NON-PHOTOREALISTIC RENDERING WITH COHERENCE FOR AUGMENTED REALITY

A Dissertation
Presented to
The Academic Faculty

by

Jiajian Chen

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Interactive Computing, College of Computing

Georgia Institute of Technology
December 2012

NON-PHOTOREALISTIC RENDERING WITH COHERENCE FOR AUGMENTED REALITY

Approved by:

Professor Blair MacIntyre, Advisor
School of Interactive Computing,
College of Computing
Georgia Institute of Technology

Professor Greg Turk, Advisor
School of Interactive Computing,
College of Computing
Georgia Institute of Technology

Professor Frank Dellaert
School of Interactive Computing,
College of Computing
Georgia Institute of Technology

Professor Irfan Essa
School of Interactive Computing,
College of Computing
Georgia Institute of Technology

Professor Eugene Zhang
School of Electrical Engineering and
Computer Science
Oregon State University

Date Approved: July 13th, 2012

ACKNOWLEDGEMENTS

I would first like to thank all my committee members for their time and interest. Your expertise and comments were invaluable during my thesis research.

I would like to thank the members of the Augmented Environments Lab and Graphics Geometry Research Group in the GVU Center at Georgia Tech.

I would also like to thank my wife, Ying, and my son, Nicholas who have given me the strength and confidence to complete the thesis.

Finally I would like to thank both my advisors, Greg Turk and Blair MacIntyre. They have been giving me tremendous help with my research in the last six years. Their guidance and support were key to my thesis work, and in pointing the direction for further research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

A seamless blending of the real and virtual worlds is key to increased immersion and improved user experiences for augmented reality (AR). Photorealistic and non-photorealistic rendering (NPR) are two ways to achieve this goal. Non-photorealistic rendering creates an abstract and stylized version of both the real and virtual world, making them indistinguishable. NPR hides unnecessary and overwhelming details of the world, and reveals and emphasizes important information of the scene. This could be particularly useful in some applications (e.g., AR/VR aided machine repair, or for virtual medical surgery) or for certain AR games with artistic stylization.

Achieving temporal coherence is a key challenge for all NPR algorithms. Rendered results are temporally coherent when each frame smoothly and seamlessly transitions to the next one without visual flickering or artifacts that distract the eye from perceived smoothness. For example, temporal coherence is maintained in brush-based NPR algorithms if the brush stroke located at a certain anchor point smoothly transitions to a corresponding anchor point from one frame to the next. NPR algorithms with coherence are interesting in both general computer graphics and AR/VR areas. Although NPR has attracted many researchers' interest in the AR/VR community, very few of the existing algorithms can support temporal coherence. Rendering stylized AR without coherence processing causes the final results to be visually distracting. While various NPR algorithms with coherence support have been proposed in the general graphics community for video processing, many of these algorithms require thorough analysis of all frames of the input video and cannot be directly applied to real-time AR applications. We have investigated existing NPR algorithms with coherence in both general graphics and AR/VR areas. These algorithms are divided into

two categories: Model Space and Image Space. We present several NPR algorithms with coherence for AR: a watercolor inspired NPR algorithm, a painterly rendering algorithm, and NPR algorithms in the model space that will support several styling effects.

CHAPTER I

INTRODUCTION

1.1 Introduction

Photorealistic and non-photorealistic rendering are two major approaches to seamlessly blend the real and virtual worlds in AR. *Non-photorealistic rendering* is the creation of images by a computer that look as though they were created by a human artist. Compared with photorealistic rendering [64], non-photorealistic rendering may be more appropriate to some AR applications. For example, an abstract stylization in AR hides unnecessary and overwhelming details of the world, and reveals and emphasizes important information of the scene. This can be particularly useful in some applications, such as AR/VR aided machine repair, or for certain AR games with artistic stylization.

A major challenge for non-photorealistic rendering (NPR) is maintaining temporal coherence for the composed AR video. Rendered results are temporally coherent when each frame smoothly and seamlessly transitions to the next one without visual flickering and artifacts that distract the eye. For example, temporal coherence is maintained in brush stroke based NPR algorithms if the brush stroke located at a certain anchor point smoothly transitions to a corresponding anchor point from one frame to the next one. NPR algorithms with coherence are interesting in both general computer graphics and AR/VR areas. Based on the literature in the AR/VR community we believe that this problem has not yet been solved. Most NPR techniques for AR/VR simply render each AR frame independently and do not consider temporal coherence at all. The resulting videos usually appear to flicker and are visually distracting. Although some NPR algorithms with good coherence (e.g., particle

based method, NPR textures and spatial-temporal volume) have been proposed in the general graphics area, applying these methods to AR/VR has many restrictions. These algorithms require analysis of the entire sequence of videos, or full knowledge of the 3D geometry of the scene. On the other hand, AR applications must run at interactive frame rates, they cannot look ahead to future video frames, and have only limited information about the geometry of the scene.

This thesis will focus on NPR algorithms with coherence for AR. We have investigated existing NPR algorithms with coherence in the general graphics and the AR/VR communities. We divide these algorithms into two major categories based on their approaches of maintaining coherence: image space algorithms and model space algorithms. We will present three NPR algorithms in both categories with coherence for AR in this thesis.

1.2 Thesis Statement

Blending the real world and virtual graphics content in a seamless way is a key to improve user experiences in Augmented Reality. Non-photorealistic rendering is an important approach to achieve this goal by stylizing both the real and the virtual content. Maintaining temporal coherence is an important attribute for the visual quality of the rendered results in NPR algorithms. We are focused on improving coherence for AR NPR by leveraging information that is extracted from both image space and model space.

1.3 Contributions

In this thesis we present three NPR algorithms with coherence for AR, two that are image space algorithms and one that is a model space algorithm. Our contribution is that we utilize the information that is extracted from both the real and virtual content to maintain temporal coherence for AR NPR in our algorithms.

1 We present a watercolor inspired NPR algorithm for AR. Our contribution is using Voronoi diagrams to produce the watercolor-like rendering style with coherence at interactive frame rates.

Our algorithm produces a watercolor-like visual effect for the AR videos. Coherence is achieved by computing the center average of edge pixels in each Voronoi cell and re-tilling Voronoi cells based on these edge centers.

2 We present a painterly rendering algorithm in image space for AR. Our contribution is using a pyramid structure to create the tensor field and warping brush anchors in feature triangles between frames.

Our algorithm renders the AR video with a paint-and-brush style. A tensor pyramid is presented to create the tensor field for each frame. Coherence is achieved by warping brush stroke anchors from frame to frame and placing brush strokes at these anchor positions. Each brush stroke is also re-shaped to provide better coherence in the final rendering.

3 We present an NPR framework that supports temporal coherence in model space for AR. Our contribution is a new anchor generation algorithm that distributes point samples on a model's surface. Another contribution is a general averaging method to make the brush strokes more coherent.

We present a model space NPR framework that supports temporal coherence for AR. Our systems targets the painterly rendering style of NPR. The contribution of our algorithm has two parts. The first part is the anchor sampling algorithm. This algorithm maintains a proper density of brush anchors on the screen, which is a desired feature for many NPR algorithms. This algorithm is also particularly suitable for AR, as we do not know the camera motion in advance. By controlling the brush anchor density in 2D and then back-projecting the anchors onto the surfaces, our algorithm

achieves better coherence for the brush stroke placement. The second contribution is our method of averaging brush properties, including their skeletons and colors, to achieve better coherence in the final rendering. Compared with existing methods, our method allows us to smoothly blend curly brush strokes with a cubic B-spline representation. We apply these methods to both static and animated models to create a painterly rendering style for AR. Compared with existing image space algorithms our method renders AR with NPR effects with a high degree of coherence.

CHAPTER II

RELATED WORK

2.1 Introduction

There are a large number of NPR algorithms for both still images and videos in the general graphics and AR/VR literature. The most applicable approaches related to our work are video NPR and real-time NPR, in which coherence is a key issue.

Based on the type of input data and the solutions of maintaining temporal coherence for the rendered results, we divide the existing algorithms in video and real-time NPR literature into two main categories: image space algorithms, and model space algorithms.

In the image space approaches the input is a sequence of real-world images (i.e., from video) or computer-synthesized images. We have only very limited knowledge of the geometric information of the scene. In contrast, we have the full 3D geometry of the scene as input in the model space approaches. Various methods of maintaining temporal coherence have been studied in these two categories. Researchers also have proposed methods that use both the information from the the images and models to improve temporal coherence. We will discuss each of these in turn.

2.2 Model Space NPR Algorithms

Model space NPR algorithms render 3D scenes with stylization. Among all of NPR algorithms that stylize 3D models and scenes, we will focus on the algorithms that can maintain temporal coherence of the final rendering for animated 3D scenes. In this category, while almost every model space NPR approach uses the geometry information of the 3D models to provide coherency, they use this information in various

ways. There are several major types of algorithms in the model space approach for maintaining coherence: particle based algorithms, pre-built non-photorealistic textures, and geometry analysis and brush parameterization based algorithms. Each major type of algorithms has different approaches to maintain coherence across the resulting images.

Among all of these NPR algorithms in computer graphics literature, probably Barbara Meier's particle based algorithm is the first and one of the most influential NPR algorithms for rendering 3D models with temporal coherence [60]. Her algorithm inspired many later researchers. A variety of NPR algorithms for rendering 3D models are proposed and studied to further improve the coherence for rendering 3D models [51, 59, 47].

2.2.1 Particle Based Algorithms

Image and video NPR has been attracting researcher's attention in computer graphics community for decades. Gooch et al.'s book gives a comprehensive overview of NPR algorithms [26].

Directly applying NPR techniques designed for still images to videos yields flicking in the final results. The brush strokes created at each individual frame cannot smoothly transition to the next frame, causing an annoying "shower door" effect. Winkenbach et al. [75] and Curtis et al. [21] observed and recognized the challenge of maintaining temporal coherence in brush-based NPR systems.

Barbara Meier presented an influential particle based NPR system that achieves high quality temporal coherence for animated 3D scenes [60]. She formally identified the "shower door" effect (i.e., the rendered scene appears as if it were being viewed through textured glass) as a major challenge for achieving frame-to-frame coherence in animations. The undesired "shower door" effect is present in some NPR algorithms because the brush stroke positions are fixed to the view plane instead of the animated

surfaces of the 3D models.

Meier's algorithm places 3D particles on the 3D model surfaces. Each particle is associated with a 2D brush stroke texture. The appearance of a brush stroke texture is controlled by the geometric and lighting properties of the surface geometry. During rendering, these particles with brush stroke textures are rendered from back to front. The combination of 3D particles and brush stroke textures forces the brush strokes to stick to the animated surfaces. Hence the brush strokes smoothly move from frame to frame without any obvious 'jumps' during animation and camera motion. Her seminal algorithm achieves temporal coherence for 3D models with a painterly rendering style. The rendering pipeline of Meier's algorithm is illustrated in Figure 1 [60].

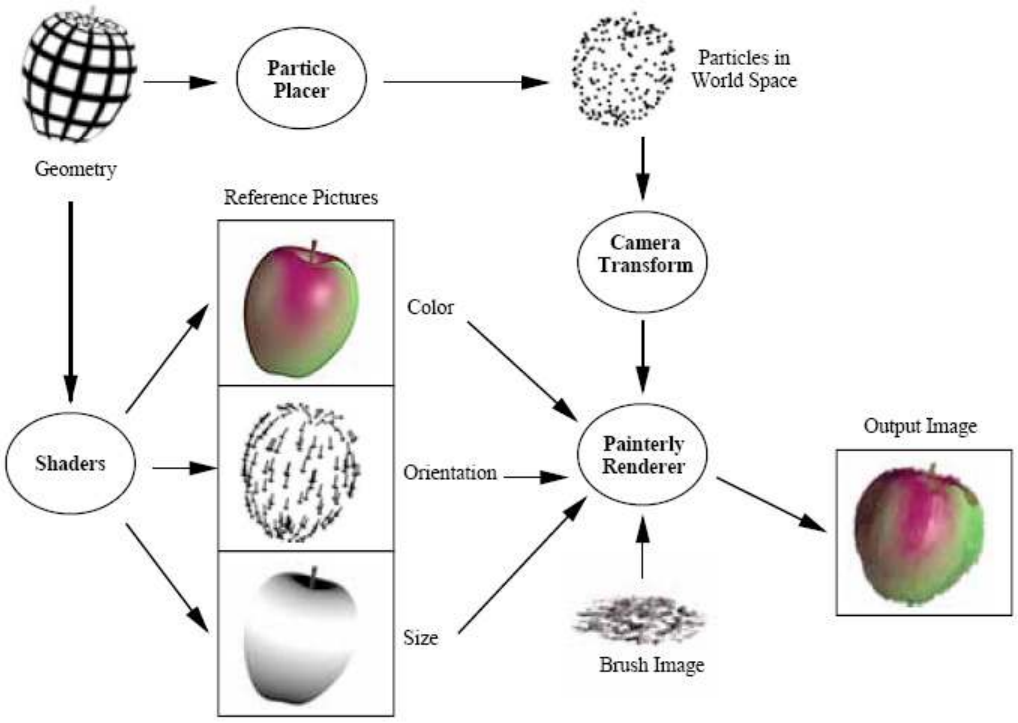

Figure 1: Meier's painterly rendering pipeline. Particles are generated on the model's surface by the particle placer. The particles, which are projected onto the screen, the reference pictures, and the brush image are input to the painterly renderer. The renderer looks up brush stroke attributes in the reference pictures at particle's projections on the screen, and then renders brush strokes for the final result. (image from [60])

Meier's algorithm generates 3D particles on model surfaces and associates them

with brush stroke textures to achieve coherence in NPR. One major factor that contributes to the success of her algorithm is the generation of 3D samples on surfaces. Her algorithm picks samples in triangles with a probability proportional to the area of the triangles. However, the sample set generated by this method is uniform across the mesh surface space instead of across screen space. As a result, the 2D projections of these 3D samples at each frame are not uniformly distributed on the screen for brush placement. The same problem shows up when we apply her algorithm to render virtual objects in AR/VR. We will discuss the importance of 3D sampling on surfaces for the purpose of NPR later in this section.

Meier's algorithm renders back-facing particles, which can introduce visual artifacts in the final rendering, since the occluded geometry may become visible in certain scenarios. On the other hand, particles will pop in and out as they become visible and invisible if the method culls the back-facing particles. This problem can be alleviated by a better sampling algorithm that can adjust the density of samples on surfaces to make brush strokes perfectly cover the screen. During the rendering of each frame, the particles are sorted by distance from the current viewpoint. There will be some popping of brush strokes in front or behind one another as the particles are animated.

Meier's inspiring algorithm has been adapted and extended by many researchers to create various NPR effects with coherence for rendering 3D models. Kowalski et al. extended Meier's work and used stroke-based procedural textures, called "graftals", to render fur, grass and trees [51]. Their approach places graftals at silhouettes but tends to omit them in interior surface regions. The appearance of graftals can be controlled by users to produce different stylizations. The key for maintaining temporal coherence in their approach is placing graftals with controlled screen-space density matching the aesthetic requirements of stylization, and sticking them to surfaces in the scene. To meet the requirement that graftals appear to stick to surfaces in the scene, they convert the 2D screen position of a graftal to a 3D position on a surface.

This is achieved in $O(1)$ time (per graftal) by using the ID reference image to find the triangle (and the exact point on the triangle with a ray test) corresponding to a given screen position. Their approach also uses multi-level hierarchy to store brushes. Their method changes the number of brush levels to be drawn to improve coherence in the camera motion, as shown in Figure 2.

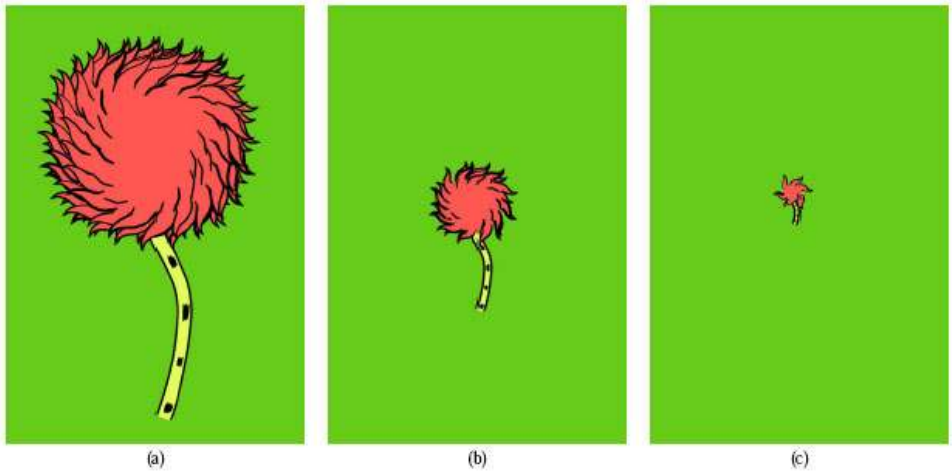

Figure 2: A treetop rendered with three-level static graftals. When the camera zooms in, all three levels are drawn (a). As the camera zooms out of the scene, only two levels are drawn (b), and finally just the base level is drawn (c). (image from [51])

Using levels of detail to adjust the number of brush textures that are applied to a surface is also useful for rendering models in AR. Because models such as plants are very common in AR games, these level of detail methods are particulary suitable for rendering objects such as trees and grasses without painstakingly analyzing the detailed geometry. This method is an alternative to distributing samples on complex geometry for maintaining coherence in NPR. When the camera zooms in or zooms out, it may be necessary to apply additional techniques such as alpha blending to make the appearance and disappearance of brush textures look more smooth.

Following Kowalski et al.'s rendering algorithm, Markosian et al. proposed to place static graftals on surfaces during the modeling phase and did not redistribute the graftals each frame to maintain temporal coherence [59]. Combing the work of

Meier and Kowalski, Kaplan et al. proposed an interactive NPR system for model rendering with better coherence support [47].

Point sampling is a key step for particle based NPR algorithms. It is also important for rendering the 3D virtual objects in AR/VR with coherence. We will discuss the variants of 3D point sampling algorithms in the following section.

2.2.1.1 Surface Sampling for Particle Based NPR Algorithms

Generating 3D anchors on model surfaces is a key step for the particle based NPR algorithms, since their 2D projections are the locations for brush placement in the rendering stage. A more general version of this problem, point sampling on arbitrary surfaces that satisfies a certain distribution, is an important research area in computer graphics. Point placement can benefit many graphics applications, such as texture mapping [70], non-photorealistic rendering [60], remeshing [69, 1], and point-based graphics rendering [29].

One popular basic solution for generating 3D samples on surfaces is to randomly generate 3D samples on triangles using barycentric coordinates in a manifold mesh. The possibility of generating a new 3D sample point on a triangle is proportional to the ratio of the area of this triangle and the total area of the mesh. This solution can be applied to arbitrary surfaces. Although the samples usually do not exactly satisfy the spatial uniformity property, this method provides a good initial sample set that can be refined by more complex sampling algorithms [7].

The quality of many sampling algorithms is often measured by the blue noise distribution. A sample set with a blue noise distribution means it has a uniform and unbiased distribution in the spatial domain, and it does not have low frequency noise and structured bias in the frequency domain. A lot of blue noise sampling algorithms have been proposed and studied by graphics researchers, such as [56, 18, 62, 74, 25, 14].

Blue noise sampling on surfaces is particularly interesting in computer graphics.

Pastor et al. proposed a point hierarchy structure for blue noise surface sampling [63]. Alliez et al. [1] and Li et al. [54] presented sampling methods with parameterizations. However these algorithms need to pre-generate data sets offline before the sampling process.

Nehab et al. used voxel grids to store samples [61]. Wei proposed a parallel dart throwing approach for blue noise sampling [74]. Nehab and Wei's approaches inspired Bowers et al., who presented a parallel Poisson disk sampling scheme that uses the GPU to sample surfaces at an interactive frame rate [7]. Fu et al. [25] and Cline et al. [14] used a geodesic distance metric for generating surface samples.

The algorithms for generating 3D samples, such as the basic sampling by triangle areas, and more advanced blue noise sampling as we discussed above, can be used for creating 3D anchors in brush based NPR. These existing algorithms produce samples on model surfaces that are evenly distributed in the spatial domain. However, we often need brush anchors that are evenly distributed in screen space for the purpose of brush stroke placement in the rendering stage. Also, most of these blue noise sampling algorithms are slow and unsuitable for interactive applications. In other words, 3D anchors generated by these existing algorithms are usable, but not ideal for brush based NPR in AR. To solve this problem, my thesis includes an algorithm for generating 3D samples on surfaces whose projections are uniformly distributed the screen space for optimized brush stroke placement.

2.2.2 Pre-built Non-photorealistic Textures

Pre-built non-photorealistic textures can also be used in the model space approach when the 3D geometry is given. These approaches create stylized NPR textures offline and map the textures to object surfaces.

Some researchers such as Horry et al. [43], Wood et al. [78], and Buck et al. [8] have built hybrid NPR/IBR (i.e., image based rendering) systems where hand-drawn

art was re-rendered for different views. These IBR based approaches directly use images created by artists to stylize the scenes for producing various rendering effects. These methods are particularly useful for VR NPR, in which we can prepare stylized textures from a set of view points offline.

Following the spirit of these IBR systems, Klein et al. used a variant of mip-map texture called an "Art Map" to render a virtual environment with coherence [48]. In a preprocessing stage, they capture photos of a real or synthetic environment, map the photos to a coarse model of the environment, and run a series of NPR filters to generate textures. At runtime, the system re-renders the NPR textures over the geometry of the coarse model, and it adds dark lines that emphasize creases and silhouettes.

The key component to achieve coherence in their algorithm is the pre-processed stylized textures. Their approach applies NPR filters to stylize each level of the mip-map texture which is the captured or synthesized image of the environment, and then maps these stylized textures onto the surfaces of the coarse 3D models during rendering. Note that the "Art Map" approach can be integrated into the particle-based algorithms, since it also applies textures to animated surfaces. In particular, it improves the coherence and visual quality when the camera zooms into and zooms out of a 3D scene. Figure 3 shows an example of the "Art Map" approach with NPR-stylized textures, which can be directly mapped on surfaces for rendering.

An advantage of this approach is the real-time speed with the use of graphics hardware. The processing of the "Art Map" can be done off-line, and the rendering still uses standard rasterization with texture mapping. This is important for interactive AR/VR applications. However, the results produced by these algorithm do not follow the process of real human artists. For example, real artists often draw brush strokes across the boundary of different objects in a painting, while the pre-built texture methods always map the stylized texture rigidly to the boundary of the object

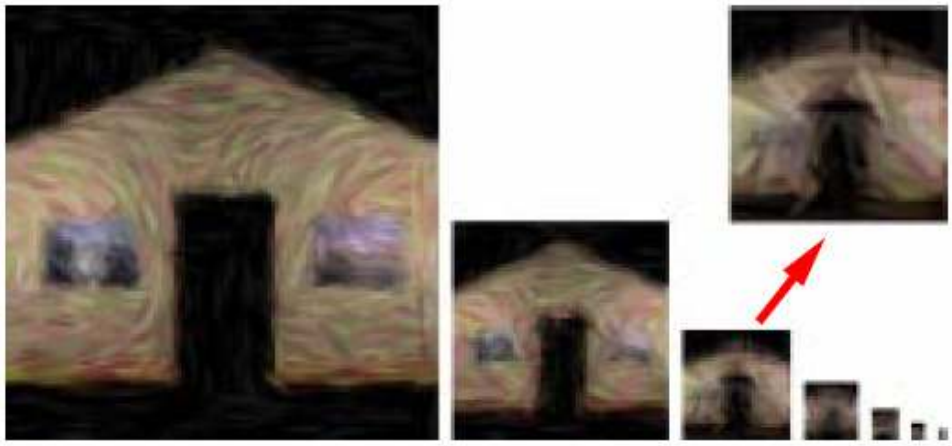

Figure 3: An example of the "Art Map". Each level of the texture is stylized, and brush stroke size is constant across multiple levels to maintain coherence. (image from [48])

geometry. Hence these algorithms do not produce faithful painterly rendering results.

Praun et al. presented a finer "Tonal Art Map" to extend Klein's method [65]. Their algorithm renders hatching strokes over surfaces. The algorithm scales strokes within the hatch images to attain appropriate stroke size and density at all resolutions. It also organizes strokes to maintain coherence across scales and tones. At runtime, hardware multi-texturing blends the hatch images over the rendered faces to locally vary tone while maintaining both spatial and temporal coherence. Figure 4 illustrates an example of the "Tonal Art Map" with a single hatching style. Note the stroke density is gradually increased from left to right, which gives inherited coherence for later texture mapping on surface.

The work described by Salisbury et al. [67] and Winkenbach et al. [75] has a similar spirit to Praun's work. They use a prioritized stroke texture, which is is a collection of strokes, that simultaneously conveys material and continuously-variable tone for NPR.

Some other researchers [58, 22, 52] have also addressed the 3D hatching problem with coherence. Markosian et al. introduced a simple hatching style indicative of a

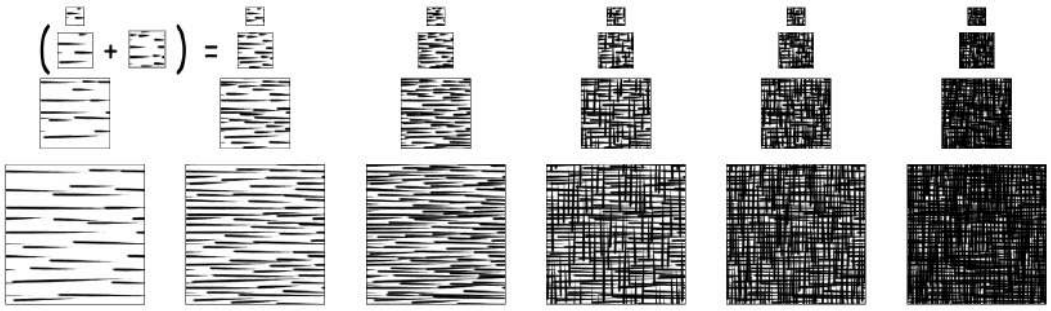

Figure 4: An example of the Tonal Art Map. Note the strokes in an image inherit all strokes to the left and top from it. (image from [65])

light source near the camera by scattering a few strokes on a surface near its silhouettes [58]. Elber showed how to render line art for parametric surfaces at interactive frame rates [22]. His algorithm maintains coherence by choosing a fixed density of strokes on the surface regardless of viewing distance. Finally, Lake et al. described an interactive hatching system with stroke coherence in image space, rather than in model space [52].

While the sketch with hatching rendering style is useful for the purpose of technical illustration in AR/VR, the speed is the bottleneck of many of existing algorithms. Also, it is a non-trivial work to prepare proper hatching textures for AR, as we do not know what will be in the future video.

2.2.3 Geometry Analysis and Brush Parameterization

We believe the two major types of approaches we explored above are typical and inspiring in maintaining coherence for many NPR styles, such as brush based painterly rendering. There are still a large number of papers that are dedicated to rendering 3D models with other NPR styles with coherence. The algorithms for maintaining frame to frame coherence in these approaches are mainly based on analysis and parameterization of brushes and object geometry. We will discuss some representative algorithms in the graphics literature below.

