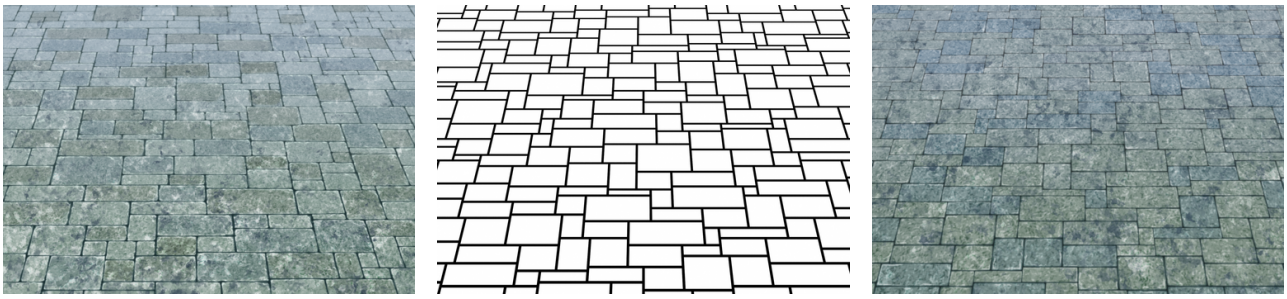
EUROGRAPHICS 2019 / P. Alliez and F. Pellacini
(Guest Editors)

Volume 38 (2019), Number 2

Generating Stochastic Wall Patterns On-the-fly with Wang Tiles

Alexandre Derouet-Jourdan[1], Marc Salvati[1] and Théo Jonchier[1,2]

[1]OLM Digital Inc, Japan
[2]ASALI-SIR, XLIM, France



(a) **(b)** **(c)**

Figure 1: *(a) Wall painted by an artist. (b) Stochastic wall pattern generated using our on-the-fly procedural algorithm. (c) Final texture after adding details. Our algorithm generates a line structure similar to the one created by the artist. This structure is used to generate brick colors and details around the edges. The global material appearance has been generated using texture bombing techniques.*

Abstract

The game and movie industries always face the challenge of reproducing materials. This problem is tackled by combining illumination models and various textures (painted or procedural patterns). Generating stochastic wall patterns is crucial in the creation of a wide range of backgrounds (castles, temples, ruins...). A specific Wang tile set was introduced previously to tackle this problem, in an iterative fashion. However, long lines may appear as visual artifacts. We use this tile set in a new on-the-fly procedure to generate stochastic wall patterns. For this purpose, we introduce specific hash functions implementing a constrained Wang tiling. This technique makes possible the generation of boundless textures while giving control over the maximum line length. The algorithm is simple and easy to implement, and the wall structure we get from the tiles allows to achieve visuals that reproduce all the small details of artist painted walls.

CCS Concepts

• ***Computing methodologies*** → ***Texturing;***

1. Introduction

The final look in movies and games is the result of the combination of textures, 3D models and illumination models. It is usually more efficient to increase the level of details through a texture rather than directly into the 3D model. With the continuous improvement in display technology (4k and 8k) and an increase of CPU/GPU power, always higher resolution textures are required. The cost of painting textures by hand is then increasing. Generating textures at render time that preserve the organic feeling of hand painted ones is the challenge that all shading artists face everyday. They typically combine noises and patterns (fractal, Perlin, Gabor, cellular, flakes, Voronoi...) with some well designed BSDF to re-create a material appearance.

Every movie and cartoon involves the creation of patterns to generate textures for backgrounds. In this context, we face the problem of generating stochastic wall patterns for stone walls and paved grounds as shown in Figure 2. These patterns appear in various constructions such as castles, temples or ruins. The problem we address in this paper focuses on generating the lines representing the edges of the bricks forming the wall. These patterns are different from a regular wall pattern with unique size of bricks (see Figure 3). Particularly, for aesthetic reasons, it is important that the generated patterns follow two constraints:

(a) (b)

Figure 2: *Examples of stochastic walls. (a) Painting. (b) Photography.*

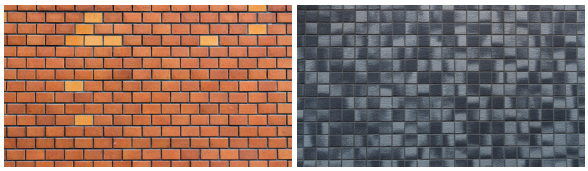


Figure 3: *Photographies of regular wall patterns. Left: alternate pattern. Right: square pattern.*

- No cross: pattern where 4 bricks share a corner.
- No long lines: brick with edges aligned in sequences.

This is an important request from our artists. They need to control the focus and the points of interest in their renderings. Cross patterns and long alignments of bricks tend to attract the eye (see Figure 18). This is backed up by works in visual perception, where it was shown that elongated blobs and crosses tend to be perceived at an early stage of vision and tend to stand out [Bec83, JB83]. So to prevent the background from attracting the eye of the viewer and put the focus on the foreground, it is crucial to prevent long lines and crosses from occurring in the wall pattern.

Previous efforts have been devoted to generating such patterns [LDG01, Miy90]. However, they do not always account for crosses or long lines. Moreover, they are offline techniques by nature, which means that the whole pattern must be created at once. In such a case, it is impossible to generate unbounded textures or to access only a part of it. In a production setting, being able to perform a GPU preview of the result or to generate arbitrarily large textures is important. This can be achieved by using an on-the-fly evaluation of fully procedural techniques (defined on a boundless domain), i.e aperiodic, parameterized, random-accessible and compact function as defined in [LLC*10].

Our goal is to generate stochastic wall patterns on-the-fly while accounting for crosses and long lines. Using such patterns, it is then easy to produce complete textures by adding colors to the bricks, noise on their edges and other details, as we present Section 3.3.

In a preliminary work, we introduced the use of Wang tiles to create walls with no cross patterns [DJMS15], albeit using an iterative algorithm and generating long lines artifacts. Wang tiles have become a standard tool in texture synthesis. They have the advantage of producing stochastic patterns, while providing access to the structure of the pattern. This feature is helpful to provide control over the rendering and shading of the structure once the pattern is generated. Also, as shown in previous work [LD05], it is possible to design on-the-fly procedural methods to evaluate Wang tilings.

We showed in a previous work [KDJO16] that the long lines artifacts is related to the orientations of the Wang tiles and that the issue can be addressed by precomputing their orientations in order to break the long lines. We called this the dappling problem and proposed an iterative algorithm to solve it.

In this paper, our contribution is to create a new and simple yet general procedural algorithm with on-the-fly evaluation that generates stochastic wall patterns while avoiding cross patterns. In addition, our algorithm provides full control over the maximum length of the lines. For this purpose, we propose an on-the-fly solution to the dappling problem we introduced in [KDJO16]. The procedural nature of the algorithm makes its GPU implementation straightforward (we provide one as additional material to the present paper).

After an overview of the related work, we explain the Wang tile model we use and the limitations of current algorithms. We then introduce our on-the-fly solution. Finally we present some result analysis and discussion about the iterative flavors of the pattern generation. We compare our results with state of the art texture synthesis methods and another wall generation algorithm before concluding about the future work.

