DOI: 10.1111/cgf.142613
EUROGRAPHICS 2021 / N. Mitra and I. Viola
(Guest Editors)
Volume 40 (2021), Number 2

Coherent Mark-based Stylization of 3D Scenes at the Compositing Stage

M. Garcia[1], R. Vergne[1], M.-A. Farhat[1], P. Bénard[2], C. Noûs[3], J. Thollot[1]

[1]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LJK, France
[2]Univ. Bordeaux, Inria, Bordeaux INP, CNRS, LaBRI, UMR 5800, France [3]Laboratoire Cogitamus

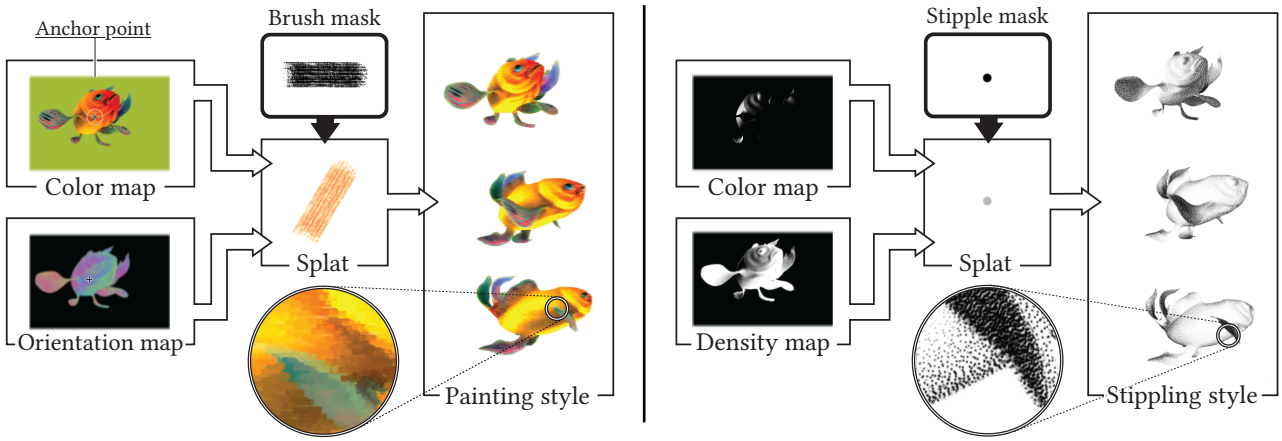

Figure 1: *Thanks to our real-time and temporally coherent anchor point generation process, we produce a wide range of styles using different control maps and splats.*

Abstract

We present a novel temporally coherent stylized rendering technique working entirely at the compositing stage. We first generate a distribution of 3D anchor points using an implicit grid based on the local object positions stored in a G-buffer, hence following object motion. We then draw splats in screen space anchored to these points so as to be motion coherent. To increase the perceived flatness of the style, we adjust the anchor points density using a fractalization mechanism. Sudden changes are prevented by controlling the anchor points opacity and introducing a new order-independent blending function. We demonstrate the versatility of our method by showing a large variety of styles thanks to the freedom offered by the splats content and their attributes that can be controlled by any G-buffer.

CCS Concepts

• ***Computing methodologies*** → ***Non-photorealistic rendering;***

1. Introduction

Stylized rendering is increasingly popular both in video games and animated films. Dedicated non-photorealistic rendering (NPR) frameworks such as BNPR, LOLLIPOPshaders or Artineering are more commonly used by the artistic community and in production pipelines, but mimicking complex artistic styles remains challenging, especially for animated scenes. As stated by Bénard et al. [BBT11], perfectly and simultaneously preserving the impression that a stylized image is drawn on a flat canvas (i.e., *flatness*) while ensuring a strong correlation between the apparent motion field of the 3D scene and the motion of its stylized depiction (i.e., *motion coherence*) and minimizing abrupt changes (i.e., *temporal continuity*) during an animation is impossible. Previous approaches, either *mark-based* or *texture-based*, compromise between these contradictory constraints, and thus manage to cover different ranges of styles from discrete to continuous. Furthermore, these NPR frameworks are usually built on top of existing 3D ren-

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd. Published by John Wiley & Sons Ltd.

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

derers — originally designed to generate photorealistic images — that cannot easily be modified to handle a large range of styles.

In this paper, we present a method that takes advantage of both mark and texture-based approaches to significantly widen this range. We propose to use marks that are coherently and dynamically positioned using a procedural, texture-based approach, which makes their density and style easily controllable. Moreover, our method can easily be integrated in any rendering pipeline as it fully operates in screen-space at the compositing stage using only 3D information stored in G-buffers (Figure 1).

To ensure motion coherence, we generate anchor points that follow the object motion field. Similarly to 3D Voronoi cellular noise, we use an implicit grid that associates a set of neighboring local object positions with a single coherent 3D position. The flatness of the style is enforced both by rendering splats in 2D and by using a fractalization mechanism to maintain a quasi-constant screen-space density of anchor points, especially when zooming in or out. Finally, to prevent popping artifacts and to ensure a stronger temporal continuity, we present a splat opacity scheme related to the anchor points distance to the object silhouettes, and we provide a new order-independent transparency blending function that avoids the blending artifacts observed in previous work.

Our full pipeline runs on the GPU, in parallel for all pixels, allowing for real-time to interactive performance. Since our method works as a post-process, it can also be used offline, e.g., to stylize high-quality animations rendered with global illumination. By varying the splats content, controlling their attributes with various G-buffers, and tuning the blending function, we show that we can reproduce most previous mark-based and some texture-based styles, as well as novel ones that depart from previous stylization techniques.

2. Related Work

The survey of Bénard et al. [BBT11] distinguishes between two main families of approaches when stylizing color regions of 3D animations: *texture-based* and *mark-based* methods. Our work belongs to the latter but borrows key ideas from the former.

Texture-based approaches. In all these methods, the marks are embedded in an image which is either mapped over the entire screen or onto the 3D scene. In either case, to preserve the flatness of the marks, their size needs to remain quasi-constant on screen. Fully working in screen-space, Cunzi et al. [CTP*03] introduce an infinite zoom mechanism, called *fractalization*, which consists in blending multiple versions of the original image with scaling factors and blending weights based on the distance to the camera. For 3D objects, dedicated mip-maps, called *art-maps* [PHWF01, SBB17] can be used to adapt the scale of the texture according to the distance to the camera, but the maximum depth range is restricted by the resolution of the largest texture. To handle arbitrary motion, Bénard et al. [BBT09] extend the fractalization algorithm in 3D, even though perspective distortion is only partially corrected. Between these full 2D or 3D techniques, patch-based approaches cover the projected 3D objects with a set of texture patches whose motion is restricted to rigid or affine transformations in screen-space [CDH06, BSM*07, KYYL08] or UV space [CDLK19] while approximating the 3D scene motion. The flatness of the marks is well preserved, but it may result in noticeable sliding artifacts when the patch size grows; yet for small patches, it becomes similar to mark-based techniques.

Mark-based approaches. As originally introduced by Meier [Mei96], these consists in attaching 2D marks (e.g., stipples, hatches, or generic texture sprites) to anchor points distributed onto, or in the vicinity, of a 3D scene. These marks can either be digitally painted by an artist [Dan99, SSGS11], or the anchor point distribution can be automatically computed to cover the 3D objects at a given camera distance [Mei96]. To allow interactive exploration, the density of points further needs to be adapted to the viewing conditions to remain uniform (e.g., Poisson disk), avoiding both holes and mark aggregates.

Such a distribution can be pre-computed in object-space for a fixed number of camera distances and stored in a hierarchical data-structure [KMN*99, CRL01, PFS03, CL06, USSK11]. At runtime, the appropriate levels are selected for the current viewpoint. By construction, the anchor points are coherent with the 3D scene, but their spacial distribution after projection may not be perfectly uniform, and the zoom range is limited by the depth of the hierarchy. To overcome these limitations, the points can be directly computed in screen-space [VBTS07, LSF10, DSZ17, USS*18] and advected along the motion field of the 3D scene. Yet, to maintain a constant density, individual points need to be added or removed, leading to temporal discontinuities.

