Chaos Mosaic: Fast and Memory Efficient Texture Synthesis

Ying-Qing XU; Baining GUO; Harry SHUM

April 20, 2000

Technical Report
MSR-TR-2000-32

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Chaos Mosaic: Fast and Memory Efficient Texture Synthesis

Category: Research

Abstract

We present a procedural method for synthesizing large textures from an input texture sample. The basis of our algorithm is the chaos mosaic, a technique for synthesizing textures with an even and visually stochastic distribution of the local features of the input sample. The chaos mosaic is fast. For synthesizing textures of the same size and comparable quality, our algorithm is orders of magnitude faster than existing algorithms. On a PC we can synthesize a 512×512 texture from a 64×64 sample in just 0.03 second. More importantly, the chaos mosaic facilitates memory efficient texture rendering through procedural texturing. Like traditional solid texture techniques, the chaos mosaic allows us to synthesize and render synthetic textures that, if stored explicitly as textures, would require prohibitively large amount of storage. As an example, we demonstrate that an $100k \times 100k$ synthetic texture can be interactively visualized on a modest PC without suffering from latency. Finally, the chaos mosaic can drastically reduce the bandwidth for interactive 3D graphics delivered across the internet.

Keywords: Procedural Texturing, Chaos Transformation, Virtual Texture, Latency, Bandwidth.

1 Introduction

Texture mapping was introduced in [Cat74] as a method of adding surface details by projecting a texture image onto an object surface. Despite its tremendous success, texture mapping suffers from three serious problems. First, the available texture is often too small to cover the object surface. In this situation, a simple tiling may introduce unacceptable artifacts in the form of visible repetition. Second, there may be no natural map from the texture space to the object surface and consequently the texture is subject to severe distortion when mapped. The third and probably the most important problem is that texture mapping is memory intensive, both in terms of capacity and bandwidth [BAC96, TK96].

Researchers have proposed texture synthesis algorithms to address some of the above problems. Two of the main approaches to texture synthesis are procedural methods [Pea85, Per85, Lew89, Ups89, WK91, Tur91] and statistical sampling methods [HB95, De 97, PS99, ZLW99, EL99]. These two approaches are complementary in their strengths and weaknesses. Procedural methods [Pea85, Per85, Lew89] can be very fast and they support memory efficient texture rendering by not storing the synthesized textures explicitly but synthesizing them on the fly [Ups89]. On the down side, existing procedural methods are only specialized emulators of the generative processes of certain types of textures, such as marbles, sea shells, and animal skins. Statistical sampling methods can synthesize a wide variety of textures, as long as appropriate sample textures are provided. Since they start from sample textures, the statistical sampling methods also eliminate the need of parameter tweaking, which is essential to traditional procedural methods. However, statistical sampling methods are quite slow and they do not consider the memory efficiency during texture rendering. In fact, some statistical sampling methods tend to use a lot of memory just for synthesizing textures (e.g., see Section 4.1).

To combine the strengths of procedural and statistical sampling methods while avoiding many of their weaknesses, we have developed a procedural method for synthesizing large textures from an input sample texture. The basis of our algorithm is the chaos mosaic, a technique for synthesizing textures with an even and visually stochastic distribution of the local features of the input texture sample. Texture synthsis with the chaos mosaic is easy and fast. For synthesizing textures of the same size and comparable quality, our algorithm is orders of magnitude faster than existing statistical sampling algorithms. For example, we can synthesize a 512×512 texture from a 64×64 sample in 0.03 second on a PC with a 450 Mhz Pentium III processor .

The chaos mosaic also addresses the memory efficiency issue. Like the procedural solid texturing techniques, the chaos mosaic allows us to synthesize and render synthetic textures that, if stored explicitly as textures, would require prohibitively large amount of storage [Pea85, Per85, Lew89, Ups89]. We introduce a compact encoding of the chaos mosaic called the virtual texture, by which textures are generated on the fly during rendring. As an example, we demonstrate that an $100k \times 100k$ synthetic texture can be interactively visualized on a modest PC without suffering system latency.

Because of its speed and memory efficiency, the chaos mosaic can significantly reduce the bandwidth

for interactive 3D graphics delivered across the internet. We can replace the synthetic textures in a VRML model by texture samples and have the VRML browser generate the synthetic textures from the downloaded texture samples upon reading the VRML model for display. The result is a drastic reduction of both downloading bandwidth and the loading time of the VRML model. Note that we can also generate the synthetic textures using statistical sampling methods [HB95, De 97, PS99, ZLW99, EL99], but we will loose interactivity because these methods are slow.

The chaos mosaic resembles the statistical sampling techniques [HB95, De 97, PS99, ZLW99, EL99] in that it generates synthetic textures from an input sample texture. In fact, by maintaining local characteristics of the sample texture while varying from it in the global form, the chaos mosaic often produces similar results as De Bonet's successful algorithm does [De 97]. However, it is worthwhile to note that the chaos mosaic is a procedural method that does no statistical modeling or analysis. In designing the chaos mosaic, we are not trying to build a statistical theory for texture modeling and analysis (e.g., see [PS99, ZLW99]). Instead, our goal is to derive a fast recipe for high-quality synthetic textures suitable for various graphics rendering taskes.

The rest of the paper is organized as follows. In section 2, we discuss the chaos mosaic in detail. In particular, we explain how to transform a tiling of the input sample texture into a synthetic texture that preserves the local features of the texture sample and has an even and visually stochastic distribution of these features. Section 3 describes procedural texturing with the chaos mosaic. We introduce the virtual texture encoding of the chaos mosaic and show how to synthesize and render large textures with the virtual texture. Results are presented in Section 4, followed in Section 5 by conclusions and suggestions for future work .

2 Chaos Mosaic

The chaos mosaic is rather unique in that it starts with a tiling of the input sample texture and transforms the tiling into a synthetic texture that has an even and visually stochastic distribution of the local features of the texture sample. In this section, we describe our techniques for creating an even and visually stochastic distribution and for preserving local features.

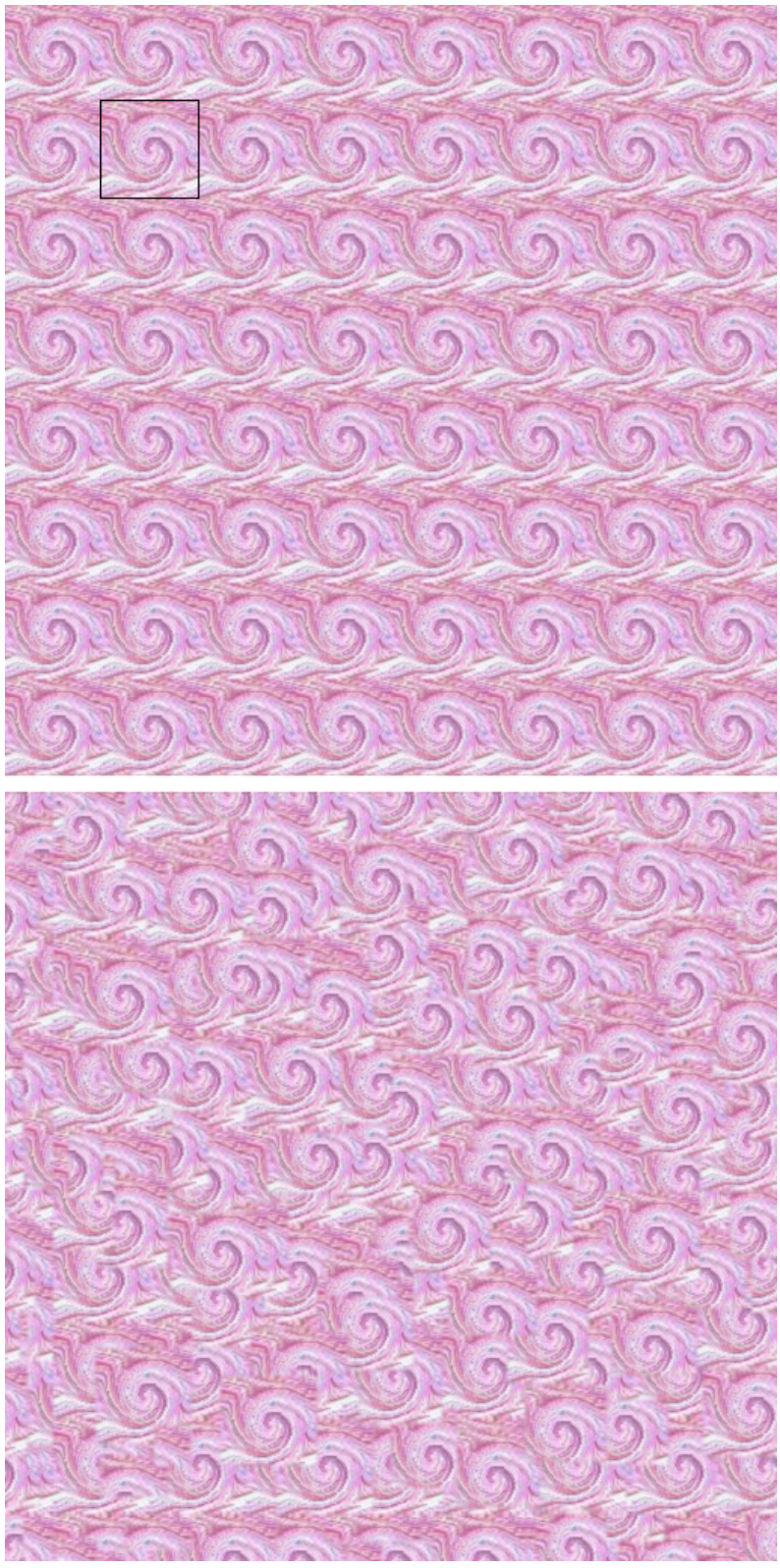

Figure 1: From tiling to the chaos mosaic. Top: a tiling of the input sample texture, with the black box indicating the input sample. Bottom: the corresponding chaos mosaic. We obtain the chaos mosaic from the tiling by applying a procedurally-defined image transformation.

