INTERACTIVE CRAYON RENDERING FOR ANIMATION

A Thesis

by

HOWARD JOHN HALSTEAD IV

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2004

Major Subject: Visualization Sciences

INTERACTIVE CRAYON RENDERING FOR ANIMATION

A Thesis

by

HOWARD JOHN HALSTEAD IV

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

December 2004

Major Subject: Visualization Sciences

ABSTRACT

Interactive Crayon Rendering for Animation. (December 2004)

Howard John Halstead IV, B.S., Texas A&M University

Chair of Advisory Committee: Dr. Donald House

This thesis describes the design and implementation of an interactive, non-photorealistic rendering system for three-dimensional computer animation. The system provides a two-dimensional interface for coloring successive frames of animation using a virtual crayon that emulates the appearance of hand-drawn wax crayons on textured paper. The crayon strokes automatically track and move with three-dimensional objects in the animation to preserve temporal coherency of strokes from one frame to the next. The system is intended to be used as an interactive renderer in conjunction with third-party three-dimensional modeling and animation tools.

To my loving mother, Carol

ACKNOWLEDGMENTS

I would like to thank the members of my thesis committee, Dr. Donald House, Dr. Ergun Akleman, and Dr. Nancy Amato. Your guidance and patience throughout the development of this thesis is sincerely appreciated.

I would also like to express my thanks to the faculty, staff, and students of the Texas A&M Visualization Laboratory for providing an incredibly rich and stimulating environment. I wish for you all the very best in your future endeavors.

Finally, I offer my deepest gratitude to my wonderful wife. Thank you so much for your love and support. With you, Virginia, everyday is Christmas.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Over the past three decades, the computer graphics community has made remarkable progress in the quest for photorealism. Driven by the desire of researchers to prove the latest graphics techniques, the incredible advances in processing power, and the public's insatiable appetite for blockbuster movies, photorealistic techniques have developed to the point that computer-generated images can no longer be distinguished from actual photographs. Perhaps the greatest testament to their acceptance and widespread use is the fact that many audiences no longer even notice the presence of computer-generated images in television and movies.

As these advances in photorealism have taken place, computer scientists and artists have begun to acknowledge the deficiencies of photorealistic imagery for many forms of expression. The inherent abstraction, exaggeration and subjectivity of works rendered in traditional media are often lost in the detail and complexity of synthetic, photoreal images. In response, many researchers have started to focus on developing rendering techniques that capture these characteristics of traditional media. The investigation and development of these rendering techniques has effected the emergence of a new research focus for the computer graphics community: non-photorealistic rendering.

But what exactly is non-photorealistic rendering (NPR)? Confusion arises due to the fact that NPR is defined in terms of what it is not: photorealistic rendering. The goal of non-photorealistic rendering is to create images that deviate from photoreal-

The journal model is *IEEE Transactions on Visualization and Computer Graphics*.

ism - images that offer an alternative method for expression and interpretation. It is perhaps easier to describe non-photorealistic rendering in terms of the images it produces. Research in non-photorealism has produced images that emulate watercolor, oil paints, pen-and-ink illustrations, pencil renderings, technical illustration, and cel animation among others.

Many NPR systems concentrate on the production of single images. In fact, the most widespread of all are the familiar paint programs installed with many computer operating systems. Other systems exist that focus on generating high- quality images in particular styles. For example, Salisbury developed a system supporting pen-and-ink illustration where the user ”paints” with stroke textures rather than drawing individual pen strokes [12]. Sousa and Buchanan implemented a system that produces high-quality images emulating the look of pencil renderings [15]. Curtis created a non-photorealistic renderer that automatically generates watercolor images using fluid dynamics [1]. Many more have been designed that emulate brush-and-ink [16], charcoal [5], and even the fantastic renderings of Dr. Seuss [4].

Though these systems are quite adept at producing single images, they are often unsuited for generating animation. For most interactive paint systems, the amount of work required to produce a single image makes animation production burdensome. These systems do not have facilities for alleviating the tedium of drawing hundreds (or even thousands) of frames. For other renderers that automatically process input images, the determination of stroke placement throughout an animation becomes problematic. If the software places the strokes randomly, the stroke coherency across frames is poor, resulting in images that appear to jitter too much. If strokes remain stationary with the respect to the image plane, the animation looks as if it is moving beneath a pane of textured glass, resulting in what many term “the shower door effect.”

Several researchers have undertaken the challenge of developing NPR systems for animation. Meier's work in automatic painterly rendering solved the stroke coherency problem using particles populated on three-dimensional objects [7]. Paint strokes in that system appear to stick to 3D objects as they animate. Litwinowicz developed a system for automatic impressionistic rendering of video segments that ensures stroke coherency using computer vision techniques [6]. The pixels in each frame of video are tracked from one frame to the next and paint strokes are programmed to automatically follow. Teece assumed a more interactive approach and developed a three-dimensional paint system to allow users to paint strokes directly on 3D objects [18]. In the same spirit, Walt Disney Feature Animation developed a production system, Deep Canvas, used by artists to paint the 3D environments for the animated feature film, Tarzan [3].

Though these systems leverage the processing power of the computer in the automatic generation of frames and solve the stroke coherency problem, their drawbacks lie in either their lack of user control or interface complexity. For example, in Meier's approach, the user does not have a direct method of specifying stroke placement, color, orientation, etc. Changes to stroke attributes must be effected through programmed surface shaders - a highly technical task.

Litwinowicz's automatic impressionistic rendering of video segments also does not provide a direct means of specifying stroke characteristics. The user specifies changes to the output through a set of rendering parameters. Though Teece's and Disney's systems provide a direct method for stroke definition, the complexity of working in three dimensions may hinder the work of some of their users. In a presentation by Eric Daniels of Disney [3], Daniels remarked that the process of using Deep Canvas went more smoothly when the user was both artistically and technically adept. Furthermore, all of these systems employ techniques that are quite different from the traditional process of working in 2D.

Two articles reflecting on the current state of non-photorealistic rendering systems highlight some important issues. In “Putting the Artist in the Loop” [14], Seims argues that though traditional paint programs provide good control and interactivity, they are burdensome when applied to animation work. On the other hand, recent automatic non-photorealistic rendering systems operate at too high a level of interaction to provide the artist with the necessary control. Seims states that, “the challenge is to find a level of interaction that minimizes the tedium of drawing and tracking individual brush strokes, and yet allows the artist full control” over the final output.

In “Computers for Artists Who Work Alone” [8], Meier argues for new research into NPR systems that are simple enough to be used by individual artists in home studios. New tools should not try to solve all problems with automatic software processes, but, instead, respect the strengths and weaknesses of both the computer and the artist. “A computer should help [artists] with the tedium or complexity of their work, but not significantly impact their style and preferred ways of thinking and working.”

The current state of research in non-photorealistic rendering for animation begs us to ask the question: how can computers support non-photorealistic animation without wresting simplicity and control from the user? To answer this question, three main problems need to be addressed: simplicity of interface, user control, and leveraging the power of the computer.

It is the opinion of the author that NPR systems should strive to provide interfaces that are as simple and intuitive to use as a pencil and paper. There is no doubt that many artists shy away from computers because of the complexity introduced when working with them. Software that is comfortable and intuitive allows the artist to spend more time creating and less time struggling. This can be achieved through integrating a system in a way that compliments the existing process rather than replacing it.

The author also believes that NPR systems need to provide the user with intuitive, direct control over the final appearance of the animation. Output generated by many systems cannot be edited directly by the user. Many rely on simulations that take a set of input variables and generate output that may only be altered by adjusting that initial set of parameters. A frustrating amount of time can be spent coaxing the computer into producing the desired output.

My position is that NPR systems should strive to identify repetitive and tedious operations and assume their responsibility. Though interactive systems provide much flexibility, many do not adequately leverage the processing power of the computer. These systems cannot offset the work required to draw hundreds of frames by hand. On the other hand, research has shown that automatic processes often make poor artistic decisions [13]. As a result, fully automated computer art has yet to rival the expression and impact of work by experienced artists. Therefore, a successful system should not rely on an algorithm to make artistic decisions for the user. Instead, the computer should be leveraged in a way that frees the artist to concentrate on solving creative problems.

In response to these issues, a two-dimensional painting system for three- dimensional computer animation could be developed that supports the artistic expression of the user while providing automatic rendering of frames. This system would allow the user to paint directly upon images output from an external 3D modeling and animation package. The paint strokes could be automatically updated in adjacent frames to give the appearance of the strokes moving with the objects they represent.

This system would allow full artistic expression of strokes in the traditional environment of two dimensions thereby avoiding the complexity of working in three dimensions. Automatic tracking of strokes would allow the computer to easily generate new frames. Not only would this reduce the workload of the user, but it would also ensure stroke coherency throughout the animation. Furthermore, these new frames could be fully editable by the user given the same tools used to define the initial frames of the animation.

Such a system would leverage the power of the computer while maintaining simplicity and control for the user. It could be designed as a pipeline of the following subsystems: a three-dimensional modeling and animation package, a 3D scene encoder, a painting interface, a stroke tracker, and a non-photorealistic renderer.

The three-dimensional modeling and animation system would be used to define, articulate, and animate models and cameras in a virtual 3D world. Such a system could make use of traditional 3D packages or possibly other systems that may be more suitable and easier to use for non-photoreal applications.

The 3D scene encoder would output reference images containing both color and geometric data of the 3D scene for each frame of the animation. The color data would be a camera-transformed snapshot of the 3D scene for one frame of animation that would be used as reference for painting the final images. The geometry data would encode information such as the location, parameterization, and identification of objects in the scene. This data would be integral to tracking strokes throughout the animation. Additional files could also be included that specify lighting information or other parameters required by the user.

The painting subsystem would be responsible for providing the user with an interface for specifying stroke positions and characteristics such as color, width, and pressure. Ideally the interactive painter would work closely with the renderer to provide interactive feedback of the appearance of the final images. Editing control would be provided here also.

The stroke tracker would be responsible for using the geometry data in the reference images to correlate the strokes with 3D objects in the scene so that they may be updated for each frame of animation. As the animation progresses, this system calculates the new positions of all strokes and sends that information to the renderer.

The rendering subsystem would be responsible for rendering the user strokes in a specified style or styles. Examples could be oil paint, charcoal, pastels, or crayon. Figure 1 shows the flow of information from one system to the next.

This thesis describes the design and implementation of such a non-photorealistic rendering system for generating animation that appears hand-drawn with wax crayons. To provide a more tractable problem, this research specifically focuses on the development of the last four components of the system: the scene encoder, the interactive painter, the stroke tracker, and the rendering subsystem.

The system described is analogous to an interactive coloring book for three-dimensional animation. The user colors on top of images output from a 3D animation package with virtual crayons. The crayon strokes are automatically correlated with the geometry in the animation. When the user goes to another frame (i.e. turns the page), the crayon strokes track the geometry and move with them.

This thesis provides an opportunity for investigating solutions to the problems of simplicity and artistic control in non-photorealistic rendering for animation. Specifically, the following techniques are explored:

- Painting of three-dimensional scenes in two dimensions (simplicity).
- Automatic correlation of two-dimensional crayon strokes with three- dimensional objects to preserve temporal coherency of crayon strokes across frames (leveraging the computer).
- An intuitive interface providing simple and direct control to the user (control).

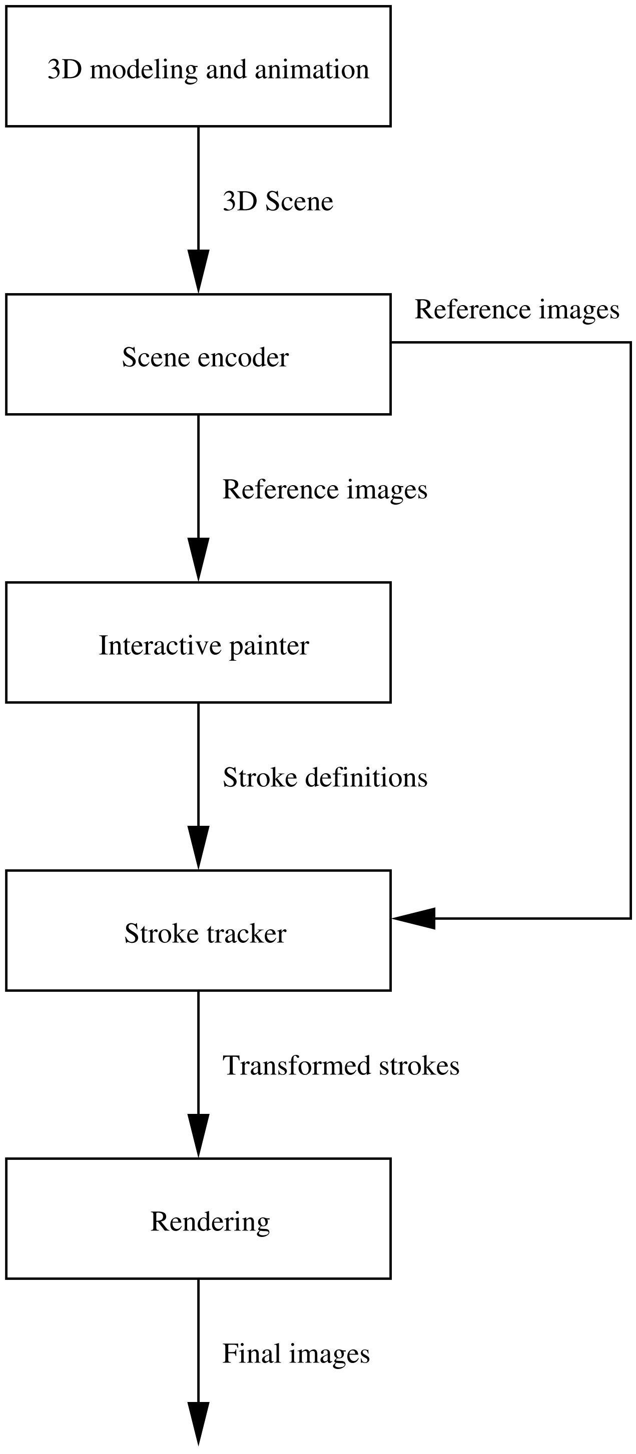


Fig. 1. System diagram.

- Emulation of the appearance of hand-drawn wax crayons on a textured canvas.

CHAPTER II

PREVIOUS WORK

A review of past NPR research reveals a variety of work concentrated on emulating traditional rendering styles such as pen-and-ink, oil paint, watercolor, and charcoal. Of particular interest to the present study are two studies that attempt to emulate the look of pencil renderings. These studies model the interaction of pencil and paper to achieve the characteristic “grainy” look that is also shared by crayon. The first study used a physically-based approach using volume rendering. The second opted for a more observational approach that yields nice results at interactive rates.

Takagi et al. [17] proposed a volumetric method for simulating the appearance of colored pencil drawings. Their method treats paper as a volume in which the pencil can enter to distribute pigment. The paper volume is initially modeled using 3D primitives corresponding to the pulp and loading matter. A random network of deformed cylinders generates a net of fibers that represents the pulp. The loading matter is added as deformed disks that are placed into gaps formed by the pulp fibers. The primitives are then 3D scan converted to generate a volume suitable for modeling paper.

Using an offset distance accessibility function developed by Miller [9], the authors determine which voxels in the paper volume are accessible to the tip of the pencil. When the tip comes into contact with an accessible voxel, pigments are deposited onto the paper simulating the effects of friction. If the accessible voxel contains loading matter or pigment, even more pigment is deposited. The resulting system produces impressive renderings, though at non-interactive rates.

