EUROGRAPHICS 2004 / M.-P. Cani and M. Slater
(Guest Editors)

Volume 23 (*2004*), *Number 3*

Procedural Texture Matching and Transformation

Eric Bourque and Gregory Dudek

Centre for Intelligent Machines, McGill University, Montréal, Québec, Canada

Abstract

We present a technique for creating a smoothly varying sequence of procedural textures that interpolates between arbitrary input samples of texture. This texture transformation uses a library of procedural shaders and selects the correct shaders and associated parameters to accomplish the task.
In general, selecting a procedural texture from a library, or finding the correct parameters to produce a smooth texture transition can be complex and time consuming. We propose a strategy for automating this process. While superficially this problem appears intractable for both humans and computational systems, its natural characteristics make a computational solution feasible. We present an algorithm and experimental results demonstrating this approach.
Transformation between two textures can then be achieved procedurally, while enforcing perceptual similarity constraints between adjacent texture frames. We describe a technique for efficiently sampling the parameter domain of a shader based on a texture similarity function to create a smooth path through its texture range. In the case of evolving between several shaders, a method is described to obtain the best jump-points which can be used to connect different shaders smoothly in texture space. Several examples of the technique are shown, and future directions as well as potential problems are discussed.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Texture

1. Introduction

This paper addresses the creation of textures and texture transformations using shaders; that is, programs that generate a desired shading or texture. Shaders are the de facto standard mechanism for synthesising photo-realistic textures. Using one can often be superficially easy, but finding the right shaders from a library, and the right settings for that shader to produce a sought-for appearance can be astonishingly difficult. In this paper we do not discuss the creation of shaders, but how to optimise the choice of a shader and its parameters to produce a desired appearance, as defined by a sample image. Of course, once such a shader and its parameters have been discovered, a user can modify the parameters to achieve alternative effects.

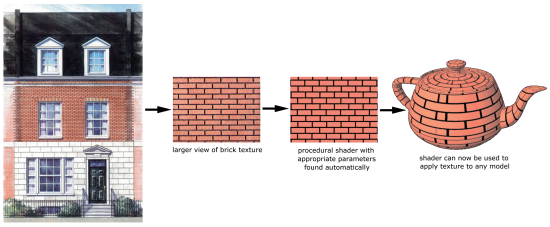


Figure 1: *An example of procedural texture matching. An input texture is acquired from an architect's drawing, and a suitable procedural shader and parameters are found to replicate the appearance of the texture so that the shader can be applied to an arbitrary model.*

To exemplify why this problem needs to be solved, consider that a library of hundreds of shaders can be available to a sophisticated user (and the number of shaders is constantly increasing). A typical shader can have five or ten parameters, and some can have substantially more. Further, the texture that is produced by a shader can vary substantially over the range of these parameters. Thus, to find a desired texture, a user must search over a complicated and high-dimensional space. Finding the right combination of parameters to define a point in this space is clearly problematic.

Our work considers the problem of starting with an initial description of a target texture (in the form of a sample image) and finding the right shader and shader parameters to match that sample as well as possible. Once we have solved this, we can consider the problem of finding a *sequence* of shaders and associated parameters that will produce a gradually varying sequence of textures that accomplish a transition from the initial texture to a final texture. We refer to this as a *texture transformation*. In general, a texture transformation may involve using more than one shader and using varying parameters for each. Significantly, a good texture transformation is one which takes a *short* path from the initial texture to the final one. This shortness is not measured with respect to the variations in the parameters, however, but with respect to the perceptual variations the texture must traverse.

Note that even with many available shaders, the space of possible images is far larger than the set of textures that can be synthesised, so some texture images will be hard to approximate. Likewise, a good texture transformation will not always be possible, particularly if the repertoire of available shaders is limited.

Our texture matching approach is based on four key stages (Figure 1 shows an example of the process):

1. A global search strategy over shaders to select ones that might produce interesting results.
2. A global search over a single shader to obtain a rough estimate of suitable input parameter settings.
3. A local search strategy to optimise parameter settings given a rough guess.
4. A perceptual texture comparison function that allows us to estimate the quality of our solution.

Note that by using procedural textures for this work, we obtain several advantages over either image samples as texture maps, or the use of stochastic image-based texture synthesis as proposed by Efros *et al.* [EL99, EF01], Wei and Levoy [WL00] and others. Namely, procedural textures (once we know the parameter settings) can be very compact, extremely flexible, of dimensionality greater than two (for solid texturing), can be evaluated in arbitrary order (useful for variable level of detail applications) and are resolution independent. Using a procedural texture also allows us to generate textures that are akin to a target in some desired way, while allowing us the freedom to make subtle changes.