When a large number of marks are blended together, these temporal events are less noticeable, and the quality of the point distribution is also less critical. Drawing inspiration from sparse convolution noises [LLC*10], a few many-marks methods have leveraged these properties. Kaplan and Cohen [KC05] and Bousseau et al. [BKTS06] use a pre-computed hierarchical 3D distribution of anchor points to produce a dynamic canvas and watercolor pigments respectively. Bénard et al. [BLV*10] generate more complex procedural patterns using Gabor kernels as marks, and introduce a level-of-detail mechanism which blends between two power-of-two sets of random points with a blending weight specifically tailored to Gabor noise. This further reduces temporal discontinuities and can be implemented on the GPU in a geometry shader, albeit with a limited number of points per triangle (unless dynamic tessellation is used). Alternatively, Bleron et al. [BVHT18] produce similar patterns by filtering a much simpler 3D Voronoi cellular noise [Wor96] with screen-space spatially-varying filters, based on geometric data stored in a G-buffer. However, since the noise is restricted to the object interior, a complex *inflation* step is required if the stylization needs to extend outside object boundaries.

Our method generalizes these previous approaches by dynamically generating a motion-coherent 3D anchor point distribution, which is based on implicit Voronoi noise generated at the compositing stage from a G-buffer. It can thus be applied as a post-process regardless of geometric representation or animation technique. The anchor point density can significantly vary — from very dense to very sparse — depending on the intended style, while remaining quasi-constant in screen-space thanks to an extended fractalization algorithm applied to the point distribution.

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

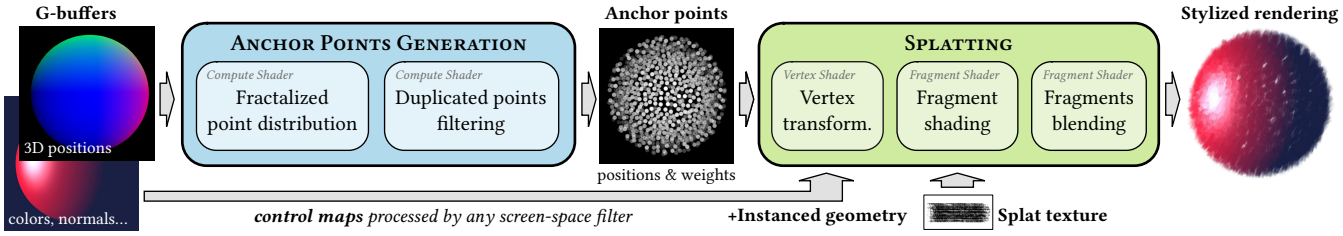

Figure 2: ***Our two-step pipeline.*** *First, a distribution of anchor points is generated from 3D positions stored in a G-buffer. Then a splat is instantiated for each anchor point, transformed, rasterized and shaded according to G-buffer data (normals, tangents, texture coordinates, etc.). Splat fragments are eventually composited together using a custom blended function.*

3. General Pipeline

Our stylization pipeline is divided into two main steps summarized in Figure 2. Starting from object-space positions stored in a G-buffer, we first generate a distribution of 3D anchors points with associated weights. For each anchor point, we instantiate a 2D splat and store the list of splat fragments covering each screen-space pixel. Eventually, we color each image pixel using an order-independent transparency algorithm and a controllable blending function.

Our post-processing pipeline makes use of input G-Buffers containing positions, normals, tangents, depths and texture coordinates. This data is used for computing splat anchors points, size, orientation and opacity. Additional maps might be used to give users more control over splat density, color, size and opacity.

4. Anchor Points Generation

Central to our approach is the generation of 3D points that are used to anchor 2D splats. Similarly to previous mark-based approaches, we make these points follow the motion field of the 3D object to ensure perfect motion coherence. Flatness perception is increased using a fractalization mechanism, and the temporal continuity of the anchor points is controlled by a weighting scheme that may act on the opacity or size of the corresponding splat. Our key contribution is to compute these points implicitly on the GPU at the compositing stage.

4.1. Points computation at a single scale

Our method takes as input a G-buffer in which each pixel $\mathbf{x}$ stores the 3D local coordinate position $\mathbf{p}(\mathbf{x})$ of an object surface. Inspired by 3D Voronoi cellular noise [Wor96], we generate one point per pixel by decomposing the space into an implicit grid, where each cell origins at $\mathbf{c}(\mathbf{x}) = floor(f \cdot \mathbf{p}(\mathbf{x}))$. The user-defined frequency parameter f modifies the size of the cells to control the number of generated points. We compute the pseudo-random points $\mathbf{p}_{ijk}(\mathbf{x})$ in the cell containing $\mathbf{p}$ as well as all its direct neighbors, and select the closest one to $\mathbf{p}$:

$$\mathbf{v}(\mathbf{x}) = \underset{\mathbf{p}_{ijk}(\mathbf{x})}{\arg\min} \left\| \mathbf{p}(\mathbf{x}) - \mathbf{p}_{ijk}(\mathbf{x}) \right\|,$$

$$\text{with:} \quad \mathbf{p}_{ijk}(\mathbf{x}) = (\mathbf{c}(\mathbf{x}) + (i,j,k)^\top + \text{rand}(\mathbf{c}(\mathbf{x}) + (i,j,k)^\top))/f,$$

where rand is a pseudo random function $\mathbb{R}^3 \to [0,1]^3$ and the vector $(i,j,k)^\top \in (\{-1;0;1\})^3$ iterates over neighborhood cells. Instead of displaying the shortest distances, as in classic Voronoi noise, we store the closest points $\mathbf{a}(\mathbf{x}) = \mathbf{M} \cdot \mathbf{v}(\mathbf{x})$ projected from local object coordinates to clip space coordinates using the Model-View-Projection matrix $\mathbf{M}$. Using the above formula, many pixels may potentially generate the same 3D position. We define $\mathcal{A}$ as the subset of anchor points positions with pairwise distinct elements:

$$\mathcal{A} = \{\mathbf{a}(\mathbf{x}) \mid \forall \mathbf{y} \neq \mathbf{x},\ \mathbf{a}(\mathbf{y}) \neq \mathbf{a}(\mathbf{x})\}.$$

Surface projection. The generated anchor points may lie outside the object boundaries, especially when the grid frequency is low. This makes the anchor point distribution hard to control, and it is preferable that the points closely stick to the object surface. We thus introduce a simple yet efficient method to project the anchor points onto the surface. For each point $\mathbf{a}_p \in \mathcal{A}$, we store the set of pixels from which it comes from:

$$\mathcal{P}_{\mathbf{a}_p} = \{\mathbf{x} \mid \mathbf{a}(\mathbf{x}) = \mathbf{a}_p\}.$$

The projected anchor point position $\mathbf{a}'_p$ is then computed as the weighted-average of the pixel positions $\mathbf{x} \in \mathcal{P}_{\mathbf{a}_p}$ and associated depths z:

$$\mathbf{a}'_p = \sum_{\mathbf{x} \in \mathcal{P}_{\mathbf{a}_p}} \|\mathbf{p}(\mathbf{x}) - \mathbf{v}_p\| \, (\mathbf{x}, z)^\top \Big/ \sum_{\mathbf{x} \in \mathcal{P}_{\mathbf{a}_p}} \|\mathbf{p}(\mathbf{x}) - \mathbf{v}_p\|,$$

with $\|\mathbf{p}(\mathbf{x}) - \mathbf{v}_p\|$ the distance between the object-space position stored in $\mathbf{x}$ and the 3D anchor point position $\mathbf{v}_p$.