Notation: We describe the size and location of a texture block B by an integer tuple (x, y, w, h), where x and y specify the top left corner of the block. The width and height of the block are given by w and h respectively. A random block B is a block whose size and location parameters x, y, w, and h are random numbers.

2.1 From Tiling to Chaos Mosaic

Fig. 1 shows an example of creating an $m \times m$ chaos mosaic M from an $n \times n$ input sample texture S. As shown in Fig. 1 (top), it is easy to construct an $m \times m$ tiling T from the input sample S. While inheriting the local features of S, the tiling T has an undesirable repetitive structure gloablly. We wish to construct the chaos mosaic M such that it satisfies the following requirements:

a) M maintains the local features of S.

b) Globally M has an even and visually stochastic distribution of the local features of S and this global distribution is quickly computable.

The first requirement of the chaos mosaic is motivated by the fact that capturing the local features is essential for the synthsized texture to have identical textural characteristics as the texture sample. This fact has been recognized for a long time by researchers who analyze/synthesize textures using statistical sampling methods [HB95, De 97, PS99, ZLW99, EL99]. Considering the input sample textures as samples of some probabilistic distributions, statistical sampling methods create new synthetic textures by estimating these probabilistic distributions and sampling from the estimations. Although the estimation and sampling strategies vary from method to method, all methods base their analysis on locally defined image features and strive to preserve local features in the synthesized textures.

The second requirement of the chaos mosaic, with its strong emphasis on the fast computation of the global distribution of the local features, is our way to eliminate the globally repetitive structure of the tiling T. Creating a stochastic distribution of the local features so as to avoid visible repetition is not new. If we generate an synthetic texture E using a statistical sampling method (e.g. [HB95]), then E will also have a stochastic distribution of local features of the input sample texture S. Such a distribution comes naturally as part of the underlying probabilistic model of the statistical sampling method.

What is new in the second requirement of the chaos mosaic is that we explicitly demand the fast computation of the global distribution of the local features. To make this fast computation possible, we relax our requirement on the global distribution to visually stochastic: there is no need for the global distribution to have a underlying probalistic model. This global distribution which is both visually stochastic and quickly computable is indeed the key to fast and memory efficient texture synthesis using the chaos mosaic. As we shall see, a properly chosen chaos iteration system can quickly create an even and visually stochastic pattern out of any input image. The chaos mosaic derives its computational efficiency from this amazing fact.

2.2 Selecting Chaos Transformation

To create an even and visually stochastic pattern, we use an iterative system from the field of deterministic chaos [Sch88]. Deterministic chaos denotes the state of disorder and irregularities generated by a nonlinear dynamic system in which previous history uniquely determines the future behavior. One such system is the iterative system based on the cat map of Arnold [AA68]. For the tiling T, the cat map is a mapping from T onto itself such that the pixel at the point $(x^l,\ y^l)$ is mapped to $(x^{l+1},\ y^{l+1})$ as follows:

$$x^{l+1} = (x^l + y^l) \bmod m \tag{1}$$

$$y^{l+1} = (x^l + 2y^l) \bmod m \tag{2}$$

Here we used the fact that T is of the size $m \times m$. Starting from any point $(x^0,\ y^0) \in T$, we can generate a sequence of points $\{(x^l,\ y^l) \in T \mid l = 0,\ 1,\ 2,\ 3,\ ...\}$ by iteratively applying the cat map. We call the resulting iteration system the cat map iteration.

Although the cat map iteration appears to be very simple, its analysis are quite mathematically involved. In this section we will keep our discussions as simple and intuitive as possible. In the appendix, we provide more mathematical explanations.

While there are many chaos transformations in the field of deterministic chaos [Sch88], we selected the cat map for two reasons. First, the cat map is an area-preserving automorphism of the torus (i.e., the cat map is an area-preserving mapping defined on the torus and the mapping is both one-to-one and onto). Because the cat map is defined on the torus, we can easily make the chaos mosaic defined on the torus

Figure 2: The cat map iteration can quickly create an even and visually stochastic pattern out of any input image. Top left: a tiled texture with 4×4 tiles. Top right: the result after one interation of the cat map. Bottom left: after three interations. Bottom right: an even and visually stochastic pattern achieved after five iterations.

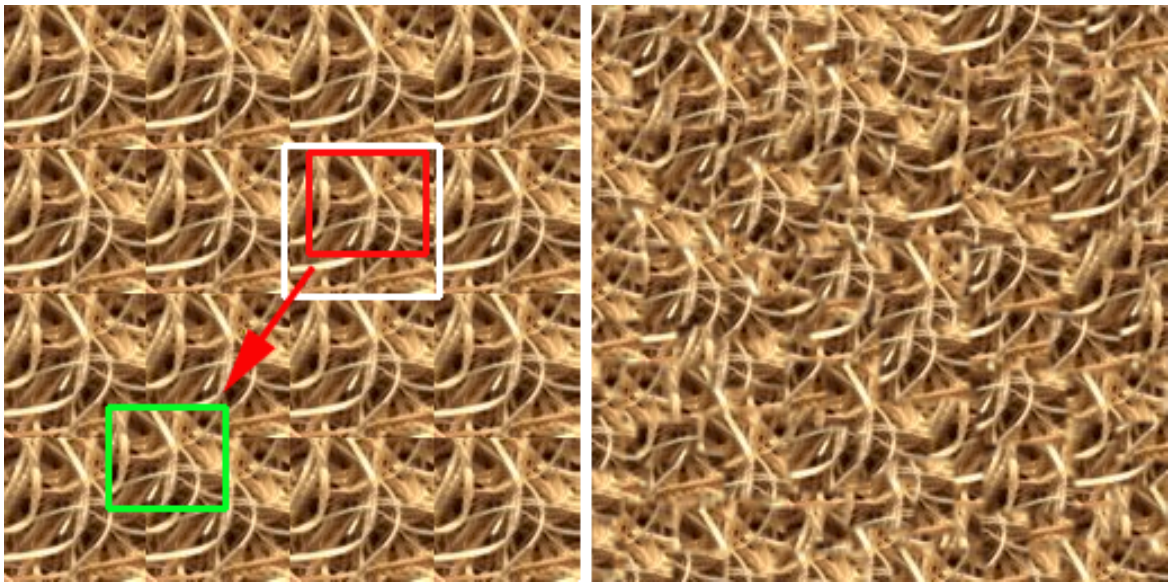

Figure 3: The chaos transformation translates the random blocks of every tile to new locations, where they are pasted over the tiling. Left: the random block (highlighted by the red box) of a tile (highlighted the white box) is translated to a new location as the green box and pasted over the tiling. Right: the synthesized texture.

as well and thus make the resulting synthetic texture tilable. Second, the cat map iteration is an ergodic system[1] well known for its strongly irregular motion. A trully amazing result of this strongly irregular motion is that the cat map iteration can create an even and visually stochastic pattern out of any input image in just a few iterations. Using a tiled texture as an example, Fig. 2 shows an even and visually stochastic pattern achieved after five iterations of the cat map. The strong irregular motion of the cat map iteration makes a visually stochastic distribution quickly computable as required by the chaos mosaic.

2.3 Preserving Local Features

Direct applications of chaos transformation to the tiling T do not preserve the local features within T. As is shown in Fig. 2, the chaos transformation destroys the local features along with the global structure of T. To preserve the local features, we apply the chaos transformation at the block level instead of the pixel

[1]A system is ergodic if its trajectory uniformly covers the energetically allowed region of classical phase space such that time averages can be replaced by the average over the corresponding phase space.

level. Specifically, we randomly choose a set of blocks within each tile of T. The chaos transformation translates the random blocks of every tile to new locations, where they are painted over the current pixels of T. See Fig. 3 for an example.

For the random blocks to capture local features of the input sample texture, we need to control the size of the blocks. On the one hand, a random block must be big enough to capture local features. On the other hand, to increase irregularity in the chaos mosaic we also require that a random block be smaller than an individual tile of T. Since the input sample texture is of size $n \times n$, we introduce the block area constraint on a random block B_γ as follows,

> **Block Area Constraint:** The block B_γ = (x_γ, y_γ, w_γ, h_γ) must be contained in the input sample texture S and that
>
> $$0.0 < c_\lambda \leq \frac{w_\gamma h_\gamma}{n^2} \leq c_\mu < 1.0.$$
>
> The constants c_μ and c_λ are the upper and lower bounds of the block area respectively.

