Universal Simulation of Textures

Gustavo Brown Rodriguez

Abstract

In this work we study the problems of texture simulation (or synthesis) and texture mixture from an information-theoretic point of view. Our goal is to apply recently developed notions of universal types and universal simulation, based on the Lempel and Ziv *LZ78* universal compression scheme, in a new framework for simulating textures and texture mixtures. A simulated texture needs to meet two basic properties; it has to be visually 'similar' to the input texture but at the same time it has to be different from the input texture. In the context of universal types, the notion of `similarity' is a statistical one: two sequences from the same universal type are, in the limit, indistinguishable by k-th order Markov models, for any value of k. The richness of the simulated output is measured, in turn, in terms of its entropy or unpredictability. We seek simulation schemes that maximize the entropy of their output given that the statistical similarity constraints are satisfied. Thus, there is a tension or trade-off between these two defining parameters of a simulation scheme. As we can measure the size of the universal type class, we can calculate the size of the set of different simulated textures our algorithm can synthesize for a given texture, and, thus, the entropy of the simulated texture, which is chosen uniformly at random from that set. Raw application of universal simulation presents some challenges when simulating practical continuous-tone textures. A series of refinements of our simulation scheme allow us to deal with such problems, strengthening our similarity criterion to produce visually better simulations. This comes at the expense of a decrease in unpredictability, which we can quantify. We extend our approach to multiple source textures, which allows us to simulate texture mixtures of user-defined proportions according to a well defined formal model. As with the case with single input textures, we give a procedure to quantify the unpredictability of the simulated texture mixtures.

Acknowledgements

I would like to thank Dr. Alfredo Viola, Dr. Gadiel Seroussi and Dr. Guillermo Sapiro for their insightful help and guidance throughout this work.

Contents

1 Introduction

Understanding texture is an important problem in image processing. The goal of *texture synthesis* is to produce, given an input texture, new textures which appear to be generated by the same process, i.e., they look similar yet they are not the same. Texture synthesis has been extensively addressed in the literature(see, e.g, [4], [15]). Among other things, texture synthesis has been used in image/video transmission applications to save bandwidth [14] (instead of sending the full background it can be used to send just a patch of it and then synthesize the full background) or in image inpainting to restore damaged photographs or replace selected areas of an image [3]. However, other texture-related fields of image sciences, like texture characterization [47, 51], texture decomposition [17], or texture mixture [2], are less understood, and are the subject of active research.

In this work we study texture images through an information-theoretic point of view, using universal type classes and universal simulation.

Throughout the thesis we will use the terms "synthesis" and "simulation" interchangeably. One of the main goals of this work is to extend the work of Seroussi [42, 43], which was primarily targeted at generic (1D) signals. We extend this approach to specifically address 2D, continuous tone, image textures. Several models for texture have been recently introduced. However, they are at most semi-automatic [51], requiring some sort of user interaction to properly model the input texture (e.g. user selection of a set of filter banks). We present an alternative model for texture characterization using universal simulation that could be helpful to understand statistical properties of textures. One of the most difficult problems when dealing with image textures is to define and characterize an appropriate source information model which could have generated the image. In this work, we investigate the application of the model of [42], as a way to provide a rigorous characterization of the statistical aspects of the texture simulation problem.

Universal simulation based on the classical *method of types* [11] was first introduced in [34]. In their setting, universality is defined in the sense that the input sequence and every simulated sequence are statistically indistinguishable under a selected parametric family of models of some *fixed* order. This notion was extended in [42], applying, in the

limit, simultaneously to models of *any order*. Our goal is to apply this notion of universal simulation in a new framework for simulating textures and texture mixtures, based on the universal type classes of [42]. In universal simulation based on universal type classes, two sequences that belong to the same *universal type class (UTC* for short) are indistinguishable under the selected family of models for any model order. Thus, after obtaining the UTC of an input texture image, we can simulate new images by sampling new instances from the UTC.

A simulated texture needs to meet two basic properties. On one hand, it has to be visually 'similar' to the input texture (i.e. 'important' features must be preserved by the simulation). This property is in essence subjective (i.e. which features are important depends on the application) and is generally the primary concern of the literature on texture synthesis algorithms. Back into our framework for simulating textures, if the statistics collected by the UTC are able to capture important features of the input texture, the simulated textures will likely show them. On the other hand, the simulated texture has to be different from the input. Just copying the input texture would not be considered an acceptable simulation, although it indeed has every feature present in it. This requirement, while intuitively assumed, has not been formalized quantitatively in previous works in the literature, and one of the main goals of this work is to provide such a mathematical formalization. Our aim is to describe schemes in which the "unpredictability" of the output texture can be shown to be maximized in a well defined mathematical sense, given that the output satisfies a certain "similarity" criterion. As we can measure the size of the class [42], we can calculate the number of different simulated textures our algorithm can synthesize. All of them, in principle, share the same features the statistics have captured.

For every simulation scheme there is always a trade-off between the ability to capture the essential features of an input texture and the richness or "unpredictability" of the simulation. In the case of universal simulation the UTC for the input texture should be rich enough to be able to simulate a broad set of textures. However, if the classes are too rich, the simulation would fail to synthesize visually similar textures.

In [42] universal classes are defined in terms of the well known Lempel and Ziv (LZ) universal compression scheme (to be more precise, the LZ78 variant [27]). The universal

type class of a one dimensional sequence (of symbols) is defined in [42] to be the set of sequences of the same length which span the same tree with the LZ *incremental parsing algorithm*. It has been proved in [42] that for any finite order k, the variational distance between the k^{th}-order empirical probability distributions of two sequences of the same universal type vanishes as $n \rightarrow \infty$. In other words, this means that, in the limit, all the sequences of the same universal type will share the same statistics for any model order.

To use this approach, we have to provide a way to preprocess an input texture (which is a 2-D signal) to produce a 1-D sequence. This is carried out by traversing the input image with a *plane filling curve* (e.g. Peano scan, Hilbert scan, etc). Once we have preprocessed the image, the UTC for the resulting sequence can be computed and new simulated sequences can be sampled from it. The main idea behind our approach is that two images which belong to the same UTC will have 'equivalent' textures, in the sense that they will share the same statistics for any order k. Thus, to simulate new images that resemble the input texture all we have to do is to sample uniformly at random from the UTC and then reverse back the traversal of the simulated sequence to get the new texture. Recall that our aim is to describe a simulation scheme in which the unpredictability of the output textures is maximized. It was shown in [42] that no simulation scheme which preserves the statistics simultaneously for all orders *k* can perform much better in this sense.

Although preservation of statistics of arbitrary order is a strong criterion of "similarity" for textures, it does not capture all aspects of visual similarity. Therefore we have to strengthen our criterion of "similarity" by providing additional constraints. For the rest of this introduction we will loosely use the term *UTC* in the following sense: we will not obtain the UTC directly from the input sequence, but from a sequence derived from it which complies with our stronger criterion of "similarity". Nevertheless, this UTC still retains the property that the simulated sequences preserve the parsing tree of the derived sequence, and therefore also preserves, asymptotically, the statistics of any order. We will justify this more precisely throughout the rest of the thesis. As a consequence of adding new similarity constraints, each UTC will be smaller and we will be able to discriminate unrelated textures better (recall the trade-off between the richness of the class and the ability to capture the essential features of a given texture). Another

challenge we face is that for continuous tone images of practical sizes, statistics of high orders gathered by the UTC might be diluted. Therefore, we apply techniques like quantization and prediction to address this issue. As we shall see, this will come at the expense of a reduction of entropy (i.e. the size of the UTC will be reduced, but taking away much of the instances which would not give an acceptable degree of visual similarity with the original texture).

We will also notice that the nature of the LZ incremental parsing algorithm often leads to visual discontinuities in portions of the simulated textures. While the statistical effect of these artifacts is not large, they are objectionable from a visual quality point of view. To address this problem, we further tune our simulation scheme to avoid this problem, again at the expense of output entropy. Furthermore we also incorporate some tools of *multi resolution analysis* (*MRA*), specifically the use of Simoncelli's *steerable pyramid decomposition* [46] , to improve the visual quality of the simulated textures and to prevent some visual artifacts.

After describing our texture simulation scheme, we extend our approach to multiple source textures, allowing the simulation of texture mixtures of user-defined proportions. Mixing textures is indeed an interesting area of research as it has not been extensively studied in literature, mainly due to the fact that the expected output has not been formally defined. Our framework provides a mechanism for synthesizing texture mixtures according to a well defined formal model.

In summary, our work presents a framework, based on recent results in universal type classes and universal simulation, for simulation and mixing of image textures, in which the "unpredictability" of the simulation can be quantified. Our proposed method can be combined with other classical image processing tools to allow for a flexible trade-off between the richness of the simulation and the ability to capture the essential features of texture.

The rest of the thesis is outlined as follows. Section 2 presents our goals and specific objectives of this thesis. Section 3 provides some necessary mathematical background (namely the concept of universal type classes and universal simulation, and a brief introduction to plane filling curves and wavelet decompositions). It also presents some definitions used in the remainder of this document. After the general overview and

mathematical background introduced on these sections, Section 4 describes some previous work in the field of texture synthesis and texture mixture. Section 5 presents our first attempt at using the concept of UTCs and universal simulation for texture simulation. Here we will describe the problems arising from the fact that images are finite length 2-D signals, as well as problems with the scanning of the image. Afterwards, in Section 6 we extend out interest to multiple texture sources and provide algorithms to simulate texture mixtures. Finally, Section 7 summarizes results, gives directions for future work and concludes the thesis.

2 Specific goals of the Thesis

The main goals of this thesis can be summarized as follows:

Introduce a framework that connects some aspects of information theory with the study of image textures. Use this framework to provide algorithms for simulation and mixture of textures, showing the trade-off between the notions of "similarity to the original" and "unpredictability" of the simulated textures.

Specific goals:

- Starting from an image that conforms (subjectively) to our definition of texture, use universal type classes and universal simulation to synthesize new images. This is addressed in section 5.1
- Measure the richness of our simulation scheme. Provide mathematical tools to determine the number of different textures our scheme can simulate for a given input texture. This is studied in section 3.5 and 5.
- Determine additional constraints the simulated textures should comply with to improve the visual results of the simulations. Sections 5.2 through 5.5 address this subject and introduce modifications to our basic simulation scheme accordingly.
- Evaluate the trade-off between the visual quality of our simulations and the unpredictability of simulations. This is also covered in Section 5.
- Given two or more input textures, define a way to simulate new images that are mixtures of them. Analyze the properties that these texture mixtures satisfy. This is addressed in section 6.

3 Mathematical Background

In this section we present some mathematical tools and definitions from the areas of image processing and information theory that we use throughout our work.

3.1 Texture

There is no formal definition for *texture*. Nevertheless, there exists some consensus on the perception of texture. Among the different informal definitions found in the literature, the words 'repetition', 'pattern', 'periodicity' come up regularly. Therefore, we may say that the concept of texture is somewhat subjective. To illustrate different approaches to the problem, we present below some of the definitions found in literature.

Gonzalez and Woods [20] state that "*The three principal approaches used in image processing to describe the texture of a region are statistical, structural and spectral. Statistical approaches yield characterizations of textures as smooth, coarse, grainy, and so on. Structural techniques deal with the arrangement of image primitives, such as the description of texture based on regularly spaced parallel lines. Spectral techniques are based on the properties of the Fourier spectrum and are used primarily to detect global periodicity in an image by identifying high energy, narrow peaks in the spectrum"*

Jain [25] asserts in his book that "*The term texture generally refers to the repetition of basic texture elements called texels. The texel contains several pixels, whose placement could be periodic, quasi-periodic or random. Natural textures are generally random, whereas artificial textures are often deterministic or periodic. Texture may be coarse, fine, smooth, granulated, rippled, regular, irregular, or linear"*

Efros and Leung [15] shortly define texture as "*some visual pattern on an infinite 2-D plane which, at some scale, has a stationary distribution*"

Even though the definitions presented here are all due to people working in the image processing field, the term texture is not bound exclusively to image textures. Think for example in the chirp of a bird, the buzz of a crowded street, or the ring of a telephone.

Those can be thought of sound textures. We might as well talk about video textures [2] (i.e. an erupting volcano).

Figure F1[1] shows some examples of image textures taken from the Brodatz texture database [5] and the Oulu texture database [36].

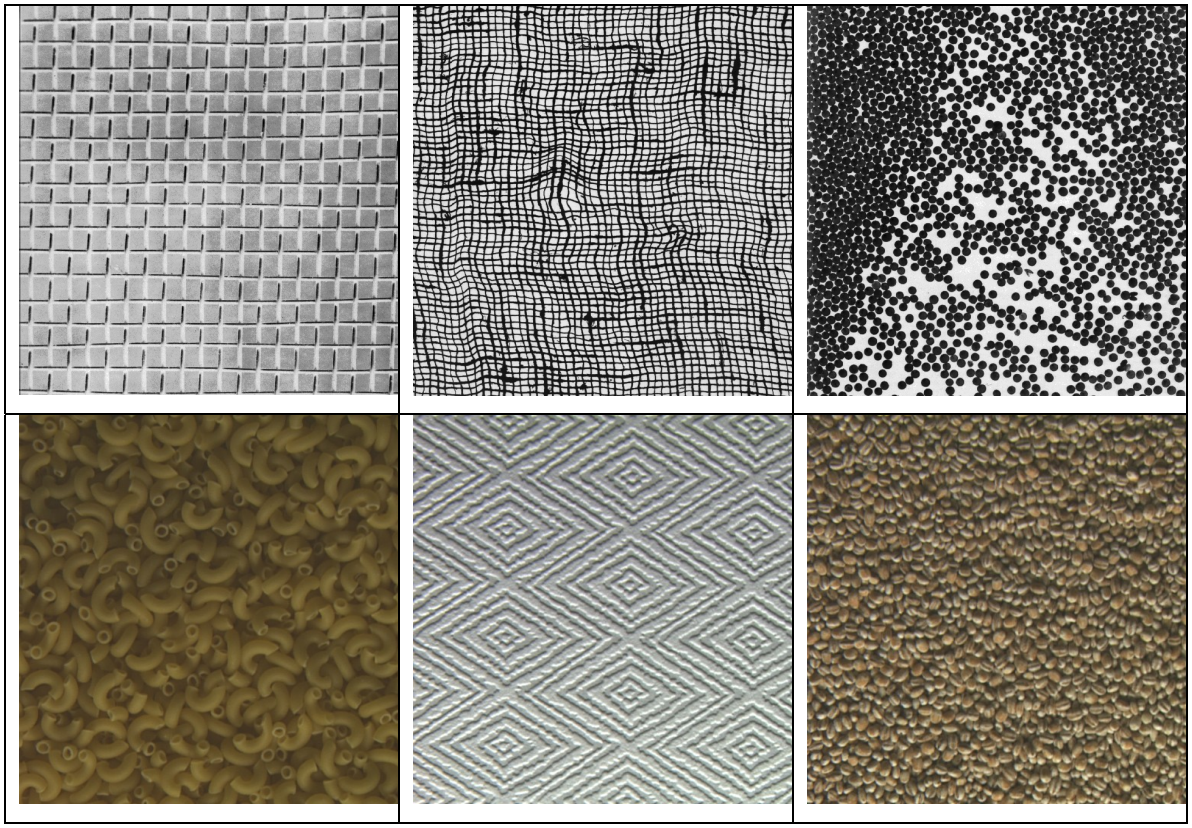

Figure (F1): Texture examples. The upper images were taken from the Brodatz texture database. The lower images were taken from the Oulu texture database.

[1] Figures which show algorithms will be labeled with the letter *A*, and figures with graphics and images will be labeled with the letter *F*.

3.2 Source models

Here we briefly describe some source models which are mentioned in this work.

We denote by x_i^j the sequence $x_i x_{i+1} \ldots x_j$, where $i \le j$, and the symbols in the sequence belong to some finite alphabet Λ of cardinality α; we sometimes omit the subscript *i* in x_i^j when *i*=1. Let's suppose we have a string or sequence x^n. A *source model Q* is a probability assignment function which gives, for every possible string, a probability of occurrence.

$$Q(x^n) = p,\ 0 \le p \le 1,\ \text{and}\ \sum_{\{x^n\}} Q(x^n) = 1$$

The *entropy* $H(X)$ of a discrete random variable *X* of alphabet Λ is a measure of the uncertainty of X, and is defined as [10]

$$H(X) = -\sum_{x \in \Lambda} p(x) \log p(x)$$

The entropy is maximized when the probability is uniformly distributed. In other words,

$$H(X) \le \log \alpha \ \text{and}\ H(X) = \log \alpha \ \text{if and only if}\ p(a) = \frac{1}{\alpha}\ \forall a \in \Lambda$$

When studying signals, it is useful to select a proper source model for the data being treated, i.e. one which captures the features of interest for the application at hand. As mentioned, textures are often characterized in terms of their local statistical properties and stationarity. Therefore, we will focus on finite memory statistical models as a useful (albeit not perfect) tool in the study of textures. We next define these models more formally.

Suppose we have an alphabet Λ of α symbols. The simplest source model is the fully parameterized *memoryless source.* It is fully characterized by a vector $\Theta = \left\{ \Theta_i, 1 \le i \le \alpha, \text{ where } \sum_{i=1}^{\alpha} \Theta_i = 1 \right\}$ of probabilities of occurrence of every symbol. In this case, the probability assignment function is

$$Q(x^n) = \prod_{i=1}^{n} \Theta(x_i)$$

where $\Theta(x_i)$ denotes the probability of occurrence of symbol x_i.

We denote the *empirical distribution* of x^n by $\hat{Q}_{x^n}$, defined as

$$\hat{Q}_{x^n}(a) = \frac{freq(a, x^n)}{n}, a \in \Lambda$$

where $freq(a, x^n)$ is the number of occurrences of symbol a in x^n. The empirical distribution $\hat{Q}_{x^n}$ defines a memoryless source, which can be used to assign probabilities to arbitrary sequences, including, in particular, x^n. In fact, $\hat{Q}_{x^n}$ is the source that assigns the highest probability to x^n among all memoryless sources over Λ, and is known as the *maximum likelihood estimator* of a memoryless source for x^n [31]. The *empirical entropy* of x^n is then defined as

$$\hat{H}(x^n) = -\sum_{i=1}^{n} \log \hat{Q}_{x^n}(x_i)$$

We now extend our discussion to *Markov sources*.

A memoryless source is a 0-order Markov source. In a *first order Markov source* the probability distribution of a symbol depends on the preceding symbol. Thus,

$$Q(x_i \mid x_1^{i-1}) = Q(x_i | x_{i-1}), 2 < i < n \text{ , and } Q(x_1) = Q_{init}(x_1),$$

where the conditional distributions $Q(\cdot|\cdot)$ and the initial distribution $Q_{init}(\cdot)$ provide the parameters defining the source. The probability assignment function for strings is

$$Q(x^n) = Q_{init}(x_1) \prod_{i=2}^{n} Q(x_i \mid x_{i-1})$$

Finally, in a *k^{th}-order Markov source* the probability of appearance of a symbol is conditioned on the last k symbols. Thus the probability assignment function is

$$Q(x^n) = Q_{init}(x_1^k) \prod_{i=k+1}^{n} Q(x_i | x_{i-k}^{i-1})$$

For simplicity, we shall write $Q(x_1^k) = Q_{init}(x_1^k)$.

A stochastic process is said to be s*tationary* when its probability distribution does not change over time and *ergodic* when its time averages equal the ensemble averages (for a more formal treatment of these properties, see, e.g. [37]). Markov sources as defined above are ergodic, and, under an appropriate choice of the initial distribution

$Q_{init}(\cdot)$, stationary. We will suppose for simplicity that the initial distribution Q_{init} equals the stationary probability distribution of the source. The family of stationary ergodic sources, however, is much richer than that of the Markov processes, although any stationary ergodic source can be approximated, in the limit, by stationary ergodic Markov sources of increasing order [10].

We define the *k^{th}-order conditional empirical distribution* of x^n, $\hat{Q}^{(k)}_{x^n}$, by

$$\hat{Q}^{(k)}_{x^n}\left(a\middle|u^k\right)=\frac{freq\left(u^k a,x^n\right)}{freq\left(u^k,x^{n-1}\right)},a\in\Lambda,u^k\in\Lambda^k$$

where $freq\left(u^k,x^n\right)$ is the number of occurrences of u^k in x^n and initial condition $\hat{Q}^{(k)}_{x^n}\left(x^k\right)=1$. The k^{th}-order conditional empirical is the maximum-likelihood estimator of a k^{th}-order probability model for x^n.

Therefore, we define the *k^{th}-order conditional empirical entropy* of x^{n} as

$$\hat{H}_k\left(x^n\right)=-\sum_{i=k+1}^{n}\log\hat{Q}^{(k)}_{x^n}\left(x_i\middle|x_{i-k-1}^{i-1}\right)$$

3.3 Lempel-Ziv universal lossless data compression algorithms

Suppose you have some data produced from an unknown source. A *universal compression scheme* for some given family of source models is an algorithm which asymptotically (in the size of the input data) compresses the input data down to the entropy of the source without any a priori information of the source distribution (except that the source belongs to some class). Whenever we talk about a universal compression algorithm we have to specify for which class of sources it is universal.

Several universal compression algorithms have been developed in the last 30 years. We will focus our attention to the Lempel-Ziv and related family of algorithms, and, in particular, the LZ78 variant [27]. From an information-theoretic point of view these algorithms are proven to compress asymptotically to the entropy of the source [10], for any stationary ergodic source. Detailed information about lossless data compression, and particularly about universal compression algorithms can be found in [10], [13], and [31].

We first give a brief description of the LZ77 algorithm. The algorithm was first made public in 1977 [29].

The idea of the algorithm is to process the source replacing already seen substrings (within a window of prescribed length) with pointers to their previous occurrences.

