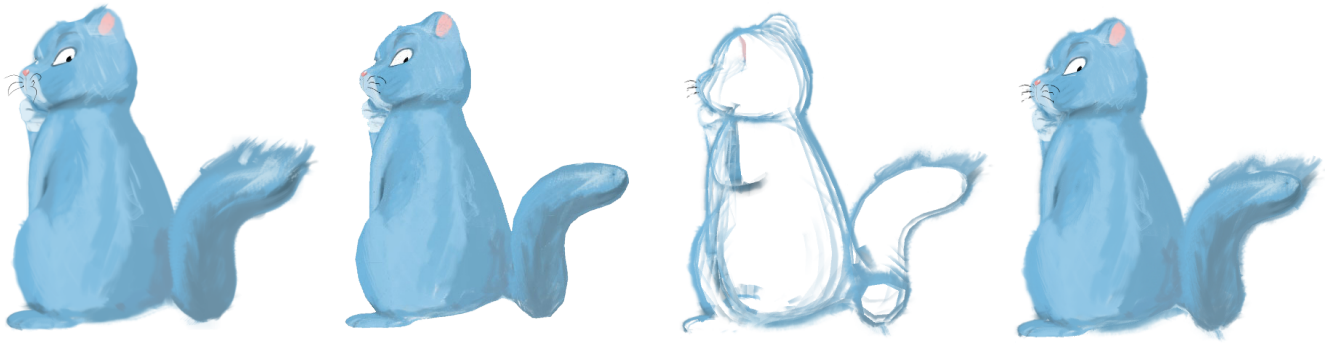
Fin Textures for Real-Time Painterly Aesthetics

Nicolas Imhof[1,2] Antoine Milliez[1,2*] Flurin Jenal[3,4] René Bauer[4] Markus Gross[1,2] Robert W. Sumner[1,2]

[1] ETH Zürich [2]Disney Research Zurich [3]Game Expressions!
[4]Zürcher Hochschule der Künste (ZHdK) Specialization in Game Design



Original scene Proxy geometry texture Fin texture Proxy and fin textures combined

Figure 1: *This painterly cat can be rendered at 165 FPS using our fin texture approximation while the original scene achieves only 4 FPS.*

Abstract

We present a novel method for real-time stylized rendering in video games. Recent advances in painterly character authoring and rendering allow artists to create characters represented by 3D geometry as well as 3D paint strokes embedded on and around that geometry. The resulting 3D paintings are rendered in screen space using special-purpose offline rendering algorithms to achieve a unique painterly style. While providing novel styles for offline rendering, existing techniques do not support real-time applications. In this paper, we propose a method to interactively render these complex 3D paintings with a focus on character animation in video games. After observing that off-surface paint strokes can be interpreted as volumetric data in the proximity of 3D meshes, we review existing volumetric texture techniques and show that they are not adapted to paint strokes, which can be sparse and have a significant structure that should be preserved. We propose a method based on fin textures in which mesh edges are extended orthogonally off the surface and textured to replicate the results of the custom offline rendering method. Our algorithm uses a per-pixel normal calculation in order to fade in fin textures along boundary views. Our results demonstrate real-time performance using a commodity game engine while maintaining a painterly style comparable to offline methods.

Keywords: real-time rendering, stylization, painterly rendering, non-photorealistic rendering

1 Introduction

Through their unique combination of visual, narrative, auditory, and interactive elements, video games provide an engaging medium of expression within our society. Video game exhibitions at top art museums such as the Museum of Modern Art [MoMA Press 2012] and the Smithsonian American Art Museum [Melissinos and O'Rourke 2012] attest to the fact that games have grown to be respected as an art form on par with film and animation. The visual design of a video game plays a significant role in the game's overall artistic impact. In the design phase, artists craft a vision for the game's look that supports the interaction style and narrative significance of the game. For example, the soft and glowing aesthetic of Flower [Chen 2009] supports the game's poetic nature, while the dark and gritty visuals of Heavy Rain [Cage 2010] enhance the game's film noir style.

Although a game's visuals contribute greatly to its overall feeling and impact, fully realizing the desired artistic vision for a game within the constraints of modern game engines is often impossible. Games are, by nature, interactive and rely on a sophisticated set of technological tools to support character models, rigging, animation, environments, camera control, lighting, texturing, and rendering within this interactive setting. The game engine, which encompasses this technology, must deliver stunning, rendered imagery at high frame rates to support smooth interaction. The strict real-time demand naturally requires tradeoffs in the engine's overall visual expressivity. As a result, the game engine may not accommodate the visual style envisioned for a game, requiring alterations to conform to the engine's technical limitations. Consequently, the final look of the game may deviate significantly from the artist's original vision.

Our work expands the aesthetic range of video game styles by reformulating costly offline expressive rendering methods to work in real time using commodity game engines. We place special attention on the difficult case of game characters, which are particularly challenging since they are animated and can be viewed from any perspective. In order to give the artist direct control over the character's visual style, we build upon a stroke-based 3D painting and animation system called OverCoat [Schmid et al. 2011; Baran et al. 2011; Bassett et al. 2013] that allows artists to craft a character's look through painting with expressivity that is akin to creating 2D concept art. While OverCoat requires a costly, custom, offline rendering step, we propose a new formulation that can be rendered in real time while maintaining comparable aesthetic quality. Since our goal is to open up new visual styles to as many game designers as possible, we develop our work for the industry standard Unity game engine [Unity 2015].