2. Previous Work

This paper deals with the selection of one or more procedural textures and associated parameters given a specification in terms of an image. This is loosely related to research which seeks to synthesise a large texture field given only a small sample of the desired texture [HB95, De 97, EL99, WL00, HJO*01, DMLG02]. While texture synthesis methods share with our work the ability to generate arbitrary texture fields from a small sample, they differ in terms of the compactness of the description, the scientific objectives, and the manner in which the results can subsequently be re-configured.

If we consider, for example, methods based on Markov models of texture [EL99, WL00, EF01] then although these techniques produce compelling results they have several limitations. For instance, the Markov framework does not allow for minor changes in the characteristics of the texture being generated (wider bricks, puffier clouds, etc.), except in some limited cases [ZZV*03]. In addition, the synthesised texture is generally limited to have a resolution no greater than the original texture, unlike those generated using procedural techniques, and are therefore not well suited to photo-realistic rendering.

In the psychophysics of texture modelling, a large body of research indicates that texture similarity can be modelled using statistical methods particularly in the Fourier domain [Jul62]. An alternative formulation of the problem uses banks of band-pass filters, for example, in the form of a Laplacian pyramid [AS87]. More recently several authors have observed that many textures can be directly synthesised from these statistical similarity measures [HB95, GF01].

A rather different approach to texture synthesis is to explicitly model the bi-directional texture function (BTF) which describes the interaction between texture and lighting as a function of angle. This can be accomplished either directly [DVNK99] or via subspace modelling [SH00] to yield highly realistic reproductions of specific physical surfaces once the requisite measurements have been acquired.

Liu *et al.* [LLSY02] propose a method for morphing between two texture samples using a pattern-based approach which requires the end-user to specify feature correspondence landmarks. Like the traditional sample-based texture synthesis techniques described above, this method is not well suited to photo-realistic rendering.

In the specific case of wood or brick textures it is possible to estimate domain-specific texture attributes to permit good quality synthesis from procedural descriptions [LP00]. How to generalise the methods to arbitrary textures is not readily apparent, but very positive results have been shown in restricted domains.

3. Texture Matching

3.1. Approach

We wish to approximate a given input target texture T using a procedural texture $p(u,v,\ldots)$. We will later show that given two sample textures, T_1 and T_2, we can produce a smoothly varying sequence of textures generated by one or more shaders.

Consider a *set* $\mathcal{P}$ of procedural textures $\{p_1,\ldots,p_n\}$, where each element, p_i, is a shader taking an arbitrary number of parameters. Given a texture target T, we wish to find

the element $p_i \in \mathcal{P}$, and the associated parameter vector $\mathbf{x}_i$ such that $p_i(\mathbf{x}_i)$ produces a texture perceptually similar to T. That is, we want to maximise a similarity measure $S()$ between the procedural candidate and the target texture: $S(T, p_i(\mathbf{x}_i))$. The implementation of $S()$ is discussed in Sec. 3.3.

The process for finding p_i and $\mathbf{x}_i$ are outlined below.

3.2. Searching in Texture Space

To find the right shader and parameters, we need to search across the range of each shader's input parameters. Unfortunately, it is unlikely that the (similarity) surface resulting from evaluating the target texture against the texture range of a particular shader will be convex. Of course, if there is a particular parameter setting which provides a good match, theoretically, exhaustive search of the parameter domain would eventually find it; however, we desire a tractable solution. This suggests a two-stage approach: a preliminary search using pre-computed data and an on-line refinement stage.

3.2.1. Global Search

As a pre-computation step, we generate a catalogue of samples in the parameter domain of each procedural texture p_i. Because it is possible that several parameter vectors will produce similar textures, we choose to adaptively sample the parameter domain of each procedural shader using a systematic subdivision technique. This allows us to avoid the costly generation of samples which do not give any novel information about the *interesting areas* of the parameter domain. A key issue, of course, is to sample densely enough to capture the expressiveness of the procedural texture, and to avoid getting caught in local minima during the next stage of our search. The texture samples in the catalogues are stored in an image database using a lossless compression format. Each sample is rendered at 256×256 pixels, with most catalogues containing on the order of 200 samples, combining for an average storage cost of 11MB per catalogue.