Let's assume we have an alphabet Λ of cardinality α, and an input string $\{x^n = x_1 \cdots x_n : x_i \in \Lambda\}$ of length n. The algorithm uses two parameters, L_s and L_n as shown in figure F2, which define the size of the look-ahead window and the past window. The *look-ahead window* is a list of L_s symbols starting at the current symbol in the input string (i.e. it always contains the next L_s symbols to be processed). Likewise, the *past window* is a list of L_n symbols which always contains the last L_n processed symbols. The algorithm uses a *working window (W)* which is constructed by appending the past window and the look-ahead window. Thus, the whole working window *W* covers $L_s + L_n$ symbols. The LZ77 algorithm processes the input string as follows. Beginning at the

current symbol location x_i, it will search in W for the longest prefix match. The *longest prefix match* is the largest substring (possibly empty) in W which both starts in the past window, and matches part of the input string starting at the current symbol x_i in the look-ahead window. The algorithm will then output a triplet *(P,L,S)* where *P* and *L* are both integers, and *S* is a symbol. P will denote the pointer location in the past window for the first matching symbol, L will indicate the length of the longest match, and S will denote the first non-matching symbol of the source. Note that P must be a number between 0 and L_n-1, and L between 0 and L_s, thus for every triplet we can measure the code output size. The actual values chosen for L_n and L_s may have impact on the algorithm performance. Greater values will allow finding longer matches, but it will also require more bits to encode the triplets.

After outputting the triplet, the window W slides L+1 elements (i.e. the number of processed symbols) so that the first symbol in the look-ahead window is the following symbol to be processed (x_{i+L+1}) . We will use L=0 in case of not finding any match in the past symbols buffer beginning at the first symbol of the look-ahead buffer. We repeat the same parsing rule until the entire input string is processed. Due to the fact that we work with finite-length input strings, we may need to either know the string length in advance or add a special symbol(marker) to denote the 'end of string' if we were working with streamed data sources.

The decoding procedure is very straightforward. Let's assume that we will output the decoded data onto an output buffer *O*. Then, for every triplet *(P,L,S)* we will append $\{O_{i-Ln+p+j} : 0 < j \le L\}$ and *S* to the output buffer. After that we set i=i+L+1 to readjust the output buffer pointer. Figure F2 shows a simple example of the encoding procedure.

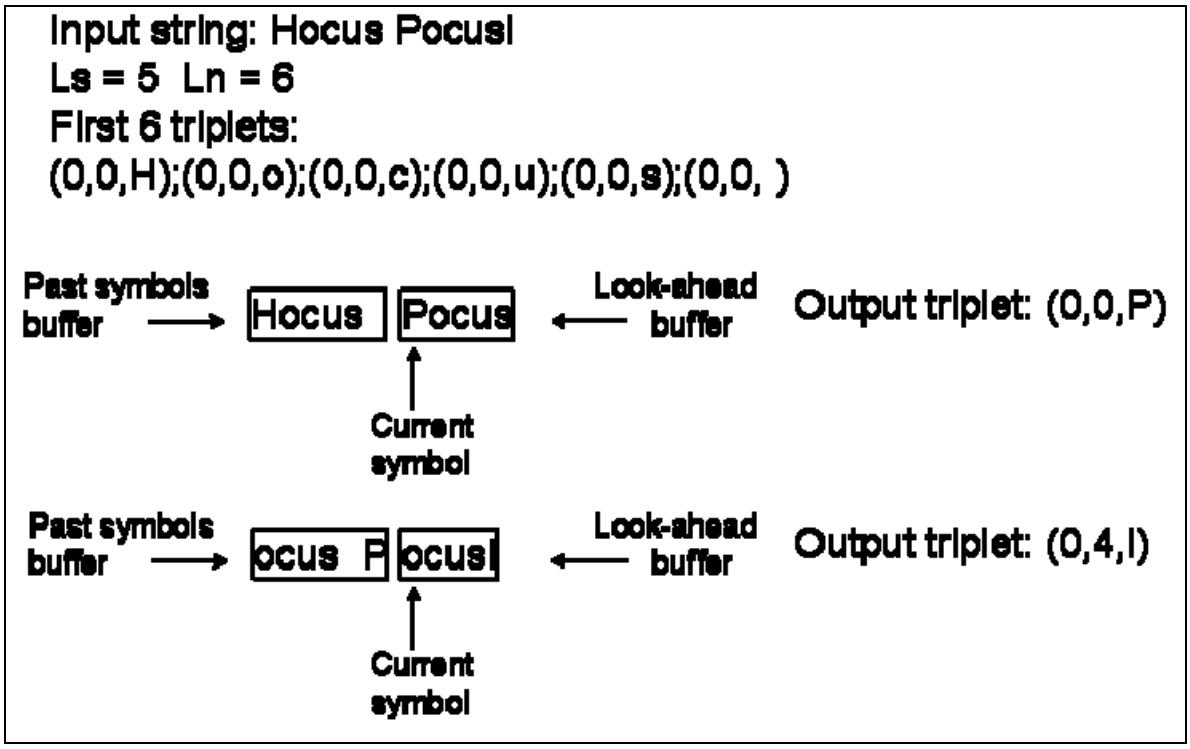

Figure (F2): Lempel-Ziv(LZ77) encoding example

We will now shift our focus to the LZ78 algorithm, which provides one of the main devices in our simulation scheme. The LZ78 *incremental parsing algorithm* [27] is also very straightforward. Instead of working with a sliding window, this algorithm uses a dictionary (usually represented by a tree) to point to previously processed substrings, which will be called *phrases*. The input parameters to the LZ78 algorithm are the alphabet size and the maximum dictionary size. As in the explanation for the LZ77 algorithm, let's suppose that we have an input string x^n of length n. The incremental parsing algorithm will split the input string x^n in p phrases such that

$$x^n = \lambda.s_1.s_2 \cdots s_p.t \text{ where } \begin{cases} s_i = s_j.S,\ S \in \Lambda \text{ for some } j < i \\ s_j.S \neq s_r \forall r < i \end{cases}$$

and t is a *tail* phrase (one that already appeared in the parsing). In this expression, λ denotes the empty-string marker and S any symbol of the alphabet Λ. For convenience, we denote $s_0 = \lambda$. Note that all phrases (except for the tail phrase) are distinct. Moreover, for a given s_i, there exists one and only one s_j such that $s_i = s_j.S$. This fact justify the

representation of the dictionary of phrases as a suffix tree with node s_i being a child of s_j following an arc labeled by S.

To encode an input string, we start the incremental parsing algorithm with an empty dictionary (just containing an empty-string maker λ), and set the current symbol x_i at the first symbol of the input string. We look in the dictionary for the longest phrase that matches the input string beginning at x_i. We denote h to the length of that phrase. Then we output a pair (P,S), where P is the location of the longest match and S is the first unmatched symbol (i.e. x_{i+h}). Next, we enlarge the dictionary adding the new phrase consisting of the concatenation of longest match found and S, i.e. x_i^{i+h}. After that we set the current symbol to be the next unprocessed symbol in the input string and repeat the procedure. The size of the output pairs grows due to the fact that the dictionary gets bigger with each round of the algorithm. Some variants of the algorithm limits the size of the dictionary (purging the oldest ones) to keep the size of the output pairs bounded.

To decode the set of output pairs back to the original string, we start again with an empty dictionary (just containing the λ empty-string marker) and process in order the input pairs. Each pair will decode into a new substring, that will be concatenated to the output string and a new item will be added to the dictionary.

As stated before, the incremental parsing algorithm is extensively used in this work. For the purpose of efficiency, we model the dictionary with an α-ary tree (α being the alphabet size of the input string). Figure A1 describes the incremental parsing algorithm using this data structure. Let x^n be a one dimensional input sequence (string) with alphabet Λ of cardinality α. We use a variable i to point to the current symbol (S), and t to point to the current node. To encode the input sequence, we set t at the root (λ) of the tree (step 1) and begin consuming data from the input string. We traverse the tree, node by node, moving to the child node with the arc labeled with the current input symbol until we get to a node that either has no children (step 3.a) or in which no outgoing arc matches the current input symbol (step 3.b). At that point we create a child node labeled with the current phrase (i.e. the phrase of the parent node plus the current symbol), and we label the arc to the child with the current symbol. Then we set our location to the root and continue parsing the remaining data. Every time a new leaf is added to the tree, a new phrase labeled by the path from the root to this leaf is generated. By construction, every

new phrase is the concatenation of an already processed phrase with the last consumed symbol. The tail depth, which is the depth of the current standing node when the algorithm finishes (step 2 and 4), is an exception to this rule. It points to a node in the tree which already existed; and it is due to the fact that we work with finite length input sequences. Figure F3 is an example of the incremental parsing algorithm showing the output pairs and the resulting parsing tree.

In this section we have presented the incremental parsing algorithm along with some brief explanations on the encoding and decoding procedures used in the LZ77 and LZ78 universal lossless data compression algorithms.

Input: input sequence (x^n) of length n

Outputs: LZ tree(**T**), tail depth(**d**)

1. Set your location **t** at the root(λ) of the tree.
2. Check if there is more data from the input sequence. If there are no more symbols (i.e. i=n) go to step 4.
3. Get the next symbol **S = x_i**, set **i = i+1**. Look for all the outgoing arcs of **t** to see if one is labeled with **S**.
   a. If there is a match, set **t** to the child pointed by the matching arc, and go to step 2.
   b. Otherwise create a child node **v** labeled with the concatenation of the phrase in **t** and **S**. Label the arc between **t** and **v** with **S**. Go to step 1.
4. Return the tree (**T**) and the integer (**d**) of depth of the last node visited by the algorithm (the *tail depth*)

Figure (A1): Lempel-Ziv(LZ78) incremental parsing algorithm

Input string x^n = 101101010110011

Incremental parsing:

1,0,11,01,010,110,011

Encoding output pairs:

(0,'1') (0,'0') (1,'1') (2,'1')

(4,'0') (3,'0') (4,'1')

Tail: λ (tail depth=0)

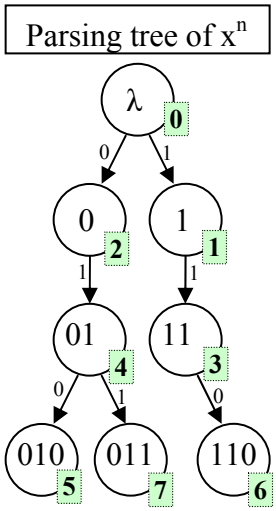

Figure (F3): Lempel-Ziv(LZ78) encoding example. The left side shows an example input string, the result from the incremental parsing and the encoding output pairs resulting from it. The right side shows the parsing tree derived from the input string. The numbers inside the squares at the right of each node represent the time of appearance of each phrase in the incremental parsing.

3.4 Plane Filling Curves

Our work makes strong use of the original Lempel and Ziv incremental parsing algorithm, which processes one-dimensional input sequences. On the other hand, image textures are two-dimensional data sequences. Thus, we need to find some means of traversing the image to feed the parsing algorithm. It is interesting to note that in [28] Lempel and Ziv already treated this problem in order to compress two-dimensional data. The method used to visit the pixels of the image is regarded as the *scan order* of the image. There are many ways of doing this, each one having its pros and cons. We will briefly discuss some of them here.

The simplest approach, called *raster scan*, is to traverse the image row by row (see left image in fig F4). This approach is very easy to implement. Its main advantage is that it does not need to have the full image available at any time. For this reason, it is mainly used in memory-constrained scenarios (i.e. JPEG-LS). Its main drawback with respect to our work will be evidenced after we develop the concept of space filling curves.

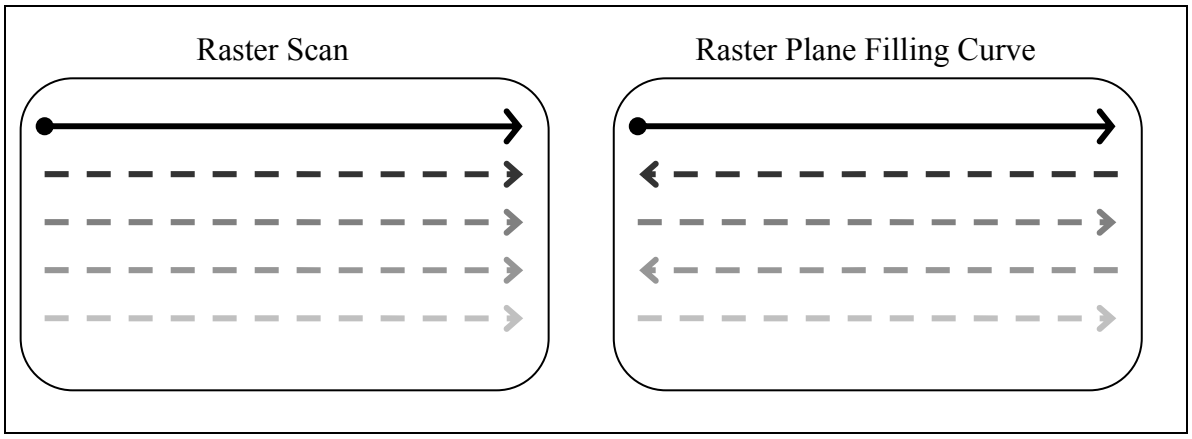

Figure (F4): Raster scan and Raster plane filling curve. The left side shows the way the Raster scan traverses the image, row by row from left to right. The right side shows a modified raster scan that conforms with the requirements to be a plane filling curve. It traverses the image also row by row, but interchanging the orientation of the scan after finishing a line.

In 1890 Giuseppe Peano introduced the notion of *space filling curves*, which are continuous mappings of the reals(R) onto a multidimensional unit hypercube [38].

More formally [21], we denote *d(.)* as the regular Euclidean distance and $[N]=\{1,\cdots,N\}$. A discrete *m*-dimensional space filling curve $\boldsymbol{C}$ of length N^m is defined as a bijective mapping:

$$C:[N^m]\rightarrow[N]^m \text{ such that } d(C(i),C(i+1))=1\ \forall i\in[N^m-1]$$

The special case of a second order hypercube is called a *plane filling curve*. The application of a plane filling curve over an image sets a unique index (also referred as the *time* variable) over each pixel in the image. The resulting mapping between the time variable and the image coordinates is bijective (i.e. given a time index one can obtain a pixel coordinate and the other way around). There are a number of plane filling curves mentioned in literature, most notably the Hilbert curve, the Sierpinksi curve and the Peano group of curves. The application of such curves to the scanning of images is referred to as the Hilbert scan, the Sierpinksi scan and the Peano scan respectively (see fig F5). A slight modification of the raster scan yields another plane filling curve (fig F4). These plane filling curves usually have some constraints on the size of the image. For example, in the Hilbert scan the image dimensions need to be equal and a power of two on all directions. However, more complex plane filling curves can be defined which do not suffer from these constraints (see for example [38]).

As images are often modeled as two dimensional, finite memory random processes, we would like to use scan orders which have the property that two points close in the scan order are also spatially close in the image. This property is indeed important because we want to model the sequences produced by the scan of the image as finite Markov random processes to feed the incremental parsing algorithm. Therefore we have to focus on the metric properties of plane filling curves.

To this extent Gotsman and Lindenbaum [21] introduce the following locality measure, $L(C)=\sum_{i,j\in\{0..N^m\},i<j}\frac{|i-j|}{d(C(i),C(j))}$ which is used as a foundation to prove that for any space filling curve there will be at least two points in *[N]*^{m} that are spatially close with respect to the Euclidean distance yet they are distant along *C*. In spite of this, one

can build space filling curves that on average perform very well (for example, the Hilbert curve). Moreover, we are more interested to know what happens with the converse (i.e. up to which extent two points that are close within the curve will be spatially apart).

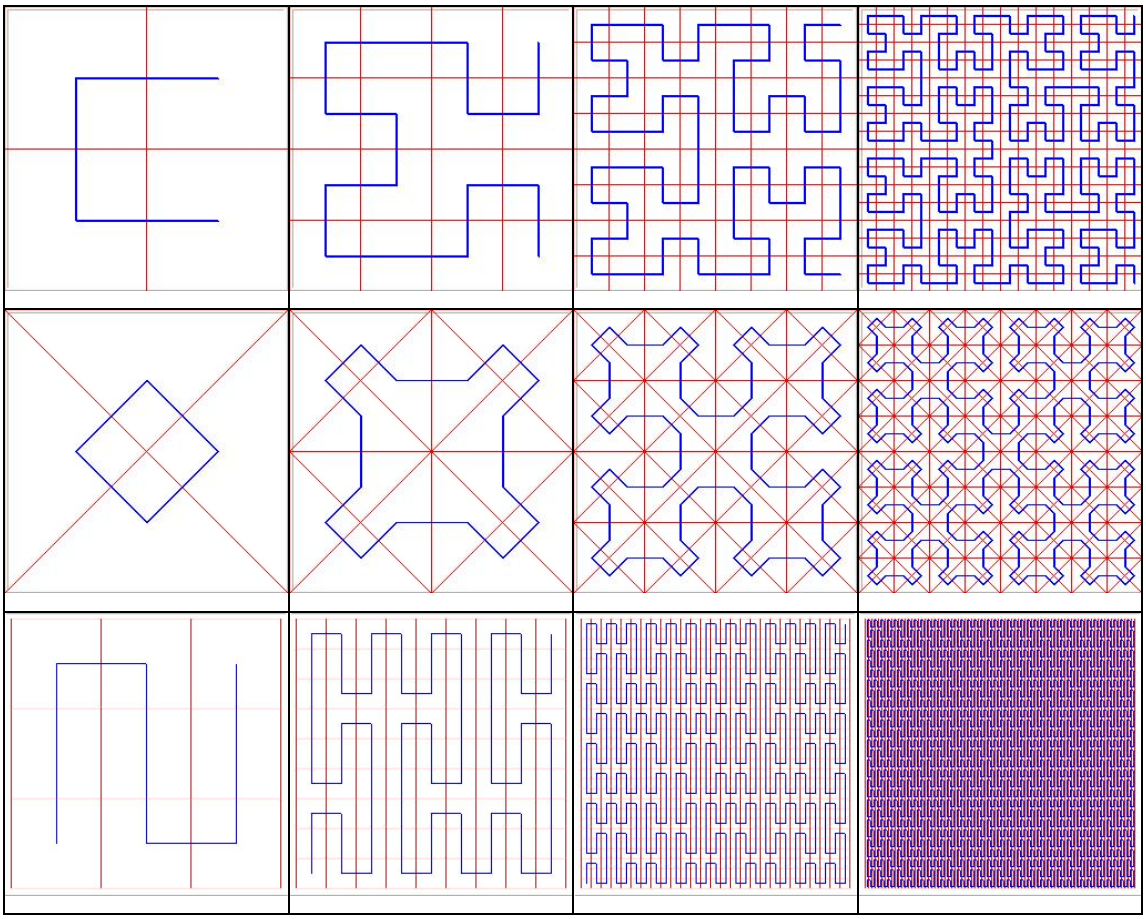

Figure (F5): Construction the Hilbert, Sierpinksi and Peano scans. The first and second row shows the construction of the Hilbert and Sierpinksi scans of sizes 2^nx2^n, for n=1..4. The third row shows the construction of the Peano scan of sizes 3^nx3^n, for n=1..4

Gotsman and Lindenbaum [21] introduced two new measures of locality and found lower and upper bounds for them. They also proved that under these measures the raster space filling curve performs extremely badly (for any two end points of a scan line, both the Euclidean distance and the distance along the curve are *N*-1). Furthermore, they prove that the Hilbert curve is close to optimal and dramatically outperforms the raster space filling curve with respect to this locality measure.

Therefore, although we have tried other order scans like the raster scan, or the Sierpinksi scan, we will mainly use the Hilbert scan.

The Sierpinksi, Peano, and Hilbert plane filling curve all share the same principle of construction. For any square image N^2xN^2 one can construct these curves recursively using rotated forms of some more basic curves. These constructions are shown in figure F5. The first row shows the construction of the Hilbert scan for images with sizes 2x2, 4x4, 16x16 and 64x64. For images with size 4x4, the scan is constructed by means of copying a rotated version of the 2x2 scan. Likewise, to construct the 16x16 scan, we use rotated versions of the 4x4 scan. This idea can be applied recursively to construct the Hilbert scan of any image of size 2^Nx2^N. The Sierpinksi of size scan 2^Nx2^N is also constructed by applying recursively the Sierpinksi scan 2^{N-1}x2^{N-1}. The Peano scan can be constructed with the same principle, although in this case the size of the image side should be a power of 3. This will present challenges in our work, which we will need to address.

Lets say we have an image of size 2^Nx2^N, and an array $H\left[0\cdots 2^{2N}-1\right]$ in which we want to copy the contents of the image. We will use the Hilbert scan for this purpose. Recall that if we traverse the image with a Hilber scan, for every pixel in the image we have an index (the time variable *t*) which maps the pixel with the "time" in which the scan passed over it. Suppose that we apply some calculations over the resulting array and then *revert back* the Hilbert scan (i.e. we traverse the image again, but this time copying the contents of H at position *t* to the corresponding location in the image). If the calculations applied to H were a shift of the contents (e.g. H[*i*] = H[*i*+*m*]) the resulting image after reverting back the Hilbert scan would show portions of the original image with rotations of multiples of 90°. Furthermore, due to the recursive nature of the construction of the Hilbert scan, these rotations may be seen at different scales depending on the magnitude of the shift. Figure F6 illustrates this problem. The first image shows a picture with some horizontal lines. In the remaining pictures we applied the calculations mentioned above with some (increasing) values of *m.* The visual consequence of these operations is that in some areas of the image, the horizontal lines become vertical lines. Moreover, for some delay values the reconstructed images doesn't even resemble lines anymore (see for example the last image in F6). In the following sections we elaborate more on this problem and on measures taken to alleviate its visual effects.

