THE HEBREW UNIVERSITY OF JERUSALEM

Texture Synthesis for Particle Infused Textures

by

Ohad Fried

A thesis submitted in partial fulfillment for the
degree of M.Sc in Computer Science

in the
Faculty of Science
School of Computer Science and Engineering

August 2012

THE HEBREW UNIVERSITY OF JERUSALEM

Abstract

Faculty of Science
School of Computer Science and Engineering

Texture Synthesis for Particle Infused Textures

by Ohad Fried

The field of texture synthesis is a mature one, with many different techniques and algorithms to accommodate for various tasks. In this paper we zoom in on a specific type of texture family - one that we denote "textures with particles". For these textures we present novel ways of synthesis with controllability. We show that our results are equal to or better than previous works in terms of quality, while allowing for an artist to generate the texture with much greater control. We allow various particle traits to be manipulated, such as size, location and shape. The resulting textures can be used in various fields, such as computer gaming, maps, CAD and 3D modeling.

Acknowledgements

I would like to thank Prof. Dani Lischinski for putting up with all my questions and comments and for helping while this thesis work took a few unexpected turns. I would like to thank my parents for having me and for not bugging me too much. I would also like to thank my cat for scratching me when I needed it the most.

This work would have never been completed without the support of the loving HUJI family.

Contents

List of Figures

Dedicated to Niki

Chapter 1

Background

1.1 Texture Synthesis 101

The art of texture synthesis has been researched and perfected in the last two decades. Before one explains how to enhance existing texture synthesis techniques, one has to explain what a texture is and what we aim to do when synthesizing textures.

1.1.1 What Do We Mean When We Say "Texture"?

The term "texture" has been defined in previous papers in different yet similar manners. For example, Wei et al. [1] give both a general interpretation used in graphics and a more specific definition relevant to synthesis.

> "...the term texture in computer graphics can refer to an image containing arbitrary patterns."
>
> "In other contexts, such as ordinary English as well as specific research fields including computer vision and image processing, textures are usually referred to as visual or tactile surfaces composed of repeating patterns, such as a fabric."

The main two features of a texture are randomness and similarity of different regions. By randomness we mean that there are no structured features or meaningful objects in the texture. Images of sand or wood contain such randomness - the lines, colors and other image elements suggest no structure. A building, for example, will not be considered a classic texture, since structured objects such as windows appear in regular locations (figure 1.1). It is important to note that the classic "texture" definition is being stretched and manipulated as more and more images are treated as textures for some purposes. For example - near regular textures [2] break the classic randomness assumption.

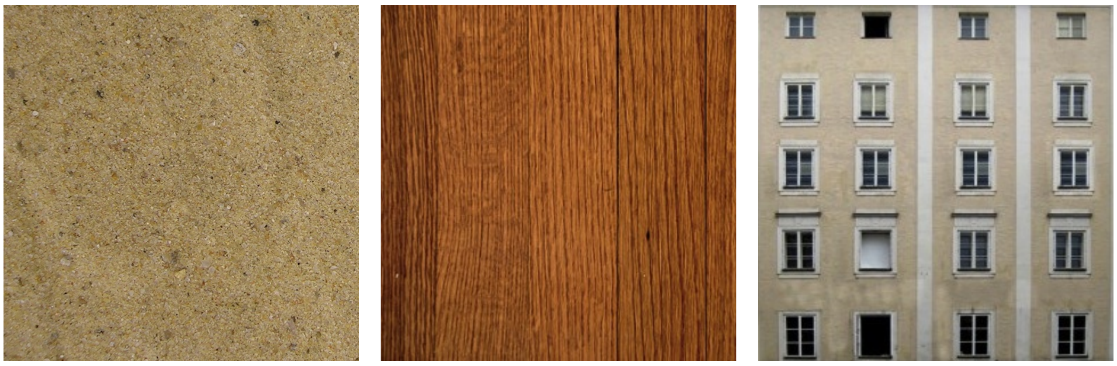

FIGURE 1.1: Several examples for textures. From left to right: sand, wood, building facade. The sand and wood are more "classical" textures, while the building defies the standard definitions.

Different regions in a texture are expected to be similar in their properties. When one looks at the fur of a cheetah, no matter where you look, it "feels" like the same material. This intuitive feeling is created by the fact that different regions of the texture are similar in terms of color values, types of edges and the relationship between different elements (see figure 1.2 for an example). This rule also has exceptions. If the captured image has some global illuminating effect, it will break the similarity assumption (more on this to come). Some works specifically handle textures where there are elements that break and interrupt the similarity [3].

This paper tries to tackle a specific type of textures. We will call such textures "textures with particles" or "particled textures". These diverse from the classic definition of a texture. For the purpose of this work, a texture is a classic texture, together with several elements spread on top of that texture. The classic texture will be denoted as background, while the elements will be called particles.

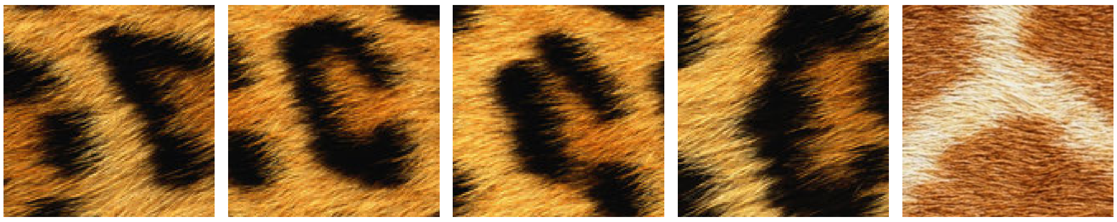

FIGURE 1.2: Several samples of patches from a cheetah texture, and one from a giraffe. While both are fur, it is very easy to spot the giraffe patch due to color, edge orientation and other properties.

1.1.2 Texture Synthesis Explained

The process of texture synthesis is defined by its output - an endless amount of images which, to a human observer, appear to be the desired texture. Note that we did not specify the input for the process, since this can vary as explained in section 1.2.1. One can think of a Turing-test inspired way to test the output of a texture synthesis process: if an image of the texture cannot be distinguished from a synthesized image by a human observer, the process is a success. While this is the holy grail, all current texture synthesis methods fail such a test. We try to achieve results which are close to the ground truth as possible.

It was said before that the process of texture synthesis aim to generate an "endless" amount of the desired texture. This is important. Some techniques will work well while creating a small patch of the texture, but will fail to create a larger patch. The need for large amounts of a specific texture is explained in section 1.1.3. Figure 1.3 shows some textures synthesized by various methods, such as [4] and [5].

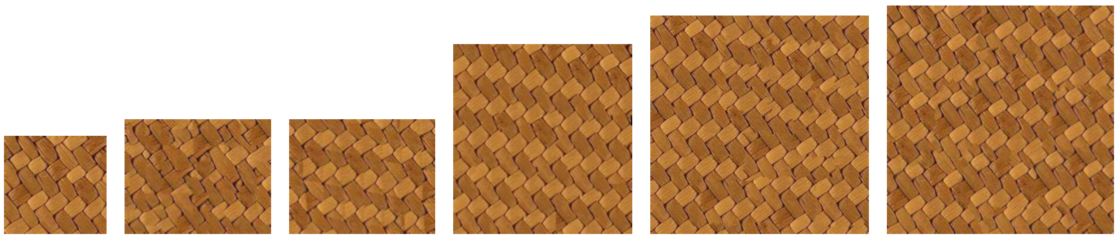

FIGURE 1.3: Example of various results of texture synthesis. From left to right: input, patch-based synthesis, regularized patch-based synthesis, near regular synthesis, image quilting and graph-cuts. Samples taken from [6]

1.1.3 What Is It Good For?

Textures appear as part of many different aspects of computer graphics and computational photography. They are so heavily used, that they received their own mechanisms (with the accompanied hardware) inside graphics chips. Textures are important for computer gaming, modeling, maps, artistic painting and more.

In computer games, textures are used to give details and interest to the "boring" polygons. Due to memory and processing constraints, complex objects are modeled using a small number of vertices, which only resemble the original object's geometry. The mechanism used to add the fine details to the model is via textures, which are painted on top of the objects. On some occasions, such textures are needed in infinite amounts. Consider, for example, walls in a 3D environment of a computer game. Many walls exist, thus a large amount of wall texture is needed. If the same texture image will be used throughout the game, the player will acquire a sense of familiarity with the specific wall patch, and notice that it appears in different locations. This, of course, breaks the realism of the game, since in real life each piece of wall is different. For such textures, the process of texture synthesis can produce the textures, along with the needed variation. Figure 1.4 shown several real world examples where on-the-fly texture synthesis could have improved the look of the game.

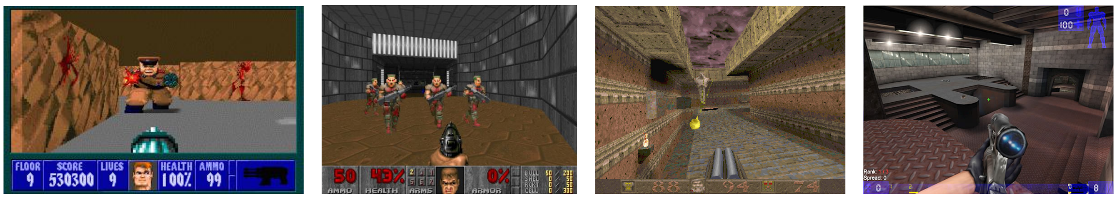

FIGURE 1.4: Screenshots from various classis first-person computer games. Notice the repeating patterns on the walls and floors, caused by the same texture tile being used several times. From left to right: Wolfenstein 3D, Doom, Quake, Unreal Tournement.

3D modeling is also a large consumer of textures. If a model is created in, for example, a CAD software, it is sometimes desirable to display it in its final form and not just as wireframe. This final form requires a texture. A marble statue will require a marble texture to be placed onto the geometry in order to look realistic. When comparing to computer games, textures for 3D modeling will usually need to be of higher quality, since they are expected to be viewed for a long time while static, as opposed to a texture within a high paced game which only flickers on the screen for a few seconds. On the other hand, these textures might not be needed in huge amounts, since the size of the model is bounded and known in advance.

Realistic mapping is a very “hot” subject which also requires a great deal of textures. Nowadays people are bored by “regular” maps that only show the roads in a cartoonish manner. They want to see the world as is, either through satellite images or, lately, as a 3D realistic model of the world. Cutting edge mapping applications allow the user to “fly” above the city, viewing roads, parks, buildings and other structures as they appear in the real world. Such an application can be considered a massive 3D model of the world. As such - it requires textures. In order to look realistic in this bird’s eye view, a patch of grass will need to be textured by grass, a wall needs to look as if it comprises of bricks and so on. This is an extension of the 3D modeling problem which requires huge amounts of textures for common materials found in our world.

Texture synthesis can also be used to create sophisticated painting tools. Instead of the usual pen or pencil tools common in drawing applications, consider a “grass” tool or a “sand” tool. On the fly synthesis will allow an artist to fill an area with the wanted material, thus providing an easy way to create novel digital images. Texture by numbers [7] is a good example of such a technique.

The above are just a few fields in which textures appear and texture synthesis proves to be useful. The list is by no way extensive, and meant only as a teaser to the importance of good texture synthesis methods.

