Synthesized textures in 3D rendering

Sam Persson

Examensarbete för 15 hp
Institutionen för datavetenskap, Naturvetenskapliga fakulteten, Lunds universitet

Thesis for a diploma in Computer Science, 15 ECTS credits
Department of Computer Science, Faculty of Science, Lund University

Abstract

In 3D rendering a common task is to render large textured surfaces, such as walls and terrains. Ideally you would want a texture that is not obviously repeating itself, but memory limits often work against this goal. This thesis explores a method using texture synthesis that aims to reduce visual repetition and memory usage at the cost of a pre-processing step and higher run time performance requirements. Various optimizations for rendering single and multiple textures using this method are compared and evaluated. A simple method to parallelize patch-based texture synthesis is also introduced. The results show that good quality and real-time performance is possible.

Sammanfattning

Inom 3D-rendering är det vanligt att rendera stora texturtäckta ytor, som väggar och landskap. På grund av minnesbegränsningar låter man ofta en mindre textur upprepas över ytan, men detta blir i vissa fall väldigt tydligt för det mänskliga ögat. Detta examensarbete undersöker en metod som med hjälp av textursyntes kan minska synliga upprepningar och minnesanvändning på bekostnad av ett förbehandlingssteg och högre prestandakrav vid körning. Olika optimeringar av metoden för att rendera enstaka och multipla texturer jämförs och utvärderas. En enkel metod för att parallellisera lapp-baserad (patch-based) textursyntes introduceras också. Resultaten visar att realtidsprestanda och god bildkvalitet kan uppnås.

Contents

1. Introduction **5**

2. Background **5**

2.1. Example based texture synthesis 5

2.2. Approximate nearest neighbor search 7

3. Using synthesized textures in real-time 3D rendering **8**

4. Implementation **10**

5. Results and performance **14**

5.1. Single texture mapping 15

5.2. Multiple texture mapping 18

6. Conclusion **20**

References **21**

Appendix A. texsyn.frag **23**

Acknowledgements

I would like to thank Tomas Akenine-Möller and especially Michael Doggett for their valuable feedback during the quite long process of creating this thesis.

I would also like to thank Sofie Samuelsson and Xix Xeaon for their proofreading and constructive criticism.

1. Introduction

Both natural and man-made materials typically contain repeating elements. Some in very regular patterns; some with nearly random distribution. Many of these patterns, or textures, can be emulated by much simpler processes than those that originally created them, a fact that can be exploited when creating digital images. This thesis explains a few techniques that is commonly used to synthesize images, and explores optimizations of one method that could be used in real-time 3D scenes, such as games and simulations.

2. Background

2.1. Example based texture synthesis

A texture in the context of image processing is a digital image composed of repeated elements. These elements or features can be repeated in regular patterns, stochastically, or anything in between, see figure 1. Texture synthesis is the process of algorithmically generating such textures. One way to do so is to construct an analytic function taking a set of parameters and outputting a texture – what is called procedural texture synthesis. But since you often want to mimic natural textures, like grass or a brick wall, coming up with a decent function is often hard. Instead, a common way is to use a sample image of a part of a texture to generate a similar, but arbitrarily large, texture. This is called example based texture synthesis.

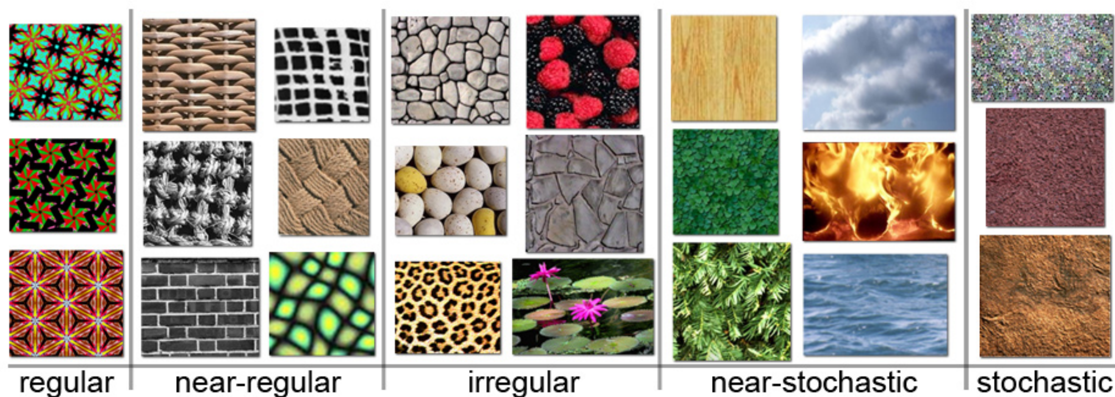


Figure 1: Different kinds of textures.

One of the simplest ways to do texture synthesis from a sample image would be to tile the sample image: copy the sample image and paste it side by side until the output image is filled. However, this would in most cases result in visible seams between the tiles, or at least lead to a highly repetitive result.

A more general idea is to generate the texture pixel by pixel; this is called pixel-based synthesis [1]. It could be described like this:

1. Pick a pixel randomly from the sample image and put it in the output image.

2. Pick another pixel from the sample image to put next to the latest pixel in the output image, by comparing the neighboring pixels in both images. Repeat until the output image is filled.

