A NON - PHOTOREALISTIC MODEL FOR PROCEDURAL PAINTERLY RENDERED TREES IN THE STYLE OF COROT

A Thesis

by

MICHAEL LOSURE

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

August 2008

Major Subject: Visualization Sciences

A NON - PHOTOREALISTIC MODEL FOR PROCEDURAL PAINTERLY RENDERED TREES IN THE STYLE OF COROT

A Thesis

by

MICHAEL LOSURE

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

Approved by:

Chair of Committee,	Donald H. House
Committee Members,	Carol LaFayette
	John Keyser
Head of Department,	Tim McLaughlin

August 2008

Major Subject: Visualization Sciences

ABSTRACT

A Non-Photorealistic Model for Procedural

Painterly Rendered Trees in the Style of Corot. (August 2008)

Michael Losure, B.A., Union College

Chair of Advisory Committee: Dr. Donald H. House

This thesis describes the development of a system for the procedural generation and painterly rendering of trees. Specifically, the rendered trees are modeled after those found in the oil landscape paintings of 19th century French painter Camille Corot. The rendering system, which is a combination of MEL-scripted Maya tools and Renderman shaders, facilitates the creation of still images that look convincingly painterly, as well as 3D animations with temporal coherence. Brush stroke properties are animated based on distance from the camera, so that traditional painting techniques for representing depth are incorporated into the computer-generated animations.

During the development process, the system was generalized to apply to other structures, such as grass and rocks, and allows for the creation and rendering of entire landscapes. Several example animations were created with the system to demonstrate the ideas developed during the process and the quality of the results.

Dedicated to mom and dad.

ACKNOWLEDGEMENTS

I would like to say thank you to my committee chair Dr. Donald House for being such a fantastic teacher and a great guy to be around, and also for indirectly introducing me to the viz lab. Also thanks to my committee members Carol LaFayette and John Keyser for their helpful questions and kindness. Thank you to Professor Walter Hatke, for helping me develop as a painter, and for teaching me ways of applying that other side of my brain to my creative endeavors. And thank you to Phillip Rollfing, for understanding when I'd disappear from work for days at a time to go write this thesis.

Thanks also to all the students, faculty, and staff of the viz lab for making it such a great place to have spent the last few years, especially to Patrick O'Brien, Tony Piedra, and Seth Freeman, for all the fun, and for one particularly large and valuable distraction from this thesis.

Thank you to my delightful girlfriend Lauren Simpson, for giving me inspiration and love, and for making it so fun to avoid my thesis, but being understanding when I didn't. And thanks to Kirby, for reminding me that proper ergonomics demand frequent breaks for fake mice, treats and general mischief.

Thanks to all my family, and lots of thanks to my parents, particularly for not giving in to my video game demands when I was younger, and for teaching me the joys of

imagination, reading and forced walks in the woods – and of course for all of their love and support, both of which were always given freely and in abundance.

TABLE OF CONTENTS

CHAPTER Page

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

Historically, most computer graphics research has focused on achieving photorealistic imagery through physical simulation, while other artistic media often purposely depart from realism to pursue more abstracted visuals. Non-photorealistic rendering (NPR) is the subgenre of computer graphics that focuses on creating expressive, artistically motivated images, often driven by concepts found in traditional art and 2D animation [9; 27]. Though NPR research is defined as any computer graphics research not devoted to photorealistic techniques, the rest of this paper will refer to the subset of NPR research devoted to replicating the look of traditional artwork.

Some NPR researchers have focused on directly simulating artistic media and processes, creating systems that let users manipulate simulated brushes and paints on digital canvases. Other researchers achieve artistic results using approaches born more from signal processing than art, such as using 2D image processing algorithms to create painterly images from photographs.

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

NPR techniques for animation attempt to replicate the visual complexity and beauty of traditional art, while simplifying the image-making process as much as possible. Creating an animation requires many, many images – films usually have 24 images per second of screen time. For this reason alone, one of the main considerations in any NPR technique is how to push as much work as possible onto the computer while leaving the user with as much artistic control as possible.

Imperfections and non-uniformity are two very important properties of human-created artwork that are often difficult to emulate with procedurally generated computer artwork. Many of the computer algorithms used to create this style of NPR imagery employ stochastic methods that work well for single frames, but introduce displeasing temporal noise in animated sequences. Thus systems for creating NPR animation must include a way of achieving temporal coherence.

Trees and natural landscapes are a serious challenge to replicate with computer graphics because of their visual complexity and high viewer familiarity. Traditional artists face the same problem and have developed a wide variety of solutions for simplifying and artistically representing nature.

The primary goal of this thesis has been the development of a system for rendering trees in a painterly style. Since oil paint as a medium allows for an immense variety of visual styles, techniques and effects, this project makes no attempt to emulate the medium in its

entirety, and instead tries to raise the overall quality of the results by narrowing the focus to the oil landscape paintings of 19th century French painter Jean-Baptiste-Camille Corot. The resulting system is a temporally coherent procedural NPR technique for creating images and animations of landscapes in the style of his paintings.

In addition to recreating Corot's style, the project focuses on the implications of adding time and motion to oil painting. Within a painting, distant objects are often depicted in a very different manner from those in the foreground, and the goal has been to allow for the creation of animations where objects undergo fluid changes in scale and distance. For example, consider a camera moving forward through a scene; as trees get closer and change from their background representation to their foreground representation, they should do so in a way that feels right for the oil medium. Brush strokes should not scale in perfect linear perspective, because the size of brush strokes an artist chooses to use do not follow the same strict mathematical rules of perspective as 3D geometry; previous 3D systems that have scaled brush strokes precisely with geometry immediately belay their digital origins, and this researcher has sought wherever possible to maintain the illusion that the animations are painted and not computer generated. Additionally, since close-up representations of trees are painted with many more brush strokes than those in the background, a system has been developed for adding and subtracting brush strokes with distance that preserves the overall volume and silhouette of the trees.

The goal has been to produce output imagery that is highly controllable by the artist in terms of brushwork, lighting effects, and other stylistic choices, not only in still images, but also over the course of an animation. In the same way that a painter directs a viewer's eye by varying brush stroke size and detail, a cinematographer uses the camera lens and focus to articulate an image. A rack focus is the film technique of dynamically changing the camera's focus from one object to another, and a similar effect is created by changing brush stroke sizes and styles over time to emphasize one section of the image over another. By finding correlations between film and painting technique, the project brings together the two separate but related visual languages.

CHAPTER II

CAMILLE COROT AND OIL PAINTING TECHNIQUE

> *...there is a great difference between an accomplished piece and a finished piece – that, in general, that which is accomplished is not finished, and something that is highly finished may not be accomplished at all* – Charles Baudelaire on Corot, 1845 [15, pg. 278]

The paintings of the French painter Jean-Baptiste-Camille Corot (1796-1875) are remarkable for their loose, painterly brushwork and distinct color palettes. His masterful use of light, color, and controversially loose brush strokes were unusual for his time and formed a bridge between the neoclassical tradition and the later impressionists [8][19].

As can be seen in Fig. 1, Corot's brushwork is at times tightly controlled and blended, and elsewhere highly visible and full of energy. By varying his brush stroke size, concentration, and style, he is able to direct the viewer's eye around his composition and lend emphasis to different areas. His trees in particular are often loosely painted, with a few distinct brush strokes giving the impression of detailed foliage. When his trees are in the extreme foreground, he often does not paint the branches, but implies their presence with a scattering of separated leaf brush strokes.

Fig. 1. "Souvenir de Mortefontaine," Corot, 1864. [30].

Painters often mix all of the colors within a given painting from a limited set of base colors. This harmonizes the image color scheme and gives the impression of light reflecting off of everything in the scene, the way it does in real life. Color hue can be used to indicate lighting; using warm colors in light areas and cool colors in darker areas can give a scene a lush, natural feel. Pure black is used very sparingly, if at all, since even the darkest colors are mixed from the base color palette. Corot's palettes are often full of pale yellows, de-saturated greens, and silvery grays, with a few sharp reds and bright colors reserved for special use.

Depth cueing is an important aspect of any believable natural image, and Corot employs several painting techniques to give his scenes depth. As can be seen in Fig. 1, Corot's paintings often make strong use of atmospheric perspective; distant foliage has cooler hues and much less defined borders, with colors blending between objects and more homogenized contrast, hue, and saturation levels.

CHAPTER III

PRIOR WORK

III.1. Painterly Rendering

There are many systems already developed for creating painterly images with a computer. Many of the techniques are based around methods of processing photographs or other 2D reference images.

One of the systems developed by Haeberli [12] tracks a user's mouse cursor moving over an image and generates brush strokes based on the mouse speed, direction and various user-specified parameters. The brush strokes are stored as data so their attributes can be manipulated later. Fig. 2 shows an example of Haeberli's system used to create several different painterly images from the same input photograph. While Haeberli's results achieve a variety of painterly looks, his systems are not particularly suited for animation due to the prohibitive amount of user interaction required and the stochastic algorithms used to generate brush strokes. One could add temporal coherence to Haeberli's system by maintaining the same brush strokes from frame to frame and changing their direction and color based on changes in the reference image sequence; however, this would make the animation seem as though it were filmed through a sheet of glass, because brush strokes would stick to the image plane instead of moving with the depicted forms. This

issue is known as the “shower door” problem and has been addressed in more recent research.

Fig. 2. Three paintings created from the same source image (using Haeberli's system) [12].