During the global search to match a target texture T, our algorithm evaluates the texture similarity function S over each of the pre-computed samples of $p_i \in \mathcal{P}$. The most promising parameter vectors $\mathbf{x}_g$ which give maximum values for $S(T, p_i(\mathbf{x}_g))$ for each of the p_i are then used as starting points during the local search in order of their admissibility as described below. If a large disparity for the best global match in p_i is detected, then that particular shader is not used in the next stage of the search.

As mentioned above, it is possible that multiple points in the parameter domain of a particular shader will generate similar textures, yet the local similarity surfaces surrounding these points can have different characteristics. Since searching in smoother spaces is both more efficient and tends to yield better results, we would like to prioritise our search based on an admissibility factor of the local smoothness.

Given a reference point $\mathbf{x}$ and a set of points $D = \{d_1, \ldots, d_n\}$ distributed in the local neighbourhood of $\mathbf{x}$, we can compute the admissibility of $\mathbf{x}$ as follows:

$$A(\mathbf{x}) = \frac{1}{||D||} \sum_{d_i \in D} \frac{S(P(\mathbf{x}), P(d_i))}{||\mathbf{x} - d_i||_2}$$

This measure can be refined incrementally by adding new points, as compute time permits, and can in fact be pre-computed when the shader is initially added to the library.

3.2.2. Local Search

Starting with each of the $p_i(\mathbf{x}_g)$'s from the global search, we perform a local optimisation to find a parameter vector which will produce a texture which best represents the target texture. For the results presented in this paper, we have used both simplex optimisation, and a gradient ascent based optimisation.

The evaluation of the similarity function $S(T, p_i(\mathbf{x}_g))$ with a new parameter vector $\mathbf{x}'_g$ entails the rendering of a texture sample since $\mathbf{x}'_g$ is not contained in the sample catalogue. Consequently, computing $\nabla S(T, p_i(\mathbf{x}_g))$ is something we would like to avoid, motivating our desire to use the simplex method when possible as it does not rely on computing derivatives. In particularly unforgiving cases, however, the user can select to use gradient based methods.

When the local search is no longer able to take a maximising step, the parameter vector which results in the greatest similarity to the target texture determines the final shader and parameter vector. The user may terminate the search at any point if the current match is to their liking, thus avoiding searching other shaders, or other starting points within the same shader.

3.3. Evaluating Texture Similarity

In order to match a synthetic texture to a target, an important requirement is a distance function to indicate the quality of a candidate match, i.e., a texture similarity function that operates on pairs of images. While a naive solution to this problem might be based on the difference between images, that would fail to capture the notion of texture fields that *look* the same even when the individual pixels are different. For example, two images of snow falling may have the same apparent textures yet no two pixels may be identical. Luckily, there is a substantial body of literature on the psychophysics of texture similarity. A key observation is that the local power spectrum of a texture is critical to distinguishing or segregating textures [Jul62] (recall that the power spectrum describes the mixture of spatial frequencies in an image and it can be obtained readily using a Fourier transform). Precisely how the power spectrum (and perhaps even the phase information) relates to texture discrimination is a matter of some debate, but using the two simple schemes

described below provides a good approximation of our natural intuition.

More specifically, current theories of texture discrimination maintain that when two textures produce a similar response to frequency-selective oriented linear filters they are perceptually similar and perhaps indistinguishable [BA88, MP90, Ber91, HB95]. This can be demonstrated through texture segregation – the notion that it is difficult to identify the borders of similar textures when they are juxtaposed.

In order to compare the results of our synthesis process with the target texture, we need a measure of the perceptual similarity of two textures, T_1 and T_2:

$$S^*(T_1, T_2) \in (0, 1]$$

where a value of 1 indicates that they are indistinguishable, and as values approach 0 the two textures are considered to be increasingly distinct. The definition of this ideal measure is the subject of ongoing research in the psychophysics community. We wish to define a computational texture similarity function S, to approximate our ideal measure S^*. One function we have used with success is:

$$S_F(T_1, T_2) = 1 - ||F_m(T_1) - F_m(T_2)||_2$$

where $F_m(T)$ is the magnitude of the amplitude image of the Fourier transform of T; the differences are weighted radially and are normalised.

