Diss. ETH No. 20579

Methods for Artistic Stylization in 3D Animation

A dissertation submitted to
ETH Zurich

for the Degree of
Doctor of Sciences

presented by
Johannes Schmid
MSc ETH CS, Switzerland
born 26 Mar 1982
citizen of Switzerland

accepted on the recommendation of
Prof. Dr. Markus Gross, examiner
Prof. Dr. Adam Finkelstein, co-examiner
Dr. Robert W. Sumner, co-examiner

2012

Abstract

This thesis presents new methods for artists to have direct control on the visual style of computer animations, in order to let more of their creative energy penetrate the production pipeline to the final result.

A focus is put on the development of a comprehensive system for the authoring and rendering of painterly 3D character animation. We build on the concept of stroke based rendering and contribute to the field in various aspects. We show how the brush stamping method for brush stroke rendering can be adapted to 3D stroke based rendering. The classic form of this method is used in many 2D digital painting applications, but the deformations and perspective properties and the rendering requirements in stroke based rendering call for extensions of the original algorithm. A persistent challenge in 3D stroke based rendering is to reconcile the depth order of paint strokes with the order in which they were painted. We present two new approaches to this problem, one of which guarantees temporal and spatial coherence to produce high-quality images, while the other is well suited for hardware acceleration and achieves interactive rendering performance.

Strokes painted in a 2D viewport window must be embedded in 3D space in a way that gives creative freedom to the artist while maintaining a high level of controllability. We address this challenge with a three-dimensional canvas defined implicitly by a scalar field. The artist shapes the implicit canvas with 3D proxy geometry and subsequent sculpting operations. An optimization procedure is then used to embed paint strokes in space by minimizing different objective criteria. This functionality allows us to implement tools for painting along level set surfaces or across different level sets of the scalar field.

We show how 3D stroke-based paintings can be deformed using standard rigging tools and propose a configuration-space keyframing algorithm for authoring stroke effects that depend on scene variables such as character pose or light position. Our system supports temporal keyframing for one-off effects during an animation. In order to ensure smooth keyframe interpolation in a high-dimensional configuration space, we develop a novel interpolation algorithm that avoids undesired stuttering artifacts when multiple keyframes are used.

Finally, we experiment the depiction of motion as a first-class entity in a traditional 3D rendering process. We extend the concept of a surface shader, which is evaluated on an infinitesimal portion of an object's surface at one instant in time, to that of a programmable motion effect, which is evaluated with global knowledge about all portions of an object's surface that pass in front of a pixel during an arbitrary long sequence of time. With this added information, our programmable motion effects can decide to color pixels long after (or long before) an object has passed in front of them. In order to compute the input

required by the motion effects, we propose a 4D data structure that aggregates an object's movement into a single geometric representation by sampling an object's position at different time instances and connecting corresponding edges in two adjacent samples with a bilinear patch.

Zusammenfassung

In dieser Arbeit werden neue Methoden für die direkte artistische Kontrolle über den visuellen Stil von Computeranimationen präsentiert. Das Ziel ist es, zu ermöglichen, dass mehr von der kreative Energie in Konzeptmalereien den Weg durch die Produktions-Pipeline zum schlussendlichen Resultat finden kann.

Der Fokus der Arbeit liegt auf der Entwicklung eines umfassenden Systems für die Ausarbeitung und Darstellung von zeichnerischen 3D-Animationen für Figuren. Das System baut auf der Technik des zeichenstrich-basierten Renderings auf, welche in dieser Arbeit in zentralen Punkten weiterentwickelt wird. Wir zeigen, wie die Stempel-Methode für das Malstrich-Rendering an die Anforderungen der Gegebenheiten im dreidimensionalen Raum angepasst werden kann. Die klassische Ausführung dieser Methode wird in vielen digitalen 2D-Malprogrammen verwendet. Im Rahmen des zeichenstrich-basierten Rendering werden Aufgrund der Deformationen und perspektivischen Eigenschaften sowie der Darstellungstechnik jedoch gewisse Erweiterungen notwendig. Ein bekanntes Problem im zeichenstrich-basierten 3D-Rendering ist die Konkurrenz von zwei Kriterien für die Darstellungsreihenfolge der Malstriche: Die Zeichnungsreihenfolge und die dreidimensionale Sichtbarkeitsreihenfolge. Wir führen zwei neue Lösungsansätze zu diesem Problem ein, von welchen der eine temporale und räumliche Glattheit garantiert und dadurch hochqualitative Bilder generiert, während der andere dank guter Kompatibilität mit Grafikbeschleunigern interaktive Darstellungsraten ermöglicht.

Digitale Malstriche, die mit einem zweidimensionalen Eingabegerät appliziert werden, müssen für die Verwendung in 3D-Animationen im dreidimensionalen Raum eingebettet werden. Unsere Anforderung an diesen Einbettungsvorgang ist, dass er dem Anwender einen hohen Grad an künstlerischer Freiheit bietet und trotzdem gut kontrollierbar bleibt. Unser Lösungsansatz zu diesem Problem ist eine dreidimensionale “Leinwand”, welche implizit durch ein Skalarfeld definiert ist. Das Skalarfeld wird als Distanzfeld zu einfachen 3D-Modellen initialisiert und kann anschliessend mit formgebenden Hilfsmitteln bearbeitet werden. Malstriche werden in diesem Skalarfeld durch ein mathematisches Optimierungsverfahren eingebettet, in welchem verschiedene Kriterien gegeneinander abgewogen werden können. Dadurch können auf einfache Art verschiedene Einbettungsmethoden realisiert werden.

Für die Animation von solchen 3D-Gemälden werden zwei sich ergänzende Techniken erläutert: Mittels einer Rigging-Methode können die Malstriche gemäss der Animation des dem Skalarfeld zugrundeliegenden 3D-Modells deformiert werden. Detailierte Effekte, die über die Animation der 3D-Modelle hinausgehen, können mittels Schlüsselbildanimation in Bezug auf den Konfigurationsraum der Animation erreicht werden. Für die Gewährleistung der glatten Schlüsselbild-Interpolation im hochdimensionalen Konfigurationsraum wurde

eine neuartige Interpolationsmethode entwickelt, die unerwünschte Stotter-Artefakte bei mehreren kolinearen Schlüsselbildpositionen verhindert.

Zuletzt wird eine neuartige Rendering-Technik zur Darstellung von Bewegung in Einzelbildern präsentiert. Das Konzept von programmierbaren Oberflächenschattierern, mit welchen das Aussehen eines einzelnen Ortes zu einem bestimmten Zeitpunkt berechnet werden kann, wird um die Zeitdimension zu "Bewegungseffektprogrammen" erweitert, welche das Aussehen eines auf der Bildfläche liegenden Punktes mittels dem Wissen um sämtliche geometrische Örter, welche innerhalb eines beliebigen Zeitbereichs durch den Bildpunkt sichtbar waren, berechnen. Dank dieser zusätzlichen Informationen können Bewegungseffektprogramme den Bildpunkt anhand von vergangenen oder zukünftigen Ereignissen einfärben, und dadurch traditionelle Bewegunseffekte wie Bewegungslinien und Bewegungsunschärfe reproduzieren. Die benötigten Informationen werden mittels einer neuartige 4D-Datenstruktur berechnet, welche die Bewegung eines Objekts auf geometrische Art und Weise aggregiert, indem Kopien des Objekts zu unterschiedlichen Zeitpunkten zusammengefügt und deren Kanten mit bilinearen Interpolationsflächen verbunden werden.

Acknowledgements

I wish to thank my advisor Prof. Markus Gross for establishing such an excellent research environment and for making it possible for me to be a part of it. With their guidance and expert advice, Prof. Gross and my direct supervisor Dr. Robert Sumner have equipped me with the skills and intuitions necessary to contribute to the science of computer graphics.

For me, the biggest challenge during my doctoral studies was finding the motivation to continue when path and outcome were uncertain and the chance of success seemed slim. Bob Sumner has always managed to bring me back on track in these situations, for which I am deeply grateful. He taught me how to approach difficult problems, which is probably the biggest lesson I am taking away from my dissertation.

A substantial amount of the work presented in this thesis was done with the help of excellent student collaborators. I would especially like to thank Huw Bowles, Martin Senn, and Katie Bassett for their central contributions, but also Claudia Kuster, Thomas Siegrist, and Philipp Simmler for their support in various parts of our work. Another collaborator who I am greatly indebted to is Dr. Ilya Baran, who was never reluctant to part with bits of his enormous knowledge of computer science and mathematics when I sought his advice. Gerhard Röthlin has contributed to the OverCoat software with many features and bugfixes, and has greatly improved the overall software architecture.

I have learned a lot about the problems we worked on in talking to people at the Walt Disney Animation Studios and Pixar. I would especially like to thank Dan Teece for his support of our projects, and Rasmus Tamstorf for providing support with respect to the Maya API and other matters.

Big thanks go the members of Disney Research Zurich and the Computer Graphics Lab at ETH Zurich for countless good discussions about research and life, and for making it such a fun place to work. I hereby formally apologize for always going home early.

Finally, I would like to express my deep personal gratitude to my parents, my sister and brother, and to Sabrina. They always were and continue to be the biggest source of inspiration in my life.

Contents

CHAPTER 1

Introduction

From its early days on, the main source of inspiration in computer graphics research has been the real world and how we perceive it. It starts with geometric models as an approximation of the shape of real objects, and continues over the visual description of materials (textures, lighting and shading models) and their faithful reproduction (rendering of illumination and shadows) to the simulation of how objects move and deform (skeleton models, physical simulation, high-level motion controllers). As a consequence, computer graphics techniques have evolved to a point where many real-world phenomena can be modeled and rendered in a quality that makes them indistinguishable from reality to our eyes.

In practical applications of computer graphics, however, photorealistic rendering and lifelike animation is not always the desired goal. Seen in a broader context, visual arts in general often does not strive for utmost realism, but instead embraces stylization to better convey impressions, emotions, and information. Stylization can be desirable both for artistic and for practical reasons. It can serve to focus the viewer's attention, to remove unnecessary information, or simply to make the image more appealing. Against this background, it is not surprising that computer graphics also experiences a strong demand for stylized depiction. Some categories of applications for stylized computer graphics evolved from their traditional counterparts: 3D cartoons, information visualization, scientific illustration, etc. Other applications, such as computer games, are endemic to computer graphics and yet have developed a need for stylization.

Figure 1.1: A comparison of images from various recent 3D animation films, showing how close they all are in visual style. For example, while the shapes of the faces vary considerably, the visual appearance is very similar in all samples. The same argument can be made for hair and clothing. Images taken from a) *Monsters vs. Aliens* (© DreamWorks Animation), b) *Cloudy with a Chance of Meatballs* (© Sony Pictures Animation), c) *Ratatouille* (Pixar), d) *Up* (Pixar), e) *Tangled* (Walt Disney Animation Studios), f) *Bolt* (Walt Disney Animation Studios). Images c)–f) © Disney Enterprises, Inc.

An example of the demand for stylization in computer graphics can be found in 3D animated feature film production. Such films have been increasingly popular in the past decade, with more than 10 films being released each year recently. However, a common criticism that has come up is that most of these films have a very similar visual style. In particular, they all exhibit a certain synthetic look that has become representative for 3D animation. They employ a varying amount and flavor of abstraction for shape, motion, and lighting, but not for shading and rendering. Figure 1.1 illustrates this argument with a number of screen captures of major recent 3D movies. The lack of visual diversity has been identified years ago, and production studios have since made significant efforts to alleviate it by broadening the spectrum of visual styles seen on screen. But it has proven to be difficult to produce a 3D animated movie that exhibits a substantially different style, is pleasant to watch for the length of a feature film, and can be realized with a reasonable amount of work with the technologies available today.

1.1 Objective

The quest of bringing artistic stylization into computer animation bridges over the disciplines of art and technology. From an artistic point of view, a major challenge is to define how visual stylizations should look when animated. There is an enormous amount of inspiration and knowledge to be found in traditional visual arts, but it is often unclear how these exemplars can be brought to life through motion in a visually pleasing way. Once there is a clear artistic vision of a stylized animation, the next challenge lies in the practical realization of this vision. Due to the amount of work and precision required for the manual creation of animations, technology is often the enabling factor in doing so. But technological research is not limited to a reactive role in the design of new animation styles. Experimentation with new technologies can be very fruitful in the development of artistic features. As John Lasseter puts is, "the art challenges the technology, and the technology inspires the art."

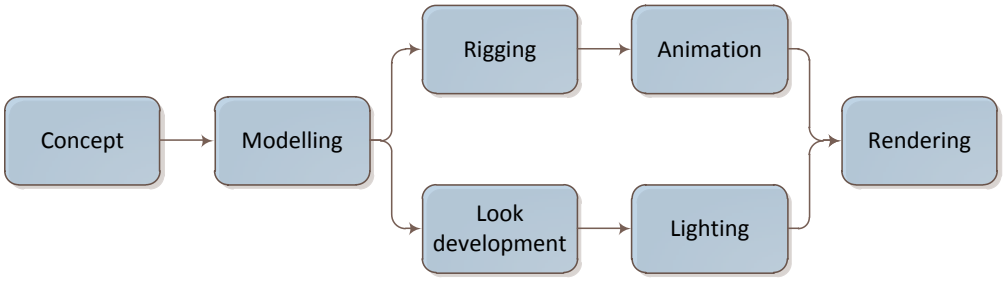


Figure 1.2: An illustration of the conventional workflow, or "pipeline", used in the production of 3D animation movies. Asset production in computer games also follows these steps closely. Each stage typically applies a number canonical mathematical models and data representations, which limits the fidelity to which the artwork from the concept stage can be reflected in the final result.

The primary concern of this thesis is to enable and facilitate new visual styles in 3D animation with the development of new technologies and tools. The biggest challenge therein lies in the rigidity of the canonical techniques used to produce 3D animations. Figure 1.2 depicts a workflow that is used to create animations both in film and in games. Painterly styles are often seen in the *Concept* stage, but have shown to be inherently difficult to achieve as the final result with a standard 3D animation pipeline. Each step in the pipeline involves mathematical models and algorithms to tackle the various problem at hand. Originally, these models were typically developed to imitate phenomena of reality. In order

Figure 1.3: The image to the left shows a concept art piece for the Disney/Pixar movie "Ratatouille" and the image to the left an actual frame from that movie. Producing a movie in the visual style shown in the concept piece would be very difficult with the canonical 3D animation pipeline and the current state-of-art in research. © Disney Enterprises, Inc.

to make production as efficient as possible, they are designed intelligently to take as much workload off the user as possible. But such automatism often comes at the cost of limiting the flexibility of the result. The models used in standard 3D animation pipelines usually do not explicitly cater to the needs of artistic stylization and can be very rigid to deal with when trying to achieve new styles. We believe that this rigidity is the main reason why animation studios with established 3D animation pipelines have found it difficult to break away from certain design attributes that are prevalent in existing productions, such as the aforementioned synthetic look (Figure 1.3).

Interestingly, this problem also existed (and continues to exist) in 2D animation. Figure 1.4 illustrates an anecdote according to which Walt Disney picked up a concept painting made by Joe Grant, one of the pivotal Disney artists, and wanted to use its visual style for animation [Thomas and Johnston, 1981]. But, the production pipeline for 2D animations at this time could not capture and reproduce the kind of style shown in Joe's painting in an efficient manner.

1.2 Overview

In this thesis, we present technologies that either extend or replace elements of the conventional 3D animation pipeline to allow more artistic control and expressiveness in the process. Chapters 2–4 combine to a system that is intended to partially replace the *Modeling*, *Rigging*, *Look dev*, and *Rendering* steps with

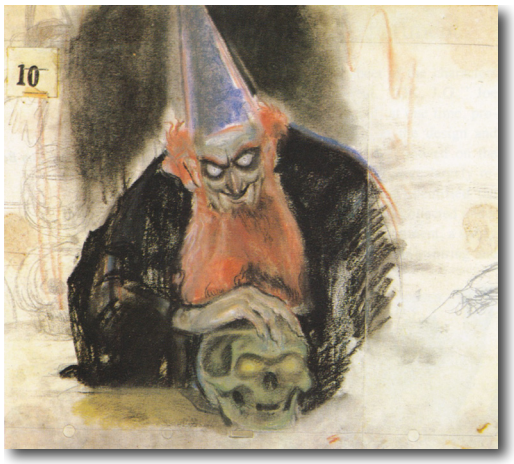


Figure 1.4: Even in 2D animation, technical limitations of the production pipeline continue to impose limitations on the visual style attainable in the end result. © Disney Enterprises, Inc.

the goal of allowing more of the artistic freedom of the *Concept* stage to permeate through the pipeline to the end result. Our system empowers its users to produce painterly-looking animations while making use of some of the powerful instruments for animation in 3D. We build upon the core concepts introduced by Deep Canvas [Katanics and Lappas, 2003]: the usage of paint strokes that are drawn onto proxy 3D geometry by the user as the central rendering primitive. We extend those concepts in Chapters 2–4 to provide greater artistic flexibility and the capability of animation.

Chapter 2 revolves around the painterly rendering of brush strokes located in space. We adapt the brush "stamping" technique that is used in many 2D digital painting applications (including Adobe Photoshop) for use in 3D animation. This technique renders brush strokes by repeatedly "stamping" a (typically square or circular) brush texture along the stroke path with close spacing, thus creating the impression of a continuous stroke. While such an approach is trivial in 2D, the notion of depth and temporal coherence requires additional consideration in the context of 3D animation, which is addressed in this chapter. In addition, we present novel solutions to the problem of respecting the paint order of brush strokes on 3D surfaces. This problem arises when paint strokes on a surface should be draw in the order that they were painted to respect the artist's intent, while paint strokes on different surfaces should be drawn according to their depth order to respect the scene's three-dimensional structure.

In Chapter 3, we address the problem of placing brush strokes in 3D space with a 2D input device (such as a mouse or a graphic tablet). The lack of depth information with these input devices leaves one degree of freedom open

when transforming the positions from 2D to 3D. We deal with this ambiguity by requiring proxy geometry in the form of a 3D mesh to be provided for each object that is to be painted. These proxies are used to roughly describe how space is occupied by objects and guide the embedding of paint strokes into space accordingly. Our system allows the exact nature of the interaction between the proxy geometry and embedded paint strokes to be defined flexibly in a mathematical optimization framework and made accessible to the user through encapsulation in different *embedding tools*. For example, one such embedding tool could place brush strokes at a certain distance to the proxy geometry, while another one could allow the user to paint perpendicular to the surface. Our results demonstrate how these simple tools allow artists to evoke certain visual characteristics in 3D paintings that would have been difficult to achieve with traditional methods.

Chapter 4 presents methods for authoring animations with the 3D painting system established in Chapters 2 and 3, with a focus on character animation. Continuing the spirit of using proxy geometry from Chapter 3, we assume that the gross motion and deformation of objects is already given for the proxy geometry, and that our system has access to the deformed vertex positions for any desired pose. Naturally, the user expects paint strokes to automatically be transformed according to the underlying proxy geometry. We propose to achieve this goal with a linear blend skinning approach. Skinning deformation method lets the 3D painting reflect the gross motion of the object, but does not provide the user with immediate control over the movement and properties of paint strokes. The need for such control arises when the result of skinning deformation is not satisfactory, or when the user wants to author motion that is not reflected in the proxy geometry, for example because it exceeds the detail level that is captured by the proxy geometry. We propose a configuration-space keyframing algorithm for authoring pose-dependent stroke effects. This mechanism allows stroke opacity or movement to be keyframed to positions in a configuration space which includes character pose parameters and other scene variables such as light positions. Our system also allows the same quantities to be key-framed in time, which enables one-off effects and tuning during the animation.

Finally, Chapter 5 presents an extension to the traditional *Rendering* stage that simplifies the art direction of the depiction of motion within a single image or a single frame of an animation. We extend the concept of a surface shader, which is evaluated on an infinitesimal portion of an object's surface at one instant in time, to that of a programmable motion effect, which is evaluated with global knowledge about all portions of an object's surface that pass in front of a pixel during an arbitrary long sequence of time. We present a data structure and algorithms to generate the necessary information and pass it to a *motion effect program*. This novel shader concept can decide to color pixels long after (or long before) an object has passed in front of them, enabling to obtain effects such as

speed lines, stroboscopic copies, streaking, and stylized blurring. By rendering different portions of an object at different times, our motion effect programs are also able to achieve perceived stretching and bending of objects.

1.3 Background

In his invitation to discuss computer depiction, Durand [Durand, 2002] highlights the difference between primary space (the 3D world in which objects live) and secondary space (the 2D canvas on which depictions of those objects are created), illustrated in Figure 1.5. This distinction, originally introduced to analyze the historical use of representation systems in engineering drawings [Booker, 1963] and fine art [Willats, 1997], provides a lens through which to explore the development of expressive depiction in computer graphics.

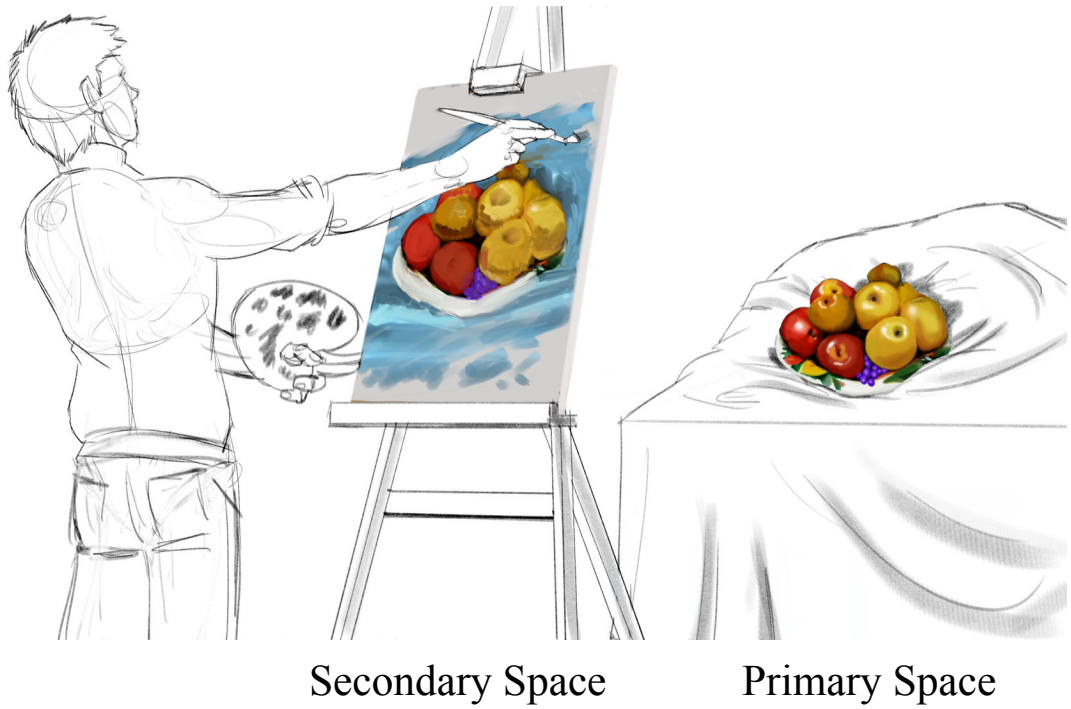


Figure 1.5: Primary space refers to the 3D world in which objects live, while secondary space denotes the 2D canvas on which depictions of those objects are created [Durand, 2002]. Our work blurs the distinction between these two spaces by upgrading strokes to the primary space and downgrading traditional 3D objects to serve only as helpers in defining an implicit canvas.

The field of non-photorealistic rendering (NPR) has developed a rich col-

lection of expressive depiction methods. Although traditional photorealistic rendering research focuses on the primary space (e.g., scene representation, visibility determination, global illumination), the NPR community first approached the problem from the opposite direction by focusing entirely on the secondary space of the 2D canvas. Haeberli's interactive "Paint By Numbers" system [Haeberli, 1990] fills a 2D canvas with brush strokes whose attributes are controlled by information contained in a photograph. This concept led to an entire sub-field of research on stroke-based rendering [Hertzmann, 2003] that encompases improved brush models [Hertzmann, 1998], image segmentation and parsing [Gooch et al., 2002, Zeng et al., 2009, Zhao and Zhu, 2010], video processing [Litwinowicz, 1997, Hertzmann and Perlin, 2000, Hays and Essa, 2004, Winnemöller et al., 2006, Lu et al., 2010, Lin et al., 2010], user-guided pen-and-ink illustration [Salisbury et al., 1997], user interaction metaphors [Schwarz et al., 2007], as well as many other advancements. Other secondary-space approaches focus on simulating the physical properties of traditional brushes and paint media such as watercolor [Curtis et al., 1997], oil painting [Baxter et al., 2001, Baxter et al., 2004, Chu et al., 2010], and pencil [Sousa and Buchanan, 2000]. Image analogies [Hertzmann et al., 2001] offer a way for automated stylization of 2D imagery on the basis of an exemplar image. All of these methods share a common focus on computation performed in the secondary space of the 2D canvas without explicit representation of the 3D world.

Inspiring secondary-space results naturally led researchers to extend expressive depiction to the primary space of 3D objects, leading to a diverse collection of non-photorealistic rendering algorithms. One topic that has received a lot of attention is the the creation of line drawings from 3D models. A sparse set of lines to describe the shape of a model can be found in silhouettes [Hertzmann and Zorin, 2000], suggestive contours [DeCarlo et al., 2003], and apparent ridges [Judd et al., 2007]. More detailed line-art illustration of surfaces can be obtained using stroke textures [Winkenbach and Salesin, 1994] and hatching [Saito and Takahashi, 1990, Winkenbach and Salesin, 1996, Hertzmann and Zorin, 2000, Praun et al., 2001, Kalogerakis et al., 2012]. Kowalski and colleagues demonstrate the procedural use of small line art elements to create stylized depictions of fur, grass, and trees [Kowalski et al., 1999]. Recently, a programmable system for the creation of line drawings from 3D scenes has been presented [Grabli et al., 2010].

Moving away from line art, but often used in conjunction with it, toon shading [Decaudin, 1996, Lake et al., 2000, Barla et al., 2006] provides a popular method to give 3D objects a cartoony appearance using a thresholded shading strategy. A more radical stylization in this direction can be obtained using the 2.5 cartoon model by Rivers and colleagues [Rivers et al., 2010]. The lit sphere [Sloan et al., 2001] allows objects to be shaded according to a painted example (hemi-)sphere. Vanderhaeghe and colleagues have recently

presented a shading primitive that is designed specifically for the stylized depiction of 3D objects [Vanderhaeghe et al., 2011]. 3D surfaces can also be stylized using art maps [Klein et al., 2000], 2D patterns [Breslav et al., 2007], solid textures [Bénard et al., 2009], and methods based on coherent noise [Kass and Pesare, 2011].

The common focus of the techniques listed in the last two paragraphs is an algorithmic mapping from primary space to secondary space that implements different expressive styles. Among these methods, Meier's painterly rendering system [Meier, 1996] marks the inception of a line of research closely related to our own. Particles attached to an object's primary-space surface are used to render secondary-space brush strokes so that the strokes stick to the object as the camera moves. This simple, though groundbreaking, concept ultimately led to the development of Disney's Deep Canvas technology [Katanics and Lappas, 2003], which replaces Meier's procedurally generated particles with an artist-driven painting system. Painted strokes are projected on the object's surface and stored along with all data associated with the painting system. A new view is rendered by "repainting," or playing back all recorded painting operations, using the camera's new view transformation. Concurrent with the development of Deep Canvas, Teece [Teece, 2000] proposed a related painting concept with a focus on interactivity. The WYSIWYG NPR system of Kalnins and colleagues [Kalnins et al., 2002] expands upon this line of work by showing how algorithmic rendering techniques such as silhouette stylization or hatching can be controlled directly by the artist via a painting interface. Our work draws heavily from these ideas and advances them in the areas of rendering, stroke placement, and animation.

1.4 Contributions

In this thesis, we make the following scientific contributions:

- We show how the brush stamping technique for brush stroke rendering can be adapted to the conditions of 3D stroke based rendering, along with some advanced features like brush parameter jittering and canvas texturing.
- We formalize the problem of combining paint order with depth order in 3D stroke based rendering. We also present two new solutions to this problem, one of which is tailored towards real-time applications and the other is suitable for high-quality rendering due to its unconditional smoothness.
- We introduce the concept of an implicit 3D canvas, which allows the full 3D space to be treated as a canvas for painting in the presence of proxy geometry.

- We present a method for flexible and customizable embedding of 2D input strokes in 3D space using a mathematical optimization framework. As an example, we implement three different embedding tools with this method.
- We present a system and workflow for authoring character animations in the context of stroke based rendering. On a high level, our system consists of two components: paint stroke skinning and configuration-space interpolation. Both components are designed to cooperate well with the painting input metaphor.
- For configuration-space interpolation, we introduce a novel high-dimensional scattered data interpolation scheme, which is particularly suited for keyframe interpolation in animation.
- We demonstrate how depictions of motion in 3D animation can be authored in a flexible, programmable fashion in a single step in the rendering process. To this end, we present a data structure that captures the motion of objects in a form which can easily be communicated to the renderer, along with an algorithm that processes this global motion data into per-pixel motion information. We show how the resulting information can be flexibly processed into depictions of motion effects on the basis of several examples.

1.5 Publications

This thesis is based on three publications that were accepted to ACM Transactions on Graphics and presented at the SIGGRAPH and SIGGRAPH Asia conferences:

- Schmid, J., Sumner, R. W., Bowles, H., and Gross, M. 2010. Programmable motion effects. In *ACM Transactions on Graphics*, 29(4), 57:1–57:9.
- Schmid, J., Senn, M. S., Gross, M., and Sumner, R. W. 2011. OverCoat: an implicit canvas for 3D painting. In *ACM Transactions on Graphics*, 30(4), 28:1–28:10.
- Baran, I., Schmid, J., Siegrist, T., Gross, M., and Sumner, R. W. 2011. Mixed-order compositing for 3D paintings. In *ACM Transactions on Graphics*, 30(6), 132:1–132:6.

Another publication concerned with the research presented in Chapter 4 of this thesis has been conditionally accepted to the *ACM Transactions on Graphics* and is currently in revision.

CHAPTER 2

Stroke Based Rendering

In this section, we present various improvements to stroke-based rendering, with a focus on scenarios where strokes have at least a partial representation in primary space (3D). In a loose definition, stroke-based rendering encompasses all image generation techniques where paint strokes are used as rendering primitives, in contrast to rendering geometric surfaces, for example. This concept has been successfully applied in image and video stylization, where a sequence of input images (photographs, video frames, rendered images) is processed to create a painterly depiction of the images. Such methods, some of which are reviewed in Section 1.3, typically work exclusively in the secondary space (as described in Section 1.3) and do not require or incorporate any primary space information like scene geometry or 3D position.

In this thesis, however, we focus on stroke-based rendering for the direct stylized depiction of three-dimensional virtual scenes. In our setting, paint strokes possess an abstract representation in the virtual 3D scene that consists of at least positional information of the stroke, which is transformed to the screen before rendering. The advantage of this approach is that manipulation in the primary space, such as camera motion or shape deformation, leads to a direct and unambiguous result in the secondary space representation. In more loose terms, the image can automatically be "re-painted" given the knowledge of the new configuration of the virtual scene.

Even though the position and movement of a stroke is governed by its pri-

mary space representation, we would like rendering to be a mostly two dimensional, secondary-space process in order to generate images that look as if they were painted in 2D. We propose to achieve a painterly appearance by ignoring much of the primary space information when creating the secondary space image. This feature is the most salient distinction between stroke based rendering and other 3D painting techniques, such as texture painting or Maya Paint Effects [Paint Effects, 2011].

Certain properties of the primary space may need to be incorporated in order to be able to convey the scene layout in a convincing and understandable way. For example, we use depth information to scale the brush width with the size of an object (Section 2.3.3) and to establish the visibility order of surfaces in the scene (Section 2.2). On the other hand, we consciously refrain from using information about the orientation of brush strokes and underlying proxy geometry, and we do not use 3D to 2D projection for anything but the centerline of a stroke. This decision is largely a stylistic choice based on our goal to create images that look as close to digital 2D painting as possible. If one is willing to relax on this ideal, however, there is a wealth of primary space data that could be used to create interesting and useful stroke effects, such as the orientation of the brush stroke with respect to the viewer or even the velocity of strokes in an animated setting.

2.1 Brush Stamping for 3D Paint Strokes

A stroke-based rendering system requires a brush model that defines how paint strokes are rendered. This model can be as simple as a single elongated and oriented brush texture per stroke [Meier, 1996, Litwinowicz, 1997, Hays and Essa, 2004, Lu et al., 2010]. However, for methods which aim to render paint strokes that are significantly longer than they are wide, the better choice is to represent strokes with a geometric curve that represents the path, or "centerline", of the paint stroke, and a method to generate the depiction of a brush stroke along that path. In the research community, the predominant method to render brush strokes from a curve is to create a geometric skeleton around the curve and render this "ribbon" with a texture or a procedural shader that creates the appearance of a brush stroke [Hsu and Lee, 1994, Northrup and Markosian, 2000, Kalnins et al., 2002, Katanics and Lappas, 2003]. This process is illustrated in Figure 2.1.

In commercial 2D digital painting packages, on the other hand, the "brush stamping" approach is by far the most commonly used brush model. In this method, which was pioneered by Alvy Ray Smith [Smith, 1982], a paint stroke is rendered by repeatedly blending a brush texture into the image along the stroke's curve (Figure 2.2). This process is conceptually simple and thus easily

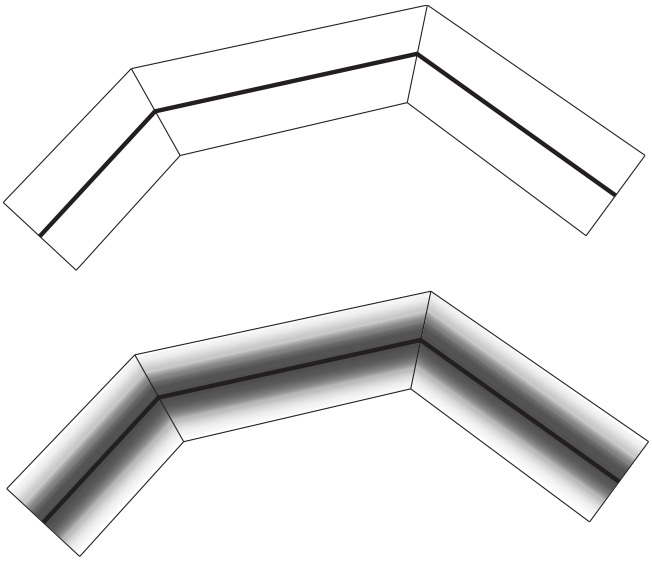


Figure 2.1: In stroke based rendering, a popular way to draw brush strokes is to create a geometric skeleton (or "ribbon") around the stroke centerline (top image) and fill it with a procedural shader or texture (bottom image).

understandable and customizable by non-technical users, yet at the same time it trivially handles paint strokes of arbitrary length and shape, which can be difficult to achieve with ribbon-based models. On the other hand, the stamping method faces issues with respect to temporal coherency when the shape and length of paint strokes change over time. This issue may not be of concern for 2D digital painting, but it readily arises in stroke-based rendering of animations.

In this section, we present a simple method for rendering tree-dimensional paint strokes with 2D brush stamping, and various extensions to improve the variability and flexibility of the resulting paint strokes.

2.1.1 Definitions

Let a paint stroke S be defined by a tuple

$$S = \langle \mathcal{P}, \mathcal{T}, \mathbf{c}, \rho, \sigma, s \rangle, \tag{2.1}$$

where $\mathcal{P}$ is a sequence of points that defines the stroke path, $\mathcal{T}$ is a brush stamp texture, ρ is the desired screen space width of the brush in pixels, σ is the desired density of brush stamps along the stroke path as a fraction of the brush stamp size, and the stroke number s is an integer that establishes an ordered sequence

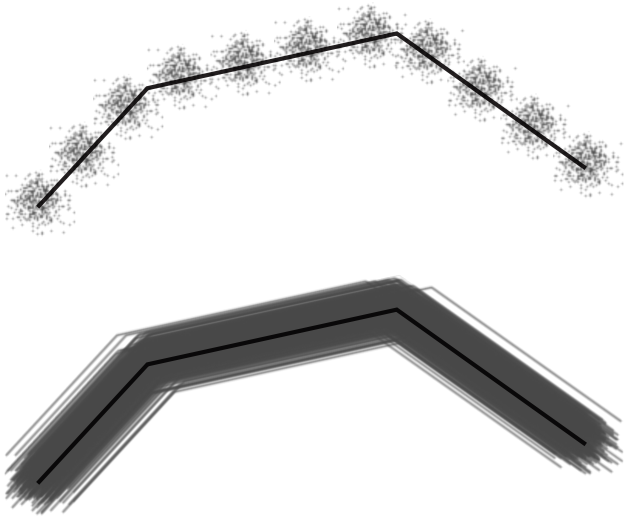


Figure 2.2: In the brush stamping approach, a roughly circular brush texture is blended into the image repeatedly along the stroke path (top image). If the stamps are placed densely enough, the appearance of a coherent brush stroke is created (bottom image).

among all strokes in a painting. Each stroke point $P_i \in \mathcal{P}$ with $i \in [1, N]$ is defined by another tuple

$$P_i = \langle \mathbf{p}_i, r_i \rangle, \tag{2.2}$$

in which $\mathbf{p}_i$ signifies the stroke point's location in 3D and r_i represents the projection of the brush width ρ into world space at position $\mathbf{p}_i$ (which is discussed in detail in Section 2.3.3). We use the notation $\hat{\mathbf{p}}_i$ to designate the screen space position of $\mathbf{p}_i$ in the current view.

The curve that represents the path of paint stroke S is defined by the stroke point positions $\mathbf{p}_i$ in the sense that the curve passes through each stroke point in sequence. In all cases, we model the curve as a polyline: the curve is defined by the linear segments connecting adjacent stroke points in $\mathcal{P}$. However, we do not see any limitation that would prohibit the usage of a higher-order curve representation, such as an interpolating B-spline.

The brush texture $\mathcal{T}$ is a raster graphic of arbitrary size, typically of roughly square format. In our implementation, the color of a paint stroke is exclusively defined by the stroke color $\mathbf{c}$, thus the only purpose of $\mathcal{T}$ is to define an opacity value for each pixel in the texture.

Some notes on terminology:

- We use the words "stamp" and "splat" almost interchangeably in this the-

sis. While "stamp" is used mainly in the abstract context of stamping, with "splat" we mean an actual textured quad that is rendered on screen.

- The word "fragment" is used with its usual meaning in computer graphics. It is a portion of a rendering primitive that falls into one pixel as computed by the rasterizer. Fragments can inherit any of the information about their corresponding strokes and splats, such as stroke number and splat depth.

2.1.2 Stamp Placement

The canonical method for brush stroke rendering with stamping in 2D is to sample the curve of a stroke at uniformly spaced intervals with respect to arc length, and at each sample location composite $\mathcal{T}$ on top of the existing image buffer with alpha blending (for example using the *over*-operator). If the color along a stroke is constant, the order in which the individual samples are processed is irrelevant, since the alpha compositing is commutative if (and only if) the colors to be composited are equal. The width of the rendered brush stroke is determined by the size of the brush texture that is composited into the image. Therefore, $\mathcal{T}$ is scaled such that its diameter matches the desired stroke width, ρ.

The arc-length distance between two samples, which we will refer to as the *spacing* of the brush, is a freely selectable parameter of the stroke that has great influence on the appearance of a stroke. It is the density with which the brush texture is reproduced along the stroke curve, and, due to the alpha compositing, it directly affects the opacity of the stroke. Typically, it is desired that the color density is distributed evenly along the stroke, and thus it follows that the sampling should be performed at uniform intervals.

A simple algorithm that satisfies these requirements is outlined in Algorithm 1. In this algorithm, *spacing* is assumed to be constant and defined in screen space length units (pixels). We will argue later that for more complex stroke based rendering applications, it is better to define spacing with respect to the brush width.

This algorithm suffers from two deficiencies when used to render brush strokes whose original representation lies in 3D space: brush width scaling and end point coherence.

Brush width scaling

In a typical 2D digital painting application, the user chooses a constant width in pixel units for the brush strokes he or she is about to paint. This width may be modulated by pen pressure or jittered along the stroke, but the base value

Algorithm 1 Uniformly spaced brush stamping

```
t ← 0, t_end ← 0
draw 𝒯 at p̂_0
for all P_i ∈ 𝒫 \ {P_N} do
  t_end ← t_end + dist(p̂_i, p̂_{i+1})
  while t + spacing < t_end do
    t ← t + spacing
    draw 𝒯 at curve position t scaled to ρ
  end while
end for
```

never changes. In a 3D painting system, we would like to mimic this behavior by defining the brush width in screen space in the current view as the user paints a stroke. This user-selected paint-time brush width is reflected by the scalar ρ in Equation 2.1.

However, as the view on the 3D painting is changed, brush strokes are expected to scale with respect to their distance from the viewer if a perspective projection is used. A simple example is a straight brush stroke that is moved away from the viewer. As the distance increases, the projection of the brush stroke gets shorter, and to maintain a uniform appearance, its width has to decrease accordingly. This effect can also vary along a stroke: suppose that the straight stroke was painted on a plane that was originally perpendicular to the viewer. If the plane and the stroke on it get tilted with respect to the viewer, one end of the stroke will be closer to the viewer than the other, and thus the width of the stroke should vary accordingly. Both effects are illustrated in Figure 2.3.

In addition, the 3D embedded representation of a brush stroke should not be restricted to be co-planar to the view plane at paint time. As a consequence, the projection of the brush width ρ to the embedded 3D positions must be allowed to vary along the stroke. We therefore compute a world space radius for each stroke point, which is reflected by the r_i in Equation 2.2. These radii are computed after a brush stroke has been applied on the canvas and transformed to 3D: for each stroke point, a line segment that has a length of ρ (the screen space brush width) and originates at the screen space location the stroke point is projected into world space. The length of the projected line segment represents the world space radius of the stroke at that point and is stored in r_i. When rendering the brush stroke from an arbitrary view, r_i can be projected back to screen space in similar fashion and used to determine the adjusted brush width. Since the sampled brush stamp positions in general do not coincide with the stroke points, we propose to use linear interpolation to compute the brush width at any location along the stroke.

