Real-time stylized rendering of 3D animated scenes

Alexandre Bleron

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Alexandre BLERON

Thèse dirigée par **Joelle THOLLOT**, codirigée par **Thomas HURTUT** et co-encadrée par **Romain VERGNE**,

préparée au sein du **Laboratoire Jean Kuntzmann** dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Rendu stylisé de scènes 3D animées temps-réel.

Real-time stylized rendering of 3D animated scenes.

Thèse soutenue publiquement le **8 novembre 2018**,
devant le jury composé de :

Mme Marie-Paule Cani
Président du jury, Professeur
Grenoble INP
M. Pascal Barla
Rapporteur du jury, Chargé de Recherche
Inria Centre Bordeaux – Sud-Ouest
M. Daniel Sýkora
Rapporteur du jury, Professeur Associé
Univ. Technique de Prague – République Tchèque
M. Cyrille Damez
Membre du jury, Ingénieur
Allegorithmic SAS – Clermont-Ferrand
M. Thomas Hurtut
Co-directeur, Professeur
Ecole Polytechnique de Montréal – Canada
Mme Joëlle Thollot
Directrice, Professeur
Grenoble INP

Abstract

The goal of stylized rendering is to render 3D scenes in the visual style intended by an artist. This often entails reproducing, with some degree of automation, the visual features typically found in 2D illustrations that constitute the "style" of an artist. Examples of these features include the depiction of light and shade, the representation of the contours of objects, or the strokes on a canvas that make a painting. This field is relevant today in domains such as computer-generated animation or video games, where studios seek to differentiate themselves with styles that deviate from photo-realism. In this thesis, we explore stylization techniques that can be easily inserted into existing real-time rendering pipelines, and propose two novel techniques in this domain.

Our first contribution is a workflow that aims to facilitate the design of complex stylized shading models for 3D objects. Designing a stylized shading model that follows artistic constraints and stays consistent under a variety of lighting conditions and viewpoints is a difficult and time-consuming process. Specialized shading models intended for stylization exist but are still limited in the range of appearances and behaviors they can reproduce. We propose a way to build and experiment with complex shading models by combining several simple shading behaviors using a layered approach, which allows a more intuitive and efficient exploration of the design space of shading models.

In our second contribution, we present a pipeline to render 3D scenes in painterly styles, simulating the appearance of brush strokes, using a combination of procedural noise and local image filtering in screen-space. Image filtering techniques can achieve a wide range of stylized effects on 2D pictures and video: our goal is to use those existing filtering techniques to stylize 3D scenes, in a way that is coherent with the underlying animation or camera movement. This is not a trivial process, as naive approaches to filtering in screen-space can introduce visual inconsistencies around the silhouette of objects. The proposed method ensures motion coherence by guiding filters with information from *G-buffers*, and ensures a coherent stylization of silhouettes in a generic way.

Résumé

Le but du rendu stylisé est de produire un rendu d'une scène 3D dans le style visuel particulier voulu par un artiste. Cela nécessite de reproduire automatiquement sur ordinateur certaines caractéristiques d'illustrations traditionnelles: par exemple, la façon dont un artiste représente les ombres et la lumière, les contours des objets, ou bien les coups de pinceau qui ont servi à créer une peinture. Les problématiques du rendu stylisé sont pertinentes dans des domaines comme la réalisation de films d'animation 3D ou le jeu vidéo, où les studios cherchent de plus en plus à se démarquer par des styles visuels originaux. Dans cette thèse, nous explorons des techniques de stylisation qui peuvent s'intégrer dans des pipelines de rendu temps-réel existants, et nous proposons deux contributions.

La première est un outil de création de modèles d'illumination stylisés pour des objets 3D. La conception de ces modèles est complexe et coûteuse en temps, car ils doivent produire un résultat cohérent sous une multitude d'angles de vue et d'éclairages. Nous proposons une méthode qui facilite la création de modèles d'illumination pour le rendu stylisé, en les décomposant en sous-modèles plus simples à manipuler.

Notre seconde contribution est un pipeline de rendu de scènes 3D dans un style peinture, qui utilise une combinaison de bruits procéduraux 3D et de filtrage en espace écran. Des techniques de filtrage d'image ont déjà été proposées pour styliser des images ou des vidéos: le but de ce travail est d'utiliser ces filtres pour styliser des scènes 3D tout en gardant la cohérence du mouvement. Cependant, directement appliquer un filtre en espace écran produit des défauts visuels au niveau des silhouettes des objets. Nous proposons une méthode qui permet d'assurer la cohérence du mouvement, en guidant les filtres d'images avec des informations sur la géométrie extraites de *G-buffers*, et qui élimine les défauts aux silhouettes.

Remerciements

Je voudrais d'abord remercier les membres de mon jury de soutenance : Pascal Barla et Daniel Sýkora, pour avoir accepté de rapporter cette thèse et pour leurs précieux retours; Cyrille Damez pour avoir accepté de participer à ma soutenance, et Marie-Paule Cani pour avoir accepté de présider ce jury.

Ensuite, je souhaiterais dire un grand merci à mes encadrants pour leur support pendant ces trois années, sans lequel je ne serais probablement pas allé bien loin: Joëlle, pour ton optimisme sans faille (et qui contrebalançait assez bien mon propre pessimisme), et pour m'avoir remotivé quand tout semblait bloqué ; Romain, pour ton aide précieuse sur un nombre incalculable de problèmes techniques dont je ne voyais pas le bout, et aussi pour être tout le temps disponible malgré ton implication dans à peu près la moitié des stages et thèses de l'équipe; et Thomas, pour m'avoir soutenu malgré la distance, pour avoir apporté un troisième point de vue, et pour arriver à comprendre les croquis sur papier lors de nos réunions par téléphone. Vous trois formez une équipe imbattable !

L'équipe Maverick n'est pas en reste : j'y ai passé trois excellentes années. Au risque de sonner faux, c'est vraiment une équipe pas comme les autres. Particulièrement, un grand merci aux doctorants, post-docs, stagiaires, et barons du troll que j'ai pu croiser pendant ce temps : Hugo, Grand Benoît, Petit Benoît, Léo, Guillaume, Beibei, Hugo, Ronak, Vincent, Jérémy, et Alban, et ceux que j'oublie car j'écris ça un peu vite.

Et enfin, merci à ma famille et mes parents pour m'avoir constamment soutenu.

Contents

Chapter 1

Introduction

1.1 Introduction to stylized rendering

Stylized rendering has also been historically called "non-photorealistic rendering" (NPR). This term naturally draws a comparison to its counterpart, photorealistic rendering: where photorealism strives to simulate as accurately as possible the behavior of light in a scene according to physical rules, stylized rendering instead tries to follow the depiction rules of illustrators, which rely on abstraction, simplification, or emphasis of particular features of a scene. Artists being human, these rules are arguably much harder to formalize.

In short, the goal of stylized rendering is to propose methods to produce images and animations, in a computer-assisted way, that exhibit "stylized" features. Typically, stylized features include the visual characteristics of hand-drawn 2D illustration, such as enhanced contour lines, or brush strokes. But more generally, this can be any visual feature that is not the result of a physical simulation of light propagation, but that instead results from an artistic intention.

In this thesis, we focus in particular on the *depiction of 3D scenes*, which is the production of a 2D picture that *represents* a 3D scene, in a chosen artistic style. Notably, this focus excludes most forms of abstract art, whereby the goal is not necessarily to depict a "real" 3D scene, but instead to express abstract visuals or concepts. It is also distinct from purely computer-generated art, or algorithmic art which is generated by an algorithm without the underlying goal of reproducing a 3D scene.

Stylized rendering methods provide some degree of automation that alleviates the work of an artist: currently, most stylized animations are made by painting every frame by hand, which requires large amounts of artist time (Figure 1.1). Automatic stylization can alleviate this process and facilitate the work of artists, allowing them to focus on the exploration of different styles: given some description of an artistic style, stylized animations could instead be produced with standard 3D animation techniques on the underlying scene, i.e. moving objects around in the scene, or moving the camera viewpoint.

The field of stylized rendering is still relevant today: while the increase

in graphics processing power fostered the developement of efficient photorealistic rendering techniques, it did not have the same impact on non-photorealistic techniques, and there is still a growing demand for looks that deviate from pure photorealism and the historical look of 3D computer graphics. This is especially true in "artistic" domains such as in animation or video games, as studios seek novel and distinctive styles to differentiate themselves from competitors. For video games, automatic stylization is even more important because of the unpredictability of some parameters: notably, the camera viewpoint and the lighting of the scene. Players can control these in a way that the artist cannot predict at design time, and the stylization technique must remain coherent under every configuration. Additionally, stylized rendering is often used in other interactive contexts, such as scientific visualization, architectural visualization, historical reconstruction, or cartography. While some of these applications are arguably less "artistic", they nevertheless share many of the same challenges: abstraction, emphasis, etc.

Automatic depiction of 3D scenes in a particular artistic style is a long-standing challenge in the computer graphics community. It differs from photorealistic rendering in the depiction of several features:

Contour lines: In traditional illustration, the contours of objects are often drawn explicitly as they convey important clues about the shape. This corresponds to the NPR subdomain of *line-based rendering*.

Light and shade: Illustrators depict illumination in a scene in ways that deviate significantly from photorealism. Examples of this include the placement and shape of highlights on an object. Again, this is for clarity and expressivity purposes. The *stylized shading* subdomain of NPR strives to model the rules for the depiction of light and shade in various contexts (paintings, technical illustration, etc.), and also to provide tools to design such shadings directly in 3D scenes.

Marks: Traditional paintings are made with marks in the picture plane (e.g. brush strokes) that do not correspond directly to a point on the surface of the depicted object. This contrasts with traditional computer graphics, where the fundamental unit of depiction is the pixel instead of the stroke: pixels map directly to a point on a surface by projection. Some techniques reproduce the appearance of paintings by directly emulating the painting process: i.e. explicitly placing and rendering discrete strokes on a canvas. This forms the domain of *stroke-based rendering*. However, this is not the only way: for instance, *appearance transfer* techniques can transfer painterly styles from an input exemplar image to a target image without explicitly manipulating discrete strokes.

There exists a great variety of artistic styles, and thus different rules and processes to depict contours and shading, and to place brush strokes, that vary from artist to artist. Traditional painting alone already represents a near-endless source of style examples, with its many different media, techniques, and tools, each with their own distinctive visual characteristics. Since more

FIGURE 1.1: Top: screen capture from *The Old Man and the Sea*, made by hand with approximately 29000 oil-painted frames. Bottom: screen capture from *Loving Vincent*, 65000 oil-painted frames, hand-painted by a team of 100 painters. Alleviating the work needed to produce such animations and allowing artists to explore new, more complex styles is one of the goals of stylized rendering.

recently, a greater variety of styles is attainable through *digital painting*, the use of computer tools that emulate painting on a digital canvas. This diversity makes it hard to formulate a common framework for stylized rendering. Instead, a lot of techniques focus on reproducing one style in particular. For example, specialized stylization techniques have been proposed to reproduce the appearance of oil paintings [Sem+16], or watercolor [Bou+06].

Another challenge lies in the fact that many artistic styles exhibit visual properties that are fundamentally related to the two-dimensional nature of the picture plane. This raises profound questions about how to translate and apply an artistic style to a 3D animation:

- First, how should the style look like when animated? For many styles, there are no animated references that we can compare the synthesized result against. Also, creating convincing animations of 3D scenes that

retain a 2D look is challenging: naive approaches to animate a style exhibit distracting visual artifacts, due to the well known problem of *temporal coherence* [BBT11] of stylized animations. We provide an overview of this problem in Section 1.3.

- Second, what user interface should we provide to control the appearance of the animated styles? What degree of control should we offer to the artist over the end result? We will see that there is usually a tradeoff between the degree of local control offered to the user and the applicability of the method in varying scenarios.

The goal of this thesis is twofold: first, to propose stylized rendering techniques that are easily and intuitively controllable by artists, and usable in interactive contexts. Second, we strive to explore new rendering techniques that can reproduce a wider range of styles, while still keeping an acceptable degree of temporal coherence.

In this regard, we make two contributions for stylizing 3D scenes in interactive contexts: a tool to design stylized shading models using intuitive primitives, and a novel screen-space rendering technique to reproduce a variety of styles using image filters with improved motion coherence.

Before going into more details about these contributions in Chapters 2 and 3, we first present general notions related to stylization of 3D scenes:

- In Section 1.2, we present the notion of style as proposed by Willats and Durand [WD05], and its different components. The vocabulary they introduced is useful to describe and classify stylization techniques in the rest of this thesis.

- In Section 1.3, an overview of the issue of *temporal coherence* of stylized animations is provided. This is useful to understand the design choices made when exploring stylization techniques.

- In Section 1.4, we explore the stylized rendering under the technical angle: i.e. how stylization is implemented in traditional real-time rendering pipelines. We describe ways to do stylization at different stages, and show that they correspond to different tradeoffs in the temporal coherence issue.

- Finally, in Section 1.5, we explore the stylization under the angle of user interaction: we list several interaction techniques, with concrete examples, and show that they do not provide the same degree of artistic control over the result.

1.2 Defining artistic style

Before diving into the technical details and challenges of rendering a 3D scene in a particular artistic style, it is necessary to clarify what is meant by "style".

Generally, the notion of artistic style encompasses many aspects. It can refer to the medium used: whether it's an oil painting, a watercolor, a digital painting, a sculpture, a 3D model, etc. It can also refer to painting techniques or the appearance of brush strokes on a picture: the thin, visible brush strokes of *impressionism* or the small dots of paint of *pointillism*. Style can also be recognized through the ways light and shade are depicted. Indeed, the treatment of lighting varies widely between different kinds of illustration: for instance, there is a clear difference between the smooth depiction of shade in renaissance paintings and the sharp contrasts between light and shadow in comic book illustration. Style is also recognizable in the shape and form of depicted objects: while some illustrators strive to reach optical realism, respecting proportions, perspective and foreshortening, others opt for more abstracted, deformed, or deconstructed shapes. This diversity of style is especially striking in comic-book illustration, which is also concerned with the depiction of movement.

At first glance, the multitude of aspects makes the notion of artistic style hard to formalize. Willats and Durand [WD05] organized these all these aspects in four *systems*, involved in the depiction of 3D scenes:

- The spatial system, which controls the mapping of 3D spatial properties to 2D properties. Typically, this represents the projection matrices used to map 3D coordinates to 2D.

- The primitive system, which "maps primitives in object space (points, lines, surfaces, volumes) to primitives in picture space (points, lines, regions)". For instance, *line-based* rendering must perform a mapping from the silhouette lines in 3D space to lines in the picture plane.

- The attribute system, which determines the rendering attributes (color, texture, width, etc.) of the picture primitives. Shading is an example of attribute system that assigns a color to primitives from a lighting environment.

- The mark system, which governs how the primitives are rendered according to their attributes. As Willats and Durand note, the marks used in traditional computer graphics are simply the pixels of the image. However, in stylized rendering, more complex marks systems can be used (e.g. *stroke-based rendering*, where primitives (points, lines, regions) are depicted by placing and rendering brush strokes on the picture plane.)

This separation into systems provides a useful framework for classifying computer depiction techniques: we occasionally refer to these terms when describing a stylization method.

The contributions we propose in this thesis are related to the attributes and mark systems. First, we propose a way to create stylized *shading*, which can be seen as an attribute system that defines the color of primitives according to parameters of the scene (the light sources). Our second contribution is a set of image processing techniques to render *stylization marks* on 3D scenes,

driven by various attributes (including shading): this falls in the category of mark systems.

1.3 Temporal coherence of animations

As we've seen, stylized depiction of 3D scenes involves both the motion of 3D objects and the motion of 2D features that live in the picture plane. There is a fundamental mismatch between those two spaces, which gives rise to the issue of temporal coherence. In this section, we explore this mismatch in more detail, and describe what is needed to ensure the temporal coherence of animations.

1.3.1 Primary and secondary space

In the context of traditional and computer depiction, a distinction is made between the *primary space*, which is the coordinate space where the scene "lives", and the *secondary space*, which is where the scene is drawn [WD05]. Primary space is the 3D cartesian space, and the secondary space is usually a 2D cartesian space that represents the canvas, physical or virtual.

Willats and Durand remarked that traditional computer graphics focus on the description of geometry in the primary space. Yet, they also argue that describing a style in secondary space (the 2D picture plane) allows for more flexibility in the specification of the depiction rules of traditional illustration, and generally leads to more intuitive user interfaces.

An illustration of this is the treatment of brush strokes: in contrast from photorealistic rendering, which uses individual pixels as the mark system, rendering techniques that strive to reproduce the look of 2D illustrations must reproduce the brush (or pencil, etc.) strokes used by the artist. These marks fundamentally "live" in the 2D picture plane: they are placed on the canvas during painting, and are not part of the artist's mental model of the 3D scene. This is why many stylized rendering techniques describe and render marks in the secondary space (also assimilated to *screen-space*, in the context of rendering), with attributes also defined in secondary space (for example the size and orientation of marks on the screen).

1.3.2 Fundamental goals of temporally coherent animations

Describing and rendering marks in the secondary space becomes more complex when animation comes into play. Indeed, when the viewpoint of the camera changes, or the objects in the scene move, the marks used to render the scene must update accordingly, and special care must be taken to ensure the *temporal coherence* of the resulting animation, as naive approaches often exhibit distracting artifacts: marks suddenly appearing and disappearing from frame to frame (*popping* artifacts), or that appear to slide on objects (*sliding* artifacts), creating parasitic *secondary motion*.

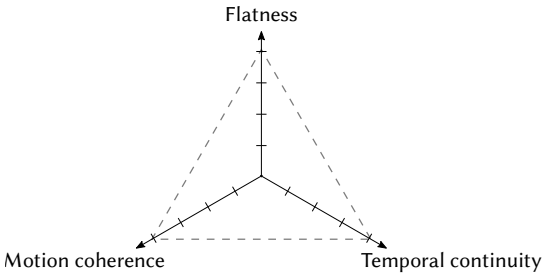


FIGURE 1.2: The three goals of temporally coherent animations: motion coherence (marks follow the motion of the underlying objects closely), temporal continuity (marks do not flicker from frame to frame), and flatness (the result has a 2D appearance, as if painted on a canvas). The tradeoff between those three properties is represented by Bénard, Bousseau, and Thollot [BBT11] as a triangle.

This is inherently a difficult problem because the motion of 2D marks is constrained to the picture plane, and is often at odds with the 3D motion of the underlying scene. Bénard, Bousseau, and Thollot [BBT11] have formalized the concept of *temporal coherence* of stylized animations in a set of three fundamental goals (Figure 1.2), and evaluated several stylization techniques against those goals:

Flatness: Flatness is what makes a stylized result look like a painting on a 2D canvas rather than a 3D texture-mapped object. Concretely, this corresponds to the capacity of a system to render stylization marks directly in the secondary space (the picture plane), independently of the geometry of the scene, that are not affected by perspective deformation and foreshortening, and maintain a constant density on the screen. It is a key component in reproducing the appearance of hand-drawn illustrations and animations.

Temporal continuity: The resulting animation should be fluid, and avoid *popping* or and *flickering* artifacts that occur when a mark used for rendering suddenly appears, disappears or flickers irregularly from frame to frame during the animation.

Motion coherence: Having the marks at fixed locations in the secondary space results in the so-called *shower-door effect* [Mei96]: the scene appears to be moving under a fixed screen, and breaks the illusion of looking at a dynamic painting. Because of that, stylization marks should maintain *motion coherence*: the motion of the marks should follow the motion of the underlying scene as much as possible.

As Bénard, Bousseau, and Thollot remarked, the mismatch between 2D and 3D cannot be fully reconciled: every solution is a tradeoff between the three goals (Figure 1.2). Ultimately, the fitness and artistic quality of a stylization algorithm is evaluated by comparing the result with an equivalent hand-drawn 2D animation, when there is one. In this regard, it is interesting

to note that hand-drawn animations themselves do not strictly follow temporal coherence rules. One example of this is temporal continuity: not only temporal continuity is hard to maintain by hand, but animators can also amplify temporal discontinuities for artistic purposes. Some stylized rendering techniques even provide ways to introduce controlled temporal discontinuities into the synthesized animation [Fiš+14].

This is also true, to some extent, for motion coherence: in highly controlled contexts such as hand-made animation, motion coherence can be managed manually by artists, and played with for various purposes. However, in interactive contexts where an artist has no control over each individual frame of the animation after rendering, it is crucial to have strong motion coherence, as uncontrolled secondary motion in the stylization is rarely desired. We tackle this issue in Chapter 3, where we propose a technique to render 3D scenes in a painterly style that ensures motion coherence.

1.4 Stylization in the real-time rendering pipeline

Stylization techniques make very different hypotheses on the nature and quantity of input data about the scene to depict. Some methods require a full 3D geometrical model of the scene, whereas others will only need information defined in the 2D image plane. They also differ by their target applications: in real-time contexts, such as video games or visualization, emphasis is put on performance, and the amount of data to be processed must be limited. Conversely, in the animation industry, the available time budget to render a frame is typically much greater, as they do not require interactivity for the spectator. Still, interactivity during the design process and the ability to quickly iterate over the result are highly valued features of stylization techniques, which reinforces the usefulness of real-time techniques.

This is why, in this thesis, we proposed stylization techniques that integrate into traditional pipelines for rendering 3D scenes in real-time, to facilitate their use by technical artists. In this section, we first present a brief overview of the standard 3D rendering pipeline, detailing the different pieces of data used (geometry, textures, etc.), and the related vocabulary that we later employ when describing stylized rendering techniques. Based on this pipeline, we will then see which degrees of freedom are available in the pipeline to do stylization.

After this section, the reader should have an overview of the process of rendering 3D scenes in real-time with the standard rendering pipeline, and how to model the appearance of objects in this context.

1.4.1 Overview of real-time rendering pipeline

In this section, we provide a brief overview of the traditional real-time rendering pipeline (summarized in Figure 1.3). Such a pipeline can be mapped directly to the hardware rendering pipeline implemented in Graphics Processing Units (GPUs). Some stages of the pipeline are programmable through

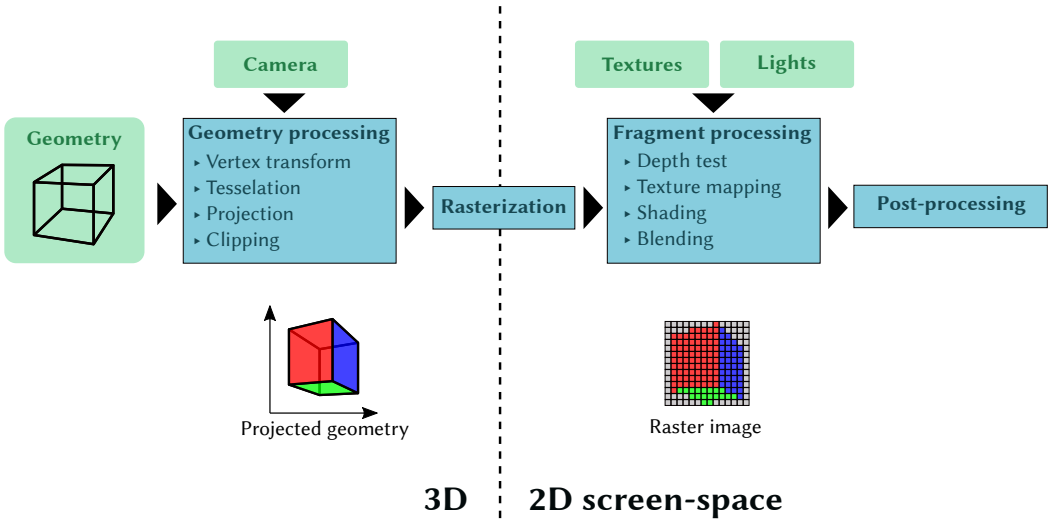


FIGURE 1.3: High-level overview of the rendering pipeline as typically implemented in hardware on a GPU. Rasterization marks the passage from 3D coordinates (object- and view-space) to 2D screen-space.

shaders: this can be used to customize the final appearance of objects on the screen.

The pipeline can be roughly divided into the following parts: first, the geometry processing stages, which transform the input geometry (typically a triangle mesh) to the view-space of the camera, and then to the 2D screen-space. This step is programmable through the vertex, tesselation and geometry shaders. Then, rasterization converts the geometry into a sequence of fragments, which are then handed to the fragment processing stage, which performs per-fragment calculations, such as shading or texture mapping. This stage is programmable through fragment shaders. Finally, the pipeline can also contain one or more post-processing stages, that perform screen-space operations on the final color image. Some examples of post-processing operations are color grading, tonemapping, contour detection and enhancement, among many others.

1.4.2 G-buffers and deferred rendering techniques

A variant of this pipeline, first proposed by Saito and Takahashi [ST90], consists in rendering the geometrical properties of the scene into screen-space images called G-buffers (see Figure 1.4). These images, containing normals, tangents and depth at every pixel on the screen, are then used for subsequent post-processing using only 2D image processing operations. Saito and Takahashi [ST90] first demonstrated this approach for extracting and rendering contour lines, and for shading.

Since then, their approach has spawned the field of *deferred rendering* techniques: The principle is to first render all data needed to compute the final color of a pixel into G-buffers, and perform the actual calculation as a post-processing pass on those G-buffers. A common instance of this is *deferred*

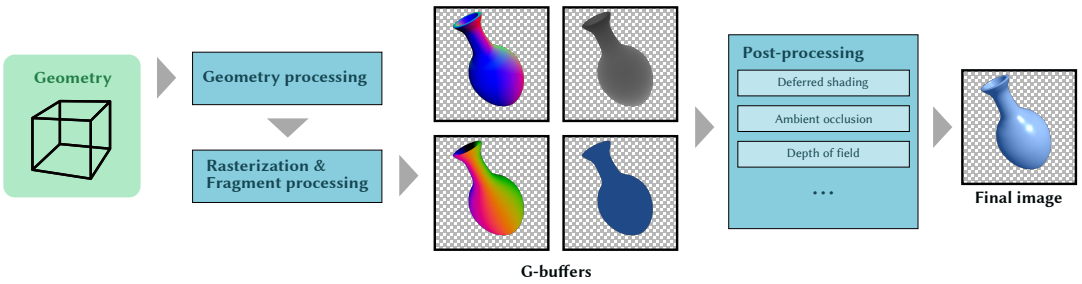


FIGURE 1.4: Overview of a deferred rendering pipeline. The scene is first rasterized into a set of G-buffers containing screen-space geometry information for visible surfaces (normals, tangents, depth, color, ...). Calculations to obtain the final result (shading, ambient occlusion, etc.) are done in a 2D post-process pass on those G-buffers.

shading, where the surface normals, diffuse color and material properties are first rendered into G-buffers, and the actual shading calculation is "deferred" to a post-processing pass.

Deferred rendering has several advantages:

- It reduces *overdraw*: the useless calculations done for each fragment that end up being discarded because they are occluded by another surface. With deferred shading, the calculations are only done once for every visible pixel on the screen, instead of once per fragment.
- Post-processing passes can employ image filters that access neighboring pixels, which is impossible with single-pass fragment processing.
- Image processing techniques used in post-process can still benefit from geometrical information inside G-buffers.

Applications of deferred rendering include contour extraction and enhancement, depth of field simulation, ambient occlusion, motion blur, etc. For a more detailed list of advantages and applications, we refer the reader to the overview of Thaler [Tha11]. In Section 3.2.3 we provide examples of deferred rendering techniques in the context of non-photorealistic rendering. Additionally, we make use of this technique in our two contributions.

1.4.3 Where to stylize?

In a real-time rendering pipeline, the visual appearance of an object is the interaction between several pieces of data at different stages of the pipeline. All of these can be manipulated to get a stylized result. We show that depending on the stage we act upon, this has a different impact on the temporal coherence of the result.

Geometry First, the geometry of the object itself can be stylized. This can actually be done by the artist before rendering: indeed, modelling a character with exaggerated, non-realistic shapes is already a form of stylization.

The animations associated to the geometry (movement and deformation of objects) also have a crucial role in the final look: it is a challenging task to author 3D animations that do not appear too synthetic and break the feeling of looking at a hand-drawn animation. However, authoring stylized shapes and animations is out of the scope of this work.

Some techniques have been proposed to alter the geometry dynamically during rendering: for instance, Rademacher [Rad99] blended between different deformations of a base model according to the camera viewpoint, in order to capture artistic distortions of 2D hand-drawn animations. Botkin [Bot09] used normal displacement to perturb the geometry of 3D models in a scene. By cycling this perturbation each frame, and blending the results across frames, they achieved an appearance reminiscent of layered, transparent brush strokes.

Since the geometry of the object itself is modified, the result stays perfectly coherent with respect to the motion of the camera or objects. However, simple geometry warping is usually not sufficient to reproduce 2D effects that have a constant size in screen-space.

Additional geometry anchored to the object can also be used as proxies to render stylization effects. One instance of this is *stroke-based rendering*, which consists in distributing flat proxy geometry that face the camera (billboards) on the surface of objects to simulate brush strokes. A review of stroke-based rendering techniques for 3D scenes is provided in Section 3.2.1.

Lighting The position, type and intensity of lights in a scene can be adjusted with an artistic intent. Some techniques ease this process by automatically adjusting lights according to user-provided constraints on the final appearance: an overview is provided in Section 2.2.1. Note that this is not specific to real-time pipelines, and also applies to offline renderers. It is also a form of stylization that is still usable within the constraints of physically-based rendering: indeed, it is very common for artists working with physically-based pipelines to "cheat" by placing lights in some view configurations to emphasize a particular element on the screen. However, this is harder to do in fully interactive contexts.

Textures Details on the surface of objects are usually stored in textures, and can be manually painted by artists. However, the main limitation of texture mapping for stylization is that all details appear flattened on the surface of the object and are affected by perspective deformation, reinforcing the 3D appearance of the result. Notably, brush strokes that live in the picture plane cannot be reproduced with standard texture mapping. Extensions to traditional texture mapping that provide better flatness have been proposed: these are reviewed in Section 3.2.2.

