Informational Texture Synthesis

Sylvain Lefebvre, Li-Yi Wei, Connelly Barnes

Informational Texture Synthesis

Sylvain Lefebvre, Li-Yi Wei, Connelly Barnes

June 28, 2021

Abstract

This paper presents a novel technique to embed messages into textures synthesized from tiles with boundary constraints. The messages are encoded in the choices that the synthesis algorithm normally makes at random during the synthesis process. The layout constraints between tiles provide additional robustness.

Images embedding messages are not distinguishable from images that are not embedding any message. The random sequence seed used during synthesis can act as a password.

1 Change log

This is an ongoing project research, the draft will be updated as we progress.

- 2021-06-28 Clarified the introduction.
- 2018-03-07 Better encoding scheme, robust decoding, preliminary results on encoding capability and robustness.
- 2018-02-13 Updated the draft with some previous work.
- 2018-02-08 Initial document.

2 Introduction

This document is a work in progress and to keep it short and to the point we skip introductory matters and focus instead of the main principle.

The synthesis algorithm described here follows an implementation similar to the *wave function collapse* (WFC) algorithm `https://github.com/mxgmn/WaveFunctionCollapse`, which builds upon the model synthesis algorithm [1]. We strongly encourage the interested reader to first get familiar with these methods. In particular [1] introduces the key idea of constraint propagation for discarding labels, which is central to the synthesis algorithm and not explained in this document.

3 Previous work

There are a number of previous work considering embedding information in synthesized textures. These works focus on 'stochastic' textures, which content is not globally organized (e.g. images of grass, pebbles, concrete, carpets). [2] proposes to encode data in synthesized textures by constraining the color of pixels spread with a specific pattern. Synthesis – inpainting – is then performed to fill-in gaps around these points. [3] encodes both the message and the exemplar into the final image. The exemplar is stored in pre-positioned square patches which locations are produced from a secret key. Synthesis is performed around these using image quilting [4]. Whenever a patch can be added, multiple candidates are considered. The candidates are ranked by how well they fit neighboring patches in the overlap region, and the rank of the selected candidate is used to encode data. [5] also relies on image quilting for synthesis around pre-positioned square patches which locations are produced by a secret key. The choice of which patch appears at each location encodes data. The authors note that approaches using patches are more robust to e.g. JPEG compression than those using pixels.

These approaches focus on synthesizing unstructured, stochastic patterns. Our technique synthesizes a different type of texture patterns which are described by tiles (square images) with assembly constraints [1]. Instead of predetermining the location of a set of tiles, we hide the data into the pseudo random sequence of choices encountered during synthesis.

4 Encoding

In the remainder it is to be understood that 'random' means 'pseudo-random using a sequential, deterministic generator'. Therefore, any random sequence produced during the algorithm can be exactly reproduced using the same starting seed.

The core principle is to layout square tiles to form a large texture. Each tile can be seen as a label (the tile id). Tiles cannot be freely placed: constraints define how tiles can be neighboring (e.g. tiles A and B may only be placed with A to the left of B). This lets artists paint pattern pieces onto the tiles which, once assembled while enforcing all constraints form a consistent pattern, e.g. a network of tubes.

Our variant of the algorithm goes as follows:

1. Initialize an array S of size $W \times H$, where each entry contains a vector of T booleans, with T the number of possible tiles. All vectors are initialized to *true*. Each boolean represents the possibility that a particular tile appears at a particular location. Initially everything is possible.

2. Visit each location of S in some *fixed* order. For instance, one can use scanline ordering (left, right then top, bottom).

3. At each location, consider the vector of possibilities. If a choice exists (e.g. more than one tile is possible), we have an opportunity to embed information. We call such a location along the sequence a *coding site*. For now let us assume a naive approach which selects the first choice to represent '0' and any other choice to represent '1'. Once the choice is made, set all other entries in the vector to $false$.

4. Propagate the constraints to cancel impossible choices in neighboring cells. This step is unchanged compared to the original algorithm [1].

5. Repeat until all cells of S have been visited. After this, a choice has been made everywhere and we can produce the final image by assembling the tiles following S.

This process can encounter a contradiction at step 3: in such cases no possibilities exist in the visited cell, due the propagation of constraints from previously selected tiles. If this happens, the algorithm given above would be stuck: the same message always produces the same choices. We discuss in Section 4.2 how to introduce randomness in the process such that the algorithm can be launched again and make different choices to recover from a contradiction.