As mentioned earlier, the reason for the uniform sampling intervals in Algorithm 1 is found in the desire of a constant density along the stroke. The

density, however, is defined by the relative amount of overlap between two consecutive splats. To keep density constant, the spacing should therefore scale proportionally with the radius of the splats along the stroke. Instead of defining spacing absolutely in terms of pixels (as in Algorithm 1), we define our spacing value σ as a fraction of the brush width. An absolute spacing value is obtained by multiplying σ with the current brush width. Since the projected brush width is not constant over a stroke, neither is the resulting spacing, and thus the sampling intervals during brush stamping are not uniform. Algorithm 2 implements adaptive sampling that takes the varying width into account.

Algorithm 2 Perspectively scaled brush stamping

$t_{next} \leftarrow 0$, $t_{end} \leftarrow 0$
for all $P_i \in \mathcal{P} \setminus \{P_N\}$ **do**
  $t_{end} \leftarrow t_{end} + dist(\hat{\mathbf{p}}_i, \hat{\mathbf{p}}_{i+1})$
  while $t_{next} < t_{end}$ **do**
    $\gamma \leftarrow \frac{t_{end} - t_{next}}{dist(\hat{\mathbf{p}}_i, \hat{\mathbf{p}}_{i+1})}$
    $r_{splat} \leftarrow \gamma \cdot transformToScreen(r_i) + (1 - \gamma) \cdot transformToScreen(r_{i+1})$
    draw $\mathcal{T}$ at curve position t_{next} scaled to r_{splat}
    $t_{next} \leftarrow t_{next} + \sigma r_{splat}$
  end while
end for

End point coherence

The brush stamping technique described so far suffers from severe temporal coherence problems in animations: if the brush strokes undergo 3D transformations, their projections to the screen plane will change in length, and thus the number of splats generated by Algorithm 2 may change from frame to frame. By construction, the algorithm will always place a splat at P_1 (the start of the stroke), but in general it will not hit P_N (the end of the stroke) exactly. As the projection of a brush stroke changes in length, splats will suddenly appear or disappear around P_N, which is perceived as a popping or flickering artifact.

We have found a simple fix to this problem that hides this aliasing problem from the viewer. To ensure that the rendering algorithm always covers the entire length of a stroke, the brush stamping algorithm is extended to always draw a last splat at P_N. In addition, the opacity of the second last splat (i.e. the last splat generated by Algorithm 2) is scaled with the proximity to P_N, such that it vanishes completely if it coincides with P_N. Figure 2.4 illustrates this process.

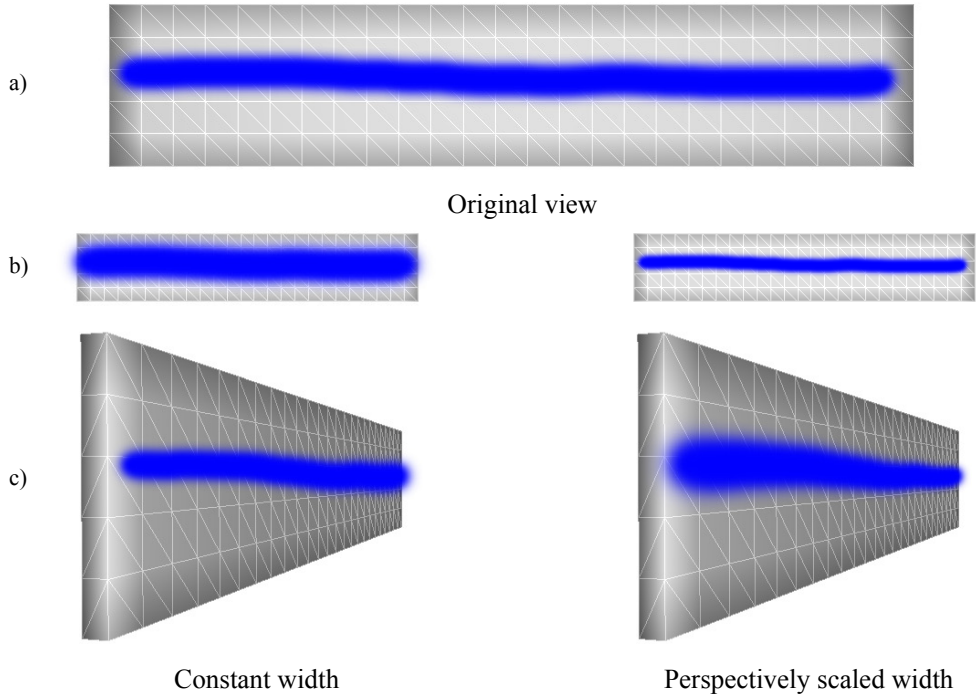


Figure 2.3: The width of a paint brush needs to be scaled according to the perspective transformation. Row a) shows the original view on a single brush stroke. In row b), the brush stroke was moved away from the viewer. If the width of the brush stroke is not scaled (left image), the proportions of the brush stroke change, as does its relative size with respect to the underlying object. Row c) shows that the width needs to be scaled at a sub-stroke resolution, otherwise long strokes exhibit distortions when viewed from steep angles as shown in c).

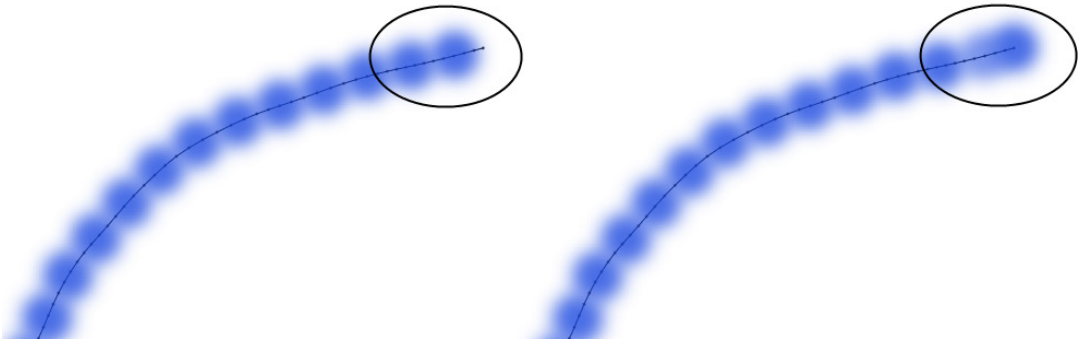


Figure 2.4: The stamp placement algorithms described in Section 2.3.3 do not ensure that the end point of a brush stroke is covered with a stamp (left hand side). This causes the end of the stroke to jump or flicker if the length of the stroke changes during an animation. A simple remedy is to always place a stamp at the last point of a stroke and to scale the opacity of the second last splat to maintain the expected color density (right hand side).

2.1.3 Stroke Transparency

The ability to change the transparency of a brush stroke is an important feature of 2D digital painting and should also be available in 3D painting. With brush stamping, there are two different approaches to controlling transparency:

1. Scale the α-value of each individual stamp before compositing.
2. Scale the α-value that results after compositing all stamps of a stroke.

Since the *over*-operator is not linear in α, the results of the two methods are not equivalent. The difference is shown in Figure 2.5. In Adobe Photoshop, the first property is called *flow*, while the second property is called *opacity*.

Figure 2.5: In the upper stroke, the α-channel of each stamp is multiplied with 0.5 before compositing. In the lower stroke, compositing is done before the resulting α-channel is multiplied by 0.5.

Adjustable *flow* is trivial to implement, since a constant scaling factor for the α-value of each primitive is typically supported by rendering architectures. Per-stroke *opacity* is much harder to obtain, because it requires a modification of the α-value *after* compositing all stamps of a stroke. This effect cannot be achieved by a constant α-factor for individual splats. Depending on the stamp spacing and the brush texture, the user can approximate *opacity* by picking the right *flow* factor. However, this factor can not be estimated automatically in a reliable fashion, because it depends both on the spacing and on the alpha values within the brush texture.

The obvious solution to obtain true per-stroke *opacity* is to composite each stroke into a separate buffer before scaling its α-value and blending it into the framebuffer. Unfortunately, when used in the context of 3D painting, this pre-buffering of strokes poses a number of challenges, which will be presented in detail in the discussion of future work (Section 2.3.3).

2.1.4 Pressure Dynamics and Parameter Jittering

In addition to the pen position, graphic tablets also record the pressure that is exerted on the tablet. This pressure value can be used to modulate brush parameters, typically brush width and *flow*, in 2D digital painting. Since our brush stamping routine (Algorithm 2) already supports brushes of non-constant width and *flow*, these effects are trivial to implement in our brush model. We simply store the pressure value at each stroke point along with the other information in Equation 2.2 and interpolate it linearly to the splat locations.

A simple method to roughen up the appearance of brushes is to randomly jitter the brush parameters along the stroke. In addition to brush width and *flow*, it is effective to randomly rotate the brush stamps around their center point, as is illustrated in Figure 2.6. These features are widely available in 2D digital painting. In order to obtain the same effects with temporal coherence in stroke-based rendering, we store a random number seed for each stroke along with the other information in Equation 2.1. The bigger challenge, however, lies in the fact that since the brush stamping routine operates in screen space, an individual splat is not tied to a particular location on the stroke. As the screen space length of a brush stroke changes, the splats "slide" over the brush stroke, which is essentially the same issue that leads to temporal coherence problems at the end of a stroke discussed in Section 2.3.3. As long as each splat has the same appearance, the sliding effect is not noticeable except at the end of a stroke. Unfortunately, it becomes very apparent when brush parameters are jittered. The only temporally coherent solution we have found to this problem is to fix the splat locations in 3D for brush strokes with jittering effects. This solution comes at the cost of forgoing the constant color density along the stroke, however, so it presents the user with a stylistic trade-off.

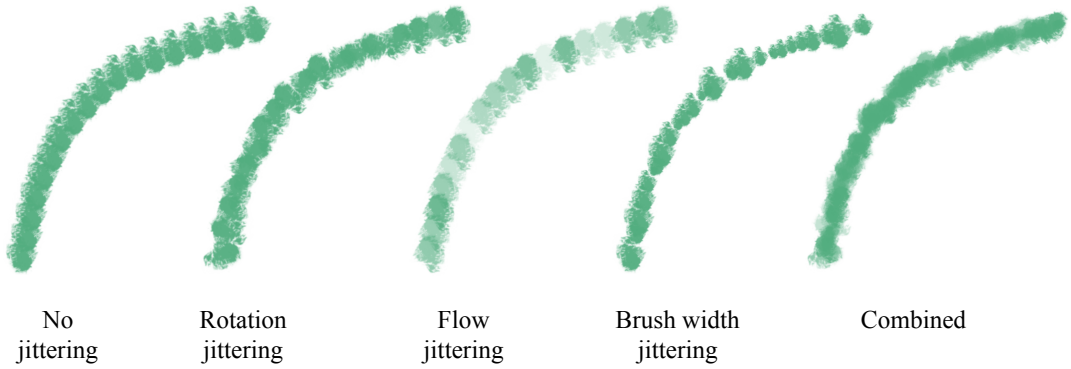


Figure 2.6: The parameters of the individual brush splats can be jittered to create more interesting brushes. The rightmost example combines rotation, flow, and brush width jittering.

2.1.5 Canvas Texture

Another brush feature that enjoys considerable popularity in 2D digital painting is canvas texturing. An arbitrary alpha texture, often resembling the texture of paper, is aligned with the canvas and used as an additional alpha modulator when compositing elements (such as brush stamps) onto the canvas. While the brush texture $\mathcal{T}$ is always co-located with each splat, the canvas texture remains fixed with respect to the canvas origin. When compositing a fragment of a splat into the canvas, the corresponding alpha values of $\mathcal{T}$, the canvas texture, and the user-chosen *flow* value are multiplied to compute the alpha value of the fragment.

When this technique is applied to moving brush strokes, the so-called "shower door" effect is perceived: since the brush strokes are moving while the canvas texture remains static, the brush strokes appear to be seen through a distortion layer, like a rippled shower door. This effect is usually disturbing to the viewer and therefore undesirable. Instead, the viewer expects the canvas texture to move along with the stroke to maintain the consistent appearance of a roughened brush stroke.

In the setting of 3D painting, the 2D projection of brush strokes may undergo complex transformations when the view is changed or the brush strokes are animated. It is a design choice to what extent these transformations should be reflected in the canvas texture of a brush stroke, as it mainly influences the visual style of the resulting animations. A common approach found in previous work is to use texture advection methods to account for arbitrary motion and deformation of objects [Bousseau et al., 2007]. Such methods compute the texture transformations incrementally based on the previous and/or future frames in the animation, which is problematic in an interactive system where there is no meaningful history let alone future of an animation.

In our situation, a method where the transformations only depend on the current state of the scene and possibly a designated rest state is preferred. We also do not want the canvas texture to become arbitrarily distorted because we would like to maintain the original visual qualities of the texture. Breslav and colleagues faced a similar challenge when placing and advecting 2D patterns along with an underlying 3D object [Breslav et al., 2007]. Inspired by Horn [Horn, 1987], they propose computing a similarity transformation consisting of rotation, translation, and uniform scaling that best maps a set of sample points from one frame to the next and applying this transformation to the pattern. We have found that this method is an ideal basis for the problem of per-stroke canvas texture transformation. In their implementation, transformations are computed on a frame-by-frame basis, but their method is also applicable when the transformation of the sample points is computed with respect to a rest state, as is depicted in Figure 2.7.

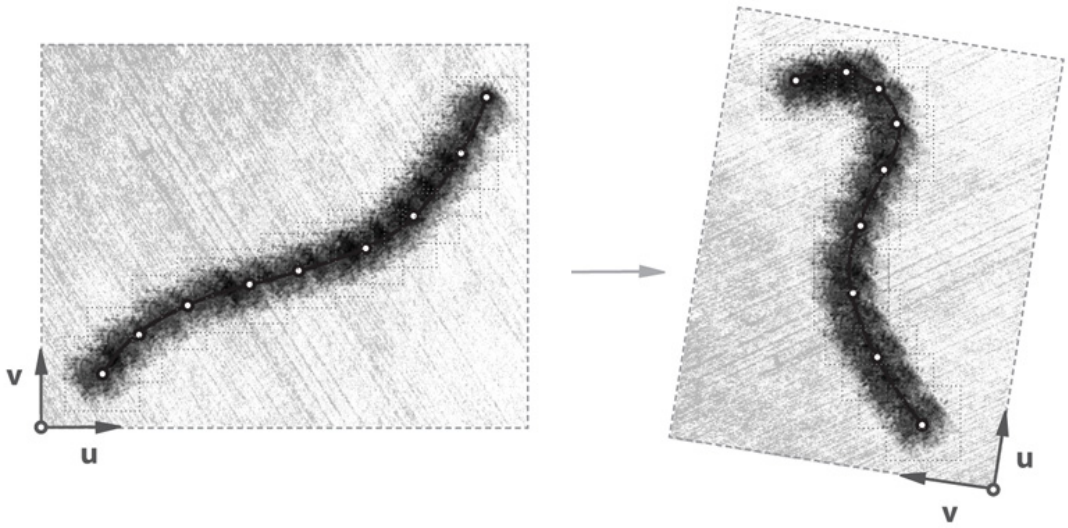


Figure 2.7: This figure illustrates the transformation the canvas texture undergoes from its rest state (left) to the current configuration (right). To avoid distortion, we compute a translation, rotation, and uniform scale that best maps the stroke points from the rest position to the current position.

In our case, each stroke represents a separate geometric entity, and therefore a separate canvas texture transformation is computed for each stroke independently. When a brush stroke is drawn, we store the original screen space coordinates of each stroke point as reference position $\hat{\mathbf{r}}_i$ along with the other data in Equation 2.2. These reference positions are directly used as canvas texture coordinates, so the behavior of the canvas texture in the original paint view is exactly the same as described for the pure 2D case at the beginning of this section. For any other view, a similarity transformation (consisting of translation, rotation, and isotropic scaling) is computed that maps the current screen space stroke points $\hat{\mathbf{p}}_i$ to the reference positions $\hat{\mathbf{r}}_i$ as well as possible. This transformation is then used to transform the corners of the brush splats (generated in the current view) into the canvas texture space.

A 2D rotation together with an isotropic scaling around the origin can conveniently be expressed as a multiplication of two complex numbers. To this end, we use the notation of a 2D position (x, y) interchangeably with that of a complex number $x + yi$. The search for an optimal canvas texture transformation can thus be expressed as an optimization for a complex number $\mathbf{z}$ which minimizes the sum of squared distances between $\mathbf{z}\hat{\mathbf{p}}_i$ and $\hat{\mathbf{r}}_i$. Horn shows that the corresponding optimal translation is obtained if the origin is moved to the

centroid of the data points [Horn, 1987]. This gives us the energy function

$$E = \sum_i \left\| \mathbf{z}(\hat{\mathbf{p}}_i - \overline{\mathbf{p}}) - (\hat{\mathbf{r}}_i - \overline{\mathbf{r}}) \right\|^2$$

with $\overline{\mathbf{p}} = \frac{1}{N} \sum_i \hat{\mathbf{p}}_i$ and $\overline{\mathbf{r}} = \frac{1}{N} \sum_i \hat{\mathbf{r}}_i$.

We take the partial derivatives of E with respect to the two components of $\mathbf{z}$ and set them equal to zero. Solving the resulting equations for $\mathbf{z}$ gives us the least-squares optimal solution

$$\mathbf{z} = \frac{1}{\sum_i \left\| (\hat{\mathbf{p}}_i - \overline{\mathbf{p}}) \right\|^2} \left(\sum_i (\hat{\mathbf{p}}_i - \overline{\mathbf{p}}) \cdot (\hat{\mathbf{r}}_i - \overline{\mathbf{r}}), \quad \sum_i (\hat{\mathbf{p}}_i - \overline{\mathbf{p}}) \times (\hat{\mathbf{r}}_i - \overline{\mathbf{r}}) \right).$$

Here, "$\times$" denotes the two-dimensional cross product $\mathbf{a} \times \mathbf{b} = a_x b_y - a_y b_x$.

Given $\mathbf{z}$, the canvas texture coordinates for any screen space location $\hat{\mathbf{x}}$ of a stroke can be computed directly:

$$(u, v) = \mathbf{z}(\hat{\mathbf{x}} - \overline{\mathbf{p}}) + \overline{\mathbf{r}}.$$

So far, our method for canvas texturing closely follows the technique described by Breslav and colleagues. We have obtained good results with it as long as brush strokes do not take on a direction that is roughly parallel to the view direction, a situation that happens, for example, when a brush stroke on a surface passes around the silhouette when it is roughly perpendicular to the silhouette. In this case, two problems occur:

- As the stroke roughly lines up with the view direction, all stroke points will be projected to a small region on screen, thus causing a large scale factor in the texture transformation.
- The orientation of the stroke with respect to the viewer will change, which causes the texture coordinates to undergo a full 180° rotation.

We avoid both effects by fading to a differently transformed texture as the stroke becomes perpendicular to the view direction. Since paint strokes are not planar in general, we compute a dominant direction for each stroke and compare it to the view direction:

$$\delta = \text{dominant stroke direction} \cdot \text{view direction}.$$

As long as paint strokes do not contain any sharp kinks and are not overly long, it is sufficient to use $\mathbf{p}_N - \mathbf{p}_0$ as the dominant direction. We have found this choice to be sufficient for our experiments, but some applications may require the use of a more sophisticated method, such as PCA.

To avoid excessive scaling, an alternative texture scaling factor is computed from the ratio between the original brush width and the average projected world space radius of the stroke points:

$$s = \frac{1}{N} \sum_i \frac{r_i}{\rho}.$$

The texture transformation is modified to reflect a scaling of s when δ is close to 1. If δ is within a certain user-chosen transition range $[\delta_0, \delta_1]$ with $\delta_0, \delta_1 \in [0, 1]$ and $\delta_0 < \delta_1$, the scaling factor is linearly interpolated between the original scaling $|\mathbf{z}|$ and s:

$$\mathbf{z}' = f(\delta)\mathbf{z},$$
$$f(\delta) = \begin{cases} 1 & \text{if } \delta \leq \delta_0 \\ 1 - \frac{\delta - \delta_0}{\delta_1 - \delta_0} \left(1 - \frac{s}{|\mathbf{z}|}\right) & \text{if } \delta_0 < \delta < \delta_1 \\ \frac{s}{|\mathbf{z}|} & \text{if } \delta \geq \delta_1. \end{cases}$$

The 180° rotation of the texture is necessary to represent the change of orientation of the stroke, but it tends to take place within only a few degrees of the angle between the dominant stroke direction and the view direction, and therefore it happens very abruptly in an animation. We avoid this unattractive effect by using a texture transformation that consists only of scaling and translation when δ is close to 1:

$$\mathbf{z}'' = |\mathbf{z}'| \ (+ 0i).$$

In the fragment shader, the canvas texture is looked up both at the location computed with $\mathbf{z}'$ and with $\mathbf{z}''$ to retrieve $\alpha_{\mathbf{z}'}$ and $\alpha_{\mathbf{z}''}$, respectively. If δ is within another user-chosen transition range $[\gamma_0, \gamma_1]$ with $\gamma_0, \gamma_1 \in [0, 1]$ and $\gamma_0 < \gamma_1$, the two results are blended linearly. Otherwise, only $\alpha_{\mathbf{z}'}$ or $\alpha_{\mathbf{z}''}$ is used, depending on whether δ is close to 0 or 1:

$$\alpha(\delta) = \begin{cases} \alpha_{\mathbf{z}'} & \text{if } \delta \leq \gamma_0 \\ \left(1 - \frac{\delta - \gamma_0}{\gamma_1 - \gamma_0}\right) \alpha_{\mathbf{z}'} + \frac{\delta - \gamma_0}{\gamma_1 - \gamma_0} \alpha_{\mathbf{z}''} & \text{if } \gamma_0 < \delta < \gamma_1 \\ \alpha_{\mathbf{z}''} & \text{if } \delta \geq \gamma_1. \end{cases}$$

In our experiments, we have achieved good results with $\delta_0 = 0.7$, $\delta_1 = 0.9$, $\gamma_0 = 0.85$, and $\gamma_1 = 0.95$. The result of $\alpha(\delta)$ can used to modulate the opacity value of a stroke to achieve a canvas texturing effect that behaves convincingly under view changes. Unfortunately, as discussed in Sections 2.1.3 and 2.3.3, stroke opacity modulation is not trivial for the brush model used in this thesis. The examples shown in Figure 2.8 were obtained using simple stroke pre-buffering, which is prone to exhibit artifacts with more complex scenes.

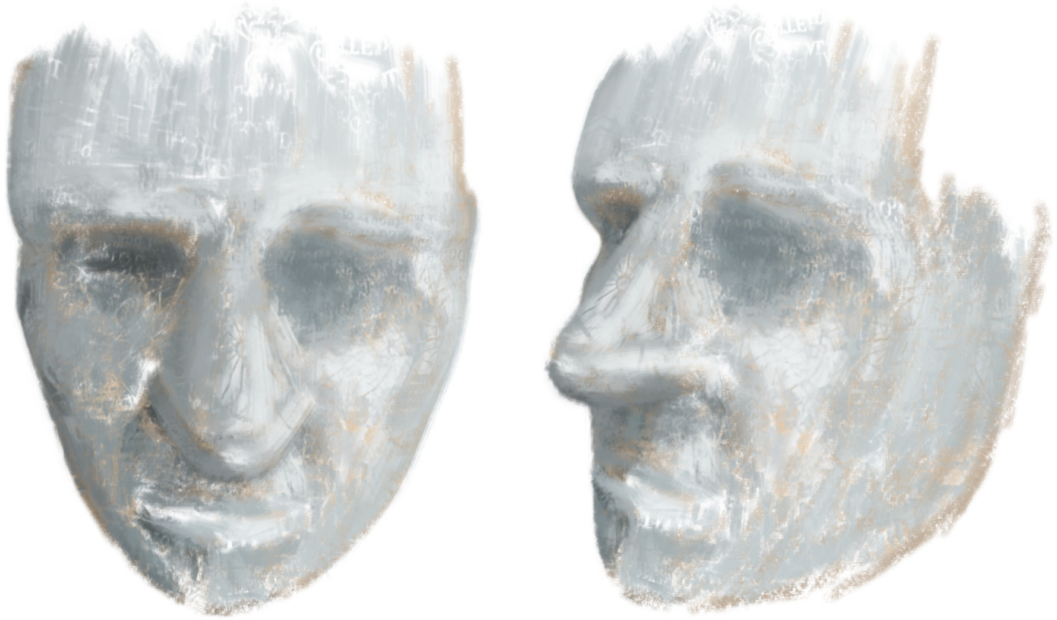


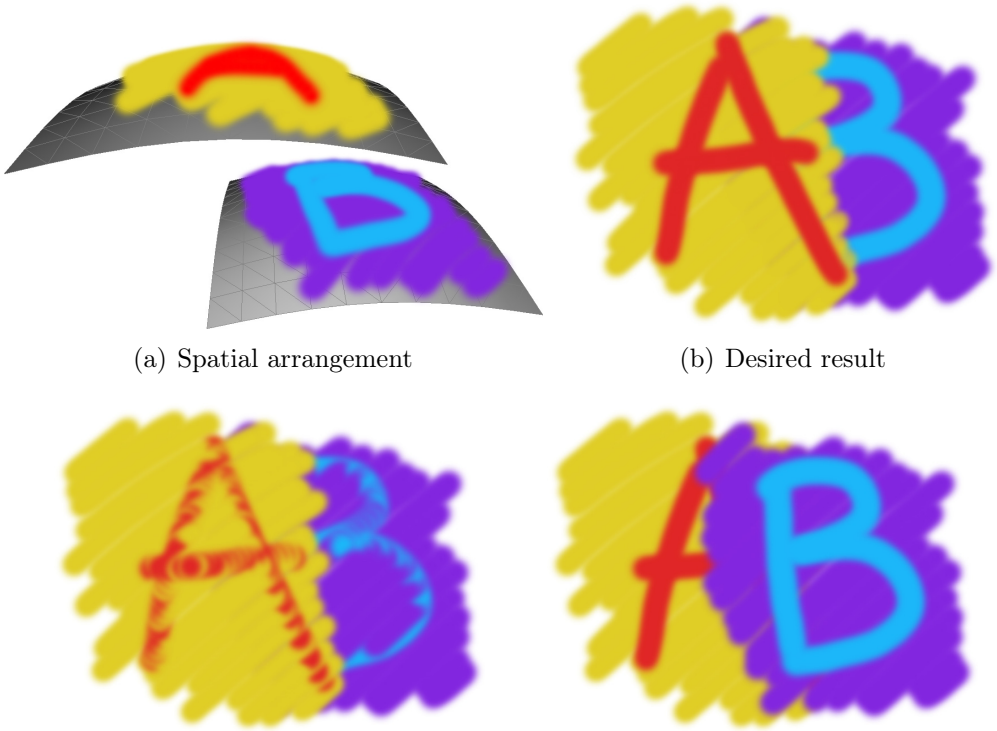
Figure 2.8: The salient texture on the brushes seen in this 3D painting were achieved with our canvas texturing method. Thanks to the view-dependent interpolation of different texture transformations, the resulting animations exhibit strong temporal coherence.

2.2 Paint Order vs. Depth Order

So far, the focus of this chapter has been on how to render a single stroke. Rendering multiple 3D paint strokes exposes a technical dilemma about the order in which the strokes should be composited. In the 2D painting metaphor, when the artist places a new paint stroke, it obscures all previous paint strokes that it overlaps. Such behavior is achieved by compositing in *paint order*. From a 3D point of view, however, strokes that are closer to the viewer should obscure those that are farther away, which amounts to compositing in *depth order*. Compositing purely in paint order negates much of the benefit of 3D painting, as the sense of tangible objects is lost when the view is changed. Compositing purely in depth order, on the other hand, precludes the artist from reliably painting over existing strokes, and thus ignores an important part of the 2D painting metaphor. The conflict is illustrated in Figure 2.9.

The inventors of Deep Canvas [Katanics and Lappas, 2003] first articulated the desire for mixed-order compositing: fragments that, in the artist's mind, belong on the same surface should be composited in paint order, while those that belong on different surfaces should be composited in depth order. Unfortunately, assigning each fragment to a specific surface is often impossible: the stroke that generated the fragment may span several surfaces, may self-occlude, or may not even conform to a surface at all. The guideline Deep Canvas adopts is therefore to choose the appropriate ordering based on a depth tolerance d: fragments whose depths are within d of each other are assumed to lie on the same surface and are composited in paint order, but fragments that are farther apart are composited in depth order. The biggest challenge in realizing this concept is to ensure that the transitions between paint order and depth order composition do not result in spatial or temporal incoherences.

Conflicting orders of compositing have recently been considered by Bruckner et al. [Bruckner et al., 2010] for illustrating 3D layers. Their goal is, in some sense, a transpose of ours: we aim to composite fragments close together in paint order and fragments far apart in depth order, while they composite user-specified adjacent layers in depth order and the results in layer order. Their method works well for illustrative rendering and shares some technical similarity with ours. The problem they are solving is more restricted than ours: in their setting, the user specifies which layers are composited in which order and this order cannot change continuously, while our method needs to smoothly transition between compositing in depth order and in paint order.



(a) Spatial arrangement (b) Desired result

(c) Strict depth order (d) Strict stroke order

Figure 2.9: The compositing order conflict illustrated on a simple example. The sequence in which the strokes were painted is: yellow, red, purple, blue. (a) illustrates the spatial arrangement of the paint strokes (the surfaces are only shown for visualization). (b) is the desired compositing result. (c) was composited in strict depth order, which ignores the paint order and leads to a kind of z-fighting among brush splats at similar depths. (d) was composited in strict paint order, which leads to an incorrect depiction of the scene arrangement (the A-plane should be shown on top of the B-plane).

2.2.1 Compositing Background

Digital compositing with the alpha channel was invented by Catmull and Smith [Smith, 1995]. Wallace [Wallace, 1981] wrote down the equation for the *over* operator, while Porter and Duff [Porter and Duff, 1984] introduced premultiplied alpha and described the algebra of compositing. Because the over operator is not commutative, the order in which fragments are composited matters, leading to sorting algorithms for real-time rendering [Mammen, 1989] and techniques for allowing the user explicit control over the order [McCann and Pollard, 2009].

In the presence of a well-defined ordering, fragments or layers are typically composited using the over operator. The formula for the over operator depends on how color and transparency are represented. Given two fragments whose colors and opacities are (h_1, α_1) and (h_2, α_2), the over operator (denoted by $\oplus$) is:

$$\begin{aligned}(h_1, \alpha_1) \oplus (h_2, \alpha_2) = \\ = \left(\frac{h_1\alpha_1 + h_2(1-\alpha_1)\alpha_2}{\alpha_1 + (1-\alpha_1)\alpha_2}, \alpha_1 + (1-\alpha_1)\alpha_2 \right).\end{aligned}$$

It is common to use the premultiplied-alpha representation, storing $c = h\alpha$ instead of h. This simplifies the over operator to

$$(c_1, \alpha_1) \oplus (c_2, \alpha_2) = (c_1 + (1-\alpha_1)c_2, \alpha_1 + (1-\alpha_1)\alpha_2).$$

In this exposition, we will work with the premultiplied-alpha representation, treating c as a 3-vector. In our implementation, we actually store $\beta = 1 - \alpha$ instead of α, which further simplifies the over operator to:

$$(c_1, \beta_1) \oplus (c_2, \beta_2) = (c_1 + \beta_1 c_2, \beta_1\beta_2).$$

From the above expression, it is easy to see that $\oplus$ is associative, but not commutative.

2.2.2 Problem Analysis

To analyze the problem systematically, consider a pixel to which n fragments are rasterized. Each fragment has a color c_i, opacity α_i, depth z_i, and stroke number s_i, and we write $\mathbf{f}_i$ to denote the entire fragment $(c_i, \alpha_i, z_i, s_i)$. The fragments are given in depth order, starting with the closest to the viewer, so $z_i \leq z_{i+1}$. We also assume that the stroke numbers s_i range between 1 and n and that no two fragments in a pixel have the same stroke number.

We now describe the properties a function $C(\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_n)$ needs to satisfy to be a good compositing function for our application. These properties express our

high-level goals: fragments close in depth should be composited in paint order while fragments further apart should be composited in depth order, and spatial and temporal coherence should be maintained. In expressing them mathematically, we strive to balance generality and the ease with which we can reason about them, but we do not attempt to formulate an exhaustive set of properties for the problem.

Because we treat fragments at the same depth as being on the same surface, we would like to composite fragments at the same depth in paint order:

Property 1. *If two fragments i and $i+1$ have the same depth and are adjacent in paint order, i.e., $z_i = z_{i+1}$ and $s_i = s_{i+1} + 1$, replacing them with their composite in paint order should not change the final output for the pixel:*

$$C(\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_n) = C(\mathbf{f}_1, \ldots, \mathbf{f}_{i-1}, \mathbf{f}_i \oplus \mathbf{f}_{i+1}, \mathbf{f}_{i+2}, \ldots, \mathbf{f}_n).$$

We treat fragments separated in depth as being on different surfaces and would like to composite them in depth order. Suppose that the user specifies a distance d such that fragments farther than d apart in depth are considered to be on different surfaces. Then, compositing them in depth order should not change the result:

Property 2. *If for some i, $z_{i+1} \geq z_i + d$, then:*

$$C(\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_n) = C(\mathbf{f}_1, \ldots, \mathbf{f}_i) \oplus C(\mathbf{f}_{i+1}, \ldots, \mathbf{f}_n).$$

A stroke whose alpha smoothly fades to zero towards its borders can nevertheless cause sharp visible edges when composited with other strokes (see e.g., Figure 2.15, bottom left). To avoid these edges, we require that a fully transparent fragment have no effect:

Property 3. *If for some i, $\alpha_i = 0$, then:*

$$C(\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_n) = C(\mathbf{f}_1, \ldots, \mathbf{f}_{i-1}, \mathbf{f}_{i+1}, \ldots, \mathbf{f}_n).$$

So far, we can construct a function that satisfies all of the above properties simply by sorting the fragments in lexicographical order by depth and then by stroke number. However, during animation, α's, depths, and colors may change, and popping must be avoided to obtain a nice rendering:

Property 4. *The mixed-order compositing function C must be continuous in all of the c_i's, α_i's, and z_i's.*

2.2.3 Existing Techniques

Not every function C that satisfies these properties is necessarily a good compositing function. For instance, C may exhibit undesirable behavior when all $z_i - z_{i-1}$ approach $d/2$ because this configuration is sufficiently far from the premises of Properties 1 and 2. Nevertheless, we have found in our experiments that in natural candidates for C, artifacts can be explained in terms of violations of these properties.

Meier [Meier, 1996] simply composites the strokes in depth order, which violates Properties 1 and 4. Luft and Deussen [Luft and Deussen, 2006] propose a blurred depth test for smooth compositing. Their goals differ from ours in that they only aim for improving temporal coherence but do not need to deal with conflicting compositing orders. Property 1 therefore does not apply. They also do not support user-specified alpha transparency, so Property 2 is trivially satisfied and Property 3 does not apply. The use of depth-dependent compositing in that method leads to a violation of Property 4, resulting in popping artifacts in their animations. For depth order, the method of Bruckner et al. [Bruckner et al., 2010] satisfies Properties 2–4, but it also is not designed to take paint order into account.

Deep Canvas clusters the fragments by z and composites each cluster separately using a combination of depth and paint order [Daniels et al., 2001]. As we understand it, this method satisfies Property 2, but the clustering is sensitive to z and can be changed by a zero-α fragment, violating Properties 3 and 4. We experimented with other methods that use clustering (including soft clustering to maintain continuity) to determine distinct surfaces, but we could not simultaneously satisfy Properties 1, 3, and 4.

In Sections 2.2.4 and 2.2.5 we present two new methods to reconcile paint and depth order in 3D painting and discuss them with respect to the desired properties.

2.2.4 Depth Offset Method

Our first compositing strategy is inspired by the layering of color in reality. Since a layer of paint has a physical thickness, subsequent layers that cover it up will have a slightly increased distance from the base surface. If all layers are composited in increasing order of this distance, the proper paint order is reproduced.

This effect can be transferred to our 3D painting setting: for each fragment to be composited for a given pixel, a modified depth value z'_i is computed according to

$$z'_i = z_i + c \cdot s_i \cdot dir_i. \tag{2.3}$$

The fragments are then sorted according to z_i' and composited in that order. Loosely speaking, this method translates the paint order into a depth offset. A constant rendering parameter c scales the magnitude of the depth offsetting. The offset must be large enough to encompass the largest depth difference between fragments of the same surface. The scalar dir_i represents the scalar projection of the desired offset direction in space to the view ray of the current pixel. To mimic the effect of physical layers of paint, this direction should be the normal of the fragment's underlying surface (if such a surface exists). In practice, however, we have experienced the best result when simply shifting the fragments into the direction of the viewer with $dir_i = 1$. Figure 2.10 illustrates the effect of depth offsetting.

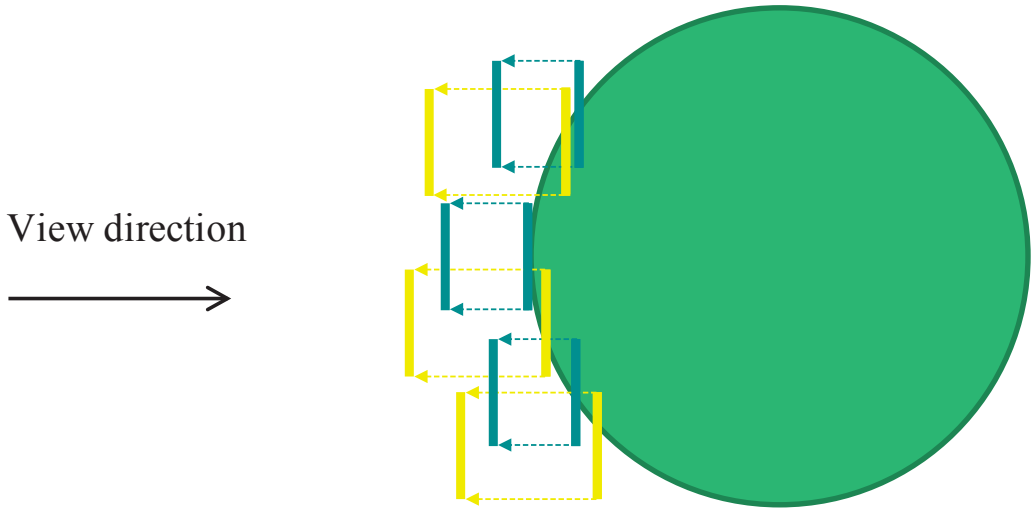


Figure 2.10: The goal of the depth offset method is to turn the paint order on a surface into a difference in depth. In the example shown here, the stroke that is represented by the yellow splats was painted after the blue one. The yellow splats therefore receive a larger depth offset and will be visible on top of the blue ones in the modified depth order. Note that the offset depth is only used for computing the order of rendering, the actual position of the splats is never modified.

Since we assume that all fragments of a splat are at the same depth z_i, the depth offset can be computed once per splat and a global rendering order for all splats can be established. Note that the offset depth value z_i' is only used for computing the rendering order, not for the location of the splat (which is still at the original depth z_i). This approach maps very well to current GPUs and yields a minimal impact on performance, since it does not require any custom per-pixel processing. Unfortunately, however, the method is not free of artifacts,

as it does not satisfy all desirable properties. Since the depth offset depends on the stroke number s_i, this method does not satisfy Property 2: large differences in s_i will lead to fragments of distant surfaces poking through closer ones. Due to the discrete compositing order, Property 4 is also violated.

2.2.5 Mixed-Order Compositing

While the depth offset method is simple and efficient, it violates some of the desired properties presented in Section 2.2.2, which is occasionally revealed in unpleasant artifacts in the output. Our second solution to the problem is a true per-pixel compositing method that satisfies all desired properties and generates smoothly varying colors in all situations. The core idea of our mixed-order compositing algorithm is to replace the color of each of the fragments with the result of compositing nearby fragments in paint order, and then composite the fragments with replaced colors in depth order. While this idea is conceptually simple, its implementation requires careful attention to ensure continuity and good performance.

The user specifies a global constant, d, so that fragments farther than d apart only composite in depth order. We therefore define the function $S(z) = (S_c(z), S_\alpha(z))$ that is the result of compositing all fragments with depths strictly between $z - d/2$ and $z + d/2$ in paint order. When there are no fragments between $z - d/2$ and $z + d/2$, we define S to be the identity color, $(\mathbf{0}, 0)$. S is a piecewise-constant function with discontinuities at $z_i + d/2$ and $z_i - d/2$. If we assign a new color to each fragment using $S(z_i)$, we would not have continuity with respect to z_i's. Instead, we smooth $S(z)$ in depth by convolving it with a box filter of width γd, where γ, with $0 < \gamma \leq 1$, specifies how much smoothing is performed. We compute the colors and alphas as:

$$(c'_i, \alpha'_i) = \frac{1}{\gamma d} \int_{z_i - \gamma d/2}^{z_i + \gamma d/2} S(z)\, dz.$$

Note that because the colors are premultiplied with alphas, this integral is correctly weighted by alpha. We replace the fragment colors, while keeping their original alpha values, setting $c''_i = c'_i \alpha_i / \alpha'_i$. Furthermore, because $S_\alpha(z) \geq \alpha_i$ over the range of integration, we have $\alpha_i \leq \alpha'_i$, and the division is well-behaved for nearly-transparent fragments. The final output is $C(\mathbf{f_1}, \ldots, \mathbf{f_n}) = (c''_1, \alpha_1) \oplus \cdots \oplus (c''_n, \alpha_n)$. Although the final output is composited in depth order, C does not exhibit discontinuities when the depth order changes because two fragments at the same depth will have the same replacement color.

Proofs

In this section, we show that the mixed-order compositing function C satisfies the desired properties presented in Section 2.2.2.

1. *Stroke order:* If fragments i and $i+1$ have the same depth, then compositing them in stroke order leaves $S(z)$ unchanged. Therefore, the replacement colors c'' for all other fragments remain the same. Let $(c_x, \alpha_x) = (c_i, \alpha_i) \oplus (c_{i+1}, \alpha_{i+1})$ be the result of compositing these fragment in stroke order. Because replacement colors are only a function of depth, we have $(c'_x, \alpha'_x) = (c'_i, \alpha'_i) = (c'_{i+1}, \alpha'_{i+1})$. Since the replacement colors are the same, up to the premultiplied alpha factor, $(c''_x, \alpha_x) = (c''_i, \alpha_i) \oplus (c''_{i+1}, \alpha_{i+1})$, and therefore the final composite result is unchanged.

