Motion-coherent stylization with screen-space image filters

Alexandre Bléron
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LJK

Romain Vergne
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LJK

Thomas Hurtut
Polytechnique Montréal

Joëlle Thollot
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LJK

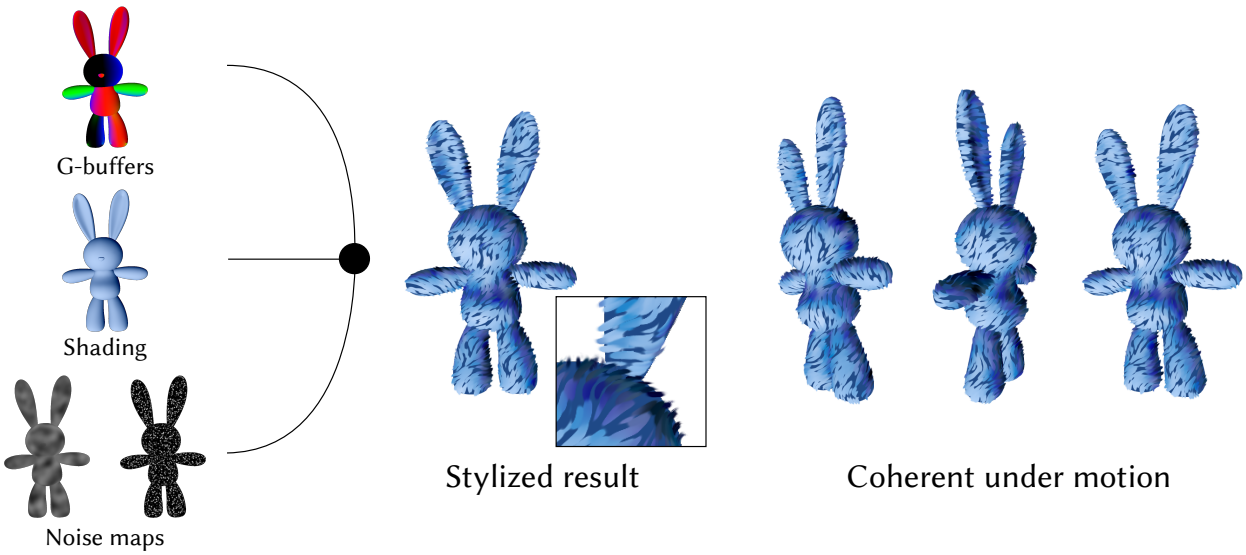

Figure 1: Using standard G-buffers and auxiliary buffers (noise, shading) as input, our pipeline can reproduce stylization effects that extend outside the original rasterized footprint of the object. Visual features produced by the filters stay coherent under motion or viewpoint changes.

ABSTRACT

One of the qualities sought in expressive rendering is the 2D impression of the resulting style, called *flatness*. In the context of 3D scenes, screen-space stylization techniques are good candidates for flatness as they operate in the 2D image plane, after the scene has been rendered into G-buffers. Various stylization filters can be applied in screen-space while making use of the geometrical information contained in G-buffers to ensure motion coherence. However, this means that filtering can only be done inside the rasterized surface of the object. This can be detrimental to some styles that require irregular silhouettes to be convincing. In this paper, we describe a post-processing pipeline that allows stylization filters to extend outside the rasterized footprint of the object by locally "inflating" the data contained in G-buffers. This pipeline is fully implemented on the GPU and can be evaluated at interactive rates. We show how common image filtering techniques, when integrated in our pipeline and in combination with G-buffer data, can be used to reproduce a wide range of "digitally-painted" appearances, such as directed brush strokes with irregular silhouettes, while keeping enough motion coherence.

CCS CONCEPTS

• **Computing methodologies** → **Computer graphics**; *Rendering*; Non-photorealistic rendering;

KEYWORDS

Screen-space filter, temporal coherence, GPU

Expressive '18, Victoria, BC, Canada

ACM Reference format:
Alexandre Bléron, Romain Vergne, Thomas Hurtut, and Joëlle Thollot. 2018. Motion-coherent stylization with screen-space image filters. In *Proceedings of The Joint Symposium on Computational Aesthetics and Sketch Based Interfaces and Modeling and Non-Photorealistic Animation and Rendering, Victoria, BC, Canada, August 17–19, 2018 (Expressive '18),* 13 pages.
DOI: 10.1145/3229147.3229163

1 INTRODUCTION

Expressive rendering strives to reproduce artistic styles by designing algorithms to mimic the artistic creation process with a computer. Depending on the application, these algorithms can use various inputs ranging from photographs, where the only available information is a flat color image, to full 3D scenes, with geometry, animation and lighting information. In this paper our goal is to stylize *3D scenes* by taking advantage of *image filters* traditionally used in image manipulation software and in the post-processing pipeline of animated movies and video games.

Stylizing animated 3D scenes is challenging, since desired properties (temporal continuity, motion coherence and flatness) cannot all be satisfied at once [Bénard et al. 2011]. For instance, following the motion field of a scene while keeping a 2D visual impression leads to an inherent contradiction: when style marks are attached to the surface of 3D objects (*e.g.* using mapped textures or anchored strokes), the resulting motion is coherent but the flatness is broken because the density and size of style marks will vary in screen-space depending on the location of the camera (zooming) or of the surface orientation (foreshortening). Conversely, attaching style marks to the 2D screen ensures all these statistics to be stable, but also leads to the well known *shower-door* effect, breaking the motion coherence.

In this work, we present a new compromise, where style marks are applied in screen space with behaviors controlled by 3D scene properties (stored in G-buffers), to ensure a high level of temporal continuity, motion coherence and flatness. The style marks are created with kernel-based 2D image filters that allow a wide range of digitally painted 2D flat looks to be obtained. In contrast to many-primitive methods, such as stroke-based rendering, they are well suited to per-pixel parallel evaluation on a GPU, which makes this approach very efficient and usable in interactive contexts.

Working in image space with G-buffers is a well-studied and widespread technique. One of its advantages is that no modification of the original geometry of the scene is required. This makes such an approach easily compatible with any compositing pipeline. However, a current limitation with image filters is that they cannot easily alter the rasterized silhouette of an object in a coherent way. Yet, irregular silhouettes contribute greatly to the perceived flatness of the look, and are a key aspect of some styles: a plausible painted fur should extend outside the silhouette of the object.

When targeting contours, filters usually alter them incoherently with respect to the motion of individual objects. As such, they are subject to the shower-door effect. Motion coherence can be improved by using 3D geometric information available in G-buffers (*e.g.* orient the marks according to surface normals, colorize them according to the shading color, etc.). However, current techniques that exploit this information are limited to filling the inside of the rasterized footprint where the geometric data is well-defined, and cannot affect the contours.

In this paper, we address this limitation by proposing a way to *inflate* the data contained in G-buffers with a controllable radius in order to have motion-coherent data outside the original footprint of the object, which is then used to drive stylization filters. This combination of motion coherence and footprint extension necessitates a careful handling of internal silhouettes because different styles can overlap at these locations. The proposed approach explicitly deals with this issue, avoiding visual artifacts. To summarize our contributions, we propose a method to stylize animated 3D objects that:

- Uses only screen-space data contained in G-buffers;
- Is fully evaluated on the GPU;
- Ensures strong motion coherence;
- Enables the design of various styles that extend beyond the rasterized footprint of the object.

2 RELATED WORK

Stylization of 3D animated objects is a widely studied subject among the expressive rendering community. We refer the reader to Bénard *et al.* [2011] for a state-of-the-art on this topic.

Depending on the use of the geometrical information, stylization methods can operate in image-space – where no 3D information is available – or in object-space. While a vast amount of image filters are available for image stylization, they do not extend easily to animated objects. In image-space the only way to compute the object motion is to compute the optical flow of the animation. This information can then be used to control the change of the filters over time to ensure some motion coherence (*e.g.* [Bousseau et al. 2007; Lu et al. 2010]). Yet it suffers from the difficulty to compute a correct and noise-free optical flow.

On the other hand, operating in object-space gives access to the 3D geometry and thus to the exact motion field of the object. However working directly in 3D space can be costly as the computation time depends on the geometrical complexity of the 3D object. That's the reason why several methods have been designed to work in 2.5D space allowing for an access to 3D information but at the image resolution. Saito & Takahashi first introduced this class of methods [1990] with G-buffers, screen-space images containing geometry information about the scene, and demonstrated their usefulness for various rendering purposes, such as shading or contour extraction and enhancement. G-buffer based methods have since been ported to GPU architectures, as first demonstrated by Nienhaus and Döllner [2004a], and are now widely used in the industry, in the compositing stage of most animation software. Our approach fits in this category: it does not require any modification of the base geometry but rather relies only on G-buffers and can consequently be easily integrated in existing deferred renderers or compositing software as a simple set of post-processing operations.

2.1 Stylizing outside the object contours

Stylizing a 3D object mostly consists in adding visual details, such as brush strokes, pigment or paper grain, fur, glow, etc. We focus in this paper on region stylization as opposed to the stylization of contours. Still our approach will allow to produce effects located near the silhouettes but we do not target styles that need contour parameterization.