Our experiments indicate that the precise values of c_μ and c_λ have little effect on the synthesized textures. For the results reported in this paper, we set $c_\lambda = 0.5$ and $c_\mu = 0.75$.

Next we need to decide how many random blocks per tile of T are needed. Ideally, each tile should be partially covered by one random block so that the chaos mosaic contains no verbatim copy of the input sample texture. Since each random block can touch at most 4 tiles, we should have more than $1/4$ random blocks per tile. Having more than one blocks per tile, however, tends to crowd some tiles with too many random blocks. For this reason, a good balance point is somewhere around one random block per tile. Our experiments suggest that the precise number of random blocks per tile is immaterial. For the results reported in this paper we simply use one random block per tile of T.

We define the cat map iteration for the random blocks as follows. For a random block B_γ = (x_γ, y_γ, w_γ, h_γ), we feed its top left corner (x_γ, y_γ) to the cat map iteration as the starting point (x_γ^0, y_γ^0). After i steps, we place the top left corner of the block B_γ at (x_γ^i, y_γ^i). Before the cat map iteration starts, there is a random block within each tile of the tiling T and the random blocks are hence evenly distributed over T. The cat map iteration redistributes the random blocks over T and produces two results. First, the redistribution quickly creates a visually stochastic pattern due to the strong irregular motion of the cat map iteration.

For the results reported in this paper, we set the number of cat map iterations to be $i = 10$. Second, the redistribution keeps the random blocks evenly distributed because the cat map iteration is ergodic and for each random block B_γ, its top left corner (x_γ^l, y_γ^l) is equally likely to be in any region of T as l increases.

Since the cat map is defined on the torus, so is the chaos mosaic M. Fig. 4 (bottom) shows a chaos mosaic mapped onto a torus. As a texture defined on the torus, the chaos mosaic tiles.

2.4 Resolving Mismatched Features

After distributing the random blocks over the tiled texture T, we have generated an even and visually stochastic distribution of the local features of the input sample texture. The remaining task is to treat the boundaries of the random blocks (and the boundaries of the tiles of T if the input sample texture does not tile seamlessly). When the input sample is a noisy texture with non-distinguishable features, the transition across the block boundary is not noticeable. On the other hand, if the input sample texture has distinguishable features, there may be mismatch between the features across the boundary as shown in Fig. 5.

One way to resolve the mismatched features across a boundary edge e is to apply a constrained synthesis technique. We first black out a block B_ϵ of pixels around the edge and then perform constrained synthesis to fill the hole B_ϵ. Typically we black out three layers of pixels around the edge e to form B_ϵ. Efros and Leung have developed a very effective texture synthesis algorithm that works well on a wide variety of textures and is especially well-suited for constrained synthesis [EL99]. We have adapted their algorithm for synthesizing the pixels in the block B_ϵ. Efros and Leung model a texture as a Markov Random Field (MRF) and grow a texture pixel by pixel from the known pixels. To synthesize a pixel p, they first find all neighborhoods in the input sample texture that are sufficiently similar to the pixel p's neighborhood ω_ρ and then randomly select one neighborhood and take its center to be the value of p. Because B_ϵ is a narrow strip along the edge e, ω_ρ contains known pixels from both sides of e and this fact should influence the choice for the pixel value of p. In practice, however, we found out that this influence by itself is not strong enough to extend features from one side to the other side. By closely examining the synthesis process, we observed that for most pixels of B_ϵ, only one neighborhood satisfies Efros and Leung's similarity criterion. As a result, the newly-synthesized pixel p is not given enough freedom to be able to differ from the blacked

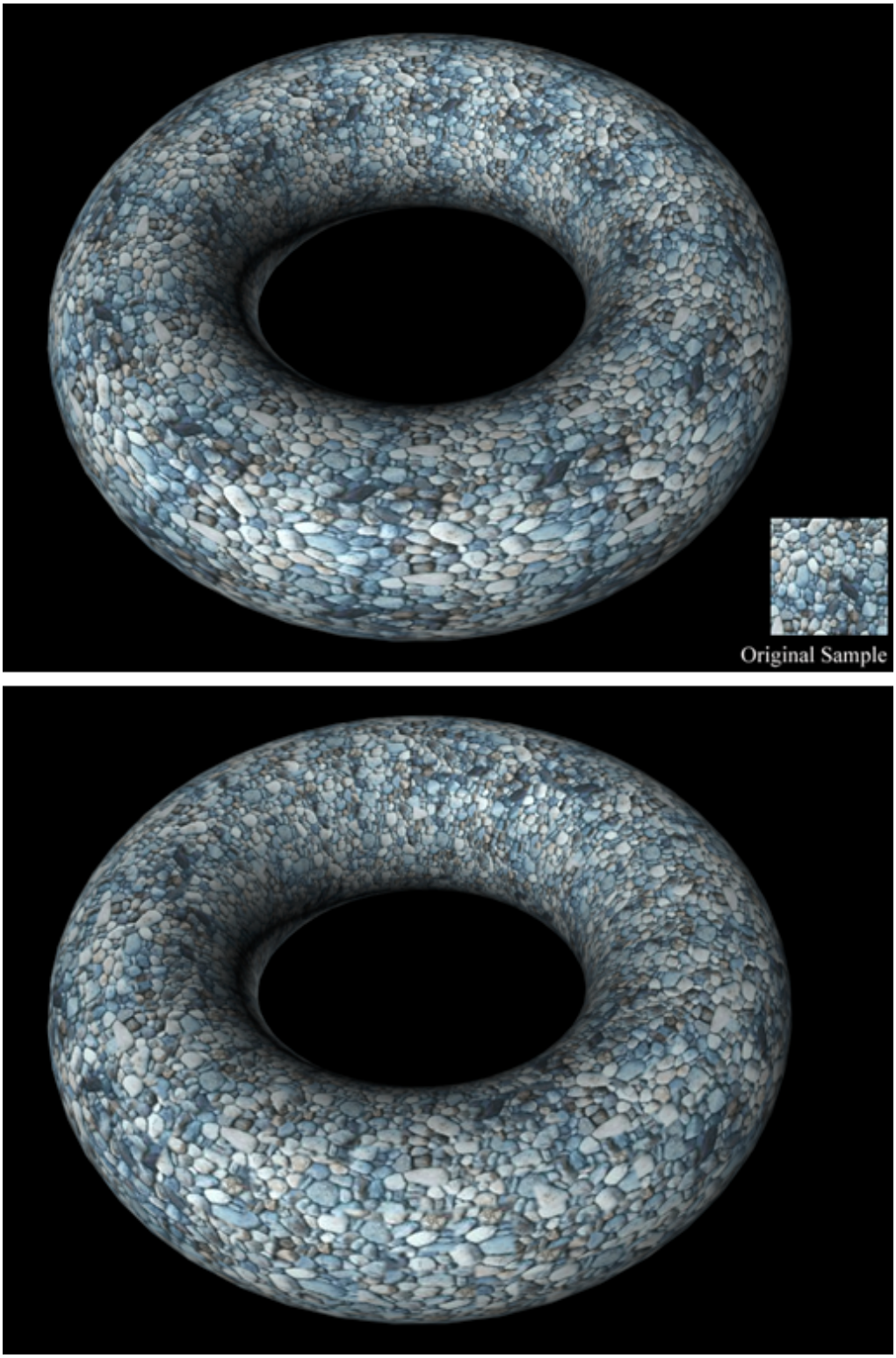

Figure 4: The chaos mosaic is defined on the torus and hence tiles. Top: the torus covered by the tiling T. We included this image for quality comparison with the chaos mosaic. Bottom: the torus covered by the chaos mosaic M.

out pixel value of p. To encourage randomness in selecting the value of p, we consider a neighborhood ω' to be sufficiently similar to ω_ρ if ω' is among the j neighborhoods that are most similar to ω_ρ where $j = 5$. With this new criterion, the constrained synthesis can resolve the mismatched features, as the example in Fig. 5 demonstrates.

Despite its effectiveness, the constrained synthesis procedure described above is quite expensive computationally. The main factors determining the cost are the sizes of the neighborhood ω_ρ and the input sample texture. For an 128×128 input sample texture and an 11×11 neighborhood ω_ρ, it can take several minutes to fill a hole of 400 pixels. In searching for a fast approximation to the above constrained synthesis procedure, we have done extensive experiments. Through these experiments we found that for most textures, a simple cross-edge filtering is sufficient to smooth out the transition across the block boundaries. All results reported in this paper are obtained using a cross-edge filtering over three layers of pixels on each side of the block boundary. The reasons for our findings are two folds. First, we rarely see severely mismatched features like those in Fig. 5, which we create by hand to highlight the issue. Second, the synthetic textures we create are for texture mapping on 3D surfaces. Due to the trilinear filtering performed during texture mapping, the image blur caused by the cross-edge filtering is hard to spot on 3D surfaces, especially in our case the affected areas are very small and they are irregularly distributed. In Section 4.2 we further assess the effect of the cross-edge smoothing on image quality.