Litwinowicz [16] developed a system for painterly animation that avoids temporal aliasing and the “shower door” effect, as well as requiring less user interaction than Haeberli's system. His system automatically analyzes the edges and features of an input image to create a painted look similar to that of Haeberli's, and uses optical flow fields to animate the image; once brush strokes are generated for the first frame of an animation, pixel motion is tracked throughout successive reference frames and the motion is applied to the brush strokes, causing them to stick to their associated objects. The main limitations of Litwinowicz's system with regards to the goals of this thesis project is the need for an input image sequence, and the overly 'automated' appearance of the results –

the brush strokes are generated without any regard for artist intent and emphasis, with a relatively uniform style applied over the entire image.

Bousseau et al. [4] improved Litwinowicz's pixel advection technique and applied it to their video watercolorization system. Their system simplifies the video images with a series of 2D image processing techniques, and then applies watercolor textures to finalize the look. One of their main contributions is the success of their algorithms for advecting the watercolor texture with the animation, with much less distortion than previous techniques. The result is a system that looks good on any still frame, but also provides a seamless, novel watercolor aesthetic to the whole animation. Corot's oil paintings are sometimes painted thinly enough that the surface of the canvas shows through, and a similar texture advection effect would probably add a lot to an animation attempting to convey his style; however, a complex 2D video processing system is outside the bounds of this research, as the focus will be on the 3D rendering and stroke generation.

Since painters usually mix their colors from a very small base palette, the stochastically generated color in Litwinowicz's and many other automatic systems often gives away its

photographic origins. Park and Yoon [20] suggest a system similar to Litwinowiczs' with an additional step for generating a limited color palette by first analyzing a number of input paintings. Since the work being proposed here is designed to resemble the paintings of one particular artist, an automated color analysis process is not needed; Corot's paintings will be analyzed manually.

Some have tried to achieve a painterly look in 3D by simply using painted texture maps on 3D geometry. This technique easily achieves temporal coherency, but the resulting imagery has an unnatural 'gift-wrapped' look, where individual brush strokes recede in space and wrap around shapes instead of lying in the image plane. Objects rendered this way also have perfectly hard, well-defined edges, which are something generally not found in painted imagery. Both of these issues are evident in rocks and tree-trunks of Fig. 3, a screenshot from the video game "The Elder Scrolls IV: Oblivion" [29]. The tree foliage seems to be made up of 2D billboards and does not exhibit the 'gift-wrapped' problem.

Fig. 3. A screenshot from the painted world of “The Elder Scrolls IV: Oblivion” [29].

The painterly rendering system developed by Meier [18] significantly manages to maintain frame-to-frame coherence while presenting a convincing painterly look that avoids the 'gift-wrapped' look as well as the 'shower door' problem. Fig. 4 illustrates Meier's rendering pipeline. Particles are generated in world space on the surface of 3D geometry, and then rendered as brush strokes in image space. To determine the properties of the rendered brush strokes, a number of reference images are first created to represent brush stroke color, orientation, size, etc. These reference images are rendered using the initial 3D geometry, lighting information, and some custom shaders. During the primary render pass, the screen-space location of each particle is used to look up stroke attribute information from the corresponding location in the reference images.

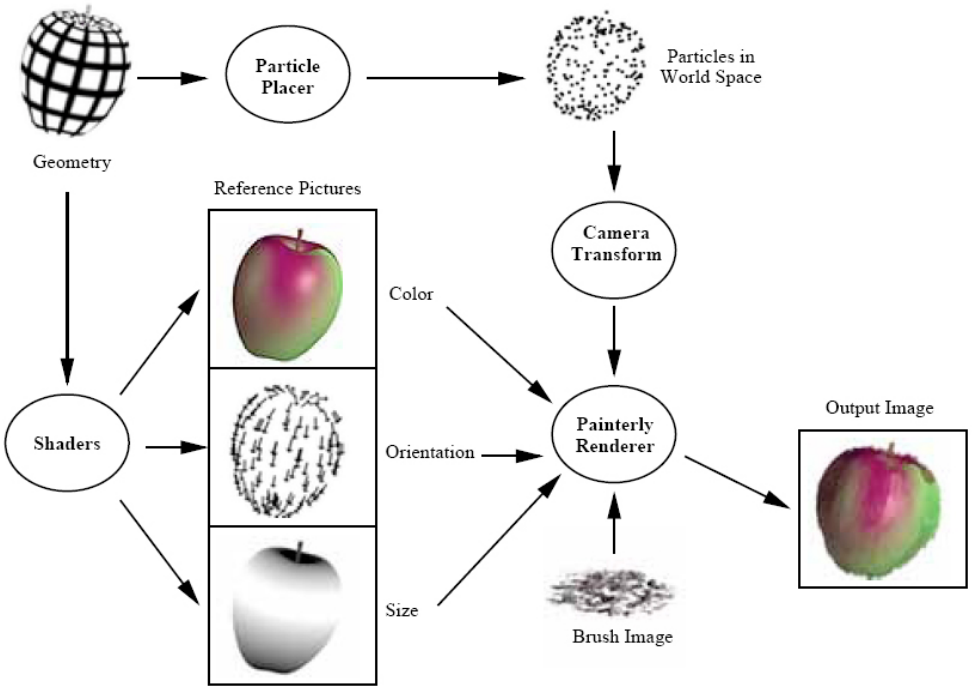

Fig. 4. Meier's rendering pipeline [18].

Meier's random-appearing brush strokes are temporally coherent, since the particles "stick" to the surface geometry during camera and object transformations. By keeping her particle strokes in world space, Meier's strokes move properly with the associated shapes instead of sticking to the image plane. Another influential idea from Meier's work is to break the painting into several layers, such as shadow, highlight, underpainting, etc., and to treat the rendering of each layer slightly differently. While this approach yields very good visual results, from a production standpoint, the need for multiple render passes adds undesirable complexity.

Davis et al. [7] present a somewhat similar way of creating painterly rendering using custom tools developed for Maya, avoiding the need to write their own custom rendering software. Their system uses particles created from surface geometry, but instead of rendering the particles themselves, they use the particles to grow splines that represent brush strokes. Each stroke is determined to be in the foreground, background or middleground, based on world coordinate space. Strokes in the background are scaled larger, to avoid the problem of strokes that look to small to have been physically painted by an artist. The brush stroke splines are then textured and rendered. Another contribution is their treatment of shadows. They identify shadow regions in the normal way, but instead of treating the shadow color as the surface color mixed with black, they use a darker shade of the surface color's complimentary color.

Many real-time systems for NPR rendering have been developed with ideas relevant to this thesis. The real-time watercolor plants of Luft and Deussen [17] use another particle-based system to achieve very convincing results. They place particles on the surface of geometry and assign a normal to each stroke particle. They then use a threshold to determine whether each particle is in a dark, medium, or light area, and render each image as a composite of three separate passes. This allows them to treat the darker areas of the painting differently from the lighter areas, which enables a more painterly look.

Haller and Sperl [13] used GPU rendering techniques to implement a real-time system based on Meier's algorithm. An initial particle set is created from geometry in a preprocessing step, and then a real-time algorithm decides which particles to render based on a desired stroke density. In cases where the camera gets too close and the initial particle set did not create enough particles, the brush stroke sizes are scaled to fill any holes. Their system uses GPU render-to-texture functionality [26] to create reference image passes in real-time, and then renders the particles as 2D billboards with brush stroke textures and attributes pulled from the reference images. They found that they could use fewer particles per object, and therefore render more objects at greater frame rates, if they rendered the final pass as a composite of their reference pass and a layer rendered from particles. Notable issues with their system include the overly uniform appearance of the brush strokes, and the 'popping' in and out of brush strokes in scenes where the camera moves.

A central concept used in real-life painting technique is the ability, and necessity, for the artist to direct the viewer's eye around the image through the designed use of light, color, brushwork, etc. The ability to design specific points of interest, as well other aspects of artistic control used by good painters, is lacking from many computer generated painterly rendering techniques. Kowalski et al [14] present a 'toon' style renderer for 'art-based' rendering that allows the user to specify visual emphasis in an image. This is done by specifying an emphasis attribute per object, which their shaders interpret in various ways to ensure that the final image's point of interest is where the artist intends.

They also specify a way of grouping silhouettes so that individual background objects do not have well-defined borders.

Artists often exaggerate the effects of light or use complimentary color hues to show changes in light. Naturalistic painters rarely use pure black to represent dark areas, and often create darker shades of color by mixing complimentary colors – not by adding black. Thus, common computer graphics lighting models, such as Phong or Lambertian shading, are not particularly conducive to art-based rendering. Several graphics researchers have investigated this problem.

Gooch et al. [10] describe a lighting model which uses a cool-to-warm color scheme that could be useful for creating painterly images. Their lighting model is designed to show volume and shading using hue and luminance changes within mid-tone colors, so that black and white can be reserved for prominent highlights and outlines. Fig. 5 shows the difference between sphere lit with their model, and spheres rendered with Phong shading. Note that even with the hue changes, their spheres still retain their "color name". In the Pixar film "Ratatouille", Director of Photography and Master Lighter Sharon Calahan approached the lighting from a painters point of view [5]; she increased the saturation levels of areas with low color values to approximate the way painters mix dark color, and she added a bit of color spill by converting the image to HSV color space and slightly blurring the hue channel.