In addition to the power spectrum of the texture sample, we have also considered the use of the histogram of the energy distribution in a Laplacian pyramidal representation of the images, as used by several authors for texture analysis and synthesis [AS87, HB95, De 97]. Pairs of textures are then compared by computing the histogram difference between the corresponding levels in their Laplacian pyramids:

$$S_L(T_1, T_2) = \sum_i w_i \left| H\left(L_i(T_1)\right) - H\left(L_i(T_2)\right) \right|$$

where H is the histogram of an image, L_i is level i in the Laplacian pyramid of a texture, T_1 and T_2 are the textures being compared, and w_i is a weighting factor.

This representation also captures the band-pass energy distribution in the image. In practise, we find the difference in both the Laplacian histograms and the Fourier power spectrum have advantages as texture similarity functions. We report illustrative results using both functions in the text below. More generic perceptually motivated image difference algorithms [NMP98] might also be well suited to the texture domain. In the context where the particular choice of texture metric is not significant, we refer to it simply as $S()$.

3.4. Examples

Textures are often divided into two classes, namely, stochastic and deterministic. Stochastic textures generally do not

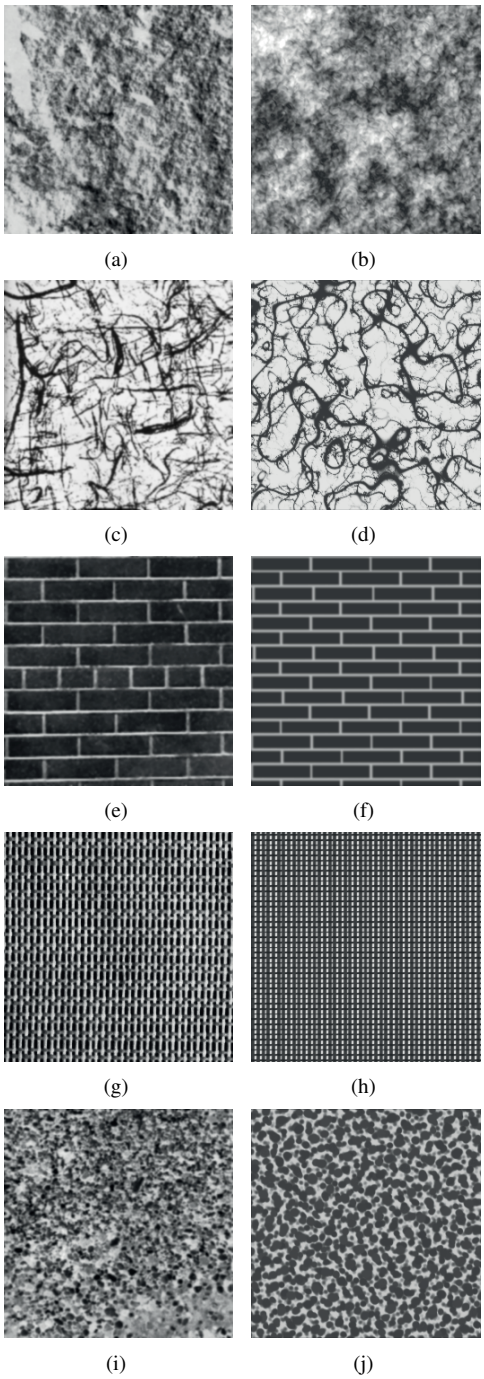


Figure 2: *Examples of procedural texture matching using Brodatz textures as targets, and a small repertoire of shaders. The images on the left are the texture targets, and the images on the right are procedurally generated using the automatically recovered shader and parameters. In every case the synthesised texture appears to be the best which could be achieved with the available shaders.*