2. *Depth order:* If $z_{i+1} > z_i + d$, then for $z \leq z_i + d/2$, $S(z)$ only depends on fragments up to i and for $z > z_i + d/2$, $S(z)$ only depends on fragments $i+1$ and after. Because $\gamma \leq 1$, the replacement color for all fragments up to i does not depend on $S(z)$ for any $z > z_i + d/2$. Similarly, the replacement color for fragments $i+1$ and after does not depend on $S(z)$ for any $z \leq z_i + d/2$. So replacement colors for fragments up to i only depend on the parameters of fragments up to i, and similarly for fragments $i+1$ and after. Therefore these two groups can be mixed-order composited separately and composited in depth order without affecting the outcome.

3. *Zero alpha:* A fragment with $\alpha_i = 0$ has no effect on $S(z)$ and does not contribute to the final composite and may therefore be removed without changing the result.

4. *Continuity:* The continuity of C in colors and alphas is clear: the over operator is continuous in color and α and which fragments are composited in what order only depends on the depths. With respect to the depths, we prove that if z_i changes by a small ϵ, while all other z's are held constant, then the change in the value of C is bounded by a function that approaches zero as $\epsilon \to 0$ and that does not depend on other variables. This is sufficient to prove that C is continuous in all depths simultaneously. The argument is technical, but the idea is simple: we bound the change in each step of the computation of C individually.

   Without loss of generality, we analyze what changes when z_i changes to $\hat{z}_i = z_i + \epsilon$, where $\epsilon > 0$. Also assume that $\epsilon^{2/3} < \gamma d/4$ and $\epsilon < 1$. First of all, $S(z) \neq \hat{S}(z)$ only at $z \in [z_i - d/2, z_i + \epsilon - d/2] \cup [z_i + d/2, z_i + \epsilon + d/2]$, so $\frac{1}{\gamma d} \int_{-\infty}^{\infty} |S(z) - \hat{S}(z)|_\infty \leq \frac{2\epsilon}{\gamma d}$. Therefore, for $j \neq i$, $|(c'_j, \alpha'_j) - (\hat{c}'_j, \hat{\alpha}'_j)|_\infty \leq \frac{2\epsilon}{\gamma d}$.

For fragment i,

$$\begin{aligned}
|(c'_i, \alpha'_i) - (\hat{c}'_i, \hat{\alpha}'_i)|_\infty &= \\
&= \frac{1}{\gamma d}\left|\int_{z_i-\gamma d/2}^{z_i+\gamma d/2} S(z)\,dz - \int_{\hat{z}_i-\gamma d/2}^{\hat{z}_i+\gamma d/2} \hat{S}(z)\,dz\right|_\infty \leq \\
&\leq \frac{1}{\gamma d}\left|\int_{z_i-\gamma d/2}^{z_i+\gamma d/2} S(z)\,dz - \int_{\hat{z}_i-\gamma d/2}^{\hat{z}_i+\gamma d/2} S(z)\,dz\right|_\infty + \\
&\qquad + \frac{1}{\gamma d}\left|\int_{\hat{z}_i-\gamma d/2}^{\hat{z}_i+\gamma d/2} S(z)\,dz - \int_{\hat{z}_i-\gamma d/2}^{\hat{z}_i+\gamma d/2} \hat{S}(z)\,dz\right|_\infty \leq \\
&\leq \frac{1}{\gamma d}\left|\int_{z_i+\gamma d/2}^{\hat{z}_i+\gamma d/2} S(z)\,dz - \int_{z_i-\gamma d/2}^{\hat{z}_i-\gamma d/2} S(z)\,dz\right|_\infty + \frac{2\epsilon}{\gamma d} \leq \frac{4\epsilon}{\gamma d}
\end{aligned}$$

For each fragment, we have bounded the change of (c', α') by $\frac{4\epsilon}{\gamma d}$, but we need a bound on $|c'' - \hat{c}''|_\infty$. For fragment $j \in \{1, \ldots, n\}$, we split our analysis into two cases, depending on its alpha: if $\alpha_j > 2\epsilon^{1/3}$, then:

$$\begin{aligned}
\left|c''_j - \hat{c}''_j\right|_\infty &= \left|\frac{c'_j\alpha_j}{\alpha'_j} - \frac{\hat{c}'_j\alpha_j}{\hat{\alpha}'_j}\right|_\infty \leq \left|\frac{c'_j\alpha_j}{\alpha'_j} - \frac{\hat{c}'_j\alpha_j}{\alpha'_j}\right|_\infty + \left|\frac{\hat{c}'_j\alpha_j}{\alpha'_j} - \frac{\hat{c}'_j\alpha_j}{\hat{\alpha}'_j}\right|_\infty \leq \\
&\leq \frac{4\epsilon}{\gamma d} + \hat{c}'_j\alpha_j\left|\frac{1}{\alpha'_j} - \frac{1}{\hat{\alpha}'_j}\right| = \frac{4\epsilon}{\gamma d} + \hat{c}'_j\alpha_j\left|\frac{\hat{\alpha}'_j - \alpha'_j}{\hat{\alpha}'_j\alpha'_j}\right| \leq \\
&\leq \frac{4\epsilon}{\gamma d} + \left|\frac{\hat{\alpha}'_j - \alpha'_j}{(2\epsilon^{1/3} - 4\epsilon/\gamma d)2\epsilon^{1/3}}\right| \leq \\
&\leq \frac{4\epsilon}{\gamma d} + \frac{2\epsilon}{\gamma d\epsilon^{2/3}} \leq \frac{4\epsilon + 2\epsilon^{1/3}}{\gamma d} \leq \frac{6\epsilon^{1/3}}{\gamma d},
\end{aligned}$$

where at the end we have used the fact that $\epsilon^{2/3} < \gamma d/4$ and $\epsilon < 1$. If $\alpha_j \leq 2\epsilon^{1/3}$, then $|c''_j|_\infty \leq 2\epsilon^{1/3}$ and $|\hat{c}''_j|_\infty \leq 2\epsilon^{1/3}$ because premultiplied-alpha color components cannot be greater than the α. Therefore $|c''_j - \hat{c}''_j|_\infty \leq 4\epsilon^{1/3}$. In either case, we have just shown that $|c''_j - \hat{c}''_j|_\infty \leq 4\epsilon^{1/3} + \frac{6\epsilon^{1/3}}{\gamma d}$. The over operator with premultiplied alpha returns a linear combination of colors with each coefficient less than or equal to one. Therefore:

$$|(c''_1, \alpha_1) \oplus \cdots \oplus (c''_n, \alpha_n) - (\hat{c}''_1, \alpha_1) \oplus \cdots \oplus (\hat{c}''_n, \alpha_n)|_\infty \leq 4n\epsilon^{1/3} + \frac{6n\epsilon^{1/3}}{\gamma d}$$

The remaining concern is that the depth order may have changed. This is not a problem because fragments close in depth have similar replacement colors. Note that because all elements of $S(z)$ are between 0 and 1, its convolution with a box of width γd is Lipschitz with constant $\frac{1}{\gamma d}$. Let us bound the

change from swapping the order in which two fragments are composited in the final stage:

$$\begin{aligned}
&|(\hat{c}_i'', \alpha_i) \oplus (\hat{c}_{i+1}'', \alpha_{i+1}) - (\hat{c}_{i+1}'', \alpha_{i+1}) \oplus (\hat{c}_i'', \alpha_i)|_\infty = \\
&\quad = \left|(\hat{c}_i'' + (1-\alpha_i)\hat{c}_{i+1}'' - \hat{c}_{i+1}'' - (1-\alpha_{i+1})\hat{c}_i'', 0)\right|_\infty = \\
&\quad = \left|\hat{c}_i''\alpha_{i+1} - \hat{c}_{i+1}''\alpha_i\right|_\infty = \\
&\quad = \left|\alpha_i\alpha_{i+1}\left(\frac{\hat{c}_i'}{\hat{\alpha}_i'} - \frac{\hat{c}_{i+1}'}{\hat{\alpha}_{i+1}'}\right)\right|_\infty = \\
&\quad = \frac{\alpha_i\alpha_{i+1}}{\hat{\alpha}_i'\hat{\alpha}_{i+1}'}\left|\hat{c}_i'\hat{\alpha}_{i+1}' - \hat{c}_{i+1}'\hat{\alpha}_i'\right|_\infty \leq \\
&\quad \leq \left|(\hat{c}_i' - \hat{c}_{i+1}')\hat{\alpha}_{i+1}' + \hat{c}_{i+1}'(\hat{\alpha}_{i+1}' - \hat{\alpha}_i')\right|_\infty \leq \\
&\quad \leq \left|\hat{c}_i' - \hat{c}_{i+1}'\right|_\infty + \left|\hat{\alpha}_{i+1}' - \hat{\alpha}_i'\right| \leq \frac{2}{\gamma d}|\hat{z}_i - \hat{z}_{i+1}|,
\end{aligned}$$

where the last inequality follows from the Lipschitz condition on the convolution of S with the box. Therefore, swapping the order in which final fragments are composited over a distance of at most ϵ changes the result by at most $\frac{2\epsilon}{\gamma d}$. Overall, using the triangle inequality, changing from z_i to $\hat{z}_i$ changes the final result in the L_∞ norm by at most $4n\epsilon^{1/3} + \frac{6n\epsilon^{1/3}+2\epsilon}{\gamma d}$. This is not tight, of course: we conjecture that C is actually Lipschitz in each variable with a constant that does not depend on n.

Algorithm

We now describe an algorithm to compute the function C efficiently in $O(n \log n)$ time and using $O(n)$ memory. The high-level procedure is to explicitly compute $S(z)$ in $O(n \log n)$ time and then define the replacement colors in linear time. Naïve algorithms for both of these tasks run in quadratic time because each distinct value of S can depend on all n fragments and each replacement color can depend on all $\Omega(n)$ distinct values of S. Our algorithm instead sweeps across depth and exploits the problem structure to compute values of S and replacement colors (c', α') incrementally.

We start by using a sort to assign fragments distinct stroke numbers from 1 to n. As previously noted, $S(z)$ only changes at $z_i + d/2$ and $z_i - d/2$, and therefore it only needs to be computed at these locations. At $z_i - d/2$, S is modified to include the new fragment z_i, and at $z_i + d/2$, S changes to no longer include z_i. To accommodate these events, we need a data structure that maintains a subset of the fragments and can add or remove a fragment from the subset efficiently. It also needs to be able to report the composite of this subset in paint order. We use a complete binary tree with n leaves, each node of which stores a color and alpha, initially $(\mathbf{0}, 0)$. Leaf s_i of the tree stores either $(\mathbf{0}, 0)$ or (c_i, α_i), and each

internal node stores the composite of its children in reverse order (because later strokes go on top). The root therefore stores the composite of all of the leaves of the tree in paint order. Inserting or deleting a fragment can be achieved by changing the appropriate leaf from $(\mathbf{0}, 0)$ to (c_i, α_i) or vice versa and updating all of the nodes on the path to the root (Figure 2.11). These updates therefore run in $O(\log n)$ time.

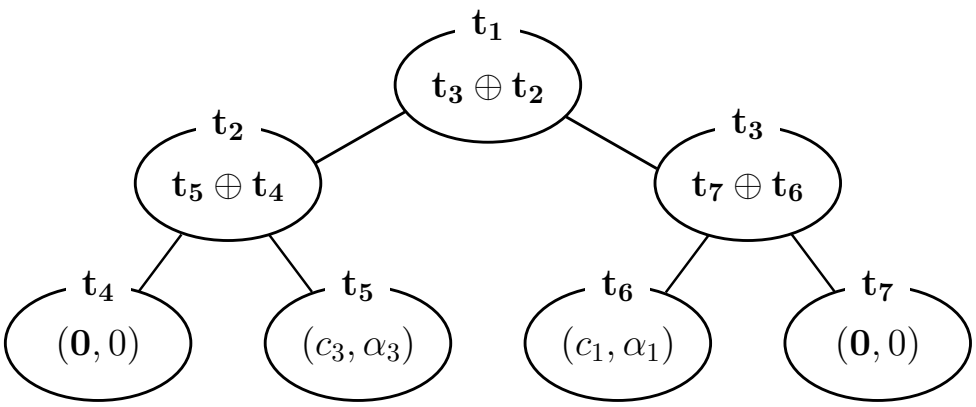


Figure 2.11: The binary tree used to compute $S(z)$. The leaves correspond to four fragments in this pixel. At the time instant shown, fragments 1 and 3 (with $s_1 = 3$ and $s_3 = 2$) are in the window. Suppose fragment 2 enters the window and $s_2 = 4$. This change requires an update of the nodes $\mathbf{t_7}$, $\mathbf{t_3}$, and $\mathbf{t_1}$ for the root to have the correct new $S(z)$.

Now that $S(z)$ is known, we compute its integral over a window of size γd around each fragment (Figure 2.12). Consider the union of the set of discontinuities of S and the points $z_i \pm \gamma d/2$. This union partitions the interval $[z_1 - \gamma d/2, z_n + \gamma d/2]$ into at most $4n$ subintervals. Within such a subinterval I, $S(z)$ is constant by construction. The set of fragments within $\gamma d/2$ of z is also constant and contiguous, consisting of all fragments from z_j to z_{j+k}, for some j and k. The contribution of I to each fragment in this set is $S(I)/\gamma d$ times the length of I. Because k may be as large as n, adding this contribution to all fragments is too expensive. However, if we maintain the integrals as differences between adjacent fragments, $(\Delta c'_i, \Delta \alpha'_i) = (c'_i - c'_{i-1}, \alpha'_i - \alpha'_{i-1})$, we can add the contribution to $(\Delta c'_j, \Delta \alpha'_j)$ and subtract it from $(\Delta c'_{j+k+1}, \Delta \alpha'_{j+k+1})$ in $O(1)$ time. We process all $4n$ subintervals by sweeping over the discontinuities of S and the points $z_i \pm \gamma d/2$ and incrementally maintaining j and k. Before doing the final composite, we compute (c'_i, α'_i) from the deltas by computing the prefix sum.

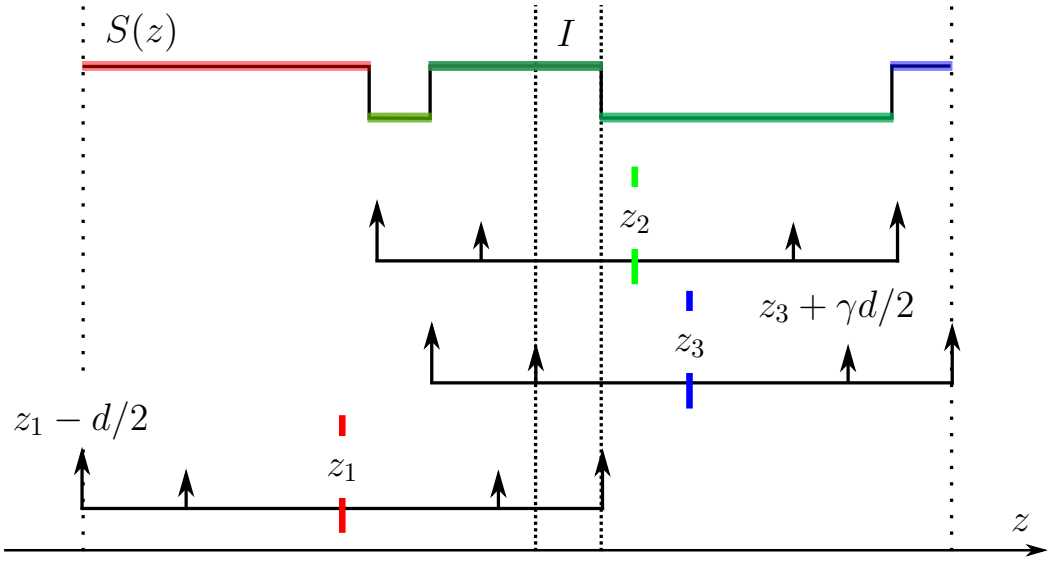


Figure 2.12: This figure illustrates the algorithm for determining the replacement colors. Three fragments are shown with their compositing windows ($\pm d$) and box filter windows ($\pm \gamma d$). The paint order is z_1, z_3, z_2. During the integration, the contribution of the interval I is added to the replacement colors of fragments z_2 and z_3. The vertical separation is used only to make the illustration less cluttered.

Parameters and Temporal Coherence

Mixed-order compositing has two adjustable parameters: the paint order window size d and smoothing width γ. The paint order window size d is crucial for an accurate reproduction of a 3D painting and needs to be chosen according to the dimensions of the scene and the nature of the brushes used. It should be large enough to ensure that strokes on a single surface are composited in stroke order, but no larger to prevent excessive blending in depth. Figure 2.13 shows a comparison of different d values.

Temporal coherence is achieved by blending different compositing orders. While this blending eliminates popping artifacts, it is not always artistically desirable. Our examples are painted in a style that works well with blending, but one can imagine 3D paintings whose artistic style would be compromised by it. The smoothing width γ allows the user to trade temporal coherence for reduced intermediate blending: a lower γ can minimize the frames with undesirable blending, while preventing hard "pops." We used $\gamma = 0.5$ for most of our examples. The effect of γ is shown in Figure 2.14.

In some cases it may be useful to vary d with depth. For example, wide brush

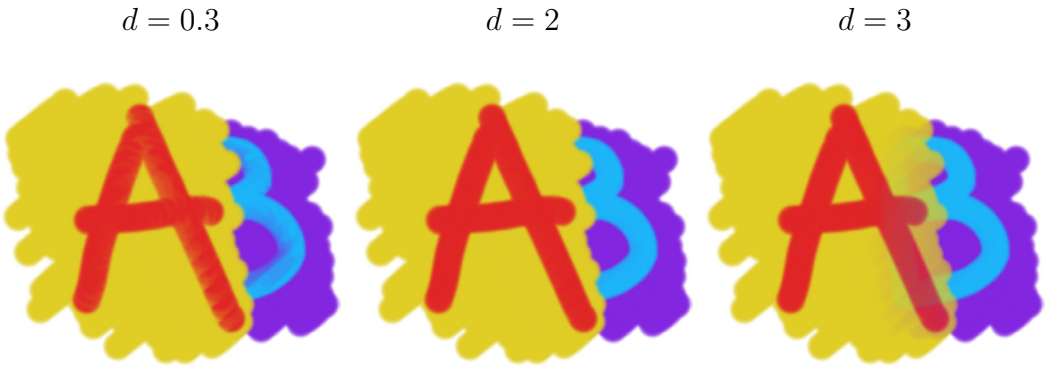


Figure 2.13: Comparison of different values for the paint order window size d. If the window is too small (left), the compositing algorithm is unable to resolve the paint order on a surface properly. If the window is too large (right), surfaces which are close in depth start to become blended in stroke order.

strokes may need to be composited in stroke order over a larger window than thin brush strokes. Our method could be extended to support a variable window size $d(z)$ as long as it satisfies the Lipschitz condition $|d(z_1) - d(z_2)| \leq |z_1 - z_2|$. This condition guarantees that no window completely contains another and allows the efficient computation of replacement colors.

2.3 Discussion

2.3.1 Implementation

The brush stamping algorithm described in Section 2.1 combined with either one of the methods for resolving the paint order on surfaces from Section 2.2 form a complete system for rendering 3D paintings with stroke based rendering. Our implementation is based on C++ and OpenGL with GLSL. The splat generation (stroke sampling) is performed entirely on the CPU. The generated splats are then processed by either the depth offset renderer or the mixed-order compositing renderer.

The depth offset renderer sorts the list of all splats according to the rule given in Section 2.2.4 and then uses conventional OpenGL with alpha blending enabled to render them in the resulting order. Since the visibility of the splats is predetermined by the rendering order, z-buffering is disabled.

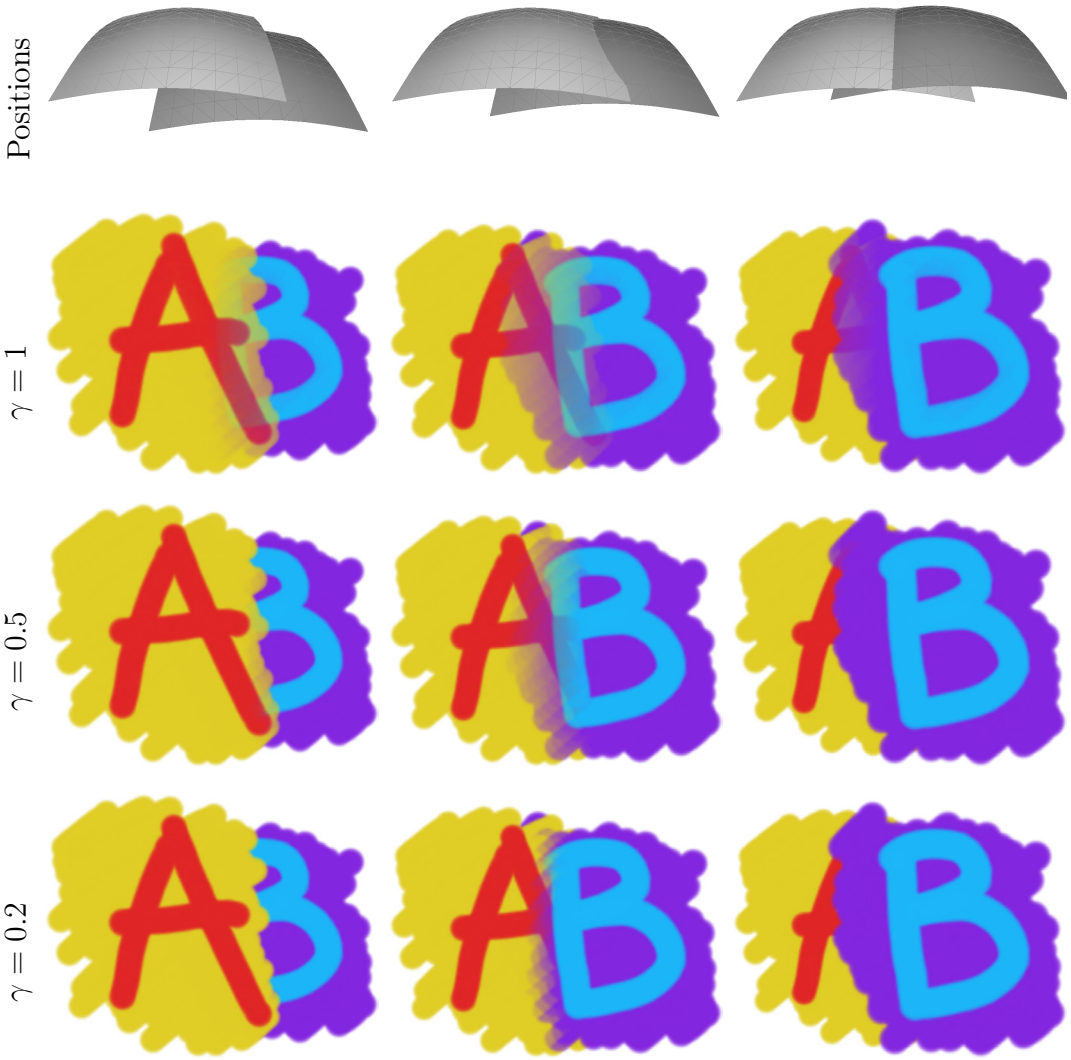


Figure 2.14: Comparison of different values for the smoothing width parameter γ. A smaller value decreases the amount of blending in the image by increasing the speed of transitions between different visibility configurations.

We have implemented to versions of mixed-order compositing: a purely CPU based implementation that does software rasterization, and a GPU implementation based on OpenGL and GLSL. Mixed-order compositing requires a persistent list of all fragments per pixel (also called "a-buffer"), which on the GPU has become practicable only in recent hardware architectures (NVIDIA GeForce 400 series and AMD Radeon 5000 series or newer). We use the *EXT_shader_image_load_store* OpenGL extension to store all fragments generated by brush splats into a buffer using a fragment shader. The order in which the splats are rasterized is irrelevant in this case. After all brush splats are rendered and stored in the a-buffer, the mixed-order composite is computed with another fragment shader and written into the framebuffer. Due to the number of splats and thus fragments in our scenes (see Table 2.1), the memory requirement of the a-buffer usually exceeds the RAM available to the GPU. We solve this problem by dividing up the desired target resolution into a number of tiles that are rendered sequentially.

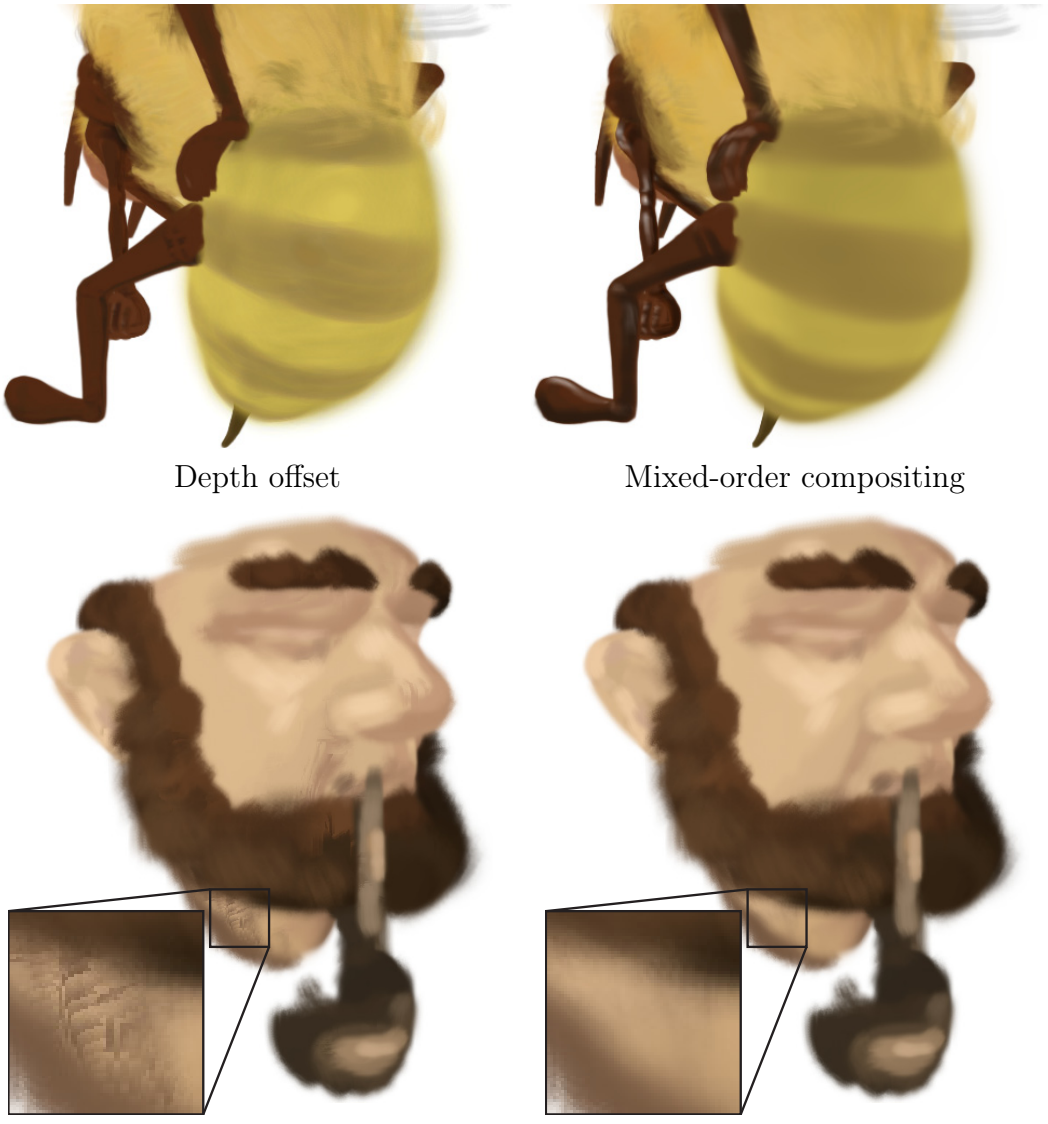
2.3.2 Results

The example 3D paintings in this sections were all created using the paint stroke embedding discussed in the following chapter, and all results shown in Chapters 3 and 4 were rendered with the techniques presented in this chapter. In Figures 2.15 and 2.16, different methods for the depth order versus paint order problem discussed in Section 2.2 are compared. Figure 2.17 shows two more 3D paintings rendered with out method.

Table 2.1 contains statistics and performance timings for the example renderings shown in this section. All performance data was measured using a single core of an Intel Core i7 2.8 GHz based computer with 12GB of RAM and a NVIDIA GeForce GTX 480 graphics card with 1.5 GB of video RAM, rendering an image of 960x720 pixels. The column "Portrait" shows the statistics of the painting and view seen in Figure 2.16, "Dog" and "Bee" in Figure 2.17, and "Captain" in the lower row of Figure 2.15.

2.3.3 Limitations, Extensions, and Future Work

The brush model and rendering methods we present form a usable basis for artists to create expressive and appealing paintings, as is demonstrated by the results in this chapter and the following chapters. However, modern 2D digital painting applications have a much richer set tools, some of which would also be of great value in the context of 3D painting. We highlight a few areas here, but since the goal of stroke based rendering is to produce images that look like



Depth offset | Mixed-order compositing

Clustering [Daniels et al. 2001] | Mixed-order compositing

Figure 2.15: A comparison between different methods to resolve the paint order on surfaces. The optimal parameters have been manually chosen for each algorithm. The left column shows artifacts on the bee's limbs and abdomen and on the captain's neck, mouth, and nose regions. These artifacts are time-varying and are especially noticeable during animations.

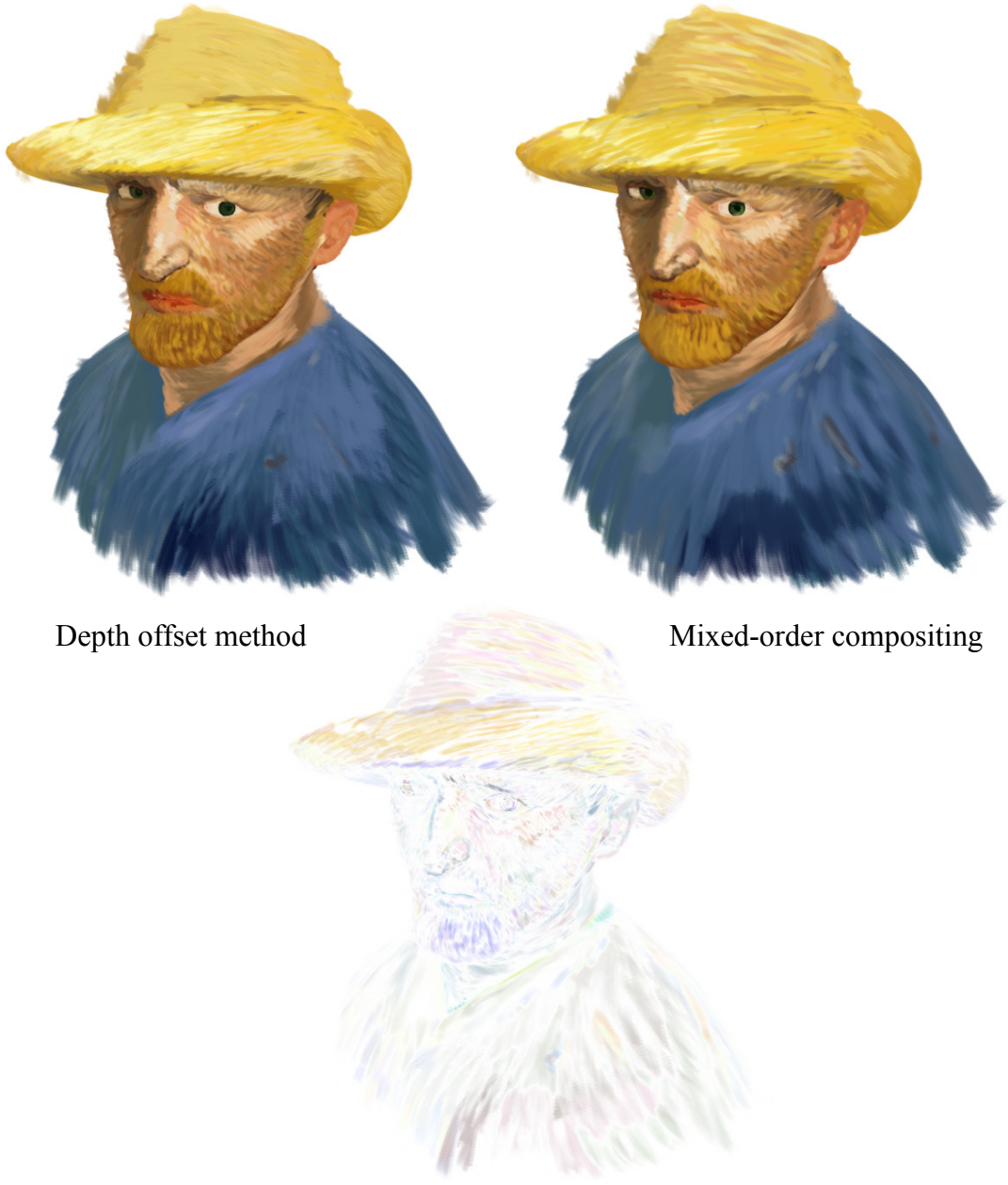


Figure 2.16: This figure shows the same painting rendered with the depth offsetting method (top left) and with mixed-order compositing (top right). The differences are most apparent on the shirt, the beard on the chin, and the right eye, where the depth offset method does an inadequate job of resolving the paint order. But, differences exist throughout the painting as the difference image at the bottom shows.

Figure 2.17: Two more examples of 3D paintings rendered with our stroke based rendering technique. In both cases, mixed-order compositing was used.

		Portrait	Dog	Captain	Bee
	Strokes	14k	29k	1.8k	20k
	Splats	246k	340k	23k	362k
	Fragments	19M	64M	31M	118M
	Max Fragments/Pixel	662	1245	643	5002
Depth offset	Total time	0.25s	0.32s	0.05s	0.30s
Mixed-order	Total time	17s	61s	26s	162s
Software	Compositing	10s	43s	18s	104s
Mixed-order	Total time	1.1s	3.9s	1.52s	9.8s
Hardware	Compositing	0.8s	3.2s	1.45s	9.4s

Table 2.1: Scene statistics and timings for our brush stroke rendering implementations for an output image of 960x720 pixels.

paintings, any tools found in these applications is of potential benefit to 3D painting.

Arguably, the most basic issue is the lack of support for true brush stroke *transparency*, as discussed in Section 2.1.3. When using mixed-order compositing, true transparency could potentially be achieved by analyzing which fragments of a pixel belong to the same stroke and computing an appropriate scale factor for the α-value of each fragment based on that. The biggest challenge in this approach is to keep the impact on computational complexity low, since mixed-order compositing already tends to dominate the rendering costs. A second option would be to use the layer blending technique that will be described in Chapter 4.4.1. But, since that technique increases the rendering complexity by a factor of the number of different transparency values present in the scene, its cost is most likely prohibitive to be used for the transparency of individual paint strokes.

Stroke pre-buffering

Finally, the transparency problem could also be solved using the stroke pre-buffering method mentioned at the end in Section 2.1.3. This method comes with its own set of outstanding issues, however. To recap, the idea behind stroke pre-buffering is to render each brush stroke into a separate buffer before compositing the resulting pixels into the final image. It has two core advantages:

- After a brush stroke has been rendered into a separate buffer, effects that should affect the stroke appearance as a whole, such as transparency or canvas texture modulation, can be applied easily.

- By compositing all the splats of a stroke with simple alpha blending first, the number of fragments that need to be processed by the more complex mixed-order compositing methods is reduced drastically, leading to better overall performance.

The biggest problems with stroke pre-buffering is the depth complexity of brush strokes: since brush strokes are curves in 3D in our application, their 2D projection may have regions where different parts of a stroke overlap. In order to resolve the visibility in paint and depth order properly, however, a composited color value needs a unique depth value that is a close approximation of all the fragments it represents. The depth complexity of the strokes therefore has to be maintained in some way.

This goal could be obtained using intelligent compositing that separates fragments of different stroke parts into layers of a deep color and depth buffer, though care would have to be taken to maintain temporal coherence when fragments change layers over time. Also, current GPUs are not particularly designed for these kinds of operations, which could negate the performance advantage gained by having less fragments in the final compositing stage.

Another option to maintain the depth complexity of brush strokes is to automatically split strokes into multiple parts that are rendered separately in such a way that no individual part has self-overlapping regions. Since the proper splits would be view-dependent, this approach also raises the issue of temporal coherence.

In general, stroke pre-buffering would be directly usable when the paint vs. depth order resolution is performed on a per-fragment basis, such as in mixed-order compositing. In the depth offsetting method, on the other hand, the rendering order of the splats of different brush strokes is interleaved, and therefore stroke pre-buffering is not trivially possible. It is questionable whether the rendering order needs to be established on a per-splat basis or if a per-stroke (or perhaps part of a stroke) rendering order would be sufficient. We have not conducted experiments in this direction.

Different compositing modes and brush models

For all of our experiments, we have exclusively used the *over*-operator in alpha compositing, which is the natural choice in many cases. It would be interesting to analyze how other commonly used compositing methods (such as darken, lighten, color dodge, color burn, etc.) could be integrated in our brush model and used in the context of stroke based rendering.

At the same time, the stamping-based brush model we build upon on is just one option among many. Its main strength lies in its versatility, but for specific

tasks a different brush model may be better suited. For example, the simulation of real-world painting techniques such as watercolor, oil, pencil, and crayon has been the target of numerous successful research efforts, some of which are discussed in Section 1.3. Such methods may be adaptable for use in 3D painting, allowing for painterly animations that mimic a specific style.

Another reason for using a different brush model can be found in rendering efficiency. While the brush stamping model is simple in concept and implementation, the number of primitives that must be rendered quickly rises into the hundreds of thousands, as becomes evident in Table 2.1. For brush strokes with little curvature and a uniform appearance along the stroke, using a ribbon-based brush model (as discussed at the beginning of Section 2.1) may lead to much better rendering performance. The two models could also co-exist, which would allow the appropriate model to be picked manually or automatically.

Level of detail and aliasing

Scaling the width of brushes as described in Section together with the 3D to 2D projection is a simple solution to the problem of making a painted object appear larger or smaller on screen, but it has several shortcomings.

Purely technical problems arise when the brush width is scaled down to sub-pixel size. In this case, a standard rasterizer misses individual splats, causing strokes to become too transparent, exhibit aliasing artifacts, and eventually vanish almost entirely. A possible remedy to this problem is to define a lower bound for the width of brushes.

A more fundamental issue, however, is level of detail. A stroke based 3D painting is best rendered roughly in the size it was painted. If it is rendered much larger, individual paint strokes become bulky and the painting may lack of detail. If it is rendered much smaller, details in the painting become too fine to represent and end up blurred or sketchy. In 2D painting, an artist would never use the same brush strokes to paint an object at different scales, but instead find the ideal level of abstraction and detail for each scale individually. Stroke based rendering therefore requires a method to handle different levels of detail smoothly in situations where an object transitions through multiple scales in size. A possible solution would be to let the user paint the object at different scales and transition between the representations, which is similar to mipmapping.

Mixed-order compositing

For large scenes, where d is much smaller than the depth range, the running time of mixed-order compositing can be improved to $O(n \log m)$ (assuming frag-

ments are given in depth order), where m is the maximum number of fragments in an interval of length d. The bottleneck is the computation of $S(z)$, which can be sped up by maintaining the fragments to be composited in stroke order in a dynamic binary tree, such as a red-black tree or a splay tree instead of our static binary tree. We did not implement this version of the algorithm because we expect that, for our scenes, the higher hidden constants of the dynamic binary tree would eclipse the potential improvement. In terms of memory, all of our steps stream over depth, so by interleaving the stages of our algorithm, memory usage can be improved to $O(m)$. Practical avenues for further optimization include compositing nearby fragments in stroke order (with bounds on the maximum incurred error) and discarding fragments obscured by other fragments closer to the viewer.

Our method assumes that fragments close together in depth are on the same surface and should thus be composited in stroke order. This assumption works well the vast majority of the time, but it may lead to unintuitive results in cases where the artist has interleaved drawing on different surfaces, or if surfaces pass through each other. Although surfaces cannot be reliably identified in general, an interesting extension of our work would be to smoothly incorporate information about distinct surfaces when it is available.

For some applications, the stroke order is irrelevant and only a temporally-coherent depth-order compositing that satisfies Properties 2–4 is needed. For such a case, we can redefine

$$S(z) = \sum_{\{i|z-d/2<z_i<z+d/2\}} (c_i, \alpha_i)$$

and leave the rest of the algorithm unchanged. Together with the box filter, the effect is that the replacement color is the average of the original fragment colors weighted by a trapezoid. This method is similar to the soft depth compositing of Bruckner and colleagues [Bruckner et al., 2010], but can be computed in $O(n)$ time (because the tree is not necessary for sums) if the fragments are given in depth order.

CHAPTER 3

Paint Stroke Embedding

An empty canvas represents the work space in which a painter realizes his or her creative vision. Working directly with brushes and paint to fill the canvas gives the artist full creative freedom of expression, evidenced by the huge variety of styles that have been explored through art's rich history. Modern digital painting software emulates the traditional painting metaphor while further empowering the user with control over layering, compositing, filtering, and other effects. As a result, digital artists have an extremely powerful, flexible, and expressive tool set for creating 2D digital paintings.

The same is not true for 3D digital painting. Most attempts to bring digital painting into the third dimension focus on texture painting or methods that project stroke centerlines onto an object's surface. The strokes must precisely conform to the object's surface, and the mathematical nature of these algorithms can betray the underlying 3D structure of the scene, leading to a "gift-wrapped" appearance. Stylistic effects that require off-surface brush strokes cannot easily be realized. Indistinct structures such as fur, hair, or smoke must be addressed using special-purpose modeling software without the direct control afforded by painting. These limitations ultimately restrict the variety of styles possible with 3D digital painting and may hinder the artist's ability to realize their creative vision.

