FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Procedural Generation of Brick and Stone Textures

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Georg Sperl
Matrikelnummer 1025854

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Projektass. Dipl.-Mediessywiss. Dr.techn. Przemyslaw Musialski

Wien, 19.09.2013

(Unterschrift Verfasser) (Unterschrift Betreuer)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Procedural Generation of Brick and Stone Textures

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Georg Sperl
Registration Number 1025854

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Projektass. Dipl.-Mediessywiss. Dr.techn. Przemyslaw Musialski

Vienna, 19.09.2013

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Georg Sperl
Borromäumstraße 34, 2362 Biedermannsdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Kurzfassung

Texturierung spielt in der Computergraphik eine sehr wichtige Rolle. Texturen können zum Beispiel flachen Objekten Farbe, Tiefe und Unebenheiten geben. Allerdings ist es oft schwer Texturen für eine Geometrie oder eine Oberfläche zu finden oder zu erstellen. Texturen manuell zu erzeugen kann sich als extrem schwer herausstellen, da Probleme mit Säumen und Verzerrung auftreten können, welche die Textur unrealistisch erscheinen lassen. Um diesen Problemen entgegenzuwirken wurden viele Arten der Textursynthese entwickelt, unter welchen prozedurale Texturen eine wichtige Position inne haben.

Die Arten von Textursynthese variieren von Beispiel-basierter Synthese zu prozeduralem Rauschen, zellulären Texturen und mehr, wodurch man eine große Zahl an unterschiedlichen Ergebnissen produzieren kann.

Das Ziel dieses Dokuments ist es, den Leser mit verwandten Themen der Textursynthese vertraut zu machen und eine Methode vorzustellen, um Texturen von zellulären Ziegelsteinmustern zu erzeugen, welche Ziegelverbände repräsentieren, die in realer Architektur verwendet werden, wobei auf korrekte Behandlung von Wand- und Wandelementrändern Acht genommen wird. Die Methode benutzt eine Zahl an Parametern, um den Syntheseprozess zu steuern. Des Weiteren wird eine Methode vorgestellt, um zufällig generierte Steinmuster zu erzeugen.

Abstract

Texturing plays an extremely important role in computer graphics. They may, for example, give flat objects colour patterns and depth or roughness. However, it is often difficult to find or create fitting textures for a geometry or surface. Creating textures by hand can prove extremely difficult as problems with seams or distortion may occur, which make the texture look unrealistic. In order to counter these problems, many different kinds of texture synthesis have been developed, among which procedural textures hold an important position.

The types of texture synthesis methods vary from example-based synthesis to procedural noise, cellular textures and more, which can produce a wide array of different results.

The aim of this paper is to introduce the reader into related topics of texture synthesis and to introduce a method for generating cellular brick pattern textures, which represent brick bonds that are used in real architecture, taking correct handling of wall and window borders into consideration. The method uses few parameters to control the synthesis process. In addition, a method for generating random stone patterns is introduced.

Contents

CHAPTER 1

Introduction

1.1 Motivation

There is a tremendous amount of papers on procedural noise, but there are only a handful of papers on cellular pattern textures, especially for generating realistic stone or brick patterns.

A large range of applications exists for realistic brick and stone textures. Such textures can be used to augment architectural models in the gaming and filming industry, urban planning and reconstruction applications, training and simulation applications (e.g. disaster control, flying simulations, etc.) and procedural modelling by creating more believable visuals of architecture intended to look realistic.

Trying to fill the gap, we introduce a new method for procedurally generating realistic brick textures, which are based on real brick bonds, such as a "*Scottish bond*" or a "*Flemish bond*".

1.2 Problem statement

The generation of brick and stone patterns is an important part of the modelling pipeline. There are a number of tools for this task (e.g. Blender). However, they usually only consider texture seams for infinite repetition, but the created patterns are very limited by their infinite repetition. They also do not consider wall elements, such as windows, where brick endings have to be taken into account. They rather just do not render the texture, instead of adjusting it.

There are solutions for generating stone wall patterns representing stones with a Voronoi diagram of joints, but they are more useful for synthesizing general stone patterns rather than brick patterns. Generally, the existing methods do not focus on brick bonds but rather on giving an abstract possibility for generating patterns.

Therefore our aim was to be able to generate textures of brick bonds with few user-controllable parameters, which consider wall elements.

Our approach was to generate the texture with cells, where each cell represents a brick. These brick cells are stacked together row by row according to the pattern of a brick bond. Brick

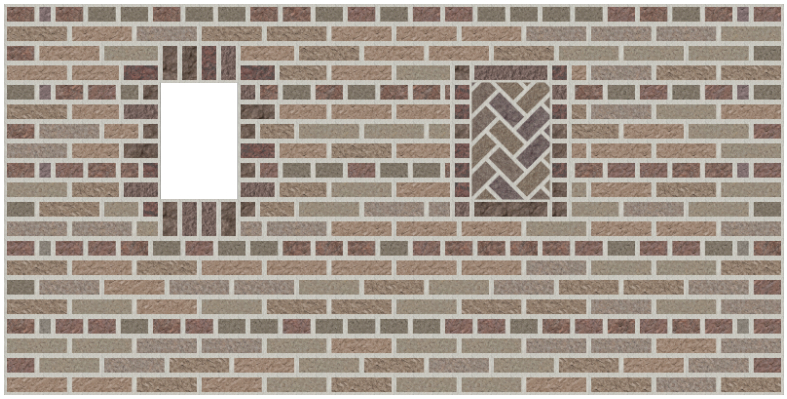


Figure 1.1: A brick wall pattern generated with our method

bonds usually have different types of row patterns, which in turn can be divided into different parts, such as a beginning, a repeating middle and an ending. To handle wall elements, we fit the bricks against them, i.e. cut them off to avoid overlapping.

The user may specify parameters, such as the width and height of a brick, the dimensions of the wall, where wall elements are located, etc. We assume that a user wants to generate a texture for a specific wall model, where the wall dimensions and the dimensions and locations of wall elements are fixed, unchangeable parameters. Since each brick bond has constraints to brick proportions, in order to have the same width for each differently patterned row some parameters may have to be modified. We approach this by trying to to fit a generated solution as closely as possible to the user's parameters. An example pattern generated with our method is shown in figure 1.1.

1.3 Thesis Structure

The structure of this thesis is as follows:

Chapter 1: Introduction
This chapter introduces the problem, motivation and the aim of the work, its general methodological approach and structure.

Chapter 2: Related Work
The state of the art of related topics, such as texture synthesis, is discussed.

Chapter 3: Methodology
The methodology of our solution is explained in detail including a general overview as well as detailed components.

Chapter 4: Implementation
The implementation of our method is discussed in more detail, such as the used frame-

works and data structures, certain class hierarchies and implementation dependent problems.

Chapter 5: Results

Our method's results and limitations are displayed and discussed with examples of parameter effects and treatment of special cases.

Chapter 6: Conclusion

The thesis is summarized, conclusions are drawn and possible future work is discussed.

CHAPTER 2

Related Work

2.1 Textures

Textures play a very important role in computer graphics. They provide the possibility to easily colourize or otherwise influence geometry via *texture mapping*. The term texture in computer graphics differs from the definition of real surface texture, where it means "*local deviations of a surface from a perfectly flat plane*", whereas in computer graphics texture usually has the meaning of a digital image, which is mapped onto surfaces.

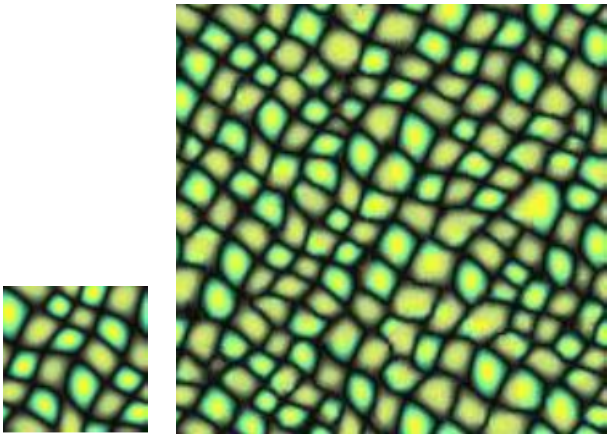
There is a wide array of different types of textures and applications, for example *bump mapping*. One can classify textures into several groups, such as *regular*, *irregular* and *stochastic*, depending on how they are created and what patterns they represent.

In the majority of cases image textures are used, which are projected onto surfaces via texture mapping. Such image textures have the advantages of being pre-computed, which makes them render faster, and there are many texture libraries with vast amounts of existing content. However, creating such image textures can be very hard. They can be created by taking and altering photographies and may therefore be very realistic or made completely by hand by *texture artists*.

Often it is too difficult or time-consuming to create textures like this, and therefore a variety of methods for synthesizing textures have been developed, which provide many additional benefits to facilitate creation, such as control over *texture seams*, distortion, texture size and much more. In the following sections we will discuss methods of texture synthesis, such as example-based texture synthesis and procedural textures. The textures generated by our method would be classified as cellular textures 2.5.

2.2 Example-based Texture Synthesis

Problems with creating or using image textures are, for example, unaligned seams, distortion and illumination. Since it can be very difficult to create specific textures procedurally, there



(a) input (b) output

Figure 2.1: Example-based texture synthesis (Figures from [1])

are many methods for generating textures by using example image textures as input. *Example-based texture synthesis* tries to combine the advantages of image textures with the advantages of procedural textures. The output texture may have arbitrary size and, depending on the method, other user-controllable benefits as well. An example is shown in figure 2.1.

Numerous different methods have been developed, for example Lagae *et al.* introduced a method to create textures with multi-resolution *noise bands*, which are parametrized to match the example texture as much as possible [2].

Wei *et al.* provided an extensive overview of the state of the art of example-based texture synthesis [1]. Most of the methods discussed rely on *Markov Random Fields*, which means that each pixel is characterized by its neighbourhood and that the synthesizable textures have to be *local* and *stationary*. Local means that a pixel is predictable by its neighbours, and stationary means that any windowed portion of the image should appear similar. In the following we will briefly discuss some of these methods.

Pixel-Based Synthesis

A random initial output is created and improved step by step. The neighbourhood of each pixel is compared to the input texture, and a fitting value is chosen. This method can be further improved with similarity sets of pixels.

Patch-Based Synthesis

It is faster to create the output by stitching together patches, rather than pixel by pixel. The neighbourhood of a patch is defined as a thin band of pixels around it. Patches may have different shapes and might therefore overlap. This has to be considered by merging, blending, warping or cutting the patches correctly.

Texture Optimization

A *quadratic energy function* is computed, which represents the mismatches of neighbourhoods between input and output. A texture is optimal when this function is minimal and thus the goal is to minimize it.

Surface Texture Synthesis

On curved surfaces problems with distortion and seams occur regularly, when using simple image textures, therefore it is better to synthesize the texture directly for those surfaces.

In the first step an *orientation field* is computed on the surface, which stores texture orientation similar to how rows and columns of pixels in image textures are oriented horizontally and vertically. This process is guided by user constraints, which are then interpolated to generate a seamless orientation field. The texture is then synthesized considering the orientation field. This can be pixel-based or patch-based.

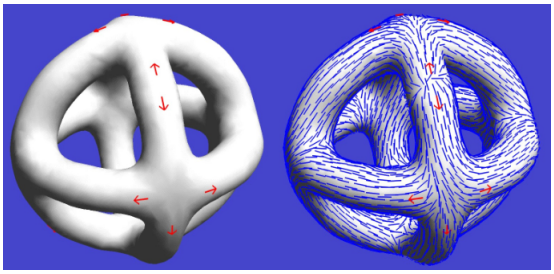


Figure 2.2: Orientation field on the surface of a curved mesh (Figure from [1])

Other Types

All of the methods listed above store the synthesized textures for later use, but this might be a waste of storage and synthesis time, since in most cases one does not need the whole texture but only the directly visible, rendered areas. Therefore *run-time synthesis* of textures can be extremely useful. Most of the methods above can be implemented at run-time as well but often much more complicated and less efficient, since they require a lot of computation and look-ups in the example texture.

To name further existing methods of example-based texture synthesis, there is *Inverse Texture Synthesis*, which basically is compression, *Image Completion*, which is repairing textures that have holes or other discontinuities, *Dynamic Texture Synthesis*, which is synthesis of time-dependent textures such as animated waves, *Resolution Enhancement*, *Solid Texture Synthesis* and many more.

2.3 Procedural Textures

The term "*procedural*" in computer science means that an entity is described by code, rather than by data. In procedural textures this means that the texture's values (e.g. colour) are generated by algorithms, such as *procedural noise*, and are not simply read from an existing file.

Procedural textures have many advantages. They have compact storage requirements, no fixed resolution or size and no undesired seams, repetitions or distortions. As they are created algorithmically with a set of modifiable parameters, they represent whole classes of textures, as opposed to a single image texture. Changing a parameter changes the texture, but generally it still looks similar. Thus the set of creatable textures defines what kind of texture class a method produces. Our method, for example, creates a variety of brick and stone textures with adjustable brick size and so on. Further advantages are that it is easy to create *solid textures*, they allow for *high quality anti-aliasing* and *texture morphing*, and they are often realizable in GPU shaders.