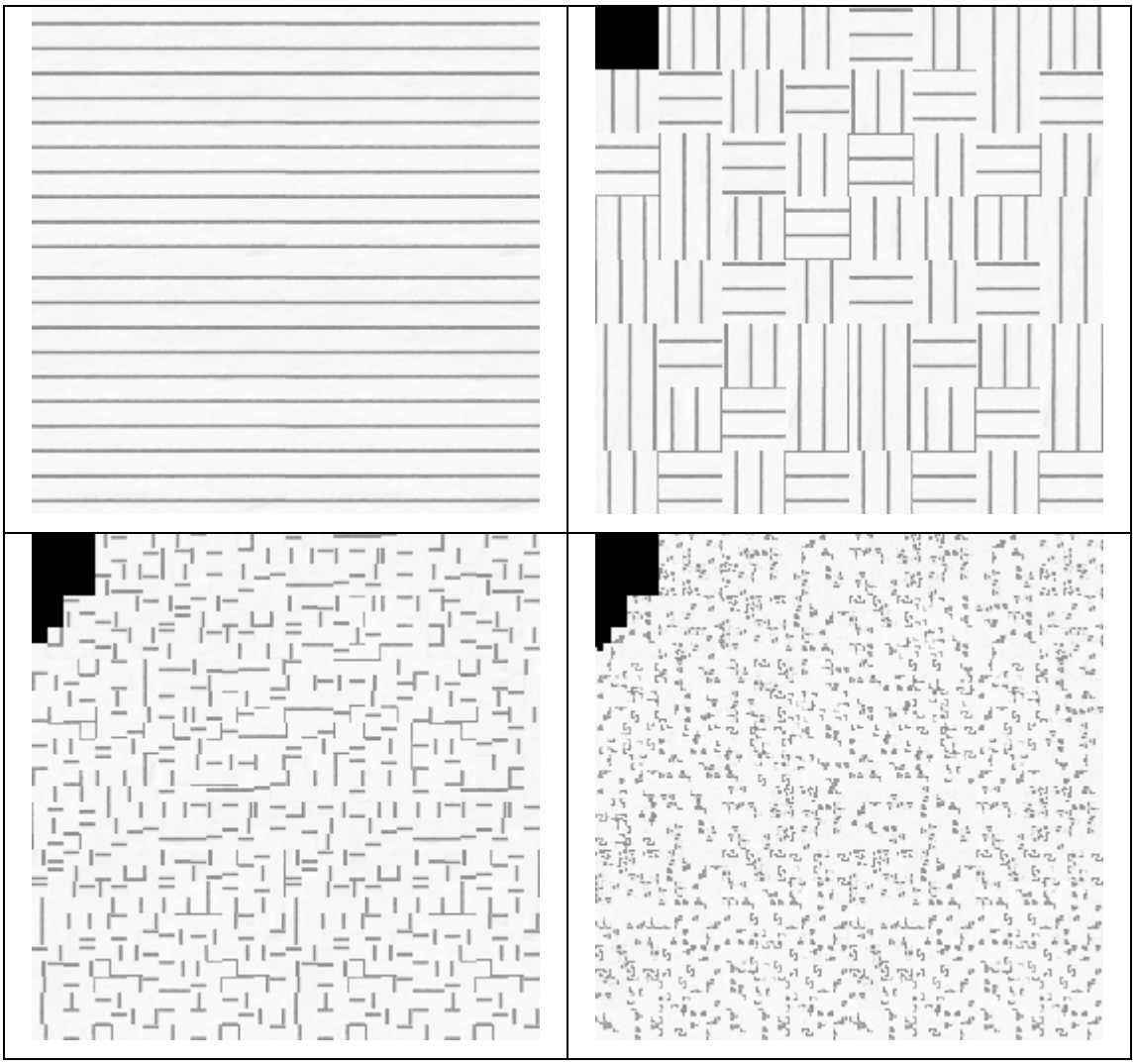

Figure (F6): Examples of the issues when reverting back a Hilbert scan after applying calculations. The top left picture shows the input image. The remaining pictures show the resulting image after applying the calculation H[i]=H[i+m] for different values of m.

3.5 Universal Type Classes

In this section we give an introduction to universal type classes and universal simulation. However, we first introduce the conventional method of types [11, 12].

We consider an alphabet Λ of cardinality α and use $x^n=x_1x_2...x_n$ to denote a sequence of length n ($x^n \in \Lambda^n$). Consider a parametric family $\mathcal{P} = \{P_\Theta\}$ of distributions on Λ^n parametrized by a vector Θ of k real-valued parameters:

$$P_\Theta : \Lambda^n \to [0,1], \ \Theta \in \mathcal{D} \subseteq \mathcal{R}^k$$

Then, for a given sequence x^n, the *type class* of x^n with respect to $\mathcal{P}$ is defined as the set [11]

$$\mathcal{T}_{x^n}^{P} = \{y^n \in \Lambda^n : P_\Theta(y^n) = P_\Theta(x^n) \ \forall \ \Theta \in \mathcal{D}\}$$

Type classes will be characterized by the combinatorial structure of x^n, and are related to the notion of *sufficient statistics*.

For fully parametrized memoryless distributions we need to know the α-1 free parameters that determine the probability of occurrence of every symbol of the alphabet. For example, consider the case of binary sequences. Thus, in this case we have α=2 and a parameter θ ($0 \le \theta \le 1$) such that $P_\theta(x_i = 1) = \theta$, i.e., this is the family of *Bernoulli* processes, *Bernoulli*(θ).

$\Lambda = \{0,1\}$, $\mathcal{P} = \{Bernoulli \ (\theta) | 0 \le \theta \le 1\}$, $P_\theta(x^n) = \theta^m(1-\theta)^{n-m}$, where m is the number of ones in x^n. In this case, it is readily verified that x^n and y^n are of the same type if and only if they have the same number of ones. For example, '1100110010' and '1010101010' are from the same type, but '1111000011' is not.

The number of type classes for memoryless distributions is polynomial in n [10]. There are at most α free parameters needed to fully parameterize a memoryless distribution with alphabet Λ of cardinality α, and each parameter can have a value among n+1 possible values. Therefore, the number of type classes of input strings of size n $(\mathcal{P}_n)$ can be upper bounded by $|\mathcal{P}_n| \le (n+1)^\alpha$. We now give bounds on the size of a type class, which are proved in [10, 11]:

$\frac{1}{(n+1)^{\alpha}} 2^{nH(P)} \le \left|\mathcal{T}_{x^n}^{P}\right| \le 2^{nH(P)}$, where $H(\cdot)$ denotes the binary entropy function $H(x) = -x\log_2 x - (1-x)\log_2(1-x)$. The fact that the number of classes is subexponential is indeed useful, for example for enumerative coding [9]. In this type of coding, every string x^n belonging to a specific type class $\mathcal{T}_{x^n}^{P}$ can be assigned a codeword of length $l(x^n) = \lceil \log_2 |\mathcal{P}_n| \rceil + \left\lfloor \log_2 \left|\mathcal{T}_{x^n}^{P}\right| \right\rfloor$ as follows: Use the first $\lceil \log_2 |\mathcal{P}_n| \rceil$ bits to describe the type class, and the remaining $\left\lfloor \log_2 \left|\mathcal{T}_{x^n}^{P}\right| \right\rfloor$ bits to identify x^n from within the class. Recall that for any model Q the "ideal codelength" of x^n under this model is $\left|-\log_2 Q(x^n)\right|$. Therefore, $l(x^n)$ will exceed this "ideal codelength" by less than $\lceil \log_2 |\mathcal{P}_n| \rceil$ for each source model in the family. The proportion of excess bits with respect to $l(x^n)$ decreases as n increases. Thus, the method of types can be used as the basis of universal source coding for the family of fully parametrized memoryless models.

Recall from the introduction of Markov sources in section 3.2 that the probability distribution of a symbol for a first order Markov source depends only on the preceding symbol. For simplicity we assume that the initial distribution equals the stationary probability distribution. Thus, for first order Markov distributions, type classes of x^n are determined by the empirical joint distribution of order 2 (i.e., there will be $\alpha(\alpha-1)$ parameters that specify the probability of occurrence of every symbol given the preceding symbol).

More generally, for finite memory Markov distributions of a given order k, if we assume once more that the initial distribution equals the stationary probability distribution, the type classes are determined by the empirical joint distribution of order $k+1$. As with the case of memoryless distributions, the growth rate in the number of type classes is subexponential. We can model Markov distributions of order k with *finite-state types* [11] with α^k states, each state having α numbers with a value among n+1 possible values. Therefore, the number of classes is upper bounded by $\left|\mathcal{P}_n^k\right| \le (n+1)^{\alpha\alpha^k}$. A lower bound was derived in [49], $\left|\mathcal{P}_n^k\right| \le Cn^{(\alpha-1)\alpha^k}$, where C is a constant which depends on the finite-state machine defining the class. The size of a class is approximately [11]

$\left|\mathcal{T}_{x^n}^{P}\right| \approx 2^{n\hat{H}_k(x^n)}$. As with the case of the method of types for memoryless sources, we can use these properties for universal source coding for the family of Markov models of order *k*. For a thorough review on the classical method of type, see e.g. [10, 11].

We now introduce a generalization of the method of types, introduced in [43] and [42]. Recall from the introduction of the Lempel-Ziv universal lossless compression schemes (section 3.3) that as a result of applying the incremental parsing algorithm (LZ78) we get a parsing of the input string. Let $T_{x^n} = \{s_1, s_2, \cdots, s_p\}$ denote the set of phrases of that parsing. Then, we define the *universal type class* (UTC) of x^n as the set [42]:

$$\mathcal{U}_{x^n} = \{y^n \in \Lambda^n : T_{y^n} = T_{x^n}\}$$

There is a bijective mapping between the set T_x^n and the tree resulting from the application of the incremental parsing algorithm defined in figure A1, thus we will refer to it as the *parsing tree of* x^n. Note that every sequence from the UTC has the same parsing tree. The parsing tree of x^n is an α-ary tree rooted at λ. Every node of the tree represents one phrase of the incremental parsing, and for every phrase s_i which is an extension of s_j with the symbol $S \in \Lambda$ added as a suffix (j>i) there will exist a branch labeled S going from the node s_i to the node s_j. Throughout this work we will make extensively use of these trees and its connection with UTCs. The definition of the UTC given in [42] states that all the sequences in a class must have the same length. Thus, even if two sequences span the same parsing tree, they need to be of the same length(the tail size has to be the same) to be part of the same UTC. When the tail size of x^n is zero, the universal type class is called the *natural UTC* of T_x^n.

Given the UTC defined by the sequence x^n, every other sequence y^n contained in the same UTC is the concatenation of a permutation of the set of phrases spanned by the incremental parsing algorithm over x^n. On the other hand, not every permutation of the set of phrases leads to a sequence contained in the same UTC, as for every phrase $s_i=s_j.S$, it is mandatory for s_j to appear before s_i. In other words, a valid permutation requires that for every phrase, the node of the parsing tree associated to it has to be reachable (i.e., all the parent nodes already been visited) when that phrase is selected. Figure F7 shows an example of sequences contained and not contained in the same UTC.

x^n = 101101010110011

Incremental parsing output:

1,0,11,01,010,110,011

y^n = 111011001011010

Incremental parsing output:

1,11,0,110,01,011,010

z^n = 110011010110110

Incremental parsing output:

1,10,0,11,01,011,0110

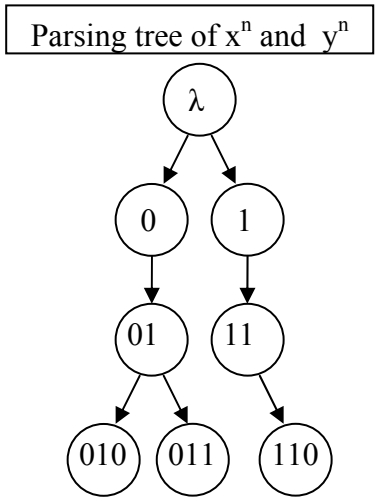

Figure (F7): UTC example. The figure of the left shows three binary input sequences (x^n, y^n, z^n) of length=15 and the incremental parsing of each of them. x^n and y^n are from the same UTC while z^n is not. The right figure shows the parsing tree of x^n and y^n (which are the same).

Recall that in the conventional method of types any pair of sequences x^n, y^n of the same class are assigned identical probabilities by any model in the class defining the types. This means that they are statistically indistinguishable by any model from the class. Lets define the *variational distance* $\|P-Q\|_1 = \sum_{w\in\Omega} |P(w)-Q(w)|$, where Ω denotes the set of possible *k-tuples* from Λ^k. In the case of the UTCs defined above, it can be shown [42] that if $y^n \in \mathcal{U}_{x^n}$, then for every integer k ≥1, the variational distance between the k^{th} order empirical distributions of x^n and y^n vanishes as $n \to \infty$. Thus, two sequences of the same universal type class are also statistically indistinguishable (in the limit) by finite memory models of any order. As with the case of conventional types, with UTCs it is also desired that the growth rate in the number of type classes be subexponential. Using the main result of [41], this is indeed proven in [42]. The number of universal types for sequences of length n is $\mathcal{N}_n = \alpha^{\left(\frac{\alpha.H(\alpha^{-1}).n}{\log(n)}.(1+o(1))\right)}$.

For the purpose of simulating individual sequences from a UTC, we need to develop some means to select sequences uniformly at random over a defined UTC. When $n \to \infty$,the simulated sequences will be statistically indistinguishable from the input

sequence (the one used to define the UTC), and yet they will be taken from the broadest pool of sequences which fulfill the statistical constraint (i.e. there will be maximum entropy in the selection of the sequence). Such sampling algorithms are presented in [42].

The algorithm shown in figure fig A2 samples a sequence uniformly at random from the UTC given a parsing tree of x^n and a tail size. We assume that we have an α-ary tree *T* (and a tail size *d*) built according to the Lempel-Ziv incremental parsing algorithm depicted in figure A1. We use a variable *t* to point to a node of T, and denote *ta* to refer the child node *a* of *t*. We also denote *cp(t)* to the number of nodes below *t* plus 1. Every node in T is decorated with a boolean which indicates if it has been visited at least once and an integer *U(t)* which contain the number of nodes below *t* which have never been visited by the algorithm yet. Whenever a node *t* is visited, we decrement *U(t)* in one unit.

Inputs: LZ tree(***T***), tail depth(***d***)

Output: Randomly sampled sequence

1. Walk through all the nodes of **T** marking them as unused, and for each node ***t*** set U(**t**) = c_p(**t**); U(**t**) denotes the number of nodes below *t* which have never been visited yet, and c_p(**t**) denotes the number of nodes below *t* plus 1. Mark the root (**λ**) of **T** as used and set U(**λ**) = U(**λ**)-1.
2. Set **t** at the root(**λ**). If U(**t**) = 0 go to step 5.
3. If **t** is marked as unused output the phrase associated to **t**, mark **t** as used, set U(**t**) = U(**t**)-1, and go to step 2
4. Draw randomly a symbol ***a*** from Λ with distribution $\text{Prob}(a=b)=\frac{U(tb)}{U(t)}, b\in\Lambda$, set U(**t**) = U(**t**)-1, set ***t*** = ***ta*** and go to step 3.
5. Pick uniformly a node ***t*** of depth **d** from **T** and output it as the tail.

Figure (A2): Random sampling from universal type

The algorithm works as follows. In step 1 we initialize U(t) for the entire tree and mark the root λ as visited. In step 2 we set *t* as the root and check *U(λ)* to see if we have visited all the nodes of the tree. In that case we go to step 5 where we pick a node of depth *d* uniformly at random, output it as the tail and finish the algorithm. Otherwise, in

step 3 we check *t* to see whether it has ever been visited in which case we output the phrase associated to *t* and return to step 2. If the node has already been visited, we select a child *a* uniformly at random, set *t* = *ta* and return to step 2.

This algorithm selects sequences uniformly at random, which yields the maximum entropy as we wished. It is the base for our work in simulation of textures and texture mixtures, and is further developed in the following sections to tailor image textures. Figure F8 shows an example of the use of this algorithm. The left image shows a black and white input texture. We apply the Hilbert scan (described in section 3.4) to get an input binary sequence. Then, we compute the UTC for that sequence following the incremental parsing algorithm shown in figure A1. The resulting parsing tree and tail size are used as inputs to the random sampling algorithm just described to simulate two new sequences. Reverting back the Hilbert scan gives as output the two images shown in the center and the right figure F8.

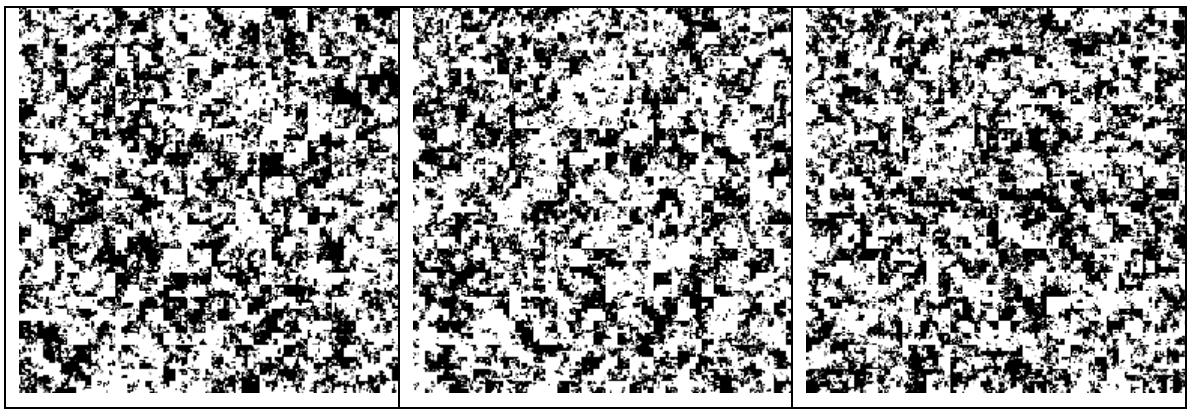

Figure (F8): Example of the random sampling algorithm. The left figure shows the input texture. The center and right figures show two different simulations.

3.6 Wavelets decomposition and the steerable pyramid

In this section we describe a *multi resolution analysis* scheme (*MRA* for short) widely used in the image processing field.

Traditional frequency analysis tools involve the use of *Fourier transform* (i.e. *Fast Fourier Transform*, *Discrete Fourier Transform*, and *Windowed Fourier Transform*). These methods decompose a periodic signal into a series of sines and cosines. Thus, any periodic signal can be analyzed either in the time domain or, applying the Fourier transform, in the frequency domain. Various approaches have extended this idea to non-periodic signals (e.g, applying a Windowed Fourier Transform). An important drawback of the Fourier analysis in some applications is that it is not local in space. On the other hand, a wavelet decomposition of a signal is localized not only in the frequency domain but also in space due to the fact that it works at different scales. This allows us to analyze the signals both at a coarse level and at a fine grain level. For example, if the analysis is applied to an image, it allows studying the objects that compose it (the coarse level) and the details of each of them (in a finer scale). A brief introduction to wavelets can be found in [22], and a thorough review in [32].

Every wavelet transform comprises the use of a kernel filter (mother wavelet) and the application of translated and dilated versions of this kernel. Usual wavelet transforms are not translation-invariant [46], [30], a feature that is usually desirable in image processing. However the steerable pyramid, introduced in 1990 by Simoncelli [46] is both translation-invariat(i.e. *aliasing-free*) and rotation-invariant. Due to these features, it has been widely used for image processing. The steerable pyramid is a linear multi-orientation image decomposition. The basis functions of the steerable pyramid are translations, dilatations and rotations of a single kernel. For more information on the design of this kernel see for example [18]. Figure F9 shows the analysis and synthesis diagram of the steerable pyramid for a single scale. The left side shows the analysis step for a single scale. The signal goes through a high pass and low pass filters. The lowpass subband (L0) is further filtered thru *n* orientation bandpass filters and a lower pass subband (L1 in the figure). This subband is decimated (by a factor of 2) and the

procedure is repeated for the next scale. The recursive application of it yields the pyramidal structure.

The synthesis procedure is usually referred as *collapsing the pyramid*. It works backwards, upsampling the input lowpass subband (L1) and applying the inverse transform to reconstruct L0. Applying a steerable pyramid decomposition to an image and subsequently collapsing the pyramid does not gives a perfect reconstruction. However, it was noted in [44] that the reconstruction errors are small enough for most applications. As it can be seen from the figure, the high frequency subband is copied 'as is'. The number of scales and orientations used are arbitrarily selected to match the needs of the application. However, it has to be taken into account that the transform is overcomplete by a factor of 4n/3. This can be a major drawback if the steerable pyramid were intended to use for image compression. However, for our work this issue will only impact the run-time performance of our algorithms. Other wavelet decompositions (e.g. orthogonal ones) do not suffer from this bloat in size, though they are generally worse for image processing as they are neither translation-invariant nor rotation-invariant.

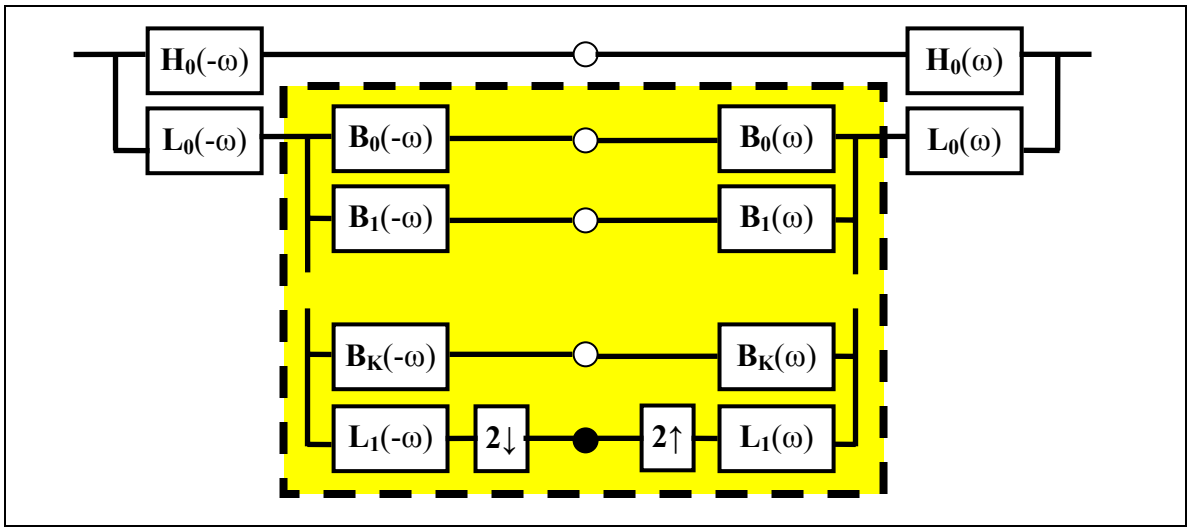

Figure (F9): Steerable pyramid analysis/synthesis. Figure taken from [44]

4 Previous Work

Before we give a detailed description of our work, we review some of the previous work in the literature related to texture simulation, universal simulation and texture mixture and their relation with our work.