*e-mail:antoine@disneyresearch.com

Technically, our method expands on the concept of shell textures [Meyer and Neyret 1998]. For each mesh polygon, we generate edge polygons, or "fins," by extending the polygon edges orthogonally off the surface. We then provide an algorithm to generate alpha textures for the fins based on an input OverCoat painting. We use a per-pixel normal calculation in order to fade in such polygons along boundary views. The method fits naturally within Unity's rendering procedure, and only requires a back-to-front polygon sort as overhead. We extend character skinning weights to deform our fin meshes to support animation. Taken together, our method can reproduce the painterly aesthetic of OverCoat at real-time speeds. Figure 1 shows one example.

Our core contributions include an edge polygon representation, a rendering procedure for per-pixel boundary visibility, and a texture calculation method designed for offline stroke-based non-photorealistic rendering methods. We show several examples of how our method expands the range of aesthetic styles that can be achieved with a commodity game engine.

2 Related Work

Stylized rendering in animation

Non-photorealistic rendering aims at depicting a digital scene using a specific visual style. Many industrial applications exist, in particular in the context of CAD, where specific shading algorithms can improve the legibility of technical illustrations, as demonstrated in [Gooch et al. 1998]. However, most existing non-photorealistic rendering techniques were developed for a specific aesthetic intent. In particular, a large palette of methods target painterly rendering, in an effort to stylize 3D objects and characters by reproducing the look of traditional 2D paintings.

Two major research directions can be identified within painterly rendering. First, following the seminal work by Meier ([Meier 1996]), a variety of methods place particles in screen space or directly onto 3D geometry to act as seeds for the instantiation of paint strokes. The other main lineage of work consists of screen space image processing methods that act on videos or rendered images. Techniques include example-based stylization [Hertzmann et al. 2001] and screen space stroke generation [Litwinowicz 1997]. We refer the reader to Hedge, Gatzidis, and Tian's review article [Hegde et al. 2013] for a more exhaustive study of existing painterly rendering methods.

In our work, we wish to place the artist at the core of the stylization process so that they can directly craft the desired painterly look. Offline methods such as WYSIWYG NPR ([Kalnins et al. 2002]), Deep Canvas ([Katanics and Lappas 2003]), and OverCoat ([Schmid et al. 2011]) share this goal. OverCoat's novelty resides in the possibility to place paint strokes around meshes, which is analogous to painting in 2D outside of strict contours, thus allowing for fluffy painterly characters to be authored and rendered. Recent extensions support character animation [Bassett et al. 2013] while maintaining motion and temporal coherence, which are challenging but critical qualities in non-photorealistic rendering [Bénard et al. 2011]. Although ideal for character stylization, OverCoat's rendering method as well as the more elaborate one published in [Baran et al. 2011], both target offline rendering, and are not suitable for framerate demanding applications such as video games. As such, we use the OverCoat framework as the basis for our research and develop an algorithm to reformulate OverCoat's costly, offline, special-purpose rendering algorithm into one that is amenable to commodity game engines in real time.

Stylized rendering in interactive applications

Some expressiveness of non-photorealistic rendering has already been leveraged in video games. In particular, emphasis has been put on flat-looking renderings, mimicking 2D celluloid animations. Such a visual style was first brought to video games using painted pre-rendered flat textures in Fear Effect [Platten 1999], and was first computed in real time in Jet Set Radio [Kikuchi 2000], thus pioneering the technique now commonly referred to as *cel shading*. Cel shading has since become common in real-time rendering applications such as video games. Many variations exist, using shaders to enforce a specific color scheme, simulate unrealistic lighting, or copy features commonly found in 2D art, such as rendering the silhouette of a character using tapered lines.

More advanced real-time stylization methods have been published in the past, aiming at achieving existing non-photorealistic rendering styles in real-time. In [Markosian et al. 1997], the authors propose a real-time method for line-based stylization. More specific styles can be achieved in real-time, such as hatching ([Praun et al. 2001]), line art rendering ([Elber 1999]) or charcoal drawing ([Majumder and Gopi 2002]). These real-time non-photorealistic rendering techniques are all fantastic for replicating particular, specialized styles, but offer only a restricted amount of flexibility to the artist over the final appearance. Fine-scale customizations in character appearance are typically not possible. Furthermore, programmable interfaces such as those inspired by cel shading require technical skills, and are an indirect way of controlling the final look of a rendering. By targeting a 3D painting system that provides direct control over character stylization, we bring this new level of stylized control to game design.

Capturing and rendering volumetric shells

Paint strokes embedded around 3D characters can be interpreted as volumetric structures. Several publications have already targeted volumetric data rendering around meshes. Indeed, volumetric structures are inherent to realistic digital scenes, and adding hair, fur, or small-scale geometry to the surface of a 3D character increases its visual richness. In the early days of computer graphics, modeling complex structures such as fur using geometry was too complex for state-of-the-art hardware. Kajiya and Kay proposed a seminal solution using texels [Kajiya and Kay 1989] that inspired researchers to use the space surrounding a mesh, commonly referred to as *shell space*, to embed renderable data.

In a similar fashion to textures that get applied to 3D models, texels with toroidal symmetry can be deformed to fit in each shell around a mesh. Such volumetric shell textures are used to add repetitive detail to a 3D model. Neyret extended that technique to shell textures around arbitrary resolution meshes to render complex natural scenes [Neyret 1998]. Further works making use of shell space include [Chen et al. 2004], in which the authors define shell texture functions, that describe complex volumetric materials around meshes for advanced rendering in the context of subsurface scattering. Porumbescu and colleagues present a bijective mapping between shell space and texture space, called a shell map, in order to synthesize geometric detail onto meshes [Porumbescu et al. 2005].