In related research, Sousa and Buchanan [15] developed a system that produces quality pencil renderings at interactive rates. Rather than relying on a highly accurate, physically-based simulation, the authors developed an observational model of paper-pencil interaction. Paper is modeled as height fields that are either procedurally generated [1, 19] or digitized from paper samples. Each location on the paper can accumulate pencil lead through friction between individual paper grains and the pencil tip. The shape of the pencil tip, the pressure applied to the tip, and the pencil hardness all contribute to the amount of lead deposited. Furthermore, not only does the pencil deposit lead, it also alters the paper heightfield by destroying grains.

Sousa and Buchanan integrated their work into an interactive illustration system that supports the use of erasers and blenders. An automatic rendering system was also developed that takes reference pictures as input. For each pixel of a reference picture, the pencil-paper model is evaluated using the intensity value of the pixel to modulate the pressure applied to the pencil.

Past research in non-photorealistic rendering systems for animation are illustrative in their handling of stroke coherency. One important study that directly investigated the stroke coherency issue was Barbara Meier's research in painterly rendering for three-dimensional animation [7]. In Meier's system, three-dimensional objects are populated with particles that stick to the surface of the geometry. These particle sets are transformed with the 3D geometry and rendered as brush strokes in the 2D image plane. Properties of strokes, including color, shape, size, texture, and orientation are encoded in a set of reference pictures. Attributes for each particle are determined by looking at the same screen space location in the reference picture as that of the particle. Meier's approach avoids the shower-door effect when strokes remain stationary to the image plane, and ensures frame-to-frame coherency of stroke positions.

Peter Litwinowicz [6] developed a system for automatically processing video segments to create an animation that appears to be hand painted in an impressionist style. Brush strokes are automatically generated based on user-defined parameters to cover the first image of the sequence. The color for each stroke is calculated by sampling the pixel in the reference image corresponding to the center position of the stroke. The gradient direction of each video image can be used to derive orientation for strokes. And random distributions for stroke attributes such as length, radius, and color are applied to give the image a more hand-crafted look.

To ensure stroke coherency, Litwinowicz employs the use of computer vision techniques to track the movement of each pixel from one frame to the next. This creates a vector field that can be used to displace the strokes from one frame to the next. As the animation progresses, new strokes are automatically added to areas lacking enough coverage while others are removed in areas becoming too dense. To help preserve detail and silhouettes, strokes are clipped against edges derived by image processing techniques from the video segment.

CHAPTER III

APPROACH

For this thesis, the Crayon Animation Tool (CAT) was developed to provide a test bed for investigating solutions to the problems of simplicity and artistic control in non-photorealistic animation. The tool operates analogously to a traditional coloring book. The artist colors on top of reference images of a 3D animation as he would if given pages of line art in a coloring book; the difference is that all strokes in the system are automatically correlated with the objects in the animation. The artist may flip to any frame of the animation to add new strokes, and all existing strokes are updated to move with the objects.

CAT is split into three subsystems: an interactive painting system, an outline generation subsystem, and a batch renderer. The interactive painting system provides facilities for defining crayon strokes for the animation. Because of difficulties with outlining objects in the animation, a separate outline generation system was created to define outline strokes. The strokes output from both of these systems are combined by the batch renderer to output the final frames of the animation.

III.1. Interactive coloring

Since the focus of this thesis is on rendering, it is assumed that 3D models and animation are specified in an existing animation package. The architecture of the coloring system is only loosely coupled with how the initial animation is developed and can support a variety of methods. In the implementation described here, the animation is defined in a system built specifically for photorealism, but there is no reason why other modeling and animation systems could not be used instead.

The data flow of the interactive system is shown in Figure 2. Modeling and animation defined within an external 3D package is imported into the reference image generator. The reference image generator provides a bridge between external 3D packages and the rest of the system. For each frame of animation, the generator outputs a single reference image.

Each reference image includes a color buffer and a geometry buffer. The color buffer is a typical rendered image of the 3D scene. This image is displayed to the artist when painting to provide a visual reference for the placement of crayon strokes. The geometry buffer provides a map for each pixel in the color buffer to a camera-independent coordinate system of the corresponding rendered model. This map provides all of the necessary data to track crayon strokes from one frame to the next. Each reference image must include these two buffers, but other buffers could be included to encode additional information. Thus, the only requirements of the modeling and animation system are that they be capable of producing data for the color buffer and geometry buffer.

The main design goal of the interactive painter is to provide the user with a two-dimensional, interactive interface for specifying crayon strokes for the final output images. The interface is designed to be simple to use so as to approximate the working conditions of traditional methods as much as possible. A color buffer from a single reference image is displayed to the user along with coloring tools. The color buffer actually lays beneath the virtual canvas of the final output image but is not included in the final rendering output. This design allows the artist to use the color buffer as a guide in specifying crayon strokes.

To begin drawing crayon strokes, the artist specifies characteristics for the stroke such as color, pressure, and width using controls provided in the painter. To draw the stroke, the user places the mouse pointer over any object depicted in the color

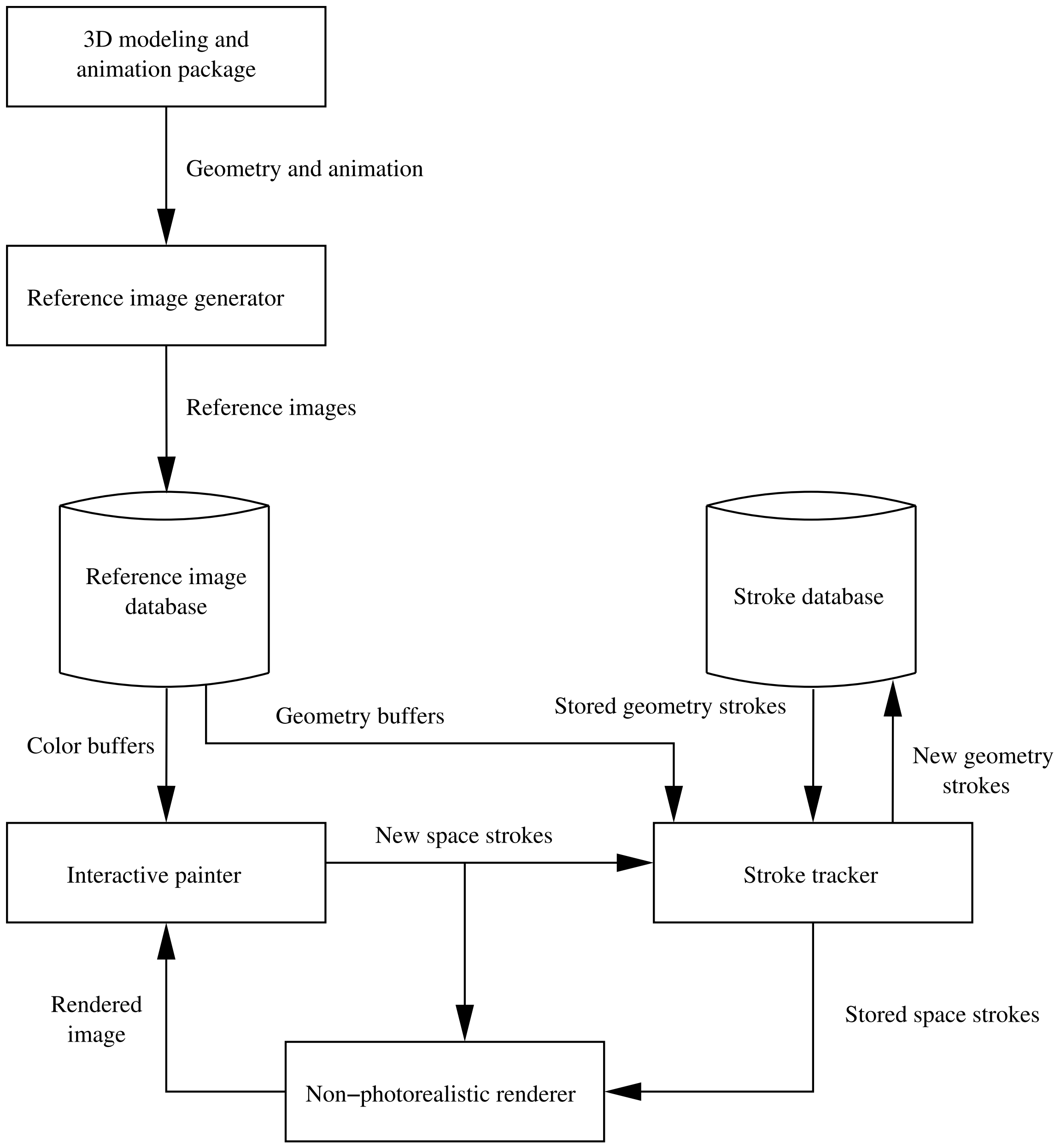


Fig. 2. Architecture of the interactive paint subsystem.

buffer image, clicks and drags. As the artist drags, the mouse is sampled and the resulting set of x-y points are stored as a new stroke. The new stroke is then sent to the rendering subsystem to be drawn. The renderer draws the new stroke and the painter updates its display with the result.

Though the research presented describes the implementation of a system for generating images that appear rendered with crayon, rendering subsystems are interchangeable within this architecture. However, the rendering technique used should be able to provide results at interactive rates if at all possible. The ability to see the final effects immediately is crucial to providing a user experience that is intuitive and enjoyable. In those situations that require time-consuming simulation, a proxy rendering subsystem could be provided that approximates the results of the final output.

There are two data representations for crayon strokes in the system. When new strokes are created, they are represented internally as x-y coordinate pairs and are referred to as "space strokes". However, in order to support the tracking of these strokes in adjacent frames, they are converted to a camera- independent representation called "geometry strokes". The stroke tracker is responsible for this conversion.

The stroke tracker converts space strokes to geometry strokes and vice versa. As mentioned earlier, for each pixel location in the reference image color buffer, there is a corresponding pixel in the reference image geometry buffer. In this implementation, each pixel in the geometry buffer encodes the u coordinate, v coordinate, and unique ID number of the NURBS model rendered at the corresponding location in the color buffer. For example, consider a NURBS sphere rendered in the color buffer at pixel location (x, y). The same pixel location (x, y) in the geometry buffer has the u and v coordinates of that sphere, as well as an ID number to identify that sphere from other geometry in the scene. Space strokes are necessarily frame-dependent, so all strokes are stored in the frame- independent geometry stroke representation.

Over the course of the animation, surfaces may appear and disappear. To handle these cases, the system provides the user the ability to draw strokes in any frame of the animation. When moving to a new frame, the appropriate reference image and strokes are updated to reflect the new frame. The user may then create new strokes for surfaces that have not been drawn with crayon yet.

III.2. Outline strokes

Artists will often outline an object to separate it from the background and other objects in the scene. Prominent features of an object may also be drawn specifically to help convey form. In this system, outlines and prominent features may be rendered using special, outline strokes. An outline stroke is a special type of stoke that can be automatically rendered, but can only exist for a single frame of animation; if outline strokes are desired in other frames, they must be recreated for those frames. This special handling arises from the fact that many outline strokes cannot be correlated with the underlying three-dimensional models.

For example, the outline of the ball in Figure 3 remains stationary relative to the image plane as the ball rotates. The outline corresponds with different regions of the ball, so it is impossible to correlate the outline with a specific parameterization of the ball. This type of outline is called a silhouette outline.

However, for outlines highlighting interior features of an object, stroke correlation can be achieved. For example, the outline stroke in Figure 4, depicting the interior deformation of the ball, rotates with the ball. The outline corresponds to the same region of the ball, so it is possible to automatically correlate the outline. This type of outline is called an interior outline.

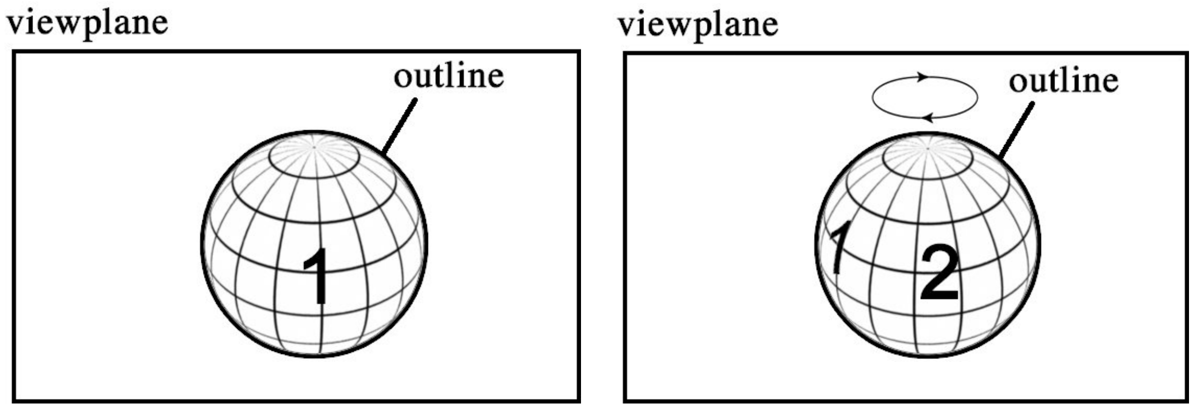


Fig. 3. Silhouette outlines.

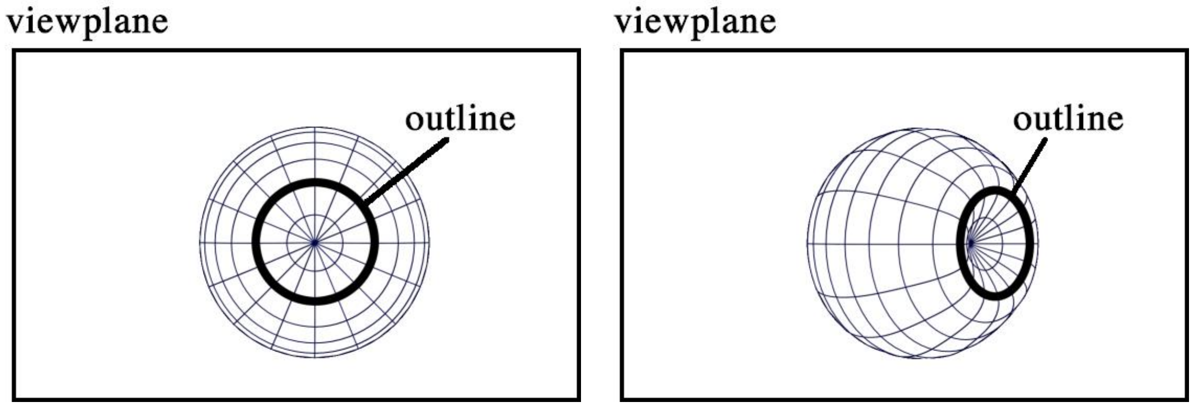


Fig. 4. Interior outlines.

Since silhouette outlines are difficult to express directly with the painter interface, it was decided that an automatic mechanism for generating them would be developed, driven by a set of user-defined parameters. And, in contrast to non- silhouette strokes, these strokes exist for only the current frame of animation and must be generated for every other frame of animation. Fortunately, strong silhouette stroke coherence arises from the temporal coherence of three- dimensional models.

To support the outline generator, reference image geometry buffers are augmented with depth and normal information. This data is loaded into the outline generator, where the user can specify characteristics for the desired strokes. The outline generator then analyzes the depth and normal data and outputs a set of space strokes for each frame of the animation. Figure 5 shows the architecture of the outline generator.

III.3. Batch rendering

Once the user is finished specifying strokes for the animation, he renders the animation using the batch renderer. For each frame of animation, the stroke tracker transforms the geometry strokes into space strokes. These are passed to the renderer along with the valid outline strokes for the current frame. The renderer draws the crayon strokes and outputs the final image. Figure 6 shows the architecture of the batch rendering system.