It is a simple and easy to use algorithm – the only parameters are the input image and the neighborhood size. If a neighborhood is seen as a vector of dimension N, where N is the number of pixels in a neighborhood, they can be compared by squared distance. This is quite costly if calculated for all possible neighborhoods, but can be accelerated using some nearest neighbor algorithm [2]. There also exist various improved algorithms that for example take spatial coherence of the selected pictures into account, such as k-coherence [3] that limits the search-space and improves both quality and speed.

Another way to increase speed and quality of the texture synthesis is to synthesize patches instead of pixels – so called patch-based texture synthesis. The pixels in a patch of the input image already fit together well, so as long as you can fit different patches together in a good way the quality of the output image should be good. And since you do not need to select as many patches as pixels to synthesize images of equal size, the speed can increase.

To fit patches together some different techniques have been used, one of the simplest being to make patches overlap and blend the images along the boundary [4]. This can sometimes lead to blurry artifacts, so other methods such as finding an optimal path on the boundary to cut through [5, 6] or warping the patches to make the edges match [7, 8] have been tried.

Yet another way to do texture synthesis is to generate tiles with a small number of border classes, such that borders of the same class match [9]. Then it is a matter of laying out the tiles while keeping the edge constraints.

2.2. Approximate nearest neighbor search

To synthesize a texture from a sample you need a way to find parts of the input image that fit well together. Many algorithms use a nearest neighbor (NN) search algorithm to do this. Finding the nearest neighbor to a point a is the same as finding the point b within a set S with the smallest distance to a, commonly measured as Euclidean or Manhattan distance. A common algorithm for finding the nearest neighbors uses the k-d tree data structure [10]. A k-d tree is a binary space partitioning tree, with each non-terminal node splitting its children along one assigned dimension, see figure 2. What dimension to split along, and where to split, is decided when building the tree; it is called a splitting rule. The *standard splitting rule* [11] is to split along the dimension with the largest variance, and to partition around the median in that dimension. This creates a balanced tree that allows for nearest neighbor search with expected-case running time of $O(\log n)$, for any fixed dimension d and assuming uniformly distributed points. However, NN search is affected by the *curse of dimensionality* [12]: in high dimensional spaces it is hard to get better than linear search ($O(dn)$). Therefore approximate nearest neighbor (ANN) algorithms are often used. These does not always find the *nearest* neighbor, but they often find one at least almost as near.

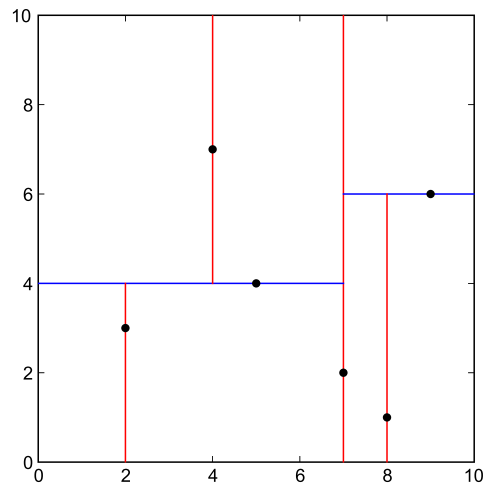


Figure 2: One possible way of splitting the points when constructing a two-dimensional k-d tree. The point along the longest vertical line represents the root node in the tree, with the two points on the horizontal lines being its child nodes.

One way to make an ANN query is to stop the search after a set number of nodes in the tree has been visited, and return the best point so far. This makes the algorithm dependent on the traversal order. Best Bin First [13] is a simple modification of the common NN algorithm for a k-d tree, that allows for a more efficient ANN search in this manner by traversing the tree in the order of a priority queue of the paths not selected, instead of the common, depth first-like traversal.

The *sliding midpoint rule* is an alternative splitting rule that has good properties for ANN [14].

3. Using synthesized textures in real-time 3D rendering

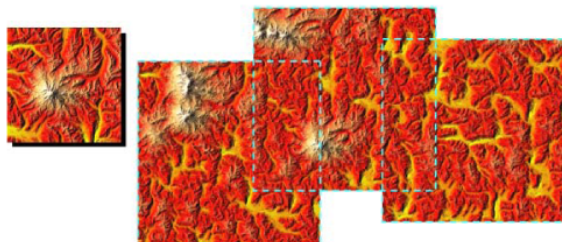
A texture in the context of 3D rendering is a digital image that is applied to a three dimensional surface – this is a somewhat different definition than what has been spoken of so far. If the surface to be textured should look like a wall or a landscape it is often a textural image you want to use. But it could also be that you want to texture a painting or a human character – in these cases the image used would probably not qualify as a textural image as defined in the context of image processing. *Real-time* is another term with different meanings in different contexts. In the context of 3D rendering it means that you need to be able to generate a new output image many times per second, with low latency.

Using a synthesized texture to texture a surface in real-time 3D rendering could be done in the same manner as if using a regular image, as long as the texture is generated beforehand. This method is sometimes used as a way to make tileable textures as a synthesized texture could easily be generated with the property that the edges match when tiled. Tileable textures are often useful for larger surfaces such as walls and terrains, since a texture for the whole surface would use a lot of memory. The disadvantage is that the repetitions can be very visible.