Shading model The shading model mediates the interaction between the incoming illumination (from the lights and secondary sources), the shape, and the surface properties (color, etc.). The combination of a shading model and its parameters (which can be spatially varying via texture mapping) is

usually called a *material*. In physically-based rendering, shading models are derived from Bidirectional Reflectance Distribution Functions (BRDFs), which describe the amount of light reflected according to incoming and outgoing light directions. In illustration, the rules for the depiction of light and shade deviate significantly from photorealism. As such, several non-photorealistic shading models have been proposed that are inspired by those rules: we review them in Section 2.2.2. Note that the shading model is a function defined on the surface of objects, in primary space: as such, the shading model alone cannot account for style marks in the picture plane. Following the classification of [WD05], the shading can be seen as an attribute system that provides the attributes of the marks to render. Stylization through shading models is the subject of our first contribution that we present in Chapter 2.

Post-processing After the scene is rendered, we get a 2D color image, onto which screen-space post-processing filters can be applied. This stage is well-suited for stylization, because screen-space directly corresponds to the secondary space, or 2D picture plane. Thus, it makes sense to perform stylization in screen-space. Furthermore, with deferred rendering, the color image is complemented by G-buffers which contain geometrical information. They make the link between object-space (3D) and screen-space (2D). This is why techniques that make use of G-buffers are sometimes called "2.5D" techniques. This offers great flexibility, but this comes at the cost of limited information about geometry, since occluded surfaces are lost. In Chapter 3 we explore the use of screen-space post-processing filters for stylization, and demonstrate how they can be guided with G-buffers to ensure motion coherence.

1.5 Artistic control of stylized rendering techniques

The challenges raised by automatic stylization are not limited to the technical aspects of rendering: depending on the application domain, stylization techniques must also provide some degree of *artistic control* (or *art direction*) and interactivity to be practically useful [Ise16]. This motivates the pursuit of stylization techniques that can render images at interactive rates, but also raises the question of how to interact with users.

Isenberg [Ise16] noted that there are different categories of target users within the field of non-photorealistic rendering, each having different expectations in terms of interactivity. On one hand, non-photorealistic rendering has been concerned with the needs of users with artistic skills, such as professional illustrators which are usually familiar with traditional or digital painting tools: they seek tools that alleviate the most tedious parts of their work, but that do not compromise the ability to do fine-grained modifications on the result. As such, for this target audience, methods providing low-level control are preferred. On the other hand, another goal of NPR is to provide stylization tools for non-artists, that do not require extensive artistic skills.

These tools naturally favor more high-level controls. In this thesis, we targeted users with technical and artistic knowledge, and sought to design tools that provide low-level control over the stylization of 3D scenes.

The type of control that can be offered to the artist also depends on the context: if the result to be displayed is a static view of a scene, then it is reasonable to allow the artist to make local edits directly on the rendered image. However, if the viewpoint is animated the edits have to be replicated for each frame of the animation, either manually or semi-automatically with a propagation method. If the viewpoint is unpredictable, such as in video games, then local edits in screen-space are outright impossible. The same is true for lights in a scene: in animated movies, artists can adjust lights to alter the appearance of objects locally. This is much more difficult to do in video games, if the lighting can be influenced dynamically by players. All of this puts constraints on the degree of control available to artists, and the type of interaction we can offer.

In the following paragraphs, we briefly describe several interaction paradigms that are found in non-photorealistic rendering techniques. We distinguish between *direct interaction with local control*, *global parameters*, *by-example approaches*, and *programmable approaches*. Individual techniques may use a combination of several of these paradigms to cover a wider range of the interaction spectrum.

Interaction through global parameters Stylization algorithms can have global parameters that control the behavior of the algorithm for the whole scene or the whole animation. They offer a very coarse-grained and high-level control over the stylization. For instance, procedural texturing techniques such as Perlin noise usually provide a fixed set of high-level parameters that control the look of the generated texture.

This kind of high-level control does not require artistic skills from users, provided that the parameters have an intuitive and predictable effect on the generated result. However, high-level control alone is not sufficient for artists and must be complemented with other inputs.

Interaction through exemplars Exemplar-based stylization (or *style transfer*) methods take the scene to stylize and a style exemplar as input, and sythesize a result with the same appearance as the input exemplar.

An example of this for 3D scenes is the lit-sphere technique of Sloan et al. [Slo+01], where the input exemplar is a shaded sphere painted by an artist. The synthesis is done by matching the normals of the exemplar sphere with the normals of the objects to stylize. Stroke-based rendering techniques also fall in this category, as users typically provide a small texture exemplar for the stamp used to draw the strokes.

Example-based methods make for very straightforward interactions with users, as they just have to provide an exemplar. However, the complexity of the input exemplars depends on the degrees of freedom in the result: i.e. if the lighting in the scene can change, then an example must be provided

for all possible lighting conditions. This can become impractical for highly dynamic scenes.

Another line of research for exemplar-based style transfer on 2D images was started by Hertzmann *et al.* with *Image Analogies*. In this framework, the input exemplar is a pair of images: a source unstylized image, and the corresponding stylized image. The image analogy algorithm learns the mapping between the two and can then transfer the stylization to another source image. It has since been extended to animations with temporal continuity [HJN03; Bén+13] and 3D renderings [Fiš+16]. More recently, Gatys, Ecker, and Bethge [GEB16] first demonstrated the use of deep neural networks to transfer style features from an exemplar to a target image. This technique has been the target of various improvements and extensions [Jin+17; SID17]. In those techniques, stylization is seen as an optimization problem: maximizing the visual similarity of the source exemplar and the target at some given scale. The main drawback is that the quality of the synthesized result is unpredictable, and there is no way to ensure that the optimization result matches the intent of the artist.

Direct interaction and local control Local control is the ability to make local edits in space (in the scene or the canvas) or in time (across the animation timeline). Local control is usually paired with direct interaction. Direct interaction is when the user interacts directly on a view of the scene to stylize, instead of indirectly through a set of adjustable parameters.

In the context of 3D scenes, local control can be done at various levels: at the parameter level, by locally adjusting parameters of the stylization algorithm in object-space (for instance, in the art-directed watercolor rendering system of Montesdeoca, Seah, and Rall [MSR16]), or at the primitive level, by directly embedding primitives in 3D space (as in *Overcoat* [Sch+11]).

One drawback is that, as more local control is provided, it can become difficult to propagate, generalize or animate the edits automatically in highly dynamic environments. However, the main advantage is that the artist has a fine-grained control over the end result and can fine-tune the appearance as precisely as needed.

User-defined rules and programs Another approach is to allow users to program their own rules in the system, with varying degrees of flexibility. This approach is widely used in practice through *shaders* in the graphics pipeline, which specify geometry transformations and shading, among other things. Several renderers and game engines propose specialized node-based editors to design shaders without having to write shader code [1] [2]. A programmable approach has also been proposed for line-based rendering by Grabli et al. [Gra+10], and implemented in the *Blender* modeling software

[1] Unreal Engine: Essential Material Concepts https://docs.unrealengine.com/en-US/Engine/Rendering/Materials/IntroductionToMaterials

[2] Unity: Shader Graph https://unity3d.com/shader-graph

as the *freestyle* rendering engine [3]. Additionally, Schmid et al. [Sch+10] proposed a programmable approach to render motion effects.

Programmable approaches can potentially provide very precise control, even in very dynamic scenarios. However, the main drawback is that some degree of technical knowledge is required from the users. Also, depending on how the program needs to be specified (e.g. in the form of code, or using a visual representation), these approaches might not be suitable for the quick exploration of different styles.

1.6 Goals of this thesis

The main motivation for this work is to render 3D scenes in new visual styles. Many stylization techniques focus on reproducing specific media or techniques: e.g. cartoon style, watercolor, oil painting, charcoal, stippling, hatching... In this work, we do not attempt to reproduce one particular style, but rather try to facilitate the exploration of the design space of 3D scene stylization, by proposing both design tools and new rendering techniques. In doing so, we want to support styles that exhibit 2D illustrative features: this is still a challenge and an active area of research, in part because of the issues linked to temporal coherence. A motivating long-term goal behind this research is to provide generic building blocks for stylization that can reproduce a variety of 2D illustrative styles on 3D scenes, with art-directed temporal coherence properties.

We more specifically target users with artistic skills. We aim to propose art creation tools that assist users rather than fully automatic techniques, and to put the artist "into the loop", as suggested by Winnemöller [Win13]. In this regard, another of our concerns is to provide techniques that can be easily plugged into existing artist workflows: for this reason, we favored post-processing techniques that require only minimal modifications to the standard real-time rendering pipeline.

1.6.1 Summary of contributions

The design space of 3D scene stylization is large: as we've seen previously, it is possible to add stylized features to the shape of the objects in the scene, the animation, the light and shadows in the scene, the materials, etc. In this work, we focus on two aspects: first, the depiction of stylized light and shade in 3D scenes through stylized *shading models*; second, the rendering of 3D scenes with stylization marks (brush strokes). Following the classification of Willats and Durand, this corresponds respectively to the stylization of the *attributes* of marks, and the rendering of the marks themselves.

[3] Freestyle introduction - Blender Manual https://docs.blender.org/manual/ko/dev/render/freestyle/introduction.html

Designing stylized shading models

Our first contribution is a workflow for designing stylized shading models on 3D objects. In computer graphics, designing a shading model that behaves in a desired way is challenging, because of the complex interactions between materials, geometry, and lighting environment. It is a time-consuming task, that often involves writing custom shader code. For photorealistic appearances, newer physically based shading techniques tend to provide an intuitive and coherent workflow for artists, but they are of limited use in the context of non-photorealistic shading styles. On the other hand, existing stylized shading techniques are either too specialized or require considerable hand-tuning of unintuitive parameters to give a satisfactory result.

In the proposed workflow, we organize the design process of individual shading effects in three independent stages: control of its global behavior on the object, addition of procedural details, and colorization. Inspired by the formulation of existing shading models, we expose different shading behaviors to the artist through *parametrizations*, which have a meaningful visual interpretation. Multiple shading effects can then be composited to obtain complex dynamic appearances. The proposed workflow is fully interactive, with real-time feedback, and allows the intuitive exploration of stylized shading effects, while keeping coherence under varying viewpoints and light configurations. Furthermore, it makes use of the *deferred shading* technique, making it easily integrable in existing rendering pipelines.

Rendering brush strokes on 3D objects with image filters

Our second contribution is more technical, and deals with the usage of image filtering techniques in the context of the stylization of 3D scenes. We recognize that a wide range of stylized effects can be achieved on 2D pictures and video using only image filtering techniques. We explore the use of image filters in the post-processing stage of 3D scenes, in combination with procedural texturing techniques, to reproduce the appearance of brush strokes on 3D objects. During this stage, having access to geometrical data in G-buffers, we show that it is possible to guide the image filters to generate strokes that are coherent with the motion of the scene. However, a naive approach has several issues regarding the coherence of the generated marks outside the silhouette of objects.

We describe a post-processing pipeline that solves these issues and allows the use of image filters that can alter the original footprint of objects in G-buffers. This pipeline is fully implemented on the GPU as a set of post-processing passes, and can be evaluated at interactive rates. We show how common image filtering techniques, when integrated in our pipeline and in combination with G-buffer data, can be used to reproduce a wide range of "digitally-painted" appearances, such as directed brush strokes with irregular silhouettes, while keeping a certain degree of motion coherence.

Chapter 2

A workflow for designing stylized shading effects

2.1 Introduction

When depicting objects, 2D illustrators use light and shade in ways that deviate significantly from realism and physical rules of light propagation. Instead, it is often used to emphasize a part of the scene that the artist deems more important, or to convey a particular mood or emotion. One of the goals of *stylized shading* techniques is to translate artistic rules for the depiction of light and shade into formalized models that can be applied automatically in the context of the rendering of animated 3D scenes on a computer. This is a difficult problem because, in general, stylized lighting effects do not reflect a physical truth and as such are not easily simulated. This difficulty is exacerbated by the fact that for some of these effects, the only 2D reference available is static: there is no consensus on how a stylized light-and-shade depiction should look like under movement of the object or of the viewpoint. They must be rendered and animated by taking into account the intention of the artist, and they are not easily unified under a single rendering framework.

In 3D computer graphics, light and shade is a part of *appearance design*, which is the process of adjusting materials, surface details, and lighting to achieve a desired look for an object in a scene. As with 2D illustration, it is possible to tweak shading and lighting in 3D scenes to achieve emphasis of a particular object, abstraction or to convey mood. However, shading design is a complex process involving multiple interconnected aspects: the appearance of an object in a 3D scene is the result of the interaction between the material, the object geometry, lighting environment, and camera viewpoint. And, contrary to 2D illustration, the artist may not have control over the viewpoint of the camera or even the lighting (e.g. in interactive art applications, such as video games). Because of that, designing a shading that accurately reproduces a desired appearance and that stays visually coherent under varying viewpoints and lighting conditions is a challenge requiring both artistic and technical skills. In many industries, such as computer animation and video games, it is a time-consuming task often entrusted to dedicated artists.

Using physically-based models for shading alleviates some of that complexity. The formulation and the parameters of these models are derived from physical phenomena: they ensure a coherent and plausible appearance

for any geometry, under any viewpoint and lighting environment. They also have the advantage of working almost directly with captured data (e.g. captured lighting environments, or scanned materials), which greatly reduces the time spent in designing materials. However, they are by design unsuitable for reproducing any kind of non-physical lighting effect commonly found in illustration.

In contrast, stylized shading models are not physically correct or plausible. Their goals are related to depiction rather than accurate light simulation. For instance, stylized shading models are used in visualization to communicate information about the shape and material of an object in a more efficient way than realistic shading models. An example of this is Gooch shading [Goo+98], which modifies in a non-physical way the illumination term to reveal surfaces in shadow. Some models have been designed to reproduce particular artistic styles in 2D illustration: for example, the well-known and well-studied *toon shading* technique was designed to mimic styles found in comic books.

However, in general, shading rules in illustration are hard to characterize, because contrary to physically based models, they are not directly linked to geometric and physical properties of the scene. Because of that, many stylized shading techniques are limited to specific styles, and usually simple dynamic behaviors.

2.1.1 Light and shade in 2D illustration

The concept of light and shade in 2D illustration is very broad: a depicted surface can be said to "catch the light" in many different ways, usually non-physical, and sometimes not consistent between different objects. An artist can play on the depiction of shading to various ends: as clues for the material of an object, to enhance specific features of a shape, to attract the focus of the spectator to some location on the image, or to set the mood of a scene. For instance, Hogarth [Hog91] proposed five different categories of light and shade in 2D illustrations with traditional media (pencil, pen-and-ink, charcoal, etc.). Each of them have different purposes: for instance, the so-called *sculptural light* is used to ensure that all details of the form of an object are revealed, regardless of whether there is an actual light shining on them.

Additionally, there are even more different rules and techniques when color is added. Custom color gradients can be used in place of photographically accurate illumination gradients for emphasis, abstraction, or as shortcuts for complex lighting effects (Figure 2.1) or convey the appearance of particular materials (Figure 2.2).

This great variety in the depiction of light and shade in both traditional and digital illustration raises the traditional question in stylized rendering: how to reproduce these appearances automatically on 3D scenes? As with other sub-domains of stylized rendering, stylized *shading* is no exception: the artistic rules for light and shade depiction are the rules of artists, and differ from traditional computer graphics or physical models in significant

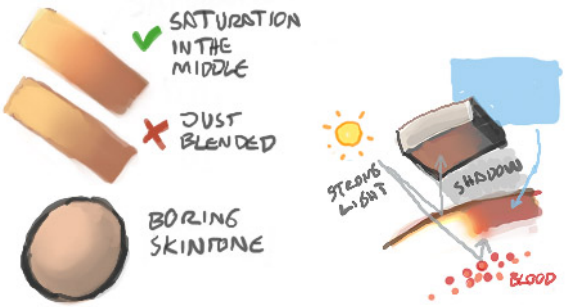


FIGURE 2.1: Illumination gradients in illustration are often modified in non-physical ways. In this example, taken from a digital painting tutorial, illumination gradients on human skin are made more appealing by artifically increasing the saturation at the transition between lit and shadowed (left). A similar technique is used to emulate the visual appearance of subsurface scattering of strong lights inside the skin (right). Source: *Basic to Advanced Color Theory and Illustration Techniques for Photoshop* http://www.floobynooby.com/ICG/artvalues.html

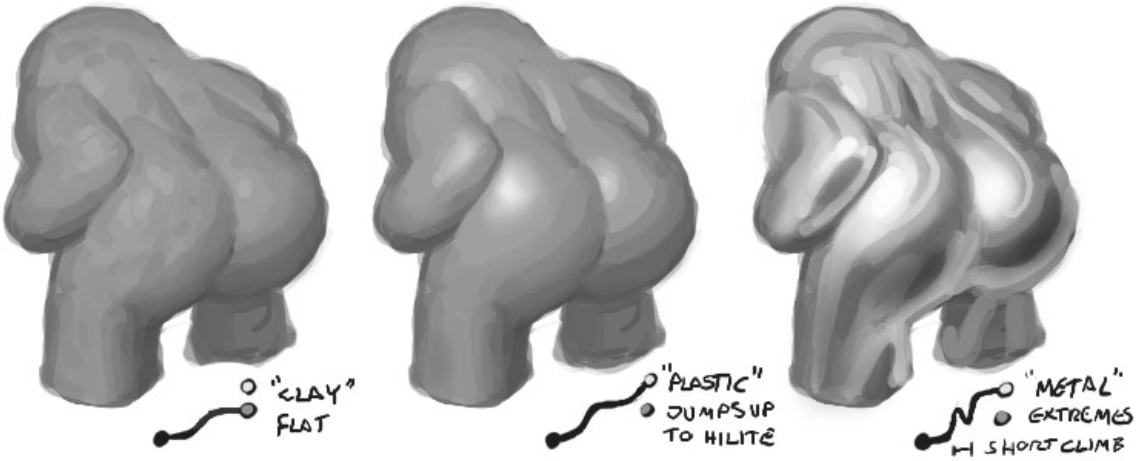


FIGURE 2.2: In the lambertian shading model, the perceived brightness of a lit surface (intensity profile) falls off linearly according to the geometry term (the cosine of the angle between the surface normal and the light direction), resulting in a smooth illumination gradient that conveys diffuse appearance (left). In illustration, simple modifications to this profile can convey dramatically different clues about the material: making the gradient "jump" when facing the light gives the impression of a specular highlight (middle), while introducing a small jump in the middle of the gradient gives off a metallic look (right). However, translating this technique to 3D is non-trivial because the appearance has to be coherent under dynamic viewpoints and lighting conditions. Source: see above.

ways. They vary in subtle ways from artist to artist, which makes it even more difficult to propose a generalized model for stylized shading.

Adding interactivity and dynamic behavior also makes the design process more difficult, as the behavior of lighting effects must now be coherent with motion in addition to shape. For instance, some effects like metallic highlights are expected to slide on the object when moving the viewpoint, contrary to diffuse lighting which stays fixed to a surface until the light themselves are moved. Thus, illustration tricks like the one shown in Figure 2.2 to convey metallic appearances will not work in dynamic 3D scenes. In a way, this means that, under animation, some "important" realistic behaviors of light and shade must be kept so that the intended appearance is not broken.

While this makes stylized shading in dynamic 3D scenes more challenging, it also greatly expands the design space for artists. Instead of reproducing specific examples of 2D light and shade into 3D, another challenge would be to propose efficient tools to let artists explore this design space, in order to create novel stylized appearances.

2.1.2 Our contribution

In this chapter, we present a method, targeted at technical artists, for designing and exploring complex stylized shading effects by combining simple building blocks, in the form of simple shading behaviors. The proposed building blocks have intuitive visual interpretations: our intent is to make them easy to combine in order to create complex shading behaviors by progressive refinement. Going further, our intent is also to provide a practical framework for decomposing complex shading effects found in illustration, and more generally, to facilitate the exploration of the design space of stylized shading.

Our main contribution is to decouple the design of individual shading effects in three independent aspects: (1) Choosing and tuning the global *shading behavior* (diffuse, specular, rim lighting, etc.) of an effect; (2) Adding details and visual complexity; (3) Colorizing the effect.

The final stylized appearance is then obtained by layering several of those shading effects. We provide interactive tools to edit each aspect, allowing the artist to precisely tune each part of the final appearance with a direct visual feedback. Our method automatically keeps a consistent result under varying viewpoints and light configurations. We show that our approach can be used to add spatially and temporally coherent details, mimicking various shading effects in a direct and flexible manner.

2.2 Related work

In computer graphics, stylized depiction of light and shade in a scene can be achieved through several ways: by manipulating the lights themselves, or by changing the way light interacts with a surface, through specialized shading models. First, we review the techniques that manipulate the lighting environment (position, intensity, color of lights) to achieve a stylized look. Note

that, with those techniques, the simulation of light can still follow physical rules. Then, we review existing literature on shading models that achieve non-photorealistic appearances.

2.2.1 Editing lighting environments

One form of lighting manipulation is *inverse lighting* techniques: given some artistic constraints on the final image (e.g. the position and color of an highlight, the size of a shadow cast by an object, etc.), the goal is to infer the parameters of the lights in the scene: their number, position, color, and type. Notable work in this category include work by Pellacini et al. [Pel+07]. However, inverse lighting is usually expressed as a complex non-linear optimization problem: it thus suffers from the common issue that the optimization result may not match the intent of the artist. Several techniques propose a more direct control over the appearance by allowing the user to make non-physical edits to an existing physical result, allowing the user to move and deform reflections [Rit+09], shadows, highlights, refractions [Rit+10; Sch+13], or even the propagation of light rays [KPD10]. However, deforming existing physical effects are impractical for stylized appearances that differ significantly from realistic results.

In contrast, the approach of Okabe et al. [Oka+07] allows users to design image-based lighting environments from scratch by directly painting the desired appearance on a 3D model. Their tool supports arbitrary BRDF models. Still, these techniques cannot be extended to take into account other scene or object attributes typically used for stylization, such as surface curvature, and as such have limited flexibility for representing arbitrary stylized shadings. Lighting environments and materials can also be acquired from real objects. Providing tools that allows artists to intuitively edit captured environments after acquisition is an important area of work [Pel10; Zub+15].

All these methods are mainly used to provide artistic direction for physically-based simulation while maintaining a plausible realistic appearance: fundamentally altering the behavior and appearance of light and shade is not their primary goal. Furthermore, a wide range of shading effects found in 2D illustration cannot be reproduced solely by tweaking the lighting environment: stylization often depends on other attributes of the scene, such as surface curvature. For such effects, specialized shading models must be used.

Lit-spheres *Lit-Spheres* (also called *matcaps*) provide another direct and flexible way to specify the appearance of an object. In the system originally described by Sloan et al. [Slo+01], the appearance is represented by an image of a sphere. The target object is then shaded by environment mapping using this image. This image can be captured, reconstructed from a drawing or a photograph, or painted using digital painting tools. It can accommodate multiple shading styles, from realistic to toon-like, without the unintuitive

control imposed by a BRDF. They give immediate plausible results with little to no manual adjustment required, and are now widely used for shading in 3D modeling software [1] [2]. However, they are limited to static lighting as the final appearance depends only on the camera viewpoint. Note that lit-spheres can capture not only shading but also textural details of the appearance. However, those details are subject to stretching and compression artifacts on shapes with a lot of curvature variations. Todo, Anjyo, and Yokoyama [TAY13] extended this technique by defining the lit-sphere in a light-dependent space, allowing dynamic lighting environments, and further refined it by adding brush stroke effects and highlight shape control that are not subject to deformations (Figure 2.3). However, the range of dynamic behaviors (i.e. how the shading reacts to position and viewpoint changes) that can be modeled with this technique is still limited.

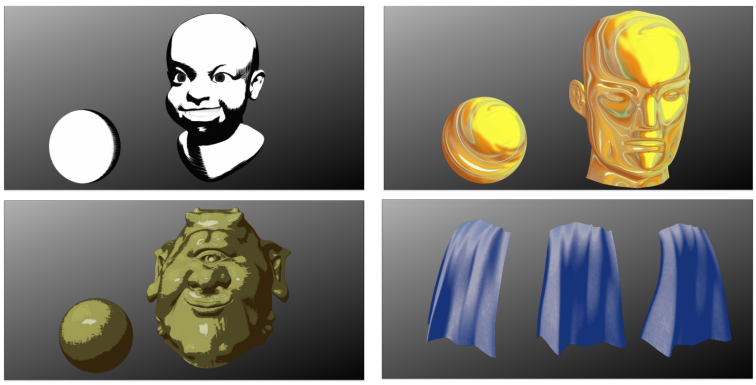


FIGURE 2.3: Examples of lit-sphere shading with the method of Todo, Anjyo, and Yokoyama [TAY13]. Images taken from the paper.

2.2.2 Stylized shading models

Simple shading models derived from physical principles of light propagation are usually not suited to reproduce light and shade found in illustrations, as they do not convey information about a shape in the most effective way possible. To illustrate this, let's consider lambertian shading, used in computer graphics as a physical model of purely diffuse surfaces. In this model, a color term c is multiplied by the incoming light I_{in} and modulated by the *geometric term* $(\max(0, \mathbf{n}.\mathbf{l}))$ that accounts for the angle at which the light is striking the surface:

$$I_{\text{lambertian}} = c \times I_{\text{in}} \times \max(0, \mathbf{n}.\mathbf{l}) \tag{2.1}$$

In this model, surfaces facing away from the light appear black and flat: details about the shape are lost. The *half-lambert* shading model is a simple

[1] Pixologic ZBrush Features http://pixologic.com/zbrush/features/Materials/

[2] Shading - Blender Manual https://docs.blender.org/manual/nb/dev/editors/3dview/properties/shading.html

non-physical alteration of the lambertian model to reveal those shadowed surfaces:

$$I_{\text{half-lambert}} = c \times I_{\text{in}}(\frac{1}{2} + \frac{1}{2}\mathbf{n}.\mathbf{l}) \tag{2.2}$$

Note that the geometric term has been replaced by the *unclamped* geometric term $\mathbf{n}.\mathbf{l}$, and rescaled so that it lies in $[0; 1]$. This simple modification appears in many stylized shading models [Goo+98; RBD06; MFE07], as a way to convey the shape of an object more clearly, regardless of lighting conditions.

In illustration, changes in illumination of a surface are also depicted by hue shifts in addition to variations in luminance: notably, the shadows tend to be depicted with a cool color instead of black. From this observation, Gooch et al. [Goo+98] proposed a shading model that reproduces these hue shifts by using a cool-to-warm color map, obtained by combining a blue-to-yellow gradient with the lambertian shading gradient (black to object color). The half-lambert modification is used to interpolate into this color map, so that the form is revealed even on surfaces facing away from the light.

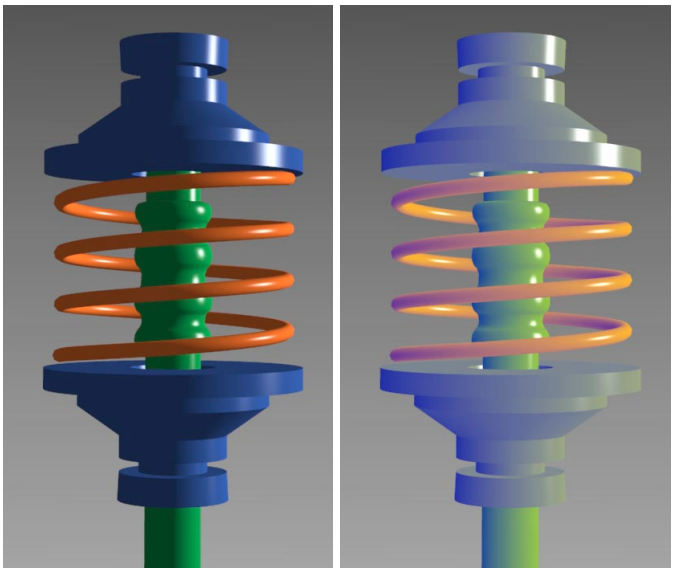


FIGURE 2.4: Comparison between standard Phong shading and Gooch shading. With Phong shading, surfaces facing away from the light appear flat, and surface detail is lost. Gooch shading reveals all surface details, even when they do not face the light. Image source: [Goo+98].

Similar hue shifts are present in watercolors: illumination in watercolor is painted with layers of pigments with varying dilution. More pigments are deposited on dark regions, while brightly lit regions actually correspond to an absence of pigments. The varying pigment density produces changes in the color temperature in addition to changes in value [LM01]. Although simulation models for the diffusion of watercolor pigments exist [Cur+97], most real-time watercolor stylization techniques use simplified shading, filtering, and color modification models to approximate diffusion effects and color variations due to pigment density [LM01; BWK05; Bou+06; MSR16].

These shading models work best when it is possible to adjust the lighting accordingly. However, in some contexts, it is not possible to control precisely the lighting. This is notably the case in video games, where designers have only limited control over the position of the lights relative to characters, as the latter can be moved around the scene. Yet, for gameplay purposes, characters should be clearly identifiable at all times, in widely varying lighting conditions. Mitchell, Francke, and Eng [MFE07] described the shading techniques used to effectively convey the shape and silhouettes of stylized characters in the video game *Team Fortress 2*. The shading model they described is composed of many components: an ambient term, diffuse and specular terms, and a rim-lighting term to reveal silhouettes. This shows that shading techniques can rapidly grow complex in highly dynamic environments: in our work, we seek a way to rapidly explore combinations of shading terms to design such complex models.

To properly depict surface details at various scales, many non-physical shading techniques have been proposed: Rusinkiewicz, Burns, and DeCarlo [RBD06] used a custom shading model based on the half-lambertian to enhance the perception of both the overall shape and details. This shading model is modulated by a user-controlled exaggeration parameter. By evaluating the shading at multiple scales, from fully detailed to heavily smoothed geometry, the user can separately control the degree of emphasis of the overall shape and of the surface details. Another common way to enhance the shape details is to modulate the basic shading term with *surface curvature* to better reveal ridges and creases [Ver+09; Ver+10; Ver+11a]. Vergne et al. [Ver+08] proposed a generalization of curvature-based shading: through the *apparent relief* descriptor, they are able to extract specific features of a shape, such as ridges, valleys and flat regions with more flexibility and more intuitive control than previous methods. The extracted information can then be used as an additional input to stylized shading models, to increase the range of achievable appearances. Note that some of these techniques make use of screen-space filtering to enhance shading.

Toon shading The intent of toon shading techniques is to mimic light-and-shade depiction in cartoons and cel animation. They use few colors (typically under five colors) to depict illumination gradients. Visually, this results in characteristic color bands. The position and size of the bands can be adjusted by artists through a color ramp.