1.2 Types of Texture Synthesis

The field of texture synthesis is a broad field, and many different approaches exist to solve the problem. This section lists the main types of texture synthesis techniques. It is not a “related work” section (for that, see chapter 3). The aim of this section is to name categories of texture synthesis techniques, allowing the reader to quickly match a given technique with its properties.

1.2.1 Synthesis by Model Vs. Synthesis from Exemplars

One way to partition the different texture synthesis techniques is according to the input for the process. One approach is to create a mathematical model for the texture. The input for such approaches is either model parameters or no input at all. Another approach is synthesis from exemplars, in which a small sample image is given. The purpose of the synthesis process is to create larger (infinite) output which resembles the input in properties.

Synthesis by Model

Synthesis by model requires the researcher to “understand” the texture and create a specific model tailored to the specific texture. These techniques have several inherent advantages and disadvantages.

Model based techniques are compact. There is no need for large samples of the wanted textures. All that is needed is an algorithm, which is a relatively short piece of text, which is compiled into a few kilobytes of code. A good example for this advantage is highlighted in *.kkrieger* which is a first person shooter with a footprint of only 97,280 bytes due to the fact that all textures are created mathematically.

Model based techniques will sometimes require parameters from the user. These parameters allow control on the generated texture. Usually, these parameters are intuitive to understand since they are part of the texture model. An artist can visualize and anticipate the result of tweaking these parameters.

The model is not easy to create. It requires a specialist to understand the texture and hypothesize regarding the appropriate model for synthesis. Moreover, the model is very specific, and each texture will require a different model. This means that synthesis by model might not be cost effective. It requires a lot of investment in order to achieve the ability to generate one single type of texture.

For a complex texture, the underlying "true" model is also complex. When trying to create a feasible model, it is simplified, thus creating results which are not up to par with the real texture. This means that model based techniques are only suitable for textures where the model can be understood and implemented, which is not a simple requirement.

Synthesis from Exemplars

The disadvantages of model based techniques (see section 1.2.1) prompted the field to move more towards example based techniques. These techniques do not try to understand a texture type, they simply strive to make more of the same. Given an input texture the goal of such techniques is to create more of that texture.

One major advantage is that one method can work for various types of textures. All that needs to be changed is the input texture, while the algorithm remains the same. Some methods (as the one in this paper) limit themselves to a specific texture family, in order to improve performance or controllability. This is still broader than the model based method - a texture family consists of an infinite number of texture types.

A disadvantage of these techniques is the fact that one never "understands" the textures. If the output is all that is needed, this is not a problem. But in the academic sense, such techniques sometimes convey a blackbox feeling - where the output was obtained through some "magic" without real understanding.

Another disadvantage is that an input is needed. An image of high quality needs to be obtained, taken from the right angle (usually straight, with no perspective effects) and with the correct lighting (without any global changes to the light). This is not a major disadvantage in today's reality where digital cameras are abundant and of high quality.

When it comes to runtime, the two categories are hard to compare. Some example based methods are realtime, while others can take minutes to produce a single image. Most model based methods are quite fast, but some models are very complex and synthesis again can take several minutes.

1.2.2 From Pixel to Patch

Within the world of sample based synthesis, all methods can be roughly divided into pixel based and patch based. The first methods that popped up were pixel based, which means that the output image is created pixel-by-pixel. Two adjacent pixels might be taken from completely different areas of the input image. These methods proved to be appropriate for textures where randomness is very prominent. When no obvious sub-structures can be seen by a human observer, these methods performed very well. When trying to deal with textures that contain some form of structure, the pixel based methods create artifacts. The output seems too random and noisy, and can be clearly distinguished from the original.

A simple observation regarding the synthesis process is that adjacent pixels in the input will usually be suitable as adjacent pixels in the output. This observation leads the way to patch based approaches. Instead of copying the pixels to the output image one by one, a patch is copied. The patches are usually an NxM rectangle (we diverse from rectangle patches in this work. More on this later on). The advantage of using patch based methods is that they succeed in maintaining the small structural elements within the texture. The challenge lies in being able to stitch the patches together in a seamless manner, avoiding seams and other stitching artifacts. Another challenge is to avoid obvious repetitions - if the same patch with the same unique look will appear in many locations within the output image, the result will look artificial.

1.2.3 Controllable Synthesis

Another way to partition the world of texture synthesis techniques is according to the amount of control these techniques allow. Different methods can vary from no control at all, through minor control over a few unintuitive parameters, to very elaborate and intuitive control over the result. There is no "best" level of control. Different methods are suitable for different needs.

When all that is needed is an endless amount of a specific texture, which matches the original in properties, having a system with no user interaction is an elegant solution. The user in this case does not want to tweak parameters and thresholds in order to create the result, since the wanted result is already specified - indistinguishable from the original.

In other cases control over the synthesis process is wanted and needed. Many parameters can be controlled. One can control the amount of randomness, the orientation of the synthesis, scale of structures that are maintained, perspective effects and more (an example of controllable synthesis can be found in [8]). The secret to a good controllable process is in the selection of the controlled parameters. These should be easily understandable and directly related to the wanted result. Bad control parameters are ambiguous thresholds that when tweaked produce unexpected results. Good controllable synthesis is enhanced by realtime performance, which enables the output to be controlled and tweaked in realtime. More on runtime in section 1.2.4.

1.2.4 Runtime

The spectrum of texture synthesis techniques spans many different runtime categories. Some techniques are realtime, allowing the results to be created in a few milliseconds. Other techniques require seconds or even minutes to generate a single image, while producing (arguably) better results. A notable example for realtime synthesis, which allows for rich user interaction, can be found in [8]. This method would not have been useful if it was not realtime. The user is presented with many sliders to play with. The output due to parameter manipulation is not always predictable, but the realtime nature of the method allows for an online adjustment workflow, where an artist simply tweaks the parameters and immediately views the result. A notable example for much slower high quality synthesis is by Zhang et al. [9]. While synthesis varies from 0.5 to 2 hours, it allows advanced results such as progressively variant textures and texture blending.

The method presented in this paper is flexible in terms of runtime. The background synthesis phase (section 4.2) can be either slow or fast, with higher or lower quality, according to the chosen synthesis algorithm. Particle synthesis (section 4.4) also varies. Simple size and orientation manipulation can be performed on the spot, while novel particle generation might take longer. The runtime can be tweaked according to the artists requirements from the generated texture and particles.

1.2.5 Textures In 3D

When dealing with 3D objects, synthesized textures will eventually find their way onto a 3D model. Texture mapping is the process of copying texture pixel values onto a model. When a flat texture is copied onto an elaborate geometry, artifacts arise. In this context, texture synthesis can be viewed as a solution to this problem. Instead of trying to map flat textures onto complex objects, we synthesize the texture directly onto the complex geometry, while taking it into account. The following subsections describe two fundamentally different approaches to synthesis in 3D.

Surface Texture Synthesis

Surface synthesis is the process of creating the texture directly on the surface, and not on a flat coordinate system. Normal synthesis will try to create a rectangular output image, with the standard x/y axes. Surface synthesis is essentially the same process, adjusted to work on a new coordinate system that matches the contour of the 3D shape. Surface texture synthesis can be thought of as a two step process - creating the coordinate system, followed by the synthesis process.

The coordinate system can be created in many ways. It is desirable that the coordinate system will match the shape's contour, since this reduces artifacts and makes the output texture more visually pleasing. Creating the coordinate system can either be a fully automatic process (according to salient points or normal directions) or a user assisted process. A user can specify the desired coordinate system in several key points on the surface of the 3D shape, and an automated process can extent this coordinate system to the entire shape. Such user assisted process can be viewed as another form of control over the synthesis process (other control methods were discussed in section 1.2.3). Different vector fields will yield textures in different orientations and with different spacial density. This paper will not discuss 3D surface modeling, but it will however deal with similar techniques in 2D in order to synthesize over non standard coordinate systems.

Solid Textures

A completely different approach used to texture 3D shapes is solid texturing. In this method, the output of the synthesis process is a 3D block. Synthesis is performed over 3 axis (and not 2 as in the standard synthesis process). The 3D texture block can be mapped directly onto a 3D shape. The downside of this approach is that it is "wasteful" in a sense. Many pixels are generated but never used. Consider for example a 3D shape whose bounding box is of size 100x100x100 pixels. A million pixels will be generated, while only a fraction of those will actually be located on the contour of the 3D shape, thus used in the output. On the other hand - if many different shapes need to be textured, they can all use the same 3D texture block, thus reducing the wastefulness.

It is important to note that usually solid textures are procedural. This helps in terms of memory usage - we do not need to store the entire 3D volume. Moreover, if constructed correctly, we do not even need to calculate the values for all pixels, only for those which are needed for the output. This in turn helps in alleviating the problem mentioned in the previous paragraph. See section 1.2.1 for more info on procedural texturing.

When using solid textures, the output will usually be quite eye pleasing. less texture distortions will occur, since the synthesis was done in 3D and not "folded" from 2D onto the 3D surface.

1.3 Result evaluation

It was already mentioned in this paper that some methods for synthesis are "good", "pleasing to the eye", "not as good", "bad" and so on. These terms are very subjective and, in a way, not very scientific. As in many other subfields of graphics and computational photography, evaluating the results of a texture synthesis process is not an easy task.

1.3.1 A Turing Test for Texture Synthesis

Consider the following task - generating a 512x512 image from a 64x64 sample of a texture. Assume that we also hold a 512x512 image of the texture as an absolute truth. Given the ground truth image I and a texture synthesis algorithm $Synt()$ the synthesis process can be marked as $O = Synt(RandPatch_{64x64}(I))$, where $RandPatch_{MxN}(I)$ indicates the process of selecting a random patch of size MxN from the input I. Now the texture synthesis process can be evaluated according to the difference between I and O. These two images are obviously different, but might not be distinguishable when it comes to the core properties of the texture. One can give both I and O to a human subject and ask which image is real and which is computer generated. If O cannot be spotted with statistical significance, we deem the synthesis process a success.

1.3.2 Automating the Quality Test

While section 1.3.1 presents a very natural way to determine if the output of a synthesis process is adequate, it has 2 major drawbacks:

- It requires a human observer.
- It outputs a good/bad binary output, and not a more fine grained result.

It is desirable to find an automatic method that produces a quantized measure of the quality of the synthesis result. Such a method must take into consideration the basic properties of textures such as randomness and repetitiveness as explained in section 1.1.1, together with coherence of the output.

One way to measure randomness would be to look for repeating patches within the output. If a large patch is repeated many times, the result should be considered less adequate. This is due to the fact that identical patches normally do not appear in natural textures. Different patches always vary to some extent. It is important to look at correctly sized patches. If the comparison window is too small, identical elements will be detected, but this is desirable and not a measure for bad textures (since in this small scall elements do repeat identically).

Repetitiveness can be measured according to equivalence of patch properties. While it is unwanted that two patches will be identical, it is very much desirable that they will be similar in properties. The properties one might measure are diverse. For example, a color histogram of the patch can be used. Another metric is the amount, orientation and connectedness of the edges within the patch. These properties and more are usually used in the synthesis process, and can also be used to evaluate the result.

Coherence is damaged the most when seams between patches are visible. The automatic process need to detect and penalize such seams. The seams should be searched in locations where they might appear (according to the specific synthesis method) - either on patch boundaries or throughout the output image.