In this chapter, we present an alternate way to define the 3D painter's workspace that targets existing limitations. We elevate the 2D painting metaphor to 3D

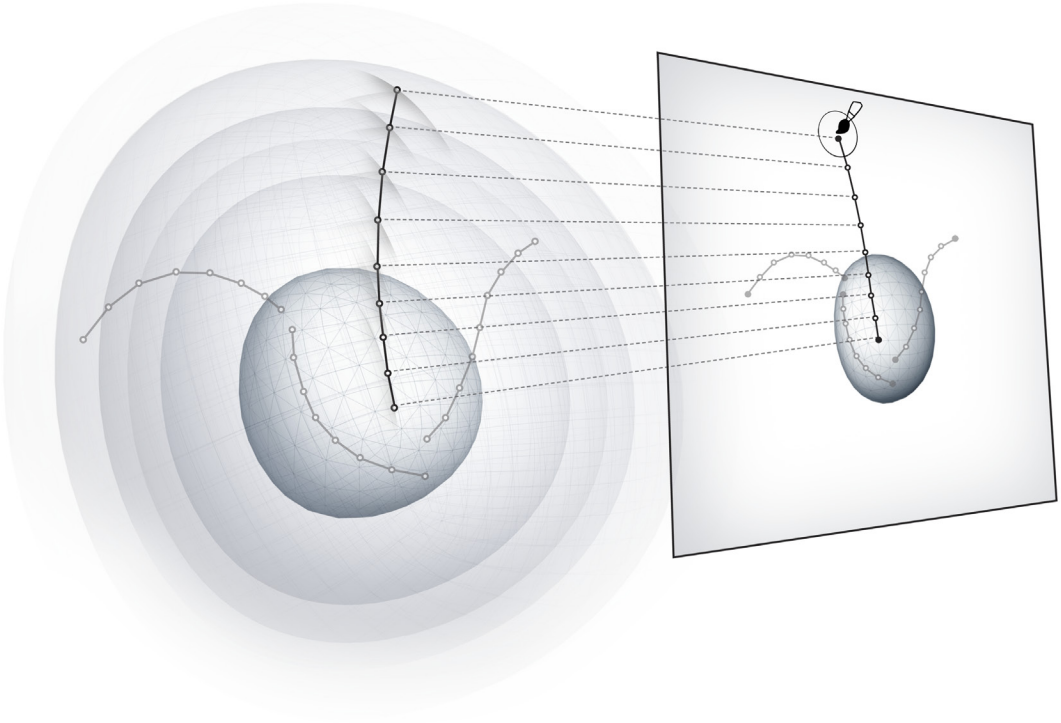


Figure 3.1: We propose embedding brush strokes painted with a two-dimensional input device in a *3D canvas* that is defined using a 3D scalar function. The scalar function is depicted by multiple shells around the object in this illustration, and it forms the basis of a customizable optimization procedure which defines the exact location along the view ray of each point on the paint stroke.

with a generalization that allows the artist to treat the full 3D space as a canvas. With this new 3D canvas, painting no longer focuses on how to paint *on* an object, but rather how to paint *in* space.

Strokes painted in the 2D viewport window must be embedded in 3D space in a way that gives creative freedom to the artist while maintaining an acceptable level of control. We address this challenge by proposing a canvas concept defined implicitly by a 3D scalar function (Figure 3.1). The artist shapes the implicit canvas by creating approximate 3D proxy geometry that defines a scalar distance field. An optimization procedure is used to embed painted strokes into 3D space by satisfying criteria defined on the scalar field and implemented as different objective terms. For example, one objective term ensures that strokes are embedded on a particular level set of the scalar field. Since any level set value can be chosen, the artist is not restricted to painting on any particular surface. Other objective criteria allow the artist to paint across level sets, allowing fur,

hair, whiskers, or other effects to be created. By formulating the optimization problem on the strokes themselves, the full scalar field need never be created and stored explicitly, leading to an efficient stroke embedding algorithm. The need for fine-scale control over the implicit canvas presents a second challenge, which we address by a unified painting/sculpting metaphor. A sculpting brush uses the same optimization procedure discussed above but creates a local change in the scalar field, resulting in outward or inward protrusions along the field's gradient. Using this sculpting tool, artists can shape the canvas before painting into it, or move strokes that have already been embedded to fine-tune the result.

3.1 Background

We analyze previous work with respect to the concept of primary and secondary space as illustrated in Figure 1.5. Existing methods such as Meier's painterly rendering system [Meier, 1996], Disney's Deep Canvas [Katanics and Lappas, 2003], the WYSIWYG NPR system [Kalnins et al., 2002], and Paint Effects of Autodesk Maya all embed strokes directly on the surface of an object or along object features such as silhouettes. Although the artist has a great deal of freedom when painting input strokes in secondary space, the strokes must conform to the surfaces of their associated primary-space objects. In this way, the functionality of these systems is intimately tied to and restricted by traditional surface representations such as polygon meshes and NURBS surfaces. The exacting nature of these surface representations may be at odds with an artist's particular style, hindering their ability to realize their artistic vision.

Other researchers have approached the problem of placing color directly in 3D with the use of specialized input devices. The CavePainting system of Keefe and colleagues [Keefe et al., 2001] uses motion capture in a virtual-reality cave to allow the artist to directly author scenes composed of ribbons, tubes, and other primitives using hand gestures. Schkolne and colleagues' Surface Drawing work [Schkolne et al., 2001] enables organic shapes to be modeled using hand gestures in a semi-immersive VR setup. Because controllability is an issue with gesture-based systems, Keefe and colleagues [Keefe et al., 2007] propose a method that uses a haptic device and 6-DOF trackers to draw lines in space in a more precise fashion. This body of work makes important advancements in human-computer interaction for ab initio design using advanced hardware devices. Our contribution is distinguished from these direct 3D painting systems in two primary ways: OverCoat extends the tablet-based 2D digital painting metaphor to 3D without the need of special hardware, thus making it more accessible and familiar to work with for artists. In addition, OverCoat's embedding procedure, brush model, and rendering algorithm merge 2D and 3D concepts to enable paintings that retain their 2D expressiveness when viewed

from any angle. Since existing direct 3D painting systems create scenes composed of 3D primitives such as ribbons, tubes, and surfaces, they cannot easily accommodate the expressive aesthetic of traditional digital painting.

Commercial modeling packages like Maya and Mudbox from Autodesk and Pixologic's ZBrush complement our work by providing tools to create 3D proxy geometry. Sketch-based modeling tools [Igarashi et al., 1999, Nealen et al., 2007], especially those that use an implicit surface representation [Karpenko et al., 2002, Schmidt et al., 2005, Bernhardt et al., 2008], are well suited since the proxy geometry need only provide a rough guide to control the implicit canvas, but is never rendered directly. Other related work [Cohen et al., 2000, Bourguignon et al., 2001, Tolba et al., 2001, Rivers et al., 2010] explores interesting concepts concerning 3D drawing, but does not address detailed 3D painting and cannot easily accommodate the expressive aesthetic we target with OverCoat.

3.2 Concept

In order to provide a controllable and effective system for 3D painting, OverCoat employs a 3D canvas based on scalar fields that are used to embed paint strokes in space. The artist sculpts proxy objects with any modeling package and imports them into the scene as triangle meshes. The objects define the overall 3D layout of the scene but need not exhibit fine geometric details since they only serve as a guide for stroke embedding and are not rendered in the final painting. Each proxy object implicitly defines a signed distance field that, conceptually, represents the object's 3D canvas.

OverCoat provides the user with a set of tools to embed paint strokes into the 3D canvas. The user first selects a proxy object and then paints into the canvas using a familiar 2D painting interface. The way in which paint strokes are embedded is determined by the semantics of the embedding tools. To obtain a maximum of flexibility and extensibility in the creation of these tools, we formulate the embedding process as a mathematical optimization problem that assigns a depth value to each point of the input paint stroke. Objective terms and constraints are used in varying combinations to obtain different embedding behaviors. These combinations are encapsulated and presented to the user as a set of different embedding tools, such as a tool to paint at a certain distance to the object, or to paint strokes that are perpendicular to the proxy surface. The embedding tools can use the scalar field magnitude, sign, and gradient in their objective terms to establish criteria that relate any position in space to the proxy object.

Another advantage of embedding paint strokes using an optimization instead of specialized heuristics (such as direct projection) is that it allows our system to

gracefully handle cases where a tool's primary goal cannot be met. For example, by incorporating a regularizing smoothness term, our level set painting tool can easily handle the case where painted strokes extend beyond the level set's silhouette. A method based on direct projection would require special heuristics since, in this case, there is no surface on which to project.

An additional sculpting tool allows the user to make localized modifications to the scalar field using the same embedding procedure. In this way, the painting interface can be used both for coloring the canvas and for manipulating the shapes.

3.2.1 Canvas Representation

OverCoat represents a 3D canvas as a scalar field $f : \mathbb{R}^3 \to \mathbb{R}$. A point $\mathbf{x}$ with $f(\mathbf{x}) = l$ is said to lie on level l. The corresponding implicit surface at level l, also called an isosurface or level set, is the set of all points $\mathbf{x} \in \mathbb{R}^3$ such that $f(\mathbf{x}) = l$. The scalar field relates the points in space to the surface of the corresponding proxy object.

In OverCoat, a proxy object is defined by a triangle mesh M that forms a closed manifold solid. The scalar field is initially defined by the signed Euclidean distance to M. Let $d(\mathbf{x}, M)$ be the shortest Euclidean distance from a point $\mathbf{x} \in \mathbf{R}^3$ to the mesh M, then

$$|f_{proxy}(\mathbf{x})| = d(\mathbf{x}, M).$$

Since M is a triangle mesh, $d(\mathbf{x}, M)$ is the minimal shortest Euclidean distance between $\mathbf{x}$ and any of the triangles T of the mesh:

$$d(\mathbf{x}, M) = \inf_{\mathbf{c} \in M} \|\mathbf{c} - \mathbf{x}\| = \inf_{T \in M} \left(\inf_{\mathbf{c} \in T} \|\mathbf{c} - \mathbf{x}\| \right).$$

To determine whether a point is inside or outside a proxy object, we require $f_{proxy}(\mathbf{x})$ to be the *signed* distance to M, such that

$$f_{proxy}(\mathbf{x}) \begin{cases} > 0 & \text{if } \mathbf{x} \text{ is } \textit{outside} \text{ the object} \\ = 0 & \text{if } \mathbf{x} \text{ is on the object } \textit{surface} \\ < 0 & \text{if } \mathbf{x} \text{ is } \textit{inside} \text{ the object.} \end{cases} \tag{3.1}$$

The scalar field $f_{proxy}(\mathbf{x})$ described so far is simply the signed distance field to M. However, the sculpting system described in Section 3.4 can inflict direct, localized changes to the field values of $f(\mathbf{x})$ so that they no longer represent distances. In either case, the scalar field is only C^0 continuous. This property

has no negative influence on the stroke embedding with the objective terms and tools presented in this paper. Problems could arise if objective terms were introduced which are more sensitive to the scalar field smoothness. In this case, a scalar field formulation with higher order continuity, such as the one described by Peng and colleagues [Peng et al., 2004], might be preferable.

We define all operations on the scalar field in a form that does not require explicit storage of the field values. Thus, we avoid the memory and computational costs of a voxel decomposition or the algorithmic complexity of more sophisticated distance-field representation methods [Frisken et al., 2000].

3.3 Stroke Embedding

The artist paints in a particular 2D view of the 3D canvas, generating an ordered sequence of n stroke points $\hat{\mathbf{p}}_i \in \mathbb{R}^2$. The goal of stroke embedding is to find 3D positions $\mathbf{p}_i \in \mathbb{R}^3$ for these points in a way that is meaningful and useful to the artist. To target an embedding algorithm that meets these workflow considerations in a flexible and extendable way, we cast the embedding of the stroke points as an optimization problem. This framework allows us to implement objective function terms that accomplish different embedding behaviors, such as painting on a level set of the 3D canvas's scalar field, or across the scalar field between two chosen level sets. Combinations of these terms are exposed to the user as different embedding tools.

To ensure that the embedded strokes match the artist's intent, it is crucial that the stroke points $\mathbf{p}_i$ project back to their original screen space locations $\hat{\mathbf{p}}_i$ in the view from which they were painted. We enforce this property strictly by parameterizing the stroke points by their view ray: $\mathbf{p}_i = \mathbf{o} + t_i\mathbf{d}_i$, where $\mathbf{o}$ is the camera position, $\mathbf{d}_i$ the view vector that passes through $\hat{\mathbf{p}}_i$ on the screen plane, and t_i the ray parameter (see Figure 3.2 for an illustration). The t_i are thus the unknown variables of the optimization.

3.3.1 Objective Terms

We propose three objective terms that provide the ingredients necessary to build OverCoat's embedding tools. The *level distance* term is minimized when all stroke points are at a particular distance from the proxy geometry. The *angle* term minimizes the curvature of the stroke and thus smoothes its embedding. The *arc length* term favors straight embeddings by minimizing the total length of a stroke.

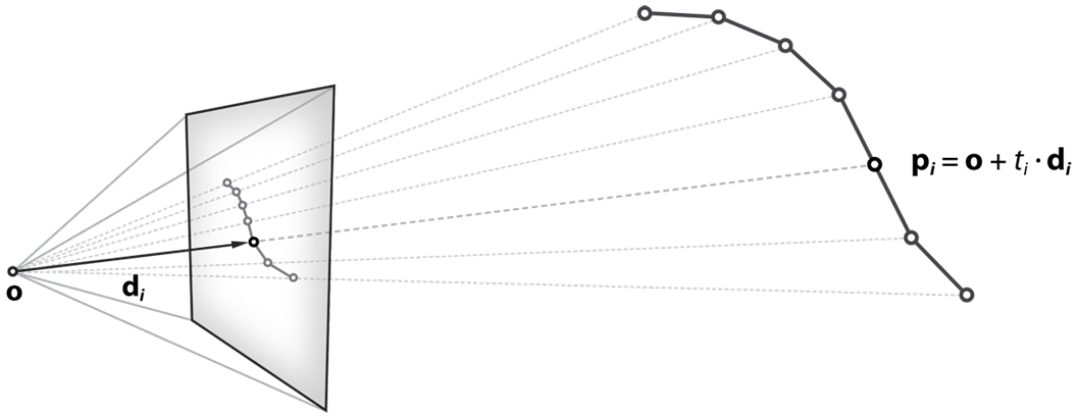


Figure 3.2: To ensure that embedded points project back to their original positions on screen, stroke points are parameterized along the view ray that passes through the input points on screen for embedding.

Level distance OverCoat allows the user to select a specific level l, and hence a specific isosurface $f(\mathbf{x}) = l$, on which to apply strokes. The corresponding objective term should ensure that all stroke points are embedded as closely as possible to the selected isosurface. The *level distance* objective term sums the difference between the actual field value $f(\mathbf{x})$ evaluated at all point locations $\mathbf{p}_i$ and the desired level l:

$$E_{level} = \sum_{i=1}^{N} \left(f(\mathbf{p}_i) - l \right)^2 . \tag{3.2}$$

Angle The *angle* objective term aims to minimize the directional deviation of consecutive line segments along a stroke. This deviation is measured by the dot product between the normalized line segments, which equals 1 when the segments are co-linear:

$$E_{angle} = \sum_{i=1}^{N-2} \left(1 - \frac{\mathbf{p}_{i+2} - \mathbf{p}_{i+1}}{\|\mathbf{p}_{i+2} - \mathbf{p}_{i+1}\|} \cdot \frac{\mathbf{p}_{i+1} - \mathbf{p}_i}{\|\mathbf{p}_{i+1} - \mathbf{p}_i\|} \right)^2 . \tag{3.3}$$

Arc length The *arc length* objective term penalizes the collective length of all segments:

$$E_{length} = \sum_{i=1}^{N-1} \|\mathbf{p}_{i+1} - \mathbf{p}_i\|^2 . \tag{3.4}$$

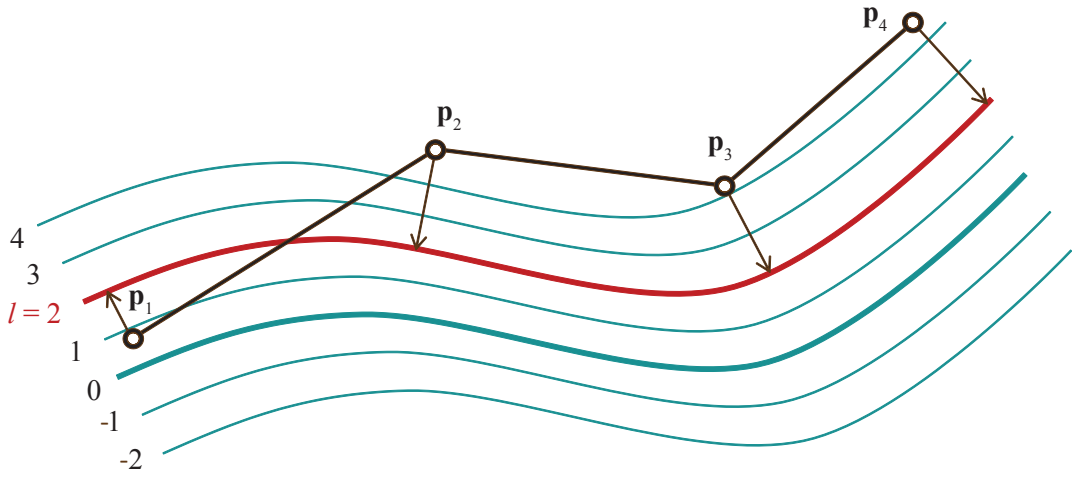


Figure 3.3: The level distance objective term is the sum of the differences between a chosen value l and the scalar field value at each stroke point. In this 2D example, the scalar field is simply the signed distance field to the thick blue line. Under this condition, the level distance term becomes the sum of distances between the stroke points and the chosen l-isoline marked in red. The thin blue lines are other isolines of the distance field.

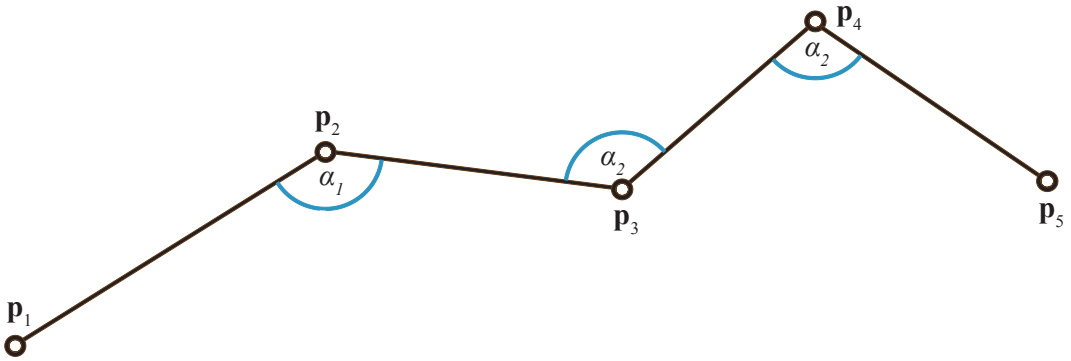


Figure 3.4: Conceptually, the angle objective term is the sum of $180° - \alpha_i$ for the α_i shown in this figure. In our actual implementation, we don't compute the angles explicitly, but use the dot product between neighboring segments instead (Equation 3.3).

3.3.2 Optimization

The goal for an embedding tool is to find ray parameter values t_i and thus 3D locations for all stroke points that minimize the weighted sum of all objective terms:

$$E = w_{level}E_{level} + w_{angle}E_{angle} + w_{length}E_{length}.$$

Individual embedding tools, described in the next section, achieve different behaviors by setting different values for the weights w_{level}, w_{angle}, and w_{length}. Since the only unknowns to the optimization are the depth values t_i, the optimization does not change the shape of a stroke in the view in which the stroke was painted.

In our implementation, we use the quasi-Newton L-BFGS method to solve this non-linear optimization problem. Like steepest descent methods, Quasi-Newton methods require only the value and the gradient of the objective function to be provided at each iteration. Successive gradient vectors are analyzed to approximate second derivatives that are used to compute a descent direction, leading to an improved convergence performance over steepest descent methods [Nocedal and Wright, 2006].

The gradient of the objective function is a vector of the function's partial derivatives with respect to the unknown variables t_i:

$$\nabla E = \left[\frac{\partial E}{\partial t_1}, \ \ldots \ \frac{\partial E}{\partial t_i}, \ \ldots \ \frac{\partial E}{\partial t_n}\right]^T.$$

Due to the linearity of differentiation, each individual partial derivative of E can simply be expressed as the weighted sum of the corresponding partial derivatives of the objective terms:

$$\frac{\partial E}{\partial t_i} = w_{level}\frac{\partial E_{level}}{\partial t_i} + w_{angle}\frac{\partial E_{angle}}{\partial t_i} + w_{span}\frac{\partial E_{span}}{\partial t_i} + w_{length}\frac{\partial E_{length}}{\partial t_i}.$$

The partial derivatives of the objective terms can be derived from the equations given in Section 3.3.1.

Level distance term derivative Since the positions of stroke points are parameterized by $\mathbf{p}_i = \mathbf{o} + t_i\mathbf{d}_i$, their derivatives with respect to the ray parameter t_i is equal to the viewing vector $\mathbf{d}_i$. According to the chain rule, the derivative of the canvas scalar field $f(\mathbf{p}_i)$ with respect to t_i is therefore

$$\frac{\partial f(\mathbf{p}_i)}{\partial t_i} = \nabla f(\mathbf{p}_i) \cdot \mathbf{d}_i, \tag{3.5}$$

which, as expected, is equivalent to the directional derivative of $f(\mathbf{p}_i)$ in the viewing direction $\mathbf{d}_i$.

When Equation 3.2 is differentiated with respect to t_i, the only non-zero term in the sum on the right-hand side is the one involving $\mathbf{p}_i$, and thus another application of the chain rule leads to

$$\frac{\partial E_{level}}{\partial t_i} = 2\left(f(\mathbf{p}_i) - l_i\right)\left(\nabla f(\mathbf{p}_i) \cdot \mathbf{d}_i\right).$$

Angle term derivative To simplify the differentiation of the angle term presented in Equation 3.3, we define

$$\begin{aligned}
\mathbf{a}_k &= \mathbf{p}_{k+1} - \mathbf{p}_k, \\
\mathbf{b}_k &= \mathbf{p}_{k+2} - \mathbf{p}_{k+1}, \\
D_k &= \frac{\mathbf{a}_k}{\|\mathbf{a}_k\|} \cdot \frac{\mathbf{b}_k}{\|\mathbf{b}_k\|} = (\mathbf{a}_k \cdot \mathbf{b}_k)\|\mathbf{a}_k\|^{-1}\|\mathbf{b}_k\|^{-1},
\end{aligned}$$

so that the angle objective term becomes

$$E_{angle} = \sum_{i=1}^{n-2} \left(1 - D_i\right)^2. \tag{3.6}$$

For any $i \in [2, N-2]$, there are exactly three D's that depend on t_i: D_i, D_{i-1}, and D_{i-2}. The partial derivative of Equation 3.6 with respect to t_i can therefore be written as

$$\frac{\partial E_{angle}}{\partial t_i} = -2\sum_{j=0}^{2}(1 - D_{i-j})\frac{\partial D_{i-j}}{\partial t_i}. \tag{3.7}$$

For the differentiation of D_{i-j}, the application of the product rule yields

$$\begin{aligned}
\frac{\partial D_{i-j}}{\partial t_i} = &\frac{\partial(\mathbf{a}_{i-j} \cdot \mathbf{b}_{i-j})}{\partial t_i}\|\mathbf{a}_{i-j}\|^{-1}\|\mathbf{b}_{i-j}\|^{-1} + \\
&+ (\mathbf{a}_{i-j} \cdot \mathbf{b}_{i-j})\frac{\partial\|\mathbf{a}_{i-j}\|^{-1}}{\partial t_i}\|\mathbf{b}_{i-j}\|^{-1} + \\
&+ (\mathbf{a}_{i-j} \cdot \mathbf{b}_{i-j})\|\mathbf{a}_{i-j}\|^{-1}\frac{\partial\|\mathbf{b}_{i-j}\|^{-1}}{\partial t_i}.
\end{aligned}$$

The partial derivatives in this equation depends on the value of j. We show the differentiation with $j = -1$; for the other j, the differentiation is analogous, but

some terms vanish because they do not depend on t_i:

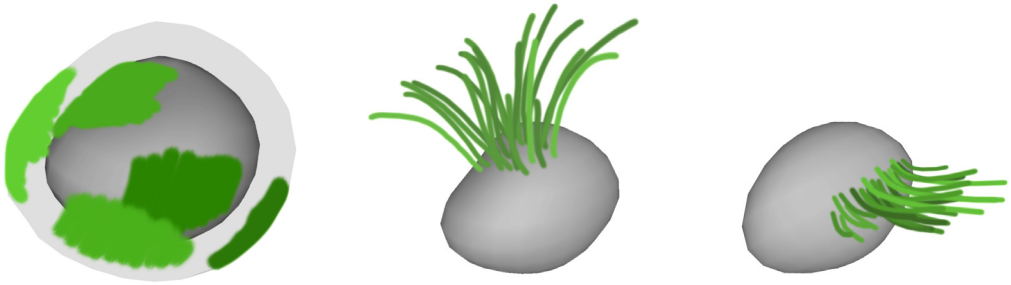
$$\begin{aligned}
\frac{\partial(\mathbf{a}_{i-1} \cdot \mathbf{b}_{i-1})}{\partial_i} &= \frac{\partial \mathbf{a}_{i-1}}{\partial_i} \cdot \mathbf{b}_{i-1} + \mathbf{a}_{i-1} \cdot \frac{\partial \mathbf{b}_{i-1}}{\partial_i} \\
&= \frac{\partial(\mathbf{p}_i - \mathbf{p}_{i-1})}{\partial_i} \cdot (\mathbf{p}_{i+1} - \mathbf{p}_i) + (\mathbf{p}_i - \mathbf{p}_{i-1}) \cdot \frac{\partial(\mathbf{p}_{i+1} - \mathbf{p}_i)}{\partial_i} \\
&= \mathbf{d}_i \cdot (\mathbf{p}_{i+1} - \mathbf{p}_i) - (\mathbf{p}_i - \mathbf{p}_{i-1}) \cdot \mathbf{d}_i \\
&= \mathbf{d}_i \cdot (\mathbf{p}_{i+1} - 2\mathbf{p}_i + \mathbf{p}_{i-1}), \\
\frac{\partial \|\mathbf{a}_{i-1}\|^{-1}}{\partial t_i} &= \frac{\partial \|\mathbf{a}_{i-1}\|^{-1}}{\partial \|\mathbf{a}_{i-1}\|} \cdot \frac{\partial \|\mathbf{a}_{i-1}\|}{\partial \mathbf{a}_{i-1}} \cdot \frac{\partial \mathbf{a}_{i-1}}{\partial t_i} \\
&= -\|\mathbf{p}_i - \mathbf{p}_{i-1}\|^{-2} \cdot \|\mathbf{p}_i - \mathbf{p}_{i-1}\|^{-1} (\mathbf{p}_i - \mathbf{p}_{i-1}) \cdot \frac{\partial(\mathbf{p}_i - \mathbf{p}_{i-1})}{\partial t_i} \\
&= -\|\mathbf{p}_i - \mathbf{p}_{i-1}\|^{-3} (\mathbf{p}_i - \mathbf{p}_{i-1}) \cdot \mathbf{d}_i, \\
\frac{\partial \|\mathbf{b}_{i-1}\|^{-1}}{\partial t_i} &= \frac{\partial \|\mathbf{b}_{i-1}\|^{-1}}{\partial \|\mathbf{b}_{i-1}\|} \cdot \frac{\partial \|\mathbf{b}_{i-1}\|}{\partial \mathbf{b}_{i-1}} \cdot \frac{\partial \mathbf{b}_{i-1}}{\partial t_i} \\
&= -\|\mathbf{p}_{i+1} - \mathbf{p}_i\|^{-2} \cdot \|\mathbf{p}_{i+1} - \mathbf{p}_i\|^{-1} (\mathbf{p}_{i+1} - \mathbf{p}_i) \cdot \frac{\partial(\mathbf{p}_{i+1} - \mathbf{p}_i)}{\partial t_i} \\
&= -\|\mathbf{p}_{i+1} - \mathbf{p}_i\|^{-3} (\mathbf{p}_{i+1} - \mathbf{p}_i) \cdot \mathbf{d}_i.
\end{aligned}$$

Arc length term derivative When differentiating Equation 3.4 with respect to t_i, exactly two elements of the sum are non-zero:

$$\begin{aligned}
\frac{\partial E_{length}}{\partial t_i} &= \frac{\partial \|\mathbf{p}_i - \mathbf{p}_{i-1}\|^2}{\partial t_i} + \frac{\partial \|\mathbf{p}_{i+1} - \mathbf{p}_i\|^2}{\partial t_i} \\
&= 2\mathbf{d}_i \cdot (\mathbf{p}_i - \mathbf{p}_{i-1}) - 2\mathbf{d}_i \cdot (\mathbf{p}_{i+1} - \mathbf{p}_i) \\
&= 2\mathbf{d}_i \cdot (2\mathbf{p}_i - \mathbf{p}_{i-1} - \mathbf{p}_{i+1}).
\end{aligned}$$

3.3.3 Embedding Tools

The objective terms presented in the previous sections provide the ingredients necessary for the embedding mechanisms shown in this thesis. To hide the complicated and technical optimization details, we encapsulate the choice of objective term weights and expose them to the user in the form of three powerful embedding tools (Figure 3.5). All examples in the thesis were painted using these three tools.



(a) Level set tool (b) Hair tool (c) Feather tool

Figure 3.5: The three embedding tools implemented by OverCoat.

Level set tool

The level set tool embeds all stroke points as closely as possible onto a selected level set surface. This goal is achieved by giving a dominant weight to the level distance term E_{level}. By itself, this term has the same effect as direct projection for paint strokes within the silhouette boundaries of the level set. When a stroke extends outside the silhouette, the closest distance solution will be roughly perpendicular to the surface in the region where the silhouette is crossed, thus creating a sharp corner along the embedded stroke. We incorporate the angle term E_{angle} to achieve a smoother transition in this case. The weights $w_{level} = 1$, $w_{angle} = 0.1$, and $w_{length} = 0$ were used for level set tool in all of our examples. If a fuzzy embedding is desired, the target level can be displaced by a random amount for each stroke, or even for each individual stroke point.

Hair and feather tool

Another set of tools allows the user to paint across level sets. The user selects a target level for both the start and the end of the stroke. The first and last

points of a painted stroke are constrained to lie on these prescribed levels using $w_{level} = 1$. The remaining stroke points, however, are optimized with $w_{level} = 0$. In the absence of a target surface, the angle objective term ensures a smooth transition between the two ends with $w_{angle} = 1$. With this term alone, the resulting embedding will be smooth, but may be extended undesirably in order to meet the angle criteria optimally, resulting in strokes that overshoot the prescribed target level set. We used the arc length term with $w_{length} = 0.05$ to regularize this behavior and cause a straighter embedding in space.

The resulting cross-level embedding can be controlled more explicitly with additional constraints. For example, the initial direction of the stroke can be prescribed by temporarily pre-pending an artificial stroke point. This point stays fixed during the optimization of the stroke, but affects the embedding solution through the angle objective term. Depending on its relative position to the first actual stroke point $\mathbf{p}_1$, it will cause the embedded stroke to leave the surface in a particular direction. For the "hair" tool, the temporary point is placed along the negative gradient direction at $\mathbf{p}_1$, causing the initial direction of the stroke to be perpendicular to the level set. The "feather" tool was realized by placing the temporary point in the direction that is tangential to the scalar field at $\mathbf{p}_1$ and has the largest angle to the straight line connecting $\mathbf{p}_1$ and $\mathbf{p}_N$. Figure 3.6 illustrates the embedding process of hair and feather strokes.

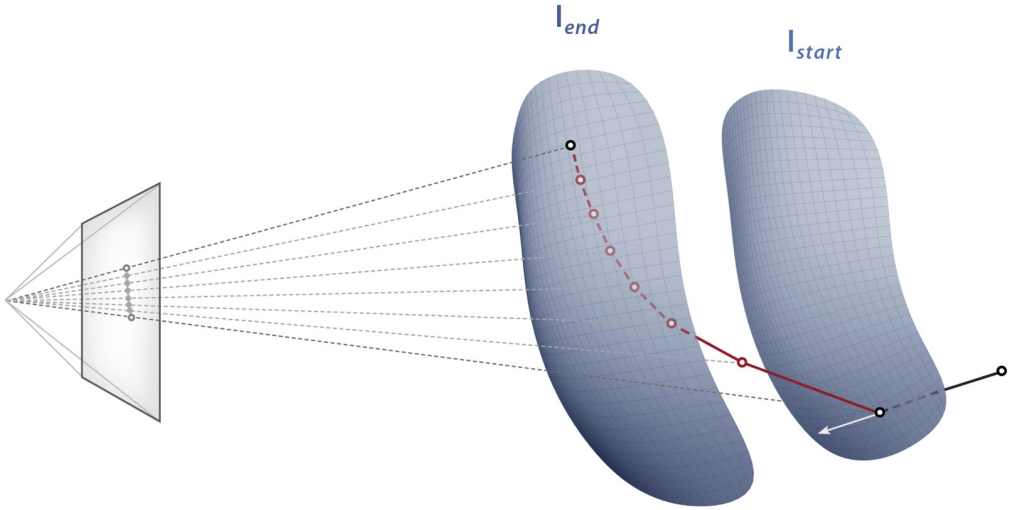
3.3.4 Distance, Derivative, and Gradient Computations

In the process of embedding strokes in space as described above, the optimization procedure must repeatedly evaluate a canvas's scalar field to calculate the field's magnitude $f(\mathbf{x})$, the gradient ∇f, and the derivative of the scalar field with respect to the ray parameter $\partial f(\mathbf{p}_i)/\partial t_i = \nabla f \cdot \mathbf{d}_i$. In the absence of sculpting operations, $f(\mathbf{x})$ is defined to be the signed smallest distance to the proxy geometry. The absolute distance value is computed by finding the closest point $\mathbf{c}$ within any primitive of the proxy geometry mesh as described in Section 3.2.1. The sign of $f_{proxy}(\mathbf{x})$ can be found by comparing the vector connecting $\mathbf{c}$ and $\mathbf{x}$ with the outward oriented surface normal the normal at $\mathbf{c}$ [Bærentzen and Aanæs, 2005]:

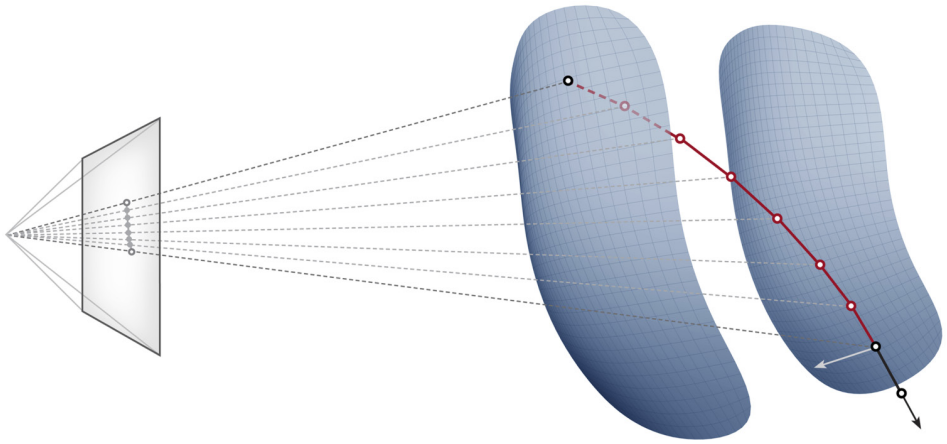
$$f_{proxy}(\mathbf{x}) = \text{sign}((\mathbf{c} - \mathbf{x}) \cdot \mathbf{n})\|\mathbf{x} - \mathbf{c}\|$$

The normal $\mathbf{n}$ is either the normal of the triangle itself if $\mathbf{c}$ lies within the triangle, or the angle-weighted normal at the vertex or edge on which $\mathbf{c}$ is located otherwise. This computation assumes that M is an orientable manifold and that all normals point outward.

The gradient is generally defined by the normalized vector between the query point $\mathbf{x}$ and its closest point on the surface. This definition can become numer-



(a) Hair tool embedding process



(b) Feather tool embedding process

Figure 3.6: The hair (a) and feather (b) tools embed the first and the last points of a paint stroke on chosen level set values l_{start} and l_{end}. Between the end points, the optimization finds a smooth embedding using the angle objective term. The angle with which the paint stroke leaves the start isosurface is controlled by temporarily pre-prending a static stroke point.

ically unstable or even invalid if $\mathbf{x}$ lies very close to or exactly on the surface. In that situation, the angle-weighted normal of the closest primitive should be used instead:

$$\nabla f(\mathbf{x}) = \begin{cases} \mathbf{n} & \text{if } \|\mathbf{x}-\mathbf{c}\| < \delta \\ \dfrac{(\mathbf{x}-\mathbf{c})}{\|\mathbf{x}-\mathbf{c}\|} & \text{otherwise.} \end{cases}$$

All scalar field evaluations are computed on the fly, so that OverCoat never needs to store the field in a discretized form. An oriented bounding box tree [Gottschalk et al., 1996] is used to accelerate the closest primitive look-ups, which allows the embedding to be performed interactively.

3.3.5 Initialization

There are several reasons that speak in the favor of providing the optimization procedure with a good initial solution. First, it avoids issues where certain objective terms may be badly behaved at artificial initial solutions such as $t_i = 1$. In our angle term, for example, the derivatives become very large if the distances between the stroke points $\mathbf{p}_i$ are small. Second, our problem formulation in general does not expose a clear global minimum, and is therefore prone to converge to undesired local minima. Finally, a good initial guess drastically reduces the number of iterations required for the optimization to converge. We therefore propose using a simple heuristic for each tool to initialize the unknown values to a configuration that is likely to be close to the optimal solution.

For the level set tool, our system uses Sphere Tracing, a ray marching technique described by Hart [Hart, 1994], to move the initial positions of the stroke points to the first intersection with the target level set. Stroke points that do not fall within the silhouette of the target level set are marked and handled separately: they are joined up into connected sequences of off-surface points. If such a sequence is adjacent to a successfully initialized stroke point on one end, the t-value of that stroke point is copied to the entire sequence. If both ends are adjacent to initialized stroke points, the bordering t-values are linearly interpolated over the sequence. Finally, if the sequence spans the entire stroke, meaning that the stroke was painted completely outside of the silhouette of the target level set, the smallest t-value encountered during ray marching on any stroke point is copied to the entire sequence/stroke.

For the hair and feather tools, only the first and the last stroke points are initialized to their respective target levels (or the closest approximation thereof), while the remaining unknowns are initialized with a linear interpolation between the two end points.

3.3.6 Stroke Refinement

If a target surface or parts of it are at a considerable angle to the screen plane, the sampling of points along the stroke from the input device may not be sufficient for the stroke to be embedded nicely in the scalar field. For example, the level objective term can only be faithful to the chosen isosurface if the stroke sampling is fine enough for the level of detail of the surface. Likewise, the angle term can only provide an effective smoothing if the sampling is appropriate. Our system therefore refines input strokes painted with the level set tool during their initialization. If the Euclidean distance between two consecutive stroke points after initialization is larger than a given threshold, a new stroke point is inserted between the existing stroke points in 2D and immediately projected according to the initialization method described above. This step is repeated until all stroke segments are at most twice as long as the shortest segment in the stroke, thus guaranteeing a roughly uniform sampling along the stroke (Figure 3.7). The position of new stroke points in 2D can be chosen according to an arbitrary subdivision scheme. We have obtained good results with the 4-point interpolatory curve subdivision method [Dubuc, 1986, Dyn et al., 1987].

Inner silhouettes The refinement process has the added benefit of detecting strokes that cross occluding contours. Painting across occluding contours can result in stroke segments bridging two parts of an isosurface that are potentially far apart in 3D. Attempting to refine such a segment causes an infinite recursion within a segment that cannot become any shorter. The end points of this segment eventually converge to two distinct 3D points that both project to the same point in 2D (Figure 3.8). This 2D point is the location where the paint stroke crosses the inner silhouette on screen. OverCoat detects this case by comparing the original segment length with the lengths of the two new segments. If the ratio is below a given threshold, the paint stroke is split into two at the location of convergence. We found that a value of 0.1 works well for the threshold ratio.

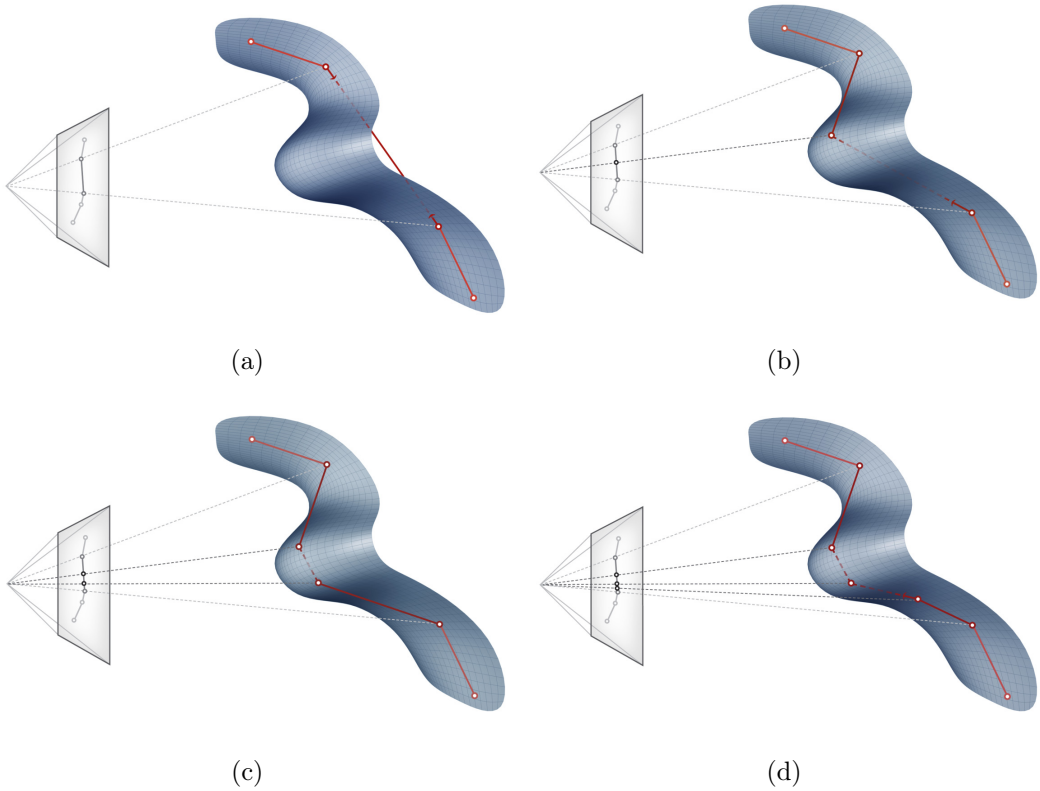


Figure 3.7: The middle segment of the original stroke in Figure (a) is subdivided by inserting a new stroke point in 2D between the segment's end points and initializing its 3D position using the initialization heuristic of the current tool (b). The right sub-segment is further subdivided in (c) and (d).