However, today's graphics hardware impose three important limits on using textures:

- The hardware used to do 3D rendering in real-time, the graphics card, has a limited amount of memory.
- Transfer between regular memory and the graphics card memory is expensive.
- The size of a texture is restricted.

This means that you cannot use very large textures, say over 8192×8192 pixels, which in case of textural images would often be useful to reduce the amount of visible repetitions.



(a) Spatial determinism – regions that overlap must match even if synthesized independently.

(b) Local evaluation – using a sequential synthesis the ghosted region need to be generated before the region in the blue rectangle can be generated.

Figure 3: Difficulties with on-the-fly synthesis [15].

One alternative is to do texture synthesis on-the-fly, in real-time. To do that you need to solve how to handle spatial determinism – the color of a pixel should remain the same

no matter the order pixels are synthesized in, and local evaluation – to decide what color a pixel should have you should only need to know the color of a small number of other pixels, see figure 3. You also need to have an algorithm that is fast enough to be executed in real-time. This has been done using an iterative approach [16] as well as using a further improved parallel texture synthesis algorithm [17]. These algorithms are however quite complex.

Another alternative is to do the actual synthesis beforehand and store only the data that is necessary to reproduce the result. This can then be used as a form of texture compression specific to textures generated using texture synthesis. This has been tried before using a tile based texture synthesis method [18]. In the case of patch-based texture synthesis it would mean storing the coordinates of the selected patches, and that is the approach taken in this thesis.

4. Implementation

The main idea is to use patch-based texture synthesis to generate a large texture, but instead of storing the generated texture, the coordinates of the patches in the input image are stored. Then a shader[1] is written that uses the input image and the patch coordinates to reconstruct the synthesized texture in real-time.

The texture synthesis part uses ANN search to select patches and blends different patches linearly of the the overlapping borders. To do ANN lookup a simple Best Bin First implementation using the sliding midpoint rule was written in C. Using existing generic ANN libraries was problematic because they could not easily be made to utilize the fact that most of the data of a patch is shared with other patches, which leads to unnecessarily high memory usage.

Patches are selected in a left-to-right, top-to-bottom order. The first patch is synthesized by picking a random patch in the input image, the rest by comparing the overlapping borders. To synthesize a patch, the borders that it will overlap in the so-far generated output image are collected in a single vector and used to look up a good match in a k-d tree of all such border zones in the input image. The output image is considered a wrapping image, so when selecting patches sequentially there are 8 different configurations of overlapping borders – see figure 4 – so 8 different k-d trees need to be constructed. To save memory, the color data of every possible border are not stored in the k-d trees, but only pointers to a pixel in an image, along with description of the shape of the border zone. Also, the k-d trees can be constructed in parallel.

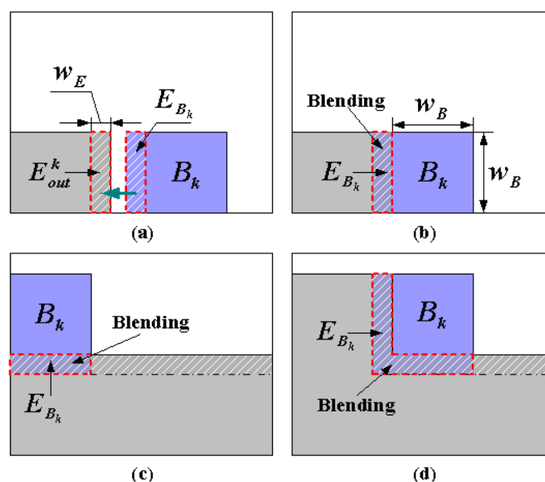


Figure 4: Four different border configurations [4].

To speed up the synthesis a simplification of the dependencies between patches have been made, see figure 5. This ignores some dependencies caused by overlapping corners (over the wrapping edge), but leads to good enough results in practice and allows for parallelization of the algorithm. Even though the actual synthesis is not used in real-time, a fast synthesis is useful when tweaking the parameters manually.

[1] A shader is a program written for a graphics processor, or GPU.

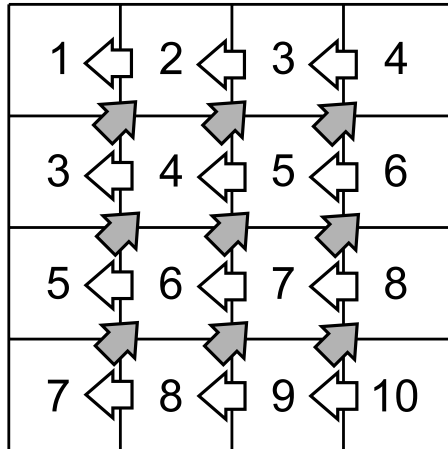


Figure 5: Simplified patch dependencies when synthesizing a texture of 4 by 4 patches, in the order left to right, top to bottom. The numbers indicate in which step each patch can be synthesized. This means that in an image with this many patches, using two processing cores can speed up the synthesis from $16k$ to $10k$, where k is the time it takes to synthesize one patch. When synthesizing larger images, more cores can be utilized and more speedups can be gained.