The reference papers in universal type classes is the work by Seroussi [42]. It shows the analogy between UTCs and traditional types, provides formulas to obtain the size and number of classes of a given sequence length, and develops algorithms for random sampling from the class. As an example application they used the random sampling algorithm to simulate sequences. They took a black and white texture image and applied a Hilbert scan to obtain a 1-d input sequence. They obtained the UTC for that input sequence, and sampled new sequences from it. Then, they reversed the Hilbert scan to get back a new texture that, as is proved in their paper, has the same statistical properties as the input texture.

We use this scheme as a starting point for our work. However, as we discuss later, there are some drawbacks in this approach especially when applied to signals over larger alphabets, for e.g. continuous tone images and textures. These challenges are briefly discussed in the closing section of [42].

Heeger and Bergen [23] provide a texture synthesis algorithm aimed at the so called "stochastic textures" (e.g. sand, grass). Their algorithm is based on histogram matching [20] of filter outputs. Initially they take a texture input image and create a temporary output image of the same size initially filled with uniform white noise. While iterating the algorithm, the output image will be modified to resemble the input texture. Their approach combines the use of histogram matching with image pyramids. In their work they used two kinds of image pyramids, the *Laplacian pyramid* and the steerable pyramid (described in section 3.6) with this last pyramid giving better results. For every iteration of the algorithm, a wavelet decomposition of the image is performed. Then, a histogram match is performed for each subband. A *histogram match* between two images A and B is a procedure in which A modifies its pixel values so that the resulting image has

the same histogram as *B*. After the histrogram match is performed, the output image pyramid is collapsed and a new histogram matching is performed. This process (decomposition, histogram matching, collapsing) is repeated a certain number of times, after which the output image will start resembling the input texture. From the point of view of a human observer, the algorithm typically converges after a few iterations. Allowing the algorithm to run too many times might lead to artifacts in the output image (mainly due to errors on the pyramid decomposition/reconstruction). For color images they point out that usual images uses a color space in which each channel is not independent of the others, so a conversion step is needed before applying the algorithm. Their approach for these types of images is to use principal component analysis (PCA) to work with decorrelated variables, thus allowing them to apply the algorithm independently to each channel. Figure F10 shows some successfully synthesized textures and also some less successful ones (taken from [16]).

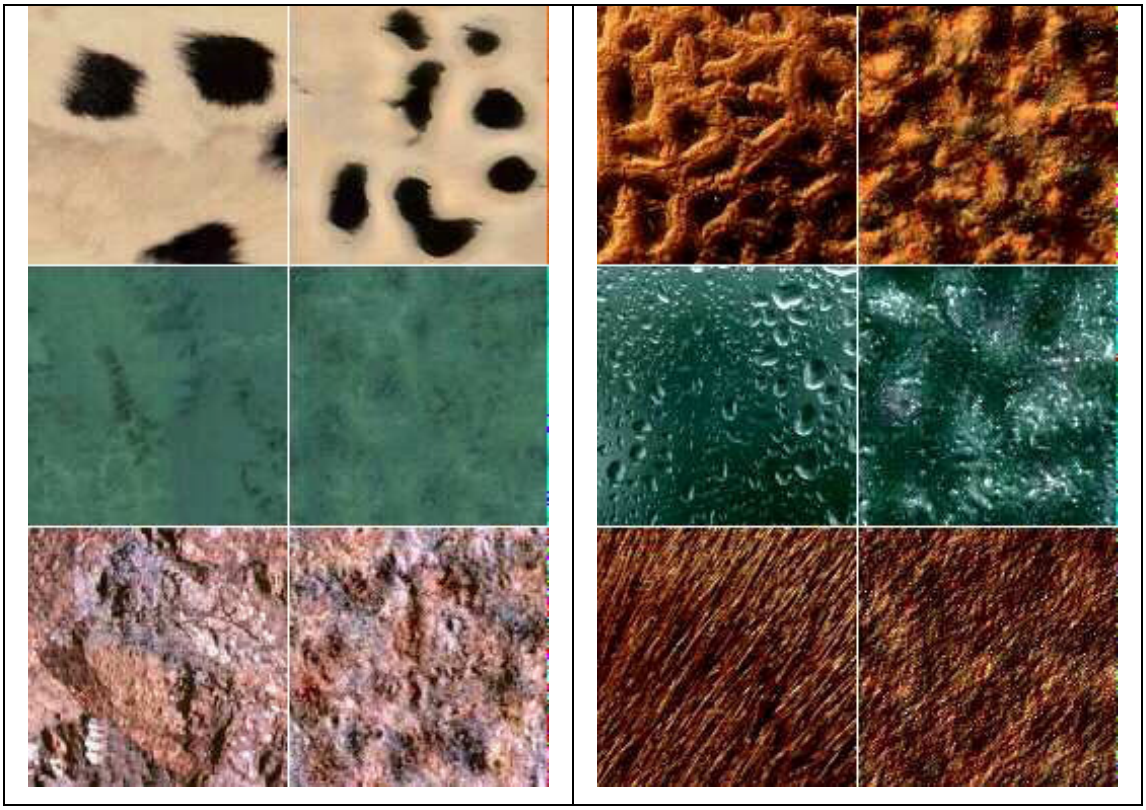

Figure (F10): Heeger and Bergen's synthesis examples. The first two columns show original textures and the successfully synthesized ones. The last two cols show original textures and their less successful synthesized versions

De Bonet and Viola [4] extend the idea of a multi-scale statistical model for natural images by using cross-scale information. While Heeger and Bergen's algorithm was well suited to work with smooth textures, more structured textures (like a wall of bricks) do not produce good results because the wavelet decompositions for these textures are not cross-scale independent (i.e. many coefficients are needed to represent a long edge). De Bonet and Viola's approach to this problem was to define a vector V for every pixel that combines the coefficient of each feature extraction filter at every scale of the pyramid (i.e. V_i are the coefficients of the filters at the i^{th} scale level). Note also that the coefficients due to the top levels of the pyramid are shared by many pixels (4^i pixels). The joint distribution of the coefficients is then modeled by a chain, in which the distribution of the coefficients of higher frequency is conditioned by the coefficients of lower frequencies:

$$p\left(\vec{V}(x,y)\right) = p\left(\vec{V}_M(x,y)\right) \times p\left(\vec{V}_{M-1}(x,y)\middle|\vec{V}_M(x,y)\right) \times \cdots \times p\left(\vec{V}_0(x,y)\middle|\vec{V}_1(x,y),\cdots\vec{V}_M(x,y)\right)$$

where M is the number of levels of the pyramid.

The conditional distributions involved here are estimated from sampled vectors taken from example images. When synthesizing a new image, they start generating the values for the lowest frequency scale of the pyramid (V_M) at every possible pixel location, and then they proceed with the next level of the pyramid. After that the pyramid is collapsed to obtain the new image. Figure F11 shows a comparison between Heeger and Bergen with De Bonet and Viola's algorithm for a structured image texture (images taken from [4]).

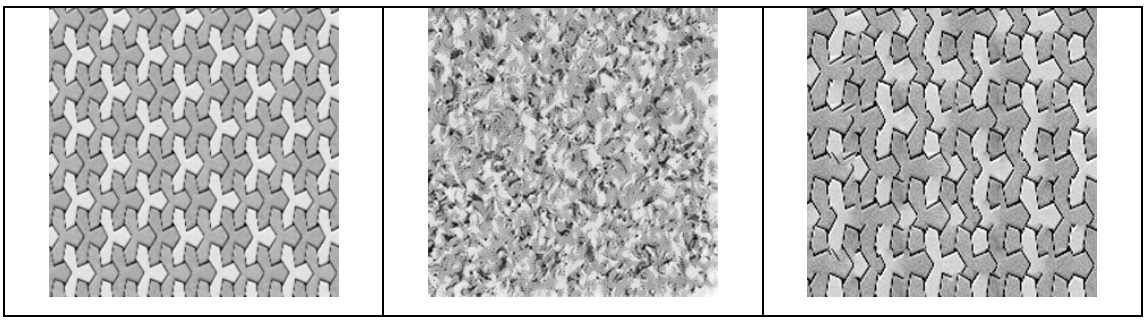

Figure (F11): Comparison between Heeger and Bergen with De Bonet and Viola's synthesis result. The left image is the input texture, the middle is Heeger and Bergen's and the last is De Bonet and Viola's

Efros and Leung [15] provide a method for inpainting by growing texture one pixel at a time. Their approach can also be used to synthesize new images by extending that idea (e.g. just consider an extension to an input texture by enlarging it with the region to synthesize). They model texture using Markov random fields (the probability distribution of pixel values only depends on the values of the pixel's neighborhood). They define a parameter L_w as the side of a square window *w(p)* centered at pixel *p*. For any given pair of pixels (p_1, p_2), they define a perceptual distance between the patches of the windows $w_1(p_1)$ and $w_2(p_2)$. This distance is derived from the normalized sum of squared differences of the corresponding pixels en each window, by convolving it with a Gaussian kernel, thus giving priority on the perceptual distance to the closer-to-center pixels of the window. Then, to derive the unknown pixel at the center of a *w*, they sample from the probability distribution $P(p|w(p))$. Of course, generally not all the pixels of w(p) (besides p,of course) will be known, so they adjust their metric to work with holes in the window. The resulting synthesized images are able to capture both stochastic and structured textures. The main drawback is that the parameter L_w must be adjusted for each texture to match the size of the biggest structure patch; and more complex images with features in various scales will not be correctly captured. As a side note, their authors also acknowledge the tendency of the algorithm to get locked in some area of the image and start creating verbatim copies of some region of the image.

Another work which synthesizes images with impressive results is the framework of *Image Analogies* which addresses the following problem [24]: "Given a pair of images A and A' being the unfiltered and filtered source images respectively, synthesize a new filtered target image B' given the unfiltered target image B." In other words, the idea is to find an image B' that relates to B in the same way that A' relates to A. This framework has been applied successfully in many applications, including texture synthesis, super resolution, texture transfer and artistic filters.

The algorithms presented in the referenced papers can be extended to generate texture mixtures. Just create a source image in which different regions are mapped with different input textures. However, this approach may lead to undesired artifacts, in many cases due to the sharp edges found between boundaries of the source textures. Other works are especially tailored to mixing textures. The work of Bar ([1], [2]), uses

statistical learning with tree mixing techniques to generate new texture mixtures. It is assumed that image textures are samples from an unknown stochastic source. Thus, after learning the underlying statistical model new textures can be sampled from it. Given the set of input textures ($s_1 \ldots s_k$), each texture is an observation from an unknown stochastic source S_k. A hypothetical source Z is defined such that Z is closest to all the sources S_i simultaneously (i.e. it minimizes the Kullbak-Leibler divergence to every S_i). Due to this fact, Z is called the mutual source of S_i (i=1..k). After obtaining the mutual source, new images which are statistically similar to S_i can be generated by sampling from Z. Instead of working directly with the input images, a wavelet decomposition (steerable pyramid) is performed to each input signal yielding k trees. Then, to sample from the mutual source, a tree mixing technique is applied, which results in a new tree being generated. Applying the inverse wavelet decomposition to the tree yields a new image texture which is statistically similar to all the input sources. Thus, when the input sources s_i come from different textures, the algorithm generates a texture mixture. As we shall see in section 6, our method to simulate texture mixtures is also based on mixing trees. When applying their algorithm to mix disparate textures, their approach tend to lock on one of the input trees, so some adjustments to the original algorithm must be taken to alleviate this problem. Our method does not suffer from these kind of problems due to the way we mix the input trees. Their method not only gives good results with texture images, it is also able to synthesize time varying textures (e.g. a crowd of people. Fire flames, etc). Figure F12 shows two synthesis done with this methodology. In the first example, the left picture is the input texture. To synthesize a new texture based on it, some overlapping regions are taken from the input texture and fed to the statistical learning algorithm giving the result shown on the second image. To generate the texture mixture shown in the rightmost picture, the first and third images were used as input sources.

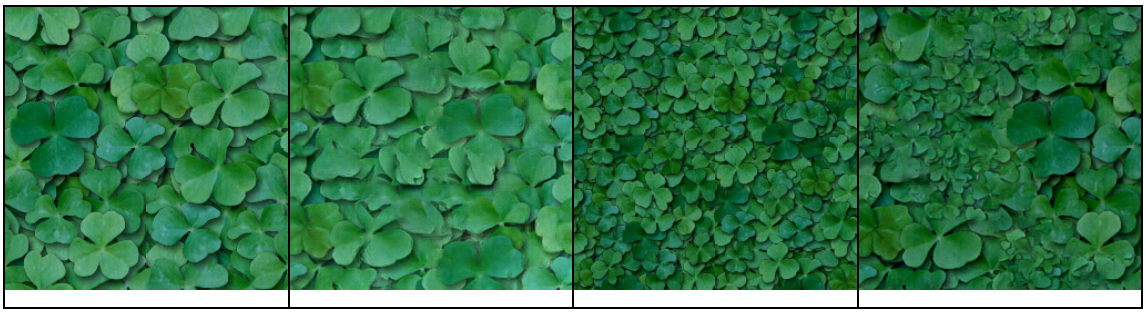

Figure (F12): Ziv Bar's texture synthesis example. From left to right: original texture A; sampled texture from A; original texture B; mixed texture from A and B

To finish this section, we mention some recently works on texture characterization. Simoncelli and Portilla presented a texture characterization via joint statistics of wavelet coefficient magnitudes [47]. Their work is based on the fact [39] that the features of real images usually contain large coefficients in local spatial neighborhoods and adjacent scales and orientation. Working with a steerable pyramid decomposition (mainly because of its properties of being translation and rotation invariant), the characterization is given by a minimal set of statistical measurements.

Another recent work by Zhu et al [51] presents a statistical theory for texture modeling (FRAME: Filters, Random fields, and Maximum Entropy). Their characterization is based on the derivation of a probability distribution *f(I)* over an ensemble of images *I* with the same texture appearances. To derive this distribution (given a set of observed textures) they follow a two step methodology. They select a set of feature extraction filters which are applied to the example observed textures to obtain the filtered textures histograms. With these histograms the marginal distribution of *f(I)* is estimated. Then a feature fusion step is performed which constructs a maximum entropy distribution *p(I)* given the marginals distributions of *f(I)*. After *p(I)* is constructed, new images with the same texture appearances can be created by sampling directly from it. In [19] Gimel, Van Gool and Zalesny recently pointed out that there are some major drawbacks on the theory (e.g. how to choose the set of filters).

5 Universal Simulation of Textures

In this section we introduce our framework for universal simulation of textures using universal type classes. The framework developed is quite flexible, allowing us to work either directly with the input texture or over a wavelet decomposition of the source textures. This section addresses the problem of universal simulation of single textures; section 6 deals with texture mixtures from multiple input textures.

5.1 Basic simulation

We begin our work by applying the scheme used in [42] for universal simulation (which was described in section 3.5). However, we use greyscale and color textures instead of binary images. Color images are usually encoded in some color space. For example, in the RGB color space the image is composed of 3 channels (Red, Green and Blue). Typical image formats like *PNG* or *GIF* use this color space. Another color space, YCbCr, is also composed by 3 channels (a luminance channel *Y*, and two chrominance channels *Cb* and *Cr*). If we re-encode an image from the RGB color space to the YCbCr color space, the luminance channel contains information from the three RGB channels as it is calculated from a weighed sum of them. Throughout our work, we will use the YCbCr color space. More specifically, we will use the luminance channel for our simulations and transfer the results to the chrominance channels, i.e., we will let the simulation on the luminance channel determine the rearrangement of pixels in the image, and apply the same rearrangement to the chrominance channels. More information on color spaces can be found in [20].

Following the extension introduced by Lempel and Ziv to their incremental parsing algorithm to deal with two-dimensional data [28], we use a plane filling curve to map the 2D input texture onto a 1D input sequence.

Roughly speaking, we traverse the input texture with some plane filling curve. In this traversal we copy the contents of the luminance channel at every visited pixel to create a 1D sequence. Then, to obtain the universal type class for the sequence we build the LZ-

tree with the algorithm described in section 3.3 (Fig A1). With the resulting tree and the tail size we generate a new sequence (simulation) following the random sampling algorithm presented in section 3.5 (Fig A2). This is basically a mapping of the occurrence in time of each pixel from the original image sequence to a new location in the simulated image sequence (i.e. a permutation of phrases from the incremental parsing). Then, we just have to revert back the scan applied to the original texture to obtain the simulated texture. When working with color images, we have to apply the results of the simulation also to the chrominance channels. This can be easily done using the indices of the permutation of phrases of the incremental parsing but this time instead of reverting back the luminance channel, we revert back the chrominance channels. In our basic simulation the scan order chosen was the Hilbert scan, mainly due to the locality measure properties [21] described in section 3.4.

One of the main contributions of this work is to measure the richness of the simulations (i.e. the size of the UTC). In other words, we would like to know how many different textures we can produce as simulations of the input texture, all complying with the "similarity" constraints imposed by our scheme. As shown in [42], by the properties of the incremental parsing algorithm, all the simulated textures are statistically indistinguishable (in the limit) by finite memory models of any order. We now elaborate on the way to measure the richness of the class of an input texture. Recall that in step 4 of the random sampling algorithm (fig A2) we draw the next node to visit (*tv*) in the tree. We can determine the probability of selection of the chosen node by computing $p_d = \frac{U(tv)}{U(t)}$. Thus, to calculate the probability of simulating a sequence we just have to compute $p_s = \prod_{\{\text{every draw}\}} p_d$. The number p_s can be easily computed as we execute the simulation algorithm, by noting (and multiplying) the probabilities of the symbols chosen in each random draw. But the algorithm selects sequences uniformly at random, thus $p_s = \frac{1}{|U_{x^n}|}$. This gives us a method to calculate the number of different textures our scheme can simulate for a given texture, by running one simulation instance on that texture.

The size of the UTC (U_x^n) is usually very big, so instead of working with such large numbers we take logarithms. Thus, instead of calculating the probability of each draw, we determine the contribution of entropy for these drawings by computing $H_d = -\log\left(\frac{U(tv)}{U(t)}\right)$. If we sum up these contributions we end up with the entropy of the full simulation. Let's take for example the texture of figure F8 shown in page 32(a 256x256 monochrome image). After computing the UTC for this image we calculated the size of U_x^n which is roughly 2^{36994}. Therefore we can synthesize 2^{36994} different textures which share the same UTC.

We now test our simulation scheme with the texture shown in figure F13a which is a photograph of a carpet (512x512 greyscale image) with a diagonal pattern (seen left-to-right it goes up-to-down). After applying a simulation we computed the output entropy which gave 544460 bits. Thus, the size of the UTC was 2^{544460}. The results of the simulation are shown on figure F13b. Clearly, the simulated image does not show what we would expect. In the following section we analyze the reasons for these results and improve our simulation scheme accordingly.

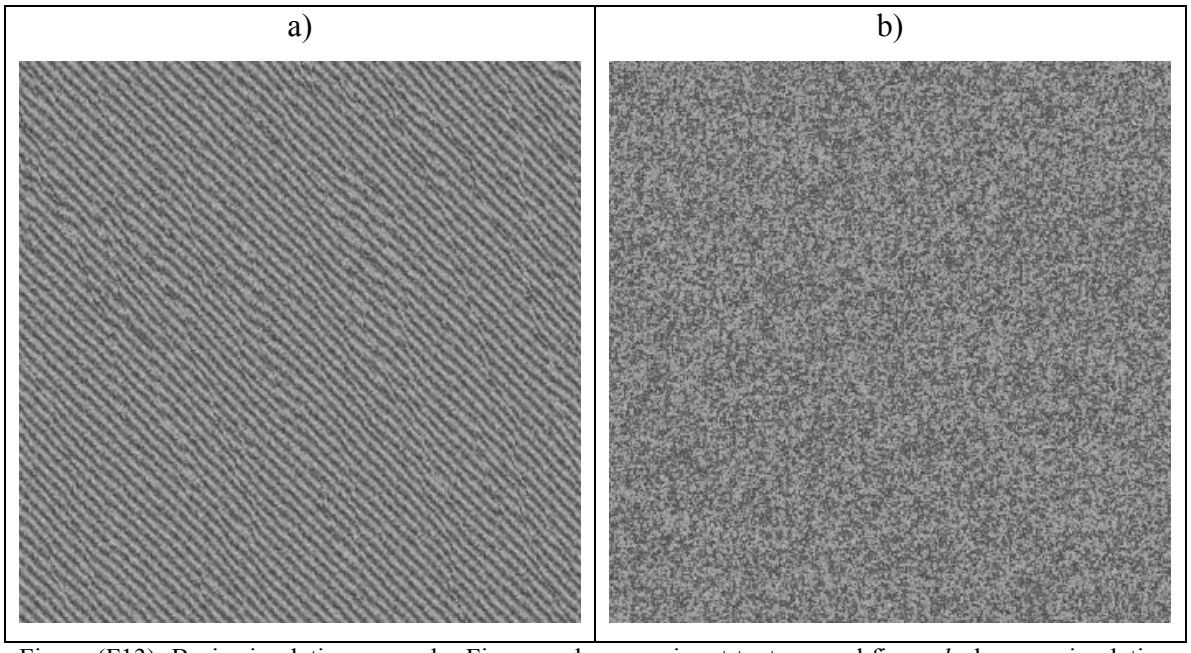

Figure (F13): Basic simulation example. Figure *a* shows an input texture, and figure *b* shows a simulation using the basic simulation scheme

5.2 The 'Context dilution' problem: quantization and vectors

The Lempel Ziv incremental parsing algorithm can be seen as a *context modeling tool* [26, 40]. Although it is generally described as an encoding based on a string algorithm that does not explicitly rely on probabilities, the LZ78 scheme can be shown to be equivalent to an encoding resulting from a probability assignment. This point of view was first described in [26]. Under this interpretation, each symbol in the input string x^n can be seen to be assigned a probability conditioned on the previous symbols in the *current phrase* of the incremental parsing. Thus, given a phrase $p = x_j^{j+s}$, the symbol x_j is assigned a probability conditioned on the empty string (i.e., unconditional), x_{j+1} is conditioned on x_j, x_{j+2} is conditioned on $x_j^{j+1}, \cdots, x_{j+r}$ is conditioned on x_j^{j+r-1}, r ≤ s. The conditional probability of $x_{j+r} = a$, for an arbitrary symbol *a*, is proportional to the number of nodes of the current parsing tree in the subtree rooted at $x_j^{j+r}a$, if such a subtree exists, or zero otherwise. Equivalently, this number is equal to the number of prior occurrences of *a* in the current context. Notice that this probability assignment is closely related to the random sampling algorithm described in section 3.5 (figure A2).