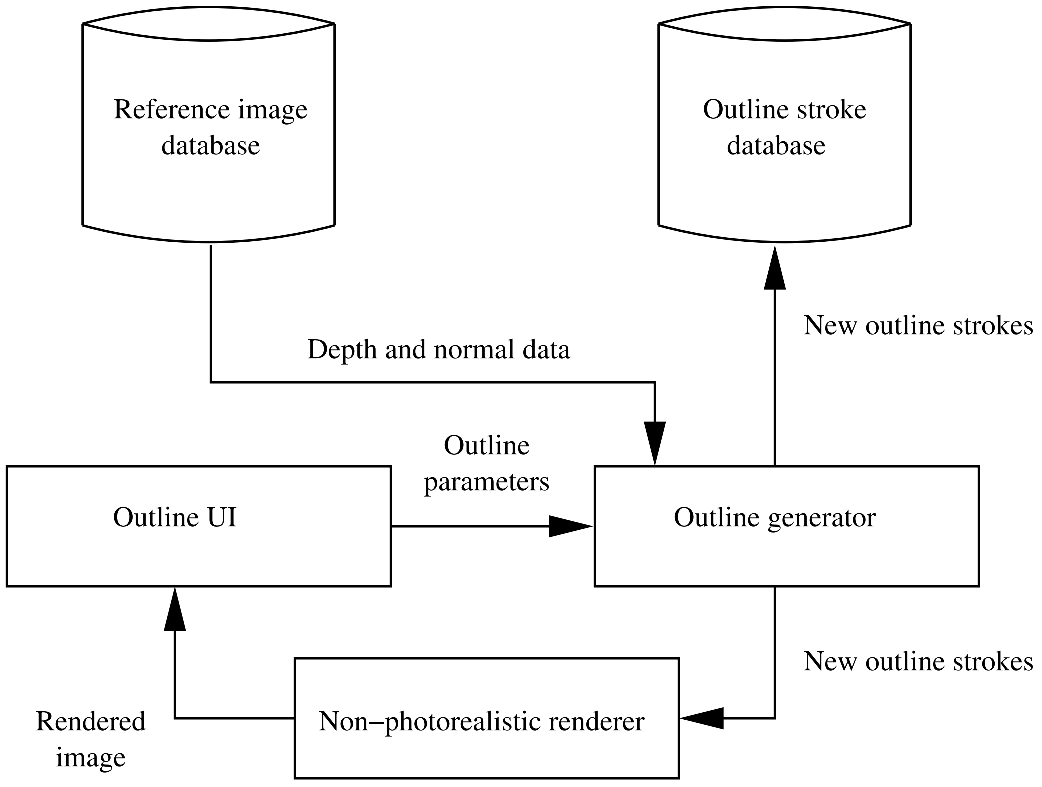


Fig. 5. Architecture of the outline generator subsystem.

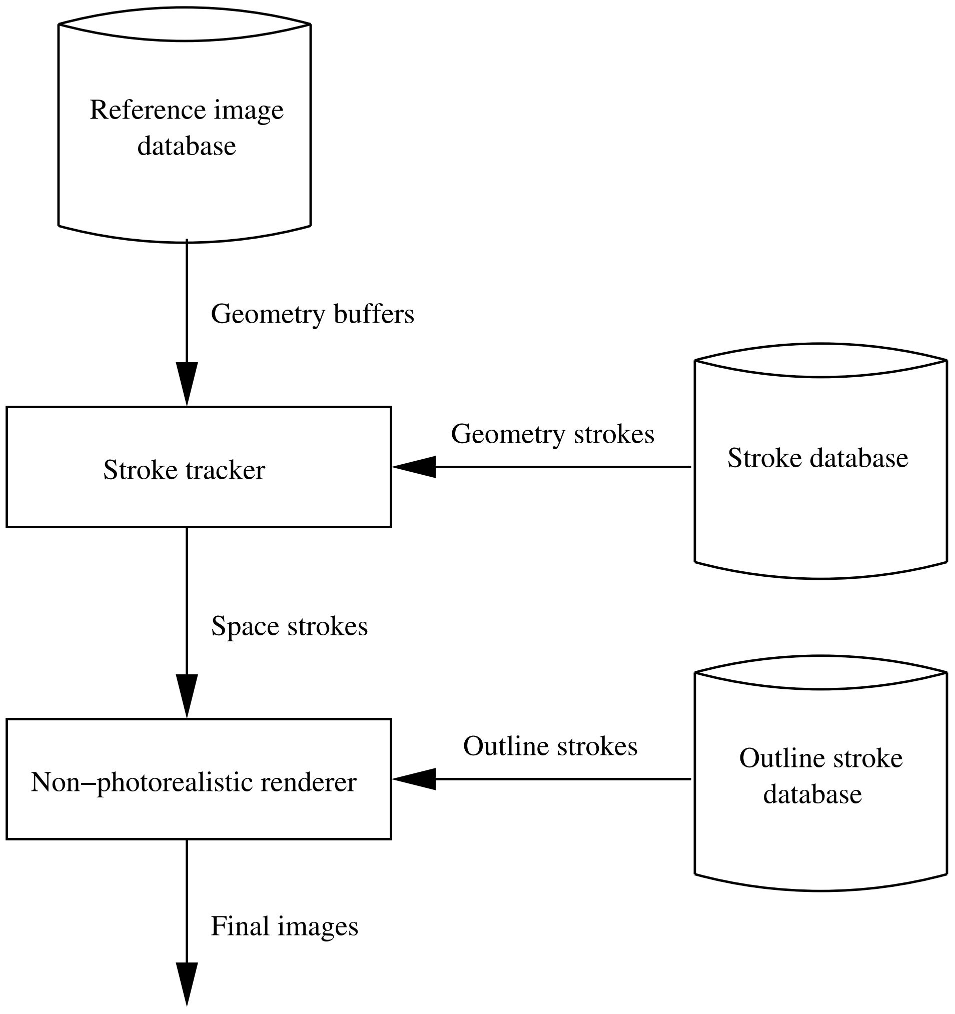


Fig. 6. Architecture of the batch rendering subsystem.

CHAPTER IV

REFERENCE IMAGES

For the prototype implementation developed for this thesis, Alias/Wavefront's Maya was used to define the initial models and animation for no other reason than the author's familiarity with the package. The reference generator was implemented using special surface shaders and Maya's general purpose renderer. In this implementation, reference images are split into several files: one for each buffer generated.

IV.1. Color buffer

The color buffer (Figure 7) is a typical shaded rendering of the 3D scene to be colored. Its primary use is to provide a guide for the placement of crayon strokes in the interactive painter. This image lies beneath the virtual canvas that the artist colors but is not included in the final rendered images. The image may be rendered in any style desired: line art, grayscale, shaded with lighting, etc.

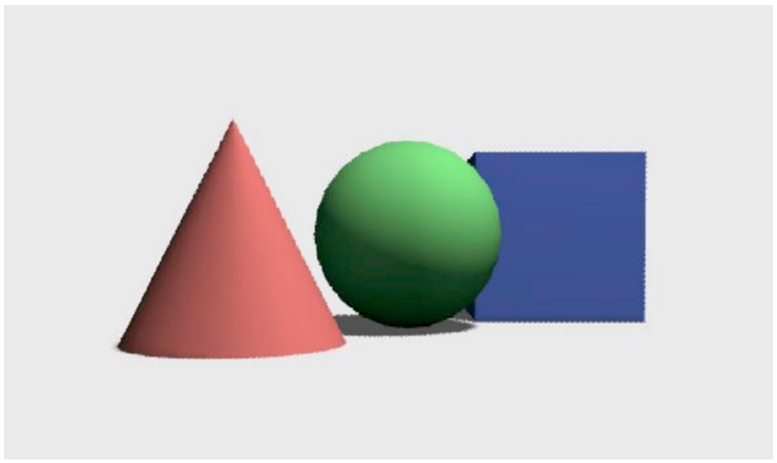


Fig. 7. A color buffer.

IV.2. Geometry (UVId) buffer

For simplicity, it was decided that the system implemented would only support NURBS geometry. Therefore, the geometry buffer encodes the u coordinate, v coordinate, and unique ID for each sample of the rendered NURBS models. This buffer is also called a UVId buffer (Figure 8) and is implemented using a special-purpose shader in Maya. In the first channel of the UVId buffer (Figure 9), the shader outputs the u coordinate of the sample being rendered. For the second channel, the shader outputs the v coordinate of the sample being rendered. In the third channel, a unique ID value for the surface is output. For demonstration purposes, this ID was assigned by hand, but an automatic method could be developed.

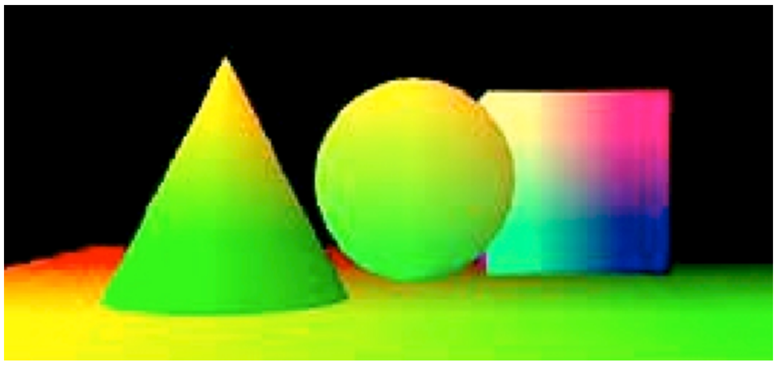


Fig. 8. A UVId buffer.

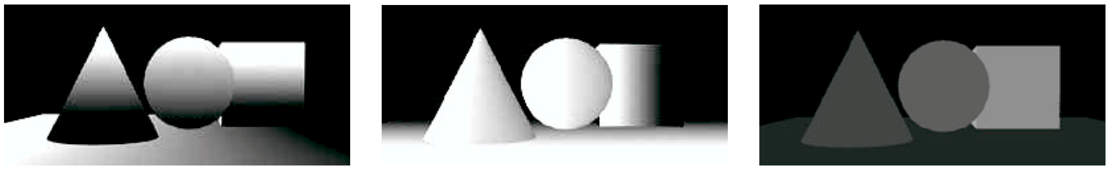


Fig. 9. Individual channels of a UVId buffer. From left, u, v, and Id.

Each pixel in the UVId buffer corresponds to a pixel at the same location in the other buffers of the reference image. When the user draws a crayon stroke over the color buffer, the corresponding pixels in the UVId are used to correlate the stroke with the geometry in the scene.

During development of the tool and animations, two issues with how UVId buffers are generated arose. First, to ensure accurate stroke tracking, a high bit depth for the UVId buffers is required. Originally, these buffers encoded u, v, and Id in 8-bits per channel. While testing the stroke tracking routines, it became apparent that 8-bits of data did not provide enough precision for encoding u and v. This was easily solved by rendering the geometry buffer to a 16-bit format.

The second area of trouble with UVId images occurs when the buffers are anti-aliased. Anti-aliasing along the edges of surfaces blends some of the pixels in the geometry buffer, resulting in erroneous results in stroke tracking. Unfortunately, despite turning off anti-aliasing controls in Maya, the blending of pixels was not totally eliminated. A post processing step was developed for the UVId buffers to remove the problem pixels from the buffer altogether. The method for handling this problem is not ideal. Each surface in the scene is rendered into an individual UVId buffer. Then the pixels in these individual buffers are compared to the problem UVId buffer. If a pixel cannot be found in the individual UVId buffers matching the current pixel in the problem image, the problem pixel's Id value is set to a special value not matching any surface in the 3D scene. This ensures that no stroke will match to this pixel. Since the number of these problem pixels is usually quite low, this proved to be an effective solution. This solution was satisfactory in completing the example animations for this thesis, but it is too time consuming (not to mention disk space intensive) to generate the required data. Ultimately, the rendering method used for generating the geometry buffer should support a single sample per pixel to ensure that no blending occurs.

IV.3. Depth and normal buffers

To generate outline strokes, support for two data sets are added to the reference images: depth buffers (Figure 10) and normal buffers. Depth buffers are grayscale images where the intensity of each pixel represents the depth of the sampled point in the scene with respect to the camera. Usually, brighter pixels represent points closer to the camera. Depth buffers are useful for finding boundaries between objects, since changes in depth are usually very gradual across an object but quite abrupt between objects. The disadvantages of depth buffers are that they are not very useful in detecting creases or folds of an object or finding boundaries between objects that abut at approximately the same depth. Normal buffers can be used to overcome this deficiency.

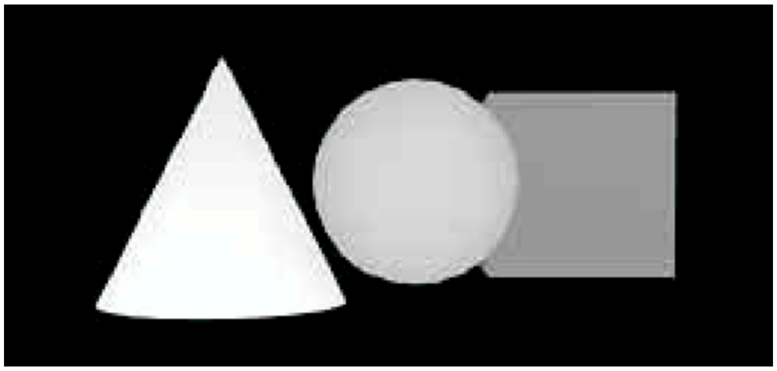


Fig. 10. A depth buffer.

Normal buffers encode changes in surface normal orientation across an object. The three components (XYZ) of the surface normal at each point are encoded in the three channels of an image. Creases and folds are represented by a rapid change in normal orientation. Areas where objects abut are also represented by a rapid change in normal orientation if the continuity across their boundary is not G1 continuous.

Most 3D packages can automatically generate depth information, but normal buffers will probably need to be custom generated. For rendering systems like Maya or RenderMan where access to geometry and shading variables is provided for each point on the object being sampled by the renderer, a straightforward method for generating a normal map can be implemented in a programmable shader:

$$R = (N_x + 1)/2 \tag{4.1}$$

$$G = (N_y + 1)/2 \tag{4.2}$$

$$B = (N_z + 1)/2 \tag{4.3}$$

where R, G, B are the red, green and blue components of the current sample, N is the surface normal at the point currently being sampled, and X, Y, and Z are the world coordinate axes, respectively.

CHAPTER V

CRAYON STROKE DEFINITION AND TRACKING

The inspiration for this thesis is the application of a coloring book paradigm to non-photorealistic rendering for animation. Analogous to a child coloring line art in a coloring book, the artist in this system draws crayon strokes on top of a reference image generated by a 3D package. However, these crayon strokes actually "stick" to an object as it animates. Given only a small number of pictures colored by the artist, the system can generate an entire crayon animation automatically.

A single reference image includes a color buffer and a geometry buffer. The color buffer is a simple color rendering of the 3D scene. This buffer is equivalent to the line drawings in a coloring book in that it provides visual reference to the artist for the placement of crayon strokes. The geometry buffer augments the color buffer by encoding data such as uv parameterization, surface Id, surface depth, and surface normal information for each sample in the color buffer.

V.1. Defining crayon strokes

Crayon strokes are defined by the artist using a mouse or stylus that is sampled regularly to generate a sequence of xy coordinates. This set of points defines the path of a crayon stroke and is called a "space stroke" to emphasize its spatial definition. Once input by the artist, the space stroke is passed to the renderer to be drawn on screen.

In addition to rendering, the system transforms each new space stroke into a frame-independent "geometry stroke". Rather than storing xy coordinates, each point in a geometry stroke stores a uv position and a surface Id. Exploiting the one-to-one correspondence between each pixel of the color buffer and the geometry buffer, the conversion from space stroke to geometry stroke is easily calculated by reading the u, v, Id data in the geometry buffer at each xy location in the space stroke.