In the following we will discuss two large groups of procedural textures, *procedural noise* and *cellular textures*.

2.4 Procedural Noise

Noise in computer graphics is generally defined as a random, unstructured pattern [3]. Noise textures are very useful to simulate details and patterns, where the true structure of an object's surface would be too complex or too small to model by hand, for example for bump maps of rough surfaces. Generating textures with procedural noise is usually more efficient and simple than other methods. The texture image is created by evaluating a noise function for each pixel, which can be done as a pre-process or at run-time. The noise function may be controllable with parameters, and, depending on the specific method, its evaluation can be implemented very efficiently in GPU shaders.

Many different types of procedural noise functions exist, such as *lattice gradient noises* and *sparse convolution noises*.

There is a broad spectrum of papers on procedural noise due to their vast number of types and applications. To list a few, Cook *et al.* introduced the *wavelet noise* [4], Frisvad *et al.* presented a fast GPU implementation of *high-quality sparse convolution noise* [5], Lagae *et al.* introduced noise based on *Sparse Gabor Convolution* [6] and Gilet *et al.* introduced *multiple kernel noise*, which further increases the variety of synthesizable textures [7].

Lagae *et al.* summarized the state of the art of procedural noise function very well [3]. In the following we will introduce some of the most important types of procedural noise listed in their paper.

Lattice gradient noises

Lattice gradient noises are generated via interpolation or convolution of random values or gradients defined on a lattice.

The most prominent example is *Perlin noise*, which was introduced by Perlin in 1985 [8]. It is one of the earliest noise functions used in computer graphics and an essential building block

for many others. In Perlin noise the lattice is filled with pseudo-randomly generated gradients by hashing the coordinates via a pseudo-random permutation and computing gradients with the results. The gradients are then interpolated with spline interpolation.

(a) (b)

Figure 2.3: (a) Perlin noise (Figure from [3]) and (b) a vase with a Perlin noise texture (Figure from [8])

Explicit noises

Explicit noises store the noise, which is generated as a pre-process as compared to direct evaluation of other noise types.

One very important example is the *wavelet noise*, introduced by Cook and DeRose in 2005 [4]. Noise coefficients, which are used later at runtime for evaluating the noise function at a given position, are pre-computed via iteratively subtracting a seed image with a down- and then up-sampled version of the same image. Only the band-limited part of the image remains because the part that is still representable after down-sampling is subtracted each time.

Another important explicit noise is *anisotropic noise*, where the frequency domain is tiled into oriented sub-bands, which are then stored. The noise values are computed during rendering as a weighted sum of these sub-bands. Anisotropic noise allows for compensation of parametric distortion and anisotropic analytic filtering.

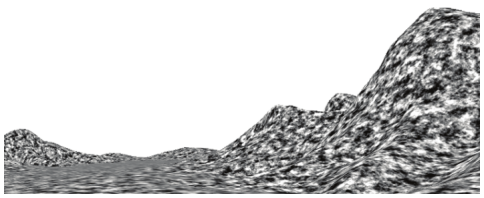


Figure 2.4: Anisotropic noise (Figure from [3])

Sparse convolution noise

Sparse convolution noise uses randomly positioned and weighted kernels to generate the noise. The noise is evaluated by considering only the necessary kernels overlapping at a given point. A kernel is convolved with a Poisson process noise, which is not defined at every pixel and therefore "*sparse*".

Two other noises extend on the general idea of sparse convolution noise, which are *spot noise* and *Gabor noise*. Spot noise is often used for visualizing scalar or vector fields on surfaces. Gabor noise uses a special Gabor kernel, which is the product of a circular Gaussian kernel and a 2D cosine.

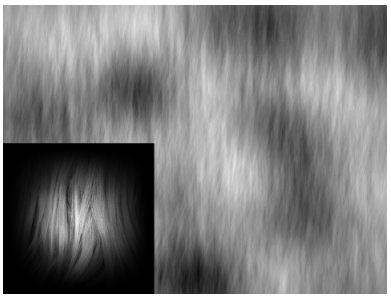


Figure 2.5: Sparse convolution noise created with the depicted kernel (left) (Figure from [3])

2.5 Cellular Textures

Procedural noise textures are already very powerful tools of texture synthesis, but they are limited to their random noise domain. *Cellular textures* are procedural textures that are, as the name implies, based on cells. These textures vary strongly in their methods of creation, distribution and rendering and can therefore produce all kinds of pattern-based textures. Cells usually interact with each other and may be arbitrarily correlated. One can, for example, represent scales or bricks with cells and, if one cell grows, its neighbours might need to shrink. Most cellular texture do not have noise functions as their basis, but they can be complimented by them.

In the following we will try to list some of the most important types of cellular texture generators.

Texture Basis Functions

Worley proposed texture basis functions, which are based on scattering *feature points* in 3D space and which do not need pre-computation [9]. Feature points are randomly distributed in $\mathbb{R}^3$. A function F_1 is now defined as the distance of any location $\mathbf{x}$ to the closest feature point. The function created is continuous and smooth and partitions the space into cellular regions like a *Voronoi diagram*. One can create an arbitrary amount of functions F_i, which represent the distance of $\mathbf{x}$ to the ith closest feature point. These functions can then be used effectively and in many ways to generate a texture, for example by simply mapping F_1 to colour and/or height,

a smoothed visualization of the Voronoi diagram is created as shown in figure 2.6. Using and combining other base functions F_i creates a variety of visually interesting patterns.

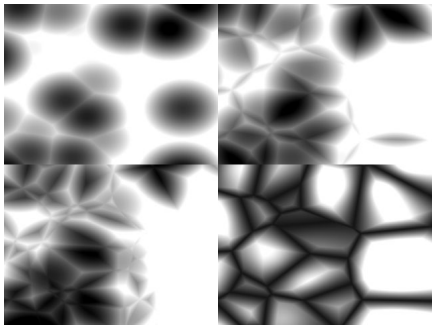


Figure 2.6: Cellular texture basis function (Figure from [9])

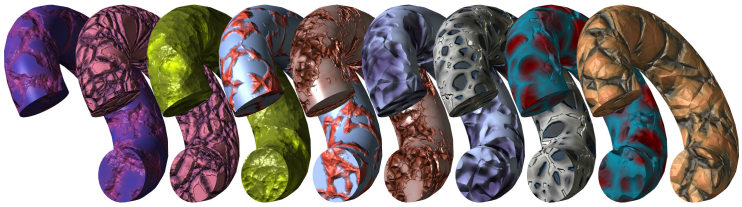


Figure 2.7: Textures created with Worley's basis function (Figure from [9])

Chan *et al.* adapted Worley's algorithm to a single-pass pixel shader implementation [10].

Our method does not use a basis function to generate the texture but instead explicitly produces the stone cells.

Cellular stone textures

A very important and often cited paper in procedural textures is Miyata's "*A Method of Generating Stone Wall Patterns*" from 1990 [11]. It describes a method of creating a variety of stone patterns by randomizing stone joints with a small set of parameters. A wall is represented by joints, which consist of nodes and links that are restricted to upper, lower, left and right. The joints are initially placed like rectangular stones with an average stone size and a size variance. The joints are then randomly deformed and subdivided fractally to create curved joints. The stone faces are also subdivided fractally to create rough surfaces. With a scan line algorithm the computed data is then stored into a bump map that can be used for rendering the stone wall. An example is shown in figure 2.8.

This method differs from our method as our stone cells are not represented by their connected joints but rather by individual cells, which offers a higher degree of freedom in placement and modification but also necessitates additional effort for correlation.

Figure 2.8: Stone wall texture (Figure from [11])

Another important method on the subject of cellular stone patterns was presented by Legakis *et al.* [12]. They presented a cellular texturing method for 3D stone walls, where the stones at the borders have correctly aligned seams and are created with chainable *pattern generators*. The input is a two-level geometry, where the first level is the basic mesh and the second level are semantic annotations like, for example, "*edge*" or "*corner*".

The chainable patterns constitute *pattern trees*, in which the individual pattern generators may pass on features to their children generators based on criteria like type, label or a specific analysis. For example, a pattern generator might pass corners to one child and edges to another, and the children might then draw these features in different colours.

To store which parts of a mesh have already been occupied, so-called *occupancy maps* are used. Occupancy maps are bit mask data structures. Legakis *et al.* proposed two dimensional arrays for faces and one dimensional arrays for edges. To correctly handle holes inside a face, the face occupancy maps are initialized as fully occupied, and then the face is rasterized into it as empty, so that holes inside faces are left as occupied.

A bi-directional mapping between the parameter space of pattern generators and the world space of the mesh is needed for rasterizing features into occupancy maps and to place cells generated in parameter space into world space for rendering.

Legakis *et al.* simplify the process of texture generation by computing the most constrained parts first. They propose the ordering of computing first corners, then edges and lastly faces. Mortar between the stones is generated by rendering a shrunken base mesh.

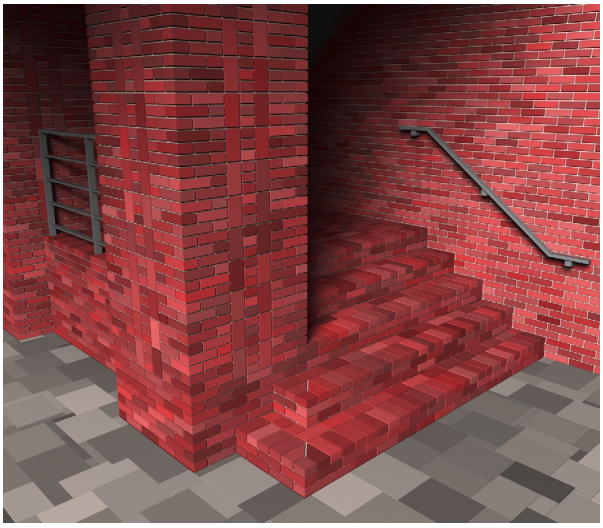


Figure 2.9: Brick pattern texture (Figure from [12])

Their method differs from ours as we do not have chainable pattern generators but instead standalone pattern generators that simulate brick bonds used in the real world. We also only use a type of occupancy map for our random stone pattern generator, but it contains indices of the generated stones as compared to only storing occupied and empty space. Another significant difference is that their method allows for 3D meshes with aligned seams, whereas our method is yet restricted to 2D. Furthermore their method provides abstract possibility for generating patterns whereas our method is explicitly focused on realistic brick bonds.

Others

Sakurai *et al.* described a method for procedural leather texture generation with a focus on structural leather elements and based on Voronoi diagrams similar to the texture generation methods of Miyata and Worley [13]. They explicitly try to simulate real leather by forming pores, wrinkles and more by modifying the nodes of the diagram.

Figure 2.10: Examples of leather textures with different parameters (Figures from [13])

A method for creating 3D cellular textures on surfaces of models was described by Fleischer *et al.* [14]. Parametrized cell particles are distributed on a surface and may be correlated to their

neighbours. This method is especially practical for creating 3D scales or fur on model surfaces. An example correlation of cell particles would be that the individual hairs of fur are locally oriented in similar directions. The particles are converted into geometry and then rendered.

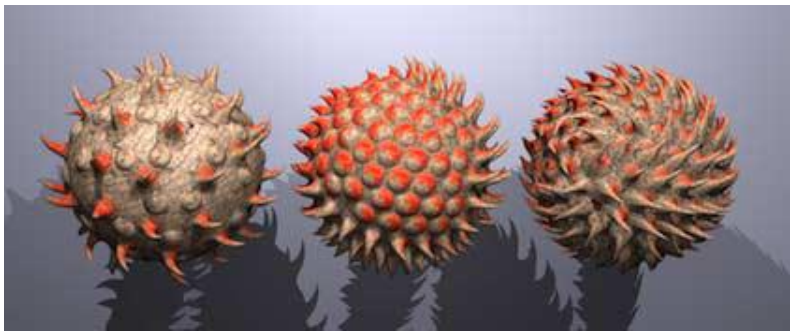


Figure 2.11: 3D cellular texture on sphere surfaces (Figure from [14])

Neyret *et al.* introduced a method for pattern-based texturing with triangular patches [15], and Lefebvre *et al.* proposed an algorithm for combining patterns in cells into textures according to user-defined controls [16]. Both methods differ from ours as they map tiles or patches onto a 3D surface and their cells represent seamlessly connectible arbitrary parts of a texture, whereas we create cellular textures on a plane representing closed brick bonds.

CHAPTER 3

Methodology

We based our method on the idea of generating the texture as a cell pattern, where each cell represents a brick or stone and the pattern is built similar to how it would be done by hand, i.e. generally brick by brick and row by row.

Our method takes a number of parameters as input, such as wall and brick dimensions. In order to fit the specific patterns into a wall of fixed dimensions, the brick dimensions have to be modified, which is later explained in greater detail. From the modified set of parameters the cells of the brick bond or stone pattern are created. The cells are then converted into framework dependent drawables to be rendered into the output texture.