Encoding information in choices. The initial proposal of encoding '0' in the first choice and '1' in any other is not a good strategy. First, it is inefficient as multiple choices provide an opportunity to encode more than just a single bit: intuitively, two choices provide one bit, four provide two bits, however an improved scheme will also exploit non power of two cases. We discuss a better approach in Section 4.1. Second, this would bias the algorithm towards always making the same choices. Consider in particular the empty message, represented by a zero bit stream. Selecting consistently the first choice will strongly bias the synthesis process and could quickly run in contradictions that could never be resolved. Instead we randomize which symbol in the stream represents which choice, using a random offset at each coding site. This is discussed in Section 4.2.

4.1 Data as a large integer

To encode data efficiently, we take an arithmetic point of view. We transform the message into a large integer. Given a message D of length L such that $D[i]$ is the i-th byte, we compute the large integer $N = \sum_{i=0}^{L-1} D[i] \cdot 2^{8 \cdot i}$.

Let us consider the sequence of number of choices in each coding site, denoted as $B_0, B_1, ..., B_i$. We encode N onto the varying radix basis formed by the B_i. We define $N_0 = N$. At a coding site j we have B_j choices. We choose the symbol as $s_j = (N_j \mod B_j)$ and update N as $N_{j+1} = N_j / B_j$, where all operations are carried with large integers.

4.2 Reintroducing randomness

If applied directly, this encoding will introduce strong bias when the message is empty (N=0), which always happens after the last bit of information is encoded. This will degrade synthesis quality, and possibly lead to contradictions.

In addition, the process would be fully determined by the message itself. This provides no randomness: if a contradiction is encountered there would be no possibility to re-run the process.

We introduce randomness by randomly offsetting the symbol encoded at each site: we encode $s_j = ((N_j + PRNG(j)) \mod B_j)$ where $PRNG$ is a pseudo-random number generator.

Changing the seed allows the algorithm to produce a different set of choices, thus affording to explore other potential solutions and attempt to resolve contradictions. The generator seed can also serve as a password, since without it the message cannot be decoded easily (see also Section 7).

5 Decoding

We assume that the decoder has access to the tileset used for synthesis. The decoding algorithm starts by reconstructing the labels from the input image. This can be done by simple image comparison if the image is unaltered, or with a more elaborate algorithm if the input has been modified (e.g. image compression, noise, etc.). Ultimately we would like to extract the data from a photograph of the texture. We describe in Section 5.1 a scheme robust to lossy compression and noise.

After obtaining the labels, the decoder runs synthesis again. However, when a choice is possible, it uses the input to determine which choice was made during the initial synthesis. This decodes one symbol of the message. At the end the bit stream is fully recovered in the form of a large integer, which is then converted back to a base of radix 256 to recover the bytes of the encoded message D.

If at some point the decoder finds a discrepancy (a set of choices that does not contain the tile selected in the input), then it can conclude that synthesis failed and was re-run. It generates the next seed for the random generator and restarts. Note that this process converges only if the correct initial seed is used, otherwise there is no guarantee this process even terminates.

5.1 Robust label extraction

The first step of decoding is, given an image, to reconstruct which tile (label) was selected in each location. Throughout this process we assume the input tileset is known to the decoder, and that all tiles are different (tiles under transforms are seen as different labels, unless they have identical content due to symmetries).

The straightforward approach for label extraction is to split the image into tiles, and then search for corresponding tiles in the tileset. This can be done by comparing the tile images, for instance measuring mean square pixel error. If

the image is unaltered, then the difference between an input tile and one of the tileset tiles will be zero. This identifies the tile selected during synthesis.

If the image has been altered, for instance after lossy compression, the tile with smallest difference can be selected. This is however fragile, since under large changes this may no longer be the tile selected during synthesis. Such an error will immediately break the decoding process, as any error cascades into a catastrophic divergence from the original sequence.

To make the process more robust, we propose to exploit the layout constraints. We indeed have a strong a priori knowledge on the input: it enforces the constraints everywhere. The key idea is to first compute matching scores between each input tile and the entire tileset. The input tiles are then sorted according to a confidence score. Intuitively a tile which matches very well a single tile of the tileset will have a high confidence, while a tile which matches equally well the entire tileset has low confidence. We then simulate a synthesis process, visiting the input by decreasing order of confidence. Each time, we select the tile that is best matching *and we propagate the constraints*. This implies that many potential candidates at neighboring locations are being removed by the constraints, resolving ambiguities. Hence, when we visit the next site, we only consider these candidates which are still available, selecting the one that is best matching.