The parameters to the synthesizing program are the input image, the number of patches in the output image, the size of a patch, and the width of a border. A good patch size is roughly the size of a feature in the input image, and the border size affects the randomness of the output, see figure 6.

The real-time part of the implementation is an OpenGL application written in Scala and using the jMonkey Engine. The relevant part is the GLSL fragment shader that uses two texture samplers to reconstruct the synthesized image and apply it to a surface, see appendix A. The first sampler is bound to the input image, the second to a 16-bit texture containing the patch coordinates, the x-coordinate in one channel and the y-coordinate in another.

The shader first determines what patch the current fragment[2] is in by multiplying the texture coordinate by the number of patches, and rounding down to the nearest integer value. The coordinate of this patch, and the three neighboring patches, are looked up in the second texture sampler. These four coordinates are used to look up four values in the input image. These are then blended together depending on where in the patch the fragment is located. If the fragment is not on a patch border, only one color value is actually used. This is wasteful, since sampling textures is expensive, but hard to avoid since GPUs are bad at rapidly changing dynamic branching. One thing can be done however – when the patches are so small on the screen that the border is hardly visible, only single patch look-up is used, skipping blending entirely. This is determined by comparing the z value of the fragment (in screen coordinates) with a shader parameter. The minimum value of this parameter before the borders between patches become visible

[2] A fragment is often the same as a screen pixel.

depends on the synthesized texture. This kind of dynamic branching does not change rapidly while traversing the screen, and gives a speed boost on modern graphics hardware. Such a method to reduce rendering complexity based on distance from the viewer is called a level of detail (LOD) technique.

There are some other interesting details in the shader. The ordinary texture look-up function uses the derivatives of the specified coordinate to select what level in the mipmap pyramid to fetch from. Slightly simplified, it can be described as: if the derivative is high, the pixel is far away and a lower resolution version of the image is used to look-up. However, since the patches can lie anywhere in the input image these derivatives are much larger on the borders between patches. The *textureGrad* operation is therefore used to specify other derivatives, that are linear across the surface, which prevents incorrect mipmap levels along the borders between patches. However, this operation is unnecessarily slow on most hardware when not using anisotropic filtering. When using only trilinear filtering it can be emulated faster using the *textureLod* operation, see *textureGrad_fake* in appendix A for details. There is also an operation named *textureGather* that can fetch four neighboring texels in one texture-fetch. This operation is used to reduce the number of texture-fetches from 8 to 6 and improves performance slightly.

In real applications, it is common that you do not only want to apply one texture to a surface, but multiple textures that blend together. You could for example want to have a path in the terrain. A common technique for doing this is to use splatting – use one extra texture (sometimes called an alpha-map) to look up where the regular textures should be applied, and how they should be blended, using one color channel for each input texture. Since this multiplies the number of texture fetches by the number of input textures, a variation of the shader was made to try to take advantage of the patch-structure to handle multitexturing. For this shader, the synthesized texture was generated from three different textures, with one k-d tree for each input texture. An alpha-map was used to see which of the k-d trees should be used for each patch synthesized, and an index for the selected texture was stored along with the patch coordinates. Then in the shader instead of one input texture, an array of input textures was used, and the stored texture index was used to select what layer in the array should be used for each texture look-up. To reduce the visibility of the borders between patches with different texture indexes, blending was increased to its maximum value when the neighboring patches lay on different textures. A disadvantage is that the same synthesis parameters have to be used for each texture, which might lead to less than optimal output.

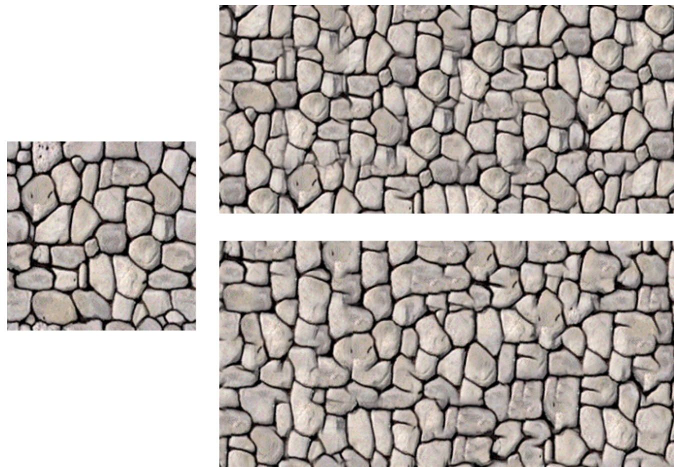


Figure 6: A texture synthesized with the implemented software. To the left is the input image. The upper right image was synthesized using a patch size of 64 pixels and a border width of 16 pixels. The lower right image was synthesized using a patch size of 32 pixels and a border width of 8 pixels. The difference is subtle, but the upper image has more blurred borders, and the lower image has more visibly repeated features.

