Master Thesis

Real-time rendering of proxy based 3D paintings using fin textures

Author(s):
Imhof, Nicolas

Publication Date:
2015

Permanent Link:
https://doi.org/10.3929/ethz-a-010541112

Rights / License:
In Copyright - Non-Commercial Use Permitted
Real-time rendering of proxy based 3D paintings using fin textures

Nicolas Imhof

Master Thesis
July 2015

Prof. Robert W. Sumner
Abstract

Recent advances in painterly character authoring and rendering let digital artists create characters represented by 3D geometry as well as 3D paint strokes embedded on and around that geometry. Existing methods provide intuitive authoring workflows based on 2D painting gestures, without constraining artists to paint directly on the geometry. Strokes painted on screen outside of the visualized 3D mesh can be embedded in the space around that geometry. The resulting paintings can be rendered in screen space and carry a unique visual style.

In this master thesis, we are in particular interested in the 3D painting authoring and rendering process presented in [SSGS11]. Such existing techniques, while providing very unique styles for the offline rendering of 3D animations, were never adapted to interactive or even real-time applications. In this master thesis, we propose the first method to render complex 3D paintings in real-time. This novel method can directly be applied to port stylized rendering to 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 which should be preserved. Taking inspiration from shell textures, we provide a method that creates relevant 3D geometry and computes alpha textures to closely approximate 3D paintings using a small number of polygons, that can be rendered in real-time in standard game engines.
Real-time rendering of proxy-based 3D paintings using fin textures

Introduction
In the recent years, the computer graphics community has shown a growing interest in achieving new visual styles for computer animated movies. Significant contributions came from Disney Research’s OverCoat software, developed in Zurich, allowing users to freely create complex 3D paintings. Recent interest also came from the computer game design community, where novel visual styles are a constant challenge in real-time. In that context, Disney Research Zurich and the Zurich University of the Arts (ZHdK) are collaborating in an attempt to display OverCoat paintings in real-time applications such as video games. Existing rendering techniques for OverCoat paintings are too involved to be directly applied and the student will have to approximate the 3D paintings using state of the art research in shell texture rendering.

Task Description
This project contains several challenges, since OverCoat uses its own file format that is not importable in any commercial game design package, and the rendering of OverCoat scenes is so far only performable in OverCoat.

Three major challenges have to be faced by the student:

- **Learning**
  The student will work in close collaboration with art students and professionals from the ZHdK. Such artists have strong creative skills but cannot contribute to the technical aspects of this project. It is required to understand the needs of a non-technical user, and make the appropriate decisions on the technical side. The student has to become familiar with OverCoat and Unity in order to implement an approximation algorithm.

- **Implementation**
  Being part of a long-term collaboration, the student is expected to put in application his scholar knowledge of project management, not forgetting that his code will surely be reused in the future. When experimenting with complex algorithms, small progressive increments are expected from the student for a continuous validation of his work.

- **Experiments**
  The developed pipeline should be tested and experimented in collaboration with the ZHdK student and validated as fail-proof. The approximation process should be simple to use and in general provide expected and satisfying results.

A final implementation running in real-time is of course the expectation of this project.

Remarks
A written report and an oral presentation conclude the thesis. The thesis will be overseen and supervised by Prof. Robert W. Sumner.

Timeline
Issue: January 25th / Submission: July 25th
Acknowledgements

First of all I would like to thank Disney Research for the great opportunity to write my thesis in such an interesting research area. In addition, my deepest appreciation goes to Prof. Robert W. Sumner for his inspiring supervision, encouragement and for sharing his deep knowledge and wisdom.

Further, I wish to express my sincere gratitude to my direct advisor at Disney Research, Antoine Milliez, for the excitement and catching spirit of adventure in regard to research and his proficient guidance with all his amazing knowledge in maths, computer graphics and animation.

In addition, I truly appreciate the creative input and artistic help from the digital artists at Disney Research, namely Alessia Marra and Maurizio Nitti, who created some of the amazing OverCoat scenes that I used for my thesis.

Speaking of artists, I must thank Flurin Jenal, a talented game designer from the Zurich University of the Arts (ZHdK), for the impressive game scenes he realised with the new method developed in this thesis.

I also would like to thank Manuel Braunschweiler, who tried to tackle the same challenge in his bachelor thesis with a different approach. He provided me with additional material and helpfully outlined the problems he ran into during his work.