Many extensions to the basic toon shading model have been proposed: for instance, X-toon [BTM06] extends the traditional toon shading by allowing users to vary tone detail according to a view-dependent scene attribute (e.g. the scene depth), effectively replacing the traditional 1D toon color ramp with a 2D texture (Figure 2.5). Another extension of toon shading was proposed by Todo, Anjyo, and Igarashi [TAI09]: their approach can reproduce various expressive lighting effects used in hand-drawn cartoon animation, such as the straight lighting effects used for the depiction of flat reflective surfaces.

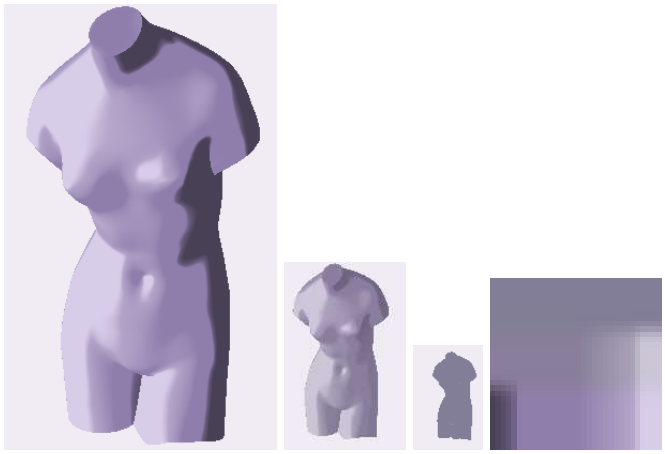


FIGURE 2.5: X-Toon shading [BTM06] example at different abstraction levels and associated 2D toon texture map. The abstraction level can be linked to view-dependent scene attributes, such as scene depth. Images taken from the paper.

Highlights Specular highlights convey important material clues about a surface. When depicting them, illustrators also take great liberties with physics. Recognizing this, several techniques have been proposed to provide artistic control over the shape and behavior of highlights on a surface. Anjyo, Wemler, and Baxter [AWB06] propose a method to alter the shape of a cartoon highlight in various ways: translation, deformation, rotation, squaring and splitting (see Figure 2.6). These properties can be animated over time with a keyframing system. BRDFshop [CPK06] allows one to create physically correct BRDFs from hand-painted highlights instead of indirectly manipulating numerical values. Finally, in the system of Pacanowski et al. [Pac+08], the shape and color gradient of the highlight can be directly sketched in a 2D plane oriented perpendicularly to the reflected direction.

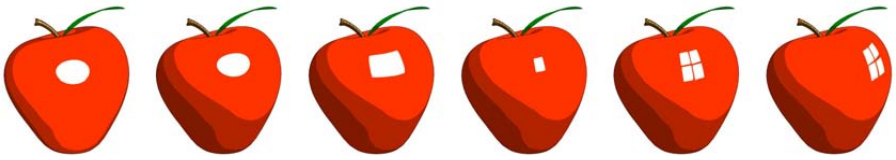
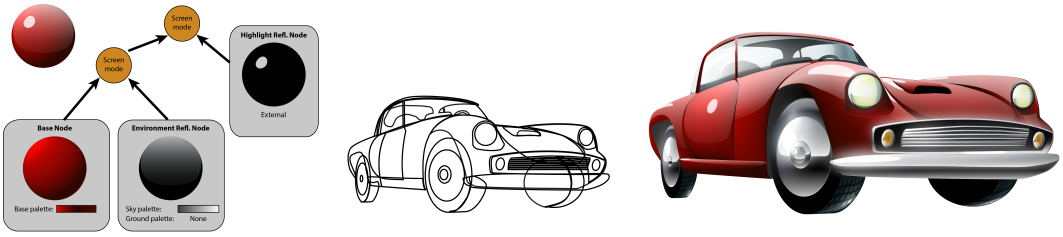


FIGURE 2.6: Tweakable light and shade: dragging the highlight on a surface, squaring, scaling, and splitting. Image source: [AWB06]

However, all these techniques are specialized for specular highlights. While they provide good artistic control and can lead to highly stylized results, they tend to have a very localized impact on the final image. In this work, we seek a more generic approach for artistic control of shading: our intuition is that there is value in a system that would use the same kind of primitives for various shading effects, and that would allow artists to combine those primitives in novel ways, instead of considering specific effects in isolation.

This is similar to *shade trees* [Coo84], and more recent graphical shader editors [3] [4] that represent the shading equations as a tree of operation nodes. However, while these approaches are very flexible, they are also very low-level, to the point that they approach the same degree of flexibility as shader code. Thus, they can be seen as a way of structuring and visualizing shader code, but do not reduce its inherent complexity and do not immediately facilitate exploration of different appearances.

In the context of 2D vector illustration, a closer approach is *Vector Shade Trees* [Lop+13]. This system can be used to construct complex appearances by arranging a set of basic *shade nodes* in a compositing tree. These nodes are specialized vector primitives that are derived from illustration guidelines for material depiction. With their system, an artist can quickly imitate the appearance of transparent, translucent or reflective objects by combining a few of these primitives. This is close to what we want to achieve in the context of 3D scenes: one of our goals is to allow artists to use some of these guidelines in the context of 3D animated scenes with dynamic lighting environments.



(a) Vector Shade Tree (b) Vector line drawing (c) Our material depiction

FIGURE 2.7: Material design in 2D illustration with *Vector Shade Trees*. The user arranges multiple basic shading primitives in a compositing tree (left). The material can then be applied to vector line drawings (middle, right). Image source: [Lop+13]

Our approach is most closely related to the work of Vanderhaeghe et al. [Van+11] with *Dynamic stylized shading primitives*. Their system can reproduce various stylized shadings with a layered combination of primitives. Primitives are composed of a base shading parametrization that can vary continuously between diffuse and specular behaviors, and parameters can be tweaked to control the anisotropy of the shading effect, and the shape of the intensity profile. The approach we propose is similar, but with lower-level primitives: Instead of having a single adjustable parametrization, we allow users to freely mix-and-match simpler parametrizations representing different shading behaviors (e.g. diffuse, specular, etc.) and scene properties (e.g. curvature) to create novel behaviors.

In the recent work on *barycentric shaders* by Akleman, Liu, and House [ALH16], shading is art-directed via user-provided *control images*, which are then mixed according to *weight images* derived from usual shading parametrizations (diffuse, specular, rim-lighting, depth) and remapping functions.

[3] Unreal Engine: Essential Material Concepts https://docs.unrealengine.com/en-US/Engine/Rendering/Materials/IntroductionToMaterials

[4] Unity: Shader Graph https://unity3d.com/shader-graph

This is similar to the system we propose, although our system does not incorporate local control via textures, and we adopted a lower-level approach in which we chose to directly expose the functions that define the illumination profile as discrete 1D texture maps, editable on-the-fly by the user with various tools. Additionally, we expose basic operations on parametrizations to combine them and create new shading behaviors, and locally perturb them to convey hints about the small-scale geometry of the surface.

2.3 Motivation and overview

Shading design is a tedious process, especially with highly dynamic 3D scenes. We've seen that a lot of different models exist, yet they share some common terms in their formulation. For example, diffuse effects are always based upon the geometric term $\mathbf{n} \cdot \mathbf{l}$. Similarly, the surface curvature is often used in shading models for emphasizing details. These models only differ by how they map those basic shading terms to actual colors on the screen.

Designers know how to combine these terms into full models to achieve artistic goals: reveal shapes in more detail, convey mood in a scene effectively, etc. But this is often done through a tedious process of fine-tuning coefficients of shading terms and writing shader code. In this chapter, we propose a tool to facilitate exploration of, and experimentation with, shading models. In our tool, the user designs a shading by layering one or more *shading effects* each representing one independent component of the resulting shading. We structure the design process of individual shading effects in clear, distinct steps:

- Choosing and tuning the *behavior* of a shading term. This is done through combinations of *base parametrizations* and *value maps*: they control how the shading effect will move and spread on the object when changing viewpoints or lights. Base parametrizations are modeled after the basic terms common to many shading models.

- Controlling the color gradient. The illumination gradient of a shading effect is controlled through a 1D color map, directly editable by the user.

- Adding small-scale details to shading effects using *perturbations* that locally modify parametrizations. To keep the appearance spatially and temporally coherent, details are generated using 3D procedural noises.

An overview of the different steps of our workflow is provided in Figure 2.8. A typical session with our tool starts with the user loading the object onto which they wish to design the shading, and setting the lighting configuration (number and position of lights). From then on, users can refine the appearance of the object on the screen by adding one or more shading effects in a layered fashion. The design process is fully dynamic: at any point the user is able to move the camera or the lights and the displayed appearance will update accordingly.

From a technical standpoint, the system is implemented on the GPU as a series of screen-space operations on standard G-buffers: surface normals and tangents, depth, etc. It thus falls in the category of *deferred shading* techniques.

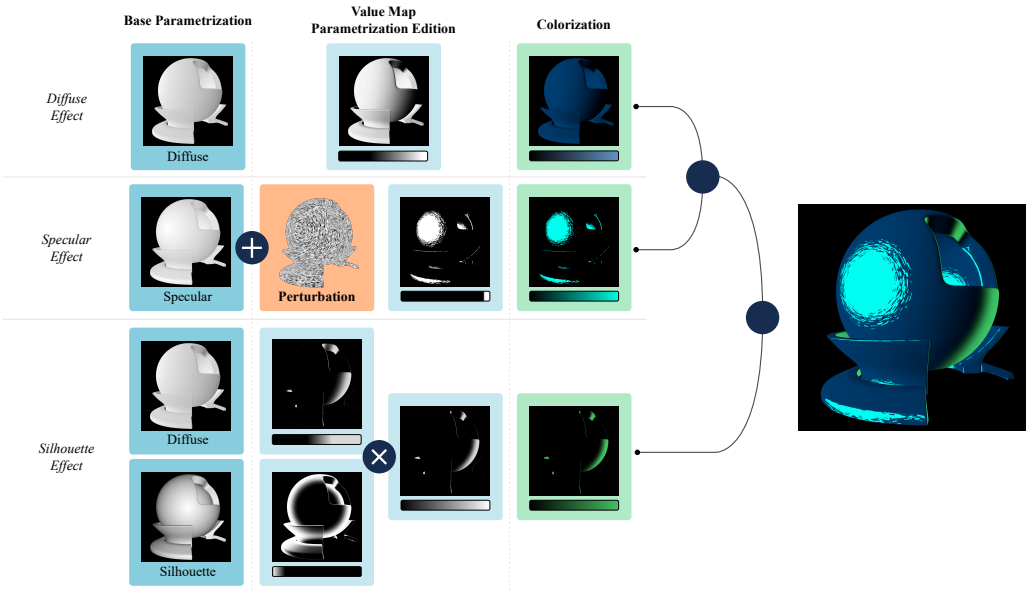


FIGURE 2.8: Illustration of our workflow showing an example with three appearance effects. A user can modify and combine *base parametrizations* to design the shading behavior (blue nodes) of an appearance effect, using value maps and combination operations. A color map (green nodes) is then applied on the designed behavior to colorize the effect. Output effects are then composited to obtain the final appearance. Perturbations (orange nodes) can be attached to every operation in order to add procedural details to an effect. The orientation of the perturbation can be controlled by the gradient of a shading behavior (as shown here), or by an external vector field, such as a tangent map.

2.4 Shading behaviors

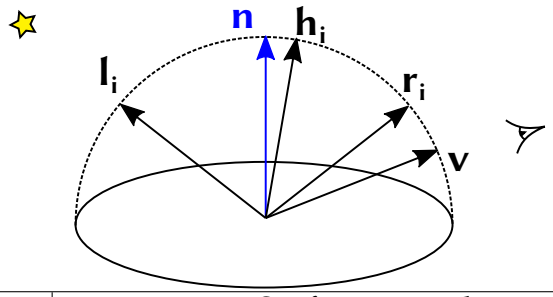
Shading behaviors describe the location and extent of a shading effect and its dynamic behavior when moving the object, the light, or the viewpoint. Users of our system design behaviors by first choosing and optionally combining *base parametrizations*. Then, a value map is applied on base parametrizations, to adjust the intensity profile of the resulting effect.

Formally, since we are in a deferred shading configuration, a shading behavior can be seen as a screen-space value: $\text{shade}(x, y)$, with (x, y) being the screen-space position. In more detail, we define a shading behavior as the application of a *value map* $f : [0,1] \rightarrow [0,1]$ on a *parametrization* $p(x, y) \in [0,1]$. For clarity we omit (x, y) in the rest of the chapter. Thus:

$$\text{shade} = f\,(p)$$

The parametrization p is a screen-space term derived from G-buffers (normals, depth, etc.) and global scene parameters (light positions, viewpoint) and that describes a base shading behavior. We provide several *base parametrizations* that represent common shading behaviors. They can be used as-is or combined to produce more complex behaviors.

2.4.1 Base parametrizations



$\mathbf{n}$	Surface normal
$\mathbf{l_i}$	Light direction: unit vector to light i
$\mathbf{r_i}$	Reflected light direction
$\mathbf{v}$	View direction: unit vector to eye position
$\mathbf{h_i} = \frac{\mathbf{l_i}+\mathbf{v}}{\lvert\mathbf{l_i}+\mathbf{v}\rvert}$	Halfway vector between the light direction i and view direction

FIGURE 2.9: Local geometry used in the definition of base parametrizations.

In computer graphics, shading models are composed of a sum of shading terms that model different behaviors. For instance, the Phong shading model [Pho75] has both a *diffuse* term and a *specular* term, for diffuse light and specular highlights. Individually, most shading terms depend on scalar values derived from the local geometry of the surface, the viewpoint, and the position of the lights. For example, terms that account for diffuse lighting have a dependency on the *geometric term* $\mathbf{n} \cdot \mathbf{l}$ (the cosine of the angle between the light and the surface normal); specular highlights depend on $\mathbf{n} \cdot \mathbf{h}$ or $\mathbf{r} \cdot \mathbf{v}$ depending on the model; Fresnel reflection effects are usually approximated with a term that depends on $\mathbf{n} \cdot \mathbf{v}$. In this work, we call these values *parametrizations*. They determine the global behavior of a shading term on a surface: how it will move and spread in response to light and view changes.

As in the technique of Akleman, Liu, and House [ALH16], we propose to use several base parametrizations to encode basic shading behaviors. These are derived from observation of the terms that appear frequently in shading models. In addition to terms derived from the local surface geometry used in physical models, we also add screen-space terms such as the screen-space surface curvature, which can be used to enhance shape features.

Diffuse behavior: Diffuse shading effects depend only on the geometric term $\mathbf{n} \cdot \mathbf{l}$. Notably, they are not view-dependent and do not slide on the object when moving the viewpoint. In our workflow, diffuse behaviors

are parameterized by an angular remapping of the geometric term:

$$p_{\text{diffuse}} = 1 - \text{acos}\left(\mathbf{n} \cdot \mathbf{l_i}\right) / \pi$$

This limits the distortion of illumination profiles on surfaces. A similar remapping is employed in the system of Vanderhaeghe et al. [Van+11]. Similarly to the half-lambert model, the geometric term is not clamped between 0 and 1 and can be used to reveal geometry that is not facing the light. Note that there is a base diffuse parametrization for each light in the scene.

Specular behavior: Similarly, specular behaviors are parameterized by:

$$p_{\text{specular}} = 1 - \text{acos}\left(\mathbf{n} \cdot \mathbf{h_i}\right) / \pi$$

where $\mathbf{h_i} = \frac{\mathbf{l_i}+\mathbf{v}}{|\mathbf{l_i}+\mathbf{v}|}$. They are used to add specular highlights to objects.

Silhouette: shading behaviors that affect the object silhouettes are parameterized by:

$$p_{\text{silhouette}} = 1 - 2 \times \text{acos}\left(\mathbf{n} \cdot \mathbf{v}\right) / \pi$$

This term ranges from 0 at grazing view angles, to 1 on normals oriented towards the camera. In shading models, dependencies on the $\mathbf{n} \cdot \mathbf{v}$ term typically appear in *rim-lighting* terms, to position lighting effects on silhouettes, or to mimic the contribution of a Fresnel term.

Curvature: Curvature shading effects are parameterized by the screen-space view-dependent mean curvature κ [Ver+11a], remapped in the $[0, 1]$ range:

$$p_{\text{curvature}} = 0.5 \times \left(1 + \tanh\left(s \times \kappa\right)\right)$$

where s is a user-controllable parameter to adjust the covered value range. It can be used in combination with other parametrizations to position shading effects on sharp object features.

Thickness: the thickness parametrization is defined by the distance between the projected front and back object faces.

$$p_{\text{thickness}} = |z_{\text{front}} - z_{\text{back}}|$$

This requires a slight modification to the classic rendering pipeline to keep track of both the frontmost and the backmost depths. We found that it can provide a good visual approximation of translucency effects for simple geometries that do not have large hollow parts.

A visualization of those parametrizations on a test object for a given light position and viewpoint is provided in Figure 2.10.

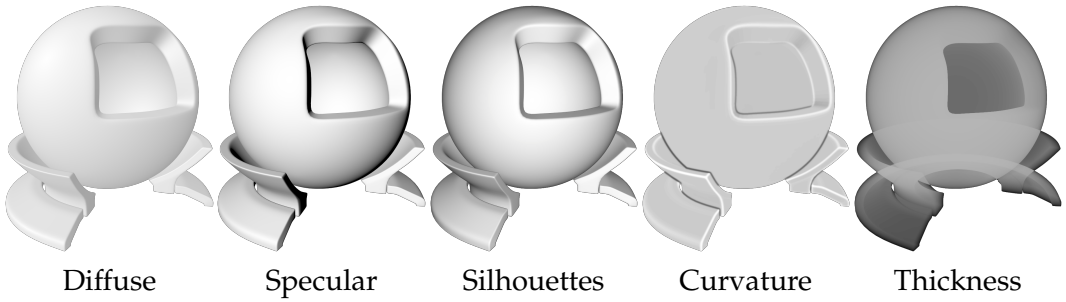


FIGURE 2.10: Visualization of base parametrizations on a given object, for a given light position and viewpoint. Parametrizations are in the $[0, 1]$ value range, from black to white. Users are able to see the how a parametrization evolves by dynamically moving lights and the camera.

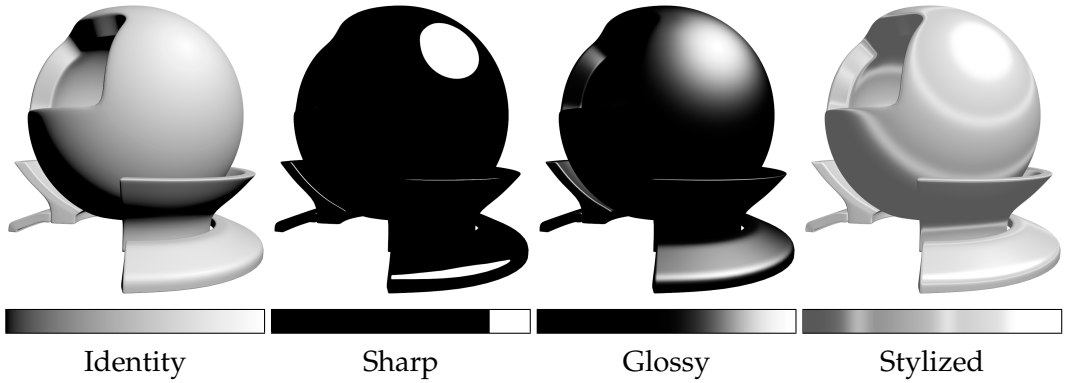


FIGURE 2.11: Use of value maps on a specular base parametrization to produce highlights of varying glossiness, and a stylized effect.

2.4.2 Value maps

The shading behavior described by a parametrization can be refined with the value map f, stored in a 1D texture. By default the value map is initialized to the identity function and does not modify the behavior of the parametrization p, so the user sees one of the images of Figure 2.10. It can be modified it to, for instance, increase the value range of a parametrization, modify the falloff at shading terminators in a manner similar to Mitchell, Francke, and Eng [MFE07], introduce toon-like hard value transitions, or change the spread and falloff of specular highlights (see Figure 2.11).

2.4.3 Composition

A user can choose one of the base parametrizations unmodified, or can design more complex behaviors by combining them together using traditional compositing operators: *multiply*, which can be used for masking part of parametrizations, akin to a logical *and* operation; and *screen*, which has the opposite effect of multiply and can be used to merge the behavior of two parametrizations together (logical *or*). In formal terms, given two parametrizations p_a

and p_b:

$$p_{\text{screen}} = 1 - (1 - p_a) \cdot (1 - p_b)$$
$$p_{\text{multiply}} = p_a \cdot p_b$$

This compositing approach to combining parametrizations provides the most flexibility and is also more intuitive to users already familiar with compositing software. In tandem with value maps, the composition operators can be used to restrain the location and spread of a shading effect. For instance, assume that the user wants a shading effect that behaves like specular highlights, but that is only present on sharp edges. This can be achieved with a base *specular* parametrization, multiplied by the *curvature* parametrization with a value map to filter desired areas of high curvature.

2.5 Perturbation terms

Parametrizations can be modified with procedural noise before applying a color map, in a manner similar to domain warping. In our system, this is implemented as *perturbations*: optional operations that modulate the parametrization p of a shading behavior with a locally varying *perturbation term* d based on solid procedural noise.

Warping locally modifies the behavior of a parametrization: it allows details that are revealed by the shading effect. In constrast, compositing the details over the final result would produce details everywhere regardless of shading and would not affect the underlying behavior of the shading. Furthermore, procedural noise allows users to quickly experiment with adding visual details on an object, without needing to modify the geometry or to manually paint textures.

In the following paragraphs, p' is the perturbed parametrization, p is the original parametrization p' is the combination of the original parameter p with a spatially varying *perturbation term* $d \in [0, 1]$, given by the evaluation of a procedural noise, using one of the operations below:

Offsetting p' is given by offsetting the original parameter with the perturbation term and clamping the result in $[0, 1]$, following the formula

$$p' = \text{clamp}\,(p + \mathit{scale} \times (d - 0.5))$$

where *scale* is a user-defined scaling factor. A typical use for this operator is to jitter the boundaries of *hard* shading features introduced in a value map (such as terminators or hard specular highlights). The strength of this effect is controlled by the scaling factor s, as shown in Figure 2.12. For parametrizations that depend on the surface normal, the offsetting operation can reproduce effects that are perceptually similar to bump mapping.

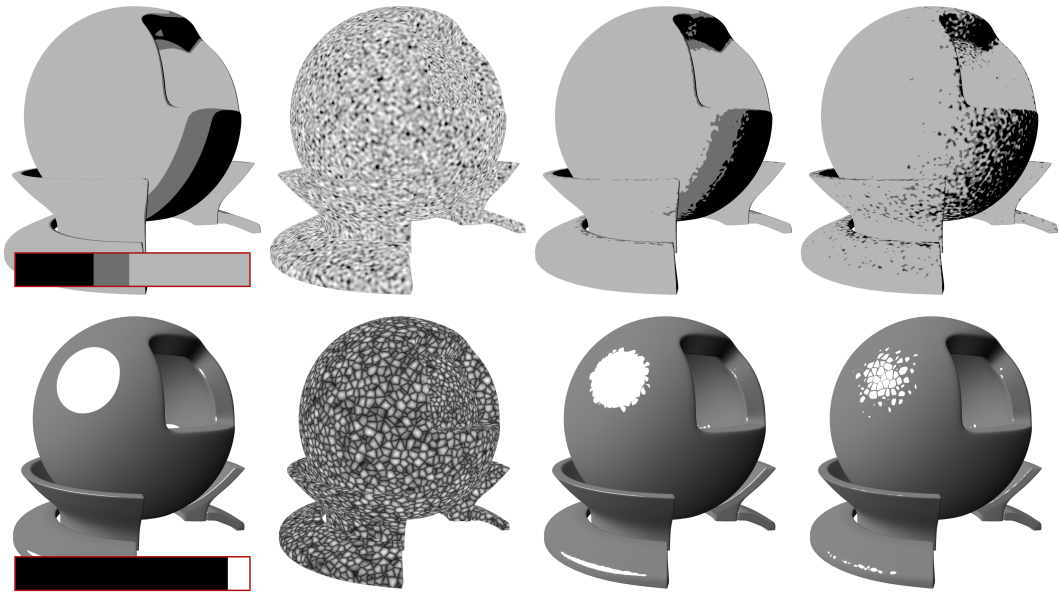


FIGURE 2.12: Offsetting examples. (Top) A toon shading transition offset with a gradient noise. (Bottom) A specular behavior offset with a cellular noise produces highlights with complex shapes. A high scale parameter increases the size of the jitter effect.

Texturing p' is given by:

$$p' = p \cdot d$$

The perturbation term will affect the object globally instead of being localized at shading terminators. Thus, in contrast with offsetting, this operation can be said to add textural details. A usage example of this operator is given in Figure 2.15(g).

Multiple perturbation steps can be chained together allowing complex dynamic stylized appearances. The perturbation term d comes from the evaluation of a 3D (solid) procedural noise in model space. This avoids the shower-door artifacts commonly found with 2D procedural noise. Another advantage of procedural noises compared to bitmap images is that they do not need any surface parametrization and can be evaluated in real-time on animated, deformable objects. In our deferred implementation, the solid noise is evaluated in screen-space, using the G-buffer containing the 3D model-space position. This results in a 2D image containing perturbation values at all positions on the screen. In our implementation, we provide two choices of noise models to generate this perturbation term:

Gradient noise: A grayscale solid gradient noise (Perlin noise) [Per85]. The user can control its frequency, amplitude (centered on 0.5) and number of octaves. For example, low amplitude gradient noise used with offsetting can produce a bumpy appearance on objects with hard shading or sharp highlights, as show in Figure 2.12.

Cellular noise: This is the model described by Worley [Wor96]. It generates structured details resembling scales with a user-defined frequency and regularity. Figure 2.12 shows cellular noise perturbing a specular parametrization to alter the shape of a specular highlight and give the impression of surface irregularities.

2.5.1 Line Integral Convolution

Additionally, to increase the range of achievable appearances with procedural solid noise, a user can optionally apply a *Line Integral Convolution*(LIC) [CL93] filter on the generated solid noise. LIC filters are mostly used for the visualization of vector fields, but have also been used for stylization purposes [LM01]. The technique we propose in our pipeline is similar to the latter: the apply the LIC filter in the 2D image containing the generated solid noise. Two options are proposed to the user for the vector field used to guide the LIC filter:

1. The LIC is guided by the screen-space gradient of the parametrization being perturbed. This gradient can optionally be rotated 90 degrees. This allows brush- and sketch-like effects that are oriented according to shading features.

2. The LIC is guided by a vector field defined on the surface of the object, projected in screen-space (for instance, the projected surface tangents or normals).

We show in Section 2.8 that this step allows the user to approximate a range of anisotropic shading effects and appearances, such as a brushed metal look, view- and light-dependent cross-hatching effects, or painterly strokes oriented according to the illumination falloff.

2.6 Colorization and compositing

Finally, the colorized result of one appearance effect is obtained by applying a 1D color map on the shading behavior: a 1D color map is a function $f : [0,1] \rightarrow [0,1]^4$ stored in a 1D RGBA texture. An artist can edit this color map to change the shading tones, alpha channel and falloff of shading features. Note that there is only one color mapping operation per shading effect.

The result of individual shading effects are then blended together according to a stack of compositing operations to obtain the final appearance, as shown in Figure 2.8. In addition to regular alpha blending, several standard compositing operations are available at this stage, including the *multiply*, *screen* and *overlay* blend modes.

2.7 User interface

During the design process, the user has to edit 1D value maps and color maps. We chose to expose them directly instead of having a model with

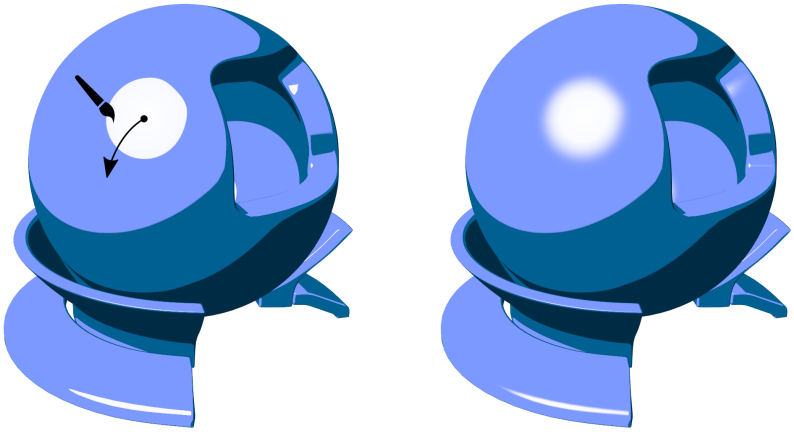


FIGURE 2.13: Blurring a specular highlight with the click-and-drag interface on the object. (Left) With the highlight color map currently selected, the user draws a stroke on the object to define the range of parameters to blur on the color map. (Right) Result of the blur operation.

parameters for artistic flexibility. However, editing these maps can also be tedious: to alleviate this, we provide editing tools to faciliate frequent operations done on color and value maps, which should be familiar to users of digital painting software:

Flat The *flat* tool paints a constant color or value over the selected parameter range. This is useful to create toon-like color bands.

Gradient The *gradient* tool paints a linear gradient. It can be used to paint smooth tonal variations.

Blur The *blur* tool applies a 1D gaussian blur over the selected parameter range. A typical use case is to increase perceived glossiness of an object by blurring the falloff of a specular highlight.

These tools operate on a parameter range of the map. The range can be selected by clicking and dragging 1D view of the map, or directly on the object by drawing a stroke: in the latter case, the parameter range is determined by looking up the values of the associated parametrization at the endpoints of the stroke. An example of user interaction is shown in Figure 2.13.