Area weights. As G-buffers have finite resolution, special attention needs to be paid to avoid popping or flickering artifacts from frame to frame. The generated points may indeed appear or disappear, especially near silhouettes, due to unstable visibility information in the input buffers. To ensure a proper temporal continuity, we thus compute a weight for each anchor point, that will typically be used as an opacity factor to make stylized splats seamlessly appear and disappear during an animation.

Intuitively, regions containing strong depth discontinuities are more prone to popping, since a given anchor point will be generated by few pixels. As we initially generate one point per pixel, the number of pixels $\#\mathcal{P}_{\mathbf{a}_p}$ generating the same anchor point $\mathbf{a}_p \in \mathcal{A}$ effectively measures the surface area that contributed to the generation of this point. For each anchor point, we thus compute its

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

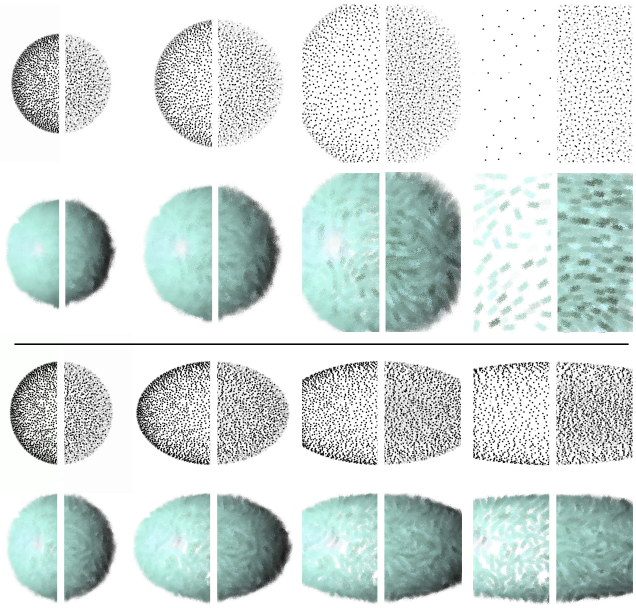

Figure 3: *Illustration of our fractalization mechanism when zooming in (top) or stretching a sphere (bottom). With fractalization (right side of each image), the density of the generated points remains constant in screen-space, preventing the creation of unwanted holes in the resulting stylization.*

weight w_{a_p} as the ratio of the area covered by these pixels and the maximum screen-space area that can potentially generates a point with a given frequency f:

$$w_{a_p} = \frac{\#\mathcal{P}_{\mathbf{a}_p}}{d_f^2 \pi}, \tag{1}$$

where d_f is the half diagonal length of a grid cell in screen-space. We finally provide users with further control over w_{a_p} using a smooth step function (Hermite interpolation) parametrized by the upper bound value $w_{\max} \in [0,1]$:

$$w'_{a_p} = 3\left(\frac{w_{a_p}}{w_{\max}}\right)^2 - 2\left(\frac{w_{a_p}}{w_{\max}}\right)^3.$$

This typically allows users to increase the global opacity of the splats, which is especially useful for styles where splats should not be blended together.

4.2. Point distribution fractalization

To increase the flatness of the final style, we need to keep the anchor point density as uniform as possible in screen-space. We compute such a point distribution implicitly using a fractalization mechanism similar in spirit to the algorithm proposed by Bénard et al. [BBT09] for solid textures. However, instead of blending several octaves of a given texture, we compute a user-controllable number of anchor point sets with associated weights for power-of-two frequencies. In addition, unlike previous approaches, our fractalization algorithm is driven by the gradient of the surface positions, which is more generic than the usual distance to the camera, since it captures both scaling effects due to camera motion in depth and distortions of deformable objects (Figure 3).

For a given position $\mathbf{p}$ at pixel $\mathbf{x}$, we compute its reference fractalization level as:

$$l_r(\mathbf{p}(\mathbf{x})) = \log_2(\|\nabla \mathbf{p}(\mathbf{x})\|).$$

The associated frequency $f_r = f/2^{l_r}$ thus increases when the gradient is low, which is the case on flat surfaces and when zooming in. According to the user-defined fractalization range n, we then create n fractalization levels on both sides of the reference level:

$$l_k(\mathbf{p}) = floor(l_r(\mathbf{p})) + k \;\text{ with }\; k \in [-n;n].$$

For each level l_k, we generate an anchor point $\mathbf{a}_p$ using the algorithm described in Section 4.1 with grid frequency $f_k = f/2^{l_k}$. To ensure smooth transitions from one level to the next, we modify the anchor point weighting scheme. We first compute the Gaussian-weighted distance between the current and reference level:

$$w_k(\mathbf{p}) = \exp\left(-\frac{\|l_k(\mathbf{p}) - l_r(\mathbf{p})\|^2}{2\sigma^2}\right),$$

with σ empirically set to $0.4n$ in all our experiments. We then modify Equation 1 in order to include this distance and modulate the weight of the anchor point $\mathbf{a}_p$:

$$w_{a_p} = \sum_{\mathbf{x}\in\mathcal{P}_{\mathbf{a}_p}} \frac{w_k(\mathbf{p}(\mathbf{x}))}{d_{f_k}^2 \pi \sum_{j=-n}^{n} w_j(\mathbf{p}(\mathbf{x}))}.$$

Intuitively, we normalize all $\{w_k(\mathbf{p})\}_k$ for a given $\mathbf{p}$ and divide each of them by their respective area coverage. Eventually, w_{a_p} is equal to the sum of normalized weights for each point $\mathbf{p}$ generating $\mathbf{a}_p$ at level k. Figure 4 illustrates the effect of this opacity scheme.

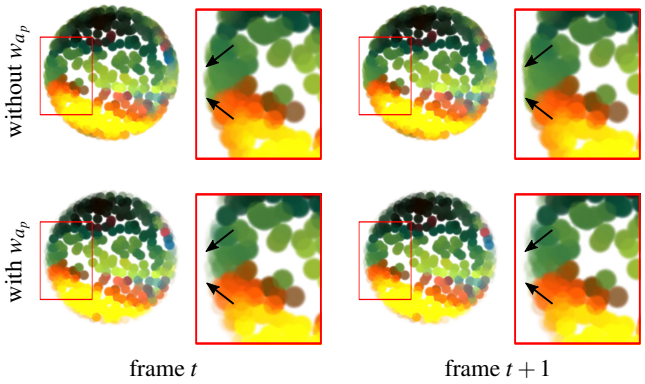

Figure 4: *Illustration of our anchor point opacity scheme. The left and right columns show two consecutive frames of a rotation around a sphere. When no opacity scheme is applied (first row), splats suddenly appear at the silhouette of the sphere, whereas our area weights smoothen the appearance of newly visible splats improving temporal continuity (second row).*

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

(a) *over* [PD84] (b) [MB13] : left $\gamma = 3 \times 10^{-3}$, right $\gamma = 6 \times 10^{-2}$ (c) [BRV*10] : left $\omega = 0.1$, right $\omega = 0.46$ (d) Our blending function: left $\omega = 0$, right $\omega = 1$

Figure 5: *Toy blending example consisting in three circular splats at a constant depth with a smooth radial transparency falloff along their edge and the following global parameters:* $[u_1 = (0,0,1), \alpha_1 = 0.8, d_1 = 0.5]$, $[u_2 = (1,0,0), \alpha_2 = 0.6, d_2 = 0]$, $[u_3 = (0,1,0), \alpha_3 = 0.4, d_3 = 1]$. *The accompanying video shows interactive variations of these parameters.*

5. Splat Blending

In this work, we assume that splats are, by default, billboard images that always face the camera. Without special treatment, when the input surface or the camera moves, certain splats may suddenly be drawn over others from one frame to the next causing popping artifacts. Since the *over* blending function [Wal81, PD84] is asymmetric, it does not solve this issue, even when the splats are not fully opaque. Instead, we need to take into account the depth of each splat to ensure temporally coherent compositing, in the same spirit as the blurred depth test of Luft et al. [LD06] or the hybrid visibility of Bruckner et al. [BRV*10]. We thus propose a new formula for blending splats together which accounts for their relative distance in depth.