Last but not least, I wish to thank all the people at Disney Research, especially the marvelous Animation Group. They made me enjoy my work-place every day and contributed to my great motivation to work on the thesis. I believe that I could have not found another thesis being more suitable for me.
Contents

List of Figures ix

1. Introduction 1
   1.1. Motivation ............................................. 1
   1.2. Ambition .............................................. 1
   1.3. Preliminary Investigations ............................. 2
   1.4. Concept ................................................. 2
   1.5. Contribution ........................................... 2

List of Tables 1

2. Related Work 3
   2.1. Stylized Rendering in Animation ....................... 3
   2.2. Stylized Rendering in Interactive Applications ........ 3
   2.3. Capturing and Rendering Volumetric Shells ............. 4
   2.4. "Kimono" Approximation ................................ 5

3. Method 7
   3.1. Overview .............................................. 7
   3.2. Fin Mesh Generation .................................. 7
   3.3. Capturing Fin Textures ................................ 9
      3.3.1. Proxy Geometry Textures ......................... 10
      3.3.2. Fin Textures ..................................... 13
   3.4. Fin Texture Rendering .................................. 14
      3.4.1. Fin Mesh Vertex Normal Interpolation ............. 14
## Contents

4. Results 17
   4.1. Frame Rate ................................................. 17
   4.2. Offline Approximation Time .................................. 19
   4.3. Rendering Quality ............................................ 19
       4.3.1. Results Review ........................................ 19
       4.3.2. Game Scenes ........................................... 24
       4.3.3. Comparison with Kimono Approximation .................. 27

5. Limitations and Future Work 31

6. Conclusion 33

A. Appendix 35
   A.1. Workflow Documentation ..................................... 35
       A.1.1. OverCoat Interface .................................... 35
       A.1.2. Approximation Interface ................................. 36
       A.1.3. Advanced Options ...................................... 36
       A.1.4. Unity Import ........................................... 38
       A.1.5. Approximation Viewer .................................. 41
   A.2. Additional Results ............................................ 42

Bibliography 45
List of Figures

2.1. Creating an equation for each pixel of the sample image by intersecting the viewing ray through the center of the pixel with the layers of the Kimono mesh. [Bra] .................................................. 6

3.1. The stroke rendering model in OverCoat (figure courtesy of Schmid and colleagues [SSGS11]) .................................................. 7
3.2. Example fins extruded from mesh edges .................................................. 8
3.3. .................................................................................................................. 8
3.4. 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. .................................................. 9
3.5. Convex case: Capturing textures for the red and blue triangles using orthographic cameras ignores the green volume. .................................................. 10
3.6. Concave case: Splats located in the green volume are considered twice, when capturing textures. Once for the red and once for blue triangle and do therefore get duplicated. .................................................. 11
3.7. Simply projecting the splats on the target plane as in (a) leads to discontinuities in the textures. Using rotations solves that problem as shown in (b). .................................................. 12
3.8. The two cameras assign the splats an inverted depth order when relying on the camera space z-position. .................................................. 12
3.9. 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. .................................................. 13
3.10. Interpolating the view direction helps to reduces hard changes in the fin textures. 14
List of Figures

3.11. 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. .......................... 15

4.1. Bee scene ......................................................... 20
4.2. Magican & Genie scene ......................................... 20
4.3. Dog scene ......................................................... 21
4.4. Cat scene (backside) .............................................. 21
4.5. Cat scene (front) .................................................. 22
4.6. The blue paint stroke is captured by all three surrounding fins. ......................... 23
4.7. The OverCoat scene (a) of painted grass is approximated using a single subdivided plane as input mesh. While in (b) the grass is still connected with 15 layers, it clearly splits up with five layers in (c), revealing the underlying structure. ........................................... 23
4.8. Game scene "Jumping panda" .................................. 24
4.9. Game scene "Overdog" ............................................ 25
4.10. Game scene "Attacking UFO" ................................... 26
4.11. The comparison of the "Dog" scene shows that in the old Kimono approximation the textures appear blurred, while for our new method the characteristics of the paint strokes are still preserved. ......................... 27
4.12. A closer look at the textures highlights the differences in quality and stroke characteristics preservation. Since the Kimono approximation solves for each texture pixel separately, single pixels can completely differ from their surroundings. .................................................. 28
4.13. In the "Element Cube" scene the layers can become disturbingly visible in the old approximation method, while the new method provides satisfying results. ......................... 28
4.14. In many cases the Kimono approximation runs into local minima, providing unsatisfying results. As the above pictures implies, even after several hours of calculation time, the textures of the "cat" scene are still in a bad condition. There is no other choice then to restart the approximation process over again and hope for better results. .............................. 29