It is the opinion of the authors that a good automated technique to measure the quality of a synthesized texture does not exist yet, and is a very promising direction for future work in the field.

1.3.3 Evaluation According to Purpose

It is important to remember the purpose of the output texture while evaluating its quality. A texture to be used in a fast-pace computer game is quite different in nature from a texture that will decorate a 3D shape within a computer generated still image. All the above techniques must take into account the purpose, putting more or less emphasis on certain output qualities.

1.4 Related Problems

The world of Texture Synthesis is closely related to other areas, most prominently to image completion and super-resolution. While it may not be obvious at first glance, many algorithms are similar or even identical in all these fields. Intuitively, one can think of an image completion task as the task of generating a texture in the target area (while usually using image patches from the very same image we are trying to fill in). Some methods for super-resolution are actually texture synthesis of high resolution textures, while using the low-resolution image as a visual cue.

1.4.1 Image Completion

An example for image completion that is closely coupled with texture synthesis can be seen in [10]. In this work, the authors use multiresolution texture synthesis in order to fill in a target area marked by the user. This allows for elimination of unwanted elements from a given image. The paper uses approximation techniques to get a coarse result, together with best fragment searches in order to turn the approximation into high quality results. All the search techniques and neighborhood matching algorithms should sound familiar to the texture synthesis savvy reader.

Another example is in the work of scene completion by Hays and Efros [11]. This work, at first glance, might seem unrelated. The authors take a large database of images and use it to replace marked areas of a target image. The synthesis here is neither pixel based nor patch base - the entire hole is replaced with a single patch from a target image. Nevertheless, the algorithm consists of best-match searches according to SSD of pixel values, and the new area is blended in using graph-cuts and poisson blending (see 4.5 on how this relates to this paper).

1.4.2 Super Resolution

It is sometimes desirable to enhance the resolution of an existing low resolution image. This allows, for example, image enlargement (without unwanted blurring) and is also important in order to make existing artwork compatible with many different screen resolutions (important for today's "retina display" shift). One sub-field of the super resolution problem considers the input image as the only input available (as opposed to using multiple similar images, a video sequence or some previously learned model). While it seems counter intuitive that an image can be enhanced in resolution while only using that single image, it is possible. The underlined assumption here is that image features exist within the single image in different scales, and one can enhance a feature that is low-resolution in the input using a high-resolution version of a similar feature. This in turn gives an opportunity to many super-resolution algorithms, which are very close in nature to texture synthesis methods. We look at the low-res image as a hint to a patch matching and stitching problem.

When using multiple images as input (or a video sequence) the problem becomes similar to texture synthesis from multiple examples. The same analogy holds, just with a more diverse input set. Some super-resolution results (along with other interesting results derived from the same method) can be seen in [7]. Also notable are [12] and (more recently) [13].

Chapter 2

Goals

The goal of this work is to provide a good texture synthesis technique, for a specific sub-family of textures. By "good" we mean that the results must be pleasing to the eye and that the synthesis must be controllable in a manner which is useful to an artist. We limit the solution to a specific sub-family of textures since this allows for a stronger prior, leading to better results. The following section will elaborate on the textures we want to deal with and on the synthesis goals.

2.1 Our Sub Problem - Textures with Particles

As explained in section 1.1.1, textures are hard to define. Depending on context, many different types of images can be considered to be textures. In this paper we are looking at a specific sub-family of textures, denoted from here on as "particled textures", "textures with particles" or "textures with elements".

Our textures hold the basic form of having some noticeable background which possesses all the usual properties of textures. On top of that background there are scattered elements, which can be identical or different from each other. If the elements can be grouped into several meaningful groups (e.g. flowers vs. fruit, or red fruit vs. blue fruit) we say that there are several different types of elements scattered on the background.

It is possible for the background to be completely covered with elements (we shall call this a null background). In this case only elements are visible. Also note that "normal" textures are a private case of our definition for particled textures - if there are zero particles the texture is normal.

In the following section we will try to explain particled textures through examples.

2.1.1 Examples

Figure 2.1 shows several examples for textures with particles. Even these simple examples show the various ways in which particles might differ. The left figure illustrates a relatively simple formation. The image consists of background which is fairly similar throughout the image. On the background there are "particles" - the holes in the material. Notice that in this image all the holes are identical and are located on a grid. When trying to synthesize such a texture, it will not look natural if the grid formation is broken. On the other hand - this regularity also creates an opportunity. With the correct controls given to an artist, an output texture might have a new hole formation which is not similar to the original, but is what is needed by the artist.

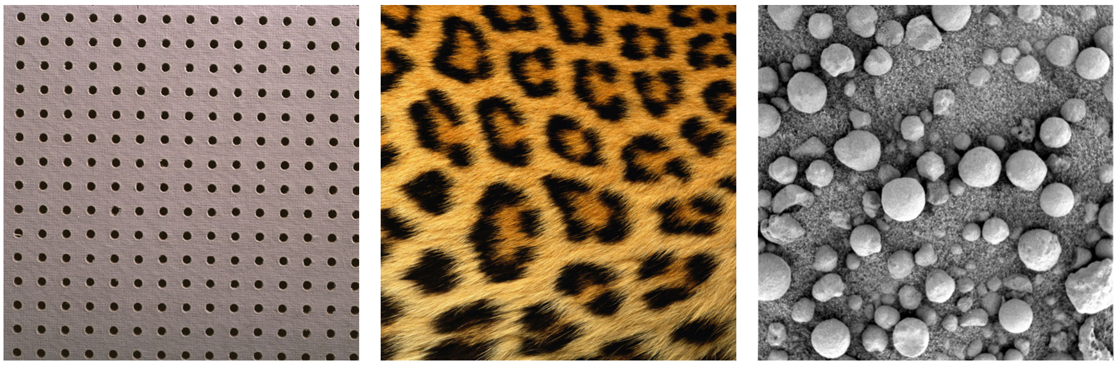

FIGURE 2.1: Various textures that contain particles. The particles may be in regular locations (left), they may be different according to their global position (middle) and they may vary in location and scale (right).

The middle image in 2.1 shows some more challenges. The background suffers a global difference in illumination and appearance. While illumination can be corrected, the fact that the length of hair changes is more challenging. When trying to synthesize such a texture, the background must be synthesized while taking global location into account. Also, the particles change according to location - from big particles with holes in the middle, to small black dots.

The rightmost image in 2.1 shows that particles within a texture may vary greatly. While it is obvious to a human observer that the image consists of sand and rocks, the rocks exhibit great variety. There is a notable difference in size, color and shape, that might make particle extraction a very hard task. Also notice that the image is in grayscale and intensity values alone cannot distinguish between background and particles, since there are various degrees of gray which consist the sand, and the same shades also consist the rocks.

Throughout this paper, particles will be marked using a particle mask - a binary image that indicate which pixels are considered background and which are a particle. Two possible extensions for such a binary image are using more than 2 values to represent several particle types, and using fractions to represent pixels which are not all-background or all-particle, but a mixture of the two.

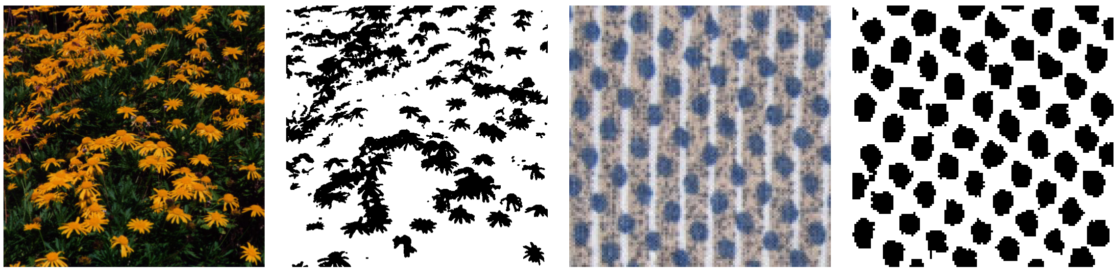

FIGURE 2.2: Textures and the corresponding particle masks. From left to right: flowers with a green background, mask for the flower image, a pillowcase, mask for the pillowcase. Notice that the pillowcase has several particle types. The displayed mask is only one possibility for a reasonable mask.

Figure 2.2 shows images and their corresponding masks. In this paper we will discuss the acquisition of such masks and how we use them in the synthesis process. Mask acquisition is not a trivial process, and is closely related to the classical "image segmentation" task. In the figure, particle are marked in black (0 or $false$ values) while the background is marked in white (1 or $true$). We will keep this convention throughout the paper. The shown masks were extracted using a simple color thresholding procedure. Later on we will discuss various methods to create the particle mask.

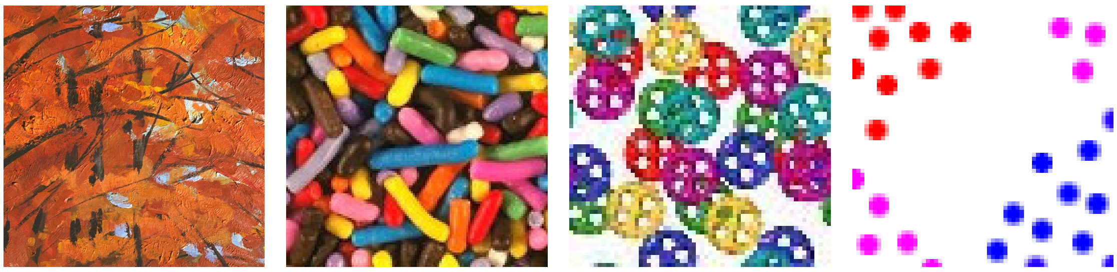

FIGURE 2.3: The same image can contain several different particle families. It is desirable to detect such families and correctly label each particle according to its type. Left - a painting. Brushstrokes are particles. Differently colored strokes should be detected as different particles. Middle left - each candy color can be considered a different particle type. Note that this particular image is unique in the sense that it contains no background, only particles. Middle right - Buttons of the same color belong to the same family. Right - three different particle types exist in this image - red dots, blue dots and pink dots.

Figure 2.3 gives several examples for textures with more than one group of particles. In the two rightmost images, there are particles with several distinct colors. While all dots might be considered the same particle type, for some applications it is desirable to treat them as three different families: red, blue and pink dots. The same goes for the buttons. Notice that this choice is somewhat arbitrary and should be specified by an artist according to the application. For example, it might be desirable to treat yellow buttons in a special manner - yielding only two particle families: yellow buttons and all other colors.

The candy image (second from the left) is similar in properties to the button image, with one major difference - there is no visible background. Again, this image can be interpreted in many ways: having a null background and 1-8 particle families, or having some background (for example, all candy except for blue candy).

The leftmost image in figure 2.3 is a paining, and different stroke types can be treated as different texture particle families. Thus 3 families exist (dark strokes, white strokes, light orange strokes) over a dark orange background.

For completeness, it is important to note that textures without particles also fit our model. These textures can be treated as particled textures with zero particles. While this shows that our model is a superset of the standard model, processing such textures with a particle oriented pipeline might prove to be wasteful. Simpler techniques can be used for “normal” textures that do not contain any particles.

2.1.2 Why are these important?...

While textures in general are very important (see section 1.1.3), it does not automatically imply that every sub-family of textures is also of interest. The sub-family might not be abundant enough or diverse enough to justify its own algorithms and methods. It is the aim of this section to justify the classification of particled textures as an important texture family.