Some semi-automatic NPR systems need human input as the starting point for

producing stylized results for 3D models. For example, Tolba et al. [68], Cohen et al. [15], and Bourguignon et al. [5] presented NPR systems that allow an artist to simply draw in the image plane and then render models with stylization. They simplified the underlying geometry in these systems. Hanrahan et al. presented the first NPR system that allows users to paint on surfaces [33]. Their system enable designers to paint textures directly on 3D models by projecting screen space brush strokes onto the 3D surfaces and then into texture space, where they are composited with other strokes. They fix the strokes on surfaces and do not change them with lighting or viewpoint, which limits the coherence and the quality of the final rendering.

Inspired by these techniques, Kalnins et al. implemented the WYSIWYG NPR system [46]. Their system can produce a sketch and hatching style for models with the user input. The system has three categories of strokes: silhouette and crease lines that form the basis of simple line drawing, decal strokes that suggest surface features, and hatching strokes to convey lighting and tone. It allows the user to control brushes in each category with various settings to achieve different stylization.

To maintain temporal coherence, their algorithm assigns consistent parameterization to the strokes in each category. Creases are parameterized and fixed on the model. Silhouettes are view-dependent from frame to frame, so it is difficult to maintain coherence for silhouettes. They adopt a simple heuristic described by Bourdev et al. [4] for the silhouette strokes. They begin by assigning all strokes an arclength parameterization. In each frame, they sample the visible silhouettes, saving for each sample its surface location and its parameter value. In the next frame, they project the sample from 3D into the image plane. Then, to find nearby silhouette paths, they search the ID reference image by stepping a few pixels along the surface normal projected to image space. If a new path is found, they register the samples parameter value with it. Since each new path generally receives many samples, they use a simple voting and averaging scheme to parameterize it. In addition to the stroke coherence,

they also maintain hatching coherence by creating a hatch group with multiple levels of detail. Note that this is similar to the idea that we discussed about pre-built NPR textures.

Hertzmann et al. also addressed generating hatching on surfaces to convey shape and tone for 3D models [41]. Kalnins et al. proposed a new algorithm for maintaining stroke coherence by parameterization propagation [45]. Their algorithm picks evenly spaced samples from each brush path at a frame f_i. Each sample contains a triplet: its 3D location on the mesh, its brush path ID, and the paramter t to be propagated. They record the 3D position as a barycentric location in a mesh triangle. In the next frame f_{i+1}, they try to register/warp the samples from frame f_i against the silhouette paths in frame f_{i+1} to propagate the parameterization.

2.2.4 Discussion

The major types of algorithms in model space that we described above are inspiring for NPR in the AR/VR domain. In AR we have the full geometry of the virtual content, so it is possible to apply a particle based algorithm for placing anchors on surfaces for brush based NPR. However, we usually do not have the camera trajectory in AR, as it is controlled by the user. The particle sampling methods used in Meier's and follow-on work are generated on model surfaces. The projections of particles on the screen are not evenly distributed in image space. As a result, the anchors can be sparse or crowded in certain areas on the screen when the camera is in motion. Other more complex sampling schemes, such as Possion disk sampling and blue noise sampling on surfaces, are designed to generate samples that are uniformly distributed over the 3D surface. On the other hand, brush anchors that are evenly distributed in image space are important for brush based NPR methods, since we need to place brush strokes at projected positions on the screen. Hence the anchors given by these sampling algorithms are not ideal for brush based NPR. To solve this problem we will

propose a better sampling algorithm for NPR in this thesis.

The pre-built NPR textures can be directly applied to a model's surface during rendering, so it is also applicable in AR NPR. An advantage of this approach is that the texture pyramid can be built offline and the texture mapping is easy to perform using graphics hardware for real-time rendering. However, this method can only be used on models with textures. Also, this rendering process does not mimic a real artist's painting process to create faithful painting results. For example, in a brush based NPR a brush stroke can cross the boundary between surfaces, but the pre-built NPR textures always align textures to the geometric boundaries. Hence this approach is only a partial solution for AR NPR.

Perhaps the biggest drawbacks for all of these algorithms is we usually do not have the detailed information about the scene geometry in the video in the context of AR. It is possible to obtain partial information about the scene, including the orientation and position of markers, from vision-based tracking. This partial information is usually not sufficient for reconstructing the scene geometry. As a result, we cannot apply the particle based methods to find accurate anchors or warp them from frame to frame for the real world content in AR.

2.3 Image Space NPR Algorithms

In image space approaches, some researchers extended NPR techniques from still images to video and used optical flow to achieve temporal coherence [55, 40]. Some other researchers treated the video as a spatiotemporal volume and analyzed the video volume globally to obtain better coherence [16].

2.3.1 Optical Flow

Litwinowitcz first proposed the use of optical flow to improve coherence for video NRP [55]. His algorithm uses the optical flow vector field as a displacement field to move the brush strokes to new locations. Brush stroke anchors are moved to new

subpixel locations from frame to frame to maintain coherence. His algorithm used Delaunay triangulation and mesh subdivision to help add new brush stroke anchors in regions that are too sparse. He also removed brush stroke anchors in regions that become overly dense. In his implementation he chose the optical flow algorithm from Bergen's work [3]. The process of moving brush stroke anchors and adding new anchors is shown in Figure 5.

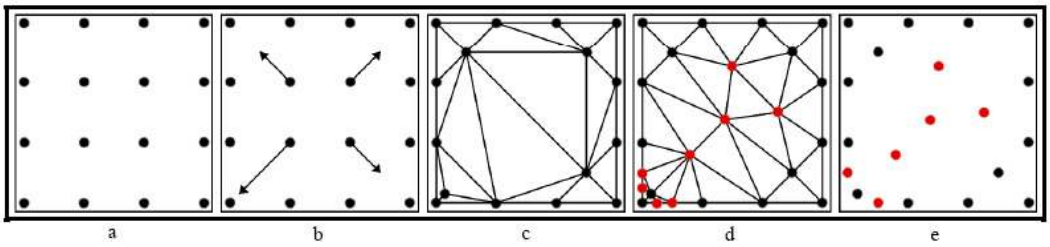

Figure 5: (a) Initial brush stroke positions shown in black dots. (b) The four middle strokes are to be moved to new locations that are computed by optical flow. (c) Delaunay triangulation of the new set of strokes. (d) Red points are new vertices that are added to maintain the brush density. (e) The updated list of brush strokes. (image from [55])

This algorithm can provide a certain degree of temporal coherence for video NPR. However, newly added strokes can appear in front of old strokes so the final result may scintillate.

Litwinowitcz's algorithm is the first approach in the literature that tries to use optical flow to maintain temporal coherence for video NRP.

Hertzmann et al. placed varied brush strokes on still images by following normals of image gradients [38]. He extended his algorithm and Litwinowitcz's work to video and also used optical flow to maintain temporal coherence [40]. This paper presented two methods for keeping temporal coherence: a crude region based paint-over method, and an optical flow based improved paint-over method. He also presented an energy function as a guide to place brush strokes on canvas, and used optical flow to warp brush stroke control points to achieve coherence [37]. Kovacs and Sziranyi proposed a similar painterly rendering technique that also uses optical flow to reduce flicker [50].

Optical flow has inspired many researchers to propose new algorithms for maintaining temporal coherence for video NPR. Many NPR algorithms involve textures, such as brush stroke textures, that simulate the appearance of various NPR effects. Bousseau et al. produced a watercolor effect for video [6]. Their method uses two different approaches to achieve temporal coherence. To preserve coherence of the watercolor texture itself, the method advects a set of pigmentation textures along lines of an optical flow field that has been computed for the video.

Hays and Essa also used an improved optical flow method to process video with different artistic styles [36]. The key difference between their approach and Litwinowicz's work is that they add more properties such as opacity to the brush strokes. Also, their approach places temporal constraints on the brush stroke properties by interpolating gradient values to improve coherence.

In general, the optical flow based NPR algorithms provide a certain degree of coherence. It is applicable for AR NPR, as it requires only the pixel information from the AR frames. The quality of coherence is decided by the underlying algorithms for estimating the pixel motion fields. The quality cannot be improved further without any advanced knowledge of the scene. Errors present in the motion field estimation can accumulate and propagate quickly to later frames. Hence the brush strokes eventually become incoherent when rendering a long sequence of frames. A possible solution to amend this drawback requires expensive manual correction, which is discussed by Green et. al. [66]. However this correction method of the motion fields is not applicable for real-time AR, since there is no opportunity for human input in rendering in AR.

2.3.2 Spatiotemporal Volume Analysis

Recently, researchers in graphics and computer vision have treated video sequences as spatiotemporal volumes by stacking sequential frames along the time-axis for analysis and post-processing. The video volume is then divided into semantic regions with coherence which can be further refined by user input. This approach improves coherence using global optimization, compared to optical flow that typically finds only pixel motion correlation between consecutive frames.

Comaniciu et al. developed a spatiotemporal segmentation technique ("kernel mean shift") for images [17]. Wang et al. proposed using "anisotropic kernel mean shift" to extend this algorithm to video [71]. The segmentation divides the video volume into pixel clusters with similar visual properties in such a way that these clusters/regions have a low spatiotemporal coherence. He also presented a semi-automatic system that can transform a video into a cartoon-like style [72]. Their approach overcomes the coherence problems by accumulating the video frames into a 3D volume and clustering pixels in spatiotemporal space. Klein et. al. also used the spatiotemporal volume in their system called "video cubism" that enables users to interactively create NPR effects [49].

Collomosse et al. improved Klein and Wang's approaches and presented a more general NPR system called the "Stroke Surface" that can create abstract animation from video with high temporal coherence [16]. A visualization of a spatiotemporal volume of a video from Collomosse's work is shown in Figure 6.

Collomosse's approach treats the video sequence as a spatiotemporal array of colors, that is a voxel volume formed by stacking sequential frames over time. There are three components in their system. A computer vision front end parses and analyzes the video. An intermediate representation provides an abstract description of the input video. A computer graphics back end synthesizes the output from the first two components and renders the video with NPR effects. The key to achieve coherence

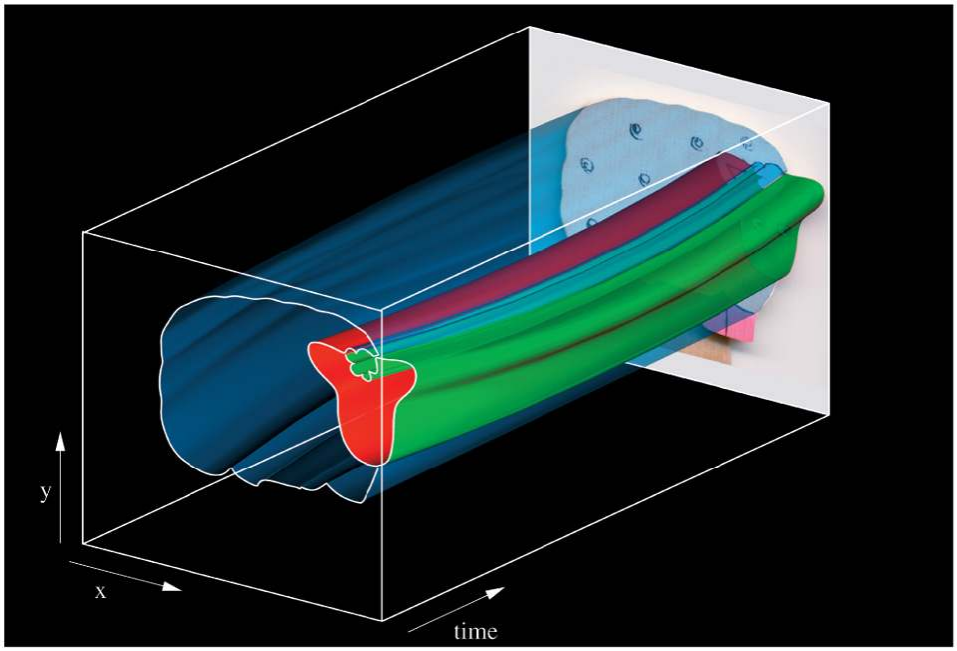

Figure 6: A visualization of the Stroke Surfaces. The sematic object (sheep) is segmented from the video sequence. The intersection between this object and the time plan generates the boundary of this object, which is described by multiple stroke surfaces. (image from [16])

in their system is to use computer vision techniques with heuristics to create semantic associations between regions in adjacent frames. The regions are recognized and connected over frames to produce video objects, as shown in Figure 6. These objects carve subvolumes through the video volume delimited by continuous isosurface patches, which are called "Stroke Surfaces". To render a frame at time t the computer graphics back end intersects the stroke surfaces with the plane $z = t$ to generate a series of splines corresponding to region boundaries in that frame.

These spatiotemporal algorithms require a thorough analysis of the whole video sequences to extract meaningful objects. The main problem of applying spatiotemporal algorithms to AR NPR is that we do not know what will be in future frames. The information we have in AR is the frames that happened before the current frame. The accuracy of sematic segmentation methods depends on the underlying computer vision techniques. This can be affected by the change of lighting conditions and other

factors in the AR video. In addition, the speed of the spatiotemporal algorithms is a bottleneck for AR NPR.

2.3.3 Image Processing Techniques for Coherence

Optical flow and spatiotemporal volumes are two major approaches in maintaining coherence for video NPR. Both of them heavily rely on the underlying computer vision and machine learning algorithms for understanding the coherence between pixels from frame to frame.

In addition to these two major approaches, a lot of researchers utilize various well-established image processing techniques to enhance coherence for video NPR. Since the techniques they employ vary and cannot be easily categorized under a general framework, we will discuss some typical and influential algorithms in this section.

Gooch et al. automatically created monochromatic human facial illustration from Difference of Gaussian (DoG) edges [27]. Inspired by their work, Winnemöller et al. proposed a real-time video abstraction scheme [77]. They define DoG edges using a smoothed step function to increase temporal coherence in animation. This version of DoG edge detector produces more robust and coherent edges in video stylization. They also perform an optional color quantization step on the video frames to create a cartoon or paint-like effects. They argue that their quantization has a significant advantage in temporal coherence. In the standard quantization a small luminance change can push a value to a different bin, thus causing a large output change for a small input change. This introduces flickering, especially for noisy input. Their soft quantization method spreads changes over a larger area, making it less noticeable. Their gradient-based sharpness control can further minimize the impact of sudden changes in low-contrast regions.

Image processing techniques for use in coherence is limited in the domain of AR. Similar to the spatiotemporal algorithms, most of the image processing algorithms

need to know the entire video sequence in advance. The important parameters in these algorithms, such as the thresholds for edge detection and color quantization, are very difficult to decide in interactive AR applications.

2.3.4 Discussion

The image space algorithms we discussed above usually cannot be directly applied to the AR/VR domain. There are several reasons. First, in real-time AR we do not have information or knowledge beyond the current frame. We can only access the information of frames that are before the current frame. On the other hand, the spatiotemporal volume algorithms and their variations heavily rely on the underlying computer vision and machine learning techniques to analyze the content of the entire video sequence. They usually need to go through the entire video sequence, and then find the segmented objects in frames for later coherence processing.

Second, AR requires real-time performance. Most of the existing image space algorithms are designed to process the video offline. They are not usually executed at interactive frame rates. The characteristics of AR make it difficult to extend these existing image space algorithms into the AR domain.

2.4 NPR Algorithms in AR/VR Areas

NPR is an important approach to blend real-world video with computer generated graphics content. This can be used in many AR applications, and as such, it has attracted researcher's attention in the AR/VR community. Quite a few NPR algorithms have been proposed and studied.

Perhaps one of the earliest pieces of works in AR NPR was done by Fischer et al. in 2005 [23, 24]. For the first time they argued that stylizing both the virtual and real content is an important alternative for creating user immersion in AR. They presented several NPR styles, such as a cartoon-like style, for AR applications. They leveraged the power of the GPU to detect strong edges in AR frames, and blend colors

to create a cartoon-like stylization for AR. They used GLSL to achieve a interactive frame rate in rendering.

Haller et al. compared photorealistic rendering and NPR for AR [30, 32]. He also presented a sketch style rendering for AR [31]. Their algorithms directly apply the NPR methods designed for still images to image sequences in video. As a result the final rendered videos appear to flicker.

2.4.1 Discussion

Not much research has been done for maintaining temporal coherence for real-time AR NPR. A typical frame from an AR video is a mixture of a computer synthesized image and a real world background. However, the algorithms in the image space and model space approaches discussed above are usually not directly applicable for AR applications.

There are several differences between NPR for AR and NPR for video and 3D models. First, AR applications usually require real-time performance. Second, we do not have any information beyond the current video frame in AR. Third, we usually do not have detailed information about the scene geometry in the video. Fourth, we do have complete information about the virtual content, and partial spatial information of the scene from tracking. These unique characteristics of AR imply that existing video NPR techniques cannot be easily extended into the AR domain.

CHAPTER III

WATERCOLOR INSPIRED NON-PHOTOREALISTIC RENDERING FOR AUGMENTED REALITY

In this chapter we present a watercolor-like rendering technique for AR. This algorithm uses the Voronoi Diagram to tile AR frames and produces a watercolor-like style in the final rendering results. The coherence is achieved by re-tiling Voronoi cells along the detected edges in each frame. This algorithm is an image space method for maintaining the coherence of the rendered AR video.

3.1 Algorithm Overview

Water-colorization for static images and videos is an interesting topic with a long history. Curtis et al. uses an ordered set of translucent glazes to model watercolor effects [21]. They use a Kubelka-Munk compositing model to simulate the optical effect of the superimposed glazes. The method operates on each frame independently. Bousseau et al. proposes a method for maintaining temporal coherence in watercolor-like video [6]. Their method employs texture advection along lines of optical flow to maintain texture coherence, and mathematical morphology to maintain abstraction coherence. All these past methods have focused on offline processing of a complete video sequence, but are too time-consuming for online rendering in a live AR video system.

The idea of using Voronoi diagrams is inspired by several NPR papers [35, 2]. Those papers use Voronoi diagrams to create various non-rigid rendering styles, such as the digital mosaic style for images and videos. The Voronoi diagram can also be used to tessellate surfaces [53]. We found that the 2D Voronoi diagrams can be

effectively used to produce a watercolor inspired effect for live AR video in real time. This approach creates a non-rigid color bleeding along the edges of objects in the video. Also, by re-tiling Voronoi cells along edges detected in the video frame, it keeps a certain degree of visual coherence in the output video.

The algorithms in this approach can be divided into two parts: creating a watercolor-like style using a 2D Voronoi diagram, and keeping visual coherence by detecting strong edges and re-tiling the Voronoi cells. In the first part, an AR video frame is processed using a Voronoi pattern to produce the color bleeding effect. In the second part, Voronoi cells are re-tiled to keep visual coherence in each independent video frame. Our algorithm uses the GPU to efficiently process the frame in shaders, so this method is applicable for real-time high resolution AR live videos with a modern graphics card.

3.2 Watercolor NPR Algorithm Descriptions

3.2.1 Using Voronoi Diagrams to Tile the Image

Given an AR image that is a blend of video and computer-generated objects, our first task is to create a tiled version of this scene. We begin by creating a tiling of the plane with a Voronoi diagram, and then we color these tiles based on the original image. We use a jittered sampling of the plane in order to create the centers of the Voronoi cells for our tiling. Assume the size of the video frame is $W \times H$, and that we will create $m \times n$ tiles. The jittered center of the Voronoi cell at the i-th row and j-th column is then chosen as follows:

$$V(i,j) = \left(\frac{W}{n}\left(j + \frac{1}{2} + R_x\right), \frac{H}{m}\left(i + \frac{1}{2} + R_y\right)\right) \tag{1}$$
$$0 \leq i < m, 0 \leq j < n, -\frac{1}{2} < R_x, R_y < \frac{1}{2}$$

To create the Voronoi cells, we use the observation that a Voronoi diagram can be created by placing a cone at each of the cell centers [35]. The portion of a given cone

that is closer than all other cones delineates one of the Voronoi cells. Using graphics hardware, these cones are approximated by collections of polygons, and the closest portions of the cones are determined by depth buffering. We rasterize $m \times n$ cones at the centers $V(i, j)$ with depth buffering enabled in orthographic projection. For example, we render 80×60 cones for a 640×480-pixel frame. During rasterization, each cone is given a unique RGB color so that the region of each cone can be identified by examining the pixel colors from the framebuffer. The framebuffer that we read back is divided into $m \times n$ Voronoi regions by the Voronoi cells. Figure 7 shows a low resolution version of what the frame buffer looks like after the cones have been rasterized. The screen is divided into many irregular hexagon-shaped Voronoi regions. The black dot in each region is the Voronoi center, which is also the vertex of the corresponding cone. Note the shape of Voronoi regions in this method depends on the locations of the cell centers. We can obtain rectangle-shaped Voronoi regions if the cell centers are located at the grid centers. In our tiling method, we choose to place the Voronoi cell centers at the jittered grid centers to obtain the hexagon pattern, since it looks less rigid and produces a more natural color bleeding effect that mimics the human artist's work.

The index of the Voronoi region for pixel (i, j) is decoded from its RGB color, since we render each cone with a unique color. These regions form a Voronoi pattern that we then use to create a watercolor-inspired style for the AR video frame. To produce the final tiled image, we color each Voronoi cell according to the average color of the original image within the cell. Figure 8 shows an original AR video frame, and the frame stylized using such a Voronoi tiling. Note that others have used Voronoi tilings to create mosaic rendering styles for single image off-line non-photorealistic rendering [35, 2].

Many watercolor paintings include dark strokes that highlight object silhouettes. In the next step we mimic this style by detecting and drawing strong edges in the

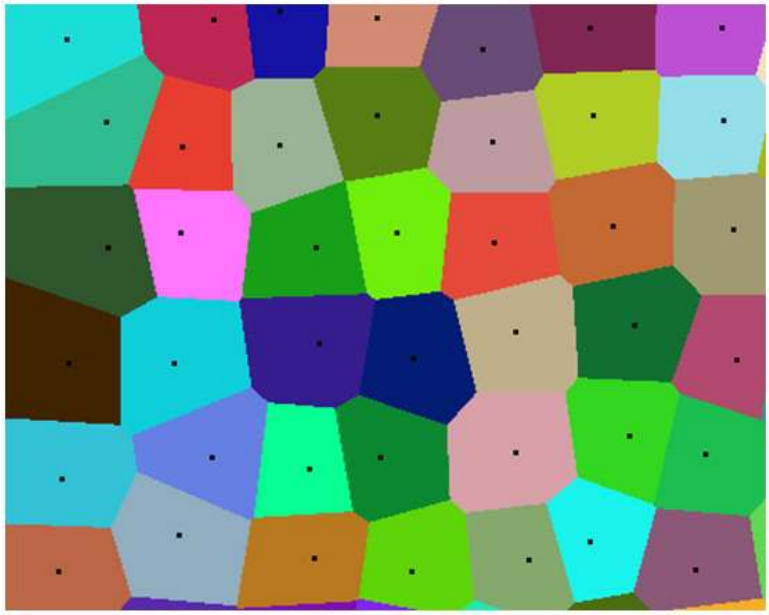

Figure 7: A Voronoi pattern in the frame buffer. We have used different colors for each region in this image to make the regions easy to see. In the actual algorithm the color of each region is determined by its region index k.

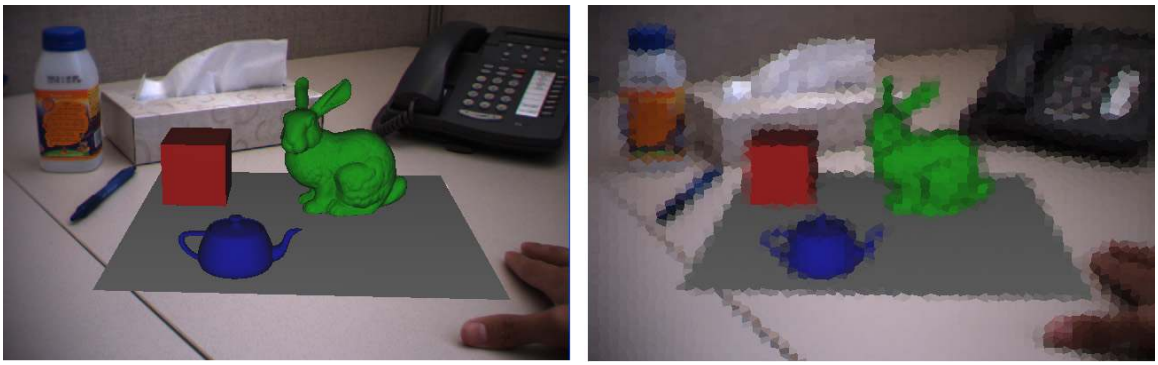

Figure 8: Left: original AR frame. Right: the AR frame tiled by a Voronoi pattern.

video frame. By combining these two steps (tiling and edge drawing) we produce a watercolor inspired NPR style for augmented reality.

3.2.2 Fast Edge Detection in an OpenGL Shader

Edge detection is a common technique for NPR rendering, since many NPR styles highlight silhouettes and strong edges. For example, Fisher et al. uses YUV color space to detect strong edges in AR video frames [23].

In our algorithm, we first render the virtual objects using a simple toon shader, and later detect edges in this AR frame. The reason that we use a simple toon shading instead of default OpenGL Gouraud shading is that we want to exaggerate the color difference between objects so we can get better edge detection results.

Our toon shader performs intensity thresholding to create just three discrete surface intensities instead of using continuous shades for an object's surface. The code of our fragment shader for toon shading is given below.

```
varying vec3 lightDir, normal;
uniform vec3 myColor;
void main()
{
    vec4 color;
    vec3 n = normalize(normal);
    float intensity = clamp(dot(lightDir, n), 0, 1);
    if (intensity > 0.7)
         color = vec4(myColor, 1.0);
    else if (intensity > 0.2)
         color = vec4(0.4 * myColor, 1.0);
    else
         color = vec4(0.1 * myColor, 1.0);
    gl_FragColor = color;
}
```

The virtual objects and the video background are rendered and combined to a texture. Edges are detected in a fragment shader in the GPU so we do not need to copy the pixel data of this texture to the CPU. The pixel information in both of RGB and YUV color space are used to detect edges. The pixel conversion between these two color spaces can be performed in the fragment shader as shown below.

```
mat4 RGBtoYUV;
vec4 rgb = texture2D(tex, texcoord);
vec4 yuv = RGBtoYUV * rgb;
```

The equation for edge detection is shown below. $\bigtriangledown$ is the color difference along the x-axis and the y-axis. If this value is larger than a threshold then this pixel is considered to be an edge pixel.

$$Edge_{YUV} = (1-\alpha)\cdot|\bigtriangledown_Y| + \alpha\cdot\frac{|\bigtriangledown_U| + |\bigtriangledown_V|}{2} \quad (2)$$

$$Edge_{RGB} = \frac{|\bigtriangledown_R| + |\bigtriangledown_G| + |\bigtriangledown_B|}{3} \quad (3)$$

$$EdgePixel = \begin{cases} \text{Yes} & \text{if } \beta\cdot Edge_{YUV} + (1-\beta)\cdot Edge_{RGB} > threshold \\ \text{No} & \text{if } \beta\cdot Edge_{YUV} + (1-\beta)\cdot Edge_{RGB} \leq threshold \end{cases} \quad (4)$$

An AR video frame with edge detection and its final rendering by combining the two steps are shown in Figure 9. The silhouettes are detected for three virtual objects (the teapot, the cube and the bunny) and for the objects in the real world.