2.5 Summary

In summary, we construct the chaos mosaic M from the tiling T through the following steps. First, we select a set of random blocks within each tile of the tiling T. Then, we distribute all the random blocks over T using the cat map iteration and paste the blocks over T. Finally, we filter the pixels near the edge of the random blocks in order to smooth out the transition between two blocks or between a block and the background tiling T. Fig. 1 (bottom), Fig. 3 (right), and Fig. 4 (bottom) are some synthetic textures constructed through these steps.

In terms of computational cost, it is easy to see that the most expensive step in constructing M is painting the random blocks over the tiling T. In our unoptimized implementation we paint one pixel at a time, which makes the overall construction cost O(n_ν) where n_ν is the number of pixels in the tiling T.

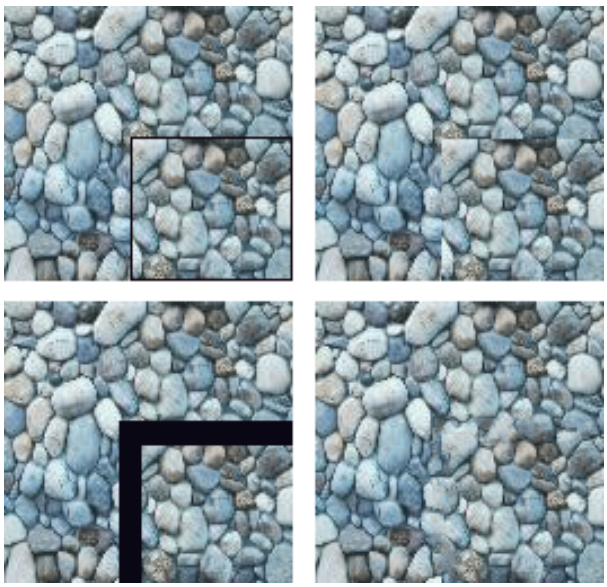

Figure 5: Resolving mismatched features on the boundary edge of a block. Top left: mismatches features on the boundary edge of the block, with the boundary edge marked by black lines. Top right: the same as the left figure but the black lines are removed for better visualization of the mismatched features. Bottom left: a few layers of pixels near the edge are blacked out to form a constrained texture synthesis problem. Bottom right: the result of constrained synthesis.

3 Virtual Texture

In this section Section 3 describes procedural texturing with the chaos mosaic. We introduce the virtual texture, an encoding of the chaos mosaic. The virtual texture is thousands of times more compact than explicitly stored textures. Yet, we can evaluate any part of a virtual texture on the fly. These properties of the virtual texture make it a powerful tool for memory efficient texture rendering.

3.1 Global Structure Encoding

The virtual texture V consists of the input sample texture S and a precomputed global structure of M. More specifically, V is a tuple (S, T_ϵ, R_ϵ) with T_ϵ and R_ϵ encoding the global structure of the chaos mosaic as follows:

- The tiling encoding T_ϵ = (m_τ, n_τ, w_τ, h_τ) records the structure of the tiling T. The integers m_τ and n_τ are the numbers of tiles in x- and y-directions, whereas w_τ and h_τ are the width and height of a tile.

- The random block sequence $R_\epsilon = \{B_0,\ L_0,\ B_1,\ L_1,\ ...,\ B_{k-1},\ L_{k-1}\}$ describes k random blocks within S that are to be pasted over the tiling T and the locations to paste them. Each random block B_γ = (x_γ, y_γ, w_γ, h_γ) is pasted in T such that the top left corner of B_γ is at L_γ = (s_γ, t_γ).

Note that the random block sequence R_ϵ also determines a fixed order to paste the random blocks. This order matters because pixels pasted later overwrite existing pixels at the same locations. Fig. 6 (top) illustrates the virtual texture encoding of the chaos mosaic.

Precomputing the virtual texture is easy. Since S and T_ϵ are given, we only have to calculate R_ϵ. By selecting a random block B_γ for each tile of the tiling T, we have implicitly set values for the parameters x_γ, y_γ, w_γ, and h_γ. Suppose that B_γ is selected within the $(k_1,\ k_2)$-th tile of T. To get the parameters s_γ and t_γ we only have to feed $x^0_\gamma = x_\gamma + k_1 w_\tau$ and $y^0_\gamma = y_\gamma + k_2 h_\tau$ to the cat map iteration and collect the result $s_\gamma = x^i_\gamma$ and $t_\gamma = y^i_\gamma$ after $i = 10$ iterations.

We can roughly estimate the storage for the virtual texture as follows. For an $m \times m$ virtual texture V with sufficiently large m, the dominate storage cost of V is that of the random block sequence R_ϵ.

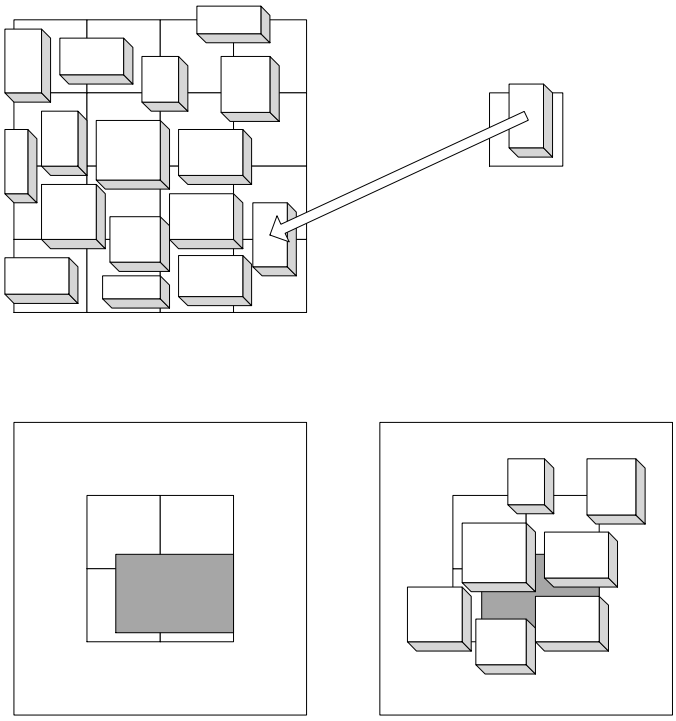

Figure 6: Virtual texture encoding and evaluation. Top row: the encoding of the global structure of the chaos mosaic. Shown on the left hand side is the structure of the tiling and the random blocks, with the random blocks indicated by 3D boxes. For each random block, we record its position within the tiling as well as its position within the input sample texture (shown by the square on the right hand side). Bottom row: the lazy evalutation of a texture block B_Π, which is shown by the shaded rectangle. Shown on the left hand side is the evaluation of the tiles corvering the B_Π. Shown on the right hand side are evaluated random blocks touching those tiles.

Typically, the input sample texture S is of size $n \times n$ with $n = 64$, $n = 128$, or $n = 256$. For such an S, a random block B_γ = (x_γ, y_γ, w_γ, h_γ) and its location L_γ = (s_γ, t_γ) require 12 bytes. With one random block per tile of T, we need roughly 12 m^2/n^2 bytes for an $m \times m$ virtual texture. This is about $n^2/4$ smaller than an $m \times m$ "real" texture with 24-bit pixels. For example, an $m \times m$ virtual texture from an 128×128 sample is 4,000 times smaller than an $m \times m$ "real" texture.

3.2 Procedural Texturing

The evaluation strategy of the virtual texture is similar to that of the procedural solid texture [Per85, Pea85, Lew89, Ups89], which procedurally defines a texture over the entire 3D space but evaluates only points on the object surfaces where the texture is needed. From the virtual texture V, we can quickly evaluate any part B_Π of the chaos mosaic on the fly. The evaluation procedure is just a lazy construction the chaos mosaic. Instead of pasting all random blocks on the complete tiling T, we only tile with the tiles touching B_Π and paste random blocks that touching those tiles. Fig. 6 (bottom) illustrates the lazy evaluation of the virtual texture.

While the above evaluation procedure works correctly for B_Π of any size, it is not the most efficient way to do single-pixel evaluation, which we need for procedural texturing [Ups89]. Consider the case when the desired pixel p is not near the boundary of a random block. Evaluating p involves determining the tile B_ρ where p belongs and the random blocks covering B_ρ. Once these random blocks are found we can determine the last random block covering p or that no random block covers p. Then we can either fetch p from the last random block or from the tile B_ρ. Finding B_ρ is a simple matter of indexing arithmetic. To efficiently find the random blocks touching B_ρ, we should precompute a matrix whose $(k1,\ k2)$-th entry contains pointers to the random blocks touching the $(k1,\ k2)$-th tile of the tiling T. As for the case when the pixel p is near the edge of a random block, we need to evaluate more than one pixels to produce a filtered result as the desired pixel.

In order for this on-the-fly texture evaluation to work correctly, the evaluation must be fast and, more importantly, it must satisfy the internal consistency requirement [FFC82, Lew89]. The internal consistency requires that if a given part of the synthetic texture is synthesized multiple times, the result must be always the same. This requirement is important when we use the synthetic texture in an animation. For the virtual

texture, the precomputed global structure ensures the internal consistency. In this respect, the virtual texture behaves just like a "real" texture.