Stroke-based approaches. Stroke-based approaches have been introduced by Meier [1996]. They rely on distributed anchor points on the 3D objects used to render small 2D primitives in image-space. The coherence of the motion of the strokes with the motion of the depicted object is directly ensured by the anchoring in 3D. However, maintaining constant density and size of strokes in image-space

requires to add or remove strokes along the animation, leading to temporal discontinuities. Most of the previous works in this category try to improve the continuity by carefully distributing the anchor points, and by blending strokes that appear or disappear in a meaningful way [Kalnins et al. 2002; Kowalski et al. 1999; Lu et al. 2010; Vanderhaeghe et al. 2007]. The *Overcoat* system, proposed by Schmid *et al.* [2011] circumvents this issue by allowing an artist to embed entire brush strokes anywhere in 3D space around a proxy object. These stroke-based approaches are well-suited for painterly styles but are costly when the number of strokes increase to reproduce higher frequency effects such as watercolor.

Texture-based approaches. To avoid temporal discontinuities and decrease the cost, texture-based approaches rely on texture mapping. The added details are contained in the texture and the mapping ensures a perfect motion coherence without temporal discontinuity. The drawbacks of these approaches are first their strong 3D look due to perspective distortion, and second, the impossibility to add textured details outside the object contour. Most of the previous works in this category try to augment the perceived flatness of the texture-mapped objects [Bénard et al. 2009, 2010]. A possible solution that allows stylization to extend outside object boundaries is to use 3D volumetric textures rendered with ray-marching [Kajiya and Kay 1989; Perlin and Hoffert 1989]. Later works focus on improving the performance of this approach to real-time frame rates by approximating volumes with layers of additional geometry [Decaudin and Neyret 2009; Lengyel et al. 2001; Meyer and Neyret 1998]. We refer the reader to the survey of Koniaris *et al.* [Koniaris et al. 2014] for a comprehensive overview of this type of approaches. A similar technique specialized for painterly appearances has been proposed by Imhof *et al.* [2015], for accelerating the rendering of *Overcoat* paintings. However, these methods retain a strong 3D look, and are poorly suited to reproduce flat styles.

Geometry modification. Another option is to modify the geometry on the fly. In [Botkin 2009], perturbation of the vertices of a 3D model is used to simulate a painterly effect. Displacement mapping techniques, combined with hardware tessellation, can also be used to efficiently add surface details to a 3D object. Nienhaus and Döllner [2004b] propose to render an inflated version of the geometry to allow coherent warping effects outside an object's silhouette. In this paper we propose a similar method to extend geometrical information away from the silhouettes. Yet unlike the previously described techniques, our approach does not need to rasterize any additional geometry. Our technique relies solely on image-space filters operating on G-buffers to produce different styles that are coherent outside the footprint of the object.

2.2 Noise textures

We employ an approach based on solid procedural noise to generate motion-coherent random variations. However, unlike previous methods, the noise is not composited directly onto the object but instead is used as an auxiliary map that drives the 2D image filters. Most texture-based stylization methods use textures as a layer composited on top of a first rendering of the 3D object. Several layers of textures can be composited on top of each other, as was previously done in watercolor rendering [Bousseau et al. 2006] but the range of resulting effects remains limited. However textures, and especially noise textures, have been widely used in photorealistic rendering to modulate not only the object color but also normals, positions, etc. A detailed survey of procedural noise functions has been conducted by Lagae *et al.* [2010]. Of particular interest to us are noise models presenting thin elongated features that can be used to mimic the appearance of brush strokes or fur. To synthesize those kind of textures, 2D techniques such as sparse convolution noise can be used. However, they cannot be used reliably with a 3D parameterization. Instead, image filtering techniques have been used to filter noise patterns in order to obtain these elongated features. For instance, Lum and Ma [2001] used line integral convolutions [Cabral and Leedom 1993] to reproduce watercolor washes. A variant of this technique has also been used to generate *glass patterns* [Papari and Petkov 2009], which have also been remarked for their painterly quality. We demonstrate the use of such filters in our pipeline, in combination with 3D solid noise, and driven by geometrical data from G-buffers. We show how they can be used to reproduce "digitally painted" styles, that range from visible discrete strokes to more continuous appearances.

To summarize, our system can be seen as a hybrid between a texture-based and an image-space approach: we use procedurally generated motion-coherent random variations both as a base for subsequent filtering in image-space and also as a way to control certain aspects of the filters. This hybrid approach allows us to strike an advantageous balance between flatness and motion coherence.

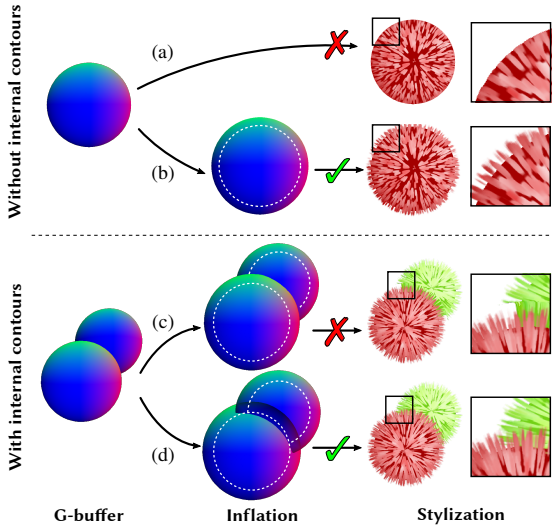

Figure 2: Overview of stylization challenges in screen-space. (a) A filter requiring data contained in G-buffers will produce discontinuities as this information is not available outside the silhouettes. (b) Locally *inflating* G-buffers along the normals allows extending the filter effect beyond the original object footprint. (c) When dealing with internal contours, a naive screen-space inflation fails where overlapping surfaces occur because a single surface is considered for each pixel. (d) Our approach uses a local segmentation in order to inflate and stylize each overlapping surface part.

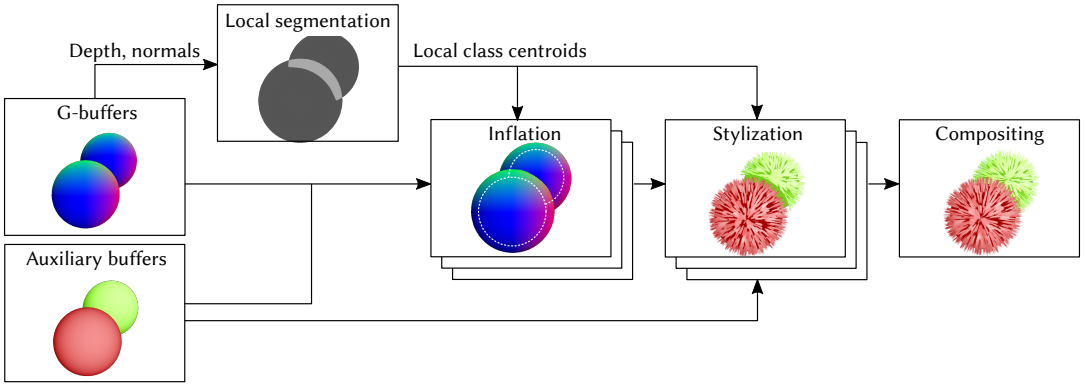

Figure 3: High-level overview of our pipeline. Starting from standard G-buffers (depth, normal, positions, etc.) and auxiliary buffers (albedo, shininess, etc), a local segmentation identifies regions where multiple surface parts overlap each other when inflated. To do so, we employ a local soft K-means segmentation based on depth and normal information. For each pixel, we obtain K weighted clusters, representing different overlapping surface parts under the filter window. The inflation algorithm relies on each cluster independently to extend any kind of data for each (overlapped) part. The stylization, based on local 2D filters, is also evaluated on each cluster using the necessary input data (inflated normals, colors, etc) as guidance. Note that the stylization is then evaluated multiple times on pixels that overlap different parts of an object. The last step consists in a compositing pass, where overlapped styles are blended to obtain the final result.

3 MOTIVATION AND OVERVIEW

To illustrate the motivation behind the pipeline proposed in this paper, let's consider a basic stylization of spheres, as shown in Figure 2. In this example, the paint strokes are obtained by simply applying a Line Integral Convolution (LIC) [Cabral and Leedom 1993] on colored spots distributed on the object surface. The spots are created using a thresholded 3D cellular noise, parameterized by surface positions (stored in a G-buffer), to ensure robust motion coherence. The LIC is then guided by screen-space surface normals to transform the spots into elongated paint strokes. The whole process is evaluated locally: the result of the filter for one pixel at location $\mathbf{p}$ is a function of all pixels under a neighborhood centered on $\mathbf{p}$ that we call the *filter window*.

With a naive approach (a), the stylization stops abruptly outside the object: the filter cannot be evaluated because normals are not defined at these image locations. Our first contribution is to fill the missing information in such a way that the object looks *inflated* (b). To that end, we propose a new image filter that locally extends any G-buffer whithout breaking the motion coherence of the original 3D scene. Once the necessary information is available everywhere under the window of the stylization filter (noise and normals in this example), the strokes can naturally extend outside the rasterized footprint of the object.