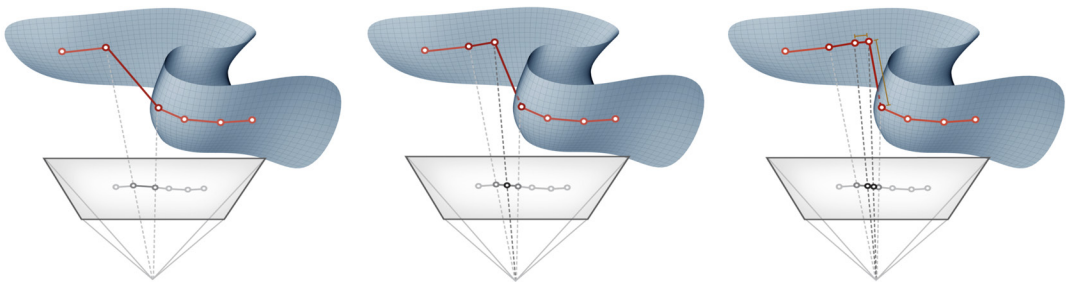


Figure 3.8: This illustration shows three consecutive refinement steps of a paint stroke crossing an occluding contour of a target isosurface. In each step, the ratio between the original segment and the smaller subsegment increases, which provides an indication for a contour crossing.

3.4 Canvas Sculpting

In the same way that paint strokes are embedded to add color to the canvas, our sculpting tool embeds sculpting strokes that alter the shape of the canvas itself. Sculpting strokes act as direct modifiers to the canvas's scalar field and thus have an influence on the embedding of subsequent strokes. A sculpting stroke defines a contribution function $C(r, R)$, where r is the smallest distance from $\mathbf{x}$ to any of the line segments of the sculpting stroke, and R is a user-defined radius of influence. OverCoat uses a cubic polynomial with local support [Wyvill et al., 1986] for $C(r, R)$:

$$C(r,R) = \begin{cases} 2\frac{r^3}{R^3} - 3\frac{r^2}{R^2} + 1, & \text{if } r < R \\ 0, & \text{otherwise.} \end{cases}$$

The scalar field is modified by adding the contributions of all sculpting strokes to the field function:

$$f(\mathbf{x}) = f_{proxy}(\mathbf{x}) - \sum_j K_j C(r_j, R_j), \tag{3.8}$$

where j enumerates all sculpting strokes with a non-zero contribution at $\mathbf{x}$, and K_j determines the magnitude of each sculpting operation. A positive value of K causes an outward deformation of the surface, while a negative value causes an inward deformation.

Adding such contribution functions locally compresses and expands the scalar field as illustrated in Figure 3.9. The amount by which a surface is shifted by a sculpting operation therefore depends both on K and previous sculpting operations in the region. OverCoat keeps the magnitude of the surface deformation approximately constant by computing K based on the scalar field value at a user-chosen distance in the gradient direction from the stroke centerline. For each point $\mathbf{p}_i$ on the sculpting stroke j with a normalized gradient vector $\mathbf{n}_i$, a local magnitude value $K_{j,i}$ is computed:

$$K_{j,i} = s_j \frac{|f(\mathbf{p}_i + s_j d_j R_j \mathbf{n}_i)|}{C(d_j R_j, R_j)}, \quad s_j = \begin{cases} 1 & \text{for outward deformation} \\ -1 & \text{for inward deformation.} \end{cases}$$

where $d_j \in]0,1[$ is a user-specified parameter that defines the magnitude of deformation for each sculpting stroke. The K_j used in Equation 3.8 is found by linearly interpolating the $K_{j,i}$ to the closest point to $\mathbf{x}$ on the sculpting stroke.

3.4.1 Impact on Stroke Embedding

Once a sculpting stroke has been embedded, it is incorporated into the evaluation of the canvas scalar field. As a consequence, the scalar field values no

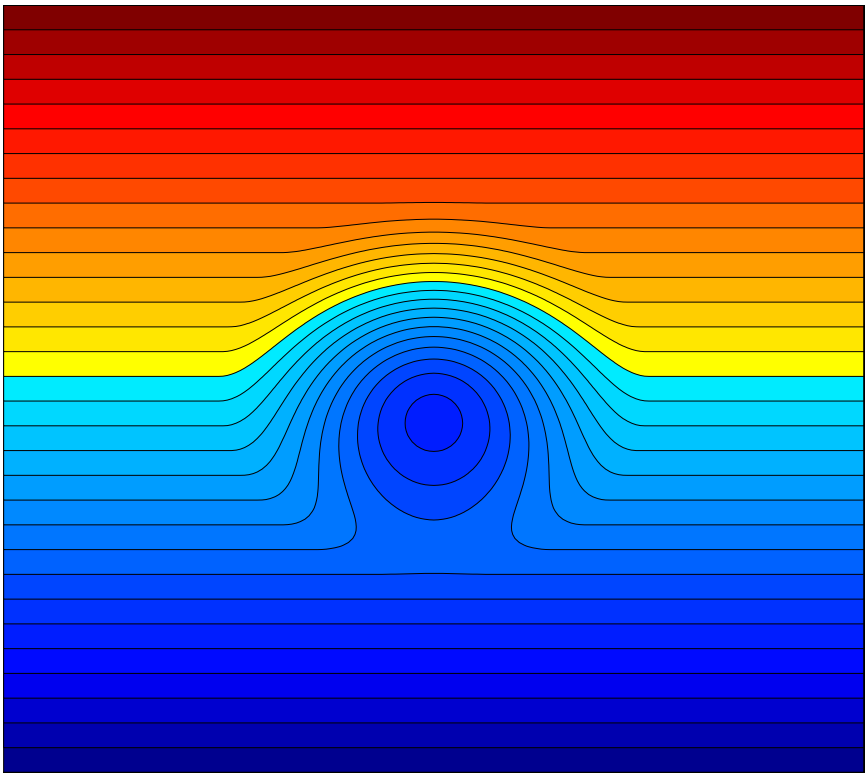


Figure 3.9: This figure shows the scalar field resulting from a flat surface with one sculpting contribution. The surface (zero level set) is between the yellow and the cyan band. The sculpting contribution locally changes the magnitude of the gradient, which is visible in the compression and expansion of the spaces between the isolines.

longer represent the distance to the zero level set. The gradient of the scalar field is augmented by the sculpting contribution functions:

$$\nabla f(\mathbf{x}) = \nabla f_{proxy}(\mathbf{x}) - \sum_j K_j \nabla C(r_j, R_j),$$

$$\nabla C(r, R) = \begin{cases} 6\frac{r^2}{R^3}\frac{\partial r}{\partial \mathbf{x}} - 6\frac{r}{R^2}\frac{\partial r}{\partial \mathbf{x}}, & \text{if } r < R \\ 0, & \text{otherwise.} \end{cases}$$

The ray marching procedure used to find an initial embedding solution (Section 3.3.5) requires a lower bound of the Euclidean distance to the target level set l to determine its step size. If the query point $\mathbf{x}$ is not within the influence region of any sculpting strokes, a practical lower distance bound is the minimum between the distance to the closest sculpting stroke and $f_{proxy}(\mathbf{x}) - l$. Otherwise,

OverCoat uses a distance bound derived according to Hart [Hart, 1994]:

$$d(\mathbf{x}, S) >= \frac{f(\mathbf{x}) - l}{1 + \sum_j |K_j| \frac{3}{2R_j}},$$

where the lower part of the fraction is the Lipschitz constant of the signed distance field of the proxy geometry (which is equal to 1) plus the Lipschitz constant of the sum of all contributions of the sculpting strokes.

To provide immediate feedback of the sculpting operations to the user, OverCoat deforms a copy of the proxy geometry by moving affected vertices along their normal to the new zero level set. If necessary, the mesh is refined to account for geometric complexity added by the sculpting tool. This copy is used for display only, while future scalar field computations use the original proxy geometry together with the sculpting stroke influence functions directly (Equation 3.8).

3.5 Results

In this section we show five complete 3D paintings created by three different artists using our prototype OverCoat software. For these paintings, the artists modeled approximate proxy geometry in Autodesk Maya, Pixologic ZBrush, or Maxon Cinema4D and imported it into OverCoat for painting. The proxy geometry does not include fine details. Instead, the artists achieved detailed results by painting strokes with the level set, hair, and feather tools, or sculpting additional details with the sculpting tool. The accompanying video contains an overview with live screen captures that show the different tools in action, a video of an artist using the system, and turntable animations of all five paintings.

The "Cat and Mouse" painting is shown in Figure 3.10 from three different viewpoints. As the close-up in Figure 3.11 shows, the cat's tail is depicted with strokes that do not conform to the proxy geometry's surface. By painting off surface, the artist gave the tail its rough, comic look. The whiskers on the cat and mouse demonstrate strokes painted in space using the hair tool. Figure 3.12 depicts an "Autumn Tree" from front and top views. In the bottom row of the figure, a rendering of the stroke centerlines is blended with the proxy geometry. It shows that the leaves are painted in the space surrounding the rough canopy geometry. "Captain Mattis" is shown in Figure 3.16. The sculpting tool was used to sculpt the Captain's beard and eyebrows. The bottom row of Figure 3.16 visualizes the original, unsculpted head, the sculpting strokes, and the final painted result. The "Angry Bumble Bee" in Figure 3.13 shows how the hair and feather tools can be used to create a fluffy appearance. The "Wizard vs. Genie" painting shown in Figure 3.15 is our most complex example. Facial features and

Example	Triangles	Strokes	Splats
Autumn Tree	29k	21k	138k
Captain Mattis	6.6k	5k	40k
Cat and Mouse	7.5k	5k	130k
Angry Bumble Bee	6.3k	20k	304k
Wizard vs. Genie	30k	24k	452k

Table 3.1: Statistics about the number of triangles in the proxy geometry, the number of paint strokes, and the number of splats generated in a typical view by the stamping algorithm described in Chapter 2.

cloth wrinkles were sculpted using OverCoat's sculpting tool, and the smoke was given a fuzzy appearance by using the random offset feature of the level set tool.

Figure 3.17 demonstrates the advantage of OverCoat over more traditional methods that restrict 3D paintings to conform tightly to the surface of scene objects. The left column in this figure shows the 3D painting as created by the artist. To create the right column, we reprojected all paint strokes onto the zero level set so that they lie exactly on the proxy geometry. In the reprojection, the silhouette of the captain's arm becomes a precise line without stylization, revealing the smooth nature of the underlying 3D geometry. Likewise, the bee's fuzzy body and hairstyle lose their expressive quality.

3.6 Limitations and Future Work

While the objective terms and embedding tools we presented were successfully used to create all the 3D paintings in this thesis, they are by no means intended to be conclusive. We believe that our system is flexible enough to allow for other embedding semantics to be implemented easily and thus presents a good basis for creative development.

In our system, the artist must paint all lighting and texture information by hand. While giving the artist the utmost level of freedom of expression, this approach requires more manual work, especially in the context of dynamic lighting. Future work could incorporate ideas from Meier's painterly rendering system [Meier, 1996] or WYSIWYG NPR [Kalnins et al., 2002] to transfer scene lighting and shading information to painted strokes.

Stylized 2D paintings often exhibit view-dependent shape changes. Our system cannot support such changes, since the 3D canvas's structure is independent of camera view. Future work could incorporate ideas from view-dependent geometry [Rademacher, 1999] into the 3D canvas authoring process.

The users of our software were successful in using Maya and ZBrush for modeling in conjunction with OverCoat. However, the system could be improved by integrating sketch-based modeling and deformation concepts [Igarashi et al., 1999, Nealen et al., 2007] directly into OverCoat.

Finally, the system discussed so far accommodates only static objects. In the next chapter, we present a concept and method to extend the system to support expressive animated characters.

Figure 3.10: "Cat and Mouse": The fuzziness of the cat's fur was achieved with paint strokes that come off the surface. The whiskers were painted using the hair tool.

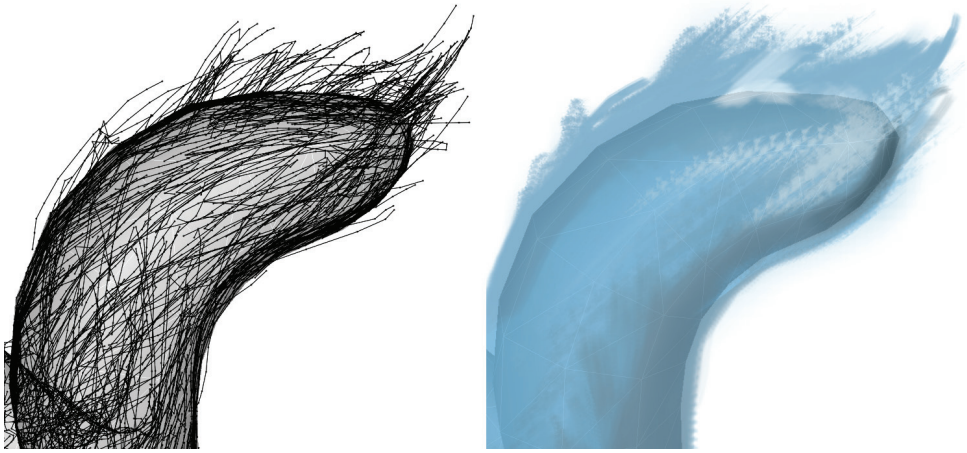


Figure 3.11: A close-up view of the cat's tail that shows how the characteristic look was achieved with paint strokes outside of the proxy geometry silhouettes.

Figure 3.12: "Autumn Tree": Leaves are individually painted strokes on offset levels of rough geometry representing canopies.

Figure 3.13: "Angry Bumble Bee": OverCoat allows painting hair and fur, as well as other structure that may not be easily represented using textured meshes.

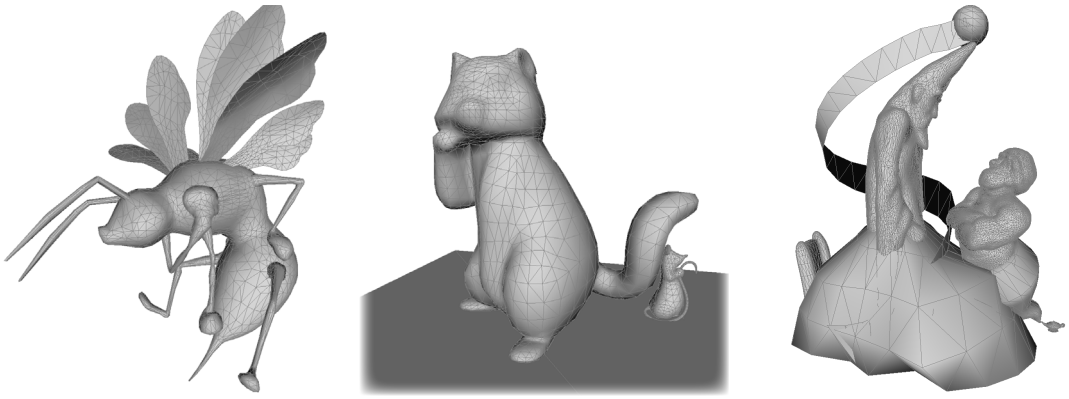


Figure 3.14: This figure shows the proxy geometry used for the "Angry Bumble Bee," "Cat and Mouse" and "Wizard vs. Genie" examples.

Figure 3.15: "Wizard vs. Genie": Since OverCoat has a unified representation for both surface and space, it is easy to depict clouds and other volumetric effects, by painting on offset surfaces. Effects such as the clouds in these images would be difficult to achieve with texture painting techniques. The beard of the wizard, most apparent in the image on the bottom right, shows an exemplary use of the feather tool.

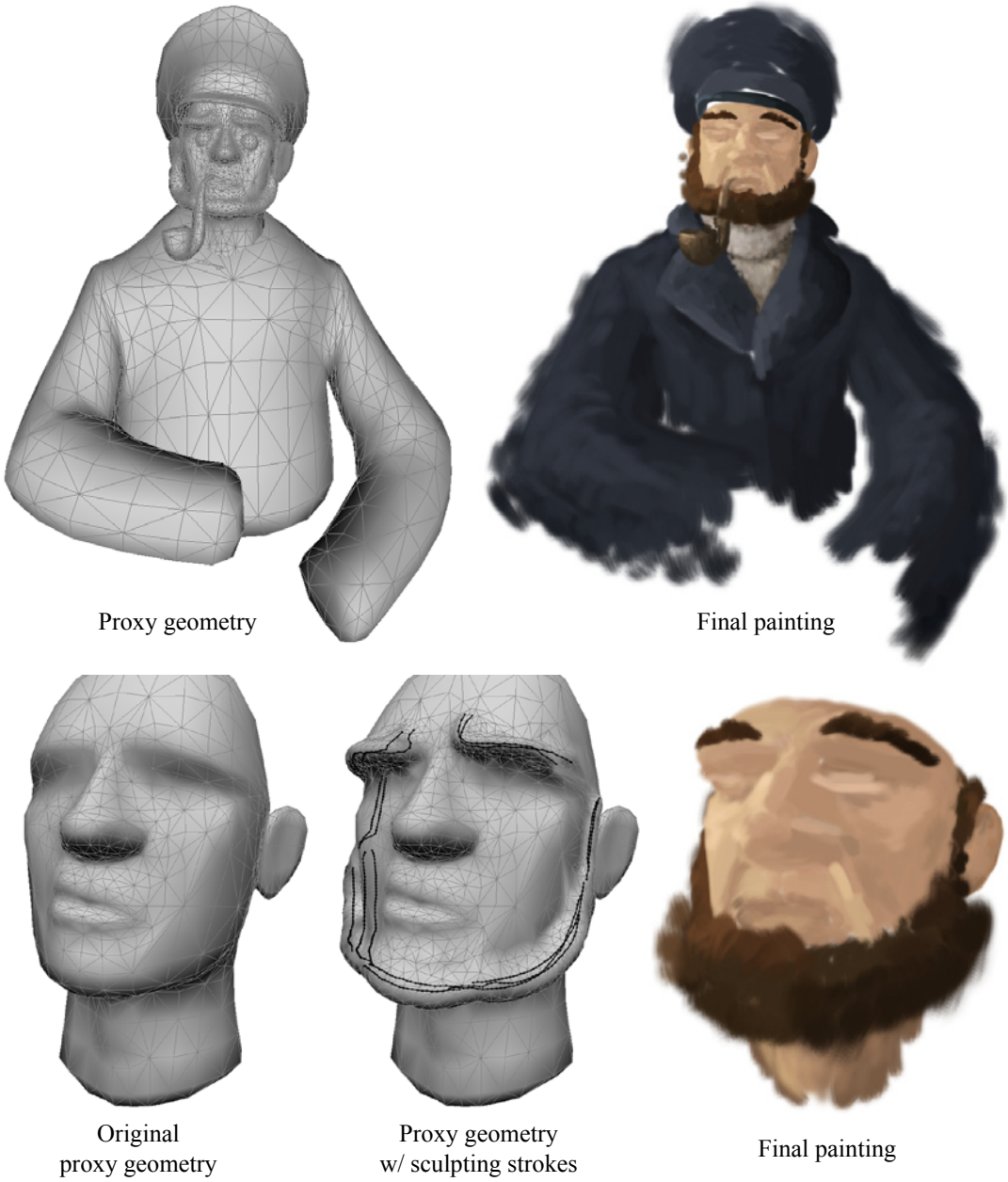


Figure 3.16: "Captain Mattis": Top row: Finished painting, paint strokes, and proxy geometry. Bottom row: original proxy geometry, geometry with sculpting strokes, and final painting.

Figure 3.17: The left column shows excerpts of the original paintings. In the right column, all strokes were projected onto the zero level set in order to highlight the benefit of our embedding method

CHAPTER 4

Animating 3D Paintings

In the two previous chapters, we have established methods that allow an artist to paint in three-dimensional space and render the resulting brush strokes in a fashion that resembles a traditional painting from any point of view. The missing piece to achieving our research goals formulated in the introduction is the ability to use these methods for animated protagonist characters. This goal is addressed in the current chapter, building on the existing system and guiding principles.

First, we show how to associate the movement of painted strokes with the movement of a character's proxy geometry so that 3D paintings can be deformed using standard rigging tools, regardless of the particular rigging algorithm employed. Next, we propose a configuration-space keyframing algorithm for authoring pose-dependent stroke effects. This mechanism allows stroke opacity or movement to be keyframed to positions in a configuration space that includes animation variables such as character pose parameters. With this mechanism, artists can create pose-dependent touch-ups to fine tune the look of a character, or larger-scale effects such as an animated facial expression. By including not only character pose but also camera and light position in the configuration space, our algorithm offers the opportunity to handle level-of-detail, view-dependent effects, and painted lighting changes with the same system. Finally, during animation, our system supports stroke-based temporal keyframing for one-off effects. Together, these tools subsume some of the work typically handled by traditional pipeline tools: a character's proxy geometry and rigging controls

can be approximate and simple, since our painterly authoring system allows the details of appearance and deformation to be painted directly by the artist.

Our primary scientific contribution in this chapter is a system for painterly character authoring and animation that provides direct control over expressive, animated character appearance. In realizing this system, we make several technical contributions, including a stroke-skinning algorithm, stroke-based deformation tools, a configuration-space stroke-keyframing method, a configuration-space interpolation algorithm, and a stroke-based temporal keyframing function. We demonstrate several characters authored with our system that exhibit painted effects difficult to achieve with traditional animation tools.

4.1 Background

The articulated animation of 3D paint strokes has not been an active topic for research thus far. The authors of WYSYWYG NPR [Kalnins et al., 2002] demonstrate one example of an animated character where paint strokes are deformed along with the surface to which they adhere. We adopt the same concept in our stroke-skinning system in a flexible way that does not restrict strokes to adhere to the proxy surface.

The configuration-space interpolation technique we propose was heavily inspired by pose-space deformation [Lewis et al., 2000], a method that adds and interpolates pose-specific sculpted shapes on top of meshes deformed using skeletal subspace deformation. Our contribution in this respect is a novel interpolation method that reduces the amount of awareness the user needs to have about the interpolation process, which is critical for our goal of providing an intuitive character authoring workflow that is based on the painting input metaphor.

4.2 Workflow Considerations

In our work, we focus on character animation authoring, which is the most challenging matter in the area of articulated animation. Character authoring traditionally includes modeling, rigging, and texturing a character, while animation involves setting the values of its rig parameters over time in order to create movement. We leverage the traditional character animation pipeline and show how to enhance it with stroke-based painting.

Authoring During the authoring phase, the artist creates a proxy model and rig for the character using traditional techniques. Since much of the character's detail will ultimately come from painting, the character's proxy geometry and

rig need only be an approximate representation. The artist then shapes the overall appearance of the character by painting it in a 3D painting system, such as the one we described in the two previous chapters. Our skinning deformation algorithm (Section 4.3) can automatically move the painted strokes together with the proxy geometry as the character is posed, allowing the artist to see the character in any configuration.

The artist shapes the overall appearance in this way, but may desire more fine-scale changes to take place at the stroke level when a particular pose is achieved. For example, when activating a facial blend shape, the artist may wish to include painted changes in the character's facial expression, as demonstrated in Figure 4.1. Our configuration-space keyframing system (Section 4.4) supports this functionality. The character's rig parameters, together with other scene variables such as light position, define the scene's configuration space. The artist can keyframe the opacity or position of any stroke to the current configuration-space variables. Our configuration-space interpolation algorithm (Section 4.4.3) ensures that the keyframes are smoothly interpolated.

Figure 4.1: A comparison of proxy geometry (left), a result after skinning deformation (middle), and after skinning and configuration-space keyframing (right). Configuration-space keyframing allows the artist to add stroke animation for features that are not present in the geometry or rig, such as the eyebrows, the cheeks, the pointed hairs, and the eye motion of this dog.

Animation In the authoring phase, the artist has created a parametric model that defines paint stroke positions and properties for any point in the high-dimensional configuration space. The goal in the animation phase is to define the temporal sequence of actions of all assets in the scene, which, in technical

terms, is achieved by creating a map from time to a point in the configuration space. This map can be created with any traditional animation software. During playback, our system deforms the painted strokes and interpolates any configuration-space keyframes on stroke movement or opacity that were created in the authoring phase. Our system supports an additional keyframing mechanism that allows stroke opacity or position to be keyed to the animation timeline (Section 4.5) independent of the configuration space. This functionality permits one-off effects relevant to the particular context of the animation.

4.3 Skinning Deformation

Skinning deformation connects the movement of strokes to the deformation of the proxy object, allowing 3D paintings to be deformed by traditional character rigs. We employ an algorithm based on linear-blend skinning [Magnenat-Thalmann et al., 1988] which is similar to the free-form deformation technique proposed by Singh and Kokkevis [Singh and Kokkevis, 2000] to accomplish this task. While linear-blend skinning is traditionally used in the context of a skeleton, we blend skinning transformations computed for each vertex of the proxy geometry. These transformations capture the space deformation in a local neighborhood around the vertices from a designated rest pose to the target pose. For a given vertex v_i and its one-ring V_i^1, our algorithm solves an orthogonal Procrustes problem to find a least-squares optimal rigid motion $\mathbf{M}$ that aligns all vertices in V_i^1 from the rest pose to the target pose.

For the actual skinning deformation, paint strokes are transformed from the rest pose by a convex combination of $\mathbf{M}_i$, where each point on a paint stroke has its own set of weights:

$$\mathbf{M} = \sum_i w_i \mathbf{M}_i. \tag{4.1}$$

Although computing good skinning weights is difficult in some applications, in our case paint strokes are typically located relatively close to the proxy geometry's surface. As a result, associating each stroke with the closest geometric primitive of the mesh is effective. If the closest point on the surface lies within a triangle, the barycentric coordinates of the closest point are used as weights w_i. When the closest point lies exactly on an edge or vertex, the barycentric coordinates of any shared triangle delivers the correct weights. Figure 4.2 shows a simple result of this skinning deformation method.

Newly applied paint strokes will typically be painted with respect to the currently active pose. Storing the stroke positions in the active pose would require storing the position of V^1 for all associated vertices and recomputing the transformations $\mathbf{M}_i$ whenever the the pose changes. We avoid this computation by storing the positions of all paint strokes in the rest pose, regardless of the pose

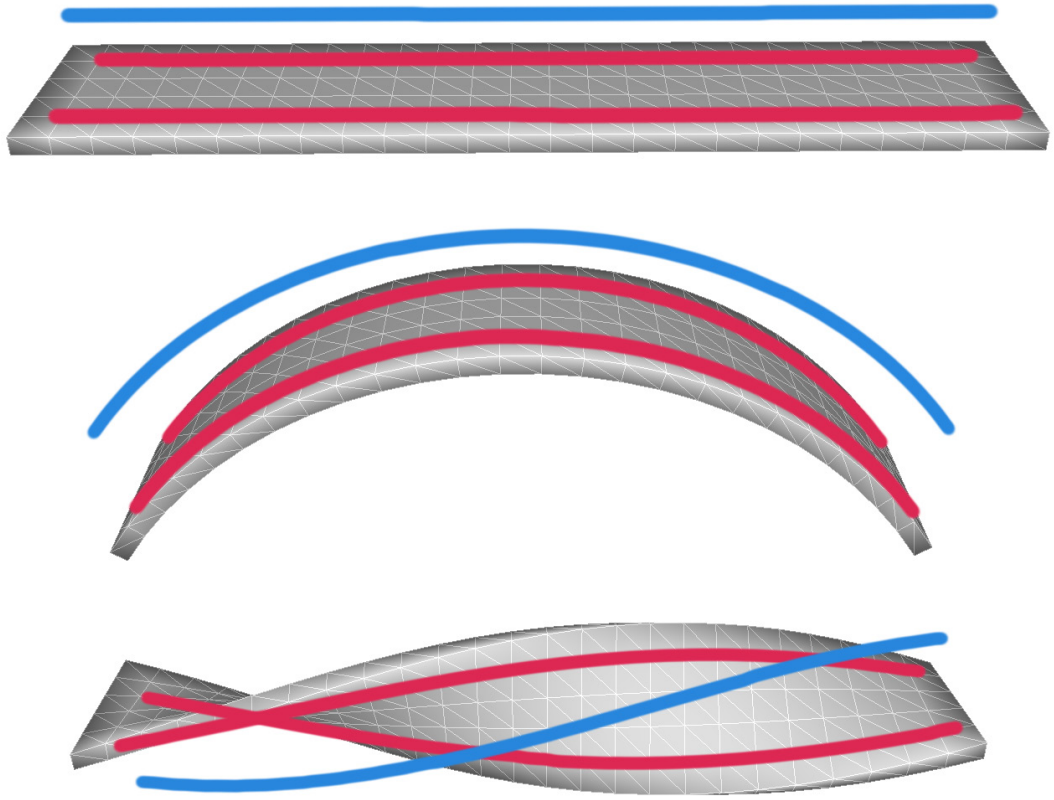


Figure 4.2: This figure shows a simple result of our skinning deformation algorithm. The red strokes are embedded on the surface, while the blue stroke is embedded above it. Since we compute skinning transformations as rotations around the centroids of mesh primitives, they naturally extend to the space surrounding the object.

in which they were painted. We compute the skinning weights in the active pose, and then apply $\mathbf{M}^{-1}$ to each stroke point to find its position in the rest pose.

4.4 Configuration-Space Keyframing

The skinning deformation described in the previous section allows control over the gross movement of strokes. However, a core advantage of 3D painting is that it gives the artist the power to directly paint details and subtle expressive elements that are not present in the proxy geometry and difficult to realize through modeling operations. Since animating such elements with traditional rigs may be cumbersome or impossible, we give artists detailed stroke-level control using a configuration-space stroke keyframing system.

The configuration space is a high-dimensional space defined by the character's rigging controls (joint angles, blend-shape variables, etc.) and other scene parameters such as light positions. A keyframe is a point in this space together with desired stroke information at that point. Our system allows stroke opacity and stroke position to be keyframed. We choose this set based on the needs encountered during the creation of our example results. However, other quantities such as color or stroke width could also be keyframed in the same manner.

Since a 3D painting may consist of thousands of paint strokes, we encourage the artist to partition the painting into layers that have a semantic meaning with respect to animation. For example, if an eyebrow is to be animated, all strokes belonging to the eyebrow should be placed in a separate layer. Keyframes are always set with respect to an entire layer. For example, a stroke position keyframe captures the positions of all paint strokes in a layer.

4.4.1 Opacity Keyframing

Opacity keyframing is conceptually simple: it involves a single opacity measure that can be set to any value between 0 and 1 at any given point in the configuration space. The opacity value should modulate the opacity of all strokes in the layer, allowing a whole group of paint strokes to fade in or out during the animation. The difficulty lies in rendering: as described in Section 2.1.3, it is impossible to achieve a uniform opacity attenuation when compositing multiple primitives without treating every pixel and fragment separately.

We resort to a simple multi-pass strategy independent of the specifics of the rendering method. The core observation is that the blend between two images where one layer was omitted in one of the images produces a semi-transparent appearance of that layer. To express this concept more formally, assume that a scene $\mathcal{S}$ consists of N layers, $\mathcal{S} = \{\mathcal{L}_1, ..., \mathcal{L}_N\}$, and $I(\mathcal{S})$ is the image generated by rendering this scene without considering layer opacities. If layer $\mathcal{L}$ should appear with opacity α and all other layers fully opaque, then the desired image $\mathcal{I}$ can be computed as a linear interpolation between $I(\mathcal{S})$ and $I(\mathcal{S} \setminus \{\mathcal{L}_k\})$:

$$\mathcal{I} = \alpha I(\mathcal{S}) + (1 - \alpha) I(\mathcal{S} \setminus \{\mathcal{L}_k\}).$$

For the more general case where each layer has an arbitrary opacity $\alpha_i \in [0, 1]$, we sort the layers such that $\alpha_i \leq \alpha_j$ if $i < j$. Setting $\alpha_0 = 0$, the desired image can be expressed as

$$\mathcal{I} = \sum_{i=1}^{N} (\alpha_i - \alpha_{i-1}) I(\{\mathcal{L}_j : j \geq i\}). \tag{4.2}$$

In the notation used above, the product of a scalar with an image corresponds to the component-wise product of each image element with the scalar, and the addition or summation of images corresponds to the component-wise addition of the respective image elements.

Computing $\mathcal{I}$ this way requires N render passes. In most cases, this number can be reduced by grouping layers with equal opacity values, which is equivalent to dropping all terms in Equation 4.2 where $\alpha_i = \alpha_{i-1}$. However, the complexity of rendering with layer opacities is still linear in the number of unique layer opacity values, which results in a significantly reduced rendering time if the opacity of many layers is keyframed.

4.4.2 Stroke Position Keyframing

Since keyframes are set per layer, a positional keyframe sets a value for all stroke points of all strokes in a layer. Setting these point positions manually would be cumbersome, and therefore we provide the artist with two tools to deform paint strokes: smudging and expansion. Both tools share a common input metaphor with the painting process: they operate on stroke paths embedded in space using the 3D painting system. Smudging moves paint stroke points along the direction of the embedded stroke, while expansion moves stroke points along the normal defined by the implicit canvas. Both tools only affect stroke points within a certain radius around the embedded path. The magnitude of the displacement can optionally be modulated by an arbitrary falloff function. We have found the cubic step function $2r^3 - 3r^2 + 1$, where r is the distance to the stroke divided by the tool radius, to produce nice results in most cases. Figure 4.3 illustrates the operations of these shaping tools.

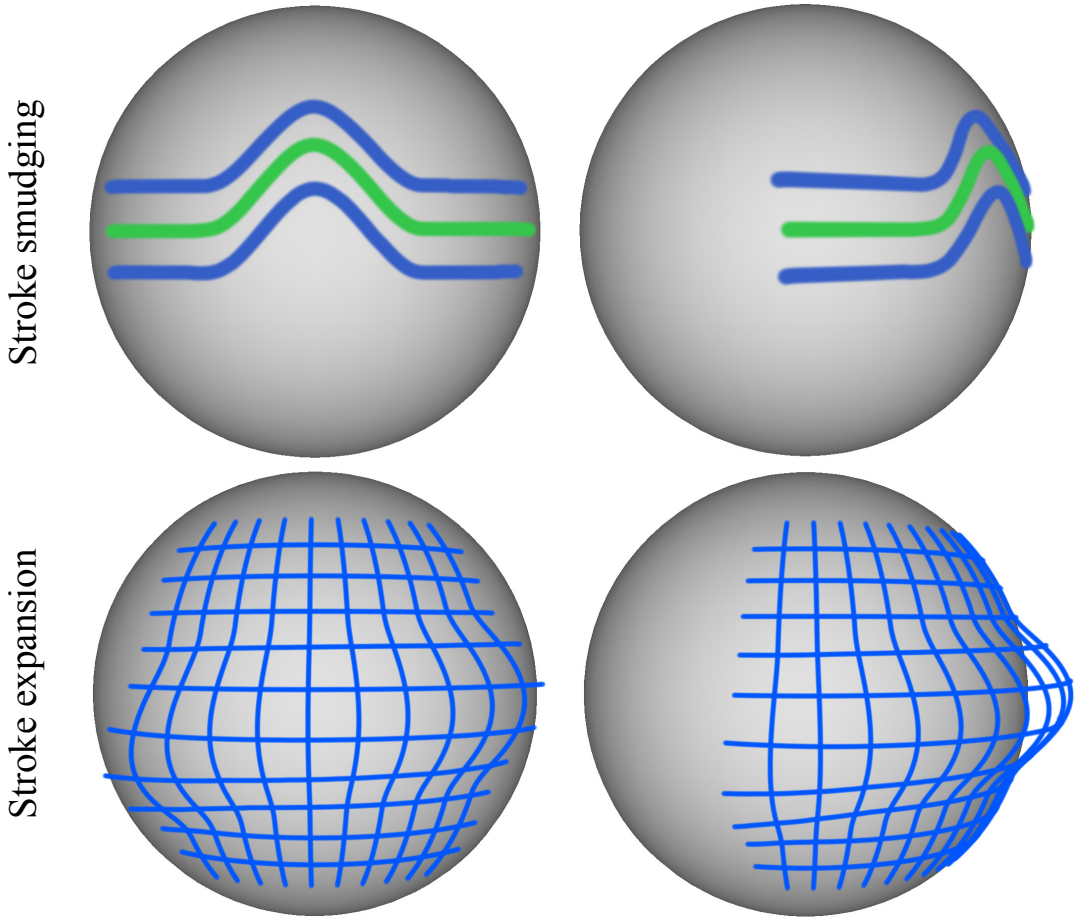


Figure 4.3: This figure shows the result of applying the stroke smudging tool to three paint strokes that were originally straight (top row), and of applying the stroke expansion tool to a grid of strokes that was originally painted on the sphere's surface (bottom row).

4.4.3 Keyframe Interpolation

Given a set of keyframes in configuration space, our system needs to be able interpolate them to all other points in configuration space. Several factors make this problem challenging:

1. The configuration space can be high dimensional, precluding geometric approaches like natural-neighbor coordinates [Sibson, 1981], finite element thin-plate spline approaches [Hegland et al., 1997], and bounded biharmonic weights [Jacobson et al., 2011].
2. The interpolant should be continuous both as a function of the interpolated configuration-space point and as a function of the keyframe locations.
3. The interpolant should not depend on parameters orthogonal to the keyed subspace: if the user specifies keyframes for an elbow, the configuration of other joints should not affect the results. This requirement precludes most radial basis functions (RBFs), including those based on thin-plate splines, Shepard interpolation [Shepard, 1968], and Gaussian process techniques (with the notable exception when the RBF is a Gaussian).
4. Outside the region bounded by the keyframes, the interpolant should level off to the value at the closest keyframe. Extrapolation requires large negative weights, which tend to amplify defects in strokes, so we found it best to avoid it.

Prior methods for pose-space interpolation used radial basis functions [Lewis et al., 2000, Igarashi et al., 2005] or even linear interpolation [Baran et al., 2009]. The latter approach cannot guarantee interpolation and Igarashi and colleagues' RBFs do not separate by dimension, violating requirement 3. Lewis and colleagues use Gaussian RBFs, which have the drawback that the covariance matrices Σ need to be specified. Kernels that are too small lead to abrupt transitions, while kernels that are too large lead to poor conditioning and overshooting. As Figure 4.4 shows, when keyframes are not uniformly distributed, it may be impossible to find Σ's that yield good results.

We therefore propose a different scheme for interpolation developed on the basis of Franke's ideas [Franke, 1977]. Let $(\mathbf{x}_i, y_i)$ be n keyframes, with $\mathbf{x}_i$ the locations in configuration space and y_i the values (e.g., layer opacities or stroke positions) and let $\hat{\mathbf{x}}$ be the point in pose space at which we would like to compute the interpolant. Like RBFs and several other schemes, our interpolant is linear in y_i, but not in $\mathbf{x}_i$ or $\hat{\mathbf{x}}$. This property implies that, for a particular $\hat{\mathbf{x}}$, we only need to find weights w_i such that $\hat{y} = \sum_{i=1}^{n} w_i y_i$ and then we can interpolate many different functions efficiently. At a high level, we start by defining n smooth "bumps" $\phi_i(\mathbf{x})$ over the configuration space, each centered at a keyframe, so $\phi_i(\mathbf{x}_j) = \delta_{ij}$. If we just used these bumps as weights, the interpolant derivative at every keyframe with respect to $\hat{\mathbf{x}}$ would be zero (see

Figure 4.5, and also [Lewis et al., 2000]), which leads to stuttering motions. We therefore locally estimate the gradient at each keyframe and use the bumps to blend the resulting linear functions.

The bumps $\phi_i(\mathbf{x})$ are defined using a smooth step function. A Hermite cubic step could be used, but we have found the quadratic-linear-quadratic step for $\gamma = 0.2$ to produce better results because its maximal derivative is smaller, which causes the interpolation to look smoother:

$$S_\gamma(t) = \begin{cases} 0 & \text{if } t \leq 0 \\ 1 & \text{if } t \geq 1 \\ \frac{t^2}{2\gamma - \gamma^2} & \text{if } 0 < t \leq \gamma \\ 1 - \frac{(1-t)^2}{2\gamma - \gamma^2} & \text{if } 1 - \gamma < t < 1 \\ 0.5 + \frac{t - 0.5}{1 - \gamma} & \text{if } \gamma < t \leq 1 - \gamma. \end{cases} \tag{4.3}$$

To compute the bump ϕ_i, we project $\hat{\mathbf{x}}$ onto the line connecting $\mathbf{x}_j$ to $\mathbf{x}_i$ for each $j \neq i$ and evaluate S for the the resulting line parameter value, so that the result is 1 if the projection is at or past $\mathbf{x}_i$ and 0 if it is at or past $\mathbf{x}_j$. We then take the product of the smooth steps toward each keyframe

$$\phi_i'(\hat{\mathbf{x}}) = \prod_{j \neq i} S\left(\frac{(\hat{\mathbf{x}} - \mathbf{x}_j) \cdot (\mathbf{x}_i - \mathbf{x}_j)}{\|\mathbf{x}_i - \mathbf{x}_j\|^2} \right) \tag{4.4}$$

and normalize the result: $\phi_i(\hat{\mathbf{x}}) = \phi_i'(\hat{\mathbf{x}}) / \sum_j \phi_j'(\hat{\mathbf{x}})$. These bumps have several useful properties: they vary between 0 and 1, they are as smooth as S, they are constant in directions orthogonal to the affine subspace spanned by $\mathbf{x}_i$'s, and $\phi_i(\hat{\mathbf{x}}) > 0$ precisely when $(\hat{\mathbf{x}} + \mathbf{x}_i)/2$ is in the Voronoi region of $\mathbf{x}_i$. However, because they are smooth and reach their extreme values at the keyframes, the gradient of ϕ_i at each $\mathbf{x}_j$ is zero, and if we use $\hat{y} = \sum_i \phi_i(\hat{\mathbf{x}}) y_i$, we will get the stepping artifacts discussed above.