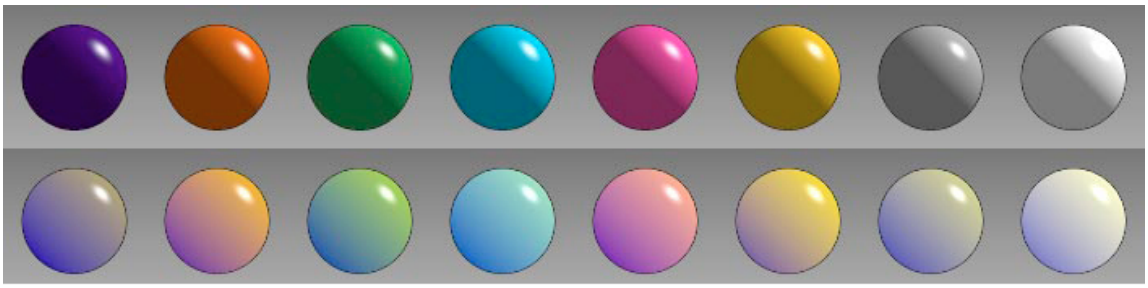

Fig. 5. Phong shading versus Gooch et al's NPR lighting model [9].

Curtis [6] and Baxter[2; 3], as well as other researchers, have developed painterly rendering systems based around the use of fluid simulation equations to model paint flow. Baxter's system allows a user to user to interact with his painting simulation using a haptic device shaped like a paint brush. As a user moves the brush in the real world, digital paint strokes are added to a virtual canvas; the pressure and speed of the user's movements are used to drive the physically simulated interaction between the brush, canvas, and paint. Baxter's system convincingly models the look of brush strokes mixing with one another, though the final results rely on the actual painting skill of the user. While systems such as Baxter's represent an important direction in NPR research, the use of physical simulation is a fundamentally different approach from that taken by this project.

Another problem facing computer-based attempts at creating painterly images is an inherent problem with the standard computer graphics RGB color model; standard RGB color-mixing models simply do not work the same way as pigment mixing in painting. Both Baxter's oil-painting system and Curtis' watercolor system use the Kubelka-Munk

pigment reflectance equations to achieve better-looking results. Fig. 6 shows an example of the different results produced by mixing yellow and blue pigments using actual paint, and those produced by linearly blending RGB colors.

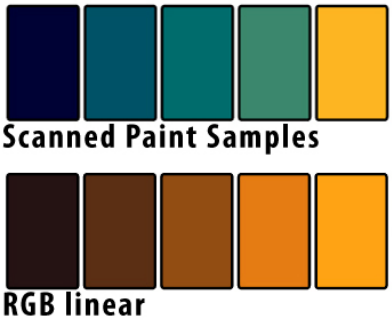

Fig. 6. Comparison of colors produced by mixing yellow and blue paint versus colors produced by RGB linear blending [3].

III.2. Procedurally Generated Trees and Plants

> *It is just as foolish for an artist, who is planning to paint out of doors, to neglect to familiarize himself with the construction of trees, as it would be for a portrait painter to understand thoroughly the anatomy of the human head and its component parts with the exception, shall we say, of the mouth; and then to paint a shapeless smootch of red where the mouth should be... [25, pg. 5]*

While a thorough botanical knowledge of trees is not necessary to artistically depict their structure, it is good to have an idea of the rules generally followed by nature. For example, as branches grow out, they will only decrease in diameter – to show a branch's diameter increasing and decreasing indiscriminately is a sure way to make it appear

unnatural. Determining and cataloging rules such as this has suggested the basis for a number of tree growing algorithms.

Lindenmayer and Prusinkiewicz devoted a lot of research to exploring “the elegance and relative simplicity of developmental algorithms, that is, the rules which describe plant development in time.”[23, v.] “L-systems” were introduced by Lindenmeyer in 1968 as a mathematical description of plant growth. L-Systems essentially consist of an initial state and a set of rules used to successively rewrite states with slightly more complicated states. The resulting system can be used to represent complex structures stemming from a few simple rules. Fig. 7 shows an L-system consisting of the initial state “b”, and the rules “b->a” and “a-> a b”.

Fig. 7. An example “L-system.” Reprinted from Prusinkiewicz, 1990 [23].

A number of researchers have developed systems with production rules designed specifically for tree branching. An early example is Honda, whose system used five observations to create a variety of tree-like shapes. Example rules from his system state

that a mother segment produces two daughter segments through each branching process, and that the daughter segments are shortened by constant ratios. The mother and daughter segments were all in the same branching plane, and the branching angles were defined with respect to the mother segment. Honda's work was extended by Aono and Kunii in a number of ways, most notably allowing the user to bias the branch positions in a particular direction to represent the effects of wind and other environmental strains [23].

Reeves and Blau [24] created a rule-based system for procedurally generating both tree branch geometry and leaf particles. Trees are built by taking user-determined parameter values, such as branching angle for the trunk and density for the leaves, and applying a set of rules. Reeves' paper also contributed a probabilistic shading model for rendering the particle leaves. Instead of standard shadow calculations, which are prohibitively costly on so many particles, shadowing is determined stochastically based on particle placement and orientation within the tree as a whole. A variety of factors raise or lower the probability that the leaves will be rendered in shadow; for example, leaves that are closer to the trunk are assumed to have a high probability of being in shadow because light must pass through the outer leaves to reach the inner leaves.

Peterson and Lee [21] identified another way to simplify the lighting process for CG trees. They noticed that when standard lighting equations without shadows were used to render clusters of randomly facing particle leaves, the resulting visual noise obscured the

overall form of the tree shapes. To solve this, they use what they call aggregate normals, which represent the normal of a simplified version of the leaf canopy's overall form. Fig. 8 shows how Peterson and Lee use a blend of the aggregate normal and the random-facing leaf normal to achieve a more visually pleasing result.

Fig. 8. Aggregate normal combined with random leaf geometry normal. Influence from aggregate normal, left to right, 100%, 50%, 0%. Reprinted from Peterson and Lee, 2006 [21].

CHAPTER IV

METHODOLOGY

IV.1. System Framework

One of the initial constraints for the painterly rendering system was that it must fit easily within an existing animation pipeline; instead of being created entirely from the ground up, the system was designed to work with Alias' Maya [11] and Pixar's Renderman [1]. Maya is a widely-used professional animation package with a powerful internal scripting language. The language, named MEL [11], provides access to core Maya functionality, and provides an interface for developing custom tools.

Renderman was chosen as the rendering software because of the power and flexibility offered by the Renderman Shading Language (RSL). RSL provides a programmable way to define an object's surface properties and light interactions. A Renderman shader can be thought of as a mini program that receives a limited set of information (lighting information, geometric surface information, etc.), and outputs a final color and opacity for each pixel based on a set of rules written by the programmer; standard computer lighting and shading models can be completely discarded and replaced with custom instructions. The language itself is similar to the C programming language, with the addition of data types specific to graphics programming, such as built in vector, color, and matrix data types and operations.

The two major components of the system are a set of procedural tree generation tools and the hand coded Renderman shaders. The tree creation scripts allow a user to adjust various parameters and then automatically generate a 3D polygonal mesh for each tree trunk and a particle field for its leaves. As the script generates a tree, it encodes the Maya object with data that will be used by the shader at render time. This way the renderer can base its shading operations on information that is not usually available; for example, the creation scripts give leaf particles a value that represents their distance along an individual branch of the tree – at render time, this value influences what color the leaves will be painted.

Because of the strong connection between the tree generation tools and the shaders, both components had to be developed in an iterative process. During the development of each version of the Renderman shaders, it often became apparent that tree creation data was needed that was not already stored in the trees; this required rewriting the tree creation scripts to store that data, rebuilding the set of testing trees, and then continuing to work on the shaders. Fig. 9 demonstrates this process.

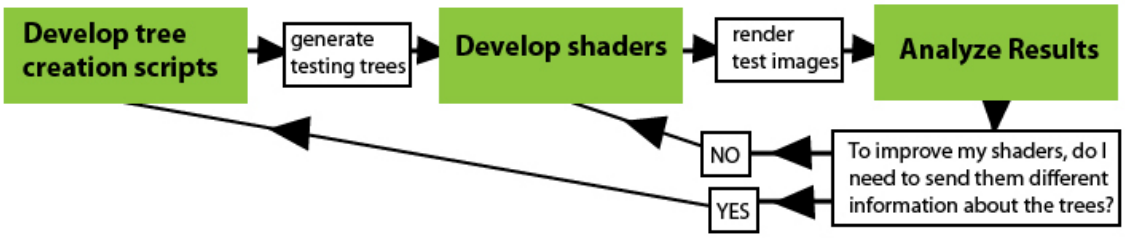

Fig. 9. The interconnected MEL script and Renderman shader development loop.

IV.2. Observations

Before implementing either of the tree creation scripts or the shaders, the first step was to study Corot's paintings and identify the most important features for the system to replicate. I was able to observe a few of his paintings in person on several occasions, but mostly worked from printed reproductions. The visual goal of the project was more the creation of a painterly style reminiscent of Corot's work than an attempt to specifically recreate any one painting, so several representative trees served as the basis for observation. Fig. 10 shows several of the trees that were used to design and evaluate the system.

Fig. 10. Examples of trees painted by Corot [30].

IV.3. Procedural Tree Generation Tools

The tree creation tools essentially consist of two MEL scripts run from within Maya. One script generates trunk and branch geometry, and the other script parses that geometry to produce particle leaves. Both scripts are dependent on user-adjusted parameters to produce controllable results.

Why generate the trees procedurally in the first place? The primary motivation for using procedural construction methods was the desire to keep the tree models themselves completely disposable. As previously shown in Fig. 9 (above), the development process was iterative; each change in the underlying structure of the trees required a new set of tree models with updated information. With a procedural work flow, the tree generation script can very quickly create a whole new batch of trees, while simultaneously determining and storing data to be used by the shaders at render time.

