Anisotropic Filtering for On-the-fly Patch-based Texturing

Nicolas Lutz, Basile Sauvage, Frédéric Larue and Jean-Michel Dischler

ICube, Université de Strasbourg, CNRS, France

Abstract

On-the-fly patch-based texturing consists of choosing at run-time, for several patches within a tileable texture, one random candidate among a pre-computed set of possible contents. This category of methods generates unbounded textures, for which filtering is not straightforward, because the screen pixel footprint may overlap multiple patches in texture space, i.e. different randomly chosen contents. In this paper, we propose a real-time anisotropic filtering which is fully compliant with the standard graphics pipeline. The main idea is to pre-filter the contents independently, store them in an atlas, and combine them at run-time to produce the final pixel color. The patch-map, referencing to which patch belong the fetched texels, requires a specific filtering approach, in order to recover the patches that overlap at low resolutions. In addition, we show how this method can achieve blending at patch boundaries in order to further reduce visible seams, without modification of our filtering algorithm.

CCS Concepts

• ***Computing methodologies*** → ***Rendering; Texturing; Antialiasing;***

1. Introduction

Adding detail to 3D surfaces in virtual scenes is often performed through the mapping of textures, as they are able not only to represent a simple color (albedo), but also complex reflectance parameters, or relief. Since very large scenes can nowadays be modelled and visualized, very large textures are required too, leading to storage problems due to graphics hardware limitations. On-the-fly texture synthesis methods have arisen to solve these issues, by generating the textures on demand at run-time, thus leading to virtually *unbounded* textures.

Among these methods, tile-based approaches are very efficient and popular. They consist in pre-computing a set of tiles which are assembled on a regular grid at run-time. However, a reduced set of tiles generally leads to repetition artifacts due to the fact that the same large pieces of contents are reused too often. Conversely, using a high number of tiles may help to remove such artifacts, but is not tractable, due to memory consumption. To get round this issue, on-the-fly patch-based approaches [VSLD13, KCD16] have proposed to pre-compute *patches*, namely contiguous regions with irregular shapes, in a repeatable tile. Then, every patch occurrence is filled-in at run-time with a *content* selected randomly (see Figure 1). This has proved to be effective for irregular and homogeneous textures with structured patterns (i.e. with sharp features), like the ones presented in Figure 2.

Texture filtering is mandatory to make real-time texturing techniques useful in practice, as it removes under-sampling artifacts which occur when viewing a textured surface from far away or at a grazing angle. In the case of on-the-fly texture synthesis, rendering requires to compute every screen-space pixel in constant time, whatever its footprint in texture space. This makes the filtering difficult: one cannot afford computing the high-resolution texture, and then averaging over an arbitrarily large footprint. Instead, filtering and synthesis have to be coupled: only the useful texture resolution must be computed. As far as we know, no such solution has been proposed for the aforementioned patch-based methods.

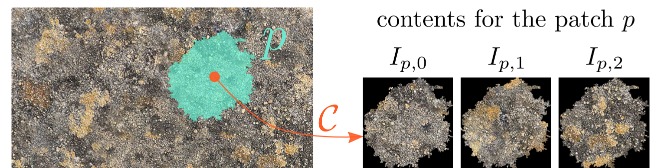

Figure 1: *Content selection process. At rendering, each occurrence of the patch p is filled with a content $I_{p,c}$ randomly chosen by $\mathcal{C}$ among a set of pre-defined candidates.*

In this paper, we propose a technique for anisotropic filtering of [VSLD13]. This solution consists in storing explicitly low resolutions of the contents, each one being pre-filtered independently. The patch-map (which indicates the patch each texel belongs to) is also MIP-mapped using a special bitwise-OR boolean operation. We show how our method allows for simple filtering at every resolution while remaining relatively cheap in terms of both memory and computational costs. After a review of the literature (Section 2), our method is detailed in Section 3 and 4, and evaluated in Section 5. We then provide some perspectives in the concluding Section 6.