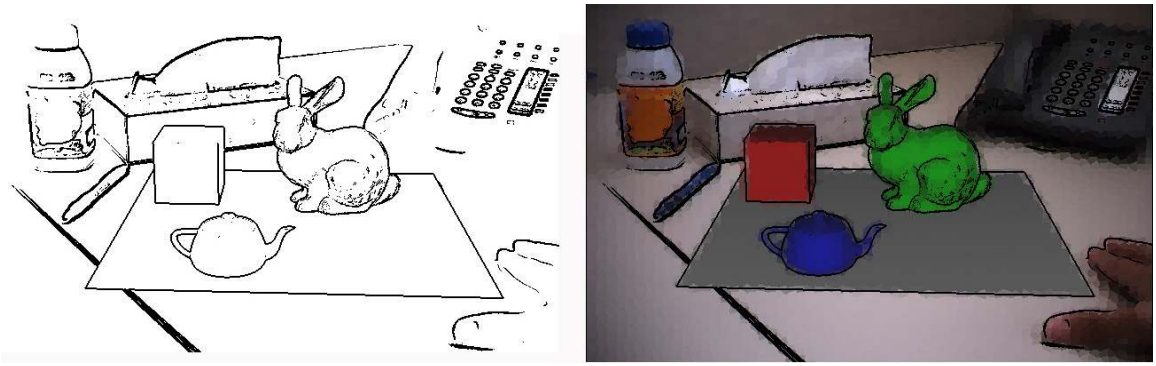

Figure 9: Left: detected edges in the AR frame. Right: final watercolor-inspired stylization of the AR frame.

3.2.3 Re-tiling Voronoi Cells for Coherence

Visual coherence is a key challenge in video NPR processing. Many papers have studied this topic. Bousseau et al. uses advection texture to keep temporal coherence in watercolor rendering for a general video [6]. However, their method is too expensive

to use for real-time AR video. We use the information in image space to provide coherence support for the watercolor rendering style.

We keep visual coherence by re-tiling Voronoi cells along the strong edges in the scene. If we keep the same Voronoi pattern and apply it to each video frame all the time, the output video would have an undesired "shower door" effect. The underlying Voronoi pattern would remain static in the rendered results. To avoid this problem we re-tile the Voronoi cells along the strong edges in each video frame, thus generating a new Voronoi pattern. Our approach is to pull each Voronoi cell *towards* strong edges in the image. This gives the appearance of color bleeding between regions since the cells will straddle two regions that are separated by an edge. Note that this is in some sense the reverse of Hausner's technique of having Voronoi cells aviod strong edges [35]. Our procedure is as follows:

1. Go through each Voronoi region to compute the average center of all edge pixels in that region. This average edge pixel location becomes the new center for a cell. If the number of edge pixels is smaller than a threshold in a region, then keep the previous center for that region.

2. Move the Voronoi cells to the new centers and rasterize the cones again.

3. Read back the frame buffer and use it as a new Voronoi pattern.

Figure 10 gives an example of re-tiling a Voronoi cell to its new center.

Ideally we want to generate a new Voronoi pattern for each video frame, but this is an expensive procedure since we need to read back the frame buffer from GPU to CPU by using `glReadPixels`. To balance the speed and quality of the output AR video, we do this re-tiling periodically, re-tiling the Voronoi cells and generating a new pattern every 10 frames. We are investigating the possibility of performing the tiling entirely on the GPU, which should allow us to generate a new tiling pattern every frame.

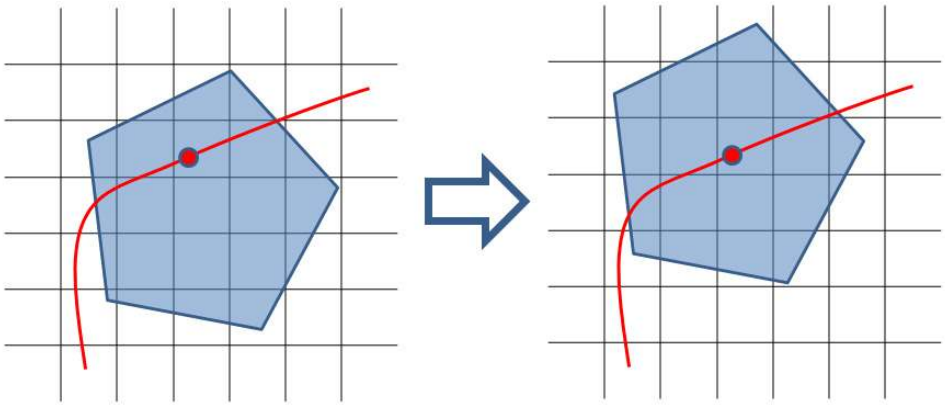

Figure 10: Move a Voronoi cell to the average center (the red dot) of all strong edge pixels in the Voronoi region.

3.3 Implementation Details

The workflow of our NPR renderer is shown in Figure 11.

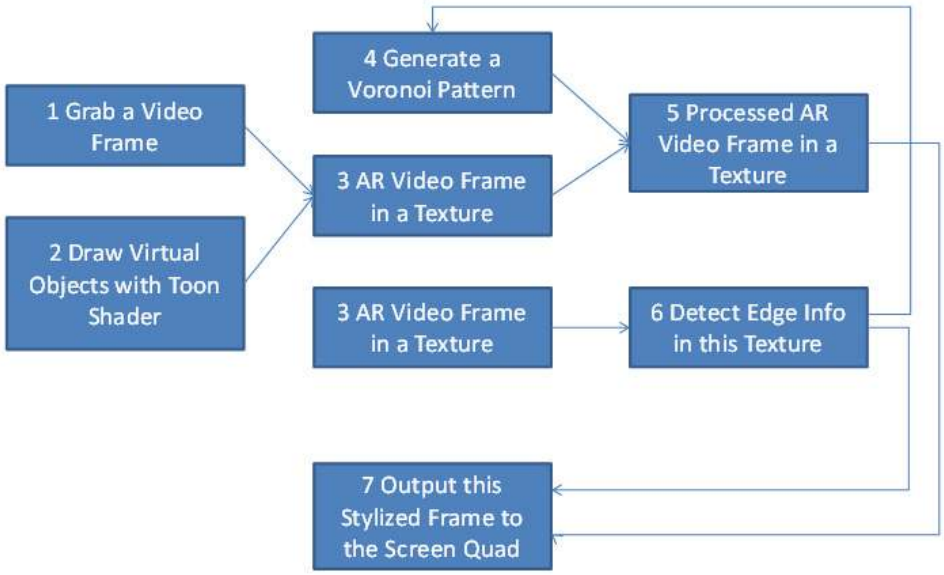

Figure 11: Workflow of the watercolor style rendering.

We used OpenGL extensions (ARB functions) and the GLSL shading language to accelerate the rendering in the GPU. All intermediate results are rendered and manipulated as textures. By directly manipulating the texture in shaders we eliminated the expensive data copy between the GPU and the CPU.

We created a Frame Buffer Object (FBO) with two frame buffers. The content

of each buffer is linked with a texture ID and it can be directly used as a texture. As shown in Figure 11, most steps are done by manipulating these two textures. Shaders are used to render the virtual objects and to detect edges. The steps in the rendering procedure are listed below. The numbers given in parentheses are shown in the workflow graph.

1. Generate an initial Voronoi pattern (4).

2. Fetch a frame from the video camera and render the video background and virtual objects (1, 2) to the first frame buffer in the FBO to generate the original AR frame as a texture (3).

3. Copy this texture from the first frame buffer to the second frame buffer in the FBO (3).

4. Use `glMapBuffer` to access (3) and modify the pixels based on the Voronoi pattern in (4), and place the processed AR video frame into a texture (5).

5. Output the stylized frame to the quad. This process is done in a fragment shader. The input to this shader are two textures, one containing the tiled image and another with the edges. The shader draws the edges on top of the tiled image.

6. Re-tile the Voronoi cells based on the detected edges.

The most expensive step is (5) in the workflow graph. This step computes the average color of all Voronoi regions in the current pattern and assigns the color to each pixel in the current frame. Also, this is the only step that can stall the rendering pipeline because it calls `glMapBuffer`. It would be possible to accelerate this by reading only a portion of the pixel buffer, processing the pixel data in a separate thread, and then reading another portion of the pixel buffer. This should increase

the speed when the pixel buffer is large. We did not resort to this in our system since our video resolution is relatively small (640×480).

Another possible way to eliminate the bottleneck of region coloring at higher resolutions is to perform the non-regular color averaging using two textures (the Voronoi pattern texture and frame texture) on the GPU. The algorithm has the following steps:

1. Generate the Voronoi pattern.

2. Use two input textures in the fragment shader: the current AR frame texture and the Voronoi pattern. For each pixel p, search its 16×16 neighbors in the Voronoi pattern texture and record the positions (texture coordinates offsets) of the neighbors that have the same color as the current pixel p.

3. Compute the average color of the pixels in this list of positions in the AR frame texture and assign this color to p in the final output image.

Note this approach is similar to performing image convolution on the GPU. The difference between our approach and the typical convolution is that in our case each pixel has a unique kernel (i.e., its value and size varies from pixel to pixel), and kernels for a same pixel change over time, where the typical convolution uses a constant kernel for all pixels at all time. To implement this method in GPU fragment shader, we find that we need to use a single loop and limit the number of searched neighbors. Otherwise the shader may have too many operators and cannot be compiled successfully for the GPU. This is an important step to achieve real-time performance for live AR video with higher resolution. The fragment shader is implemented as follows.

```
1 uniform sampler2D voronoi_pattern;
2 uniform sampler2D ar_frame;
3 varying vec2 texcoord;
4
```

```
5  void main()
6  {
7      int max_size = 16;
8      vec3 avg_color = vec3(0.0, 0.0, 0.0);
9      vec3 ctr = tex2D(voronoi_pattern, texcoord).rgb;
10     int col, row;
11     int count = 0;
12     for (int i = 0; i < max_size * max_size; ++i)
13     {
14      row = i / max_size;
15      col = i - row * max_size;
16      row -= max_size >> 1;
17      col -= max_size >> 1;
18      vec2 offset = vec2(col * 1.0/512.0, row * 1.0/512.0);
19      if (tex2D(voronoi_pattern, texcoord + offset).rgb == ctr)
20      {
21          ++count;
22          avg_color += tex2D(ar_frame, texcoord + offset);
23      }
24     }
25     avg_color /= (count + 1);
26     gl_FragColor.rgb = avg_color;
27     gl_FragColor.a = 1.0;
28 }
```

Figure 12 illustrates an example of averaging pixels in a non-rectangular Voronoi cell using GLSL.

Our system runs on a Windows PC with Xeon 2.2GHz CPU and an nVidia G7950 graphics card. This graphics card supports OpenGL extensions (e.g., Pixel Buffer Object and Frame Buffer Object) and GLSL. We use a PointGrey flea camera, which provides a crisp and bright video that is superior to Webcam videos. Pictures in this

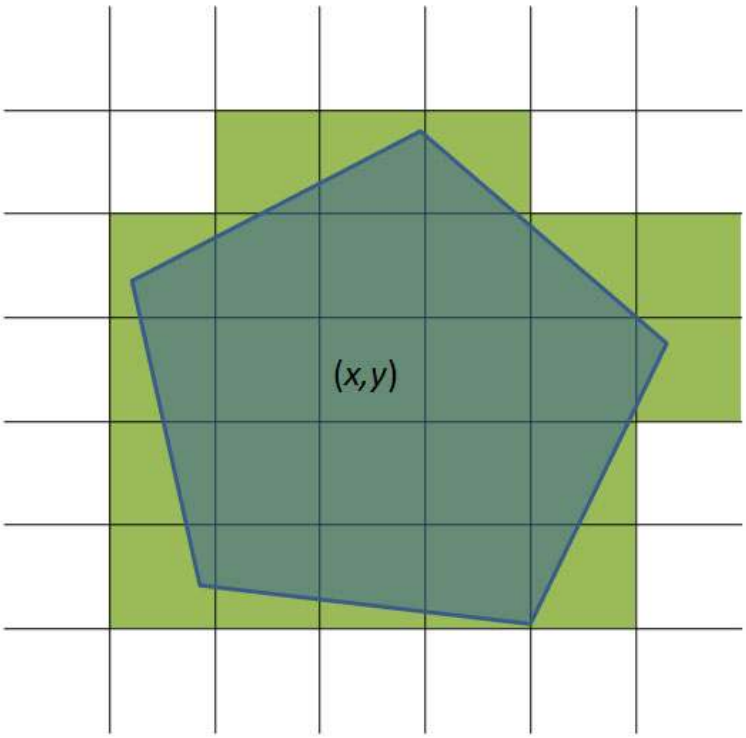

Figure 12: Averaging pixels in a Voronoi cell with GLSL. The blue area is a non-rectangular Voronoi cell. The green area is the pixels in the AR frame that are covered by the rasterized cell. The input of the fragment shader is the Voronoi pattern texture and the current AR frame texture. The output color of a pixel in a Voronoi cell is the average color of all pixels in this cell (the green pixels).

chapter are captured from a live AR video running in real-time. The video resolution is 640×480 and the average frame rate is over 15fps. Our algorithm uses the GPU to effectively process the image in shaders, so the method is applicable for higher resolution AR videos with a fast graphics card.

3.4 Results

Figure 13 shows several screen shots from a live AR video processed by our watercolor algorithms. The algorithm is running at real-time speed for stylizing the AR video.

This work has been published in VRST 2008 [10].

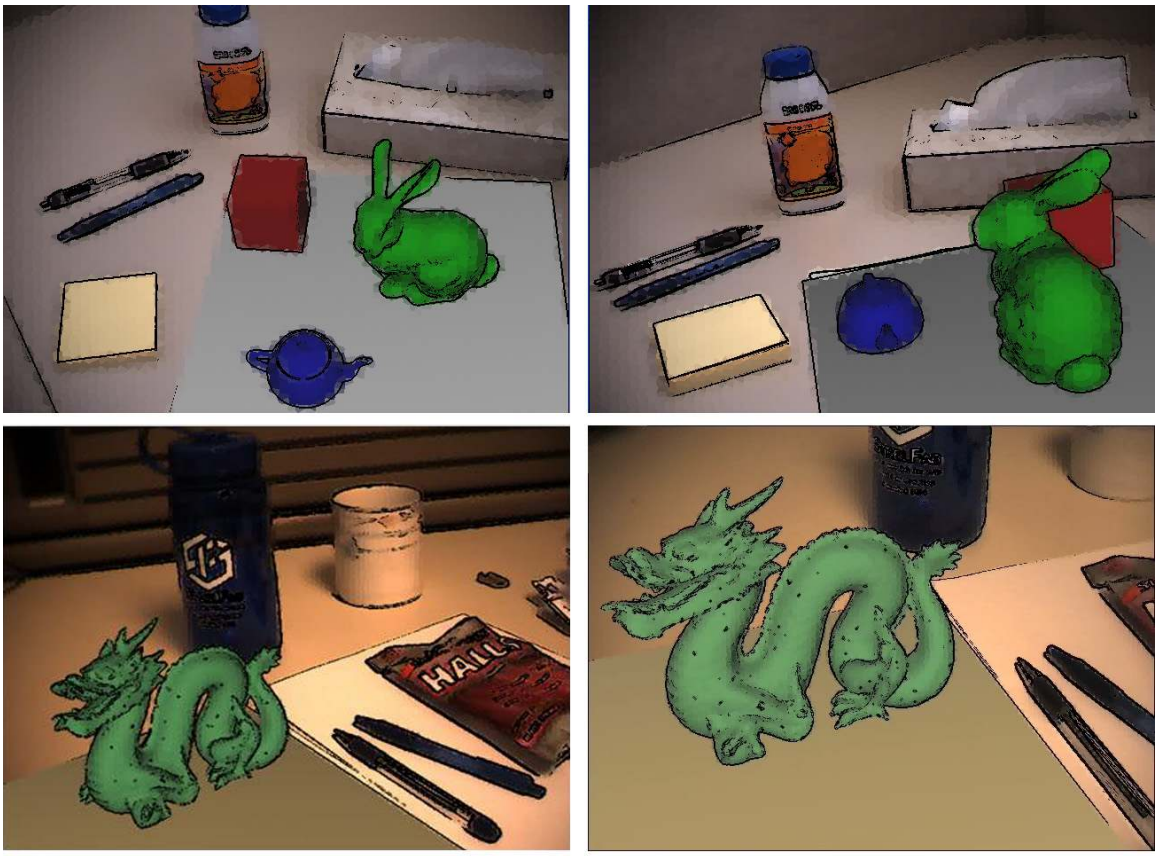

Figure 13: Screen shots from a watercolor stylized AR video. Top row: three virtual objects (a teapot, a cube and a bunny) standing on a real table. Bottom row: A virtual dragon standing on a real table.

3.5 Discussion

We have demonstrated the use of Voronoi diagrams to create a real-time watercolor inspired style for live AR video. We achieve visual coherence by detecting strong edges and re-tiling the Voronoi cells along these edges. It would be possible to use simulated paper and canvas textures as background for our algorithm, as done by [19].

This NPR algorithm provides temporal coherence by re-tiling Voronoi cells along detected strong edges of the AR frames. The success of re-tiling relies on the underlying edge detection algorithms. The coherence is lost if an edge is lost in detection in a new frame. The accuracy of the edge detection is restricted by the video quality and can be affected by exposure and lighting change in the scene. Hence the coherence in

this algorithm is limited.

CHAPTER IV

PAINTERLY RENDERING WITH COHERENCE FOR AUGMENTED REALITY

Painterly rendering is an important category in the NPR literature. Such algorithms can be extended to many brush based stylizations.

In this chapter, we propose a painterly rendering algorithm that demonstrates significant coherence for AR. This algorithm is an image space method for maintaining coherence.

This algorithm creates a painterly rendering effect for AR video by generating a tensor field and placing bump-mapped curly brush strokes in the resulting frames. It uses the correspondences of feature points across frames to warp and reshape the brush strokes to maintain the temporal coherence. We choose the painterly rendering style in our system, but our approach of maintaining temporal coherence can be applied to other NPR algorithms (e.g., a sketching rendering style with hatching textures).

4.1 Algorithm Overview

The flowchart of our painterly rendering algorithm is shown in Figure 14.

We use multiple layer painting to compose the final result. In the tensor field creation, we divide the input frame into regions and label each region as "highly-detailed", "middle-detailed" and "texture-less" based on each region's overall edge magnitude. We use different sets of properties for brush strokes to paint these regions from the coarse layer to fine layer. During the painting, we allow a brush stroke from a coarse layer to cut across regions in the fine layer, but do not allow this in the opposite direction.

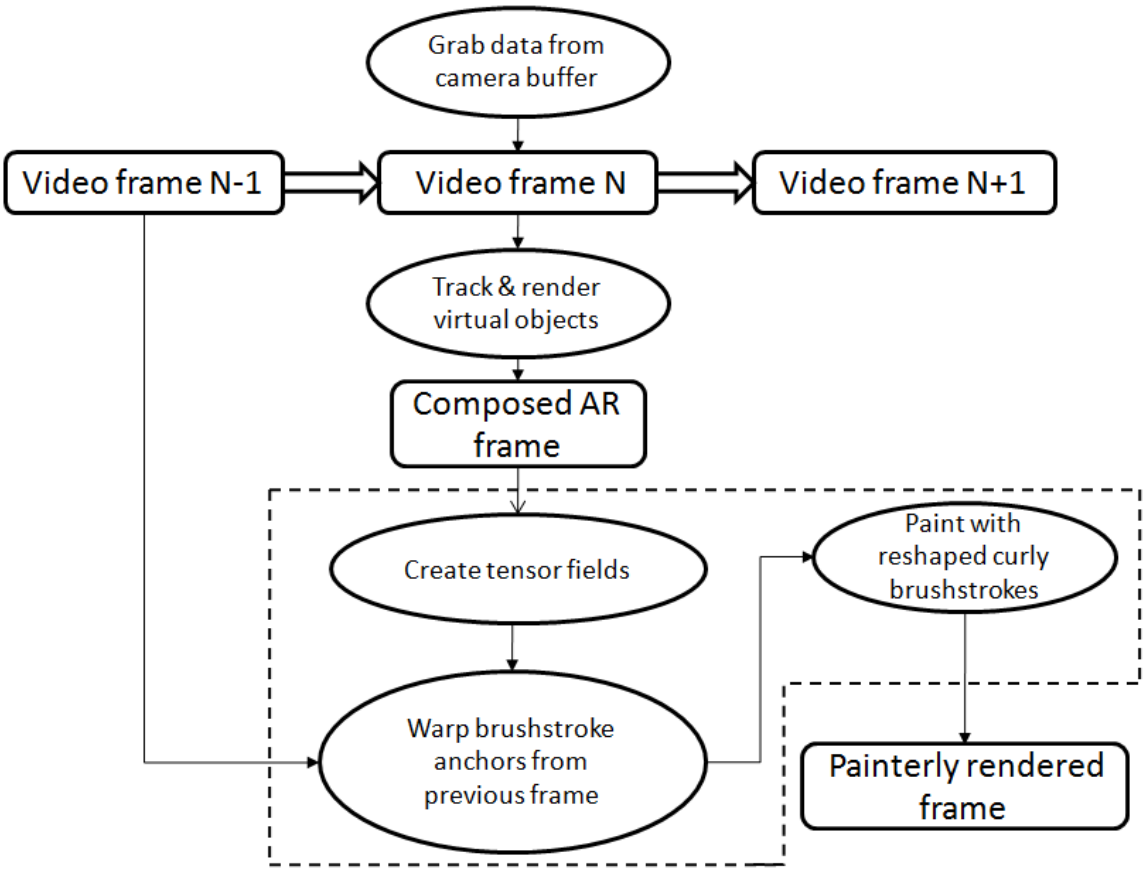

Figure 14: Flowchart of our painterly rendering algorithm for AR. Three major steps are enclosed by the dashed lines.

The final painting result is created by placing bump-mapped curly brush strokes in an AR frame. The placement of brush strokes is guided by tensor fields of each frame. Temporal coherence is maintained by warping the brush stroke anchors and reshaping the brush strokes from the previous frame to the current frame. We track feature points across frames, and brush stroke anchor warping is done by using the three nearest neighboring feature points in barycentric coordinates. To our knowledge this is the first attempt to achieve coherence of painterly rendering for AR.

4.2 Tensor Field Creation

To guide the orientation of the brush strokes, we first compute the tensor field on an AR frame. One of the major advantages of a tensor field over a vector field is it

allows direction ambiguity. The tensor field at a pixel does not distinguish between the direction of θ and $\theta + \pi$ [79].

In the vector field, the magnitude P and orientation θ of a pixel can be computed by the following image gradient operator:

$$\begin{aligned} G_x &= 2I(x+1,y) - 2I(x-1,y) + I(x+1,y+1) \\ &\quad -I(x-1,y+1) + I(x+1,y-1) - I(x-1,y-1) \qquad (5) \\ G_y &= I(x-1,y+1) + 2I(x,y+1) + I(x+1,y+1) \\ &\quad -I(x-1,y-1) - 2I(x,y-1) - I(x+1,y-1) \qquad (6) \\ P &= \sqrt{G_x^2 + G_y^2} \qquad (7) \\ \theta &= \arctan\left(G_y/G_x\right) \qquad (8) \end{aligned}$$

$I(x, y)$ is the gray scale intensity of a pixel at (x, y).

Note that many other edge detection algorithms in computer vision literature, such as the Harris corner detector [34], will also work. In our experiment, this operator from Canny edge detection is relatively fast and produces a good enough initial tensor fields. Once the edge magnitude P and orientation θ of a pixel is computed, its tensor value M is defined as follows:

$$M = P \begin{bmatrix} \cos 2\theta & \sin 2\theta \\ \sin 2\theta & -\cos 2\theta \end{bmatrix} \qquad (9)$$

The eigenvector of this matrix M's major eigenvalue gives the orientation θ for this pixel. As mentioned above, this representation allows for sign ambiguity. The tensor field has been used in scientific visualization [79] and video processing [44]. Some researchers also use the idea of the tensor field for different applications, such as street modeling [9].

Alternatively, we can define the following 2×1 vector:

$$v = \begin{bmatrix} P & \theta \end{bmatrix}^T \tag{10}$$

at each edge pixel. Furthermore, we can omit the magnitude component P and use only the angle θ at each edge pixel to create a scalar field, since the direction information is all that is needed by our algorithm. However, both the vector field and the scalar field of angles are directional, while the tensor field representation supports sign ambiguity, a major advantage of the tensor field. As a result, the linear interpolation of the vector field and the scalar field may result in singularities as intermediate values pass through zero. It is possible to avoid this problem by always capping the angle θ to the range $[-\pi, \pi]$ when we interpolate the values in the vector field. However this is not an elegant solution, with the resulting algorithm needing conditional statements to compute the new angle. On the other hand, the tensor field representation allows us to interpolate new tensor values simply and elegantly, as *a linear combination of tensor values*. We can also apply filters (such as a Gaussian filter) to the tensor values by convoluting the filter kernel and the tensor values directly, without worrying about the sign ambiguity of the angle θ. This benefit becomes more important as we construct and use the tensor pyramid later in this section.

To reduce the image noise, we compute a threshold on the pixel edge magnitude P. In our system, we choose the threshold value as $\frac{1}{3}$ of the maximum P of all pixels in a frame. It should also be possible to use the intensity histogram of P to find a proper threshold that separates the edge pixels and non-edge pixels. After thresholding, we have a tensor field map associated with the video frame. Each strong edge pixel whose magnitude is above the threshold has been assigned a tensor field value M.

For the pixels whose magnitude is below the threshold, we can use global radial basis interpolation to compute their tensor values [36]. A linear interpolation of a vector field usually generates singularity points, where the magnitude of the vector

field becomes zero if two nearby vector values have opposite directions. Tensor fields can lessen this problem because the interpolation of two tensor values with opposite orientations does not produce zero. The interpolated tensor field value for a non-edge pixel is computed as follows:

$$M = \sum_{i \in \text{edge pixels}} e^{-\frac{d_i}{\sigma}} P_i \begin{bmatrix} \cos 2\theta_i & \sin 2\theta_i \\ \sin 2\theta_i & -\cos 2\theta_i \end{bmatrix} \tag{11}$$

The exponential component is the weight that controls the contribution of a strong edge pixel P_i to the tensor value. d is the Euclidean distance between an edge pixel $P(x_i, y_i)$ and the non-edge pixel $P(x, y)$. σ controls the width of the attenuation window. Small σ makes the tensor values smoother, but it is possible to lose local details. The linear global interpolation method and its variations need to loop through all strong edge pixels to compute the weighted sum.