This process significantly improves robustness. Results are reported in Figures 1 and 2 for both jpeg compression and noise (erasing a percentage of pixels with a random color). As can be seen the improvement is significant for each tileset.

Computing confidence. This is an area of improvement. Right now we use the difference between the second best and best matching across the tileset (intuitively, if this difference is large then the 'best' is a much better match than all others).

Discussion. What makes a tileset more robust than another is a topic of further study. Robustness could also be improved by selecting a better difference metric, as well as a better confidence measure.

6 Implementation

A proof of concept is available at `http://members.loria.fr/Sylvain.Lefebvre/infotexsyn/`. Code will be released when ready, but it should be simple to implement after reading this document.

7 Discussion

Tilesets This algorithm works with any tileset that provides choices (e.g. a tileset with a single tile provides none). Some tilesets will allow to encode larger

messages than others in a same space. Whether and why a tileset allows for more information space in an intriguing theoretical question that we intend to explore.

Encoding capacity can be measured experimentally, trying to encode increasingly longer random messages with a limited number of tiles. Using this technique, we determined the encoding capabilities presented in Table 1. It is also interesting to consider this data in light of decoding robustness (Section 5.1).

Circles	Circuit	Knots	Castle	Rooms	Summer	Squares
2.39	1.98	1.8	0.76	0.69	0.47	0.24

Table 1: Encoding capabilities of different tiles sets, in bits per tile, measured experimentally.

Messages Another intriguing question is whether pathological messages exist, that would consistently lead to a contradiction regardless of the initial random seed. Such message would be impossible to encode.

Image size As it is difficult to ensure a message will fit, and since the synthesis is fast, we devise the following approach. First, the algorithm determines the size in bits of the message and uses previously computed average capabilities of the tileset to estimate the size of S (assuming some aspect ratio, e.g. 1:1). If the process terminates before all bits are encoded, S is enlarged and the process resumes. If on the contrary a lot of space is left empty, S could be reduced for a tighter fit. Note that messages are always padded with zeros until synthesis completes.

Steganography While this is a form of steganography, it is unclear yet whether our approach is a good or bad steganography approach.

Cryptography It is intriguing to consider how 'secret' the encoding is. In particular, if the PRNG seed is lost, how difficult would it be for an adversary to extract the message?

Embedding the tileset We currently assumed that the decoder has access to the tileset used for synthesis. It would be interesting to consider whether the tile set could be embedded within the texture. This should ensure that both the tiles and constraints are embedded, and that decoding remains robust.

References

[1] Paul Merrell. Example-based model synthesis. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 105–112, 2007.

[2] Hirofumi Otori and Shigeru Kuriyama. Data-embeddable texture synthesis. In *International Symposium on Smart Graphics*, pages 146–157, 2007.

[3] Kuo-Chen Wu and Chung-Ming Wang. Steganography using reversible texture synthesis. *IEEE Transactions on Image Processing*, 24(1):130–139, 2015.

[4] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 341–346, 2001.

[5] Zhenxing Qian, Hang Zhou, Weiming Zhang, and Xinpeng Zhang. Robust steganography using texture synthesis. In *Advances in Intelligent Information Hiding and Multimedia Signal Processing*, pages 25–33. 2017.