We will be referencing the following brick positions in this paper.

- Stretcher: *A brick laid on its long side with the narrow side exposed*
- Header: *A brick with its shortest side exposed*
- Queen-closer: *A brick cut in half down its length, i.e. a half header*

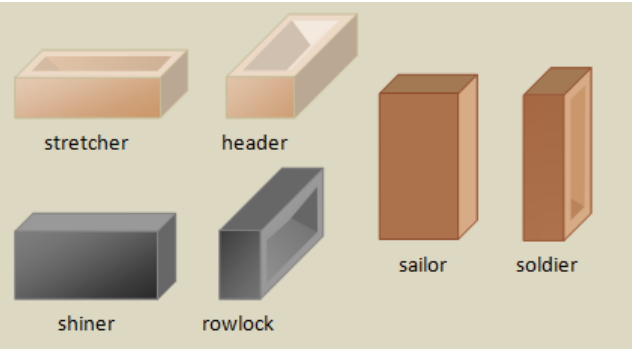


Figure 3.1: Various brick positions [17]

3.1 Application Flow

We will discuss the sub-processes in greater detail afterwards, but the general process of our application is as follows. After initialization, we take the the input parameters, from the graphical user interface or a XML-file. The parameters are then validated with simple checks, i.e. if the specified brick dimensions are greater than zero or if the location of a wall element is inside the bounds of the wall, but we generally assume that the user makes sensible parameter choices.

Afterwards, depending on the parameters, such as pattern type, element type and border type, the necessary generators are initialized. The first step is generating the borders of wall elements and its row limits. Then the actual pattern for the wall is created.

The generated pattern data, which includes the bricks of the wall pattern and the bricks of wall element borders, is then converted into drawable entities and finally rendered. In this step we also convert and draw non-cell-based objects like, for example, the fill of our Herringbone-bond-Element. We also provide the functionality for saving the rendered image to a PNG-file.

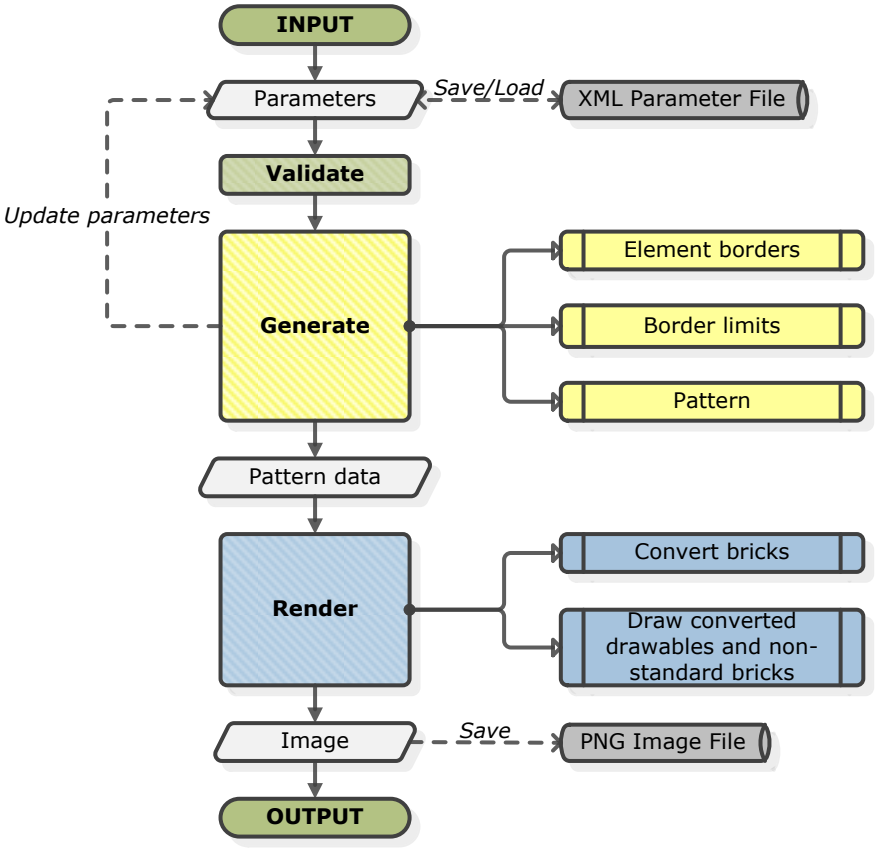


Figure 3.2: Flow chart depicting the application flow

3.2 Pattern Parameters

We use the following list of parameters to generate our patterns.

- Plane dimensions: *width, height*
- Colour theme: *if the brick cells are to be rendered with colours or bitmap textures*
- Variation seed: *seed used in random number generation for varying brick colours*
- Mortar width
- Elements
  - Type
  - Location: *x, y*
  - Dimensions: *width, height*
- Element groups
  - Type
  - Location: *x, y*
  - Dimensions: *width, height*
  - Distance: *the distances between consecutive elements*
  - Amount: *the amount of elements per row or column*
- Brick pattern generators
  - Type
  - Brick dimensions: *width, height, depth*
  - Frequency: *used in modifying the Scottish bond pattern*
- Stone pattern generators
  - Seed: *for random number generation*
  - Minimum stone size

As we already explained, to be able to generate a brick bond pattern, sometimes parameters have to be modified in order to create a realistic solution. We assumed that the plane dimensions and locations and dimensions of the wall elements are unmodifiable parameters as one possible application of our method could be generating a pattern for an existing wall mesh of fixed size with fixed wall elements. The parameters that are variable in our method are the brick and stone size.

3.3 Pattern Data

The pattern data is the transient data created from the generators, which is necessary for rendering or controlling the further process. It includes the brick cells of the wall pattern and of each element's borders as well as the row limits of a wall element. The row limits indicate where the wall element (including its borders) start and end is located in each row. These limits are stored in a data structure and used in fitting the bricks of the main pattern and other elements to another elements' borders.

3.4 Generation

The pattern data is initialized empty and together with the pattern parameters given to a specific pattern generator, depending on the choice of pattern type. We devised a method for creating brick bond patterns and stone patterns. Both types have in common that the individual stones or bricks are represented by cells. We also implicitly include mortar between the cells, by reducing the brick or stone size and rendering the mortar around the cell in the remaining space.

Another shared concept is how we recalculate the row height, i.e. brick or stone height. Since we want that wall elements start and end exactly on top or bottom of a row respectively, we have to calculate the row height in a way that the rows below and on top of every element fit perfectly. We do this by first finding all enclosed areas between wall and element borders. A picture of such enclosed areas is depicted in figure 3.3. After this, we find the greatest common divisor of the area. Then we try to fit the users preferred brick height into this area because, if it fits, it will naturally fit into all enclosed spaces and thereby create brick rows that fit perfectly and will not have to be clipped. If the height is greater than the greatest common divisor area, the areas size is returned as modified height. Else we compute the number of times the height would fit into the area and then divide the area by that number, giving us an only slightly altered, rounded up height from the preferred one. If the height already fits perfectly, this process will return the same height.

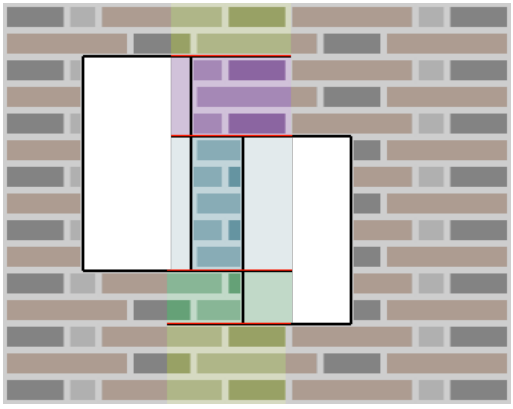
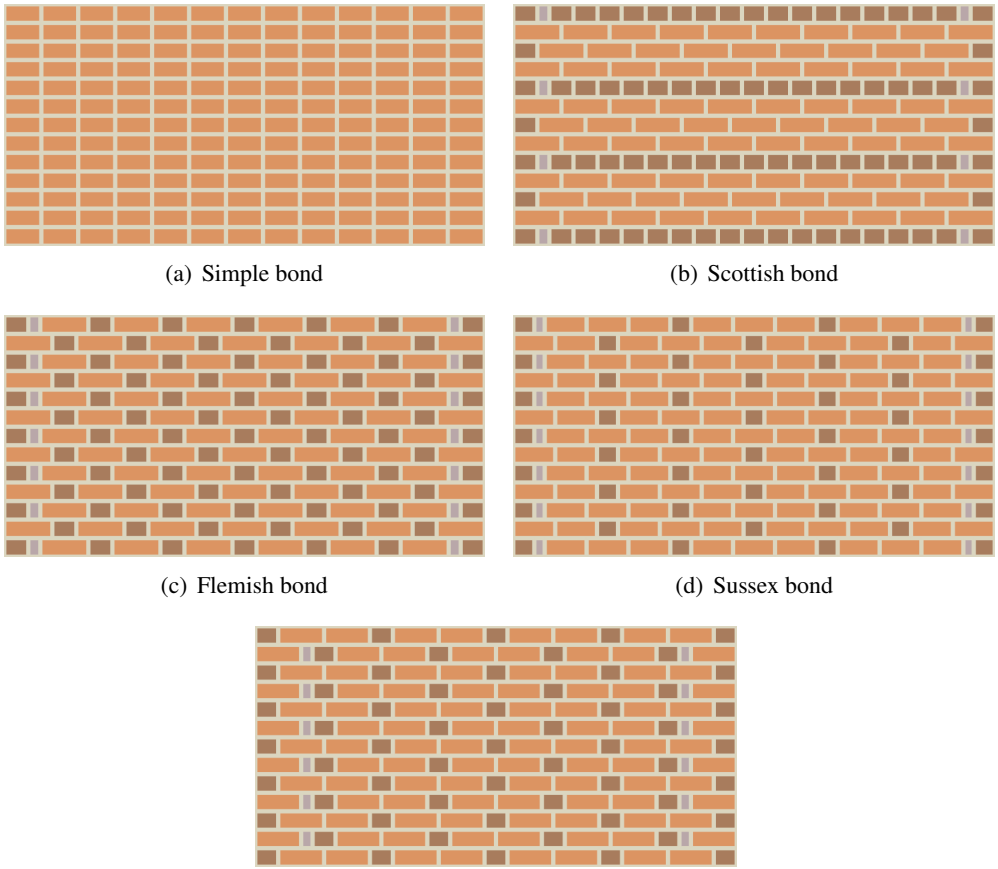


Figure 3.3: The fixed parts of a wall between borders and element bounds are marked in colour

3.5 Brick Pattern Generators

The types of brick patterns we implemented are a simple brick pattern, where header bricks are stacked exactly on top of each other, a *Scottish bond*, *Flemish bond*, *Sussex bond* and a *Monk bond* pattern, all of which are depicted in figure 5.3.



(a) Simple bond

(b) Scottish bond

(c) Flemish bond

(d) Sussex bond

(e) Monk bond

Figure 3.4: Different brick bonds patterns

The first step is to fit the brick height to the wall's height, as explained above. Afterwards, the rest of the brick dimensions are fitted to the width of the wall. One cannot use the same method for this, as there different types of rows, each of which may have repeating and non-repeating parts.

Therefore, we analysed the rows of the patterns and created simple regular expression patterns for them. There may be static parts at the beginning and ending (usually mirrored at each side) but in the middle there is always a repeating pattern. By comparing different rows of the same pattern, we could devise formulas for each rows for the amount of bricks per type in one variable n, which is the amount of repetitions of the row's repeating part. We then try to compute

n by adding the minimum amounts of bricks and trying to add the repeating pattern as often as possible and counting the repetitions. An example of a thus derived formula is $2 \cdot H + (n+1) \cdot S$, where H is the width of a header, i.e. a brick's depth, and S the width of a stretcher. Since we want to compute H and S so that the row equals exactly the width of the wall, we have (in this case) two variables because we already know n. This constitutes a system of linear equations and, since our chosen patterns are not unrealistically complex, the systems are solvable. The specific equations for our implemented patterns will be explained in more detail in the following subsections.

Subsequently, the wall elements' borders and limits are generated, which we will discuss later in section 3.7.

Then the individual bricks are created according to the pattern. Each brick is fitted against all borders of wall elements with the help of their limits. We assume that the patterns are made up of rows, and therefore we can store element limits used in brick fitting as row limits with a starting and ending point. If the row of a brick has such stored limits, it is checked whether the brick is completely or partially inside or outside, and the brick is accordingly modified, i.e. ignored, shortened or split.

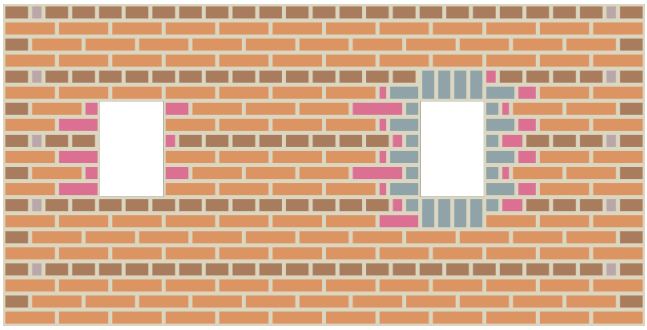


Figure 3.5: Fitting of bricks against borders, modified bricks are coloured in magenta

In the following explanations we will use n for the number of repetitions of a row's repeating pattern, w for the width of the wall, and S, H and Q for the widths of a stretcher, header and queen-closer brick respectively.