As can be seen in Fig. 10, above, Corot's tree trunks are generally fairly simplified and abstracted. He rarely depicts intricate tangles of branches, usually letting a few separate strokes and the arrangement of his leaves imply the underlying branching structure. Accordingly, the procedural tree system could attain an appropriate level of detail with a trunk and one level of branches.

The tree creation tools conceptually separate the tees into trunks, branches, and leaves, and allow individual control over each component. When a user runs the first tree generation script, it loads a new window containing a viewport, several buttons and some adjustable parameters. The GUI shown in Fig. 11 provides tools for separately building the trunk and branches, with parameters separated based on their relevant components. In the "create geometry" portion of the GUI, clicking on "trunk" will generate the trunk, "branches" adds branches to the existing trunk, and "LOD" will create a model of the trunk with less detail.

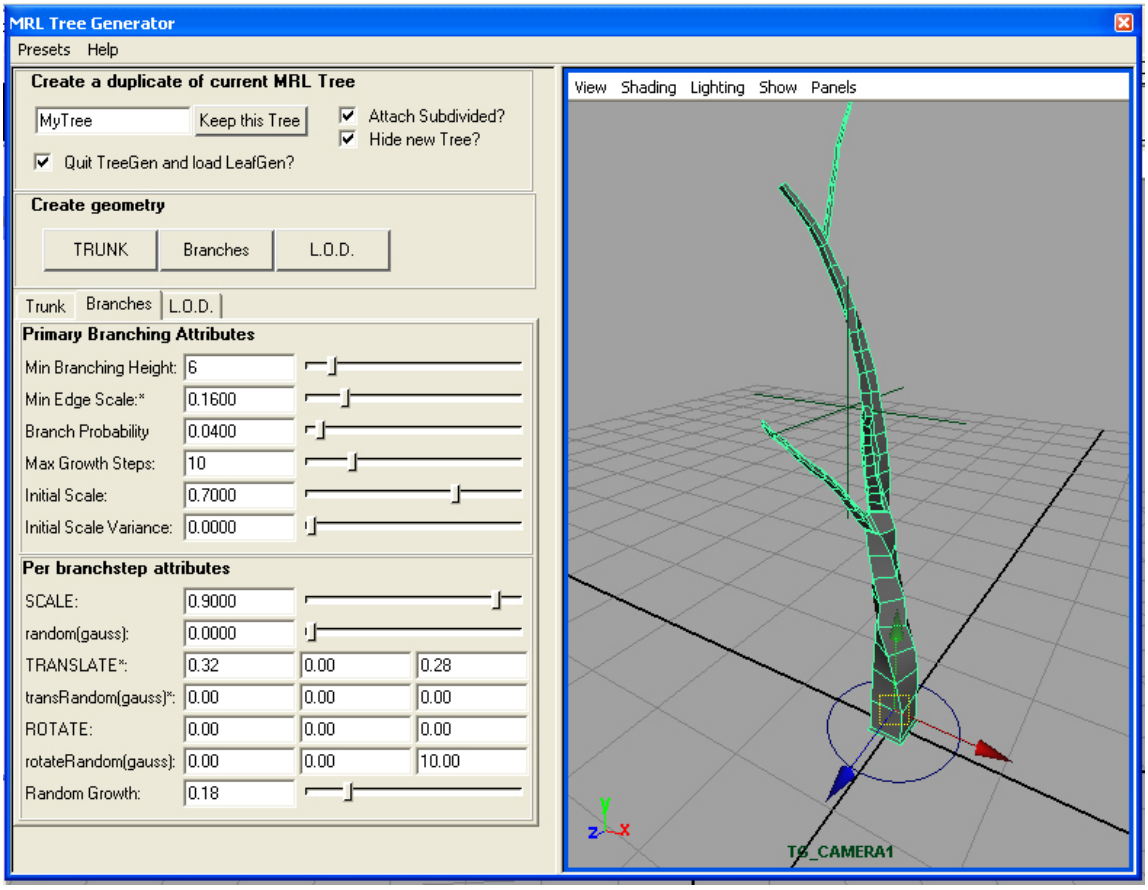

Fig. 11. GUI for MEL tree generation script.

In order to generate a trunk, the script uses MEL commands to create a cube, and then performs repeated face extrusions until the trunk reaches a maximum size or a maximum number of growth steps occur. The user does not see the trunk building process – they simply set the parameters, click the build button, and then choose to either accept or rebuild the completed trunk model. The first attribute, named "unit", sets the overall world-space scale of the tree units, and any other attribute that specifies a length or size is expressed in terms of the tree unit. The other available parameters control the initial size of the trunk, and the amount of extrusion, twist and taper at each extrusion step. The user controls these parameters by setting a base value and the amount of gaussian variation from that value. By adding random variation with a gaussian distribution, the user can quickly generate many unique trees that belong to the same family.

The branching function parses each face of the trunk geometry and determines, based on user-adjusted parameters, whether or not branching occurs; if branching does occur, the algorithm extrudes the face to create a new branch. The branching section of the GUI is separated into primary branching attributes and per-branch-step attributes.

Primary branching attributes include values for minimum and maximum vertical trunk heights that are eligible for branching, an overall probability that branching will occur, and attributes for controlling the maximum length of the new branches. They also include attributes for the first extrusion of a new branch. By starting with a cube, the

trunk generator ensures that any face on the trunk is quadrilateral, but since the extrusion algorithm allows for twisting and general randomness, a given face on the trunk may not be square or planar. Therefore, the first branch extrusion step only extrudes a small amount, and then manually adjusts the new vertex locations in an attempt to force the new face to be both planar and square. The two left hand images in Fig. 12 show that keeping the extruded faces square ensures that the final subdivided model will have a generally cylindrical shape to its branches. Also, the smaller extrusion height on the first step greatly improves the shape of the model around areas where branching occurs. The right hand side of Fig. 12 shows the undesired extra volume that occurs around the point of branching if the extrusion height is not minimized for the first step.

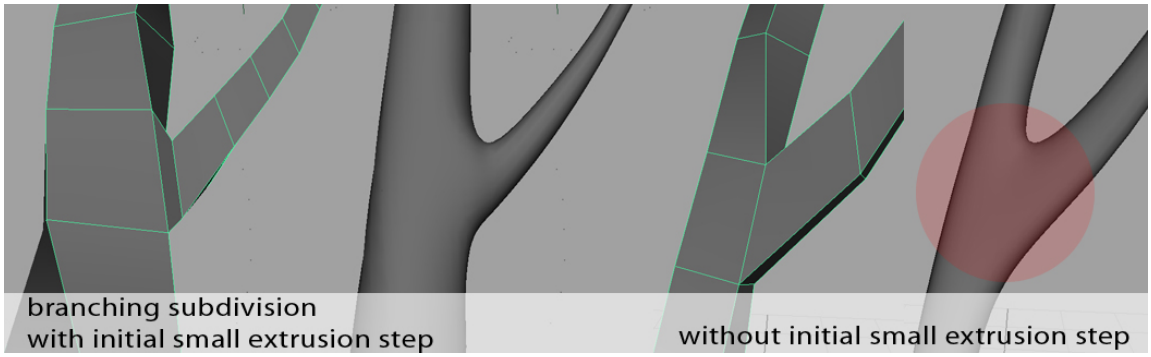

Fig. 12. Tree subdivision with and without a small first step extrusion height.

Once the first extrusion takes place, the branching algorithm continues making extrusions in a very similar way to the trunk creation algorithm. The per-branch-step attributes allow control over the extrusion length, twist and taper at each branch step. If the user is unsatisfied with the branches that are created, they may simply run the script

again to reset the tree to only the trunk model and create a new set of branches; the randomness in the script ensures that the new set of branches will be different.

Once a satisfactory tree is created, the user can enter a name and then save the tree. The script ensures that each tree has a unique name by adding a numbered suffix if necessary. Each tree is actually a group of objects, including the trunk geometry and a subdivided model of the trunk for rendering. The tree group also includes a locator object, to which are attached custom attributes for storing general information about the tree; for example, attributes for storing the beginning and ending vertices of each branch of the tree are added to the data locator, since that information is needed by the leaf generation script. Each object in the group is organized with a specific naming scheme so that the other tools can perform global operations on the trees. These tools are described later in this section.

The leaf generation script allows the user to select any existing tree object and then either generate new leaves or modify existing ones. The original leaf generation script generated NURBS spheres in formations around the tree branches. However, the results of attempting to shade the NURBS spheres in a painterly manner were unsuccessful. Since the surface normals of the NURBS spheres were used for lighting calculations, the underlying spherical shapes were always apparent in the renders. Even though some results looked somewhat promising in a still frame, any animation that rotated around the trees made it immediately clear that they were simply a cluster of warped spheres.

Switching to a particle representation of the leaves, a decision inspired by Reeves [24], brought much more success. The final leaf generation script parses the tree geometry and emits leaf particles, based on user parameters and data stored in the tree group's data locator. The GUI, as seen in Fig. 13, is very similar to the tree generation script, with the parameters grouped into several sections. All of the parameters that specify lengths or distances are expressed in terms of the tree object's unit size.