Taking inspiration from previous work on commutative blending functions for order-independent transparency [MB13], our method aims at making parts of the splats that are near to each other depth-wise more transparent and blend them with respect to their distance to the "nearest" splat, defined below. Let us consider a per pixel list of N overlapping splat fragments [YHGT10], each fragment $i \in [1,N]$ is defined by a color u_i, a transparency value α_i and a normalized depth $d_i \in [0,1]$. Our key idea is to make fragments that are both close to the camera and opaque more visible. To this end, we first compute a reference depth d_{ref}, which corresponds to the smallest alpha-weighted depth of the overlapping splats, and a reference alpha value α_{ref} using the regular *over* function [PD84]:

$$\begin{cases} d_{ref} &= \min\limits_{i \in [1,N]} (\alpha_i (d_i - 1) + 1) \\ \alpha_{ref} &= 1 - \prod\limits_{i=1}^{N} (1 - \alpha_i). \end{cases}$$

For each splat fragment, we then compute a visibility factor v_i that depends on both the proximity to the reference depth d_{ref} and alpha value α_{ref}:

$$v_i = \frac{1}{N} \left(1 - \min(\max(d_i - d_{ref}, 0)\,(1 + \omega\,\alpha_{ref}), 1)\right) \alpha_i,$$

where $\omega \in [0,1]$ is a global blending strength factor controlled by the user. This formula makes splat visibility decrease faster when ω increases as illustrated in Figure 6. For instance, when $d_{ref} = 0$ and $\omega = 1$, all splats with $d_i \geq 0.5$ will be invisible and will not contribute to the final blending. Notice that ω is weighted by α_{ref} to avoid fully masking the nearest splat when it is semi-transparent.

From this visibility factor we compute new alpha values ensuring that the final transparency matches the reference α_{ref}:

$$\alpha'_i = \frac{v_i}{\sum_{j=1}^{N} v_j} \alpha_{ref}.$$

Intuitively, this formula normalizes the visibility factors and assigns to each fragment the corresponding fraction of the reference transparency. The final color and opacity of a given pixel is then obtained with a weighted sum of the new transparency values:

$$\begin{cases} u_{blend} &= \sum_{i=1}^{n} \alpha'_i u_i / \sum_{i=1}^{n} \alpha'_i \\ \alpha_{blend} &= \sum_{i=1}^{n} \alpha'_i \end{cases}$$

As our blending function does not rely on depth ordering, it allows splats to seamlessly move in front of each other over time. The final pixel color can also easily be blended with a background image using the *over* operator.

We compare in Figure 5, on a toy example composed of three circular splats, the results obtained from three different approaches and our method. First, we show the asymmetric *over* operator. Second, we show the commutative blending function of McGuire and Bavoil [MB13] (Equ. 6 & 10 in the paper, where the depth is further adjusted by a factor γ). Third, we show the distance-based blending operator of Bruckner et al. [BRV*10] (Equ. 1 & 2 in the paper) assuming that all splats belong to a single layer (as defined in the paper) and that the reference depth is the average distance of the splats to the camera. Similarly to McGuire and Bavoil, we cannot fully reproduce the behavior of the *over* operator, but our results are reasonably similar and we avoid the rigging artifacts noticeable at the boundary of the overlapping splats. The operator of Bruckner et al. does not seem well-suited for splats blending since it produces results that are quite different from the *over* operator. For low blending factors $\omega \in [0:1]$, the result is similar to an average, whereas for high ω value, splats that are far away from the reference depth tend to be masked, which may even give the impression that splats are swapped as shown in Figure 5 (the blue splat seems closer to the camera than the red splat). Figure 6 further compares our approach with the *over* operator on a rotating sphere.

6. Style Examples

We can produce a wide variety of styles depending on the chosen texture, size, orientation and opacity of the splats, and the density of anchor points. Brush splats can produce painterly renderings ranging from pointillism to impressionism, or more blended painting

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

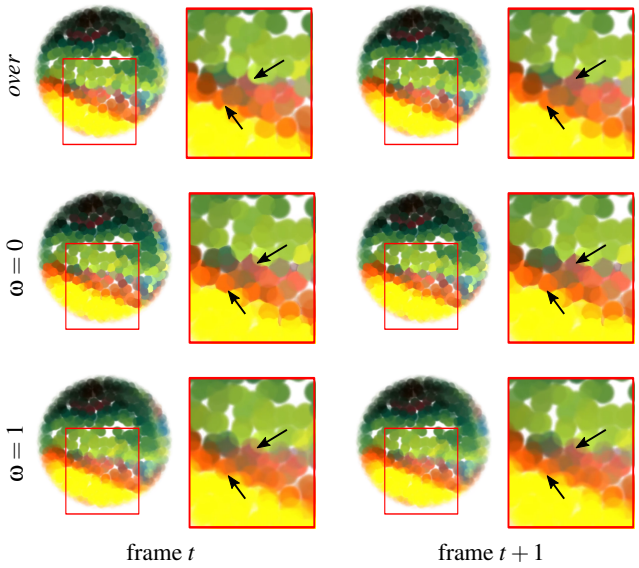

Figure 6: *Comparison between our blending scheme and the over operator [PD84]. The left and right columns show two consecutive frames of a rotation around a sphere. Notice the popping artifacts occurring when one splat come across another using the over operator (first row); our depth-aware order-independent scheme prevents such temporal instabilities (second and third rows). In addition the blending strength parameter ω allows the user to control the perception of individual marks.*

styles according to user-defined parameters. A very high number of small splats allows us to generate stippling, while several big splats may be combined to produce almost continuous textures. The user can control the rendering via several global parameters (summarized in Table 1), which can be further refined with screen-space control maps stored in G-buffers, as shown in Figure 1. As long as temporally coherent filters are used to generate such auxiliary maps, the anchor points generated by our method ensure temporally coherent results. Nevertheless, special care must be taken to avoid producing aliasing at each step of the pipeline. We usually apply some amount of smoothing (Gaussian blur) to each map to limit this problem.

Our method also applies to animated objects like articulated characters and deformable environments, as long as temporally coherent input positions are provided. In practice, we baked their rest pose positions into a texture and mapped it onto the object surface so that they get deformed during the animation. We include several animated objects in our results: a deforming blob and twisting torus, an articulated dancer and a procedurally generated ocean.

Table 1: *Global parameters of our pipeline. Users can tune five parameters allowing them to create various styles as well as variations of the same style.*

Notation	Range	Description
f	$[0:+\infty]$	Sampling frequency
s_d	$[0:1]$	Splat size (scaling factor)
$w_{\max}$	$[0:1]$	Anchor point maximum opacity
ω	$[0:1]$	Blending strength
n	$[[0;+\infty]]$	Number of fractalization levels

In the following, we present a set of results trying to reproduce standard styles found in the NPR literature, but also more experimental styles leveraging the variety of splats that we can use. Indeed, the splat itself may consist in a simple alpha mask but can also contain color and normal information or even a procedurally generated and animated content. The corresponding videos and additional examples are provided in the supplemental materials.

Painterly. Using a color buffer to define the color of the splats, we can produce painterly results similar to previous stroke-based rendering methods (Figure 7). The brush stroke used as the splat texture can be enhanced with a normal map, allowing for shading the strokes themselves. An orientation map can be used to align the brush strokes along, for instance, the surface normals, tangents or principal directions of curvature.

We eventually show a result combining two painterly styles. We used Blender to render G-buffers with high-quality lighting effects and combined a style with randomly oriented brushes for the cups and a pointillism style for the teapot and floor. All stylization attributes such as density, size and orientation are generated on the object surface as textures to ensure motion coherence.