A.1. Different scales for the genie’s body and smoke are used in (a), while in (b) only the smoke of the genie is approximated. .......................... 35
A.2. Approximation interface integrated in Overcoat ........................ 37
A.3. Different Unity texture settings illustrated on the blue cat. Artifacts appear at the fins’ end in (a). No such artifacts are visible in (b). ......................... 38
A.4. Different Unity texture formats illustrated on the blue cat’s tail. ..................... 39
A.5. The two Unity scripts that are necessary for the approximations to look correct. The *KimonoRenderer* in (a) ensures a back-to-front sorting of the mesh triangles on every frame. The *KimonoShader* controls the fading in and out of the fins. ......................... 40
A.6. As a possible example on how to extend the current shader, this cat scene uses an extended shader with additional color option for the fins. ......................... 40
A.7. Modifying textures ................................................................................................................. 41
A.8. *KimonoViewer* .................................................................................................................... 42
A.9. Van Gogh scene ....................................................................................................................... 43
A.10. Cat and Mouse scene - Both images show an approximation of the original scene with additional shadow projectors. .................................................. 43
A.11. Panda scene ................................................................. 44
A.12. Various approximated scenes ............................................ 44
List of Figures

xii
Introduction

1.1. Motivation

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 [MoM12] and the Smithsonian American Art Museum [MO12] 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 [Che09] supports the game’s poetic nature, while the dark and gritty visuals of Heavy Rain [Cag10] 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. The final look of the game may deviate significantly from the artist’s original vision. Extending the spectrum of visual styles achievable in real-time applications is therefore an ongoing research area.

1.2. Ambition

Our work attempts to expand 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 use a stroke-based 3D painting and animation system called OverCoat [SSGS11, BSS+11, BBS+13] 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 the same 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 [Uni15].
1. Introduction

1.3. Preliminary Investigations

Since its foundation in the year 2008, considerable interest and effort was put into non-photo-realistic rendering at Disney Research. This effort manifested itself in projects such as OverCoat. In a presentation by Disney Research at ZHdK, in which OverCoat was introduced to the art students, Flurin Jenal, a ZHdK game designer, envisioned combining OverCoat’s unique visual style with an interactive video game. He appealed with his idea to Disney Research and originated the initial motivation. Thereafter, Disney Research announced work on a bachelor thesis that aimed at finding a way to tackle the technical challenges. Although not perfect, the results, accomplished by using the OverCoat approximations of that thesis in a real-time game, evinced major potential.

We comment on the technical approach of said thesis in the related work chapter in section 2.4 and compare the results achieved by its method and ours in section 4.3.3.

1.4. Concept

Technically, our method expands on the concept of shell textures [MN98]. For each mesh polygon, we generate edge polygons, or "fins", by extruding 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.

1.5. Contribution

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. Taken together, our method can reproduce the painterly aesthetic of OverCoat at real-time speeds, even on low-end devices. We show several examples of how our method expands the range of aesthetic styles that can be achieved with a commodity game engine.
Related Work

2.1. 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 [GGSC98]. 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 ([Mei96]), 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 [HJO+01] and screen space stroke generation [Lit97]. We refer the reader to Hedge, Gatzidis, and Tian’s review article [HGT13] for a more exhaustive study of existing painterly rendering methods.

In our work, we wish to place the artists at the core of the stylization process so that they can directly craft the desired painterly look. Offline methods such as WYSIWYG NPR ([KMM+02]), Deep Canvas ([KL03]), and OverCoat ([SSGS11]) 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 [BBS+13] while maintaining motion and temporal coherence, which are challenging but critical qualities in non-photorealistic rendering [BBT11]. Although ideal for character stylization, OverCoat’s rendering method as well as the more elaborate one published in [BSS+11], 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.