However, this *inflation* is ambiguous when parts of an object are overlapping each other (*e.g.* when dealing with internal contours). This is illustrated with two spheres in Figure 2(c), where we extend the front-most surface information of the G-buffers. In that case, the red strokes are coherent, but the green ones produce a spatial discontinuity because the filters do not have access to the information from the second sphere behind. Our solution (d) is to keep track, at each pixel, of all the inflated surface parts that overlap. We then perform the stylization separately for each of those. For that we propose a local segmentation method, based on soft K-means [Bauckhage 2015], that finds and classifies the surface parts that will overlap when inflated. This information is handed over to the inflation and stylization passes so that they can filter each surface part independently.

Figure 3 provides an overview of our pipeline. In the following sections, we detail the local segmentation and screen-space inflation algorithms, and how to integrate image filters into our pipeline.

4 INFLATION AND SEGMENTATION

In this section, we show how G-buffers can be locally inflated and segmented to provide motion-coherent data inside and outside the footprint of the rasterized objects to the stylization pass.

4.1 Object Inflation

We extend G-buffer data by mimicking a real inflation of the original 3D object, that is, displacing vertices along their associated normals before rasterization. Let us consider a pixel $\mathbf{p_0}$, being the center of a circular neighborhood of radius r (the filter window) in a G-buffer. Our goal is to look for pixels $\mathbf{p}$ in the neighborhood that would be at the $\mathbf{p_0}$ position if the corresponding object had really been inflated before being rasterized. Intuitively, this can be done by checking if, when a point $\mathbf{p}$ is displaced along the projected screen-space normal with a given radius, the point equals $\mathbf{p_0}$. In other words, for every pixel $\mathbf{p_0}$, we look for points $\mathbf{p}$ such that $\mathbf{p_0} = \mathbf{p} + r\mathbf{n}(\mathbf{p})$, where $\mathbf{n}(\mathbf{p})$ is the projected screen-space normal at point $\mathbf{p}$. Points $\mathbf{p}$ that satisfy this equation can be seen as *anchor points* as they can be used to inflate any G-buffer (by simply fetching the data at point $\mathbf{p}$ instead of $\mathbf{p_0}$).

In practice, we work on discrete buffers, and this equation is usually not satisfied. Instead, we assign a weight $w_\mathbf{p}$ to each point under the filter window that evaluates the *fitness* of the point for inflation. We define it as a Gaussian of the distance between the

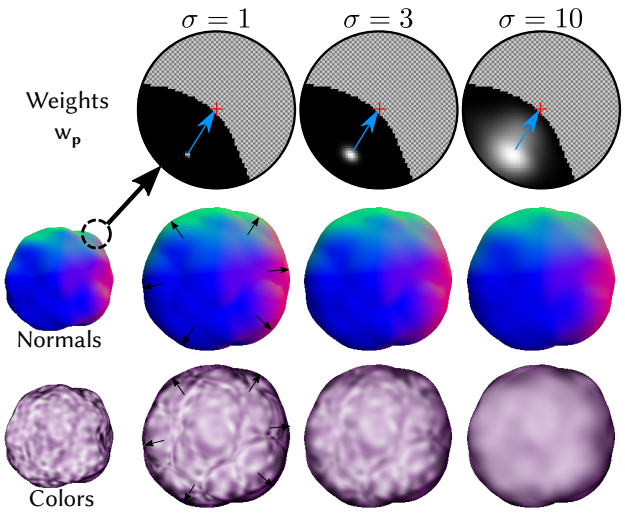

Figure 4: Inflation: (top) effect of σ on the size of the clusters, (middle) inflated normal map, (bottom) inflated color. Low σ values make the averaged region more localized, as few normals are considered fit for inflation. Conversely, higher σ values result in a wider region being averaged when inflating G-buffer data. This can lead to more stable animations, at the price of a loss of detail on high-frequency data, as shown with the noisy color image in the bottom row.

warped pixel $\mathbf{p} + r\mathbf{n}(\mathbf{p})$ and the filter window center $\mathbf{p_0}$:

$$w_{\mathbf{p}} = \exp\left(\frac{-\|\mathbf{p_0}, \mathbf{p} + r\mathbf{n}(\mathbf{p})\|^2}{\sigma^2}\right), \tag{1}$$

where σ controls the tolerance towards the displacement error.

The inflated data at $\mathbf{p_0}$ is then calculated by averaging the data under the filter window S, weighted by $w_{\mathbf{p}}$:

$$I_{\text{inflated}} = \frac{\sum_{\mathbf{p}\in S} w_{\mathbf{p}} I(\mathbf{p})}{\sum_{\mathbf{p}\in S} w_{\mathbf{p}}}, \tag{2}$$

where $I(\mathbf{p})$ represents the buffer data being inflated (depth, normals, colors, etc.). A visualization of weights and inflated data is provided in Figure 4.

4.2 Weighted Clustering

When different parts of the surface are overlapping in the inflated version (due to internal contours, multiple silhouettes, T-junctions, etc.), the relation $\mathbf{p_0} = \mathbf{p} + r\mathbf{n}(\mathbf{p})$ can be satisfied by more than one anchor point $\mathbf{p}$ under the filter window. This is shown in Figure 5, where several clusters of weighted points stand out. In that case, we would like Equation 2 to be applied independently on each cluster to ensure that each part is correctly inflated. Figure 5 illustrates what happens with naive approaches, in contrast to our solution, on a style similar to the one shown in Figure 2, where a LIC is oriented along surface normals to "smudge" colors outside the silhouettes of the object. In the second row, the normals of all clusters are averaged together, resulting in an unnatural direction field for guiding the style. In the third row, only the weights of the top-most surface are considered. As a unique direction is computed per pixel, only the front surface will be correctly stylized, producing discontinuities when styles of front and back surfaces should overlap. Our solution is presented in the last row, where we locally segment and stylize each overlapping part independently before recomposing the final result.

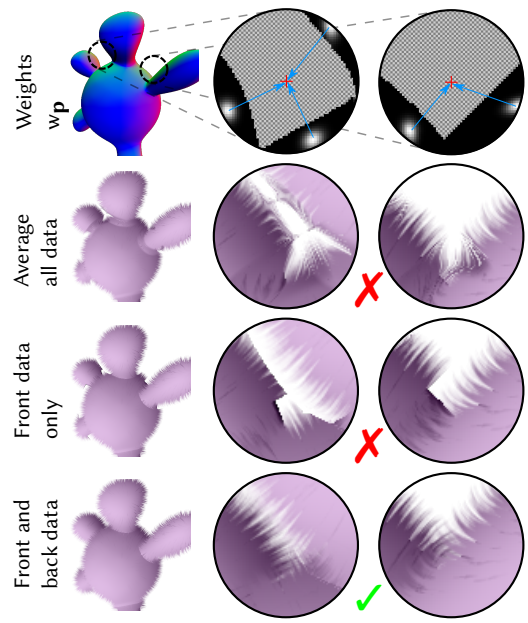

Figure 5: Handling overlapping inflated data on a style guided by screen-space normals. (Top) normals and weights obtained on two filter windows. (Second) averaging all data leads to unnatural direction fields. (Third) averaging only front-most surfaces produces discontinuities in the resulting stylization. (Bottom) in our approach the weight maps are first segmented into clusters to stylize each part independently before compositing them.

Our segmentation is done locally over each filter window. It is based on a weighted soft K-means [Bauckhage 2015] since it is relatively fast and well suited to parallel GPU evaluation. The data to be segmented takes the form $d(\mathbf{p}) = (x, y, z)$, where (x, y) is the position of the pixel and z is the depth of the surface at this position. Note that all components are remapped into the range $[-1; 1]$ so that they have the same impact on the segmentation. Each data point is weighted by $w_{\mathbf{p}}$ (Equation 1). Intuitively, the K-means will segment different clusters of weights (using $w_{\mathbf{p}}$) and thus clearly distinguish pixels that belong to different parts of a surface in a filter window. Adding z enforces this segmentation and separates parts that contain the same normal with different depths.

Initialization. The initial clusters are distributed linearly along the z dimension inside the filter window:

$$\mu_i^{(0)} = \left(0, 0, z_{\min} + i\frac{z_{\max} - z_{\min}}{K}\right), \tag{3}$$

where $\mu_i^{(0)}, i \in [0, K-1]$ is the cluster i at step 0 and $z_{\min/\max}$ are the minimal and maximal surface depths under the filter window. We found this simple and fast initialization to work well in most

cases, as overlapping inflated parts usually tend to have significantly different depths.

Class membership weights. Each data point is soft-assigned to a cluster, meaning that each of them can partially belong to all classes. The class membership weight of the data point at $\mathbf{p}$ to class k is noted $m_k(\mathbf{p})$, with $\sum_{k=0}^{K-1} m_k(\mathbf{p}) = 1$. It can be interpreted as the probability that $\mathbf{p}$ effectively belongs to cluster k. We define the membership weights $m_k(\mathbf{p})$ for the data point $d(\mathbf{p})$ as the soft-max distance from the point to the class centroid, as suggested in [Bauckhage 2015]:

$$m_k(\mathbf{p}) = \frac{e^{-\beta \|d(\mathbf{p})-\mu_k\|^2}}{\sum_{k=0}^{K-1} e^{-\beta \|d(\mathbf{p})-\mu_k\|^2}}, \tag{4}$$

β being the stiffness parameter, controlling the segmentation sensitivity: a higher stiffness value will result in more clearly separated classes, but with an increasing risk of over-segmentation of nearby points.