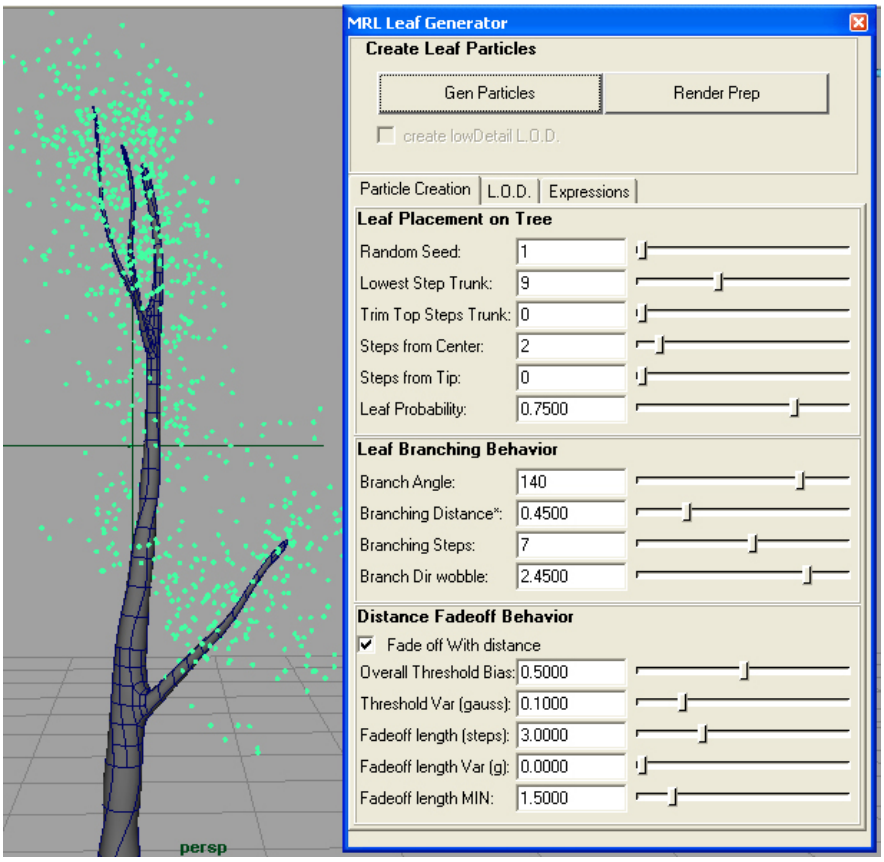

Fig. 13. GUI for MEL leaf generation tool.

Since the actual tree geometry only has one level of branching, the leaf generation algorithm attempts to imply a more complicated structure by generating leaf particles in branching patterns. The algorithm parses each vertex of the tree geometry, determines if it is eligible for branching, and then determines if branching does occur. User parameters can limit leaf branching eligibility by culling vertices at the beginning or ending of each branch, and by setting the trunk height range at which branching may occur. The user also sets the overall probability that is used to determine if leaf branching will occur on an eligible tree vertex.

The leaf branching behavior section of the GUI provides the user with parameters for leaf branching angle, length, variation, and number of particles per branch. If branching does occur, the script emits the specified number of particles outward from the tree. For each particle, the algorithm calculates a normal vector based on the branching angle and the allowed stochastic variation. Each particle is placed the specified distance away from the previous one in the direction of its normal. The result is a series of particles that form an implied branch stemming from the tree geometry, as can be seen in Fig. 14. Each particle stores information about its creation, for use later by the shader. The values it stores include its normal vector, its distance from the original branching vertex, its distance from the center of the tree, and a unique particle ID. All randomness is seeded from the seed parameter so that the same input parameters will always produce the same output results; changing only the seed parameter will produce similar results, depending on how much randomness is allowed.

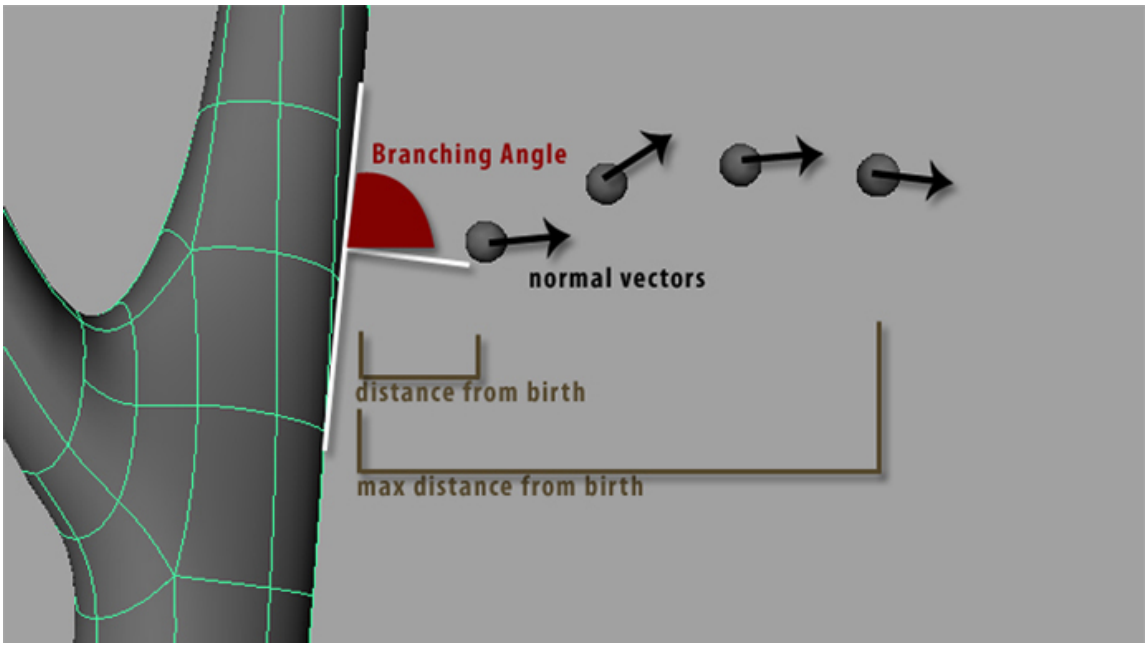

Fig. 14. Data recorded during the leaf generation process.

The particles are rendered as two-dimensional rectangular images, known as sprites, which are constrained to always face the camera. Each individual sprite is colored to look like a cluster of brush strokes, and since the sprites are 2D and always facing the camera, the strokes appear to be painted in the image plane. The particles are sorted by depth before rendering so that sprites that are closer to the camera appear to be painted on top of more distant brush strokes. The size of each sprite is determined based on user parameters and environmental factors, such as the particle's location within a tree. The size can also be dynamically changed during an animation.

One downside to using the sprites is that their square blocky shape is so different from the final brush strokes that it is hard to visualize the actual shape of the tree when working in Maya. Because of this, the leaf creation script allows the option to build proxy NURBS spheres in addition to the leaf particles. Fig. 15 shows the difference between the different tree representations, and demonstrates the usefulness of the NURBS proxies for visualizing the scene. When NURBS proxies are generated, they are automatically set so that they do not render or cast shadows.

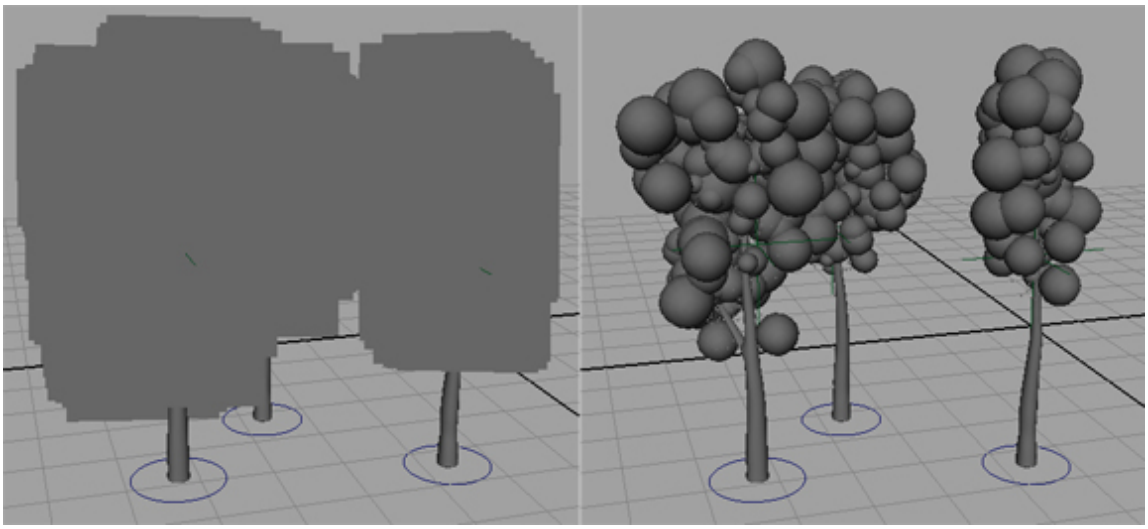

Fig. 15. Sprite and NURBS sphere representations of the same trees.

IV.4. Shading the Trees

As mentioned previously, each leaf particle stores data about its relationship to the overall tree, as well its parent tree's relationship to the entire scene. When a scene is rendered, the Renderman leaf shader uses each particle sprite's data to render it as a cluster of brush strokes of the appropriate size, style, and color.

The shading process is separated into several main steps, each of which can be modified using the shader parameters. The main steps are

- calculating illumination using particle's normal vector,
- modifying illumination based on particle data,
- calculating shadows,
- computing brush stroke mask, based on illumination and particle data,
- using illumination to pick color from color palette,
- modifying color based on atmospheric perspective,
- applying brush stroke mask to color and rendering.