These methods target offline rendering through ray-tracing, or geometry generation. Meyer and Neyret [Meyer and Neyret 1998] introduce a technique to slice a shell texture into layered polygons, enabling real-time rendering of shell textures using z-buffers. Sliced, or layered shell textures were then used for rendering fur using level of detail [Lengyel 2000] and over arbitrary surfaces [Lengyel et al. 2001]. These methods are powerful for stochastic and repetitive data like fur, but are not directly applicable in our context, since we aim at reproducing 3D paintings where every

locally painted detail contributes to the artist's intended character design.

In order to render 3D paintings using shell textures, we must devise a new method for reformulating paint strokes rendered in screen space as shell textures. Some existing techniques target the related problem of rendering complex 3D data acquired from screen space capture. In particular, [Matusik et al. 2002] capture *opacity hulls*, which are rendered by projecting surfels onto the screen. Further work presented in [Vlasic et al. 2003] provide a hardware-oriented algorithm to render *opacity light fields* using a multi-pass rendering. Both methods impose specific rendering algorithms that target real-time lighting. Our method is designed to be compatible with commodity game engines and only requires back-to-front polygon sorting and a one-pass OpenGL rendering. More recently, Okabe and colleagues [Okabe et al. 2015] compute 3D models of fluid in motion from images. These three methods must infer the shape of the captured object or phenomenon from the 2D views they use as input. Our fin texture method makes use of the 3D paintings' proxy geometry which gives the artist control over the complexity and topology of the mesh.

3 Method

Overview

Our method takes as input a 3D painting as well as an offline non-photorealistic renderer for such paintings. For this paper we used the OverCoat method described in [Schmid et al. 2011] where a 3D painting consists of paint strokes positioned in 3D space around a proxy geometry. At render time, those paint strokes are projected to the camera space and are populated with paint splats, as shown in Figure 2.

Taking inspiration from shell textures, we use the proxy geometry to generate a fin mesh that will ultimately be rendered in real-time. We use OverCoat's rendering routine to compute a texture for each polygon of the fin mesh. At runtime, the fin mesh is rendered using per-fragment alpha blending depending on the camera state. In this section, we explain the different steps of the fin mesh construction as well as the texture acquisition. In section 4, we present the results rendered using this method.

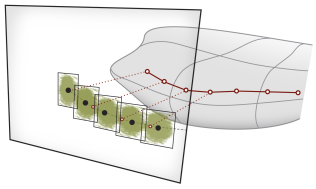


Figure 2: *The stroke rendering model in OverCoat (figure courtesy of Schmid and colleagues [2011])*

3.1 Fin Mesh Generation

Paint strokes located around 3D geometry can be seen as volumetric data. Traditionally, volumetric data around a mesh can be rendered using ray tracing as originally presented in [Kajiya and Kay 1989], or using shell textures as can be seen in [Meyer and Neyret 1998]. We are approximating paintings that cannot be exactly represented by 3D data since they are rendered in screen space, making a traditional ray tracing approach unsuitable. We therefore choose to follow an approach inspired by shell textures, and create a fin mesh by extruding the edges of a proxy 3D mesh into fin quads.

The 3D painting used as input in our method comprises a 3D proxy mesh. We extrude each edge of the proxy geometry by offsetting its vertices along their normal. Figure 3 shows this process on a toy example, while Figure 4 shows a triangle mesh and the generated fin mesh using this method.

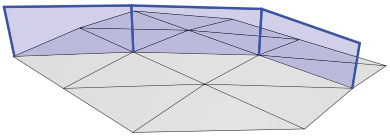
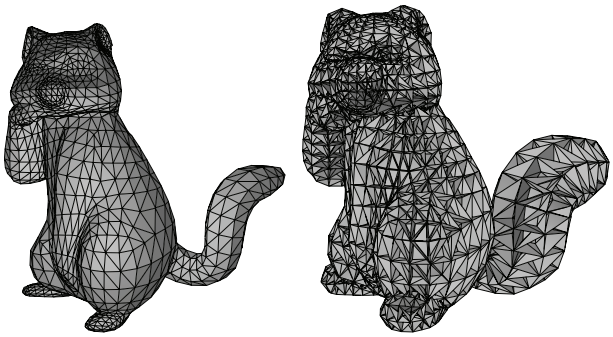


Figure 3: *Example fins extruded from mesh edges*



(a) Cat proxy geometry (b) Fin mesh

Figure 4

To approximate a 3D painting in an ideal way, the volume spanned by the fins should tightly enclose the paint strokes. We let the user define what offset should be used when extruding the proxy geometry edges. In cases of non-uniform paint repartition, the user can split the proxy geometry in parts and prescribe different offsets for each of the parts. In figure 5, the strokes around the cat's tail are much further away from the proxy geometry than the strokes covering the cat's body, and different offsets are used for those parts.

After the texture generation described in section 3.2, a final optimization process is applied to the mesh that discards the geometry that is not necessary due to completely empty or transparent textures.