2.2. 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 [Pla99], and was first computed in real-time in Jet Set Radio [Kik00], thus
2. Related Work

pioneering the technique now commonly referred to as \textit{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 [MKG+97], the authors propose a real-time method for line-based stylization. More specific styles can be achieved in real-time, such as hatching ([PHWF01]), line art rendering ([Elb99]) or charcoal drawing ([IMG02]). 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.

2.3. 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 [KK89] that inspired researchers to use the space surrounding a mesh, commonly referred to as \textit{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 [Ney98]. Further works making use of shell space include [CTW+04], 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 [PBFJ05].

These methods target offline rendering through ray-tracing or geometry generation. Meyer and Neyret [MN98] 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 [Len00] and over arbitrary surfaces [LPFH01]. 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, [MPN+02] capture *opacity hulls*, which are rendered by projecting surfels onto the screen. Further work presented in [VPM+03] 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 [ODAO15] 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.

### 2.4. "Kimono" Approximation

Clearly most related to our venture is the work of Manuel Braunschweiler in [Bra]. In his bachelor thesis, Manuel Braunschweiler approaches the challenge of depicting 3D paintings with a multi-layered, textured mesh. Because of its nested structure, the author assigned the term "Kimono" to the approximation method. The challenge to craft the textures for the mesh is approached in a strictly mathematical way by setting up a non-linear system of equations. Several sample images of the targeted scene are captured from different view perspectives and used to derive per-pixel based equations as visualized by Figure 2.1. The system of equations is then solved in a least squares sense.

Although mathematically elegant, this approach faces some serious drawbacks. First, solving such a system with millions of equations requires a long offline computation time of several hours. Second, the approximation process often runs into local minima and is not trivial to set up since there are several decisions to be made by the user. In terms of quality, the method does not preserve the inherent structure of the paint strokes. Instead, the resulting textures appear blurred and various artifacts show up. The most disruptive issue is that the object’s silhouettes reveal the underlying layered structure.

Likewise, we support multiple layers in our approach; however, they are not the pillar of our method. In fact, we show in section 4.3 that good results can be achieved using just one single layer. Furthermore, we use "fin textures" together with per-fragment fading in and out to hide the layers at the silhouettes. The most noticeable difference is how we solve for the textures. We apply various transformations to the splats of the paint strokes in order to be able to directly render parts of the texture, which helps us in preserving the intended structure of the paint strokes. In section 4.3, we compare the results achieved by Kimono and by our method, and show an increase in quality and reliability as well as a reduction in offline pre-computation time.
2. Related Work

Figure 2.1: Creating an equation for each pixel of the sample image by intersecting the viewing ray through the center of the pixel with the layers of the Kimono mesh. [Bra]
Method

3.1. Overview

Our method takes as input a 3D painting with a low polygon mesh as well as an offline non-photorealistic renderer for such paintings. For this master thesis we used the OverCoat method described in [SSGS11] 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 3.1.

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 3.1: The stroke rendering model in OverCoat (figure courtesy of Schmid and colleagues [SSGS11])](image)

3.2. 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 [KK89], or using shell textures as can be seen in [MN98]. 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.
3. Method

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 3.2 shows this process on a toy example, while Figure 3.3 shows a triangle mesh and the generated fin mesh using this method.

![Example fins extruded from mesh edges](image1)

**Figure 3.2:** Example fins extruded from mesh edges

(a) Cat proxy geometry  (b) Fin mesh

**Figure 3.3.**

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 3.4, 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 3.3, a final optimization process is applied to the mesh that discards the geometry that is not necessary due to completely empty or transparent textures.

![Offset mesh](image1)

![Offset mesh with 3D painting](image2)

**Figure 3.4:** 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.

### 3.3. Capturing Fin Textures

The fin rendering, as explained in section 3.4, consists of depicting the original 3D painting using a textured proxy geometry mesh as well as textured fin quads that are faded in as their normals 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 [MN98], 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 as depicted in Figure 3.1. 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 polygon. Directly using that rendering method would mean that the stroke is sampled differently when captured with unequal view perspectives onto adjacent proxy geometry polygons, which inevitably leads to
3. Method

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, especially at polygon borders.

3.3.1. 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 3.5. In the concave
case represented by Figure 3.6, we end up rendering splats multiple times in different places. 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 3.5: Convex case: Capturing textures for the red and blue triangles using orthographic cameras ignores the green volume.](image)

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 whose 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
3.3. Capturing Fin Textures

Figure 3.6: Concave case: Splats located in the green volume are considered twice, when capturing textures. Once for the red and once for blue triangle and do therefore get duplicated.

defined by $t$. Figure 3.7 shows the difference of rotating a splat in its correct position rather than projecting it onto the plane defined by $t$. 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.