Usually this interpolation becomes a computational bottleneck if there are a large amount of edge pixels initially detected. In practice, it may also give a zero tensor value for non-edge pixels due to large textureless regions without strong edges. For example, if the distance between a non-edge pixel and all edge pixels is too big, its tensor value M will be almost zero. In the final painting these unassigned regions become "holes" that are not covered by any brush strokes. The zero tensor problem may be avoided by normalizing the tensor value by the sum of the weights as follows:

$$M = \frac{\sum_{i \in \text{edge pixels}} e^{-\frac{d_i}{\sigma}} P_i \begin{bmatrix} \cos 2\theta_i & \sin 2\theta_i \\ \sin 2\theta_i & -\cos 2\theta_i \end{bmatrix}}{\sum_{i \in \text{edge pixels}} e^{-\frac{d_i}{\sigma}}} \tag{12}$$

However, normalization does not solve the computational problem of looping over all strong edge pixels. In practice, this quickly becomes a computational bottleneck in many scenarios. To avoid this bottleneck, we use a tensor pyramid structure

(described below) that limits the computation cost, since only a convolution with a Gaussian filter kernel is performed per region in each level. This convolution is done by a linear weighted combination of tensor values. The weights are from the Gaussian filter kernel, which also eliminates the expensive computation cost of the exponential weights $e^{-\frac{d_i}{\sigma}}$ in the global radial basis interpolation. Also, since the initial tensor field of a frame is usually non-uniformly distributed, a region that has a dense group of edge pixels (e.g., a tree) may strongly bias the interpolated results. The tensor pyramid approach avoids this bias.

To address these problems in the global radial basis interpolation, we propose a tensor pyramid for computing the tensor values for an AR frame. Our idea is inspired by the work from Gortler et al. [28]. In their paper an image pyramid is built and a pull-push algorithm is used to interpolate the sparse lumigraph data. Similarly, we build a pyramid of the tensor field. The steps of the algorithm are given below.

1. Divide the original video frame to grid-based regions, and sum and normalize the tensor values in each region.

2. For a region that is not assigned a tensor value, use radial basis interpolation to compute its tensor, but search only a small 5×5 window patch centered at this region.

3. Apply a low-pass Gaussian filter to the tensor values, and then sub-sample to create a new tensor field at a higher level.

4. Repeat Step 2, until we have reached the top of the pyramid and built the tensor field of size 1×1.

Note the difference between our method and Gortler's original interpolation method in step 2. Gortler's original method averages and normalizes data in regions at each level. In our method, at each level of the tensor field pyramid we first search a 5×5

window centered at a region to compute its tensor field. We then apply a Gaussian filter to create a new higher level of the tensor field. The purpose is to keep fine local details for later painting. Local tensor values near to a strong edge should follow the direction of that edge. This makes the brush strokes in the area look smoother. Figure 15 illustrates the creation of the tensor field pyramid.

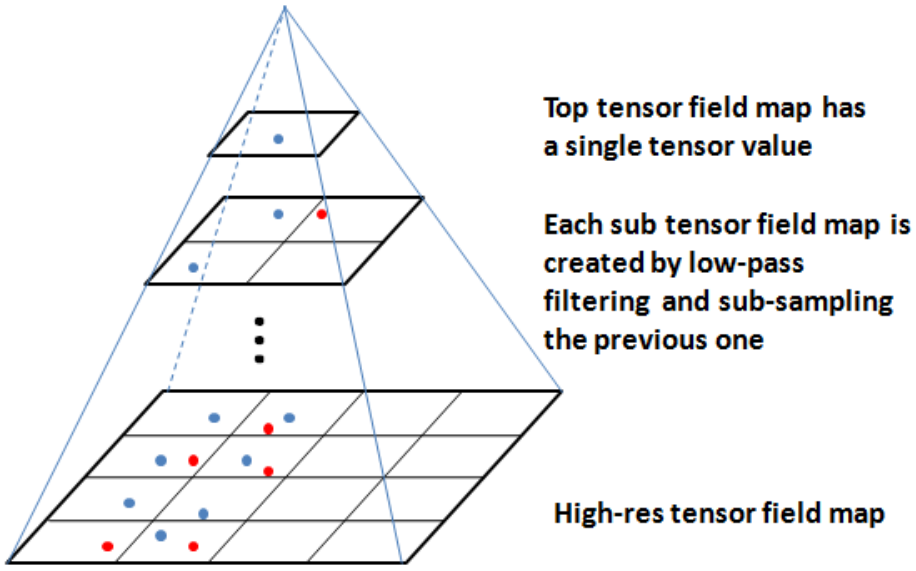

Figure 15: Three levels of the tensor field pyramid, created from bottom to top. At each level, blue dots are the tensor fields from strong edge pixels in regions, and red dots are the tensor fields created by interpolation in the region's local patch window. A higher level is created by applying a Gaussian low pass filter to the lower level and then sub-sampling the tensors at the lower lever.

Once the pyramid is created, we can look up a tensor value for any pixel in a frame. During the painting, if we need a tensor field of a pixel, first we check if it already has a tensor value assigned at the bottom level of the tensor pyramid (i.e., it is an edge pixel). If it does, we just use its value, otherwise we go up in the tensor field pyramid, and check if the corresponding region containing this pixel has a tensor value in this level, continuing until a value is found.

This tensor field pyramid algorithm has several advantages over the global interpolation method and its variations. First, we reduce the influence of non-uniformly distributed data by dividing the tensor field into regions and normalizing the tensor

values in each region. Second, it is a fixed cost algorithm that is much faster than global interpolation, since it only needs to check a 5×5 window for the unassigned regions at each level, instead of looping through all edge pixel/regions. The small window size also produces better results for our painting purpose. For example, the tensor field of pixels that are close to a strong edge will follow its tensor value. Hence it preserves fine local details better. Third, it avoids the zero tensor value ("holes") in textureless regions.

In our multi-layer painting, we divide a frame into regions with different size for different layers, and then create the tensor pyramid. The tensor value in a region gives the direction of a brush stroke across this region.

Figure 16 shows a visualization of the tensor field for an AR frame. A short line segment is drawn along the direction of the tensor field in the image.

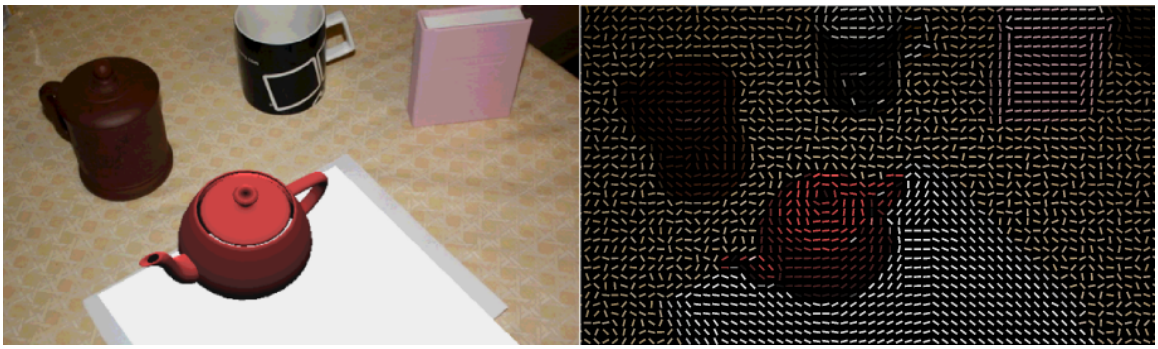

Figure 16: Tensor field visualization of an AR frame. There are a virtual teapot, two real cups and an album on the table.

4.3 Feature Point Based Brush Stroke Anchor Warping

In order to maintain coherence we need to move and repaint brush strokes based on the correspondences across frames. We track feature points across frames. For the non-feature pixels in the frame, we warp the brush strokes that are located at these pixels from frame to frame. The warping is done using barycentric coordinates.

4.3.1 Initialization of Brush Strokes at the First Frame

For multi-layer painting, we place brush strokes in each layer. The first frame is evenly divided into $M \times N$ grid regions. The initial anchor points of the brush strokes at each layer are the jittered centers of these regions.

4.3.2 Feature Points Tracking

We use video feature tracking to help produce brush strokes that are coherent between frames. In general any tracking methods that generate feature points across frames can be used. In our system the features points we choose are Harris corners [34]. We use SIFT to build gradient-histogram descriptors to find matched feature points from frame to frame [57].

4.3.3 Brush Stroke Anchor Warping

In AR we can only use information before the current frame, since we do not know what will be in later video frames. This characteristic of AR imposes a major constraint for coherence processing. We cannot analyze the whole sequences of the video or apply any global optimization across the frames [16].

To maintain the temporal coherence for brush strokes we warp their anchor points from the previous frame to the current frame in barycentric coordinates. Assume we have a set of matched feature points between two consecutive frames. For a brush stroke starting at position P in the previous frame, we find its three nearest neighboring feature points P_1, P_2, P_3, and compute P's barycentric coordinates $C = \begin{bmatrix} u & v & w \end{bmatrix}^T$ with respect to these three points. We then find the correspondent matches of these three points in the second frame P_1', P_2', P_3'. The matched feature points are found by using a ratio-testing of the gradient histogram feature vectors of feature points. We warp point P to the current frame, whose position P' is given by $\begin{bmatrix} P_1' & P_2' & P_3' \end{bmatrix} \times C$. P' is the new anchor location of the brush stroke. The procedure is shown below.

$$P = uP_1 + vP_2 + wP_3, \text{ where } u + v + w = 1 \tag{13}$$

$$P = \begin{bmatrix} P_1 & P_2 & P_3 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} \tag{14}$$

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} P_1 & P_2 & P_3 \end{bmatrix}^{-1} P \tag{15}$$

A close form solution of the barycentric coordinates of P is:

$$u = \frac{(y_2 - y_3)(x - x_3) + (x_3 - x_2)(y - y_3)}{detT} \tag{16}$$

$$v = \frac{(y_3 - y_1)(x - x_3) + (x_1 - x_3)(y - y_3)}{detT} \tag{17}$$

$$w = 1 - u - v \tag{18}$$

$$T = \begin{bmatrix} x_1 - x_3 & x_2 - x_3 \\ y_1 - y_3 & y_2 - y_3 \end{bmatrix} \tag{19}$$

Figure 17 illustrates how we warp a single brush anchor inside a triangle formed by its three neighboring feature points from the previous frame to current frame. Note this procedure still works when the anchor is located outside of the triangle.

When the scene is in motion, the warped brush strokes can become overly dense or sparse in some area. The overly dense brush strokes in an area may cause unnecessary repainting in this area, while the sparse brush strokes may not cover the area and lead to holes. We need to remove or add new brush strokes in these cases. In the multi-layer painting, the frame is divided into regions. We use an array to record the number of brush anchors in each region. If there are not enough brush anchors in a region, we evenly-subdivide this region, and then add new brush strokes to each sub region. Similarly we can remove brush strokes in overly dense areas. After adding

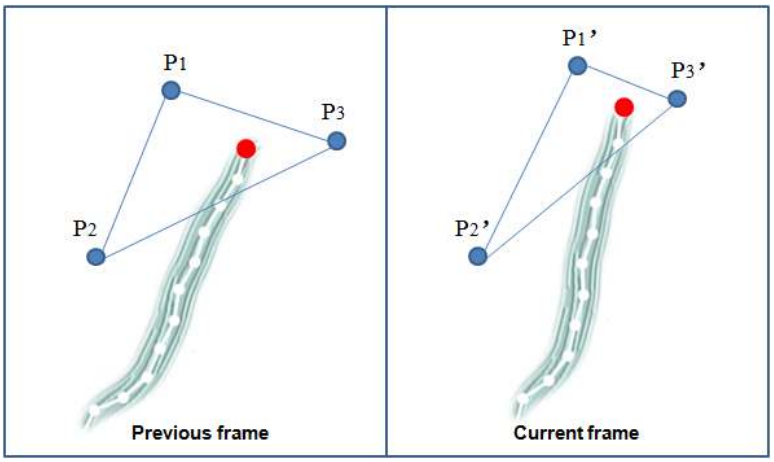

Figure 17: Warping of a brush stroke anchor point P denoted by a red dot.

or removing brush anchors in each region we update the list of brush stroke anchors correspondingly.

The correctness of the anchor point warping is based on the following observation: when the camera is far enough from the scene, if a non-feature point P is co-planar with some feature points in the previous frame, we can compute the correspondent $P^{'}$ in the current frame by warping. The warping gives the new anchor position of an existing brush stroke, and makes it coherent from frame to frame. Optical flow can also warp brush stroke anchors by estimating the movement of an individual pixel. Optical flow based brush stroke anchor warping is suitable for general videos where precise tracking may be unavailable, as shown in [16, 36]. The feature point tracking based brush stroke anchor warping gives more stable results in AR since we have feature tracking in the scene.

In our algorithm, a brush stroke anchor P could be outside of the triangle formed by its three nearest neighboring feature points. In this case the warping still works if P is close to these neighbors (i.e., these four points are approximately co-planar). Otherwise it may give undesirable results. Figure 18 shows an example from two consecutive AR frames. The matched SIFT feature points across frames are linked by colored lines. The anchors of painted brush strokes are correspondingly warped based

on these matches. Four corresponding areas from the painted results are highlighted at the bottom. Note that in some areas the brush stroke anchors are inside of the triangle formed by neighboring feature points, and in some other areas they are outside of the feature point triangles. Although the later case is undesirable, the warping still works when the camera is far enough from the scene.

Note that the feature triangles can straddle object boundaries (e.g., some vertices on objects and others on the background video). This is another undesirable situation, as the warping of a anchor point is supposed to be in a feature triangle that is on the same surface or plane. We should be able to alleviate this problem by looking at the video background and objects separately. If we mark pixels as 'object pixels' and 'background pixels', we could find feature triangles whose three vertices are all on the object or on the background, and use them for anchor warping. We could achieve this for virtual objects by distinguishing between the pixels from the video and our rendered virtual objects. We could further reduce this undesirable effect if we segment the video and separate physical objects from the background.

We use SIFT tracking to find matching feature points between the previous and current frames. Tracked feature points in frame f_n and f_{n+1} may disappear in frame f_{n+2}, but we are still able to warp brush stroke anchors since we can find another set of matches between frame f_{n+1} and f_{n+2}. If the tracking is temporarily lost during the camera motion we can use the latest list of brush stroke anchors as the starting points to paint the new frame. Once the tracking works again we continue warping the brush stroke anchors if a large number of matched feature points are found in the new frame, or re-initialize all brush stroke anchors and restart warping. We always try to use the warping option in our system when the tracking recovers after a temporary loss. If the number of matched feature points is very small in this case we re-initialize all brush anchors and restart warping for the following frames.

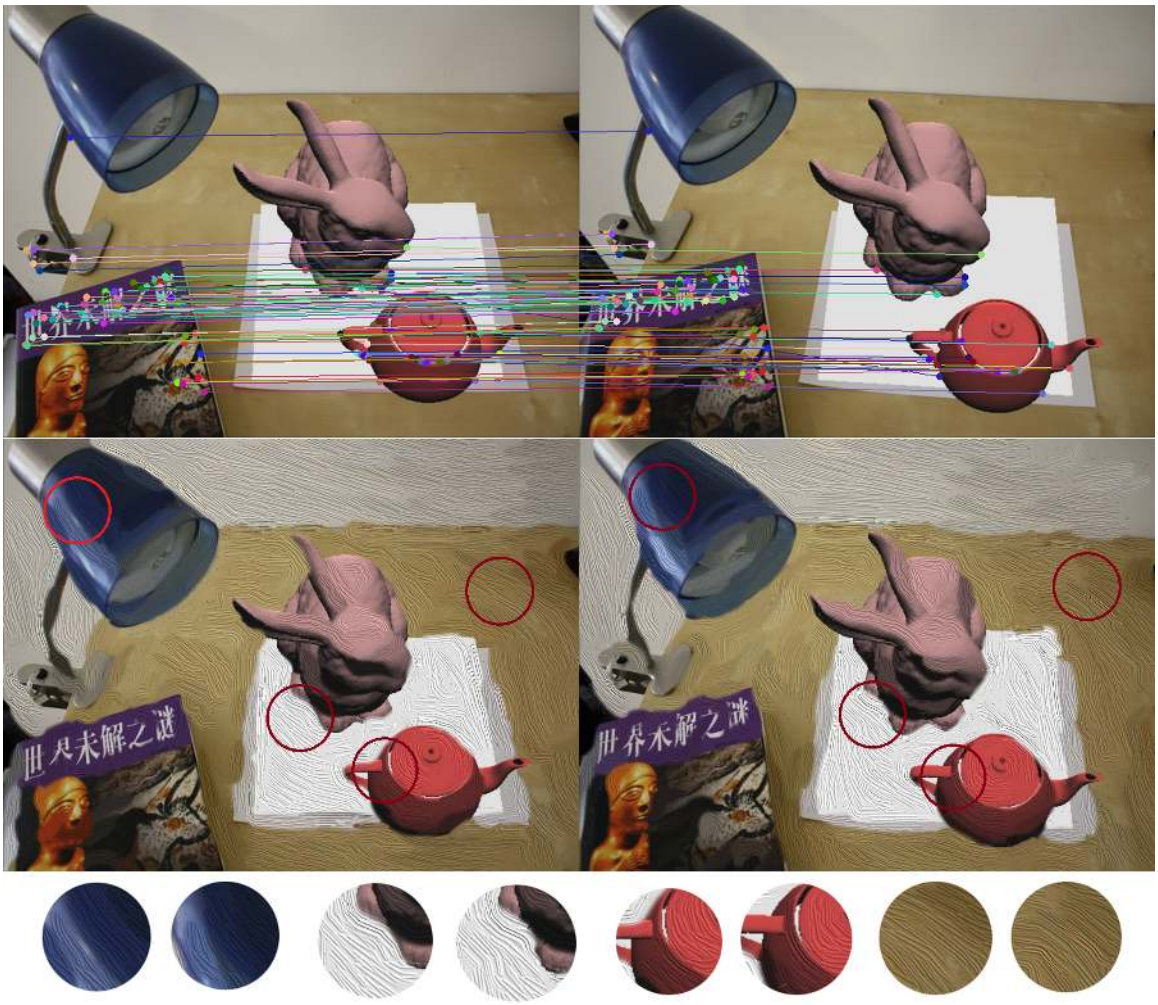

Figure 18: Top row: Matching SIFT feature points in two frames. Feature points are Harris corners. 256 features detected in the left frame, and 206 features detected in the right frame. ($CornerThreshold = MaxCornerStrength/5$). 80 matches are found. Matching features are linked with colored lines. Bottom row: Painted results for these two frames. Brush strokes in four correspondent areas are shown at the bottom. Note that in some areas the brush stroke anchors are inside the triangles formed by neighboring feature points, as well in some areas the anchors are outside the feature triangles.

4.4 Bump-mapped Curly Brush Strokes

We create long curly brush strokes based on the algorithm of Hertzmann et al. [38]. The difference between our algorithm and Hertzmann's original one is that we use bump mapping to simulate the lighting of textured brush strokes that are present in real world paintings. Hertzmann et al. also proposed to use a height map to create lighting effects, but this is not directly used in the creation of a single brush stroke [39]. Another difference is we choose cut-off angles as the stop condition for brush strokes with multiple segments, compared to the work of Hertzmann et al. and Hays et al. [38, 39, 36]. Hertzmann et al. use only the color difference as the stop condition for creating curly brush strokes. Hayes et al. allow only straight brush strokes with a single segment.

For each brush stroke we define a set of properties, including the width, segment length in a single travel step, minimum and maximum travel steps, and a color difference threshold. To reduce the curvature change in the brush stroke we also define a cut-off angle as an extra property. We choose the different sets of property values for each layer in our painting.

We start from an anchor point to compute each curly brush stroke. To elongate a brush stroke we travel a short distance (i.e., segment length) in the tensor field direction and this brings us to a new region. The tensor field direction is ambiguous by a half-turn, so we select the direction that will not cause a kink in the stroke, thus minimizing stroke curvature. We repeatedly extend the stroke until the color in the new region varies too much from the color in the starting region, or the direction change of two consecutive segments in a brush stroke exceeds the cut-off angle threshold. Compared to the fine layer, the coarse layer has a smaller cut-off angle and a bigger color difference threshold, since we observed real artists often tend to cover the background with thick and straight brush strokes first, and use thin and curly brush strokes to refine details later. The color difference can be measured in

RGB or HSV space.

In each step along the stroke we record the position of the regions as control points. Once we have all the control points $C_0, C_1, \ldots, C_{n-1}, C_n$, we compute a cubic B-spline curve based on these control points. To force the result curve to interpolate at the start and end points, we use triple knots at C_0 and C_n. As a result, the input control point set for the cubic B-spline interpolation is $C_0, C_0, C_0, C_1, C_2, \ldots, C_{n-2}, C_{n-1}, C_n, C_n, C_n$. Correspondingly we will get $n + 2$ segments. In each segment we choose 4 sample points at $t = 0, \frac{1}{4}, \frac{1}{2}$ and $\frac{3}{4}$. In the final painting we will have $4(n + 2) + 1$ sample points. We divide the curve into $4(n + 2)$ subcurves and draw each subcurve as a quad. Figure 19 shows a parameterized brush stroke with 11 control points.

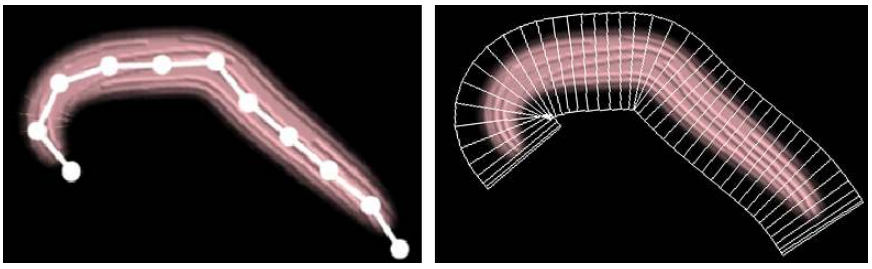

Figure 19: Left: A curly brush stroke with 11 control points (triple knots at begin and end points). Right: The brush stroke is divided into quads and rendered.

Figure 20 shows two of the various styles of brush strokes used in our paper, each of which has an alpha mask and brush stroke texture.

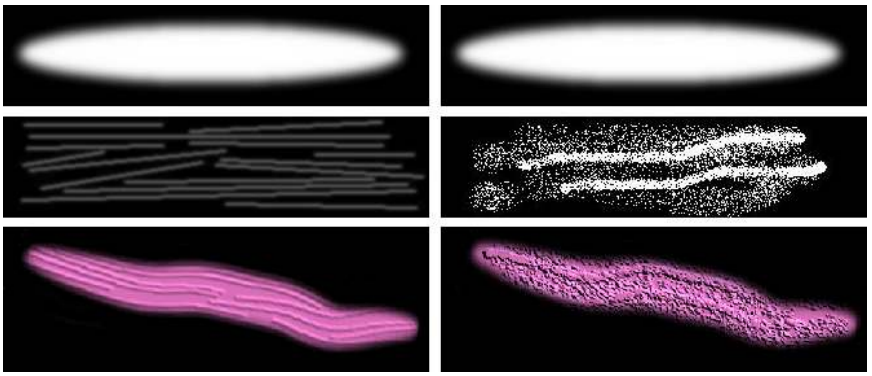

Figure 20: Each column is a type of a brush stroke. From top to bottom: an alpha mask, brush stroke texture, and example of a final composed brush stroke on screen.

Figure 21 visualizes the brush stroke skeletons of an AR frame. Each brush stroke is rendered as a colored curve.

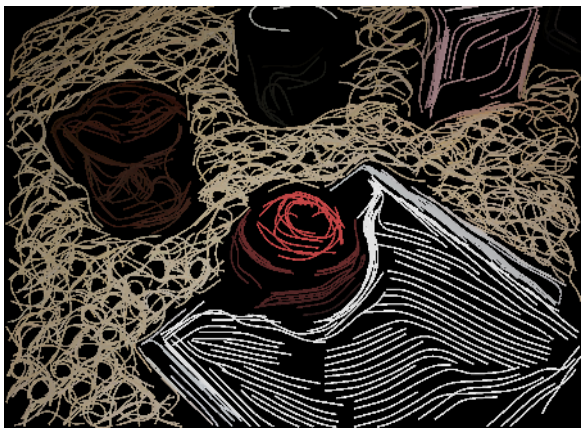

Figure 21: Brush stroke skeleton visualization on the coarse layer of an AR frame.

To obtain better visual results, we use bump mapping to paint each brush stroke, instead of texture mapping. Each brush stroke also has an alpha mask to mimic the smooth and natural brush stroke shape from a human artist's painting. This is also used in Hays et al.'s work [36]. Bump-mapping can simulate the subtle lighting effects for the brush stroke and drastically improves the visual quality of the final painting. We implement bump mapping using GLSL on the GPU for speed. To get correct lighting, we pass the orientation of each painting segments as a tangent vector to the vertex shader. In the fragment shader, the surface normal is estimated from a gray-scale brush stroke texture (e.g., Figure 20 middle row) as follows:

```
1 vec3 BumpNorm = vec3((left.x - right.x) , (bottom.x - top.x) , 1);
2 BumpNorm = normalize(BumpNorm);
```

The 'left, right, bottom, top' variables are the colors of the four adjacent pixels in the texture.

Multiple brush strokes could be placed across a same area on a frame, which causes unnecessary repainting. To avoid this problem, we choose a proper width for each brush stroke. Once a brush stroke is drawn, we mark the regions covered by this

brush strokes as "painted".

Figure 22 shows the result of the final painting. Note bump-mapping simulates the lighting effect of artistic painting and adds fine details to the result.

Figure 22: Final painting with curly brush strokes. Note the fine details in the painting. Three different styles of brush strokes at the coarse and fine level are shown at the bottom row.

4.5 Coherent Curly Brush Strokes

In order to keep temporal coherence we warp the anchor points of brush strokes from the previous frame to current frame. However, simple repainting a brush stroke at the corresponding anchor points at the current frame sometimes produces poor coherency. The computation of a brush stroke curve is decided by the local tensor field and color difference of regions around the anchor, so the shape of a curly brush stroke in two frames could vary significantly, even if its anchor points are coherent among the frames. Hence it is also necessary to change the shape of a brush stroke in the current frame to make it similar to the previous one.

Our assumption is that a coherent brush stroke in two consecutive frames should

have similar shapes that do not vary too much. To make the shape of a brush stroke in the current frame resemble the previous one, we first compute a new curly brush stroke in the current frame, and then we blend it with the corresponding brush stroke from the previous frame that has the same anchor point. Both brush strokes are parameterized. We compute the average of the two parameterized brush strokes, and then use the blended brush stroke for painting in the current frame. If the two brush strokes do not have a same number of control points, we resample both of them to obtain the same number of sample points, and then compute the average brush skeleton. Figure 23 shows the blending method.

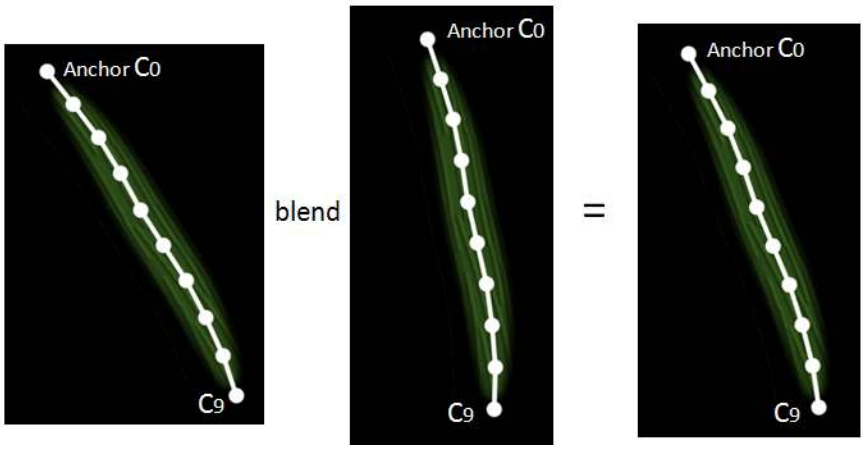

Figure 23: Blending of two parameterized brush strokes. Each brush stroke has 10 control points.