Textures with particles are abundant

We have already shown many examples for textures with particles. Flowers on grass, spots on fur, strokes on canvas and rocks on sand are all manifestations of textures with particles. It is the authors’ claim that not only these textures are abundant, they even appear more often than “normal” textures in natural settings. It is very hard to find a “clean” texture patch, and much effort is put into acquiring clean texture images. In nature, it is more common to find objects placed on top of surfaces, as opposed to clean surfaces, and materials will usually consist of several sub materials or different orientations of matter. For that reason, textures with particles are abundant.

Textures with particles are diverse

Being abundant is not enough. If all particled textures would have been essentially the same texture, the problem would have been less interesting. Luckily, the family of textures with particles is extremely diverse. The diversity manifests itself in the amount and shape of the particles, the properties of the background and the actual materials and object that make up the texture.

Textures with particles require added control

Creating better texture synthesis results is a challenge and an important task. Creating more control over the synthesis process can prove to be even more interesting. Artists nowadays not only require "better", they also require the result to be exactly as they imagined it. Particled textures hold a great opportunity in terms of control. Splitting the texture into background and elements allows for intuitive element manipulation. Section 2.3 discusses further the control goals in this project.

2.1.3 Mathematical Model

Texture synthesis is often modeled as a Markov Random Field. The synthesis process acts as an MRF in the sense that a synthesized pixel or patch is only dependent on its surrounding. Given the neighborhood pixels, we get some probability for the given pixel values in the currently synthesized location. Moreover, consider an output image where some of the pixels were already synthesized, creating several distinct "holes". These holes, according to the MRF assumption, are independent, since we already know all the output pixel values for pixels that create a strict separation between the holes (this can be viewed as a global markov property). For an illustration, see figure 2.4.

When dealing with particled textures the MRF model still applies. If classical synthesis is applied, the standard MRF analogy is valid with no changes. For our method (as explained later), the background synthesis phase is strictly an MRF. Particle manipulation is a more controllable process that is coupled with user input. Some phases during the particle manipulation phase (most notable - particle enhancement after resizing, see section 4.4.1) again abide to the MRF assumptions.

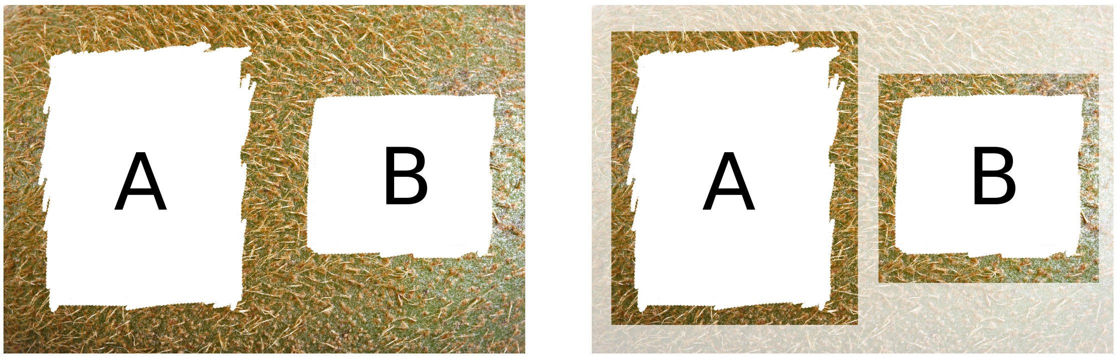

FIGURE 2.4: Texture synthesis is often referred to as a process that satisfies the Markov Random Field properties. Left: the image shows a partial output image of a kiwi skin. The white area marked as A is independent of the white area marked as B, since they are separated by already-determined pixels. Right: during the synthesis process, only the highlighted pixels will be used for patch determination. It is clear from the image that the synthesis of the two patches is independent.

2.2 Synthesis

In terms of synthesis, the goals of this project are quite straightforward. Existing texture synthesis techniques perform reasonably well, and the aim is to at least match such existing techniques. Every solution that will perform as good or better than current techniques is reasonable. The reason that equal performance is considered acceptable is because more control will be introduced into the synthesis. Results with equal quality and better control over the end result will be an improvement.

2.3 Giving Control

As hinted before, a major part of this work is about control. We not only desire to synthesize the texture, we also want to closely control the appearance of the output. Such control will be over the particles within the texture. The following subsections describe the parameters we wish to control.

2.3.1 Size

Variation of size is a very natural and useful manipulation. To go back to the previously given examples, one can now create sand with rocks of a specific size or size range, one can make the flowers on the grass unnaturally tiny or the spots on the fur unnaturally large. The artist will be given two parameters to play with - average size (bounding box diagonal or pixel area) and variance. More elaborate controls might allow for size variation according to x/y location (useful for perspective effects) or different size/variance parameters in different areas of the image.

It is desirable for the result to look pleasing, even when the particle is greatly enlarged compared to the original. In order to do so, extra algorithms are needed as a “fix phase” for the generated particle (see section 4.4).

2.3.2 Location

The location of the particles should also be manipulable. We would like to be able to specify where the particles will be generated and how densely. In terms of UI, there are several possible ways to specify particle location.

The user can “paint” particles wherever she wants. In a sense, this is like a particle brush, and particles will appear interactively on the output image according to the input strokes. This is very useful to create effects such as the ones demonstrated in the banner images of both [5] and [14].

Another option is for the user to specify density and randomness, when randomness here refers to the amount in which the particle locations differ from a completely regular grid. Randomness of zero will place all elements on a rectangular grid. The grid resolution will be determined according to the density parameter.

Particle locations can also follow a given density field. Each particle location will be sampled according to the field, resulting in an image that resembles the field yet displays randomness (due to the stochastic sampling process) which is pleasing to the eye.

There are many more possible ways to allow for the user to specify particle location. All are implementable, and rely on the underlying ability to place a particle wherever we wish on the synthesized background.

2.3.3 Shape

The shape of a particle should be manipulable. An artist might want to deform an existing particle, while keeping its “feeling”. This implies a transformation from the original particle shape to the newly specified shape, along with needed correction due to unwanted deformations.

In this paper when we talk about shape manipulation, we refer to the process of changing the shape of a single particle. For a discussion about creating brand new particles or mixing together a group of existing particles, see section 2.3.4.

2.3.4 Novel Particles

One of the main problems with texture synthesis from examples arises when the same input patch is used in several locations of the output in a way which is obvious. When a prominent feature of the image is repeated, it makes the output feel unnatural, since in real life no two elements are the same. This problem is emphasized when we deal with particled textures. Particles are, by definition, prominent texture features. If the same particle is used in many locations of the output, the result will not look natural. Even if the particle is rotated and resized, it might still be obvious that the same particle was duplicated.

Since the input image has a finite number of particles, one must be able to create novel particles that do not appear in the input. Two different approaches can be taken for the process of novel particle creation:

- **Mask creation followed by texture synthesis.** One can think of novel particle creation as a two step process. The first step is to decide upon the mask of the new particle. This can either be specified by the user, randomly generated, or a combination of two existing masks ([15] gives one method to combine two binary images). After a new mask is acquired, one needs to synthesize the pixels for the new element. This can be done with standard pixel based techniques, while changing the axis from the standard x/y to follow the contour or backbone of the particle. An example of mask manipulation followed by texture synthesis can be seen in [9].

- **Combining two or more existing particles.** This method is a one stage approach. We wish to take two or more existing particles and combine them in a seamless manner. We need to find good seams within the particle so that the hybrid particle will look natural. [16] gives a good method to create particle hybrids. The downside is that particles need to be well aligned for this method to work. Making the method work with unaligned and not so similar particles is a good direction for future work.

2.4 Runtime Goals

This project gives much more emphasis on result quality than on fast computation time. The logic being that we first need to find a good method that works on our textures, and only later (as future work) can try to improve the performance of our method. Our goal in terms of runtime was for the algorithm to perform fast enough to make it useable. This implies realtime performance for any interactive parts, while allowing slower performance for purely automatic parts.

Chapter 3

Related Work

The field of texture synthesis is quite mature, with many articles and methods. In this chapter we will try to point out a few articles which are closely related to our goals, challenges and algorithms. This is not meant to be an overview of published work in the field of texture synthesis. For a very good overview on example based texture synthesis, see [1].

In their "Image Analogies" paper [7], Hertzmann et al. allow the transfer of a relationship between two input images onto a third image, in order to create the analogous output image. The goal is for image B' to relate to image B the same as A' relates to A, when A, A' and B are given as input and B' is the desired output. The algorithm uses ANN search [17] over multiscale descriptors to find the best matching target pixel. It uses a controllable parameter to decide between good approximation (best patch match) and good coherence (close pixels remain close). This general setting allows for various applications, including texture synthesis and super-resolution. For texture synthesis the inputs A and B are left blank, thus instructing the framework to use pixels from A' to create a new image B'. This approach is advantageous because of the tunable parameter that prevents inconsistencies in the output (due to hectic synthesis that takes each pixel from a different area of the input). That being said, the method is still pixel based and not patch based (although multiscale), thus making complex textures with features harder to synthesize successfully.

Ashikhmin [14] was among the first to use entire patches from the input image (and not single pixels) in order to produce a more coherent result. His algorithm is faster than image analogies, but suffers from obvious horizontal lines in the output image. These lines result from the fact that once a patch is determined, all pixels up to the edge of the input image are taken. Once this edge is reaches - a visible seam is introduced onto the output image.

Several methods (such as [4, 5]) use overlapping patches during the synthesis process. The overlapping area is examined and a best-seam is calculated (either by dynamic programming in [4] or graph-cuts in [5]). These methods are very similar to our background synthesis phase (section 4.2). To be exact, we use a variation on the graph-cut technique. Instead of using rectangular patches on a regular grid, we allow for patches of arbitrary size and location. This makes the synthesis process more computationally intensive, but produces better results.

In "Self-Similarity Based Texture Editing" [18], the emphasis is on editing an existing texture and not on creating a novel one. The approach entails the user editing a single location within a texture that contains a repeating pattern, and the framework duplicates the change over the entire texture. Moreover, they allow for some regions to be emphasized, effectively supplying a way to resize texture elements. Although the element resize ability sounds close to ours, the two methods are fundamentally very different. Brooks et al. do not separate elements from background and thus cannot manipulate elements in a way different than a simple resize. Also, the resizing is essentially zooming in on specific pixels, which cause artifacts and only suitable for simple particles (they improve this by a second texture synthesis phase, but this fix is also very limited).

In 2003 Zhang et al. showed a very nice technique for synthesizing progressively varying textures [9]. The technique uses either a distortion field or a manipulated texton map to guide the variation of the synthesis. The texton map approach resembles ours in the sense that elements are separated using a mask, which is then manipulated and used to control the synthesis. This approach allows for variation in size and location of particles. It does not allow individual particles to display great variance, since only two or three channel masks are used. More importantly, the synthesis in Zhang et al. is pixel based and not patch based, which allows them to work only on a very limited (simple) set of textures. For more complex input images, patch based methods (as is ours) must be used to increase output coherence. The same limitation applies for "Parallel Controllable Texture Synthesis" [8]. This is a very impressive GPU accelerated synthesis technique that reaches realtime performance. The synthesis is controllable in aspects such as amount of multiscale randomness, and also allows for feature drag-and-drop. In the words of the authors, the main disadvantage again lies in the pixel based nature of the method:

> "The main weakness of our approach is the well known drawback of neighborhood-based per-pixel synthesis: it performs poorly on textures with semantic structures not captured by small neighborhoods."