Pen-and-ink. Figure 8 shows pen-and-ink examples. For the stippling style, we use a simple dot colored using a gray-scale Phong shading of the model as input color map. The splat density is inversely proportional to the intensity of the shading, and further refined using an input diffuse texture whose gradient depicts areas of high details on the model and in which the density is increased. Such a style raises two challenges: (1) it requires a very high number of splats (we used $f = 85$ with $n = 3$ fractalization levels, e.g., generating 131 101 points at the first frame of the stippled sphere animation); (2) it is prone to aliasing due to the small size of the dots. We therefore rendered the models at high resolution (1600×1600 pixels) to produce our final results.

Using the same input maps but taking advantage of an orientation map, we created a hatching style using a simple black segment as the splat. Cross-hatching is easily obtained by compositing two layers of hatches oriented orthogonally. Here again, a high number of splats is needed to ensure a detailed depiction of the model.

Textures. Using large splats containing a textured patch, we can produce continuous texture effects similar to spot noise [vW91]. We show in Figure 9 the use of a splat containing a triangle pattern. Depending on the size of the splat, we control the size and precision of the resulting texture. To better depict visual details of the underlying 3D model, we use the model color for each fragment of the splat instead of a single color per splat, i.e., the color at the anchor point position. We also show a more abstract style obtained using a splat containing bubbles, with a very low frequency of anchor points, a simple blue shading and a constant orientation, which increases the abstraction effect.

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

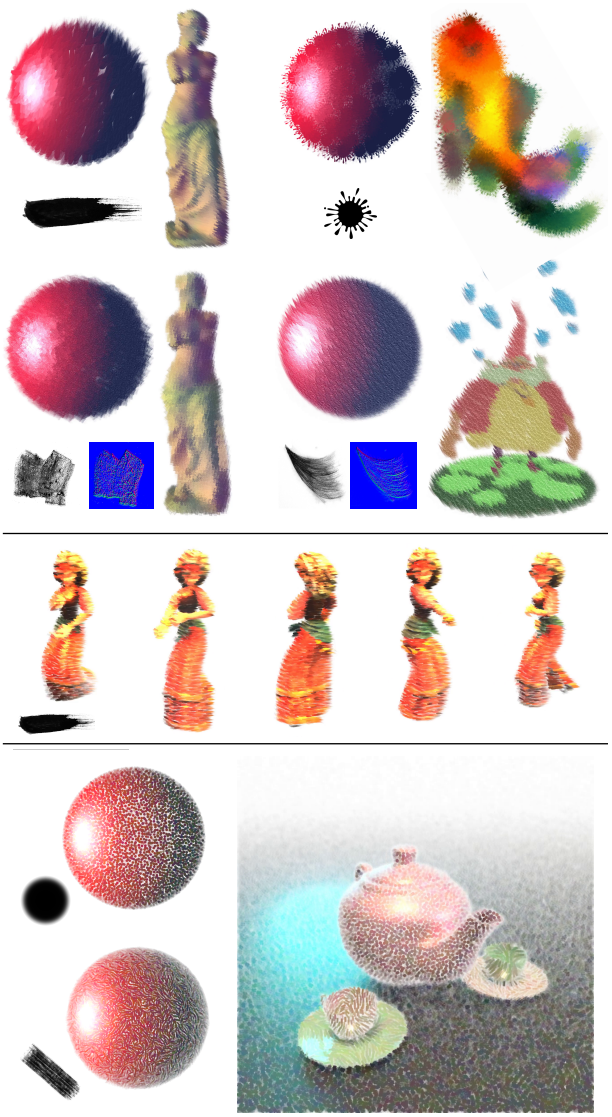

Figure 7: *Painterly style: (top) a large variety of styles is achieved by changing the splat texture and using various orientation maps; (middle) an articulated dancer is stylized coherently by backing its rest pose positions into a texture and mapping it back on its surface; (bottom) the G-buffers can be generated offline to obtain complex lighting effects in the color map.*

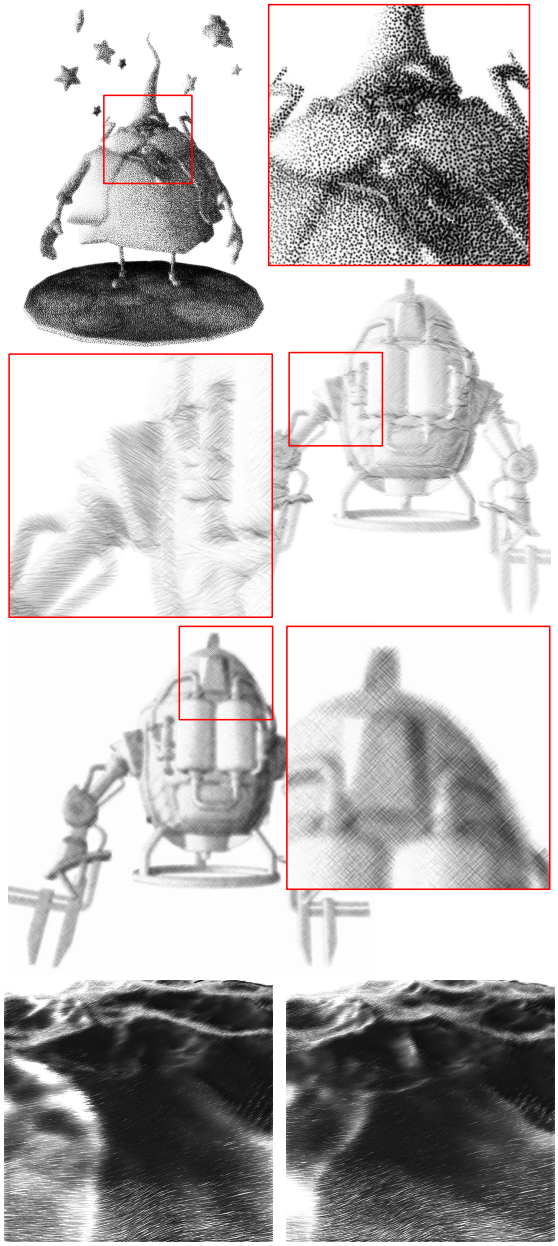

Figure 8: *Pen-and-ink: (grom top to bottom) stippling, hatching following the principal direction of curvature, cross hatching, and hatching following the bi-tangent of an animated ocean.*

Procedural splats. Since our pipeline works as a post-process on the GPU, we can directly compute the splat content online using any procedural generation approach. We show in Figure 10 (top) a Gabor noise example producing results inspired by those of Bénard et al. [BLV*10]. Each splat is a simple Gabor kernel randomly oriented using a 3D gradient noise but still drawn in 2D. For this particular example, we changed the blending function to the average sum of the splats to respect the standard Gabor noise formulation. We also produced a watercolor example following Bénard et al. [BLV*10] recipe that shows a comparable behavior in terms of temporal coherence.

Finally we show an animated fiber style where each splat is a simple implicit function based on the distance to a segment. Each segment is deformed by a sine wave whose frequency and phase are randomly selected using 3D gradient noises as input. Animating these parameters over time leads to coherent fiber motion as shown in Figure 10 (bottom) and in the supplemental video.