One of the problems arisen when we use the incremental parsing as a modeling tool applied to practical images is that of "*context dilution*" which will be addressed in this section.

Since the symbol alphabet in continuous tone images is relatively large (typical images are 8 bits per pixel per channel), there are usually very few exact context repetitions in an image of practical size, except for very short context lengths. Therefore, context quantization techniques have been applied in literature to help modeling tools get more statistically significant context models for this type of images (see, e.g, [7, 8, 35, 50]). In the LZ setting, this means that phrases will tend to be quite short, and will not capture higher order dependencies in the data. To ameliorate this problem, we employ symbol quantization and allow "approximate matches" in the incremental parsing (i.e. we

group nearby pixel levels together). This will allow the dictionary to collect longer phrases that better capture image patterns. In the traditional incremental parsing, each arc corresponds to a single symbol and each node corresponds to a phrase which is the concatenation of the parent's phrase with symbol associated with the arc that connects both nodes. On the other hand, when we use quantization, the whole set symbols which (after quantization) are clustered together, are associated with a single arc. However, we only quantize the data for the purpose of building the parsing tree. Therefore, each node is labeled with the exact phrase used to reach that node. This way we keep track of the exact, unique input string that lead to the creation of the node. Afterwards, when producing the simulated output texture, the original strings are faithfully reproduced, thus preserving the statistics of the texture.

The quantization techniques incorporated to our simulation scheme directly affect the trade-off between the richness of the UTC and the visual quality of the simulated textures. As we modify our similarity constraints to allow for approximate matches, the incremental parsing will contain longer phrases. Thus, the simulated textures will be composed from copies of larger patches from the input texture. This, of course, comes at the expense of some loss of entropy which will be measured at the end of this section. In the end of this section we show, numerically, the loss of entropy of the output due to the data processing we add to our framework.

Our simulation scheme is further refined to work with vector input data sequences instead of one-dimensional ones. When working with vectors, a preprocessing stage is applied to the input texture to group a set of pixels into a vector. Instead of traversing the image pixel by pixel we take chunks of **n** pixels at a time. As a result, we build the simulated texture from small patches (e.g. 4 pixels) instead of single pixels. To do this, we define some distance measure between two vectors, and combine it with a quantization stage. As it was noted above, this can be used in the incremental parsing algorithm to produce longer paths in the parsing tree. For example we can define some quantization threshold for the distance measure. Below this threshold we will say that two vectors are close enough to be grouped together in the same LZ-tree node. This can be regarded as an extension of the "approximate matches" discussed above.

Further refinements to this idea are made. The quantization threshold is adaptively set in order to achieve a certain mean length for the phrases in the LZ-tree, which can be used as a parameter of the algorithm. We use a variable threshold in the sampling algorithm which depends on the depth of the current standing node of the tree (we decrease the threshold in deeper nodes of the parsing tree). Furthermore, we automatically select the value of the threshold for each depth level using a two-stage procedure to achieve certain properties for the tree (e.g., number of outgoing paths for each node, or mean path length). We defined a target phrase length which indicates the mean length of the phrases in the parsing tree. To achieve this phrase length, on a first stage we apply the sampling algorithm controlling the quantization threshold by applying slight modifications while building the tree. Let $c_p(\lambda)$ denote the number of nodes in the parsing tree and n the number of vectors currently processed. Then our estimator $\hat{e}$ of the mean phrase length is computed as $\hat{e} \equiv n/c_p(\lambda)$. After adding a new node to the tree we recalculate this value and if the result differs from the target phrase length more than a certain threshold we modify the quantization threshold. On a second stage, we fix the quantization threshold values for every tree depth and reapply the sampling algorithm with these threshold values. This modification greatly improves the visual quality of the simulated textures. However, as in the case of quantization explained above, this comes at the expense of a reduction in the entropy of the output. However, properties of statistical similarity are kept unaffected by these manipulations.

The framework allows for more operations over the source texture when applied to vectored input data. As an example, we implemented a simple pixel predictor for the pixels in the group it belongs. The pixels belonging to a vector are ordered by the scan applied to the input texture, therefore we know from the properties of plane filling curves that the distance between two consecutive pixels in a vector is one. Thus, for each pixel in the vector, we use the value of the previous pixel as a predictor for current value. So, instead of having a sequence of pixel values, we have a sequence of prediction-errors.

We applied a simulation using the quantization techniques described above to the texture shown in figure F14a. For this simulation we also arranged the input to use vectors of size 8 and a target sequence length of 128. The result of one simulation is shown in figure F14c. In another simulation (using the same input image) we used a

simple predictor together with quantization and vectorization. In this simulation the value of the previous pixel in the Hilbert scan is used as a predictor of the next pixel.

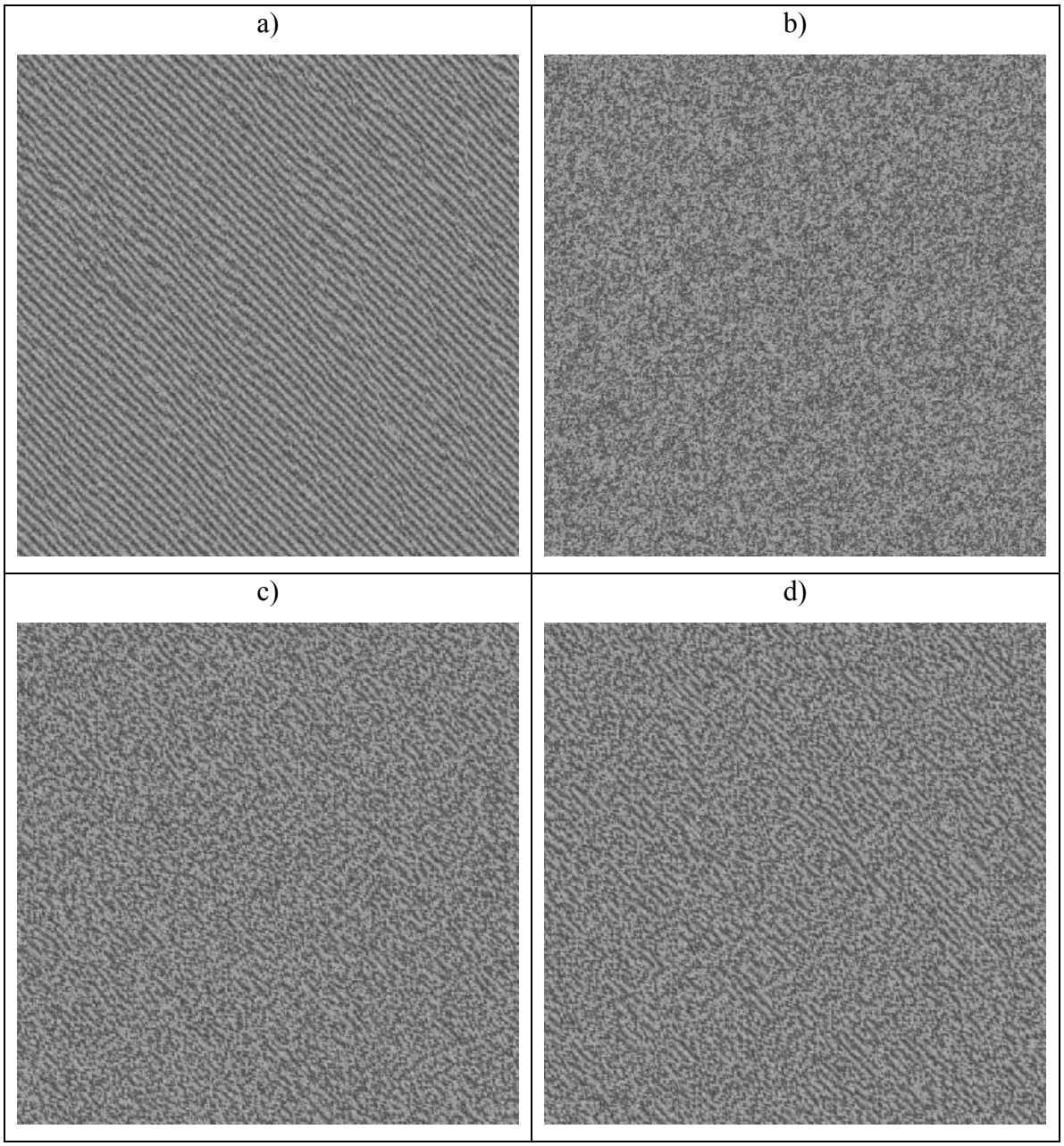

Figure (F14): Simulation example across successive refinements of our simulation scheme. Figure *(a)* shows the input texture. The rest of the figures show simulations using: (*b)* basic scheme. (*c)* vectored data (size 8) and quantization. (*d*) vectored data (size 8), quantization and the simple pixel predictor.

Figure F14d shows the result of a simulation using this scheme. We computed the entropy of the output using the same formulas described in section 5.2. For the simulation shown in figure F14c the output entropy was ~53720 bits, which is a reduction of

~4,9*10^5 bits of output entropy compared to the results for the basic simulation scheme. The visual improvement in the simulated texture is evident. While the basic simulation scheme failed to capture the patterns of the texture, this simulation shows some patches with the diagonal pattern shown in the input texture. Thus, while the use of quantization removed most of the output entropy, it helped to capture in part the features of the texture. For the simulation using the simple predictor (which is shown in figure F14c), the output entropy was ~14300 bits which yield a reduction of ~5,3*10^5 bits out output entropy again compared to the results for the basic simulation scheme. In this case, the simulated texture seemed to capture better the patterns of the input texture, resulting in a texture which looks sharper. The table from figure F15 shows a column with the output entropy for these simulations and in another column the loss of entropy with respect to the basic simulation scheme.

Simulation Scheme	Input parameters	Output entropy (bits)	Delta output entropy compared with Basic simulation scheme(bits)
Basic	N/A	544460	0
Quantization + Vectors	Vector size: 8	53720	~4,91*10^5
Quantization + Vectors + Predictor	Vector size: 8 Predictor: previous pixel in the Hilbert scan	14300	~5,30*10^5

Figure (F15): Number bits of output entropy for different simulation schemes. The input texture is shown in figure F14a. The simulated textures for the Basic, Quantization+Vectors and Quantization+ Vectors+Predictor are shown in figures F14b, F14c, F14d respectively.

5.3 The 'Loss of Context' problem

Another problem when using Lempel Ziv as a context modeling tool is that after one gets to an unused node (thus outputting the related phrase) we start again from the root of the tree, so there might be no visual coherence between two consecutive phrases. Basically this means that after we output a phrase (which originally belong to some part of the input texture), the next phrase we output might come from a distant place in the input texture. There is no context preserving mechanism which gives, at some extent, preference to select a phrase which comes "near" the last output phrase in the input texture. We refer to this issue as the *'Loss of context'* problem. Although the effect of this discontinuity is statistically negligible, it is visually unpleasant.

To induce our simulation framework to output more visually coherent contiguous phrases we extend the Lempel-Ziv incremental parsing algorithm to collect side information while constructing the tree. We also extend the random sampling algorithm accordingly in order to use that information to 'restart' in a deeper node of the tree after we output a phrase.

The extended incremental parsing algorithm, described in figure A3, extends the basic algorithm by collecting side information about the input data. The idea is to use the last m symbols of each phrase processed by the algorithm to locate the context in which the following phrases occurred. As in the case of the basic incremental parsing algorithm, we denote by x^n a one-dimensional input sequence with alphabet Λ of cardinality α. Let m be an integer denoting the maximum path length of side information to collect. We construct two α-ary trees: T with root λ and I with root μ. T is, as in the case of the basic incremental parsing algorithm, the LZ-tree. On the other hand, I denotes a *back reference tree*. The nodes of I represent suffixes of phrases in T and are used in an extended sampling algorithm to preserve the context after we output a phrase. As in the case with the parsing tree, the arcs between two nodes in I are labeled with a symbol S, and the suffix associated with each node is the concatenation of the node's parent suffix with S.

Inputs: input sequence (x^n) of length **n**, maximum side information tree depth(**m**)

Outputs: LZ tree(***T***), tail depth(***d***), back reference tree(***I***)

1. Set **j** at the root(μ) of **I**.
2. Set **t** at the root(λ) of **T**.
3. Check if there is more data from the input sequence. If there are no more symbols (i.e. i=n) go to step 7.
4. Get the next symbol ***S*** = $\mathbf{x_i}$, set **i = i+1**. Look for all the outgoing arcs of ***t*** to see if one is labeled with ***S***.
   a. If there is a match, move **t** to the node pointed by the matching arc and go to step 3.
   b. Create a child node ***v*** labeled with the concatenation of the phrase in ***t*** and ***S***. Label the arc between ***t*** and ***v*** with ***S***, *and set* ***t***=***v***. Add ν to the set of nodes of **j**. Let *l* be *min*(**m**, *depth*(**t**)). Ensure the path of the last *l* nodes of **t** in **T** also exist in **I**(starting from μ), adding the missing nodes along the paths to **I** and set **j** to the node correspondent to **t** in **I**.
5. Go to step 2.
6. Return the trees (**T**, **I**) and the integer (**d**) of the depth of the last node visited by the algorithm (the *tail depth*)

Figure (A3): Extended Lempel-Ziv incremental parsing

The length of a suffix is bounded by *m* which means that we will preserve the context of at most *m* symbols. Every node *i* in *I* points to a set of nodes of *T* which we refer as *preferred phrases* of *i.* In the extended sampling algorithm, the preferred phrases of a suffix will be favored as continuations of a phrase in a simulated sequence when the suffix matches the phrase's suffix. The extended parsing algorithm works as follows. We use the variable *t* to point to a node of *T* and *i* to point to a node of *I*. After adding a new node to the LZ-tree we store, in the back reference tree, a pointer to the newly created node. The pointer is stored in the node of *I* that represents the *context* of the previously processed phrase. Here when we refer to the context of a phrase, we are denoting the last *m* symbols of the phrase or, when the phrase is less than m symbols length, the whole phrase. Aside from the initialization of *I* and step 4b*,* the algorithm is the same as in the case of the basic incremental parsing. In step 4b of the algorithm, we store a pointer to the

output phrase into the context of the previous phrase (i.e. we mark the output phrase as a preferred for the previous phrase).

Initially each node in the back reference tree it is likely to list few nodes in the LZ-tree. Nonetheless since the back reference tree is an *m*-depth tree, after some significant amount of data is processed each node will point to many preferred phrases.

The example in figure F16 shows the initial steps in a sample sequence in which for simplicity we chose *m*=2. The figure shows the two trees after a small amount of data has been processed.

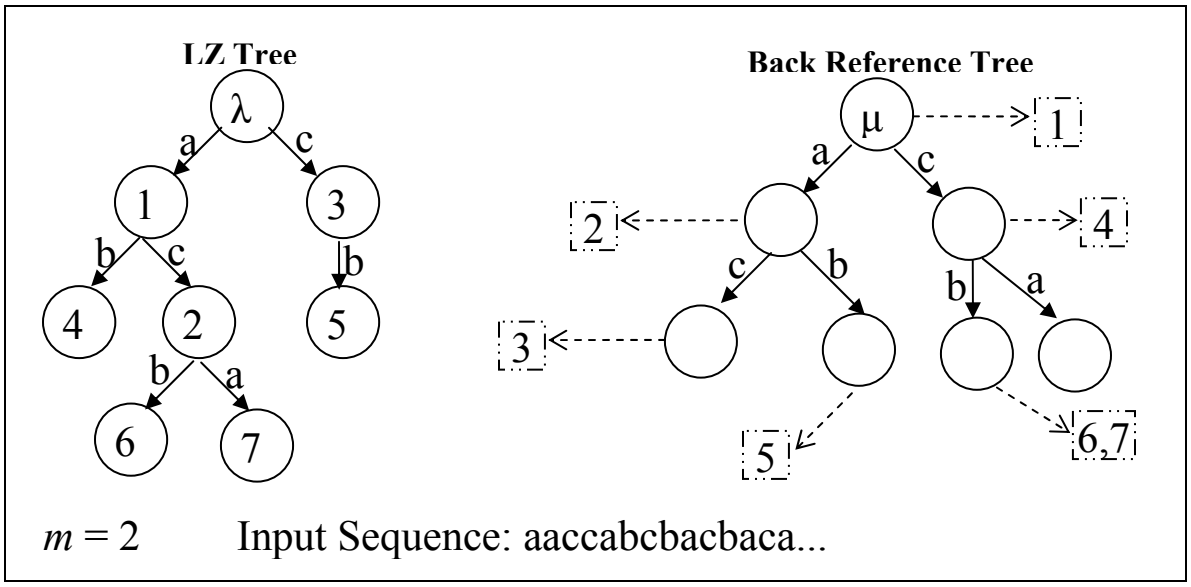

Figure (F16): Back reference tree example

After applying the extended incremental parsing we end up with two trees. On one hand we have the typical LZ-tree. On the other hand, we have the back reference tree which contains information about all the preferred phrases of each phrase processed in the incremental parsing. Thus, for each phrase *p*, a list of preferred phrases which comply with the following rule is kept: every preferred phrase must have the property that the *m*-suffix of the previous phrase in the incremental parsing matches the *m*-suffix of *p* (i.e. they share the same previous context).

We use the back reference tree in the extended random sampling algorithm shown in figure A4. For every phrase *p*, the list of preferred phrases associated with *p* will be

favored as continuations of *p*, thus preserving a length-*m* context in the phrase transition. In the extended random sampling algorithm, we introduce a constant *c* that denotes the restarting depth of the preferred phrases. In step 1b of the initialization stage of the algorithm, every preferred phrase contained in *I* is visited to check whether the phrase depth is greater than *c*. In such case, we replace the phrase with the first *c* symbols of it. In step 3 (which is executed at the beginning of the algorithm and then every time we output a phrase *p*), the list of preferred phrases for *p* is revisited. For every preferred phrase, we verify whether the node in the LZ tree associated to it is reachable (i.e. if it has not already been visited). If there are no reachable nodes we restart from the root. Otherwise we draw randomly the restarting node from the pool of reachable nodes. In terms of the parsing tree, this modification causes the sampling algorithm to restart, if possible, at depth *c* in the tree after outputting a phrase, rather than restarting from the root. When no preferred phrases are reachable, the conventional restart rule is applied.

Inputs: LZ tree(***T***), tail depth(***d***), back reference tree(***I***), context depth **c**

Output: Randomly sampled sequence

1. Initialization.
   a. Walk through all the nodes of **T** marking them as unused, and for each node ***t*** set U(**t**) = c_p(**t**). Mark the root (**λ**) of **T** as used and set U(**λ**) = U(**λ**)-1.
   b. Walk thru all the nodes **j** of **I**. For every node **v** in the set of nodes of **j** check to see if the depth(**v**) > **c**. In that case replace **v** with the ancestor node **v'** along the path of **v** with depth(**v'**) = **c**
   c. Set **j** at the root(μ) of **I**
2. Set **t** at the root(**λ**). If U(**t**) = 0 go to step 6.
3. Try to set a context for **T**. Lets define the function R(**v**) to be 1 if the node **v** has been used and 0 otherwise, and $U^* = \sum_{b \in set(v)} U(b).R(b)$, where $set_j(v)$ denotes the set of nodes in **j** which are rooted at λ.
   a. If no node is reachable(used), set **t** = λ
   b. Otherwise draw randomly a node **t** from the set of nodes of **j** with distribution $\text{Prob}(t=b) = \frac{U(b).R(b)}{U^*}, b \in set_j$
4. If **t** is marked as unused:
   a. Output the phrase associated to **t**, mark **t** as used, set U(**t**) = U(**t**)-1.
   b. Let $l = min(\text{depth(I)}, depth(\mathbf{t}))$. Set **j** = μ. Traverse **j** in I along the last **l** symbols of the phrase associated to **t**
   c. Go to step 2
5. Draw randomly a symbol ***a*** from Λ with distribution $\text{Prob}(a=b) = \frac{U(tb)}{U(t)}, b \in \Lambda$, set U(**t**) = U(**t**)-1, set ***t*** = ***ta*** and go to step 4.
6. Pick uniformly a node ***t*** of depth **d** from **T** and output it as the tail.

Figure (A4): Random sampling from universal type with side information

Continuing with the examples given in section 5.2, we applied this simulation scheme to the input texture shown in figure F18a. The results of the simulation using

quantization, vectors of size 8 and side information (with $m = c = 6$) are shown in figure F18c. The output entropy was 39090 bits. In figure F19c we show the result of a simulation in which, additionally, we used the simple predictor described in section 5.2. In this case, the output entropy was ~11400 bits. The table from figure F17 shows the output entropy and the loss of entropy with respect to the basic simulation scheme. The results of these simulations show an improvement over the reproduction of local features of the texture. However, global features like the orientation of patterns are not well preserved. The reason for this problem lays in the way the Hilbert scan performs the traversal of an image. In the following section we propose of modification on the way we scan the image to improve the results.

Simulation Scheme	Input parameters	Output entropy (bits)	Delta output entropy compared with Basic simulation scheme(bits)
Basic	N/A	544460	0
Quantization + Vectors	Vector size: 8	53720	~$4{,}91*10^5$
Quantization + Vectors + Side information	Vector size: 8 m = c = 6	39090	~$5{,}05*10^5$
Quantization + Vectors + Predictor	Vector size: 8 Predictor: previous pixel in the Hilbert scan	14300	~$5{,}30*10^5$
Quantization + Vectors + Predictor + Side information	Vector size: 8 Predictor: previous pixel in the Hilbert scan m = c = 6	11400	~$5{,}33*10^5$

Figure (F17): Number bits of output entropy for different simulation schemes. The input texture is shown in figure F18a and figure F19a. The simulated textures for Quantization+Vectors and for Quantization+Vectors+Side Information are shown in figure F18b and F18c. The simulated textures for Quantization+Vectors+Simple Predictor and for Quantization+Vectors+Simple Predictor + Side Information are shown in figures F19b and F19c.