(a) Offset mesh (b) Offset mesh with 3D painting

Figure 5: *The resulting mesh, when offsetting the vertices by their normals, is shown in* (a). *The OverCoat painting in* (b) *is mostly contained within the volume spanned by the fin mesh.*

3.2 Capturing Fin Textures

The fin rendering, as explained in section 3.3, consists in depicting the original 3D painting using a textured proxy geometry mesh as well as textured fin quads that are faded in as they face the camera position.

We provide two different approaches for capturing the textures for the proxy geometry and fins. Both methods share the idea of rendering for each polygon its neighboring volume using a camera aimed as orthogonally as possible at the polygon. This technique is similar to the slicing technique employed in [Meyer and Neyret 1998], however theirs cannot be directly applied to the context of screen space rendered paint strokes. Moreover, in our context both types of polygons will be rendered using different procedures which both require specific texture captures.

In the original OverCoat method, 3D paint strokes are sampled in screen space. This means that as the camera moves around a paint stroke, its projected length on the screen changes, and the number and position of the paint splats it carries constantly changes. In our work, paint strokes can often span more than one proxy geometry triangle. Directly using that rendering method would mean that the stroke is sampled differently when captured onto adjacent proxy geometry triangles, which inevitably leads to discontinuities. We therefore force the paint strokes to be sampled in world space instead of screen space for our application. This leads to minimal changes in appearance while allowing us to capture consistent textures.

Proxy geometry textures

In order to generate the texture for a single proxy geometry polygon, we first triangulate it. We then successively place an orthographic camera orthogonally above each triangle and set its viewport to match the triangle shape. The near and far planes of the camera are set to only capture the volume between the triangle and the top of the fins it touches. Challenges arise from such a simple approach and we explain here how we tackle them.

First, rendering using an orthographic camera placed above two adjacent triangles will ignore a volume above their common edge in the convex case, as shown in Figure 6. In the concave case, we end up rendering splats multiple times. We tackle this problem by first projecting each splat carried by a paint stroke onto its closest mesh polygon. We refer to the splat's new position as the *projected splat position*. Note that since splats then lie directly on polygons, special care has to be taken depending on the renderer's precision. Typically, offsetting the near and/or far planes of the orthographic camera by an *epsilon* ensures that all splats will be rendered during the texture capture phase.

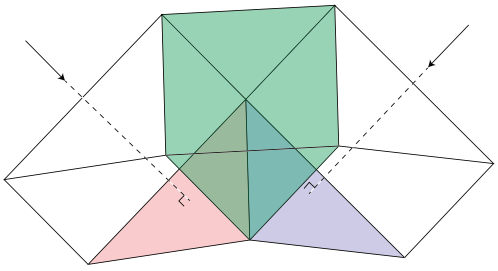
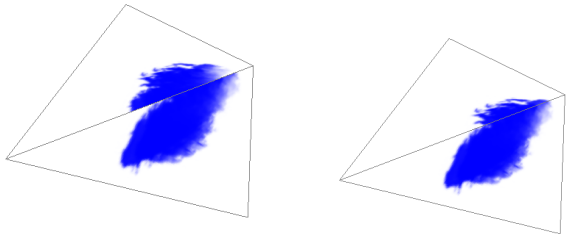


Figure 6: *Capturing textures for the red and blue triangles using orthographic cameras ignores the green volume.*

Another challenge comes from splats that contribute to more than one triangle. This scenario happens in all the 3D paintings we observed, for example when painting using a large paint stroke diameter relative to the proxy geometry triangle size, or when paint strokes are embedded close to proxy geometry edges. This means that splats cannot be discarded during the capture solely based on the position of their center. When rendering the volume above a proxy mesh triangle t, we therefore consider all splats which radius puts them within reach of t. Each of those splats can potentially contribute to the texture being computed. Splats placed on an adjacent triangle are rotated around the shared edge between the two triangles to lie in the plane defined by t. Figure 7 shows the difference of rotating a splat in its correct position rather than offsetting it. For more distant splats, we use the shortest path from the splat's projected position to the current triangle t and execute a series of such rotations along the edges that connect the triangles of the path, which corresponds to a "wrapping" of the splat around the mesh.



(a) Projecting splats (b) Rotating splats

Figure 7: *Simply projecting the splats on the target plane as in* (a) *leads to discontinuities in the textures. Using rotations solves this problem as shown in* (b).

When computing the texture for a specific triangle t, several splats in its vicinity are projected onto t. When rendering using an orthographic camera placed above t, the splats are therefore at an equal depth from the camera, although their original positions would exhibit a depth order. Moreover, splats on adjacent triangles are seen at inconsistent depths from different cameras, as shown in Figure 8. In order to render the splats onto the texture with consistent depth information, we therefore sort them based on the distance between their original position and the proxy mesh. This ensures a consistent depth ordering of the splats even across triangle borders. Note that for splats having exactly the same distance to the proxy geometry, we render the most recently painted stroke last, inspired by OverCoat [Schmid et al. 2011].