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

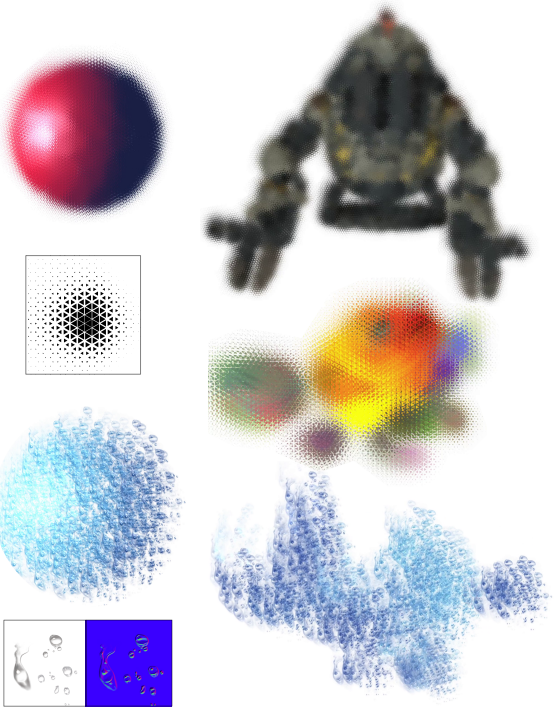

Figure 9: *Large and content-rich splats allow to create complex textures.*

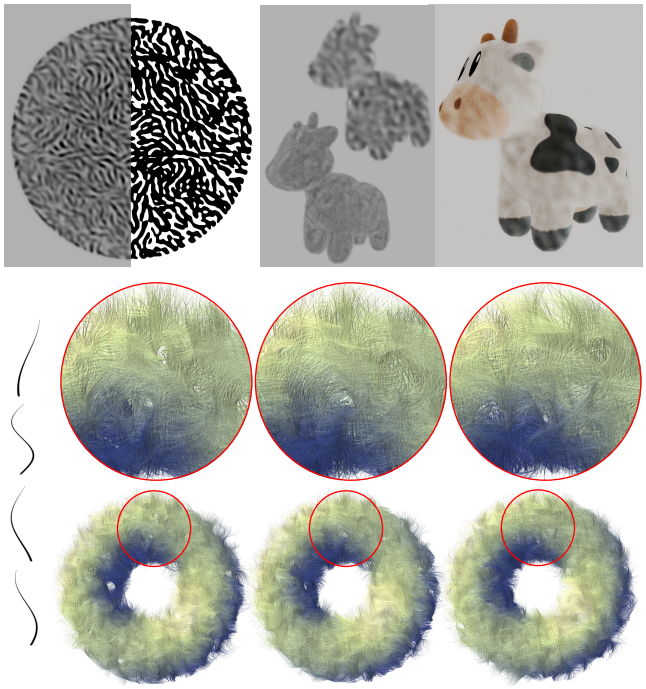

Figure 10: *Procedural splats: (top) dynamic Gabor noise used to generate a temporally coherent watercolor style, (bottom) implicit animated fiber splat.*

7. Implementation

Our method fully runs on the GPU using OpenGL 4.3 inside the node-based compositing system "Gratin" [VB15]. The anchor point generation algorithm described in Section 4 is implemented with two *Compute* shaders. For every pixel of the input G-buffer, the first shader generates $2n+1$ anchor points, with n the number of fractalization levels. Each of these points is stored at its corresponding screen-space position inside a per-pixel linked list, using atomic operations on integer counters. Ideally, we would have liked to avoid storing duplicated points by filtering them by their depth, but current GPU architectures do not provide atomic operations on 32 bits floating point buffers. Alternatively, we use a second shader that processes each linked list in parallel, computing the averaged projected position of the duplicated anchor points and their associated area weights.

The generated points can then be processed with a regular OpenGL pipeline using *Vertex* and *Fragment* shaders. The simplest approach consists in emitting as many `GL_POINTS` as anchor points and drawing each of them as a 2D sprite centered at their projected position. However, it offers restricted stylization options: it only allows global control over the point size and texture. A more generic approach leverages OpenGL instancing capability to draw any 2D geometry per anchor point. Every vertex of each instance can then be freely transformed in the *Vertex* shader according to its corresponding anchor point (identified by `gl_InstanceID`), but also any additional information read from the input G-buffer, such as the underlying surface normal.

The splat blending method detailed in Section 5 is implemented as a post-processing step using a standard order-independent transparency technique [YHGT10]. Instead of simply compositing the output fragments using OpenGL blending modes, we store their color, alpha and depth in a per-pixel linked list inside an offscreen buffer. During a final screen-space pass, for each pixel in parallel, the reference depth is first determined. The visibility of all fragments for a given pixel is then computed relatively to this reference and their own opacity; all fragments are eventually blended together using this quantity.

As demonstrated in Section 6, our method can be applied to 3D scenes composed of several objects as well as skinned characters (teapot and dancer examples in Figure 7). To handle multiple objects, we segment the scene with an ID Buffer and use one Model-View-Projection matrix per object to generate the anchor points. For skinned characters, we bake their rest pose positions into textures and wrap them around the objects. The anchor points are then generated by sampling those textures (a single matrix is thus needed); their position is computed as the average of the screen-space positions of the pixels generating the same anchor points as described in Section 4.1.

In Figure 11 (top) we show the performance achieved by this implementation on a Nvidia Geforce RTX 2070 graphic card, for three constant point densities and two output image resolutions (1K and 4K), using 2D splats whose size ranges from 10×10 to 400×400 pixels. In our experimentation, the three point densities were used for different artistic styles: High point densities (yellow curves)

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

for stippling and hatching styles (see Figure 8), moderate densities (red curves) for painterly styles (see Figure 7) and low point densities (blue curves) for more abstract styles (see Figure 9). The same overall trend is observed for both output image resolutions with an expected $\times 4$ difference factor for low point densities (blue curves) or small splat sizes. When the splat size or density significantly increases, this difference is progressively reduced to below a $\times 2$ factor since many splats overlap in screen-space. Real-time to interactive performances are maintained for 1K outputs as long as reasonable splat sizes and densities are used, which is sufficient for style design and animation preview. Final high-quality rendering at a 2K resolution can take up to a few seconds.

In Figure 11 (bottom), we show the GPU memory footprint of this implementation for a 1K output image resolution. As expected, the memory usage evolves in a quadratic fashion with respect to splat size with a difference factor rising from $\times 18$ to $\times 200$ for high point densities. This trend is due to the linked-lists of splat fragments being stored at each pixel of the image as required by our order-independent transparency blending algorithm (the memory footprint of the anchor points list is constant for a given splat size). For high point densities and large splats (above 100x100 px), the memory footprint exceeds 12 GB, which is above the storage capacity of most modern GPUs. However, in practice, we did not encounter any style that requires such an extreme configuration. Note that the GPU memory of the linked-lists must be pre-allocated anyway; our implementation checks that the available memory is not exceeded. Beyond this limit, points and fragments are omitted and our algorithm cannot ensure that popping artifacts will not occur.

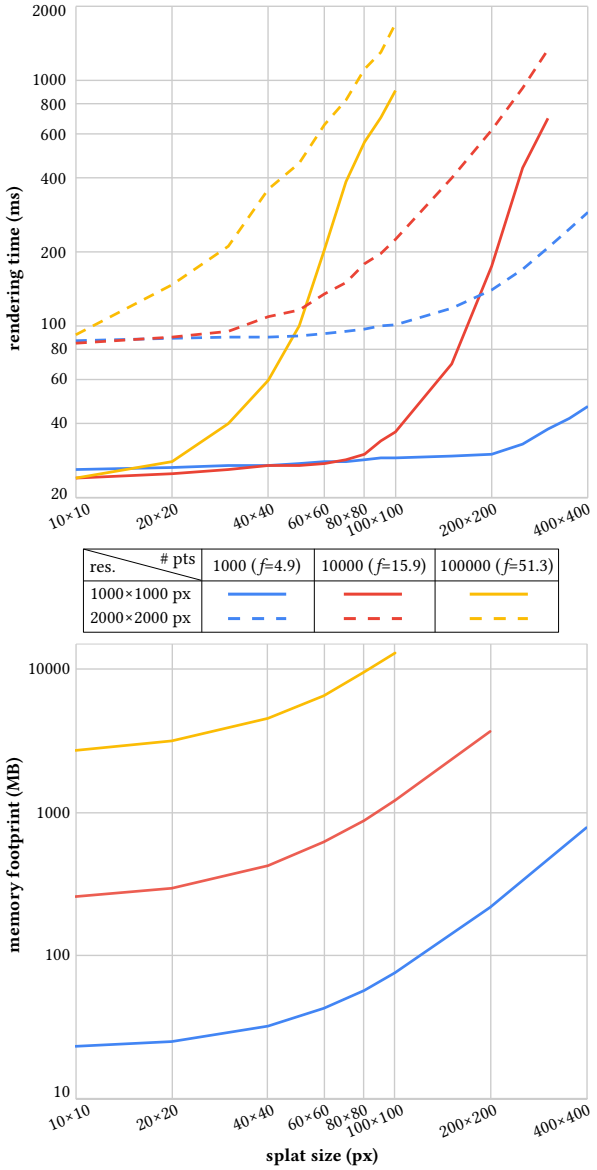

Figure 11: *Rendering time (top) and GPU memory footprint (bottom) with respect to the number of anchor points, 2D splat size and output image resolution (logarithmic scale on both axes).*