5. Results and performance

The performance of the software was measured on a Windows 7 machine with an AMD Phenom II X6 1090T CPU running at 3.20 GHz and an nVidia GeForce GTX 560 Ti GPU. To synthesize an image of size 1024×1024 pixels from an input image of size 194×194, with 1024 32×32 patches and a border width of 16 pixels, takes about 11 seconds, of which 1 seconds are for constructing the k-d trees and 10 seconds for synthesizing the patches. The time to construct k-d trees depends on the input image size and the patch size and border width, while the time to synthesize patches depend mostly on the number of patches to synthesize and the number of nodes in the k-d tree visited per look-up. Synthesizing very large images can take hours, but you can synthesize a small texture to verify that the parameters give a good output, and then synthesize a large image over night to use when rendering. This part of the synthesis is only a pre-processing step, so optimizing its performance was not a primary goal for this thesis, but instead making the real time rendering as fast as possible.

For all real-time scenes, a synthesized texture applied to a 3D-terrain was rendered to a 1680×1050 screen with the filtering of the input image-texture set to trilinear and the filtering of the coordinate texture set to nearest neighbor. The real-time performance was measured in frames per second, FPS, when the frame rate had stabilized. The theoretical throughput was estimated for an AMD Radeon HD 6970 GPU using AMD's GPU ShaderAnalyzer utility. This is a quite different GPU than the one used for testing, but it represents the same generation of graphics cards and the results should still give a good picture of the relative complexity of the different shaders.

5.1. Single texture mapping

To compare single texture mapping performance a synthesized texture of 256×256 patches of size 32×32 pixels and a border width of 8 pixels was used. This means that the output image was 8192×8192 pixels large, a size that only more recent GPUs can handle when used as a regular texture. The input image was of size 196×196 pixels. The procedural texture uses four layers of simplex noise, representing a minimum amount of processing needed for a decent procedural texture. The regular texture shaders use the input image to the texture synthesis and the output synthesized texture respectively, to compare the memory usage.

Table 1: Performance results rendering a single texture.

Shader	FPS	Theoretical throughput (Mpixels/s)	memory usage (kB)
Synthesized texture without LOD	1 060	3 911	413
Synthesized texture with LOD	1 216	5 029	413
Procedural texture	371	2 139	0
Regular texture (input image)	1 816	14 080	110
Regular texture (output image)	1 816	14 080	268 000

Using the synthesized texture shaders the rendered image looks almost exactly the same as when using the output image as a regular texture. That is, except for some very slight “jumping” in the far distance when the camera is moved, coming from the nearest neighbor filtering of the coordinate texture; when a patch is smaller than a pixel, choosing a patch becomes an almost random process. However, at this distance the input-texture is so blurred from its trilinear filtering that this is rarely noticeable. Compared with using a tiled image the size of the input image any visual repetitions are drastically reduced, see figure 7.

With the LOD shader, about a half of the screen can use the simpler branch with next to no visible difference, see figure 8. The theoretical performance is estimated based on this assumption.

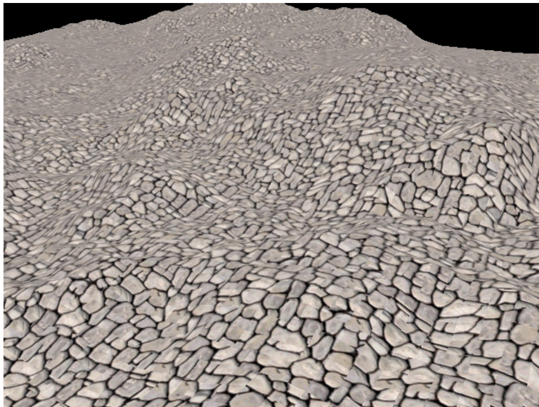
The difference in memory usage between the regular texture shaders is huge, but in an actual application it might lie somewhere in-between. You seldom actually need a 8192×8192 pixel texture, but to avoid the repetitions visible in figure 7a you might need one larger than for example 512×512 pixels.

The poorer performance of the synthesized texture shaders comes mostly from the fact that 6 texture fetches are needed for every fragment in the worst case, and only one is needed when using a regular texture. Another factor is that the patch coordinates are seemingly randomly distributed, which could be bad for texture caching when the patches are small. The performance is still many times higher than the 60 FPS that most displays can show, and also a lot better than the simple procedural shader that was compared with.

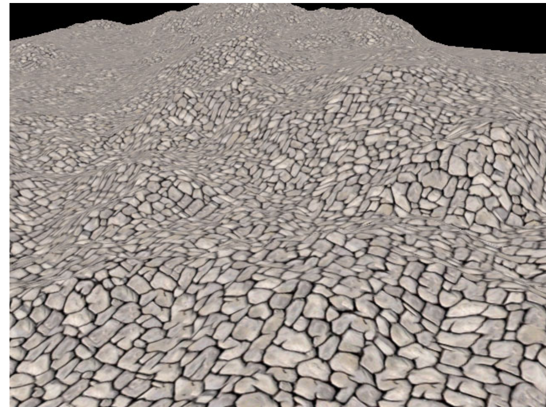
(a) A 512×512 pixel image is tiled over the surface. You can see the same feature appearing over and over again.

(b) A synthesized texture is applied to the surface. Parts of the texture are still repeated, but in an irregular manner.

Figure 7: Comparison between tiling and a synthesized texture.



(a) With no blending. The inconsistencies between patches should be visible among the closest patches.



(b) With blending on the about 50 % pixels closest to the camera.