2.8 Results

Our workflow is implemented in C++ and OpenGL. All operations are done on the GPU. While there is considerable room for improvement on the performance of our implementation (most of our image passes could be fused in one single shader pass), our system still keeps interactive frame rates (> 20 FPS), even for complex styles. Note that our system only requires normal and depth maps for calculating the base parametrizations, and position maps

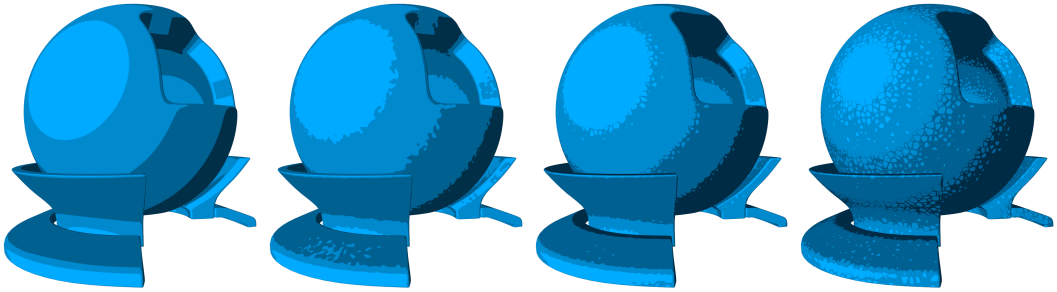


FIGURE 2.14: Simple toon shading (a). Our system easily allows details to be added with different offsetting noises (b,c) and offsetting scales (c,d).

for coherent 3D noise. Thus, it can be easily integrated into any pipeline that produces these three outputs.

A user starts by loading a mesh from a file, and by adding a first appearance effect. The initial view shows the default shading behavior. From then on, the user can edit the value map or color map by clicking and dragging on the mesh or directly on the associated 1D textures, add a perturbation operation, or add new appearance effects. All modifications are reflected in real-time on the mesh. The designed appearance stays independent of the model used for editing: this allows the user to change the mesh during a design session and resume editing operations on this new mesh seamlessly. Our interface also provides controls for adjusting global parameters, such as the position of the lights (azimuth and elevation), or the scaling parameter for the surface curvature. These controls are shared by all appearance effects. The user can also move the camera at any time by clicking and dragging on the object.

Directional lights can be added in a scene as needed, each light having its own associated *Diffuse* and *Specular* parametrizations. This way, a user can bind shading features to one particular light. Their directions can be controlled individually, or rotated all at once as a single lighting environment. By progressively adding different effects that depend on different lights, users can emulate complex lighting environments, as shown in Figure 2.15(e).

Figure 2.14 shows how our framework can accommodate simple toon shading effects with one appearance effect parameterized by a *Diffuse* behavior (a). In (b-d) we add a perturbation operation with the offsetting mode, using unmodified gradient noise (b), and cellular noise (c,d). In (c,d) we vary the scale parameter s of the offsetting to increase or decrease the amount of added detail.

In Figure 2.15, we show various styles achievable with our system. Simple toon shading (a) is implemented with only one appearance effect. The shading tones were painted directly on the object using the flat tool. In (b), the same color map was smoothed and parameterized by a *Silhouette* behavior instead of *Diffuse*. A glossy specular highlight was added as a second appearance effect. Figure 2.15(c) illustrates the use of the thickness base parametrization to give a convincing impression of a translucent material.

In (d), we combined a simple diffuse effect with a silhouette lighting effect appearing only in unlit regions. In (e), we illustrate the ability of our system to bind appearance effects to different lights: two glossy specular effects were added and parameterized by two different lights. In (f), we offset a specular highlight with a high-frequency cellular noise to reproduce a glinting effect. The use of procedural noise allows a coherent result when moving the light. In (g), we illustrate the effect of the *texturing* perturbation mode. Two perpendicularly oriented noises are combined with a diffuse behavior using a texturing operation. The result is oriented along the image-space gradient of the diffuse term and follows its variations. Finally, in (h) we re-used the metallic appearance shown in (b) and applied an offsetting operation with an oriented noise in order to reproduce a brushed metal appearance.

In Figure 2.16 we illustrate the behavior of a complex combination of appearance effects under varying lighting conditions. The appearance shown is composed of a base diffuse effect (orange), and an edge lighting effect (cyan) that appears only on object silhouettes and regions of high curvature. The behavior of this effect was designed by merging the *Silhouette* and *Curvature* behaviors using the *Screen* mode, and by multiplying the result with a remapped *Diffuse* behavior, so that the effect appears only in the unlit part of the object. By providing simple compositing operations to combine and remap shading behaviors, our system allows fast prototyping of such complex shading behaviors without the need for extensive technical knowledge.

Figure 2.17 shows the decomposition of an appearance effect based on surface curvature: we combined, using the multiply blending mode, value-mapped *Specular* (b) and *Curvature* (a) behaviors to obtain highlights localized on sharp object edges that behave specularly (c). The final result is shown in (d). We then transferred this style to other objects, as shown in the bottom row. The size of the edge highlight effect can be modified either by tweaking the value map used to select the desired curvature range or by adjusting the global scaling parameter for curvature.

2.9 Conclusion

We proposed a workflow and a set of editing tools to design stylized appearances on 3D objects coherent under light and viewpoint changes. Our main contribution is the separation of the design of individual appearance effects in three aspects: *shading behavior* of the effect, addition of *procedural details*, and *colorization*. This decomposition offers a more precise and predictable result than offline appearance transfer techniques, while being more flexible and easier to use for non-technical artists than specialized shading models. Our workflow allows stylization results to be calculated in real-time and is thus usable in interactive contexts such as visualization, 3D modeling, or video games.

We have provided several base parametrizations that encode common shading behaviors and proposed ways to control their size and placement on an object. For non-technical artists, we feel that this approach is more

direct for designing complex behaviors than hand-written shaders, and resembles the shading design process in 2D digital painting. It could however benefit from more base parametrizations: notably, a limitation of our current system is the inability to reproduce convincing refraction effects for translucent objects. More work is needed in this direction to provide intuitive base parametrizations for these effects. The same is true for global illumination effects such as color bleeding between objects in a scene, or subsurface scattering, for which we should provide specialized parametrizations.

Currently, the parametrizations are only defined on pixels covered by the rasterized object. This means that all shading and appearance effects are cut at object silhouettes. A possible improvement to our system would be to extend parametrizations in a region surrounding the object in order to create appearance effects that alter the object silhouette, such as a glow effect. In Chapter 3, we describe the challenges of doing so, and propose solutions.

Concerning procedural details, the combination of 3D gradient noise and line integral convolution ended up being sufficient for a wide range of appearances. Our system can also be used with cellular noise to produce medium-scale details, but could be extended with other kinds of structured noises.

The use of 3D noise produces details that are fully coherent with scene motion. However, there are cases where a more 2D aspect is desired: in this case, our stylization approach could be extended with coherent 2D noise primitives [Bén+10]. Another limitation of our system in this aspect is that small-scale procedural details on shading features are not preserved when transferring the appearance to a mesh with high-frequency geometric detail. In these scenarios, we would want shading effects to ignore the high-frequency geometric details. This could be done by rendering parametrizations with a smoothed out mesh, or by prefiltering the existing parametrizations to remove high-frequency variations.

In our model, complex shading behaviors can be obtained by combining parametrizations using standard compositing operators and value maps. In a future work, we would like to automate part of this process by inferring parametrizations from an artist provided value distribution made with conventional digital painting tools.

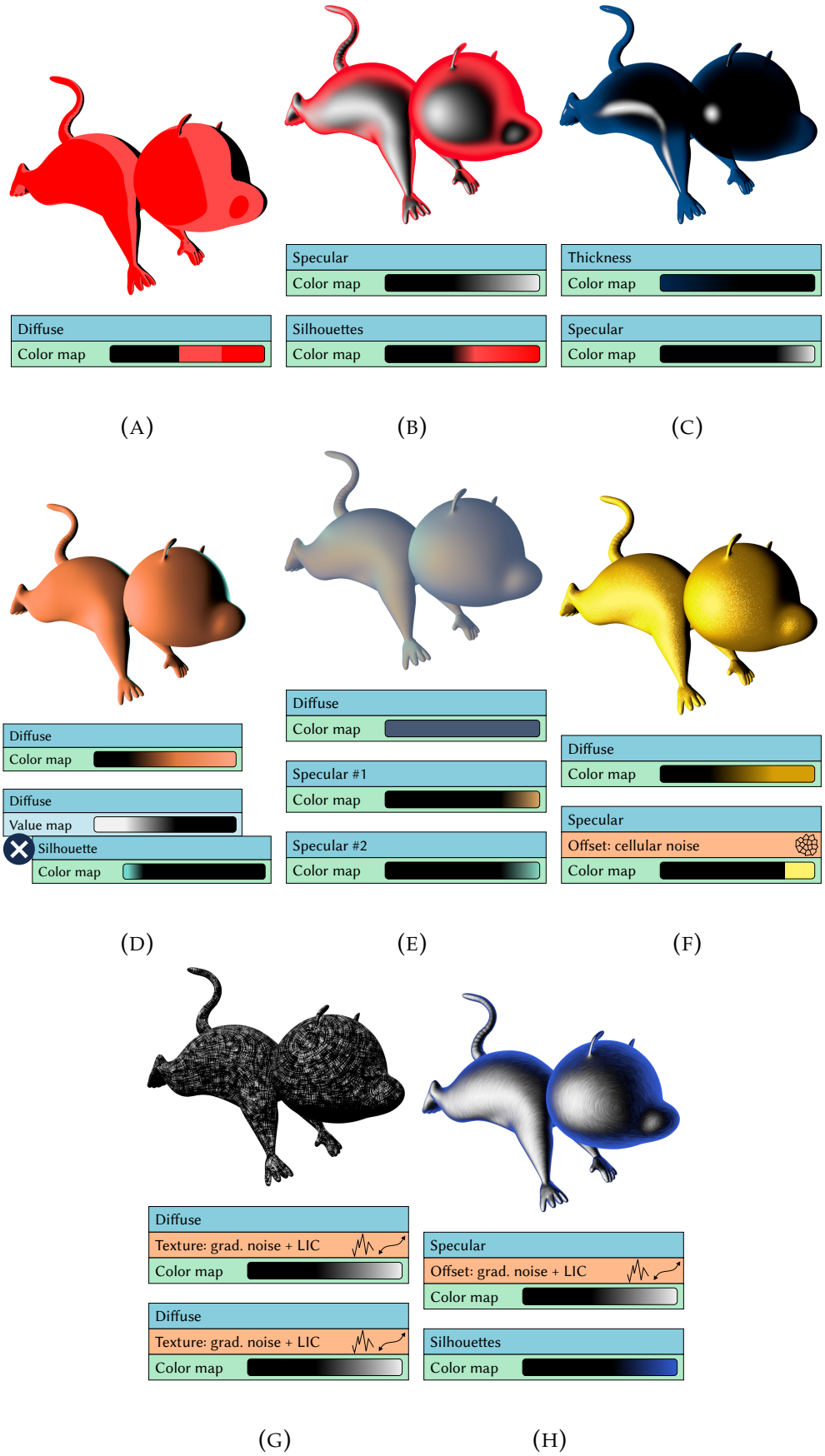


FIGURE 2.15: Various styles obtained with our system, and corresponding layer decompositions. See Section 2.8 for details.

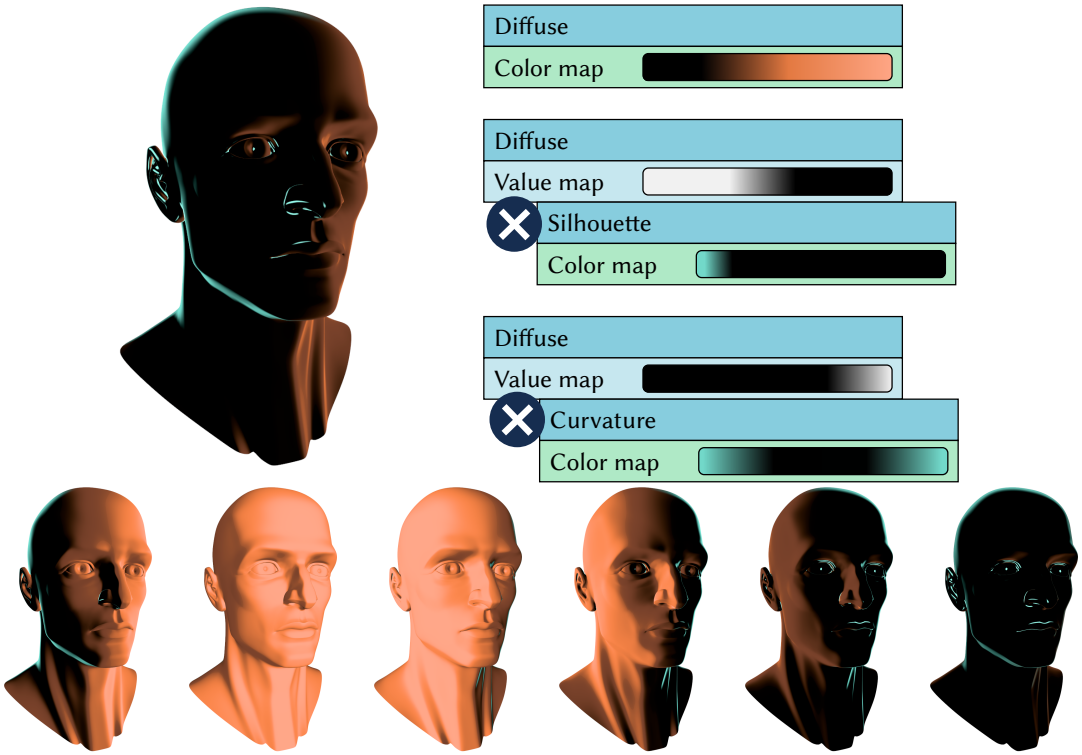


FIGURE 2.16: Our system allows complex spatially and temporally coherent behaviors with varying light directions. Here, curved regions and silhouettes are enhanced with a bluish color when the diffuse component gets darker.

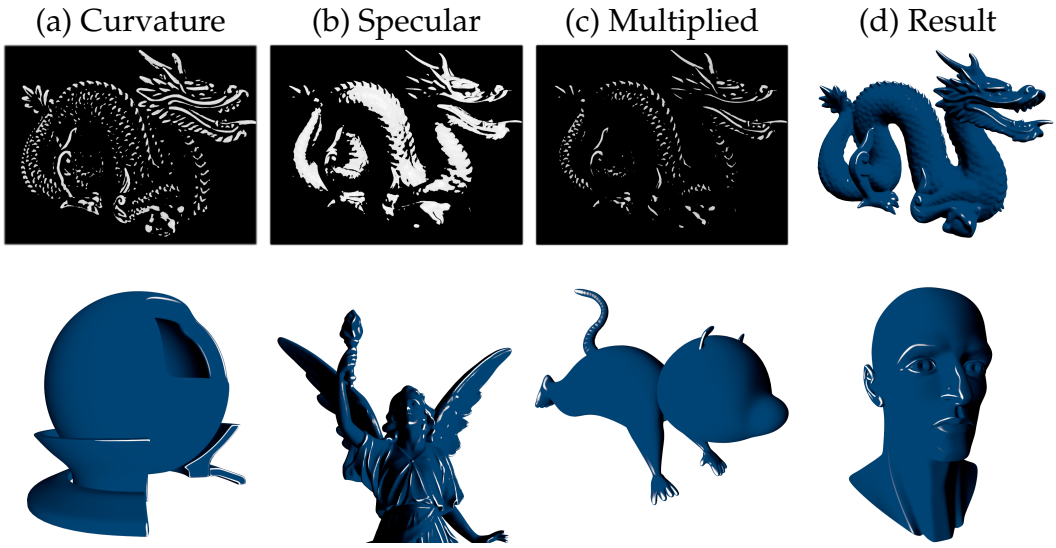


FIGURE 2.17: Combining parameterizations: user-defined curvature (a) and specular (b) behaviors are multiplied together (c) in order to ensure that the highlights appear only on highly curved regions (d). The corresponding style is then directly transferred to other input objects, as shown in the bottom row.

Chapter 3

Motion coherent stylization with marks in post-processing

3.1 Introduction

In both traditional and digital painting, the artist creates the picture by placing and accumulating *marks* on the canvas. These marks can be, among other things, pencil lines that depict contours, or brush strokes that fill the inside of objects. Similarly to the way artists depict light and shade, the distribution, shape and size of marks on the canvas, vary widely from artist to artist. This is a very versatile process in terms of the achievable appearances: in some styles, the marks are clearly visible on the final result, with their boundaries clearly defined. In others, the marks blend continuously, blurring the boundaries and making them indistinguishable from each other, resulting in a smoother appearance. Because of that, marks are essential components in the definition of an artistic style. Some styles with prominent and easily recognizable brush stroke techniques are shown in Figure 3.1.

Note that these marks fundamentally live in the *secondary space*, i.e. the 2D picture plane. Notably, from a computer graphics point of view, this means that such marks should not be affected by perspective transformation, foreshortening, or zooming of the camera: they should maintain a constant size on the screen regardless of where the camera is and how the surface is oriented in 3D. Because of that, such stylization marks should preferably be drawn as 2D primitives in screen-space. However, stylizing animated 3D scenes this way is challenging, because of the mismatch between the *flatness* of the marks (their 2D appearance and behavior on the picture plane), and the underlying 3D geometry and motion of the scene. As we've seen in Section 1.3, this mismatch is the source of various *temporal coherence* artifacts, detailed in the report by Bénard, Bousseau, and Thollot [BBT11]. As a result, all stylization methods that strive to reproduce 2D marks are tradeoffs between preserving flatness and avoiding temporal coherence artifacts.

Numerous stylized rendering techniques have been proposed to simulate the visual appearance of traditional painting marks, forming the field of *painterly rendering*. A straightforward approach is *stroke-based rendering*, which simulates the painting process by placing and rendering many small discrete textured primitives representing the painting marks. However, they are directly affected by temporal continuity issues. Also, the individual stroke

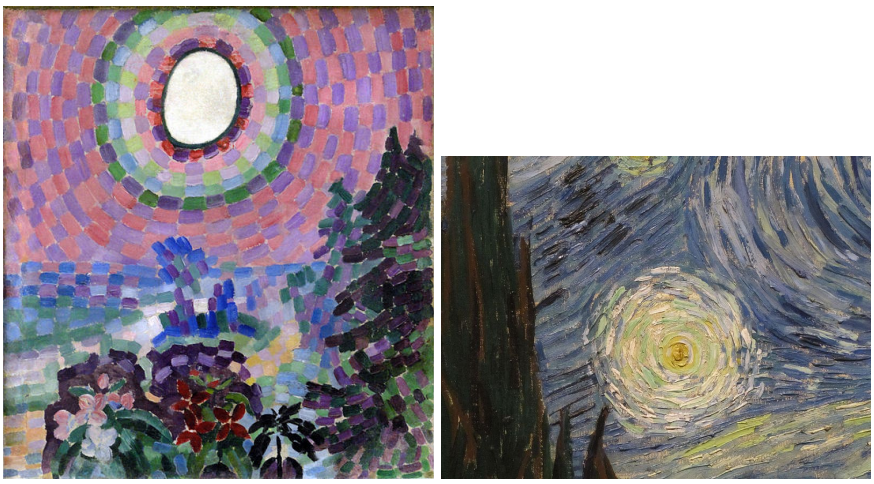


FIGURE 3.1: Importance of strokes in defining and recognizing an artistic style in traditional media: individual paint strokes are highly regular, and clearly distinguisable in the pointillist painting by Delaunay (left). In contrast, in this detail from Van Gogh's *The Starry Night* (right), the brush strokes are much less regular: they vary in size and shape. In both cases, the orientation of the strokes does not follow the same rules in every part of the image: some strokes seem to be oriented in the direction of light propagation (of the sun on the left, and of the stars on the right), while others conform to tangible objects in the scene. In the context of animated 3D scenes, controlling the distribution and attributes of strokes on the screen without introducting *temporal coherence* artifacts is one of the main challenges of the NPR community.

primitives should be drawn in a consistent order to avoid flickering, which is a non-trivial problem.

Promising advancements in the rendering of stylization marks have been made with techniques based on dynamic textures [BBT09] and procedural noise [Bén+10; KP11]. However, with these models the range of achievable appearances tends to be more limited. In the case of procedural noise models, it is also more difficult to generate highly structured marks.

Meanwhile, good stylization results have been obtained for images and videos with local image filtering techniques. Local image filters can be evaluated independently for each pixel, and depend on a fixed-size neighborhood (the *filter window*) around the pixel on the original image, as illustrated in Figure 3.2. Common examples of filters used for visual abstraction and simplification of pictures are morphological operators (erosion, dilation), and bilaterial filtering [TM98]. Furthermore, image filters can be guided to follow local image structure (such as contours or luminance edges) using flow-based filtering techniques. And, in combination with procedural noise, they can be used to reproduce brush-like appearances [Sem+16]. Finally, since image filtering techniques fundamentally operate in the picture plane, the resulting appearances tend to have a convincing 2D look.

FIGURE 3.2: Local image filtering process (here: bilateral filter). A pixel in the output image depends only on a neighborhood of fixed size of the original image, centered on the pixel.

3.1.1 Stylization of 3D scenes with local image filters

In this work, we explore the use of guided local image filters, in combination with sources of visual detail (procedural noise, for instance), to reproduce stylization marks on 3D scenes, during the post-processing stage: i.e. when the scene has been rasterized into G-buffers. This is motivated by several reasons:

- First, we feel that image filtering techniques have not been widely explored in the context of the stylization of 3D scenes in real-time. Many post-processing techniques have been proposed for the extraction and enhancement of contours, for abstraction, or to reproduce various shading effects (e.g. ambient occlusion), but very few methods exist to generate coherent stylization marks at this stage.

- Image filtering techniques map very well to evaluation on a GPU, and real-time performance is now relatively easy to achieve. Furthermore, extensive literature exists on how to speed up the computation of image filters on GPU.

- Whereas stroke-based rendering lends itself to reproducing styles with distinguishable marks, techniques based on image filtering might a better fit for styles where the color of adjacent strokes can smoothly mix.

- From a practical point-of-view, post-processing techniques are usually favored by technical artists as they can be easily plugged into rendering pipelines, the majority of which are already using some form of deferred shading and readily expose the G-buffers of the scene. No modification of the original geometry is required. Stroke-based rendering techniques, in contrast, require significant modifications to the pipeline, and additional geometry passes.

3.1.2 Challenges and contribution

This set of advantages seems to make image filtering an attractive choice for stylization. It is also a good compromise for temporal coherence: with image filtering it is easy to maintain a flat appearance, yet, by guiding image filters with G-buffer data, it is possible to maintain some degree of motion coherence. However, this approach presents some challenges because of one desired property: being able to render style marks slightly outside the rasterized footprint of objects. The issue is that local image filters that are guided by geometric data from G-buffers are limited to filling the inside of the rasterized footprint of objects. This is a significant drawback, because the style marks tend to stop abruptly at silhouettes, creating a dissonance between the stylized interior of the object and the fact that we can still see the unaltered silhouette of the underlying geometry (see Figure 3.11).

Our core contribution is centered around allowing image filters to alter the perceived silhouette of an object in a way that stays globally coherent with the motion. We propose a set of screen-space operations to *inflate* the data contained in G-buffers with a controllable radius in order to have motion-coherent data outside the original footprint of the object. This extended data can then be used to guide local image filters. This combination of motion coherence and footprint extension requires a careful handling of internal silhouettes because different styles can overlap at these locations. The approach that we present here explicitly deals with this issue, avoiding visual artifacts.

To summarize, our contribution allows the creation of stylization pipelines for 3D scenes that:

- Use only screen-space data contained in G-buffers;
- Are fully evaluated on the GPU;
- Ensure strong motion coherence;
- Enable the design of various styles that extend beyond the rasterized footprint of the object.

This work has been published and presented at the Expressive 2018 conference [Blé+18].

Structure of this chapter First, an overview of existing mark-based stylization techniques is proposed in Section 3.2. In Sections 3.3 and 3.6, we present a typical structure of stylization pipeline based on a combination of local image filtering guided by G-buffers, and solid procedural noise. Through this example, we illustrate the issues that arise at object silhouettes when reproducing marks. In Sections 3.4 and 3.5, we detail our *G-buffer inflation* method and how we handled the difficult cases of internal silhouettes. Many local image filters can be adapted to work with our method: we show some examples of achievable styles in full pipelines in Section 3.7.

3.2 Related Work

Rendering a 3D scene in a way that is visually reminiscent of the brush strokes of an artist is a long standing problem in stylized rendering. This is usually referred to as *painterly rendering*. Painterly rendering techniques have been proposed for static images, videos and 3D scenes. For an overview of painterly rendering techniques, we refer the reader to the reviews of Bénard, Bousseau, and Thollot [BBT11] and Hegde, Gatzidis, and Tian [HGT13].

In this section, we explore painterly rendering techniques, with an emphasis on *region* stylization, under three different angles: first, techniques that achieve a painterly style by explicitly rendering discrete paint strokes sequentially on a canvas: these methods form the field of *stroke-based rendering*; then, techniques that emulate the impression of brush strokes with *texturing*; and finally, techniques where the marks are implicit and result from the application of *image processing* techniques.

3.2.1 Explicit marks: stroke-based rendering

Following the *Paint by Numbers* system of Haeberli [Hae90], considerable work has been done on stroke-based rendering from 2D images [Her98; GCS02; HE04; Zen+09; ZZ11]. The general principle behind these approaches has been summarized by Hertzmann [Her03]: it consists in "painting" color regions of an image by placing discrete marks (brush strokes, stipples, etc.) on a canvas, optimizing their size, length, and orientation according to some depiction goals. These methods have been extended to video by moving the strokes according to the optical flow [Lit97].

When 3D information is available, temporal coherence can be improved by anchoring the marks to the 3D geometry, as was first proposed by Meier [Mei96]: their system distributes particles on 3D surface that serve as anchors for drawing small, straight textured strokes in screen-space, sometimes called "splats". A particle-based approach is also employed by Curtis [Cur98] to produce sketchy animated drawings.

However, maintaining constant density and size of strokes in image-space requires adding or removing strokes along the animation, leading to temporal discontinuities. Most of the previous works in this category try to improve the continuity by carefully distributing the anchor points, and by blending strokes that appear or disappear in a meaningful way [Kow+99; Kal+02; Van+07; LSF10].

Stroke placement can also be guided by the user, as in the WYSIWYG NPR system of Kalnins et al. [Kal+02]: within their system, users annotate the 3D model with textured strokes to represent silhouettes, creases, or regions via hatching. When the viewpoint changes, the system automatically adapts the position, length and density of the strokes to maintain the original appearance, if necessary by synthesizing longer strokes from a smaller exemplar.

Strokes can also be placed manually by the artist in 3D, generalizing the concept of painting to the 3D space. This is the case of Disney's *Deep Canvas*

FIGURE 3.3: Appearance obtained using Meier's stroke-based painterly rendering system. The image is composed of many small textured splats anchored to an underlying geometry to preserve motion coherence. Image from [Mei96].

[KL03], and more recently, the *Overcoat* system [Sch+11]: Overcoat provides a 3D canvas around a proxy object inside which artists can embed arbitrary stroke paths. No automatic placement is made: an artist draws 2D strokes on a view of the object, which are then embedded in 3D space with an optimization process. The stroke themselves are rendered by distributing splats along the stroke paths. The system has been extended with skinning and keyframing-based animations of strokes [Bas+13]. Finally, the rendering of the strokes can be improved by using example-based texture synthesis instead of splats [Zhe+17].

The main drawback of stroke-based rendering techniques for animations is the difficulty of maintaining both a constant screen-space density of strokes (which is important for *flatness*), and minimizing the sudden changes in the distribution of the strokes (which is important for *temporal continuity*). Common issues with temporal continuity are the popping of the strokes as their anchor points become disoccluded, and sudden changes in the rendering order of strokes. For 3D painting approaches such as Deep Canvas and Overcoat, another thing to consider is the order of the strokes as painted by the artist, which presents a challenge for compositing [Bar+11]. However, one advantage with stroke-based rendering is that it is easy to alter the silhouette of an object, as strokes drawn in screen-space do not have to "stay inside" the rasterized footprint of the object on which they are anchored (only the anchor points need to).

In general, stroke-based approaches are well-suited for styles with clearly distinguishable strokes, but may be less adapted to styles where marks can smoothly blend between each other. Smoother appearances have been demonstrated in Overcoat, but this necessitates a high density of semi-transparent strokes: the amount of geometry required in this case can be costly. In real-time contexts such as video games, this prompted alternative approaches and approximations [Imh+15], or highly specialized GPU rendering pipelines [Eva15].

FIGURE 3.4: WYSIWYG NPR system of Kalnins et al. [Kal+02]. Users have annotated the contours, and shaded areas with textured strokes. The system ensures a consistent look when changing viewpoints and zooming. Images from [Kal+02]

FIGURE 3.5: 3D painting obtained with the Overcoat system, composed of approximately 5000 strokes. Strokes are painted manually by an artist on a 2D view, and automatically embedded by the system in a 3D space surrounding the proxy object. Result from [Sch+11].

3.2.2 Texture-based approaches

To avoid temporal discontinuities and decrease the cost of rendering additional geometry, stylization marks can instead be rendered into a texture beforehand, and then applied to an object by texture mapping. Texture mapping ensures a perfect motion coherence of the marks without temporal discontinuities. In the case where a parametrization of the surface is not available, *solid texturing* techniques can be employed, where the textures are defined as 3D volumes, and texture coordinates are derived from object-space vertex coordinates.

To generate those textures, procedural texturing techniques can be used. We refer the reader to the survey conducted by Lagae et al. [Lag+10] for a comprehensive overview of the different kinds of noise functions and their uses in computer graphics. Examples of widely used procedural models include Perlin Noise [Per85] or Cellular Noise [Wor96]. Most procedural

models are usable as 3D *solid noise* for use in solid texturing scenarios: see Lewis [Lew89] for an overview. Procedural models are attractive in computer graphics because of their ability to be evaluated on-the-fly, and their low memory cost compared to texture images. They are also *randomly accessible* [Lag+10], meaning that they can be easily evaluated at different locations in parallel, which is a necessary property for efficient GPU evaluation.