Centroid update. For each iteration of the algorithm, we update the class centroids $\mu_k \in \mathbb{R}^3$ from the weighted data points under the filter window S using the following formula:

$$\mu_k^{(t+1)} = \frac{\sum_{\mathbf{p}\in S} w_{\mathbf{p}} m_k^{(t)}(\mathbf{p}) d(\mathbf{p})}{\sum_{\mathbf{p}\in S} w_{\mathbf{p}} m_k^{(t)}(\mathbf{p})}. \tag{5}$$

We stop the process when a specified maximum number N of iterations is reached. We used $N = 8$ for all the examples shown in the paper. The result of the clustering is a fixed K number of classes represented by their centroid $\mu_k = (\mu_{k,x}, \mu_{k,y}, \mu_{k,z})$, each representing a local surface part. We finally sort those clusters by depth, μ_0 being the closest to the camera, and μ_{K-1} the farthest.

Inflation. Following the clustering, the inflated G-buffer data for class k can be calculated by modifying Equation 2 as follows:

$$I_{\text{inflated,k}} = \frac{\sum_{\mathbf{p}\in S} m_k(\mathbf{p}) w_{\mathbf{p}} I(\mathbf{p})}{\sum_{\mathbf{p}\in S} m_k(\mathbf{p}) w_{\mathbf{p}}}. \tag{6}$$

Figure 6 shows the result of our segmentation on three local windows. We used $K = 3$ in all the results shown in this paper as there is rarely more than 3 distinct surface parts under the filter window, but more clusters could be used if needed (especially when using very large neighborhoods). Thanks to the soft assignement used in the K-means algorithm, multiple clusters can overlap when the number of surface parts is less than K. For instance, this is shown in 6(a). All clusters are located at the same place then all pixels equally belong to the three of them. The corresponding inflation on a normal map is shown on the right, where we α-blend the resulting normals when multiple parts were overlapping.

5 STYLIZATION

In this section, we describe how 2D image filters used for stylization are integrated in our pipeline.

5.1 Assigning clusters to individual pixels

We want to apply sylization filters independently for each distinct surface part under the filter window. So we need to be able to mask out all pixels not belonging to the surface part being considered, as

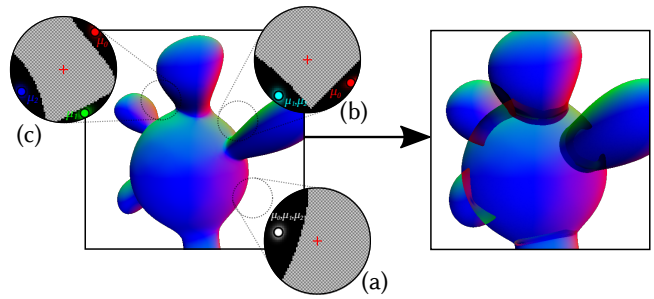

Figure 6: (Left) K-means segmentation results with K = 3. Under filter windows where only one or two distinct surface parts are present, the clustering associates multiple centroids to the same surface part. (Right) visualization of the inflated normal map. Regions with overlapping inflated normals are represented by blending them on top of each other.

otherwise data belonging to different surface parts may be averaged together (*e.g.* the color of two different surfaces may bleed onto each other, which is not the desired result).

Our K-means segmentation locates anchor points on these surface parts, but the class membership weights m_k do not allow us to directly say whether a pixel belongs to a given surface part. Indeed, as seen in Figure 4, the weights quickly decrease according to the distance between the pixel and the class centroid. For most pixels we have $w_{\mathbf{p}} \approx 0$, meaning that their normals are all pointing "away" from the window center $\mathbf{p_0}$. Still it does not mean that they do not belong to a surface part, it rather means than when inflated they do not reach the window center. However, the class membership weights m_k cannot be relied upon for pixels with a low weight in the segmentation (*i.e.* $w_{\mathbf{p}} \approx 0$).

Instead of using the class membership weights m_k, our strategy for deciding whether a pixel belongs to a surface part detected by the K-means is to simply consider the unnormalized depth differences and the normal deviations between the current pixel and the centroid that corresponds to the surface part. The pixel is considered as belonging to the surface part if this difference is under a certain threshold. In the end, pixels not assigned to any previously detected surface part are put in a so-called *residual class*, and treated separately in the stylization.

The depth and normal deviations $\Delta_{\text{z,k}}(\mathbf{p})$ and $\Delta_{\text{n,k}}(\mathbf{p})$ for a given point $\mathbf{p}$ with respect to centroid μ_k are defined as follows:

$$\Delta_{\text{n,k}}(\mathbf{p}) = \exp\left(-\sigma_n \operatorname{acos}\left(\mathbf{n}(\mu_k) \cdot \mathbf{n}(\mathbf{p})\right)\right), \tag{7}$$

$$\Delta_{\text{z,k}}(\mathbf{p}) = \exp\left(-\sigma_z \left|z(\mu_k) - z(\mathbf{p})\right|\right), \tag{8}$$

with $\mathbf{n}(\mu_k)$ the approximate normal associated with class k, and $z(\mu_k)$ the screen-space depth of the centroid. We note $\mathbf{n}(\mathbf{p})$ the screen-space normal at $\mathbf{p}$. As either $\Delta_{\text{n,k}}$ or $\Delta_{\text{d,k}}$ may be more suited to discriminate between classes, depending on the shape of the object in the filter window, we take the minimum of the two:

$$\Delta_{\text{k}}(\mathbf{p}) = \min(\Delta_{\text{n,k}}(\mathbf{p}), \Delta_{\text{z,k}}(\mathbf{p})). \tag{9}$$

The σ_n and σ_z parameters control the tolerance of the classification to depth and normal differences, respectively. They are scene-wide parameters and can be adjusted by the user as needed.

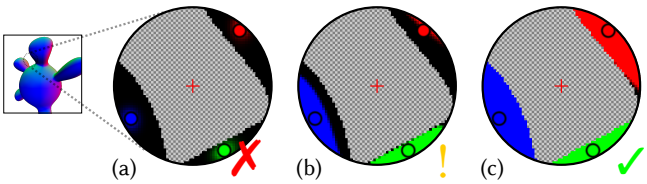

Figure 7: Extending the initial segmentation to all pixels under the filter window. Red: class 0; green: class 1; blue: class 2. (a) The initial K-means segmentation is only reliable for pixels with significant weights $w_{\mathbf{p}}$. (b) Prior to adjusting σ_n and σ_z to the current scene, the class assignments may be imprecise or incomplete (black parts are unclassified regions). (c) Adjusted σ_n to increase the tolerance to normal variations with respect to the normal at the cluster center. The distinct surface parts are properly detected and classified.

Note that one of σ_n and σ_z can be set to zero to disable the influence of the normal variations and depth variations, respectively. The membership of a pixel to class k, noted m'_k, is obtained by thresholding the final weight Δ_{k}:

$$m'_k = \begin{cases} 0 & \text{if} \quad \Delta_{\mathrm{k}} < 0.5 \\ 1 & \text{otherwise} \end{cases} \tag{10}$$

It is important for the filters that as much pixels as possible under the filter window are properly assigned to a matching surface part. It is up to the user to ensure that the surface parts are properly segmented by adjusting the σ_n and σ_z parameters. Figure 7 illustrates this process. In Section 7, we discuss guidelines on the input geometry and filter window size to avoid over-segmentation. Note that, as with the K-means segmentation, the resulting classes can overlap (a pixel under the filter window can belong to more than one class), and thus $\sum_{k=0}^{K-1} m'_k(\mathbf{p}) \neq 1$.

Residual class. Pixels that have not been assigned into any previously detected class (i.e. $\forall k, m'_k = 0$) are put in the *residual class*. They represent all pixels for which we could not assign a surface part previously detected by the K-means segmentation. Still, this residual class needs to be taken into account by the filters to avoid holes in the stylized result, so it is treated similarly to the classes detected by the segmentation. We define the membership weight of a pixel $\mathbf{p}$ to the residual class, $m'_{\text{residual}}(\mathbf{p})$, as:

$$m'_{\text{residual}}(\mathbf{p}) = \max\left(0, 1 - \sum_k m'_k(\mathbf{p})\right). \tag{11}$$

The residual class holds all surface parts that contain no anchor point under the filter window (often, because their anchor points are occluded). Thus, we cannot obtain coherent data for those surfaces by inflation. Nevertheless, extended G-buffer data for the residual class, required by the filters, is still calculated as the average of all its pixels:

$$I_{\text{inflated,residual}}(\mathbf{p}) = \frac{\sum_{\mathbf{p}\in S} m'_{\text{residual}}(\mathbf{p}) I(\mathbf{p})}{\sum_{\mathbf{p}\in S} m'_{\text{residual}}(\mathbf{p})}. \tag{12}$$

5.2 Stylization filters

In this section, we describe several concrete examples of stylization filters that we use to generate the results presented in Section 6. We show how these filters can be made motion-coherent and silhouette-aware with relative ease inside our pipeline. The stylization filter is calculated separately for each class, meaning all pixels that do not belong to the surface part being considered are masked out during evaluation. The stylization filter is also calculated for the residual class, resulting in a total of $K + 1$ evaluations per pixel. The final result will be obtained by blending each stylized class on top of each other.