Simple Pattern Generator

This pattern simply consists of stretcher bricks stacked exactly on top of each other creating a pattern of rows and columns. To adjust the brick width, there is a single equation: $n \cdot S = w$. We calculate n as mentioned above by counting the amount of bricks that fit into the row with the preferred brick width.

An example image is shown in figure 5.3 (a).

Scottish Bond Pattern Generator

The Scottish bond has three distinct types of rows.

- Type 1: $HQH[HH]^*QH$
- Type 2: SS^+
- Type 3: HS^+H

By looking at type 2 and 3 one can derive the equation $S = 2 \cdot H$, which is easy to see without repetition. We decided on which bricks per row are required at minimum by looking at all row types and their minimum bricks with the fixed relation between headers and stretchers. Note that a queen-closer is per definition half a header. After this we could reformulate these regular expressions as:

- Type 1: $(2n + 4) \cdot H = w$
- Type 2: $(n + 2) \cdot S = w$
- Type 3: $2 \cdot H + (n + 1) \cdot S = w$

Now we calculate n with our usual method and the row type 3, since it uses both brick types and therefore modifies both only a little.

The equation of type 2 and the relation between headers and stretchers immediately give us the fitted brick dimensions.

The Scottish bond usually has five rows of alternating type 2 and 3 before another row of type 1 is added, but we also added a frequency parameter to select the interval of repetition, thereby being able to create an *American bond*, which has nine rows in between, and others.

An example image is shown in figure 5.3 (b).

Flemish Bond Pattern Generator

The Flemish bond has two distinct row types.

- Type 1: $HQS[HS]^*QH$
- Type 2: $SHS[HS]^*$

The equation $S = 2 \cdot H$ is again valid, which is trivial to see when leaving out the repeating pattern ($n = 0$).

- Type 1: $(n + 1) \cdot S + (n + 3) \cdot H = w$
- Type 2: $(n + 2) \cdot S + (n + 1) \cdot H = w$

We can compute n the like we did for the Scottish bond with one of the rows by adding up bricks and counting the maximum possible repetitions.

Now we have to solve the equation system constituted by the two row types with the solution being: $H = \frac{w}{3n+5}$. We can again compute S afterwards with the simple relation between headers and stretchers.

An example image is shown in figure 5.3 (c).

Sussex Bond Pattern Generator

The Sussex bond has two distinct row types.

- Type 1: $HQSSS[HSSS]^*QH$
- Type 2: $SS[HSSS]^*HSS$

As before the equation $S = 2 \cdot H$ is valid (visible with $n = 0$).

- Type 1: $(3n + 3) \cdot S + (n + 3) \cdot H = w$
- Type 2: $(3n + 4) \cdot S + (n + 1) \cdot H = w$

The solution of the equation system is: $H = \frac{w}{7n+9}$.
An example image is shown in figure 5.3 (d).

Monk Bond Pattern Generator

The Monk bond has two distinct row types.

- Type 1: $H[SSH]^+$
- Type 2: $SQH[SSH]^*QS$
- Type 1: $(2n + 2) \cdot S + (n + 2) \cdot H = w$
- Type 2: $(2n + 2) \cdot S + (n + 2) \cdot H = w$

Since the two equations are the same and we therefore only have one equation with two variables, the equation system is under-determined. We solve this by using the formula $S = 2 \cdot H$ which is valid in the other brick bonds.

The solution to the equation system is now: $H = \frac{w}{5n+6}$.
An example image is shown in figure 5.3 (e).

3.6 Stone Pattern Generator

Our method of creating stone wall patterns is based on the idea of dividing the wall into a two-dimensional array of stones of minimum size, which are then merged to create larger stones. The minimum size of these stones is the greatest common divisor of the sizes of any larger, merged stone, and hence we call them *gcd-stones*.

First we compute the gcd-stone size. We assume squared gcd-stones and use the same method for finding the brick height in brick patterns, which is fitting the height into the greatest common divisor area of all areas enclosed between wall and element borders, but we include the horizontal areas as well in our computation. Thereby we compute a stone size that fits into every part of the wall, so that wall elements are surrounded perfectly.

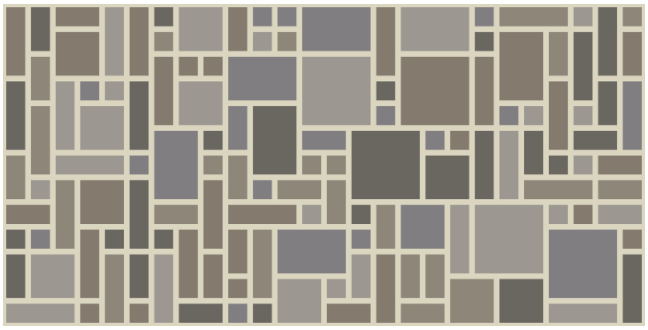


Figure 3.6: A random stone pattern created with our method

Then we initialize an *occupancy map*, which is a two dimensional array of indices. The occupancy map represents the stones on the wall, and its indices represent the individual stones and occupied spaces.

The occupancy map is initialized as completely empty. Afterwards the wall elements including their created borders are rasterized into the occupancy map as occupied. For the remaining spaces the following stone growing/merging algorithm is used.

As long as the occupancy map is not fully occupied, the next empty space is taken and a gcd-stone is created with a new index. The stone's size is then increased randomly and then decreased to fit into the wall with regards to already occupied spaces from other stones or wall elements. We increase the stone size by first trying to let it grow horizontally once with a probability of 50%, and then a second time with a probability of 50%. The same is done vertically.

The occupancy map is thereby filled with stones represented by different indices. The indices are converted into stone cells, which may later be rendered. Each group of identical indices constitutes one stone. This produces a random pattern of rectangular stones.

3.7 Wall Elements

Wall elements are fixed position and fixed size elements of any kind on the wall, e.g. windows. They may have a border of bricks around them. As we already stated, to fit the bricks of the general pattern to the wall elements and their borders, we store the limits for each row and each element against which new bricks are fitted.

We create a *limit map*, which is our data structure for storing limits mapped by row height, for each element. The limit map is initialized with limits for each row a wall element overlaps as the wall elements start and end. It is then modified when adding border bricks, either by adding limits for a row or expanding them. It is necessary to initialize the limit maps for each element before starting to create their borders, as border bricks have to be fitted against other wall elements, even if their borders have not yet been created, which happens iteratively, i.e. one element after another.

Since it is possible that wall elements leave no space between them and the wall borders for a border of bricks around them, this has to be considered when creating the borders.

Wall elements and their borders are considered in the stone pattern generation by reserving areas in the occupancy map.

We implemented two types of borders apart from using no border:

- A simple border that has side columns of rectangular bricks and top and bottom rows of modified stretcher bricks. The side bricks have the size of the row height and size of the modified stretcher is calculated with the method of trying to fit as many bricks as possible into the available space.

- A border whose side columns have bricks of alternating width and whose top and bottom rows are pseudo-soldier bricks (pseudo because it is a standing up brick, but its standing height is simply the height of two rows and not the longest length of the brick). The soldier bricks width is calculated by trying to fit the row height into the available space. Problems may arise with other wall elements and their borders, as our soldier bricks occupy two rows and we only provide limits and fitting for single rows. We solved this by only allowing soldier bricks when the full element length is free in two rows.

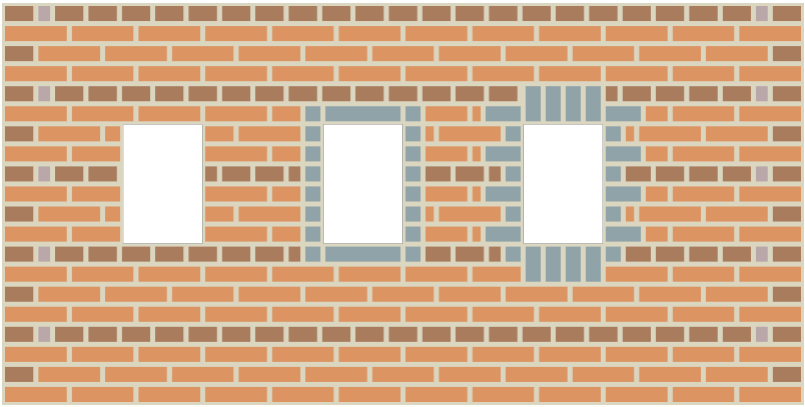


Figure 3.7: Different types of borders around wall elements

Herringbone Bond

We implemented an example of a non-empty wall element that is filled with non-standard bricks. The *Herringbone bond* is a brick pattern where bricks are tilted $45°$ and difficult to represent with our standard cells. We solved this by creating its bricks normally (i.e. horizontally and vertically) in a tilted coordinate system and then tilting them back into the normal coordinate system. We have to calculate the the positions of the element's corners in the tilted coordinate system trigonometrically as they are needed for loop control.

To fit the bricks directly to the wall element's available space, we produce as many bricks as are necessary for occupying the whole space and clip them to the element's size.

We use the brick height of the general pattern also as the brick height in the element but then use double the height as width. This looks better because it creates more columns of bricks that are visually of a fitting size, as compared to using the dimensions of the other bricks directly.

An example is shown in figure 3.8

Element groups

Furthermore we added the functionality of creating groups of similar elements. A group is simply defined by a basic wall element, the amount of rows and columns and the distance between consecutive elements in a row or column.

An example of a group of four elements is shown in figure 3.8

Variation

We have not yet talked about brick colour or texture, since our main focus is to generate a realistic pattern with cells and these cells can be textured arbitrarily while rendering. Nevertheless, we added the option of varying a bricks colour or texture with a randomly generated variation index, which is used in rendering, beside the brick type, to decide on a specific colour or texture. An example of variation is shown in figure 3.8.

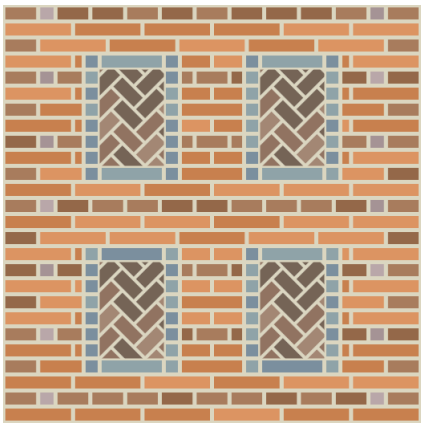


Figure 3.8: A group of Herringbone bond elements with colour variation

3.8 Rendering

In the rendering step the bricks and border bricks are converted into drawable entities whilst considering their type and their variation index for colour or texture. Afterwards these entities are rendered.

We also have the possibility of rendering non-standard entities, such as non-standard cells directly. We create the filling of the Herringbone bond wall element here, as we use framework dependent geometric and renderable classes for this.

CHAPTER 4

Implementation

4.1 Framework

We implemented our method on a PC running Microsoft Windows 8 Pro x64 in Visual Studio 2012. The programming language was C# using the Windows Presentation Foundation (WPF) framework with the `Model-View-Viewmodel` pattern (MVVM) for the user interface and rendering. We used the NotifyPropertyWeaver Visual Studio add-in for facilitating property updates in the GUI.

4.2 Input

Pattern Parameters

The `PatternParameter` class contains all necessary parameters for generating our patterns. The individual parameters are editable in the GUI, which is explained in the next subsection. The class consists of general parameters, such as the mortar width and the colour theme, and other types of parameters sets, such as `GeneratorParameters` and a list of `ElementParameters`. The individual parameters are either `double` or `enum`. The class `ElementGroupParameters` extends the class `ElementParameters` with the count and distance parameters and can therefore be displayed in the same list box in the GUI.

All parameters are serializable, annotated with further XML attributes and can be serialized and de-serialized with the .NET XML serialization framework, which gives us the possibility to store and load parameters from XML files.

Graphical User Interface

The user interfaces is defined as an XAML file. The general structure is as follows: A menu bar at the top, expandable sections for controlling and editing parameters on the right side, and the rest is the canvas on which the patterns are rendered. Noteworthy is that the parameters are bound

to the PatternParameters property via MVVM bindings, the double valued parameters are displayed with two decimal places and the buttons are bound to RelayCommands that execute methods. Figure 4.1 depicts our GUI with different expanded sections.

4.3 Generation

Bricks

We use the Brick class for both bricks and stones. The class extends Quad, which simply contains a starting and ending Point, with a ColorIndex, an int value representing a colour or texture, and an uint as the variation, which represents different colours or textures for the same colour index.

We use a BrickFactory to create bricks either by a brick type enum or directly with location, size and colour index.