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 3.8. 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 [SSGS11].

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 Figure 3.9 (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 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 3.9 (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 and was comfortable to implement.
3. Method

Figure 3.7.: Simply projecting the splats on the target plane as in (a) leads to discontinuities in the textures. Using rotations solves that problem as shown in (b).

Figure 3.8.: The two cameras assign the splats an inverted depth order when relying on the camera space z-position.
3.3. Capturing Fin Textures

(a) Incorrect orientation  
(b) Fixed orientation

Figure 3.9: 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.

3.3.2. 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 cover the complete volume spanned by the proxy mesh triangle and its connected fins. Similar to the ignored volume depicted in Figure 3.5, 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 3.10 shows how this method helps remove discontinuities between neighboring fin textures.
3. Method

(a) No interpolation  (b) Smooth transition due to interpolation

Figure 3.10.: Interpolating the view direction helps to reduce hard changes in the fin textures.

3.4. 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, fin textures should only be visible when their normal is close to orthogonal with the camera plane. Existing methods such as [LPFH01] 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.

3.4.1. 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 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.

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.
3.4. Fin Texture Rendering

![Figure 3.11:](image)

**Figure 3.11:** 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.

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 3.11. 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.
3. Method
Results

All the results shown in this master thesis are rendered in Unity [Uni15] using our custom Unity surface shader for blending polygons in and out, as well as our back-to-front polygon sorting implementation. Besides the rendered images in section 4.3.2. The supplemental video, created during the thesis, shows side by side comparisons of OverCoat paintings with the output of our method, as well as live game sequences, that prove 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.

Prior to showing the results in section 4.3, we present the obtained frame rate for our rendered scenes in section 4.1 and explain the offline approximation running duration in section 4.2.

4.1. Frame Rate

Table 4.1 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 speedup of one to two orders of magnitude compared to the original OverCoat scenes, making them suitable for real-time applications such as games.
4. Results

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

Table 4.1.: 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. Images of all the listed scenes are shown in section 4.3.2.
4.2. 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 speedup which allows regular scenes, like the ones referred to in table 4.1, 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 to achieve good results in terms of quality.

4.3. Rendering Quality

In this section we first review the results achieved by our method. We then compare our results in section 4.3.3 with the results achieved by the method described in [Bra]. More results by our method can be viewed in the appendix in section A.2.

4.3.1. Results Review

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 3.4.

Contrary to existing shell texture methods such as [LPFH01], 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 by our algorithm 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 presented in Figure 4.1, 4.2 and 4.3, respectively.
4. Results

Figure 4.1: Bee scene

(a) OverCoat rendering  
(b) Approximation rendering

Figure 4.2: Magican & Genie scene

(a) OverCoat rendering  
(b) Approximation rendering
4.3. Rendering Quality

Figure 4.3: Dog scene

(a) OverCoat rendering

(b) Approximation rendering

Figure 4.4: Cat scene (backside)

(a) OverCoat rendering

(b) Approximation rendering
4. Results

![Cat scene (front)](image)

(a) OverCoat rendering  
(b) Approximation rendering

**Figure 4.5:** Cat scene (front)

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 subsection 3.3, 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 4.6.

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 subsection 3.3, and we capture strips of fins using a single camera view, to avoid discontinuities as mentioned in subsection 3.3. Figure 4.7 shows painted grass on a flat plane mesh. If the plane tessellation 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 4.6. 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 subsection 3.4 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 mentioned video 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.
4.3. Rendering Quality

Figure 4.6.: The blue paint stroke is captured by all three surrounding fins.

Figure 4.7.: The OverCoat scene (a) of painted grass is approximated using a single subdivided plane as input mesh. While in (b) the grass is still connected with 15 layers, it clearly splits up with five layers in (c), revealing the underlying structure.
4. Results

4.3.2. Game Scenes

In this section we present three game scenes created by Flurin Jenal, a ZHdK game designer, using our new method. The first game scene, appearing in Figure 4.8, contains a bouncing panda in front of a hand-drawn mountain landscape. The second game scene, depicted in Figure 4.9, lets the player take control over the "Overdog", the goal being to always walk straight, while deforming its body parts in rhythm with the music. A devastating UFO attack is simulated in the last scene illustrated in Figure 4.10.