Although we illustrate our method with the reproduction of hand painted walls with cartoon style, there is no restrictions to use our stochastic wall structure to render photo-realistic walls.

2. Related work

To create material appearance, artists use a combination of raster textures and procedural textures. There is no limit but the skill of the artist to what they may paint in a raster texture. Because of the need of very high resolution textures, multiple methods have been proposed to generate them. It is a very long studied problem and it is out of the scope of the present paper to discuss it extensively. In the following, we present a few techniques that have been proposed to tackle this problem. We can distinguish solutions proposed to this problem into two families, example-based techniques and purely generative ones.

2.1. Example-based texture synthesis

Texture synthesis generates large size textures based on an exemplar of limited size [DBP*15, KSE*03, LH06, YBY*13]. These methods either work by reproducing statistical properties of the exemplar [GLLD12] or by aggregating patches of the exemplar locally, satisfying local continuity of the structure in the larger texture [LH06, KNL*15]. Large textures can be generated on the fly [VSLD13], without storing at any time the result in memory. For an extensive overview of these methods, please consult [RDDM17]. As we show Section 5, the strong structure of the wall pattern is hard to capture for such techniques. They produce bend lines, non-rectangular bricks or arbitrarily long lines. Moreover, the control over the final result is limited. For example it would be very hard to control the color of individual bricks or the space between bricks of a wall pattern produced by such a method. The reason is that even if the visual structure is reproduced, the algorithm does not have the knowledge of the underlying structure of the pattern.

2.2. Rectangular packing and tiling

Outside of the computer graphics community, the problem of generating arrangements of rectangles has been extensively studied in combinatorics and discrete mathematics. Domino tiling consists in tiling a given polygon with rectangles of size 2×1 [KP80]. Tatami tiling is a constrained domino tiling where no four dominos share a corner, which correspond to our no cross constraint [ERWS11]. Another similar problem is the rectangular packing which consists in packing with no overlap as many rectangles of various sizes inside a given larger rectangle [LMM02]. In this case, the constraint is to minimize the area of the larger rectangle that is not covered.

These works have a great theoretical value but are hard to apply practically to our specific problem. Tatami tilings are limited to two different sizes of rectangles, and the rectangular packing doesn't account for crosses and leaves holes in the pattern.

2.3. Purely generative texture synthesis

Purely generative methods like noises or pattern arrangements create textures from a small set of parameters or a description of the wanted result [Per85, Wor96, LLDD09, SP16, LHVT17]. For example, texture bombing [Gla04] has been used successfully [SRVT14] to generate on the fly the equivalent of 100k textures while preserving the hand painted feeling. We use this technique to add details to the final result in Section 3.3.

Some dedicated techniques have been proposed [LDG01, Miy90] to create wall patterns. However the control over the size of the bricks and the occurrence of crosses and long lines remain an issue. Also the iterative nature of those methods prevents an on-the-fly evaluation. The technique we presented in [DJMS15] is based on Wang tiles [Wan61] which have been introduced in computer graphics to generate aperiodic stochastic textures [CSHD03, Sta97]. They have been proven to be usable in an on-the-fly context [LD05, SD10, Wei04]. Although the technique presented in [DJMS15] accounts for cross patterns, it cannot guarantee in itself that the lines will remain short. We introduced in [KDJO16] an algorithm that can be used to limit the lines' lengths in the pattern by precomputing the orientations of the Wang tiles. Unfortunately, this approach is iterative and limited to fixed size textures and cannot be adapted easily for an on-the-fly evaluation. Wang tiles have also been used to generate point distributions with a blue noise Fourier spectrum [KCODL06]. In that paper, the authors introduce a hierarchical Wang tiling algorithm. Essentially, the algorithm first generates a tiling with a low resolution. Then the tiling is locally refined. This way, the algorithm can produce point sets with non uniform density.

In this paper, we propose a new procedural technique to evaluate on-the-fly a Wang tiling specially tailored for the Wang tiles designed in [DJMS15]. Our technique enforces a fixed upper bound on the lines' lengths in the pattern. It is inspired by [KCODL06]. We first generate a subdivision of the texture space which has the property of limiting the lines' lengths, based on the ideas from [KDJO16]. Then we construct the final tiling following the subdivision. The Wang tiles we use allows us to solve both steps at once, by generating the subdivision implicitly.

3. Walls with Wang tiles

In this section we describe how we use Wang tiles to generate the visual goal. First we recall the Wang tile model as introduced in [DJMS15] and then we explain how to use the generated structure to render the walls.

3.1. Wang tiles

Wang tiles are square tiles with colors on the edges as shown in Figure 4. Tiles are placed edges to edges in the tiling space. A tiling is valid when every two tiles sharing an edge have the same color on this edge. We identify the problem of tiling to a problem of edge coloring like in [LD05]. In that configuration, for a tile in (i, j), we denote $H_{i,j+1}$, $V_{i,j}$, $H_{i,j}$ and $V_{i+1,j}$ the top, left, bottom and right edges, as well as their colors. Wang tiles have the characteristics of

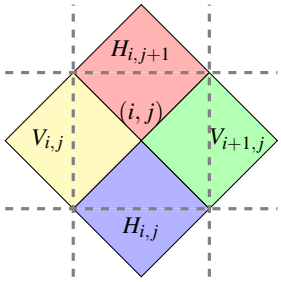


Figure 4: *Model of Wang tile. In a valid tiling, the color of a shared edge is the same for the two tiles sharing the edge. The tile (i, j) is made from the colors $H_{i,j+1}$, $V_{i,j}$, $H_{i,j}$ and $V_{i+1,j}$*

providing local continuity on the edges of the tiles.

3.2. Wang tile model

The stochastic wall patterns we generate consist of a set of rectangular bricks in various sizes. They don't contain cross patterns as these break the randomness and attract the eye. That is the reason why, when painting by hand, artists avoid such patterns (see Figure 2 (a)).

Expressing stochastic wall patterns with Wang tiles is straightforward. Each tile models the corner junctions of four bricks. This is done by mapping the four colors of each tile to the bricks' edges' placement as shown in Figure 5. However to avoid non rectangu-

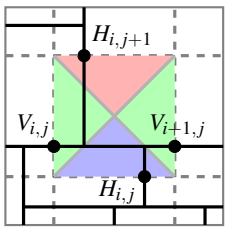


Figure 5: *Modeling a wall pattern with Wang tiles. Tile colors are mapped to the bricks edges positions.*

lar bricks (see Figure 6 (c) (d) (e)) and avoid cross patterns (see

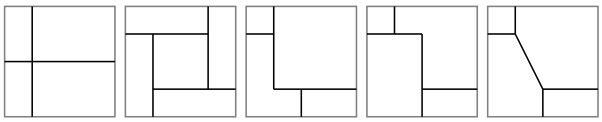


Figure 6: *The tiles removed from the Wang tile set. (a) cross tiles ($H_{i,j+1} = H_{i,j}$ and $V_{i+1,j} = V_{i,j}$). (c) (d) (e) non rectangular tiles/(b) extra brick tiles ($H_{i,j+1} \neq H_{i,j}$ and $V_{i+1,j} \neq V_{i,j}$).*