"Appearance-Space Texture Synthesis" [19] builds on "Parallel Controllable Texture Synthesis" and improves the results by using richer feature vectors (plus dimensionality reduction) for the synthesis. As a pixel-based technique, it suffers the same drawbacks.

In their work on near regular textures [2], Liu et al. present a framework for detecting semi-regularity in textures and use it for texture manipulation. Their work allows variation in two dimensions - position regularity and difference from mean color value. This color variation and location variation, while impressive, relies heavily on the fact that the input image is near-regular. In a sense their paper is one that zooms in on a specific sub-family of textures, in order to produce better results. Our paper is the same, but we tackle a different family (particled textures). Notice that the two families are not disjoint - some textures belong to both. Even for those, the possible outputs differ between the two methods. We, for example, can take a texture with elements in near regular locations and create from it a texture with elements in arbitrary locations.

While most state-of-the-art techniques for example based texture synthesis use patch based synthesis, "Texture Optimization for Example-based Synthesis" took a different approach [20]. They treat the problem as an energy problem and perform global optimizations via an EM like algorithm. They also provide control over the result in the form of a flow field - the texture is synthesized with some warping. This method is unable to control only the background / particles of the texture, since the optimization is global. It would be interesting to adapt warp field synthesis to our patch based technique.

In "Detail preserving shape deformation in image editing" [21] Fang et al. allow the user to draw lines over a feature of the input and manipulate the lines in order to create a novel image. The manipulation causes a warping effect, followed by graph-cut texture synthesis and poison blending. This process resembles our algorithm for particle resizing (section 4.4.1). It would be interesting to implement their full solution to allow for more elaborate particle manipulations. Risser et al. [16] synthesize output hybrids given a few input exemplars. The downside of their method is that the exemplars need to resemble each other and be aligned in order to produce convincing results. Lepage et at. [22] created a nice framework to process materials and separate them into layers according to user scribbles. It would be interesting to apply our methods on their material separation results.

The work which probably resembles ours the most, both in name and in contents, is the "Texture Particles" paper by Dischler from 2002 [23]. The article is focused on the same sub-family of textures we try to tackle. The main differences between our approach and "Texture Particles" is in the image segmentation and synthesis control.

When it comes to image segmentation, Dischler uses two approaches:

1. ”A brute-force scissoring technique similar to the one used by the lapped textures”. Is plain words, this means that the user manually outlines image region using a commercial drawing package. This outlining process is facilitated by gradient-seeking tools. The paper indicates that the technique ”is inappropriate for textures characterized by a lot of small features, since it might require too much work from the user”. That is why they also introduce method 2.

2. Gaussian filter (for smoothing), plus simple RGB quantization for the actual segmentation.

Our segmentation technique (see sections 4.1 and 4.3) is more sophisticated than the RGB quantization yet requires less user interaction than the scissoring technique.

When it comes to synthesis control, Dischler offers several “extra features” for the synthesis process:

1. Modifying density of specific element types to create different results.

2. Modifying the size of particles (simple resize, without extra algorithms).

3. “Particle mixing” - synthesize the whole image with several element types, then drop some elements (this is mixing in the sense that elements from 2 sources are synthesized on a single image, not in the sense that new particles are created from several particle examples).

Since the above is not a major part of Dischler’s solution, we are able to greatly improve on previous results. We allow density modification, along with other option for particle location manipulation, as specified in 2.3.2. We also allow for size modification, yet our method is not a simple resizing filter. We use texture synthesis and super-resolution techniques to create visually pleasing particles, even for very small or very large resize factors (see section 4.4). Instead of mixing particles in the Dischler sense, which simply means borrowing particles from one image to be synthesized on a different image, we allow for novel particles to be created (section 2.3.3).

Chapter 4

The Pipeline

In this chapter we will present our pipeline for controllable synthesis of textures with particles. Each stage of the pipeline is self contained. Different algorithms can be swapped in and out as needed. Some of the steps are completely novel, while others are an implementation of a well known method, adapted to our needs. We will start off by explaining about background separation.

4.1 Background Separation

A particle texture contains, by definition, one background element and several particle groups. We would like to use the background element to produce an endless amount of background texture, onto which we will add particles. In order to create the background, we first need to know which parts of the input image are background pixels. We will denote the process of identifying background pixels as "background separation". Some examples of textures with a corresponding background mask can be seen in figure 2.2.

It is important to note that the background mask does not have to be comprehensive. We do not need to detect *all* background pixels, we only need to detect enough so that the synthesis result will look adequate. This makes the process faster and easier than particle separation (as explained in section 4.3).

4.1.1 Background Separation via Color Difference

A straightforward method to achieve background separation is according to color values. The algorithm is as follows:

1. The user clicks on different texture parts. At least two clicks are required (and usually suffice) - one on background area and one on particle area.

2. The image is blurred to prevent artifacts due to high frequency noise.

3. Each MxN patch of the image (here M and N are fixed constants) is compared against the marked background patch and particle patch. We use a simple 2-norm of the difference between pixel values. For example, the background score is calculated according to the following formula:

$$backgroundScore = \sqrt{\sum_i \sum_j (Patch_{MxN}(I)_{ij} - BackgroundPatch_{ij})^2} \tag{4.1}$$

   And the particle score is calculated in a similar manner, swapping $BackgroundPatch$ with $ParticlePatch$. Matlab code for the score calculation can be found in section A.1.

4. Each pixel is marked as background if $backgroundScore < particleScore$, otherwise it is a particle pixel.

For the 3-category version, we calculate several particle scores. A pixel will belong to particle type 1 iff $particleScore1$ is the minimal out of all scores for that pixel. Sample input/output for this method can be seen in figure 4.1.

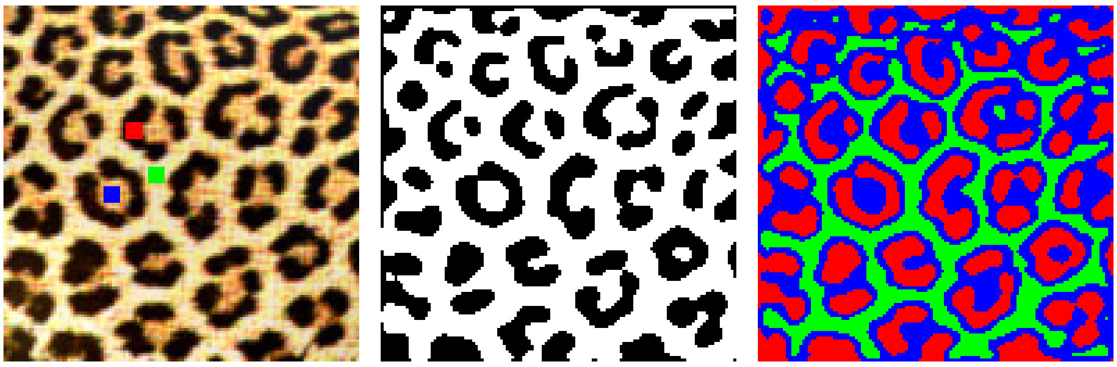

Figure 4.1: One possible way to extract background pixels. The user specifies an area which is background and an area which is particles (two clicks total, shown as greed and red squares on the left image) and the thresholding mechanism produces the result shown in the middle. More degrees of separation can be created. A separation into 3 categories is shown on the right image (user required to click on 3 locations: green, red and blue squares on left image).

4.1.2 Background Separation via Background Patch Similarity

The previous method used a comparison between two (or more) values. These values indicated how much a given patch resembles a background patch vs. how much it resembles a particle patch. The underline assumption is that if a patch resembles a particle more than the background, than it must be a particle. This assumption does not always hold true, especially when there are pixels which are neither pure background nor particle. In order to deal with such situations, another method had to be employed.

Background separation can be based solely on resemblance to a background sample. This will ensure that pixels we denote as background are indeed similar to the background, and not only "more similar" than to the particle sample. The method resembles the previous one in terms of the metric used. For each pixel we calculate the 2-norm of the difference between the surrounding patch and the sample background patch. Only this time we have no other score to compare it against. The solution in this case is to use a threshold value.

The problem with methods that rely on a threshold lies in the selection of such a value. Two main approaches exist: either set the threshold automatically or give simple interactive tools for the user to set the threshold. In this case there is no "correct" threshold, so an interactive UI was built that enables users to specify a good threshold value. The important part here is that the process is fast, meaning that the UI is updated in real-time. All a user has to do is play around with a single slider, until the result suits her needs. Figure 4.2 shows several different results for different threshold values.

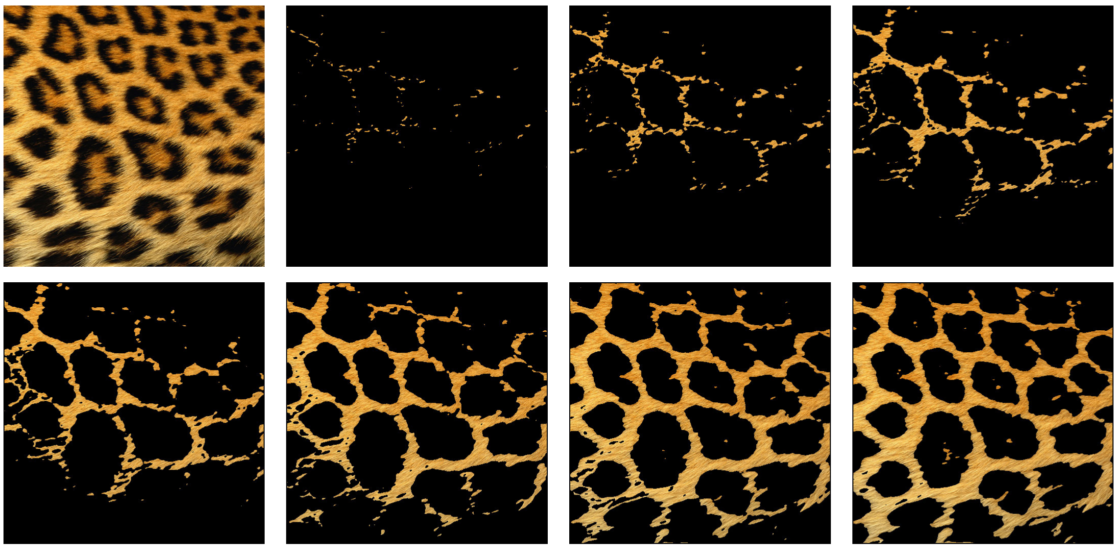

Figure 4.2: Background separation using a threshold. Top left: original image. Threshold values (left to right, top to bottom): 0.23, 0.35, 0.47, 0.59, 0.82, 1.06, 1.30

4.2 Background Synthesis

After the task of background separation is completed, we are free to start synthesizing the background image. The inputs for this stage are the texture sample and a background mask which specifies which of the pixels are considered background pixels. The output for this stage is an image (of arbitrary size) containing only the background, which does not contain any visible artifacts.