Auxiliary buffers. Our styles are based on several buffers commonly used in image-space stylization approaches. The simplest one is the color $I^{\text{color}}(\mathbf{p}) \in [0; 1]^4$. Depending on the desired style it can be a flat color, a texture mapped on the object, a simple shading, or a more complex result obtained with standard procedural texturing techniques. All color values in the pipeline are considered to be in premultiplied alpha space: given a color $C = (c_r, c_g, c_b, c_a)$ of opacity c_a, $wC = (wc_r, wc_g, wc_b, wc_a)$ is the same base color with opacity wc_a.

In most styles, we use 3D procedural noises $I^{\text{noise}}(\mathbf{p}) \in [0; 1]$ as a basis for generating motion-coherent procedural details. It is generated in post-processing from the object-space position buffer and relieves the users of the burden of providing their own noise-like texture on the object, although this workflow is also possible. Finally we use 2D direction flows $I^{\text{flow}}(\mathbf{p})$ derived from available G-buffers (normals, tangents, curvature, etc.).

All these buffers can be inflated when needed, allowing to compute filters in a controllable radius outside the original rasterized footprint of the object, with correct overlaps.

Line Integral Convolution filter. A typical stylization filter is the Line Integral Convolution (LIC) [Cabral and Leedom 1993] that can be used to produce fibers like textures for fur or brush strokes. This class of filter has previously been used for stylization purposes [Lum and Ma 2001; Papari and Petkov 2009].

To reproduce a painterly appearance, we multiply the color input I^{color} by the noise texture I^{noise}, modulating the opacity of the color input. The intent is to mask out colored spots in the original color input, that are then spread along the provided flow field by the LIC filter, creating elongated color flats reminiscent of brush strokes.

For the filter window centered at $\mathbf{p_0}$, the resulting filtered value for class k is defined by:

$$C_k(\mathbf{p_0}) = \sum_{s=-L}^{L} \frac{f(s/L) m'_k(\kappa(s)) I^{\text{noise}}(\kappa(s)) I^{\text{color}}(\kappa(s))}{f(s/L)}, \tag{13}$$

with L being the length of the convolution path, and $f : [-1; 1] \rightarrow [0; 1]$ a convolution profile used to control the spread of the colored spots. The per-pixel class membership weights $m'_k(\kappa(s))$ are used to mask out all pixels outside of the considered classes. The convolution path $\kappa(s)$ is defined as:

$$\kappa(s) = \mathbf{p_0} + s \times I^{\text{flow}}_{\text{inflated,k}}(\mathbf{p_0}), \tag{14}$$

i.e. a straight line oriented along the inflated flow. The length of the convolution path should be chosen so as to avoid going outside the filter window.

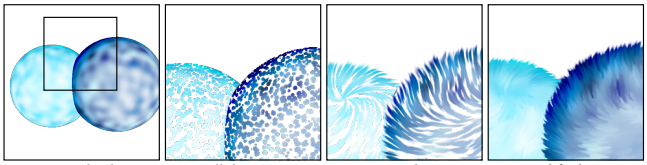

Figure 8: Comparison of the original and our modified formulation of the LIC filter.

Modified Line Integral Convolution filter. Due to the opacity modulation of the color input, the result of the LIC is semitransparent. Standard procedural texturing techniques can be used to process the opacity of the result in order to produce less washed-out strokes. Alternatively, we propose a minor variation in the normalization term in Formula 13 to fill the opacity gaps between the strokes, producing a more continuous painterly appearance:

$$C'_k(\mathbf{p_0}) = \sum_{s=-L}^{L} \frac{f(s/L)m'_k I^{\text{noise}}(\kappa(s)) I^{\text{color}}(\kappa(s))}{f(s/L)\left((1-m'_k) + m'_k\, I^{\text{noise}}(\kappa(s))\right)} \tag{15}$$

with $m'_k = m'_k(\kappa(s))$. A comparison of the resulting appearances is shown in Figure 8.

LIC control. The number of strokes and their weights can be controlled through the noise parameters. In practice, we used a thresholded cellular noise [Worley 1996] to reproduce most stroke-like appearances. This noise masks out round colored spots that serve as "seed points" for stroke generation. Their size and smoothness are controlled through the thresholding parameters. The resulting aspect of the strokes can also be controlled through the integration profile function f. In combination with the modified LIC filter in Equation 15, we found that a gaussian integration profile with an user-adjustable width α (*i.e.* $f(x) = e^{-\frac{x^2}{\alpha^2}}$) provides intuitive control over the resulting appearance: low values of α tend to produce more clearly separated color flats, while increasing α results in a flatter profile that tends to blend the color of neighboring strokes together, resulting in a more continuous appearance.

Brush convolution filter. The LIC filters previously described can be generalized by replacing the integration path by an arbitrary user-provided footprint. This is inspired by Implicit Brushes [Vergne et al. 2011], and sparse convolution noise [Lewis 1984]. When used in tandem with thresholded cellular noise, the brush pattern is replicated on every cell, and can be used to reproduce a variety of different styles with proper handling of silhouettes. Examples of such styles are given in Figure 12. The result of those filters can be processed afterwards with standard procedural texturing techniques to refine the style (thresholding, color mapping, etc.).

5.3 Blending of intermediate filter results

The filter results for each individual class $C_k, k \in [0; K-1]$, and the filter result for the residual class, C_r, need to be composited together in depth order to get the final stylized result. The classes $k \in [0; K-1]$ are sorted by depth. As the presence of a residual class is often due to occlusions by classes on top, we consider that

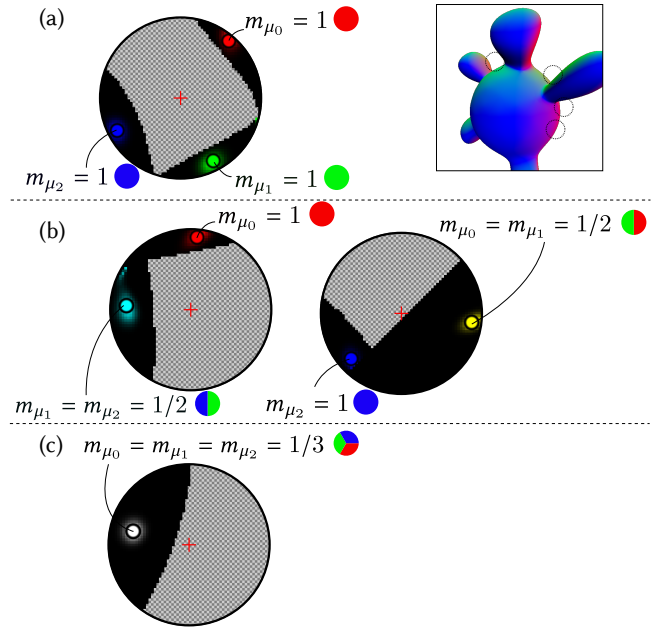

Figure 9: Contribution weights for $K = 3$ under 3-cluster (a), 2-cluster (b) and single (c) configurations. Class 0,1 and 2 associated colors are respectively red, green and blue. The contribution weight of the middle class m_{μ_1} can be used to identify the current configuration.

the residual class is behind all other classes during compositing. Thus, the final compositing order should be first C_{K-1} composited over C_r, then C_{K-2} composited over the previous result, and so on up to C_0.

However, as the segmentation algorithm may produce overlapping classes when the number of effective surface parts are less than K under the classification window, we must avoid compositing twice filter results that correspond to the same surface part, since it may result in wrong colors for semitransparent styles. A robust way to handle this is to first identify sets of overlapping classes, average the filter results that belong to a set of overlapping classes and then blend the result of each set.

To detect if a class k overlaps with another, we consider $m_{\mu_k} = m_k(\mu_k)$, the class membership weights of the centroids. If $m_{\mu_k} < 1$ then the centroid is also partially assigned to another class, meaning an overlap. We therefore can consider m_{μ_k} as the contribution weight of class k. Due to classes being sorted by depth, overlaps can only happen between successive classes. Note that the residual class never overlaps with any other class, so it is unconditionally composited.