8. Evaluation

The evaluation of Non-Photorealistic Rendering techniques is a challenge in itself, especially when dealing with animations since no ground truth or perfect solution exists. Please note that we do not aim to evaluate the final "look" of the stylized renders, as this is an overly subjective assessment and largely depends on the artistic expertise of the user who produced those results. Instead, we propose to discuss and clarify the positioning of our method with respect to previous mark-based approaches. To conduct this comparison as fairly as possible, we first present a set of criteria that, in our opinion, every mark-based method should ensure. We then detail how each class of mark-based methods behaves with respect to these criteria and compare them to our approach, as summarized in Table 2. We use the following criteria for this comparison:

1. **Flatness**: give the impression that the image is drawn on a flat canvas;
2. **Motion coherence**: ensure a strong correlation between the motion of the marks and the apparent motion of the 3D object;
3. **Temporal continuity**: minimize abrupt changes of the marks (e.g., popping and flickering);
4. **Ease of integration**: ensure that the method easily fits into existing rendering pipelines;
5. **Range of style**: cover a wide range of styles;
6. **Dynamic behavior**: easily adapt to animated and deformable objects.

The first three criteria, proposed by Bénard et al. [BBT11], apply to any stylization method that addresses the temporal coherence problem. The other three measure the versatility of such a method. As introduced in Section 2, we divide mark-based stylization methods into four categories according to their sampling approach:

- **Static object-space**: pre-computed 3D distribution,
- **Dynamic object-space**: dynamic distribution computed over the 3D geometry,
- **Dynamic screen-space**: dynamic distribution computed from what is visible on screen,
- **Many-marks**: dense dynamic 3D distribution.

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

Table 2: *Mark-Based stylization approaches comparison.*

		Flatness	Motion coherence	Temporal continuity	Ease of integration	Range of style	Dynamic behavior
Static object-space	e.g., [Mei96]	−−	++	+−	−	+−	−−
Dynamic object-space	e.g., [CRL01]	+	++	+	−	+−	−
Dynamic image-space	e.g., [LSF10]	++	+−	−	++	−	+
Many-marks	e.g., [KC05]	+	++	+−	−	+−	−
Ours		+	++	+−	+	+	++

Static object-space approaches mostly suffer from flatness issues as marks cannot be added or removed when the camera viewpoint changes, producing holes or aggregates. Dynamic object-space methods better preserve flatness through complex level of detail (LOD) mechanisms, but those require heavy pre-computations. Furthermore, these mechanisms are only computed in a discrete fashion, which leads to temporal instabilities during transitions, and, as these methods rely directly on the geometry to sample anchor points, their integration into existing pipelines is difficult. Dynamic image-space methods can perfectly satisfy the flatness criteria, but they require an accurate motion field of the scene to preserve motion coherence, and suffer from popping artifacts due to the sudden appearance/disappearance of anchor points. Moreover, most of those approaches have difficulty in dealing with a very high number of points, hence limiting the range of style that they can handle. Finally, many-mark methods use continuous LOD mechanisms that require many anchor points to ensure temporal continuity and thus restrict them to specific styles. In addition, they also rely on the 3D geometry for point sampling, which makes them hard to integrate in existing pipelines.

Similarly to all dynamic image-space methods, our approach provides both flatness and motion coherence by construction, and can only easily process the visible geometry, which prevents, for instance, strokes appearing from hidden parts of the 3D scene. However, unlike previous work, our adaptive opacity factor combined with our continuous LOD mechanism smooth the appearance (resp. disappearance) of anchor points at disocclusions (resp. occlusions), and our blending operator further enforces temporal continuity. Please refer to the accompanying video for a comparison with the method of Lu et al. [LSF10] that clearly illustrates the benefits of these new components. Also note that this blending scheme could be used in previous methods to limit the popping artifacts due to the mark drawing order.

Since our method entirely works at the post-processing stage, it can be directly integrated into any existing rendering pipelines. As it is based on G-Buffers and 2D textures, it is sensitive to noise and aliasing present in these inputs, although this can be largely mitigated by pre-filtering. Our anchor point generation technique itself can produce aliasing artifacts at high frequencies, especially when the projected cell area of the virtual grid is lower than the pixel size. This limitation only occurs for very high frequency styles like stippling or hatching. From our tests, we ascertain that these artifacts are resolved by super-sampling, but this solution is hardly practical as it requires to generate $(s+2)^2$ anchor points per pixel (with s the super-sampling factor), which quickly overloads the GPU capacity.

Finally our method covers the range of styles of most previous mark-based methods, from sparse stroke-based styles to many-marks continuous styles. All the computation is done at runtime in a procedural fashion allowing for a fully dynamic approach.

9. Conclusions and Perspectives

We introduced a new stylization technique based on an implicit and temporally coherent generation of 2D marks. Our approach fully operates at the compositing stage, making it suitable for both real-time and offline pipelines, as demonstrated with the teapot global illumination scene. We have presented stylization results where 3D models are rotating or deforming while preserving motion coherence and temporal continuity, thanks to a new order-independent blending function, a default opacity policy based on a novel coverage criteria, and an extended fractalization mechanism.

Thanks to the freedom offered by the choice of the splat content and the control of its attributes via G-buffers, our method can already produce an unprecedented range of different styles. Yet we would like to investigate further the use of procedurally generated splats and their animation. For example, a *Geometry* shader could be used to generate more complex splat geometry for each anchor point, e.g., to create curved strokes following the direction of maximum curvature of the underlying 3D object. Moreover implicit splats could even make use of 3D content to obtain volumetric effects, if an adapted blending operation is defined. Besides, we would also like to investigate alternative ways to maintain temporal continuity when splats appear and disappear. Instead of changing their opacity, we could for instance modify their size, or explore different blending schemes in other color spaces.

Acknowledgements

We would like to thank Maxime Isnel, Laurence Boissieux and Sunrise Wang for their valuable discussions, suggestions, and assistance with this work, and the reviewers for their constructive comments. The following models come from Sketchfab under Common Creative Attribution license: Wiz Biz by tommonster, Shiny Fish by GenEugene, Mech Drone by Willy Decarpentrie.

References

[BBT09] BÉNARD P., BOUSSEAU A., THOLLOT J.: Dynamic solid textures for real-time coherent stylization. In *I3D 2009 - Symposium on Interactive 3D graphics and games* (2009), pp. 121–127. doi:10.1145/1507149.1507169. 2, 4

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.