Figure 6 (a)), the tile set is restricted. To enforce rectangular bricks and avoid cross patterns, only "vertical" and "horizontal" tiles are considered. These constraints are shown in Figure 7 and formulated in Equations (1) and (2) . This avoids all the tiles introducing non

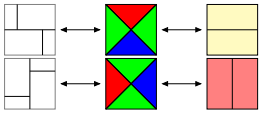


Figure 7: *Correspondence between the brick pattern(left), the Wang tile (center) and the orientation (right). Top line describes horizontal tiles, and bottom line vertical ones.*

rectangular bricks (such that $H_{i,j+1} \neq H_{i,j}$ and $V_{i+1,j} \neq V_{i,j}$).

$$\text{Vertical constraint } H_{i,j+1} = H_{i,j} \text{ and } V_{i+1,j} \neq V_{i,j} \quad (1)$$

$$\text{Horizontal constraint } H_{i,j+1} \neq H_{i,j} \text{ and } V_{i+1,j} = V_{i,j} \quad (2)$$

This also excludes tiles introducing an extra brick in the center (Figure 6 (b)). It would not be complicated to consider them, but they introduce tiny bricks that stand out. For n_c colors, we obtain a tile set of size $2 * n_c^2 * (n_c - 1)$ (see Figure 8 for the tile set with 3 colors, 36 tiles)

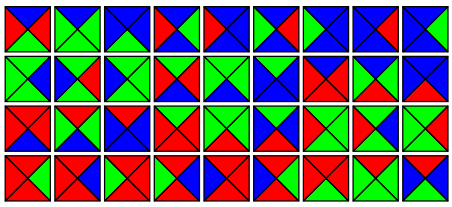


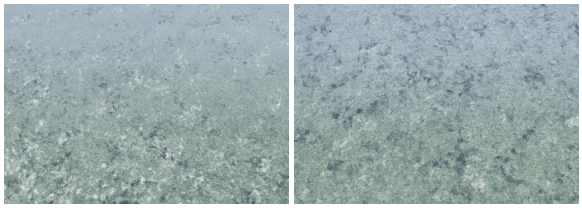
Figure 8: *Tile set example with 3 colors, 36 possible tiles.*

3.3. Rendering and shading

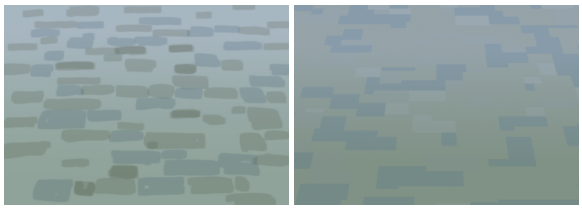
In this section we focus on the visual appearance of the bricks, given a stochastic wall structure of rectangular bricks without cross patterns. Observing the hand painted wall (see Figure 2), we notice the following features created by the artist:

- The global rock material.
- Color variation for each brick.
- Variable space between the bricks and corner roundness.
- Highlights and scratches near the edges of the bricks.

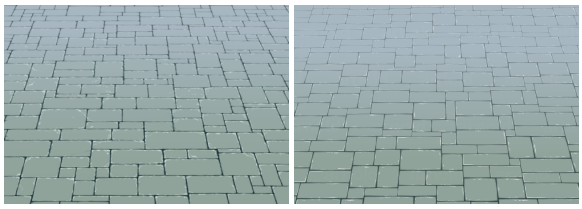
In Figure 9, we decompose the painting and our rendering in those features. We then explain how we use the stochastic wall pattern to reproduce all those features.



(a) *The global appearance: artist (left), texture bombing (right).*



(b) *The brick color painted: artist (left), from wall structure (right).*



(c) *The lines and edge highlights: artist (left), from brick parameterization (right).*

Figure 9: *Comparison of the features of both artist painted wall (left) and the generated wall rendering (right).*

Space between bricks, corner roundness and edge scratches are features localized near brick edges. Thanks to the brick structure we can generate a Cartesian parameterization as well as a polar parameterization that includes corner roundness (see Figure 10). From that structure, it's easy to generate normal maps to be used in a photo-realistic context.

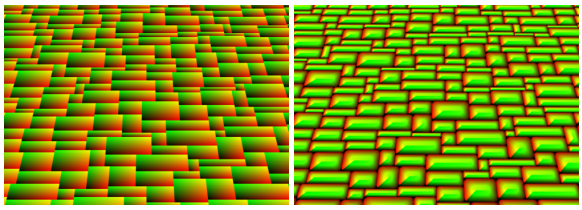


Figure 10: *UV parameterization of brick. Cartesian (left), polar parameterization with corner roundness (right).*

Distance to brick edges can be combined with some noise function to create irregular space between bricks (see Figure 9 (c)). The same distance can also be used to localize and apply highlights and scratch textures (see Figure 9 (c)). The color variation is simply enforced by generating a color based on the brick id or

center position (see Figure 9 (b)). We can also use a texture map to determine this color. As for the global material appearance it may be obtained by a combination of hand-painted and procedural textures. In our case we use texture bombing techniques as in the brush shader [SRVT14] to generate all those details while keeping the hand-painted touch (see Figure 9 (a)).

Without the wall/brick structure, such visual features would be really difficult to reproduce.

4. Algorithm

This section presents our procedural algorithm for wall pattern generation. As seen in various procedural texture generation papers like [Wor96], we make a correspondence between a given request point $P(x,y)$ and an underlying "virtual" grid cell (i,j) (for example integer part of scaled coordinates). In our algorithm, we build two functions $h'(i,j)$ and $v'(i,j)$ implementing procedural tiling while avoiding long lines:

$$H_{i,j} = h'(i,j) \quad V_{i,j} = v'(i,j).$$

In the following sections, we start by explaining the long line problem inherent to [DJMS15]. We start by giving a general procedural solution (Section 4.2) that do not consider the long line problem. Then we explain how to build h' and v' on top of the general solution to avoid long lines procedurally in Section 4.3.

4.1. The long line problem

The Wang tile set we use has been introduced along with an algorithm to generate a repeatable wall pattern in [DJMS15]. However long lines occur in walls generated by this algorithm (see Figure 11 and 12).

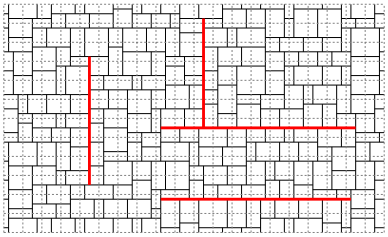


Figure 11: *Long lines stand out and attract the eye in a wall pattern.*

The reason behind the occurrence of long lines in Figure 11 is simple. Tiles are separated in two categories: the vertical and horizontal ones. So the probability to have a horizontal/vertical line of more than three tiles length is $1/2^3 = 0.125$. The occurrence of long lines on the side in Figure 12 is inherent to the sequential algorithm of [DJMS15]. We can see that the length of the lines on the side is increasing with the number of connections. In a sequential algorithm, the last tile of a row is solved with 3 constraints. With n_c connections the probability that the last constraints match is roughly $1/n_c$. This means that the probability the last tile of a row is horizontal is roughly $1/n_c$. The more connections you have, the longer the vertical lines on the side are. For the same reasons long horizontal lines occur on the top. The stochastic variation of Wang tiling [KCODL06] does not suffer from this limitation. However the stochastic nature of the algorithm makes difficult the control of the length of the straight lines shown in Figure 11. The dappling algorithm of [KDJO16] can be combined to the sequential algorithm of [DJMS15] to solve the issue of long lines for non-repeatable walls. We use that algorithm as a general reference to compare against our procedural approach in Section 5.