Figure 8: The effect of blending between patches.

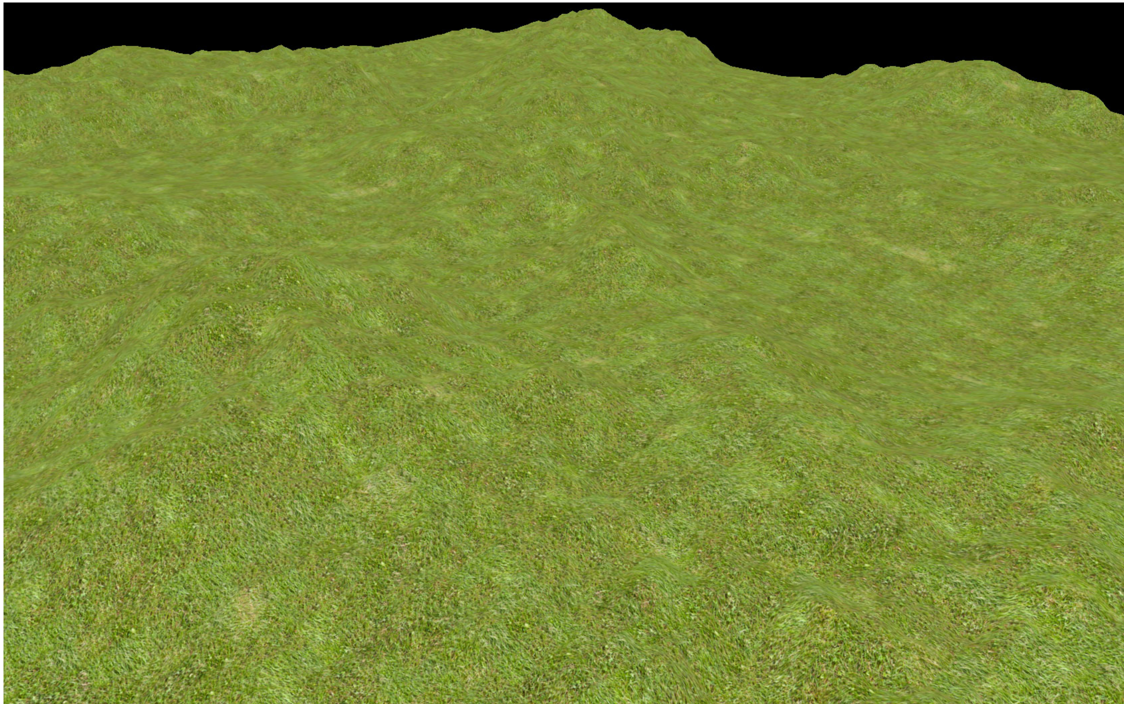


Figure 9: This image was rendered using a synthesized texture of 128×128 patches of size 64×64 pixels, and a border width of 8 pixels. This synthesized texture looks good as far as repeated features are concerned, but somewhat sharp differences in brightness are visible. This shows that the input image should have an even luminosity across the whole image.

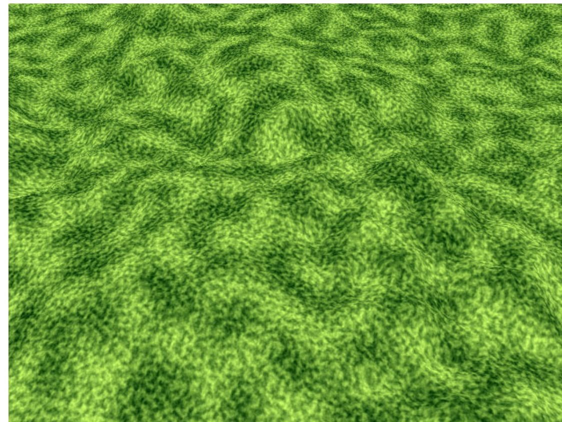


Figure 10: The procedural texture used for comparison. It should be possible to get much better visual quality at about the same performance, but that is not in the scope of this thesis.

5.2. Multiple texture mapping

Multi-texturing performance was compared using 3 textures of size 256×256 pixels and an alpha-map of 1024×1024 pixels. The texture array version of the synthesized texture was generated using 128×128 patches of size 64×64 pixels and a border width of 8 pixels, and uses the alternate shader that takes advantage of the patch-structure of the synthesized texture. The shader using splatting and synthesized textures used three different synthesized textures with different parameters, and uses LOD.

Table 2: Performance results rendering multiple textures.

Shader	FPS	Theoretical throughput (Mpixels/s)	memory usage (MB)
Synthesized texture using texture array	1000	3 520	0.672
Synthesized texture splatting	512	~1 600	5.49
Regular splatting	1307	8 800	3.73

The visual quality, see figure 11, is much improved using synthesized texture splatting over using regular splatting; repeated features are hardly visible at all. Although the performance hit is quite large, it is still many times faster than what is necessary for real-time rendering.

The shader using the texture array on the other hand gives a completely different result. The path and the borders between different textures look blocky, and smother transitions and mixed textures are nowhere to be seen. A smaller patch size might reduce the blockiness but would also reduce blending between textures and introduce sharper borders, and in addition might reduce the quality of the synthesized texture. For some applications this visual style could still fit in, but it might be hard to use it for realistic images. However, only for this shader is the performance not limited by the number of textures used.