Our background synthesis technique is an extension of the technique introduced by Kwatra et al. [5] and the reader is advised to become familiar with their method. Many background synthesis techniques assume that the input is pure background, thus they are able to use whichever patch from the input in the synthesis process. Since we have pixels that we must not use (particle pixels) we cannot use any arbitrary patch. There have been works in the past that tackled such a problem, but they simply chose to only use rectangular patches which consist of pure background (for example, see [3]). The problem here is that the method is wasteful. There might be a rectangle which is almost pure background, but due to a single bad pixel such a rectangle will not be used in the synthesis process. This wastefulness takes its toll especially when the input texture is small.

Our approach is different. We allow for the patches that are used in the synthesis process to be of arbitrary shape. This allows us to use *all* background pixels in the synthesis process, but adds extra complexity to the algorithm. The main phases of our algorithm are as follows:

1. Iterate till completion:
   (a) Find the edges of the background generated thus far (or some seam that needs fixing).
   (b) Select a pixel along that edge.
   (c) Find a patch that best corresponds to the area surrounding the pixel (only compare valid areas, e.g. background pixels).
   (d) Merge in the new area (using graph-cuts as explained below).
   (e) Copy other pixels which are near the overlapping area.

Step (c) is done by simple pixel comparison, as explained in section 4.1.1. The only part that needs further explaining is how we merge in the new area and extra pixels.

4.2.1 Graph Cuts

Graph cuts are widely used for various partitioning problems. In the context of texture synthesis, Kwatra et al. [5] explain the technique well. The general idea is to create a graph from the pixels, thus transforming the best-seam problem into a min-cut problem. Best-seam refers to the problem of, given 2 input patches of the same size, deciding which pixels should be taken from which patch, so that the "seam" will not be visible. If two adjacent pixels are taken from two different patches, it is desirable for the pixels to resemble the corresponding pixels in the other patch. This means that the swap from the first patch to the second patch will look natural.

In order to translate the problem to a graph-cut problem, we construct the following graph (for now we will assume both patches are rectangular):

- Create a node for each pixel location.
- Create source/sink nodes.
- Connect the source to each location in the leftmost column of patch locations with weight ∞.
- Connect the sink to each location in the rightmost column of patch locations with weight ∞.
- Connect each pair of adjacent pixels s and t with weight $||Patch1(s) - Patch2(s)|| + ||Patch1(t) - Patch2(t)||$. The term $PatchX(y)$ refers to the color values of pixel y within patch X. The norm can change (we use 2-norm). Other more sophisticated cost functions can also be used.

After the graph had been constructed, we calculate min-cut over it. The minimal cut corresponds to the location where it is best to move from one patch to the other. Each side of the cut will be taken from a different input patch in order to generate the result.

Note that the above pseudo code is a simplified version. For a more elaborate version that also includes the possibility to fix "bad decisions" that were previously taken, see [5].

Until now, we assumed the patches are all rectangular, as in the original paper. Our novelty here is in allowing the patches to be of arbitrary shape. This in turn creates a problem: which pixels should be forced to be taken from which patch? While this was simple for rectangles (leftmost column and rightmost column), it is not trivial for arbitrary shapes.

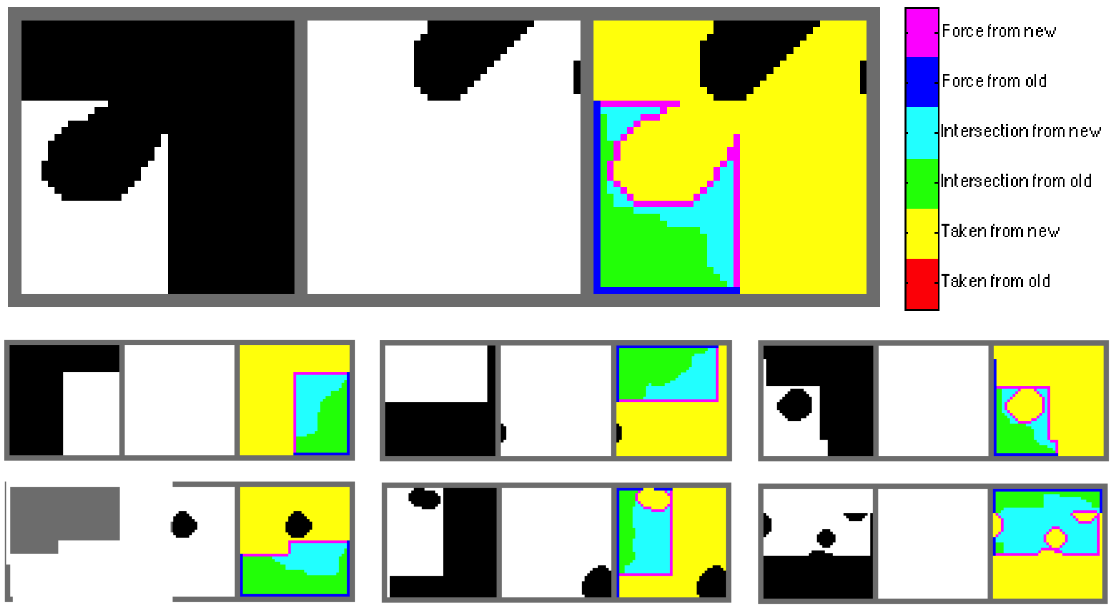

Figure 4.3: Different regions which are part of the graph-cut logic. 7 examples, each consists of 3 images. Left: mask of pixels which are currently in the region. Middle: mask of new patch that we wish to add to the region. Right: map of different pixel types. Purple - pixel connected to sink, thus forced to be taken from the new patch. Blue - pixel connected to source, thus forced to be taken from the old patch. Light blue - pixels taken from new patch, due to the decision of the graph-cut algorithm. Light green - pixels taken from the old patch, due to the decision of the graph-cut algorithm. Yellow - pixels which are only present in the new patch, thus taken from it. Red - pixels which are only present in the old patch, thus taken from it (do not exist due to the new patch selection criteria). Black - these pixels are not background pixels, thus left blank (will be filled in a different iteration of the algorithm).

Figure 4.3 shows a color coded explanation regarding the different pixel types in our algorithm. We need to carefully select which pixels we want to force to be taken from the already generated area and which from the new patch we wish to add. Since the patches are of arbitrary shape, we need to devise a way to select such pixels. The full algorithm can be viewed in section A.2. The main idea is that pixels forced to be taken from what we have synthesized thus far hold two traits:

- They are on the edge of the new mask.
- They are part of the intersection between the old mask and the new mask.

The complementary condition determines which pixels are forced to be taken from the new patch (the above is an over simplification, since there are some corner cases that can be viewed in section A.2).

4.3 Particle Separation

The task of particle separation, at first glance, seems very much like the task of background separation. We already showed algorithms to distinguish the background from the particles (section 4.1), so as a side effect we surely separated the particles, did we not?

The above is not accurate, due to the fact that for the process of background synthesis we need *some* background pixels and not *all* background pixels. This means that the complementary group of pixels does not consist only of particle pixels. In this section we will show better methods to accurately separate the particles from the background.

For some simpler textures, the basic approaches work well. Figure 4.4 shows particles detected according to color value thresholding, as explained in section 4.1.1.

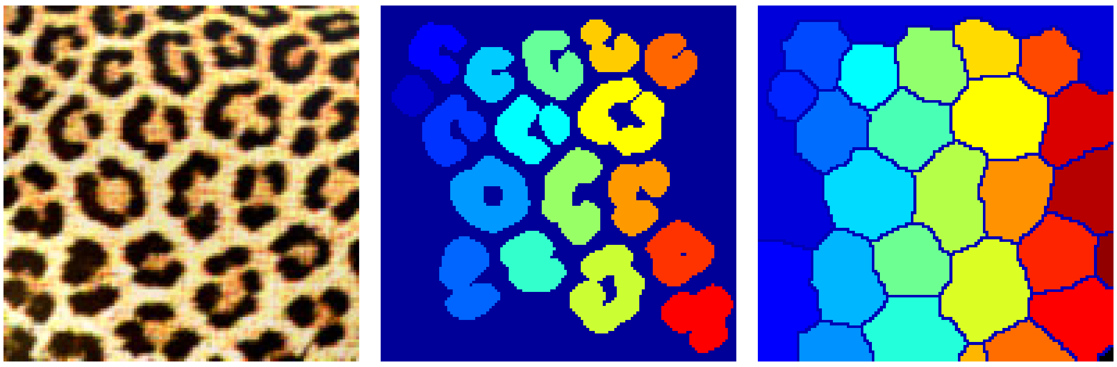

FIGURE 4.4: Left: original image. Middle: particles extracted using color difference thresholding. Right: maximal usable area for particle blending. Each cell contains a single particle together with as large border as possible, to facilitate blending into the background.

The left image shows the original texture. In the middle we can see the detected particles. Each particle is marked with a different color. The background is marked in dark blue. This output was produced after two steps of removing unsuitable particles - according to size and according to location. In terms of size, particles which are too large are removed (X percent of total pixels in image). This prevents the background from being detected as a particle, and also removes touching particles which are not separated correctly. In terms of location, particles which touch the edge of the image are also removed, since these are assumed to be cropped.

On the right we can see the output of a watershed transform over the image mask. This kind of particle separation is very useful if we do not need to extract the exact shape of the particle. Instead, we want large rims around the particles that can be used when blending the particle back onto the background. Note that this kind of mask is not useful if we need the exact shape, for example when we want to classify particles into groups or when we want to create novel particles from existing shapes. For this reason both exact masking (middle) and wide masking (right) are needed.

As explained before, some textures are too complex for simple color-based methods to work. In such cases we might get a particle mask which contains good quality particles, along with partial particles, fused particles and noise. One way to get high quality particles from such a mixture of results is to simply discard unwanted regions. Similar to what was shown above, we can discard according to size and location, but also according to shape (no holes, resembling some predetermined shape) and color differences within the particle (assuming particles are homogenous). This simple fix can yield high quality particles from difficult textures, at the cost of losing some of the particles (thus getting less diversity for the synthesis process).

Another method that proved useful is to separate elements using the background mask and a single user click on the wanted element. Since we know that the click is indeed on an element and that the (loosely created) background mask is indeed background, the problem becomes a classification problem. We classify all the uncertain pixels around the user input according to the two groups. The code for this method can be viewed in section A.3. Note that the function $GetFeatureVectors()$ can be arbitrary complex. A more complex feature vector will generally yield better results.

4.4 Particle Manipulation

In this work we are dealing with textures that have particles. Both input and output images have such particles. The particles from the input can simply be transferred as is onto the output, but usually we want to create more controllable diversity within the particles. This section specifies various methods in which we allow the manipulation of particles.

4.4.1 Size Manipulation

We give the artist an opportunity to change the size of particles. This is very useful if one wants to emphasize / de-emphasize the importance of particles within the image. Also, an artist can change the size of specific elements, giving focus to some area of the generated image.

A simple approach, as shown in [23], would be to resize the particles before adding them back onto the background image. The resizing can use, for example, bicubic interpolation in order to determine the pixel value in a given output location (or any other suitable interpolation). This method is suitable for resize factors which are close to 1, but fails when the particle is enlarged or scaled down by a large factor. In order to support large resize factors, we needed something better than a simple resize procedure.

