that every destination we stored in the jump map would be valid *given those offsets*. On surfaces, however, we no longer know the offsets that shall be used, as they depend on the mesh topology and the user's scale and orientation parameters. Thus, there is now the possibility that a jump listed in a jump list is not a valid destination for a particular vertex. We must therefore explicitly test destinations for validity according to the criteria just outlined in the previous section.

A potential problem here is that all of the destinations listed at a particular neighbour may be invalid. Instead of attempting a backtracking algorithm which would test alternate neighbours, we build a large selection table consisting of all of the valid entries from *all* available already-synthesized neighbours. Thus, only if there are no valid destinations at all (from any neighbour) are we forced into a purely random assignment for a vertex. This simplifies the algorithm considerably, since the same processing is performed for all vertices, for all neighbours: at each neighbour, we add all of its suggested valid destinations to the selection table; then, when all neighbours have been processed, we select a destination from the selection table with a single random number as before. This processing emphasizes the fact that our approach is a probabilistic algorithm for finding good matches. In particular, if multiple neighbours suggest the same destination, that destination is *likely* to be a good match, since it matches well over a space of neighbouring directions. Reflecting this, that destination's probability within the larger selection table is in fact higher relative to other destinations. Conversely, outlier destinations suggested by only a single neighbour, which are thus unlikely to be good matches, have their probability lowered relative to other more popular destinations. This reasoning gives further support to the ideas behind our emerging front-based vertex ordering: the more neighbours which are available when we process a vertex, the more advantage we get from this probabilistic matching.

Note that in the situation that all jumps suggested by all neighbours are valid, as is always the case in the image synthesis algorithm, building a large selection table and choos-

ing from it is equivalent, probabilistically, to choosing a neighbour at random, and then choosing a destination from those suggested by the chosen neighbour. Thus, our image-based texture synthesis algorithm performs the same kind of probabilistic matching that the surface algorithm does. We simply organize the computation differently in the image domain to avoid the overhead of building the selection table at each pixel, taking advantage of the regularity of the output.

6.7 Texture Synthesis Results on Surfaces

We show some typical texture synthesis results on surfaces for a variety of surfaces and textures in Figure 19. Note that even in regions of high curvature, there is no apparent distortion of the textures beyond that implied by the user's orientation field. The models used ranged from 20,000 to 40,000 vertices, and were textured at a rate of about 200,000 vertices per second on an Athlon 1800+ PC. This is easily fast enough to allow intuitive, interactive specification of the orientation and scale parameters on these models. Note that the speed of this algorithm depends linearly on the number of vertices in the model, and is independent of both the texture and surface used. All of the results shown use the emerging front-based vertex ordering and texture blending.

As with the image-based algorithm, the quality of our results is clearly best on stochastic textures, and can be worse than other algorithms on ordered textures. However, one interesting effect is that texturing directly on surfaces tends to allow a greater degree of inhomogeneity in the sample textures. In particular, both the water drops texture on the winged dragon in Figure 19, and the lined shirt texture used in Figure 12, have slight colour shifts across the sample images. While difficult to perceive normally, these colour shifts become quite apparent when pixels from different areas of the sample images are placed next to one another. Because of this, these textures pose a significant challenge to image-based texture synthesis algorithms. When mapped directly on surfaces, however, the colour shifts often simply give a false impression of a variation in shading, implying additional geometric detail on the surface. Also note that artifacts produced on the wicker texture (Figure 8) are much less noticeable and objectionable when synthesis occurs directly on a surface (Figure 11).

We present some comparative results in Figures 20, 21, and 22. Note that these comparisons are not exact, in that the meshes, textures, and texturing parameters (scale, orientations) are not identical. The comparative results for the other methods were taken from the respective paper for each method, and are thus simply images for which we have attempted to visually match the data used as closely as possible. Also, note that the random element to all of these methods means results may vary significantly from run to run.

Typically, as shown in Figures 20 and 21, our results are slightly worse in quality than the vertex colouring methods [Wei and Levoy 2001; Turk 2001]. However, we obtain results over three orders of magnitude faster than these techniques, and may adjust the scale of the texture significantly without remeshing. Artifacts produced by their algorithms include slight texture distortion in regions of high curvature (due to the need to locally parameterize the surface for resampling), and the possibility of falling into a bad part of the texture search space and synthesizing "garbage" (a well-known phenomenon with non-parameteric synthesis techniques [Wei and Levoy 2000]). In contrast, artifacts with our method are primarily texture discontinuities. With stochastic textures, these are generally unnoticeable, but become very distracting with ordered textures. Since our technique exclusively copies patches of texture from the sample image, there is no possibility of syn-

Fig. 19. Surfaces textured with our algorithm.

thesizing garbage, and as discussed earlier, our results are distorted only where required by the orientation and scale fields.