Pattern Generators

All pattern generators implement the IPatternGenerator interface, which declares the Generate method. This method accepts the PatternData, which holds a List of Quads and a List of WallElements, and the PatternParameter instances as parameters, creates the bricks of the pattern and adds them to the brick list.

We decide upon whether to choose a brick pattern generator or the stone generator by which tab is currently opened in the expandable section of the GUI, i.e. the brick or stone tab.

Brick Pattern Generators

The brick pattern generators have the common superclass AbstractBrickPatternGenerator, which implements the method FitAndAddBrick. This method takes a generated brick, fits it against all wall elements and adds it to the brick list. The GeneratorFactory class creates a brick pattern generator from a generator type enum.

Notably the brick pattern generators cast the double values of the current position to decimal in their loop conditions to avoid problems with rounding errors, where too many or too little bricks would be created else.

```
for (double y = 0; (decimal)y < (decimal)planeHeight; y += height)
```

Stone Pattern Generator

Since we only have one stone pattern generator we do not use any special factory.

The stone pattern is created with a occupancy map. Our implementation is the OccupancyMap class, which serves as a wrapper around a two-dimensional int array for storing the stone indices and provides further functionality. These include filling of rectangular areas inside the map and flood filling. We use negative indices for reserved places of wall elements and their borders, zero for empty spaces and positive indices for stones. Notable is the method ToQuads, which accepts the gcd-stone size as parameter and converts the indices of

the occupancy map into stones. For each index that is not yet converted, it looks for the smallest and largest x and y positions and creates a `Brick` object respectively. The method also uses a clone of the same occupancy map, on which each checked index is set to 0 in order to specify which indices and parts have already been converted.

Wall Elements

The wall element parameters are inflated by the `ElementInflater` class before the generators are created. Depending on whether it is a `ElementParameters` or a `ElementGroupParameters` instance, the parameters are inflated into different amounts of wall elements, which are added to the list in the `PatternData` instance. Element parameters contain their type and their type of border as enumerations.

The `WallElement` class is the super class of all wall elements and provides the methods `FitBrick`, `InitializeLimitMap` and `CreateBorders`. `FitBrick` fits a brick specifically to this element, by looking up the row's limit and, if there is one, possibly shortening or splitting the brick. Because of the possibility of splitting the method accepts and returns lists of bricks. `InitializeLimitMap` initializes the element's row limits as the elements main area. `CreateBorders` inflates the border generator by type and creates the border bricks while updating row limits. The class also declares two abstract methods, `Initialize` and `CreateFill`. `Initialize` is called by pattern generators to initialize the wall element with computed parameters and create initial row limits by calling `InitializeLimitMap`. `CreateFill` is called in the rendering step to create a Drawing of non-standard elements, as is the case in the Herringbone-bond element.

Herringbone Bond Element

We create the bricks in the tilted coordinate system with our `BrickFactory` class. They are later converted into Geometry objects, which are then transformed with rotation of $-45°$ and translation into the normal coordinate system. We use individual `GeometryGroup` objects for bricks and mortar, on which the transformations are executed. The mortar geometries are added to one `GeometryDrawing`. Each brick geometry is added to an individual `GeometryDrawing` because they might have different colour variations. All drawings are then added into a single `DrawingGroup` and returned for rendering.

Border Generators

Border generators implement the `IBorderGenerator` interface, which defines the `GenerateBorders` method. This method accepts an instance of `BorderGeneratorParameters`, which is a structure containing the necessary parameters for border creation, and returns a list of the generated border bricks. `BorderGenerators` are created by the `BorderGeneratorFactory` by an `enum` border type.

We implemented the `LimitMap` as a class containing a list of nodes. A `Node` represents a row limit. It is defined by its height and has a starting and ending point. All values are `double`. When adding a new limit, the list is searched for a limit with the same height, and if it exists

it is updated by comparing and taking the minimum and maximum limits, i.e. a limit can only be expanded. The limit heights are stored and looked up with heights rounded to five decimal places due to rounding error problems.

In the concrete border generators we use temporary lists for storing the generated bricks. The list is iterated afterwards and each brick is fitted against all other wall elements and then a limit is created for the updated brick.

Double-valued Euclidean Algorithm

We implemented the *Euclidean algorithm* for `double` values, which is used in the greatest common divisor area computation, e.g. for adjusting row heights. Instead of using the modulo operator, we subtract the smaller number from the larger one until it would leave a negative rest. This gives us a much more accurate result than the modulo operator.

Result: The greatest common divisor of two `double` values

```
1  Algorithm EuclidDouble(double a, double b)
2      if a < b then
3          swap a and b
4      end
5      rest ←a;
6      while rest − b >= 0 do
7          rest = rest − b;
8      end
9      if (rest == 0) then return b;
10     else return EuclidDouble(b, rest);
```

Algorithm 4.1: Euclidean algorithm for `double` values

4.4 Output

Brick Converter

Bricks are converted into `DrawingGroup` objects that include both the brick and the mortar, since we defined mortar implicitly in a brick cell. We defined that a brick is too small, if there is more mortar than brick in a cell. Specifically if the mortar size is larger than half of the smallest brick dimension, only the mortar is drawn. If the mortar size is set to 0, only the brick is drawn. In any other case separate drawings are created for the brick and the mortar. The mortar is created with a `StreamGeometry` to create the rectangular shape with the rectangular hole where the actual brick is.

Brushes

The `BrushLoader` class provides methods for loading `ImageBrush` objects from texture files.

The `BrushMap` class is very important in rendering. We specified all types of different bricks or stones in the `ColorIndex` enumeration, i.e. header, stretcher, border, stone and more. Mortar is excluded from this and stored separately as a single `Brush` in the `BrushMap`. The `BrushMap` contains a `Dictionary` that maps a `ColorIndex` to a list of brushes. Which brush of a list is chosen depends on the variation of the brick. A brick brush is looked up by `ColorIndex` and variation. We map the variation integer to a valid index of a colour-index-specific list by calculating $index = variation\%variationcount$, where $variationcount$ is the amount of stored brushes for that specific colour index. This allows us to add variable amounts of brushes for each type of brick. The `BrushMap` is initialized with a `ColorTheme`, which declares which kinds of brushes are loaded, i.e. `SolidColorBrush` or `ImageBrush` objects.

We downloaded free textures for our brushes from `http://seamless-pixels.blogspot.co.at/` [18] and then edited them strongly by down-scaling, hue-shifting, adjusting saturation and contrast, merging, overlaying etc in order to have harmonizing textures.

Problems of texture stretching, blurring or tiling occured, which may be DPI-dependent errors between the image files and the WPF canvas. We solved this by setting the `ViewportUnits` to absolute units and the `Viewport` to down-scaled dimensions of the true image in our `ImageBrush` objects.

4.5 Debug

We implemented a `Debug` class with properties hooked to the GUI to facilitate debugging of brick fitting, variations, Herringbone bond element and borders. Its properties are accessible globally and can therefore be used anywhere in conditional statements. To be more specific, we implemented `bool` debug flags for deciding whether black line borders should be drawn around the walls and wall elements, whether the Herringbone bond element's bricks should be left unclipped, whether fitted bricks should be coloured magenta and whether no colour or texture variations should be used for bricks and stones.

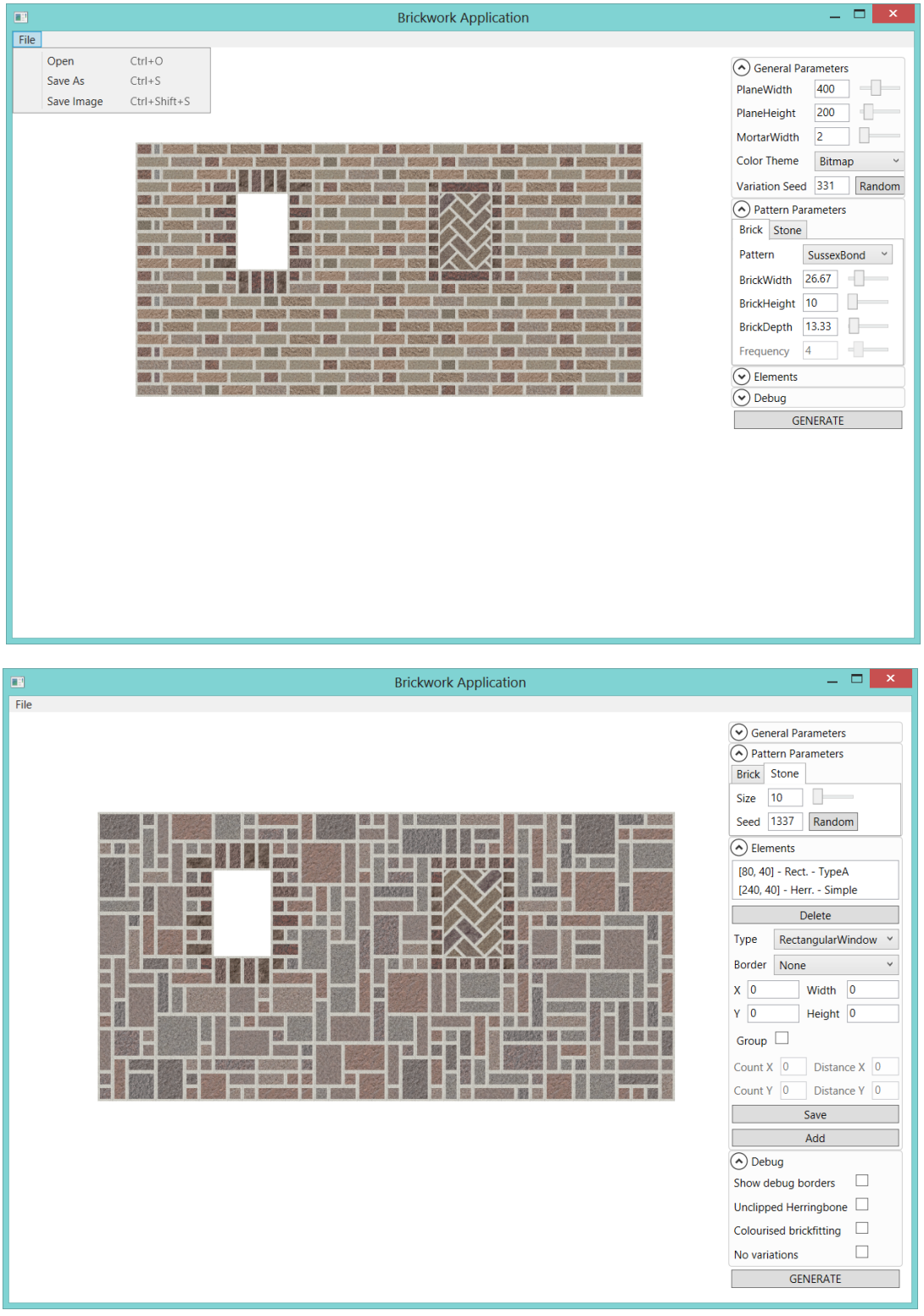


Figure 4.1: Our GUI with different expanded sections

CHAPTER 5

Results

In the following we are going to show the results of our implemented method. We will discuss some important special cases along with rendered examples.

5.1 Variation

Figure 5.1 depicts a Flemish bond pattern with the solid colour theme, where each brick type has two slightly different colours. In figure 5.2 three example variations of the bitmap theme are shown. They use the same parameters except for a different variation seed and have a higher number of varying textures.