We use a method that borrows from existing image upscaling methods (such as [24]) and present it here in the context of texture synthesis. The important thing to remember is that a resized particle will have much in common with an original particle at some zoom level (for example, if you zoom onto the contour of a rock, it does not matter if the rock is small or huge, the contour will have similar properties). We use this property to enhance upscaling results. The first step is to resize a particle using bicubic interpolation. The next step is to "texture synthesize" over the resized particle, using the particle as hint for the synthesis process. We look at patches and find the best matching patch (from all particles of the given texture). We use the center pixel of the matched patch as the new pixel value in our output. This pixel based texture synthesis can be seen as a correction phase for the resize stage. For some particle types, this approach improves the result, but is not good enough. For such complex particles we use an iterative approach, where we enlarge the particle by a small fraction in each iteration (and perform the correction phase). Combining a few of these small steps together yields the wanted resize factor. Results are shown in figure 4.5.

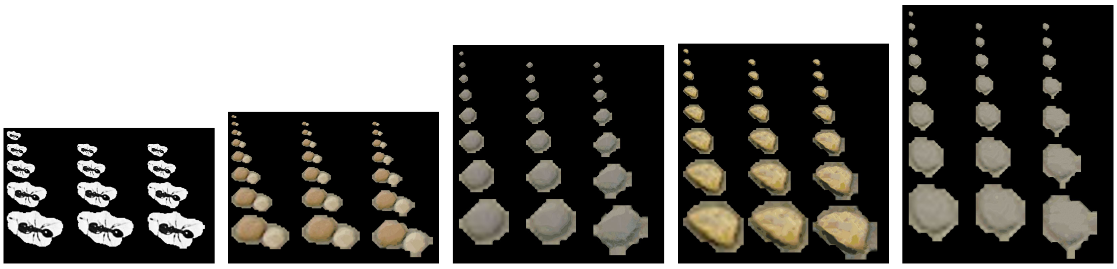

FIGURE 4.5: Particle resize. Original particle is shown in the top left of each image. Left column: bicubic interpolation. Middle column: bicubic + correction phase. Each particle is generated from the original. Right column: bicubic + correction. Each particle is generated from the previous level. Note that resize factors are exaggerated (up to 8x) and usually will be lower for real world scenarios.

4.4.2 Angle Manipulation

It is sometimes desirable to manipulate the angle of particles. More over, it is sometimes desirable to first understand the natural orientation of a particle, and later to manipulate such orientations. The process of angle manipulation consists of 2 steps:

1. **Finding the natural orientation of the particle.** Some particles have no such orientation or one that is not detectable according to shape. Others do have a main axis that determines the orientation. Consider images of ants or cars for example - the direction in which the particle is the longest is the main axis for that particle. We find the direction in which the particle is longest using PCA. A code snippet for aligning all particles according to their main axis can be seen in section A.4.

2. **Rotating particles according to the user's needs.** A user can specify an angle for each particle, for all particles or parameters for a distribution from which the angle will be sampled. Angle can also be correlated with other parameters, such as x/y location.

Figure 4.6 shows the result of controllable particle rotation.

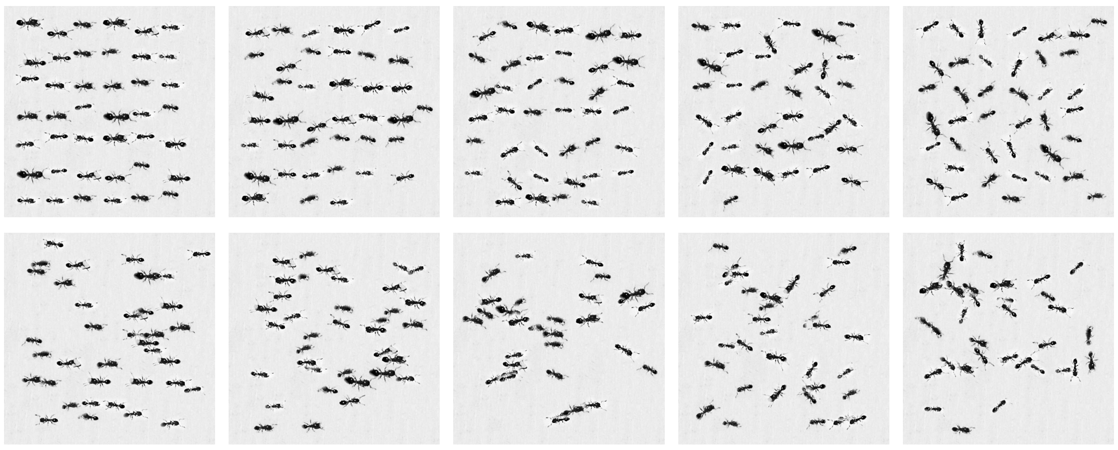

FIGURE 4.6: Synthesis of ants with various degrees of rotation. Top row: x/y locations are fixed on a grid. Bottom row: x/y locations are randomly perturbated away from the grid (stdev = 40). From left to right, rotational stdev increases (0, 10, 20, 30 and 40). Note that original image (not shown) had all ants in various orientation. The main axis of each particle had to be found and all particles were aligned according to it.

4.4.3 Location Manipulation

Each particle in the output image must have an assigned location. Such location can be set on a regular grid or perturbated away from such grid (figure 4.6). It can also be randomly generated with a density parameter that determines how close together are the particle (figure 4.7). Such a density parameter can be fixed for the whole image or changed for different areas. Also, particle locations can be sampled from a mask of valid locations or from a distribution function that determines the probability of a particle in each location (figure 4.8).

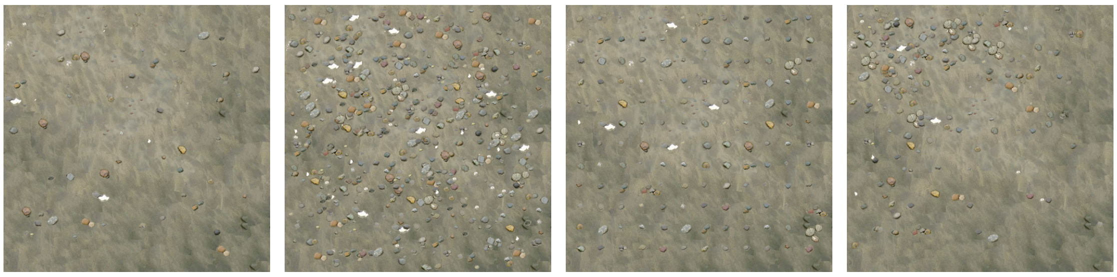

FIGURE 4.7: Several examples of generated textures with manipulated particle locations. Left: synthesis with sparse particles. Middle-left: synthesis with dense particles. Middle-right: particles are synthesized on a regular grid. Right: density is correlated with x/y location.
Only particles from the input were used, without creating novel particles (thus some repetition is visible). The manipulated parameters are x/y location and rotation (random rotation sampled from a gaussian). Input texture can be seen in figure 4.8

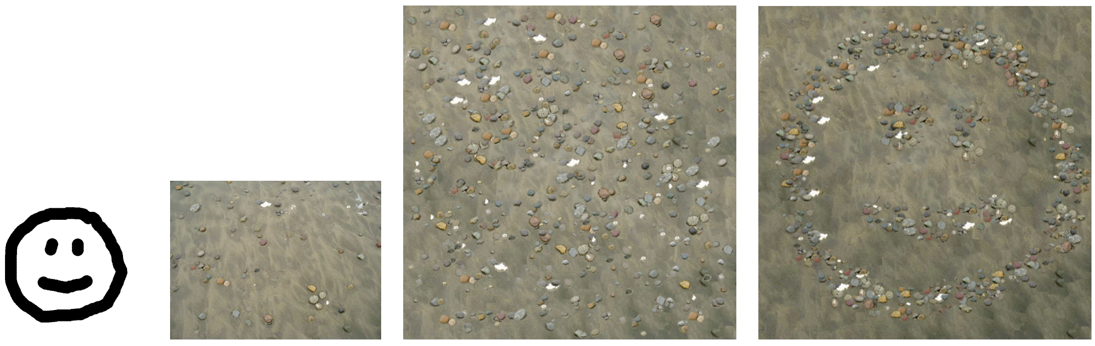

FIGURE 4.8: Left: mask specifying wanted output region. Middle-left: input texture. Middle-right: normal synthesis (with dense particle placement). Right: synthesis according to mask. Rocks form the shape of a smiley face.

4.4.4 Shape Manipulation

Up until now all particle manipulations kept the basic shape of the particle. Even if the particle mask was changed (during resize) it still was an exact copy of the original mask, in a different scale. It is sometimes desirable to create novel shapes for particles. Two basic methods were tested. The first involves an artist plotting the desired particle mask. This gives complete control, but can be a lot of work which is sometimes unnecessary. If all that is needed is the creation of novel shapes for particles, a more automatic approach can be employed. We take two existing particle shapes and interpolate them together using the method in [15].

After a new mask is created, one still needs to create the actual particle (e.g. pixel values). This can be done by either deforming an existing particle onto the new shape, or by synthesizing the new particle pixel by pixel, with axis which are aligned to the contour of the shape. If deformation is the chosen method, a second “fix” stage can be used, similar to the one described in section 4.4.1.

The creation of novel particles was tested by the authors, but did not reach maturity at the time of writing this paper. It is one of the major directions for future work deriving from this paper.

4.5 Background-Particle Composition

After a proper background was synthesized and all particles are ready to be added, we need to seamlessly add each particle to its appointed location. The simplest approach is to paste the particle pixels in the designated location. This approach proves to be too simple, and only works if the particle mask is extremely accurate or if the region surrounding the particle matches the region of background onto which the particle is placed. A better solution had to be used.

Poisson image blending [25] is a proven method for blending areas from two or more different images. It involves a linear solver to obtain pixel values given the Laplacian of an image area and some boundary conditions. For a comprehensive description of the method, see original paper. Specifically, results relevant for particled textures can be seen in figure 3 of that article.

While poisson image blending does the job, in most cases it is possible to use much simpler methods. In the general problem for poisson image blending, it was needed to be able to blend areas from completely unrelated images. In our case, we have a much stronger prior. Both the background and the particles derive from the same source image. The background is not strictly a part of the original image, but a synthesis product derived from it. This still implies that basic properties such as color values and gradient sparsity remain the same. Particles are also manipulated but, by definition, keep their visual properties. All the above means that, in most cases, the area around a given particle will resemble a background area (unless the original background had some global illumination effect of very diverse color values). This allows for two simple methods to be used for background-particle composition:

1. Composition using graph-cuts.
2. Composition using α-blending

Composition using graph-cuts is very simple in our case, since we already have the graph-cut machinery in place. All we need to do is connect the middle of the particle to the source vertex and the bounding region of the patch to the sink vertex, and use the exact same mechanism used in the background synthesis process. The “middle” of a particle can easily be obtained by erode operations, until some percentage of the pixels is left. This method was tested and proved useful for many scenarios, but an even simpler method also worked.

α-blending is probably the simplest way to merge two images. Due to the nature of our problem, even this simple method proved adequate in most cases. All examples in this paper, unless stated otherwise, use simple α-blending.

Chapter 5

Discussion

In this paper we have shown a novel way to analyze and synthesize textures with particles. We demonstrated a few methods to separate the particles from the background and to synthesize a new background image. On top of the synthesized background we have shown how to add particles, with various degrees of freedom. An artist can control the look of the particles, their location, orientation and size and even guide the synthesis with a mask or a probability function. We have shown that our method is a generalization of standard texture synthesis, and demonstrated results which are comparable to the current state-of-the-art. Our control options over the synthesis process are either completely new, or give better results than previous attempts to provide the same controllability.