It is worthwhile to note that the existing statistical sampling methods [HB95, De 97, PS99, ZLW99, EL99] are not suitable for procedural texturing. First, these methods are quite slow and they require a long processing time even for a small part of the texture. Second, if we apply a statistical sampling method to synthesize a small piece of the texture at a time, there is no easy way to satisfy the internal consistency requirement.

3.3 Applications

Memory Efficient Texture Rendering: The virtual texture is an effective way to reduce the system latency caused by a synthetic texture that does not fit into the system memory, either because the scene geometry has taken too much space already or because the synthetic texture is itself too big. System latency is one of the most fudamental issues in graphics system design [TK96] and researchers have developed various techniques to reduce or hide system latency. Interactive rendering systems handle system latency by careful algorithms design and pipelining [TK96]. For scanline-based rendering [CCC87] and ray tracing [PKGH97], caching is an excellent way to reduce system latency. Caching works well as long as the system memory is bigger than the working set of the scene; when the system memory is smaller than the working set, caching is not effective and the rendering task may not get a chance to finish at all [PKGH97].

The virtual texture can reduce system latency for two reasons: the virtual texture is itself very small and it supports procedural texturing, by which textures are generated on the fly as needed. Procedural texturing based on the virtual texture can be used the same way as procedural solid texturing [Pea85, Per85, Lew89], which is already a standard features of most of today's ray tracing systems as well as any rendering system using the RenderMan interface [Ups89, HL90].

As an interactive demonstration of capability of the virtual texture, we have developed an interactive system for displaying large synthetic textures based on on-the-fly evaluation of the virtual texture. This system, which allows the user to see any part of the synthetic texture through a window, has two outstanding features. First, the system can load a very large synthetic texture quickly because an $m \times m$ virtual

texture is thousands times smaller than an $m \times m$ "real" texture. Second, the user can view any part of a very large (e.g. $100k \times 100k$) synthetic texture interactively without suffering system latency. See Section 4 for detailed report on experimental results.

The basic idea of our virtual texture display system can be extended to interactive texture mapping of large synthetic textures. For simplicity we consider a scene in which geometry primitives use different parts of the same texture. Before rendering each frame, we determine a rectangular region B_σ in the texture space for every visible primitive P_σ in the scene. We then evaluate a block B_Σ of the virtual texture V such that B_Σ covers B_σ for every P_σ. Finally, we perform texture mapping with the evaluated texture block B_Σ. Our experimental results are discussed in Section 4.

In principle, the virtual texture can also be used to reduce the memory bandwidth due to large synthetic textures. Today, bulk of the IHVs have moved to the so-called "pull" architecture [CBS98]. In this architecture, textures are stored in the system memory and the graphics accelerator pulls texels from the system memory to the accelerator's on-chip texture cache during texture mapping. Since memory bandwidth is improving at a much slower rate than that of memory capacity [TK96], today's memory system cannot keep pace with the escalating demand for texture bandwidth. Because the virtual texture is so small, it can fit into the L2 cache of a memory system that uses multi-level texture caching [CBS98] and thus eliminate the need to fetch texels from system memory. For example, a 5000×5000 virtual texture with an 128×128 24-bit input sample is about 67 Kb, which fits into a typical L2 texture cache ($2 \sim 8$ Mb) [CBS98].

Bandwidth Reduction: The Virtual Reality Modeling Language (VRML) is a standard language for describing interactive 3D objects and worlds delivered across the Internet. The chaos mosaic allows us to replace the synthetic textures in a VRML model by their texture samples. After the VRML model is downloaded, the VRML browser generates the synthetic textures from the downloaded texture samples upon reading the VRML model for display. Handling synthetic textures in the VRML models this way drastically reduces the downloading bandwidth as well as the loading time of the VRML models. Note that in theory we can also generate synthetic textures in the VRML models using statistical sampling methods [HB95, De 97, PS99, ZLW99, EL99] but there are two problems. First, the user will not be able to display the VRML models immediately after downloading because a long time is needed to generate

Texture Size	T(H)	T(CM)	Mem(H)	Mem(CM)
400×400	157	0.026	8Mb	4.8Mb
512×512	278	0.042	12Mb	5.0Mb
800×800	579	0.101	27Mb	6.2Mb
1024×1024	1112	0.151	44Mb	7.4Mb

Table 1: Timing and memory usage comparison between the chaos mosaic and Heeger's algorithm. The "T(H)" column lists timings of Heeger's algorithm in seconds. The "T(CM)" column lists timings of the chaos mosaic in seconds. The "Mem(H)" and "Mem(CM)" report the memory usages of Heeger's algorithm and the chaos mosaic respectively.

the synthetic textures from the texture samples. Second, the statistical sampling methods explicitly store the synthesized textures and hence can only handle relatively small textures.

4 Results

4.1 Synthesis Speed

To compare our synthesis speed with existing statistical sampling methods, we have implemented Heeger's algorithm [HB95], which is one of the more efficient statistical sampling methods. Table 1 provides the statistics for synthesizing textures of various sizes from an 128×128 input texture sample. These statistics were gathered on a PC with a 450 Mhz Pentium III processor and 128 Mb of main memory. Roughly speaking, the chaos mosaic is about 6,500 times faster than Heeger's algorithm. Notice that as the the size of the synthetic image increases, the memory usage grows much faster for Heeger's algorithm than for the chaos mosaic.

In terms of quality, both Heeger's algorithm and the chaos mosaic work well on noisy textures with non-distinguishable features. When the input sample texture has distinguishable features, Heeger's algorithm ceases to be effective while the chaos mosaic continues to produce good results. We shall compare the chaos mosaic with De Bonet's algorithm [De 97], which is more successful in capturing distinguishable

features.

4.2 Texture Quality

In order to study image quality, we have done extensive testing with the chaos mosaic. The companion CDROM contains over 600 examples from our testing results. These 512×512 textures are generated from either 64×64 or 128×128 input sample textures taken from various sources including De Bonet's and Brodatz's collections.

First, we compare the image quality of the chaos mosaic with De Bonet's algorithm. We choose to compare quality with his algorithm for two reasons. First, De Bonet's algorithm is at or near the state of the art in terms of quality. The more important reason is that De Bonet has done extensive testing with his algorithm and has published his test results over the internet [De].

In general De Bonet's algorithm and the chaos mosaic produce textures of similar qualities. The two techniques have some differences, as Fig. 7 demonstrates. Fig. 7 (top right) and Fig. 7 (middle left) are diferent versions of a 192×192 texture generated by De Bonet's algorithm with randomness thresholds of 750, and 1250 respectively. Fig. 7 (bottom) is a 400×400 texture by the chaos mosaic. The chaos mosaic captures local features well and have a high degree of randomness. De Bonet's algorithm also captures local features well for small randomness thresholds, but the resulting Fig. 7 (top right) is similar to the tiling shown in Fig. 7 (top left). To get a higher degree of randomness, we can try to increase the randomness threshold in De Bonet's algorithm. However, after the randomness threshold passes 750, further increasing unfortunately has the side effect of destroying local features, as shown in Fig. 7 (middle left).

A main strength of the chaos mosaic is its ability to create a visually stochastic distribution of local features. For repeated textures such as the one in Fig. 8, the chaos mosaic generates the best result with the number of cat map iteration set to zero, in which case the texture synthesized is just a tiling. Fig. 8 (top) shows the textures synthesized by the chaos mosaic with increasing degree of randomness. For comparison, Fig. 8 (bottom) shows textures synthesized by De Bonet's algorithm with increasing degree of randomness.

Next, we study the effect of cross-edge smoothing on the quality of the synthesized texture. In most