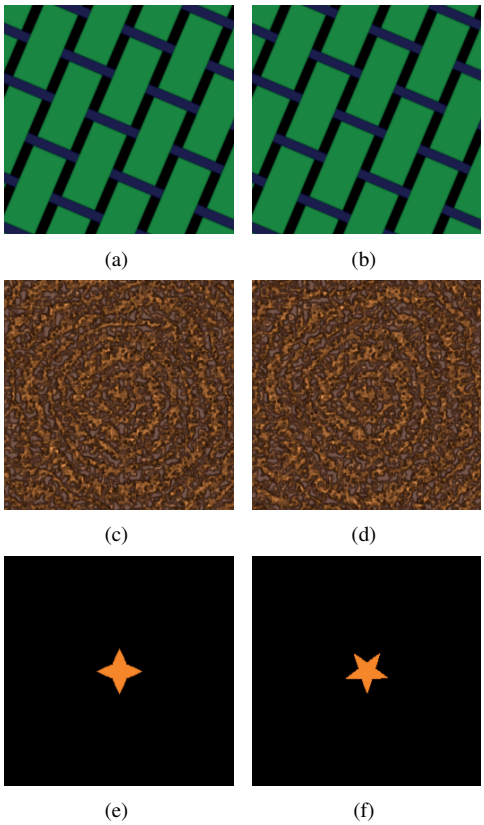


Figure 3: *Examples of procedural texture matching using synthetic images as targets. Using procedurally generated textures as targets should guarantee a good match since we know beforehand that the desired texture lies within the texture range of the shaders being searched. However, only the first (a) target generated a match with the exact same parameters. The second example (c) is a close match, but with entirely different parameters, and in the last example (e), the match is poor. This is discussed further in Sec. 3.4.*

contain any easily identifiable primitives, whereas deterministic textures largely consist of well-defined primitives combined with a set of rules governing their placement. In practise, many textures exhibit some combination of properties from both classes. Prior work in the field of texture synthesis tends to focus on only one of these texture classes, the predominant methods being based on Markov random fields which assume that the desired targets are stationary, local, stochastic textures. The deterministic texture synthesis methods attempt to measure domain specific attributes, and therefore can not be used to synthesise stochastic textures.

In order to demonstrate our method of procedural texture matching, we have chosen some sample target textures from the Brodatz album [Bro66]. These textures are commonly used as a reference point for various texture algorithms in the perception community, and are therefore equally well suited to exemplify our procedural texture matching framework. The matching examples shown in Fig. 2 demonstrate positive results for both stochastic and deterministic textures. The average match time was 12 minutes, with under 100 iterations for all cases.

While most of the examples show matches which are perceptually very similar to their targets, there is one texture (Fig. 2(i)) for which a less suitable match was found. It will not always be the case that we can find a close match if the target texture is not contained in the texture ranges of the procedural textures which we are searching. In this situation, we can only hope to find a procedural texture p and a parameter vector $\mathbf{x}$ such that $p(\mathbf{x})$ is as perceptually similar as possible to the target texture T.

In addition to the natural texture examples, we have included some examples of searching for particular instances of synthetic textures which were themselves generated procedurally (Fig. 3). One of these examples fails to find a convincing match (Fig. 3(f)), and this is due to the fact that one of the parameters determines the number of points in the star which is intended to be an integer value. For this shader, the sampling phase did not produce the desired value of 4 but rather produced floating point values close to 4. This shader gives degenerate results when given non-integer parameters which means that the energy space between integer values is highly non-convex, thus preventing our algorithm from finding the exact result.

All of the results shown in this paper were obtained using a small collection of publicly available general purpose shaders, none of which were specifically written to replicate the appearance of any of the given natural target textures.

4. Texture Transformation

Once we are able to recover a procedural texture and suitable parameters to replicate a given texture sample, we can consider transforming those textures over time. We would like to be able to specify the starting and ending textures, as well as optional *key-textures* (specific textures that the transformation must contain) which can be placed in between the starting and ending textures.

The main goal of procedural texture transformation is to vary a shader's parameters over time so that the perceived difference between adjacent texture samples is minimised. This kind of smooth transition in a controlled environment has several applications in the field of graphical animation.

4.1. Approach

As a first step, the texture transformation algorithm must identify the initial and final shaders and their respective parameters. These are typically recovered from real or synthetic images using the texture matching approach described

above, although they can also be specified manually. The following temporal framework is desired: given a series of input textures $T_1, \ldots, T_n$, find a corresponding set of procedural shaders and their associated parameters $p_1(\mathbf{x}_1), \ldots, p_n(\mathbf{x}_n)$. Using the p_i, we want to produce a continuously changing texture $C(t), t \in [0,1]$ such that $C(0) = p_1(\mathbf{x}_1)$, $C(1) = p_n(\mathbf{x}_n)$, each T_i is used as a key-texture and $S(C(t), C(t + \Delta t))$ (the similarity of adjacent frames) is maximised.

Transformation between two texture samples can be divided into two classes: (1) transformation within the same shader, and (2) transformation between different shaders. These will each be discussed separately below.

4.2. Transformation Within a Shader

In order to evolve between two parameter vectors within the same shader, we again use a two-stage algorithm with some pre-computed data. After creating a catalogue of samples during the pre-computation step when a new shader is added to the library, a fully connected graph is constructed where the nodes are the samples from the catalogue, and the edge weights correspond to the similarity measure.