4.6 Results

We applied our painterly rendering algorithm to both static images and AR videos. The results are shown in Figure 24 and Figure 26. The rendered static images are comparable to Hays and Essa's work [36], which is one of the highest quality methods that we know. We also show a sequence of frames from an AR video to demonstrate the coherence.

The algorithm processes AR video with coherence in image space, as we categorized in Section 2. We choose a painterly rendering style in our system, but the

algorithm may be extended to other brush based NPR styles for AR.

Currently our algorithm is focused on producing coherent painterly rendering effects for AR and it is not running in real-time. In our system the average processing time per AR frame is divided into four major stages: (1) tensor field creation, (2) Harris corner detection and SIFT matching, (3) brush stroke anchor warping, and (4) final painting. We tested the speed on a PC with Core2Duo 2GHz CPU and ATI5730 graphics card. The result is shown in Table 1. The bottleneck is the SIFT matching where we run ratio test for all detected corner pairs, and brush stroke anchor warping where we find nearest neighbors for each brush stroke anchor and compute its barycentric coordinates. However, the original SIFT ratio test can be replaced with a faster version. The neighbor finding can also be accelerated by using KD-tree-like data structure. These improvements can lead to the real-time performance in the future.

Table 1: Average Processing Time per AR frame. (Unit: ms)

Frame Resolution	Stage 1	Stage 2	Stage 3	Stage 4	Total
1440×1050	75	420	215	30	740
1024×768	50	173	125	27	375

This work has been published in ISMAR 2010 [11] as a poster and in ISVRI 2011 [12] as a full paper.

4.7 Discussion

This algorithm maintains temporal coherence in the final rendering by warping corresponding brush stroke anchor points and placing brush strokes at these anchors from frame to frame. A brush stroke anchor in the current frame is warped to the next one by computing its location in barycentric coordinates. The degree of coherence in this algorithm relies on the anchor warping method. The anchor warping loses accuracy if we lose tracking of the feature points in frames. Error in the feature point matching

Figure 24: Top row: original photos. Bottom row: our painterly rendering results, which are comparable with Hays and Essa's work [36].

Figure 25: A zoom-in view of the peach painting. Note the details of brush strokes in this region.

test may produce cross-matchings in certain scenarios (e.g., a tree scene that has a lot of Harris corner feature points). This situation will also break the coherence.

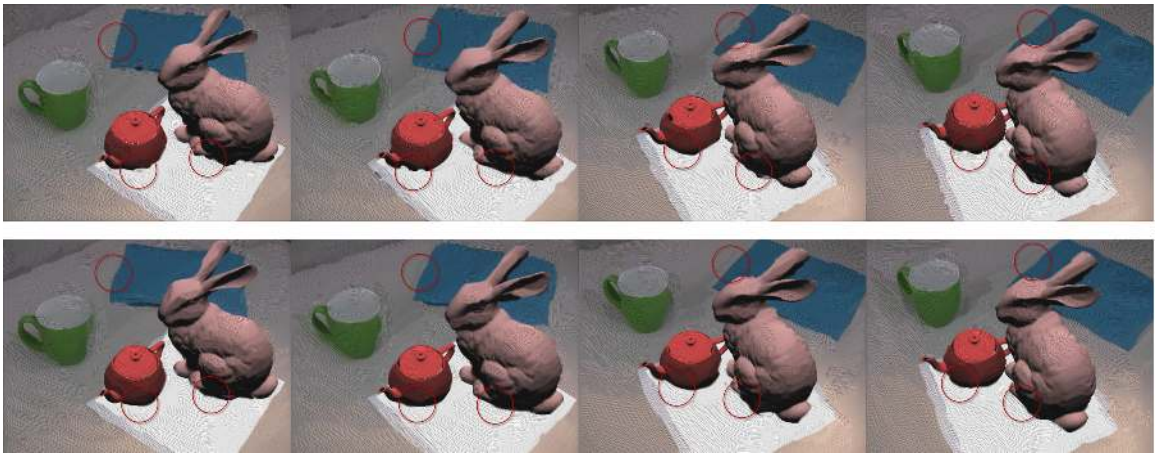

Figure 26: Top row: painted AR frames without coherence processing. Notice the brush strokes in circled areas appear and then disappear between frames. Bottom row: the same sequence of frames with coherence processing. Notice coherent brush strokes in circled areas.

CHAPTER V

NON-PHOTOREALISTIC RENDERING ALGORITHMS IN MODEL SPACE WITH COHERENCE FOR AUGMENTED REALITY

5.1 Algorithm Overview

We propose an NPR algorithm in model space for AR in this chapter. This method adopts the same method of creating tensor fields and rendering brush stroke textures as we described in the previous chapter. To achieve better coherent results by leveraging the information of the virtual models, our algorithm generates anchor points on a model's surface and adjusts their density on the screen. Our method also averages brush shapes from frame to frame.

The flowchart of our NPR framework in model space is shown in Figure 27.

We create a painterly rendering style for AR in our NPR framework. In brush-based NPR, the key for maintaining temporal coherence is to keep brush stroke textures smoothly transitioning from frame to frame. There are three major steps in our framework.

First we create tensor fields on AR frames to guide the orientation of the brush strokes. The tensor values are interpolated by a tensor pyramid structure.

Second, to achieve a high degree of temporal coherence for the final rendered results, we distribute 3D brush anchors on the model surfaces. We update the 3D anchor list to maintain their density in screen space from frame to frame. The 3D anchors are uniformly distributed in image space with our sampling algorithm, which is suitable for brush texture placement in NPR.

Third, we also keep the appearance of brush strokes coherent by averaging their

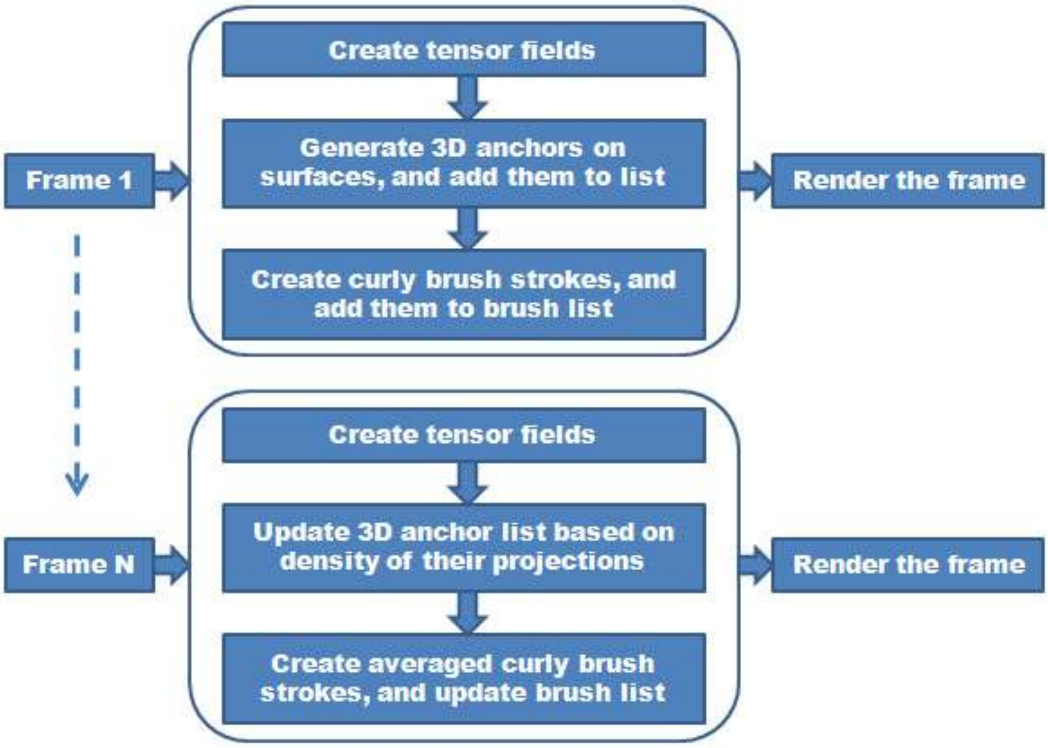

Figure 27: Flowchart of our NPR framework for AR.

properties between frames. The major properties of a single brush stroke include its skeleton and color. Our method smoothly blends curly brush strokes with a cubic B-spline representation. The second and third steps are the key for maintaining temporal coherence in our framework.

Finally we stylize the AR frame by placing brush strokes at the anchors on the screen. We will discuss the details of tensor field creation, brush anchor generation on surfaces, and property averaging of curly brush strokes in the following sections.

5.2 Tensor Field Creation

We adopt the same approach that is described in the previous chapter to create tensor fields and assign tensor values to pixels in each frame. This process can be parallelized by GPU programming packages such as CUDA. We will discuss how to accelerate the tensor field creation using CUDA as a part of our future work in the final chapter.

5.3 Brush Anchor Generation on Surfaces

A key for keeping coherence in brush stroke based NPR is finding the correspondence of the brush stroke anchors and placing the brush strokes at these anchor positions. Generating 3D samples is an important topic in computer graphics. A widely-used algorithm is to generate samples on the triangles of a manifold surface using barycentric coordinates. This algorithm randomly picks a triangle each time to create a new sample on it, with a probability proportional to the ratio of the area of this triangle and the total area of the mesh. We will compare the anchor points generated on a static model by this algorithm and our new algorithm later in this section. Details of this algorithm is listed as follows, assuming we have a manifold mesh that has N triangles.

1. The area of each triangle $S_1, S_2, ..., S_N$ is first computed, and the sum of areas of triangles S in the mesh. Next, the ratio between accumulated areas staring from the first triangle are computed, with the area sum S. These ratios are stored in an array.

$$S = \sum_{i=1}^{N} S_i \tag{20}$$

$$r_i = \frac{\sum_{j=1}^{i} S_j}{S} \tag{21}$$

2. A random number p between 0 and 1 is generated, and binary search is used in the sorted array of ratios r_i to find the first ratio that is bigger than p. The corresponding index i gives the triangle in which a new sample will be generated.

3. The second step is repeated to generate a set of samples on surfaces. These samples are evenly distributed across the object's surface.

This algorithm can produce random samples on model surfaces but the projection of these samples are not evenly distributed in image space. Hence these samples are not ideal for brush-based NPR algorithms. Also, the number of samples this algorithm gives is fixed. It is difficult to generate new samples in order to adjust the density of anchor positions on the screen in this method. In NPR algorithms we often need to maintain the density of anchors in each region on the screen for placing brush strokes. For example, we need to add or remove anchors when the camera is moving close to or far away from the models that we are painting.

To solve the stroke placement problem, we create an anchor list and update the list based on the density of their 2D projections on the screen. Our new algorithm for brush anchor placement makes use of an image-space grid and back-projection onto the 3D model's surface. The size of grid cells in our implementation is 8×8 pixels. For the first frame we pick jittered centers of grid cells in screen space and back-project them onto the model surface to obtain the initial 3D anchor list.

In the following frames, we dynamically add and remove 3D anchors in each frame to maintain the density of 2D projections of the 3D anchors in our list. If certain areas have too many anchors, we simply remove 3D anchors whose projections are inside the area. We keep updating the current anchor list from frame to frame.

After these 3D anchors are projected to the screen for a frame, it is still possible that certain areas are not covered by any anchors. Hence no brush strokes will be placed in these area and it will cause "holes" as a result. To solve this problem, we propose an algorithm to dynamically add new 3D anchors to our anchor list. These new 3D anchors are generated at each frame by back-projecting 2D points from image space into the model surface. The pseudo code of this algorithm is listed as follows.

```
1 Compute the 2D projection of each 3D anchor in the current anchor list
2 Update the number of anchors in each grid
3 for every grid in the screen
4 {
```

```
    if (the number of anchors in this grid <= lower_threshold)
    {
        Randomly pick a point P inside this grid;
        Back-project P to 3D model surfaces, and find the corresponding
            3D location Q on mesh for P;
        Add Q to current anchor list;
    }
}
```

Note that the back-projection of a point from the 2D screen to a 3D mesh is a non-trivial problem, given the fact that the projection matrix returned by our AR tracker contains components for camera calibration. It is difficult to use the regular OpenGL routine to compute the intersection of the eye ray with a mesh with this non-standard perspective matrix. To avoid this problem, we propose a much simpler and clearer way for the back-projection. The pseudo code of our back-projection is shown below.

```
Render the mesh with only color. The color value of a triangle is the
    index of it in the mesh.
// return a 3D anchor from mesh for a given grid
Back_Projection(grid index, mesh)
{
    Randomly pick P inside grid, get the pixel color at P;
    Decode the color to index i;
    Generate a random 3D point Q in i-th triangle in mesh;
    while (projection of Q is not inside the grid)
    {
        Generate a random 3D point Q in i-th triangle in mesh;
    }
    return Q;
}
```

In this back-projection routine we generate the random 3D point Q with barycentric coordinates (x, y, z) uniformly distributed in the i-th triangle as follows:

$$u, v = Unif[0, 1] \tag{22}$$

$$x = 1 - \sqrt{u} \tag{23}$$

$$y = \sqrt{u}(1 - v) \tag{24}$$

$$z = \sqrt{u}v \tag{25}$$

Although our trial-and-error method has a loop, it is very efficient in most cases. Models we are working with usually have a large number of triangles. The projection area of a single triangle is usually very small in most camera scenarios. Hence we usually produce a new anchor Q in only one trial in our testing. The back-projection method also by-passes the problem of computing intersections between the eye ray and triangles with a complex non-regular OpenGL perspective matrix.

This algorithm can be used for both static models and animated models. For static models, we store 3D positions of each anchor in the 3D anchor list. We use the color polygon ID to find the triangle index in the mesh, and then generate a new 3D anchor using barycentric coordinates. We update the 3D anchor list from frame to frame. For models with rigid body animation, the process is similar. We just need to apply the corresponding world view matrix to each anchor when we compute the 2D projection at each frame.

For models that are animated using mesh skinning, we store the triangle index and the barycentric coordinates of each anchor in the 3D anchor list. To compute the projection of a 3D anchor at a certain frame, we need to compute the position of the three vertices in the triangle based on the corresponding bone transformation and bone weight matrix. We then compute the new 3D location of this anchor in the current frame using its barycentric coordinates inside this triangle, and finally

compute its 2D projection on the screen. In the back-projection process, we first find the triangle index using color polygon ID, then generate a new anchor in the triangle and store its index and barycentric coordinates to the anchor list.

Figure 28 compares the anchor points produced by the basic point sampling algorithm, and our new algorithm. The red dots in the left column are visible 2D projects of the anchor points produced by the basic algorithm. Note these positions are not evenly distributed on the screen from frame to frame. On the other hand, the anchor points produced by our new algorithm (the blue dots in the right column) are uniformly distributed on the screen covered by the model. These points are also adjusted from frame to frame to maintain a proper density, which is better for brush stroke placement.

Figure 29 shows an alien spaceship, which is a static model, with over 7,000 anchors. The purple dots are the new anchors added for the current frame. The blue dots are existing anchors that are visible for the current frame in the current anchor list. Note that there is only a small number of purple anchors that are newly added to each frame, and most of anchors are blue ones after only a few "warm-up" frames. This means that we achieve a satisfactory anchor density quickly in this algorithm. This is exactly what we want for obtaining a stable anchor list for coherent painting.

Figure 30 shows a running superhero model with skinning animation with over 1,000 anchors. Note that anchors stick to the model's surface during the animation and that they are updated from frame to frame to maintain proper density on the screen.

This algorithm can also be applied to generate an arbitrary number of point samples on any 3D models. Compared to the original point sampling algorithm based on triangle areas, and other more complex point sampling algorithms (e.g., blue noise sampling), our algorithm has several advantages. First, it generates 3D samples whose projections are evenly distributed in 2D image space. Other sampling

algorithms produce samples that are uniformly distributed over a model's surface. As a result, their 2D projections on the screen can be sparse or overly dense in different regions. In our sampling algorithm, the 2D projections are immediately ready for use in brush stroke placement. This is a desired feature for many NPR algorithms. Second, it is easier to add or remove anchors to maintain a proper density of brush stroke anchors on the 2D screen, since we adjust the 3D anchor list based on the density of their 2D projections from frame to frame. Third, it should be possible to implement and speed up this algorithm with GPU based parallelization techniques such as CUDA. Our sampling algorithm is straightforward to parallelize, since the generation of an anchor in each region is independent of other anchors.

5.4 Brush Shape Coherence

In order to maintain temporal coherence we move the anchor points of the brush strokes from the previous frame to the current frame. However, repainting a brush stroke at the corresponding anchor points of the current frame sometimes produces poor coherence results. The shape of a textured brush stroke in the final painting is given by its *skeleton*, which is a sequence of control points. The computation of brush stroke control points is decided based on the local tensor field and the color difference of regions around the anchor. The shape of a curly brush stroke in two frames can vary significantly, even if its anchor points are coherent. In other words, tensor field incoherence between frames can cause the brush strokes to flicker.

There are several possible solutions to alleviate this flickering problem. One solution is to smooth the tensor field in the current frame based on the tensor information from the previous frames. It should be possible to achieve this by using region segmentation and doing the tensor field computation per-region. For example, we could first track and segment objects from the AR video, and then use interpolation to smooth tensor values in corresponding regions of the matched objects that we detect.

Note this solution is similar in spirit to the spatiotemporal volume analysis approach, which also uses the segmentation method to find the corresponding regions between frames for coherence processing. Once we segment the video and find corresponding objects and regions, the information can also be used to help clip brush strokes between regions, so that the brush strokes will not cross the boundary between the foreground objects and the background video when we paint the final layer.

Another possible solution is to change the shape of a brush stroke in the current frame to make it similar to the previous one. In additional to the shape of a brush stroke skeleton, we also need to add constraints to cap the change of a brush stroke color between two consecutive frames. We explore this solution in the remainder of this section.

Our assumption is that a coherent brush stroke in two consecutive frames should have similar shapes and colors that do not vary too much. We introduce a new algorithm to average the properties of two corresponding brush strokes, including their shapes and colors, between the current frame and the previous frame. We maintain a history list of the properties of the brush strokes at each 3D anchor in our algorithm. At each new frame, we first compute the new properties of a brush stroke (e.g., the averaged color and the skeleton control points) based on the information from the current frame. We then interpolate the new properties with the previous properties in the history list of this brush stroke. We use the interpolated results to paint the brush, and then update the history list.

In our implementation, we heavily use the vector class in STL to create and maintain the history list of brush stroke properties. For the fix-sized properties such as color (i.e., the color is a RGB triplet at each frame for each brush in the list), we use the following data structure to record the history of color of each brush stroke in our brush list:

```
1 // mycolor_3d is a struct which contains RGB values
```

```
2 std::vector< std::vector<mycolor_3d> > brush_color_list;
```

For the variable-sized properties such as control points (i.e., the size of control points of a brush stroke at each frame is not fixed), we use the following data structure to record the history for all brushes in our list:

```
1 // mypoint_2d is a struct which contains x, y coordinates
2 std::vector< std::vector< std::vector<mypoint_2d> > > brush_path_list;
```

The interpolation of colors of two corresponding brush strokes can be computed in many ways. We choose the following weighted average function that produces satisfactory results:

```
1 mycolor_3d new_color = (1 - alpha) * current_color + alpha *
    previous_color;
```

We ran several experiments to pick the optimal value for the coefficient alpha. Choosing alpha as 0.95 gives coherent results for the brush stroke skeleton and color in general.

The interpolation of the brush stroke skeleton is more complex. In our algorithm a brush stroke skeleton is described by a cubic B-spline representation. Given two sequences of 2D control points, we first describe the basic case, in which the two sequences have an equal number of control points. We start from the first control point in the current frame. For each new segment, we average the direction from two brush strokes, and then elongate the brush stroke skeleton to the new ending point. Since the length of each segment is equal in our algorithm, we also need to normalize the direction at each step. We continue doing this until we reach the end of both sequences.

We use the following routine to average directions to create a new brush stroke skeleton control points:

```
1 void AverageTwoBrush(std::vector<mypoint_2d>& b1, std::vector<mypoint_2d
    >& b2, std::vector<mypoint_2d>& result)
2 {
3    int size = b1.size() < b2.size() ? b1.size() : b2.size();
4    result.push_back(b2[0]);
5    float len = sqrt((b1[1].x - b1[0].x) * (b1[1].x - b1[0].x) + (b1[1].y
         - b1[0].y) * (b1[1].y - b1[0].y));
6    for (int i = 0; i < size - 1; ++i)
7    {
8       mypoint_2d dir1, dir2;
9       dir1.x = b1[i+1].x - b1[i].x;
10      dir1.y = b1[i+1].y - b1[i].y;
11      dir2.x = b2[i+1].x - b2[i].x;
12      dir2.y = b2[i+1].y - b2[i].y;
13      float alpha = 0.95f;
14      mypoint_2d sum_dir;
15      sum_dir.x = dir1.x * alpha + dir2.x * (1 - alpha);
16      sum_dir.y = dir1.y * alpha + dir2.y * (1 - alpha);
17      float len2 = sqrt(sum_dir.x * sum_dir.x + sum_dir.y * sum_dir.y);
18        // avoid divide by zero
19      len2 += 0.00005f;
20      sum_dir.x = len * sum_dir.x / len2;
21      sum_dir.y = len * sum_dir.y / len2;
22      mypoint_2d next;
23      next.x = result[i].x + sum_dir.x;
24      next.y = result[i].y + sum_dir.y;
25      result.push_back(next);
26   }
27 }
```

Note that we choose an alpha value of 0.95 to average the direction for each segment to create the new control point sequence.

Based on this basic case, we then need to consider the scenarios in which the two brush strokes have different numbers of control points. If the size of the brush in the current frame is smaller than in the previous frame, we can still use the routine above to compute the new average brush skeleton by using only part of the control points at the beginning of the previous one. If the size of the brush in the current frame is larger than in the previous frame, we do not have the information (e.g., control points) beyond the last segment from the previous frame to match in the current frame. One possible solution to this problem is to use only the information of the control points in current frame to continue the brush skeleton elongation. However this solution is not ideal.

We propose a general solution to deal with the problem of averaging two sequences of control points in all types of scenarios. We force the elongation of each brush to have the same number of control points at each frame for the purpose of interpolation, but we use only the valid part of the averaged brush control points for painting textures in each frame. There are several advantages to this solution. First, it avoids the problem of lacking information during interpolation. Second, it allows us to handle several complex scenarios in a unified routine as shown above. Third, it produces more clear and elegant code that has fewer conditional branches and runs faster. The algorithm is illustrated in Figure 31.

Figure 32 visualizes the brush skeletons of AR frames using our algorithm. In these results the brush skeletons stick to the object surfaces, which gives a very high degree of coherence for brush skeletons in the painting.

Figure 33 shows another example of coherent brush skeletons with an animated model.

5.5 Comparison of Algorithms and Results

To verify the success of our attempt at achieving better coherence for AR NPR, we make a series of comparisons between algorithms in this section. We compare image sequences from AR videos rendered by different versions of our AR NPR algorithms, including the image space and model space algorithms. We also compare the results with different levels of coherence that are produced by algorithms with different settings.

First, we compare the coherence of rendered AR video produced by the image space algorithm and the model space algorithm. We test our results on both static and animated 3D models. The models we choose to render are most common in most AR applications, such as AR games. We divide them into the following categories: humanoid, animal, vehicle and architecture.

Figures 34 and 35 show a painterly rendered alien spaceship and a tower, against a video background produced by our new model space algorithm. The brush strokes move smoothly from frame to frame without any obvious 'jumps'. Also note the coherent brush strokes at the spike-shaped areas in these two models across these frames.

Figures 36 and 37 compare the rendered results of a static dragon model produced by the image space algorithm as we described in the previous chapter and our new model space algorithm. While the results produced by both algorithms look acceptable, the brush strokes with our new model space algorithm in the right column are significantly more stable across frames. In particular, note the brush stroke textures on the dragon wing region in both results, as shown in Figure 37.

Figures 38 and 39 show a painterly rendering example of a running superhero model with skinning animation. Note the extremely stable and coherent brush textures on the chest area of the running superhero that are produced by our new model space algorithm.

We also explored the coherence of model space algorithm with different settings. These settings include different strategies of blending brush skeletons, averaging brush colors, and clipping brush strokes to edges.

Figure 40 compares the results of an alien spaceship model rendered by our new model space algorithm with different settings for the brush stroke averaging. The images in the left column are rendered with the a small blending weight $w1 = 0.05$ with the previous brush stroke properties including the color and skeleton. The images in the right column use a larger blending weight of $w1 = 0.95$. Note the rendered results in the right column appear more coherent, especially at the spike-shaped area and the flat interior of the alien spaceship model. When we choose a small blending weight $w1 = 0.05$ for the brush averaging we use the information that is mostly extracted from the current frame to create the brush stroke. Even if the anchor positions are perfectly coherent in this case the results still flicker because many brush skeletons change significantly between two consecutive frames, as shown in the images in the left column.

Figure 41 compares the results produced by our new model space algorithm with different settings for the edge clipping. The images in the left column are rendered without edge clipping, while images in the right column are with edge clipping. Note the artifact of color bleeding at the edges between the virtual objects and the real background in the images in the left column (e.g., the brush strokes at the boundary between the superhero's legs and the background). This artifact also appears when using the image space algorithm. Due to the inaccuracy of edge detection it is difficult to perfectly align the brush stroke textures on the edges. Once a brush stroke that crosses the boundary is created at a certain frame, it sticks to the model's surface and is carried into the later frames. This causes another undesirable artifact in the video, in addition to the color bleeding. We are able to alleviate these two problems in our model space algorithm by using the information from both the virtual models and the

real background. We clip brush stroke skeletons at the boundary between the virtual objects and the real background, since we are able to obtain an accurate boundary of the virtual objects from the frame buffer. Also, in the brush stroke averaging we calculate the properties of a brush stroke from the control points in the skeleton only inside the model area, if this brush stroke anchor is in the model. Note the images in the right column have most of these two artifacts removed.

The images in the right column still have some undesirable artifact, such as the color bleeding between the yellow belt and the green body of the model. It may be possible to reduce these artifacts by choosing a smaller color threshold for brush elongation in image space. If these two parts are separate meshes in the model, we can render them separately and then clip the brush strokes to the boundaries to alleviate this artifact.

We also explored different rendering styles in our model space algorithms. We tested our results with the same brush stroke textures, with and without bump mapping. Figure 42 shows an alien spaceship model rendered with brush texture only (the left column), and with bump mapping (the right column). The rendered results with bump mapping enabled simulate the subtle lighting effects of the canvas surface in the real world.