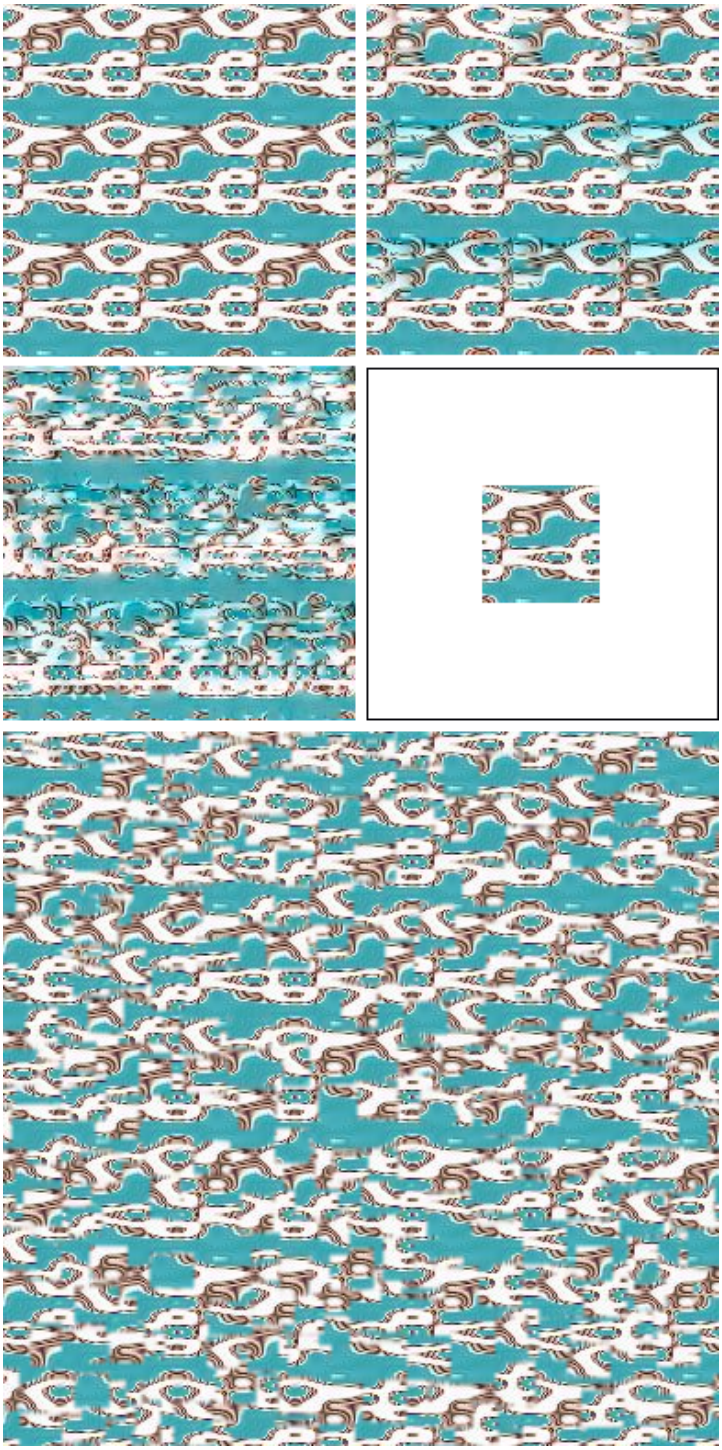

Figure 7: Comparison of image quality of the chaos mosaic and De Bonet's algorithm. Top left: a tiling of 3×3 tiles. Top right: De Bonet's result with the randomness threshold of 750. Middle left: De Bonet's result with the randomness threshold of 1250. Middle right: Input sample texture. Bottom: the result of the chaos mosaic.

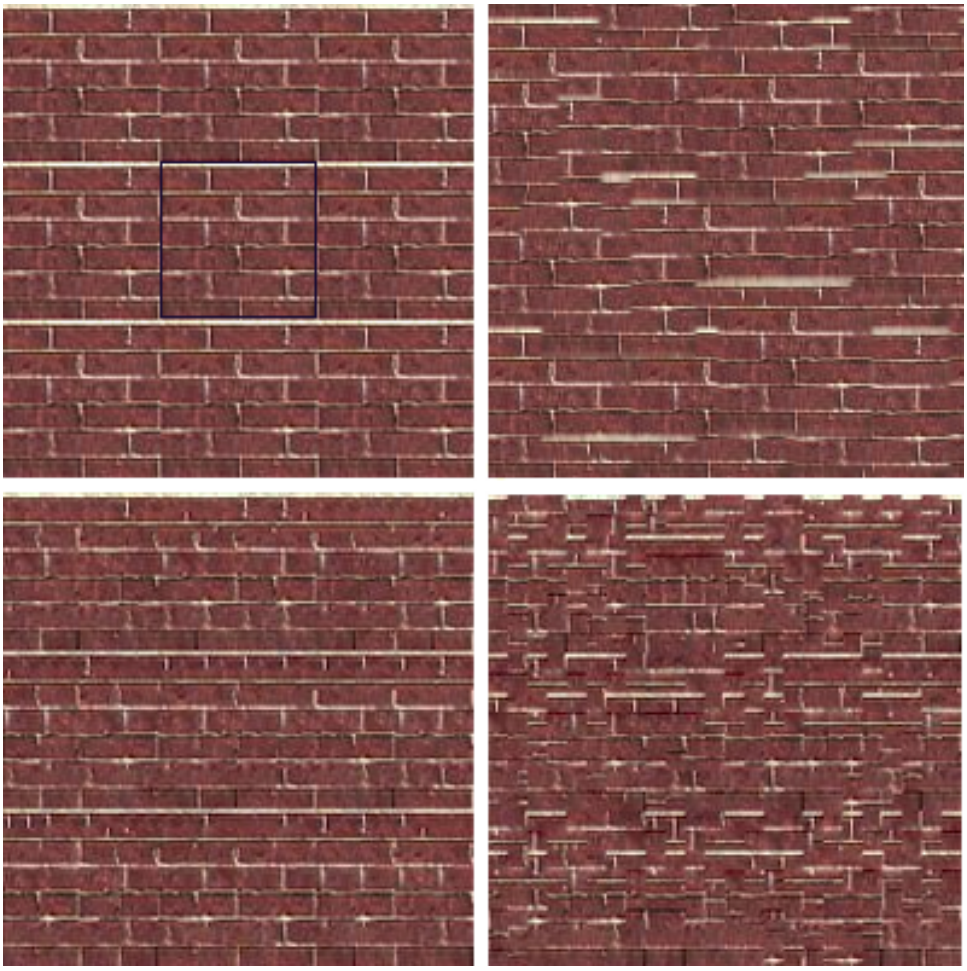

Figure 8: Handling repeated textures. Top left: The chaos mosaic zero cat map iteration. This is just a tiling of the input sample texture, which is indicated by the black box. Top right: The chaos mosaic with ten cat map iterations. Bottom left: De Bonet's result with randomness threshold of 750. Bottom right: De Bonet's result with randomness threshold of 1250.

cases, as Fig. 1, Fig. 3, Fig. 4, Fig. 7 exemplify, the mismatched features are not a serious problem and the cross-edge smoothing is sufficient to hide the edge of the random blocks. An exception is shown in Fig. 9. In this case, because the background is a light color and the features are small, the mismatched features and image blur caused by cross-edge smoothing are easily visible. Nevertheless, the synthesized texture in Fig. 9 (bottom) is still of reasonable quality. In contrast, if the local features are not captured at all as is the case with De Bonet's algorithm, the synthesized texture is dramatically worse as shown in Fig. 9 (top right).

4.3 Memory Efficient Texture Rendering

Fig. 10 shows the user interface of our interactive texture rendering system. Using the mouse, the user can select any part of a large texture for displaying through a $w \times w$ window, where $w = 256$ for us. For a large virtual texture, the synthetic texture is never created in its entirety; instead the system dynamically evaluates and displays the chosen $w \times w$ block of the texture. For a large "real" texture, we keep it in the disk storage. When the user has chosen a $w \times w$ block of the texture for display and the block is not currently in the main memory, the user will experience system latency .

With the virtual texture, our interactive texture rendering system allows us to view interactively very large textures, such as $100k \times 100k$ textures, without suffering system latency. To measure system latency for "real" textures stored on the hard drive, we have experimented with a $10k \times 10k$ texture on a PC running Microsoft Windows 98 on an AMD K7 processor. The PC has 128Mb of main memory and a 9 Gb Quantum Fireball hard drive. The lantency for the "real" texture on the hard drive can be as big as 1.0 second per frame and is on average 0.7 second per frame. This latency is mainly affected by the PC's caching strategy and the hard drive's the seeking speed. The viewing of the virtual texture is interactive at 0.017 second per frame.

Fig. 11 (bottom) is a snapshot from an interactive rendering system that uses the virtual texture for texture mapping. The pathway in the middle is texture mapped by on-the-fly evaluation of a large virtual texture and hence there is no visible repetition. The pathway is made of quads, each corresponding to a 512×512 texture block within a very large virtual texture. When a quad is become visible, its corresponding texture block is evaluated and mapped onto the quad. Note that the existing graphics API's (e.g. Direct