![Game scene "Jumping panda"

Figure 4.8: Game scene "Jumping panda"
4.3. Rendering Quality

Figure 4.9: Game scene "Overdog"
4. Results

Figure 4.10: Game scene "Attacking UFO"
4.3. Rendering Quality

4.3.3. Comparison with Kimono Approximation

Compared to the results produced by the method described in [Bra], our method achieves an overall better visual quality. This can be directly seen in Figure 4.11, where we compare the two methods on the example of the "Dog" scene. The close look at the different textures in Figure 4.12 shows clearly that our method preserves the characteristics of the paint strokes better and provides sharper results. In Figure 4.13, we show that with our method we eliminate the most striking issue, namely the visible layer structure, that appears prominently in the Kimono approximations. Another advantage of our method is that we allow different sizes for the covering hulls of parts of our approximations. As can be seen on the dog’s tail in Figure 4.11, our method is able to represent the strokes of the tail that go far off from the proxy geometry’s surface.

![Figure 4.11.](image)

(a) OverCoat rendering  (b) Kimono approximation  (c) Our method

*Figure 4.11.: The comparison of the "Dog" scene shows that in the old Kimono approximation the textures appear blurred, while for our new method the characteristics of the paint strokes are still preserved.*

Additionally our method does not suffer from blurred triangles due to small textures dimensions. Increasing the texture size for the Kimono method equivalently increases the number of unknowns and leads therefore to drastically longer offline computation times. In our method, the output texture dimensions barely contribute to the pre-computation time. Furthermore our offline approximation time is in general much lower. For the "Element Cube" scene shown in Figure 4.13, the Kimono algorithm had to run about 16 hours. Our approximation finished in a few minutes.

In many cases the Kimono approximation runs into local minima, providing unsatisfying results. Even after several hours of calculation time, the textures of the "cat" scene are still in a bad condition as shown in Figure 4.14. Our method acts usually as expected and provides reasonable results.
4. Results

Figure 4.12.: A closer look at the textures highlights the differences in quality and stroke characteristics preservation. Since the Kimono approximation solves for each texture pixel separately, single pixels can completely differ from their surroundings.

Figure 4.13.: In the "Element Cube" scene the layers can become disturbingly visible in the old approximation method, while the new method provides satisfying results.
4.3. Rendering Quality

![Figure 4.14: In many cases the Kimono approximation runs into local minima, providing unsatisfying results. As the above pictures implies, even after several hours of calculation time, the textures of the "cat" scene are still in a bad condition. There is no other choice then to restart the approximation process over again and hope for better results.](image)
4. Results
Limitations and Future Work

Our results reveal the limitations of our method and indicate several possible directions for future work.

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.

Parts of 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 concave 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.

Taking this paradigm to the next level, one could try to circumvent the dependency of the results on adequate input geometry by designing an algorithm that crafts a completely new geometry depending on the positions of the scene’s paint strokes. This adapted geometry could possibly achieve more accurate results and could alleviate the need for the artist to construct the precise geometry beforehand. One could still allow to modify the generated geometry manually, so that the artist keeps direct control and influence via the input mesh.

To the best of our knowledge, no real-time lighting solution for 3D paintings such as those presented in [SSGS11] 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.
5. Limitations and Future Work
Conclusion

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-fragment interpolated 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.
6. Conclusion
Appendix

A.1. Workflow Documentation

This section describes how to use the integrated interface in OverCoat to generate an approximation from a painted scene. Moreover, it explains how to correctly import the approximation in Unity and how to apply possible modifications to alter its appearance.

A.1.1. OverCoat Interface

To give full control to the artist over the approximation process, the interface has been directly integrated in OverCoat. As mentioned in section 3, the first step is to tell OverCoat which paint strokes should be captured to be part of the crafted approximation. This can be done by selecting proxy geometries and scaling their layer level so that the spanned volume tightly encloses the paint strokes. The algorithm allows several proxy geometries to be selected with different scales. Figure A.1 illustrates different possibilities to approximate the “Genie” scene. Note that even though it is visible on the screen, the shown geometry will not be visible in the final approximation, unless explicitly stated in the advanced options as shown in A.1.3. Indeed, only the visible scene geometry is considered by the approximation method for generating the output mesh. This concept generalizes from geometry to paint strokes. Only the strokes that are visible on the screen appear in the final approximation. This can be useful for switching off parts of the scene and creating specific approximations as shown in Figure A.1 (b).