Figure 11: A visual comparison of the different multi-texturing techniques. The top-most image uses the texture-array based shader, the middle image uses synthesized texture splatting and the bottom images uses regular splatting.

6. Conclusion

The results show that patch-based, pre-generated, synthesized textures can be used in real-time 3D rendering to reduce video memory usage or increase texture quality at the cost of performance. Whether this tradeoff is worthwhile depends on the intended application, but most modern applications should have bottlenecks in other parts of rendering than in for example terrain texturing. The synthesized texture splatting-shader could probably be used in some modern games without much modification, and improve visual quality with only slightly worse performance. And a similar method could be used for offline rendering to reduce rendering times for some scenes in CGI-videos, since more textures can fit in memory. But the technique is no silver bullet – it only works for textures that can be generated using patch-based synthesis in the first place.

The multi-texturing technique using a texture array would probably need much improvement before practical use. One way to improve it could be to store two texture indexes per patch and one value specifying how to blend between them. Or to introduce noise when selecting level in the texture array. Or it might be that a completely different multi-texturing optimization can be adapted for synthesized textures in a better way.

One interesting aspect of this method of using texture synthesis to texture large areas is that it is highly automatic and not particularly customizable. This appeals to programmers, but perhaps not as much to texture artists, who might want full control over the result. One solution that allows for full control is what is called megatextures or clipmaps [19] that allows usage of huge textures by streaming the parts that are visible to the graphics card. The initial texture could of course still be generated using texture synthesis. Comparing these methods to the methods in this thesis could be an opportunity for future research.

Other areas of future research may include finding better ways to join patches than blending, how to handle normal and specular maps, improving anisotropic filtering quality, and whether more adaptable or even dedicated hardware could improve performance.

References

[1] A Efros and T Leung. Texture synthesis by non-parametric sampling. In *Proceedings of the International Conference on Computer Vision*, volume 2 of *ICCV '99*, pages 1033–1038, 1999.

[2] L-Y Wei and M Levoy. Fast texture synthesis using tree-structured vector quantization. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 479–488, 2000.

[3] X Tong, J Zhang, L Liu, X Wang, B Guo, and H-Y Shum. Synthesis of bidirectional texture functions on arbitrary surfaces. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 665–672, 2002.

[4] L Liang, C Liu, Y-Q Xu, B Guo, and H-Y Shum. Real-time texture synthesis by patch-based sampling. *ACM Transactions on Graphics*, 20:127–150, July 2001.

[5] A Efros and W T Freeman. Image quilting for texture synthesis and transfer. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 341–346, 2001.

[6] V Kwatra, A Schödl, I Essa, G Turk, and A Bobick. Graphcut textures: Image and video synthesis using graph cuts. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 277–286, 2003.

[7] C Soler, M-P Cani, and A Angelidis. Hierarchical pattern mapping. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 673–680, 2002.

[8] Q W Yizhou. Feature matching and deformation for texture synthesis. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 364–367, 2004.

[9] M F Cohen, J Shade, S Hiller, and O Deussen. Wang tiles for image and texture generation. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 287–294, 2003.

[10] J Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517, September 1975.

[11] J H Friedman, J L Bentley, and R A Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3:209–226, September 1977.

[12] R B Marimont and M B Shapiro. Nearest neighbour searches and the curse of dimensionality. *Journal of the Institute of Mathematics and its Applications*, 24:59–70, 1979.

[13] J Beis and D Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition*, CVPR '97, pages 1000–1006, 1997.

[14] S Maneewongvatana and D M Mount. It's okay to be skinny, if your friends are fat. In *Center for Geometric Computing 4th Annual Workshop on Computational Geometry*, pages 84–89, 1999.

[15] L-Y Wei, S Lefebvre, V Kwatra, and G Turk. State of the art in example-based texture synthesis. In *Eurographics '09 State of the Art Reports (STARs)*, pages 93–117, 2009.

[16] L-Y Wei and M Levoy. Order-independent texture synthesis. Technical Report TR-2002-01, Computer Science Department, Stanford University, April 2002.

[17] S Lefebvre and H Hoppe. Parallel controllable texture synthesis. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, pages 777–786, 2005.

[18] L-Y Wei. Tile-based texture mapping on graphics hardware. In *ACM SIGGRAPH 2004 Sketches*, SIGGRAPH '04, pages 67–, 2004.

[19] C Tanner, C Migdal, and M Jones. The clipmap: A virtual mipmap. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 151–158, 1998.

Appendix A texsyn.frag

The uniforms should be set as follows:

- m_Diffuse: a texture array of the input images (at least one).
- m_Coords: a luminance/alpha texture with patch coordinates, divided by the input image size.
- m_TextureIndex: optional, an alpha texture with the texture indexes of the patches.
- m_ps: the patch size divided by the input image size.
- m_bs: the border width divided by the patch size.
- m_np: the number of patches per side.
- m_lod: optional, a value specifying the maximum distance where the simple branch should be taken.

To enable all optimizations, TEXTURE_GATHER should be defined. That requires a GPU and drivers that support OpenGL 4.0 or higher.