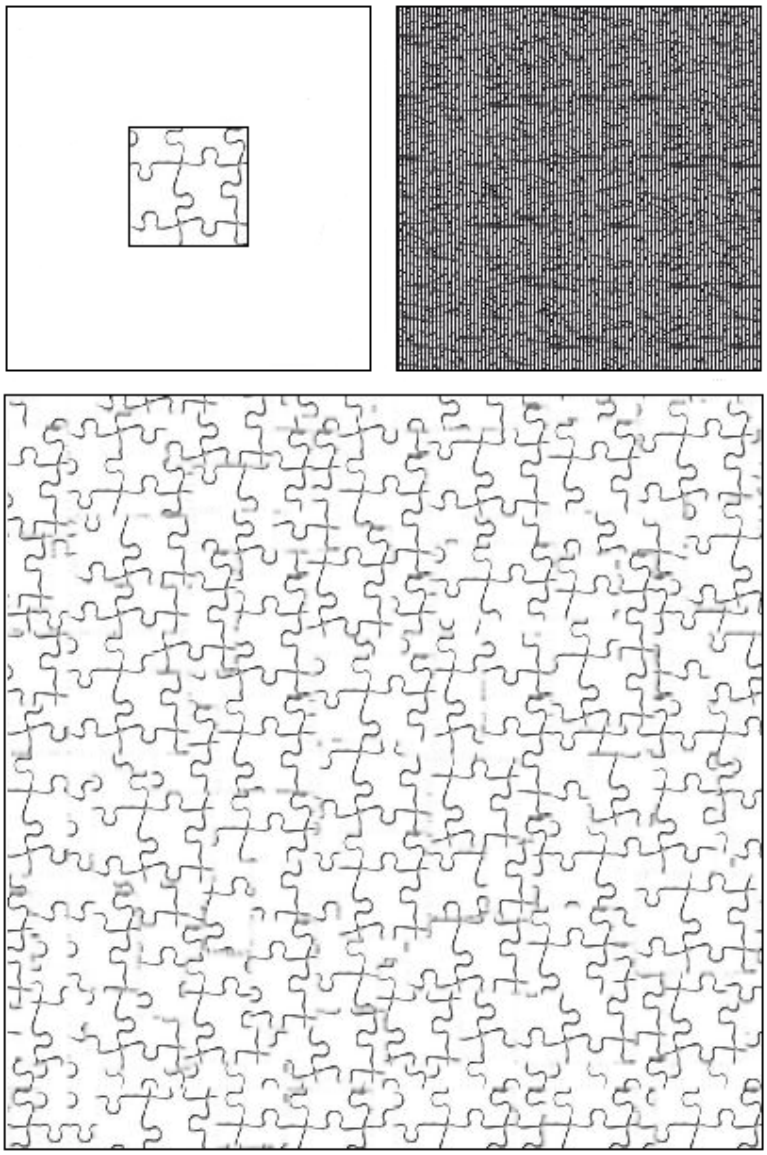

Figure 9: The effect of corss-edge smoothing on the quality of synthesized texture. Top left: the input sample texture. Bottom: the result of the chaos mosaic. Notice the mismatched features and image blur caused by the cross-edge smoothing. Top right: De Bonet's result with randomness threshold of 500. Results with other randomness thresholds are similar.

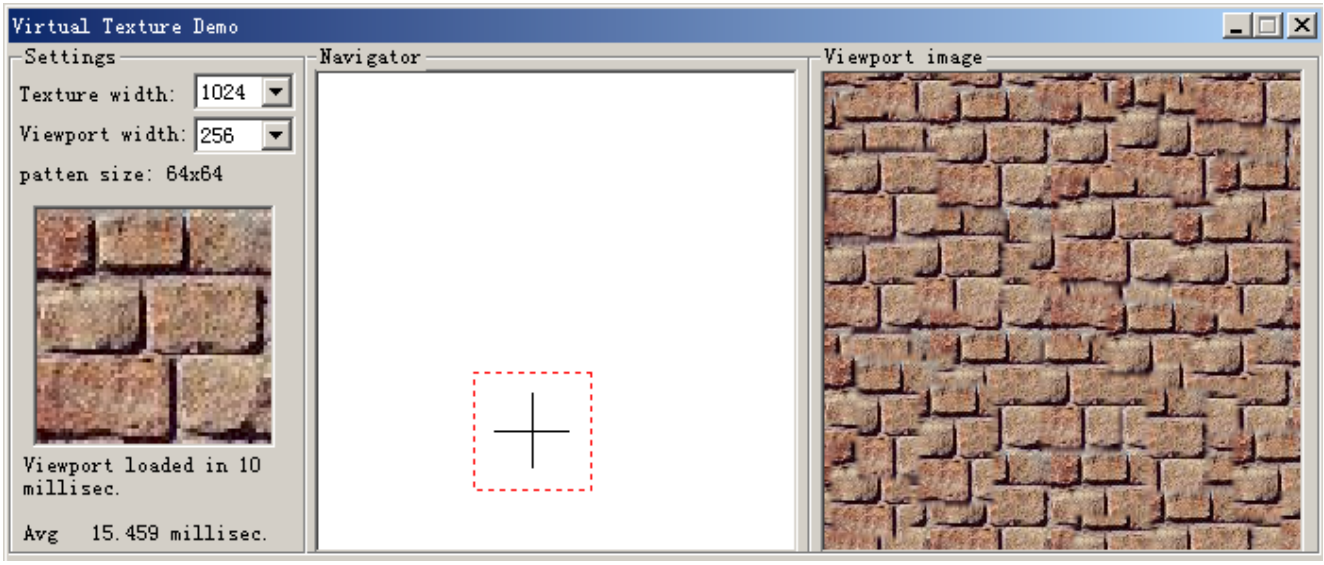

Figure 10: The user interface of our interactive texture rendering system. The navigator window corresponds to the entire virtual texture. The red box in the navigator window indicates the size and location of the evalauted texture block, which is displayed in the viewport. In the above example, the virtual texture is of size 1024×1024 and the viewport is of size 256×256. The user can view any part of the virtual texture by moving the red box around with the mouse.

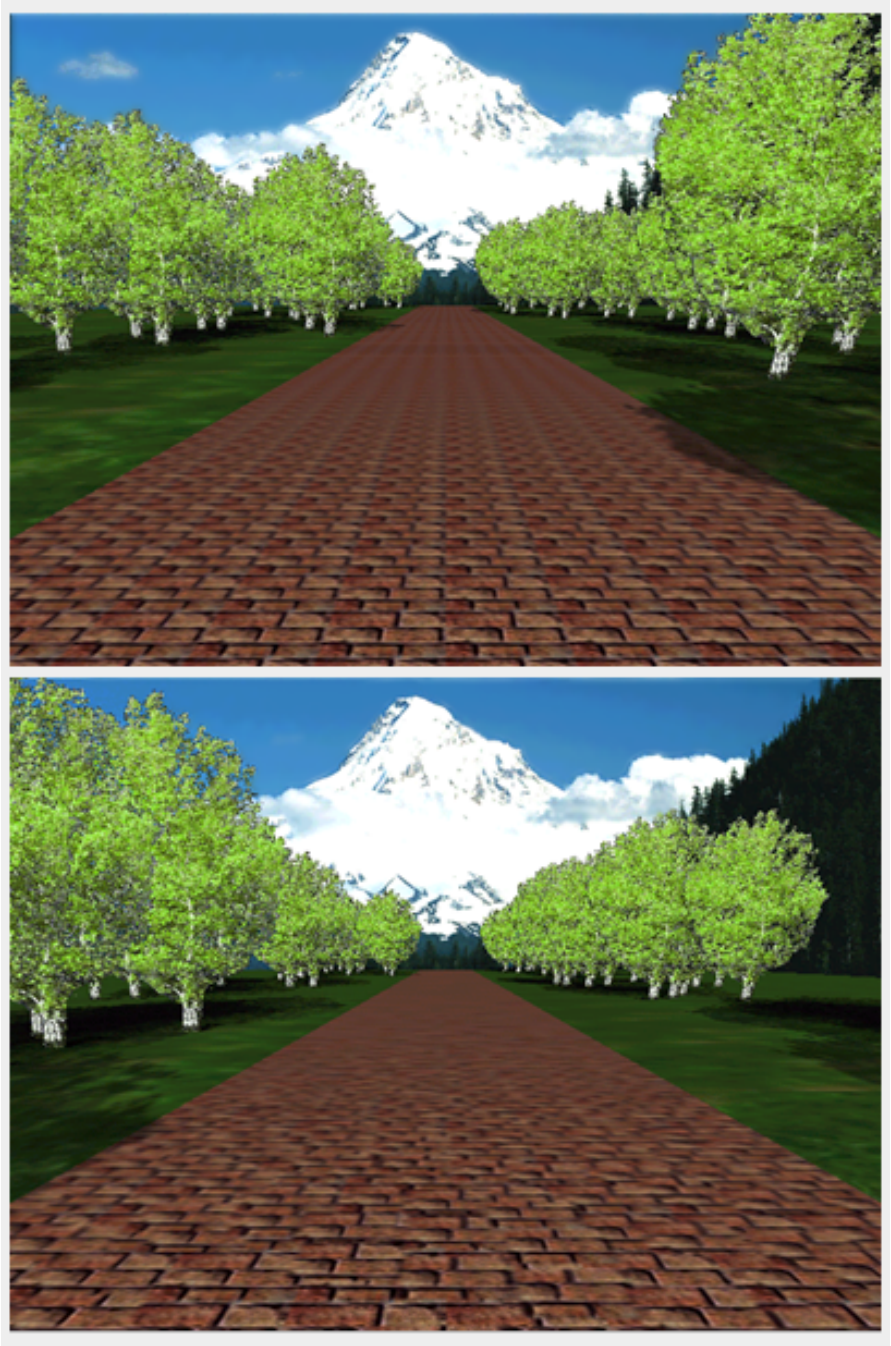

Figure 11: Snapshots of an interactive rendering system that uses the virtual texture for texture mapping. The pathway in the middle is texture mapped using two methods. Top: traditional texture mapping by repeating the input texture. Bottom: texture mapping with on-the-fly evaluation of a virtual texture constructed from the input sample texture.

3D) are not designed with the virtual texture in mind, and under these API's it is difficult to implement texture mapping with the virual texture in a general setting. Nevertheless, our system provides insight into the feasibility of such implementations. In terms of speed, our system is almost as fast as the conventional texture mapping shown in Fig. 11 (top), in which we can easily see the visible repetition on the pathway.