IV.4.1. Calculating and Modifying Illumination

The system is designed for outdoor landscape rendering, and the lighting is generally assumed to come from the sun. Multiple lights can be used, but the system may not respond in the same way as a standard CG scene would. All of the results images were created with only one light, and shading calculations generally assume that only one light is used. The lighting calculation begins with the standard diffuse lighting equation, which is just the dot product of the particle's normal and the direction vector to the light. This value is a floating point number, which is clamped between 0 and 1. The value is stored by the shader in a variable named `lightness`, which is used later to select a color.

Consider sunlight shining on a tree with dense leaves. The leaves on the outside of the tree are most likely to receive sunlight, while leaves closer to the center of the tree are increasingly more likely to be in shadows cast by the outer leaves. This effect can be seen in Corot's trees in Fig. 10 (above). The outer leaves are painted with individual strokes of lighter colored paint, while the interior foliage is depicted with larger, darker strokes. The Renderman shaders reflect this concept by modifying the `lightness` of a particle sprite based on its position within a tree. As particles are created, they store one value that represents their distance outwards along their individual branch, and another that represents their distance from the center of the tree (the "center" is a locator that can be manipulated by the user to give a particular portion of the tree a fuller look). The shader converts these values to ratios between 0 and 1, and then modifies the particle lightness accordingly. Leaves on the outside are made a lot brighter, and leaves on the very inside are made a lot darker. The effect is generally limited to the outer innermost leaves by passing the distance ratios through gain functions prior to using them.

IV.4.2. Shadows

Shadowing initially caused a great deal of trouble, because the particle sprites cast shadows as if they were solid squares instead of individual brush strokes. Shadows cast on the ground had hard square edges, and self-shadows on the trees looked awful. Using the proxy NURBS spheres to cast shadows yielded good results on the ground and other

trees, but still had large self-shadowing artifacts. The final system solves this problem by forcing Renderman to evaluate the shader when it creates the shadow map; when Renderman evaluates the shader, it uses the true opacity of the sprites, so the particles cast shadows as if they were brush strokes instead of flat rectangles.

Instead of stopping when it hits a surface, light in the physical world continues to bounce around and illuminate objects that are not directly in the original path of the light. This indirect illumination does not normally occur in computer graphics, and when only one standard CG light is used, an object in shadow generally ends up being colored black. This is not desirable, because it completely eliminates any detail in shadowed areas and looks unnatural. To counteract this, the shader simply rescales the 0 to 1 shadow multipliers to range from 0.5 to 1.0. This means that objects in full shadow are treated as being half-illuminated. In certain situations this looks unnatural, but is generally a large improvement over using fully opaque shadows. The shader also passes the shadows through a user-controlled bias function, which allows the user to universally darken or lighten shadows as needed.

IV.4.3. Using Lightness to Select a Color

Color works very differently in standard painting and computer graphics methods. In computer graphics, color is determined by multiplying an object's default surface color by the diffuse coefficient, which is the amount of light hitting the surface; this means

that when an object is not lit, its color fades off to black. A painter however, may use completely different colors for light and dark parts of the same object. The dark and light colors are usually mixed from the same base color, but the dark colors are created by adding in a complimentary color – not by adding black. Additionally, a painter will generally mix all of the colors in a scene from a relatively small color palette; this generally adds a lusher more natural feel to a painting, and accounts for the way color is transmitted from object to object when light bounces around the physical world. A major component of this project's painterly look is that it uses each particle's `lightness` not as a diffuse coefficient, but as a lookup for a color palette.

The user specifies parameters that determine the ranges of what is considered dark, medium or light. The user also specifies appropriate colors each for dark areas, mid-tones and light areas, and arranges them from dark to light in three separate *color splines*. A color spline is a function that accepts a value from 0 to 1 and returns a color based on a list of base colors given to the spline; each color in the spline is evenly spaced, and any value along the spline is interpolated from the two colors it falls between. Fig. 16 shows an example color palette.

Fig. 16. Example color palette used for tree leaves.

The shader lighting equations determine whether a given particle lies in the dark, medium or light value range, and then uses the `lightness` value to look up a color from the appropriate spline. An additional parameter allows the user to add a little bit of noise to the color lookup, which creates color variance within each brush stroke. This is meant to emulate the natural variance that occurs within brush strokes, as the paint mixes in slightly different ways within each stroke due to physical properties of the canvas, paint and brush. Shadow colors also benefit from the use of color splines. Since the shadow calculations are applied to the `lightness` variable instead of the final color, a pixel that is in shadow simply looks up a darker color from the palette, instead of multiplying a final color by black.

IV.4.4. Generating Brush Stroke Masks

The shader prevents each sprite from rendering as a solid rectangle by creating an opacity mask to give the appearance of individual brush strokes. The opacity masks are created dynamically using thresholded noise functions. Fig. 17 shows 1D and 2D example noise functions. Note that although they appear random, they are also continuous without any distinct jumps in value. The noise function in Renderman always outputs a number between 0 and 1, and the same input will always produce the same result.

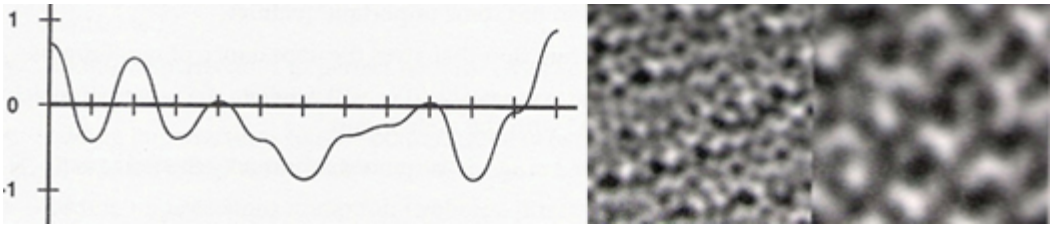

Fig. 17. 1D (left) and 2D noise functions. Two different frequencies of 2D noise are shown [26].

A sprite has an internal coordinate system, referred to in the shader as *s* and *t* coordinates, which run continuously from 0 to 1 along each direction of the sprite. A smooth-step function has two thresholds; any value below the low threshold is treated as 0, any value above the high threshold is treated as 1, and values in between the two thresholds are interpolated between 0 and 1. Fig. 18 shows the effects of using a sprite's *st* coordinates as inputs to a 2D noise, and then passing the results through smooth-step function. Varying these thresholds and the frequency of the noise can give the give the sprite appearances ranging from small non-uniform brush strokes to one large smudge of color. The bigger the different between the thresholds, the more feathering will appear on the edges of the strokes, as is demonstrated by the right-hand side of Fig. 18.

Fig. 18. Brush stroke masks created by passing noise through a smooth-step function. From left to right: low noise frequency, high frequency, and high frequency with feathered edges.

.

The full portion of the shader code for creating the brush strokes is shown in Fig. 19. The first line of the code calculates a frequency for the brush stroke noise function. The frequency of the function determines how quickly the values change, and, therefore, how large or small the brush strokes will appear; the higher the frequency, the more the strokes will appear to be small individual dots of color. The variables `centerFrequFactor` and `lightnessFrequFactor` are user-adjusted weights that determine how much the noise is influenced by the particle's distance from the center and the amount of light illuminating the particle, respectively. As mentioned previously, this idea comes from the observation that Corot's trees are painted as small detailed splotches of color on top of larger, looser brush strokes in the dark areas. Giving the noise a higher frequency for particles that are further from the center of the tree and in more direct light will keep those brush strokes smaller and allow the larger brush strokes on more inward particle sprites to show through. The `label` variable is a unique ID

number (between 0 and 1) that is stored in each particle, and helps to ensure that all particles will have unique brush stroke patterns.

```
//COMPUTE STROKE OPACITY
//**********************
//calculate appropriate frequency and generate brush stroke noise
float strokeFrequ = (1 + centerDist*centerFrequFactor +
    lightnessForStrokeFrequency*lightnessFrequFactor)+(1*label);
float stroke = noise(strokeFrequ*(s+4.122*label), strokeFrequ*(t+3.313*label));

//feather edges of particle sprite to prevents hard edges on sides
float strokeEdge = sin(PI*s) * cos((PI/2)-PI*t);
strokeEdge = bias( strokeFullness, clamp(strokeEdge, 0,1) );

//apply edge mask to stroke and then convert noise to individual strokes
stroke = stroke*strokeEdge;
stroke = smoothstep( strokeFeatherMin, strokeFeatherMax, stroke);

//clamp and set final output opacity, reduce opacity is beyond its distance threshold
float distanceFadePP = gain( 0.7, opacityPP );
Oi = clamp(stroke, 0.0, maxStrokeOpacity) * distanceFadePP;
```

Fig. 19. RSL shader code for generating the brush stroke masks.

The second line of code shown in Fig. 19 is the primary noise function that determines the brush stroke noise. This would be enough for areas in the center of each sprite, but would leave undesirable hard lines on the edges of the sprites. To prevent these hard edges, the next few lines of shader code create a rounded feather mask on the edges of the sprite. The sin and cos functions create a rounded shape with values of 1 in the center of the sprite, and smoothly changing values that fall off to 0 around the edges. The next line uses a bias function to adjust the size of this mask. The default bias settings push the shape all the way to the edge so that the maximum area of the sprite is still visible. Fig. 20 shows edge masks created with bias values ranging from 0.8, on the left, to 0.2, on the right. Multiplying the results of the stroke noise function with the

stroke edge mask creates a noise function without hard edges. This mask could have been created with a texture map, but by producing it procedurally, the system is able to modify it during the course of an animation – the grass shader makes use of this functionality and is discussed later on in this chapter.