[BBT11] BÉNARD P., BOUSSEAU A., THOLLOT J.: State-of-the-art report on temporal coherence for stylized animations. *Computer Graphics Forum 30*, 8 (2011), 2367–2386. doi:10.1111/j.1467-8659.2011.02075.x. 1, 2, 9

[BKTS06] BOUSSEAU A., KAPLAN M., THOLLOT J., SILLION F. X.: Interactive watercolor rendering with temporal coherence and abstraction. In *International Symposium on Non-Photorealistic Animation and Rendering (NPAR)* (2006), ACM. 2

[BLV*10] BÉNARD P., LAGAE A., VANGORP P., LEFEBVRE S., DRETTAKIS G., THOLLOT J.: A dynamic noise primitive for coherent stylization. *Computer Graphics Forum 29*, 4 (2010), 1497–1506. doi:10.1111/j.1467-8659.2010.01747.x. 2, 7

[BRV*10] BRUCKNER S., RAUTEK P., VIOLA I., ROBERTS M., SOUSA M. C., GRÃŰLLER M. E.: Hybrid visibility compositing and masking for illustrative rendering. *Computers & Graphics 34*, 4 (2010), 361 – 369. doi:10.1016/j.cag.2010.04.003. 5

[BSM*07] BRESLAV S., SZERSZEN K., MARKOSIAN L., BARLA P., THOLLOT J.: Dynamic 2d patterns for shading 3d scenes. *ACM Trans. Graph. 26*, 3 (2007), 20–es. doi:10.1145/1276377.1276402. 2

[BVHT18] BLÉRON A., VERGNE R., HURTUT T., THOLLOT J.: Motion-coherent stylization with screen-space image filters. In *Expressive '18 - The Joint Symposium on Computational Aesthetics and Sketch Based Interfaces and Modeling and Non-Photorealistic Animation and Rendering* (2018), ACM, pp. 1–13. doi:10.1145/3229147.3229163. 2

[CDH06] COCONU L., DEUSSEN O., HEGE H.-C.: Real-time pen-and-ink illustration of landscapes. In *Proceedings of the 4th International Symposium on Non-Photorealistic Animation and Rendering* (2006), pp. 27–35. doi:10.1145/1124728.1124734. 2

[CDLK19] CURTIS C., DART K., LATZKO T., KAHRS J.: Real-time non-photorealistic animation for immersive storytelling in "age of sail". *Computers & Graphics* (2019). doi:10.1016/j.cagx.2019.100012. 2

[CL06] CHI M.-T., LEE T.-Y.: Stylized and abstract painterly rendering system using a multiscale segmented sphere hierarchy. *IEEE Transactions on Visualization and Computer Graphics 12*, 1 (2006), 61–72. 2

[CRL01] CORNISH D., ROWAN A., LUEBKE D.: View-dependent particles for interactive non-photorealistic rendering. In *Proceedings of the Graphics Interface 2001 Conference* (2001), pp. 151–158. 2, 10

[CTP*03] CUNZI M., THOLLOT J., PARIS S., DEBUNNE G., GASCUEL J.-D., DURAND F.: Dynamic canvas for immersive non-photorealistic walkthroughs. In *Proc. Graphics Interface* (2003). 2

[Dan99] DANIELS E.: Deep canvas in Disney's Tarzan. In *SIGGRAPH 99 Conference Abstracts and Applications* (1999), ACM, p. 200. doi:10.1145/311625.312010. 2

[DSZ17] DEUSSEN O., SPICKER M., ZHENG Q.: Weighted linde-buzo-gray stippling. *ACM Trans. Graph. 36*, 6 (2017). doi:10.1145/3130800.3130819. 2

[KC05] KAPLAN M., COHEN E.: A generative model for dynamic canvas motion. In *Proceedings of the First Eurographics Conference on Computational Aesthetics in Graphics, Visualization and Imaging* (2005), pp. 49–56. doi:10.2312/COMPAESTH/COMPAESTH05/049-056. 2, 10

[KMN*99] KOWALSKI M. A., MARKOSIAN L., NORTHRUP J. D., BOURDEV L., BARZEL R., HOLDEN L. S., HUGHES J. F.: Art-based rendering of fur, grass, and trees. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques* (1999), pp. 433–438. doi:10.1145/311535.311607. 2

[KYYL08] KIM Y., YU J., YU X., LEE S.: Line-art illustration of dynamic and specular surfaces. In *ACM SIGGRAPH Asia 2008 Papers* (2008). doi:10.1145/1457515.1409109. 2

[LD06] LUFT T., DEUSSEN O.: Real-time watercolor illustrations of plants using a blurred depth test. In *Proceedings of the 4th International Symposium on Non-Photorealistic Animation and Rendering* (2006), ACM, pp. 11–20. doi:10.1145/1124728.1124732. 5

[LLC*10] LAGAE A., LEFEBVRE S., COOK R., DEROSE T., DRETTAKIS G., EBERT D., LEWIS J., PERLIN K., ZWICKER M.: A survey of procedural noise functions. *Computer Graphics Forum* (2010). doi:10.1111/j.1467-8659.2010.01827.x. 2

[LSF10] LU J., SANDER P. V., FINKELSTEIN A.: Interactive painterly stylization of images, videos and 3d animations. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2010), pp. 127–134. doi:10.1145/1730804.1730825. 2, 10

[MB13] MCGUIRE M., BAVOIL L.: Weighted blended order-independent transparency. *Journal of Computer Graphics Techniques (JCGT) 2*, 2 (2013), 122–141. 5

[Mei96] MEIER B. J.: Painterly rendering for animation. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (1996), ACM, pp. 477–484. doi:10.1145/237170.237288. 2, 10

[PD84] PORTER T., DUFF T.: Compositing digital images. *SIGGRAPH Comput. Graph. 18*, 3 (1984), 253–259. doi:10.1145/964965.808606. 5, 6

[PFS03] PASTOR O. M., FREUDENBERG B., STROTHOTTE T.: Real-time animated stippling. *IEEE Comput. Graph. Appl. 23*, 4 (2003), 62–68. doi:10.1109/MCG.2003.1210866. 2

[PHWF01] PRAUN E., HOPPE H., WEBB M., FINKELSTEIN A.: Real-time hatching. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (2001), p. 581. doi:10.1145/383259.383328. 2

[SBB17] SUAREZ J., BELHADJ F., BOYER V.: Real-time 3d rendering with hatching. *Visual Computer 33*, 10 (2017), 1319–1334. doi:10.1007/s00371-016-1222-3. 2

[SSGS11] SCHMID J., SENN M. S., GROSS M., SUMNER R. W.: Overcoat: an implicit canvas for 3d painting. *ACM Trans. Graph. 30* (2011), 28. 2

[USS*18] UMENHOFFER T., SZIRMAY-KALOS L., SZÉCSI L., LENGYEL Z., MARINOV G.: An image-based method for animated stroke rendering. *The Visual Computer 34*, 6-8 (2018), 817–827. doi:10.1007/s00371-018-1531-9. 2

[USSK11] UMENHOFFER T., SZÃL'CSI L., SZIRMAY-KALOS L.: Hatching for motion picture production. *Computer Graphics Forum 30*, 2 (2011), 533–542. doi:10.1111/j.1467-8659.2011.01878.x. 2

[VB15] VERGNE R., BARLA P.: Designing gratin, a gpu-tailored node-based system. *Journal of Computer Graphics Techniques (JCGT) 4*, 4 (2015), 54–71. URL: http://jcgt.org/published/0004/04/03/. 8

[VBTS07] VANDERHAEGHE D., BARLA P., THOLLOT J., SILLION F. X.: Dynamic point distribution for stroke-based rendering. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques* (2007), Eurographics Association, pp. 139–146. 2

[vW91] VAN WIJK J. J.: Spot noise texture synthesis for data visualization. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques* (1991), ACM, pp. 309–318. doi:10.1145/122718.122751. 6

[Wal81] WALLACE B. A.: Merging and transformation of raster images for cartoon animation. *SIGGRAPH Comput. Graph. 15*, 3 (1981), 253–262. doi:10.1145/965161.806813. 5

[Wor96] WORLEY S.: A cellular texture basis function. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* (1996), pp. 291–294. doi:10.1145/237170.237267. 2, 3

[YHGT10] YANG J. C., HENSLEY J., GRÜN H., THIBIEROZ N.: Real-time concurrent linked list construction on the gpu. In *Proceedings of the 21st Eurographics Conference on Rendering* (2010), pp. 1297–1304. doi:10.1111/j.1467-8659.2010.01725.x. 5, 8

© 2021 The Author(s)
Computer Graphics Forum © 2021 The Eurographics Association and John Wiley & Sons Ltd.