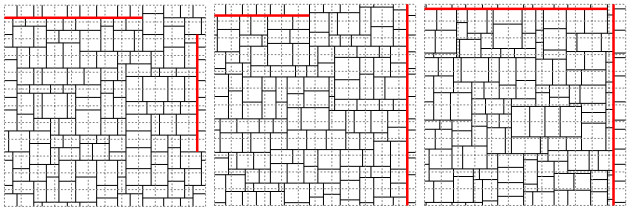


Figure 12: *Notice the long lines on the side when the number of connections is increasing (from left to right: 3, 5, 10 connections).*

4.2. General solution

The general on-the-fly tiling method is based on a property of our Wang tiles set, enunciated in [DJMS15], that a 2×2 square can always be tiled, for any boundary coloring.

The idea is to cut the grid into 2×2 squares. We need to determine the color of their outer edges.

For each cell (i,j), we compute the bottom left cell index (i',j'), which identify a 2×2 square, with

$$i' = i - i\%2 \quad j' = j - j\%2,$$

where $x\%y$ is the positive remainder in the Euclidean division of x by y.

We use h and v to associate pseudo-random values to the outer edge color ($H_{i',j'+2}$, $H_{i',j'}$, $H_{i'+1,j'}$, $H_{i'+1,j'+2}$, $V_{i',j'+1}$, $V_{i',j'}$, $V_{i'+2,j'}$, $V_{i'+2,j'+1}$) as shown in Figure 13 (a):

$$h(i,j) = \mathcal{H}(i,j)\%n_c \quad v(i,j) = \mathcal{V}(i,j)\%n_c,$$

where $\mathcal{H}$ and $\mathcal{V}$ are hash function and associate "random" integer values to (i,j). In practice we use a FNV hash scheme [FNV91] to seed a xorshift random number generator from which we pick a value.

However to satisfy the constraints (Equations (1) and (2)) of our Wang tile model, we cannot use pseudo-random values for the inner edge. We compute their colors using the result from [DJMS15]: we solve the interior of the 2×2 square by considering the different cases on the border. This gives us the colors $H_{i',j'+1}, H_{i'+1,j}$ and $V_{i',j'+1}, V_{i'+1,j'+1}$. Denoting h_2 and v_2 the function combining the two coloring of the edges (both outer and inner edges of 2×2), we have

$$H_{i,j} = h_2(i,j) = \begin{cases} h(i,j) & \text{if } j\%2 = 0 \\ \text{solved by } 2\times 2 \text{ solver} & \text{otherwise,} \end{cases}$$
$$V_{i,j} = v_2(i,j) = \begin{cases} v(i,j) & \text{if } i\%2 = 0 \\ \text{solved by } 2\times 2 \text{ solver} & \text{otherwise.} \end{cases}$$

An example of 2×2 solution is shown in Figure 13 (b). [DJMS15] proves that there is a solution, without explicitly giving

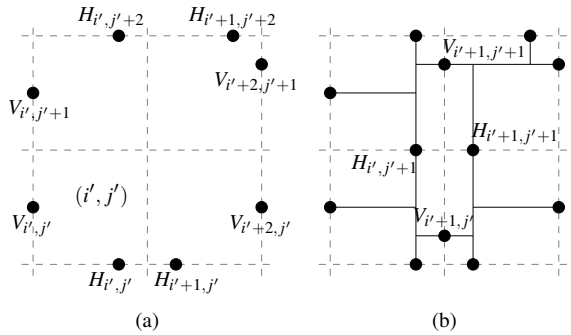


Figure 13: *Given a set of colors (a), an example solution (b).*

it. Solution can be built by constraints based or backtracking algorithms. In our implementation, we choose to consider all possible cases of color matching for opposite edges. It leads to 16 possible cases that are listed in appendix A.

4.3. Restricting the line length with dappling

In Figure 14, we can see how the dappling algorithm of [KDJO16] can produce random distributions while avoiding the long lines. The vertical tiles are represented in red and the horizontal ones in

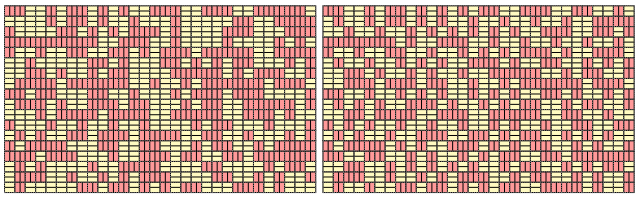


Figure 14: *Left: without dappling, we can notice the long continuous horizontal and vertical lines. Right: iterative dappling algorithm from [KDJO16] (maximum line length of 2)*

yellow. Vertical successions of red tiles and horizontal successions of yellow tiles result in long lines. By combining this dappling algorithm with the previous tiling algorithm of [DJMS15], it is possible to generate stochastic patterns with a given maximum length of lines. However this approach is strictly iterative. We propose an on-the-fly method to achieve similar results and restrict the longest line length to an arbitrary value n.

The idea behind the dappling in [KDJO16] is to traverse the configuration diagonally and correct the dappling when the number of consecutive horizontal or vertical tiles is too large. It is not possible to use this method in an on-the-fly fashion as it necessitates constructing the whole configuration until the requested cell. What we propose is to force the correction, that is, we "preemptively" correct the dappling automatically, whether the correction is necessary or not. In the following, we explain the core idea behind the on-the-fly evaluation of the dappling. Then, we explain how to evaluate on-the-fly a tiling that satisfies the dappling constraints directly, without having to build explicitely the dappling beforehand.

We start with the case of an upper bound n greater than 2, and then consider the special case of $n = 2$ which is simpler and can be treated more efficiently.

4.3.1. Case of n > 2

For a maximum number of n consecutive tiles of a same orientation, we use 2×2 checkerboards (see Figure 15) on diagonals. Each diagonal is separated from the other one by $n-2$ cells horizontally and vertically (see Figure 16). The cells inbetween the

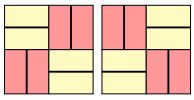


Figure 15: *2 possible checkerboard patterns.*

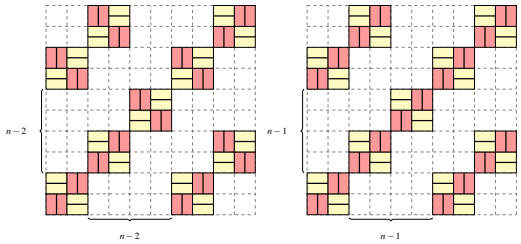


Figure 16: *Dappling pattern for even n (left), odd n (right). For even values of n, we use both checkerboard patterns randomly. For odd values of n, we pick at random one of the two checkerboard patterns and use it everywhere.*

2×2 checkerboards are not constrained and can be oriented in any way. It is easy to see that in such a case, there are no alignment of more than n consecutive tiles with the same orientation ($n-2$ for the cells inbetween and $+2$ for the checkerboards). Because the checkerboards are of size 2×2, this technique only works for even values of n. It is possible to solve the same problem for odd values of n: we use only one type of checkerboard and separate each diagonal from the other one by $n-1$ cells horizontally and vertically.