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

DOI: 10.2312/egs.20191019

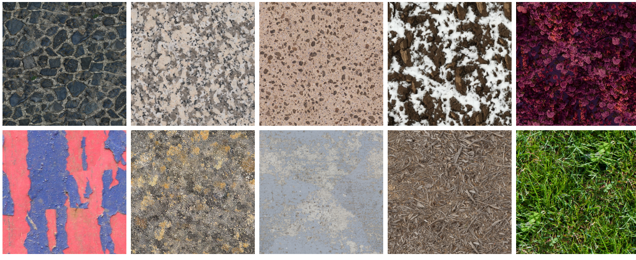

Figure 2: *Examples of irregular, homogeneous and structured textures, particularly suitable for content exchange.*

2. Related work

In this paper, we focus on real-time texture synthesis, and specifically tile-based methods. The quality of the results of the methods which fall in this category is not meant to be competitive against the best offline synthesis results, but rather to provide a good balance between quality, size, and speed. These methods generate an output by putting tiles next to each other, in a way such that there is no visible seam between adjacent tiles. To achieve this, both Wang Tiling [CSHD03, Wei04, LD05] and Corner Tiling [LD06, SD10] indicate the compatibility of the edges or corners of square tiles, and then assemble them in an aperiodic tiling. The filtering problem for these methods has been solved by [Wei04], by packing the tiles as a correct tiling in a texture which is then filtered, allowing borders of adjacent tiles to blend into each other at low resolutions without introducing visual artifacts.

A drawback of these methods is that repeating the same set of squared tiles introduces visually unpleasant repetition artifacts, unless the amount of different edges (resp. corners) is increased, which exponentially increases the corresponding amount of tiles. Both [VSLD13] and [KCD16] solve this issue by partitioning the tiles into regions called patches, whose contents can be exchanged for different, pre-determined alternative contents taken from the tile itself, or from a larger input. While [VSLD13] segments the entire tile with a single set of patches, at the risk of introducing visible seams near the patch borders, [KCD16] limits seams by computing different sets of non-overlapping patches, but at the cost of re-introducing possible alignments, especially at the tile corners. For these methods, the filtering solution proposed by [Wei04] is impractical, since it would require to generate tiles representing every possible combination of contents, yielding a memory complexity which grows exponentially with the number of patches.

Even if the decomposition of the input tile into patches may seem to be related to the problem of multi-chart texture filtering, approaches like [RNLL10] are not adapted to our case, because contents that must be blended are only known at rendering time, thus preventing an a priori filtering of texture seams. To our knowledge, no solution has been proposed yet for the filtering of on-the-fly patch-based texturing.

3. Problem statement

3.1. Content exchange

The algorithm of [VSLD13] works as follows. For clarity, we consider a single periodic tile. The tile is partitioned into P patches numbered $1 \leqslant p \leqslant P$. They are encoded by a patch-map $\mathcal{P}$ such that $\mathcal{P}(\boldsymbol{x}) = p$ if the texel $\boldsymbol{x}$ belongs to patch p. Because the tile is assumed to be periodic, the patch-map is also repeated periodically. Let $I_{p,c}$ be the c-th content of patch p: $I_{p,c}(\boldsymbol{x})$ yields a texel value (Figure 1). Without loss of generality, we assume that every patch is given the same number C of contents, numbered $1 \leqslant c \leqslant C$. At each tile repetition, each patch p is repeated, but its content changes: a function $\mathcal{C}(p)$ yields a random content number c. The synthesized texture is then defined, for any position $\boldsymbol{x}$, as

$$I(\boldsymbol{x}) = I_{\mathcal{P}(\boldsymbol{x}),\mathcal{C}(\mathcal{P}(\boldsymbol{x}))}(\boldsymbol{x}). \tag{1}$$

3.2. Filtering

Once the texture I is mapped to a 3D scene, every screen-space pixel has a projected footprint in texture space. The filtering consists in approximating the average of I over this footprint. In a standard pipeline with pre-computed I, it relies on a MIP-map $I^{(\ell)}$ of the texture: as the level ℓ increases, I is down-sampled by averaging over $2^\ell \times 2^\ell$ square regions (decreasing resolution). In our context, the challenge consists in computing $I^{(\ell)}$ on-the-fly.

Tiling techniques solve this problem by MIP-mapping the tiles offline and by assembling the appropriate resolution at run-time [Wei04]. This does not work in our case because the patches have irregular shapes. So, at low resolution, the square regions may overlap several patches whose contents are known only at run-time. Pre-computing all possible combinations of contents is not tractable either: it would produce C^P tiles, which cannot be stored and pre-filtered.

4. Filtering content exchange

Our method follows the same intuition as [Wei04]: since we cannot assemble I offline and pre-filter it as $I^{(\ell)}$, we pre-filter the data so as to compute $I^{(\ell)}$ on demand at run-time. We achieve this by:

- Pre-filtering the patch-map as a MIP-map $\mathcal{P}^{(\ell)}$ in such a way that the overlap of patches can be recovered at every level ℓ.
- Pre-filtering all contents $I_{p,c}^{(\ell)}$ and packing them in a texture atlas.
- Assembling $I^{(\ell)}$ at run-time from $\mathcal{P}^{(\ell)}$ and the content atlas.