This geometry stroke representation enables the tracking of strokes as objects move during the animation. For example, when the frame changes, the system searches the new geometry buffer for the xy location of each uvId point in the geometry stroke. The resulting sequence of xy coordinates defines a space stroke and is promptly sent to the renderer for drawing. The next section discusses the conversion from geometry to space stroke in more detail. Transforming geometry strokes

Consider a geometry stroke that consists of an ordered set of points defining a path for a crayon. Each point has a u and v coordinate and a surface Id. To transform the stroke into a space stroke and render it for the current frame, we must determine where to place each point in the framebuffer (i.e. the canvas). Therefore, for each point in the stroke we need to find the location of the pixel in the geometry buffer that most closely matches the u, v, and Id of that point. Once found, we transform the point to that location, and proceed to the next point in the stroke. Once all points have been transformed, we pass the resulting space stroke to the renderer so that it can draw a crayon stroke along that path.

V.2. Accelerating the transformation process

To accelerate the transformation process, we bucket-sort all u, v, and Id information stored in the geometry buffer. This scheme allows us to quickly determine if uv samples exist in specific regions of a surface's uv space. The resulting performance is much better than that of a brute-force, sequential search.

For each unique surface Id value found in the buffer, we create an NxN array of buckets that store the uv coordinates and xy position for each sample with matching Id. The buckets are indexed by a hashed u and v value. The hash function is designed to place neighboring u and v values into the same bucket. A uv pair is placed into a bucket using the following hash function

$$b_i = round(u * (N - 1)) \tag{5.1}$$

$$b_j = round(v * (N - 1)) \tag{5.2}$$

where b_i, b_j are the bucket indices and N is the length of one side of the array.

V.3. Stroke transformation

At this point we have a set of bucket arrays containing uv samples for each unique surface (Id) rendered in the geometry buffer. Our next step is to parse the list of strokes, and transform them to their correct positions in the image plane. For each point in a stroke, we search through the bucket arrays to find the xy location of the closest matching uv sample in the geometry buffer. A close match is determined by

$$\sqrt{(u - u_{sample})^2 + (v - v_{sample})^2} \leq e \tag{5.3}$$

where e is an error threshold. If no closest matches for the point are found, the point is not rendered and a discontinuity is introduced into the stroke. If multiple matches are found with the same error, the centroid of the corresponding xy locations is calculated and the point is transformed to that location. Once all of the stroke points have been transformed, the resulting space stroke is sent to the renderer. The renderer then draws a crayon stroke along the path.

V.4. Bucket array sizes and choosing an error threshold

We can calculate a bucket array size based on the error threshold value such that the maximum number of buckets that need to be searched is held constant at the current bucket plus it's eight neighbors. The formula for calculating N below

$$N = floor\frac{1}{e} \tag{5.4}$$

ensures that the “width” of each bucket does not exceed the error threshold. Therefore, in cases where no matches are found in the bucket that the sample is hashed to, we only need to search the eight neighboring buckets for possible matches.

Note that accuracy of stroke tracking should ultimately drive decisions for determining good values for e and N. For this thesis, e was determined empirically. However, a more general approach would probably require closer inspection of the screen resolution of each object. For example, finding the maximum distance between adjacent uvId samples and then halving that value may provide a decent per-object error threshold.

V.5. Algorithm for transforming geometry strokes

The algorithm for transforming geometry strokes to space strokes follows:

```
Generate bucket arrays and sort geometry buffer data;
for each stroke {
    for each stroke point {
        find bucket array with matching ID;
        identify bucket neighborhood with close matches;
        search bucket neighborhood for close matches
        if (close matches are found) {
           calculate XY location;
           add to crayon path;
        } else {
           introduce discontinuity into crayon path;
        }
    }
}
```

V.6. Stroke occlusion and clipping

As objects in the scene animate, crayon strokes may occlude one another. Consider the stroke drawn on the sphere in Figure 11. When this stroke is created, every sample point of the stroke maps to a pixel in the image. If the sphere rotates during the animation, there is a good chance that the region of the sphere where the stroke is drawn will begin to rotate out of view (i.e. the sphere occludes part of the stroke). Since the geometry buffer only stores data for visible surface points, there is no straightforward method of determining how the occluded portions of the stroke should be transformed. However, if we have a high enough sample resolution for the stroke, occlusion is easily handled by simply ignoring the occluded points: we only draw between points that are visible and adjacent to each other.

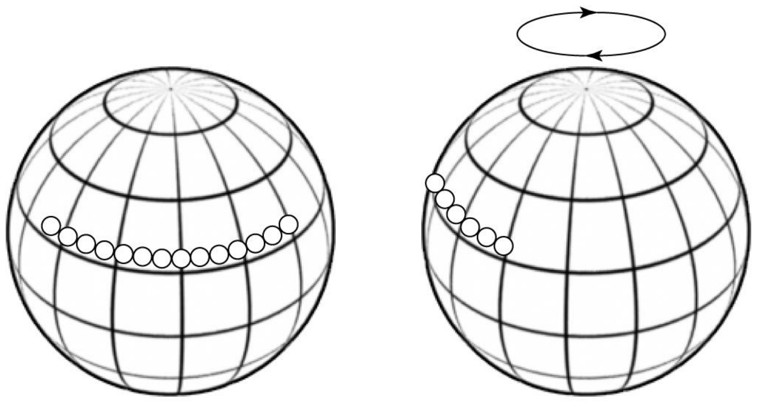


Fig. 11. Self-occlusion.

In Figure 12, we see that our occlusion scheme also works in situations where the stroke is occluded by another object. In this example, a crayon stroke is drawn across a surface. In later frames of the animation, another surface passes in front of the stroke and occludes it. The samples in the middle of the stroke cannot be mapped,

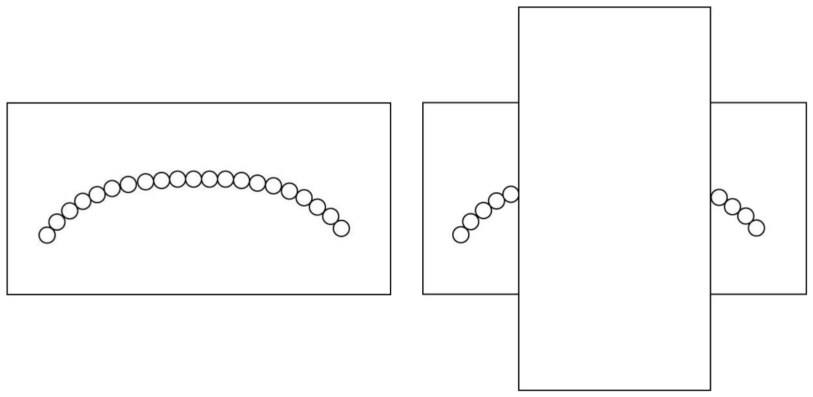


Fig. 12. Occlusion by another object.

so the path of the stroke is split into two parts.

CHAPTER VI

REAL-TIME EMULATION OF WAX CRAYONS ON PAPER

Emulating the look of crayon drawings requires that our rendering model account for at least a few characteristics of the medium that distinguish it from work rendered using other techniques. The most important characteristic simulated in this model is the tendency of crayon to highlight the texture of paper.

All paper has an inherent grain that can be visualized as a landscape of peaks and valleys providing a rough surface for removing pigment from a crayon. As a crayon is rubbed across the paper, the tip of the crayon slides across peaks and skips over valleys leaving a characteristic textured stroke. Modeling this low-level interaction between crayon and paper is necessary to achieve a characteristic crayon appearance.

Another modeled characteristic of crayon drawing is the reduction of friction due to wax buildup on the paper. As a crayon moves across the paper, wax is deposited along with pigment. This wax tends to reduce the friction generated between the crayon tip and paper surface, and gradually less pigment is deposited in areas already colored. Accounting for this enables the rendering model to emulate the effect of repeatedly covering an area of the paper so that it is filled with an even saturation of color (i.e. repeated strokes of the same pressure tend to fill uncolored areas without significantly affecting colored areas). This type of behavior is characteristic of working with crayon.

Of course, since we are modeling pigments, crayon colors interact according to the principles of subtractive color mixing. The rendering model accounts for this effect, providing the artist with intuitive feedback as he colors.

Accounting for the interaction of crayon and paper, the reduction of friction due to crayon wax buildup, and subtractive color mixing produces results that approximate the look of actual crayon drawing. The model presented is loosely based on Sousa and Buchanan's work in the modeling of graphite pencil materials [15]. The crayon and paper models used here are similar to the pencil and paper models they used. However, several changes have been made to the interaction of these models to account for the required crayon characteristics mentioned above and to remove undesired complexity.

VI.1. The paper model

The paper model is represented as a heightfield that can be synthesized using procedural methods or digitized samples. Several researchers have found that a combination of Perlin's noise [10] and Worley's cellular texture basis functions [19] can produce images that emulate the surface of paper quite well. On the other hand, Sousa and Buchanan suggest using digitized paper samples. Both methods produce a grayscale map where bright pixels represent peaks in the paper texture and dark pixels represent valleys.

Modeling the interaction of crayon and paper focuses on the contact of the crayon tip with individual paper grains. A paper grain, h(x,y), is located at pixel (x,y) in the paper image with a height equal to the normalized grayscale intensity of the pixel ($h(x,y) \in [0,1]$). When the crayon tip moves across the grain, the grain removes a portion of crayon material. The amount of material distributed to the grain is proportional to the pressure applied to the crayon and the grain's height. Higher paper heights get more pigment; lower paper heights get less.

VI.2. The crayon model

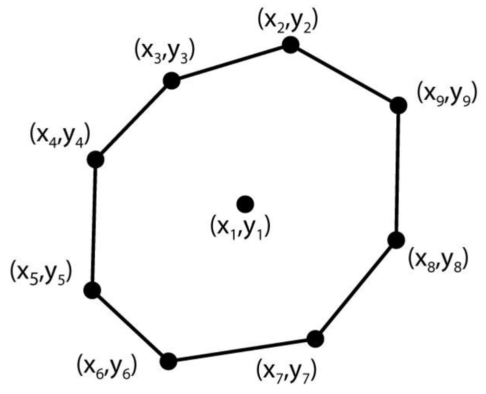


Fig. 13. Crayon tip shape.

The crayon model is comprised of a polygonal tip shape, a set of pressure distribution coefficients, and a color. The tip shape is defined as a polygonal surface

$$T_s = \{S, \{x_i, y_i\} : 1 \leq i \leq n\} \tag{6.1}$$

where S is a scalar factor that accounts for the diameter of the crayon and the coordinate pair, (x_i, y_i), is one of n vertices comprising the polygonal surface (Figure 13).

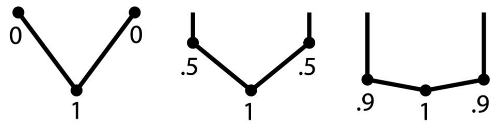


Fig. 14. Pressure coefficients change tip profile.

The pressure distribution coefficients determine what percentage of the tip surface makes contact with the paper. Higher coefficient values correspond to more surface contact. The set of pressure coefficients, P_c, is defined as

$$P_c = \{c_i, 1 \leq i \leq n\} \tag{6.2}$$

where c_i are pressure coefficients for each vertex of the polygonal tip shape.

This representation offers flexibility in defining the tip shape. For crayons, the shape is usually approximately circular unless the side of the crayon is used to draw. In this case, the tip is better approximated by a rectangular shape. The pressure coefficients provide an intuitive mechanism for shaping the profile of the tip as shown in Figure 14.

The shape of an actual crayon will change as the stroke is drawn across the paper. This change will certainly have some effect on the appearance of the stroke. However, the effect is probably minimal, and to avoid unnecessary complexity, this model assumes that the shape remains constant throughout the length of the stroke.

VI.3. Crayon-paper interaction

With the crayon and paper models defined, we can discuss the process of rendering a crayon stroke. A stroke path is sent to the renderer as an ordered sequence of points defining line segments. The interaction of the crayon and paper models is then evaluated at each point along this path. The algorithm proceeds as follows:

```
for each position along the stroke path {
    compute the pressure applied across crayon tip;
    identify paper grains beneath crayon tip;
    for each grain beneath crayon tip {
        calculate the depth of the crayon "into" the paper surface;
        calculate the maximum amount of material that could be deposited;
        calculate the percentage of the maximum amount to deposit;
        calculate the amount to deposit (before friction adjustment);
        attenuate amount deposited according to friction;
        deposit actual amount of material removed to grain;
    }
}
```

VI.4. Computing pressure and identifying paper grains

At each position along the stroke path, we begin by computing the actual pressure at each vertex of the polygonal tip surface. The pressure at each vertex is calculated as

$$p_i = p \times c_i \tag{6.3}$$

where $p \in [0, 1]$ is the pressure applied to the crayon by the user, and c_i is the pressure distribution coefficient for vertex i. Projecting the tip shape onto the paper surface (Figure 13), we bilinearly interpolate the pressure across the tip surface by scan converting each triangle in the tip (Figure 15). This scan conversion process also identifies which paper grains are beneath the crayon tip. For each grain, $h_{(x,y)}$, on the paper surface identified by the scan conversion, a bilinearly interpolated pressure, p_a (pressure applied), is calculated for the grain.

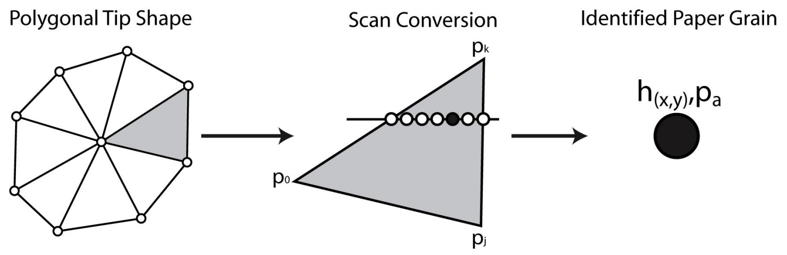


Fig. 15. Identifying paper grains in contact with the crayon tip.

VI.5. Processing the amount of material removed by the grain

Next, we need to determine exactly how much crayon material is removed by the paper grain. We first calculate the depth of the crayon, d_c, into the paper surface:

$$d_c = (1 - p_a) \times D_{lo} + p_a \times D_{hi} \tag{6.4}$$

where $D_{lo}, D_{hi} \in [0,1]$ are the user-specified values for crayon depth at a pressure of 0 and at a pressure of 1, respectively. These parameters are used to define paper roughness by identifying what range of grain heights is accessible by the crayon tip.

Next, the maximum amount of material, f_{max}, that could be deposited at any grain in the paper, given the current pressure, is calculated as

$$f_{max} = (1 - p_a) \times F_{lo} + p_a \times F_{hi} \tag{6.5}$$

where F_{lo} is the maximum amount of material that can be deposited at a low pressure of 0, and F_{hi} is the maximum amount of material that can be deposited at a high pressure of 1. Both of these values are supplied by the user. The percentage of f_{max}, $f_{\%}$, that can be deposited at the current grain is dependent on the crayon depth and the grain height, and is calculated as

$$h_c = 1 - d_c \tag{6.6}$$

$$f_{\%} = max(0, \frac{h_{(x,y)} - h_c}{(1 - h_c)}) \tag{6.7}$$

where h_c is the height of the crayon from the "bottom" of the paper surface. This equation models the observation that taller paper grains receive more material than shorter ones. And those paper grains that are below the depth of the crayon tip do not receive any material at all. The amount of material "bitten", f_b, by the paper grain is then calculated as

$$f_b = f_{\%} \times f_{max} \tag{6.8}$$

To account for a reduction in friction due to the crayon material already present, $f_{(x,y)}$, we scale f_b by the function, $f_a()$. $f_a()$ is a friction adjustment factor yielding values from 0 to 1. For this thesis, $f_a()$ was designed to reduce the amount of friction once $f_{(x,y)}$ reaches a saturation amount, f_{sat},

$$f_{sat} = M_s \times (f_{\%} \times f_{max}) \tag{6.9}$$

where M_s is a user-defined multiplier that controls how quickly a grain becomes saturated. We've shown that repeated strokes across a grain at constant pressure will deposit $f_{\%} \times f_{max}$ amount of material for each stroke (ignoring friction). Therefore, M_s can be interpreted as the number of strokes required to saturate a grain at a given pressure.