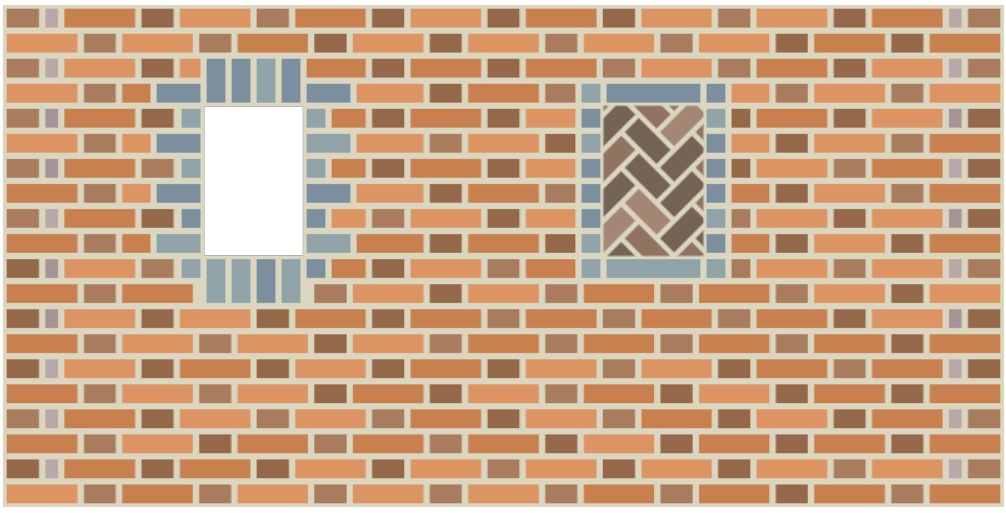


Figure 5.1: Colour variation

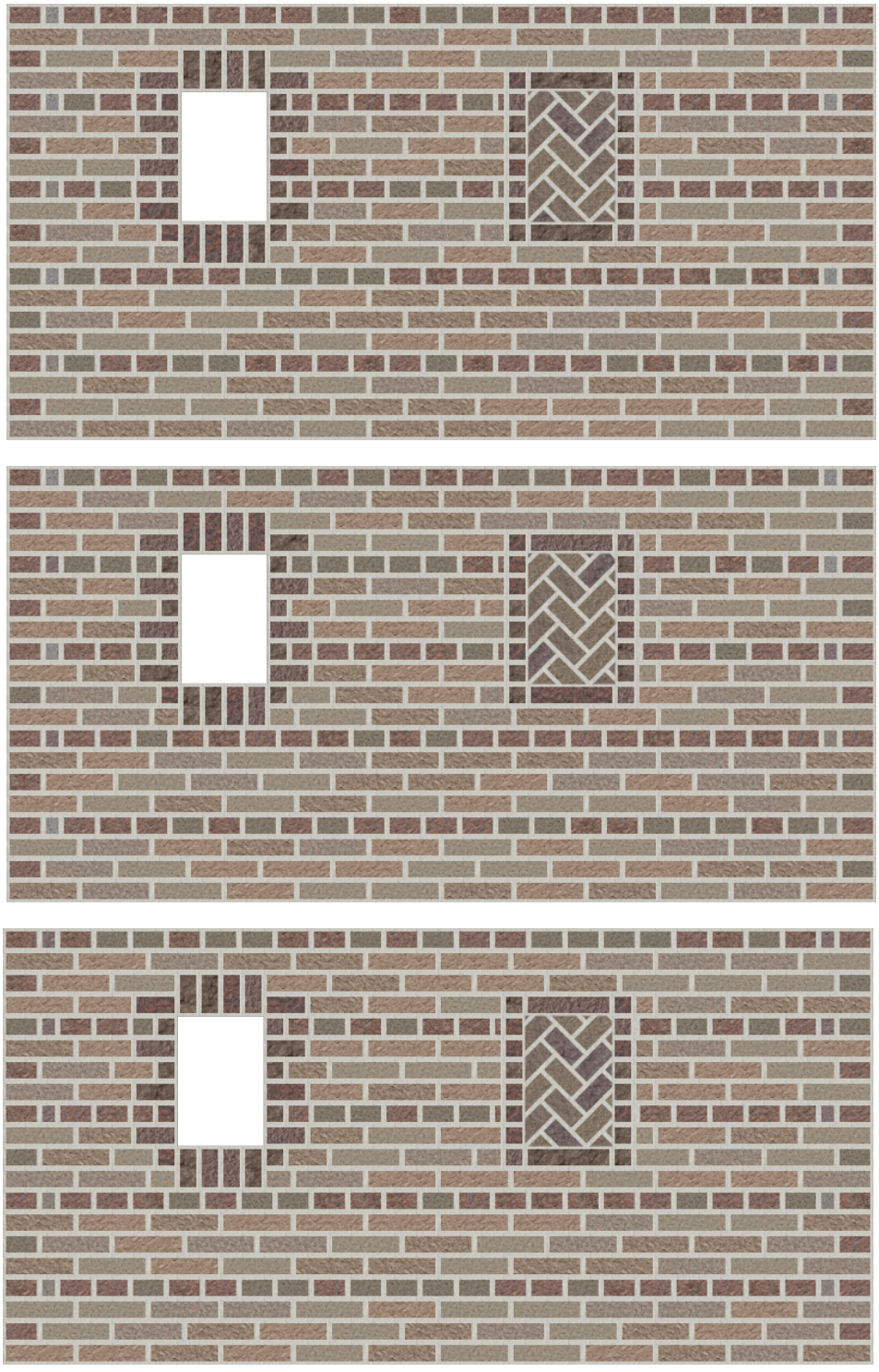


Figure 5.2: Bitmap texture variation with different variation seeds

5.2 Patterns

The following examples show the different patterns our method is able to create. Figure 5.3 shows the different brick patterns and figure 5.4 shows three random stone patterns that only differ in their seed parameter.

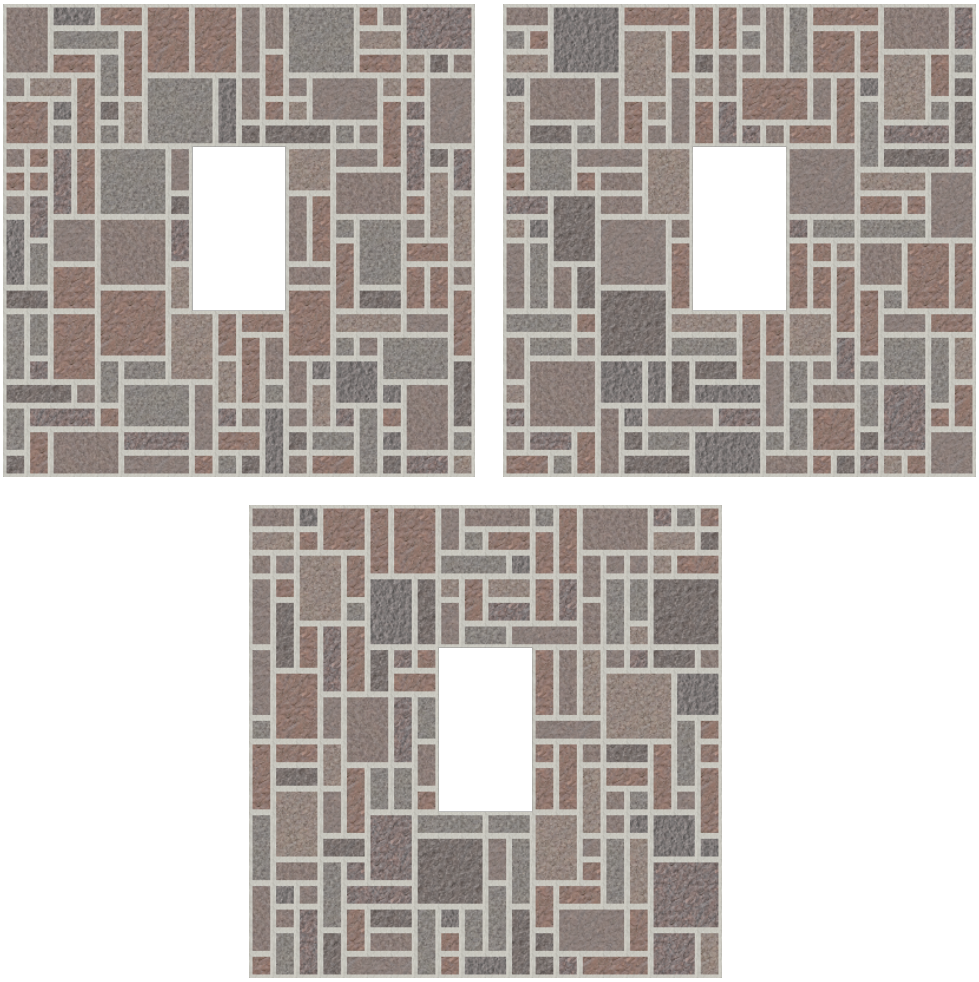
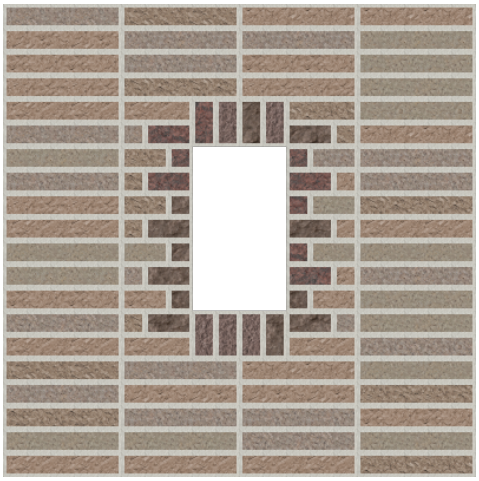


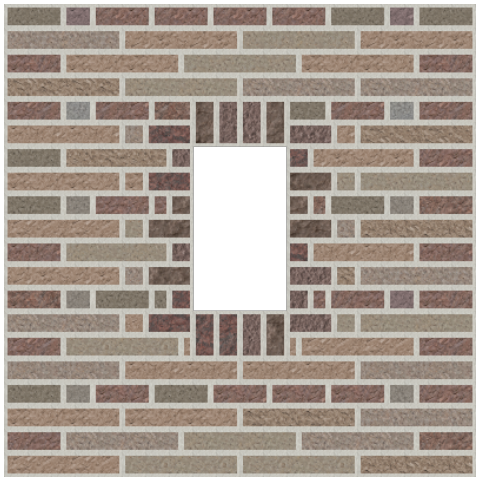
Figure 5.4: Stone patterns with different initial seeds

5.3 Mortar

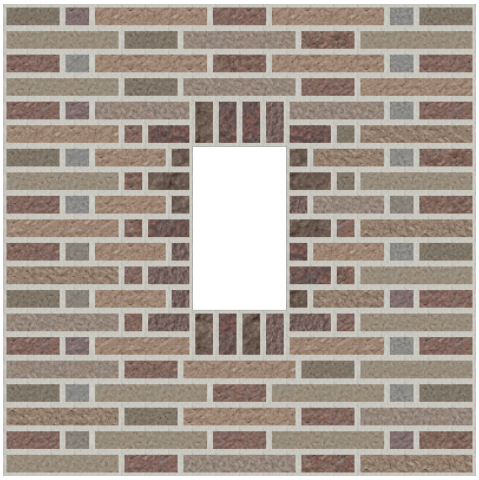
The mortar strength is adjustable globally in our method. The same wall and pattern with different mortar strengths is depicted in figure 5.5. One can see the special case of mortar exceeding the threshold of a brick's smallest halved dimension when comparing figure 5.5 (c) and figure 5.5 (d) where e.g. the queen-closer bricks are rendered as mortar only.