5.6 Discussion

5.6.1 Discussion of Algorithms

Although the models shown here have different numbers of triangles, in our model space algorithm the total number of brush stroke textures that are placed on screen is capped by the video frame size (640×480 pixels), the grid size (8×8 pixels), and brush density per grid (2 brush strokes per grid). This is another advantage of our anchor generation method. The total number of brush strokes is about $9,600$ per frame. Each brush stroke can have up to 20 quads, so the total number of polygons

rendered per frame is about 192,000. Hence the rendering is not the bottleneck of our system.

Using our NPR framework we should be able to create other NPR effects, such as sketch and hatching style, by using cross-hatching textures instead of brush stroke textures. Brush stroke anchors located on surfaces smoothly transition from frame to frame to provide temporal coherence. The final rendering is still composed in 2D since we place curly brush strokes at AR frames. Compared with painterly rendering algorithms in image space [12], our new model space framework provides better coherent brush strokes from frame to frame.

This algorithm can be used to render 3D graphics content in AR. A problem of using this method in model space is that in AR we usually do not have accurate information of the scene geometry except for the 3D graphical models. If we can reconstruct the 3D scene from the video then this algorithm can be directly used to render the video content with coherence. However, if the precise reconstruction cannot be done at interactive frame rates then we can use image space algorithms to process the video content with coherence, which leads to a hybrid combination for rendering AR. In this thesis we rendered the 3D graphics model with our new model space method, and rendered video content with an image space method. Both of them fit into our general rendering framework with coherence support.

5.6.2 Evaluation of Video Quality

In Section 5.5 we render the same video sequences using the image space algorithm and the model space algorithm. We also render using the model space algorithm with different settings. We mainly use subjective measurements to judge the quality of coherence in the renderings. Objective measurements are possible for the anchor distribution: our anchor generation algorithm gives the ground truth for anchor between

frames, and it can be proven that their projections are evenly distributed in the image space, compared with the basic point sampling algorithm. However, it is difficult to create objective measurements to compare the quality of the final rendering. We visualize the brush skeletons to show the coherence improvement, but the effect on human visual perception of coherence in the final painted results is more difficult to quantify.

Researchers have been used various subjective and objective metrics to help evaluate the perceptual quality of images and video. Traditional error summation methods have been used as an objective measurement for image quality by many researchers. Wang and Bovik presented a universal objective image quality index to assess the various image processing applications without employing the human visual system [73]. Claypool and Tanner asked test subjects to rate a *quality opinion score* ranging from 1 to 1000 after watching a video clip to measure the effects of jitter and packet loss on perceptual quality of streaming video [13]. Winkler and Mohandas gave a comprehensive review of subjective experiments and objective metrics of video quality and their uses [76].

We are focused on the perceptual quality and coherence of videos that are rendered by NPR algorithms. Many objective metrics such as the error summation cannot be used in NPR, as we usually do not have a corresponding stylized image or video to use as a ground truth to compare with our rendered results. Subjective experiments may be more useful to evaluate the quality of stylized videos in NPR. We used informal methods to compare the video quality in this comparison chapter, but a formal user study would be useful to better quantify the quality and coherence of our results. For example, a questionnaire or survey could be used to collect scores for perceived video quality and level of coherence of videos that are rendered by different algorithms with different settings. Interviews could be conducted after the study to collect user's feedback and comments. In addition to the subjective scores and comments, objective

measurements such as the movement of subjective's eye gaze when they are watching rendered videos may be recorded to help evaluate which parts of the video sequence are attracting user's attention (and thus may exhibit distracting artifacts). Although most of the collected data would still be subjective, this information may be useful to help better understand and evaluate the coherence problem in NPR for AR.

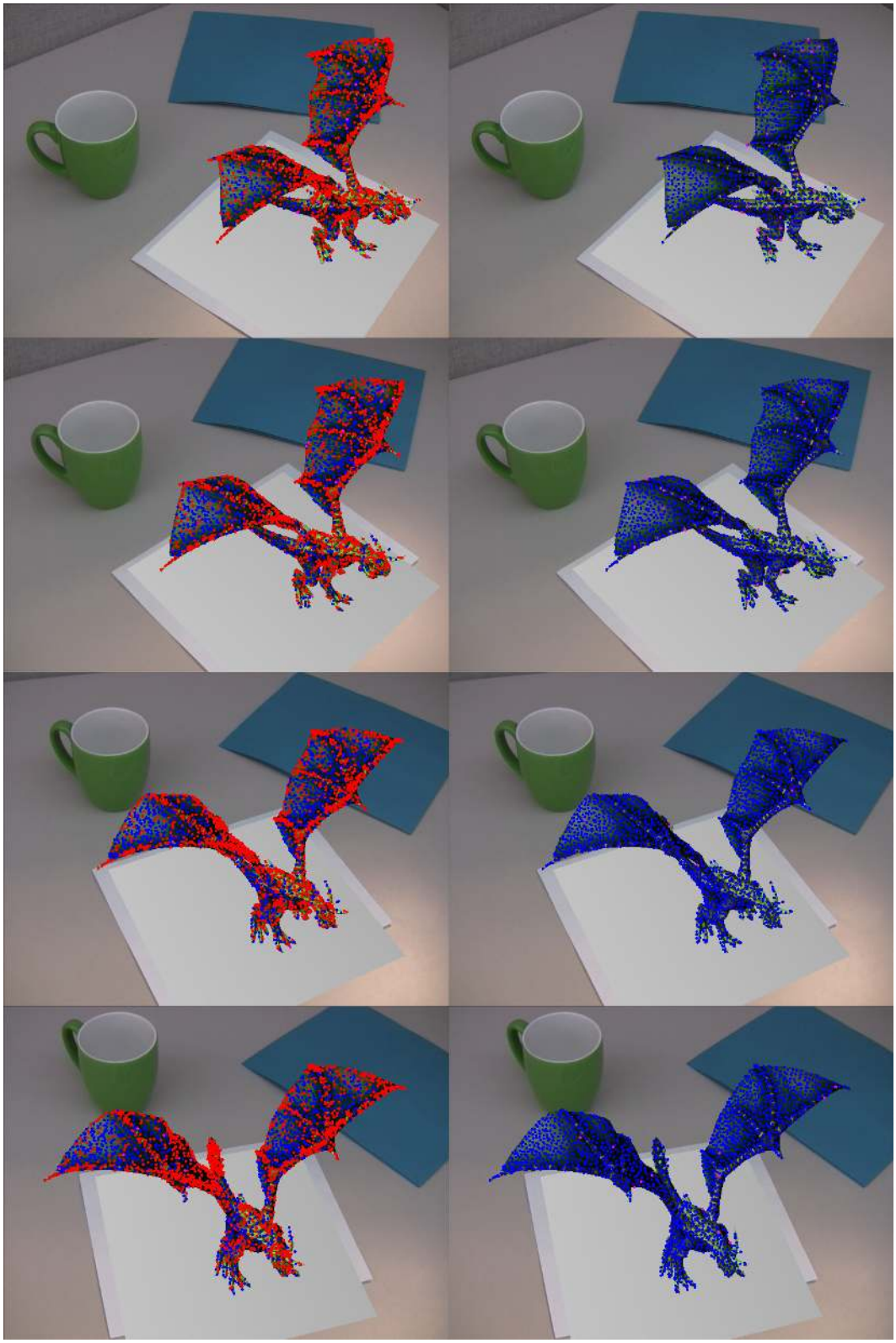

Figure 28: Comparison of the anchor points produced by the basic point sampling on surface algorithm, and our new algorithm. The red dots are visible samples at each frame produced by the basic algorithm. The blue dots are visible samples produced by our new algorithm.

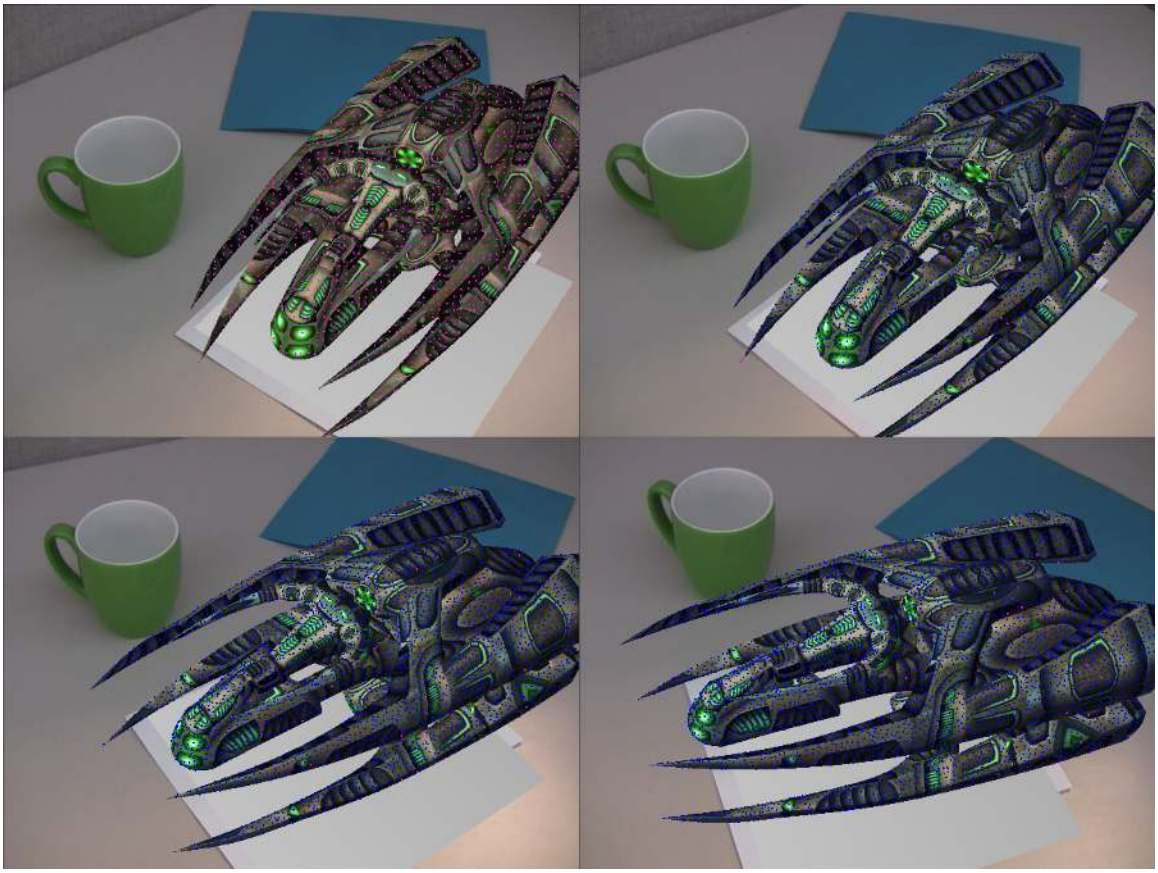

Figure 29: Brush anchor visualization for a static alien spaceship model. Four screens are created at frame 0, 20, 40, 60 from a video at 30 fps.

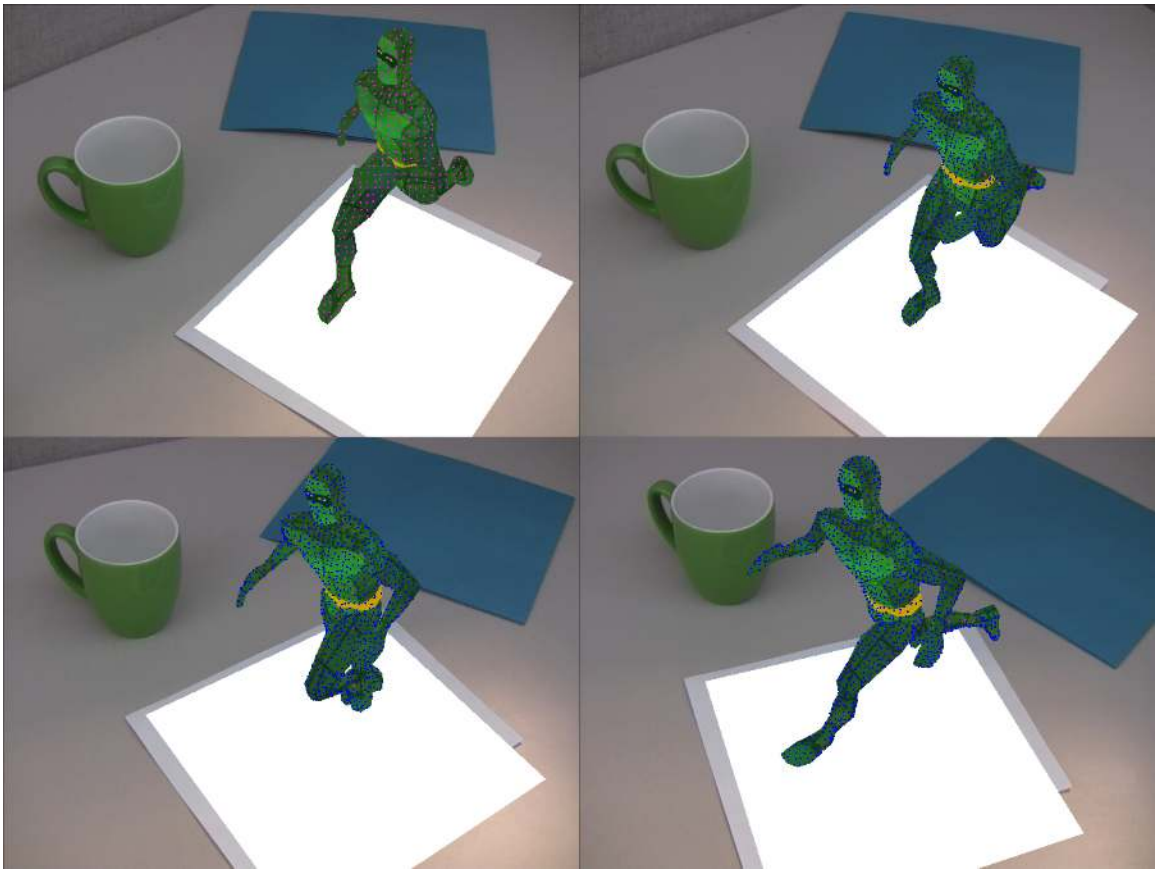

Figure 30: Brush anchor visualization for the running superhero model with skinning animation. Four screens are created at frame 0, 20, 40, 60 from a video at 30 fps.

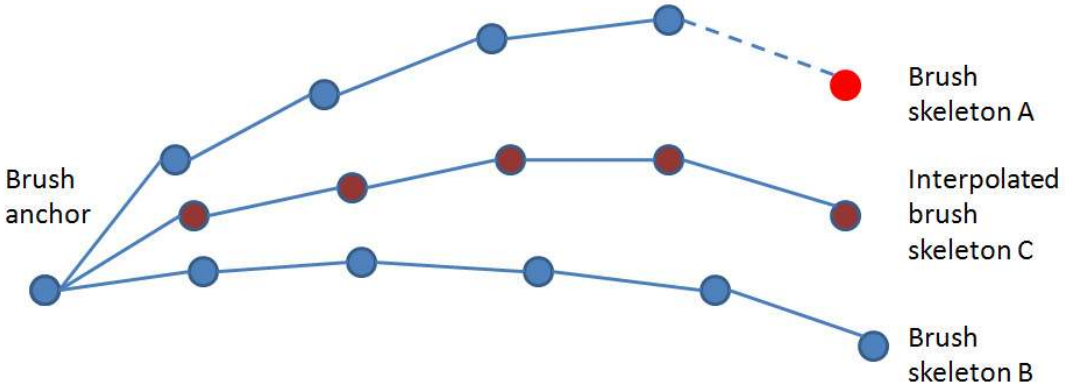

Figure 31: A general solution for averaging brush stroke skeletons with different sizes at two consecutive frames. The maximum size of a brush stroke is 5 segments and 6 control points in this case. Brush stroke skeleton A is from the previous frame. We force it to extend to 5 segments, and use only the first 4 segments in painting. Brush stroke skeleton B is initially created by local tensor fields from the current frame, and then averaged with A to interpolate C. C is the brush stroke skeleton we use for the final painting at this anchor point in the current frame. We also record C into the brush history list for this anchor in this frame.

Figure 32: Brush stroke skeleton visualization for the alien spaceship model. Four screens are created at frame 0, 20, 40, 60 from a video at 30 fps.

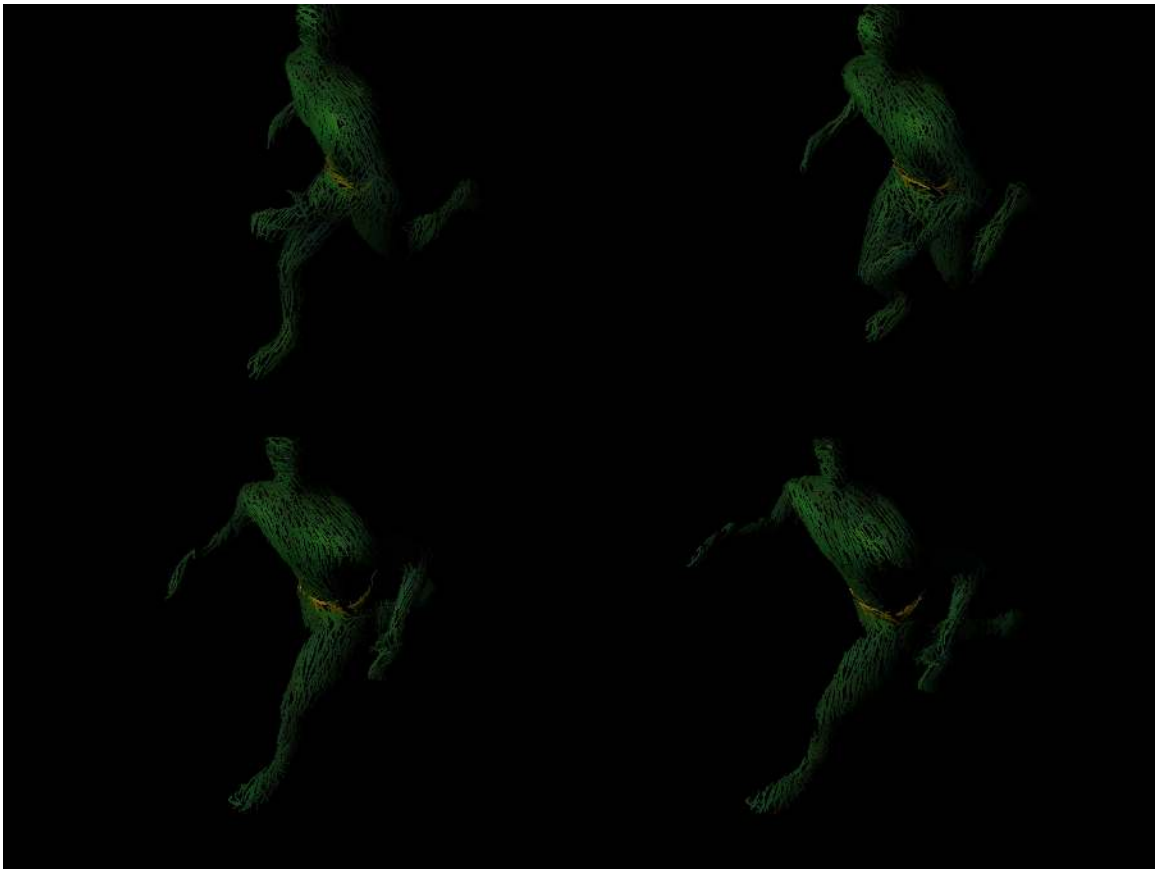

Figure 33: Brush stroke skeleton visualization for the running superhero model. Four screens are created at frame 0, 20, 40, 60 from a video at 30 fps.

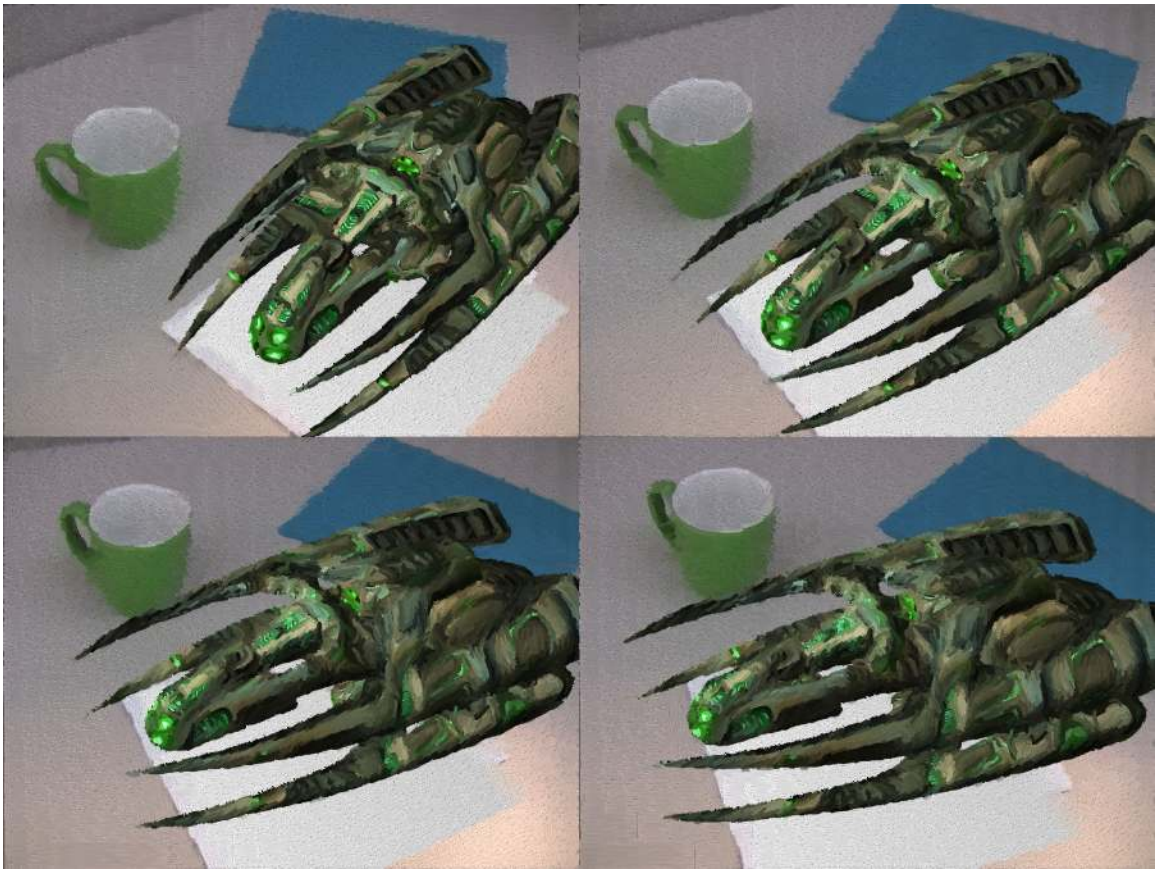

Figure 34: An alien spaceship with the real background painted with coherence by our model space algorithm.

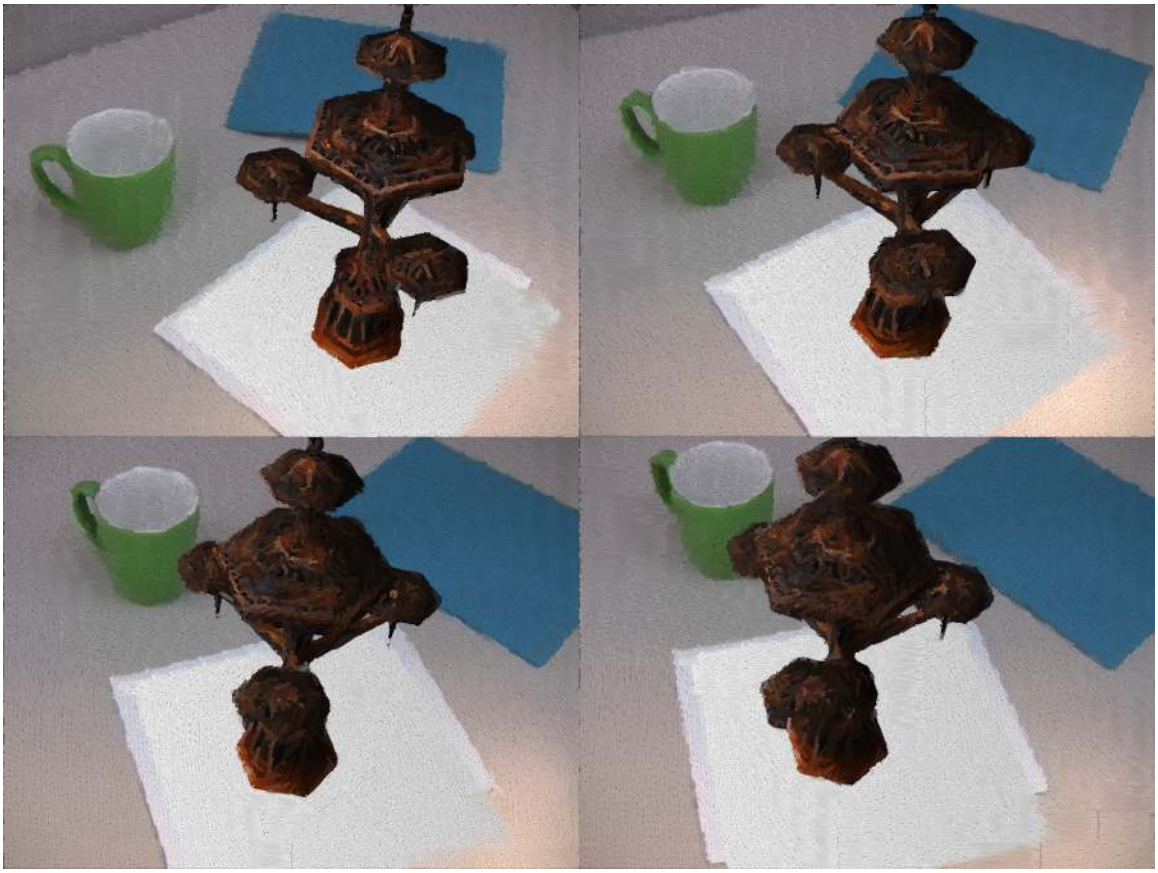

Figure 35: A tower with the real background painted with coherence by our model space algorithm

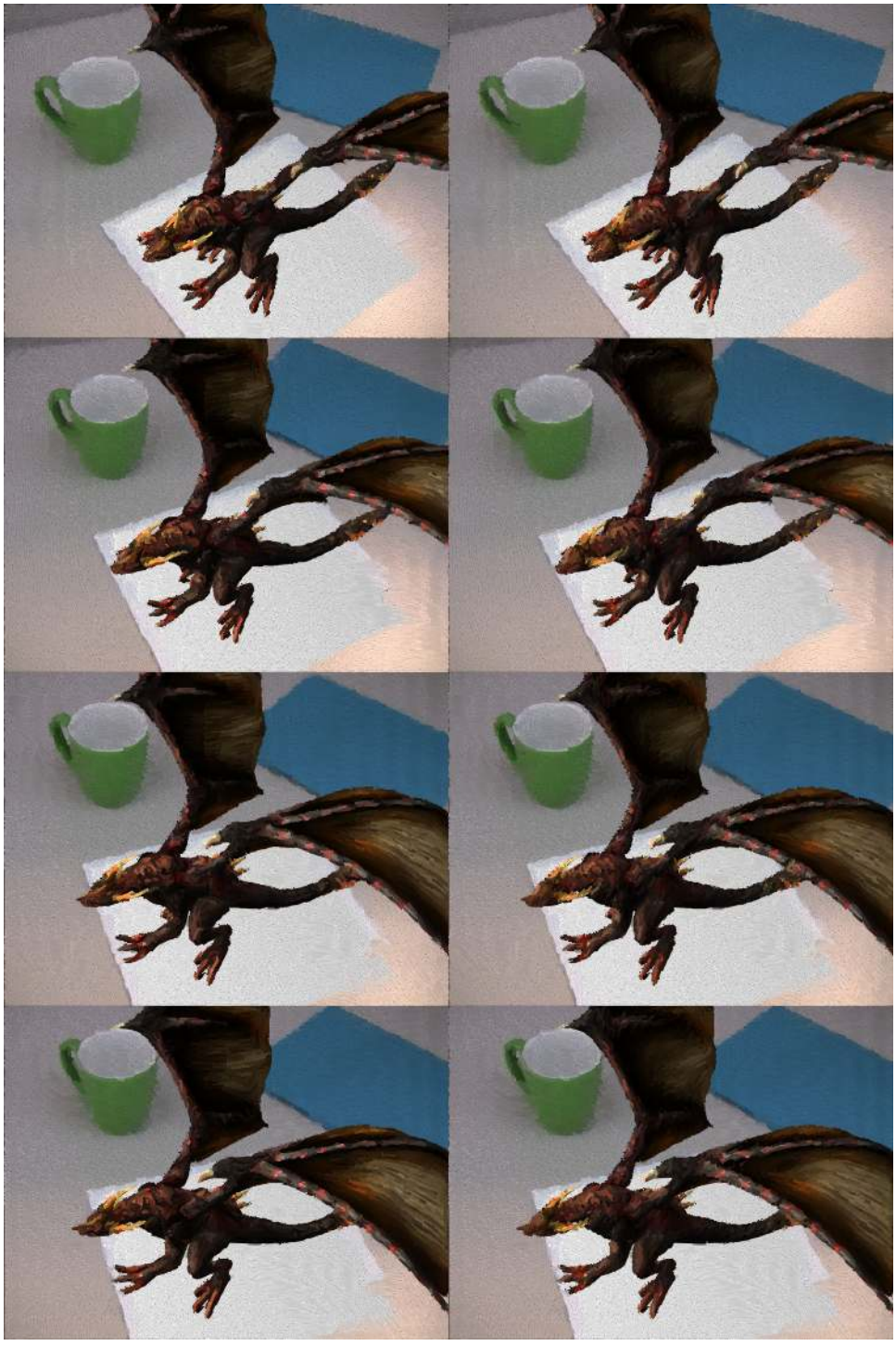

Figure 36: A painted dragon model. While each individual frame looks acceptable in both image space method (left) and our new model space method (right), careful examination reveals significantly greater brush stroke coherence between the model space images.