In the first stage of the transformation, we find the closest sample from the catalogue for the end-point parameter vectors as described in Sec. 3.2.1 above. We then compute the shortest path along the edges e_i through the sample graph using Dijkstra's algorithm [Dij59] with a path cost function of $\sqrt[n]{e_1^n + \ldots + e_m^n}$. This cost function allows paths which go through more samples from the catalogue to be encouraged by increasing the value of n.

To refine the chosen path, our second stage uses an adaptive linear subdivision technique. While the similarity measure between adjacent samples is less than a pre-specified value, another sample is inserted at the midpoint between the two samples in the shader's parameter space. This recursive solution assures that no two adjacent samples will have a large perceptual disparity.

There is, of course, the possibility that with a particularly uncooperative shader, repeated bisection of the parameter values will not result in adjacent samples falling below the perceptual threshold. While this has not been the case with the shaders we have tested thus far, we describe a possible approach for this situation in Sec. 5.

With this framework, key-textures are easily specified. If it is desirable to use a particular parameter vector at some point for a shader $p(\mathbf{x}_b)$ while evolving from $p(\mathbf{x}_a)$ to $p(\mathbf{x}_c)$ (denoted by $\rightsquigarrow$), the approach described above can be used to compute the paths from $p(\mathbf{x}_a) \rightsquigarrow p(\mathbf{x}_b)$, and from $p(\mathbf{x}_b) \rightsquigarrow p(\mathbf{x}_c)$. These paths can then simply be concatenated. This can be repeated for as many key-textures as necessary.

This approach provides a perceptually smooth path in the texture range of the particular shader due to the adaptive sampling of the parameter domain. This ensures that the shader is sampled more densely where the corresponding texture range is more volatile, thus avoiding the large changes between successive frames commonly associated with uniform sampling. Conversely, fewer samples are used when the texture range is more static – just enough to show a minimal change between frames.

4.3. Transformation Between Different Shaders

The technique described in Sec. 4.2 is only well suited to evolving within a procedural shader since different shaders no longer share a common parameter domain. In order to evolve *between* distinct shaders, we must first find the points in each shader's parameter domain whose associated textures are (perceptually) closest to each other. These *jump points* allow the technique described above to be used to evolve each texture to the closest jump point in its respective shader. For example, to evolve from $p_k(\mathbf{x}_a) \rightsquigarrow p_l(\mathbf{y}_a)$, we would first find the best jump point between p_k and p_l, that is, the point in each shader's parameter domain ($\mathbf{x}_j$ and $\mathbf{y}_j$) which gives a maximum similarity measure $S(p_k(\mathbf{x}), p_l(\mathbf{y})) \; \forall \; [\mathbf{x}, \mathbf{y}]$. The paths from each shader to their respective jump point can then be concatenated:

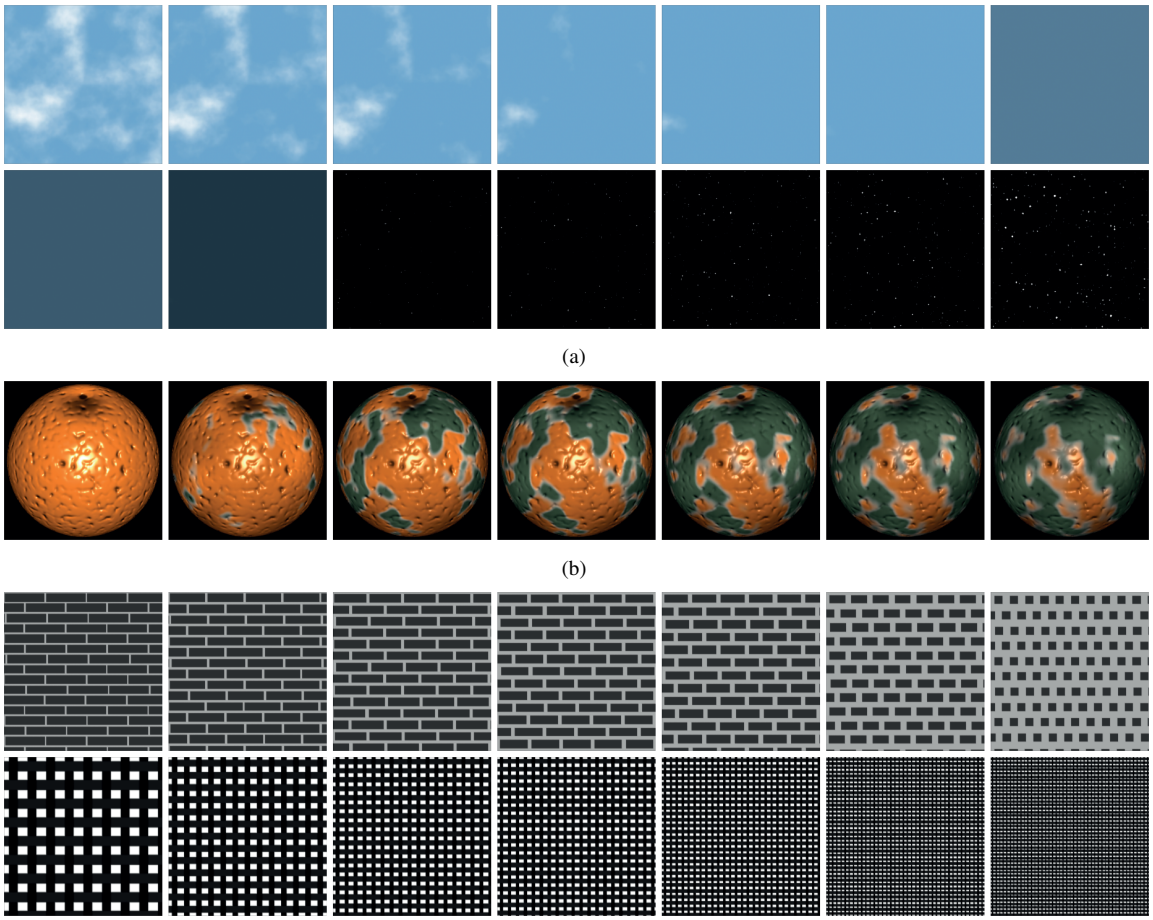
$$p_k(\mathbf{x}_a) \rightsquigarrow p_k(\mathbf{x}_j) \; : \; p_l(\mathbf{y}_j) \rightsquigarrow p_l(\mathbf{y}_a)$$

Since finding these jump points via exhaustive search is prohibitively costly, an approach similar to that described in Sec 3 can be used to reduce the computational burden. By adaptively sampling each shader sparsely, and comparing each sample to the (also sparse) samples of the other shader, the best candidate jump *regions* can be found. To narrow these regions to the actual jumps points, local optimisations of the similarity between the current candidate and the candidate in the other shader's jump region are performed. This is repeated in alternation until the distance travelled during a step for each candidate is negligible.

In the case where the shaders have limited texture ranges, or share little resemblance, the best jump points may not be very similar. Various strategies for dealing with this situation are presented in Sec. 5.

4.4. Examples

Several example frames from procedural texture transformations are shown in Fig. 5. For the first example (Fig. 5(a)), the real world targets shown in Fig. 4 were used to specify the starting and ending frames. In this particular case, the best jump point between the cloud shader and the star field shader results from parameters which produce a simple coloured background. This is actually the best match from a perceptual viewpoint since clouds and stars are distinct textures.

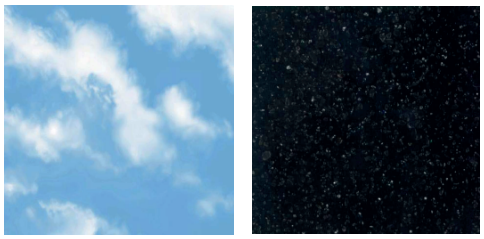


(a)

(b)

(c)

Figure 5: *Some example texture transformations. (a) is a transformation from the matches corresponding to the real world images shown in Fig. 4. (b) is a transformation from manually specified starting and ending textures. (c) is a transformation from Fig. 2(f) to Fig. 2(h).*



(a) (b)

Figure 4: *Texture targets used for the first and last frames shown in the transformation in Fig. 5(a).*

The second example (Fig. 5(b)) shows how the transformation can be used to create smoothly varying intermediate texture frames when the conditions for the endpoints are manually specified.

The third example is a transformation based on two of the Brodatz textures (Figs. 2(e) and 2(g)). All of these examples exhibit the smooth perceptual transition desired for effective texture transformation.

5. Discussion

We have presented a technique that encompasses the automated selection of procedural textures and also creates sequences of procedural textures given only a pair of images. Accomplishing this depends on pre-computing an automatically selected catalogue of representative samples of each shader. It also requires a perceptual texture similarity measure, an on-line search algorithm, and a perceptually-driven iterative subdivision procedure.