4.1. Pre-filtering the patch-map

The patch-map is encoded as a bitmask of size P: at the higher resolution the p-th bit of $\mathcal{P}^{(0)}(\boldsymbol{x})$ is set to 1 if $\boldsymbol{x}$ belongs to the patch p. Then the MIP-map is computed by using a bitwise-OR instead of an average. Namely, the value $\mathcal{P}^{(\ell)}(\boldsymbol{x})$ at texel $\boldsymbol{x}$ is the bitwise-OR of its four parents in the previous level $\mathcal{P}^{(\ell-1)}$.

As a consequence, the p-th bit of $\mathcal{P}^{(\ell)}(\boldsymbol{x})$ equals to 1 iff the patch p overlaps the texel $\boldsymbol{x}$ which corresponds to a $2^\ell \times 2^\ell$ square at high resolution. These patches are the ones whose contents contribute to the final color $I^{(\ell)}(\boldsymbol{x})$.

4.2. Pre-filtering the contents

To compute the pre-filtered contents $I_{p,c}^{(\ell)}$, we proceed as follows. For each content c of each patch p :

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

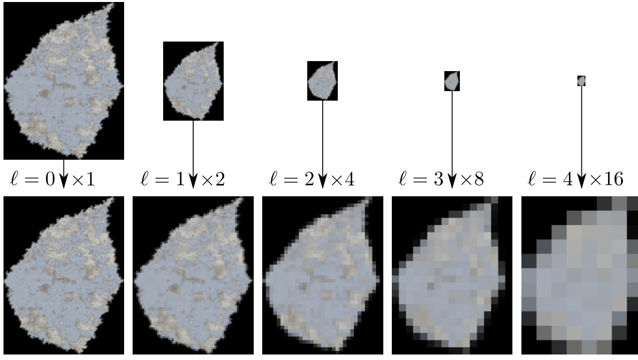

Figure 3: *Each content $I_{p,c}$ is pre-filtered at different levels / resolutions. Top: fixed resolution. Bottom: fixed size.*

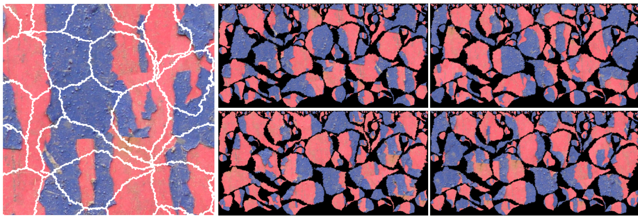

Figure 4: *Packing of the contents. Left: the high resolution ($\ell = 0$) does not need special storage. Right: for each content id c, all low-resolution ($\ell \geq 1$) contents $\{I^{(\ell)}_{p,c}\}_{c,\ell}$ are packed in an atlas.*

- The tile is filled with zeros, except for the patch p, which is filled with content $I_{p,c}$.
- The tile is MIP-mapped using four-to-one standard averaging. Boundary texels are thus blended with the null-valued background (see Figure 3).
- At every level $\ell \geq 1$, the content $I^{(\ell)}_{p,c}$ is cut out and packed in an atlas (see Section 4.4).

4.3. Assembling the filtered output

When rendering the texture, we need to compute I on demand for any level ℓ and at any texel $\boldsymbol{x}$. We compute it as

$$I^{(\ell)}(\boldsymbol{x}) = \sum_{p \in \mathcal{P}^{(\ell)}(\boldsymbol{x})} I^{(\ell)}_{p,\mathcal{C}(p)}(\boldsymbol{x}), \tag{2}$$

where the sum blends all the contents $I^{(\ell)}_{p,\mathcal{C}(p)}$ overlapping $\boldsymbol{x}$ at level ℓ. The patches p involved are recovered from the pre-filtered patch-map $\mathcal{P}^{(\ell)}$. Thus, equation (2) is an exact evaluation of the MIP-map of I, without the need to store it explicitly. Then, standard filtering operations, such as trilinear and anisotropic filtering, can be applied as usually done with a MIP-map.