Lapped Textures [Praun et al. 2000], a generalization of the Chaos Mosaic [Xu et al. 2000] to surfaces, produces results equal or lower in quality to jump map synthesis. Since Lapped Textures are produced by repeatedly pasting a single patch over the surface, they typically suffer from noticeable repetition. As they locally parameterize the surface as well,

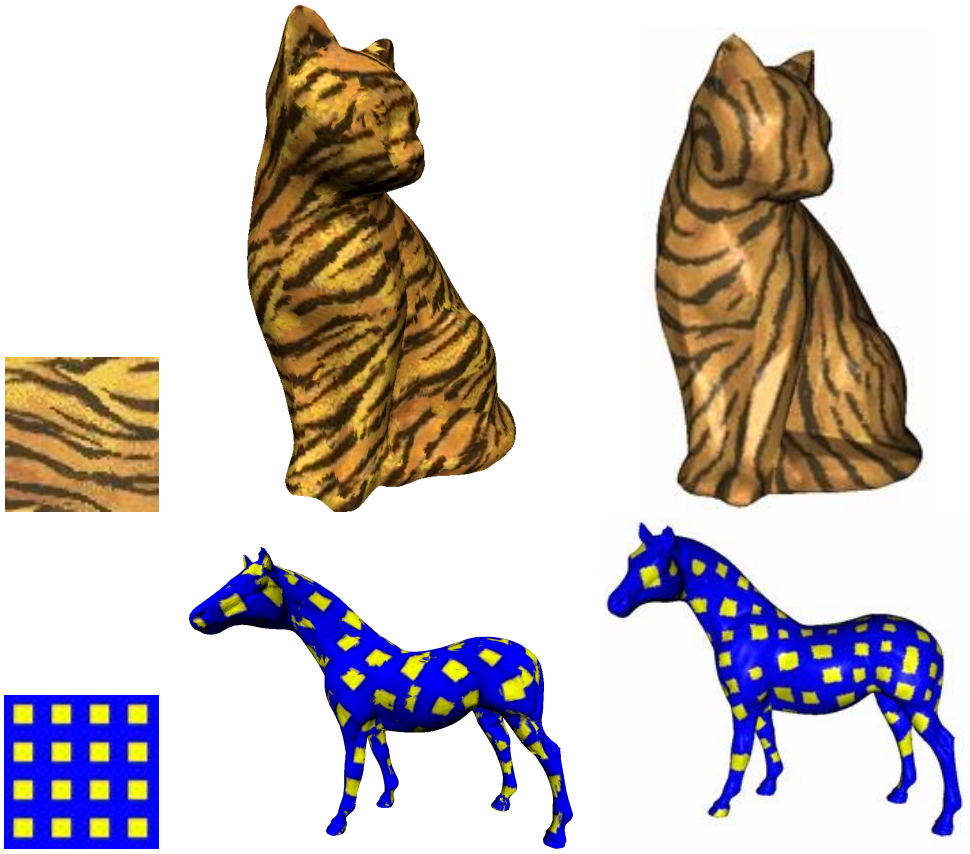


Fig. 20. Comparison with Wei and Levoy's algorithm. Left to right: sample, jump map result, Wei and Levoy's result.

there is also some unintended distortion. Since the patch is overlapped arbitrarily on the surface, there are typically texture discontinuities as in our method, alleviated somewhat with texture blending. While much faster than the vertex colouring methods, generating Lapped Textures remains over two orders of magnitude slower than our approach: they require 20 seconds to 6 minutes to texture models averaging 5000 faces; with our method, comparable models on comparable hardware require 0.05 seconds to texture.

7. EXTENDING IMAGE SYNTHESIS

In this section, we explore some powerful extensions to image synthesis brought about by our development of texture synthesis on surfaces. First, we detail a patch-based image synthesis algorithm which outperforms the pixel-based algorithm while delivering higher quality results on more structured textures. Following that, we show how the user inputs to surface-based texture synthesis can be applied to images as well, yielding interesting new effects purely within the image domain.