To include it in the general solution of Section 4, we need to adjust the hash functions h_2 and v_2 and replace them by dappling enabled ones h' and v'.

2×2 checkerboards are aligned on the diagonal: $i'\%n = j'\%n$, with $i' = i - i\%2, j' = j - j\%2$. Considering a checkerboard with its bottom left cell in (i', j'), there is exactly one horizontal tile between two horizontal edges opposed on the outer border, $H_{i',j'}$ and $H_{i',j'+2}$. It means that we necessarily have $h'(i',j') \neq h'(i',j'+2)$. The same reasoning applies for v'. So we need to create h' and v' to enforce that condition.

$$h'(i,j) = \begin{cases} h_{d1}(i,j,h(i,j+2)) & \text{if } j\%2 = 0 \text{ and } i'\%n = j'\%n \\ h(i,j) & \text{if } j\%2 = 0 \text{ and } i'\%n \neq j'\%n \\ \text{solved by } 2\times 2 \text{ solver} & \text{if } j\%2 \neq 0 \end{cases}$$

$$v'(i,j) = \begin{cases} v_{d1}(i,j,v(i+2,j)) & \text{if } i\%2 = 0 \text{ and } i'\%n = j'\%n \\ v(i,j) & \text{if } i\%2 = 0 \text{ and } i'\%n \neq j'\%n \\ \text{solved by } 2\times 2 \text{ solver} & \text{if } i\%2 \neq 0 \end{cases}$$

This defines all the outer edges of the 2×2 squares. We then use the general solution to solve each 2×2 locally. h_{d1} and v_{d1} compute random colors different from their input and are defined in Appendix A.

4.3.2. Case of n = 2

In the special case of $n = 2$, we just need to generate a dappling with randomly chosen 2×2 checkerboards (from Figure 15).

The idea to create a hash function that generate such a dappling is to consider the grid by pack of four cells, enforcing in the middle the checkerboard condition (opposite outer edge color are different). Similarly to the case n>2, we get the condition $h'(i',j') \neq h'(i',j'+2)$. Since in this case $n-2=0$, there is another checkerboard with its bottom left corner in $(i',j'+2)$, meaning $h'(i',j'+2) \neq h'(i',j'+4)$. Combining the two constraints we get $h'(i',j'+2) = h_{d2}(i',j'+2,h'(i',j'),h'(i',j'+4)$. If we choose $h'(i',j')$ and $h'(i',j'+4)$ arbitrarily, we can compute $h'(i',j'+2)$. The same reasoning apply to v' and can be summed up as

$$h'(i,j) = \begin{cases} h_{d2}(i,j,h(i,j-2),h(i,j+2)) & \text{if } j\%4 = 2 \\ h(i,j) & \text{if } j\%4 = 0 \\ \text{solved by } 2\times 2 \text{ solver} & \text{otherwise.} \end{cases}$$

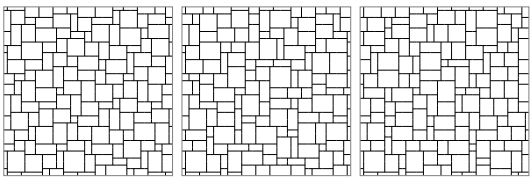
$$v'(i,j) = \begin{cases} v_{d2}(i,j,v(i-2,j),v(i+2,j)) & \text{if } i\%4 = 2 \\ v(i,j) & \text{if } i\%4 = 0 \\ \text{solved by } 2\times 2 \text{ solver} & \text{otherwise.} \end{cases}$$

h_{d2} and v_{d2} compute random colors different from their input and are defined in Appendix A.

5. Results and Discussion

5.1. Visual results and comparison

Thanks to our algorithm, we can control over the maximum length of the lines of the stochastic wall pattern. Output result of the pattern for maximum line length $n = 1$, $n = 3$, $n = 5$ are shown in Figure 17. We could also reproduce the artist painted wall patten as shown in Figure 1.



$n = 1$ $n = 3$ $n = 5$

Figure 17: *Output of our algorithm for $n = 1$, $n = 3$, $n = 5$*

The algorithm from [DJMS15] has been used in production and long lines were standing out (Figure 18 (left)). Limiting the maximum length to $n = 2$ with our new algorithm, the output looks much more natural (Figure 18 (right)).

Existing methods failed to reproduce our wall pattern structure. We tried the box packing method from [Miy90] (see Figure 19). It produces elongated brick and does not give control over cross

Figure 18: *Production result using [DJMS15] (left), using our algorithm ($n = 2$) (right). Our method breaks the long lines artifacts behind the character. These lines attract the eye, breaking them restores the focus on the character. Pokémon Generations Episode 3: The Challenger : ©2016 Pokémon. ©1995-2016 Nintendo/Creatures Inc. /GAME FREAK inc.*

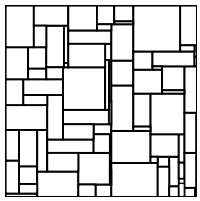
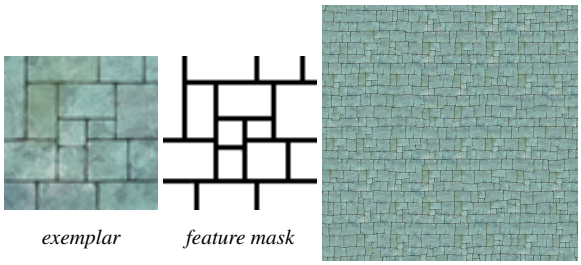


Figure 19: *Box packing method*

patterns or line length. We have access to the structure of the bricks, but the method is iterative and cannot be evaluated on-the-fly.

Figure 20 and 21 we show the results of two patch-based methods [LH06, KNL*15] on a hand-painted wall sample. These two methods are designed to take into account not only the color of each pixel but also its distance to features like the brick edges in our case. Unfortunately, texture synthesis methods can only reproduce approximative look, introducing holes or bending the lines, which makes them unsuited for production.

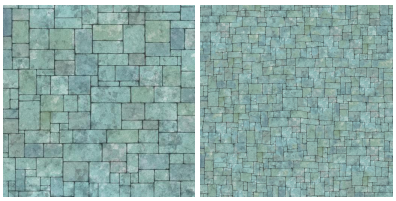


exemplar *feature mask* *output*

Figure 20: *Result of [LH06]. The mask is used to compute the distance to the bricks' edges. The computation time is low (just a few seconds). Although this method manages to keep the general aspect of a wall, we can see that a lot of bricks have non rectangular shapes and that some edges are not properly connected.*

5.2. GPU implementation

The GPU implementation of the procedural approach was straightforward in OpenGL3 and WebGL (we provide a ShaderToy implementation of the case $n = 2$). We use integer based hash functions



exemplar *output*

Figure 21: *Result of [KNL*15]. The result is impressive and close to our goal. However, we can still see bended lines and non rectangular bricks. Also, the computation time of the authors' implementation is large (around 40 minutes).*

in our production implementation. However, the integer support seems to be limited in OpenGL, and we then switched to different, float based hash functions.