5 Conclusion

We have presented an algorithm for synthesizing large textures from an input texture sample. Our algorithm combines the strengths of traditional procedural methods and statistical sampling methods. Our method is fast. It facilitates memory efficient texture rendering by way of procedural texturing. Like statistical sampling methods, our method can synthesize a wide variety of textures as long as a sample texture is provided. Our method is easy to use and requires no parameter tweaking as traditional procedural methods do.

As Turk pointed out before [Tur91], acquiring texture, texture mapping, and texture sampling are interrelated taskes. In this work, we have shown that procedural technique can be combined with input texture sample to facilitate the process of acquiring textures and memory efficient texture rendering. For future work, we plan to extend the ideas presented here to texture synthesis on surfaces of arbitrary topology. Also of interest is the interplay between texture synthesis artifacts and the texture sampling strategy used in texture rendering. Finally, we are interested developing texture synthesis techniques that minimize the memory bandwidth during texture mapping.

References

[AA68] V. I. Arnold and A. Avez. *Ergodic Problems of Classical Mechanics*. Benjamin, 1968.

[BAC96] A. C. Beers, M. Agrawala, and N. Chaddha. Rendering from Compressed Textures. *Proceedings of SIGGRAPH 96*, August 1996.

[Cat74] E. E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Deptartment of Computer Science, University of Utah, December 1974.

[CBS98] M. Cox, N. Bhandari, and M. Shantz. Multi-Level Texture Caching for 3D Graphics Hardware. In *Proceedings of the 25th International Symposium on Computer Architecture*, 1998.

[CCC87] R. L. Cook, L. Carpenter, and E. Catmull. The Reyes Image Rendering Architecture . *Computer Graphics (Proceedings of SIGGRAPH 87)*, July 1987.

[De] J. S. De Bonet. `www.ai.mit.edu/~jsd/Research/TextureSynthesis/`.

[De 97] J. S. De Bonet. Multiresolution Sampling Procedure for Analysis and Synthesis of Texture Image. In *Computer Graphics Proceedings, Annual Conference Series*, pages 361–368, August 1997.

[EL99] A. A. Efros and T. K. Leung. Texture Synthesis by Non-Parametric Sampling. In *Proceedings of International Conference on Computer Vision*, 1999.

[FFC82] A. Fournier, D. Fussell, and L. Carpenter. Computer Rendering of Stochastic Models. *Communications of the ACM*, 25(6):371–384, June 1982.

[Gu90] Y. Gu. Evidences of Classical and Quantum Chaos in the Time Evolution of Non-Equilibrium Ensembles. *Physics Letters*, A149(2,3):95–100, September 1990.

[HB95] D. J. Heeger and J. R. Bergen. Pyramid-Based Texture Analysis/Synthesis. In *Computer Graphics Proceedings, Annual Conference Series*, pages 229–238, July 1995.

[HL90] P. Hanrahan and J. Lawson. A Language for Shading and Lighting Calculations. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4), August 1990.

[Jan69] R. Jancel. *Fundations of Classical and Quantum Statistical Mechanics*. Pergamon, 1969.

[Lew89] J.-P. Lewis. Algorithms for Solid Noise Synthesis. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, volume 23, pages 263–270, July 1989.

[Pea85] D. R. Peachey. Solid Texturing of Complex Surfaces. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3), July 1985.

[Per85] K. Perlin. An Image Synthesizer. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, volume 19, pages 287–296, July 1985.

[PKGH97] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering Complex Scenes with Memory-Coherent Ray Tracing. *Proceedings of SIGGRAPH 97*, August 1997.

[PS99] J. Portilla and E. Simoncelli. Texture Modeling and Synthesis Using Joint Statistics of Complex Wavelet Coefficients. In *Proceedings of IEEE Workshop on Statistical and Computational Theories of Vision*, 1999.

[Sch88] H. G. Schuster. *Deterministic Chaos: An Introduction*. VCH Verlagsgesellschaft mbH, 1988.

[TK96] J. Torborg and J. Kajiya. Talisman: Commodity Real-Time 3D Graphics for the PC. *Proceedings of SIGGRAPH 96*, August 1996.

[Tur91] G. Turk. Genereating Textures on Arbitrary Surfaces Using Reaction-Diffusion. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 289–298, July 1991.

[Ups89] S. Upstill. *The RenderMan Companion*. Addison Westley, 1989.

[WK91] A. Witkin and M. Kass. Reaction-Diffusion Textures. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 299–308, July 1991.

[ZLW99] S. C. Zhu, X. Liu, and Y. N. Wu. Exploring Texture Ensembles by Efficient Markov Chain Monte Carlo. In *Proceedings of IEEE Workshop on Statistical and Computational Theories of Vision*, 1999.

Appendix: The Chaos Behaviors of the Cat Map

If we cosider the torus as the $(p,\ q)$ phase space with $|p| \leq \frac{1}{2}$ and $|q| \leq \frac{1}{2}$, then the cat map iteration describes the chaotic motion of a point mass of one degree of freedom (DOF) subject to periodic external force [Jan69]. The Hamiltonian of a point mass with one DOF is

$$H(t) = \frac{1}{2}p^2 + \frac{K}{2}q^2 \sum_{m=-\infty}^{\infty} e^{i2\pi mt}$$

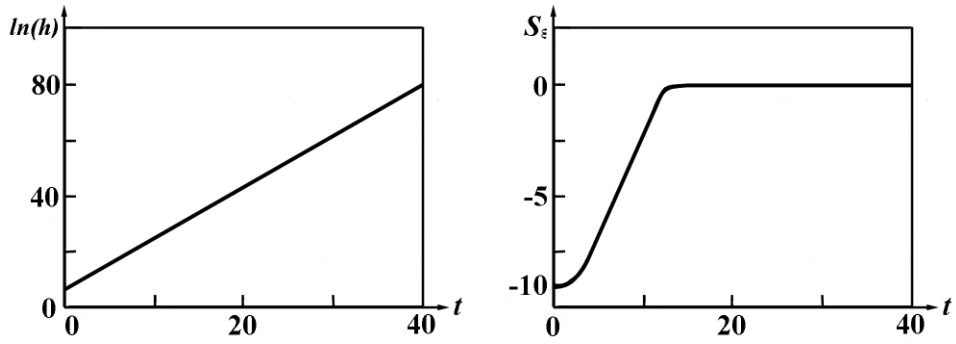

Figure 12: Left: the exponential growth of $h(\rho)$ confirms the chaotic behaviors of the cat map. Right: the fact that S_ϵ quickly approaches its maximum value after about ten iterations indicates that an even and visually stochastic distribution can be achieved in about ten ieterations.

The corresponding Hamilton's equations, after discretization, are

$$p_{k+1} = (p_k - Kq_k) \bmod 1 \tag{3}$$

$$q_{k+1} = (p_k + (1-K)q_k) \bmod 1 \tag{4}$$

The cat map corresponds to the case of $K = -1$.

As an non-equilibrium ensemble, the cat map iteration has two statistical quantities that describe its phase-space distribution. The coarse-grained entropy $S_\epsilon = -k\int_\Gamma \rho_\epsilon \ln(\rho_\epsilon) d\Gamma$, where ρ_ϵ represents the coarse-grained density of Ehrenfests [Jan69], is a measure of macroscopic uniformity of the phase-space distribution. The other quantity, which describes the microscopic heterogeneity of the phase-space distribution, is defined by $h(\rho) = \lim_{\epsilon\to 0} \frac{S_\epsilon - S}{\epsilon}$, where $S = -k\int_\Gamma \rho \ln(\rho) d\Gamma$ is the Gibbs entropy [Gu90]. The quantity $h(\rho)$ is important because whether the non-equilibrium ensemble exhibits deterministic stochasticity or chaos behaviors depends on whether $h(\rho)$ has exponential growth. For the cat map, the coarse-grained entropy

$$S_\epsilon = -\ln(\sum_{m_0,n_0} e^{-2\pi^2(\epsilon^2 a_k^2 + a^2)(m_0^2+n_0^2)})$$

where $a_k = \frac{m_k^2+n_k^2}{m_0^2+n_0^2}$ with

$$m_k = \frac{m_0}{2\sqrt{5}}((\sqrt{5}-1)\lambda_1^k + (\sqrt{5}+1)\lambda_2^k) - \frac{n_0}{\sqrt{5}}(\lambda_1^k - \lambda_2^k)$$

$$n_k = \frac{n_0}{2\sqrt{5}}((\sqrt{5}+1)\lambda_1^k + (\sqrt{5}-1)\lambda_2^k) - \frac{m_0}{\sqrt{5}}(\lambda_1^k - \lambda_2^k)$$

$$\lambda_1 = \frac{1}{2}(3 + \sqrt{5}) \text{ and } \lambda_2 = \frac{1}{2}(3 - \sqrt{5}).$$

The expression of $h(\rho)$ is more complex and can be found in [Gu90]. In Fig. 12, we plot S_ϵ and $h(\rho)$ as funtions of the discrete time $t = k$. The exponential growth of $h(\rho)$ confirms the chaotic behaviors of the cat map. The fact that S_ϵ quickly approaches its maximum value after about ten iterations indicates that an even and visually stochastic distribution can be achieved in about ten ieterations.