The stepping artifacts can be avoided by estimating the gradient $\mathbf{g}_i$ from the keyframe values y_i and using it as the interpolant close to keyframes:

$$\hat{y} = \sum_{i=1}^{n} \phi_i(\hat{\mathbf{x}}) \left(y_i + \mathbf{g}_i \cdot (\hat{\mathbf{x}} - \mathbf{x}_i) \right). \tag{4.5}$$

We estimate $\mathbf{g}_i$ by minimizing a least-squares energy based on the distance and direction of neighboring keyframes:

$$E(\mathbf{g}_i) = \sum_{j \neq i} \frac{a_j^2}{\|\mathbf{x}_j - \mathbf{x}_i\|^2} \left(\mathbf{g}_i \cdot (\mathbf{x}_j - \mathbf{x}_i) - b_j (y_j - y_i) \right)^2, \tag{4.6}$$

where a_j quantifies how much we trust the j^{th} neighbor and b_j is used for forcing the gradient at extreme keyframes to zero to avoid overshooting. We use the

bumps to evaluate $a_j = \phi_i(\alpha \mathbf{x}_i + (1-\alpha)\mathbf{x}_j)$ with $\alpha = 0.5$. To set the gradient to zero at extreme points, we determine whether there are keyframes on both sides of $\mathbf{x}_i$ in the $\mathbf{x}_j$ direction:

$$b_j = S\left(-\beta \cdot \min_{k \neq i} \frac{(\mathbf{x}_k - \mathbf{x}_i) \cdot (\mathbf{x}_j - \mathbf{x}_i)}{\|\mathbf{x}_k - \mathbf{x}_i\| \|\mathbf{x}_j - \mathbf{x}_i\|}\right) \tag{4.7}$$

with $\beta = 5$. The problem of minimizing the energy E is typically under constrained, as no information is available in directions orthogonal to the affine subspace of the keyframes. To satisfy requirement 3, the gradient needs to be zero in these directions. This condition is achieved by adding the Tikhonov regularization term $\epsilon\|\mathbf{g}_i\|^2$ to the energy function with $\epsilon = 0.2$. We then minimize E using singular value decomposition.

As discussed earlier, this interpolation method only depends linearly on the y_i's. Example basis functions generated with this scheme are shown in Figure 4.6. Because configuration space is hard to visualize, we show an application of our interpolation to spatial keyframing [Choi et al., 2008] in Figure 4.7. Although the presented method has parameters α, β, γ, and ϵ, they are independent of the domain of $\mathbf{x}$, unlike the Σ's in RBF interpolation. We used the same parameter values for all of the results shown in this chapter ($\alpha = 0.5$, $\beta = 5$, $\gamma = 0.2$, $\epsilon = 0.2$). Another concern is that, like prior methods, our scheme does not always satisfy requirement 4: the gradients at extreme keyframes may not end up zero and overshooting can occur. Although it is not hard to construct synthetic examples where this happens, we did not run into this issue when producing our animated characters.

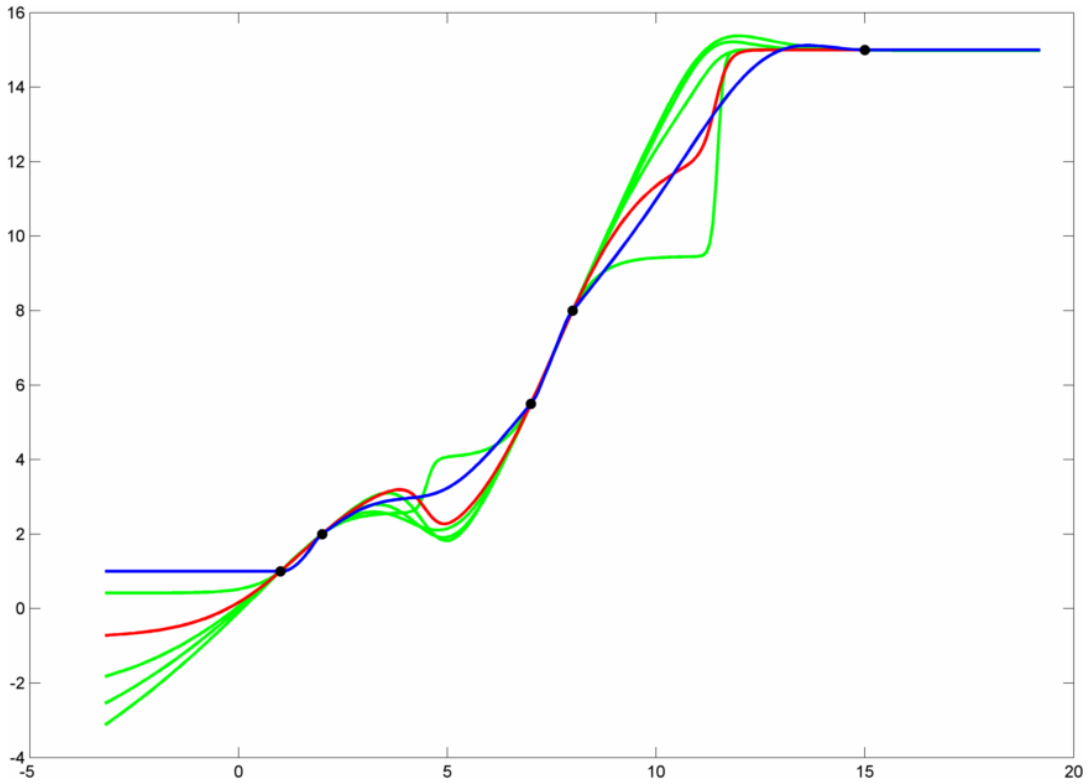


Figure 4.4: A comparison of our method (blue) and Gaussian radial basis functions with kernel radii $1, 2, 3, 4, 5$ (green and red). The RBF interpolant with kernel radius 2 is shown in red. This radius is both large enough to cause an overshoot after the point at $x = 2$ and small enough to have a step around $x = 12$, demonstrating that picking a good kernel radius can be an impossible task.

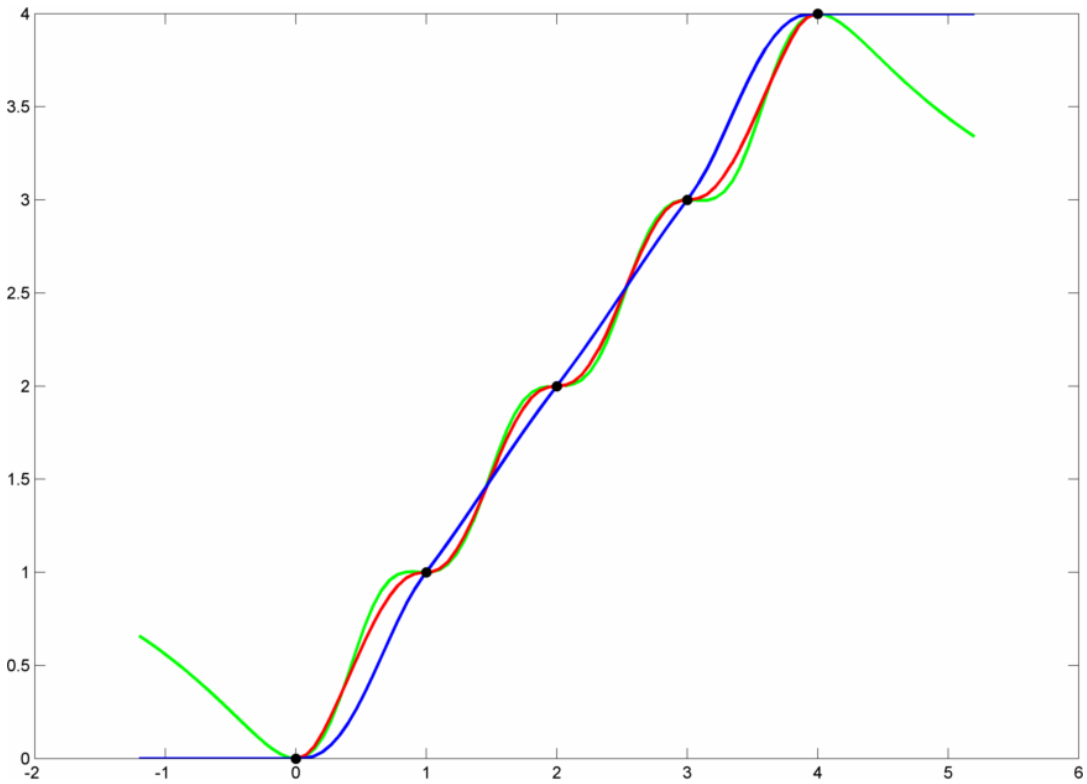


Figure 4.5: A comparison of our method (blue), Shepard interpolation with the $1/d^2$ kernel (green), and using our ϕ's directly (red). The task is to interpolate five regularly spaced points on a line. The steps produced by the other two methods manifest themselves as stuttering during animation playback.

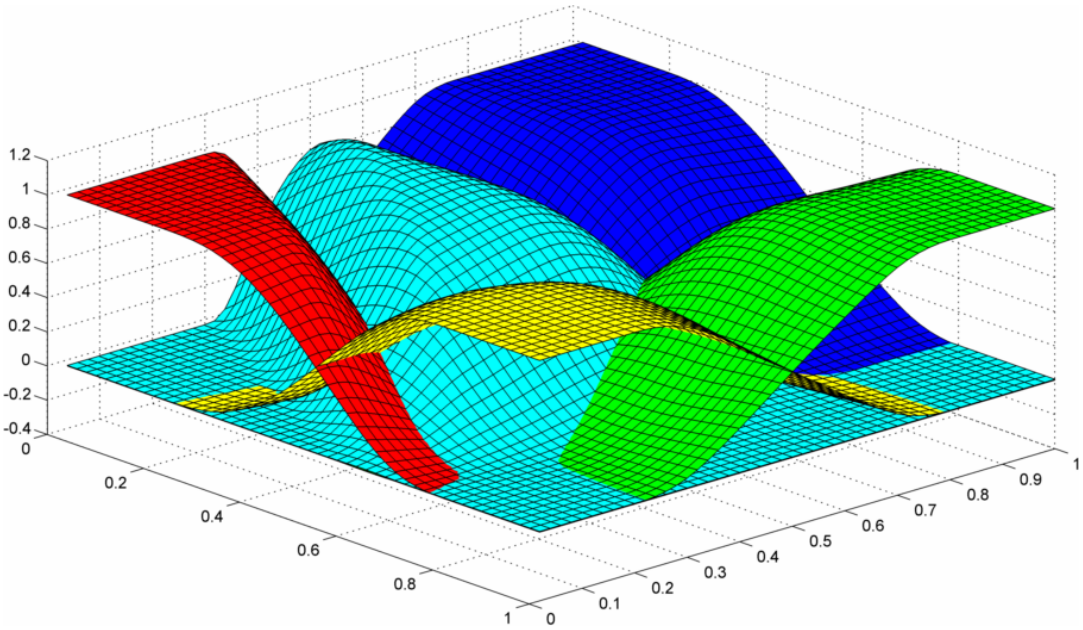


Figure 4.6: Basis functions w generated by our interpolation scheme in 2D. The "keyframes" are at $(0.2, 0.2)$, $(0.8, 0.8)$, $(0.2, 0.8)$, $(0.8, 0.2)$, and $(0.3, 0.4)$.

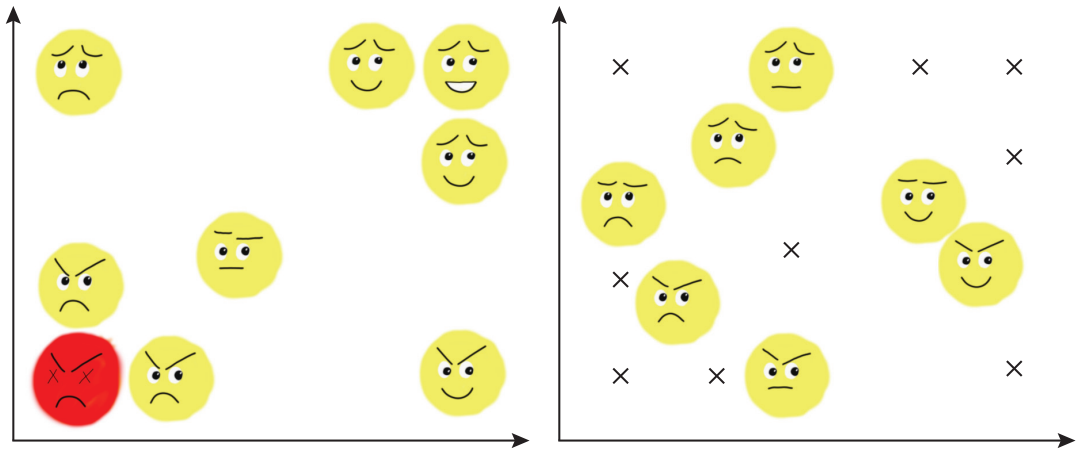


Figure 4.7: A painted smiley face has been keyframed to show various expressions. The configuration space is defined by the 2D position of the face. The left figure shows all keyframes at their respective 2D positions. The right figure shows interpolated faces superimposed on the keyframes.

4.5 Temporal Keyframing

The configuration-space keyframing system described above focuses on character authoring, and is used to incorporate stroke-based effects that are common to certain character poses or other repeated scene conditions. When a particular animation is created, the artist may wish to add one-off effects specific to one point in the animation's timeline. We enable this kind of edit by allowing stroke opacity and position to be keyed to the animation timeline.

Since animation takes place after authoring, temporal keyframes should override any existing configuration-space keys. To bound the regions where temporal keyframes are in effect, we employ the concept of "sentinel keys." In contrast to "normal" temporal keys, sentinel keys do not carry any scene parameter data (such as stroke positions or opacity values). Let the current animation time be designated by t. If t is between two normal temporal keyframes, the result of applying temporal keyframing should be the interpolation between the two keyframe values. Since we have a simple one-dimensional interpolation problem in this case, any interpolation suitable for normal keyframing can be used. For the sake of simplicity, we used the method described in the previous sections with good results. If t is between two sentinel keys (or between a sentinel key and the beginning/end of the animation), temporal keyframing is inactive. In this case, the keyframed value is determined entirely by the configuration-space keyframes.

To avoid discontinuities around a range in time where temporal keyframes exist, there needs to be a smooth transition between the value determined by the configuration-space keyframes and by the temporal keyframes. Therefore, if t is between a sentinel key and a temporal keyframe, the keyframed value is interpolated between the result of configuration-space keyframing and the value of the neighboring temporal keyframe. Figure 4.8 illustrates the transitions around a region of temporal keyframing.

4.6 Results

We authored and animated several example characters to demonstrate our skinning deformation system and the range of applications for our configuration-space and temporal keyframing methods. All characters are purely stroke-based paintings created with the stroke embedding technique described in the last chapter and rendered using mixed-order compositing.

The dog example (Figure 4.1) demonstrates both skinning and configuration space keyframing. In the rig's blend shape, only the snout, ears, and jaw move. We use stroke position keyframing to make the hair stand up, create eyebrows,

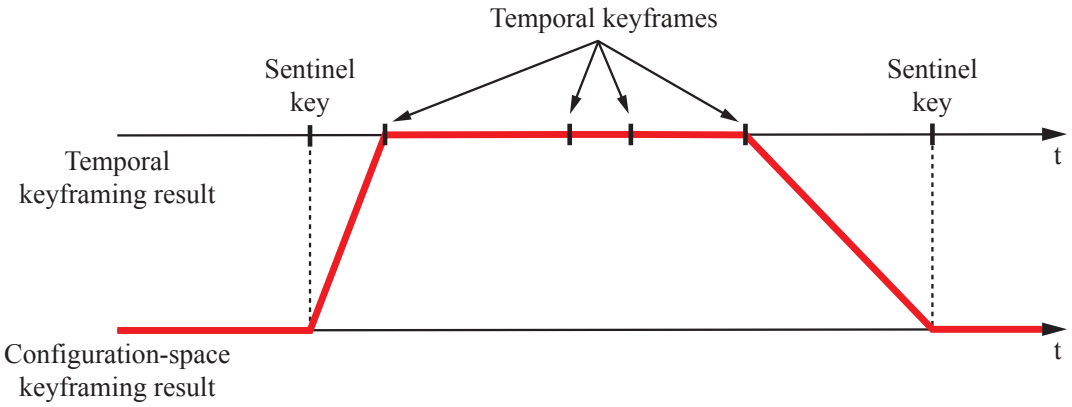


Figure 4.8: Temporal keyframes override the result of configuration-space keyframing. Sentinel keys are used to bound the regions where temporal keyframing is active. In this figure, the upper arrow represents the parameter values that result from evaluating the temporal keyframes, while the lower arrow represents the parameter values resulting from the configuration-space keyframes. Between the sentinel keys and their neighboring temporal keyframes, the results of temporal keyframing and configuration-space keyframing are interpolated to ensure a smooth transition.

animate the pupils, and pull the cheeks out. Opacity keyframing is used to add highlights and shadows around the cheeks and eyebrows.

Lighting is a crucial element in many animations. Figure 4.9 shows a painted lighting example in which a light source's position is included in the configuration space. Keyframes for stroke positions and opacities are added for different positions of the light source to create the illusion of a moving highlight, shadow, and shading. In this example, no actual lighting calculations are made. Everything is accomplished with configuration-space keyframing.

The blowfish animation (Figure 4.10) makes use of all aspects of our animation system. The mesh animation was created with blend shapes and skeletal rigging and exhibits the overall deformation of the body and the movement of the fins. The eyes, however, are fixed in the original animation and were animated by smudging and keyframing the pupils to a two-dimensional eye target position in the configuration space. The spikes were folded out with the smudging tool and keyframed to the blend-shape weight that corresponds to the blown-up shape of the fish. Spike-stretching is avoided by constraining all points along one spike to a single point on the surface. The blend-shape weight parameter also causes the cheeks and lips to turn red via opacity keyframing. As the fish inflates, the distances between paint strokes increase, leading to gaps in the surface. We

Figure 4.9: Painted lighting effects such as the highlight, shading, and shadow of this apple can be achieved by including the light's position in the configuration space and setting stroke position and stroke opacity keyframes to the light's position.

filled these gaps with additional paint strokes that are keyframed to appear only as the body expands. The difference is visible in the blowfish comparison segment in the video. The blinking eyes were realized by keying opacity to a separate parameter in the configuration space. Figure 4.11 shows three frames of the blowfish animation with and without the configuration-space effects.

Finally, the ballerina character, shown in Figure 4.12, has a detailed rig that allows the animated painting to rely heavily on stroke skinning. However, the input animation exhibits a number of rigging artifacts, including the leg protruding through the skirt and collapsing deformation in the shoulders. Such artifacts can easily be remedied with our configuration-space keyframing system, as is illustrated in Figure 4.13. In the particular case of the shoulders, we modified the behavior of the problematic strokes by keying new stroke positions to the angles of the shoulder joints using 10 keyframes to fix both shoulders. The leg protrusion was fixed with three pose keyframes to move the skirt upward. Apart from alleviating issues in the input animation, we also embellished the animation with an effect that causes the ballerina's tutu to twist in reaction to her pirouette. This effect was achieved with three temporal keyframe during her spin that marks the maximum deformation of the fabric.

4.7 Discussion

In this chapter, we have presented a system for authoring and animating painterly characters that incorporates much of the expressive freedom of 2D concept painting into the character animation pipeline. In essence, our work

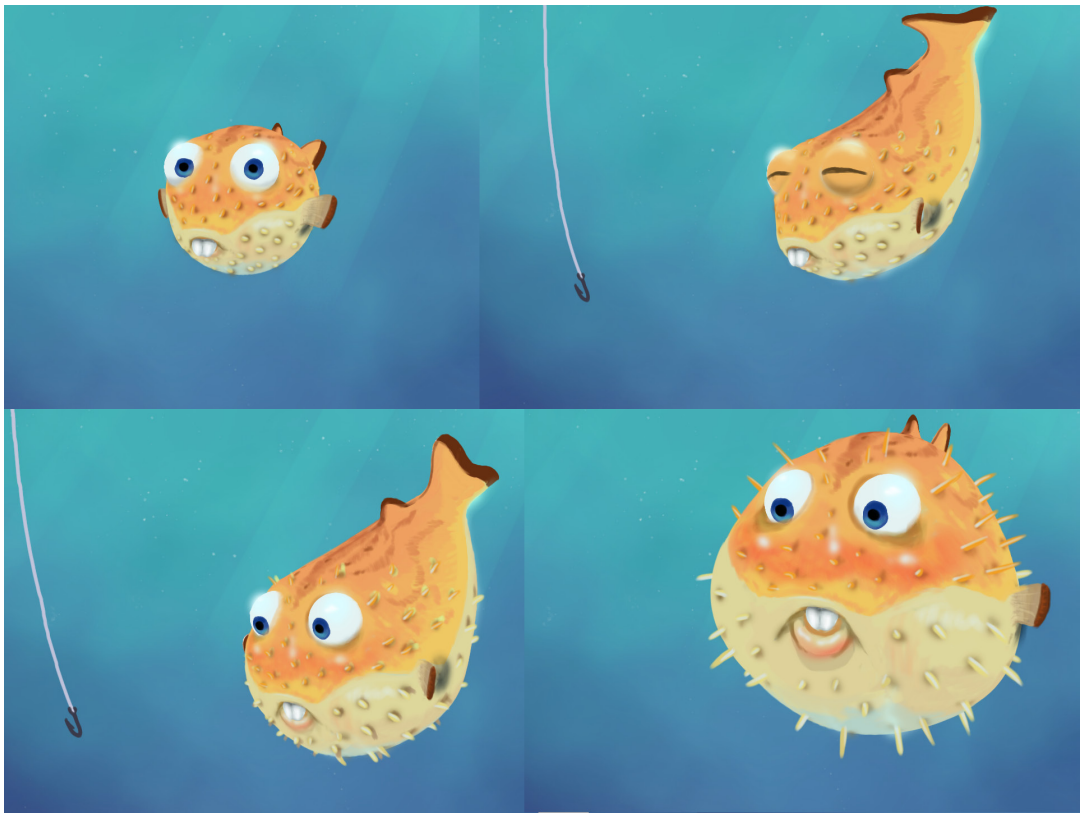


Figure 4.10: A blowfish gets a shock when catching sight of a fishing hook.

upgrades painting from its restricted role in concept design and texturing to become an integrated piece of expressive depiction that impacts modeling, rigging, posing, lighting, and rendering. We show how painted strokes can be deformed together with the character's surface. Our configuration-space and temporal keyframing system allows artists to fine-tune the movement of strokes in order to accomplish stroke-level effects that are difficult or impossible to achieve using more traditional modeling and animation tools. We have demonstrated results for facial animation, animated lighting, and full body animation ranging in style from comical and cartoony to fine-art impressionism.

An animated painting brings together two seemingly incompatible worlds since three-dimensional movement must be conveyed using a medium that has been static for thousands of years. Our system attempts to give the animator explicit control over this movement. Discovering the boundaries of this control and how to move past them to create effects and styles we have not yet dreamed is a future direction of research that relies on art just as much as it relies on science.

Although our system gives the artist new control over painterly animation, it also comes with many limitations that offer exciting opportunities for future

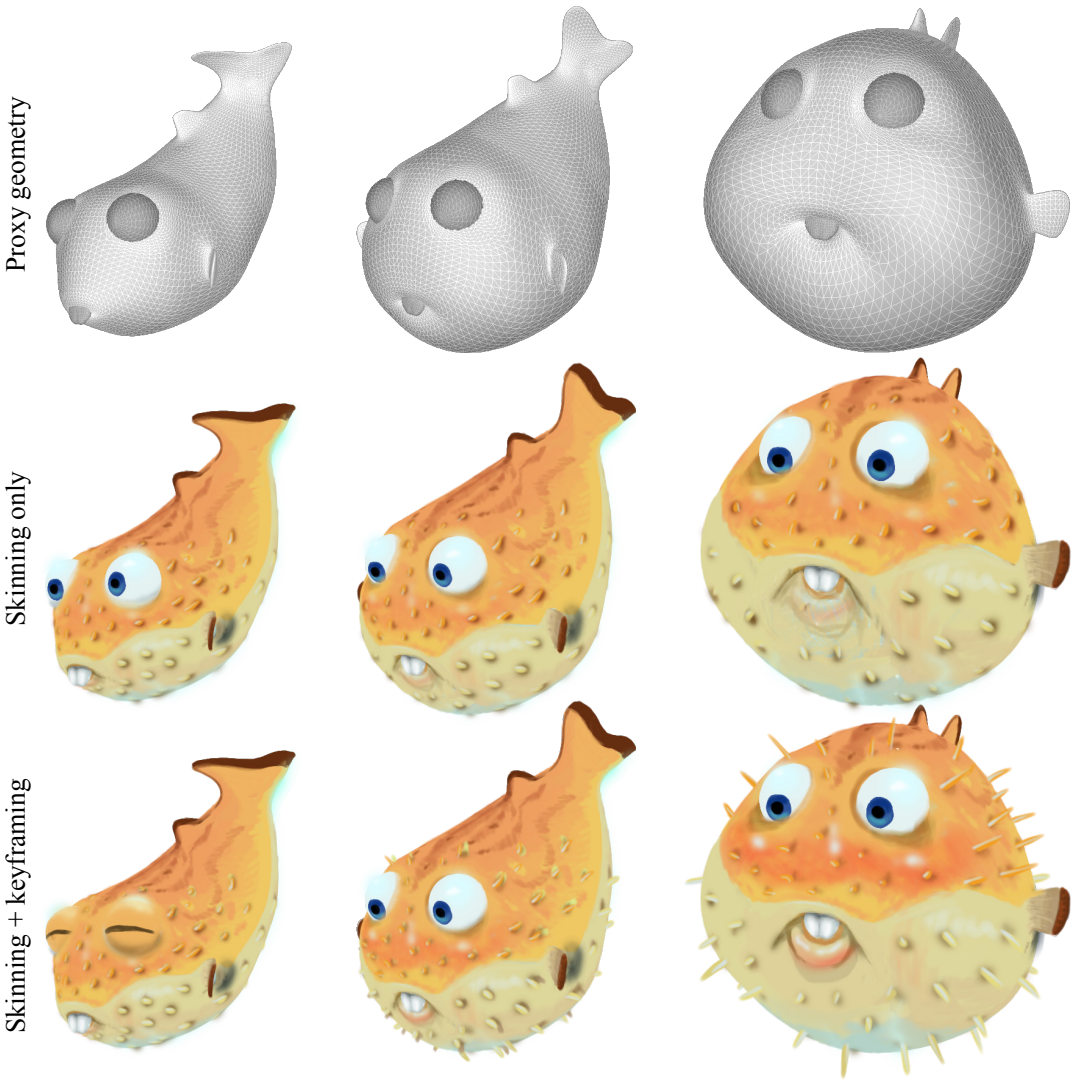


Figure 4.11: An illustration of the different authoring steps for the blowfish example. The top row shows the animated proxy geometry, the middle row shows the painted result with skinning deformation only, and the bottom row shows the result with configuration-space keyframing. The middle row lacks the blinking and spike animations, and shows holes in the blown-up pose because the original paint strokes have moved too far apart.

Figure 4.12: Poses of a painterly ballerina authored and animated with our system.

work. The freedom to shape the depiction of painted strokes in different poses, under different lighting conditions, and at different times can mean that character authoring becomes more laborious. The extra time required for painting is somewhat balanced by the need for a simpler geometric model and rigging deformations. Nonetheless, an interesting area of future work lies in exploring methods to fill a character with strokes without having to paint each and every one individually. Such a system must strike an effective balance between its automatic nature and the level of artistic control. Lighting represents a challenge in and of itself. Our current system allows painted lighting effects, such as the moving shadow, highlight, and shading on the apple in Figure 4.9. This example is somewhat simplistic, and the amount of work involved in creating extremely complex lighting effects can make authoring them impractical. Incorporating automatic lighting into our animation system is thus an important area of future work. As mentioned above, the key challenge involves automating the process without stealing the artists expressive control. Image-space effects, such as silhouettes, are not easily accomplished by our system. Incorporating the WYSIWYG annotation system [Kalnins et al., 2002] for silhouettes, creases, and hatching is thus an interesting future work direction. On the technical side, our configuration-space keyframe interpolation algorithm avoids stuttering when interpolating multiple keyframes, but does not provide precise control over the interpolation procedure. Generalizing the spline control typically found in animation packages to higher dimensions in an efficient way is a challenging area of future work.

The painterly animation and rendering system presented thus far enlivens painterly depictions of objects and characters with motion. An additional dimension of stylization can be found in the depiction of motion itself with the use of transient “annotations” in a still image or the individual frames of an

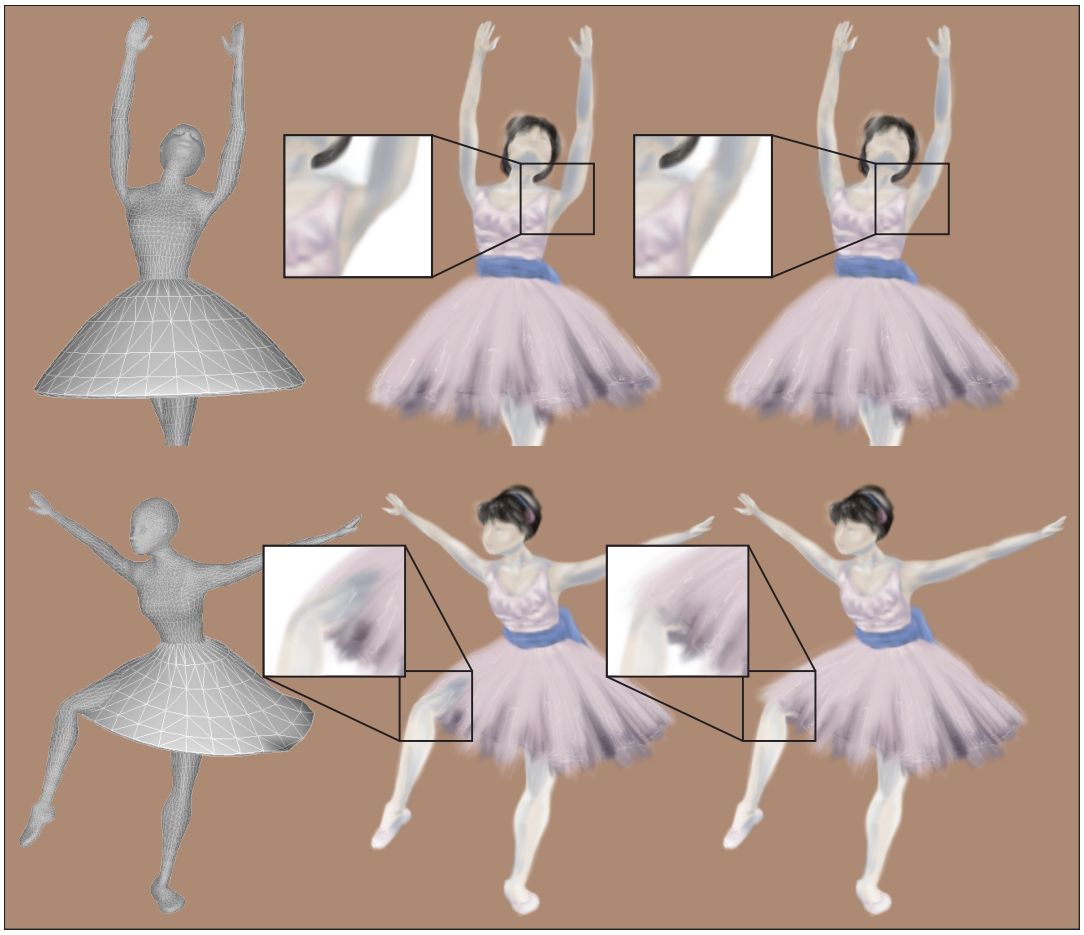


Figure 4.13: In the ballerina example, we used configuration-space keyframing (right column) to fix issues that arose from the proxy geometry animation (left and middle columns), such as the skinning artifacts in the shoulder regions and the leg protruding through the skirt.

animation. We have investigated this topic in the context of more traditional rendering and present a system for stylized rendering of motion in the next chapter.

CHAPTER 5

Stylized Rendering of Motion

A huge variety of animation techniques, ranging from keyframing to dynamic simulation to motion capture, populates the animation toolbox and can be used to create dynamic and compelling worlds full of action and life. During the rendering process, production-quality renderers employ motion blur as a form of temporal anti-aliasing. While motion blur effectively removes aliasing artifacts such as strobing, it does so at the cost of image clarity. Fast-moving objects may be blurred beyond recognition in order to properly remove high-frequency signal content. Although correct from a signal-processing point of view, this blurring may conflict with the animator's concept of how motion should be treated based on his or her creative vision for the scene. Since the appreciation of motion is a perceptual issue, the animator may wish to stylize its depiction in order to stimulate the brain in a certain manner, in analogy to the way an impressionist painter may stylize a painting in order to elicit some aesthetic response. To this end, comic book artists, whose entire medium of expression is based on summarizing action in still drawings, have pioneered a variety of techniques for depicting motion. Similar methods are employed in 2D hand-drawn animation to emphasize, accent, and exaggerate the motion of fast-moving objects, including speed lines, multiple stroboscopic copies, streaking, stretching, and stylized blurring. Figure 5.1 shows some examples of motion effects in comics and cartoons. Although these effects have played an important role in traditional illustration and animation for the past century, computer-graphics animation cannot accommodate them in a general and flexible manner

because production renderers are hard-wired to deliver realistic motion blur. As such, stylized motion effects are relegated to one-off treatment and infrequently used.

Figure 5.1: Motion effects were pioneered by comic and cartoon artists. For example, the upper two images from Marvel's X-Men use speed lines to illustrate the dynamic action in the scenes. The images in the lower row are frames from the cartoons "Donald Duck - Golden Eggs" and "Aladdin." Although the motion effects may not be clearly visible in the fraction of a second that they are displayed when played back as a movie, they contribute to the perception of the motion of objects. All images are © Disney Enterprises, Inc.

In this chapter, we experiment with motion effects as first-class entities within the rendering process. Rather than attempting to reproduce any particular style, we aim at creating a general-purpose rendering mechanism that can accommodate a variety of visualization styles, analogous to the way surface shaders can implement different surface appearances. Many effects from traditional mediums are directly related to an object's movement through a region of space and take the form of transient, ephemeral visuals left behind. This observation motivates a simple yet powerful change in the rendering process. We extend the concept of a surface shader, which is evaluated on an infinitesimal portion

of an object's surface at one instant in time, to that of a programmable motion effect, which is evaluated with global knowledge about all portions of an object's surface that pass in front of a pixel during an arbitrary long sequence of time. Figure 5.2 illustrates the process. With the added motion information, our programmable motion effects can decide to color pixels long after (or long before) an object has passed in front of them, enabling speed lines, stroboscopic copies, streaking, and stylized blurring. By rendering different portions of an object at different times, our effects also encompass stretching and bending. Other effects that extend beyond the object's position in space, such as clouds of dust or smoke, flashes of color that fill the frame, and textual annotations (e.g., "BANG" or "POW") are not addressed by our framework. Traditional motion blur is a special case within our system, implemented as a motion effect program that averages the relevant surface contributions during a specified shutter time. In general, however, our method dissolves the classic notion of a scene-wide shutter time and allows each motion effect program to independently specify its operating time range, so that a single rendered frame may compose information from different periods of time.

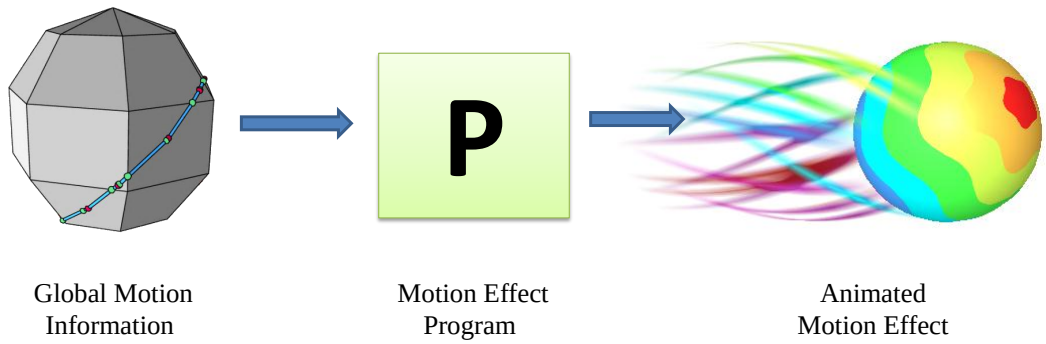


Figure 5.2: Our system computes precise information about which part of an object has passed under a given pixel within a certain time period. This motion data can be processed by a user-supplied motion effect program into a pixel color that is representative of the desired depiction of motion.

From a technical standpoint, the most challenging aspect of our system is efficiently computing global information about an object's movement. We make this computation in the context of a ray tracer, where a single ray cast is modified to find every part of an object that has moved past a given pixel throughout an arbitrary long time range. Since the motion of an object may be complex, an analytical solution to the problem is not feasible. Instead, we construct a new geometric object called a *time aggregate object* (TAO) that aggregates an object's movement into a single geometric representation. This representation

is augmented with additional information that enables the reconstruction of a set of points representing the path along the surface of the object that is visible to the pixel as the object moves through the scene, along with the associated times. Using this global information, one can develop shading algorithms that utilize information about an object's movement through an arbitrarily large window of time.

Our primary contribution is an approach to motion depiction for three-dimensional computer animation that fits naturally into current rendering paradigms and offers the same generality and flexibility as programmable surface shading. We also make the technical contribution of the TAO data structure and present several examples of programmable motion effects.

5.1 Background

James Cutting [Cutting, 2002] presents a detailed treatise about motion depiction in static images that examines parallels in art, science, and popular culture. He identifies five categories that encompass the vast majority of motion depiction techniques used to-date: photographic blur, speed lines, multiple stroboscopic images, shearing, and dynamic balance.

Of these categories, photographic blur has received by far the most attention within computer graphics, as an answer to temporal aliasing problems in computer animation. Since a rendered animation represents a sampled view of continuous movement, disturbing aliasing artifacts such as strobing can occur if the temporal signal is sampled naively [Potmesil and Chakravarty, 1983]. Consequentially, temporal anti-aliasing is a core component of all production-quality renderers, and the research community has developed many sophisticated algorithms for this purpose. These methods can be seen as convolution with a low-pass filter to remove high-frequency details, yielding a blurry image. Or, in analogy to traditional photographic processes, they can be interpreted as opening a virtual shutter for a finite period of time during which fast-moving objects distribute their luminance over regions of the film, creating blurred motion.

Sung, Pearce, and Wang [Sung et al., 2002] present a taxonomy of motion blur approaches, with associated references, and reformulated these published methods in a consistent mathematical framework. Two features of this framework help distinguish existing motion-blur algorithms from our presented work. First, existing methods can be interpreted as scene-wide integration over a fixed shutter time. In our work, we abandon the notion of a fixed shutter time and empower individual motion effect programs to determine the time range over which to operate. In this way, a single rendered image may combine information from a variety of different time ranges. This added flexibility incurs a more

complicated compositing situation (Section 5.3.6) when multiple motion effects overlap in depth. Second, existing methods employ a spatio-temporal reconstruction filter responsible for averaging contributions from different times. In our method, we generalize this filtering concept to an arbitrary program that allows a variety of different looks to be expressed. The averaging operation used in traditional motion blur becomes a special case within this framework.

The value of stylized motion depiction is evident from its treatment in traditional artistic mediums, including the "Futurism" art movement of the 1900's [Hulten, 1986] and techniques taught as tools-of-the-trade to comic book artists [McCloud, 1993] and animators [Goldberg, 2008, Whitaker and Halas, 2002]. Perceptual studies even provide direct evidence that speed lines influence low-level motion processing in the human visual system [Burr and Ross, 2002]. Due to the importance of stylized motion depiction, many researchers have explored ways to incorporate it into computer-generated imagery, animation, photographs, and video.

Masuch and colleagues [Masuch et al., 1999] describe the use of speed lines, stroboscopic copies, and arrows in 3D graphics as a post-processing operation, Lake and colleagues [Lake et al., 2000] present a similar method for speed line generation, and Haller and colleagues [Haller et al., 2004] present a system for generating these effects in computer games. These methods are specialized for the targeted effects and may not generalize easily to different visual styles. A framework for generating visual cues based on object motion is introduced by Nienhaus and Döllner [Nienhaus and Döllner, 2005]. This method allows one to define rules for the depiction of certain events or sequences based on a scene graph representation of geometry and a behavior graph representation of animation. The authors do not address the issue of how to implement the depictions in a generalized fashion. Researchers also present algorithms to add stylized motion cues to 2D animation [Hsu and Lee, 1994, Kawagishi et al., 2003] and video [Bennett and McMillan, 2007, Collomosse et al., 2005], to filter motion in images [Liu et al., 2005] or 3D animation [Wang et al., 2006, Noble and Tang, 2007, Chenney et al., 2002] in order to create a magnified or cartoony effect, to create stylized storyboards from video [Goldman et al., 2006], and to summarize the action in motion-capture data [Assa et al., 2005, Bouvier-Zappa et al., 2007] or sequences of photographs [Agarwala et al., 2004].

Taken all together, these methods cover the five categories of motion depiction techniques that Cutting [Cutting, 2002] proposes. However, existing algorithms target specific looks and must explicitly parameterize stylistic deviations (e.g., Haller and colleagues' system [Haller et al., 2004] represents speed lines as connected linear segments with a parameter to control line thickness over time). The central thrust and distinguishing characteristic of our contribution is an open-ended system for authoring motion effects as part of the

rendering process. We extend the concept of programmable surface shading [Hanrahan and Lawson, 1990] to take the temporal domain into account for pixel coloring. While our motion effect programs define how this extra dimension should be treated for a certain effect, they can still call upon conventional surface shaders for the computation of surface luminance at a given instant in time. We show examples in four of the five categories proposed by Cutting (stylized blurring, speed lines, multiple stroboscopic images, and shearing). And, notably, within these categories, different styles can be achieved with the same flexibility afforded by programmable surface shaders.