We reach real time performance even with high number of pixel samples as shown in Table 1.

Pixel Samples	1	2	8	16	32
VGA(1024x768)	520	515	450	180	148
HD(1920x1080)	200	220	187	70	60

Table 1: *GPU algorithm performance (in frame per seconds). GPU results have been obtained using a Nvidia Quadro K620.*

5.3. Computation time and discussion

We measure and compare the performance of our on-the-fly evaluation against the general iterative algorithm, combination of the sequential algorithm of [DJMS15] with the solution to the dappling problem of [KDJO16]. Both algorithms give the same kind of results, and computation time are given in Table 2. Although

	Initialization	Time for 10^7 computation (x NP)
S/SD	2.1/2.5	790 (1x)
F/FD	0	3320 (4.2x)/3726 (4.7x)

Table 2: *CPU algorithms performance measured in ms. Time for the retrieval of a brick, using a 100x100 grid. Sequential algorithm without/with dappling (S/SD $n = 2$). On-the-fly evaluation without/with dappling(F/FD $n = 2$)*

the on-the-fly evaluation is 4 to 5 times slower than the iterative version, in production rendering context, this timings remain negligible compared to the full rendering times. It represents 2% of our wall rendering time (roughly 2 minutes per frame). The computation time difference comes from the multiple evaluation of the same tile in the on-the-fly approach. To compute one tile, it is necessary to compute some local neighboring tiles. Without memory, when we compute the next tile, some of the same neighboring tiles are computed again. We are confident that this computation time can be reduced by a carefully designed local cache system to avoid some re-computations.

We evaluate the quality of the dappling results by computing histograms of the number of consecutive tiles of same orientation in rows (see Figure 22). We use the original dappling method from [KDJO16] for the sequential algorithm, because it is the most general solution as far as we know. Figure 22 shows that the proportion of lines with the same length are similar with either on-the-fly evaluation and sequential algorithms. However we found a flaw in the original algorithm from [KDJO16]. In the case $n = 4$, the number of lines of length 4 is oddly equivalent to line of length 3. Our assumption is that every lines over 4 of length will be clamped to 4 and then artificially increase their occurrence, which inserts a bias in the result. Our on-the-fly evaluation technique produces a better line length distribution than the previous method.

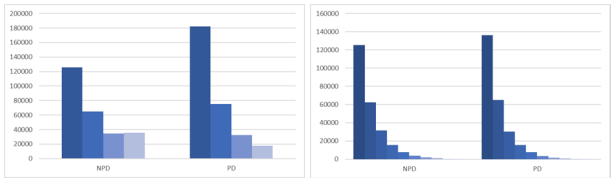


Figure 22: *Histogram of the vertical line length occurrence. Comparison between non procedural (NPD) and procedural (PD) version. Left: with maximum line length $n = 4$. Right: with maximum line length $n = 10$.*

The iterative approach is faster than our algorithm. It requires memory proportional to the number of bricks. It is also customizable and allows the caching of per brick information (color, randomization values...). In the case of a small wall pattern, using memory is not a big issue. But the fact that the memory used by this approach is unbounded makes it problematic in the case of a very large wall when the memory budget is tight. Our on-the-fly method generates unbounded textures without any repetitions, without using memory, But its cost is an increased computation time because it needs to recompute all the bricks information constantly. We believe that this cost could be reduced in some cases by crafting a bounded memory local cache system. We are currently investigating this approach.

6. Conclusion and future work

By designing custom hash functions for our specific problem, we succeed to provide a simple yet general solution to the generation of stochastic wall patterns. Our algorithm is fully procedural, avoids common visual artifacts and gives control to the user over the maximum line length. The computation overhead is low and the GPU implementation enables preview of the result, making it ready to integrate into movie production pipeline, extending artists' creation palette.

We are now considering the inclusion of multi resolution Wang tiles ([KCODL06]) or the combination of various sets of tiles to enable more variations in the brick sizes and patterns. We are also thinking about using border constrained 2D Wang tiling solutions in the context of texture synthesis. We are also working on the 3D procedural texture generation of stochastic wall patterns, and we

think about extending those results to general voxelization problems. Our intuition is that the stochastic structure of the underlying grid may improve the quality of volume rendering and collision detections.

References

[Bec83] BECK J.: Textural segmentation, second-order statistics, and textural elements. *Biological Cybernetics 48*, 2 (Sep 1983), 125–130. URL: https://doi.org/10.1007/BF00344396, doi:10.1007/BF00344396. 2

[CSHD03] COHEN M. F., SHADE J., HILLER S., DEUSSEN O.: Wang tiles for image and texture generation. In *ACM SIGGRAPH 2003 Papers on - SIGGRAPH 03* (2003), Association for Computing Machinery (ACM). URL: http://dx.doi.org/10.1145/1201775.882265, doi:10.1145/1201775.882265. 3

[DBP*15] DIAMANTI O., BARNES C., PARIS S., SHECHTMAN E., SORKINE-HORNUNG O.: Synthesis of complex image appearance from limited exemplars. *ACM Transactions on Graphics 34*, 2 (mar 2015), 1–14. URL: http://dx.doi.org/10.1145/2699641, doi:10.1145/2699641. 2

[DJMS15] DEROUET-JOURDAN A., MIZOGUCHI Y., SALVATI M.: Wang Tiles Modeling of Wall Patterns. In *Symposium on Mathematical Progress in Expressive Image Synthesis (MEIS2015)* (2015), vol. 64 of *MI Lecture Note Series*, Kyushu University, pp. 61–70. 2, 3, 5, 6, 7, 8

[ERWS11] ERICKSON A., RUSKEY F., WOODCOCK J., SCHURCH M.: Monomer-Dimer Tatami Tilings of Rectangular Regions. *Electronic Journal of Combinatorics 18*, 1 (2011). URL: http://www.combinatorics.org/Volume_18/Abstracts/v18i1p109.html. 3

[FNV91] FOWLER G., NOLL L. C., VO P.: Fowler / Noll / Vo (FNV) Hash, 1991. URL: http://isthe.com/chongo/tech/comp/fnv/. 5

[Gla04] GLANVILLE R. S.: Texture bombing. In *GPU Gems*, Fernando R., (Ed.). Addison-Wesley, 2004, pp. 323–338. 3

[GLLD12] GALERNE B., LAGAE A., LEFEBVRE S., DRETTAKIS G.: Gabor noise by example. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2012) 31*, 4 (July 2012), 73:1–73:9. doi:10.1145/2185520.2335424. 2

[JB83] JULESZ B., BERGEN J. R.: Human factors and behavioral science: Textons, the fundamental elements in preattentive vision and perception of textures. *Bell System Technical Journal 62*, 6 (1983), 1619–1645. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1983.tb03502.x, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.1538-7305.1983.tb03502.x, doi:10.1002/j.1538-7305.1983.tb03502.x. 2