However, texture-mapped marks, procedural or not, maintain a strong 3D appearance, as they are affected by perspective distortion and foreshortening, and do not maintain a constant density on the screen when zooming. Texture mapping techniques specialized for NPR have been designed to create a flatter appearance. Klein et al. [Kle+00] proposed *art maps*, an adaptation of the classic mip-mapping technique for non-photorealistic appearances. During texturing, the mip-map level of a texture is selected based on projected size of a screen pixel on the surface. By putting marks at different scales in each mip-map level, they are able to maintain a constant mark size on the screen regardless of foreshortening. A similar approach has been used for real-time hatching with *tonal art maps* [Pra+01; Web+02]. The *Dynamic Solid Textures* of Bénard, Bousseau, and Thollot [BBT09] adapt this principle to solid textures with a *fractalisation* approach, in which the original solid texture is blended multiple times at doubling frequencies. They adjust these frequencies dynamically to allow for continuous "infinite zoom", keeping a similar texture appearance at multiple zoom levels.

Alternatively, texturing a scene in screen-space provides a good flat appearance, as the texture is not subject to perspective deformations. However, this is subject to the shower-door effect. Techniques have been propsosed to prevent this: Cunzi et al. [Cun+03] used a 2D similarity transform that matches the motion of the scene to animate the screen-space texture. Bousseau et al. [Bou+06] blended multiple layers of screen-space noise, each layer being anchored to a point on the 3D object.

However, while these approaches successfully avoid the shower-door effect, they are more suited to stochastic textures containing small-scale and random features (e.g. paper) but not adapted to large-scale structured features (e.g. brush strokes). Other methods are better suited for that: In Breslav et al. [Bre+07] approach, objects are textured in screen-space with small patterns anchored to an object, which are deformed from frame to frame to ensure a continuous flat appearance. Bénard et al. [Bén+10] proposed a specialized procedural noise model based on Gabor noise [Lag+09], and showed a wide variety of styles with a 2D appearance (Figure 3.6). Yet, textural details generated with only procedural noise still keep a very synthetic look.

Volumetric texturing Texturing can be extended outside the surface of an object by using 3D volumetric textures rendered with ray-marching [KK89; PH89]. Later works focused on improving the performance of this approach to real-time frame rates by approximating volumes with layers of additional geometry [MN98; Len+01; DN09]. We refer the reader to the survey of Koniaris et al. [Kon+14] for a comprehensive overview of this type of approaches.

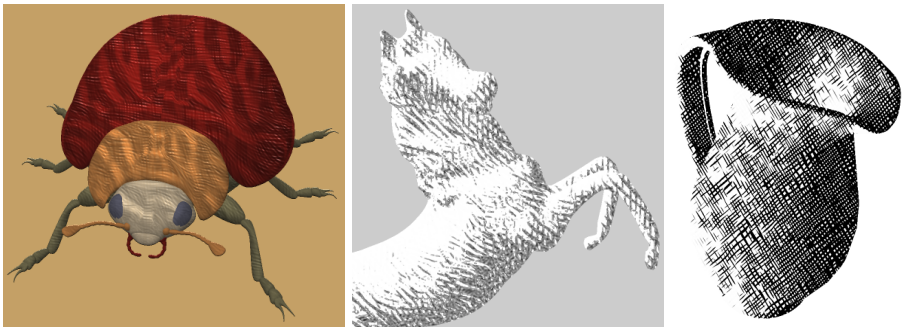


FIGURE 3.6: Appearances obtained with the NPR Gabor noise of Bénard et al. [Bén+10], reproducing painterly styles with marks. Images taken from the paper.

However, the goal of these methods is to be as close as possible to a realistic appearance, and are usually poorly suited to reproduce flat styles.

3.2.3 Image processing techniques

Image filtering techniques are ubiquitous and used in all domains dealing with images: medical imaging, computer vision, computer graphics, ... and for various purposes: denoising, detection of contours, visual simplification, etc. Originally, image processing techniques are made to work on classic color images. They can be used in the 3D rendering pipeline of 3D scenes in the post-processing stage. However, some image processing techniques are tailored for use in a post-processing context, and make use of the additional data contained in G-buffers (depth and normal information, notably). For an overview of image processing techniques in the context of the stylization of images and videos, we refer the reader to the survey of Kyprianidis et al. [Kyp+13]. We distinguish *global* and *local* image filtering techniques: *local* filtering techniques work on a per-pixel basis, each pixel of the output image depending only on a fixed pixel neighborhood in the original image (Figure 3.2), whereas global filtering techniques cannot be expressed efficiently this way. An example of global image algorithm is mean shift [CM02], which has been used in stylization to segment regions of similar color and texture for abstraction of images and videos [DS02; Bou+07]. Appearance transfer techniques derived from *image analogies* [Her+01] are other examples.

Here, we put emphasis on local image filtering techniques, as they can be implemented efficiently on a GPU. Examples of local image filtering techniques include morphological operations (used in the watercolor pipeline of Bousseau et al. [Bou+06]), and bilateral filtering [TM98]. Flow-guided filtering techniques, in which the filter kernels are oriented according to a 2D flow, have been used for abstraction [KD08; KLC09; KK11], and synthesis of paint strokes [Sem+16].

In this work, instead of abstracting details away from a picture, we seek to use local image filters to instead *add* stylization marks to a rendering of a 3D scene. In previous literature, elongated brush-like marks have been synthesized by filtering procedurally generated noise patterns with an anisotropic

image filter. Commonly used for this purpose is the Line Integral Convolution [CL93], which was originally proposed for the visualization of vector fields. This technique has been adapted to reproduce watercolor washes in texture space [LM01] and pencil drawings [MNI01; YMI04]. A variant of this technique has also been used to generate *glass patterns* [PP09], which have also been remarked for their painterly quality. Semmo et al. [Sem+16] used a flow-based Gaussian filter for synthesizing oil paint textures (Figure 3.7).

We follow this general principle of filtering noise to create marks and use it as a post-process pass, driven by geometrical data from G-buffers, to stylize 3D scenes coherently. To summarize, our system can be seen as a hybrid between a texture-space and an image-space approach: we use motion-coherent noise textures mapped on objects as a base for subsequent filtering in image-space, and also as a way to control certain aspects of the filters. This hybrid approach allows us to strike an advantageous balance between flatness and motion coherence. We show how they can be used to reproduce "digitally painted" styles, that range from visible discrete strokes to more continuous appearances.

FIGURE 3.7: Oil paint filtering of Semmo et al. Paint strokes were generated by blurring noise using a flow-guided Gaussian filter. Images from [Sem+16].

Post-processing techniques Even in the context of 3D scenes, local image filters are still widely used in the form of *post-processing* passes. Using *G-buffers* [ST90], post-processing filters also have access to geometry information in screen-space (sometimes called "2.5D" information, as it is 3D information stored in 2D images). G-buffer based methods have since been ported to GPU architectures, as first demonstrated by Nienhaus and Döllner [ND04a]. They are now widely used in the industry, as standard buffers in compositing software, and also in the rendering pipeline of many video games, in the form of deferred shading. Because of their simplicity, the fact that their complexity is independent of the input geometry, and the fact that they can be very efficiently implemented on GPUs, they are an attractive

alternative to object-space techniques in many scenarios such as contour extraction and stylization [ST90; Lee+07; Ver+11b], ambient occlusion, depth of field effects, etc.

A large number of NPR techniques make use of the geometry information in G-buffers. In addition, there exists hybrid methods that combine object-space processing and screen-space post-processing on G-buffers [TC00; ND04b; BWK05; MSR16]. Notably, Nienhaus and Döllner [ND04b] use an hybrid approach to render sketchy contour lines: in order to have motion-coherent noise to perturb the contour lines, which lie *outside* the rasterized footprint of the object, they first inflate the initial geometry, and render the inflated version textured with procedural noise.

Expanding the data contained in G-buffers is an essential part of our pipeline, in order to generate marks that extend outside the screen-space boundaries of the object. In contrast to the approach of Nienhaus and Döllner, our technique is purely screen-space, and does not require additional geometry passes: we expand the G-buffer data with an *inflation filter*.

3.3 Motivation and overview

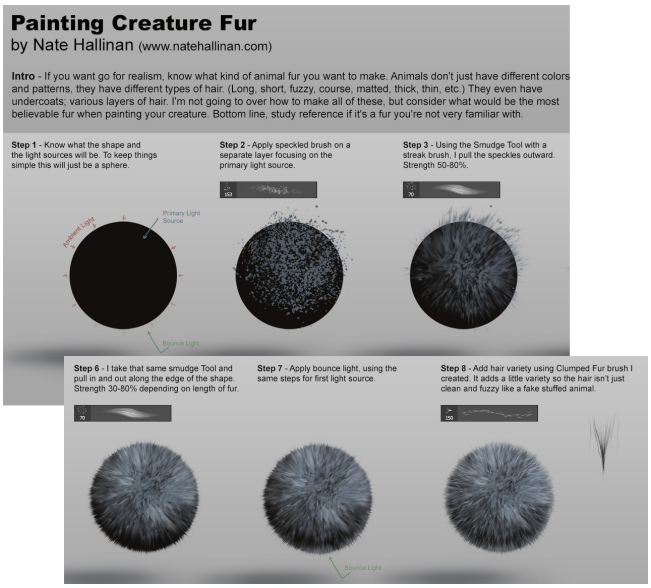


FIGURE 3.8: Tutorial on how to digitally paint fur: the artist starts by laying down splats of paint, then spreads it outwards the surface of the object (black sphere) with a *smudge* tool. This painting process is an inspiration for an automatic painterly stylization pipeline based on a combination of procedural noise (for the splats), and guided image filtering (for the smudging). © Nate Hallinan (Cropped version of https://www.deviantart.com/natehallinanart/art/Painting-Creature-Fur-423413686)

To illustrate the motivation behind the proposed pipeline, consider Figure 3.8, which is a tutorial on how to paint fur on an object with digital painting

tools. The artist starts by painting color splats with a brush tool. Then, using the *smudge* tool, the artist spreads the splats along the surface of the sphere, creating strands of fur. By repeating this process multiple times, a fur-like appearance can be obtained.

Our intuition is that it should be possible to reproduce this process with a combination of procedural texturing (for the initial paint splats) and an anisotropic image filter (for the smudging). To ensure a coherent motion of the strands of fur, the filter should be guided by a flow derived from the geometry data in G-buffers (e.g. the projected screen-space normals). As we've seen in the previous section, methods exist that can reproduce volumetric fur in real-time. But they tend to keep a 3D look at all times. In that aspect, image filters can more accurately model what the artist does with paint using the smudging tool.

An example of such a pipeline is summarized in Figure 3.9. First, spots of color are masked from a textured and shaded color input with a thresholded solid cellular noise, to ensure robust motion coherence. The solid noise can be generated from the object-space position G-buffer. Then, a convolution filter is applied, with an anisotropic spatial kernel of radius r, and locally rotated according to an input flow derived from G-buffers (here: the projected screen-space normals), to transform the spots into elongated paint strokes.

This particular pipeline is generalizable to other kinds of styles. In our work, we consider pipelines that have the same general structure and components as Figure 3.9:

- One or more source of motion coherent details (e.g. procedural noise) that serve as "seeds" for the marks: this role is fulfilled by solid noise in our previous example, but other techniques could be used [BBT09; Bén+10].
- Operations that combine the motion-coherent details with other scene attributes (e.g. shaded color).
- An image filter, usually guided by G-buffers, that synthesizes marks from the motion-coherent detail.

The only constraint that we impose on the image filter is locality: the result of a filter operation at a pixel should only depend on a local pixel neighborhood of radius r around the pixel in the input images, that we call the *filter window*. This allows the filters to be efficiently implemented on GPUs.

However, this pipeline has two main issues. First, the stylization stops abruptly at the silhouette of objects. We solve this by inflating the footprint of objects contained in G-buffers, in order to guide the filters in a controllable radius around the object. We describe this technique in Section 3.4. This, in turn, raises an issue when distinct surface parts are present under a filter neighborhood, because the inflation produces incoherent data. We explicitly deal with this in our technique by recognizing that, in those cases, the inflation and filtering steps should be done separately for each object. In Section 3.5, we explain the issue in more detail, and describe the local segmentation algorithm that we used to detect the offending cases and perform

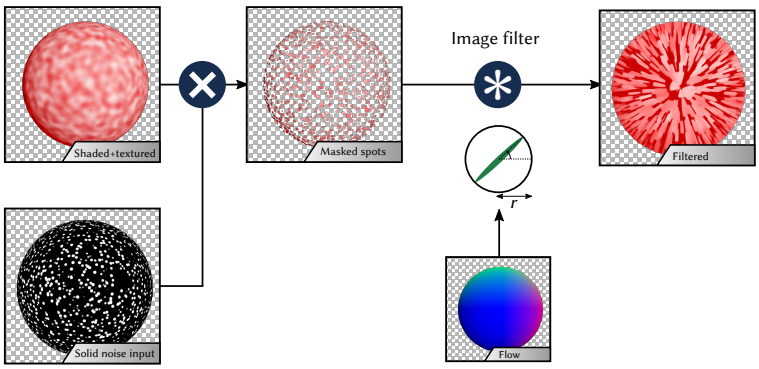


FIGURE 3.9: Schematic overview of a screen-space stylization pipeline to reproduce stylization marks with a combination of procedural solid noise and image filtering. First, spots on the shaded and textured input image are masked with cellular solid noise. Those masked spots are then filtered with a local image filter, guided by a 2D flow derived from the scene G-buffers. Using a filter with an elongated footprint, we are able to generate stroke-like features.

inflation separately. Finally, in Section 3.6, we cover the details of filter evaluation and blending, and show how to integrate image filters into our pipeline. Figure 3.10 provides an overview of the full pipeline.

3.4 Inflation

3.4.1 Problem overview

In the pipeline described in 3.9, the marks generated by the filters cannot go beyond the footprint of the object in the G-buffers (Figure 3.11(a)), as the data needed to guide the filter is not defined there. Yet, having irregular silhouettes contributes greatly to the perceived flatness of the look, and are a key aspect of some styles: for instance, a plausible painted fur should extend outside the silhouette of the underlying object.

Our first contribution is to fill the missing information by inflating the footprint of objects in G-buffers (Figure 3.11(b)). To that end, we propose a local image filter that performs inflation in screen-space, that is equivalent to inflating an object geometrically before rasterization by translating vertices along their normals. This equivalence is important to preserve the motion coherence of the extended G-buffer data. Once the necessary information for guiding the filter is available, the strokes can naturally extend outside the rasterized footprint of the object.

3.4.2 Filter formulation

We extend G-buffer data by mimicking a real inflation of the original 3D object, as if we displaced the vertices of the model along their associated normals before rasterization. Let us consider a pixel $\mathbf{p_0}$, being the center of a

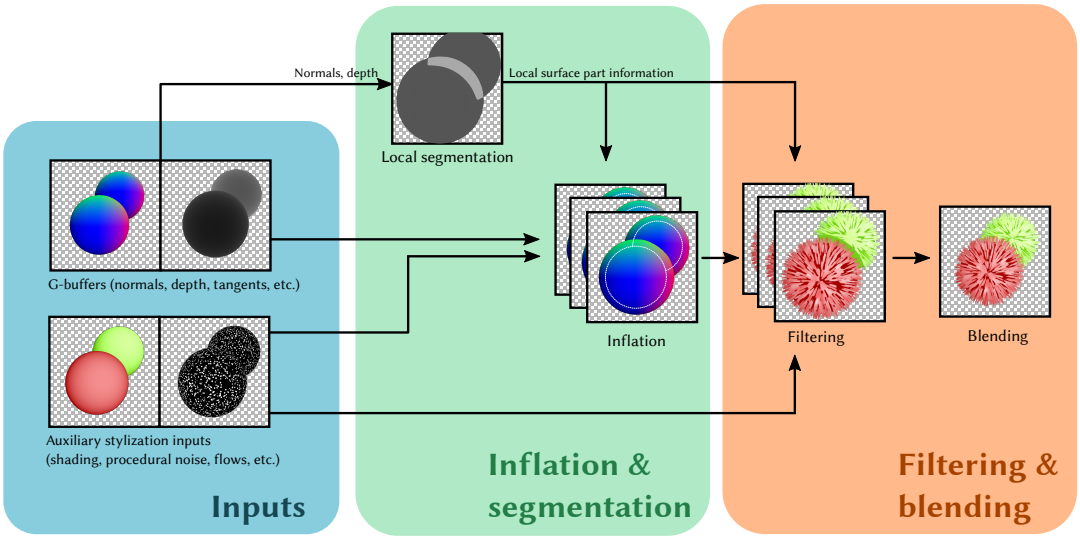


FIGURE 3.10: Our pipeline takes standard G-buffers (depth, normal, positions, etc.) and auxiliary buffers (scene shading, procedural noise, etc) as input. From the depth and normal buffers, a segmentation algorithm identifies the distinct surface parts under every filter neighborhood. The algorithm we employ is a soft K-means segmentation. The local information about surface parts is then passed on to the inflation pass, that generates inflated versions of all G-buffers and auxiliary inputs needed to drive the image filter. The local image filter is then evaluated separately for each detected surface part with the necessary input data (inflated G-buffer data, colors, etc) as guidance. Finally, the last step consists in blending the filter results at locations with stylization overlaps.

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

In practice, we work on discrete buffers, and this equation is usually not satisfied. Instead, we assign a weight $w_\mathbf{p}$ to each point under the filter window that evaluates the fitness of the point for inflation. We define it as a Gaussian of the distance between the warped pixel $\mathbf{p} + r\mathbf{n}(\mathbf{p})$ and the filter window center $\mathbf{p_0}$:

$$w_\mathbf{p} = \exp\left(\frac{-\|\mathbf{p_0}, \mathbf{p} + r\mathbf{n}(\mathbf{p})\|^2}{\sigma^2}\right), \tag{3.1}$$

where σ controls the tolerance towards the displacement error.

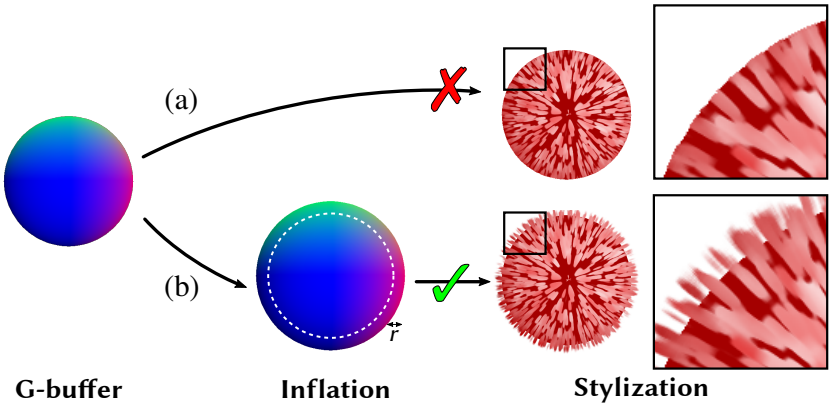


FIGURE 3.11: (a) The local image filter guided by G-buffer data cannot be evaluated outside the footprint of the object in the G-buffers: the generated marks stop abruptly at the contour. (b) We "inflate" the footprint of objects with the radius of the filter window r, in all G-buffers that are used by the filter, allowing the filter to be evaluated outside the original footprint, and the marks to go beyond the contours.

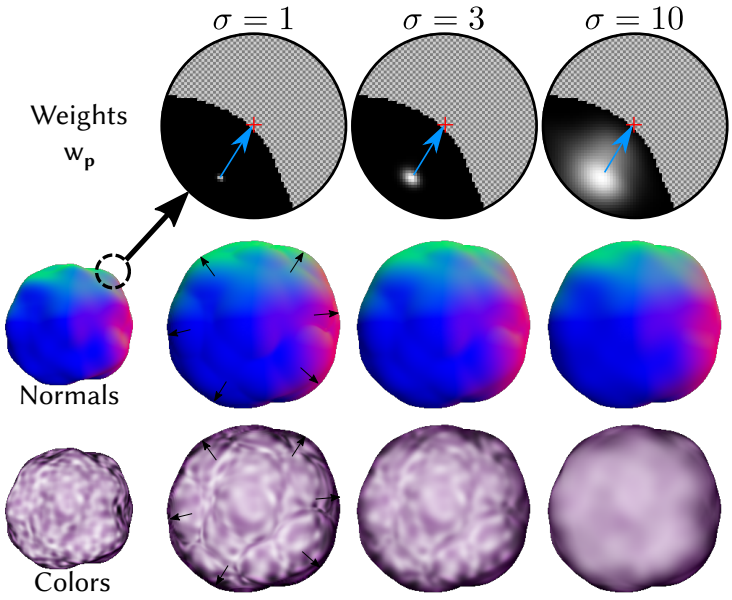


FIGURE 3.12: Inflation: (top) effect of σ on the size of the clusters, (middle) inflated normal map, (bottom) inflated color. Low σ values make the averaged region more localized, as few normals are considered fit for inflation. Conversely, higher σ values result in a wider region being averaged when inflating G-buffer data. This can lead to more stable animations, at the price of a loss of detail on high-frequency data, as shown with the noisy color image in the bottom row.

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

$$I_{\text{inflated}} = \frac{\sum_{\mathbf{p}\in S} w_\mathbf{p} I(\mathbf{p})}{\sum_{\mathbf{p}\in S} w_\mathbf{p}}, \tag{3.2}$$

where $I(\mathbf{p})$ represents the buffer data being inflated (depth, normals, colors, etc.). A visualization of weights and inflated data is provided in Figure 3.12.

3.5 Segmentation

3.5.1 Problem overview

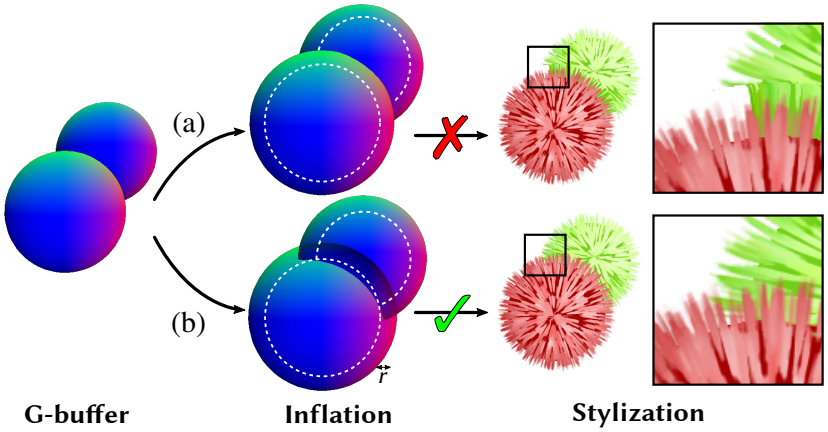


FIGURE 3.13: (a) When dealing with internal contours, a naive screen-space inflation fails where overlapping surfaces occur because a single surface is considered for each pixel. (b) Our approach uses a local segmentation in order to inflate and filter separately each overlapping surface part.

Inflation and filtering are ambiguous when parts of an object are overlapping each other (*e.g.* when there are internal contours). This is because inflation and filtering have no knowledge of multiple surface parts that may exist under a filter window.

This is illustrated with two spheres in Figure 3.13: visually, we would expect two sets of overlapping strokes in different directions (one for each sphere). The naive approach (in Figure 3.13(a)), keeps only one direction, which results in the green strokes not being properly oriented.

The main insight in our method is that in cases where multiple surface parts (separated by a contour) are present under a filter window, filtering should be done *separately* for each part. This means that, at these locations, the local image filter should be evaluated *multiple times per pixel*, with different guiding parameters for each detected surface part, and then blended together. The inflation should also be done separately, since each surface part may propagate different data.

A large portion of the proposed pipeline is dedicated to the detection and proper handling of these *stylization overlaps*. Our solution (Figure 3.13(b)) is to locally detect pixels affected by stylization overlaps by identifying and locating the different surface parts under their associated filter neighborhood. For that, we propose a local segmentation method, based on soft K-means [Bau15]. This information is handed over to the inflation and stylization passes so that they can inflate and filter each surface part independently.

3.5.2 Algorithm

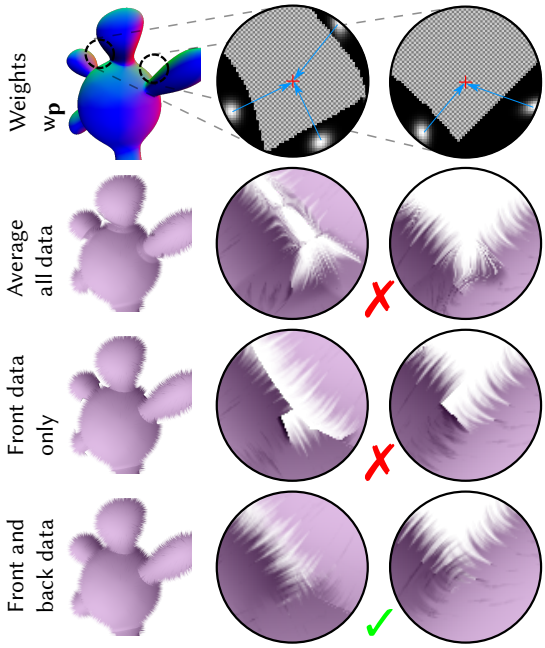


FIGURE 3.14: Handling overlapping inflated data on a style guided by screen-space normals. (Top) Normals and weights obtained on two filter windows. Multiple separate clusters of $w_{\mathbf{p}}$ are visible under windows with more than a single surface part. (Second) Averaging all data leads to unnatural direction fields. This is what happens by default with the naive inflation filter. (Third) averaging only front-most surfaces produces discontinuities in the resulting stylization. (Bottom) in our approach the weight maps are first segmented into clusters to stylize each part independently before compositing them.

When different parts of the surface are overlapping in the inflated version (due to internal contours, multiple silhouettes, T-junctions, etc.), the relation $\mathbf{p_0} = \mathbf{p} + r\mathbf{n}(\mathbf{p})$ can be satisfied by more than one anchor point $\mathbf{p}$ under the filter window. This is shown in Figure 3.14, where several clusters of weighted points stand out, and corresponds to the case where multiple surface parts are present under a filter window.

In that case, we would like Equation 3.2 to be applied independently on each cluster to ensure that each part is inflated separately. Figure 3.14 illustrates what happens with naive approaches, in contrast to our solution, on a style similar to the one shown in Figure 3.11, where an image filter is oriented along surface normals to "smudge" colors outside the silhouettes of the object. In the second row, the normals of all clusters are averaged together, resulting in an unnatural direction field for guiding the style.

In the third row, only the weights of the top-most surface are considered. As a unique direction is computed per pixel, only the front surface will be correctly stylized, producing discontinuities when styles of front and back surfaces should overlap. Our solution is presented in the last row, where

we locally segment and stylize each overlapping part independently before recomposing the final result.

Note that if the surface parts belonged to distinct objects in the scene, they could be separated by assigning different identifiers to objects and recovering them in a G-buffer. However, in the general case, internal silhouettes can appear even within the same object: this would require users of our system to split the geometry of objects in parts that do not exhibit internal silhouettes under any viewpoint. This is complex to do manually, and an automatic approach would still require an upstream pre-processing of the geometry of the scene.

Instead, our approach is to segment the clusters of w_p under a filter window. This classification has to be done locally under every filter window (i.e. for an input size of width W and height H, the classification must be done under $W \times H$ filter windows). Thus, in order to maintain interactive frame rates, the algorithm must not be too computationally expensive.

We chose the *weighted soft K-means* [Bau15] algorithm, as it is less computationally and memory-intensive than other classes of segmentation methods. It is also well suited to parallel GPU evaluation in a fragment shader. Note that K-means classifies input data into a fixed number of K clusters. However, there may be less than K surface parts under a filter window: to avoid over-segmentation in those cases, the *soft* variant of K-means was used, which allows overlapping clusters. As an alternative to soft K-means, clustering with a *Gaussian mixture model* was also tested, but did not significantly improve the quality of the resulting segmentation, and required more memory to store the additional shape parameters of the gaussians for each pixel. Mean shift was also considered, but was found to be too memory-intensive to be implemented inside a fragment shader.

Input data The data to be segmented takes the form $d(\mathbf{p}) = (x, y, z)$, where (x, y) is the offset of the pixel under the filter window and z is the depth of the surface at this position. Note that all components are remapped into the range $[-1; 1]$ so that they have the same impact on the segmentation. Each data point is weighted by $w_\mathbf{p}$ (Equation 3.1). Intuitively, the K-means will segment different clusters of weights (using $w_\mathbf{p}$) and thus clearly distinguish pixels that belong to different surface parts in a filter window. Adding z improves this segmentation for parts that contain the same normal with different depths.

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

$$\boldsymbol{\mu}_k^{(0)} = \left(0, 0, z_{\min} + k\frac{z_{\max} - z_{\min}}{K}\right), \tag{3.3}$$

where $\boldsymbol{\mu}_k^{(0)} \in \mathbb{R}^3, k \in [0, K-1]$ is the centroid of the cluster k at step 0 and $z_{\min/\max}$ are the minimal and maximal surface depths under the filter window. We found this simple and fast initialization to work well in most

cases, as overlapping inflated parts usually tend to have significantly different depths. We also experimented with distributing the initial clusters along the first principal component of the data points under the filter window, obtained with principal component analysis. However, we saw no discernible improvement over the simple initialization.

Cluster membership weights Each data point is soft-assigned to a cluster, meaning that each of them can partially belong to all clusters. The cluster membership weight of the data point at $\mathbf{p}$ to cluster k is noted $m_k(\mathbf{p})$, with $\sum_{k=0}^{K-1} m_k(\mathbf{p}) = 1$. It can be interpreted as the probability that $\mathbf{p}$ effectively belongs to cluster k. We define the membership weights $m_k(\mathbf{p})$ for the data point $d(\mathbf{p})$ as the softmax distance from the point to the cluster centroid, as suggested by Bauckhage [Bau15]:

$$m_k(\mathbf{p}) = \frac{e^{-\beta \|d(\mathbf{p}) - \boldsymbol{\mu}_k\|^2}}{\sum_{k=0}^{K-1} e^{-\beta \|d(\mathbf{p}) - \boldsymbol{\mu}_k\|^2}}, \tag{3.4}$$

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

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

$$\boldsymbol{\mu}_k^{(t+1)} = \frac{\sum_{\mathbf{p} \in S} w_{\mathbf{p}} m_k^{(t)}(\mathbf{p}) d(\mathbf{p})}{\sum_{\mathbf{p} \in S} w_{\mathbf{p}} m_k^{(t)}(\mathbf{p})}. \tag{3.5}$$

We stop the process when a specified maximum number N of iterations is reached. We used $N = 8$ for all the examples shown in the paper. The result of the algorithm is a fixed K number of clusters represented by their centroid $\boldsymbol{\mu}_k = (\mu_{k,x}, \mu_{k,y}, \mu_{k,z})$, each representing a local surface part. We finally sort those clusters by depth, $\boldsymbol{\mu}_0$ being the closest to the camera, and $\boldsymbol{\mu}_{K-1}$ the farthest.