Once the grain becomes saturated, $f_a()$ gradually reduces the amount of material deposited according to an exponential function

$$f_a = \begin{cases} 1 & f_{(x,y)} < f_{sat} \\ e^{-\phi\left(f_{(x,y)} - f_{sat}\right)} & f_{(x,y)} \geq f_{sat} \end{cases} \tag{6.10}$$

where ϕ controls how quickly friction is attenuated, and $f_{(x,y)}$ is the total amount of material deposited at the grain. With an attenuation value from $f_a()$, the final value of f_b is then attenuated,

$$f_b = f_b \times f_a() \tag{6.11}$$

and the total amount of material at height $h_{(x,y)}$ is updated as

$$f_{(x,y)} = f_{(x,y)} + f_b \tag{6.12}$$

VI.6. Rendering

The color of the crayon is represented in CMY color space to accommodate subtractive color mixing. When an amount, f_b, of crayon material of color, c_b, is deposited at a paper grain, the new accumulated color becomes a weighted average of the current paper grain color and the color being added

$$c_{(x,y)} = \frac{c_{(x,y)} \times f_{(x,y)} + c_b \times f_b}{f_{(x,y)} + f_b} \tag{6.13}$$

where $c_{(x,y)}$ is the color of the paper grain $h_{(x,y)}$.

The final rendering of the paper requires converting the accumulated pigment color at each paper grain to RGB space, as well as calculating an opacity value to mix this color with the underlying paper color. Converting the pigment color to RGB is trivial:

$$red = 1 - cyan \tag{6.14}$$

$$green = 1 - magenta \tag{6.15}$$

$$blue = 1 - yellow \tag{6.16}$$

The opacity of the pigment color at each paper height is calculated as

$$\alpha_{(x,y)} = min(\ 1, \tfrac{f_{(x,y)}}{F_t}\ ) \tag{6.17}$$

where $f_{(x,y)}$ is the total amount of crayon wax deposited at $h_{(x,y)}$, and F_t is the user-supplied, maximum amount of material any paper height can hold. This should be considered a fairly loose interpretation of F_t, and the value of F_t is often adjusted to achieve the desired rendering results regardless of its physical meaning. When the paper is finally rendered, the color at each pixel (x,y), is calculated as

$$pixel_{(x,y)} = (1 - \alpha_{(x,y)}) \times C_{paper} + \alpha_{(x,y)} \times c^{rgb}_{(x,y)} \tag{6.18}$$

where C_{paper} is the user-supplied paper color and is the RGB equivalent of the paper grain color, $c_{(x,y)}$.

CHAPTER VII

GENERATING OUTLINE STROKES

The algorithm for generating outline strokes is based on a method developed by Curtis [2]. The first step in the process identifies edges in the scene for each frame of animation and writes them to an edge map. Next, a vector field is generated with vector orientations aligned along the identified edges. The edge map and vector field are then input to a physically-based particle system that traces the edge map with particles driven by the vector field. As the particles move along the edges, paths for outline strokes are created. Drag and random forces are modeled in the system to provide extra user control and emulate a hand-drawn look. The rest of this chapter investigates each step of the outline stroke algorithm in depth.

VII.1. Depth and normal buffers

To generate outline strokes, we require depth and normal buffers for each reference image of the animation. Depth buffers are useful for detecting boundaries between objects, since changes in depth are often quite abrupt in such areas. However, they are not useful in detecting creases or folds within an object or for finding boundaries between objects that abut at approximately the same depth. These shortcomings are addressed with normal buffers. Normal buffers encode changes in surface normal orientation and are useful for detecting creases and folds in an object. The boundaries between objects that abut at the same depth can also be readily detected if they do not exhibit G1 continuity across their border.

VII.2. Edge detection with convolution

A simple method for detecting edges in an image is to convolve the image with edge detecting kernels such as those of the Sobel filter [11]. The kernels for the Sobel filter are defined as:

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{7.1}$$

Now, if we let $I(x, y)$ be a grayscale image, we can convolve $I(x, y)$ separately to create two images:

$$\begin{aligned} I_x(x, y) &= I(x, y) * S_x \\ I_y(x, y) &= I(x, y) * S_y \end{aligned} \tag{7.2}$$

where $*$ indicates convolution. Convolution with the Sobel kernels approximates differentiation or calculating the gradient of the image. To create an edge map, we compute the magnitude of the gradient in both x and y directions and apply a simple threshold function to the result:

$$I_{mag}(x, y) = \sqrt{I_x^2(x, y) + I_y^2(x, y)} \tag{7.3}$$

$$E(x, y) = \begin{cases} 1 & I_{mag}(x, y) \geq T \\ 0 & I_{mag}(x, y) < T \end{cases} \tag{7.4}$$

$E(x, y)$ is a bitmap containing the edges of the grayscale image $I(x, y)$. The value of the constant T in the threshold function is a significant parameter. High values of T make edge detection very selective which helps to filter any noise in the image. This parameter is made available to the user to tweak the results of the edge detection process.

Edge detection on the depth and normal maps is performed independently of one another. The results are combined using a simple max function to create a final edge image:

$$E(x, y) = max(E_d(x, y), E_n(x, y)) \tag{7.5}$$

where E is the final edge image, E_d is the edge image generated from the depth buffer, and E_n is the edge image from the normal buffer.

VII.3. Generating a vector field

A vector field for the particle system is generated using a three-step process. First, the gradient of the depth and surface normal buffers is calculated and rotated by 90 degrees (actually this is the same as the normal of the gradient; this is used so that the vector field is aligned along edges). As in edge detection, the gradient is calculated through convolution with the Sobel kernels. The rotated gradient vectors are stored in a map that we will call $forcefield_1$. Second, we calculate the magnitudes of the gradient vectors in $forcefield_1$, store them in a temporary map, and then calculate the gradient of this map. This result is stored in $forcefield_2$. The final vector field is created by simply summing $forcefield_1$ with $forcefield_2$.

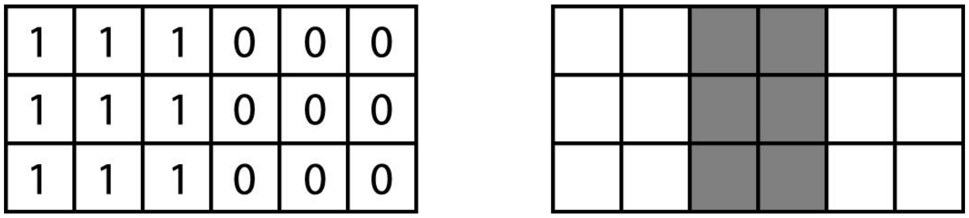


Fig. 16. An example pixel map and its corresponding edge.

For example, let's look at a small area of an image (Figure 16). Each box represents a pixel. The numeral 1 corresponds to maximum pixel intensity and 0

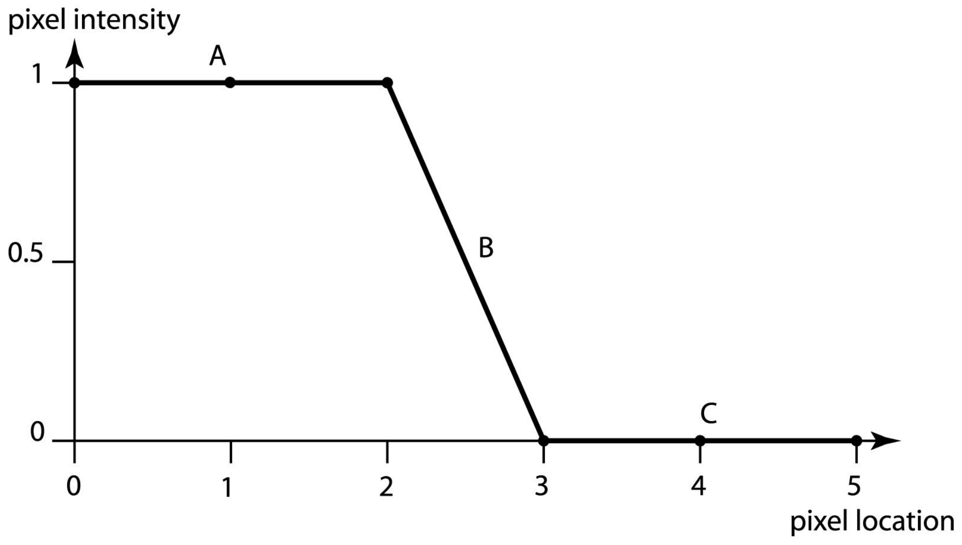


Fig. 17. Pixel intensities across the middle scanline.

corresponds to minimum pixel intensity. Intuitively, if we perform edge detection on just this area an edge would be found at the third and fourth pixels of each scanline. If we graph pixel intensities across the middle scanline, one possible representation is shown in Figure 17.

The gradient of the graph at areas A and C are both equal to 0. This gradient is analogous to the gradient we would calculate using convolution. Applying the Sobel kernels, the gradient at point B, which is halfway between the third and fourth pixels, is -0.5. Since the gradient is negative, the direction of the gradient with respect to the scanline is shown in Figure 18.

If we rotate the gradient vectors by 90 degrees and chart the gradient vectors on individual pixels, we have the result shown in Figure 19. All gradient vectors in the diagram have length 0.5. As desired, the gradient vectors are aligned with the edge. Any particle moving through this area will have a force applied in this direction along the edge. $forcefield_1$ is generated by calculating these gradient vectors across

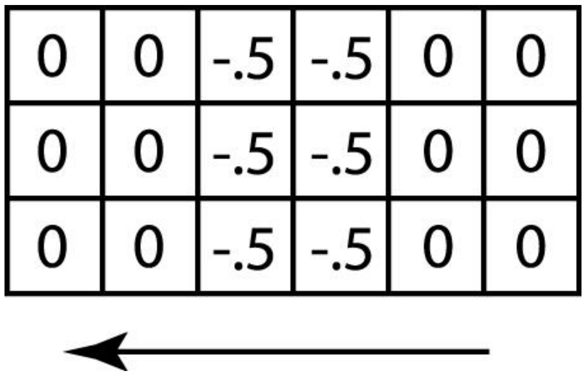


Fig. 18. Direction of gradient with respect to pixel map.

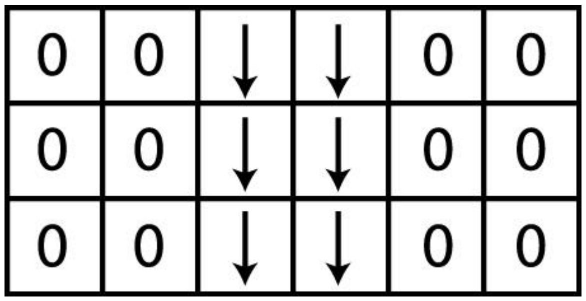
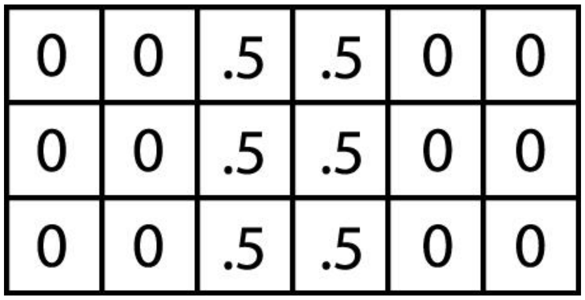


Fig. 19. Gradient vectors rotated by 90.

the entire image and rotating all of them by 90 degrees. To calculate $forcefield_2$, we first find the magnitude of all the gradient vectors in $forcefield_1$. As mentioned above, the length of these vectors is 0.5 (Figure 20).



0	0	.5	.5	0	0
0	0	.5	.5	0	0
0	0	.5	.5	0	0

Fig. 20. The magnitude of gradient vectors in $forcefield_1$.

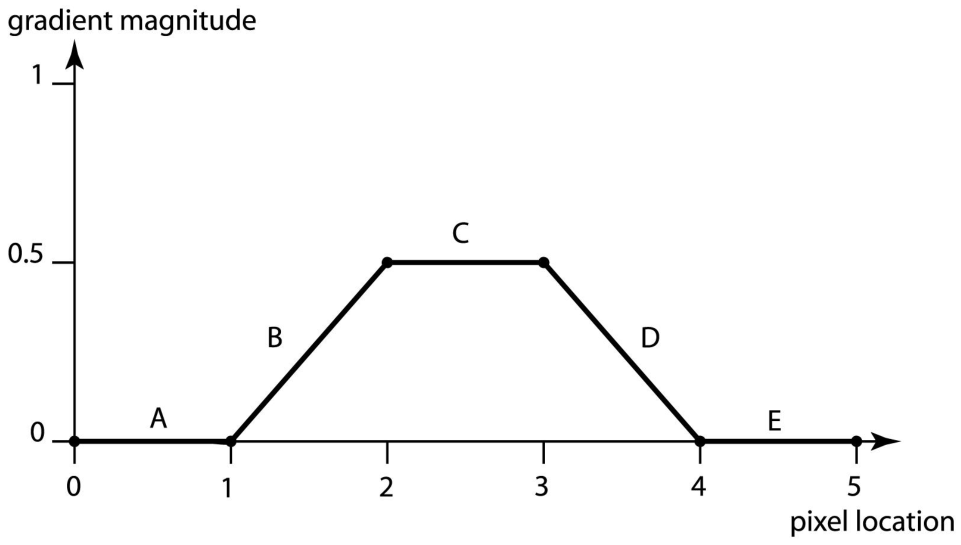


Fig. 21. Gradient magnitude vs. pixel location.

Figure 21 depicts a graph of the middle scanline with gradient magnitude on the y-axis and pixel positions on the x-axis. Applying the Sobel kernels, the gradient of the graph at areas A, C, and E is equal to 0. The gradient at B is 0.25 and D

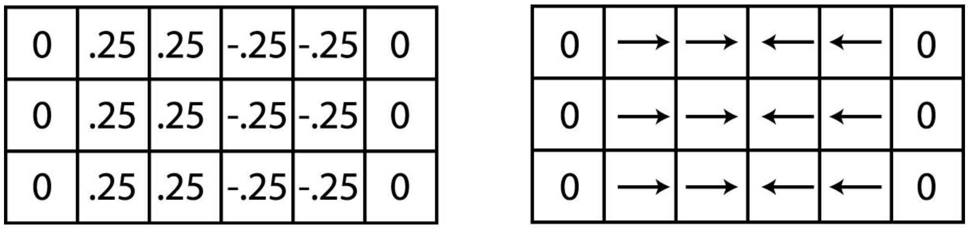


Fig. 22. Pixel gradients and their directions.

is -0.25. Figure 22 illustrates the gradient of the graph using our scanline diagram. This operation applied across $forcefield_1$ yields $forcefield_2$.

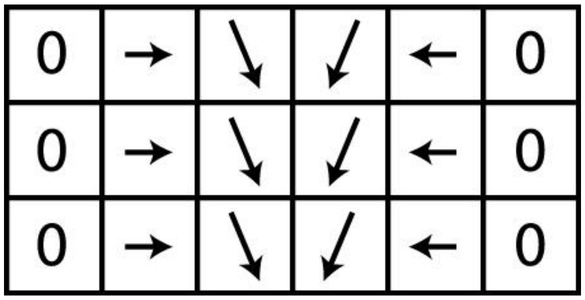


Fig. 23. Result of summing $forcefield_1$ with $forcefield_2$.