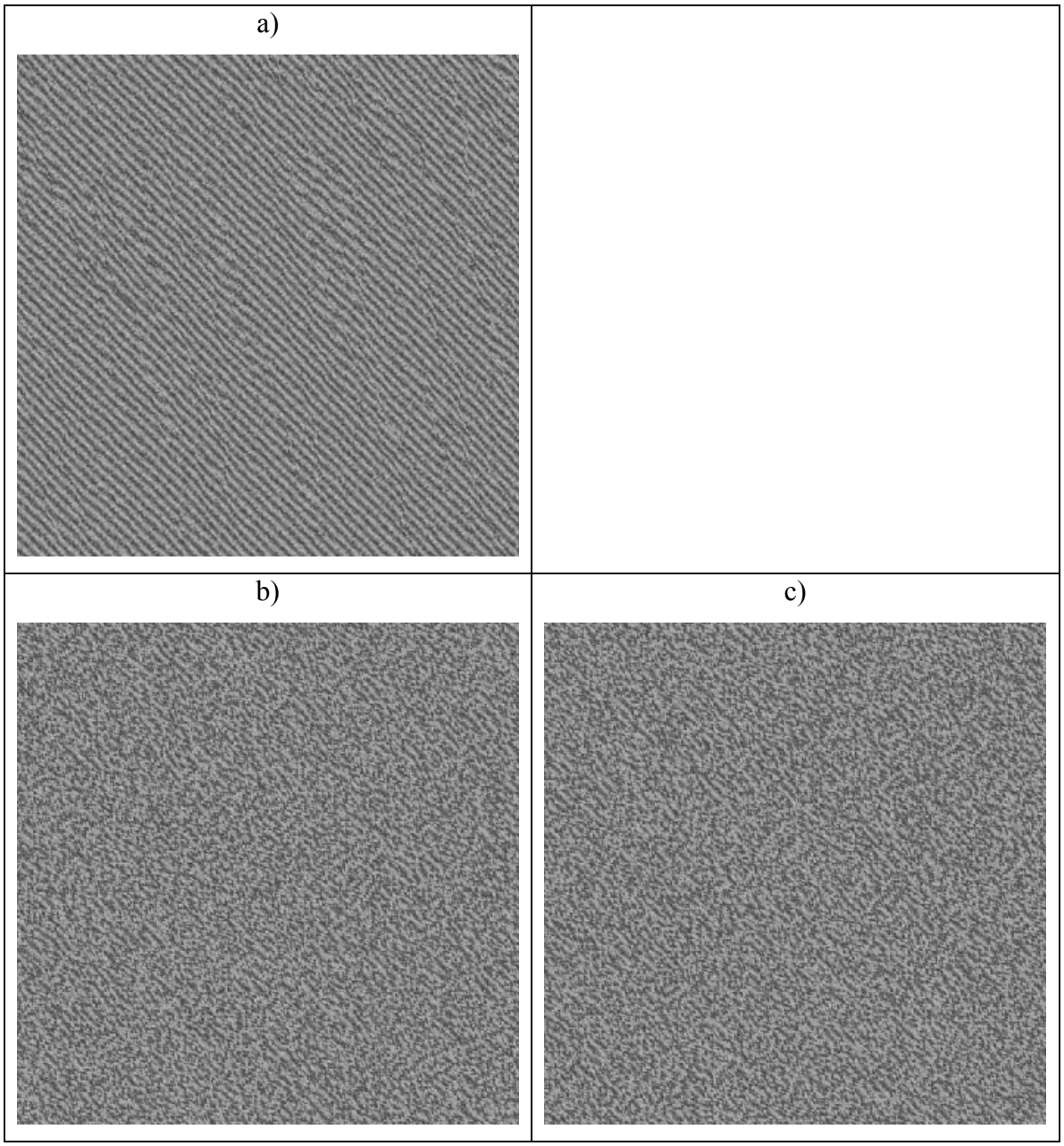

Figure (F18): Simulation example with side information.
Figure *a* shows the input texture. Figures *b* shows a simulation using vectored data (size 8) and quantization. Figure *c* shows a simulation using vectored data (size 8), quantization and usage of side information with parameters $m = c = 6$.

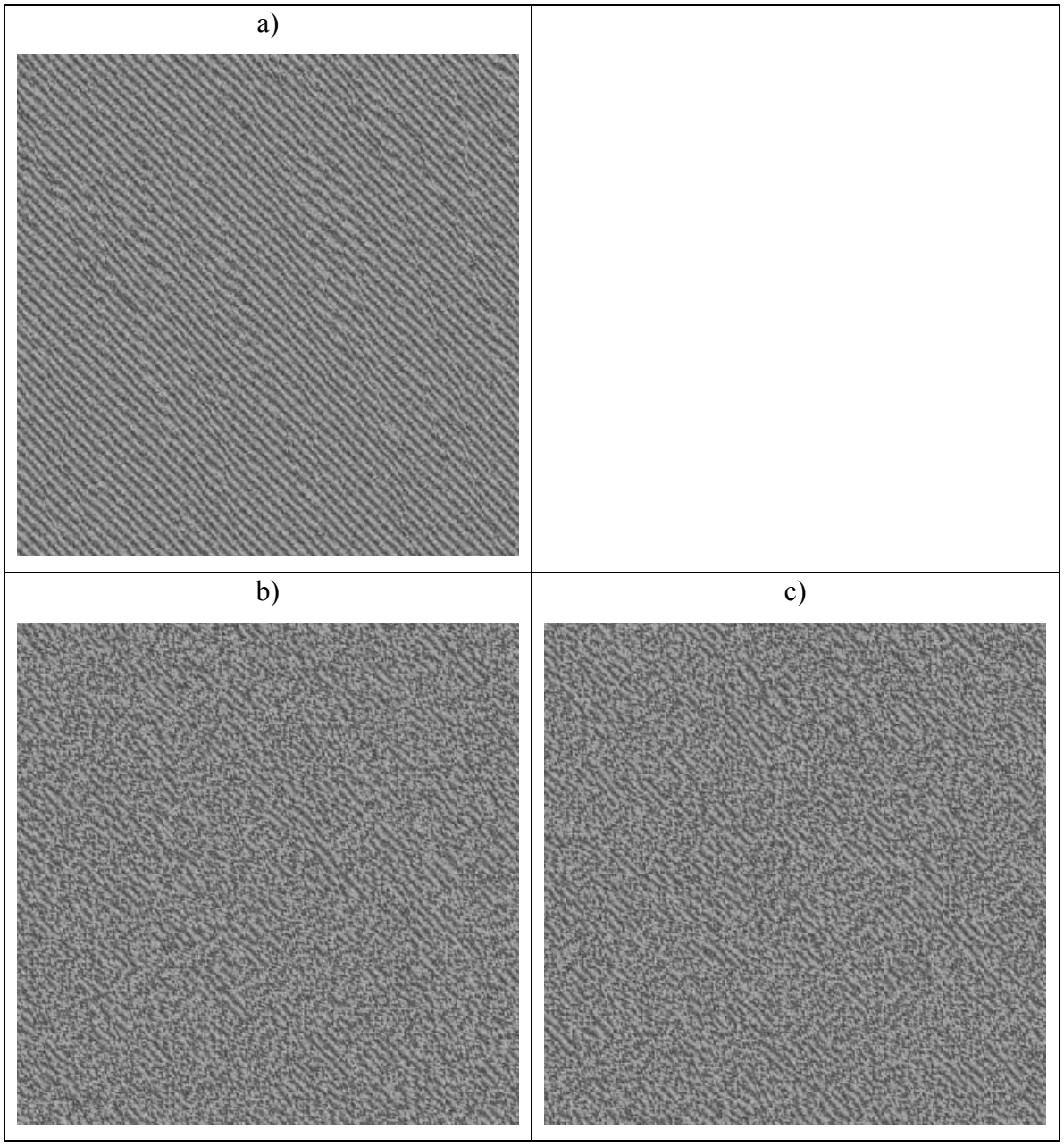

Figure (F19): Simulation example with side information.
Figure *a* shows the input texture. Figures *b* shows a simulation using vectored data (size 8), quantization and the simple pixel predictor. Figure *c* shows a simulation using and vectored data (size 8), quantization, the simple pixel predictor and usage of side information with parameters $m = c = 6$.

5.4 Improving the Hilbert scan

The simulation schemes we have introduced present some challenges in the way we scan the image to get a 1D sequence, as it was noted in the previous section. The following example will show these problems more evidently. The input texture shown in figure F20 consists of a set of horizontal lines. The texture from the right is a simulated texture using our basic scheme. The result of the simulation is not what we have expected. The problem lays in the way the usual Hilbert curve performs the scan. It was shown in section 3.4 that if we shift in time some part of the sequence the resulting image when we revert back de Hilbert scan shows portions of the original image with rotations in multiples of 90°. Now recall that the result of a simulation is a permutation of the phrases from the incremental parsing of the input sequence. Thus the simulated image will show patches of the original image but with different orientations. Note that this problem arises on other plane filling curves (i.e. Peano, Sierpinksi) as well. Simpler scan orders like a raster scan are also of no use mainly due to their poor locality properties.

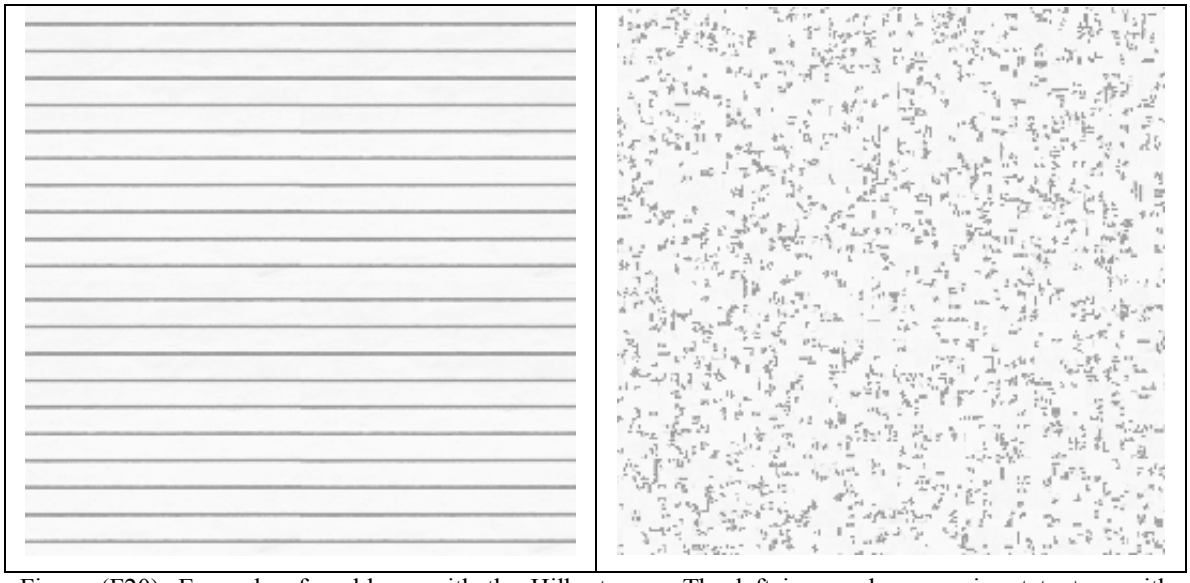

Figure (F20): Example of problems with the Hilbert scan. The left image shows an input texture with horizontal lines. The right image shows a simulated image using the basic simulation scheme.

Figure F21b show the result of applying the original Hilbert scan for the example we have been working with in the previous section using the simulation scheme presented in section 5.3 (using vectors of size 8, the simple pixel predictor and m = c = 6). The

input texture shows a diagonal pattern, (seen left-to-right it goes up-to-down). As it was noted before, the simulated image failed to capture some of the global features (orientation) of the input texture. It can be seen that in some regions the pattern is now inverted (i.e. it goes left-to-right, down-to-up). This is a direct consequence of the problem described above.

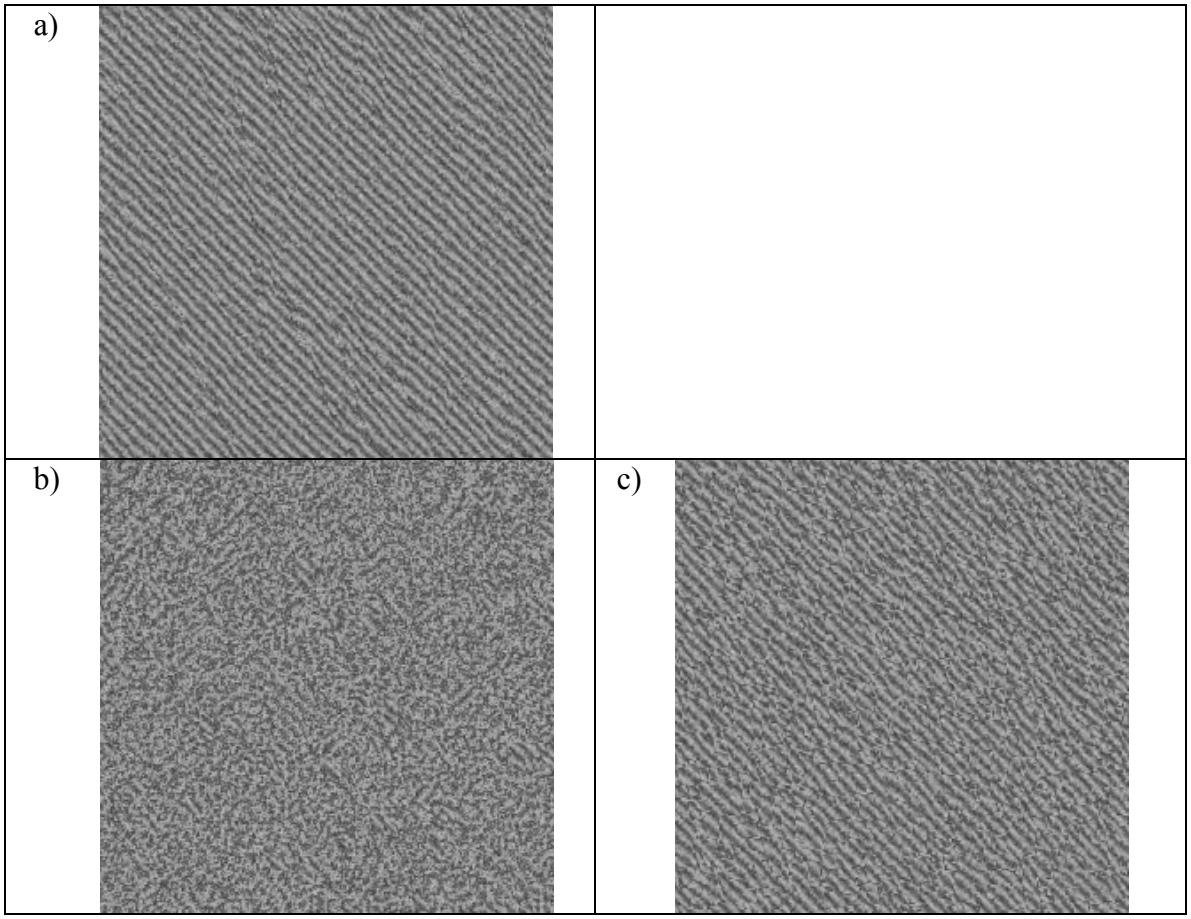

Figure (F21): Problems with the Hilbert scan in simulated images. Figure *a* shows the input texture. Figure *b* shows the simulated image using vectored data (size 8), quantization, the simple pixel predictor, and side information with parameters $m = c = 6$. Figure *c* shows the simulated image with the modified Hilbert scan and using the same parameters as figure *b*.

To avoid this problem we consider two possible scenarios. One would be to not map the 2D input image texture onto a 1D input sequence (i.e. don't use a plane filling curve at all). This would require to use some other two-dimensional extension to the incremental parsing algorithm for two dimension (other than the one presented by [28] which uses the Hilbert scan). Such extensions have been considered in [48], although

they have not become popular in image compression. They would also require reformulating the notion of UTC and the sampling algorithm. This approach presents interesting open problems for future work.

An alternative scenario would be to adjust the way the scan order works in such a way that a shift in time does not lead to rotations in the reconstructed image. This is the solution we have adopted in the current work. We have modified the plane filling scan algorithm as follows. For the sake of concreteness we assume we are working with the Hilbert scan, though this modification is applicable to any plane filling curve. We use a cursor **C** with coordinates (C_x, C_y) to traverse the image. Initially the cursor starts in one of the image's corners. As we pass through the image following the Hilbert curve we move the cursor accordingly. We use $\mathbf{D_i}$ to indicate the direction (up, down, left or right) in which the cursor moves at time index **i**. Thus, after applying the scan we end up with a 1-d input sequence **S** (just like the usual scan) and with another 1-d sequence **D** containing the direction the cursor moved at any instant of time. Then, to reconstruct a sequence back onto the image we apply the direction information to move across the image instead of the usual traversal made by the Hilbert scan. We start by locating the cursor C in the corner of the image. Then, as we consume the sequence S_i over the time index i, we put back the pixel's information in the location pointed by C. After that, we move the cursor in the direction pointed by D_i (instead of moving in the direction the Hilbert scan would go), and repeat again the process for the next element of the sequence (i.e. i = i+1).

When we apply the basic or the extended random sampling algorithm to get the permutation of the original sequence we also permute the sequence of $\mathbf{D_i}$. Thus, the curve used when applying the reconstruction algorithm is different from the curve used when scanning the input image. Also, there won't be any rotations of the subsequences (i.e. the relative position of two pixels from the same subsequence when located in the image will remain unchanged). However, this algorithm has an important drawback. The curve used to reconstruct the simulated texture is no longer guaranteed to be plane-filling; some image coordinates will be visited more than once (we refer to this as *collisions*) whereas others might never be visited. To alleviate this problem to some extent we feed the incremental parsing algorithm with the input texture more than once. Then, following the

ideas from the universal delay-limited simulation [33], when we reconstruct the image we use as much of the simulated sequence as needed to fill the output image (though, in practice, we 'consume' the full simulated sequence). Figure F21c show the output of a simulation from the input texture shown in figures F21a using the modified Hilbert scan with the extended simulation scheme with vector data of size 8, quantization, simple pixel predictor and side information with parameters m = c = 6. We can see in this simulation that the orientation of the output textures is now correct.

We would like to measure the number of different textures we can simulate using this modified Hilbert scan. The main challenge we face is that when we reverse the scan, a location in the image may be visited more than once. Thus, part of a previous outputted phrase may be occluded by new phrases. Although at this moment we have not been able to calculate with precision the number of different textures this scheme can simulate, we can at least give an upper bound. Assume we have an input texture and we apply a Hilbert scan obtaining a sequence x^n. Then we concatenate this sequence r times and denote this new sequence y^{rn}, i.e. $y^{rn} = x^n x^n \cdots x^n$ (r times). If we feed the incremental parsing algorithm with y^{rn} we can calculate the size of the UTC of y^{rn} using the formulas described in section 5.2. Let's denote $L(x^n)$ the size of the UTC of x^n. Therefore the sampling algorithm can produce $L(y^{rn})$ different sequences for the UTC of y^{rn}. After reversing the Hilbert scan with our modified procedure, the output texture will have the same size as the input image and, as we described above, some phrases will be overwritten by later phrases in the sampling algorithm. Therefore we suggest that the number of different textures our simulation scheme using the modified Hilbert scan can produce is upper-bounded by $L(y^{rn})$. Obtaining more accurate results is a matter of future work.

5.5 Wavelets smoothing

The left and center columns of Figure F23 show, respectively, some input textures and the result of the simulation scheme presented in section 5.3 with the modified Hilbert scan. As it can be clearly seen, there are noticeable artifacts ("*false contours*") mostly in the interface between the different patches. This is due to the fact that two patches which are 'together' in the simulated image may come from different places from the input texture. While the context preservation techniques presented in previous sections helped to preserve the context between phrases, this is clearly are not enough since the context preservation is inherently local. Moreover, due to the way the Hilbert scan traverses the image (making 90° turns), the snippets taken from the input texture tend to have sharp edges. Another important characteristic of textures is that they have features in different scales. For example, the texture in figure F22a shows wall of bricks. However if we look at a finer scale we can see the texture within each brick. Figure F22b shows a magnification of the patch inside the blue rectangle of figure F22a. The introduction of multi-scale analysis techniques would help us to capture the features of textures that are at different scales.

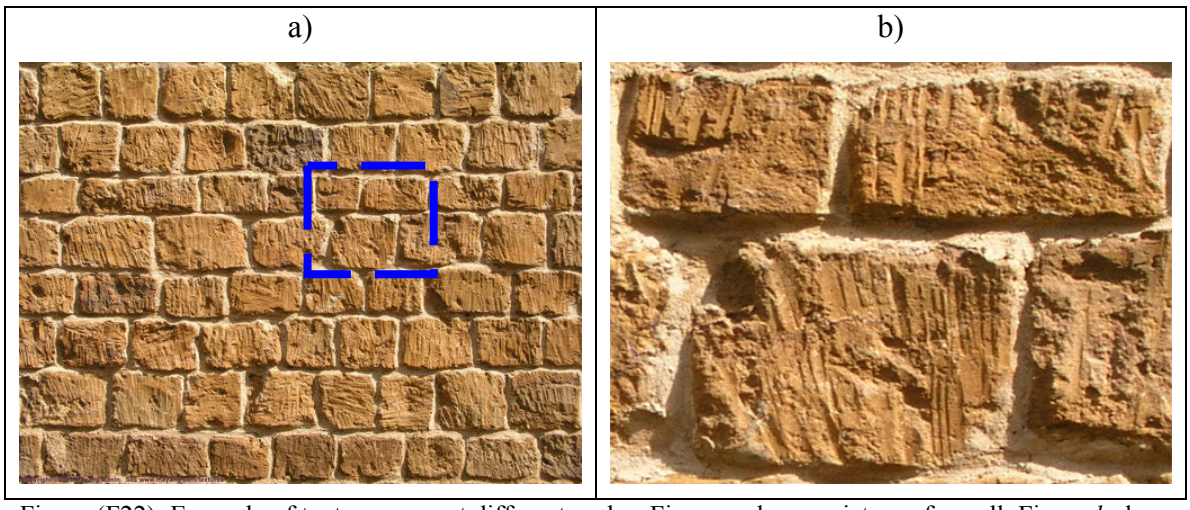

Figure (F22): Example of textures seen at different scales. Figure *a* show a picture of a wall. Figure *b* show the region of the wall contained inside the blue rectangle, where the features of texture within each brick can be seen.