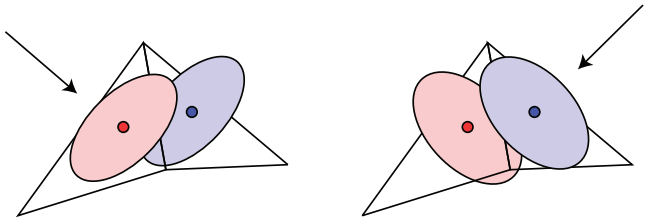
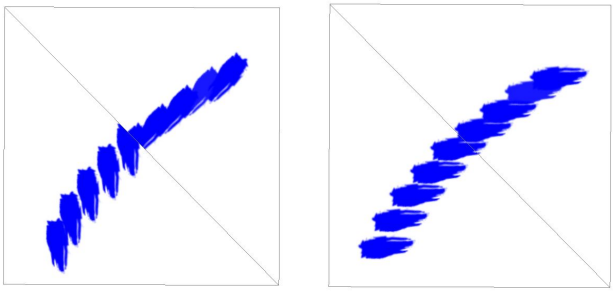


Figure 8: *The two cameras assign the splats an inverted depth order when relying on the camera space z-position.*

Finally, since the orientation of splats in OverCoat is defined in screen space, viewing paint strokes from cameras with different up vectors causes inconsistencies in the rendered splat orientations, as seen in 9 (a). In general it is not possible to assign a continuous orientation across a mesh without singularities, not even on simple meshes such as topological spheres, as stated by the Poincaré-Hopf



(a) Uncorrected orientation (b) Fixed orientation

Figure 9: *Dealing with the splats' orientation is necessary when rendering small parts of the scene to generate textures. Discontinuities arise at the polygon borders if the camera orientation changes.*

theorem. In our case however, we obtain satisfying results simply by assigning a global orientation to all triangles. We compute that global orientation by projecting a single arbitrary vector onto each face. For faces where such a projection does not exist, we propagate the direction using the neighboring faces' triangles. Figure 9 (b) shows the influence of using such corrected orientations. While one could envision defining a consistent orientation for every stroke individually, we found that our method provided good results.

Fin textures

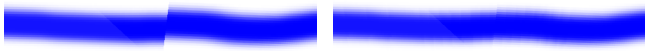
Fins are represented as quads and may not be planar. If we triangulate each fin and capture the textures for each triangle separately, special care will have to be taken to avoid discontinuities between the obtained textures, in a similar fashion as the method described for the proxy geometry texture. However, in most cases fin quads are not strongly distorted. We therefore simplify the texture capture by using one stroke rendering pass for each fin. We place the camera so that it faces the lowest triangle orthogonally. Since the lowest triangle shares an edge with the proxy geometry, we can assume that it is less dependent on the fin deformation.

In a similar fashion to the proxy geometry texture capture, the near and far planes of the camera are adjusted to contain the whole volume spanned by the proxy mesh triangle and its connected fins. Analog issues arise from that method. For example, similar to the ignored volume depicted in Figure 6, rendering the volume spanned by fins and cropping it onto a fin quad will ignore some splats. In general, fixing this issue by projecting splats as described above introduces unpleasing perspective distortions, and the best way to remove such artifacts is to modify the input mesh to be locally smoother.

Since the view direction can change dramatically from one fin quad to the next, discontinuities between neighboring fins cannot be avoided. We however propose a solution consisting in capturing each fin texture several times, while interpolating the camera position from a fin to the next. We generate the final fin texture by concatenating strips of the multiple generated textures. Figure 10 shows how this method helps remove discontinuities between neighboring fin textures.

3.3 Fin Texture Rendering

To render our approximation scenes, we apply at each frame a back-to-front sorting on all the polygons and use per-fragment alpha blending to achieve visually correct results with our semi-transparent textures. While the proxy geometry is always rendered,



(a) No interpolation (b) With view interpolation

Figure 10: *Interpolating the view direction helps to reduces hard changes in adjacent fin textures. While the change in the view direction is clearly visible in (a), a smooth transition is achieved in (b).*

fin textures should only be visible when their normal is close to orthogonal with the camera plane. Existing methods such as [Lengyel et al. 2001] blend whole fin quads in and out at once, which provides satisfying results when used to render noisy structures like fur. However, in our context, having a fin blended in while its neighbors are not visible creates discontinuities, and makes the fin structure obvious to an observer. Since our goal is to give the impression that a 3D painting is being rendered in real-time, we fade fin textures in and out per fragment, using a normal continuously interpolated across fin polygons.

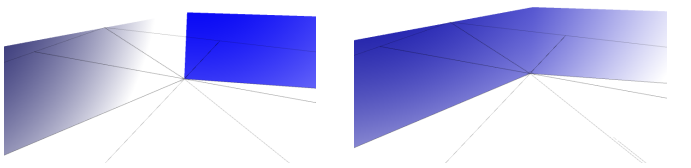
Contrary to manifold meshes, interpolated normals across fins are not straightforward to define for fin meshes. We therefore add an additional step to our workflow prior to rendering, that computes proper vertex normals for fin meshes.

Fin mesh vertex normal interpolation

When rendering fins, we want to use a per-fragment normal direction for blending. That normal direction should continuously transition from a fin to its next visible neighboring fins, to avoid discontinuities in the fin mesh rendering. While defining a continuous normal direction on a manifold mesh can easily be obtained by interpolating vertex normals across faces, the concept of a "neighboring fin" is not well defined in our case. We observed that when viewing a fin mesh from an arbitrary angle, fins that are visible and appear to be neighbors are fins that make a close to flat angle with each other. This makes sense, since two fins that share an edge will both be visible if they are both facing the camera.