We describe here for $K = 3$, but generalizable to higher values of K, an enumeration of all the possible class configurations using the middle class weight m_{μ_1} to distinguish between the four cases, shown in Figure 9. Let C_f be the final composited result,

- if $m_{\mu_1} \approx 1$ then all classes are likely distinct, and all intermediate results should be composited on top of each other:

$$C_f = C_0 \textbf{ over } C_1 \textbf{ over } C_2 \textbf{ over } C_r$$

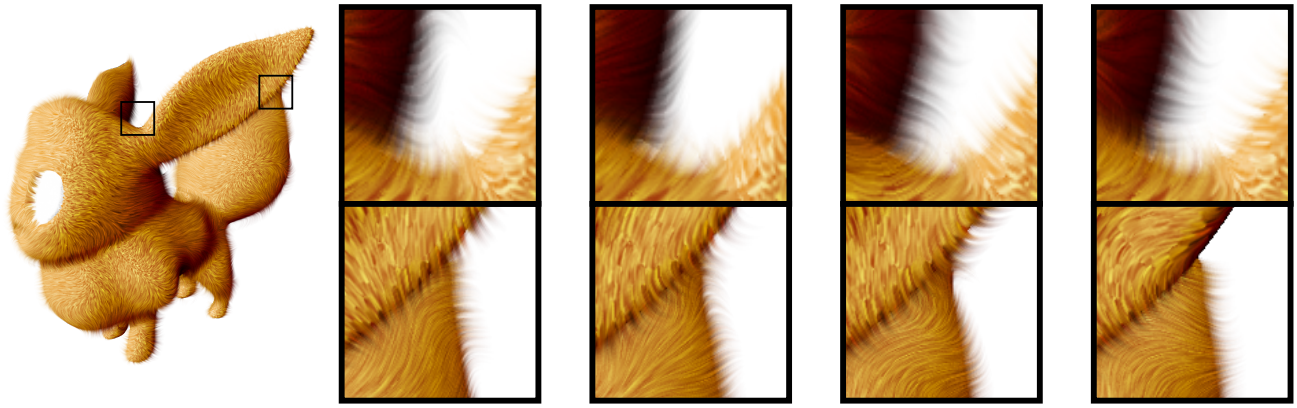

Figure 10: Random variations introduced by solid noise can be animated, such as in this example with a fur-like style. See the accompanying video for the full animation. Segmentation parameters: $\beta = 15, \sigma_z = 20, \sigma_n = 2$

- if $m_{\mu_1} \approx 1/2$ then m_{μ_1} is overlapping either m_{μ_0} or m_{μ_2}:
  - if $m_{\mu_0} < m_{\mu_2}$ then m_{μ_1} is likely overlapping m_{μ_0}, and m_{μ_2} is a distinct class:
  $$C_f = \frac{m_{\mu_0} C_0 + m_{\mu_1} C_1}{m_{\mu_0} + m_{\mu_1}} \textbf{ over } C_2 \textbf{ over } C_r$$
  - if $m_{\mu_2} > m_{\mu_0}$ then m_{μ_1} is likely overlapping m_{μ_2}, and m_{μ_0} is a distinct class:
  $$C_f = C_0 \textbf{ over } \frac{m_{\mu_1} C_1 + m_{\mu_2} C_2}{m_{\mu_1} + m_{\mu_2}} \textbf{ over } C_r$$
- finally, if $m_{\mu_1} \approx 1/3$ then all three classes are likely overlapping:
$$C_f = \frac{m_{\mu_0} C_0 + m_{\mu_1} C_1 + m_{\mu_2} C_2}{m_{\mu_0} + m_{\mu_1} + m_{\mu_2}} \textbf{ over } C_r$$

In practice, the class membership weight m_{μ_1} may take intermediate values between 1, 1/2 and 1/3. To avoid discontinuities, we calculate the blending result for all four cases and interpolate between those according to the value of m_{μ_1}. Note that the standard "**over**" compositing operator can be replaced by other blend modes for more flexibility in the resulting style.

6 RESULTS

Figures 1, 10 and 11 present several styles using our line integral convolution, showing that it can create discrete stylization marks that extend outside the original object. The accompanying video shows them in motion demonstrating the coherence of the motion of the marks. Unless specified otherwise, all results presented in this section have been made with $N = 8$ K-means iterations, and with $\sigma = 1$.

In Figure 11(a), a LIC is directed by a constant flow, whereas in Figure 1, solid gradient noise is used to locally perturb the orientation of the directing flow (surface tangents). This produces a wavier appearance. Auxiliary maps that drive the stylization filters can also vary over time, as shown in Figure 10 where a gradient noise is animated to evoke a fur moved by the wind.

Figures 11(b,c,d,f,g) illustrate various painterly appearances obtained with our modified LIC filter. In Figure 11(d), the filter is directed by the inflated surface tangents; whereas in Figures 11(c,f,g), the filter is directed by the inflated screen-space normals. These filters can be used in various ways to create novel styles: in Figure 11(b), we introduce perturbations in the directing flow, but also introduce color variations along the integration path, producing a fur-like effect.

More geometrical effects can also be emulated with image filters, such as the wobbled silhouette of Figure 11(e). Here, the integration profile f of the modified LIC filter is designed to simulate a displacement along the directing flow (normals). The length of the integration path, and thus the amount of displacement, is locally modulated by a low-frequency solid noise that has been inflated outside the original object footprint. In Figure 11(h) a low-frequency solid gradient noise is applied on the object, slightly inflated, then processed in the stylization step to extract isolines. The result is composited on top of the original object. The accompanying video also shows this style with an animated solid noise.

Finally, Figure 12 shows several examples obtained with the brush convolution filter. The ability to choose any arbitrary brush footprint provides great artistic freedom.

Performance: our pipeline was implemented as a set of (non-optimized) OpenGL fragment shaders in the Gratin software [Vergne and Barla 2015]. Its performance is mainly affected by the radius of the filter window as well as the target resolution. The stylization filter complexity also has an impact on performances. The following table provides the number of frames per second for varying radii, resolutions and style modes. These numbers were measured on the "bunny" object, on the styles in Figures 1 (LIC) and 12 (brush), using $K = 3$ and with a Nvidia GeForce GTX 1080 Ti graphics card.

Resolution	Filter	$r = 15$	$r = 20$	$r = 30$
512x512	LIC	7.3	3.8	2
	Brush	3.4	1.5	0.8
1024x1024	LIC	4.3	2.9	0.54
	Brush	0.8	0.4	0.2

Interactive framerates are usually obtained with simple styles and small screen resolutions. Optimizations necessary to improve performances and allow our pipeline to be included in real-time applications such as video-games are left for future works.

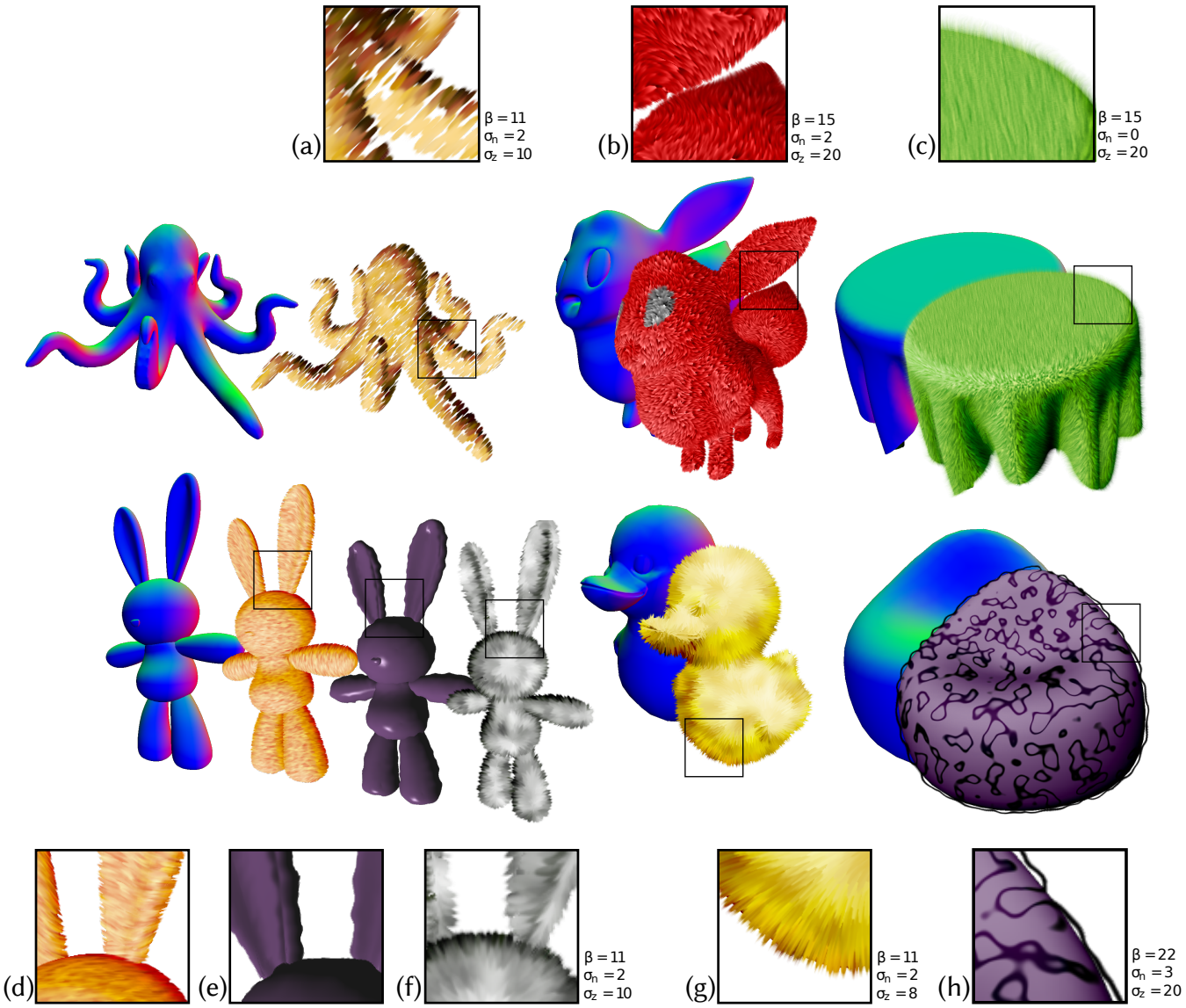

Figure 11: Various results obtained with LIC based filters. Model (b) by Sketchfab user Katerina Novakova (CC-BY-SA). Bunny model (d,e,f) by Sketchfab user Bernardo 3D (CC-BY). Octopus model by Lukáš Marek (CC-BY).