The two vector fields can be combined to create a single vector field that guides a particle along an edge: $forcefield_1$ provides the primary direction for particles to travel while $forcefield_2$ helps keep the particle from wandering off the edge. Simply summing $forcefield_1$ with $forcefield_2$ yields the desired vector field (Figure 23). This is the vector field input into the physically-based particle system.

VII.4. Particle system and stroke generation

Using the edge map and vector field as input, a physically-based particle system emulates the look of hand-drawn crayon strokes by tracing the edges in the edge map. The particle system operates by randomly generating particles within the edge image. If a particle is generated in an area that has an edge, it is swept up by the vector field and begins to move along that edge. Otherwise, the particle immediately dies, and a new one is generated.

As the particle moves along the edge it creates control points for an outline stroke. To avoid tracing the same edge several times, the particle also erases the edge image as it moves so that no more particles may create strokes there. When the particle wanders off into an area that does not have an edge, it dies. Only one particle is alive at a time.

The particle system is physically-based in that each particle has mass, and the system calculates the velocity and acceleration for the particle according to the vector field. By introducing a drag force to the system, the path of the particle can be controlled so that strokes may conform tightly or loosely to the edge depending on the drag coefficient. A random force vector may also be applied to introduce wiggle to the strokes. The simulation continues until all edges are erased or a maximum number of particles are generated.

CHAPTER VIII

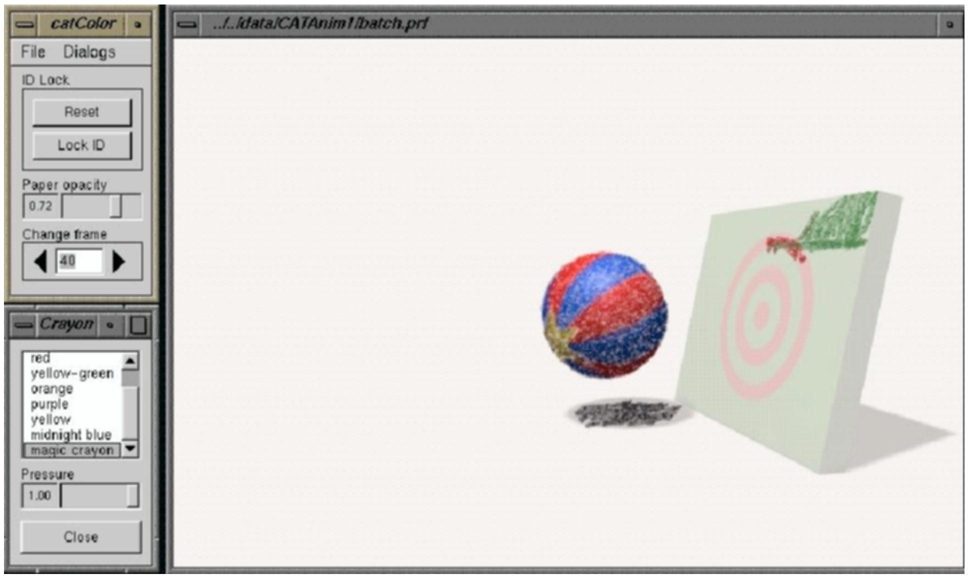
RESULTS

Three software components were developed to realize a prototype implementation of the Crayon Animation Tool. The interactive painter provides facilities for an artist to draw on top of reference images and define strokes for the final rendered images. The outline generator provides an automatic method for defining silhouette outlines. And the batch renderer provides a simple command-line facility for rendering the strokes defined in the interactive painter and combining them with those created by the outline generator.

Each tool requires a profile data file. The profile specifies where the reference images, paper data, crayon strokes, and crayon boxes can be found. Each crayon box is implemented as a simple data file that stores all of the crayons that can be used to render the animation. Each crayon in the data file is given a name, a set of points that describe the tip shape, a multiplier for the tip shape that specifies the diameter, and a color.

Once a profile is loaded in the interactive painter, the color buffer for the first frame of the animation is displayed to the user. A slider is provided to adjust the opacity of the paper displayed above the color buffer, allowing the artist to partially or fully obscure the underlying color buffer while drawing. The interactive painter also provides the artist with controls to change the current frame of the animation. This allows the artist to draw strokes in any frame of the animation, so that new strokes may be added as objects move and new surfaces are uncovered. The user interface for the interactive painter is shown in Figure 24.

To begin drawing crayon strokes, the artist chooses a crayon from the crayon box



Paper opacity 0.72

Change frame 40

Pressure 1.00

Fig. 24. The interactive painter.

window. A variety of crayon colors and sizes may be selected here. A slider at the bottom of the window provides a control for adjusting the pressure of the stroke.

The crayon box also supports the use of a magic crayon. Instead of a single color, a magic crayon determines its color from the color buffer of the current reference image. For each stroke drawn with this crayon, the color of the stroke is determined by looking up the pixel in the color buffer at the starting position of the stroke. This setup is useful for capturing any lighting and shading depicted in the color buffer.

To draw the stroke, the artist places the mouse or stylus cursor over the desired region of the paper, clicks and drags. As the artist drags, a crayon stroke is formed along the path of the mouse cursor and is displayed to the screen. For each point along the stroke, the tool locates the corresponding u, v, and Id coordinates in the geometry buffer. This data is assigned to each point of the stroke so that the tool can track the stroke in adjacent frames. When a new frame is rendered, the point is transformed to the new position defined in the geometry buffer and rendered.

The interactive painter also provides a facility to ensure that new strokes are only applied to a specific surface. When the user selects the “Lock Id” button, all new strokes are applied to a specific surface. This prevents the user from accidentally drawing across surface boundaries. Drawing across surface boundaries can be problematic when two adjacent surfaces are moving at different velocities. The resulting stroke may experience unwanted stretching or shrinking as the frame changes. Once the user is finished drawing crayon strokes, they are saved to a file where they can be retrieved by the batch renderer when the final images are rendered.

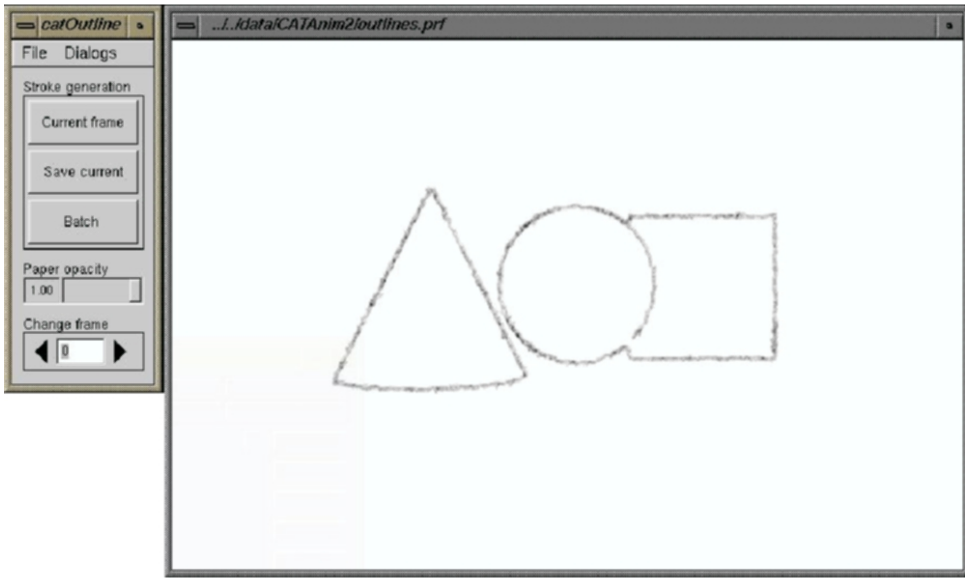


Fig. 25. The outline generator.

The outline generator shown in Figure 25 also uses a profile to locate the necessary data files for generating outline strokes. Facilities are provided to tweak simulation parameters such as the number of particles used, drag forces, and edge detection thresholds. The artist may try different settings and crayons until the desired look is achieved. Once the parameters have been defined, the artist may then render outline strokes for the entire animation and store them in a stroke file for the batch renderer.

The batch renderer is the simplest of the three tools. From the command line the user specifies a profile, the range of frames to render, and a header string for the output filenames. The renderer reads all of the stroke files specified in the profile, and renders the final images of the animation.

Using the implemented Crayon Animation Tool, two animations were produced. The first animation depicts a physically-based ball that collides with a wall and bounces to a rest. The scene was first modeled and animated in Maya where the majority of production time was spent. Some simple shading and lighting was applied to the scene. Each reference image output consisted of a color buffer and a geometry buffer.

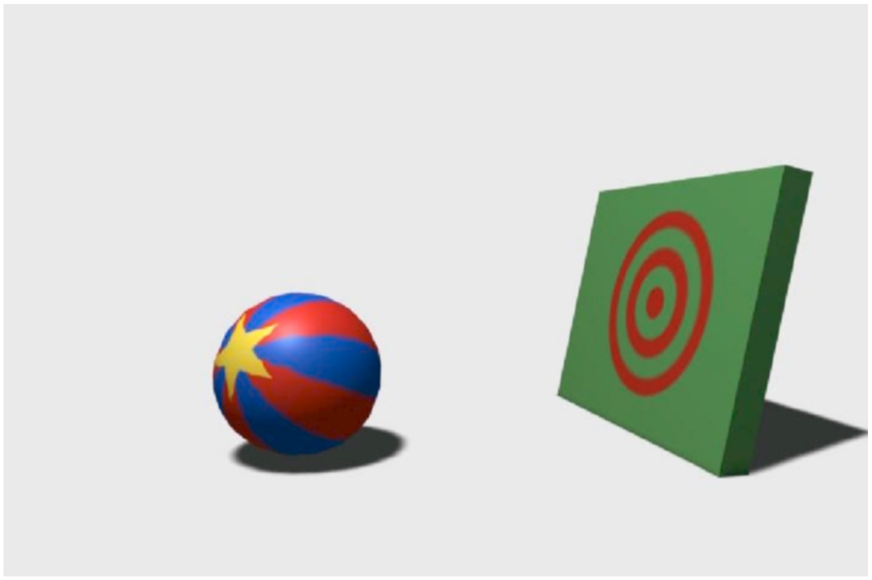


Fig. 26. A color buffer from the first animation.

The reference images were imported into the interactive painter where the crayon

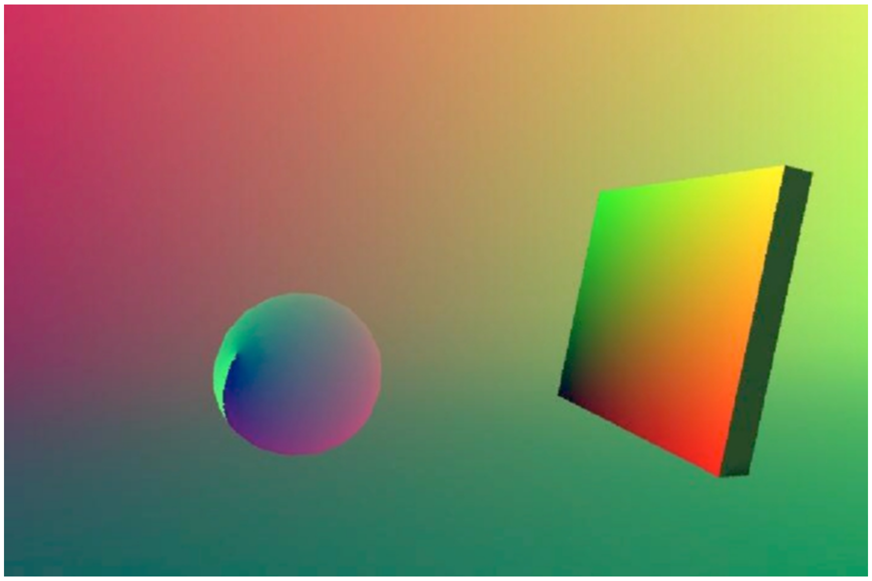


Fig. 27. A UVId buffer from the first animation.

strokes for the final output images were defined. For color, a magic crayon was used to look up the color of the pixel in the color buffer at the starting location of each stroke. This technique proved to be quite useful in quickly defining strokes that preserved the lighting and shading elements of the color buffer. Figure 26 and Figure 27 show the data files used to define one of the final images of the animation. Figure 28 shows several final frames from the animation.

The second animation produced depicts a still life of three simple, geometric primitives: a cone, a cube, and a sphere. The camera is animated in this example, as it rotates around the primitives. Again, the scene was first modeled and animated in Maya with some simple shading and lighting applied. Each reference image output consisted of a color buffer, a geometry buffer, and a depth buffer to support outlines.

The color and geometry buffers were imported into the interactive painter, where the magic crayon was used to define the crayon strokes. The cube is actually a

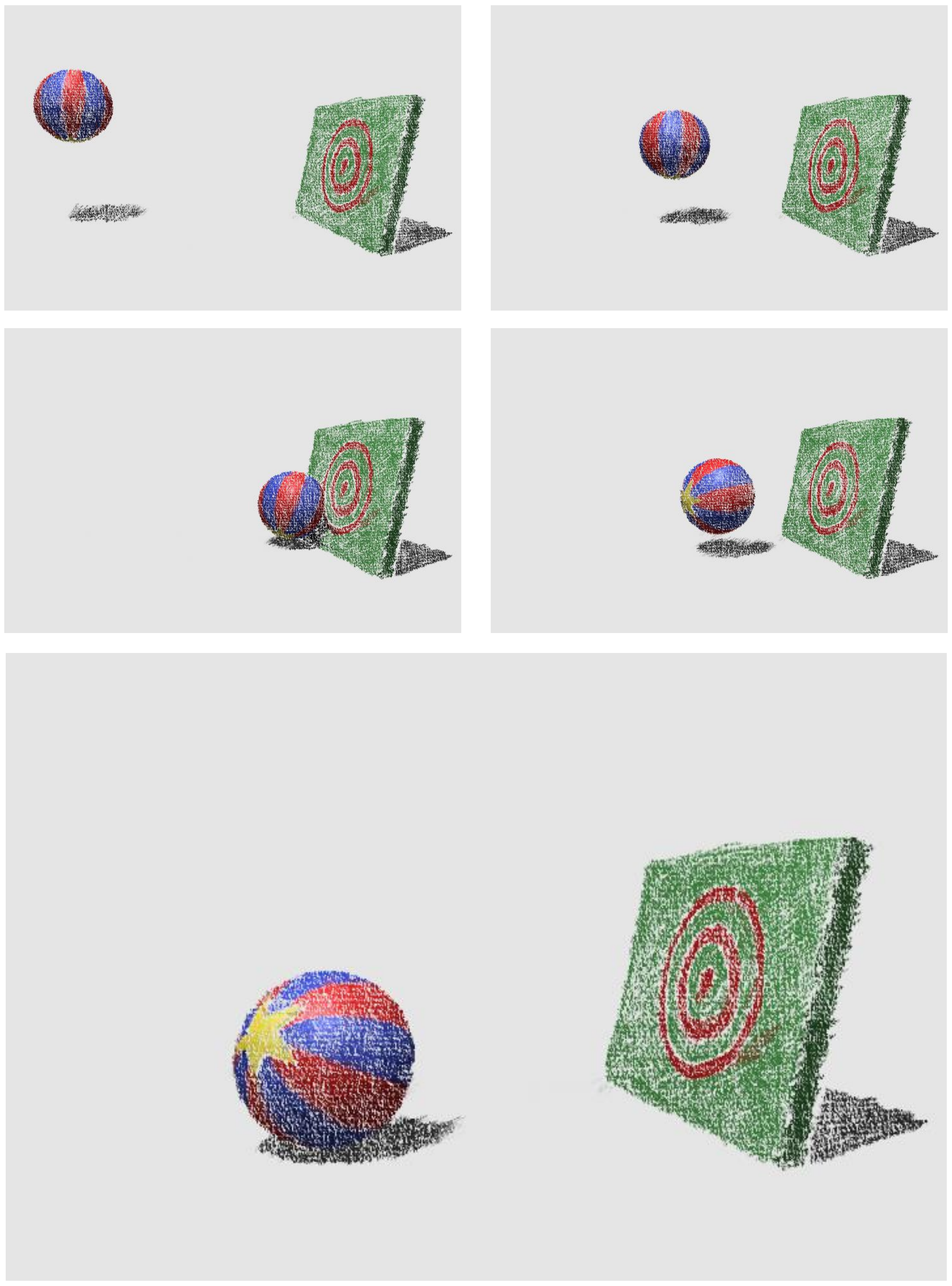


Fig. 28. Several frames from the first animation. The shadows were achieved using a magic crayon on the background flat.