We therefore conduct an additional processing step on our fin mesh, that needs to run only once after the fin mesh generation. During that processing step, we assign to each fin its best neighboring fin. Since each fin quad is based on a proxy mesh edge, two fin quads are as good neighbors as their respective edges on the mesh make an flat angle. Our neighbor assignment is fairly simple: to each mesh edge, we first find the pair of its best neighboring unmarked edges at its two ends, while discarding pairs of neighbors making a too sharp angle (we chose $\pi/2$ as an angular threshold for discarding). Once all pairs are listed, we define two edges as neighbors if they listed each other as their best neighbor, and we mark them. We then iterate until no new matching is found. Finally, edges left unassigned are paired to their best neighbor, even if that one is already marked.

The neighboring information on the edges is transferred to the fins they support, and this information on fin continuity lets us render fin meshes without discontinuities, as exhibited in Figure 11. In some cases, a fin does not admit a valid neighbor, due to the proxy geometry not providing the corresponding edge with neighbors making a valid (over $\pi/2$) angle. This can create artifacts where a sole fin is rendered on the silhouette of a mesh. Using proxy geometry with a consistent edge flow helps avoiding these issues.



(a) Fin mesh with face normals (b) Adjusted fin mesh vertex normals

Figure 11: *Using the fin mesh's face normals for the fading results in harsh discontinuities as shown in the left figure. Calculating the interpolated vertex normals on the fin mesh helps to obtain smooth transitions on the fins.*

Scene	OverCoat FPS	Unity FPS	OverCoat Splats	Unity Triangles
Cat	4.2	165	196468	18214
Magicans	0.7	55	1469633	49790
Panda	12.5	190	60992	16204
Bee	1.3	125	861114	23604
UFO	X	72	417215	27122
Van Gogh	1.4	180	527143	16205
Dog	1.5	120	677355	28231

Table 1: *Comparison of frame rates when rendering paintings and their fin-texture approximations. Note that there is no value for the OverCoat FPS field of the UFO since that character is assembled in Unity from five independent OverCoat scenes.*

4 Results

All the results shown in this paper are rendered in Unity [Unity 2015] using our custom Unity surface shader for blending polygons in and out, as well as our back-to-front polygon sorting implementation. Rendered images are attached to the end of this paper in section 6. The supplemental video shows side by side comparisons of OverCoat paintings with the output of our method, as well as live game sequences, that proof that our implementation can be used conveniently in commodity game engines such as Unity. The output of our method can easily be integrated with other special effects, as is demonstrated with the smearing and stretching effects shown in the video.

Frame Rate

Table 1 shows the frame rate obtained when rendering different scenes using OverCoat, as well as rendering the fin-texture approximations using our method in Unity. Our measurements were taken running Unity on an Intel Core i7 2.80 GHz, with 12 GB of RAM and an NVIDIA GeForce GTX 580. The frame rate of the approximated scenes is mostly dependent on the number of triangles of the input geometry. Our triangle sorting script uses a fast linear-time bucket sort, and most of the rendering time in Unity is spent accessing the mesh data using the Unity API, and writing it back once sorted. Note that for static game objects, the mesh vertex positions do not change and the mesh data does not need to be read at every frame. Since the bucket sort does not guarantee the absolute correct order of triangles, flickering artifacts can appear if the number of buckets is chosen to be too low. Therefore, we added to our sorting algorithm the possibility to adapt the number of buckets used. In our tests, 1000 buckets were usually enough to avoid popping artifacts. The FPS table shows that our approximations can be rendered with a substantial speed up of one to two orders of magnitude compared to the original OverCoat scenes, making them suitable for real-time applications such as games.

Offline Approximation Time

The running time of the offline algorithm linearly depends on the number of triangles of the proxy mesh. Without fin interpolation, for each triangle $L + 3$ scene renderings are necessary, where L is the number of layers used and 3 is the number of fins per triangle. The second important factor is the rendering time for the scene. We render a lot of images that only show a small part of the scene. Therefore we calculate the bounding box of each stroke and use it to skip strokes efficiently when they are not intersecting the rendered volume. The bounding boxes provide a huge speed up which allows regular scenes, like the ones referred to in table 1, to be approximated in ten to twenty minutes. Using many fin view direction interpolation steps can increase the approximation running time up to one or two hours, but are not necessary in general.

Rendering Quality

The rendered approximations are a satisfying reproduction of the original 3D paintings. We noticed that input meshes modeled as quad meshes, even if they are then triangulated, achieve in general better results than models of arbitrary mesh topology. Indeed, they exhibit a consistent edge flow, with few sharp angles between adjacent mesh edges. The fins created from such edges have a similar normal to their neighboring fins, which is beneficial for our results as explained in section 3.3.

Contrary to existing shell texture methods such as [Lengyel et al. 2001], we only used a proxy mesh and fins, and did not define layered textures. We observed that in most cases our method achieves satisfying results, while benefiting from a lower polygon count. A low polygon count naturally increases the frame rate during rendering, but also helped us target real-time rendering using the Unity game engine. Indeed, that engine enforces a maximum of 65536 triangles or vertices in a single mesh, and that limit can be quickly reached when using many layers or complex input meshes. Nonetheless, a single fin quad per edge was sufficient to render the fluffy bee body, the yellow smoke around the genie, or the dog's tail in figure 14.

Rare artifacts along object silhouettes can appear due to poorly suited fins. Indeed, if the original painting exhibits a striking feature such as a single paint stroke relatively far from all its neighboring fins, it can be captured by several fins, or by a distant fin, as described in 3.2, and the feature can then be duplicated when rendering the approximation of the painting, or seen away from its intended location, as shown in Figure 12.