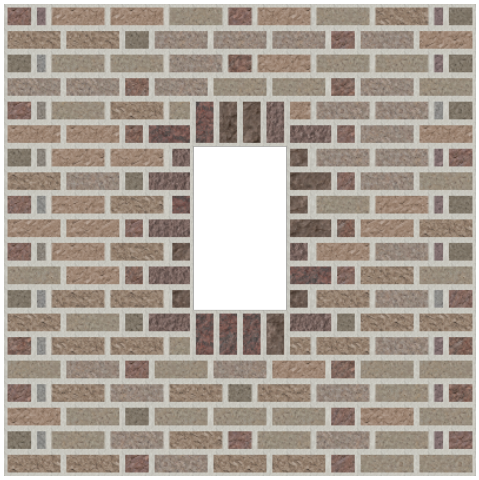
(a) Simple brick bond



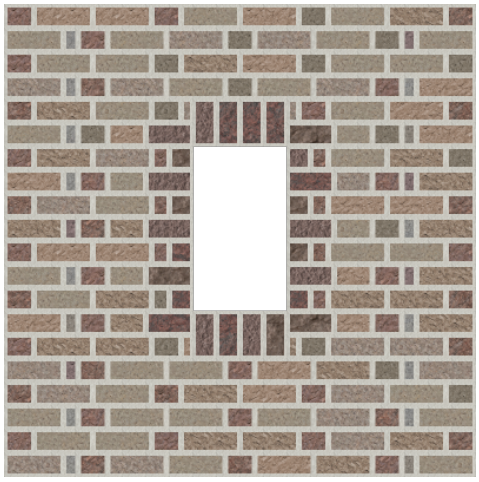
(b) Scottish bond



(c) Flemish bond

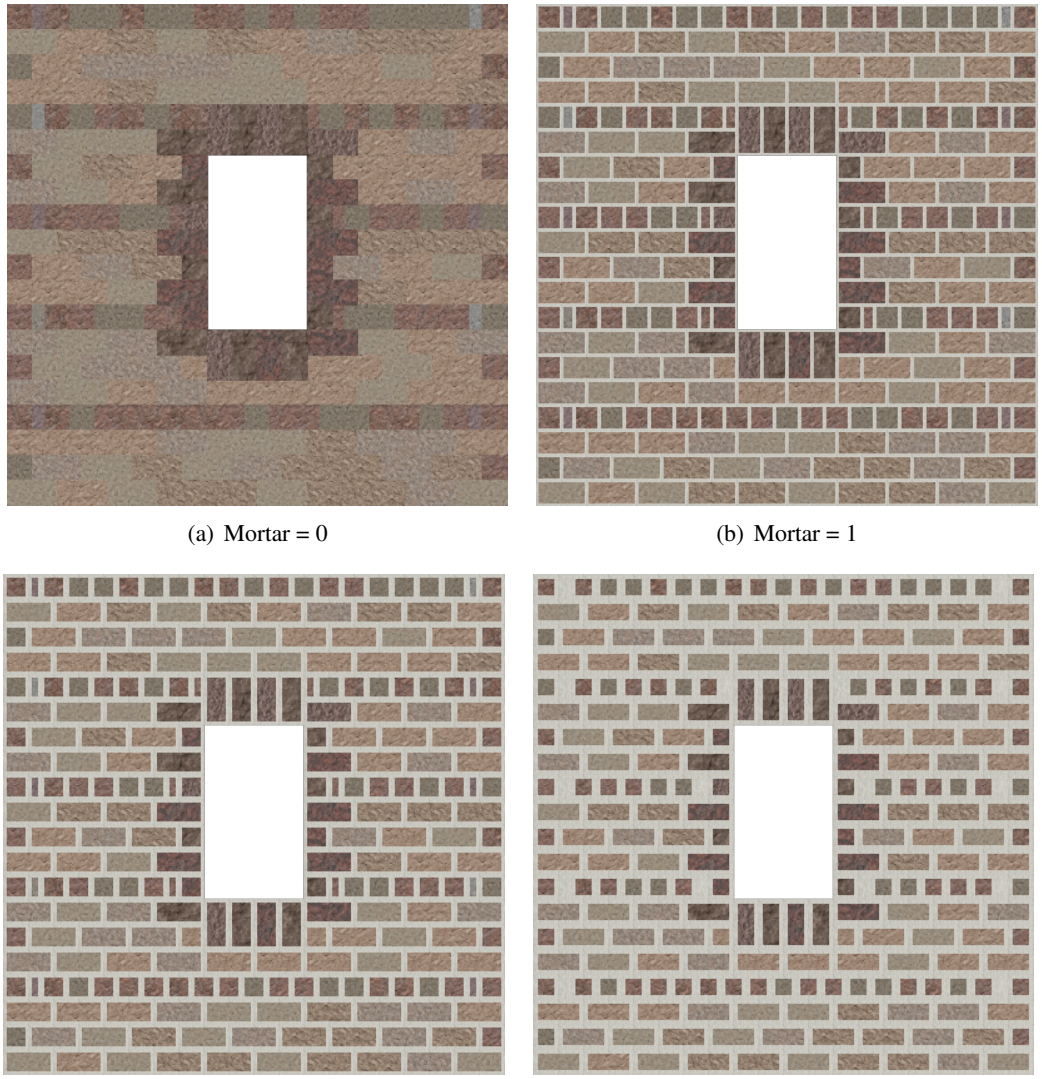


(d) Sussex bond



(e) Monk bond

Figure 5.3: Brick bonds with bitmap textures



(a) Mortar = 0 (b) Mortar = 1

(c) Mortar = 2 (d) Mortar = 3

Figure 5.5: Different mortar strengths

5.4 Special Cases

The following examples depict some interesting special cases, which should either be avoided or occur only in specific situations.

Borders

Figure 5.6 depicts examples of wall elements being too close to or even directly at the wall borders, so that their borders either have to be limited or can't be drawn at all.

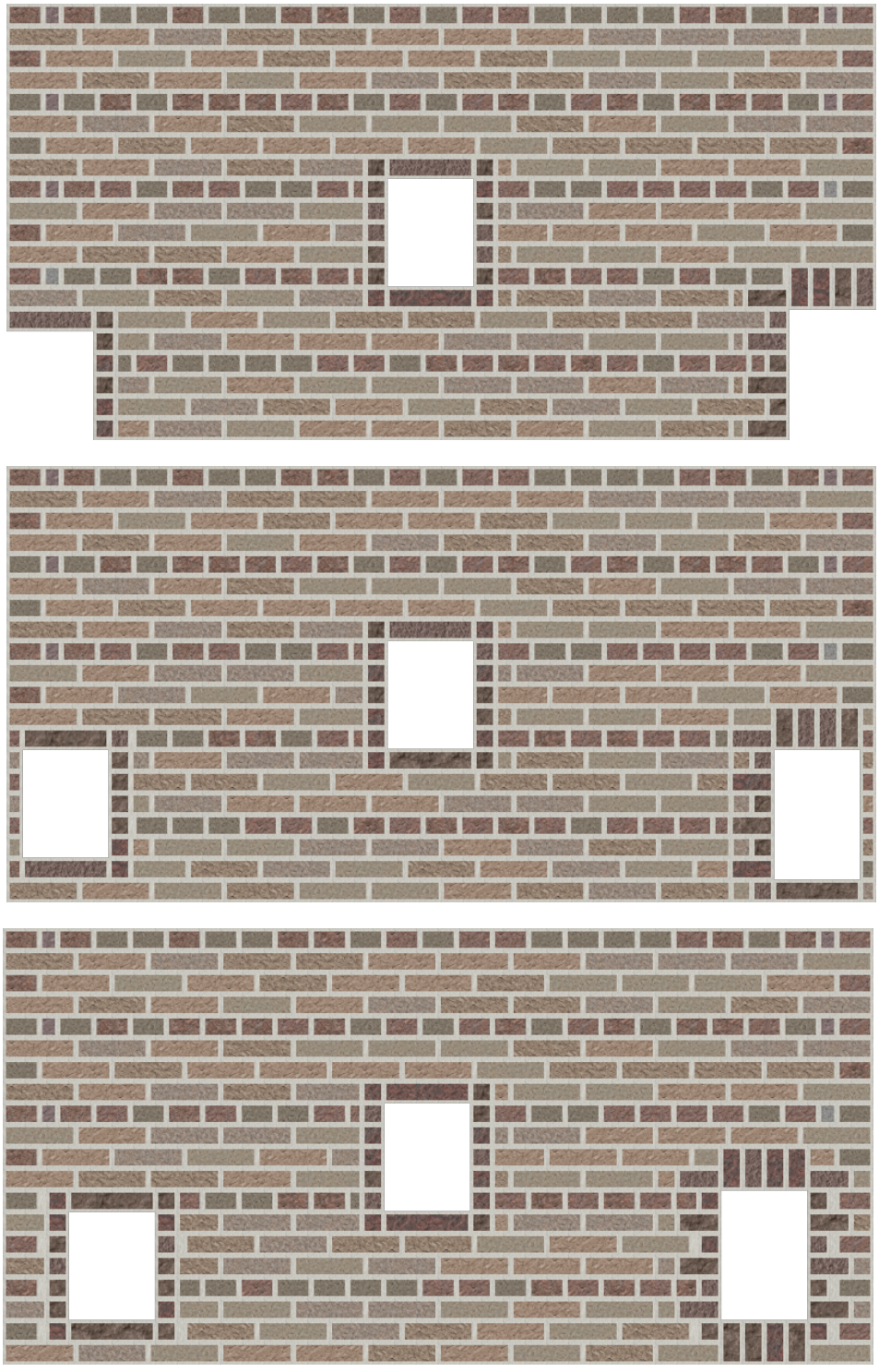
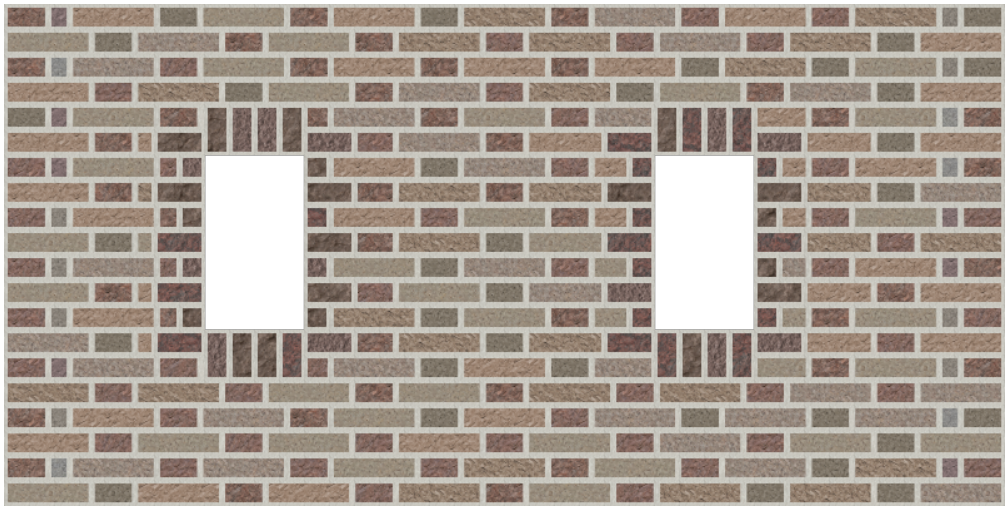


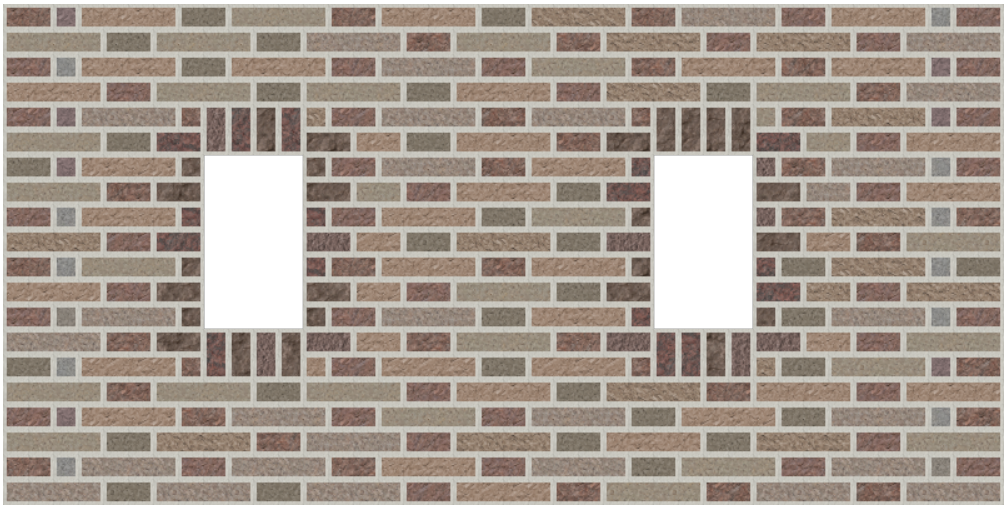
Figure 5.6: Special cases of border handling on wall borders

Brick Enclosure

Since our second border type uses alternating brick widths for different rows and since the general pattern's bricks are cut off at the elements' borders, it can happen that bricks are almost enclosed in the remaining spaces of the alternation, as is depicted in figure 5.7 (a) on the left side of the left window. This artefact can however be avoided rather easily by adjusting the brick parameters. Figure 5.7 (b) is the same pattern as before but with a slightly modified brick width, which thus solves the randomly occurring artefact.



(a) Bricks are enclosed on the left side of the left element



(b) Slightly altered brick widths solve the artefact of enclosure

Figure 5.7: Example of enclosed bricks

Element Overlap

It is generally not an allowed, standard use case of our method to overlap wall elements. However, since we considered wall element border overlap, it is possible to overlap empty windows. An example is shown in figure 5.8 with different borders.

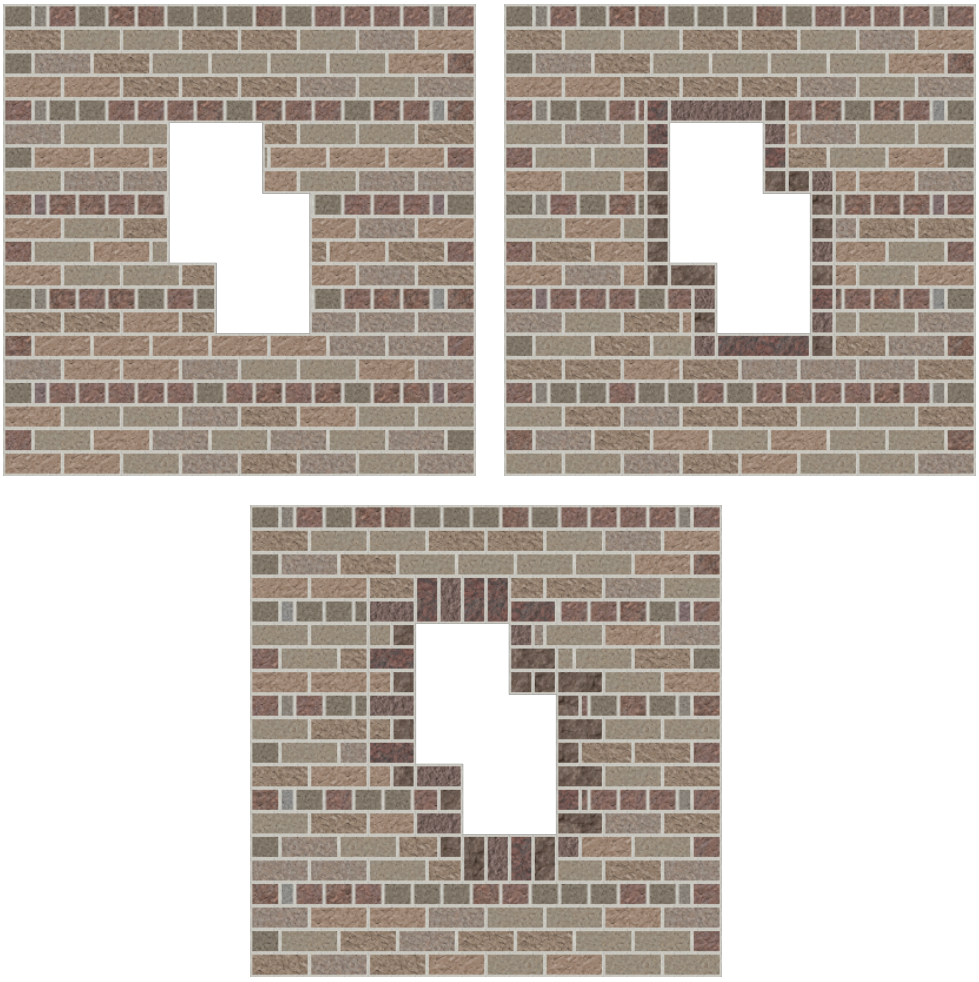


Figure 5.8: Special case of overlapping windows

5.5 Multiple Floors

Figure 5.9 shows an example of a wall using a modified Scottish bond pattern to visualize different floors.