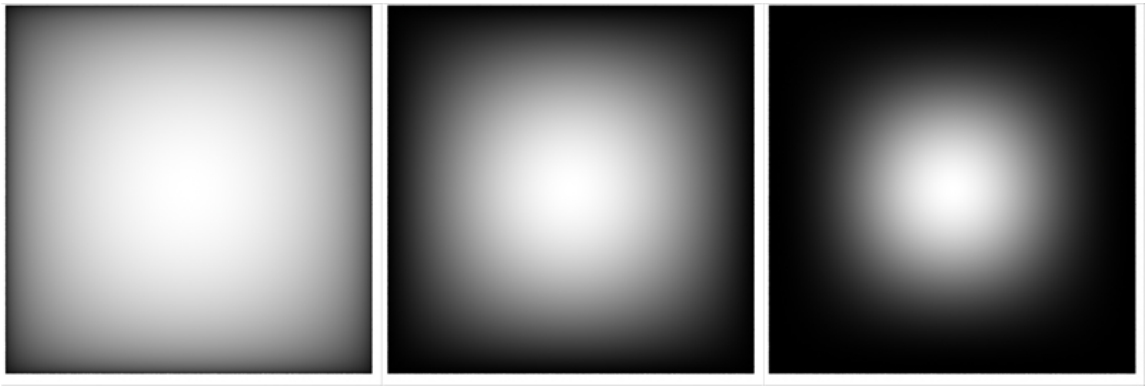

Fig. 20. Sprite edge masks. Bias values from left to right are 0.8, 0.5, 0.2.

The next line of code is the smooth-step function that transforms the noise function into the actual brush strokes. The final line clamps the opacity to a user-defined maximum opacity and then modifies it based on the distance from the camera. Clamping the opacity to a number slightly less than 1 ensures that all strokes mix slightly with the strokes behind them and helps with the overall painterly look. The opacity adjustments based on distance are used to fade out stroke detail when objects are away from the camera and are discussed in detail later in this chapter.

IV.5. Artistic Control and Shader Interface

Almost all of the shader processes discussed so far use parameters that are controllable by the user. One difficult aspect of designing any shader is figuring out how to give the user as much control as possible without presenting them with too many confusing variables and too many degrees of freedom. Since this is a research project and not a shader that will be handed off to another artist for use, I have erred on the side of control over ease of use. SLIM is a graphical interface program for changing the value of Renderman shader input parameters. Fig. 21 shows the SLIM interface to the particle sprite leaf shader.

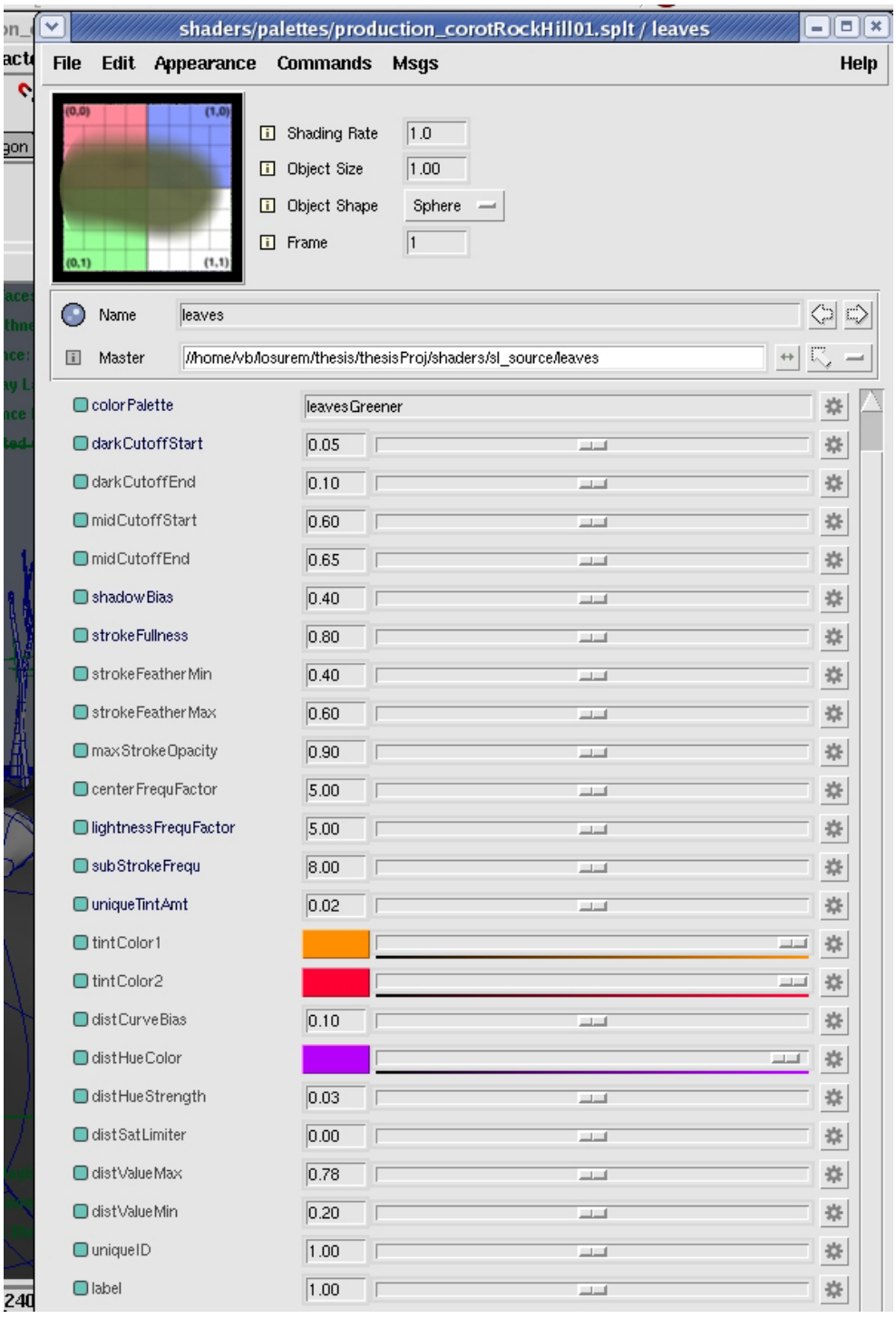

Fig. 21. SLIM interface to the leaf shader's parameters.

The first four parameters allow the user to adjust the lightness ranges that determine whether a particle is considered dark, medium or light; for example, any *lightness* value below the darkCutoffStart is evaluated on the dark color spline, and any value between the darkCutoffEnd and the midCutoffStart is evaluated in the medium color spline. Values that lie between the darkCutoffStart and darkCutoffEnd are evaluated on both color splines and then mixed together. Splitting the colors into the three ranges and allowing the user control over the boundaries of those ranges allows greater control than if all colors were represented as one continuous spline.

The shadow bias parameter is used to darken or lighten shadows. During lighting calculations a shadow multiplier between 0.5 and 1.0 is calculated and then passed through a bias function. A bias value of less than 0.5 will make shadows darker and a bias of greater than 0.5 will lessen the effect of all shadows. A bias of 0.5 will leave the shadow values alone.

A parameter named strokeFullness is used to bias the particle sprites edge mask. A higher number will push the mask closer to the edge borders, making more of the sprite visible. A lower number will increase the mask and effectively make the stroke area smaller. The strokefeatherMin and strokeFeatherMax are used by the stroke smooth-step function to clamp the brush stroke noise values into distinct splotches, and the maxStrokeOpacity is used to clamp the final stroke opacity to a maximum value.

The `centerFrequFactor` and `lightnessFrequFactor` are used to increase the frequency of the brush stroke noise – and therefore reduce the size of the brush strokes – for leaf particle sprites that are further away from the tree center and in greater illumination, respectively. The `subStrokeFrequ` is used for the color variation noise within a particular brush stroke.

Every tree is given a unique ID during its creation, which can be used to apply an overall unique tint to individual trees. The tint color is interpolated between the two tint colors based on the unique tree ID, and the `uniqueTintAmt` determines the amount of the tint color to mix into the tree's normal color. Since the tint color is uniformly applied over the entire tree, very low tint amounts must be used to avoid washing out the tree colors. If large color differences are desired between trees, the user should instead use separate leaf shaders with different color palettes.

The parameters with the `dist` prefix determine how the tree changes in appearance when it is far away from the camera. The specifics of the camera distance effects are discussed in the next section, but they are mostly designed to emulate how atmosphere effects color.

The remaining parameters are all data that is stored in the particles and should not be changed by the user. These parameters include the leaf's normal, information about its position within a tree and within the world, and unique Ids for both the tree and the leaf.

Ideally, these parameters would not be visible to the user, but it is the only way to pass custom data directly from Maya into the Renderman shader.

IV.6. Animation

In a painting, brush stroke size does not scale linearly with perspective – distant objects are not just painted with tiny strokes, but instead are generally painted with fewer brush strokes and less detail. Because of this, one of the original guidelines for the system is that it must be able to use different representations for trees that are in the background than those in the foreground, and it must be able to smoothly animate between representations. The first step in working towards this goal was developing a system for defining what constitutes foreground and background.