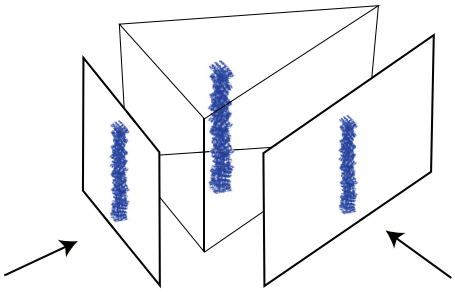


Figure 12: *The blue paint stroke is captured by all three surrounding fins.*

In the absence of well-fitting proxy geometry for sparse paint strokes, additional layers can help establish a good real-time approximation of the input painting. Our method can easily support such a layering. By connecting the edges at the top of fin quads into new triangles, and basing new fins on those triangles, layered shells can be created. Capturing textures for the layered geometry can be done using the method described in 3.2, and we capture

(a) (b) (c)

Figure 13: *The OverCoat scene* (a) *of painted grass is approximated using a single subdivided plane as input mesh. While in figure* (b) *the grass is still connected with 15 layers, it clearly splits up with five layers in figure* (c)*, revealing the underlying structure.*

strips of fins using a single camera view, to avoid discontinuities as mentioned in 3.2. Figure 13 shows painted grass on a flat plane mesh. If the plane tesselation does not closely match the positions of the grass strokes, fins cannot capture the grass appearance in a satisfying way, as previously described and illustrated in Figure 12. Adding layers help convey the volumetric appearance of the grass.

One important variable in our implementation is the function for blending fin textures in and out. The default function described in 3.3 was satisfying in most of our tests. However, in specific regions where fins are relatively large compared to the mesh they base on, the fin geometry becomes visible to the user. We let the user correct this behavior by allowing them to change the fade in speed and threshold for selected parts of a character. The cat example visible in the trailer of this paper has longer fin quads along its tail than on its body, which was specified by the user to capture the whole volume spanned by the 3D paint strokes. The structure of such long fins can easily be seen using a default implementation, the user could therefore decide to blend the tail fins 50% faster than the body ones, to create a fuzzier effect on the tail.

5 Conclusion And Future Work

We presented a novel algorithm that uses precomputation to generate fin-texture approximations of complex 3D paintings that preserve the artist's original design. The approximations can be rendered in real-time and combine layer textures (on and around the proxy geometry) together with fin textures that are orthogonal to the proxy geometry and fade in using a per-vertex normal value when facing the camera. Our results show that layer textures on the proxy geometry reduce 3D paintings to textured meshes which exhibit sharp edges on screen, while the generated fin textures help convey the original painterly style in real-time by covering the sharp borders of the layer textures. While our composited results provide a satisfying depiction of the input 3D paintings, our experience helped us identify limitations of capturing and using textures from shells constructed by offset mesh generation.

First, as previously stated, using complex meshes or high numbers of shell layers can result in polygon counts that overshoot the Unity limitation. A possible solution to bypass this limitation is to split complex objects into several independent ones. In our current implementation, the back-to-front sorting of polygons is performed per object. Splitting paintings into several game objects would require adapting our sorting algorithm to be performed globally. While a naive implementation would slow down the rendering time, adding a collision detection pass across game objects would help optimize the polygon sorting to remain local across colliding objects.

The dependency of the results on adequate input geometry as well as the shell distortion could be circumvented by a better construction algorithm for the offset mesh. Although more elaborate algorithms for creating smooth offset meshes exist, to the best of our knowledge, no existing solution focuses on avoiding distorted shells. A possible solution could be to formulate the offset mesh construction as an optimization problem that would solve for the offset vector from each vertex on the proxy geometry mesh while penalizing the distortion of the shells. Additional energy terms could be used to avoid the self-intersection of shells or to force fin quads to be planar, making the capture of their texture more accurate.

An interesting way to further improve the appearance of the fins would be to create offset meshes with independent topology. Thus, parts of a mesh with a particularly high curvature could be subdivided as they are offset, while highly convex parts could be decimated as they are offset. In such a process, self intersections could be avoided and the number of polygons in offset meshes would be locally more suited to our application.

To the best of our knowledge, no real-time lighting solution for 3D paintings such as those presented in [Schmid et al. 2011] has been proposed. Indeed, 3D paint strokes rendered in screen space do not admit an exact volumetric description, and cannot be represented as a manifold. Adapting existing shading algorithms to such paintings is therefore an ill-defined problem. In our context, the shell mesh and fins approximating a painting have an exact 3D representation and could in theory be lit using traditional shading principles. Special care would have to be taken for fins as they should not be lit according to their normal, but according to the normal of the mesh layer to which they are attached. Whether an accurate 3D shading would be aesthetically pleasing when used on stylized structures such as our paintings remains to be explored.

References

BARAN, I., SCHMID, J., SIEGRIST, T., GROSS, M., AND SUMNER, R. W. 2011. Mixed-order compositing for 3d paintings. *ACM Transactions on Graphics 30*, 6.

BASSETT, K., BARAN, I., SCHMID, J., GROSS, M., AND SUMNER, R. W. 2013. Authoring and animating painterly characters. *ACM Trans. Graph. 32*, 5 (Oct.), 156:1–156:12.

BÉNARD, P., BOUSSEAU, A., AND THOLLOT, J. 2011. State-of-the-art report on temporal coherence for stylized animations. *Computer Graphics Forum 30*, 8, 2367–2386.

CAGE, D. 2010. *Heavy Rain*. Quantic Dream, February.