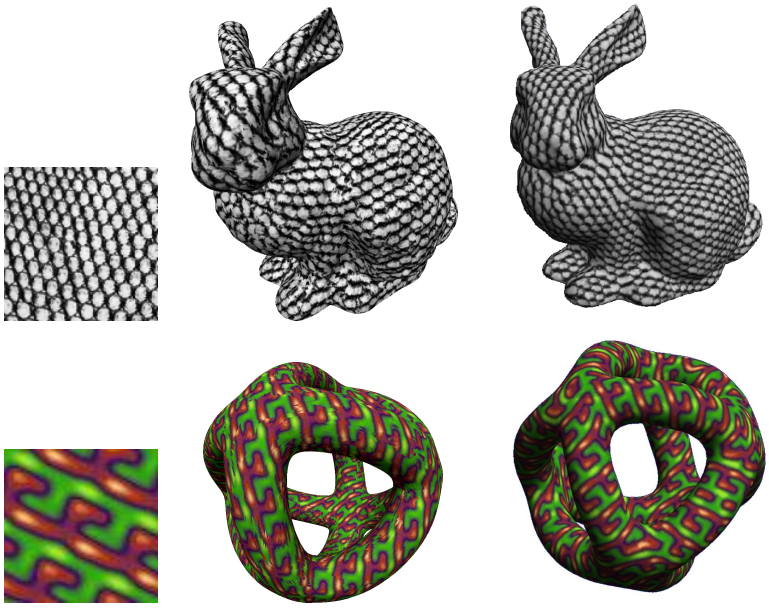


Fig. 21. Comparison with Turk's algorithm. Left to right: sample, jump map result, Turk's result.

Fig. 22. Comparison with Lapped Textures. Left to right: sample, jump map result, Lapped Textures result.

7.1 Patch-based Image Synthesis

Our algorithm for texturing surfaces from an example texture leads directly to a novel image synthesis algorithm which explicitly uses patches, giving improved results for more structured textures. In essence, our surface-based algorithm has shown that jump maps remain effective even when the offsets used between neighbours are relatively large. Thus, the simple specialization to images is to make a jump decision relatively infrequently, only once per patch, rather than once per pixel, and to consider all the available neighbouring pixels when making the jump decision.

There are many possible variations on this approach; we describe just one with which we have observed higher quality results on ordered textures. We use regular patches of a size decided by the user, but which should be large enough to capture the largest texture features, and perhaps a degree of their interaction (e.g., not just leaves, but patterns of leaves). We typically use patches on the order of one-fifth to one-half of the sample image size. We then simply synthesize each patch in order through the output image, following a Hilbert path-based ordering of patches (rather than pixels). At each new patch, we build a selection table from which the texture coordinates of the middle of the patch are assigned. In building this table, we use each neighbouring already-synthesized *pixel* of the patch as a potential source for the patch's address, adding only their *valid* destinations. In this case, valid destinations are those from which the entire patch may be textured.

This new patch-based algorithm seems to perform better than the pixel-based algorithm for many ordered textures (see Figure 23), since patches are more likely to capture the high level structures well. Synthesis time is only about two to three times faster, for while far fewer decisions need to be made in the course of synthesis, each decision involves much more work in building the selection table. For more natural or stochastic textures, the pixel-based algorithm produces higher-quality results, as the boundaries it produces are more random and thus less noticeable for these textures.

7.2 Controlling Image Synthesis

For specifying how a texture was to be applied to a surface, the user was responsible for two parameters: the apparent scale of the texture on the surface, and the orientation of the texture with respect to the surface. However, it is quite easy to allow the user to specify these same parameters when synthesizing *images* as well. In effect, we are treating the output image as a quad mesh, and simply directly applying our surface-based texture synthesis algorithm to this quad mesh. In practice, we extend the image-based algorithm to handle varying distances between pixels and floating point addresses, so that we can take advantage of the regularity of the output.

Thus, the user may specify an orientation field over the output image, as well as a scale field. We demonstrate a couple of simple examples in Figure 24. First, we control the scale of the output across the image, increasing the distance between pixels towards the top of the image in order to simulate perspective. In the second example, we generate an orientation field where every pixel's orientation points toward the center of the image, and increase inter-pixel distances near the center of the image as well. Note that proper filtering is required for these results; a particular output pixel must be a filtered combination of all of the input pixels mapped into it.