Figure A.1.: Different scales for the genie’s body and smoke are used in (a), while in (b) only the smoke of the genie is approximated.
A. Appendix

A.1.2. Approximation Interface

Once a scene is ready to be approximated, the approximation interface, shown in Fig. A.2), can be found under Tools -> Show Approximation Window or alternatively by pressing Ctrl + A. In general, the advanced options do not need to be changed and are only intended for experienced users or for very specific purposes. Detailed information about each setting is provided below:

**Select Folder**
Provides the directory where the files will be created.

**Output file name**
Specifies the base name of the created files.

**Number of layers**
Defines how many mesh layers are created.

**Output texture size**
Defines the pixel-dimensions of the created textures. Possible options are 512, 1024, 2048 and 4096 pixels. Note that 2048 is the recommended maximum since the current Unity version does not seem to use the 4096 sized texture correctly. In our tests, it appeared that Unity downscales the texture which results in the same artifacts as described in A.1.4 with activated mip maps.

**Texture quality**
Defines the size of the images taken by the algorithm when rendering the scene. Smaller values reduce the time needed for the algorithm to terminate but can reduce the quality of the final output textures.

**Fin add top/bottom space**
With these parameters, additional space can be given to the fins at the top and bottom.

A.1.3. Advanced Options

**Interpolate view direction**
This parameter allows the user to specify that the algorithm should perform additional interpolation steps as described in section 3.3 and as appears in Figure 3.10. The parameter value represents the maximum non-interpolated change in the angle of the viewing direction from a fin to its neighbors.

**Show strokes as world space sampled**
Ensures that all strokes are world-space-sampled to avoid inconsistent splat positions.

**Project splats on layers**
This option tells the algorithm to project all splats on the closest layer to avoid missing or duplicated splats.
**A.1. Workflow Documentation**

![Figure A.2: Approximation interface integrated in Overcoat](image)

**Uniform splat orientation**
Lets the algorithm try to compute a uniform orientation for the splats across the geometry. This option is only noticeable if the scene contains splats which are orientation dependent.

**Horizontal fin view direction**
When activated, the view direction is manipulated to be parallel to the base triangle, when rendering images for the fins.

**Speed up with bounding boxes**
The algorithm computes bounding boxes for the strokes, which are then used to easily skip splats that can not contribute to the rendered volume. This option drastically decreases the overall approximation time in general.

**Only keep fins of quad edges**
The algorithm tries to quadrangulate the given input meshes. For each found quad, the inner edge is then excluded from the final output mesh.
A. Appendix

Keep to combine with next approximation
This option tells the algorithm to keep the data from the current approximation and to add it when creating the next approximation. This way, multiple scenes can be loaded and approximated independently but are still combined in the output textures and mesh.

A.1.4. Unity Import

To import the approximation in Unity, one simply adds the object file as well as the two textures to Unity. To ensure correct appearance, the settings of the textures need to be modified. The texture type should be changed manually from Textured to Advanced. This allows to see and modify more options concerning the texture in Unity. The flag "Generate Mip Maps" should be switched to the unchecked status, since the automatic mip map generation leads to problems with the semitransparent textures. From the way our layer and fin textures are structured, artifacts appear especially at the in general transparent fin ends, since the transparent pixels get merged with non-transparent pixels from the fin located above in the texture. Figure A.3 shows the slightly visual artifacts that arise when mip maps are switched on.

![Figure A.3: Different Unity texture settings illustrated on the blue cat. Artifacts appear at the fins’ end in (a). No such artifacts are visible in (b).](image)

A second necessary change on the texture options is to choose the correct color format. Leaving it at "Compressed" leads to incorrect colors especially in semi-transparent regions, which again is particularly visible on the fins. Good results are achieved when changing the color format to "Automatic Truecolor" as can be seen in Figure A.4.

For the imported model geometry, the settings don’t need to be changed. However, it is worth mentioning, that the setting "Normals" should be on "Import", since our geometry exhibits a special structure because of the fin mesh, and therefore the normal calculation method of Unity fails to deliver the correct normals.