CHEN, Y., TONG, X., WANG, J., LIN, S., GUO, B., AND SHUM, H.-Y. 2004. Shell texture functions. In *ACM SIGGRAPH 2004 Papers*, ACM, New York, NY, USA, SIGGRAPH '04, 343–353.

CHEN, J. 2009. *Flower*. Thatgamecompany, February.

ELBER, G. 1999. Interactive line art rendering of freeform surfaces. *Computer Graphics Forum 18*, 3, 1–12.

GOOCH, A., GOOCH, B., SHIRLEY, P., AND COHEN, E. 1998. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '98, 447–452.

HEGDE, S., GATZIDIS, C., AND TIAN, F. 2013. Painterly rendering techniques: a state-of-the-art review of current approaches. *Computer Animation and Virtual Worlds 24*, 1, 43–64.

HERTZMANN, A., JACOBS, C. E., OLIVER, N., CURLESS, B., AND SALESIN, D. H. 2001. Image analogies. In *Proceedings of*

the 28th Annual Conference on Computer Graphics and Interactive Techniques, ACM, New York, NY, USA, SIGGRAPH '01, 327–340.

KAJIYA, J. T., AND KAY, T. L. 1989. Rendering fur with three dimensional textures. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '89, 271–280.

KALNINS, R. D., MARKOSIAN, L., MEIER, B. J., KOWALSKI, M. A., LEE, J. C., DAVIDSON, P. L., WEBB, M., HUGHES, J. F., AND FINKELSTEIN, A. 2002. WYSIWYG NPR: Drawing strokes directly on 3D models. *ACM Transactions on Graphics (Proc. SIGGRAPH) 21*, 3 (July), 755–762.

KATANICS, G., AND LAPPAS, T. 2003. Deep canvas: integrating 3d painting and painterly rendering. *Theory and Practice of Non-Photorealistic Graphics: Algorithms, Methods, and Production Systems*.

KIKUCHI, M. 2000. *Jet Set Radio*. SmileBit, June.

LENGYEL, J., PRAUN, E., FINKELSTEIN, A., AND HOPPE, H. 2001. Real-time fur over arbitrary surfaces. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, ACM, New York, NY, USA, I3D '01, 227–232.

LENGYEL, J. 2000. Real-time fur. In *Rendering Techniques 2000*, B. Péroche and H. Rushmeier, Eds., Eurographics. Springer Vienna, 243–256.

LITWINOWICZ, P. 1997. Processing images and video for an impressionist effect. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '97, 407–414.

MAJUMDER, A., AND GOPI, M. 2002. Hardware accelerated real time charcoal rendering. In *Proceedings of the 2Nd International Symposium on Non-photorealistic Animation and Rendering*, ACM, New York, NY, USA, NPAR '02, 59–66.

MARKOSIAN, L., KOWALSKI, M. A., GOLDSTEIN, D., TRYCHIN, S. J., HUGHES, J. F., AND BOURDEV, L. D. 1997. Real-time nonphotorealistic rendering. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '97, 415–420.

MATUSIK, W., PFISTER, H., NGAN, A., BEARDSLEY, P., ZIEGLER, R., AND MCMILLAN, L. 2002. Image-based 3d photography using opacity hulls. *ACM Trans. Graph. 21*, 3 (July), 427–437.

MEIER, B. J. 1996. Painterly rendering for animation. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '96, 477–484.

MELISSINOS, C., AND O'ROURKE, P. 2012. *The Art of Video Games: From Pac-Man to Mass Effect*. Welcome Books, March.

MEYER, A., AND NEYRET, F. 1998. Interactive volumetric textures. In *Rendering Techniques '98*, G. Drettakis and N. Max, Eds., Eurographics. Springer Vienna, 157–168.

MOMA PRESS, 2012. MoMA acquires 14 video games for architecture and design collection. Press Release, December.

NEYRET, F. 1998. Modeling, animating, and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics 4*, 1 (Jan.), 55–70.

OKABE, M., DOBASHI, Y., ANJYO, K., AND ONAI, R. 2015. Fluid volume modeling from sparse multi-view images by appearance transfer. *ACM Transactions on Graphics (Proc. SIGGRAPH 2015) 34*, 4, 93:1–93:10.

PLATTEN, J. Z. 1999. *Fear Effect*. Kronos Digital Entertainment.

PORUMBESCU, S. D., BUDGE, B., FENG, L., AND JOY, K. I. 2005. Shell maps. In *ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, SIGGRAPH '05, 626–633.

PRAUN, E., HOPPE, H., WEBB, M., AND FINKELSTEIN, A. 2001. Real-time hatching. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '01, 581–.

SCHMID, J., SENN, M. S., GROSS, M., AND SUMNER, R. W. 2011. Overcoat: an implicit canvas for 3d painting. *ACM Trans. Graph. 30* (August), 28:1–28:10.

UNITY, 2015. Unity Technologies. www.unity3d.com.

VLASIC, D., PFISTER, H., MOLINOV, S., GRZESZCZUK, R., AND MATUSIK, W. 2003. Opacity light fields: Interactive rendering of surface light fields with view-dependent opacity. In *Proceedings of the 2003 Symposium on Interactive 3D Graphics*, ACM, New York, NY, USA, I3D '03, 65–74.

6 Additional Results

Figure 14: *Left: OverCoat rendering, right: fin mesh rendered using our method*

Figure 15: *From left to right: Panda game scene, "Dog vs. Bird" characters, UFO game scene*