4.4. Storage

Our method requires storing the patch-map at all resolutions as an array of bitmasks (of P bits for P patches), as well as every resolution of the contents. To limit the impact of irregular patch shapes on the memory, we build an atlas (one for each content identifier c)

Data (N/C)	No filtering (Mb)		Our filtering (Mb)	
	contents	patch-map	contents	patch-map
512^2 / 3	**0.79**	**0.26**	**2.0** (+150%)	**1.4** (+433%)
512^2 / 6	**0.79**	**0.26**	**3.2** (+300%)	**1.4** (+433%)
1024^2 / 3	**3.1**	**1.05**	**8.1** (+150%)	**5.6** (+433%)
1024^2 / 6	**3.1**	**1.05**	**13** (+300%)	**5.6** (+433%)
Asymptotic	$\Theta(N)$		$\Theta(NC)$	

Table 1: *Memory footprint analysis for different input sizes N and different numbers of contents C. The additional cost for the patch-map is fixed (+433%). The cost for the contents is +50% per content, which must be compared to +33% if the packing were perfect.*

which encompasses the low resolution contents $I^{(\ell)}_{p,c}$ of every patch ($\ell \geq 1$, $p \in \mathcal{P}$), as shown in Figure 4. Note that we store only isotropic MIP-map levels, from which anisotropic filtering can be computed. Since contents of the highest resolution do not present the same overlapping issues as those of lower resolutions, they do not need to be stored in this atlas, but can be accessed directly from the input tile using translation offsets, like in [VSLD13, KCD16]. To use this atlas, it is necessary to store the origins of the content in the atlas and of the related patch in the input tile, for each resolution. It takes up a few more kilo-bytes, which is negligible.

4.5. Blending the contents

Patch-based texturing tends to introduce visible seams at patch boundaries. They can be hidden by blending adjacent contents. Our technique can filter the blended contents by making the following changes:

- The patches are dilated, which sets several bits to 1 in $\mathcal{P}^{(0)}(\boldsymbol{x})$ when $\boldsymbol{x}$ is near a boundary.
- Blending weights are defined in overlapping areas. In our tests, we used a simple decreasing distance to the patch boundaries.
- $I_{p,c}$ are weighted accordingly to get the high-resolution contents $I^{(0)}_{p,c}$.

Then our pipeline (Sections 4.1, 4.2, and 4.3) is unchanged, generating the lower resolutions which automatically integrate the blending weights in the filtered contents. This enables to get the blending almost for free: a slight additional cost of memory is due to the storage of blending weights and more terms are involved in the sum of equation (2).

5. Results

Figure 5 presents the results of our filtering technique : it is indistinguishable from the ground truth (middle), while unfiltered results (left) are noisy at low resolution. The accompanying video shows how the filtering reduces flickering artifacts in dynamic scenes. Note that some flickering remains, which is not due to pre-filtering: it is also present in the ground truth and can be further reduced with supersampling or temporal anti-aliasing. Figure 6 shows the improvement of a simple blending, based on the texel distance to the patch boundaries. In regions where seams between different contents remain visible, blending reduces their visual impact. This

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

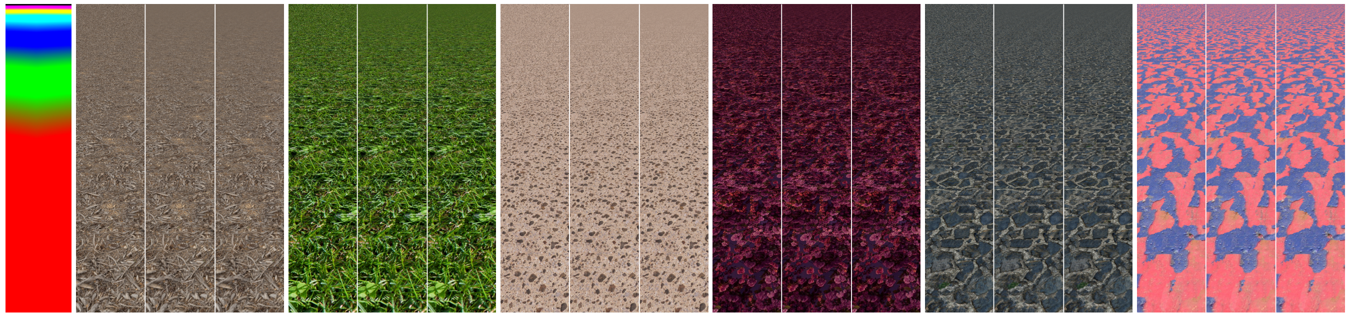

Figure 5: *Results for various examples. Our filtering method (right) is compared to the ground truth (middle) and no filtering (left). The ground truth is computed by an exact filtering of the high resolution. The leftmost view indicates the MIP-map levels used.*