After correcting the texture settings, two more steps need to be done. Every Unity game object that originates from an OverCoat approximation requires a specialized shader as well as a back-to-front triangle sorting script. The per-frame back-to-front sorting script needs to be added to the game object on the same hierarchy as the MeshFilter, MeshRenderer or SkinnedMeshRenderer to be able to access the mesh data. Note, that for static game objects
A.1. Workflow Documentation

that can only be seen from a certain view direction, the back-to-front sorting script can be run only once and then be removed or deactivated to save performance. In Unity’s inspector, the script provides two modifiable parameters. The first one is the number of buckets to use for the bucket sort. The default value, as seen in Figure A.5 (a), is 1000, which generally is sufficient. However, it should be increased if flickering or wrongly sorted triangles appear. The second parameter specifies if the variable vertex positions of an animated character should be calculated by the GPU or the CPU. On the one hand, having the work done by the CPU instead of the GPU increases the calculation time of the script. On the other hand, retrieving the vertex positions of a skinned mesh from the GPU using the Unity API is faster, but allocates a new vertex array every frame. This creates a considerable amount of garbage from which performance spikes arise, that can be perceived as periodical lags, when the garbage collector becomes active.

Unlike the KimonoRenderer, the KimonoShader script offers several tunable parameters as Figure A.5 (b) suggests. In the first two texture slots, the model’s layer and fin texture should be referenced. The sliders for the layers and fins visibility allow to assign each of them a separate opacity.

Modifications

Additional functionality can be easily integrated into the shader script. For example, an artist might want to apply a different color to the fin textures to intensify their appearance or to have them look similar to a Toon shader. Figure A.6 shows the cat scene rendered with such a modified shader.

An alternative way to alter the appearance of an approximation is to directly modify the layer and fin textures with an external tool such as Photoshop. Figure A.7 shows the UFO scene once with the original textures and once with adjusted colors. While this may seem straightforward, care has to be taken because of the special pre-multiplied image format of the textures. For example in Photoshop, when simply deleting parts of the texture to make them appear completely transparent, the deleted parts will surprisingly appear white in Unity. The problem is that the saved color is a transparent white and not a transparent black, which is needed for the
A. Appendix

Figure A.5.: The two Unity scripts that are necessary for the approximations to look correct. The *KimonoRenderer* in (a) ensures a back-to-front sorting of the mesh triangles on every frame. The *KimonoShader* controls the fading in and out of the fins.

Figure A.6.: As a possible example on how to extend the current shader, this cat scene uses an extended shader with additional color option for the fins.
pre-multiplied alpha blending to work correctly.

![Figure A.7: Modifying textures](image)

(a) Original texture  (b) Modified with Photoshop

**Figure A.7.:** Modifying textures

### A.1.5. Approximation Viewer

Importing an approximation into Unity does not take much time. However, a simple viewer application, that was programmed during the thesis, allows to even faster visualize the approximations. Besides the geometry file and the two textures, the approximation algorithm outputs a text file that can be used by the viewer to directly review the approximation. Figure A.8 shows the viewer application. Additionally, the viewer offers the possibility to consider only a single shell of the approximation.
A. Appendix

Figure A.8: KimonoViewer

A.2. Additional Results

In this section we present further results that we did not show in section 4. Figure A.9 shows a depiction of Van Gogh and allows to directly compare the approximation with the OverCoat scene. In figure A.10, the mouse of the original OverCoat scene is added to the approximation. Additionally, shadow projectors are added to the scene to enhance the spatial impression. In figure A.11, the panda, that is used in the "Jumping Panda" game scene, can be compared to its original OverCoat painting from the side. Last, various approximations of different scenes are shown in the Figure A.12.
A.2. Additional Results

(a) OverCoat rendering  
(b) Approximation rendering

*Figure A.9.*: Van Gogh scene

*Figure A.10.*: Cat and Mouse scene - Both images show an approximation of the original scene with additional shadow projectors.
A. Appendix

Figure A.11.: Panda scene

Figure A.12.: Various approximated scenes
Bibliography


Bibliography


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

**Title of work** (in block letters):

Real-time rendering of proxy based 3D paintings using fin textures

**Authored by** (in block letters):

*For papers written by groups the names of all authors are required.*

**Name(s):**

Imhof

**First name(s):**

Nicolas

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

**Place, date**

25.07.2015

**Signature(s)**

Imhof

*For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.*