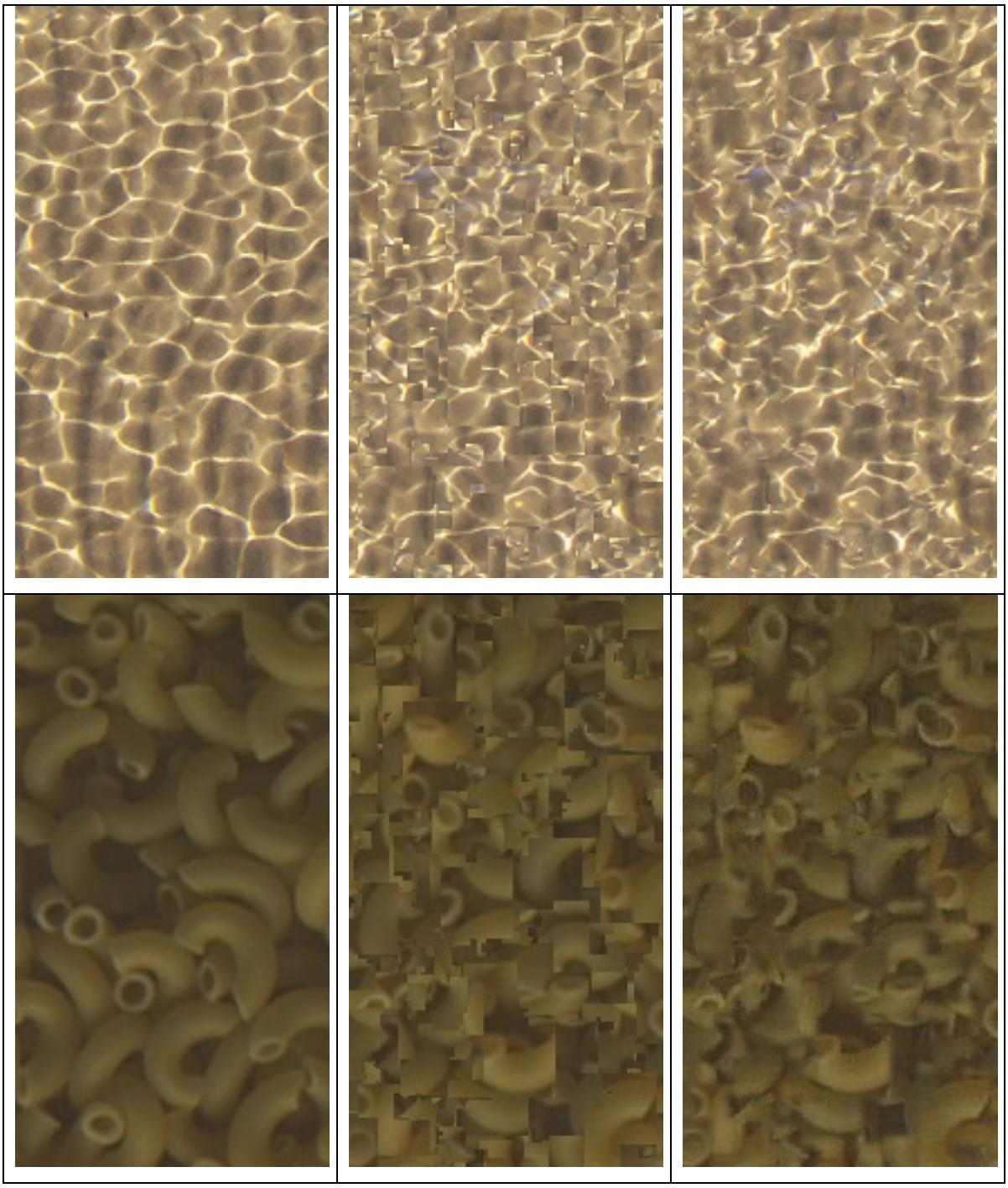

Figure (F23): Simulation results with/without wavelets smoothing. The left column contains the source textures (LakeTahoe, Pasta006), the center and right columns contains the simulated image without and with wavelets smoothing respectively.

In order to help reduce the false contours described above, we used the smoothing scheme described in figure A5. For the wavelet decomposition we used Simoncelli's steerable pyramid ([45, 46]) which was introduced in section 3.6.

Inputs: Source image texture

Outputs: Simulated image texture

1. Obtain the universal type class for the source image texture.
2. Randomly sample a new sequence from the universal type. This will be regarded as the permutation **p** over the original sequence.
3. Reconstruct the simulated texture. Obtain the wavelet decomposition for the simulated texture (steerable pyramid *A*)
4. Obtain the wavelet decomposition for the original texture (steerable pyramid *B*)
5. Traverse every band in the highest frequency scale (and the high frequency residue) of the steerable pyramid *B* to get the one-dimensional sequence of that band. Apply the permutation **p** over it, and insert the result back into the steerable pyramid *A*.
6. Collapse the steerable pyramid *A* to get the smoothed simulated texture.

Figure (A5): Texture simulation with wavelets smoothing

The algorithm in figure A5 works as follows. We first obtain the universal type class for the source texture (step 1) and get a simulated sequence as we would usually do (step 2). Let denote *p* to the index permutation of the incremental parsing (i.e. we are referring the index mappings that are taken in the permutation, this being independent of the actual values of the sequence). In steps 3 and 4 we obtain the wavelet decomposition of the simulated texture and the original texture. These are the steerable pyramid *A* and *B* respectively. Then in step 5, we start by applying a Hilbert scan over every band of the highest scale of the steerable pyramid *B*. Recall from the introduction of the steerable pyramid (in section 3.6) that the size of the highest scale bands is the same as the size of the input image. Therefore, the corresponding the sequences will be the same as the size of the input texture's sequence. Then, we apply the index permutation *p* to these sequences and revert back the Hilbert scan onto the corresponding band and scale of the steerable pyramid *A*. We apply the same procedure to the high frequency residue (which again has the same size as the input image). Upon finishing step 5, steerable pyramid *A* still has the coarser levels of the simulated texture's steerable pyramid but the band in the highest scale (and the high frequency residue) contains the data from the processed

pyramid *B*. We finish the procedure in step 6 by collapsing the resulting "hybrid" steerable pyramid to obtain a new simulated texture. This visually improved image removes most of the artifacts mentioned earlier. The right column of figure F23 shows the results of the application our simulation scheme with the wavelets smoothing algorithm. Figure F24 shows more examples of simulations using wavelets smoothing.

Figure (F24): Simulated texture mixtures. The left column shows input textures. The right column shows simulated textures using wavelets smoothing

6 Mixing Textures via Universal Simulation

In this section we address the problem of simulation of mixtures of two or more input textures. As it was described in previous sections, it is possible to use an algorithm for texture synthesis to build texture mixtures the following way: First generate an intermediate texture by copying all input textures one after the other in some fixed way. This intermediate texture has the features of all the source images. Then use the synthesis algorithm using the intermediate texture as input to generate new textures. Although this technique is very straightforward it generally gives bad results, especially when mixing textures with disparate features. Moreover, when mixing textures in this way, we have absolutely no control over the spatial distribution of each input texture over the simulated image because the synthesis algorithm has no knowledge about the fact that it is really working with a set of input textures. In other words, the algorithm still uses a single input texture and the fact that this input was generated by some preprocessing stage using many textures is unknown to it.

Therefore, we extend our framework to work with many input texture images. This way the algorithm has prior knowledge that what we want to simulate is a texture mixture so we can use this information to give better results. We assume to have N input textures, and a set of target mixture ratios. The set of *target mixture ratios* is a set $\{R_1,\cdots,R_N\}$ with $\sum_{i=1}^{n} R_i = 1$ where R_i denotes the desired proportion of the corresponding input texture in the output texture mixture. The simulation scheme, which is an extension of the algorithm A4 described in section 5.3, will generate a simulated texture mixture in which the k^{th} order statistics (for all k), approaches the weighed mixture of the k^{th} order statistics of each individual texture used in the mix. As in the case of previous sections, we are referring to the statistics of the individual textures using vectored data and quantization.

As with the simulation of single textures presented in the previous section, we can divide the process in two steps. We denote B_i the i^{th} input texture, and m_i the number of

pixels that B_i has. The first step consists in building the parsing tree for each of the N input sequences using the extended incremental parsing algorithm described in section 5.3. However, instead of using the whole image to feed the incremental parsing, we take a patch of $R_i m_i$ pixels from it. The patch we use from B_i is the one conformed by the first $R_i m_i$ pixels visited when we apply a Hilbert scan to B_i. In other words, to obtain the 1D sequence associated to B_i we stop the traversal of the texture when we have visited $R_i m_i$ pixels. After feeding the incremental parsing algorithm with the resulting sequences, we get N parsing trees ($T_1,...,T_N$), each one corresponding to their respective input texture. The second step (the mixture random sampling) is described in figure A6 and works as follows. We start with a set $\{T_1,\cdots,T_N\}$ of input parsing trees, a set $\{R_1,\cdots,R_N\}$ of target mixture ratios mixture proportions, a back reference tree I and an the total input sequence length n, $n=\sum_{i=1}^{N} R_i m_i$. The back reference tree is a combination of all the back reference trees resulting from the extended incremental parsing of the input textures. For the purpose of achieving the ratio of every texture we will use a set $\{C_1,\cdots,C_N\}$ of *auxiliary counters* which denotes the amount of data the algorithm has taken from every tree, and the current output sequence length L. Step 1 of the algorithm initializes the variables that we use. In step 2 we select randomly the tree that will be used to output the next phrase with a distribution that leads to the desired mixture ratio. Then, in step 3 we try to set a context for the selected tree using the side information collected in the extended incremental parsing. We traverse the tree choosing the next arcs in the same way we did in the basic sampling algorithm. Step 4 checks to see if an unused node has been reached. In that case, in step 4a we check to see if we have fully visited the selected tree. In that case we select a node at random that will give exactly the desired ratio for the selected tree (i.e. we output a tail phrase). In step 4b we output the phrase associated with the selected node. Then in step 4c we update C_i and L and if we have outputted the prescribed output sequence length n we finish the algorithm. At the end of this process, we obtain the simulated texture mixture with the prescribed input proportions.

Inputs: Set $\{T_i, 1 \le i \le N\}$ of LZ trees, set $\{R_i, 1 \le i \le N\}$ of target mixture ratios, back reference tree I, context depth c, total input sequence length n.
Output: Randomly sampled mixed sequence

1. Initialization
   a. Walk thru all the nodes of every tree $\{T_i, 1 \le i \le N\}$ marking them as unused, and for each node t set $U(t) = c_p(t)$. Mark the root (λ_i) of T_i as used and set $U(\lambda_i) = U(\lambda_i) - 1$.
   b. Walk thru all the nodes j of I. For every node v in the set of nodes of j check to see if the depth(v) > c. In that case replace v with the ancestor node v' along the path of v with depth(v') = c
   c. Set j at the root(μ) of I.
2. Select randomly an index tree h with distribution $\text{Prob}(h = k) = \frac{nR_k - C_k}{n - L}, 1 \le k \le N$, and set $t = \lambda_i$.
3. Try to set a context for T_i. Lets define the function $R(v)$ to be 1 if the node v has been used and 0 otherwise, and $U^* = \sum_{b \in set(v)} U(b).R(b)$, where $set_i(v)$ denotes the set of nodes in i which are rooted at λ_i.
   a. If no node is reachable(used), set $t = \lambda_i$
   b. Otherwise draw randomly a node t from the set of nodes of i with distribution $\text{Prob}(t = b) = \frac{U(b).R(b)}{U^*}, b \in set(i)$
4. If t is marked as unused or if $U(\lambda_i) = 0$:
   a. If $U(\lambda_i) = 0$, pick uniformly at random a node v of depth $nR_i - C_i$ from T_i, and set $t = v$.
   b. Output the phrase associated to t, mark t as used, set $U(t) = U(t) - 1$, set $C_i = C_i + \text{depth}(t)$.
   c. Let $l = min(\text{depth}(I), depth(t))$. Set $j = \mu$. Traverse j in I along the last l symbols of the phrase associated to t.
   d. Set $L = L + \text{depth}(t)$. If $L \ge n$ finish the algorithm.
   e. Otherwise go to step 2
5. Draw randomly a symbol a from Λ with distribution $\text{Prob}(a = b) = \frac{U(tb)}{U(t)}, b \in \Lambda$, set $U(t) = U(t) - 1$, set $t = ta$ and go to step 4.

Figure (A6): Mixture random sampling from multiple universal type classes

We now give a procedure to measure the richness of the simulations of our scheme to simulate texture mixtures. The procedure, which is an extension of the one described in section 5.1, works as follows. In step 2 of the sampling algorithm described in figure A6 we select the parsing tree we will use to output the next phrase. The probability that the j^{th} parsing tree is selected in the draw is $p_t = \frac{nR_j - C_j}{n - L}$. Once we have selected a parsing tree and we select the restarting node inside that tree, the algorithm continues to traverse the tree until it reaches a node which has not been visited before.

Now recall that in step 5 of the algorithm we draw the next node to visit (*tv*) in the tree in which the probability of selection of the chosen node is determined by $p_d = \frac{U(tv)}{U(t)}$. Thus, after outputting a phrase *q*, the probability of selection of that phrase is $p_q = p_t \prod_{\left\{\begin{array}{l}\text{every node draw}\\ \text{carried out to select } q\end{array}\right\}} p_d$. Finally, we can calculate the probability of the output sequence p_s by multiplying the probabilities of selection of all the phrases. Thus, $p_s = \prod_{\{\text{every phrase}\}} p_q$. It can be shown that the proof of uniformity of the output distribution from [42] extends to the algorithm of figure A6. In [42], when traversing the parsing tree, we select a branch from the current node at random with probability proportional to the number of phrases remaining unused in the subtree rooted at the target of the branch. Here, the same argument applies, as the random selection in step 2 chooses one of the texture trees with probability proportional to the number of its phrases that remain unused, and the rest of the random selection steps proceed as in [42]. Consequently $p_s = \frac{1}{L}$ where *L* is the number of different sequences our simulation scheme can produce. This number is usually very big, so we take logarithms of the probabilities we compute and sum the contributions of entropy for every draw. At the end we get the entropy of the full simulation.

The false contours described in section 5.5 also appear in these mixtures. Hence we extend the wavelets smoothing algorithm to work with multiple input textures. In this case though, we have to perform the permutation over the higher scales of the steerable pyramid of every input texture, and mix them accordingly onto the output steerable pyramid. We also use the modified Hilbert scan described in section 5.4 to deal with the orientation issues described there. Figures F26-F28 shows some texture mixture examples. The simulation was done using quantization, vectors of size 6 and side information (with $m = c = 6$), and we applied the wavelet smoothing to the resulting textures. In the examples, we start from a single input texture, and progressively decrease its proportion over a second input texture. The table from figure F25 shows the output entropy for the simulation of the mixture of the input images shown in figure F26

(BarleyRice009 and BarleyRice010) with the proportions ranging [0.1i, 1-0.1i] using quantization, vectors of size 8, target sequence length of 192 and side information (with $m = c = 6$), without the modified Hilbert scan.

Figure F29 shows the simulation texture mixtures from three input textures, using the same parameters for the simulation scheme. In this simulations we mixed three textures from the Oulu texture database (pasta001, pasta005, pasta006) [36] in a variety of different proportions. More examples of texture mixture simulations can be found in [6]. In some of them, two input textures were mixed in proportions [0.025i, 0.025(40-i)] for $0 \leq i \leq 40$. The resulting frames were combined into video clips showing the mixture progression.

Input proportions: [A, B] Texture A: BarleyRice009 Texture B: BarleyRice010	Output entropy (bits)
[100, 0]	13060
[90, 10]	12880
[80, 20]	11400
[70, 30]	11640
[60, 40]	8400
[50, 50]	9050
[40, 60]	9770
[30, 70]	10440
[20, 80]	13080
[10, 90]	13570
[0, 100]	14180

Figure (F25): Number bits of output entropy for different proportions of texture mixture. The input textures are shown in the first image of figure F26 (BarleyRice009) and in the image of figure F28 (BarleyRice010). The left column shows the proportions of each input texture in the mixture. The right column shows the output entropy of the simulation. For these We used the simulation scheme for texture mixtures, using quantization, vectors of size 8, the simple predictor and side information with m=c=6, without the modified Hilbert scan.

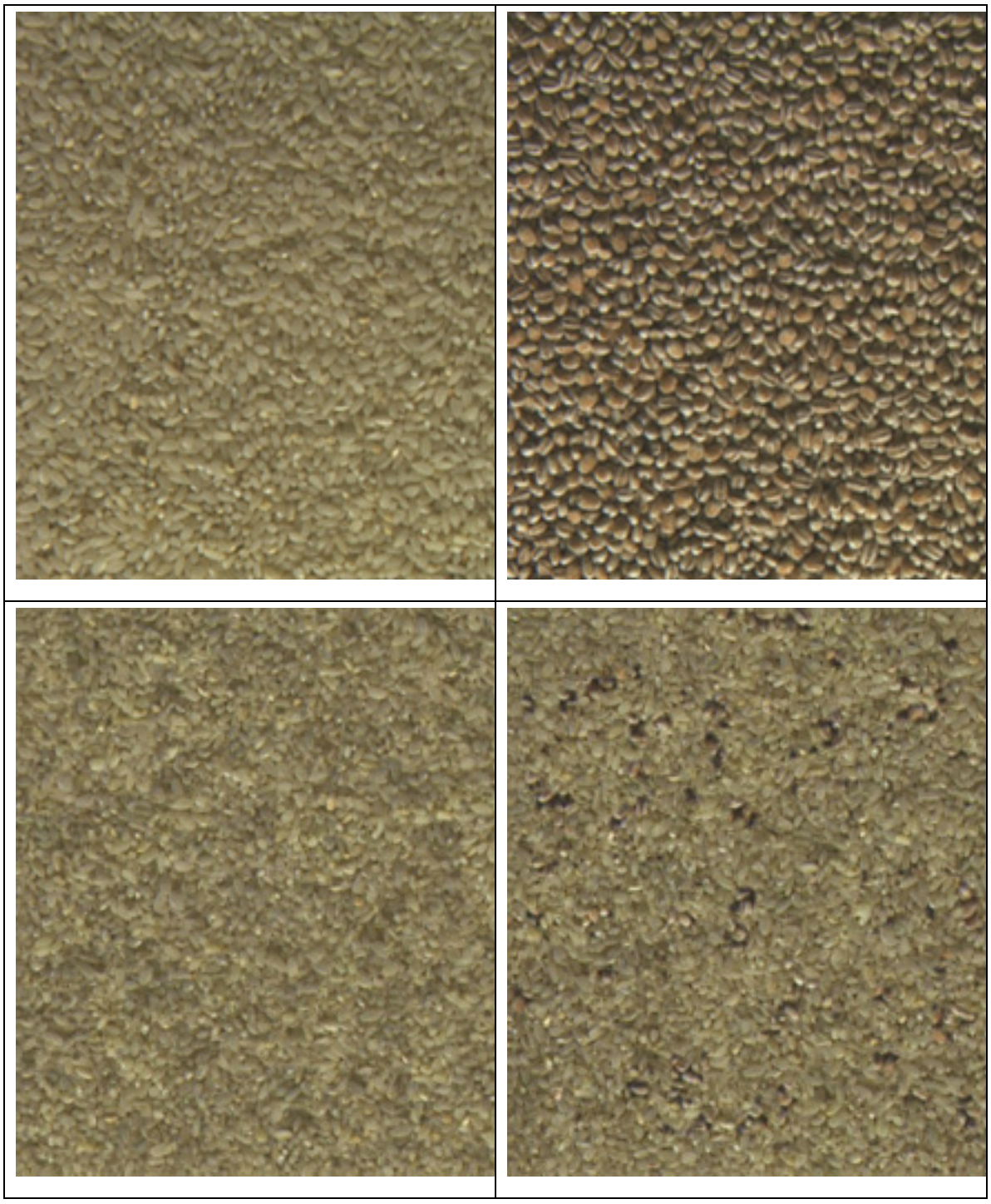

Figure (F26): Simulated texture mixtures. The top row contains the input textures (BarleyRice009, BarleyRice010). In the bottom row at the left: a simulated texture of BarleyRice009, and at the right: a simulated texture mixture (proportions 10-90).