In order for the parameter estimation technique to succeed, the ensemble of procedural shaders must be large enough to approximate the specified texture target. If this is not the case, it will be detected by a large residual error in the similarity measure.

In the present work we assume that the transitions (jump points) between shaders can be determined without using an intermediate shader. In more difficult cases, to achieve a smooth transition from one shader to another may entail a longer trajectory by way of one or more other shaders. We are currently investigating this problem using stochastic search methods. As an alternative to planning a trajectory through disparate shaders, it is also possible to select a jump point between shaders and then use morphing techniques to smooth the transition. We find this latter approach less appealing since it entails a challenging correspondence problem to automate the morph.

The path planning process in texture space is based on graph search, and refinement using adaptive linear subdivision. In some cases the adaptive linear subdivision may not succeed, or we may wish to have more precise control over the sequence of textures. For example, we might want to avoid some types of appearance or shader parameter vectors while guaranteeing we evolve in a specific fashion. We are currently exploring the use of high-dimensional path planners to enable this type of control.

References

[AS87] ADELSON E., SIMONCELLI E.: Orthogonal pyramid transforms for image coding. In *SPIE Visual Communications and Image Processing II* (1987), vol. 845, pp. 50–58.

[BA88] BERGEN J., ADELSON E.: Early vision and texture perception. *Nature 333* (1988), 363–364.

[Ber91] BERGEN J.: *Spatial Vision*, vol. 10. CRC Press, 1991, ch. 5: Theories of visual texture perception, pp. 114–134.

[Bro66] BRODATZ P.: *Textures – A Photographic Album for Artists and Designers*. Dover, 1966.

[De 97] DE BONET J.: Multiresolution sampling procedure for analysis and synthesis of texture images. In *SIGGRAPH* (1997), pp. 361–368.

[Dij59] DIJKSTRA E. W.: A note on two problems in connexion with graphs. *Numerische Mathematik 1* (1959), 269–271.

[DMLG02] DISCHLER J.-M., MARITAUD K., LÉVY B., GHAZANFARPOUR D.: Texture particles. *Computer Graphics Forum 21*, 3 (2002).

[DVNK99] DANA K., VAN GINNEKEN B., NAYAR S., KOENDERINK J.: Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics 18*, 1 (January 1999), 1–34.

[EF01] EFROS A., FREEMAN W.: Image quilting for texture synthesis and transfer. In *SIGGRAPH* (2001), pp. 341–346.

[EL99] EFROS A., LEUNG T.: Texture synthesis by non-parametric sampling. In *International Conference on Computer Vision (ICCV)* (September 1999), vol. 2, pp. 1033–1038.

[GF01] GURNSEY R., FLEET D.: Texture space. *Vision Research 41*, 6 (2001), 745–757.

[HB95] HEEGER D., BERGEN J.: Pyramid-based texture analysis/synthesis. In *SIGGRAPH* (1995), pp. 229–238.

[HJO*01] HERTZMAN A., JACOBS C., OLIVER N., CURLESS B., SALESIN D.: Image analogies. In *SIGGRAPH* (2001), pp. 327–340.

[Jul62] JULESZ B.: Visual pattern discrimination. *IRE Transactions on Information Theory IT-8* (1962), 84–92.

[LLSY02] LIU Z., LIU C., SHUM H.-Y., YU Y.: Pattern-based texture metamorphosis. In *Pacific Graphics* (2002), pp. 184–191.

[LP00] LEFEBVRE L., POULIN P.: Analysis and synthesis of structural textures. In *Graphics Interface* (May 2000), pp. 77–86.

[MP90] MALIK J., PERONA P.: Preattentive texture discrimination with early vision mechanisms. *Journal of the Optical Society of America 7*, 5 (May 1990), 923–932.

[NMP98] NEUMANN L., MATKOVIC K., PURGATHOFER W.: Perception based color image difference. In *Proceedings of Eurographics* (1998), vol. 17, pp. 233–241.

[SH00] SUEN P., HEALEY G.: The analysis and recognition of real-world textures in three dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI) 22*, 5 (May 2000), 491–503.

[WL00] WEI L.-Y., LEVOY M.: Fast texture synthesis using tree-structured vector quantization. In *SIGGRAPH* (2000), pp. 479–488.

[ZZV*03] ZHANG J., ZHOU K., VELHO L., GUO B., SHUM H.-Y.: Synthesis of progressively-variant textures on arbitrary surfaces. *ACM Transactions on Graphics 22*, 3 (2003), 295–302.