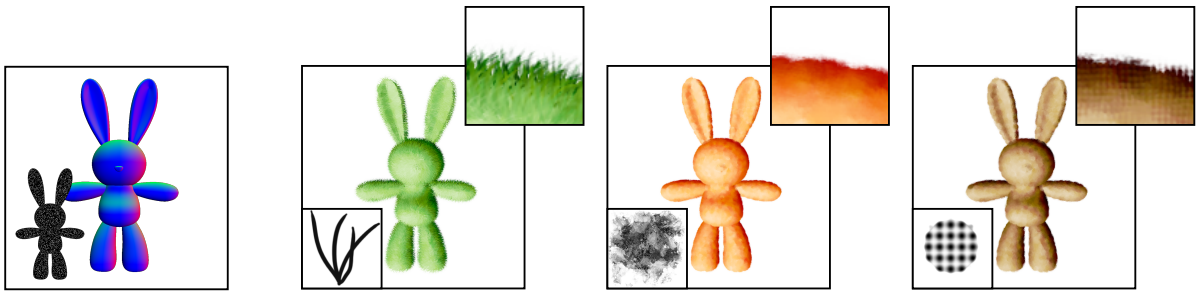

Figure 12: Styles obtained with the brush convolution filter over a thresholded cellular noise. By varying the brush pattern, a variety of different appearances can be reproduced. In the "grass" example, the brush patterns are oriented by the inflated screen-space normals. For these results $\beta = 11, \sigma_z = 10, \sigma_n = 2$.

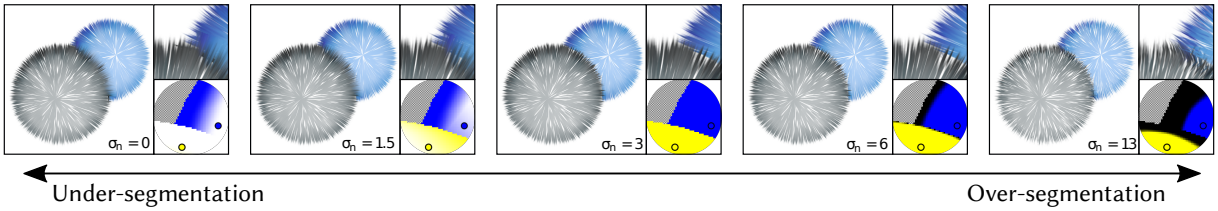

Figure 13: Influence of σ_n on the segmentation and filtered result. Pixels assigned to the residual class are shown in black in the classification result (lower-right windows). For these results $\beta = 11, \sigma_z = 10$.

7 DISCUSSION

Segmentation quality. Our local segmentation produces best results when the filter window contains enough data to cluster and with objects having rather smooth geometry. Local oversegmentation and per-pixel classification errors tend to occur in regions with large variations of depth or normal orientations, such as heavily slanted areas or regions with high curvature. In such cases, noticeable visual artifacts or holes may appear in the result. A good rule of thumb is to choose a filter window smaller than the smallest geometrical feature in the G-buffers but large enough to have enough data (*e.g.* $r > 10$). This, however, limits the size of the stylized features, and the amount of geometric detail in the original geometry. However, the impact of the latter is rather low, since the fine geometric features of highly detailed meshes would be lost anyway after applying the stylization filters, as shown in Figure 14.

Large filter windows are more likely to contain more overlapping inflated surface parts, and may require increasing K. As shown in Figure 15, if the effective number of distinct surfaces parts is higher than K, some of them will be grouped in the resulting classes, which may lead to undesirable artifacts in the inflated buffers. By following the rule of thumb above and limiting the filter window size, we found that smooth objects with low geometrical detail rarely need more than $K = 3$ classes for a correct detection of overlaps in inflation.

Finally, the segmentation is affected by the amplitude of depth discontinuities at silhouettes. It is sometimes necessary to increase the stiffness parameter β and/or adjust the per-pixel classification weights σ_z and σ_n to increase the sensitivity to depth differences, and, if necessary, increase the weight given to normal differences. Figure 13 shows visual artifacts that might appear in case of undersegmentation and oversegmentation, for varying values of σ_n. It shows a simple visual style consisting of marks directed by the surface normals. For high values of σ_n, the per-pixel segmentation may become too conservative and assign fringe pixels to the residual class. As the residual class spans unrelated surface parts, the corresponding inflated data blends two conflicting orientations, resulting in a distracting interaction at the boundary between the two spheres ($\sigma_n = 13$). Conversely, low values of σ_n or outright disabling the contribution of normals by setting $\sigma_n = 0$, can lead to undersegmentation: the surface parts are not properly segmented during filtering and the color of different surface parts may bleed onto each other ($\sigma_n = 0, \sigma_n = 1.5$).

Inflation of concave surface parts. The inflation result is not well defined in concave surface parts, as an infinite number of surface points may overlap when inflated. In those configurations, the weights $w_{\mathbf{p}}$ form elongated clusters with no local maximum, and are hence not easily detected as a single cluster by the local segmentation. The K-means algorithm will distribute class centroids along the cluster, splitting it into multiple classes in an unpredictable way. This is an inherently difficult case for stylization, as it is difficult to find a coherent anchor point on the surface. It is not handled in our pipeline, but we found that in practice those cases are rare and easily identifiable.

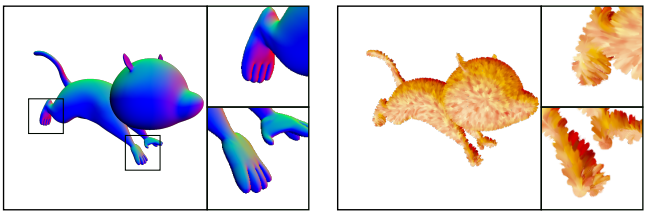

Figure 14: Geometry details finer than the filter size are lost during the stylization process, and can lead to visual artifacts as K-means does not handle the many small overlapping surface parts.

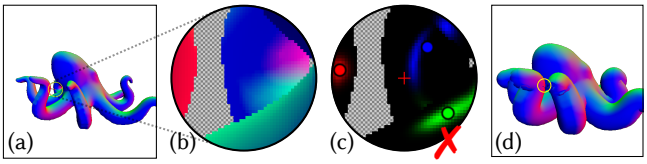

Figure 15: Segmentation failure for $K = 3$ classes. Under the filter window, 4 distinct surface parts overlap when inflated. The K-means algorithm merged two unrelated surface parts in the same class (blue class), impacting the inflated result.

Undetected surface parts and residual class. Some surface parts under the filter window are missed by the K-means segmentation pass when no pixel belonging to the surface part has an adequate screen-space normal that displaces it towards the window center. In most cases, this happens because the corresponding anchor point for inflation is occluded by another surface. Significant complexity is needed during per-pixel segmentation to account for missed surface parts. Our solution, which is to manually separate all pixels not belonging to a detected class into a residual class, cannot accurately preserve motion coherence, as the extended data for this class is not obtained by inflation. More generally, it can be seen as a limitation

of screen-space algorithms: information about occluded surfaces are lost, but is sometimes needed to get coherent data. However, this slight incoherence is usually not noticeable as it is often occluded by the stylization of other classes.

Disocclusions. A pervasive issue in stylized rendering techniques based on stroke primitives is how to handle disocclusions when a primitive becomes visible during camera or object motion. Our pipeline is not based on stroke primitives, but instead uses a combination of solid noise, 2D filtering, and standard procedural texturing techniques to achieve a similar appearance. Yet, this process is also subject to visual artifacts due to disocclusions. Depending on the stylization filter, noise features that become suddenly visible can introduce large changes in the results from one frame to another. This is particularly prevalent with the styles based on cellular noise and the modified LIC filter, as they tend to spatially "amplify" the point-like cells, producing noticeable *popping* artifacts with large filter sizes. This could be improved as a future work by prefiltering the noise at disocclusion boundaries (silhouettes).

Pipeline parameters. In addition to the parameters of the stylization filters, which are out of the scope of our contribution, the user may have to adjust several parameters of the segmentation (σ, β, σ_n, σ_z, filter window radius r) to obtain an accurate result for a given object. This can be a tedious process as some segmentation failure cases may only appear under certain viewpoints. Currently, a visualization of the local segmentation results under a filter window is provided to ease this process, but an improvement would be to infer some of those parameters, either from the geometry if it is available, or from a global analysis of the G-buffers contents.

Comparison with object-space inflation. Our approach does not require modification of the input geometry or the rendering pipeline. An alternative to our screen-space inflation algorithm could be done during the vertex processing stage, as in [Nienhaus and Döllner 2004b], and overlapping surface parts could be recovered using a *depth peeling* technique. Such an approach may be more efficient for high inflation radii at the cost of a more complex rendering pipeline. However our per-pixel classification stage would still be necessary in order to assign each pixel to a surface part within a filtering window.