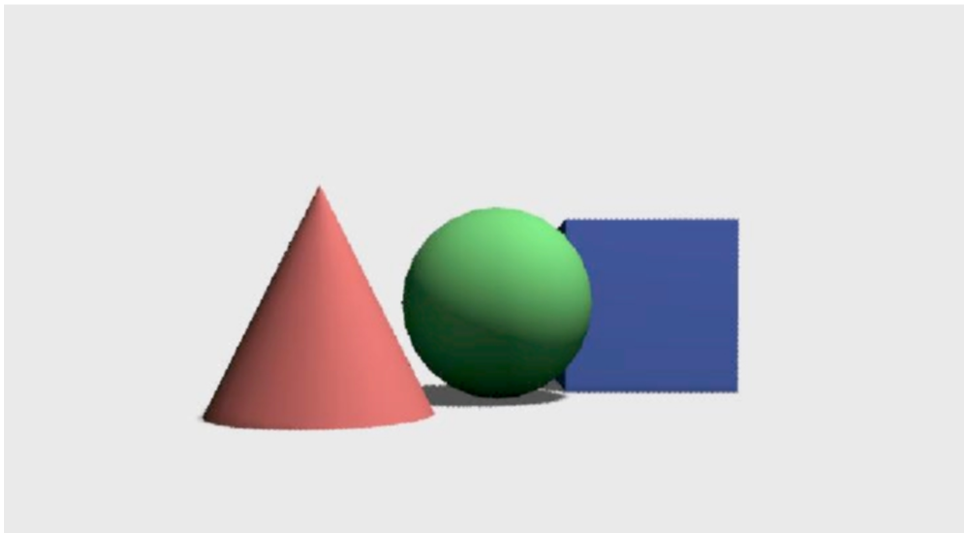


Fig. 29. A color buffer from the second animation.

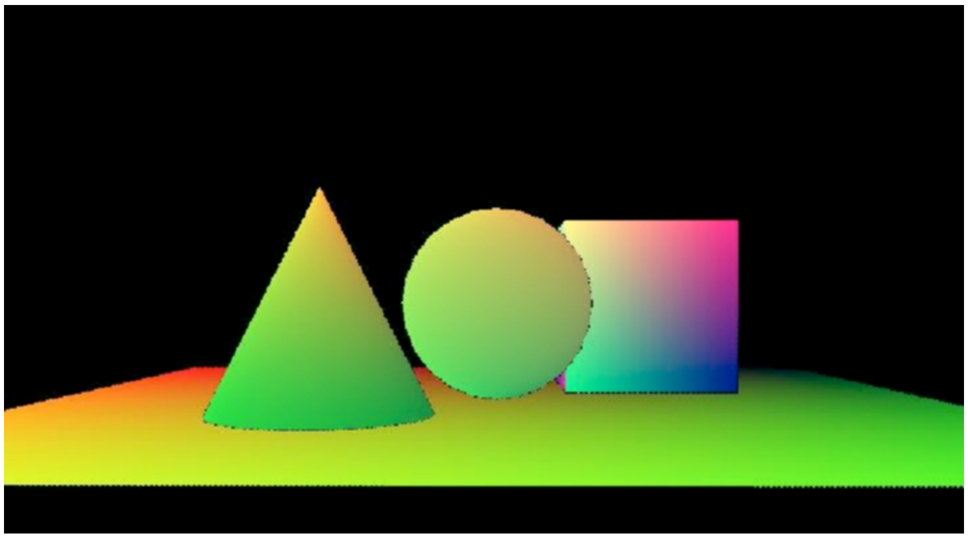


Fig. 30. A UVId buffer from the second animation.

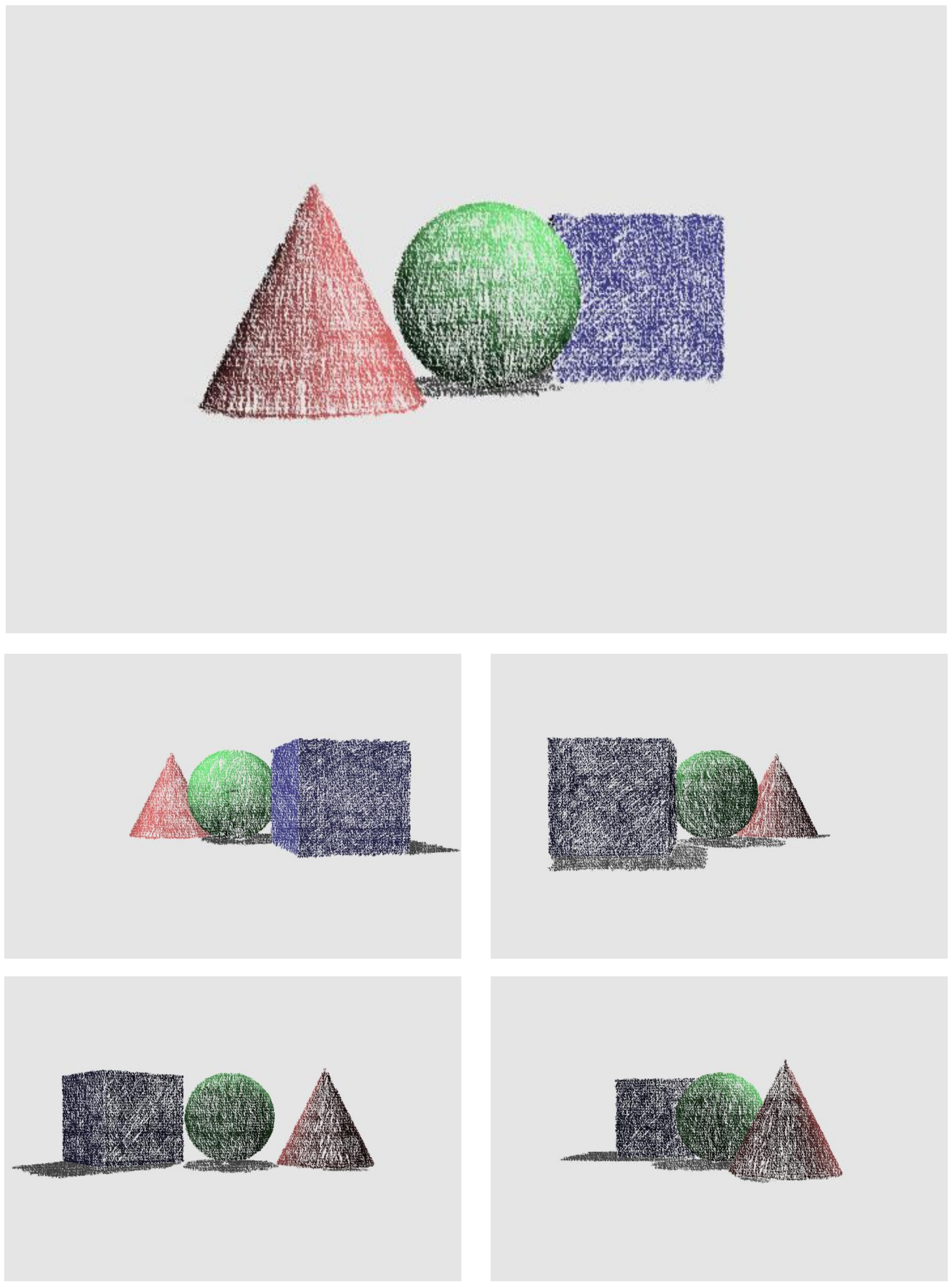


Fig. 31. Several frames from the second animation.

combination of six separate surfaces, and the strokes applied to it had no difficulty in spanning across surface boundaries (each point of a stroke has a surface Id coordinate, so the stroke may have many points with different Ids). Figure 29 and Figure 30 show the color buffer and geometry buffer. Figure 31 shows several final frames from the animation.

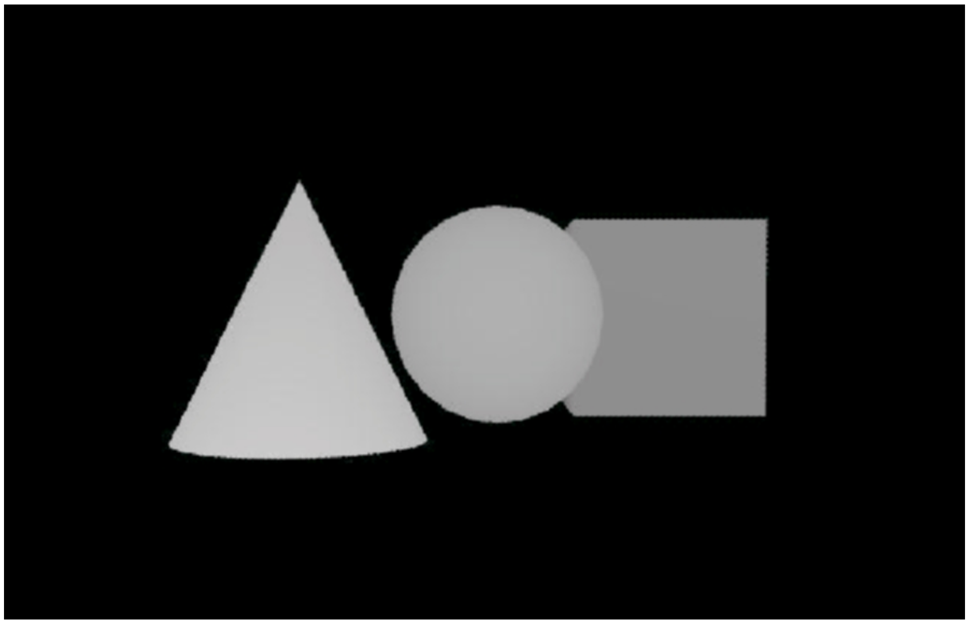


Fig. 32. A depth buffer from the second animation.

The depth buffers for the animation were imported into the outline generator. Here, which crayon to use was specified as were the simulation parameters. For each frame of animation, the program generated a set of outline strokes. The batch renderer then combined these strokes with those defined in the interactive painter and rendered the final images. Figure 32, Figure 33, and Figure 34 depict a depth buffer, a corresponding rendered outline, and the final image rendered with outlines.

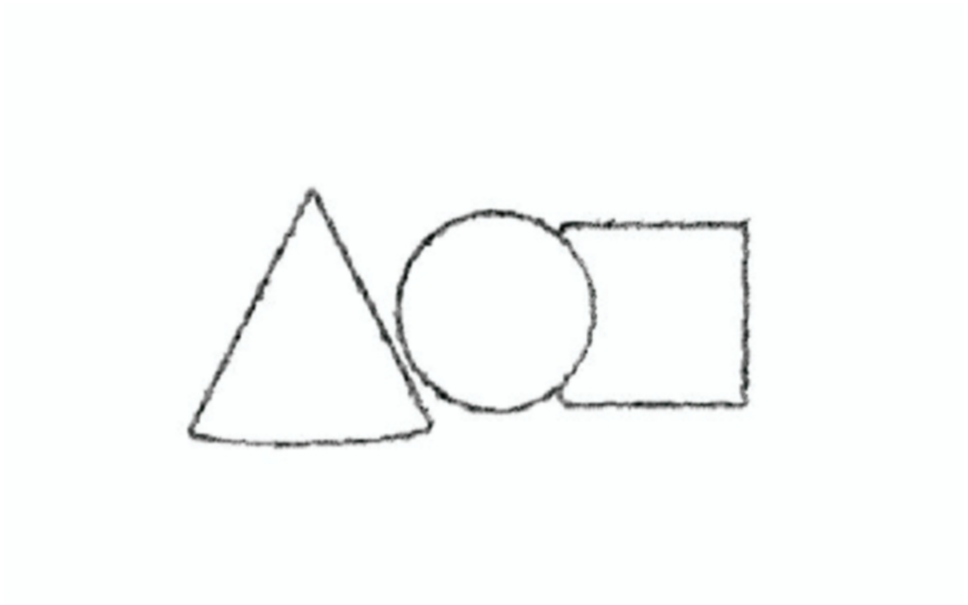


Fig. 33. An outline render from the second animation.

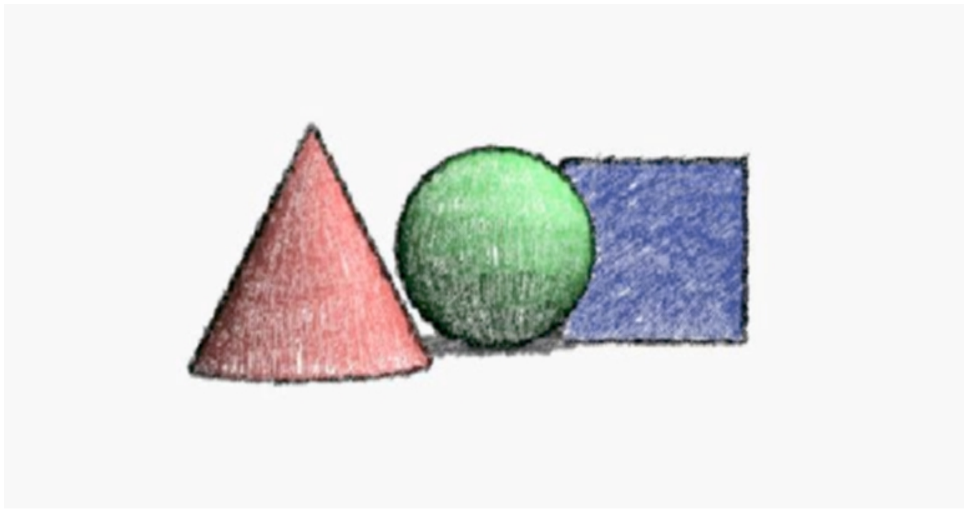


Fig. 34. A final crayon render with outlines from the second animation.

CHAPTER IX

DISCUSSION

The production of two short animations provided an opportunity to evaluate the results of this thesis. Specifically, how well does the stroke tracker reduce the amount of time and effort to produce an animation? Is the interface to the system simple and intuitive? Does the system provide direct control over the final output image? And does the renderer provide a convincing emulation of actual crayon drawing?

In the author's estimation, stroke tracking proved to be an excellent leverage of the computer's processing power. The method provides excellent stroke coherency, and significantly reduced the amount of time for production versus a hand-drawn approach. The accuracy of stroke coherency was high enough to warrant adding a slight jitter to the strokes to produce a more convincing hand-drawn appearance.