This kind of application is not possible with most alternative techniques. Purely image domain methods would require local resampling of the output, as well as searching in

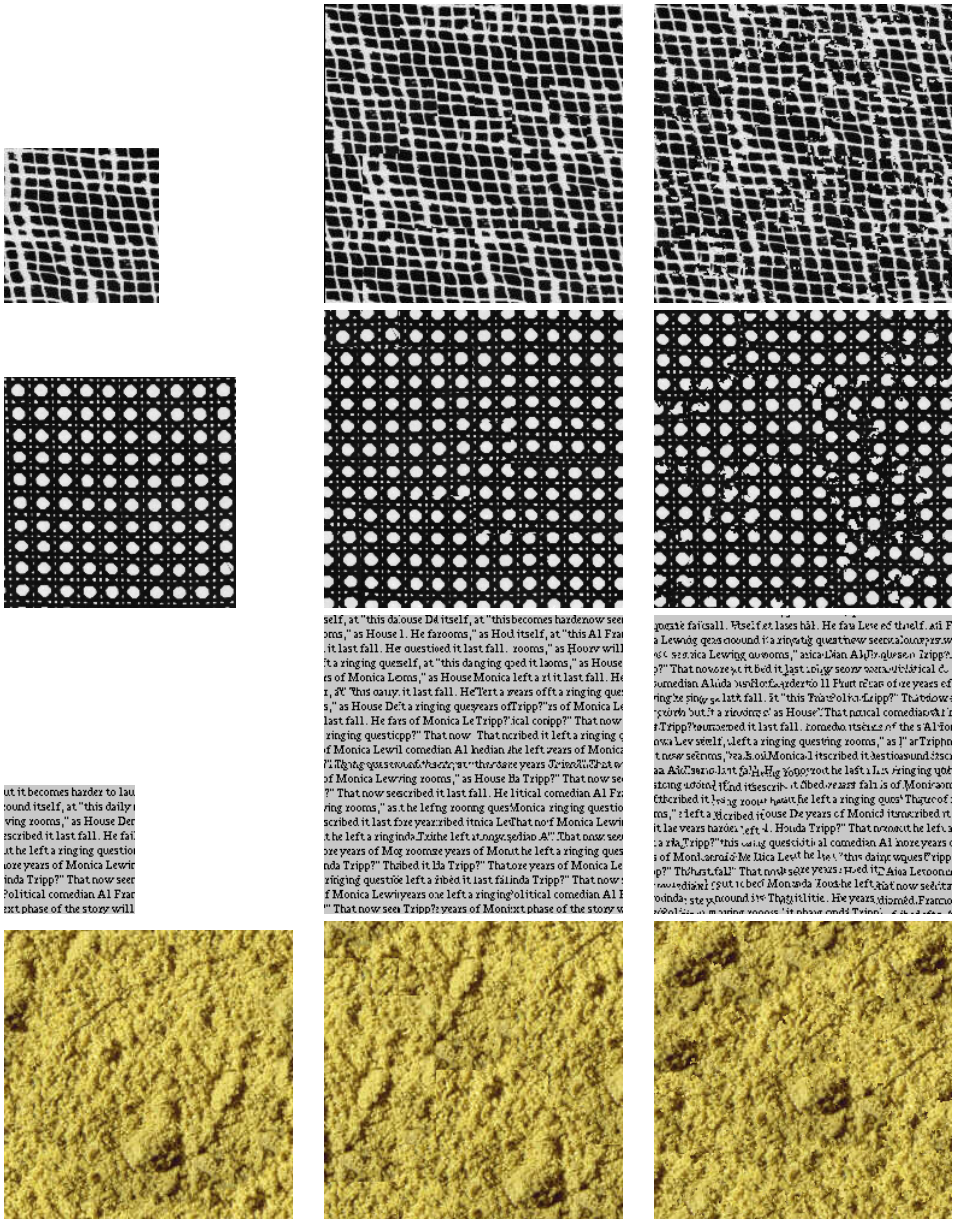


Fig. 23. Patch-based image synthesis results. Left to right in each column: sample, patch-based jump map synthesis results, pixel-based jump map synthesis results. The patch-based approach handles ordered textures better, but can cause noticeable artifacts in stochastic textures.

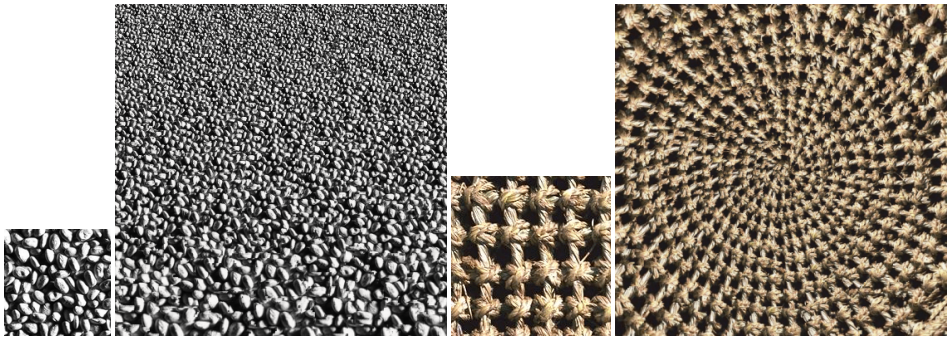


Fig. 24. Controlling Image Synthesis. Left to right: sample image; simulated perspective by increasing the inter-pixel distances towards the top of the image; sample image; applying an orientation field to the image where each pixel's up direction points toward the image center, and inter-pixel distances increase toward the center.