Figure 3.15 shows the result of our segmentation on three local windows. We used $K = 3$ in all the results shown in this paper as there is rarely more than 3 distinct surface parts under the filter window, but more clusters could be used if needed (especially when using very large neighborhoods). Thanks to the soft assignement used in the K-means algorithm, multiple clusters can overlap when the number of surface parts is less than K. This is shown in Figure 3.15(a). If all clusters are located at the same place, then all pixels equally belong to the three of them. The corresponding inflation on a normal map is shown on the right, where we α-blend the resulting normals when multiple parts are overlapping.

Inflation Following the K-means, the inflated G-buffer data for cluster k can be calculated by modifying Equation 3.2 as follows:

$$I_{\text{inflated,k}} = \frac{\sum_{\mathbf{p}\in S} m_k(\mathbf{p}) w_{\mathbf{p}} I(\mathbf{p})}{\sum_{\mathbf{p}\in S} m_k(\mathbf{p}) w_{\mathbf{p}}}. \tag{3.6}$$

We now have, for each pixel, K inflation results, that correspond to at most K different surface parts.

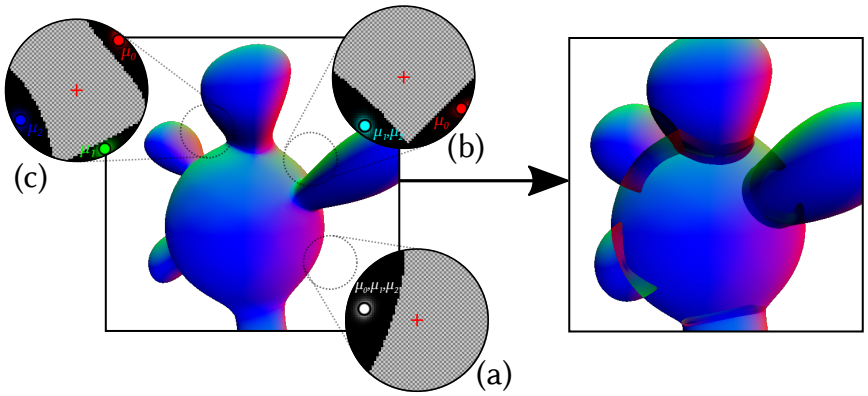


FIGURE 3.15: (Left) K-means segmentation results with $K = 3$. Under filter windows where only one or two distinct surface parts are present (a,b), the clustering associates multiple centroids to the same surface part. (Right) visualization of the inflated normal map. Regions with overlapping inflated normals are represented by blending them on top of each other.

3.6 Filtering and Blending

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

3.6.1 Assigning clusters to individual pixels

We want to apply stylization filters independently for each distinct surface part under the filter window. So we need to be able to mask out all pixels not belonging to the surface part being considered, as otherwise data belonging to different surface parts may be averaged together (*e.g.* the color of two different surfaces may bleed onto each other, which is not the desired result).

Our K-means segmentation locates anchor points on these surface parts, but the cluster membership weights m_k do not allow us to directly say whether a pixel belongs to a given surface part. Indeed, as seen in Figure 3.12, the weights quickly decrease according to the distance between the pixel and the cluster centroid. For most pixels we have $w_{\mathbf{p}} \approx 0$, meaning that their normals are all pointing "away" from the window center $\mathbf{p_0}$. Still it does not mean that they do not belong to a surface part, it rather means than when

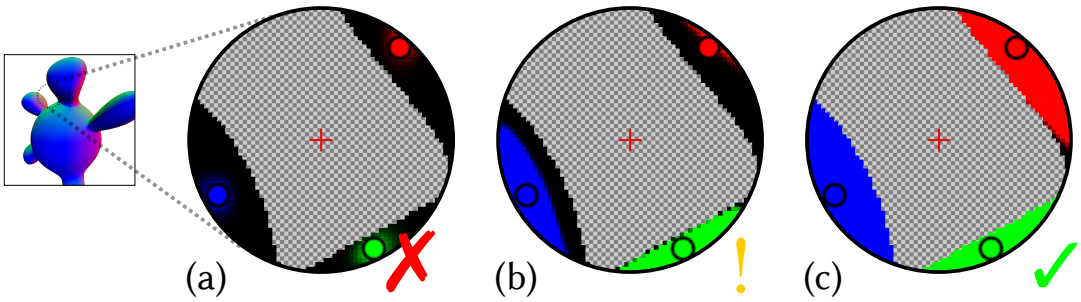


FIGURE 3.16: Extending the initial segmentation to all pixels under the filter window. Red: cluster 0; green: cluster 1; blue: cluster 2. (a) The initial K-means segmentation is only reliable for pixels with significant weights $w_{\mathbf{p}}$. (b) Prior to adjusting σ_n and σ_z to the current scene, the cluster assignments may be imprecise or incomplete (black parts are unclassified regions). (c) Adjusted σ_n to increase the tolerance to normal variations with respect to the normal at the cluster center. The distinct surface parts are properly detected and classified.

inflated they do not reach the window center. However, the cluster membership weights m_k cannot be relied upon for pixels with a low weight in the segmentation (*i.e.* $w_{\mathbf{p}} \approx 0$).

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

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

$$\Delta_{\mathrm{n,k}}(\mathbf{p}) = \exp\left(-\sigma_n \operatorname{acos}\left(\mathbf{n}(\boldsymbol{\mu}_k) \cdot \mathbf{n}(\mathbf{p})\right)\right), \tag{3.7}$$

$$\Delta_{\mathrm{z,k}}(\mathbf{p}) = \exp\left(-\sigma_z \left|z(\boldsymbol{\mu}_k) - z(\mathbf{p})\right|\right), \tag{3.8}$$

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

$$\Delta_{\mathrm{k}}(\mathbf{p}) = \min(\Delta_{\mathrm{n,k}}(\mathbf{p}), \Delta_{\mathrm{z,k}}(\mathbf{p})). \tag{3.9}$$

The σ_n and σ_z parameters control the tolerance of the classification to depth and normal differences, respectively. They are scene-wide parameters and can be adjusted by the user as needed. Note that one of σ_n and σ_z can be set to zero to disable the influence of the normal variations and depth variations, respectively. The membership of a pixel to cluster k, noted m'_k, is

obtained by thresholding the final weight Δ_k:

$$m'_k = \begin{cases} 0 & \text{if} \quad \Delta_k < 0.5 \\ 1 & \text{otherwise} \end{cases} \tag{3.10}$$

It is important for the filters that as much pixels as possible under the filter window are properly assigned to a matching surface part. It is up to the user to ensure that the surface parts are properly segmented by adjusting the σ_n and σ_z parameters. Figure 3.16 illustrates this process.

In Section 3.8, we discuss guidelines on the input geometry and filter window size to avoid over-segmentation. Note that, as with the K-means segmentation, the resulting clusters can overlap (a pixel under the filter window can belong to more than one cluster), and thus $\sum_{k=0}^{K-1} m'_k(\mathbf{p}) \neq 1$.

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

$$m'_{\text{residual}}(\mathbf{p}) = \max\left(0, 1 - \sum_k m'_k(\mathbf{p})\right). \tag{3.11}$$

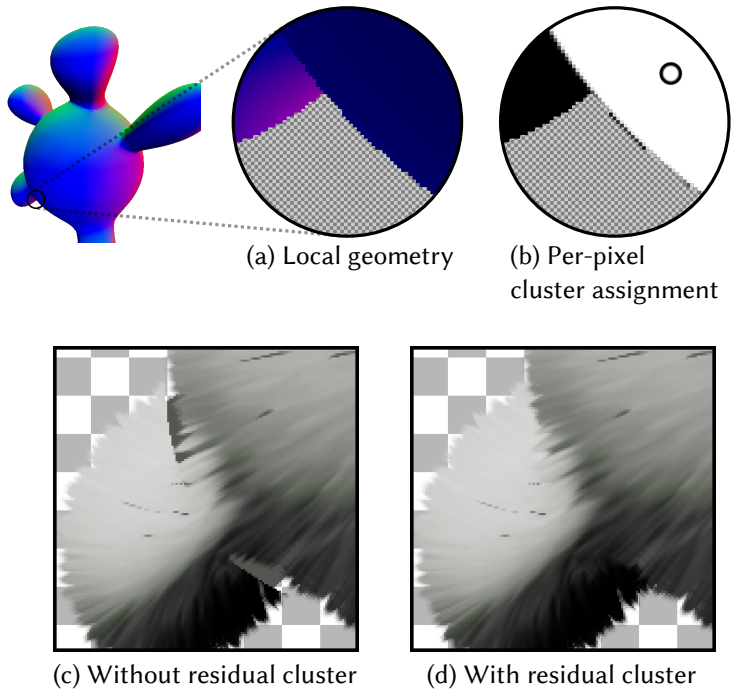
The residual cluster holds all surface parts that contain no anchor point under the filter window (often, because their anchor points are occluded). Thus, we cannot obtain coherent data for those surfaces by inflation. Nevertheless, extended G-buffer data for the residual cluster, required by the filters,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

$$I_{\text{inflated,residual}}(\mathbf{p}) = \frac{\sum_{\mathbf{p}\in S} m'_{\text{residual}}(\mathbf{p}) I(\mathbf{p})}{\sum_{\mathbf{p}\in S} m'_{\text{residual}}(\mathbf{p})}. \tag{3.12}$$

In practice, this approach properly fills stylization holes due to imprecise segmentation, as shown in Figure 3.17, and the lack of motion coherence is not easily noticeable.

3.6.2 Stylization filters

In this section, we describe several concrete examples of stylization filters that we use to generate the results presented in Section 3.7. We show how these filters can be made motion-coherent and silhouette-aware with relative ease inside our pipeline. The stylization filter is calculated separately for each surface part, meaning all pixels that do not belong to the surface part being considered are masked out during evaluation. The stylization filter is also calculated for the residual cluster, resulting in a total of $K + 1$ evaluations per



(a) Local geometry (b) Per-pixel cluster assignment

(c) Without residual cluster (d) With residual cluster

FIGURE 3.17: Clustering failures and importance of the residual cluster. Top row: (a) Local geometry under the offending filter neighborhood. (b) Clustering result and cluster assignment of the pixels. In black: pixels without an assigned cluster. The soft k-means segmentation missed a surface part (the leg) because the associated surface normals do not point towards the center of the filter window. This leads to pixels without an assigned cluster, that together form the *residual cluster*. Bottom row: (c) Not taking into account the residual cluster results in visible holes in the filtered result. (d) Our solution filters the residual cluster separately to fill the holes.

pixel. The final color is obtained by blending intermediate filter results in depth order.

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

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

All these buffers can be inflated when needed, making it possible to apply filters outside the original rasterized footprint of the object, with correct overlaps.

Line Integral Convolution filter A typical stylization filter is the Line Integral Convolution (LIC) [CL93] that can be used to produce fiber-like textures for fur or brush strokes. This class of filter has previously been used for stylization purposes [LM01; PP09].

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

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

$$C_k(\mathbf{p_0}) = \sum_{s=-L}^{L} \frac{f(s/L)m'_k(\boldsymbol{\kappa}(s))I^{\text{noise}}(\boldsymbol{\kappa}(s))I^{\text{color}}(\boldsymbol{\kappa}(s))}{f(s/L)}, \tag{3.13}$$

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

$$\boldsymbol{\kappa}(s) = \mathbf{p_0} + s \times I^{\text{flow}}_{\text{inflated,k}}(\mathbf{p_0}) \tag{3.14}$$

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

Modified Line Integral Convolution filter Due to the opacity modulation of the color input, the result of the LIC is semitransparent. Standard procedural texturing techniques can be used to process the opacity of the result in order to produce less washed-out strokes. Alternatively, we propose a minor variation in the normalization term in Formula 3.13 to fill the opacity gaps between the strokes, producing a more continuous painterly appearance:

$$C'_k(\mathbf{p_0}) = \sum_{s=-L}^{L} \frac{f(s/L)m'_k I^{\text{noise}}(\boldsymbol{\kappa}(s))I^{\text{color}}(\boldsymbol{\kappa}(s))}{f(s/L)\left((1-m'_k) + m'_k\, I^{\text{noise}}(\boldsymbol{\kappa}(s))\right)} \tag{3.15}$$

with $m'_k = m'_k(\boldsymbol{\kappa}(s))$. A comparison of the resulting appearances is shown in Figure 3.18.

LIC control The number of strokes and their weights can be controlled through the noise parameters. In practice, we used a thresholded cellular noise [Wor96] to reproduce most stroke-like appearances. This noise masks out round colored spots that serve as "seed points" for stroke generation.

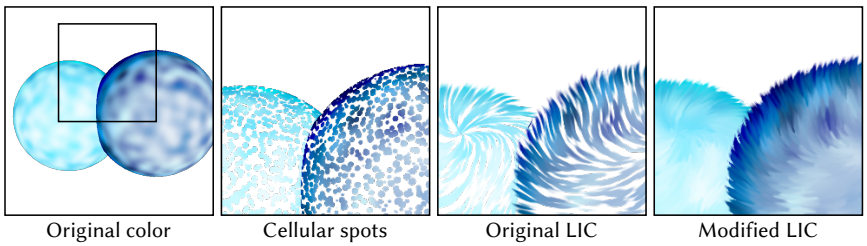


FIGURE 3.18: Comparison of the original and our modified formulation of the LIC filter.

Their size and smoothness are controlled through the thresholding parameters. The resulting aspect of the strokes can also be controlled through the integration profile function f. In combination with the modified LIC filter in Equation 3.15, we found that a gaussian integration profile with an user-adjustable width α (*i.e.* $f(x) = e^{-\frac{x^2}{\alpha^2}}$) provides intuitive control over the resulting appearance: low values of α tend to produce more clearly separated color flats, while increasing α results in a flatter profile that tends to blend the color of neighboring strokes together, resulting in a more continuous appearance.

Brush convolution filter The LIC filters previously described can be generalized by replacing the integration path by an arbitrary user-provided footprint. This is inspired by Implicit Brushes [Ver+11b], and sparse convolution noise [Lew84]. When used in tandem with thresholded cellular noise, the brush pattern is replicated on every cell, and can be used to reproduce a variety of different styles with proper handling of silhouettes. Examples of such styles are given in Figure 3.24. The result of those filters can be processed afterwards with standard procedural texturing techniques to refine the style (thresholding, color mapping, etc.).

3.6.3 Blending of intermediate filter results

The filter results for each individual cluster $C_k, k \in [0; K-1]$, and the filter result for the residual cluster, C_r, need to be composited together in depth order to get the final stylized result. The clusters $k \in [0; K-1]$ are sorted by depth. As the presence of a residual cluster is often due to occlusions by classes on top, we consider that the residual cluster is behind all other clusters during compositing. Thus, the final compositing order should be first C_{K-1} composited over C_r, then C_{K-2} composited over the previous result, and so on up to C_0.

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

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

We describe here for $K = 3$, but generalizable to higher values of K, an enumeration of all the possible cluster configurations using the middle cluster weight m_{μ_1} to distinguish between the four cases, shown in Figure 3.19. Let C_f be the final blended result,

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

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

- if $m_{\mu_1} \approx 1/2$ then m_{μ_1} is overlapping either m_{μ_0} or m_{μ_2}:
  - if $m_{\mu_0} < m_{\mu_2}$ then m_{μ_1} is likely overlapping m_{μ_0}, and m_{μ_2} is a distinct cluster:

$$C_f = \frac{m_{\mu_0} C_0 + m_{\mu_1} C_1}{m_{\mu_0} + m_{\mu_1}} \textbf{ over } C_2 \textbf{ over } C_r$$

  - if $m_{\mu_2} > m_{\mu_0}$ then m_{μ_1} is likely overlapping m_{μ_2}, and m_{μ_0} is a distinct cluster:

$$C_f = C_0 \textbf{ over } \frac{m_{\mu_1} C_1 + m_{\mu_2} C_2}{m_{\mu_1} + m_{\mu_2}} \textbf{ over } C_r$$

- finally, if $m_{\mu_1} \approx 1/3$ then all three clusters are likely overlapping:

$$C_f = \frac{m_{\mu_0} C_0 + m_{\mu_1} C_1 + m_{\mu_2} C_2}{m_{\mu_0} + m_{\mu_1} + m_{\mu_2}} \textbf{ over } C_r$$

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

3.7 Results

Figures 3.20, 3.22 and 3.23 present several styles using our line integral convolution, showing that it can create discrete stylization marks that extend outside the original object. These styles can be seen in motion, demonstrating the coherence of the marks, in the supplemental video of the associated

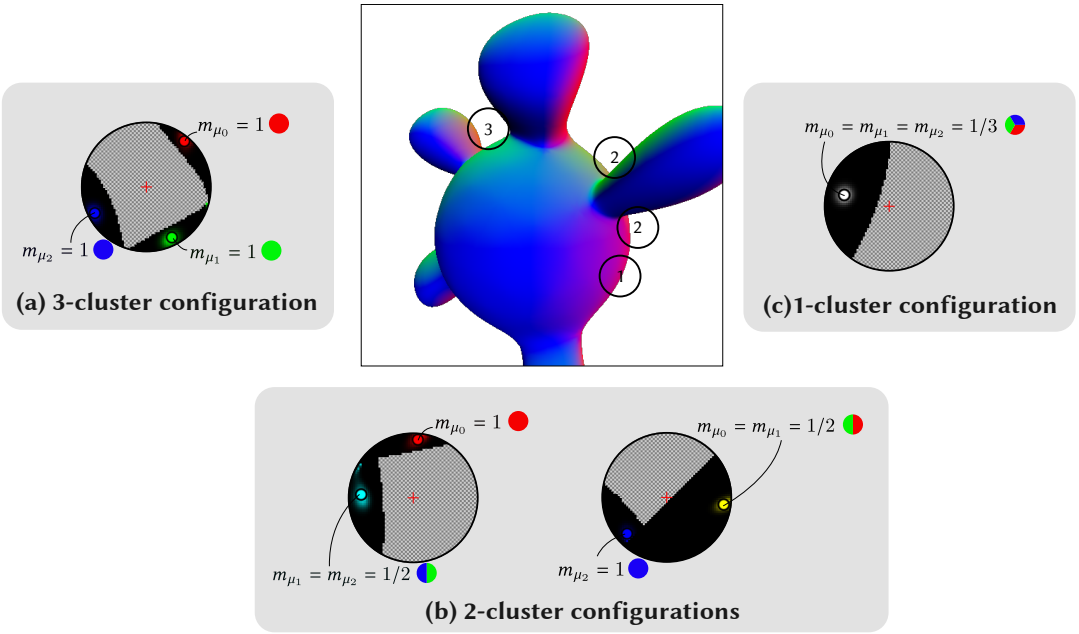


FIGURE 3.19: Contribution weights for $K = 3$ under 3-cluster (a), 2-cluster (b) and single cluster (c) configurations. Clusters 0,1 and 2 associated colors are respectively red, green and blue. The contribution weight of the middle cluster m_{μ_1} can be used to identify the current configuration.

paper [Blé+18]. Unless specified otherwise, all results presented in this section have been made with $N = 8$ K-means iterations, and with $\sigma = 1$.

In Figure 3.23(a), a LIC is directed by a constant flow, whereas in Figure 3.20, solid gradient noise is used to locally perturb the orientation of the directing flow (surface tangents). This produces a wavier appearance. Auxiliary maps that drive the stylization filters can also vary over time, as shown in Figure 3.22 where a gradient noise is animated to evoke a fur moved by the wind.

Figures 3.23(b,c,d,f,g) illustrate various painterly appearances obtained with our modified LIC filter. In Figure 3.23(d), the filter is directed by the inflated surface tangents; whereas in Figures 3.23(c,f,g), the filter is directed by the inflated screen-space normals. These filters can be used in various ways to create novel styles: in Figure 3.23(b), we introduce perturbations in the directing flow, but also introduce color variations along the integration path, producing a fur-like effect.

More geometrical effects can also be emulated with image filters, such as the wobbled silhouette of Figure 3.23(e). Here, the integration profile f of the modified LIC filter is designed to simulate a displacement along the directing flow (normals). The length of the integration path, and thus the amount of displacement, is locally modulated by a low-frequency solid noise that has been inflated outside the original object footprint. In Figure 3.23(h) a low-frequency solid gradient noise is applied on the object, slightly inflated, then processed in the stylization step to extract isolines. The result is composited on top of the original object.

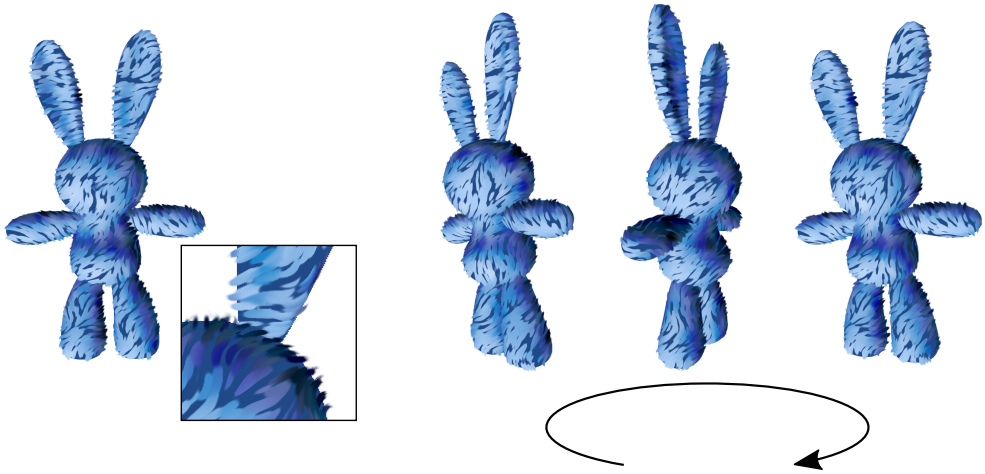


FIGURE 3.20: Using standard G-buffers and auxiliary buffers (noise, shading) as input, our pipeline can reproduce stylization effects that extend outside the original rasterized footprint of the object. Visual features produced by the filters stay coherent under motion or viewpoint changes. The complete post-processing pipeline to generate this style is shown in Figure 3.21.

Finally, Figure 3.24 shows several examples obtained with the brush convolution filter. The ability to choose any arbitrary brush footprint provides great artistic freedom.

3.7.1 Performance

Our pipeline was implemented as a set of (unoptimized) OpenGL fragment shaders in the Gratin software [VB15]. Its performance is mainly affected by the radius of the filter window as well as the target resolution. The stylization filter complexity also has an impact on performances. The following table provides the number of frames per second for varying radii, resolutions and style modes. These numbers were measured on the "bunny" object, on the styles in Figures 3.20 (LIC) and 3.24 (brush), using $K = 3$ and with a Nvidia GeForce GTX 1080 Ti graphics card.

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

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

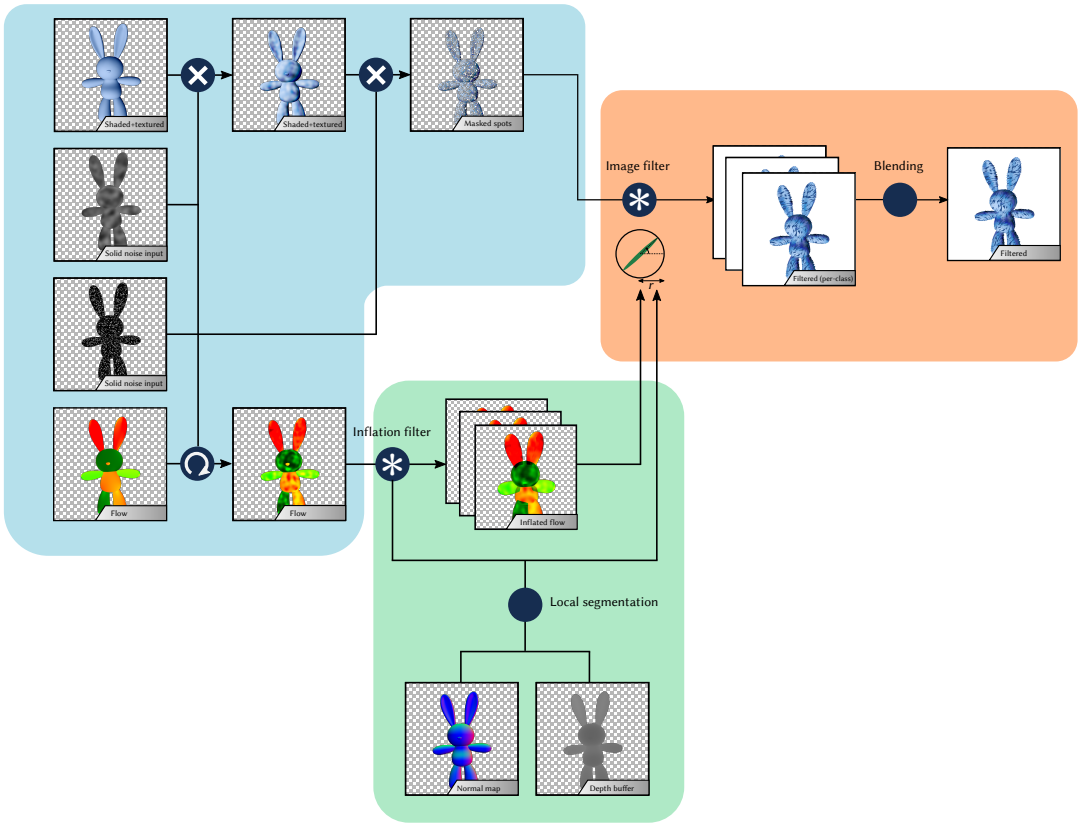


FIGURE 3.21: Complete post-processing pipeline to generate a painterly style. First, the G-buffers and auxiliary buffers (noise maps) are processed (blue region) to generate the inputs to the filtering stage (orange region). The flow guiding the filter is inflated (green region), using the results of the local segmentation of overlapping silhouettes.

3.8 Discussion

Segmentation quality Our local segmentation produces best results when the filter window contains enough data to cluster and with objects having rather smooth geometry. Local oversegmentation and per-pixel classification errors tend to occur in regions with large variations of depth or normal orientations, such as heavily slanted areas or regions with high curvature. In such cases, noticeable visual artifacts or holes may appear in the result. A good rule of thumb is to choose a filter window smaller than the smallest geometrical feature in the G-buffers but large enough to have enough data (*e.g.* $r > 10$). This, however, limits the size of the stylized features, and the amount of geometric detail in the original geometry. However, the impact of the latter is rather low, since the fine geometric features of highly detailed meshes would be lost anyway after applying the stylization filters, as shown in Figure 3.26.

Large filter windows are more likely to contain more overlapping inflated surface parts, and may require increasing K. As shown in Figure 3.27, if the effective number of distinct surfaces parts is higher than K, some of them will

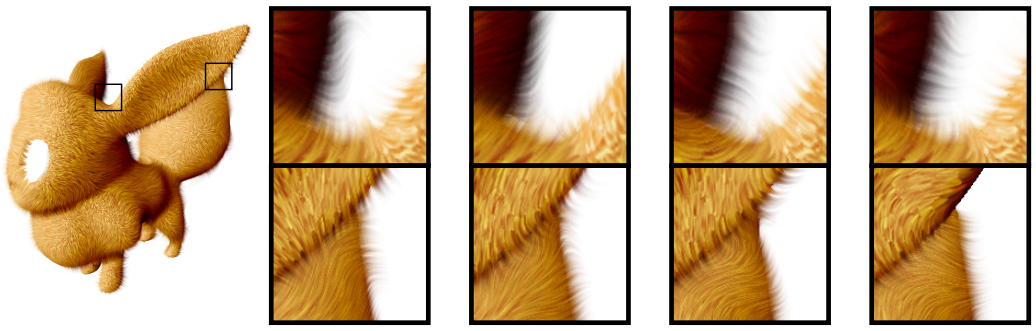


FIGURE 3.22: Random variations introduced by solid noise can be animated, such as in this example with a fur-like style. Segmentation parameters: $\beta = 15, \sigma_z = 20, \sigma_n = 2$

be grouped in the resulting clusters, which may lead to undesirable artifacts in the inflated buffers. By following the rule of thumb above and limiting the filter window size, we found that smooth objects with low geometrical detail rarely need more than $K = 3$ clusters for a correct detection of overlaps in inflation.

Finally, the segmentation is affected by the amplitude of depth discontinuities at silhouettes. It is sometimes necessary to increase the stiffness parameter β and/or adjust the per-pixel classification weights σ_z and σ_n to increase the sensitivity to depth differences, and, if necessary, increase the weight given to normal differences. Figure 3.25 shows visual artifacts that might appear in case of undersegmentation and oversegmentation, for varying values of σ_n. It shows a simple visual style consisting of marks directed by the surface normals. For high values of σ_n, the per-pixel segmentation may become too conservative and assign fringe pixels to the residual cluster. As the residual cluster spans unrelated surface parts, the corresponding inflated data blends two conflicting orientations, resulting in a distracting interaction at the boundary between the two spheres ($\sigma_n = 13$). Conversely, low values of σ_n or outright disabling the contribution of normals by setting $\sigma_n = 0$, can lead to undersegmentation: the surface parts are not properly segmented during filtering and the color of different surface parts may bleed onto each other ($\sigma_n = 0, \sigma_n = 1.5$).