5.2 Method Principles

Many stylized motion effects from traditional mediums summarize an object's movement over a continuous range of time with transient, ephemeral visuals that are left behind. Motivated by this observation, we propose an alternative rendering strategy that operates on the scene configuration during an arbitrarily long time range T. In this section, we introduce the concept of motion effect programs, our *time aggregate object* data structure, and the renderer's compositing system. Section 5.3 discusses more specific implementation details.

5.2.1 Motion Effect Programs

In analogy to a state-of-the-art renderer that relies on surface shaders to determine the color contributions of visible objects to each pixel, we delegate the computation of a moving object's color contribution within the time range T to *motion effect programs*. A motion effect program needs to know which portions of all surfaces have been "seen" through a pixel during T. In general, this area is the intersection of the pyramid extending from the eye location through the pixel corners with the objects in the scene over time. Although Catmull [Catmull, 1978] presents an analytic solution to this pixel coverage problem for static scenes, extending it to the spatio-temporal domain is non-trivial. As such, we follow the approach of Korein and Badler [Korein and Badler, 1983] and collapse the pyramid to a single line through the pixel center before computing analytic coverage. The surface area seen by the pixel for a particular object then becomes a line along the surface, which we call a *trace*.

A motion effect program calculates a trace's contribution to the final pixel color. In doing so, it utilizes both positional information (the location of a trace on the object's surface) and temporal information (the time a given position was seen) associated with the trace. It can evaluate the object's surface shaders as needed and draw upon additional scene information, such as the object's mesh

data (vertex positions, normals, texture coordinates, etc.), auxiliary textures, the camera view vector, and vertex velocity vectors.

5.2.2 Time Aggregate Objects

Computing a trace is a four-dimensional problem in space and time, where intersecting the 4D representation of a moving object with the plane or surface defined by the view ray yields the exact trace. Unfortunately, since the influence of the underlying animation mechanics on an object's geometry can be arbitrarily complex, a closed-form analytic solution is infeasible. Monte Carlo sampling [Cook et al., 1984] could be considered as an alternative, since it is used by prominent production renderers [Sung et al., 2002] to produce high-quality motion blur. However, it is also not effective in the present scenario since the time period associated with a trace may be very short in comparison to the time range over which the motion effect is active. For example, a ball may shoot past a pixel in a fraction of a second but leave a trailing effect that persists for several seconds. A huge number of samples distributed in time would be required in order to effectively sample the short moment during which the ball passes.

Consequentially, we propose a new geometric data structure that allows our system to reconstruct a linear approximation of the full trace from a single ray cast. Our data structure is inspired by the 4D polyhedra used in Grant's temporal anti-aliasing method [Grant, 1985] and aggregates an object's geometry sampled at a set of times t_i (Figure 5.3) into a single geometric primitive. In addition, corresponding edges of adjacent samples are connected by a bilinear patch, which is the surface ruled by the edge as its vertices are interpolated linearly between t_i and t_{i+1}. We call the union of the sampled object geometry and swept edges a *time aggregate object* (TAO).

The intersection of a view ray with a bilinear patch of the TAO represents a time and location where the ray, and thus also a trace, has crossed an edge of the mesh. By computing all such intersections (not just the closest one) and connecting the associated edge crossings with line segments, we obtain a linear approximation of the trace. Intersections with the sampled geometry represent additional time and space coordinates, which can be used to improve the accuracy of the trace approximation. The accuracy of the approximated traces thus depends on the number of TAO intersections, which scales with both the geometric complexity of the object's geometry and the number of samples used for aggregation. In practice, we can use finely sampled TAOs without a prohibitive computation time, since the complexity of ray-intersection tests scales sub-linearly with the number of primitives when appropriate spatial acceleration structures are used [Fóris et al., 1996].

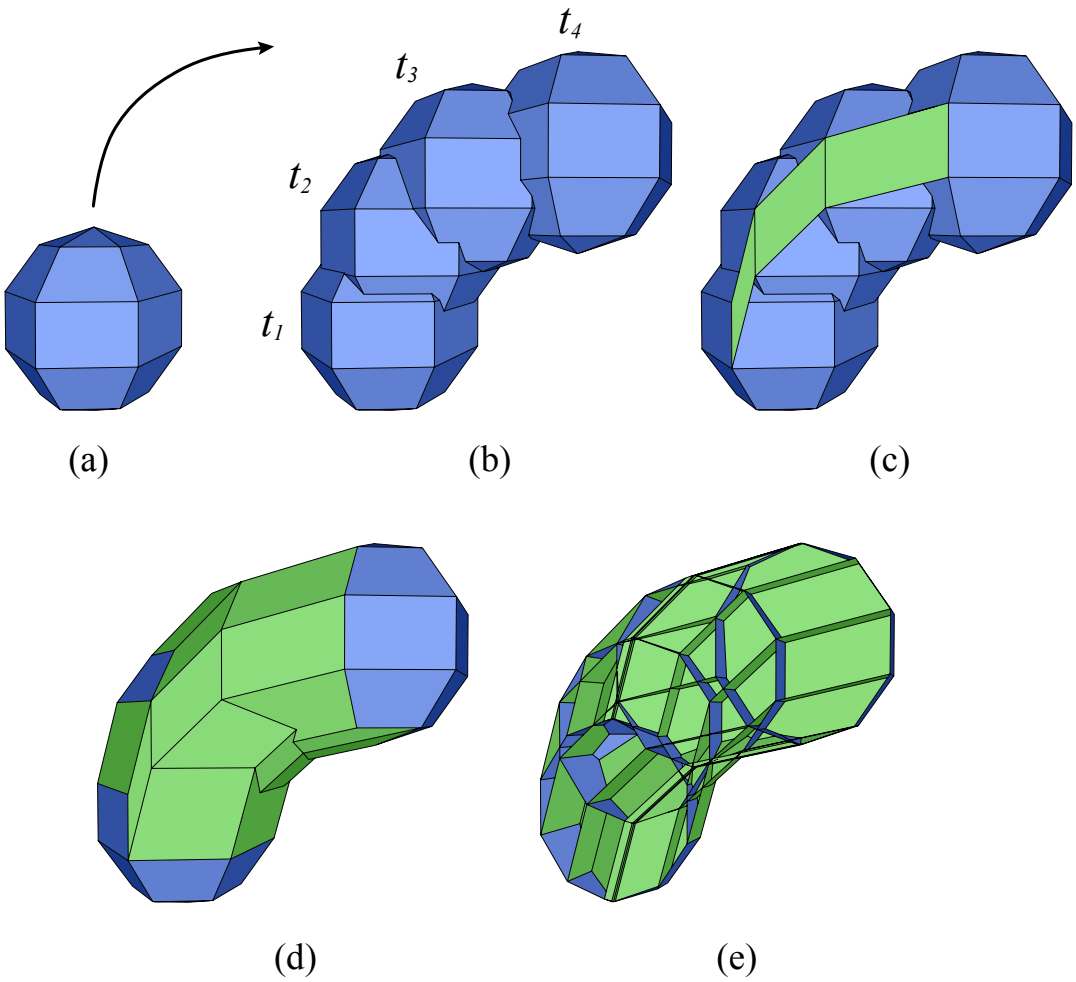


Figure 5.3: The *time aggregate object* (TAO) data structure encodes the motion of an object (a) using copies of the object sampled at different times $t_1 \ldots t_4$ (b) and bilinear patches that connect corresponding edges in adjacent samples edges. These patches are shown in (c) for one edge and in (d) for the whole mesh. This data structure is not just the convex hull of the moving object, but has a complex inner structure, as seen in the cutaway image (e).

5.2.3 Compositing

A motion effect program acts on a trace as a whole, which may span a range of depths and times. If objects and contributions from their associated programmable motion effects are well separated in depth, our renderer can composite each effect's total contribution to the image independently according to depth ordering. The compositing algorithm used can be defined by the effect itself, drawing upon standard techniques described by Porter and Duff [Porter and Duff, 1984]. Compositing becomes more complex when multiple traces overlap, since there may be no unequivocal ordering in depth or time. Additionally, different motion effect programs may operate over different, but overlapping, time domains since a single scene-wide shutter time is not enforced.

We resolve this ambiguity by introducing additional structure to the way in which motion effects operate. All traces are resampled at a fixed scene-wide resolution. Each motion effect program processes its traces' samples individually, and outputs a color and coverage value if that sample should contribute luminance to the rendered pixel. Our compositing system processes the output samples in front-to-back order, accumulating pixel color according to the coverage values until 100% coverage is reached.

5.3 Implementation

We have implemented our method as a plug-in to Autodesk Maya. Our system is divided into two parts: a module to create and encapsulate TAOs from animated 3D objects, and a new rendering engine, which generates images with motion effects. The rendering engine computes a color for each pixel independently by computing all intersections of the pixel's view ray with the TAOs, and connecting them to form a set of traces. Then, it calls the motion effect programs for each object, which compute that object's and effect's contribution to the pixel color using the object's traces. Finally, a compositing step computes the resulting pixel color.

5.3.1 TAO Creation

We implement the TAO concept as a custom data structure within Maya. We assume that animated objects are represented by triangle meshes with static topology. Our system builds a TAO by sampling an object's per-vertex time dependent data (e.g., vertex positions, normals, and texture coordinates) at a set of times t_i, aggregating those samples into a single geometric primitive, and connecting adjacent edges with bilinear patches (Figure 5.3). For a mesh

with N_E edges and N_F triangles, N_t object samples yield a TAO with $N_t \cdot N_F$ triangles and $(N_t - 1) \cdot N_E$ bilinear patches.

Figure 5.4: Undersampled (left) versus properly sampled (right) motion. The trace sampling rate is 10 times larger in the right image.

The density and placement of the object sample times t_i determine how well the motion of an object is approximated by our TAO data structure. Since the approximation is linear by nature, it perfectly captures linear motion. For rotation and non-rigid deformation, however, proper sampling of the motion is necessary (Figure 5.4). Our systems supports both a uniform sampling and an adaptive sampling strategy that starts with a uniform temporal sampling and repeatedly inserts or deletes sample positions based on the maximum non-linearity α_i between the vertex positions of adjacent samples:

$$\alpha_i = \max_j angle(\mathbf{v}_j(t_{i-1}), \mathbf{v}_j(t_i), \mathbf{v}_j(t_{i+1})), \tag{5.1}$$

where $angle(A, B, C)$ is the angle between the line segments AB and BC and $\mathbf{v}_j(t_i)$ is the position of vertex j in object sample i. First, the adaptive sampling strategy iteratively removes sample times whenever α_i is smaller than a given coarsening threshold. In a second step, it inserts two new samples at $\frac{t_i - t_{i-1}}{2}$ and $\frac{t_{i+1} - t_i}{2}$ if α_i exceeds a refinement threshold. In order to keep the maximum number of object samples under control, the refinement criterion is applied iteratively on the sample with the largest α_i until the maximum number of samples is reached. In practice, we assume a certain amount of coherence in the motion of nearby vertices, and our system approximates the optimal sample placement by considering only a subset of the mesh vertices in Equation 5.1.

5.3.2 TAO Intersection

Our renderer generates a view ray for each pixel according to the camera transform. Each ray is intersected with the primitives of the TAO (mesh faces and bilinear patches) to compute all intersections along the ray, not just the one which is closest to the camera. Normals, texture coordinates, and other surface properties are interpolated linearly over the primitives to the intersection point and stored.

For the intersection with bilinear patches, we use the algorithm described by Ramsey and colleagues [Ramsey et al., 2004]. If an edge connects $\mathbf{v}_j$ and $\mathbf{v}_k$, then for each $0 < i < N_i - 1$ a bilinear patch is defined as

$$\begin{aligned}\mathbf{p}_{ijk}(r,s) = (1-r)(1-s)\cdot\mathbf{v}_j(t_i) + (1-r)\cdot s\cdot\mathbf{v}_k(t_i)\\ + r\cdot(1-s)\cdot\mathbf{v}_j(t_{i+1}) + r\cdot s\cdot\mathbf{v}_k(t_{i+1}).\end{aligned} \tag{5.2}$$

Solving for a ray intersection yields a set of patch parameters (r, s) that corresponds to the intersection point. With parameter r, we can compute the time at which the ray has crossed the edge:

$$t = t_i + r\cdot(t_{i+1} - t_i). \tag{5.3}$$

The parameter s represents the position along the edge at which the crossing has happened. It can be used to interpolate surface properties stored at the vertices to the intersection point, and to compute the position of the intersection point in a reference configuration of the object. Mesh face primitives in the TAO are intersected according to standard intersection algorithms. A parametric representation of the intersection points with respect to the primitives is necessary for interpolating the surface properties.

To accelerate the intersection computation, we partition the image plane into tiles and create a list of all primitives for which the bounding box intersects a given tile. Each view ray only needs to be intersected with the primitive list of the tile containing the ray.

5.3.3 Trace Generation

The set of all intersection points of a ray with the TAO can be used to reconstruct the locus of points traced by the ray on the moving object. Consider the interpretation of the ray-TAO intersection points on the input mesh. As an individual triangle passes fully by the ray, the ray crosses the edges of the triangle an even number of times, assuming that the ray was not intersecting at the beginning or at the end of the observed time. Each edge crossing corresponds to an intersection of the ray with a bilinear patch of the TAO, and vice versa. Since our system is restricted to a piecewise linear approximation of

motion, we assume that the trace forms a linear segment in between two edge crossings. The task of constructing the trace is thus equivalent to connecting the intersection points in the proper order.

To facilitate this discussion, we refer to the volume covered by a triangle swept between two object samples within the TAO as a Time Volume Element (TVE). This volume is delimited by two corresponding triangles that belong to adjacent object samples and the three bilinear patches formed by the triangle's swept edges (Figure 5.5). Since there is an unambiguous notion of inside and outside, a ray originating outside of a TVE will always intersect the TVE an even number of times, unless it exactly hits the TVE's boundary. A pair of consecutive entry and exit points corresponds to a segment of the trace on the triangle. Therefore, by sorting all ray-TVE intersection points according to their distance from the viewer and pairing them sequentially, we can construct all trace segments that cross the corresponding triangle within the time range spanned by that individual TVE.

Next, we consider trace connectivity. TVEs that share a common TAO primitive (triangle or bilinear patch) are adjacent in space. If the shared primitive is a triangle, the TVEs were formed by the same triangle of the input mesh in adjacent object samples. If it is a bilinear patch, the triangles that formed the TVEs are adjacent to one another on the input mesh, separated by the edge that formed the patch. This combined adjacency information determines the connectivity of trace segments associated with neighboring TVEs. Our system processes TVEs in turn to reconstruct the individual trace segments, and then uses this connectivity information to connect them together into different connected components.

5.3.4 Trace Resampling

At this point, we have the traces that the current pixel's view ray leaves on moving objects in the form of connected sequences of intersections with the TAO. Each intersection point corresponds to an animation time and a position on an object, and, in conjunction with the trace segments that connect to it, we can determine and interpolate surface properties such as attached shaders, normals, and texture coordinates.

In order to facilitate compositing, our system imposes a consistent temporal sampling on all traces by dividing them into individual trace fragments of fixed spacing in time. This resampling should be dense enough so that depth conflicts among different traces can be resolved adequately. The visibility of individual trace fragments during each sampling interval is also determined at this stage. Our system does not delete occluded or back facing trace fragments, however.

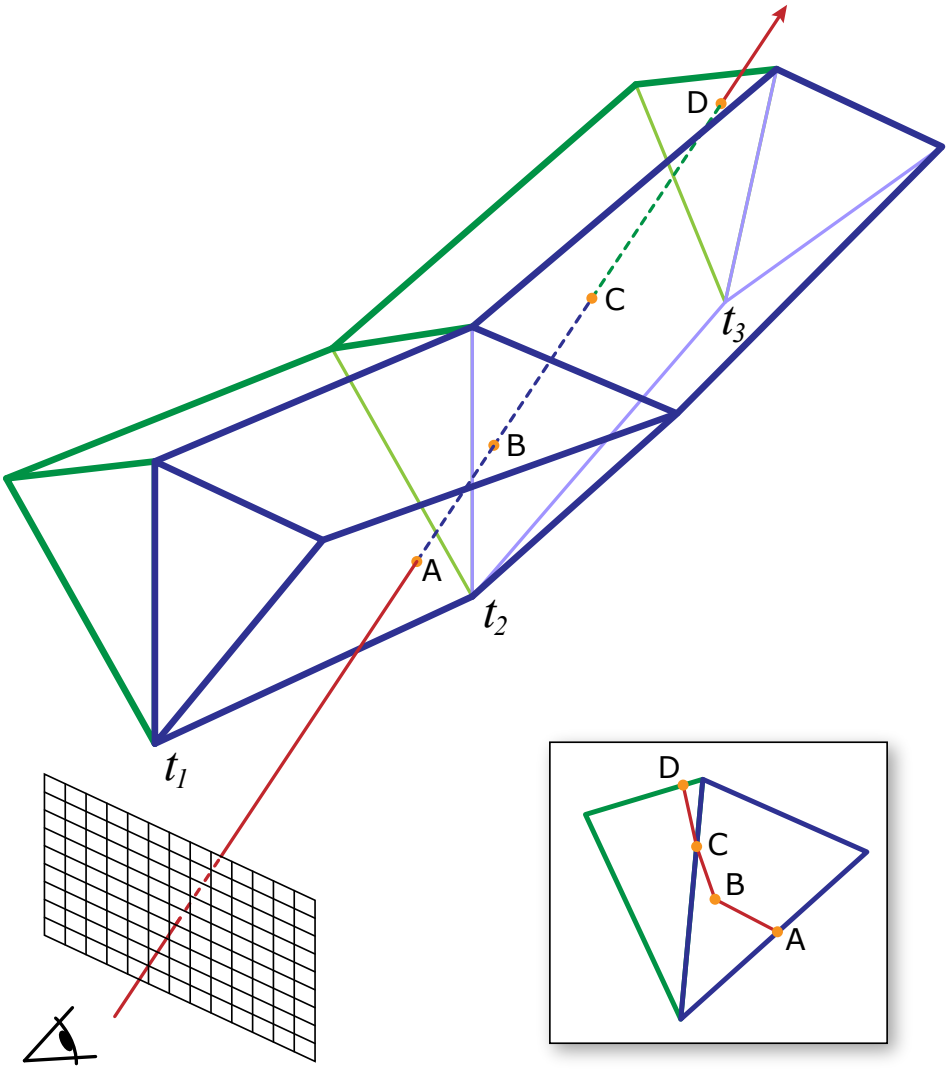


Figure 5.5: This figure shows two triangles at 3 consecutive object samples, t_1, t_2, and t_3, that results in four Time Volume Elements (TVEs). A ray intersects the TVEs four times. At intersection A, the ray enters the blue triangle though an edge. Point B is an intersection with a sampled mesh triangle, indicating that the ray moves from one time sample to the next. In C, the ray leaves the blue and enters the green triangle, which it finally exits at D. The reconstructed trace visualized on the two triangles is shown in the inset figure.

Instead, it is left up to the motion effect program to decide whether obstructed segments should contribute to pixel coverage.

5.3.5 Motion Effect Programming

A motion effect program operates on the resampled representation of a trace. It is called once per object and processes all trace fragments associated with that object. The motion effect program has two options when processing an individual trace fragment. It can simply discard the fragment, in which case no contribution to the final pixel color will be made. Or, it can assign a color, depth, and pixel coverage value and output the fragment for compositing. The coverage value determines the amount of influence a fragment has on the final pixel color in the compositing step. When making this decision, the motion effect program can query an object's surface shader, evaluate auxiliary textures (e.g., noise textures, painted texture maps, etc.), or use interpolated object information.

The effect program can also subdivide the trace even further if the effect requires a denser sampling of the object's surface. For example, further subdivision could be the needed if the effect must integrate a surface texture of high frequency. Decreasing the trace resampling distance (Section 5.3.4) has a similar effect, although it affects all motion effects in the scene.

5.3.6 Compositing

By emitting trace fragments, each motion effect program specifies its desired contribution to the pixel color. The compositing engine combines all of the emitted fragments to determine the final pixel color by processing fragments in depth order (front to back) using a clamped additive model based on the coverage values. The coverage values of the fragments, which range between 0 and 1, are accumulated until a full coverage of 1 is reached or until all fragments have been processed. If the coverage value exceeds 1, the last processed fragment's coverage is adjusted so that the limit is reached exactly. The final pixel color is computed by summing the processed fragment colors weighted by the corresponding coverage values. The accumulated coverage value is used as the alpha value for the final pixel color.

5.4 Results

In this section, we show a number of results obtained with our programmable motion effect renderer and describe how the effect programs were set up to achieve these results.

5.4.1 Motion Effects

To better explain the mechanism of programmable motion effects, we supply pseudo-code algorithms for five basic effects used in our examples. These algorithms show the framework of each effect using a common notation. A trace segment *ts* consists of two end points, denoted *ts.left* and *ts.right*. Each of the end points carries information about the animation time at which this surface location has been intersected by the view ray and linearly interpolated surface properties. They may be interpolated further with the INTERPOLATE function, which takes a trace segment and a time at which it should be interpolated as parameters. By *ts.center*, we designate the interpolation of the end points' properties to the center of the segment. The SHADE function evaluates the object's surface shader with the given surface parameters and returns its color. The animation time of the current frame being rendered is denoted as "current time." The output of a motion effect program is a number of fragments, each with a color and coverage value, that represent the effect's contribution to the pixel color. Passing a fragment to the compositing engine is designated by the keyword **emit**.

Instant Render The most basic program renders an object at a single instant in time. The program loops through all trace segments and checks whether the desired time lies within a segment. If so, it interpolates the surface parameters to that point, evaluates the surface shader accordingly, and emits just one fragment to the compositing stage with a coverage value of 1. Any surfaces behind that fragment will be hidden, as is expected from an opaque surface.

Algorithm 3 INSTANT RENDER(*time*)

```
for all trace segments ts do
  if ts.left.time <= time < ts.right.time then
    eval_at := INTERPOLATE(ts, time)
    color := SHADE(eval_at)
    coverage := 1
    emit fragment(color, coverage)
  end if
end for
```

Weighted Motion Blur Photorealistic motion blur integrates the surface luminance within a given shutter time. We extend this process with an arbitrary weighting function $w(t)$ that can be used both for realistic and stylized blurring. For photorealistic blur, the weighting function is the temporal sampling reconstruction filter function. A flat curve corresponds to the commonly used box filter. In general, the weighting function need not be normalized, which means that luminance energy is not necessarily preserved. This flexibility increases the possibilities for artistic stylization. For example, a flat curve with a spike at $t = 0$ results in motion blur that has a clearly defined image of the object at the current frame time.

Algorithm 4 MOTION BLUR($w(t)$)

```
for all trace segments ts do
  if ts time interval overlaps with {t | w(t) > 0} then
    clip ts against {t | w(t) > 0}
    color := SHADE(ts.center)
    δt := ts.right.time - ts.left.time
    coverage := δt · w(ts.center.time)
    emit fragment(color, coverage)
  end if
end for
```

The effect program in Algorithm 4 describes weighted motion blur in its most basic form. A more advanced implementation could take more than one surface shading sample within a segment and use a higher order method for integrating the luminance. With the given effect program, however, the fidelity of the image can be improved by increasing the global trace sampling rate (Section 5.3.4). A comparison of motion blur from our implementation with that from production renderers is shown in Figure 5.6.

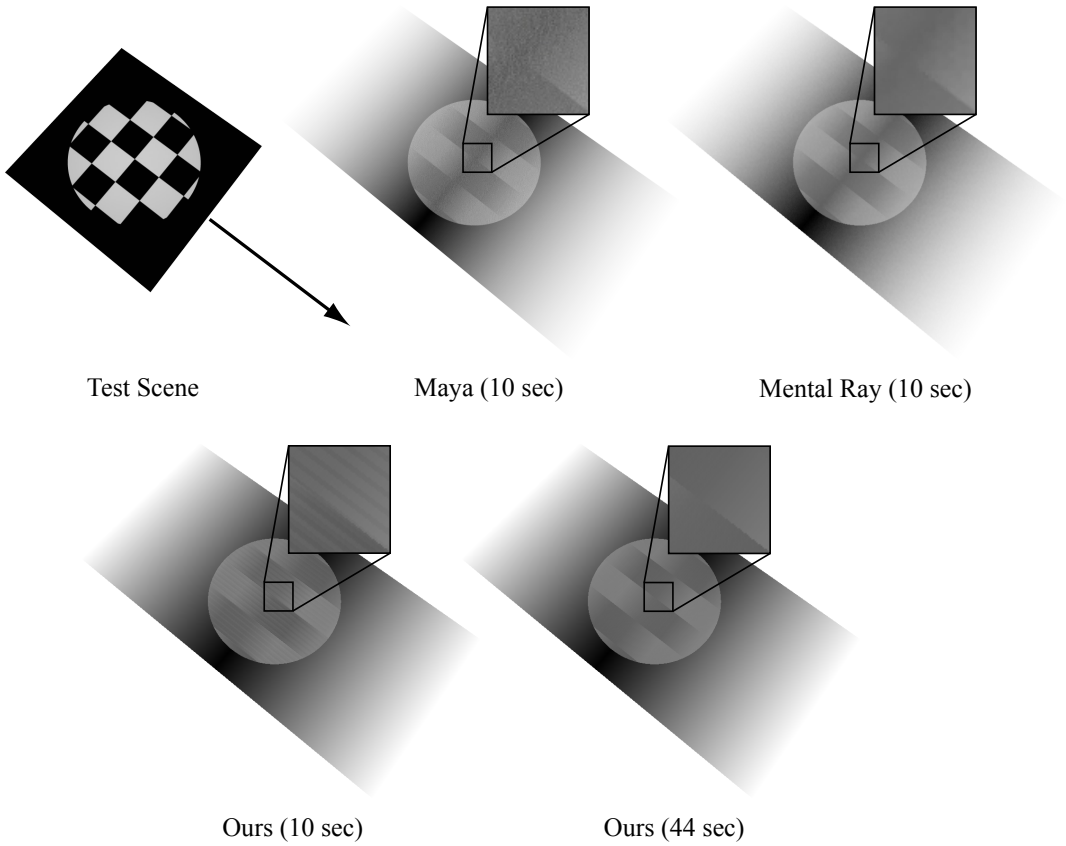


Figure 5.6: A comparison of our motion blur implementation with production renderers. The test scene, rendered without blur, consists of a translating checkerboard square with a stationary spotlight. The three middle images were created with equal render time. The rightmost image shows a high-quality render obtained with our system with a denser trace sampling.

Speed Lines Speed lines are produced by seed points on an object that leave streaks in the space through which they travel. They can be distributed automatically or placed manually. The program computes the shortest distance between trace segments and seed points. If the distance is smaller than a threshold, the seed point has passed under or close to the pixel, and the pixel should be shaded accordingly. A falloff function based on distance or time can be used to give the speed line a soft edge.

Algorithm 5 SPEED LINES(*seed vertices*, *width*, *length*)

```
for all trace segments ts do
  for all seed vertices v do
    if DISTANCE(ts, v) < width
    and current time − ts.center.time < length then
      color := SHADE(ts.center)
      compute coverage using a falloff function
      emit fragment(color, coverage)
    end if
  end for
end for
```

Time Shifting The time shifting program modulates the instantaneous time selected from the trace. We use it in conjunction with the INSTANT RENDER so that each pixel in the image may represent a different moment in time. If fast-moving parts of an object are shifted back in time proportional to the magnitude of their motion, these parts appear to be lagging behind. They "catch up" with the rest of the object when it comes to a stop or changes direction. This effect is only visible if an object's motion is not uniform across the surface, as in the case of rotation. In this algorithm, v_{motion} designates the motion vector of the corresponding part of the surface.

Algorithm 6 TIME SHIFT(*ts*, *shift magnitude*)

```
time shift := |ts.center.v_motion| · shift magnitude
shift time values in trace segment ts according to time shift
return ts
```

Stroboscopic Images This effect places multiple instant renders of the object at previous locations. It imitates the appearance of a moving object photographed with stroboscopic light. The intensity of the stroboscopic images is attenuated over time to make them appear as if they are washing away. We keep the locations of the stroboscopic images fixed throughout the animation, but they could also be made to move along with the object. The falloff function

can be composed of factors considering the time of the stroboscopic image, geometric properties of the mesh (e.g., the angle between the motion vector and the normal at a given point), or an auxiliary modulation texture to shape the appearance of the stroboscopic images.

Algorithm 7 STROBOSCOPIC IMAGES(*spacing*, *length*)

```
for all trace segments ts do
  if ts.left.time < current time − length then
    t1_mod := ts.left.time modulo spacing
    t2_mod := ts.right.time modulo spacing
    if t1_mod <= 0 < t2_mod then
      color := SHADE(ts.center)
      compute coverage using a falloff function
      emit fragment(color, coverage)
    end if
  end if
end for
```

5.4.2 Examples

Translating and Spinning Ball The results in Figure 5.7 demonstrate speed lines in different visual styles. The first two images use the SPEED LINES effect (Algorithm 5) with a falloff function that fades in linearly at both ends of the speed lines and modulates the width of the speed line over time. For the last image, the distance of the seed point to the current pixel is used to manipulate the normal passed to the surface shader, giving a tubular appearance. In all images, the INSTANT RENDER effect was used to render the solid appearance of the ball.

Bouncing Ball The example in Figure 5.8 shows a bouncing toy ball, rendered with a modified version of the MOTION BLUR effect (Algorithm 4). The effect program computes a reference time for the input trace, and uses the difference between this reference time and the current time to determine the amount of blur, or the shutter opening time in conventional terms. As a result, the blur increases toward the end of the trail.

Spinning Rod This result shows a rod spinning about an axis near its lower end. Figure 5.9 (a) combines a MOTION BLUR effect (Algorithm 4) that uses a slightly ramped weighting function with an INSTANT RENDER effect (Algorithm 3) to render a crisp copy of the rod. Figure 5.9 (b) additionally uses

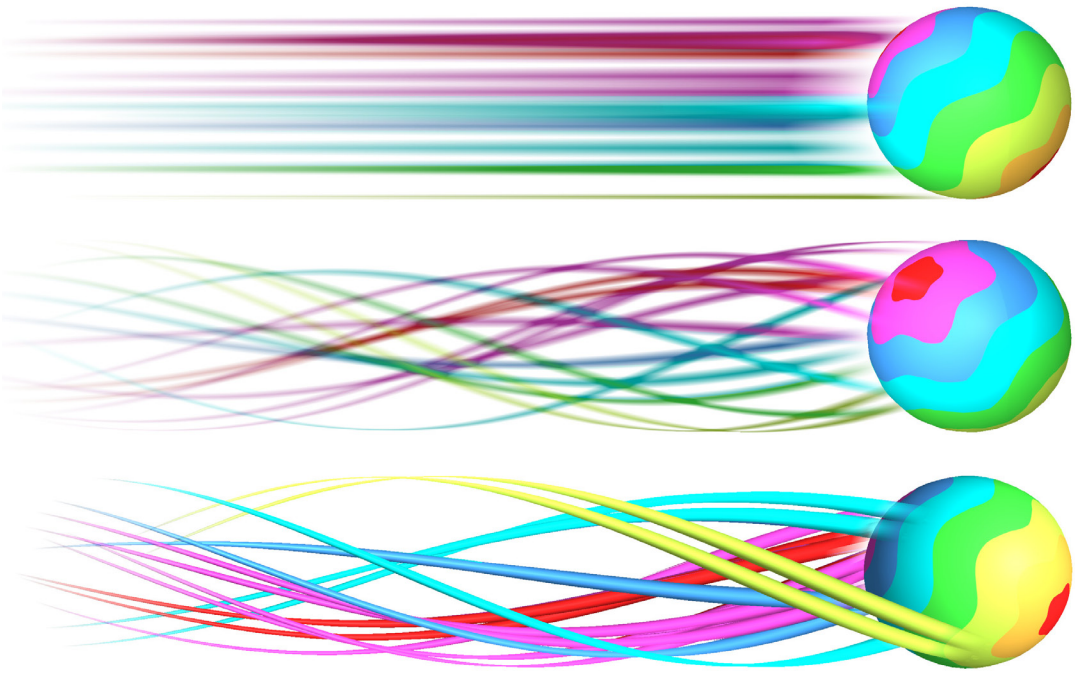


Figure 5.7: Different speed-line styles.

the TIME SHIFT function (Algorithm 6) with a negative shift magnitude so that quickly moving surface parts lag behind. As shown in the accompanying video, these parts "catch up" when the rod stops moving or changes direction. In Figure 5.9 (c), MOTION BLUR is replaced with a STROBOSCOPIC IMAGES effect (Algorithm 7). The falloff function fades the stroboscopic images out as time passes and additionally uses the angle between the motion vector and the surface normal to make the rod fade toward its trailing edge.

Toy UFO In Figure 5.10, we have attached speed lines to a UFO model to accentuate and increase the sensation of its speed. To keep the effect convincing when the camera is moving with the UFO, the length and opacity of the speedlines are animated over time. An animated noise texture is sampled using the texture coordinates from the seed point of each speed line, which gives the attenuation value for that speed line.

Pinocchio's Peckish Pest The last set of examples shows our results on a production-quality scene. Figure 5.11 uses a combination of speed lines and weighted motion blur. The motion blur's weighting curve has a spike around the current time, so that Pinocchio is shown clearly in every frame, even when

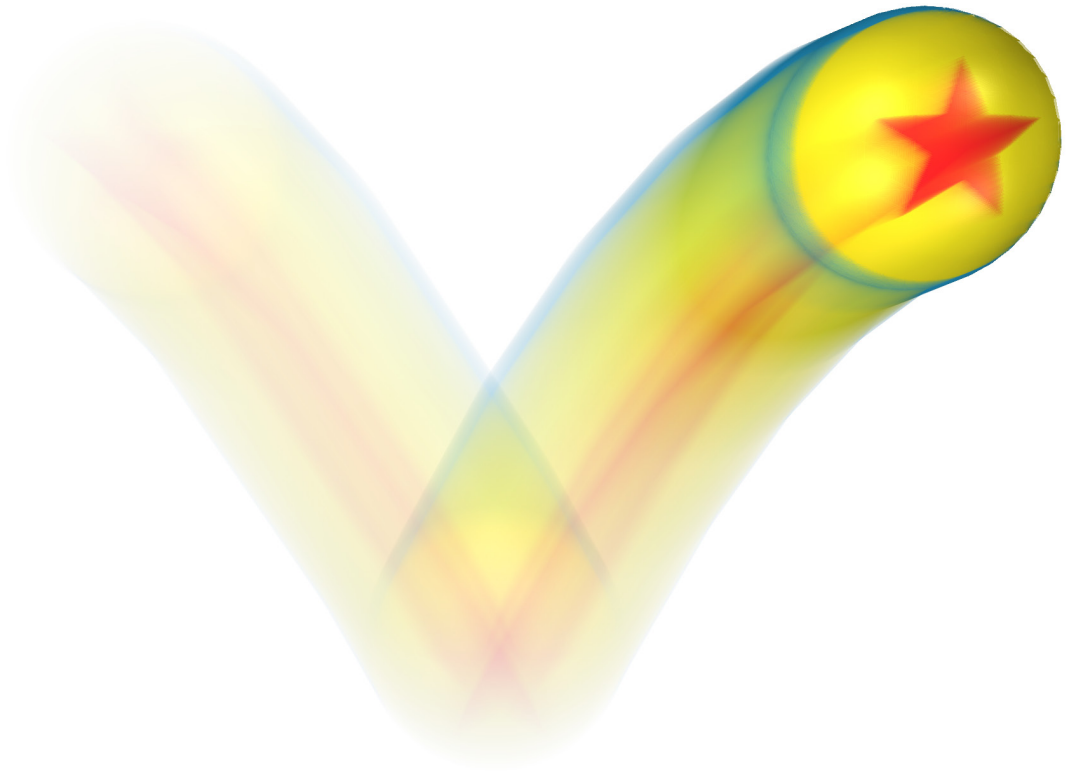


Figure 5.8: A bouncing ball rendered with non-uniform blur.

he is moving quickly. In Figure 5.12 a comparison between conventional motion blur (also rendered by our system) and our weighted motion blur is made.

Statistics about all examples are shown in Table 5.1. The values are taken from a representative frame of each animation, where the full TAO is within the visible screen area. Render time depends greatly on the motion and the amount of screen space covered by the objects. The numbers shown result from rendering an image with 1280x720 pixels on an 8-core 2.8 GHz machine.

5.5 Conclusion

In this paper, we have presented a novel approach to depict motion in computer-generated animation. Our method fits naturally into current rendering paradigms and offers the same generality and flexibility as programmable surface shading. Our results demonstrate that it is a powerful platform for experimenting with different depiction styles.

Limitations of our work motivate a number of rich future research possibilities.

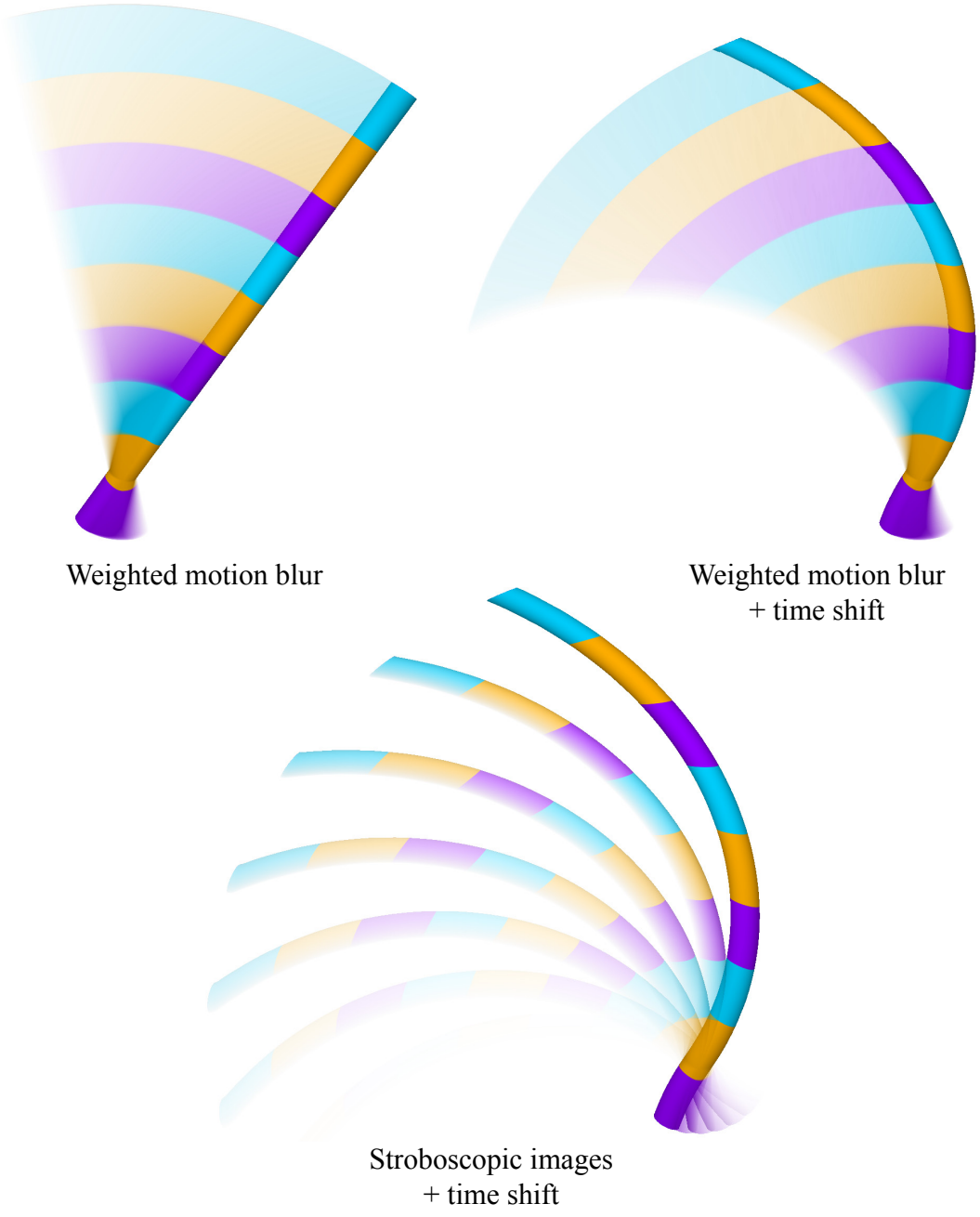


Figure 5.9: A spinning rod showing weighted motion blur (with and without time shift) and stroboscopic images with time shift.

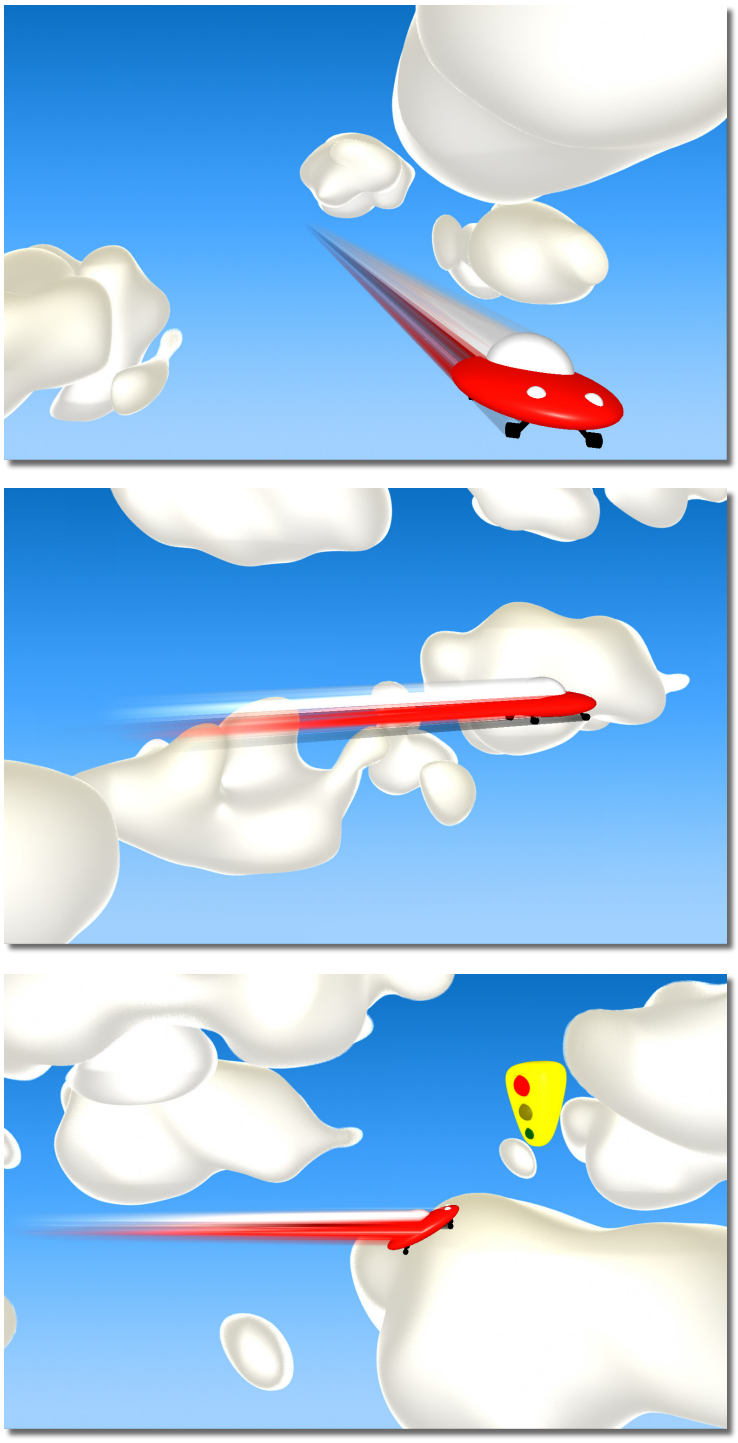


Figure 5.10: Speed lines applied to a flying UFO animation.