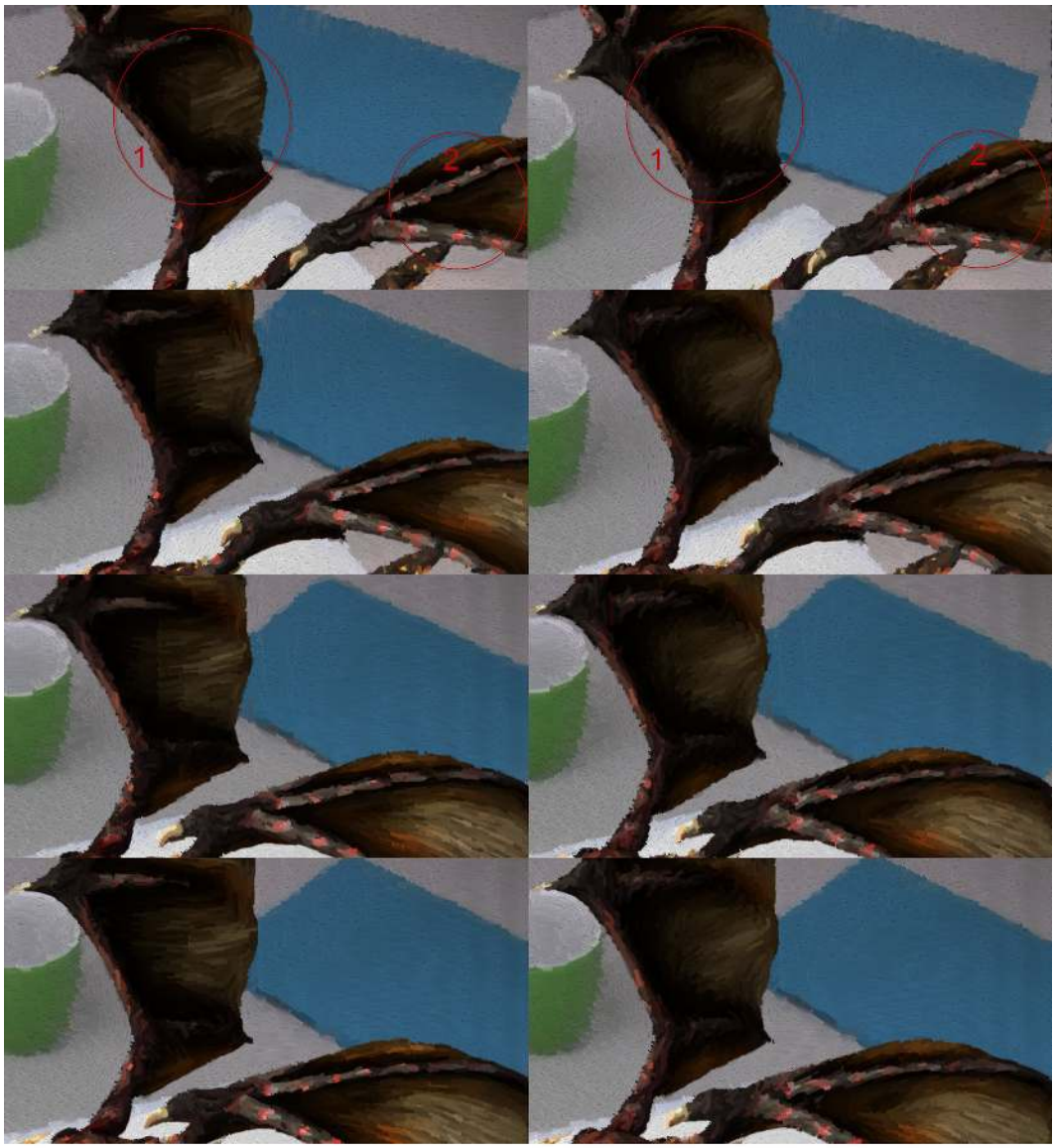

Figure 37: A zoomed in view of the dragon wing region. (Left: image space method. Right: our new model space method.) Although both results look good (the brush strokes in Circle 2 look coherent in the both methods), the brush strokes in Circle 1 are significantly more stable in the model space method than in the image space method.

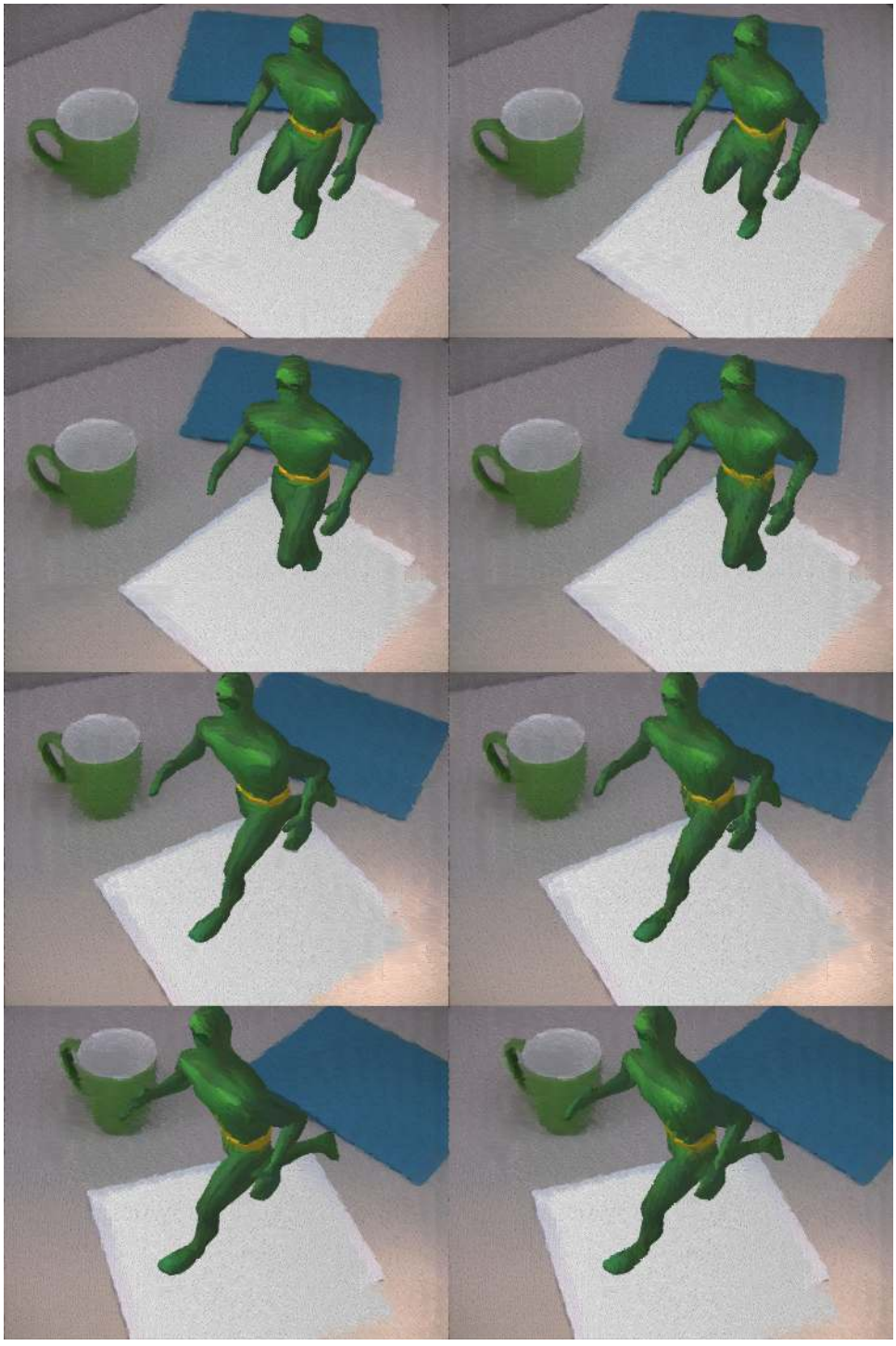

Figure 38: A painted running superhero model. (Left: image space method. Right: our new model space method.)

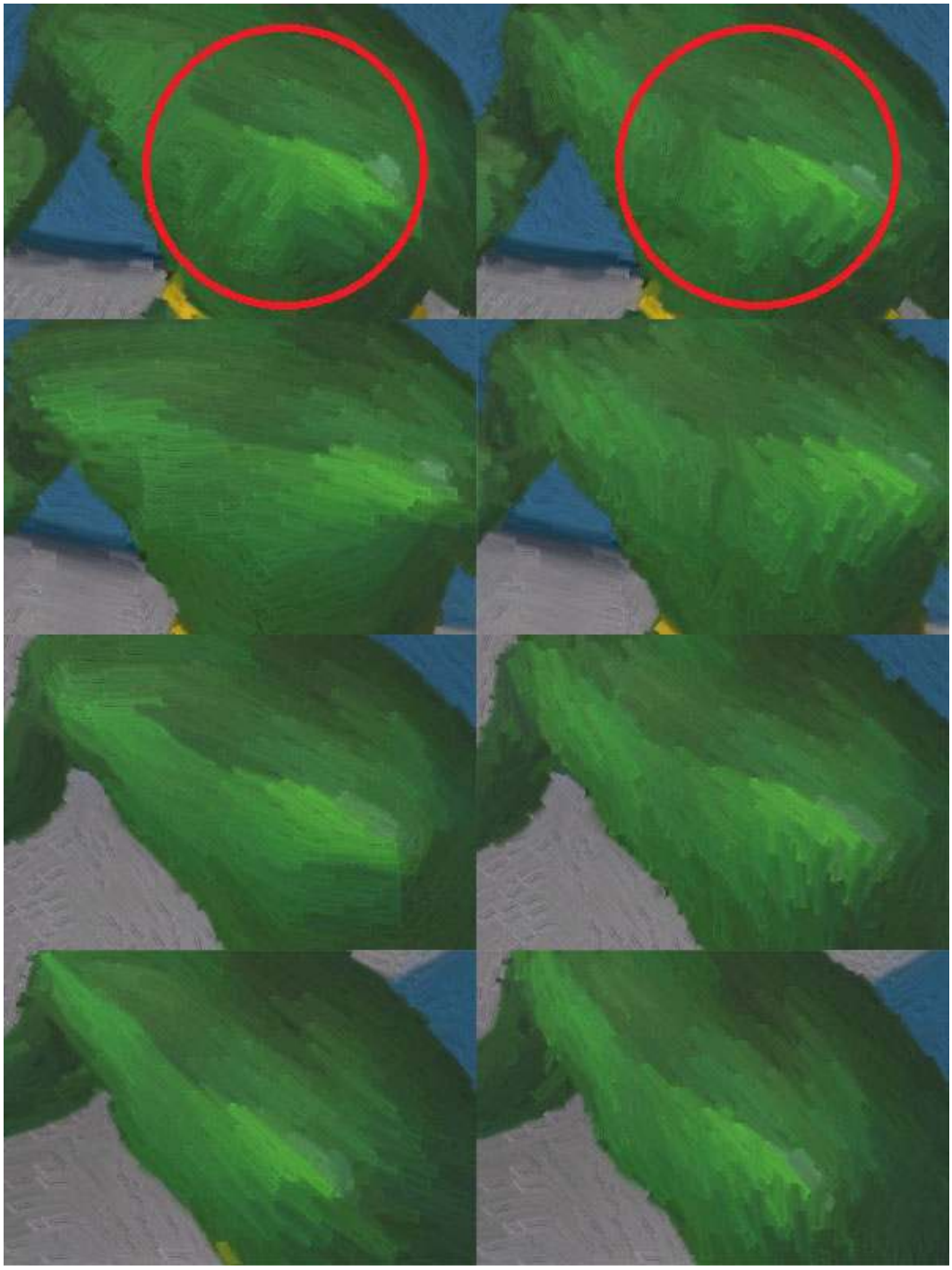

Figure 39: A zoomed in view of the chest area of the animated model. (Left: image space method. Right: our new model space method.) Note the brush strokes in the circle are significantly more stable in the model space method than in the image space method.

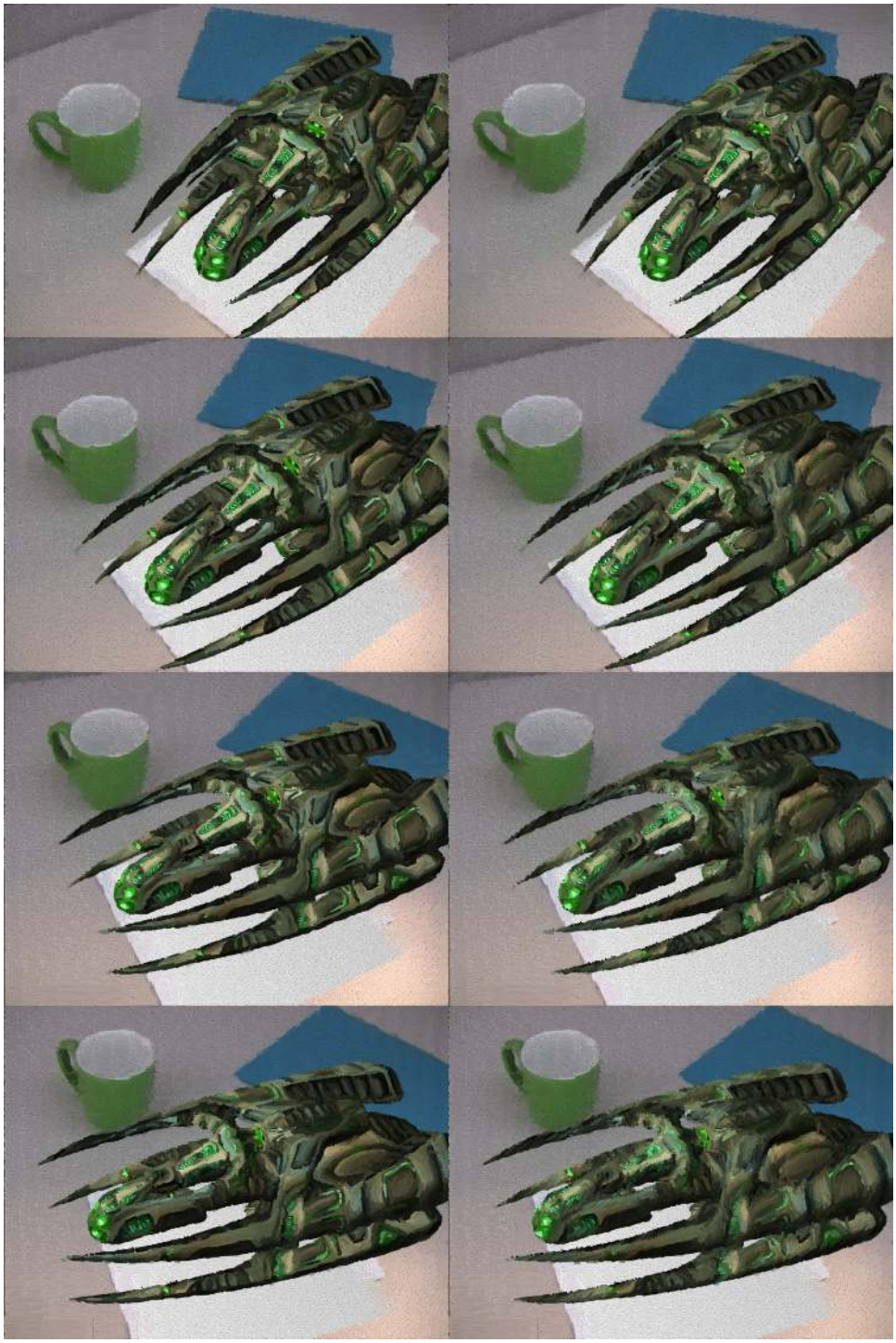

Figure 40: A painted alien spaceship model. Left: our model space algorithm with a small blending weight with the previous brush stroke's properties. Right: the same model space algorithm with a large blending weight with the previous brush stroke's properties.

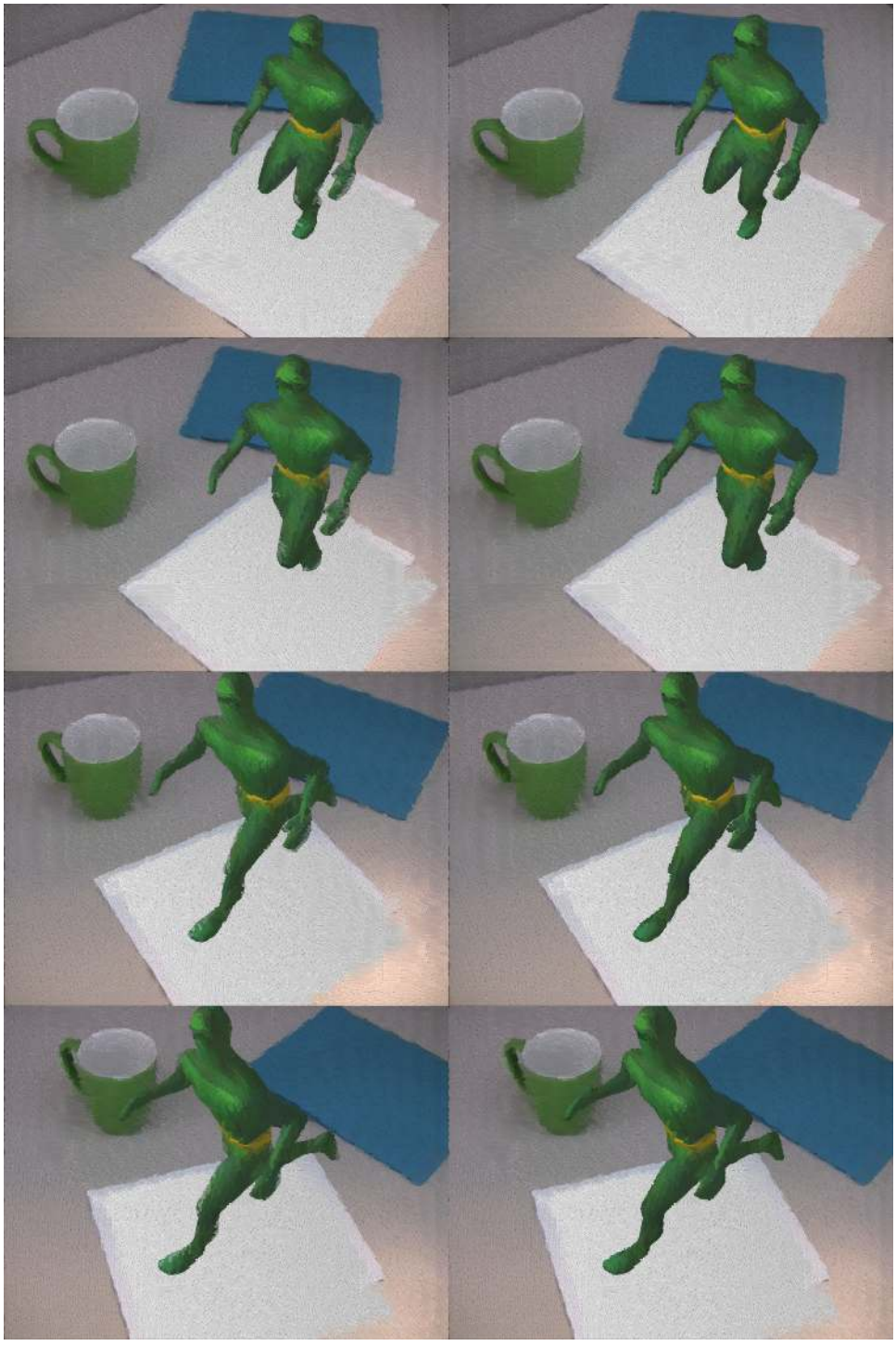

Figure 41: A painted running superhero model. Left: our model space algorithm without edge clipping. Right: the same model space algorithm with edge clipping. Note the artifacts in the images in the left column, including the color bleeding along edges, and incorrect brush stroke movement between frames. These artifacts are removed by using the model information in the right column.

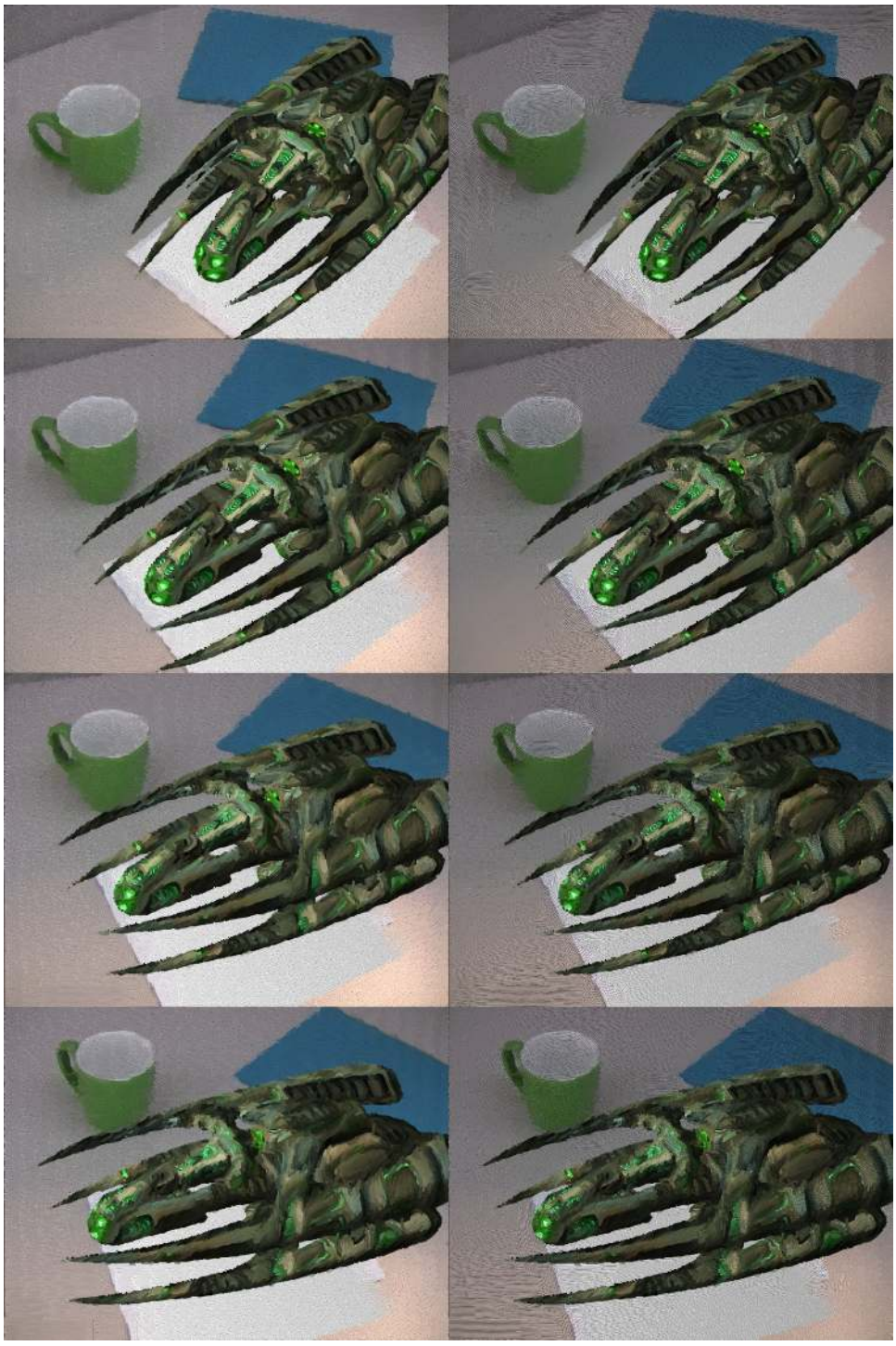

Figure 42: A painted alien spaceship model. Left: model space algorithm without bump mapping. Right: model space algorithm with bump mapping.

CHAPTER VI

CONCLUSION AND FUTURE WORK

We explored a variety of NPR algorithms with coherence in both the computer graphics and the AR/VR areas. We divided these algorithms into two major categories: model space algorithms and image space algorithms.