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

Undetected surface parts and residual cluster Some surface parts under the filter window are missed by the K-means segmentation pass when no

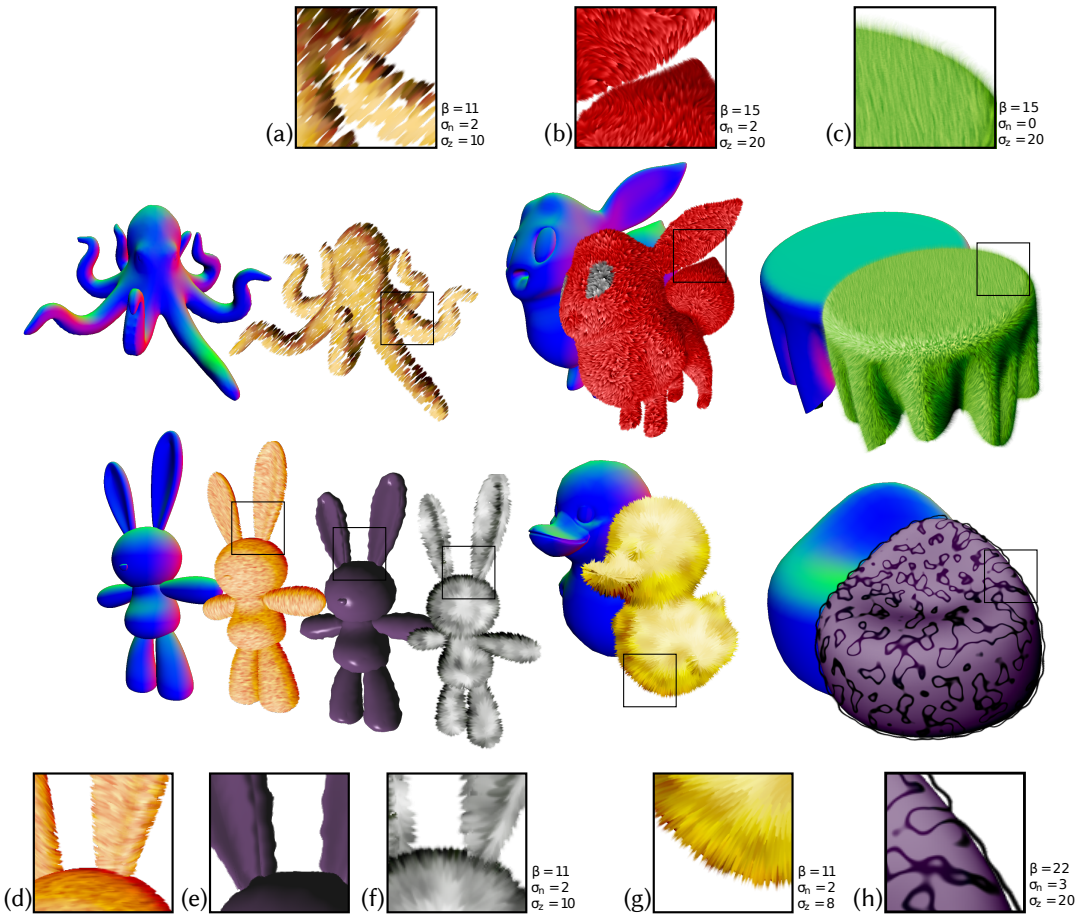


FIGURE 3.23: Various results obtained with LIC based filters. Model (b) by Sketchfab user Katerina Novakova (CC-BY-SA). Bunny model (d,e,f) by Sketchfab user Bernardo 3D (CC-BY). Octopus model by Lukáš Marek (CC-BY).

pixel belonging to the surface part has an adequate screen-space normal that displaces it towards the window center. In most cases, this happens because the corresponding anchor point for inflation is occluded by another surface. Significant complexity is needed during per-pixel segmentation to account for missed surface parts. Our solution, which is to manually separate all pixels not belonging to a detected cluster into a residual cluster, cannot accurately preserve motion coherence, as the extended data for this cluster is not obtained by inflation. More generally, it can be seen as a limitation of screen-space algorithms: information about occluded surfaces are lost, but is sometimes needed to get coherent data. However, this slight incoherence is usually not noticeable as it is often occluded by the stylization of other surface parts.

Disocclusions and aliasing A pervasive issue in stylized rendering techniques based on stroke primitives is how to handle disocclusions when a primitive becomes visible during camera or object motion. Our pipeline is not based on stroke primitives, but instead uses a combination of solid noise,

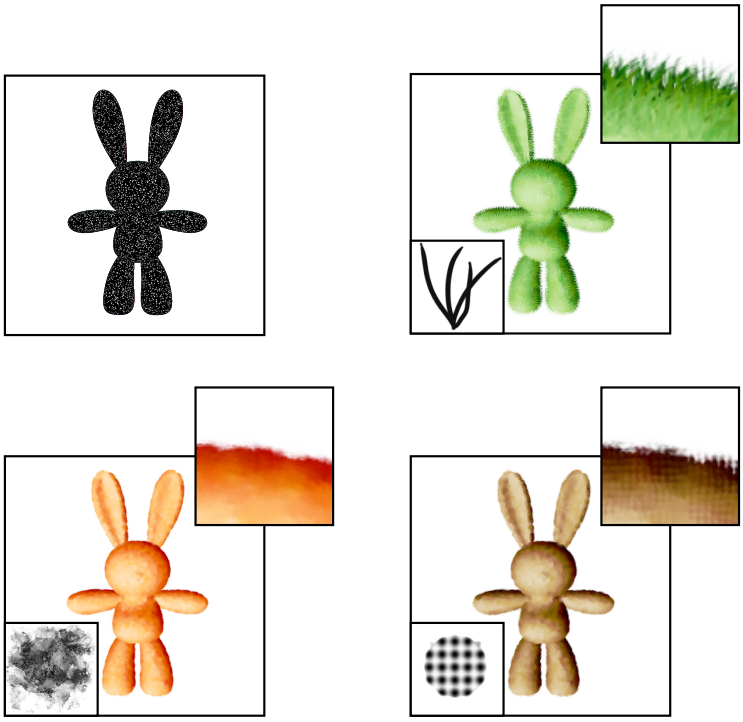


FIGURE 3.24: Styles obtained with the brush convolution filter over a thresholded cellular noise. Top left: cellular noise pattern before convolution. Top right, bottom row: result after convolution with the specified brush footprint. By varying the brush pattern, a variety of different appearances can be reproduced. In the "grass" example (top right), the brush patterns are oriented by the inflated screen-space normals. For these results $\beta = 11, \sigma_z = 10, \sigma_n = 2$.

2D filtering, and standard procedural texturing techniques to achieve a similar appearance. Yet, this process is also subject to visual artifacts due to disocclusions and aliasing. Depending on the stylization filter, noise features that become suddenly visible can introduce large changes in the results from one frame to another (popping). The same is true for aliasing artifacts in the noise, which tend to be amplified by the stylization filters. This could be improved as a future work with better sampling and filtering of the noise.

This effect can also be seen with aliasing due to rasterization (jagged edges). Increasing the resolution of the G-buffer can mitigate this, at the cost of speed and memory usage. Hardware anti-aliasing techniques such as multisample anti-aliasing could also be used, but would require modifications to the segmentation pass to work with multisampled textures. Another solution to alleviate various kinds of aliasing issues would be temporal supersampling. This is left as a possible future improvement to the pipeline.

Pipeline parameters In addition to the parameters of the stylization filters, which are out of the scope of our contribution, the user may have to adjust several parameters of the segmentation (σ, β, σ_n, σ_z, filter window radius r) to obtain an accurate result for a given object. This can be a tedious process as

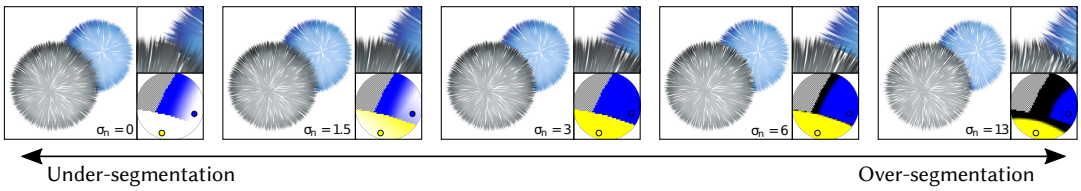


FIGURE 3.25: Influence of σ_n on the segmentation and filtered result. Pixels assigned to the residual class are shown in black in the classification result (lower-right windows). For these results $\beta = 11, \sigma_z = 10$.

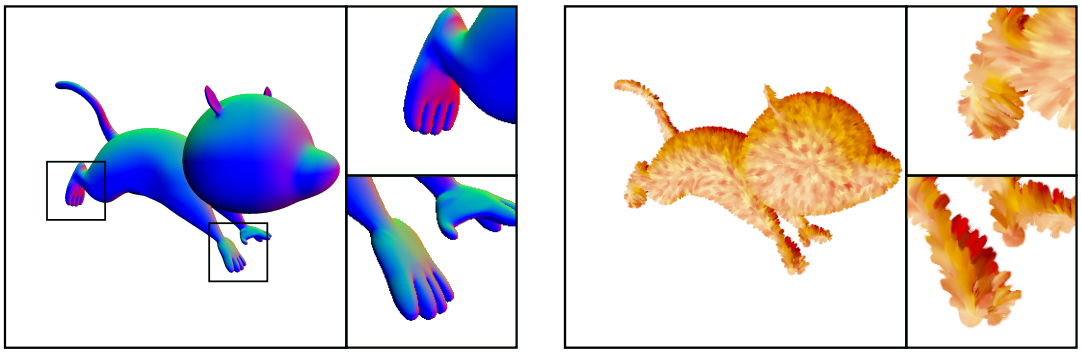


FIGURE 3.26: Geometry details finer than the filter size are lost during the stylization process, and can lead to visual artifacts as *K*-means does not handle the many small overlapping surface parts.

some segmentation failure cases may only appear under certain viewpoints. Currently, a visualization of the local segmentation results under a filter window is provided to ease this process, but an improvement would be to infer some of those parameters, either from the geometry if it is available, or from a global analysis of the contents of the G-buffers.

Comparison with object-space inflation Our approach does not require modification of the input geometry or the rendering pipeline. An alternative to our screen-space inflation algorithm could be done during the vertex processing stage, as in the technique of Nienhaus and Döllner [ND04b], and overlapping surface parts could be recovered using a *depth peeling* technique (storing multiple depths per pixel). Such an approach may be more efficient for high inflation radii at the cost of a more complex rendering pipeline. However our per-pixel classification stage would still be necessary in order to assign each pixel to a surface part within a filtering window.

3.9 Conclusion

We presented a post-processing stylization pipeline that allows stylization that extends outside object boundaries. Our main contribution in this paper is a post-processing technique to extend image filters that depend on G-buffer data outside the rasterized object boundaries in a motion coherent

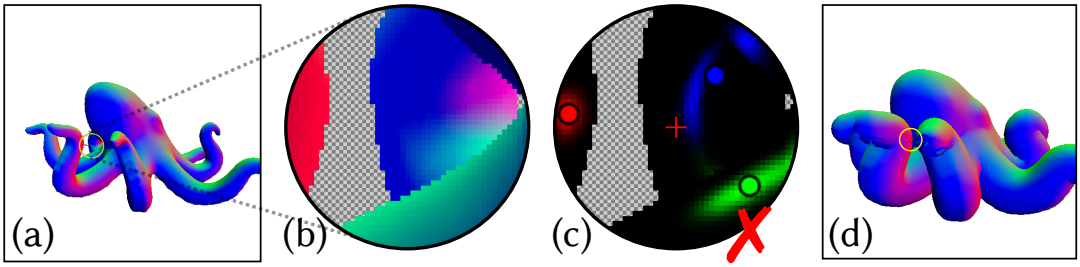


FIGURE 3.27: Segmentation failure for $K = 3$ clusters. Under the filter window, 4 distinct surface parts overlap when inflated. The K-means algorithm merged two unrelated surface parts in the same cluster (blue cluster), impacting the inflated result.

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

Chapter 4

Conclusion and Perspectives

We have presented two contributions in the field of stylization of 3D scenes: a tool to design and explore stylized shading models by composition of simple primitives, and a generic post-processing pipeline that uses image filtering techniques to stylize 3D scenes in a painterly style. In this chapter, we discuss what potential improvements on those two aspects (stylized depiction of light and shade, and painterly stylization) could be made in future work.

4.1 Shading design

We have presented a tool to design stylized shading models. We mainly targeted technical artists having previous experience in desiging shading models in 3D scenes. One of the goals of this tool was to facilitate the exploration of stylized depictions of light and shade, and avoid spending time writing shader code by hand. This, in turn, would allow artists to converge more quickly to a shading model that best fits their needs for a given 3D object or complete 3D scene.

4.1.1 User study

As a future work, we would like to conduct an actual user study on those artists, and measure the time saved compared to writing shader code by hand. A possible evaluation protocol would be to have artists reproduce a given shading style in a limited time using both approaches (directly writing shaders, and using our tool) and compare the time spent for each approach. It would also be interesting to compare our layered approach to shading design with the node-based shader editors implemented in many game engines and 3D modeling software. This would allow us to compare the efficiency of a representation in successive layers versus a more free-form node graph representation.

More than making the exploration of the design space of shading faster, one of our goals was also to enable the discovery of new shading models. To that end, we proposed ways to combine parametrizations to create novel shading behaviors but it is still unclear whether this “bottom-up” approach to shading design can lead to the discovery of new styles for 3D scenes. More people trying the tool are needed to conclusively answer this question.

4.1.2 Casual usage by non-technical users

It would also be interesting to perform a user study on non-technical users, outside the audience that we originally targeted. We believe, after having used the system for a while, that such a tool has potential uses for non-technical users, as an easily accessible appearance design tool. Such a study would provide clues as to what kind of interface and vocabulary would be necessary for less technically-skilled users. We expect that some work will have to be done on the shading primitives exposed to the users, and also the vocabulary employed: in this work, we focused on artists who have knowledge of shading terms employed in computer graphics (e.g. diffuse and specular components, rim-lighting terms, etc.), whereas non-technical users may not be familiar with this vocabulary. In practice, traditional illustrators employ quite different vocabulary and classifications when talking about light and shade: for instance, Hogarth [Hog91] proposed five different categories of light depiction: *single-source light*, *double-source light*, *diffused light*, *moonlight*, and *sculptural light*. This differs considerably from the primitives that we proposed. It might be interesting to propose higher-level "presets" that directly reproduce those shading types used in traditional illustration.

4.1.3 Extend the range of achievable appearances

This last point also raises the question of the range of appearances that we can reproduce with our system, especially when considering illustrative shading styles. For instance, in the so-called *sculptural lighting* style, each continuous part of a shape is shaded "as if the light falls on the center" (of the shape) [Hog91], independently of whether the shape is actually facing a light, or the camera: this allows revealing each continuous part of a shape with maximum clarity (Figure 4.1).

Currently, there is no way to capture the "center" of a shape with our parametrizations, and no way to accurately reproduce that kind of effect. Some techniques, such as *exaggerated shading* [RBD06], dyamically adjust the position of the light locally, in order to reveal shapes even in regions not facing the light (Figure 4.2). This may be better suited as a starting point to reproduce this particular shading style. As future work, we would like to integrate such techniques into our system as additional parametrizations.

Another interesting area of research that we would like to pursue is to find a way to more directly express some concepts of shading in 2D illustration into shading models for 3D objects. This also entails extending shading models with terms that only have meaning in screen-space (instead of only using local surface properties): for instance, to reproduce sculptural shading, propose a shading model that depends on the 2D center of shapes in screen-space.

FIGURE 4.1: Sculptural shading example by Burne Hogarth. This shading style reveals all parts of the shape, producing highlights centered on each continuous component of the shape. Image source: [Hog91]

4.1.4 Integration into existing modeling software

One way of having more users trying our tool is to integrate it into existing 3D creation toolsets. Commercial plugins for stylized shading design exist [1] [2] [3] [4] but usually target specific looks (toon shading, in particular). Our system could be easily implemented into existing rendering pipelines as a deferred shading pass. Also, while our system is currently implemented with multiple screen-space passes (one per layer) for historical reasons, it could be easily merged into a single pass: thus, as a future extension, it should be feasible to export the layers as one fragment shader that would be usable in game engines, for instance.

4.1.5 Inference from hand-painted input

Another axis of research that we would like to explore is the possibility of automatically inferring a shading model from a hand-painted input, in complement of the manual edition of layers. With stroke annotations painted by the user, the system would deduce on-the-fly the combination of input parametrizations that best fits the shading intent of the user.

In the general case, given a finished painting, this is difficult to do. An online approach is more suited to this case, as the user could progressively refine the result of the inference by painting additional strokes. Additional

[1] PSOFT Pencil+ https://www.psoft.co.jp/en/product/pencil/3dsmax/

[2] ToonKit http://cogumelosoftworks.com/index.php/toonkit/

[3] Maneki® http://maneki.sh/

[4] cebas finalToon™ 4.0 https://www.cebas.com/index.php?pid=productinfo&prd_id=192

(A) Diffuse shading (B) Exaggerated shading

FIGURE 4.2: Comparison between simple diffuse shading and exaggerated shading [RBD06]. Exaggerated shading can reveal shape even in regions not lit by the diffuse term, by locally adjusting the lights. Such a technique might be a good starting point for reproducing a variety of illustrative styles. Images taken from the supplemental material of [RBD06].

hints may be provided by the user besides strokes, e.g. in the form of constraints for the inference algorithm.

This would be similar to the style transfer method proposed by Fišer et al. [Fiš+16], although in their system, the relation between the real illumination of the scene and the painted depiction is captured globally on the whole image, and then synthesized with image analogies. In our case, the inference result would be an explicit shading model (i.e. a function of the local surface properties, viewpoint, and lights) and usable as such in real-time contexts. Such an approach would be especially powerful when combined with a fallback to the manual editing of layers, and possibly, with local control of the shading (e.g. through normal maps, or through perturbations): for instance, we could propose a painting interface that would let the user paint local tone variations on the surface of an object and automatically translate them into coherent local adjustments of the geometry or the normal map. This way, with additional local control, we could potentially cover both ends of the interaction spectrum, and it would be a significant step towards a comprehensive framework for stylized depiction of light and shade.

4.2 Rendering in a painterly appearance

Discrete stylization marks, in the form of brush strokes, seems to be a distinctive feature of so-called painterly artistic styles. The ability to distinguish individual brush strokes in the end result seems to reinforce the feeling of

looking at a hand-painted picture. It is thus an important feature to reproduce in stylized rendering.

In Chapter 3, we have presented a way to render painterly strokes that are coherent with the motion of the scene with a combination of procedural noise and local image filters guided by G-buffers. We saw that it is more complicated than it seems at first glance because of coherence issues at silhouettes. The core of our contribution is to propose a generic way to solve these issues, and enable the use of any kind of local image filter in that context, to reproduce painterly appearances.

As we've seen, this approach to rendering marks avoids some drawbacks of stroke-based rendering and might be more flexible in terms of achievable appearances: again, a thorough study with more users is required to confirm this. The protocol for such a study would be similar to the one for the shading design tool: first, present participants with a reference style to reproduce, and a set of stylization filters integrated within our pipeline; then, measure how efficiently the participants can achieve a desired look from a reference (time spent, and degree of satisfaction with the result).

4.2.1 User interface for desiging image filters

So far our work on this approach to stylization has mostly been under a technical angle: how to resolve the issues that appear when using image filters naively with G-buffers.

We've demonstrated that such a system can reproduce interesting painterly appearances, but the examples that we show have been hand-crafted with a combination of procedural noise and custom image filters that have limited artistic control. The brush convolution filter provides some artistic control, but we feel that this only scratches the surface of what is achievable with local image filters guided by G-buffers. Currently, an artist has to resort to the low-level implementation of the filters in shader code in order to try out new effects with complex dynamic behaviors: thus, our system is currently not suitable for fast experimentation with styles.

As with shading, a more efficient way of exploring this design space is needed. An area of research that is complementary to our contribution that we would like to explore is the artist-guided design of image filters: that is, provide an interactive tool to design local image filters, from the input of an artist, instead of having to program them manually.

In this regard, a first improvement of the current system would be to unify all filter parameters that affect the appearance under a common interface. To illustrate this, consider the style presented in Figure 3.21. Currently, the parameters controlling the appearance are split between the noise used to generate the "anchor points" (frequency, jitter, amplitude, thresholding parameters), the filter itself (filter radius, convolution profile), the flow used to guide the filter (either the screen-space normals, tangents, or a constant flow), and the noise used to perturb the latter (type, frequency, amplitude, etc.). All of these are specified in separate parts of the pipeline.

From a user interface point of view, it would be beneficial to abstract the internal structure of the pipeline and provide a unified interface to control the stylization process as a whole, with parameters that have an intuitive effect on the appearance. For this, it is possible to take inspiration from the parameters exposed in most digital painting software to control brush parameters: e.g. size of the splats, color, rotation, jitter, opacity falloff, and in our case, the length and density of the strokes on the surface.

More advanced behaviors, such as filters with dynamically varying footprints, could be also be controlled by the user, potentially through a programmable approach, abstracting away the details of filter evaluation and the handling of silhouettes. There again, it is possible to take inspiration from existing painting software that provides programmable control over brush appearance.

4.3 Digital painting case study

In this thesis, our approach to stylization has been mostly "bottom-up": providing low-level primitives that, through exploration of the design space, would eventually allow the reproduction of existing 2D illustrative styles. Still, we feel that there is also value in a "top-down" study of styles in 2D illustration, that consists in "reverse-engineering" existing styles from a 2D hand-painted reference. Many works on stylization followed this approach, by studying one particular technique, style, or media.

In this regard, the field of *digital painting* seems to be largely untapped. In the broad sense, digital painting comprises a set of algorithms, rendering techniques, and interaction techniques to make paintings on a 2D digital canvas. In itself, digital painting can be used in the same way than traditional media, and specialized tools exist that simulate traditional paint and provide a natural painting experience [5]. But, in general, digital painting software can provide a very low-level control over the painting process, and, at the same time, efficient shortcuts to paint large amount of detail, usually faster than traditional painting techniques.

An example of this is the way digital artists use custom brushes to quickly place patterns, and add details to paintings (e.g. leaves, grass, fur, water: see Figure 4.3). These brushes have a variety of adjustable parameters (such as size, rotation, jitter, pressure sensitivity, etc.). Some tools provide even more complex control over the brushes in the form of small programs [6]. These brushes can also be used to control the mixing of colors through *smudge* tools. The precise control available to artists has led to the creation of painting techniques specific to the digital medium. This, in turn, contributed to the development of digital painting as a recognizable style in itself.

[5] Rebelle https://www.escapemotions.com/products/rebelle/

[6] BLACK INK http://blackink.bleank.com/

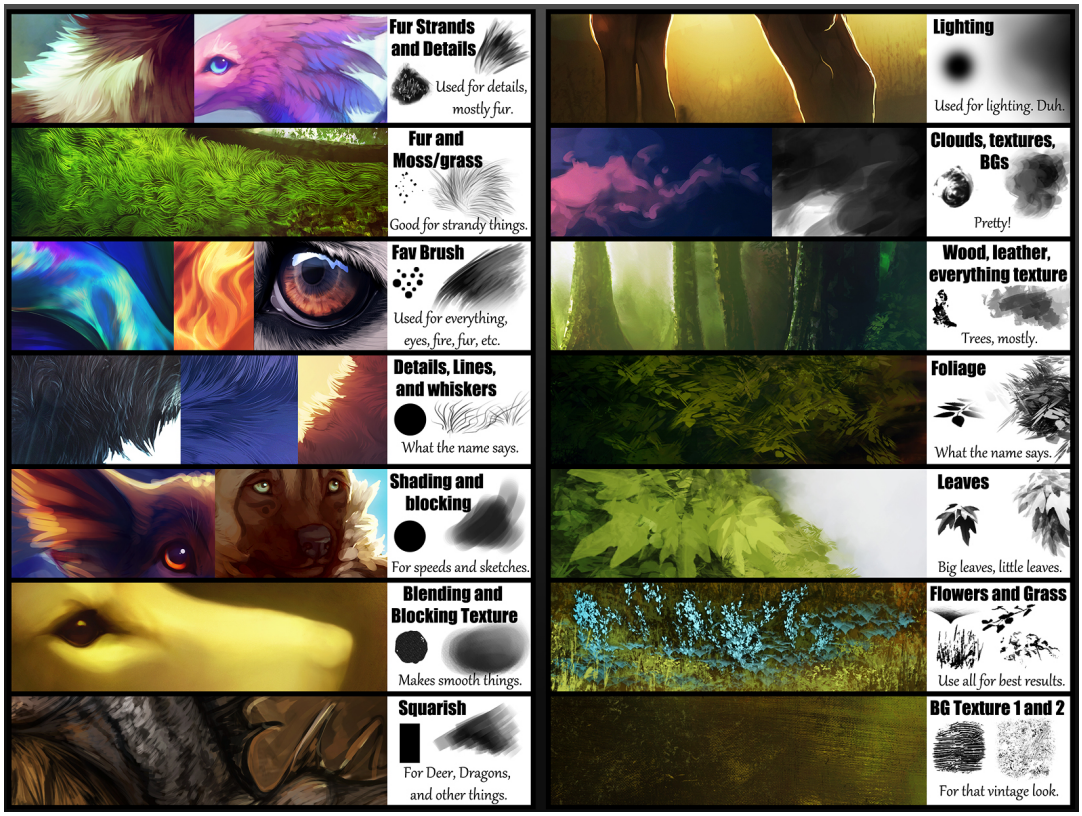


FIGURE 4.3: Examples of custom digital brushes used for various purposes. Cropped version of http://cobravenom.deviantart.com/art/My-Brush-Pack-565102840 by DeviantArt user CobraVenom.

An interesting aspect of digital painting is that the blending of paint on a canvas is still algorithmic in nature. While it is difficult to accurately reproduce the visual richness and nuances of paint in traditional media with conventional digital painting tools, digital painting has the advantage of being more directly expressible in terms of simple algorithms, without resorting to physical simulation. Conventional digital painting tools also share some vocabulary with computer graphics (e.g. the names of the different blending modes), which might lessen the distance between artists and researchers. All of this, in our opinion, makes digital painting a promising target of study, not only as a target for reproduction of traditional media, but also as a standalone media for artistic expression, with its own characteristic styles.

This axis of research is also motivated by the large amount of reference art available. This is due to several factors: the democratization of digital painting tools (such as graphics tablets) and thus a lower barrier to entry; more sophisticated software with advanced brush simulation models; and, importantly, the ability to easily share and publish digital art online.

The actual painting process is complex, but fortunately it is well documented by artists themselves. An exemple of this are material studies, such as the one shown in Figure 4.4, which are useful as reference points to compare our results to. Additionally, a wide corpus of step-by-step "tutorial"

paintings are available, and provide valuable insight on how artists naturally decompose the painting process. For instance, taking the example of Figure 4.5, we see that the painting of the fur is done with multiple layers of brush strokes with an uniform appearance, followed by a final color correction that accounts for lighting. Reproducing the final result in one go with a single image filter seems daunting. But reproducing each layer individually, following the artist-provided decomposition, is much more manageable. This could be implemented by complementing our layer-based shading design tool with a screen-space post-process of each layer to generate brush strokes, thus combining our two contributions into a more comprehensive system.

This is not a trivial addition, however, as shading and strokes are closely interlinked: our shading design tool would have to be extended to provide the local attributes of the filters, in addition to their color (for instance, the orientation of the strokes, their width, etc.). This is technically possible with the stylization pipeline presented in Chapter 3, but relies on the user programming its desired behavior by hand with shaders. In this regard, considerable work is needed to provide an intuitive interface for artists: for instance, as with shading, the formulas giving the stroke parameters could be inferred from a user-provided exemplar set of strokes.

This also raises the question of the placement of the strokes. For strokes that represent concrete, material details in the scene, such as hair or fur, the strokes should be anchored on the object. To begin, anchor points could be procedurally generated using solid noise, like we did, or even manually distributed by an artist if desired. But stroke placement is more complicated for "immaterial" features, such as highlights or contours, because we expect those features to not be anchored on the surface, but to move and deform in response to changes in the viewpoint. A large body of work has already been done on drawing stylized contours in a temporally coherent way. But single, long strokes are also used to depict shading features, such as the sharp, thin white highlights on the "blood" material in Figure 4.4, or the red highlights in the creases of the "cloth" material. Depending on the complexity of the shading model used to produce those highlights, it can be difficult to coherently place and draw those strokes across frames. For such effects, it might be more appropriate to use an image filtering approach that does not rely on anchor points.

To conclude, we feel that digital painting has recently grown into a full-fledged medium for artistic expression, as artists adopted digital painting tools into their workflow and started to create techniques specifically targeting this medium. The ability to easily manipulate and share digital media allowed the wide diffusion of those techniques, in the form of online tutorials and step-by-step paintings, which provide valuable insight into the artistic depiction process. Additionally, the digital nature of the media might make those techniques more easily translatable into rendering algorithms for the stylization of 3D scenes. For these reasons, we would like to make digital painting a basis for future work in new stylization techniques.