arbitrary rotations of the input to reproduce the desired orientations. In the surface domain, vertex colouring techniques [Wei and Levoy 2001; Turk 2001; Zhang et al. 2003] would generally require severe remeshing of the output image in order to capture the detail of the distortion properly. Thus, these already slow methods would take a proportionately longer time to compute a result. Effective false perspective results were demonstrated with Graphcut Textures [Kwatra et al. 2003], but these results required a pre-process to create scaled versions of the sample image, and searching over these multiple samples rather than a single sample image. The methods most similar to ours [Soler et al. 2002; Magda and Kriegman 2003] are most likely to be adaptable to support these kinds of operations, but as mentioned earlier both of these algorithms are much slower than ours and not quite interactive in speed. For this application, where determination of the correct scale and orientation parameters is a strong focus of the user, the ability to easily experiment with different parameters implies a strong requirement for interactive speed.

8. APPLICATIONS

The speed of texture synthesis enabled by the jump map framework in turn enables a number of applications which were difficult or impossible with previous approaches. Texture paint-brushes, for example, allow the painting of surfaces with a continuous texture. While texture paint-brushes are not novel in the image domain, our method allows interactive texture painting on surfaces as well. This can be particularly useful for touching up any rare visible artifacts in a fully-synthesized surface.

Another useful application is for systems which generate geometry dynamically. For example, games may have objects which get blown up into many pieces, or need to randomly generate objects to populate the game world (consider an asteroid field in a space-based game, for example). With jump map-based texture synthesis, these objects can now have textures synthesized on them from general texture imagery, without unduly burdening the runtime system.

Perhaps most useful in practice has been improvements made possible by the jump map for simple interactive modelling or rapid prototyping applications. Consider the problem of assigning a texture to an object. With our system, a user can interactively decide from among a library of textures, seeing instant results as to how the object appears with a

selected texture (our system achieves real-time interaction for models on the order of a few tens of thousands of faces). More importantly, the user burden of deciding *how* a texture should appear on a surface is greatly reduced with our system. The apparent size of the texture with respect to the surface, as well as its orientation (i.e., which direction on the surface corresponds to "up" in the texture) are parameters which, invariant of the automatic texture synthesis algorithm used, *must* be decided by the user. Previously, if a bad decision were made regarding these parameters, the cost of resynthesizing the texture on the surface to attempt to alleviate the problem would be quite burdensome, providing a disincentive to experiment. Using the jump map-based approach, an artist can vary these parameters continuously, receiving instant feedback as to the appearance of the object as these parameters change.

As the orientation of the texture is typically specified by a vector field on the surface, this is particularly powerful when used in conjunction with tools for editing the orientation field. Previous approaches primarily rely on using a static, pre-specified orientation field based on interpolating a set of anchor vectors. However, it can be quite unintuitive how this interpolation proceeds over a surface, possibly producing singularities in undesirable locations. The speed of our approach allows interactive specification of the orientation field, allowing operations to locally "twist" the texture on the surface in a specific region (locally manipulating the vector field), or perform global rotations of the texture on the surface. As each orientation change is made, the surface may be completely (or, if possible, locally) retextured, giving the artist immediate useful feedback in response to their efforts.

Note that some algorithms [Soler et al. 2002; Zhang et al. 2003; Magda and Kriegman 2003; Kwatra et al. 2003] generate higher quality results when a complete result is available to work from (i.e., multiple passes can improve results). Generally, this is due to having a full set of neighbourhood information to match against, rather than only that part of the neighbourhood which has already been synthesized on the current pass. A natural application, then, is to use jump map-based texture synthesis for interactive specification of the output texture, and then use the jump map-based result as input to an alternative, higher-quality algorithm. The user, while specifying textures, is effectively given a preview of the final results from the interactively-generated jump map-based result. Once the user is satisfied with the texture parameters, the higher quality algorithm may be invoked offline to refine the results generated by the jump map-based algorithm. Thus, our techniques can improve the effective speed and quality of other higher-quality, but slower multi-pass texture synthesis algorithms.