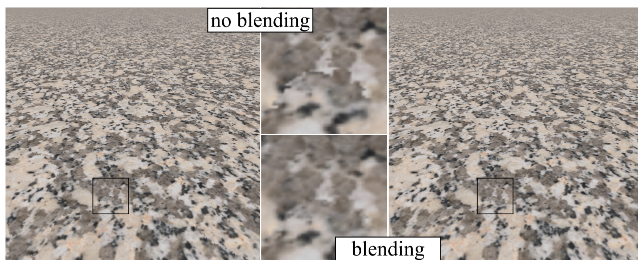

Figure 6: *Rendering of a texture with (bottom / right) and without (top / left) blending at patch boundaries. Seams between different contents are less visible when blending is enabled.*

blending has been enabled for all the results presented in the paper. Although the blending we used is very simple, more complex, content-dependant weights may be used as well.

Table 1 lists the memory usage, which grows linearly with the number N of pixels and the number C of contents. For typical use cases, with a tile of 1024^2 texels and 6 contents, the additional cost is about $5\times$, which makes it possible to generate high-resolution and unbounded textures with less than $20Mb$. We rendered our results on a steep surface covered by $2.6G$ texels ($51.2K \times 51.2K$) using a 800×700 pixels viewport. We enabled trilinear filtering, linear magnification, and anisotropic filtering. Rendering was running at 90 frames per second on an NVidia GTX 1060. Note that these performances could be improved by bypassing equation (2) at very low resolutions (i.e. far away), by lowering the degree of anisotropic filtering (we used 16 samples at most), and by improving the implementation to reduce cache misses.

6. Conclusion and future works

We presented in this paper a filtering technique for on-the-fly patch-based texturing approaches, which deals with the difficult challenge of managing different contents coming from non-adjacent regions of the input texture and randomly chosen at rendering time. Pre-filtering both the contents and the patch-map allows for a runtime evaluation of the resulting texture MIP-map. To achieve this, our method requires to store explicitly the pre-filtered low-resolution contents, as well as the patch-map at each resolution. Yet, the total memory storage still remains acceptable, typically a few tens of Mb for unbounded textures. Moreover, we showed that our algorithm is able to perform blending at patch boundaries by only a slight modification of our pre-processing. It reduces the visual impact of seams potentially appearing in on-the-fly patch-based texturing. This technique performs well on a modern GPU, and can be implemented with standard shader languages, leading to real-time performances.

We are interested in investigating how our filtering technique can be extended. Indeed, it has been especially designed to work for [VSLD13], but cannot be applied directly to [KCD16] due to differences in data representation. We would like to extend the algorithm so as to be able to filter the latter as well. Following the same direction, we think that our algorithm is able to filter tile-based approaches too, by considering the set of tiles as the different contents of a unique square patch, and we would like to conduct experiments to validate this assumption. Finally, we want to investigate the possibility of using our method for more complex data that cannot be linearly filtered, such as normal maps or shadow maps.

References

[CSHD03] Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang tiles for image and texture generation. *ACM Transactions on Graphics*, 22(3):287–294, 2003. 2

[KCD16] Martin Kolář, Alan Chalmers, and Kurt Debattista. Repeatable texture sampling with interchangeable patches. *The Visual Computer*, 32(10):1263–1272, 2016. 1, 2, 3, 4

[LD05] Ares Lagae and Philip Dutré. A procedural object distribution function. *ACM Transactions on Graphics*, 24(4):1442–1461, 2005. 2

[LD06] Ares Lagae and Philip Dutré. An alternative for wang tiles: Colored edges versus colored corners. *ACM Transactions on Graphics*, 25(4):1442–1459, 2006. 2

[RNLL10] Nicolas Ray, Vincent Nivoliers, Sylvain Lefebvre, and Bruno Lévy. Invisible seams. In *Proceedings of the 21st Eurographics Conference on Rendering*, EGSR'10, pages 1489–1496. Eurographics Association, 2010. 2

[SD10] Thomas Schlömer and Oliver Deussen. Semi-stochastic tilings for example-based texture synthesis. *Computer Graphics Forum*, 29(4):1431–1439, 2010. 2

[VSLD13] Kenneth Vanhoey, Basile Sauvage, Frédéric Larue, and Jean-Michel Dischler. On-the-fly multi-scale infinite texturing from example. *Transactions on Graphics*, 32(6):208:1–208:10, 2013. (Proceedings of Siggraph Asia'13). 1, 2, 3, 4

[Wei04] Li-Yi Wei. Tile-based texture mapping on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '04, pages 55–63. ACM, 2004. 2

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.