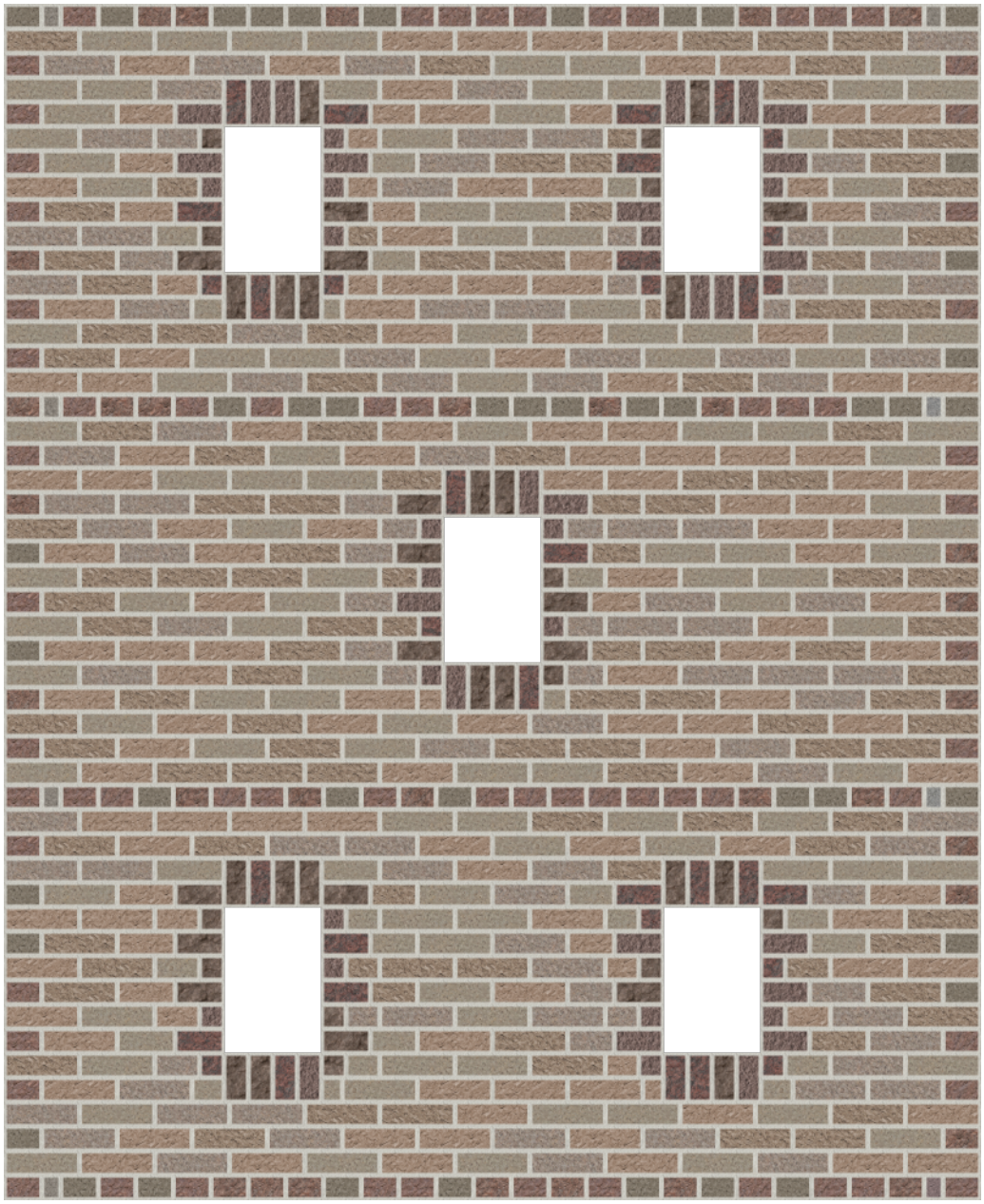


Figure 5.9: A wall with multiple floors

5.6 XML Parameters

All our input parameters are annotated for XML serialization, which allows us to save and load parameter sets. One example XML file is shown in listing 28.

```
<?xml version="1.0" encoding="utf-8"?>
<PatternParameters xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Plane Width="400" Height="200" />
  <ColorTheme>Bitmap</ColorTheme>
  <VariationSeed>891</VariationSeed>
  <MortarWidth>1</MortarWidth>
  <WallElements>
    <ElementGroup X="80" Y="60" Width="40" Height="70">
      <Type>RectangularWindow</Type>
      <BorderType>TypeA</BorderType>
      <DistanceX>140</DistanceX>
      <DistanceY>0</DistanceY>
      <CountX>2</CountX>
      <CountY>1</CountY>
    </ElementGroup>
  </WallElements>
  <Generator>
    <Type>FlemishBond</Type>
    <BrickWidth>40</BrickWidth>
    <BrickHeight>10</BrickHeight>
    <BrickDepth>20</BrickDepth>
    <Frequency>4</Frequency>
    <Seed>1337</Seed>
    <GcdSize>5</GcdSize>
  </Generator>
</PatternParameters>
```

Listing 5.1: An example XML parameter file

5.7 Limitations

Our method can produce many real brick bond patterns and one can add more by following the general procedure of finding row patterns and formulating equations. There are, however, limitations.

We generate our texture in software and not in GPU shaders. Patterns where bricks occupy more than one row, as is the case with patterns that contain soldier bricks, or have other multi-row constraints, are not yet possible, since our brick fitting method is based on single rows with the element's row limits. We do not have a structure for neighbouring stones, which could provide easier methods for correlating bricks, e.g. if one brick shrinks another neighbour grows etc. Our stone pattern only creates rectangular stones. If the position and dimensions of wall elements and the dimensions of the wall are only a little off thus creating gcd areas that are too small, our method creates bricks with the height of that small area and thereby too many rows. This could

however be countered with allowing “soft” boundaries of wall elements, i.e boundaries that are not taken into account for gcd computation and against which bricks are fitted by simply cutting them.

There are many possible improvements to our method, which will be discussed in the next chapter.

CHAPTER 6

Conclusion

6.1 Summary

This thesis focused on introducing a method for procedurally generating realistic brick bond patterns and random stone patterns for walls with windows and other wall elements by using cells. First we introduced our general idea and motivation. Then related topics were discussed. Afterwards we explained the method in greater detail and showed prominent parts of our specific implementation. We also provided many image results and discussed special cases. Our method differs from related work as it creates realistic brick bond patterns with consideration of windows, whereas related methods create random or more abstract patterns.

6.2 Conclusion

In conclusion this project had many difficulties beginning with the definitions of how a single pattern might be built, along with deciding on and assessing constraints. Constituting the correlation between wall elements and their borders with the general pattern, i.e. defining how to compute the row height and when to clip bricks in what way, was also a major stepping stone. We also focused on programming our method in a very generic and extensible way with abstractions of pattern generators, converters to convert cells into renderable entities, etc. This facilitates modifiability, extensibility and maintainability for any future work or changes. Our method and implementation thereof are furthermore aimed to provide simple and intuitive use.

Moreover, we can conclude that it would be possible to create even more realistic textures by incorporating procedural noise for generating colours and roughness of individual bricks and mortar.

In addition, despite the numeric nature of computation and number storage it is possible to avoid problems caused by rounding errors by using data types that provide more accuracy or by explicitly setting a fixed amount of decimal places and rounding.

The applications of our method cover a broad range. The textures it is able to produce can be used in any application that contains architectural models. Since the textures are generated to be realistic, they can augment those models, giving the application more believability and realism. Such architectural models are used and incorporated in the filming industry, urban planning and reconstruction applications, as well as in training and simulation applications.

Procedural modelling is another important area where our method could be applied effectively. Models created in procedural modelling are generated with a set of parameters. One could use these parameters more or less directly for generating a texture specifically fitted to a generated model. Since procedural modelling often involves parameter tweaking and regeneration of different models multiple times, these specifically matched textures create realistic visuals in every instance and substitute stretched, tiled or simple colour textures.

6.3 Future Work

Ultimately we will list possible improvements and extensions to our method.

- More brick bonds can be added.
- Patterns with multi-row constraints can be implemented.
- The individual stones, bricks and the mortar can be texturised using procedural noise.
- The patterns can be made more realistic by using the computed data in bump mapping or displacement mapping.
- The random stone pattern generator could be split into sequential, chain-able and interchangeable pattern generators guided by constraint data structures such as the occupancy map.
- The stone colour can be chosen using the four colour theorem.
- The probabilities of growing or merging stones could be computed with Markov chains.
- Neighbourhood data structure can be introduced for simplifying correlated modification of bricks.
- The method can be extended to 3D using additional constraints along mesh edges.
- More advanced types cells can be introduced, e.g. tilted cells, non-rectangular cells, etc.
- Soft element boundaries can be implemented, where bricks are cut along their height rather than fitted against them.

Bibliography

[1] Li-Yi Wei, Sylvain Lefebvre, Vivek Kwatra, and Greg Turk. State of the art in example-based texture synthesis. In *STAR - State of The Art Report*, 2009.

[2] Ares Lagae, Peter Vangorp, Toon Lenaerts, and Philip Dutré. Procedural isotropic stochastic textures by example. *Computers & Graphics*, 34(4):312–321, 2010.

[3] Ares Lagae, Sylvain Lefebvre, Rob Cook, Tony DeRose, George Drettakis, D.S. Ebert, J.P. Lewis, Ken Perlin, and Matthias Zwicker. State of the art in procedural noise functions. In Helwig Hauser and Erik Reinhard, editors, *EG 2010 - State of the Art Reports*. Eurographics, Eurographics Association, May 2010.

[4] Robert L. Cook and Tony DeRose. Wavelet noise. *ACM Trans. Graph.*, 24(3):803–811, July 2005.

[5] Jeppe Revall Frisvad and Geoff Wyvill. Fast high-quality noise. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, GRAPHITE '07, pages 243–248, New York, NY, USA, 2007. ACM.

[6] Ares Lagae, Sylvain Lefebvre, George Drettakis, and Philip Dutré. Procedural noise using sparse gabor convolution. *ACM Trans. Graph.*, 28(3):54:1–54:10, July 2009.

[7] G. Gilet, J-M. Dischler, and D. Ghazanfarpour. Multiple kernels noise for improved procedural texturing. *Vis. Comput.*, 28(6-8):679–689, June 2012.

[8] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985.

[9] Steven Worley. A cellular texture basis function. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 291–294, New York, NY, USA, 1996. ACM.

[10] Bryan Chan and Michael McCool. Worley cellular textures in sh. In *ACM SIGGRAPH 2004 Posters*, SIGGRAPH '04, pages 18–, New York, NY, USA, 2004. ACM.

[11] Kazunori Miyata. A method of generating stone wall patterns. *SIGGRAPH Comput. Graph.*, 24(4):387–394, September 1990.

[12] Justin Legakis, Julie Dorsey, and Steven Gortler. Feature-based cellular texturing for architectural models. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 309–316, New York, NY, USA, 2001. ACM.

[13] Kaisei Sakurai, Kazunori Miyata, Naoki Kawai, and Kazuo Matsufuji. Procedural modeling of leather texture with structural elements. In *Proceedings of the Fifth Eurographics conference on Natural Phenomena*, NPH'09, pages 1–8, Aire-la-Ville, Switzerland, Switzerland, 2009. Eurographics Association.

[14] Kurt W. Fleischer, David H. Laidlaw, Bena L. Currin, and Alan H. Barr. Cellular texture generation. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 239–248, New York, NY, USA, 1995. ACM.

[15] Fabrice Neyret and Marie-Paule Cani. Pattern-based texturing revisited. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '99, pages 235–242, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[16] Sylvain Lefebvre and Fabrice Neyret. Pattern based procedural textures. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, I3D '03, pages 203–212, New York, NY, USA, 2003. ACM.

[17] Wikipedia. Brickwork — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Brickwork, 2013. [Online; accessed 17.09.2013].

[18] Seamless-Pixels. Free seamless textures. http://seamless-pixels.blogspot.co.at/, 2013. [Online; accessed 17.09.2013].