9. CONCLUSIONS AND FUTURE WORK

We have presented fast algorithms for image texture synthesis and texture synthesis directly on 3D surfaces. Our approach is based on first analyzing the sample texture offline to create a jump map, and then using the jump map to synthesize new texture similar to the example texture at interactive rates. While there are recent very high-speed approaches for synthesizing images[Cohen et al. 2003], they do not generalize easily to surfaces. To our knowledge, our algorithm is the only one to deliver interactive synthesis speed on surfaces of reasonable size, about an order of magnitude faster than the fastest alternative[Magda and Kriegman 2003]. We further do not require the mesh to be resampled or use additional texture memory as the scale of the texture on the mesh is changed. Our framework allows very useful novel modelling tools for interactively specifying a texture on a mesh, while

allowing the easy incorporation of new effects such as progressive variation.

A primary avenue for future work is the need for more powerful and intuitive user interaction mechanisms for specifying how textures are to be mapped on surfaces. While our ability to specify sources and sinks in the orientation field have proven useful, as well as global and local rotations of the vector field, the ability to easily specify *lines* of singularities in the orientation field across prominent ridges of a surface would be quite powerful. A particularly intuitive approach would be to simulate a "virtual comb" to direct the orientation field locally. We expect that combining these interaction techniques with the speed of our approach will deliver a very easy-to-use, powerful system for interactive modelling of textures on surfaces.

For the patch-based image synthesis algorithm, it may be possible to use a lower sampling density for the sources examined from each neighbouring patch. Additionally, it may be possible to define a per-input-pixel "suggested patch size" during preprocessing. In effect, this would say that after jumping to a particular position, a jump should not be taken within a certain number of pixels. This would be somewhat similar to the *texton masks* used in synthesizing progressively-variant textures[Zhang et al. 2003]. We could also relatively easily incorporate texton masks into the analysis phase, potentially increasing the quality of jumps stored in the jump map. These modifications amount to a more pixel-oriented algorithm which attempts to prevent breaking up prominent texture features. Another alternative, inspired by Nealen and Alexa's approach [2003], is to perform a pixel-based post-process near regular patch boundaries, effectively performing a second pass over the output to attempt to de-regularize the patch boundaries.

ACKNOWLEDGMENTS

This work was supported in part by a grant from the National Science Foundation (CCR-0086084). Textures used are courtesy www.scenic-route.com, www.mayang.com, and www.grsites.com, as well as the Brodatz and VisTex collections. Models are from Stanford, Georgia Tech, www.3dtotal.com, and www.3dcafe.com. Thanks to the many researchers who have published results or code on the web to allow for easy comparison of techniques.

REFERENCES

ARIKAN, O. AND FORSYTH, D. A. 2002. Interactive motion generation from examples. In *Proceedings of SIGGRAPH 2002*. ACM SIGGRAPH, San Antonio, TX, 483–490.

ASHIKHMIN, M. 2001. Synthesizing natural textures. In *Proceedings of 2001 ACM Symposium on Interactive 3D Graphics*. ACM SIGGRAPH, North Carolina, 217–226.

ASHIKHMIN, M. 2003. Fast texture transfer. *IEEE Computer Graphics and Applications 23,* 4, 38–43.

BAGOMJAKOV, A. AND GOTSMAN, C. 2002. Universal rendering sequences for transparent vertex caching of progressive meshes. *Computer Graphics Forum 21,* 2, 137–148.

BERTALMIO, M., VESE, L., SAPIRO, G., AND OSHER, S. 2003. Simultaneous structure and texture image inpainting. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2003)*. Vol. 2. IEEE Computer Society, 707–712.

CARR, N. A. AND HART, J. C. 2002. Meshed atlases for real-time procedural solid texturing. *ACM Transactions on Graphics 21,* 2 (Apr.), 106–131.

COHEN, M. F., SHADE, J., HILLER, S., AND DEUSSEN, O. 2003. Wang tiles for image and texture generation. *ACM Transactions on Graphics, SIGGRAPH 2003 22,* 2 (July), 286–294.

CRIMINISI, A., PEREZ, P., AND TOYAMA, K. 2003. Object removal by exemplar-based inpainting. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2003)*. Vol. 2. IEEE Computer Society, 721–728.

DE BONET, J. S. 1997. Multiresolution sampling procedure for analysis and synthesis of texture images. In *Proceedings of SIGGRAPH '97*. ACM SIGGRAPH, Los Angeles, CA, 361–368.

DISCHLER, J.-M., MARITAUD, K., LÉVY, B., AND GHAZANFARPOUR, D. 2002. Texture particles. In *Eurographics 2002 Conference Proceedings*. Eurographics Association, Saarbr ucken, Germany.