All our input images can be fed into existing texture synthesis algorithms, producing visually pleasing results. We believe the specific attention to layers of texture element allows us to create control and variance in the output image, in a novel manner. Out approach can be thought of as an extension of existing texture synthesis methods, specifically - graph-cut textures, and if we set the element layers to be empty in our algorithm - the two methods overlap.

Our approach is compromised of several steps, some are completely new and others use well-known well-tested synthesis techniques. Each stage of the pipeline can be enhanced given new algorithms that will surely arise. As all pipelines, our approach is only as strong as its weakest link. All stages of the pipeline were designed with a plug-n-play approach. For example, one can easily substitute the element detection method or the background synthesis algorithm. Our approach is limited by its subparts. If an input image contains elements which are hard to detect by a computer, we will not be able to synthesize it in the “advanced” way, falling back to texture synthesis without element layers.

Working on this paper meant we had to pick and choose which directions to pursue, and many direction that were left untouched are an excellent opportunity for future research.

5.1 Future Work

- **Find a good automated technique to measure the quality of a synthesized texture.** This future work is not specific to textures with particles, and it seems will benefit all texture synthesis techniques. The aim is to create some standardized way to automatically evaluate the quality of texture synthesis results.
- **Make our synthesis run faster.** As explained before, the goal of this project was quality and not speed. Speeding up the current implementation (that might spend several minutes to generate a single texture) would benefit the method a great deal.
- **Extending the "image hybrids" method [16] to support particles which are not perfectly aligned, in order to generate new particles.** Image hybrids show great promise as a source for novel texture particles. The current method needs some extensions so that it will fit the diversity of particles found on a single input image. Some automatic alignment and clustering is needed in order to make the method feasible.
- **Other methods for new particle creation.** The possibilities for new particle creation were not all explored in this paper. New techniques for the creation of novel particles should be invented and researched.
- **Create better interactive tools for particle placement and manipulation.** Supplementing our algorithm with a comfortable UI (and not a proof-of-concept UI as the one we supplied) will greatly enhance its usability. It will allow real world artists to experiment with these new manipulation techniques, providing valuable feedback.
- **3D texture synthesis and video synthesis.** Extend our method to work on a 3D volume instead of a 2D surface.

Appendix A

Some Code Snippets

A.1 Calculating patch resemblance score

```
function [output] = patchResemblance(inputPatch, inputImg)
%PATCHRESEMBLANCE Calculate patch difference for all image coordinates

    imgRows = size(inputImg,1);
    imgCols = size(inputImg,2);
    patchRows = size(inputPatch,1);
    patchCols = size(inputPatch,2);

    output = zeros(imgRows, imgCols);

    % extend input with 2 to handle borders
    % we use 2 since most side patches (all except corner are ¬half the
    % size of center patches)
    inputImg = padarray(inputImg, [floor(patchRows/2) ...
    floor(patchCols/2)], 2);

    % calculate score for all pixels
    for row=1:imgRows
        for col=1:imgCols
            imgPart = inputImg(row:row+patchRows−1, ...
    col:col+patchCols−1, :);
            diffMatrix = imgPart − inputPatch;
            output(row, col) = norm(diffMatrix(:));
        end
    end
end
```

A.2 Pixel types for graph-cut

```
% validAreaMask — area in the texture thus far which is valid (within ...
    the patch location
% we are currently looking at).
% patchToCompareMask — mask of the new patch we wish to add.

% intersectionMask — all the pixels that are both in the original and
% the new patch
intersectionMask = patchToCompareMask & validAreaMask;

% For the graph—cut, we need to determine which pixels we want to force
% from the new patch and which from the original
% Orig pixels are those that are both on the edge of the new mask and
% part of the intersection, new pixels are those that are both on the
% edge of the orig mask and part of the intersection. Also — for orig
% pixels we need to remove those that are part of new pixels.

newPatchPixels = findEdgePixels(patchToCompareMask) & intersectionMask;
newPatchPixels(1:end,1) = false;
newPatchPixels(1:end,end) = false;
newPatchPixels(1,1:end) = false;
newPatchPixels(end,1:end) = false;

origPatchPixels = findEdgePixels(validAreaMask) & intersectionMask & ¬...
    newPatchPixels;

% If the above process yields no origPatchPixels, we need a different
% strategy (will happen when orig is fully within new). In this
% case we simply erode the original until we have the correct amount of
% pixels.
if (¬any(origPatchPixels(:)))
    origPatchPixels = intersectionMask;
    while (sum(origPatchPixels(:)) > sum(newPatchPixels(:)))
        origPatchPixels = imerode(origPatchPixels, ones(3));
    end
end
```

A.3 Detecting a single element using a background mask

```
function [classification] = DetectSingleElement(img, backgroundMask, ...
    elementMask)
    BACKGROUND_PIXELS_NEEDED = 50;

    % we do not need all the pixels in background mask. We only use the
    % closest pixels to elementMask (which are for sure background). We
    % simply dilate elementMask and check the number of pixels in the
    % intersection, until enough (BACKGROUND_PIXELS_NEEDED) pixels have
    % been encountered.

    elementSurrounding = imdilate(elementMask, strel("disk", 1));
    tmp = elementSurrounding & backgroundMask;
    while (sum(tmp(:)) < BACKGROUND_PIXELS_NEEDED)
        elementSurrounding = imdilate(elementSurrounding, ...
    strel("disk", 1));
        tmp = elementSurrounding & backgroundMask;
    end
    backgroundMask = tmp;

    % For each pixel which is part of the mask, a row will be created in
    % the output - containing the feature vector for that pixel
    % The vectors are ordered according to the pixel locations - left to
    % right, top to bottom
    edgeImg = edge(rgb2gray(img));
    elementFeatures = GetFeatureVectors(img, edgeImg, elementMask);
    backgroundFeatures = GetFeatureVectors(img, edgeImg, backgroundMask);
    allFeatures = [elementFeatures ; backgroundFeatures];
    % label elementFeatures as 0 and backgroundFeatures as 1
    group = [zeros(size(elementFeatures, 1), 1) ; ...
             ones(size(backgroundFeatures, 1), 1)];

    % Create the interest group of pixels which we want to classify
    CLASSIFICATION_AREA = 21;
    classificationMask = imdilate(elementMask, strel("disk", ...
    CLASSIFICATION_AREA));
    sample = GetFeatureVectors(img, edgeImg, classificationMask);

    classification = classify(sample,allFeatures,group);
end
```

A.4 Aligning particles to same orientation

```
% leave particle in same x/y position
outputGallery{I}.row = inputGallery{I}.row;
outputGallery{I}.col = inputGallery{I}.col;

% Find main axis using PCA
[rows, cols] = find(inputGallery{I}.mask);
[coeff, ¬] = princomp([rows cols]);
angle = atand(coeff(1,1) ./ coeff(2,1));

% Rotate the image and mask
outputGallery{I}.img = imrotate(inputGallery{I}.img, angle);
outputGallery{I}.mask = logical(imrotate(inputGallery{I}.mask, angle));
```

Bibliography

[1] LY Wei, S Lefebvre, V Kwatra, and G Turk. State of the art in example-based texture synthesis. *Eurographics 2009, State of the Art Report, EG-STAR*, 2009.

[2] Y Liu and W Lin. Near-regular texture analysis and manipulation. *ACM SIGGRAPH 2004 Papers*, 2004.

[3] J Lu, J Dorsey, and H Rushmeier. Dominant texture and diffusion distance manifolds. *Computer Graphics Forum*, 28(2):667–676, 2009.

[4] AA Efros and WT Freeman. Image quilting for texture synthesis and transfer. *Proceedings of SIGGRAPH 2001*, pages 341–346, 2001.

[5] V Kwatra, A Schodl, I Essa, and G Turk. Graphcut textures: image and video synthesis using graph cuts. *ACM Transactions on ...*, 2003.

[6] W C Lin, J H Hays, C Wu, V Kwatra, and Y Liu. A comparison study of four texture synthesis algorithms on regular and near-regular textures. *Robotics Institute, Carnegie Mellon University*, 2004.

[7] A Hertzmann, CE Jacobs, N Oliver, B Curless, and DH Salesin. Image analogies. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 327–340, 2001.

[8] Hugues Hoppe Sylvain Lefebvre. Parallel controllable texture synthesis. pages 1–10, May 2005.

[9] J Zhang, K Zhou, L Velho, B Guo, and HY Shum. Synthesis of progressively-variant textures on arbitrary surfaces. *ACM Transactions on Graphics (TOG)*, 22 (3):295–302, 2003.

[10] Iddo Drori, Daniel Cohen-Or, and Hezy Yeshurun. Fragment-based image completion. *SIGGRAPH '03: SIGGRAPH 2003 Papers*, July 2003.

[11] James Hays and Alexei A Efros. Scene completion using millions of photographs. *SIGGRAPH '07: SIGGRAPH 2007 papers*, August 2007.

[12] W T Freeman, T R Jones, and E C Pasztor. Example-based super-resolution. *Computer Graphics and Applications, IEEE*, 22(2):56–65, 2002.

[13] Y HaCohen, R Fattal, and D Lischinski. Image upsampling via texture hallucination. In *Computational Photography (ICCP), 2010 IEEE International Conference on*, pages 1–8, 2010.

[14] M Ashikhmin. Synthesizing natural textures. *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 217–226, 2001.

[15] Marcin Iwanowski. Generalized Morphological Mosaic Interpolation and Its Application to Computer-Aided Animations. In *CAIP '01: Proceedings of the 9th International Conference on Computer Analysis of Images and Patterns*. Springer-Verlag, September 2001.

[16] E Risser, C Han, R Dahyot, and E Grinspun. Synthesizing structured image hybrids. *ACM SIGGRAPH 2010 papers*, pages 1–6, 2010.

[17] Sunil Arya, David M Mount, Nathan S Netanyahu, Ruth Silverman, and Angela Y Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *An optimal algorithm for approximate nearest neighbor searching in fixed dimensions*, December 1995.

[18] S Brooks and N Dodgson. Self-similarity based texture editing. *ACM Transactions on Graphics (TOG)*, 21(3):653–656, 2002.

[19] S Lefebvre and H Hoppe. Appearance-space texture synthesis. *ACM Transactions on Graphics (TOG)*, 25(3):541–548, 2006.

[20] V Kwatra, I Essa, and A Bobick. Texture optimization for example-based synthesis. *ACM Transactions on . . .*, 2005.

[21] Hui Fang and John C Hart. Detail preserving shape deformation in image editing. *ACM Transactions on Graphics (. . .*, 26(3):12, July 2007.

[22] Daniel Lepage and Jason Lawrence. Material matting. In *SA '11: Proceedings of the 2011 SIGGRAPH Asia Conference*. ACM Request Permissions, December 2011.

[23] J.M. Dischler, K. Maritaud, B. Lévy, and D. Ghazanfarpour. Texture particles. *Computer Graphics Forum*, 21(3):401–410, 2002.

[24] Gilad Freedman and Raanan Fattal. Image and video upscaling from local self-examples. *Transactions on Graphics (TOG*, 30(2), April 2011.

[25] P Pérez and et al. Poisson image editing. *ACM Transactions on Graphics (TOG)*, 2003.