Simplicity and control are often conflicting problems that require divergent solutions. In the present system, the success of these two objectives is mixed. Painting three-dimensional animation in two-dimensions was quite intuitive and simple to understand for the most part. Additionally, coloring on top of a color buffer was instrumental in the placement of crayon strokes. However, user confusion arises when strokes are drawn across surface boundaries that are not animating at the same velocity. For example, if the stroke is drawn across two planes moving in opposite directions, the stroke will stretch between them as they move farther apart.

The present system provides a facility for avoiding cases like these, but a better one could be devised. One method would be to only allow strokes to be drawn on a single, conceptual model rather than only on a single surface. For example, the ball in the first animation was colored separately from the target. The system could prevent the user from trying to draw a stroke crossing both models. Note that a single, conceptual model is different from a single surface. One side of a cube can be a single surface, but the entire cube is a single, conceptual model. Such a system could be easily implemented by including a new buffer in the reference image that identifies conceptual models in the scene.

The interface of the present system could definitely be further simplified. For example, the management of data files and profiles could be handled by the system entirely, freeing the artist to focus on more creative problems. The controls could be presented more effectively, perhaps organized in a manner similar to an artist's drafting table. For example, the crayon box could be implemented quite nicely using graphic representations of different color crayons and crayon tips.

The present system lacks some basic editing controls. There is no undo function, which would be required in a real production system. An eraser function would also be a nice addition. There is no way to edit the shape of a stroke that has already been drawn or keyframe values for its attributes. However, the challenge is to add these facilities without sacrificing the simplicity of the interface. One nice characteristic of the present system is that the interface does not overwhelm the user with controls.

Because it is driven by a simulator and not by hand, the outline generator is unintuitive to use and lacks direct control. An alternative to this would be to trade the benefits of an automatic process in return for a more simple and direct method. For instance, the interactive painter could support the specification of space strokes that are not correlated with any objects in the 3D scene. This method could lead to a new workflow for the artist, where the geometry strokes are defined during the first color passes for the animation, and the outlines or other space strokes are defined during subsequent passes.

The crayon renderer provides a reasonable approximation of actual crayon draw-

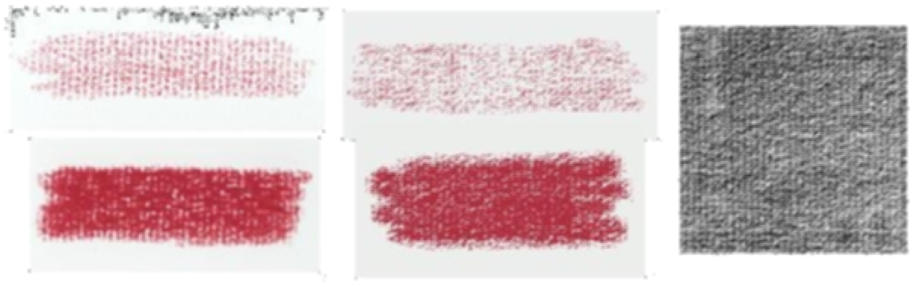


Fig. 35. Rendering comparisons on corrugated paper. From left: actual, rendered, paper texture.
Settings used: $F_{lo} = 2, F_{hi} = 10F_t = 5, D_{lo} = 0.4D_{hi} = 0.9, M_s = 3, \phi = 1.5.$

ing. The paper textures are the most important element required for re- creating the look of natural media. Figure 35 shows side- by-side comparisons of actual crayon samples and those rendered by CAT. The first column shows crayon samples drawn at high and low pressures on a corrugated paper texture. A sample of the paper texture in column 3 was acquired using a digital camera and input into CAT. The second column shows the result in CAT.

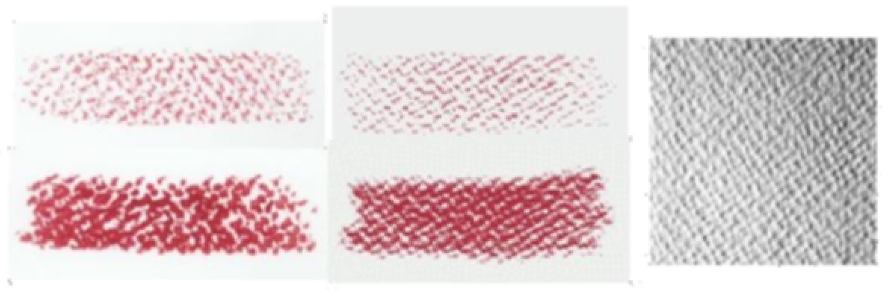


Fig. 36. Rendering comparisons on watercolor paper. From left: actual, rendered, paper texture.
Settings used: $F_{lo} = 1.1, F_{hi} = 5, F_t = 5, D_{lo} = 0.11, D_{hi} = 0.4, M_s = 3, \phi = 1.5.$

Figure 36 shows another set of comparisons on watercolor paper. The paper sample used was also acquired using a digital camera, and this test exhibits a limitation of cameras. When using a camera, the resulting image is capturing the paper's response to light rather than an actual heightfield. In this case, the resulting texture is quite different in comparison to the real paper, and this results in a different character to the crayon drawing in CAT.

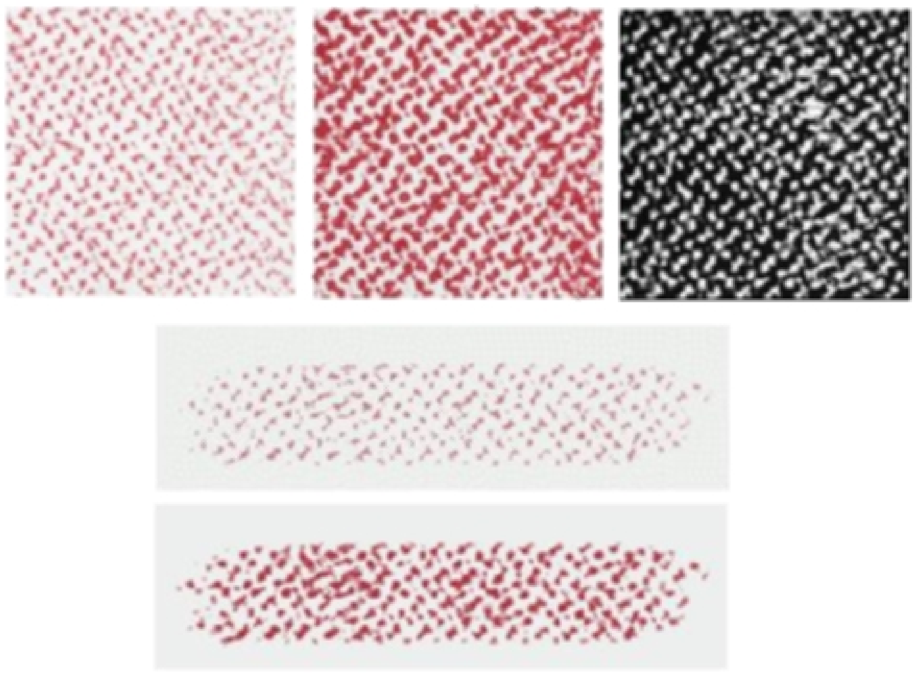


Fig. 37. Building a more accurate watercolor texture. Top images, from left: paper sample shaded at low intensity, paper sample shaded at high intensity, resulting heightfield. Bottom images: rendering at low pressure, rendering at high pressure.
Settings used: $F_{lo} = 1, F_{hi} = 2.5, F_t = 10, D_{lo} = 0.25, D_{hi} = 0.8, M_s = 3, \phi = 1.5$.

Another method for acquiring paper textures can be developed by hand-shading a paper sample at varying pressures. The first two images of Figure 37 shows the same watercolor paper sample shaded with a crayon at low and high pressure. Using a scanner and an image manipulation program, one can composite a rudimentary heightfield (third image), which can then be input into CAT. The resulting crayon renderings (bottom images) more closely match the real-life samples.

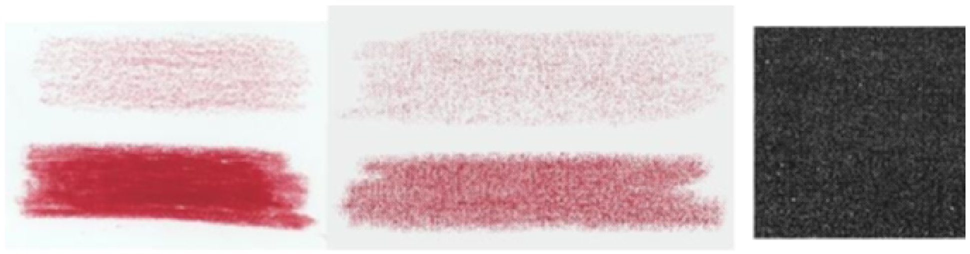


Fig. 38. Rendering comparisons on smooth paper. From left: actual, rendered, paper texture.
Settings used: $F_{lo} = 1, F_{hi} = 5, F_t = 7.5, D_{lo} = 0.8, D_{hi} = 0.99, M_s = 3, \phi = 1.5$.

Shading an actual paper sample also proves to be a useful way for acquiring paper textures that are too smooth for shooting with a camera. Figure 38 shows crayon comparisons for a smooth sheet of notepad paper. In this example, the emulator does a reasonable job at lower crayon pressures. However, it isn't able to capture the quality of the actual crayon drawing at high pressure. On smooth papers, the crayon material begins to build up into layers at high crayon pressure. The resulting look is less dependent on the paper texture and more dependent on how crayon material smears and sticks together.

IX.1. Future work

There are many areas where this research could be extended. One would be to concentrate on building a more artist-friendly interface that would hide much of the file management as well as provide more intuitive controls for the user. The system could also be adapted to support other geometry types such as polygons or subdivision surfaces. For example, the geometry buffer could encode 4-channels for these, an x_{ref}, y_{ref}, z_{ref}, and ID where x_{ref}, y_{ref}, z_{ref} are the x,y,z coordinates of each point on the model in some default reference pose. Three-dimensional bucket arrays could be developed for handling stroke tracking.

The emulator could be updated to handle other effects of crayon drawing. Paying attention to the gradient of the paper grains would help model highlighting of paper texture according to stroke direction. Also, crayons are very soft and modeling tip shape changes according to pressure may augment the illusion of working with real media. As mentioned earlier, a more robust emulator could handle effects of smearing on smooth papers at high pressure. Also, modeling paper grain damage due to high pressure would be worth investigating. Or the emulator could be entirely replaced with another rendering method that might emulate watercolor, pen-and-ink, etc.

The system could be coupled with a system like the one designed by Litwinowicz [6] to process video segments. The user could define the initial strokes for the animation, which would then be transformed by the optical flow of pixels from one frame to the next. As areas became too dense, strokes could be killed if necessary. And in areas that require more strokes, the user could define new strokes. The resulting system would allow the user to have more control over the final look of the animation while still leveraging the computer in the automatic tracking of strokes.

CHAPTER X

CONCLUSION

We successfully implemented a non-photorealistic rendering system for generating animation that emulates the appearance of hand-drawn wax crayon on paper. This project allowed us to experiment with several techniques for simplifying and improving the usability of non-photorealistic rendering for animation. We implemented a method for automatically correlating two-dimensional crayon strokes with three-dimensional objects and developed an interface for the painting of three-dimensional scenes in two dimensions. We also investigated methods for ensuring simplicity and control.

Three software tools were developed during the implementation of the present system. An interactive painter provides an intuitive interface for the specification of crayon strokes. An outline generator employs a method for the automatic generation of silhouette outlines. A batch renderer combines the results of the interactive painter and the outline generator to produce the final rendered images of the system.

Two animations were produced to evaluate the system. The first provided a case study of a simple scene with a stationary camera. A second animation explored the use of an animated camera.

Automatic correlation of crayon strokes significantly decreased the amount of time required to render the animation by hand while preserving excellent stroke coherency across frames. A simple interface for painting three-dimensional scenes in two-dimensions provided an intuitive method for specification of crayon strokes. Possibilities for enhancing the simplicity and control of the system were identified and discussed.

REFERENCES

[1] C. J. Curtis, S. E. Anderson, K. W. Fleischer, and D. H. Salesin, "Computer-Generated Watercolor." *SIGGRAPH 97 Conference Proceedings*, pp. 421-430, August 1997.

[2] C. Curtis, "Loose and Sketchy Animation." *SIGGRAPH 98 Conference Abstracts and Applications*, p. 317, July 1998.

[3] E. Daniels, "Deep Canvas in Disney's Tarzan." *SIGGRAPH 99 Conference Abstracts and Applications*, p. 200, August 1999.

[4] M. A. Kowalski, L. Markosian, J. D. Northrup, L. Bourdev, R. Barzel, L. S. Holden, and J. Hughes, "Art-Based Rendering of Fur, Grass, and Trees." *SIGGRAPH 99 Conference Proceedings*, pp. 433-438, August 1999.

[5] J. Lansdown and S. Schofield, "Expressive Rendering: A Review of Nonphotorealistic Techniques." *IEEE Computer Graphics and Applications*, vol. 15, no. 3, pp.29-37, May 1995.

[6] P. Litwinowicz, "Processing Images and Video for an Impressionistic Effect." *SIGGRAPH 97 Conference Proceedings*, pp. 407-414, August 1997.

[7] B. J. Meier, "Painterly Rendering for Animation." *SIGGRAPH 96 Conference Proceedings*, pp.477-484, August 1996.

[8] B. Meier, "Computers for Artists Who Work Alone." *ACM SIGGRAPH Computer Graphics*, vol. 33, no. 1, pp. 50-51, February 1999.

[9] G. Miller, "Efficient Algorithms for Local and Global Accessibility Shading." *SIGGRAPH 94 Conference Proceedings*, pp. 319-326, July 1994.

[10] K. Perlin, "An Image Synthesizer." *SIGGRAPH 85 Conference Proceedings*, pp. 287-296, July 1985.

[11] T. Saito and T. Takahashi, "Comprehensible Rendering of 3D Shapes." *SIGGRAPH 90 Conference Proceedings*, pp. 197-206, August 1990.

[12] M. P. Salisbury, S. E. Anderson, R. Barzel, and D. H. Salesin, "Interactive Pen-and-Ink Illustration." *SIGGRAPH 94 Conference Proceedings*, pp. 101-108, July 1994.

[13] S. Schofield, "NPR - The Artist's Perspective." *SIGGRAPH 99 Course Notes: "Non-Photorealistic Rendering"*, pp. 4(1)-4(13), August 1999.

[14] J. Seims, "Putting the Artist in the Loop." *ACM SIGGRAPH Computer Graphics*, vol. 33, no. 1, pp. 52-53, February 1999.

[15] M. C. Sousa and J. W. Buchanan, "Observational Models of Graphite Pencil Materials." *Computer Graphics Forum*, vol. 19, no. 1, pp. 27-49, March 2000.

[16] S. Strassman, "Hairy Brushes." *SIGGRAPH 86 Conference Proceedings*, pp. 225-232, August 1986.

[17] S. Takagi, M. Nakajima, I. Fujishiro, "Volumetric Modeling of Artistic Techniques in Colored Pencil Drawing." *SIGGRAPH 99 Conference Abstracts and Applications*, pp. 238, August 1999.

[18] D. Teece, "3D Painting for Non-Photorealistic Rendering." *SIGGRAPH 98 Conference Abstracts and Applications*, p. 248, July 1998.

[19] S. P. Worley, "A Cellular Texturing Basis Function." *SIGGRAPH 96 Conference Proceedings*, pp. 469-476, August 1996.

VITA

Howard John Halstead IV
19532 Barclay Rd.
Castro Valley, CA 94546
jhalstead@pixar.com

Education

M.S. in Visualization Sciences	Texas A&M University, December 2004
B.S. in Computer Engineering	Texas A&M University, May 1997

Employment

Technical Director	Pixar Animation Studios, June 2000 - Present Film credits: Finding Nemo, Cars (2005)