DRORI, I., COHEN-OR, D., AND YESHURUN, H. 2003. Fragment-based image completion. *ACM Transactions on Graphics, SIGGRAPH 2003 22,* 2 (July), 303–312.

EBERT, D., MUSGRAVE, K., PEACHEY, D., PERLIN, K., AND WORLEY, S. 1994. *Texturing and Modeling: A Procedural Approach*. AP Professional, Cambridge, MA.

EFROS, A. A. AND FREEMAN, W. T. 2001. Image quilting for texture synthesis and transfer. In *Proceedings of SIGGRAPH 2001*. ACM SIGGRAPH, Los Angeles, CA, 341–346.

EFROS, A. A. AND LEUNG, T. K. 1999. Texture synthesis by non-parametric sampling. In *IEEE International Conference on Computer Vision*. IEEE Computer Society, Corfu, Greece, 1033–1038.

FLEISCHER, K. W., LAIDLAW, D. H., CURRIN, B. L., AND BARR, A. H. 1995. Cellular texture generation. In *Proceedings of SIGGRAPH '95*. ACM SIGGRAPH, Los Angeles, CA, 239–248.

HARRISON, P. 2001. A non-hierarchical procedure for re-synthesis of complex textures. In *Winter School of Computer Graphics (WSCG'01) 2001*, V. Skala, Ed. 190–197.

HEEGER, D. J. AND BERGEN, J. R. 1995. Pyramid-based texture analysis/synthesis. In *SIGGRAPH '95 Proceedings*. ACM SIGGRAPH, 229–238.

HERTZMANN, A., JACOBS, C. E., OLIVER, N., CURLESS, B., AND SALESIN, D. H. 2001. Image analogies. In *Proceedings of SIGGRAPH 2001*. ACM SIGGRAPH, Los Angeles, CA, 327–340.

HERTZMANN, A. AND SEITZ, S. M. 2003. Shape and materials by example: A photometric stereo approach. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR '03)*. Vol. 1. IEEE Computer Society, Madison, WI, 533–540.

JIA, J. AND TANG, C. K. 2003. Image repairing: Robust image synthesis by adaptive nd tensor voting. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2003)*. Vol. 1. IEEE Computer Society, 643–650.

JOLLIFE, I. T. 1986. *Principal Component Analysis*. Springer-Verlag, New York.

KOVAR, L., GLEICHER, M., AND PIGHIN, F. 2002. Motion graphs. In *Proceedings of SIGGRAPH 2002*. ACM SIGGRAPH, San Antonio, TX, 473–482.

KWATRA, V., SCHÖDL, A., ESSA, I., TURK, G., AND BOBICK, A. 2003. Graphcut textures: Image and video synthesis using graph cuts. *ACM Transactions on Graphics, SIGGRAPH 2003 22,* 2 (July), 277–286.

LEE, J., CHAI, J., REITSMA, P. S. A., HODGINS, J. K., AND POLLARD, N. S. 2002. Interactive control of avatars animated with human motion data. In *Proceedings of SIGGRAPH 2002*. ACM SIGGRAPH, San Antonio, TX, 491–500.

LEGAKIS, J., DORSEY, J., AND GORTLER, S. J. 2001. Feature-based cellular texturing for architectural models. In *Proceedings of SIGGRAPH 2001*. ACM SIGGRAPH, Los Angeles, CA, 309–316.

LEUNG, T. K. AND MALIK, J. 1999. Recognizing surfaces using three-dimensional textons. In *IEEE International Conference on Computer Vision (ICCV '99)*. IEEE Computer Society, Kerkya, Greece.

LIANG, L., LIU, C., XU, Y.-Q., GUO, B., AND SHUM, H.-Y. 2001. Real-time texture synthesis by patch-based sampling. *ACM Transactions on Graphics (TOG) 20,* 3, 127–150.

LIU, C., SHUM, H.-Y., AND ZHANG, C.-S. 2001. A two-step approach to hallucinating faces: Global parametric model and local non-parametric model. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR '01)*. Vol. 1. IEEE Computer Society, Kauai, Hawaii, 192–198.

MAGDA, S. AND KRIEGMAN, D. 2003. Fast texture synthesis on arbitrary meshes. In *Proceedings of the Eurographics Symposium on Rendering 2003*. Eurographics Association, Leuven, Belgium, 82–89.

MOUNT, D. M. 1998. Ann programming manual. Tech. rep., Department of Computer Science, University of Maryland, College Park, Maryland.