Creating NPR effects in AR/VR is an important approach to seamlessly blend the real and virtual content. This is also key to immersed user experiences in AR/VR applications. AR/VR NPR has unique characteristics compared with regular video NPR. In AR we have both real-world video content and computer-generated virtual graphics content. We have partial information of the scene from vision-based tracking, and complete information of the virtual objects. We cannot access any information beyond the current frame. AR also requires real-time performance in rendering. Coherence is a challenge in NPR algorithms. The characteristics of AR/VR make it even more difficult to achieve coherence. We usually cannot directly apply existing video NPR and real-time NPR algorithms into the domain of AR/VR.

In this thesis we presented three NPR algorithms for AR: a watercolor-inspired rendering in image space, and a painterly rendering algorithm in image space and a painterly rendering algorithm in model space. The watercolor-inspired rendering algorithm utilizes the Voronoi diagram to create a watercolor-like effect that mimics a real artist's work. This method maintains coherence in image space by re-tiling Voronoi cells along strong edge pixels in each frame.

The two painterly rendering algorithms place brush stroke textures on the screen to create the final result. The key to maintain coherence in brush stroke based NPR algorithms is keeping coherence for brush strokes across frames. The image

space version achieves coherence by warping brush stroke anchors with barycentric coordinates in feature point triangles from frame to frame. The model space version achieves coherence by generating 3D anchors on surfaces and averaging brush stroke properties between frames. We proposed using back-projection to generate anchors that are uniformly distributed in image space. We also presented a general solution for averaging brush strokes skeletons with a cubic B-spline representation.

Currently our algorithm creates NPR effects for AR/VR with coherence. Although our algorithms are currently running offline, they can be further optimized to achieve interactive frame rates. We analyzed the bottleneck of performance in our current algorithms. There are several improvements that can speed up the rendering. For example, we can accelerate the creation of the initial tensor field and the final tensor pyramid by using GPU-based parallel programming techniques such as CUDA. Similarly, the anchor sampling method can also be parallelized. Another direction to explore is to study how to make better use of the information extracted from the video and graphics content. The information extracted from video can be potentially useful to refine coherence for the graphics content, and vice versa. We will discuss these two parts in turn in the following sections.

6.1 GPU-accelerated Tensor Field Creation

This section we first review the algorithm for creating tensor fields. We then suggest a way to use the multi-core GPU to accelerate the creation of the initial tensor field and tensor pyramid.

6.1.1 Review of Tensor Field Creation

Although brush stroke anchors are generated on 3D model surfaces in our model space algorithms, we still need to create tensor fields for each AR frame in 2D image space for brush stroke shape creation. We discussed the details of the tensor field pyramid in the previous chapter. We have the following steps for creating tensor field pyramid

and assigning a tensor value to each pixel in the AR frame.

1. Divide the original video frame into grid-based regions, and sum and normalize the tensor values in each region.

2. For a region that is not assigned a tensor value, use radial basis interpolation to compute its tensor, but search only a small 5×5 window patch centered at this region.

3. Apply a low-pass Gaussian filter to the tensor values, and then sub-sample to create a new tensor field at a higher level in the pyramid.

4. Repeat Step 2, until we have reached the top of the pyramid and built the tensor field of size 1×1.

This tensor field pyramid algorithm has several advantages over the original global radial basis interpolation. First, we reduce the influence of non-uniformly distributed data by dividing the tensor map into regions and normalize the tensor field in each region. Second, it is much faster than a simple global interpolation, since it only needs to check a 5×5 window for the unassigned regions at each level, instead of all edge pixel/regions. The small window size also produces better interpolation results for our painting purpose (e.g., the tensor field of pixels that are close to a strong edge will strictly follow its tensor value). Hence it preserves fine local details better. Third, it avoids the problem of holes in textureless regions.

6.1.2 GPU-accelerated Tensor Field Creation

The algorithm of tensor field creation described in the previous subsection can be accelerated by using GPU based parallel computing techniques, such as GLSL, CUDA or Stream. All these techniques utilize the many cores of modern GPU and parallelize algorithms to obtain massive algorithm speedups. Tensor field creation is the bottleneck of many NPR algorithms that need the tensor values at each pixel location. In

this subsection I suggest the use of these techniques to accelerate the process of tensor field creation for a composed AR image. There are two major steps of computing tensor values: the initial tensor field creation, and tensor field pyramid creation.

To compute an initial tensor field for a composed AR image, we use the image gradient operator to get the magnitude and orientation of each pixel, and then keep only the strong pixels whose magnitudes are bigger than a predefined threshold. This process can be parallelized by computing the magnitude and orientation of many pixels at the same time. One possible method to implement this is using GLSL. In our previous work [10], we create a black-and-white sketch by coloring only the strong edge pixels in GLSL. The process to compute initial tensor values is similar to this using GLSL. We create a float-point target texture (i.e., 32 bits per color channel) to store the tensor values we found from a composed AR frame texture. The magnitude and orientation of a pixel are stored in the red and green channels in the target texture. We override the fragment shader to compute the tensor values for each pixel and store the results to the target texture. The fragment shader is shown below.

```
1  uniform sampler2D texture;
2
3  void main()
4  {
5      float offset = 1.0f/512.0f;
6     vec2 texcoord = gl_TexCoord[0].xy;
7
8      // find the gray-scale intensity of the center pixel and its 8
           neighbors
9     float ctr = texture2D(texture, texcoord).r;
10     float top = texture2D(texture, texcoord + vec2(0, offset)).r;
11     float bottom = texture2D(texture, texcoord + vec2(0, -offset)).r;
12     float left = texture2D(texture, texcoord + vec2(-offset, 0)).r;
13     float right = texture2D(texture, texcoord + vec2(offset, 0)).r;
```

```
14      float topleft = texture2D(texture, texcoord + vec2(-offset, offset))
            .r;
15      float topright = texture2D(texture, texcoord + vec2(offset, offset))
            .r;
16      float bottomleft = texture2D(texture, texcoord + vec2(-offset, -
            offset)).r;
17      float bottomright = texture2D(texture, texcoord + vec2(offset, -
            offset)).r;
18
19      // compute the magnitude and orientation for this pixel
20      float Gx = 2 * right - 2 * left + topright - topleft + bottomright -
             bottomleft;
21      float Gy = topleft + 2 * top + topright - bottomleft - 2 * bottom -
            bottomright;
22
23      float P = sqrt(Gx * Gx + Gy * Gy);
24      float theta = atan(Gy/Gx);
25
26      // encode magnitude and orientation of this pixel to red and green
            channel in target texture
27    gl_FragColor = vec4(P, theta, 0, 1);
28  }
```

Once we calculate the initial tensor field we could then create the tensor pyramid recursively using GLSL. The tensor values are encoded in the color channels of textures in each pyramid level. To create a new level of the tensor field we create a new target float-point texture which is half-size of its lower level texture, and then assign values to each pixel in the target texture by averaging the tensor values of a 2×2 area in the lower level texture.

However there is some drawbacks of using GLSL to implement the parallelized algorithm of tensor field creation. First the tensor values are encoded in float-point

texture and computed in the fragment shader. The routine is difficult to debug and hard to understand and maintain. Second this method is not flexible for adding more values to the textures, since there are only four color channels that can be used for each pixel. This is inconvenient if we need to expand this algorithm in the future to compute more than four components at once. (This number limit can be bypassed if we use more than one textures to store higher-dimensional data for each pixel. However this approach will bring more confusion and difficulties to the implementation.)

Another method of parallelizing the tensor field creation is using CUDA. CUDA provides a more comprehensive data structure for parallel computing. The tensor values can be computed and stored in an array in GPU memory. CUDA divides computing resources of multi-core GPUs into grids, and each grid contain multiple threads. We compute the tensor values for multiple pixels in the threads in parallel.

To compute the initial tensor field we can put pixel data of the composed AR frame in a device array. We can also create a target array to store the tensor values of the result. There are several different ways of dividing the computing workload: row style, column style and pixel style. In the row style we divide the 2D data array of the AR frame to rows and compute the tensor values for each row of pixels in a CUDA call thread. The code for computing tensor values in parallel in row style is shown below.

```
1 // CUDA code for computing tensor values.
2 __global__ void
3 d_init_tensor_nontex(float *grid, float *mag, float *ori, int w, int h)
4 {
5   // row index y
6   int y = blockIdx.x*blockDim.x + threadIdx.x;
7
8   mag = &mag[y*w];
```

```
9   ori = &ori[y*w];
10
11  // compute magnitude and orientation for each pixel in y-th row
12  for (int x = 0; x < w; x++) {
13   float Gx = 2 * grid[y*w+(x+1)%w] - 2 * grid[y*w+(x-1+w)%w] + grid[((y
        +1)%h)*w+(x+1)%w] - grid[((y+1)%h)*w+(x-1+w)%w] + grid[((y-1+h)%h
        )*w+(x+1)%w] - grid[((y-1+h)%h)*w+(x-1+w)%w];
14    float Gy = grid[((y+1)%h)*w+(x-1+w)%w] + 2 * grid[((y+1)%h)*w+x] +
         grid[((y+1)%h)*w+(x+1)%w] - grid[((y-1+h)%h)*w+(x-1+w)%w] - 2 *
         grid[((y-1+h)%h)*w+x] - grid[((y-1+h)%h)*w+(x+1)%w];
15
16    mag[x] = sqrt(Gx * Gx + Gy * Gy);
17    ori[x] = atan(Gy/Gx);
18  }
19 }
```

Similarly we can do this in the column style or pixel style. Depending on the data to be accessed and the multi-core GPU itself some access style could be faster than the others. We use different routines to access the memory. Some access style may have fewer cache misses and better memory coherence in data fetching. Hence some approaches may provide a greater speedup.

Besides the ways of dividing the workload, there are two different memory access styles: global memory access and texture access. Global memory access is access of the data in a device array by index. In texture access we first store the data to a 2D texture, and then access data by normalized texture coordinates (s, t). Hence we have $3 \times 2 = 6$ possible combinations (3 ways of dividing workload, and 2 ways of data access) to compute the initial tensor values in parallel. It is worth finding the optimal combination for our algorithm in modern GPUs.

We can create the tensor pyramid after we get the initial tensor field. We can then create 2D data array for each level of the pyramid. Each array is quarter size

of the array in the lower level. The tensor values of a pixel in an array are computed in parallel by averaging the tensor values of correspondent pixels in the lower level. Similarly we have 6 combinations of possible routines for creating a new pyramid level.

6.2 Hybrid Non-Photorealistic Rendering Algorithms in Augmented Reality

In Chapter 3, 4 and 5 we present two image space algorithms and one model space algorithm to maintain coherence when rendering NPR effects in AR video. The proposed model space NPR algorithm achieves better coherence in the final rendering than the image space algorithms. However, an advantage of image space algorithms for AR is that they do not require any additional information about the scene beyond what would normally be provided by the camera tracking software. Model space algorithms can be applied to if we have not only the 3D graphical models of the virtual content, but also the complete geometric information of the scene. Complete real-time construction of the 3D geometry of real world scenes is still an active research topic in the computer graphics and computer vision areas.

A hybrid NPR algorithm with coherence is also possible in addition to the image space and model space algorithms. To achieve coherent results, we apply an image space algorithm to the video content and a model space algorithm to the graphics content, and combine the rendered results together to obtain the final composed AR images.

We used the direct combination of image space and model space algorithms to render AR videos in this dissertation. However, we believe there is more information that can be extracted from the video and the computer graphics content. The information that we extracted from video, such as the orientation and position of markers to track the pose of the camera, can be potentially used to refine the rendering of the graphics content. For example, we place brush stroke textures on the 2D image in the

final rendering. It may be possible to obtain more accurate tensor information from models and use this information to smooth the overall tensor field for the entire AR frame. It is also possible to use the information to help clip the brush stroke textures along the boundary between the video and graphics content.

It should also be possible to leverage segmentation algorithms from computer vision to improve the coherence. We have already presented an averaging method to make the brush strokes more coherent, to cope with the incoherent tensor field. If we segment the video and detect matched objects and regions across frames, we should be able to smooth the tensor values in the corresponding regions to alleviate the flickering problem that is caused by the incoherent tensor field. Optical flow could also help create a more coherent tensor field. For example, if we use optical flow algorithms to find the movement of corresponding pixels between frames, the tensor value at each pixel in the current frame can be smoothed by the tensor at the corresponding pixel in the previous frame. This information can also be used to help clip brush strokes at the boundary between regions. These improvements may be useful for both the image space algorithms and the model space algorithms, since both of them need a coherent tensor field to create brush strokes.

REFERENCES

[1] Alliez, P., Meyer, M., and Desbrun, M., "Interactive geometry remeshing," in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, (New York, NY, USA), pp. 347–354, ACM, 2002.

[2] Battiato, S., Blasi, G. D., Farinella, G. M., and Gallo, G., "Digital mosaic frameworks: An overview," in *Computer Graphics Forum 07*, 2007.

[3] Bergen, J. R. and Hingorani, R., "Hierarchical motionbased frame rate conversion," *Technical Report of David Sarnoff Research Center*, 1990.

[4] Bourdev, L., "Rendering Nonphotorealistic Strokes with Temporal and Arc-Length Coherence," Master's thesis, Brown University, Providence, RI, 1998.

[5] Bourguignon, D., paule Cani, M., and Drettakis, G., "Drawing for illustration and annotation in 3d," in *Computer Graphics Forum*, pp. 114–122, 2001.

[6] Bousseau, A., Neyret, F., Thollot, J., and Salesin, D., "Video watercolorization using bidirectional texture advection," *ACM Transaction on Graphics*, vol. 26, no. 3, p. 104, 2007.

[7] Bowers, J., Wang, R., Wei, L.-Y., and Maletz, D., "Parallel poisson disk sampling with spectrum analysis on surfaces," *ACM Trans. Graph.*, vol. 29, pp. 166:1–166:10, Dec. 2010.

[8] Buck, I., Finkelstein, A., Jacobs, C., Klein, A., Salesin, D. H., Seims, J., Szeliski, R., and Toyama, K., "Performance-driven hand-drawn animation," in *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, NPAR '00, (New York, NY, USA), pp. 101–108, ACM, 2000.

[9] Chen, G., Esch, G., Wonka, P., Müller, P., and Zhang, E., "Interactive procedural street modeling," in *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, (New York, NY, USA), pp. 103:1–103:10, ACM, 2008.

[10] Chen, J., Turk, G., and MacIntyre, B., "Watercolor inspired non-photorealistic rendering for augmented reality," in *Proceedings of the ACM Symposium on Virtual Reality Software and Technology, VRST 2008, Bordeaux, France, October 27-29, 2008*, pp. 231–234, 2008.

[11] Chen, J., Turk, G., and MacIntyre, B., "Painterly rendering with coherence for augmented reality," in *IEEE ISMAR*, 2010.

[12] Chen, J., Turk, G., and MacIntyre, B., "Painterly rendering with coherence for augmented reality," in *IEEE ISVRI*, 2011.

[13] Claypool, M. and Tanner, J., "The effects of jitter on the peceptual quality of video," in *Proceedings of the seventh ACM international conference on Multimedia (Part 2)*, MULTIMEDIA '99, (New York, NY, USA), pp. 115–118, ACM, 1999.

[14] Cline, D., Jeschke, S., Razdan, A., White, K., and Wonka, P., "Dart throwing on surfaces," *Computer Graphics Forum*, vol. 28, pp. 1217–1226, 6 2009.

[15] Cohen, J. M., Hughes, J. F., and Zeleznik, R. C., "Harold: a world made of drawings," in *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, NPAR '00, (New York, NY, USA), pp. 83–90, ACM, 2000.

[16] Collomosse, J. P., Rowntree, D., and Hall, P. M., "Stroke surfaces: Temporally coherent artistic animations from video," *IEEE TVCG*, vol. 11, pp. 540–549, Sept./Oct. 2005.

[17] Comaniciu, D. and Meer, P., "Mean shift: A robust approach toward feature space analysis," *IEEE PAMI*, vol. 24, no. 5, pp. 603–619, 2002.

[18] Cook, R. L., "Stochastic sampling in computer graphics," *ACM Trans. Graph.*, vol. 5, pp. 51–72, Jan. 1986.

[19] Cunzi, M., Thollot, J., Paris, S., Debunne, G., Gascuel, J.-D., and Durand, F., "Dynamic canvas for non-photorealistic walkthroughs," in *Graphics Interface (GI'03)*, 2003.

[20] Curtis, C., Gooch, A., Gooch, B., Green, S., Hertzmann, A., Litwinowicz, P., Salesin, D., and Schofield, S., "Stylized augmented reality for improved immersion," *SIGGRAPH 99 Course Notes*, 1999.

[21] Curtis, C. J., Anderson, S. E., Seims, J. E., Fleischer, K. W., and Salesin, D. H., "Computer-generated watercolor," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, (New York, NY, USA), pp. 421–430, ACM Press/Addison-Wesley Publishing Co., 1997.

[22] Elber, G., "Interactive line art rendering of freeform surfaces," *Computer Graphics Forum*, vol. 18, no. 3, pp. 1–12, 1999.

[23] Fischer, J. and Bartz, D., "Real-time cartoon-like stylization of AR video streams on the GPU," in *Technical Report WSI-2005-18, University of Tubingen*, 2005.

[24] Fischer, J., Bartz, D., and Strasser, W., "Artistic reality: fast brush stroke stylization for augmented reality," in *VRST*, pp. 155–158, 2005.

[25] Fu, Y. and Zhou, B., "Direct sampling on surfaces for high quality remeshing," in *Proceedings of the 2008 ACM symposium on Solid and physical modeling*, SPM '08, (New York, NY, USA), pp. 115–124, ACM, 2008.

[26] Gooch, B. and Gooch, A., *Non-Photorealistic Rendering*. AK Peters Ltd, july 2001. ISBN: 1-56881-133-0.

[27] Gooch, B., Reinhard, E., and Gooch, A., "Human facial illustrations: Creation and psychophysical evaluation," *ACM Trans. Graph.*, vol. 23, pp. 27–44, Jan. 2004.

[28] Gortler, S. J., Grzeszczuk, R., Szeliski, R., and Cohen, M. F., "The lumigraph," in *SIGGRAPH*, pp. 43–54, 1996.

[29] Grossman, J. P. and Dally, W. J., "Point sample rendering," in *Rendering Techniques 1998*, pp. 181–192, Springer, 1998.

[30] Haller, M., "Photorealism or/and non-photorealism in augmented reality," in *ACM International Conference on Virtual Reality Continuum and its Applications in Industry*, pp. 189–196, 2004.

[31] Haller, M. and Landerl, F., "A mediated reality environment using a loose and sketchy rendering technique," in *ISMAR*, (Washington, DC, USA), pp. 184–185, IEEE Computer Society, 2005.

[32] Haller, M. and Sperl, D., "Real-time painterly rendering for mr applications," in *International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia*, 2004.

[33] Hanrahan, P. and Haeberli, P., "Direct wysiwyg painting and texturing on 3d shapes," in *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '90, (New York, NY, USA), pp. 215–223, ACM, 1990.

[34] Harris, C. and Stephens, M., "A combined corner and edge detector," *4th Alvey Vision Conference*, pp. 147–151, 1988.

[35] Hausner, A., "Simulating decorative mosaics," in *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 573–580, ACM, 2001.

[36] Hays, J. and Essa, I. A., "Image and video based painterly animation," in *NPAR*, pp. 113–120, ACM, 2004.

[37] Hertzmann, A., "Paint by relaxation," in *Proceedings of Computer Graphics International Conference*, pp. 47–54, 2001.

[38] Hertzmann, A., "Painterly rendering with curved brush strokes of multiple sizes," in *SIGGRAPH*, pp. 453–460, 1998.

[39] Hertzmann, A., "Fast paint texture," in *NPAR*, p. 91, 2002.

[40] Hertzmann, A. and Perlin, K., "Painterly rendering for video and interaction," in *NPAR*, pp. 7–12, 2000.

[41] Hertzmann, A. and Zorin, D., "Illustrating smooth surfaces," in *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, (New York, NY, USA), pp. 517–526, ACM Press/Addison-Wesley Publishing Co., 2000.

[42] Hertzmann, A. and Zorin, D., "Illustrating smooth surfaces," in *SIGGRAPH*, pp. 517–526, 2000.

[43] Horry, Y., Anjyo, K.-I., and Arai, K., "Tour into the picture: using a spidery mesh interface to make animation from a single image," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, (New York, NY, USA), pp. 225–232, ACM Press/Addison-Wesley Publishing Co., 1997.

[44] Kagaya, M., Brendel, W., Deng, Q., Kesterson, T., Todorovic, S., Neill, P. J., and Zhang, E., "Video painting with space-time-varying style parameters," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, pp. 74–87, Jan. 2011.

[45] Kalnins, R. D., Davidson, P. L., Markosian, L., and Finkelstein, A., "Coherent stylized silhouettes," in *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, (New York, NY, USA), pp. 856–861, ACM, 2003.

[46] 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., "Wysiwyg npr: drawing strokes directly on 3d models," in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, (New York, NY, USA), pp. 755–762, ACM, 2002.

[47] Kaplan, M., Gooch, B., and Cohen, E., "Interactive artistic rendering," in *NPAR*, pp. 67–74, 2000.

[48] Klein, A., Li, W., Kazhdan, M. M., Corrêa, W. T., Finkelstein, A., and Funkhouser, T. A., "Non-photorealistic virtual environments," in *SIGGRAPH*, pp. 527–534, 2000.

[49] Klein, A. W., Sloan, P.-P. J., Finkelstein, A., and Cohen, M. F., "Stylized video cubes," in *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '02, (New York, NY, USA), pp. 15–22, ACM, 2002.

[50] Kovacs, L. and Sziranyi, T., "Creating animations combining stochastic paintbrush transformation and motion detection," *Pattern Recognition, International Conference on*, vol. 2, p. 21090, 2002.

[51] Kowalski, M. A., Markosian, L., Northrup, J. D., Bourdev, L., Barzel, R., Holden, L. S., and Hughes, J., "Art-based rendering of fur, grass, and trees," in *SIGGRAPH*, pp. 433–438, 1999.

[52] Lake, A., Marshall, C., Harris, M., and Blackstein, M., "Stylized rendering techniques for scalable real-time 3d animation," in *Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, NPAR '00, (New York, NY, USA), pp. 13–20, ACM, 2000.

[53] Lévy, B. and Liu, Y., "Lp centroidal voronoi tessellation and its applications," *ACM Trans. Graph.*, vol. 29, pp. 119:1–119:11, July 2010.

[54] Li, H., Lo, K.-Y., Leung, M.-K., and Fu, C.-W., "Dual poisson-disk tiling: An efficient method for distributing features on arbitrary surfaces," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, pp. 982–998, Sept. 2008.

[55] Litwinowicz, P., "Processing images and video for an impressionist effect," in *SIGGRAPH*, pp. 407–414, 1997.

[56] Lloyd, S. P., "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, vol. 28, pp. 129–137, 1982.

[57] Lowe, D. G., "Object recognition from local scale-invariant features," in *ICCV*, pp. 1150–1157, 1999.

[58] Markosian, L., Kowalski, M. A., Goldstein, D., Trychin, S. J., Hughes, J. F., and Bourdev, L. D., "Real-time nonphotorealistic rendering," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, (New York, NY, USA), pp. 415–420, ACM Press/Addison-Wesley Publishing Co., 1997.

[59] Markosian, L., Meier, B. J., Kowalski, M. A., Holden, L., Northrup, J. D., and Hughes, J. F., "Art-based rendering with continuous levels of detail," in *NPAR*, pp. 59–66, 2000.

[60] Meier, B. J., "Painterly rendering for animation," in *SIGGRAPH*, pp. 477–484, 1996.

[61] Nehab, D. and Shilane, P., "Stratified point sampling of 3D models," in *Eurographics Symposium on Point-Based Graphics*, pp. 49–56, February 2004.

[62] Ostromoukhov, V., Donohue, C., and Jodoin, P.-M., "Fast hierarchical importance sampling with blue noise properties," in *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, (New York, NY, USA), pp. 488–495, ACM, 2004.

[63] Pastor, O. M., Freudenberg, B., and Strothotte, T., "Real-time animated stippling," *IEEE Comput. Graph. Appl.*, vol. 23, pp. 62–68, July 2003.

[64] Pessoa, S. A., de S. Moura, G., Lima, J. P. S. M., Teichrieb, V., and Kelner, J., "Photorealistic rendering for augmented reality: A global illumination and BRDF solution," in *IEEE VR*, pp. 3–10, 2010.

[65] Praun, E., Hoppe, H., Webb, M., and Finkelstein, A., "Real-time hatching," in *SIGGRAPH*, p. 581, 2001.

[66] S. Green, D. Salesin, S. S. A. H. and Litwinowicz, P., "Non-photorealistic rendering," *ACM SIGGRAPH, NPR Course Notes*, 1999.

[67] Salisbury, M. P., Wong, M. T., Hughes, J. F., and Salesin, D. H., "Orientable textures for image-based pen-and-ink illustration," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, (New York, NY, USA), pp. 401–406, ACM Press/Addison-Wesley Publishing Co., 1997.

[68] Tolba, O., Dorsey, J., and McMillan, L., "A projective drawing system," in *Proceedings of the 2001 symposium on Interactive 3D graphics*, I3D '01, (New York, NY, USA), pp. 25–34, ACM, 2001.

[69] Turk, G., "Re-tiling polygonal surfaces," in *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '92, (New York, NY, USA), pp. 55–64, ACM, 1992.

[70] Turk, G., "Texture synthesis on surfaces," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, (New York, NY, USA), pp. 347–354, ACM, 2001.

[71] Wang, J., Thiesson, B., Xu, Y. Q., and Cohen, M., "Image and video segmentation by anisotropic kernel mean shift," in *ECCV*, pp. Vol II: 238–249, 2004.

[72] Wang, J., Xu, Y., Shum, H.-Y., and Cohen, M. F., "Video tooning," *ACM Transactions on Graphics*, vol. 23, pp. 574–583, Aug. 2004.

[73] Wang, Z. and Bovik, A., "A universal image quality index," *IEEE Signal Processing Letters*, vol. 9, pp. 81 – 84, 2002.

[74] Wei, L.-Y., "Parallel poisson disk sampling," in *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, (New York, NY, USA), pp. 20:1–20:9, ACM, 2008.

[75] Winkenbach, G. and Salesin, D. H., "Rendering parametric surfaces in pen and ink," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, (New York, NY, USA), pp. 469–476, ACM, 1996.

[76] Winkler, S. and Mohandas, P., "The evolution of video quality measurement: From PSNR to hybrid metrics," *IEEE Transactions on Broadcasting*, vol. 54, pp. 660–668, Sept. 2008.

[77] Winnemöller, H., Olsen, S. C., and Gooch, B., "Real-time video abstraction," in *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, (New York, NY, USA), pp. 1221–1226, ACM, 2006.

[78] Wood, D. N., Finkelstein, A., Hughes, J. F., Thayer, C. E., and Salesin, D. H., "Multiperspective panoramas for cel animation," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, (New York, NY, USA), pp. 243–250, ACM Press/Addison-Wesley Publishing Co., 1997.

[79] Zhang, E., Hays, J., and Turk, G., "Interactive tensor field design and visualization on surfaces," *IEEE TVCG*, vol. 13, no. 1, pp. 94–107, 2007.