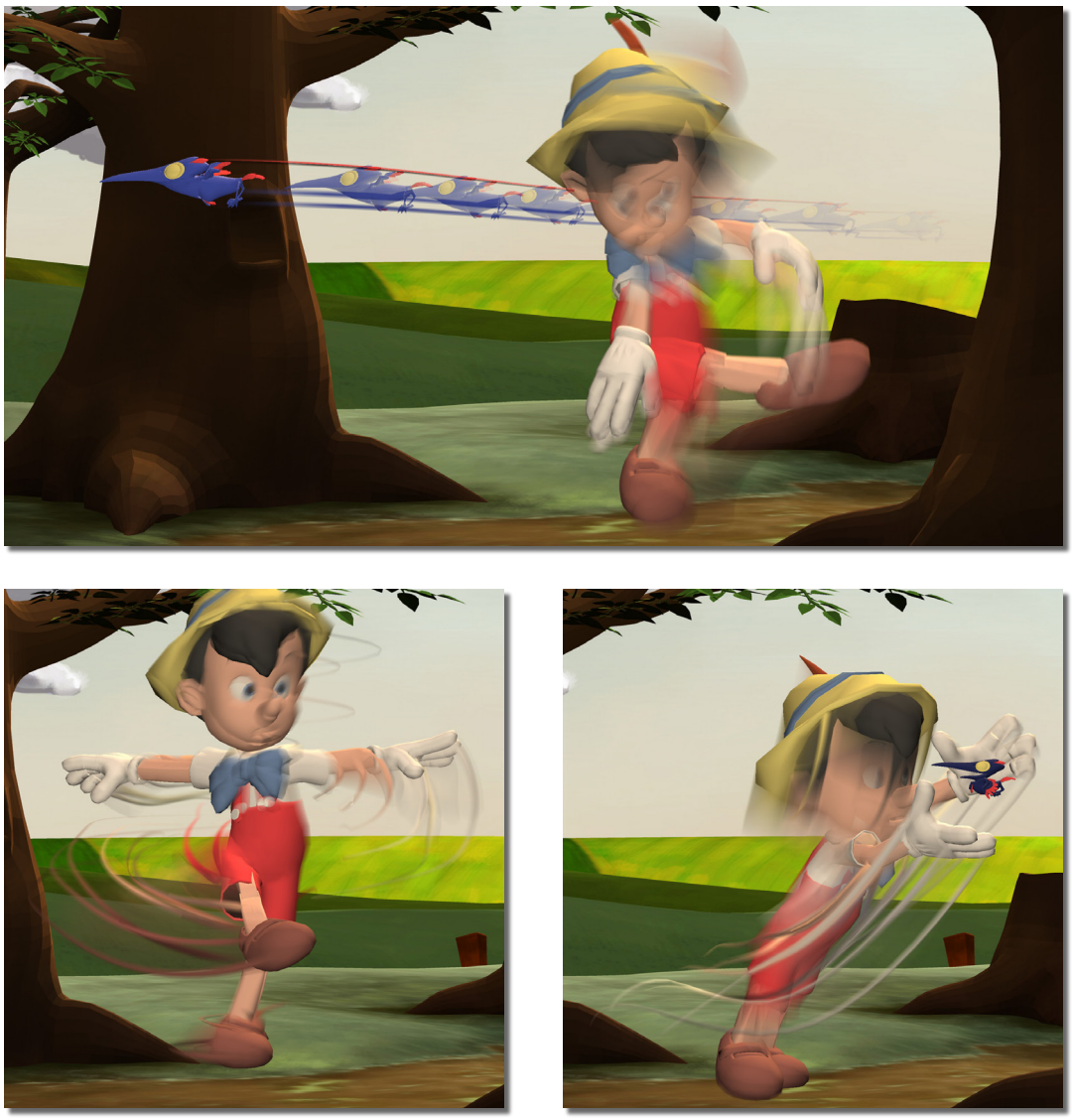


Figure 5.11: Results from applying speed line and motion blur effects (bottom), and the stroboscopic image effect (top) to an animation of Pinocchio and a woodpecker.

Figure 5.12: A comparison of conventional motion blur (left) with our weighted motion blur including speed lines (right).

In our current implementation, the screen-space region of the motion effect is limited to the convex hull of the object's motion, since the motion effect programs operate only on pixels that intersect the TAO. One future research avenue would be to consider installing the motion depiction framework further upstream in the rendering process as a geometry shader, allowing new geometry to be created and expanding the range of possible depiction styles.

We experiment with the core idea of programmable motion effects and provide a proof-of-concept that they can be used to express a variety of motion depiction styles. However, our renderer implements only the most basic functionality. Most importantly, by analyzing only one ray for each pixel, we ignore the problem of spatial aliasing. In the example renderings shown in this thesis, spatial aliasing was mitigated with simple spatial supersampling. This approach causes a large performance overhead, which could potentially be improved by exploring coherence between neighboring rays or by solving a more complex TAO intersection problem. Also, our renderer does not consider global illumination, reflection, refraction, caustics, participating media, or other effects that are standard in production renderers. One immediate avenue of future work is applying the core principle of recursive ray tracing to our framework by casting secondary rays for shadowing, reflection, and refraction. Naturally, computation time may become an issue if many secondary rays are used.

The performance of our system is reasonable for the examples shown, with most frames requiring only a few minutes to render. The slowest aspect of the system lies in the way it interfaces with Maya. Creating the TAO structure requires sampling the animation system at tens or hundreds of sample positions to cache time varying mesh data. Additionally, many of our motion effect

Example	# Triangles	TAO	Trace	Memory	Time
Spinning Ball	1200	160	0.1	12	2.6
Rod	80	100	0.1	0.5	0.25
UFO (shot 2)	3400	20	0.5	4.5	2
Pinocchio (shot 2)	37k	60	0.005	130	4.3
Pinocchio (shot 3)	122k	60	0.005	300	65

Table 5.1: Example statistics for a representative frame. The values shown in the columns are (from left to right): number of source mesh triangles, number of object samples (Section 5.3.1), trace sampling distance (Section 5.3.4), memory required to store the TAO in MiB, render time in minutes.

programs evaluate an object's surface shader or other auxiliary textures. If the parameters of these shaders and textures are themselves animated, then each and every evaluation must call back to the animation software in order to work in the proper temporal context. Due to Maya's system design and assumptions about the distinct separation of animation and rendering, such out-of-context shader network evaluations are prohibitively expensive. This observation speaks to a future animation and rendering design that does not draw a hard line between the two, but rather couples both as tightly as possible.

Our current TAO data structure requires the mesh connectivity to be constant throughout the animation. This limitation prohibits the use of our system in scenarios where the connectivity changes, such as in the presence of level of detail or certain physical simulation techniques. A related issue comes with the requirement for the motion to be sampled consistently within one TAO, even if some parts of the input object move with a different complexity than other parts. Sampling each primitive's motion at its optimal rate would improve performance and flexibility.

The implementation presented in this paper uses a ray tracing approach for rendering, but we believe that the general concept can be adapted to other rendering paradigms. It would be interesting to investigate how programmable motion effects can be implemented on GPUs and in the Reyes architecture [Cook et al., 1987]. An alternative approach would be to replace the TAO data structure with an image sequence that stores additional per pixel data such as face correspondence and surface parameters. Motion effect programs could combine a number of these images to create one frame. This approach can be seen as an extension of deferred shading for motion depiction.

CHAPTER 6

Conclusion

To conclude this thesis, we review the key contributions of our work in Section 6.1, we list the main limitations of our methods along with ideas on how to mitigate them in Section 6.2, and we finish with some broader insights that could facilitate future research in the area in Section 6.3.

6.1 Summary of Contributions

In this thesis, we have presented various advancements in the realm of artistic stylization for 3D animation. In the first part, we have developed a comprehensive system for the animation of characters and objects that uses 3D animation techniques for motion but mimics the appearance and input metaphor of 2D digital painting packages. Several scientific contributions were necessary to achieve this goal:

- We have adapted the brush stamping technique for use in stroke-based rendering. This brush model is popular in 2D digital painting due to its flexibility and simplicity, and its power is proven by a rich body of quality 2D paintings created with it. However, the traditional method is suitable only for brush strokes that are applied once and never re-rendered in different shapes. It does not take into account the changes in length and the influence of perspective that are present in 3D painting. We show how

the rendering algorithm can be adapted to support these requirements in the presence of various brush stroke effects, such as canvas texturing.

- We have presented new solutions to the problem of rendering brush strokes in the presence of conflicting paint and depth orders. This problem has been present in all past work on 3D-based brush stroke rendering, and we have shown that none of the existing solutions perform satisfactory in our application. Our first suggested solution of depth offsetting is not devoid of artifacts, but it can be implemented very efficiently on current graphics acceleration hardware and is therefore suitable for approximate preview rendering in an interactive stroke-based rendering system. Our second solution, mixed-order compositing, fulfills all conditions we have found to be necessary for artifact-free brush stroke rendering and therefore presents an excellent solution for high-quality rendering. The algorithm has a good asymptotic run-time performance of $O(N \log n)$, but due to its requirement of storing and processing all fragments for each pixel separately, it does not adapt well to current GPU architectures and thus does not deliver interactive refresh rates on practical scenes.
- 3D painting with 2D input devices was effectively limited to projection onto the surface of 3D objects so far. We have presented a method that allows the artist to treat the full 3D space as a canvas while still using the successful concept of proxy geometry as a general guide. The problem of embedding a 2D paint stroke in 3D space is ambiguous and the desired solution depends on the artist's intentions. Therefore, we approach the problem with a flexible optimization framework that can accommodate various criteria for the embedding. As a proof of concept, we define an example set of such criteria and show how they can be exposed to the user in an intuitive way.
- We have presented a system and workflow to author the animation of stroke-based 3D painted characters. To this end, we have adapted several techniques from related fields to our usage scenario: skinning deformation, sketch-based free-form deformation, and configuration-space and temporal keyframing. We have shown how these techniques can be arranged in a workflow that is intuitive and familiar to artists. The system was designed to meet the needs of character animation, which is one of the most difficult disciplines in animation, but it is also applicable to many other animation tasks where articulated motion dominates.
- We have developed a novel interpolation method that is tailored for configuration-space keyframing. Such an interpolation method has to be able to handle a high-dimensional domain while exhibiting some qualities that are specifically required for keyframe animation, such as exact interpolation, overall smoothness, and insensitivity to irrelevant dimensions. We have found that traditional methods do not satisfy these qualities suf-

ficiently. Our new interpolation method was designed with these qualities in mind, and we have shown how it can be used for configuration-space keyframing in the context of animating 3D paintings.

In addition to our efforts in the field of stroke-based rendering and animation, we have also investigated the depiction of motion in a more general 3D animation environment. We have presented a framework that extends the traditional rendering approach to treat the scene not only at a single instant of time, but over a certain period in time. With this extension, we have enabled the renderer to display the motion of objects in a single image. To allow for flexible customization and stylization of how motion is depicted, we have presented a programmable interface that continues the spirit of programmable surface shading. The data required at the interface is generated using a novel data structure and algorithms to capture the motion of objects over time in a piece-wise linear approximation.

6.2 Limitations

While the limitations of our methods have been discussed in detail at the end of the individual chapters, we summarize the most important ones here and discuss some additional ideas for future research.

The brush model we have chosen for our 3D painting system has some distinctive qualities: its flexibility, simplicity, and familiarity. But there are many potential alternatives that offer other qualities. For example, there have been excellent research projects in the realistic simulation of traditional paint media (watercolor, oil, pencil, etc). Adapting such technologies to 3D-based stroke rendering could greatly broaden the range of visual styles achievable in 3D painting. On the other hand, the search for new styles can also lead away from the aim of achieving traditional 2D looks and focus on more three-dimensional representations of brush strokes, such as volumetric ones. We expect that some of the problems addressed in this thesis will also apply to other brush models and, at the same time, each model will likely come with its own challenges.

In our 3D painting system, we have used layers of brush strokes mainly to make keyframe animation tractable and in addition as a means to introduce a logical structure to the paintings. However, we believe that the layering system has the potential of becoming much more powerful by attaching additional semantics to it. One example is the notion of compositing modes known from Photoshop and other similar programs. The interaction of 3D brush stroke layers under different compositing modes is a promising topic for future research. Layers could also be used to provide additional information for resolving the conflict between depth and paint order in a more controlled fashion.

While the animation tools we present in Chapter 4 closely resemble traditional methods for 3D animation, they stand out as rather technical in an environment where most interaction is based on the 2D painting input metaphor. The usability of our system could be improved with further abstractions of the animation mechanics from the user, such as sketch-based posing and deformation and high-level semantic controls for keyframing.

A similar argument also applies to programmable motion effects as described in Chapter 5. The programming interface ensures that a maximum amount of flexibility is provided for the design of motion depictions. But, since stylization is an artistic process, a more artist-friendly interface could greatly enhance the usability of our concepts. One could envision an example-based system where the user sketches a prototype of the desired motion effect into the image that is automatically propagated over time and space by novel algorithms.

Finally, our animation system was primarily designed to handle articulated motion, which is the dominant aspect when animating characters. Some types of motion, however, cannot easily be captured with this paradigm. Many visual effects, such as liquids, fire, and other turbulent gaseous media, do not exhibit enough structural coherence to be well approximated by the proxy geometry concept we use. While physically-based simulation could be used to animate brush strokes in such cases, the stylized visual nature of brush strokes calls for matching stylization in the animation, which is typically hard to achieve with physically-based simulation. An alternative could perhaps be found in procedural and example-based animation. There are also many situations where structural coherence is given, but the motion is very complex and detailed, making it impractical to produce with our system. Examples of such scenarios include the animation of hair or vegetation (trees, grass, etc.). We conjecture that such situations could be handled with a combination of large-scale articulated animation along the lines of what we described in this thesis, and a system for secondary motion that could be based on physical simulation or procedural animation.

6.3 Outlook

Stylized rendering and animation remains to be one of the major challenges in computer graphics. While the research community has already come up with an impressive range of solutions, only very few of them have found regular use in tools and productions outside of academia. The reasons for this lack of adoption are somewhat diffuse, but there are a two factors that we believe are of key importance for future research in the area.

Coming up with a “new visual style” is first and foremost an artistic process. Therefore, it is very important that technological research goes hand in hand

with evaluation and inspiration from artists. Artists often have very specific requirements both on the usage and the output of a technology. Since these requirements can be of fundamental nature, the dialog between technological and artistic research should take place from the first conceptual stage throughout the project. For example, it may be better to design a technology around the artist's need for controllability rather than to build a control mechanism around an existing technology.

What we ideally seek is not a single new look, but the tools and knowledge that enable us to create a diversity of new looks. Such a diversity is most likely not achievable with a single piece of technology. On the one hand, this factor is a motivation to keep research broad and to investigate into completely new directions. On the other hand, there should also be an effort to bring things back together. We believe that there is great potential in the careful combination of existing techniques, for example if line art is combined with 3D painting. Such a combination of different techniques, each with its own strengths and weaknesses, may succeed where no single technique was successful.

We believe that there is much to be done in this area of research that provides the exciting opportunity to bridge the worlds of art and technology. Following up on those opportunities should lead to some fundamentally new visual styles for animation.

List of Figures

List of Tables

Bibliography

[Agarwala et al., 2004] Agarwala, A., Dontcheva, M., Agrawala, M., Drucker, S., Colburn, A., Curless, B., Salesin, D., and Cohen, M. (2004). Interactive digital photomontage. *ACM Transactions on Graphics*, 23(3):294–302.

[Assa et al., 2005] Assa, J., Caspi, Y., and Cohen-Or, D. (2005). Action synopsis: pose selection and illustration. *ACM Transactions on Graphics*, 24(3):667–676.

[Bærentzen and Aanæs, 2005] Bærentzen, J. A. and Aanæs, H. (2005). Signed distance computation using the angle weighted pseudonormal. *IEEE Transactions on Visualization and Computer Graphics*, 11:243–253.

[Baran et al., 2009] Baran, I., Vlasic, D., Grinspun, E., and Popović, J. (2009). Semantic deformation transfer. *ACM Transactions on Graphics*, 28(3):36:1–36:6.

[Barla et al., 2006] Barla, P., Thollot, J., and Markosian, L. (2006). X-toon: an extended toon shader. In *Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, NPAR '06, pages 127–132.

[Baxter et al., 2001] Baxter, B., Scheib, V., Lin, M. C., and Manocha, D. (2001). DAB: interactive haptic painting with 3D virtual brushes. In *Proceedings of SIGGRAPH 01*, Annual Conference Series, pages 461–468.

[Baxter et al., 2004] Baxter, W., Wendt, J., and Lin, M. C. (2004). IMPaSTo:

a realistic, interactive model for paint. In *Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*, NPAR '04, pages 45–148.

[Bénard et al., 2009] Bénard, P., Bousseau, A., and Thollot, J. (2009). Dynamic solid textures for real-time coherent stylization. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D '09, pages 121–127.

[Bennett and McMillan, 2007] Bennett, E. P. and McMillan, L. (2007). Computational time-lapse video. *ACM Transactions on Graphics*, 26(3):102:1–102:6.

[Bernhardt et al., 2008] Bernhardt, A., Pihuit, A., Cani, M.-P., and Barthe, L. (2008). Matisse: Painting 2D regions for modeling free-form shapes. In *Eurographics Workshop on Sketch-Based Interfaces and Modeling*, pages 57–64, Annecy, France.

[Booker, 1963] Booker, P. (1963). *A history of engineering drawing*. Chatto & Windus.

[Bourguignon et al., 2001] Bourguignon, D., Cani, M.-P., and Drettakis, G. (2001). Drawing for illustration and annotation in 3D. *Computer Graphics Forum*, 20(3):114–122.

[Bousseau et al., 2007] Bousseau, A., Neyret, F., Thollot, J., and Salesin, D. (2007). Video watercolorization using bidirectional texture advection. *ACM Transactions On Graphics*, 26(3):104:1–104:7.

[Bouvier-Zappa et al., 2007] Bouvier-Zappa, S., Ostromoukhov, V., and Poulin, P. (2007). Motion cues for illustration of skeletal motion capture data. In *Proceedings of the 5th international symposium on Non-photorealistic animation and rendering*, NPAR '07, pages 133–140. ACM.

[Breslav et al., 2007] Breslav, S., Szerszen, K., Markosian, L., Barla, P., and Thollot, J. (2007). Dynamic 2D patterns for shading 3D scenes. *ACM Transactions On Graphics*, 26(3):20:1–20:6.

[Bruckner et al., 2010] Bruckner, S., Rautek, P., Viola, I., Roberts, M., Sousa, M. C., and Gröller, M. E. (2010). Hybrid visibility compositing and masking for illustrative rendering. *Computers and Graphics*, 34(4):361 – 369.

[Burr and Ross, 2002] Burr, D. C. and Ross, J. (2002). Direct evidence that "speedlines" influence motion mechanisms. *Journal of Neuroscience*, 22(19).

[Catmull, 1978] Catmull, E. (1978). A hidden-surface algorithm with anti-aliasing. *Computer Graphics*, 12(3):6–11.

[Chenney et al., 2002] Chenney, S., Pingel, M., Iverson, R., and Szymanski, M. (2002). Simulating cartoon style animation. In *Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, NPAR '02, pages 133–138.

[Choi et al., 2008] Choi, B., You, M., and Noh, J. (2008). Extended spatial keyframing for complex character animation. *Computer Animation and Virtual Worlds*, 19(3-4):175–188.

[Chu et al., 2010] Chu, N., Baxter, W., Wei, L.-Y., and Govindaraju, N. (2010). Detail-preserving paint modeling for 3D brushes. In *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '10, pages 27–34.

[Cohen et al., 2000] Cohen, J. M., Hughes, J. F., and Zeleznik, R. C. (2000). Harold: a world made of drawings. In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, NPAR '00, pages 83–90.

[Collomosse et al., 2005] Collomosse, J. P., Rowntree, D., and Hall, P. M. (2005). Rendering cartoon-style motion cues in post-production video. *Graph. Models*, 67(6).

[Cook et al., 1987] Cook, R. L., Carpenter, L., and Catmull, E. (1987). The Reyes image rendering architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, pages 95–102.

[Cook et al., 1984] Cook, R. L., Porter, T., and Carpenter, L. (1984). Distributed ray tracing. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, pages 137–145.

[Curtis et al., 1997] Curtis, C. J., Anderson, S. E., Seims, J. E., Fleischer, K. W., and Salesin, D. H. (1997). Computer-generated watercolor. In *Proceedings of SIGGRAPH 97*, Annual Conference Series, pages 421–430.

[Cutting, 2002] Cutting, J. E. (2002). Representing motion in a static image: constraints and parallels in art, science, and popular culture. *Perception*, 31(10).

[Daniels et al., 2001] Daniels, E., Lappas, A., and Katanics, G. T. (2001). *Method and apparatus for three-dimensional painting.* US Patent 6268865.

[DeCarlo et al., 2003] DeCarlo, D., Finkelstein, A., Rusinkiewicz, S., and Santella, A. (2003). Suggestive contours for conveying shape. *ACM Transactions on Graphics*, 22(3):848–855.

[Decaudin, 1996] Decaudin, P. (1996). Cartoon-looking rendering of 3D-scenes. Technical Report 2919, INRIA Rocquencourt.

[Dubuc, 1986] Dubuc, S. (1986). Interpolation through an iterative scheme. *Journal of Mathematical Analysis and Applications*, 114(1):185 – 204.

[Durand, 2002] Durand, F. (2002). An invitation to discuss computer depiction. In *Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, pages 111–124.

[Dyn et al., 1987] Dyn, N., Levin, D., and Gregory, J. A. (1987). A 4-point interpolatory subdivision scheme for curve design. *Computer Aided Geometric Design*, 4(4):257 – 268.

[Fóris et al., 1996] Fóris, T., Márton, G., and Szirmay-Kalos, L. (1996). Ray shooting in logarithmic time. In *WSCG'96 - 4-th International Conference in Central Europe on Computer Graphics and Visualization'96*.

[Franke, 1977] Franke, R. (1977). Locally determined smooth interpolation at irregularly spaced points in several variables. *IMA Journal of Applied Mathematics*, 19(4):471–482.

[Frisken et al., 2000] Frisken, S. F., Perry, R. N., Rockwood, A. P., and Jones, T. R. (2000). Adaptively sampled distance fields: a general representation of shape for computer graphics. In *Proceedings of SIGGRAPH 2000*, Annual Conference Series, pages 249–254.

[Goldberg, 2008] Goldberg, E. (2008). *Character Animation Crash Course!* Silman-James Press.

[Goldman et al., 2006] Goldman, D. B., Curless, B., Salesin, D., and Seitz, S. M. (2006). Schematic storyboarding for video visualization and editing. *ACM Transactions on Graphics*, 25(3):862–871.

[Gooch et al., 2002] Gooch, B., Coombe, G., and Shirley, P. (2002). Artistic Vision: painterly rendering using computer vision techniques. In *Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, NPAR '02, pages 83–90.

[Gottschalk et al., 1996] Gottschalk, S., Lin, M., and Manocha, D. (1996). OBB-tree: a hierarchical structure for rapid interference detection. In *Proceedings of SIGGRAPH 96*, Annual Conference Series, pages 171–180.

[Grabli et al., 2010] Grabli, S., Turquin, E., Durand, F., and Sillion, F. X. (2010). Programmable rendering of line drawing from 3D scenes. *ACM Transactions on Graphics*, 29(2):18:1–18:20.

[Grant, 1985] Grant, C. W. (1985). Integrated analytic spatial and temporal anti-aliasing for polyhedra in 4-space. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, pages 79–84.

[Haeberli, 1990] Haeberli, P. E. (1990). Paint by numbers: abstract image representations. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, pages 207–214.

[Haller et al., 2004] Haller, M., Hanl, C., and Diephuis, J. (2004). Non-photorealistic rendering techniques for motion in computer games. *Comput. Entertain.*, 2(4).

[Hanrahan and Lawson, 1990] Hanrahan, P. and Lawson, J. (1990). A language for shading and lighting calculations. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, pages 289–298.

[Hart, 1994] Hart, J. C. (1994). Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12:527–545.

[Hays and Essa, 2004] Hays, J. and Essa, I. (2004). Image and video based painterly animation. In *Proceedings of the 3rd international symposium on Non-photorealistic animation and rendering*, NPAR '04, pages 113–120.

[Hegland et al., 1997] Hegland, M., Roberts, S., and Altas, I. (1997). Finite element thin plate splines for surface fitting. *Computational Techniques and Applications: CTAC97*, pages 289–296.

[Hertzmann, 1998] Hertzmann, A. (1998). Painterly rendering with curved brush strokes of multiple sizes. In *Proceedings of SIGGRAPH 98*, Annual Conference Series, pages 453–460.

[Hertzmann, 2003] Hertzmann, A. (2003). A survey of stroke-based rendering. *Computer Graphics and Applications, IEEE*, 23(4):70 – 81.

[Hertzmann et al., 2001] Hertzmann, A., Jacobs, C. E., Oliver, N., Curless, B., and Salesin, D. H. (2001). Image analogies. In *Proceedings of SIGGRAPH 2001*, Annual Conference Series, pages 327–340.

[Hertzmann and Perlin, 2000] Hertzmann, A. and Perlin, K. (2000). Painterly rendering for video and interaction. In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, NPAR '00, pages 7–12.

[Hertzmann and Zorin, 2000] Hertzmann, A. and Zorin, D. (2000). Illustrating smooth surfaces. In *Proceedings of SIGGRAPH 2000*, Annual Conference Series, pages 517–526.

[Horn, 1987] Horn, B. K. P. (1987). Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society of America A*, 4(4):629–642.

[Hsu and Lee, 1994] Hsu, S. C. and Lee, I. H. H. (1994). Drawing and animation using skeletal strokes. In *Proceedings of SIGGRAPH 94*, Annual Conference Series, pages 109–118.

[Hulten, 1986] Hulten, P. (1986). *Futurism & Futurisms*. Abbeville Press.

[Igarashi et al., 1999] Igarashi, T., Matsuoka, S., and Tanaka, H. (1999). Teddy: a sketching interface for 3D freeform design. In *Proceedings of SIGGRAPH 99*, Annual Conference Series, pages 409–416.

[Igarashi et al., 2005] Igarashi, T., Moscovich, T., and Hughes, J. (2005). Spatial keyframing for performance-driven animation. In *2005 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 107–116.

[Jacobson et al., 2011] Jacobson, A., Baran, I., Popović, J., and Sorkine, O. (2011). Bounded biharmonic weights for real-time deformation. *ACM Transactions on Graphics*, 30(4):78:1–78:8.

[Judd et al., 2007] Judd, T., Durand, F., and Adelson, E. (2007). Apparent ridges for line drawing. *ACM Transactions on Graphics*, 26(3):19:1–19:7.

[Kalnins et al., 2002] Kalnins, R. D., Markosian, L., Meier, B. J., Kowalski, M. A., Lee, J. C., Davidson, P. L., Webb, M., Hughes, J. F., and Finkelstein, A. (2002). WYSIWYG NPR: drawing strokes directly on 3D models. *ACM Transactions on Graphics*, 21(3):755–762.

[Kalogerakis et al., 2012] Kalogerakis, E., Nowrouzezahrai, D., Breslav, S., and Hertzmann, A. (2012). Learning hatching for pen-and-ink illustration of surfaces. *ACM Transactions On Graphics*, 31(1):1:1–1:17.

[Karpenko et al., 2002] Karpenko, O., Hughes, J. F., and Raskar, R. (2002). Free-form sketching with variational implicit surfaces. *Computer Graphics Forum*, 21(3):585–594.

[Kass and Pesare, 2011] Kass, M. and Pesare, D. (2011). Coherent noise for non-photorealistic rendering. *ACM Transactions on Graphics*, 30(4):30:1–30:6.

[Katanics and Lappas, 2003] Katanics, G. and Lappas, T. (2003). Deep Canvas: integrating 3D painting and painterly rendering. In *Theory and Practice of Non-Photorealistic Graphics: Algorithms, Methods, and Production Systems*, ACM SIGGRAPH 2003 Course Notes.

[Kawagishi et al., 2003] Kawagishi, Y., Hatsuyama, K., and Kondo, K. (2003). Cartoon blur: non-photorealistic motion blur. *Computer Graphics International Conference*.

[Keefe et al., 2007] Keefe, D., Zeleznik, R., and Laidlaw, D. (2007). Drawing on air: input techniques for controlled 3D line illustration. *IEEE Transactions on Visualization and Computer Graphics*, 13:1067–1081.

[Keefe et al., 2001] Keefe, D. F., Feliz, D. A., Moscovich, T., Laidlaw, D. H., and LaViola, Jr., J. J. (2001). CavePainting: a fully immersive 3D artistic medium and interactive experience. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 85–93.

[Klein et al., 2000] Klein, A. W., Li, W. W., Kazhdan, M. M., Correa, W. T., Finkelstein, A., and Funkhouser, T. A. (2000). Non-photorealistic virtual

environments. In *Proceedings of SIGGRAPH 2000*, Annual Conference Series, pages 527–534.

[Korein and Badler, 1983] Korein, J. and Badler, N. (1983). Temporal anti-aliasing in computer generated animation. In *Computer Graphics (Proceedings of SIGGRAPH 83)*, pages 377–388.

[Kowalski et al., 1999] Kowalski, M. A., Markosian, L., Northrup, J. D., Bourdev, L., Barzel, R., Holden, L. S., and Hughes, J. F. (1999). Art-based rendering of fur, grass, and trees. In *Proceedings of SIGGRAPH 99*, Annual Conference Series, pages 433–438.

[Lake et al., 2000] Lake, A., Marshall, C., Harris, M., and Blackstein, M. (2000). Stylized rendering techniques for scalable real-time 3D animation. In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, NPAR '00, pages 13–20.

[Lewis et al., 2000] Lewis, J. P., Cordner, M., and Fong, N. (2000). Pose space deformations: a unified approach to shape interpolation and skeleton-driven deformation. In *Proceedings of SIGGRAPH 2000*, Annual Conference Series, pages 165–172.

[Lin et al., 2010] Lin, L., Zeng, K., Lv, H., Wang, Y., Xu, Y., and Zhu, S.-C. (2010). Painterly animation using video semantics and feature correspondence. In *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '10, pages 73–80.

[Litwinowicz, 1997] Litwinowicz, P. (1997). Processing images and video for an impressionist effect. In *Proceedings of SIGGRAPH 97*, Annual Conference Series, pages 407–414.

[Liu et al., 2005] Liu, C., Torralba, A., Freeman, W. T., Durand, F., and Adelson, E. H. (2005). Motion magnification. *ACM Transactions On Graphics*, 24(3):519–526.

[Lu et al., 2010] Lu, J., Sander, P. V., and Finkelstein, A. (2010). Interactive painterly stylization of images, videos and 3D animations. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 127–134.

[Luft and Deussen, 2006] Luft, T. and Deussen, O. (2006). Real-time watercolor illustrations of plants using a blurred depth test. In *Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, NPAR '06, pages 11–20.

[Magnenat-Thalmann et al., 1988] Magnenat-Thalmann, N., Laperrière, R., and Thalmann, D. (1988). Joint-dependent local deformations for hand animation and object grasping. In *Graphics Interface*, pages 26–33.

[Mammen, 1989] Mammen, A. (1989). Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics & Applications*, 9(4):43–55.

[Masuch et al., 1999] Masuch, M., Schlechtweg, S., and Schulz, R. (1999). Speedlines: depicting motion in motionless pictures. In *ACM SIGGRAPH 99 Conference abstracts and applications*.

[McCann and Pollard, 2009] McCann, J. and Pollard, N. (2009). Local layering. *ACM Transactions on Graphics*, 28(3):84:1–84:7.

[McCloud, 1993] McCloud, S. (1993). *Understanding Comics*. Kitchen Sink Press.

[Meier, 1996] Meier, B. J. (1996). Painterly rendering for animation. In *Proceedings of SIGGRAPH 96*, Annual Conference Series, pages 477–484.

[Nealen et al., 2007] Nealen, A., Igarashi, T., Sorkine, O., and Alexa, M. (2007). FiberMesh: designing freeform surfaces with 3D curves. *ACM Transactions on Graphics*, 26(3):41:1–41:9.

[Nienhaus and Döllner, 2005] Nienhaus, M. and Döllner, J. (2005). Depicting dynamics using principles of visual art and narrations. *IEEE Computer Graphics and Applications*, 25(3).

[Noble and Tang, 2007] Noble, P. and Tang, W. (2007). Automatic expressive deformations for implying and stylizing motion. *The Visual Computer*, 23(7).

[Nocedal and Wright, 2006] Nocedal, J. and Wright, S. (2006). *Numerical Optimization, Series in Operations Research and Financial Engineering*. Springer, New York, 2nd edition.

[Northrup and Markosian, 2000] Northrup, J. D. and Markosian, L. (2000). Artistic silhouettes: a hybrid approach. In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, NPAR '00, pages 31–38.

[Paint Effects, 2011] Paint Effects (2011). Painting in 3D using Paint Effects. In *Autodesk Maya Learning Resources*. http://download.autodesk.com/us/maya/2011help.

[Peng et al., 2004] Peng, J., Kristjansson, D., and Zorin, D. (2004). Interactive modeling of topologically complex geometric detail. *ACM Transactions on Graphics*, 23(3):635–643.

[Porter and Duff, 1984] Porter, T. and Duff, T. (1984). Compositing digital images. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, pages 253–259.

[Potmesil and Chakravarty, 1983] Potmesil, M. and Chakravarty, I. (1983). Modelling motion blur in computer-generated images. In *Computer Graphics (Proceedings of SIGGRAPH 83)*, pages 389–399.

[Praun et al., 2001] Praun, E., Hoppe, H., Webb, M., and Finkelstein, A. (2001). Real-time hatching. In *Proceedings of SIGGRAPH 2001*, Annual Conference Series, pages 579–584.

[Rademacher, 1999] Rademacher, P. (1999). View-dependent geometry. In *Proceedings of SIGGRAPH 99*, Annual Conference Series, pages 439–446.

[Ramsey et al., 2004] Ramsey, S. D., Potter, K., and Hansen, C. (2004). Ray bilinear patch intersections. *Journal of Graphics, GPU, and Game Tools*, 9(3).

[Rivers et al., 2010] Rivers, A., Igarashi, T., and Durand, F. (2010). 2.5D cartoon models. *ACM Transactions on Graphics*, 29(4):59:1–59:7.

[Saito and Takahashi, 1990] Saito, T. and Takahashi, T. (1990). Comprehensible rendering of 3-D shapes. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, pages 197–206.

[Salisbury et al., 1997] Salisbury, M. P., Wong, M. T., Hughes, J. F., and Salesin, D. H. (1997). Orientable rextures for image-based pen-and-ink illustration. In *Proceedings of SIGGRAPH 97*, Annual Conference Series, pages 401–406.

[Schkolne et al., 2001] Schkolne, S., Pruett, M., and Schröder, P. (2001). Surface drawing: creating organic 3D shapes with the hand and tangible tools. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 261–268.

[Schmidt et al., 2005] Schmidt, R., Wyvill, B., Sousa, M., and Jorge, J. (2005). Shapeshop: Sketch-based solid modeling with blobtrees. In *Eurographics Workshop on Sketch-Based Interfaces and Modeling*, pages 53–62.

[Schwarz et al., 2007] Schwarz, M., Isenberg, T., Mason, K., and Carpendale, S. (2007). Modeling with rendering primitives: an interactive non-photorealistic canvas. In *Proceedings of the 5th international symposium on Non-photorealistic animation and rendering*, NPAR '07, pages 15–22.

[Shepard, 1968] Shepard, D. (1968). A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM national conference*, pages 517–524.

[Sibson, 1981] Sibson, R. (1981). *Interpolating multivariate data*, pages 21–36. John Wiley & Sons.

[Singh and Kokkevis, 2000] Singh, K. and Kokkevis, E. (2000). Skinning characters using surface oriented free-form deformations. In *Graphics Interface*, pages 35–42.

[Sloan et al., 2001] Sloan, P.-P., Martin, W., Gooch, A., and Gooch, B. (2001). The lit sphere: a model for capturing NPR shading from art. In *Graphics Interface 2001*, pages 143–150.

[Smith, 1982] Smith, A. R. (1982). Paint. *Tutorial: Computer Graphics*, pages 501–515.

[Smith, 1995] Smith, A. R. (1995). Alpha and the history of digital compositing. In *Microsoft Technical Memo #7*.

[Sousa and Buchanan, 2000] Sousa, M. C. and Buchanan, J. W. (2000). Observational models of graphite pencil materials. *Computer Graphics Forum*, 19(1):27–49.

[Sung et al., 2002] Sung, K., Pearce, A., and Wang, C. (2002). Spatial-temporal antialiasing. *IEEE Transactions on Visualization and Computer Graphics*, 8(2).

[Teece, 2000] Teece, D. (2000). Animating with expressive 3D brush strokes (animation abstract). In *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*.

[Thomas and Johnston, 1981] Thomas, F. and Johnston, O. (1981). *The Illusion of Life: Disney Animation*, page 210. Disney Editions.

[Tolba et al., 2001] Tolba, O., Dorsey, J., and McMillan, L. (2001). A projective drawing system. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 25–34.

[Vanderhaeghe et al., 2011] Vanderhaeghe, D., Vergne, R., Barla, P., and Baxter, W. (2011). Dynamic stylized shading primitives. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering*, NPAR '11, pages 99–104.

[Wallace, 1981] Wallace, B. A. (1981). Merging and transformation of raster images for cartoon animation. In *Computer Graphics (Proceedings of SIGGRAPH 81)*, pages 253–262.

[Wang et al., 2006] Wang, J., Drucker, S. M., Agrawala, M., and Cohen, M. F. (2006). The cartoon animation filter. *ACM Transactions on Graphics*, 25(3):1169–1173.

[Whitaker and Halas, 2002] Whitaker, H. and Halas, J. (2002). *Timing for Animation*. Focal Press.

[Willats, 1997] Willats, J. (1997). *Art and representation: new principles in the analysis of pictures*. Princeton University Press.

[Winkenbach and Salesin, 1994] Winkenbach, G. and Salesin, D. H. (1994). Computer-generated pen-and-ink illustration. In *Proceedings of SIGGRAPH 94*, Annual Conference Series, pages 91–100.

[Winkenbach and Salesin, 1996] Winkenbach, G. and Salesin, D. H. (1996). Rendering parametric surfaces in pen and ink. In *Proceedings of SIGGRAPH 96*, Annual Conference Series, pages 469–476.

[Winnemöller et al., 2006] Winnemöller, H., Olsen, S. C., and Gooch, B. (2006). Real-time video abstraction. *ACM Transactions On Graphics*, 25:1221–1226.

[Wyvill et al., 1986] Wyvill, G., McPheeters, C., and Wyvill, B. (1986). Data structure for soft objects. *The Visual Computer*, 2(4):227–234.

[Zeng et al., 2009] Zeng, K., Zhao, M., Xiong, C., and Zhu, S.-C. (2009). From image parsing to painterly rendering. *ACM Transactions On Graphics*, 29:2:1–2:11.

[Zhao and Zhu, 2010] Zhao, M. and Zhu, S.-C. (2010). Sisley the abstract painter. In *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*, NPAR '10, pages 99–107.

Johannes Schmid

Date of Birth:	26 Mar 1982
Nationality:	Switzerland
E-Mail:	johannes.schmid@grob.org

Education

since Aug 2008	Ph. D. student at ETH Zurich and Disney Research Zurich.
Apr 2008	Master of Science ETH in Computer Science. Specialization in Computational Science, minor in System Software Engineering.
Aug 2002 – Apr 2008	Student in computer science at ETH Zurich.

Employment

May 2012 – Jul 2012	Software Engineer at Disney Research Zurich, *Switzerland*.
May 2008 – Apr 2012	Research Assistant at ETH Zurich, *Switzerland*.
Oct 2006 – Jun 2007	Research Assistant at ETH Zurich, *Switzerland*.
Apr 2005 – Oct 2005	Software Development Intern at Maplesoft, *Waterloo, ON, Canada*.

Scientific Publications

Sumner, R. W., Schmid, J., Pauly, M. 2007. Embedded deformation for shape manipulation. In *ACM Transactions on Graphics*, 26(3), 80:1–80:7.

Schmid, J., Sumner, R. W., Bowles, H., and Gross, M. 2010. Programmable motion effects. In *ACM Transactions on Graphics*, 29(4), 57:1–57:9.

Schmid, J., Senn, M. S., Gross, M., and Sumner, R. W. 2011. OverCoat: an implicit canvas for 3D painting. In *ACM Transactions on Graphics*, 30(4), 28:1–28:10.

Baran, I., Schmid, J., Siegrist, T., Gross, M., and Sumner, R. W. 2011. Mixed-order compositing for 3D paintings. In *ACM Transactions on Graphics*, 30(6), 132:1–132:6.