[KCODL06] KOPF J., COHEN-OR D., DEUSSEN O., LISCHINSKI D.: Recursive wang tiles for real-time blue noise. In *ACM SIGGRAPH 2006 Papers on - SIGGRAPH '06* (2006), Association for Computing Machinery (ACM). URL: http://dx.doi.org/10.1145/1179352.1141916, doi:10.1145/1179352.1141916. 3, 5, 8

[KDJO16] KAJI S., DEROUET-JOURDAN A., OCHIAI H.: Dappled tiling. In *Symposium on Mathematical Progress in Expressive Image Synthesis (MEIS2016)* (2016), vol. 69 of *MI Lecture Note Series*, Kyushu University, pp. 18–27. 2, 3, 5, 6, 8

[KNL*15] KASPAR A., NEUBERT B., LISCHINSKI D., PAULY M., KOPF J.: Self tuning texture optimization. *Computer Graphics Forum 34*, 2 (may 2015), 349–359. URL: http://dx.doi.org/10.1111/cgf.12565, doi:10.1111/cgf.12565. 2, 7, 8

[KP80] KLARNER D., POLLACK J.: Domino tilings of rectangles with fixed width. *Discrete Mathematics 32*, 1 (1980), 45 – 52. URL: http://www.sciencedirect.com/science/article/pii/0012365X80900989, doi:https://doi.org/10.1016/0012-365X(80)90098-9. 3

[KSE*03] KWATRA V., SCHÖDL A., ESSA I., TURK G., BOBICK A.: Graphcut textures: Image and video synthesis using graph cuts. In *ACM SIGGRAPH 2003 Papers on - SIGGRAPH '03* (2003), Association for Computing Machinery (ACM). URL: http://dx.doi.org/10.1145/1201775.882264, doi:10.1145/1201775.882264. 2

[LD05] LAGAE A., DUTRÉ P.: A procedural object distribution function. *ACM Transactions on Graphics 24*, 4 (October 2005), 1442–1461. URL: http://doi.acm.org/10.1145/1095878.1095888, doi:10.1145/1095878.1095888. 2, 3

[LDG01] LEGAKIS J., DORSEY J., GORTLER S.: Feature-based cellular texturing for architectural models. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 2001), SIGGRAPH '01, ACM, pp. 309–316. URL: http://doi.acm.org/10.1145/383259.383293, doi:10.1145/383259.383293. 2, 3

[LH06] LEFEBVRE S., HOPPE H.: Appearance-space texture synthesis. In *ACM SIGGRAPH 2006 Papers on - SIGGRAPH '06* (2006), Association for Computing Machinery (ACM). URL: http://dx.doi.org/10.1145/1179352.1141921, doi:10.1145/1179352.1141921. 2, 7

[LHVT17] LOI H., HURTUT T., VERGNE R., THOLLOT J.: Programmable 2d arrangements for element texture design. *ACM Trans. Graph. 36*, 4 (May 2017). URL: http://doi.acm.org/10.1145/3072959.2983617, doi:10.1145/3072959.2983617. 3

[LLC*10] LAGAE A., LEFEBVRE S., COOK R., DEROSE T., DRETTAKIS G., EBERT D., LEWIS J., PERLIN K., ZWICKER M.: A Survey of Procedural Noise Functions. *Computer Graphics Forum* (2010). doi:10.1111/j.1467-8659.2010.01827.x. 2

[LLDD09] LAGAE A., LEFEBVRE S., DRETTAKIS G., DUTRÉ P.: Procedural noise using sparse gabor convolution. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2009) 28*, 3 (July 2009), 54–64. doi:10.1145/1531326.1531360. 3

[LMM02] LODI A., MARTELLO S., MONACI M.: Two-dimensional packing problems: A survey. *European Journal of Operational Research 141*, 2 (2002), 241 – 252. URL: http://www.sciencedirect.com/science/article/pii/S0377221702001236, doi:https://doi.org/10.1016/S0377-2217(02)00123-6. 3

[Miy90] MIYATA K.: A method of generating stone wall patterns. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1990), SIGGRAPH '90, ACM, pp. 387–394. URL: http://doi.acm.org/10.1145/97879.97921, doi:10.1145/97879.97921. 3, 7

[Per85] PERLIN K.: An image synthesizer. *SIGGRAPH Comput. Graph. 19*, 3 (July 1985), 287–296. URL: http://doi.acm.org/10.1145/325165.325247, doi:10.1145/325165.325247. 3

[RDDM17] RAAD L., DAVY A., DESOLNEUX A., MOREL J.-M.: A survey of exemplar-based texture synthesis. *Annals of Mathematical Sciences and Applications* (2017), To appear. 2

[SD10] SCHLÖMER T., DEUSSEN O.: Semi-stochastic tilings for example-based texture synthesis. *Computer Graphics Forum 29*, 4 (aug 2010), 1431–1439. URL: http://dx.doi.org/10.1111/j.1467-8659.2010.01740.x, doi:10.1111/j.1467-8659.2010.01740.x. 3

[SP16] SANTONI C., PELLACINI F.: gtangle: A grammar for the procedural generation of tangle patterns. *ACM Trans. Graph. 35*, 6 (Nov. 2016), 182:1–182:11. URL: http://doi.acm.org/10.1145/2980179.2982417, doi:10.1145/2980179.2982417. 3

[SRVT14] SALVATI M., RUIZ VELASCO E., TAKAO K.: The brush shader: A step towards hand-painted style background in cg, 2014. 3, 5

[Sta97] STAM J.: *Aperiodic Texture Mapping*. Tech. rep., ERCIM, 1997. 3

[VSLD13] VANHOEY K., SAUVAGE B., LARUE F., DISCHLER J.-M.:

On-the-fly multi-scale infinite texturing from example. *ACM Transactions on Graphics 32*, 6 (nov 2013), 1–10. URL: http://dx.doi.org/10.1145/2508363.2508383, doi:10.1145/2508363.2508383. 2

[Wan61] WANG H.: Proving theorems by pattern recognition II. *Bell System Technical Journal 40* (1961), 1–42. 3

[Wei04] WEI L.-Y.: Tile-based texture mapping on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware - HWWS '04* (2004), Association for Computing Machinery (ACM). URL: http://dx.doi.org/10.1145/1058129.1058138, doi:10.1145/1058129.1058138. 3

[Wor96] WORLEY S.: A cellular texture basis function. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1996), SIGGRAPH '96, ACM, pp. 291–294. URL: http://doi.acm.org/10.1145/237170.237267, doi:10.1145/237170.237267. 3, 5

[YBY*13] YEH Y.-T., BREEDEN K., YANG L., FISHER M., HANRAHAN P.: Synthesis of tiled patterns using factor graphs. *ACM Transactions on Graphics 32*, 1 (jan 2013), 1–13. URL: http://dx.doi.org/10.1145/2421636.2421639, doi:10.1145/2421636.2421639. 2

A. Base case solver

We separate the 16 cases for the equalities $H_{i,j+2} = H_{i,j}$ $H_{i+1,j} = H_{i+1,j+2}$ $V_{i,j+1} = V_{i+2,j+1}$ $V_{i,j} = V_{i+2,j}$. The solutions are given as 4 values for respectively $V_{i+1,j+1}$, $H_{i,j+1}$, $V_{i+1,j}$ and $H_{i+1,j+1}$.