8 CONCLUSION

We presented a post-processing stylization pipeline that allows stylization that extends outside object boundaries. Our main contribution in this paper is a post-processing technique to extend image filters that depend on G-buffer data outside the rasterized object boundaries in a motion coherent way, and with proper handling of overlaps at silhouettes. This pipeline only requires standard G-buffers as input (normal map, depth map), allowing it to be embedded in the compositing stage of most rendering software. This technique was then put to the test by integrating several well-known image filtering techniques. Instead of striving to reproduce a particular artistic style, our intent is to provide a generic way to make arbitrary image filters motion-coherent and silhouette-aware. We impose a very limited amount of restrictions on the image filters, thus providing a flexible stylization method capable of reproducing a wide variety of appearances. In the future, we would like to integrate more image filtering techniques into our pipeline, and further experiment to discover novel styles. In the process, we plan to take inspiration from styles found in digital paintings as we feel that such styles have not yet been widely explored in the context of the interactive rendering of 3D scenes.

REFERENCES

Christian Bauckhage. 2015. Lecture Notes on Data Science: Soft k-Means Clustering. (10 2015).

Pierre Bénard, Adrien Bousseau, and Joëlle Thollot. 2009. Dynamic Solid Textures for Real-Time Coherent Stylization. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*. ACM, ACM, Boston, MA, Etats-Unis, 121–127. http://maverick.inria.fr/Publications/2009/BBT09

Pierre Bénard, Adrien Bousseau, and Joëlle Thollot. 2011. State-of-the-Art Report on Temporal Coherence for Stylized Animations. *Computer Graphics Forum* 30, 8 (Dec. 2011), 2367–2386. DOI:https://doi.org/10.1111/j.1467-8659.2011.02075.x

Pierre Bénard, Ares Lagae, Peter Vangorp, Sylvain Lefebvre, George Drettakis, and Joëlle Thollot. 2010. A Dynamic Noise Primitive for Coherent Stylization. *Computer Graphics Forum (Proceedings of the Eurographics Symposium on Rendering 2010)* 29, 4 (june 2010), 1497–1506. http://maverick.inria.fr/Publications/2010/BLVLDT10

Isaac Botkin. 2009. Painting with polygons: non-photorealistic rendering using existing tools. In *SIGGRAPH 2009: Talks*. ACM, 22.

Adrien Bousseau, Matthew Kaplan, Joëlle Thollot, and François Sillion. 2006. Interactive watercolor rendering with temporal coherence and abstraction. In *International Symposium on Non-Photorealistic Animation and Rendering (NPAR)*. ACM. http://maverick.inria.fr/Publications/2006/BKTS06

Adrien Bousseau, Fabrice Neyret, Joëlle Thollot, and David Salesin. 2007. Video Watercolorization using Bidirectional Texture Advection. *ACM Transaction on Graphics (Proceedings of SIGGRAPH 2007)* 26, 3 (2007). http://maverick.inria.fr/Publications/2007/BNTS07

Brian Cabral and Leith Casey Leedom. 1993. Imaging Vector Fields Using Line Integral Convolution. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '93)*. ACM, New York, NY, USA, 263–270. DOI:https://doi.org/10.1145/166117.166151

Philippe Decaudin and Fabrice Neyret. 2009. Volumetric Billboards. *Computer Graphics Forum* (2009). http://www-evasion.imag.fr/Publications/2009/DN09

Nicolas Imhof, Antoine Milliez, Flurin Jenal, René Bauer, Markus Gross, and Robert W. Sumner. 2015. Fin Textures for Real-time Painterly Aesthetics. In *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games (MIG '15)*. ACM, New York, NY, USA, 227–235. DOI:https://doi.org/10.1145/2822013.2822021

J. T. Kajiya and T. L. Kay. 1989. Rendering Fur with Three Dimensional Textures. *SIGGRAPH Comput. Graph.* 23, 3 (July 1989), 271–280. DOI:https://doi.org/10.1145/74334.74361

Robert D. Kalnins, Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Joseph C. Lee, Philip L. Davidson, Matthew Webb, John F. Hughes, and Adam Finkelstein. 2002. WYSIWYG NPR: Drawing Strokes Directly on 3D Models. *ACM Transactions on Graphics (Proc. SIGGRAPH)* 21, 3 (July 2002), 755–762.

Charalampos Koniaris, Darren Cosker, Xiaosong Yang, and Kenny Mitchell. 2014. Survey of texture mapping techniques for representing and rendering volumetric mesostructure. *Journal of Computer Graphics Techniques* (February 2014). http://opus.bath.ac.uk/44666/

Michael A. Kowalski, Lee Markosian, J. D. Northrup, Lubomir Bourdev, Ronen Barzel, Loring S. Holden, and John F. Hughes. 1999. Art-Based Rendering of Fur, Grass, and Trees. In *Proceedings of SIGGRAPH 99 (Computer Graphics Proceedings, Annual Conference Series)*. 433–438.

Ares Lagae, Sylvain Lefebvre, Rob Cook, Tony Derose, George Drettakis, David S. Ebert, J.P. Lewis, Ken Perlin, and Matthias Zwicker. 2010. A Survey of Procedural Noise Functions. *Computer Graphics Forum* 29, 8 (2010), 2579–2600. DOI:https://doi.org/10.1111/j.1467-8659.2010.01827.x

Jerome E. Lengyel, Emil Praun, Adam Finkelstein, and Hugues Hoppe. 2001. Real-Time Fur over Arbitrary Surfaces. In *2001 ACM Symposium on Interactive 3D Graphics*. 227–232.

John-Peter Lewis. 1984. Texture synthesis for digital painting. In *ACM SIGGRAPH Computer Graphics*, Vol. 18. ACM, 245–252.

Jingwan Lu, Pedro V. Sander, and Adam Finkelstein. 2010. Interactive Painterly Stylization of Images, Videos and 3D Animations. In *Proceedings of I3D 2010*.

E. B. Lum and Kwan-Liu Ma. 2001. Non-photorealistic rendering using watercolor inspired textures and illumination. In *Proceedings Ninth Pacific Conference on Computer Graphics and Applications. Pacific Graphics 2001*. 322–330. DOI:https://doi.org/10.1109/PCCGA.2001.962888

Barbara J. Meier. 1996. Painterly Rendering for Animation. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96)*. ACM, New York, NY, USA, 477–484. DOI:https://doi.org/10.1145/237170.237288

Alexandre Meyer and Fabrice Neyret. 1998. *Interactive Volumetric Textures.* Springer Vienna, Vienna, 157–168. DOI: https://doi.org/10.1007/978-3-7091-6453-2_15

Marc Nienhaus and Jürgen Döllner. 2004a. Blueprints: illustrating architecture and technical parts using hardware-accelerated non-photorealistic rendering. In *Proceedings of Graphics Interface 2004 (GI 2004).* Canadian Human-Computer Communications Society, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 49–56. DOI: https://doi.org/10.20380/GI2004.07

Marc Nienhaus and Jürgen Döllner. 2004b. Sketchy drawings. In *Proceedings of the 3rd international conference on Computer graphics, virtual reality, visualisation and interaction in Africa.* ACM, 73–81.

G. Papari and N. Petkov. 2009. Continuous Glass Patterns for Painterly Rendering. *IEEE Transactions on Image Processing* 18, 3 (March 2009), 652–664. DOI: https://doi.org/10.1109/TIP.2008.2009800

Ken Perlin and Eric M Hoffert. 1989. Hypertexture. In *ACM SIGGRAPH Computer Graphics*, Vol. 23. ACM, 253–262.

Takafumi Saito and Tokiichiro Takahashi. 1990. Comprehensible Rendering of 3-D Shapes. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '90).* ACM, New York, NY, USA, 197–206. DOI: https://doi.org/10.1145/97879.97901

Johannes Schmid, Martin Sebastian Senn, Markus Gross, and Robert W. Sumner. 2011. OverCoat: an implicit canvas for 3D painting. *ACM Trans. Graph.* 30, Article 28 (August 2011), 10 pages. Issue 4. DOI: https://doi.org/10.1145/2010324.1964923

David Vanderhaeghe, Pascal Barla, Joelle Thollot, and Francois X. Sillion. 2007. Dynamic Point Distribution for Stroke-based Rendering. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques (EGSR'07).* Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 139–146. DOI: https://doi.org/10.2312/EGWR/EGSR07/139-146

Romain Vergne and Pascal Barla. 2015. Designing Gratin, A GPU-Tailored Node-Based System. *Journal of Computer Graphics Techniques* 4, 4 (2015), 17. https://hal.inria.fr/hal-01254546

Romain Vergne, David Vanderhaeghe, Jiazhou Chen, Pascal Barla, Xavier Granier, and Christophe Schlick. 2011. Implicit Brushes for Stylized Line-based Rendering. *Computer Graphics Forum* 30, 2 (April 2011), 513–522. DOI: https://doi.org/10.1111/j.1467-8659.2011.01892.x

Steven Worley. 1996. A Cellular Texture Basis Function. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '96).* ACM, New York, NY, USA, 291–294. DOI: https://doi.org/10.1145/237170.237267