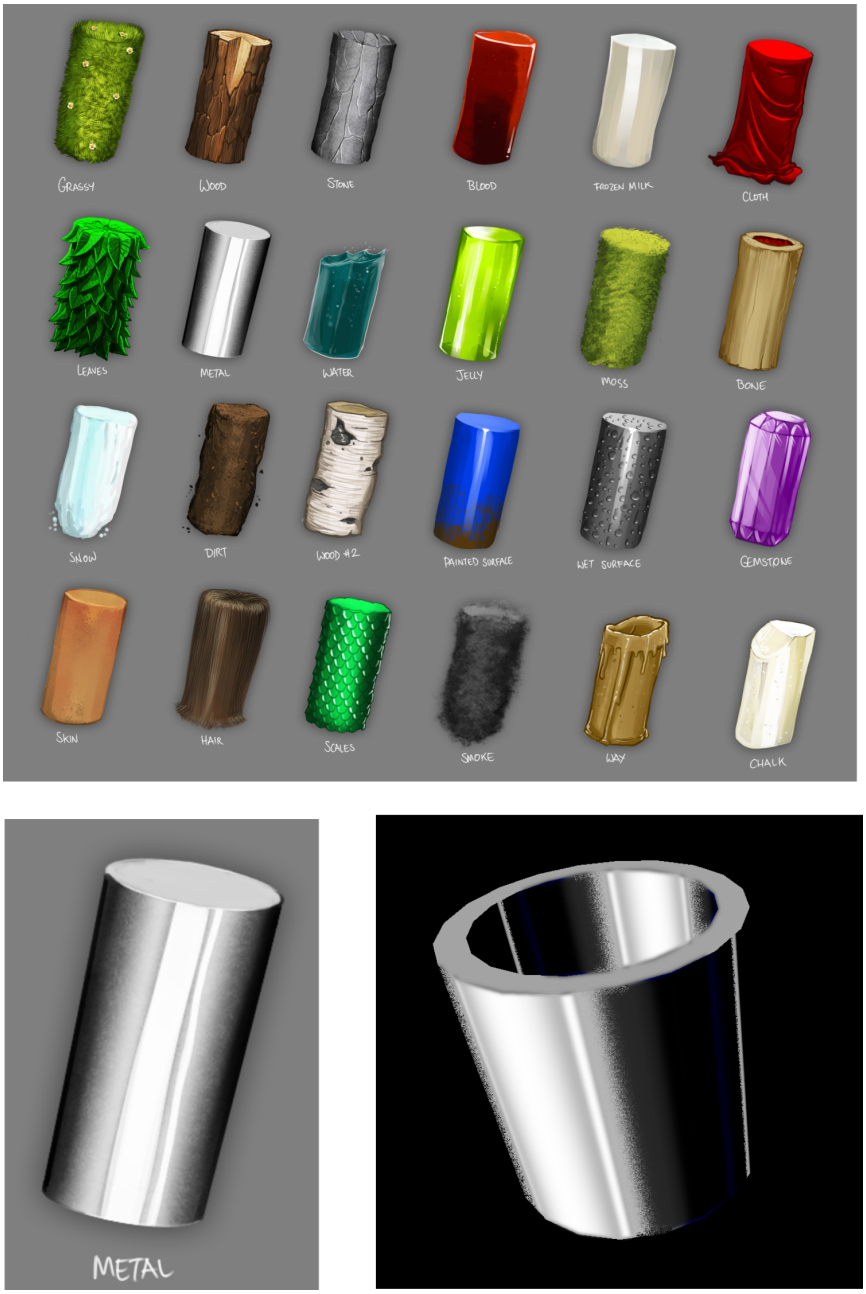


FIGURE 4.4: Top: digital painting study of various materials. Picture by *tudormorris* on Reddit (https://imgur.com/gc1HUJz). Bottom left: zoom-in on the "metal" material. Bottom right: our attempt at reproducing the "metal" material with our shading design tool.

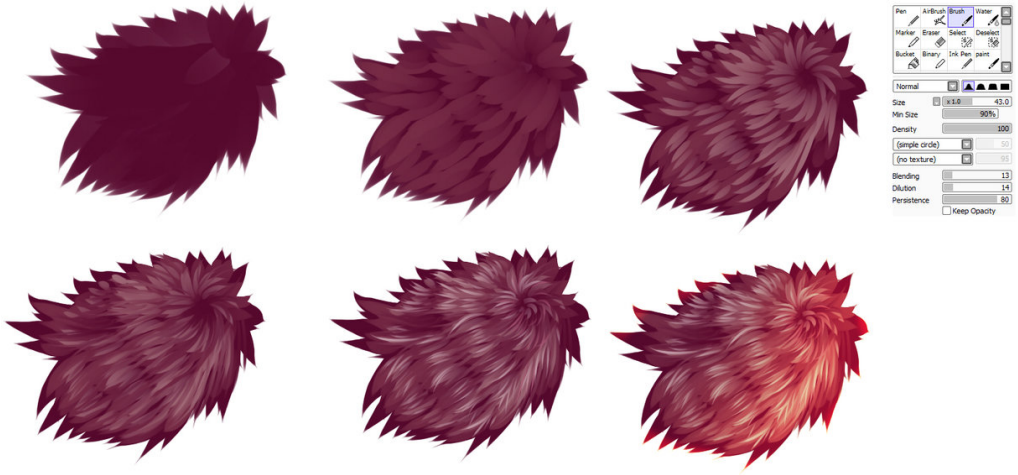


FIGURE 4.5: Step-by-step painting of fur. A large number of tutorial decompositions of digital painting techniques are made available by digital artists and could serve as a starting point for further study, in view of automatic reproduction of those styles on 3D objects. Picture by DeviantArt user ryky (https://www.deviantart.com/ryky/art/Easy-fur-tutorial-411937415)

Bibliography

[ALH16] Ergun Akleman, S Liu, and Donald House. "Barycentric Shaders: Art Directed Shading Using Control Images". In: *Proceedings of Expressive 2016*. 2016.

[AWB06] Ken-ichi Anjyo, Shuhei Wemler, and William Baxter. "Tweakable Light and Shade for Cartoon Animation". In: *Proceedings of the 4th International Symposium on Non-photorealistic Animation and Rendering*. New York, NY, USA: ACM, 2006, pp. 133–139.

[Bar+11] Ilya Baran, Johannes Schmid, Thomas Siegrist, Markus Gross, and Robert W. Sumner. "Mixed-order compositing for 3D paintings". In: *ACM Transactions on Graphics* (2011).

[Bas+13] Katie Bassett, Ilya Baran, Johannes Schmid, Markus Gross, and Robert W Sumner. "Authoring and Animating Painterly Characters". In: *ACM Trans. Graph.* 32.5 (Oct. 2013), 156:1–156:12.

[Bau15] Christian Bauckhage. *Lecture Notes on Data Science: Soft k-Means Clustering*. 2015.

[BBT09] Pierre Bénard, Adrien Bousseau, and Joëlle Thollot. "Dynamic Solid Textures for Real-Time Coherent Stylization". In: *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*. Boston, MA, Etats-Unis: ACM, Feb. 2009, pp. 121–127.

[BBT11] Pierre Bénard, Adrien Bousseau, and Joëlle Thollot. "State-of-the-Art Report on Temporal Coherence for Stylized Animations". In: *Computer Graphics Forum* 30.8 (Dec. 2011), pp. 2367–2386.

[Bén+10] Pierre Bénard, Ares Lagae, Peter Vangorp, Sylvain Lefebvre, George Drettakis, and Joëlle Thollot. "A Dynamic Noise Primitive for Coherent Stylization". In: *Computer Graphics Forum (Proceedings of the Eurographics Symposium on Rendering 2010)* 29.4 (June 2010), pp. 1497–1506.

[Bén+13] Pierre Bénard, Forrester Cole, Michael Kass, Igor Mordatch, James Hegarty, Martin Sebastian Senn, Kurt Fleischer, Davide Pesare, and Katherine Breeden. "Stylizing animation by example". In: *ACM Transactions on Graphics* 32.4 (2013).

[Blé+18] Alexandre Bléron, Romain Vergne, Thomas Hurtut, and Joëlle Thollot. "Motion-coherent stylization with screen-space image filters". In: *Expressive '18 - The Joint Symposium on Computational Aesthetics and Sketch Based Interfaces and Modeling and Non-Photorealistic Animation and Rendering*. Victoria, Canada, Aug. 2018.

[Bot09] Isaac Botkin. "Painting with polygons: non-photorealistic rendering using existing tools". In: *SIGGRAPH 2009: Talks*. ACM, 2009, p. 22.

[Bou+06] Adrien Bousseau, Matthew Kaplan, Joëlle Thollot, and François X Sillion. "Interactive watercolor rendering with temporal coherence and abstraction". In: *International Symposium on Non-Photorealistic Animation and Rendering (NPAR)*. Annecy, France: ACM, 2006.

[Bou+07] Adrien Bousseau, Fabrice Neyret, Joëlle Thollot, and David Salesin. "Video Watercolorization using Bidirectional Texture Advection". In: *ACM Transaction on Graphics (Proceedings of SIGGRAPH 2007)* 26.3 (2007).

[Bre+07] Simon Breslav, Karol Szerszen, Lee Markosian, Pascal Barla, and Joëlle Thollot. "Dynamic 2D Patterns for Shading 3D Scenes". In: *ACM SIGGRAPH 2007 Papers*. SIGGRAPH '07. New York, NY, USA: ACM, 2007.

[BTM06] Pascal Barla, Joëlle Thollot, and Lee Markosian. "X-Toon: An extended toon shader". In: *International Symposium on Non-Photorealistic Animation and Rendering (NPAR'06)*. Ed. by Douglas DeCarlo and Lee Markosian. Annecy, France: ACM, June 2006.

[BWK05] J Burgess, G Wyvill, and S A King. "A system for real-time watercolour rendering". In: *International 2005 Computer Graphics*. June 2005, pp. 234–240.

[CL93] Brian Cabral and Leith Casey Leedom. "Imaging Vector Fields Using Line Integral Convolution". In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 1993, pp. 263–270.

[CM02] Dorin Comaniciu and Peter Meer. "Mean shift: A robust approach toward feature space analysis". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2002).

[Coo84] Robert L. Cook. "Shade Trees". In: *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '84. New York, NY, USA: ACM, 1984, pp. 223–231.

[CPK06] Mark Colbert, Sumanta Pattanaik, and Jaroslav Krivanek. "BRDF-Shop: Creating physically correct bidirectional reflectance distribution functions". In: *IEEE Computer Graphics and Applications* 26.1 (2006), pp. 30–36.

[Cun+03] Matthieu Cunzi, Joëlle Thollot, Sylvain Paris, Gilles Debunne, Jean-Dominique Gascuel, and Frédo Durand. "Dynamic Canvas for Immersive Non-Photorealistic Walkthroughs". In: *Proc. Graphics Interface*. Halifax, Canada: A K Peters, LTD., 2003.

[Cur+97] Cassidy J. Curtis, Sean E. Anderson, Joshua E. Seims, Kurt W. Fleischer, and David H. Salesin. "Computer-generated watercolor". In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques - SIGGRAPH '97* (1997).

[Cur98] Cassidy J Curtis. "Loose and Sketchy Animation". In: *ACM SIGGRAPH 98 Electronic Art and Animation Catalog*. New York, NY, USA: ACM, 1998, pp. 145–.

[DN09] Philippe Decaudin and Fabrice Neyret. "Volumetric Billboards". In: *Computer Graphics Forum* (2009).

[DS02] Doug DeCarlo and Anthony Santella. "Stylization and abstraction of photographs". In: *ACM Transactions on Graphics* (2002).

[Eva15] Alex Evans. *Learning from failure: A Survey of Promising, Unconventional and Mostly Abandoned Renderers for 'Dreams PS4', a Geometrically Dense, Painterly UGC Game*. 2015.

[Fiš+14] Jakub Fišer, M. Lukáč, O. Jamriška, M. Čadík, Y. Gingold, P. Asente, and D. Sýkora. "Color me noisy: Example-based rendering of hand-colored animations with temporal noise control". In: *Computer Graphics Forum* 33.4 (2014).

[Fiš+16] Jakub Fišer, Ond\vrej Jamriška, Michal Lukáč, Eli Shechtman, Paul Asente, Jingwan Lu, and Daniel Sýkora. "StyLit: Illumination-Guided Example-Based Stylization of 3D Renderings". In: *ACM Transactions on Graphics* 35.4 (2016).

[GCS02] Bruce Gooch, Greg Coombe, and Peter Shirley. "Artistic Vision : Painterly Rendering Using Computer Vision Techniques". In: *NPAR 02 Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering* (2002).

[GEB16] L. A. Gatys, A. S. Ecker, and M. Bethge. "Image Style Transfer Using Convolutional Neural Networks". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 2414–2423.

[Goo+98] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. "A Non-photorealistic Lighting Model for Automatic Technical Illustration". In: *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 1998, pp. 447–452.

[Gra+10] Stéphane Grabli, Emmanuel Turquin, Frédo Durand, and François X Sillion. "Programmable rendering of line drawing from 3D scenes". In: *ACM Transactions on Graphics (TOG)* 29.2 (2010), p. 18.

[Hae90] Paul Haeberli. "Paint by numbers: abstract image representations". In: *ACM SIGGRAPH Computer Graphics* (1990).

[HE04] James Hays and Irfan Essa. "Image and video based painterly animation". In: *Proc. NPAR* (2004).

[Her+01] Aaron Hertzmann, Charles E Jacobs, Nuria Oliver, Brian Curless, and David H Salesin. "Image Analogies". In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 2001, pp. 327–340.

[Her03] A. Hertzmann. "A survey of stroke-based rendering". In: *IEEE Computer Graphics and Applications* 23.4 (July 2003), pp. 70–81.

[Her98] Aaron Hertzmann. "Painterly rendering with curved brush strokes of multiple sizes". In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques - SIGGRAPH '98*. 1998.

[HGT13] Siddharth Hegde, Christos Gatzidis, and Feng Tian. "Painterly rendering techniques: a state-of-the-art review of current approaches". In: *Computer Animation and Virtual Worlds* 24.1 (Jan. 2013), pp. 43–64.

[HJN03] R. Hashimoto, H. Johan, and T. Nishita. "Creating various styles of animations using example-based filtering". In: *Proceedings Computer Graphics International 2003*. July 2003, pp. 312–317.

[Hog91] Burne Hogarth. *Dynamic Light and Shade*. Watson-Guptill, 1991.

[Imh+15] Nicolas Imhof, Antoine Milliez, Flurin Jenal, René Bauer, Markus Gross, and Robert W Sumner. "Fin Textures for Real-time Painterly Aesthetics". In: *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games*. New York, NY, USA: ACM, 2015, pp. 227–235.

[Ise16] Tobias Isenberg. "Interactive NPAR: What type of tools should we create?" In: *Proceedings of the Joint Symposium on Computational Aesthetics and Sketch Based Interfaces and Modeling and Non-Photorealistic Animation and Rendering*. Eurographics Association, 2016, pp. 89–96.

[Jin+17] Yongcheng Jing, Yezhou Yang, Zunlei Feng, Jingwen Ye, Yizhou Yu, and Mingli Song. "Neural Style Transfer: A Review". In: *arXiv:1705.04058 [cs, eess, stat]* (May 2017).

[Kal+02] Robert D Kalnins, Lee Markosian, Barbara J Meier, Michael A Kowalski, Joseph C Lee, Philip L Davidson, Matthew Webb, John F Hughes, and Adam Finkelstein. "WYSIWYG NPR: Drawing Strokes Directly on 3D Models". In: *ACM Transactions on Graphics (Proc. SIGGRAPH)* 21.3 (July 2002), pp. 755–762.

[KD08] Jan Eric Kyprianidis and Jürgen Döllner. "Image Abstraction by Structure Adaptive Filtering". In: *Proc. EG UK Theory and Practice of Computer Graphics* (2008).

[KK11] Jan Eric Kyprianidis and Henry Kang. "Image and Video Abstraction by Coherence-Enhancing Filtering". In: *Computer Graphics Forum* 30.2 (2011), pp. 593–602.

[KK89] J T Kajiya and T L Kay. "Rendering Fur with Three Dimensional Textures". In: *SIGGRAPH Comput. Graph.* 23.3 (July 1989), pp. 271–280.

[KL03] George Katanics and Tasso Lappas. "Deep Canvas: Integrating 3D Painting and Painterly Rendering". In: *Theory and Practice of Non-Photorealistic Graphics: Algorithms, Methods, and Production Systems*. Ed. by Mario Costa Sousa. Vol. 10. New York: ACM SIGGRAPH, 2003.

[KLC09] H. Kang, Seungyong Lee, and C. K. Chui. "Flow-based image abstraction". In: *IEEE Transactions on Visualization and Computer Graphics* (2009).

[Kle+00] Allison W. Klein, Wilmot Li, Michael M. Kazhdan, Wagner T. Corrêa, Adam Finkelstein, and Thomas A. Funkhouser. "Non-photorealistic Virtual Environments". In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 527–534.

[Kon+14] Charalampos Koniaris, Darren Cosker, Xiaosong Yang, and Kenny Mitchell. "Survey of texture mapping techniques for representing and rendering volumetric mesostructure". In: *Journal of Computer Graphics Techniques* (Feb. 2014).

[Kow+99] Michael A Kowalski, Lee Markosian, J D Northrup, Lubomir Bourdev, Ronen Barzel, Loring S Holden, and John F Hughes. "Art-Based Rendering of Fur, Grass, and Trees". In: *Proceedings of SIGGRAPH 99*. Aug. 1999, pp. 433–438.

[KP11] Michael Kass and Davide Pesare. "Coherent noise for non-photorealistic rendering". In: *ACM SIGGRAPH 2011 papers on - SIGGRAPH '11*. 2011.

[KPD10] William B Kerr, Fabio Pellacini, and Jonathan D Denning. "BendyLights: Artistic Control of Direct Illumination by Curving Light Rays". In: *Computer Graphics Forum* (2010).

[Kyp+13] Jan Eric Kyprianidis, John Collomosse, Tinghuai Wang, and Tobias Isenberg. "State of the 'Art: A taxonomy of artistic stylization techniques for images and video". In: *IEEE Transactions on Visualization and Computer Graphics* (2013).

[Lag+09] Ares Lagae, Sylvain Lefebvre, George Drettakis, and Philip Dutré. "Procedural noise using sparse Gabor convolution". In: *ACM Transactions on Graphics* (2009).

[Lag+10] Ares Lagae, Sylvain Lefebvre, Rob Cook, Tony Derose, George Drettakis, David S Ebert, J P Lewis, Ken Perlin, and Matthias Zwicker. "A Survey of Procedural Noise Functions". In: *Computer Graphics Forum* 29.8 (2010), pp. 2579–2600.

[Lee+07] Yunjin Lee, Lee Markosian, Seungyong Lee, and John F Hughes. "Line Drawings via Abstracted Shading". In: *ACM Trans. Graph.* 26.3 (July 2007).

[Len+01] Jerome E Lengyel, Emil Praun, Adam Finkelstein, and Hugues Hoppe. "Real-Time Fur over Arbitrary Surfaces". In: *2001 ACM Symposium on Interactive 3D Graphics*. Mar. 2001, pp. 227–232.

[Lew84] J. P. Lewis. "Texture synthesis for digital painting". In: *ACM SIGGRAPH Computer Graphics* 18.3 (1984), pp. 245–252.

[Lew89] J. P. Lewis. "Algorithms for solid noise synthesis". In: *ACM SIGGRAPH Computer Graphics* (1989).

[Lit97] Peter Litwinowicz. "Processing images and video for an impressionist effect". In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques - SIGGRAPH '97*. 1997.

[LM01] E B Lum and Kwan-Liu Ma. "Non-photorealistic rendering using watercolor inspired textures and illumination". In: *Proceedings Ninth Pacific Conference on Computer Graphics and Applications. Pacific Graphics 2001*. 2001, pp. 322–330.

[Lop+13] Jorge Lopez-Moreno, Stefan Popov, Adrien Bousseau, Maneesh Agrawala, and George Drettakis. "Depicting Stylized Materials with Vector Shade Trees". In: *ACM Transactions on Graphics (SIGGRAPH Conference Proceedings)* 32.4 (2013).

[LSF10] Jingwan Lu, Pedro V Sander, and Adam Finkelstein. "Interactive Painterly Stylization of Images, Videos and {3D} Animations". In: *Proceedings of I3D 2010*. Feb. 2010.

[Mei96] Barbara J Meier. "Painterly Rendering for Animation". In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 1996, pp. 477–484.

[MFE07] Jason Mitchell, Moby Francke, and Dhabih Eng. "Illustrative rendering in team fortress 2". In: *Proceedings of the 5th international symposium on Non-photorealistic animation and rendering*. ACM, 2007, pp. 71–76.

[MN98] Alexandre Meyer and Fabrice Neyret. "Interactive Volumetric Textures". In: *Eurographics Workshop on Rendering techniques (Rendering Techniques'98)*. Ed. by George Drettakis and Nelson Max. Vienna, Austria: Eurographics, June 1998, pp. 157–168.

[MNI01] Xiaoyang Mao, Yoshiyasu Nagasaka, and Atsumi Imamiya. "Automatic Generation of Pencil Drawing from 2D Images Using Line Integral Convolution". In: *Proc. International Conference on Computer Aided Design and Computer Graphics (CAD/GRAPHICS'01)*. 2001.

[MSR16] Santiago E Montesdeoca, Hock-Soon Seah, and Hans-Martin Rall. "Art-directed Watercolor Rendered Animation". In: *Non-Photorealistic Animation and Rendering*. Ed. by Pierre Bénard and Holger Winnemöller. The Eurographics Association, 2016.

[ND04a] Marc Nienhaus and Jürgen Döllner. "Blueprints: illustrating architecture and technical parts using hardware-accelerated non-photorealistic rendering". In: *Proceedings of Graphics Interface 2004*. School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada: Canadian Human-Computer Communications Society, 2004, pp. 49–56.

[ND04b] Marc Nienhaus and Jürgen Döllner. "Sketchy drawings". In: *Proceedings of the 3rd international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*. ACM, 2004, pp. 73–81.

[Oka+07] Makoto Okabe, Yasuyuki Matsushita, Li Shen, and Takeo Igarashi. "Illumination brush: Interactive design of all-frequency lighting". In: *Computer Graphics and Applications, 2007. PG'07. 15th Pacific Conference on*. IEEE, 2007, pp. 171–180.

[Pac+08] Romain Pacanowski, Xavier Granier, Christophe Schlick, and Pierre Poulin. "Sketch and paint-based interface for highlight modeling". In: *Eurographics Workshop on Sketch-Based Interfaces and Modeling*. 2008, pp. 7–23.

[Pel+07] Fabio Pellacini, Frank Battaglia, Keith Morley, and Adam Finkelstein. "Lighting with Paint". In: *ACM Transactions on Graphics* 26.2 (June 2007), Article 9.

[Pel10] Fabio Pellacini. "envyLight: An Interface for Editing Natural Illumination". In: *ACM Trans. Graph.* 29.4 (July 2010), 34:1–34:8.

[Per85] Ken Perlin. "An Image Synthesizer". In: *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '85. New York, NY, USA: ACM, 1985, pp. 287–296.

[PH89] Ken Perlin and Eric M Hoffert. "Hypertexture". In: *ACM SIGGRAPH Computer Graphics*. Vol. 23. ACM, 1989, pp. 253–262.

[Pho75] Bui Tuong Phong. "Illumination for Computer Generated Pictures". In: *Commun. ACM* 18.6 (June 1975), pp. 311–317.

[PP09] Giuseppe Papari and Nicolai Petkov. "Continuous Glass Patterns for Painterly Rendering". In: *IEEE Transactions on Image Processing* 18.3 (Mar. 2009), pp. 652–664.

[Pra+01] Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein. "Real-time Hatching". In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 581–.

[Rad99] Paul Rademacher. "View-dependent Geometry". In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 439–446.

[RBD06] Szymon Rusinkiewicz, Michael Burns, and Doug DeCarlo. "Exaggerated Shading for Depicting Shape and Detail". In: *ACM Transactions on Graphics, Proceedings of ACM SIGGRAPH 2006 (Boston, MA, July 30–August 3, 2006)* 25.3 (2006), pp. 1199–1205.

[Rit+09] Tobias Ritschel, Makoto Okabe, Thorsten Thormählen, and Hans-Peter Seidel. "Interactive Reflection Editing". In: *ACM Trans. Graph. (Proc. SIGGRAPH Asia 2009)* 28.5 (2009).

[Rit+10] Tobias Ritschel, Thorsten Thormählen, Carsten Dachsbacher, Jan Kautz, and Hans-Peter Seidel. "Interactive On-surface Signal Deformation". In: *ACM Trans. Graph.* 29.4 (July 2010), 36:1–36:8.

[Sch+10] Johannes Schmid, Robert W. Sumner, Huw Bowles, and Markus Gross. "Programmable Motion Effects". In: *ACM SIGGRAPH 2010 Papers*. SIGGRAPH '10. New York, NY, USA: ACM, 2010, 57:1–57:9.

[Sch+11] Johannes Schmid, Martin Sebastian Senn, Markus Gross, and Robert W Sumner. "OverCoat: an implicit canvas for 3D painting". In: *ACM Trans. Graph.* 30.4 (Aug. 2011), 28:1–28:10.

[Sch+13] Thorsten-Walther Schmidt, Jan Novak, Johannes Meng, Anton S Kaplanyan, Tim Reiner, Derek Nowrouzezahrai, and Carsten Dachsbacher. "Path-Space Manipulation of Physically-Based Light Transport". In: *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2013)* 32.4 (Aug. 2013).

[Sem+16] Amir Semmo, Daniel Limberger, Jan Eric Kyprianidis, and Jürgen Döllner. "Image Stylization by Interactive Oil Paint Filtering". In: *Comput. Graph.* 55.C (2016), pp. 157–171.

[SID17] Amir Semmo, Tobias Isenberg, and Jürgen Döllner. "Neural Style Transfer: A Paradigm Shift for Image-based Artistic Rendering?" In: *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*. NPAR '17. New York, NY, USA: ACM, 2017, 5:1–5:13.

[Slo+01] Peter-Pike Sloan, William Martin, Amy Gooch, and Bruce Gooch. "The Lit Sphere: A Model for Capturing NPR Shading from Art". In: *Proceedings of Graphics Interface 2001*. Toronto, Ont., Canada, Canada: Canadian Information Processing Society, 2001, pp. 143–150.

[ST90] Takafumi Saito and Tokiichiro Takahashi. "Comprehensible Rendering of 3-D Shapes". In: *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 1990, pp. 197–206.

[TAI09] Hideki Todo, Ken Anjyo, and Takeo Igarashi. "Stylized lighting for cartoon shader". In: *Computer Animation and Virtual Worlds* 20.2-3 (2009), pp. 143–152.

[TAY13] Hideki Todo, Ken Anjyo, and Shun'Ichi Yokoyama. "Lit-Sphere Extension for Artistic Rendering". In: *Vis. Comput.* 29.6-8 (June 2013), pp. 473–480.

[TC00] S. M. F. Treavett and M. Chen. "Pen-and-ink rendering in volume visualisation". In: *Proceedings Visualization 2000. VIS 2000 (Cat. No.00CH37145)*. Oct. 2000, pp. 203–210.

[Tha11] Jonathan Thaler. "Deferred Rendering". In: (Feb. 2011).

[TM98] Carlo Tomasi and Roberto Manduchi. "Bilateral filtering for gray and color images". In: *Computer Vision, 1998. Sixth International Conference on*. IEEE, 1998, pp. 839–846.

[Van+07] David Vanderhaeghe, Pascal Barla, Joelle Thollot, and Francois X Sillion. "Dynamic Point Distribution for Stroke-based Rendering". In: *Proceedings of the 18th Eurographics Conference on Rendering Techniques*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, pp. 139–146.

[Van+11] David Vanderhaeghe, Romain Vergne, Pascal Barla, and William Baxter. "Dynamic Stylized Shading Primitives". In: *NPAR '11:Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*. Vancouver, Canada, Aug. 2011, pp. 99–104.

[VB15] Romain Vergne and Pascal Barla. "Designing Gratin, A GPU-Tailored Node-Based System". In: *Journal of Computer Graphics Techniques* 4.4 (2015), pp. 54–71.

[Ver+08] Romain Vergne, Pascal Barla, Xavier Granier, and Christophe Schlick. "Apparent Relief: A Shape Descriptor for Stylized Shading". In: *Proceedings of the Sixth International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2008, June 9–11, 2008, Annecy, France)* (2008).

[Ver+09] Romain Vergne, Romain Pacanowski, Pascal Barla, Xavier Granier, and Christophe Schlick. "Light Warping for Enhanced Surface Depiction". In: *ACM Transaction on Graphics (Proceedings of SIGGRAPH 2009)* 28.3 (2009).

[Ver+10] Romain Vergne, Romain Pacanowski, Pascal Barla, Xavier Granier, and Christophe Schlick. "Radiance Scaling for Versatile Surface Enhancement". In: *I3D '10: Proc. symposium on Interactive 3D graphics and games*. Boston, United States: ACM, Feb. 2010.

[Ver+11a] Romain Vergne, Romain Pacanowski, Pascal Barla, Xavier Granier, and Christophe Schlick. "Improving Shape Depiction under Arbitrary Rendering". In: *IEEE Transactions on Visualization and Computer Graphics* 17.8 (June 2011), pp. 1071–1081.

[Ver+11b] Romain Vergne, David Vanderhaeghe, Jiazhou Chen, Pascal Barla, Xavier Granier, and Christophe Schlick. "Implicit Brushes for Stylized Line-based Rendering". In: *Computer Graphics Forum* 30.2 (Apr. 2011), pp. 513–522.

[WD05] John Willats and Fredo Durand. "Defining pictorial style: Lessons from linguistics and computer graphics". In: *Axiomathes* 15.3 (2005), pp. 319–351.

[Web+02] Matthew Webb, Emil Praun, Adam Finkelstein, and Hugues Hoppe. "Fine Tone Control in Hardware Hatching". In: *Proceedings of the 2Nd International Symposium on Non-photorealistic Animation and Rendering*. NPAR '02. New York, NY, USA: ACM, 2002, 53–ff.

[Win13] Holger Winnemöller. "NPR in the Wild". en. In: *Image and Video-Based Artistic Stylisation*. Ed. by Paul Rosin and John Collomosse. Computational Imaging and Vision. London: Springer London, 2013, pp. 353–374.

[Wor96] Steven Worley. "A Cellular Texture Basis Function". In: *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: ACM, 1996, pp. 291–294.

[YMI04] Shigefumi Yamamoto, Xiaoyang Mao, and Atsumi Imamiya. "Colored pencil filter with custom colors". In: *Proceedings - Pacific Conference on Computer Graphics and Applications*. 2004.

[Zen+09] Kun Zeng, Mingtian Zhao, Caiming Xiong, and Song-Chun Zhu. "From image parsing to painterly rendering". In: *ACM Transactions on Graphics* (2009).

[Zhe+17] Ming Zheng, Antoine Milliez, Markus Gross, and Robert W Sumner. "Example-based Brushes for Coherent Stylized Renderings". In: *Proceedings of the Symposium on Non-Photorealistic Animation and Rendering*. New York, NY, USA: ACM, 2017, 3:1–3:10.

[Zub+15] Carlos J Zubiaga, Adolfo Muñoz, Laurent Belcour, Carles Bosch, and Pascal Barla. "MatCap Decomposition for Dynamic Appearance Manipulation". In: *Eurographics Symposium on Rendering 2015*. Darmstadt, Germany, June 2015.

[ZZ11] Mingtian Zhao and Song-Chun Zhu. "Customizing painterly rendering styles using stroke processes". In: *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on non-photorealistic animation and rendering*. ACM, 2011, pp. 137–146.