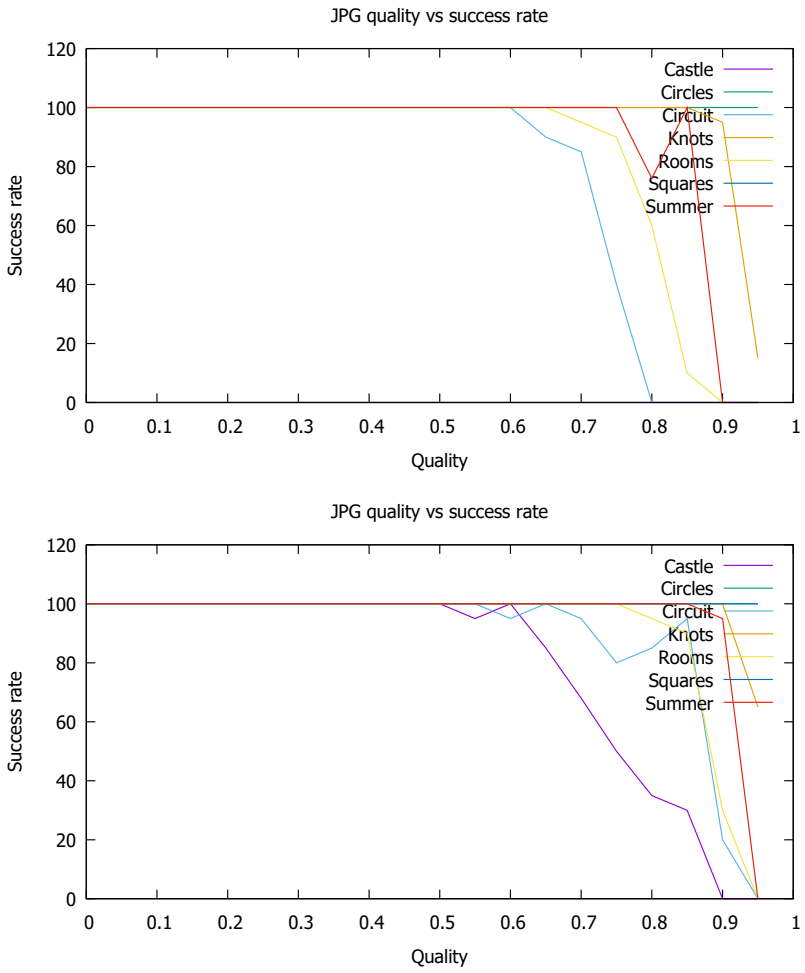

Figure 1: Robustness to jpg compression for various tilesets, plotting decoding success rate versus jpeg quality (0: lossless, 1: maximum compression, using libjpeg setting). **Top:** simple selection of the best matching tile. **Bottom:** robust scheme exploiting constraints.

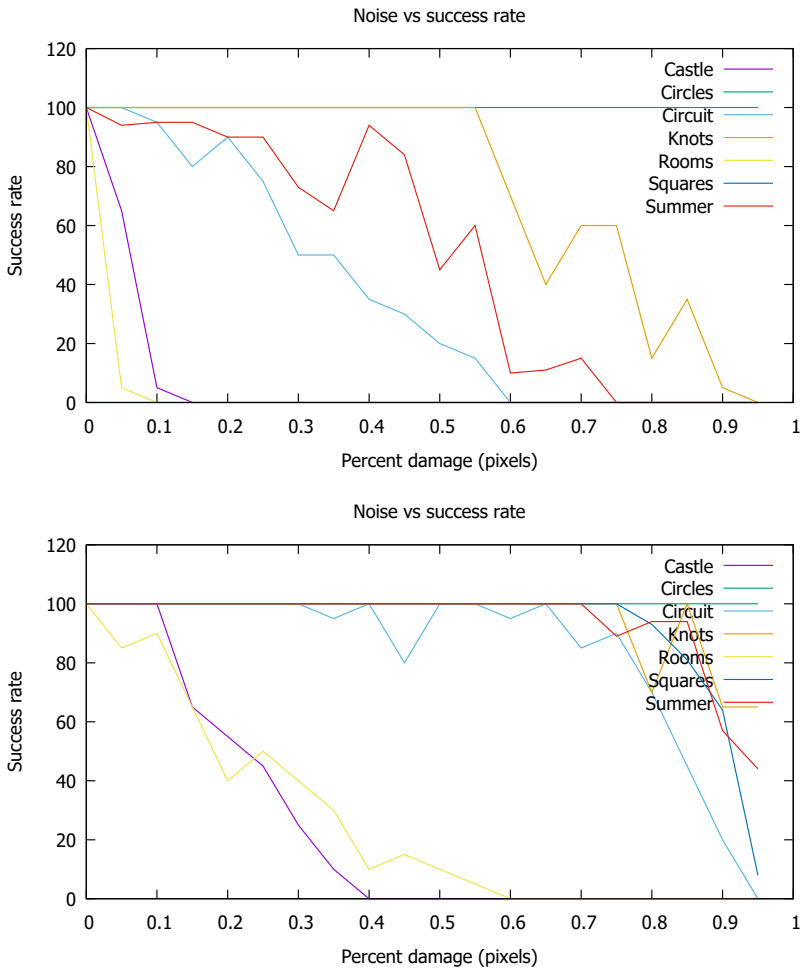

Figure 2: Robustness to noise, plotting decoding success rate versus amount of noise. A percentage of pixels is selected and erased by a random color, with 1 meaning 50% of pixels damaged. **Top:** simple selection of the best matching tile. **Bottom:** robust scheme exploiting constraints.