NEALEN, A. AND ALEXA, M. 2003. Hybrid texture synthesis. In *Proceedings of the Eurographics Symposium on Rendering 2003*. Eurographics Association, Leuven, Belgium, 97–105.

NEYRET, F. AND CANI, M.-P. 1999. Pattern-based texturing revisited. In *Proceedings of SIGGRAPH '99*. ACM SIGGRAPH, Los Angeles, CA, 235–242.

PIPONI, D. AND BORSHUKOV, G. 2000. Seamless texture mapping of subdivision surfaces by model pelting and texture blending. In *Proceedings of SIGGRAPH 2000*. ACM SIGGRAPH, New Orleans, LA, 471–478.

PRAUN, E., FINKELSTEIN, A., AND HOPPE, H. 2000. Lapped textures. In *Proceedings of SIGGRAPH 2000*. ACM SIGGRAPH, New Orleans, LA, 465–470.

SAGAN, H. 1994. *Space-Filling Curves*. Springer-Verlag, New York.

SANDER, P. V., GORTLER, S. J., SNYDER, J., AND HOPPE, H. 2002. Signal-specialized parameterization. In *Proceedings of the Thirteenth Eurographics Workshop on Rendering Techniques*. Eurographics Association, Pisa, Italy, 87–98.

SCHÖDL, A., SZELISKI, R., SALESIN, D., AND ESSA, I. 2000. Video textures. In *Proceedings of SIGGRAPH 2000*. ACM SIGGRAPH, New Orleans, LA, 489–498.

SHEFFER, A. AND HART, J. C. 2002. Seamster: Inconspicuous low-distortion texture seam layout. In *Proceedings of IEEE Visualization 2002*. IEEE Computer Society, Boston, MA, 291–298.

SOLER, C., CANI, M.-P., AND ANGELIDIS, A. 2002. Hierarchical pattern mapping. In *Proceedings of SIGGRAPH 2002*. ACM SIGGRAPH, San Antonio, TX, 673–680.

TONG, X., ZHANG, J., LIU, L., WANG, X., GUO, B., AND SHUM, H.-Y. 2002. Synthesis of bidirectional texture functions on arbitrary surfaces. In *Proceedings of SIGGRAPH 2002*. ACM SIGGRAPH, San Antonio, TX, 665–672.

TURK, G. 2001. Texture synthesis on surfaces. In *Proceedings of SIGGRAPH 2001*. ACM SIGGRAPH, Los Angeles, CA, 347–354.

WEI, L.-Y. AND LEVOY, M. 2000. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of SIGGRAPH 2000*. ACM SIGGRAPH, New Orleans, LA, 479–488.

WEI, L.-Y. AND LEVOY, M. 2001. Texture synthesis over arbitrary manifold surfaces. In *Proceedings of SIGGRAPH 2001*. ACM SIGGRAPH, Los Angeles, CA, 355–360.

WEI, L.-Y. AND LEVOY, M. 2002. Order-independent texture synthesis. Tech. Rep. TR-2002-01, Computer Science Department, Stanford University. Apr.

WELSH, T., ASHIKHMIN, M., AND MUELLER, K. 2002. Transferring color to grayscale images. In *Proceedings of SIGGRAPH 2002*. ACM SIGGRAPH, San Antonio, TX, 277–280.

XU, Y.-Q., GUO, B., AND SHUM, H. 2000. Chaos mosaic: Fast and memory efficient texture synthesis. Tech. Rep. MSR-TR-2000-32, Microsoft Research. April.

YING, L., HERTZMANN, A., BIERMANN, H., AND ZORIN, D. 2001. Texture and shape synthesis on surfaces. In *Proceedings of the Twelfth Eurographics Workshop on Rendering Techniques*. Eurographics Assocation, London, UK, 301–312.

ZELINKA, S. AND GARLAND, M. 2002. Towards real-time texture synthesis with the jump map. In *Proceedings of the Thirteenth Eurographics Workshop on Rendering Techniques*. Eurographics Association, Pisa, Italy, 99–104.

ZELINKA, S. AND GARLAND, M. 2003. Interactive texture synthesis on surfaces using jump maps. In *Proceedings of the Eurographics Symposium on Rendering 2003*. Eurographics Association, Leuven, Belgium, 90–96.

ZHANG, J., ZHOU, K., VELHO, L., GUO, B., AND SHUM, H.-Y. 2003. Synthesis of progressively variant texture on arbitrary surfaces. *ACM Transactions on Graphics, SIGGRAPH 2003 22,* 2 (July), 295–302.

Received Month Year; revised Month Year; accepted Month Year