1. solver 0000
   - $V_{i,j+1}, H_{i,j}, V_{i+2,j}, H_{i+1,j+2}$
   - $V_{i+2,j+1}, H_{i,j+2}, V_{i,j}, H_{i+1,j}$
2. solver 0001
   - $v_{d2}(i+1,j+1,V_{i,j+1},V_{i+2,j+1}), H_{i,j+2}, V_{i,j}, H_{i+1,j+2}$
   - $V_{i+2,j+1}, H_{i,j+2}, V_{i,j}, h_{d2}(i+1,j+1,H_{i+1,j},H_{i+1,j+2})$
   - $V_{i,j+1}, h_{d2}(i,j+1,H_{i,j+2},H_{i,j}), V_{i,j}, H_{i+1,j+2}$
3. solver 0010
   - $V_{i,j+1}, H_{i,j}, v_{d2}(i+1,j,V_{i,j},V_{i+2,j}), H_{i+1,j}$
   - $V_{i,j+1}, H_{i,j}, V_{i+2,j}, h_{d2}(i+1,j+1,H_{i+1,j},H_{i+1,j+2})$
   - $V_{i,j+1}, h_{d2}(i,j+1,H_{i,j+2},H_{i,j}), V_{i,j}, H_{i+1,j}$
4. solver 0011
   - $V_{i,j+1}, h_{d2}(i,j+1,H_{i,j+2},H_{i,j}), V_{i,j}, h_{d2}(i+1,j+1,H_{i+1,j},H_{i+1,j+2})$
   - $V_{i,j+1}, H_{i,j}, v_{d2}(i+1,j,V_{i,j},V_{i+2,j}), H_{i+1,j}$
   - $v_{d2}(i+1,j+1,V_{i,j+1},V_{i+2,j+1}), H_{i,j+2}, V_{i,j}, H_{i+1,j+2}$
5. solver 0100
   - $V_{i,j+1}, h_{d2}(i,j+1,H_{i,j+2},H_{i,j}), V_{i,j}, H_{i+1,j}$
   - $v_{d2}(i+1,j+1,V_{i,j+1},V_{i+2,j+1}), H_{i,j+2}, V_{i,j}, H_{i+1,j}$
   - $V_{i,j+1}, H_{i,j}, v_{d2}(i+1,j,V_{i,j},V_{i+2,j}), H_{i+1,j}$
6. solver 0101
   - $V_{i,j+1}, H_{i,j}, v_{d2}(i+1,j,V_{i,j},V_{i+2,j}), H_{i+1,j}$
   - $V_{i+2,j+1}, H_{i,j+2}, V_{i,j}, h_{d2}(i+1,j+1,H_{i+1,j},H_{i+1,j+2})$
7. solver 0110
   - $v_{d2}(i+1,j+1,V_{i,j+1},V_{i+2,j+1}), H_{i,j+2}, V_{i,j}, H_{i+1,j}$
   - $V_{i+2,j+1}, H_{i,j}, V_{i+2,j}, h_{d2}(i+1,j+1,H_{i+1,j},H_{i+1,j+2})$
8. solver 0111
   - $V_{i,j+1}, h_{d2}(i,j+1,H_{i,j+2},H_{i,j}), V_{i,j}, h_{d2}(i+1,j+1,H_{i+1,j},H_{i+1,j+2})$
9. solver 1000
   - $V_{i+2,j+1}, H_{i,j+2}, V_{i+2,j}, h_{d2}(i+1,j+1,H_{i+1,j},H_{i+1,j+2})$
   - $v_{d2}(i+1,j+1,V_{i,j+1},V_{i+2,j+1}), H_{i,j+2}, V_{i+2,j}, H_{i+1,j+2}$
   - $V_{i+2,j+1}, H_{i,j+2}, v_{d2}(i+1,j,V_{i,j},V_{i+2,j}), H_{i+1,j}$
10. solver 1001
    - $V_{i+2,j+1}, H_{i,j+2}, v_{d2}(i+1,j,V_{i,j},V_{i+2,j}), H_{i+1,j}$
    - $V_{i,j+1}, h_{d2}(i,j+1,H_{i,j+2},H_{i,j}), V_{i,j}, H_{i+1,j+2}$
11. solver 1010
    - $v_{d2}(i+1,j+1,V_{i,j+1},V_{i+2,j+1}), H_{i,j+2}, V_{i+2,j}, H_{i+1,j+2}$
    - $V_{i+2,j+1}, h_{d2}(i,j+1,H_{i,j+2},H_{i,j}), V_{i,j}, H_{i+1,j}$
12. solver 1011
    - $V_{i,j+1}, h_{d2}(i,j+1,H_{i,j+2},H_{i,j}), V_{i,j}, h_{d2}(i+1,j+1,H_{i+1,j},H_{i+1,j+2})$
13. solver 1100
    - $v_{d2}(i+1,j+1,V_{i,j+1},V_{i+2,j+1}), H_{i,j+2}, v_{d2}(i+1,j,V_{i,j},V_{i+2,j}), H_{i+1,j}$
    - $V_{i,j+1}, h_{d2}(i,j+1,H_{i,j+2},H_{i,j}), V_{i,j}, H_{i+1,j}$
    - $V_{i+2,j+1}, H_{i,j+2}, V_{i+2,j}, h_{d2}(i+1,j+1,H_{i+1,j},H_{i+1,j+2})$
14. solver 1101
    - $v_{d2}(i+1,j+1,V_{i,j+1},V_{i+2,j+1}), H_{i,j+2}, v_{d2}(i+1,j,V_{i,j},V_{i+2,j}), H_{i+1,j}$
15. solver 1110
    - $v_{d2}(i+1,j+1,V_{i,j+1},V_{i+2,j+1}), H_{i,j+2}, v_{d2}(i+1,j,V_{i,j},V_{i+2,j}), H_{i+1,j}$
16. solver 1111
    - $v_{d2}(i+1,j+1,V_{i,j+1},V_{i+2,j+1}), H_{i,j+2}, v_{d2}(i+1,j,V_{i,j},V_{i+2,j}), H_{i+1,j}$
    - $V_{i,j+1}, h_{d2}(i,j+1,H_{i,j+2},H_{i,j}), V_{i,j}, h_{d2}(i+1,j+1,H_{i+1,j},H_{i+1,j+2})$

with v_{d1} and v_{d2} (resp. h_{d1} and h_{d2}) to compute random colors different from its input:

$$v_{d1}(i,j,c) = \begin{cases} m & \text{if } m < c \\ m+1 & \text{if } m \geq c \end{cases}$$
$$\text{where } m = \mathcal{V}(i,j)\%(n_c-1)$$

$$v_{d2}(i,j,c_1,c_2) = \begin{cases} m' & \text{if } m' < \min(c_1,c_2) \\ m'+1 & \text{if } \min(c_1,c_2) \leq m' < \max(c_1,c_2)-1 \\ m'+2 & \text{if } m' \geq \max(c_1,c_2)-1 \end{cases}$$
$$\text{where } m' = \mathcal{V}(i,j)\%(n_c-2).$$