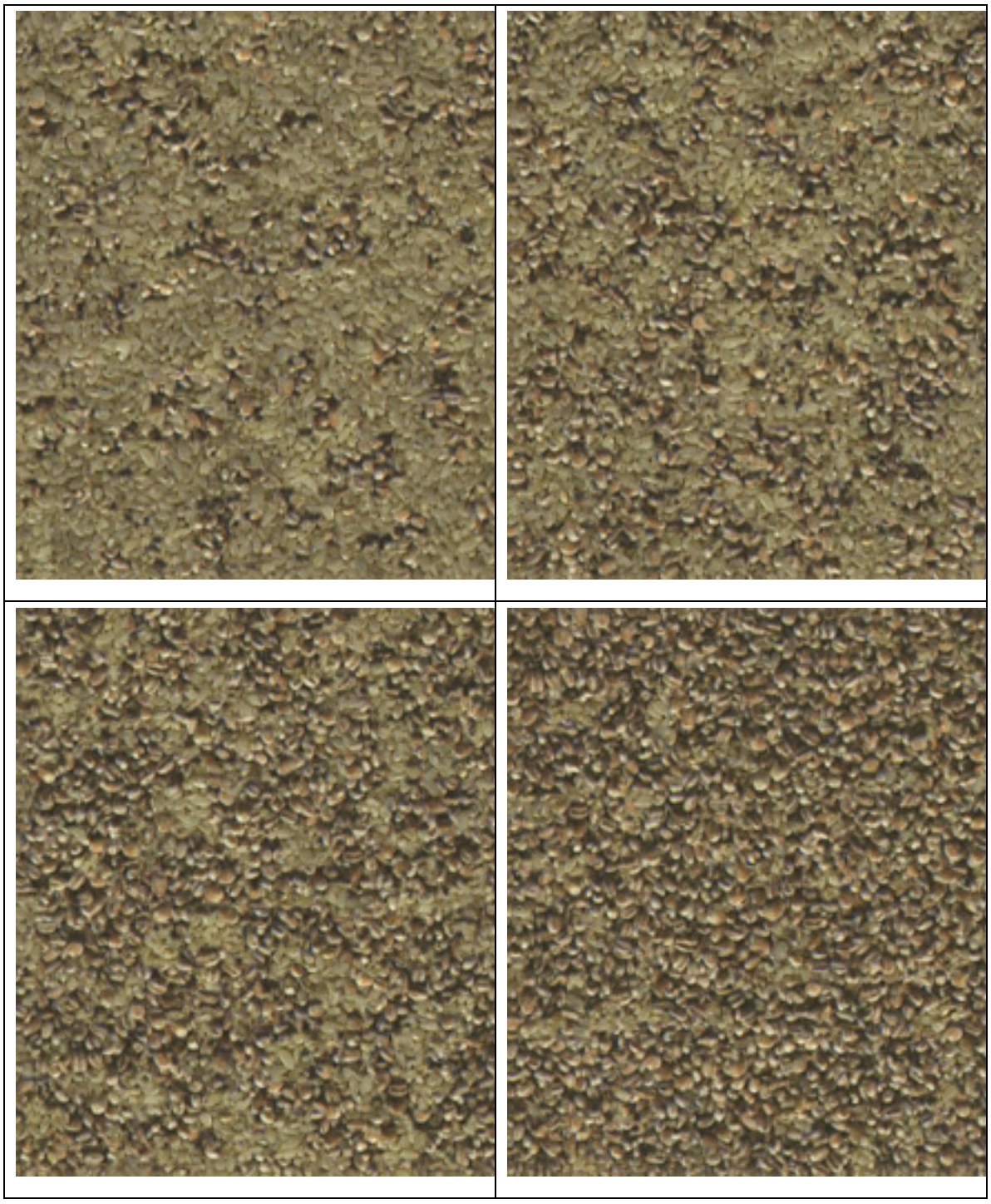

Figure (F27): Simulated texture mixtures. The top row at the left: proportions 30-70; at the right: proportions 50-50. The bottom row at the left: proportions 70-30; at the right: proportions 90-10.

Figure (F28): Simulated texture mixtures. A simulated texture of BarleyRice010.

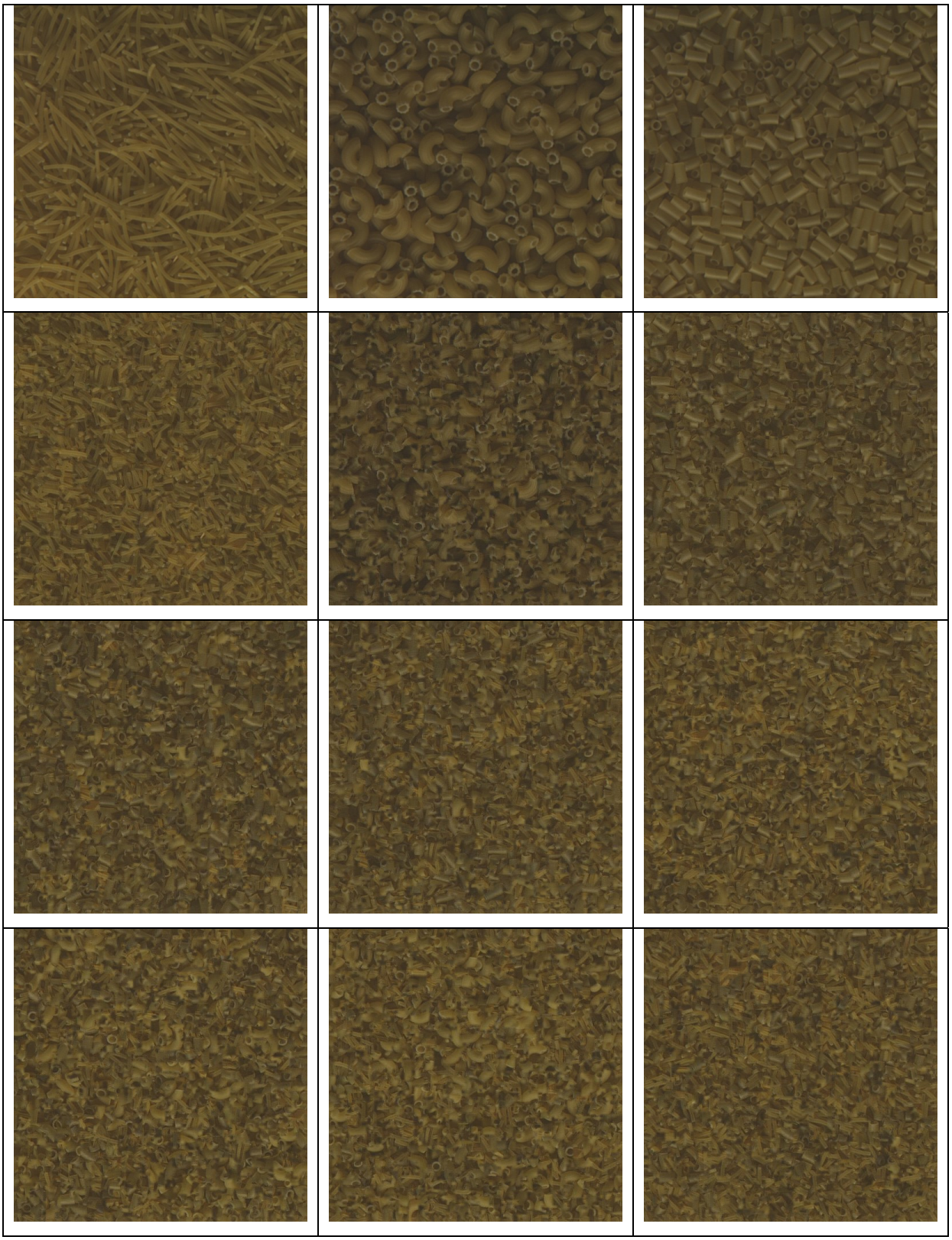

Figure (F29): Simulated texture mixtures. The top row shows the input textures. The proportions for the simulations shown in the remaining rows have, from left to right and top to bottom, 100-0-0, 0-100-0, 0-0-100, 10-30-60, 25-25-50, 33-33-33, 25-50-25, 30-60-10, and 60-10-30, respectively.

7 Conclusions and Future Work

In this section we summarize the results obtained in this work and propose several directions for future research.

7.1 Results

We have introduced the use of universal type classes [42] based on the incremental parsing of Ziv and Lempel [27] in texture simulation and mixing. We believe that these information-theoretic tools add great value to these applications, as they provide a formal approach to the problem of texture simulation. These tools provide a way to measure the "unpredictability" of the simulated output texture, or, equivalently, the number of possible output textures that comply with a given notion of "similarity". Thus, a trade-off is established between similarity and output entropy, which we highlight in various examples in this work. Previous works in our direction ([33, 42]) settled the mathematical background and addressed mainly the case of one-dimensional input sequences and studied image texture simulation in a basic form. However, for continuous-tone images of practical sizes, the basic approach does not give satisfactory results, due to the issues discussed in section 5. We have shown how some of these issues can be addressed, by means of various modifications to the basic scheme, which result in schemes along the similarity-entropy trade-off mentioned above. Clearly, further refinements and improvements are possible along that trade-off, but we believe that the initial results shown in this work show the promise of the approach.

We began our work using the simulation scheme described in [42]. With this simulation scheme, two textures are said to be "similar" if the sequences resulting from the Hilbert scan belong to the same UTC. In other words, we say they are similar if they are statistically indistinguishable (in the limit) by finite memory models of any order. The universal simulation does a faithful job at capturing the constraints of the implicit statistical model used, and delivers results that shows how the chosen model 'sees' the

texture. In a first approach we tried to see if those statistics were enough to capture the essential features of the input texture. For every simulation scheme there is always a trade-off between the ability to capture the essential features of an input texture and the richness or "unpredictability" of the simulation. In section 5.1 we gave a procedure to measure the number of different textures this simulation scheme can synthesize for a given input texture, therefore giving a numerical value for the trade-off described above. The results of this scheme applied to continuous-tone textures gave poor results. This suggested that our similarity criterion had to be strengthened. In section 5.2 we discussed the 'context dilution problem', which is one of the problems of using the Lempel-Ziv incremental parsing algorithm as a context modeling tool. To solve this problem, we introduced the use of quantization and use of small vectors (typically patches of 4 or 8 pixels). As a consequence our simulation scheme was able to capture better the features of the texture, and at the same time reduced the unpredictability of the simulations. In section 5.3 we discussed another problem of using the incremental parsing algorithm as a context modeling tool, the loss of context occurring at phrase boundaries. We extended our simulation scheme to collect side information in an incremental parsing algorithm, and used that information as an aid to preserve context in an extended sampling algorithm. This gave more visually coherent simulations again at the expense of a decrease in the entropy in the output. In section 5.4 we addressed an issue with the Hilbert scan used to get a 1D sequence from an input texture that had the side-effect of giving simulated images with patches with wrong orientation. The simulated images produced by the simulation schemes described in sections 5.1-5.4 contain visual artifacts like blocking and false contours that distinguishes them from the original textures. To improve the visual quality of the simulated textures and to prevent some of these artifacts we extended our simulation scheme with the use of the steerable pyramid decomposition.

We evaluated the trade-off between the unpredictability and the visual quality of our simulations. This can be seen in the tables which show the output entropy and the figures that illustrate the results of our simulation schemes after the additional constraints added to narrow our 'similarity' criterions were introduced.

Synthesis of single source input textures has been widely addressed with success in literature. However, creating texture mixtures by applying a synthesis procedure to

several input images has not been as extensively studied. The second part of this work extends the techniques and formalism developed for simulation to the problem of texture mixtures. According to our setting, the k^{th} order statistics of a mixture show approach, asymptotically, the weighed mixture of the k^{th} order statistics of each individual texture used in the mixing. It is of practical interest to create mixtures with prescribed proportions of the input textures. Therefore, in section 6 we modified our simulation scheme to work with several input textures and to synthesize texture mixtures with the desired proportions. Output textures created with our simulation schemes can be seen in section 6 of this thesis and in the web page set up for this purpose [6].

7.2 Future Work

To end this work, we propose several directions for future research.

- **Extend the universal simulation scheme to work natively with 2D sequences.**
  In section 5.4 we dealt with a problem with the way the plane filling curves scan an input image. The use of the classic Hilbert scan (and related plane filling curves) in our simulation schemes, which at this point is needed to convert the 2D sequence into a 1D input sequence, produced simulated textures that had patches of the original image with different orientations. We partially solved that problem by modifying the usual scan procedure with a directional version. We would like to extend the universal simulation scheme to work natively with the 2-d sequences provided by the input textures.

- **Integrate more thoroughly the use of multi-resolution analysis tools.**
  In this work we have used the steerable pyramid wavelets decomposition as an aid to resolve the false contours visible in the simulated textures. We would like to incorporate more thoroughly these techniques to capture the features of a given texture at different orientations/scales. Here we might extend our use of vectors to form them with the coefficients from many bands/scales of a wavelets decomposition.

- **Study other models of universal type classes**
  Throughout this work we have used universal type classes based on the Lempel and Ziv incremental parsing algorithm. However, we can define a model for universal type classes based on other universal compression algorithms (e.g. prediction by partial matching, context trees). These kinds of models for UTC have not been studied yet, and are a matter of further investigation.

- **Improve the simulation of texture mixtures**
  As we have mentioned in section 6, we created a series of texture mixtures of two input textures with proportions ranging from 0 to 100% within each input texture.

With the resulting set of output textures we created a small video to show how the mixture slides from one input texture to the other. However, no temporal constraints were taken on these mixtures which lead to discontinuities between each frame and the next. A possible future work might involve the creation of time varying textures ([1], [2]).

References

[1] Z. Bar-Joseph, "Statistical Learning of Multi-Dimensional Textures," The Hebrew University of Jerusalem, MSc. thesis. 1999.

[2] Z. Bar-Joseph, R. El-Yaniv, D. Lischinski, and M. Werman, "Texture Mixing and Texture Movie Synthesis using Statistical Learning," in *Proc. IEEE Transactions on Visualization and Computer Graphics* 2001, pp. 120–135.

[3] M. Bertalmio, G. Sapiro, V. Caselles, and C. Ballester, "Image inpainting," in *Proc. of the 27th Annual Conference on Computer Graphics and interactive Techniques*, New Orleans, Louisiana, 2000, pp. 417-424.

[4] J. S. D. Bonet and P. Viola, "A Non-parametric Multi-Scale Statistical Model for Natural Images," in *Proc. 1997 conference on Advances in neural information processing systems*, Denver, Colorado, 1997, pp. 773-779.

[5] P. Brodatz, "Textures: A Photographic Album for Artists and Designers," Dover, New York 1996. Electronic resource. Available [online] at

[6] G. Brown, "Thesis examples web page," 2005. Electronic resource. Available [online] at *http://www.fing.edu.uy/~gbrown/universal-simulation.html*.

[7] B. Carpentieri, M. J. Weinberger, and G. Seroussi, "Lossless compression of continuous-tone images," *Proceedings of the IEEE*, vol. 88, no. 11, pp. 1797-1809, Nov. 2000.

[8] B. Carpentieri, M. J. Weinberger, and G. Seroussi, "Lossless compression of continuous-tone images," *Proceedings of the IEEE*, vol. 88, no. 11, pp. 1797-1809, 2000.

[9] T. M. Cover, "Enumerative source coding," *IEEE Transactions on Information Theory*, vol. 19, no. 1, pp. 73-77, Jan. 1973.

[10] T. M. Cover and J.A.Thomas, *Elements of Information Theory*. New York: John Wiley & Sons, Inc., 1991.

[11] I. Csiszár, "The Method of Types," *IEEE Transactions on Information Theory*, vol. 44, no. 6, pp. 2505-2523, Oct. 1998.

[12] I. Csiszár and J. Körner, *Information Theory: Coding Theorems for Discrete Memoryless Systems*. New York: Academic, 1981.

[13] S. Deorowicz, "Universal lossless data compression algorithms," Silesian University of Technology, Phd. thesis. 2003.

[14] A. Dumitras and B. G. Haskell, "An encoder-decoder texture replacement method with application to content-based movie coding," *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 14, no. 6, pp. 825-840, Jun. 2004.

[15] A. A. Efros and T. K. Leung, "Texture Synthesis by Non-parametric Sampling," in *Proc. IEEE International Conference on Computer Vision*, Corfu, Greece, Sep. 1999, pp. 1033.

[16] T. F. El-Maraghi, "An Implementation of Heeger and Bergen's Texture Analysis/Synthesis Algorithm," Dept. of CS, University of Toronto, Toronto, Ontario, Canada, Sep. 1997. Available [online] at *http://www3.cc.gatech.edu/classes/AY2004/cs4495_fall/Materials/Texture_El_Maraghi.pdf*.

[17] J. M. Francos, A. Z. Meiri, and B. Porat, "A unified texture model based on a 2-D Wold like decomposition," *IEEE Transactions on Signal Processing*, vol. 41, no. 8, pp. 2665-2678, Aug. 1993.

[18] W. T. Freeman and E. H. Adelson, "The Design and Use of Steerable Filters," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 13, no. 9, pp. 891-906, 1991.

[19] G.Gimel'farb, L. V. Gool, and A.Zalesny, "To FRAME or not to FRAME in Probabilistic Texture Modelling," in *Proc. 17th International Conference on Pattern Recognition (ICPR'04)*, Aug. 2004, pp. 707-711.

[20] R. C. Gonzales and R. E. Woods, *Digital Image Processing*, 2nd ed. New Jersey: Prentice Hall, 2002.

[21] C. Gotsman and M.Lindenbaum, "On the Metric Properties of Discrete Space-Filling Curves," *IEEE Transactions on Image Processing*, vol. 5, no. 5, pp. 794-797, 1996.

[22] A. Graps, "An Introduction to Wavelets," *IEEE Computational Science and Engineering*, vol. 2, no. 2, pp. 50-61, 1995.

[23] D. J. Heeger and J. R. Bergen, "Pyramid-Based Texture Analysis/Synthesis," in *Proc. SIGGRAPH '95*, 1995, pp. 229-238.

[24] A. Hertzmann, C. E. Jacobs, N. Oliver, B. Curless, and D. H. Salesin, "Image Analogies," in *Proc. the 28th International Conference on Computer Graphics and Interactive Techniques*, Los Angeles, CA, 2001, pp. 327-340.

[25] A. K. Jain, *Fundamentals of Digital Image Processing*. New Jersey: Prentice Hall, 1989.

[26] G. G. Langdon, "A Note on the Ziv-Lempel Model for Compressing Individual Sequences," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 284-287, Mar. 1983.

[27] A. Lempel and J. Ziv, "Compression of individual sequences via variable-rate coding," *IEEE Transactions on Information Theory*, vol. 24, pp. 530-536, 1978.

[28] A. Lempel and J. Ziv, "Compression of Two-Dimensional Data," *IEEE Transactions on Information Theory*, vol. 32, no. 1, 1986.

[29] A. Lempel and J. Ziv, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, 1977.

[30] S. Li and J. Shawe-Taylor, "Comparison and Fusion of Multiresolution Features for Texture Classification," University of Southhampton, UK, 2004.

[31] D. MacKay, *Information Theory, Inference, and Learning Algorithms*: Cambridge University Press, 2003.

[32] S. Mallat, *A Wavelet Tour of Signal Processing*: Academic Press, 1998.

[33] N. Merhav, G. Seroussi, and M. J. Weinberger, "Universal Delay-Limited Simulation," in *Proc. ISIT05*, 2005.

[34] N. Merhav and M. J. Weiberger, "On universal simulation of information sources using training data," *IEEE Transactions on Information Theory*, vol. 50, no. 1, pp. 5-20, 2004.

[35] G. Motta, E. Ordentlich, I. Ramirez, G. Seroussi, and M. J. Weinberger, "The DUDE framework for continuous tone image denoising," in *Proc. IEEE International Conference on Image Processing*, Genova, Italy, Sep. 2005, pp. 345-348.

[36] T. Ojala, T. Mäenpää, M. Pietikäinen, J. Viertol, J. Kyllönen, and S. Huovinen, "Outex - New framework for empirical evaluation of texture analysis algorithms," in *Proc. 16th International Conference on Pattern Recognition*, Quebec, Canada, 2002, pp. 701-706, Available [online] at http://www.outex.oulu.fi.

[37] P. Peebles, *Probability, Random Variables and Random Signal Principles*, 4th ed. Boston: McGraw-Hill Inc, 2001.

[38] A. Perez, S. Kamanta, and E. Kawaguchi, "Peano Scanning of Arbitrary Size Images," in *Proc. IEEE International Conference of Pattern Recognition*, 1992, pp. 565-568.

[39] J. Portilla and E. Simoncelli, "A Parametric Texture Model Based on Joint Statistics of Complex Wavelet Coefficients," *International Journal of Computer Vision* vol. 40, pp. 49-70, 2000.

[40] J. Rissanen, "A universal data compression system," *IEEE Transactions on Information Theory*, vol. 29, no. 5, pp. 656-664, Sep. 1983.

[41] G. Seroussi, "On the number of t-ary trees with a given path length," Hewlett-Packard Laboratories Technical Report HPL-2004-127, July 2004. Available [online] at http://arxiv.org/abs/cs.DM/509046. To be published in *Algorithmica*.

[42] G. Seroussi, "On Universal Types," *IEEE Transactions on Information Theory*, vol. 52, no. 1, pp. 171-189, Jan. 2006.

[43] G. Seroussi, "On Universal Types and Simulation of Individual Sequences," in *Theoretical Informatics: 6th Latin American Symposium, Lecture Notes in Computer Science*, M. Farach-Colton, Ed. Berlin: Springer-Verlag, 2004, pp. 2976.

[44] E. Simoncelli, "The steerable pyramid." Electronic resource. Available [online] at http://www.cns.nyu.edu/~eero/steerpyr/.

[45] E. Simoncelli and W. Freeman, "The steerable pyramid: a flexible architecture for multi-scale derivative computation," in *Proc. IEEE International Conference on Image Processing*, 1995, pp. 444-447.

[46] E. Simoncelli, W. Freeman, E. Adelson, and D. Heeger, "Shiftable multiscale transforms," *IEEE Transactions on Information Theory*, vol. 38, no. 2, pp. 587-607, 1992.

[47] E. Simoncelli and J. Portilla, "Texture characterization via joint statistics of wavelet coefficient magnitudes," in *Proc. 5th IEEE International Conference on Image Processing*, 1998.

[48] J. A. Storer, "Lossless Image Compression using Generalized LZ1-Type Methods," in *Proc. 1996 Data Compression Conference*, Snowbird, Utah, Mar. 1996, pp. 290-299.

[49] M. J. Weinberger, N. Merhav, and M. Feder, "Optimal Sequential Probability Assignment for Individual Sequences," *IEEE Transactions on Information Theory*, vol. 40, no. 2, pp. 384-396, Mar. 1994.

[50] M. J. Weinberger, G. Seroussi, and G. Sapiro, "The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS," *IEEE Transactions on Image Processing*, vol. 9, no. 8, pp. 1309-1324, Aug. 2000.

[51] S. C. Zhu, Y. N. Wu, and D. Mumford, "FRAME: Filters, random field and maximum entropy: Towards a unified theory for texture modeling," *International Journal of Computer Vision* vol. 27, no. 2, pp. 107-126, Mar. 1998.

List of Figures