For older GPUs, modify the version on the first line to, for example, #version 330.

```
1  #version 420
2
3  uniform sampler2DArray m_Diffuse;
4  uniform sampler2D m_Coords;
5  #ifdef TEXTURE_INDEX
6  uniform sampler2D m_TextureIndex;
7  #endif
8
9  uniform float m_ps;
10 uniform float m_bs;
11 uniform float m_np;
12 #ifdef LOD
13 uniform float m_lod;
14 #endif
15
16 in vec2 texCoord;
17 in vec4 pos;
18
19 vec4 textureGrad_fake(sampler2DArray sampler, vec3 P, vec2 dPdx, vec2 dPdy)
20 {
21    vec2 dx = dPdx * textureSize(sampler, 0).xy;
22    vec2 dy = dPdy * textureSize(sampler, 0).xy;
23    float d = max(dot(dx, dx), dot(dy, dy));
24
25    return textureLod(sampler, P, 0.5 * log2(d));
26 }
27
28 #ifndef TEXTURE_GRAD
29 #define textureGrad textureGrad_fake
30 #endif
31
32 vec4 textureSynth(sampler2DArray sampler, sampler2D coords,
33 #ifdef TEXTURE_INDEX
```

```
34    sampler2D textureIndex,
35  #endif
36    vec2 P,
37  #ifdef LOD
38    float lod, float z,
39  #endif
40    float ps, float bs, float np)
41  {
42    vec2 coord = P * np + 0.5;
43
44    vec2 patchCoord = floor(coord);
45    vec2 inPatchCoord = coord - patchCoord;
46
47    vec2 dFdxCoord = dFdx(coord*ps);
48    vec2 dFdyCoord = dFdy(coord*ps);
49
50  #ifdef LOD
51    if(z > lod)
52    {
53      vec2 v = vec2(inPatchCoord.x < 0.5?-0.5:0.5,
54                    inPatchCoord.y < 0.5?-0.5:0.5);
55
56      vec2 t = texture2D(coords, P).zw;
57
58      vec3 u = vec3((t + (inPatchCoord - v + bs/2.0)*ps), 0.0);
59      #ifdef TEXTURE_INDEX
60        u.z = texture2D(textureIndex, P).a * 256.0;
61    #endif
62
63    return textureGrad_fake(sampler, u, dFdxCoord, dFdyCoord);
64    }
65  #endif
66
67    #ifndef TEXTURE_GATHER
68    vec2 ipatch[4] = vec2[](
69      (patchCoord+vec2(-0.5, 0.5)) / np,
70      (patchCoord+vec2( 0.5, 0.5)) / np,
71      (patchCoord+vec2( 0.5,-0.5)) / np,
72      (patchCoord+vec2(-0.5,-0.5)) / np
73    );
74    #endif
75    vec4 ipatchx;
76    vec4 ipatchy;
77    #ifdef TEXTURE_GATHER
78      ipatchx = textureGather(coords, patchCoord/np, 2);
79      ipatchy = textureGather(coords, patchCoord/np, 3);
80    #else
81      for(int i=0; i<4; i++)
82      {
83        ipatchx[i] = texture2D(coords, ipatch[i]).z;
84        ipatchy[i] = texture2D(coords, ipatch[i]).w;
85      }
86    #endif
87    #ifdef TEXTURE_INDEX
88      vec4 tindex;
89      #ifdef TEXTURE_GATHER
90        tindex = textureGather(textureIndex, patchCoord/np, 3) * 256.0;
91      #else
92        for(int i=0; i<4; i++)
93        {
94          tindex[i] = texture2D(textureIndex, ipatch[i]).a * 256.0;
95        }
```

```
96       #endif
97       if(tindex.xyzw != tindex.yzwx)
98       {
99         bs = 1.0; // Blend maximally between different layers.
100      }
101    #endif
102
103    vec4 colors[4];
104    const vec2 offsets[4] = vec2[] (
105      vec2(0.5,-0.5), vec2(-0.5,-0.5), vec2(-0.5,0.5), vec2(0.5,0.5)
106    );
107    for(int i=0; i<4; i++)
108    {
109      vec3 u = vec3((vec2(ipatchx[i],ipatchy[i])
110              + (inPatchCoord + offsets[i] + m_bs/2.0)*ps),
111                    0.0);
112      #ifdef TEXTURE_INDEX
113        u.z = tindex[i];
114      #endif
115      colors[i] = textureGrad(sampler, u, dFdxCoord, dFdyCoord);
116    }
117
118    vec2 transition = clamp((inPatchCoord-0.5)/bs+0.5, 0.0, 1.0);
119    vec4 a = mix(colors[3], colors[2], transition.x);
120    vec4 b = mix(colors[0], colors[1], transition.x);
121    return mix(a, b, transition.y);
122  }
123
124  layout(location=0, index=0) out vec4 frag_color;
125  void main(void)
126  {
127    frag_color = textureSynth(m_Diffuse, m_Coords,
128  #ifdef TEXTURE_INDEX
129    m_TextureIndex,
130  #endif
131    texCoord,
132  #ifdef LOD
133    m_lod, pos.z,
134  #endif
135    m_ps, m_bs, m_np);
136  }
```