Renderman has a built in level-of-detail (LOD) control, and initially the system tried to make use of this functionality. With Renderman's built in LOD, a user can specify multiple models, and the renderer will automatically interpolate between them. The main purpose of the Renderman system is to reduce the complexity of scenes by using less complex models when objects are too small on the screen for a viewer to tell the difference. This means that there is no continuous interpolation between models, but instead one model simply fades into another by animating the opacity of each. Because of this, it is generally only necessary to have two or three different models. The ranges for where the transitions occur are specified in screen space area. These attributes work

very well for their intended purpose, but make Renderman LOD completely inappropriate for system that needs a continuous change over a large range of distances. Additionally, the appropriate level of detail needs to be decided by distance from the camera, not screen space size. For example, a giant mountain may be extremely far in the background and need to be painted as such, but may also still occupy a fairly large portion of the screen.

The LOD system developed for this project uses a fairly simple and intuitive way of letting the user specify which objects are foreground and which are background. The GUI created by the scene management script, shown in Fig. 22, provides the user with several useful tools for building animations. Clicking the create camera button, listed under the Scene prep tab, builds a camera with custom attributes and two circle objects constrained so they can only move along the camera's view axis; these circles allow a user to specify the distance ranges, as shown in Fig. 22. Anything closer to the camera than the near circle is rendered with the highest detail possible, and anything farther away than the distant object is rendered with the lowest detail. Objects between the two circles are assigned an interpolated distance value between 0 and 1, and rendered accordingly. The distance to the camera is calculated relative to a plane perpendicular to the view vector, which means that any objects in this plane are considered to be the same distance away. If a camera is animated, these circles follow with it, but can also be animated individually for added effect.

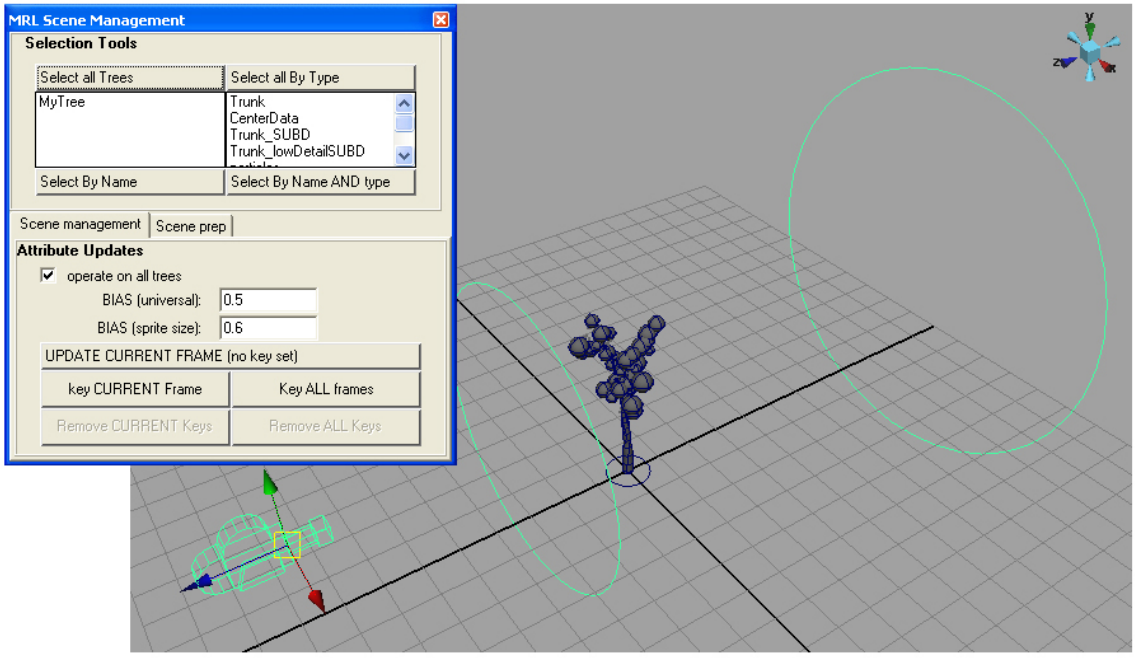

Fig. 22. Scene management MEL script GUI.

Maya allows a user to define relationships between objects using MEL-scripted expressions. Expressions are evaluated every time an event, such as changing the current animation frame, forces an update. The system uses an expression to keep track of the world space distance between the camera and its circle tools, and stores these distances in custom attributes. There is a separate pair of distance circles that is used to specify distances for the tree trunks, and these are handled the exact same way.

The scene management tool has a button that parses a scene and updates every object's distance values. For trees, it calculates the distance between the camera and the center of the tree, compares that distance to those of the camera distance circles, and then updates the tree's distance value accordingly. For testing scenes, it is generally only necessary to

update the current frame; for animations, there is another button that goes frame by frame and sets animation keys for the distance values so that they will animate correctly over the course of an animated shot.

The particle leaf objects have expressions attached to them that update several of the individual particle attributes, based on the tree's distance value. When individual leaf particles are created, they are assigned two distance value thresholds. These thresholds are used to adjust the particle's opacity based on distance from the camera. The particle is either made fully opaque, fully transparent, or, if the value falls between the thresholds, is given an interpolated opacity. The effect of this is that the particles will slowly disappear as the camera moves away from the tree. The leaf creation script allows the user to specify the minimum separation for these thresholds in order to control the distance of the fade. At render time, the shader also puts a slight gain on the opacity values in order to ease in and out of the fade. The leaf creation script also allows the user to specify a gaussian variance on the thresholds so that each leaf begins to fade away at a slightly different time.

The distance thresholds are assigned to the particles such that those farthest away from the center of the tree disappear sooner than those on the inside. Since particles on the inside have lower frequency strokes, the effect is that the apparent detail of the trees is reduced. Fig. 23 shows the same tree represented as near, far, and several stages in between. In order to maintain a relatively constant silhouette and overall tree size, the

Maya expressions continuously scale up the remaining particles to account for the missing leaves. The leaf shader also must account for the removal of leaf particles. In the high detail representation of the trees, interior leaves are artificially darkened to account for the probability that they are in shadow from the exterior leaves. As the tree gets farther away from the camera and the exterior leaves disappear, this darkening effect is diminished and then removed, to prevent the tree from appearing to darken. The shader handles this by including the distance value in its probabilistic lighting calculations.

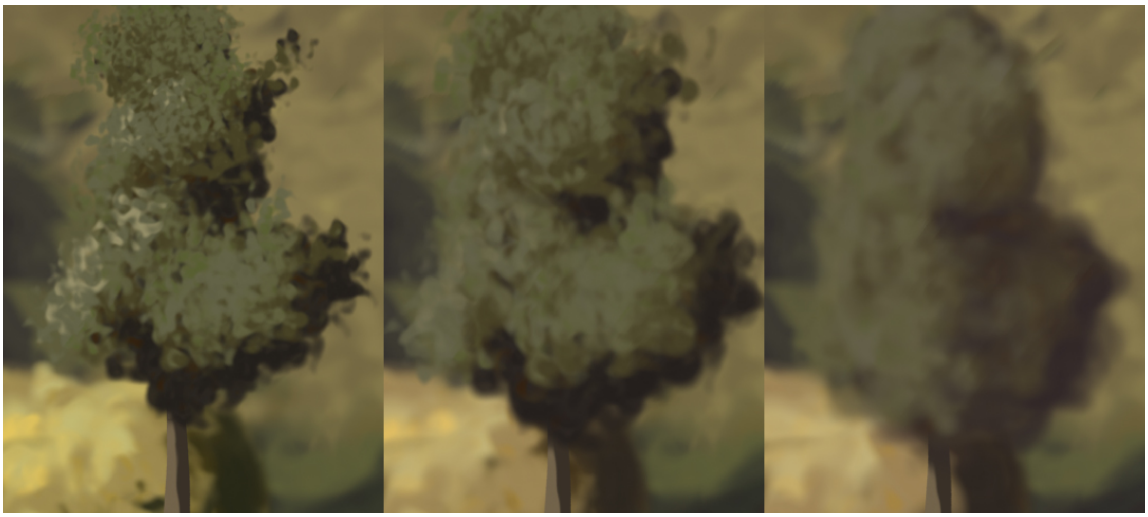

Fig. 23. Renderings of the same tree at three different depths.

In painting, the effect that atmosphere has on objects that are distant to the viewer is referred to as atmospheric perspective. In addition to losing detail clarity, distant objects tend to be painted with less contrast and saturation, and tend to have their hue shifted towards the sky color. The shader has parameters that account for each these attributes, and their effects can be seen on the trees in Fig. 23 (above). The attribute named

`distHueColor` is mixed with a distant tree's color in an amount determined by the `distHueStrength` parameter. The saturation of distant objects may be limited by the `distSatLimiter`, and the value of distant colors may be clamped by the `distValueMix` and `distValueMax` parameters. The `distCurveBias` is used to bias distance of the object from the camera, effectively changing distance values from a linear interpolation into a curve that increases more with distance.

IV.7. Generalizing the Leaf Shader for Grass, Rocks, Etc.

It was difficult to evaluate the success of the painterly rendering system when the only results were trees by themselves. By adjusting the leaf shader to fit more general forms, I was quickly able to populate an entire scene with grass, rocks, and other surfaces that fit within the same painterly style as the trees. The generalized shader was designed specifically for grass, but purposely left general enough to accommodate other surfaces.

The system's generalized brush stroke creation tool, as shown in Fig. 24, is another MEL script that allows a user to select any NURBS surface and convert it into a field of particle sprites. The script divides the NURBS surface into a user specified number of divisions in the u and v directions, and then emits a specified number of particles randomly within each section of the surface. The reason for dividing the surface instead of generating particles randomly over the entire surface is because the user can specify

separate *u* and *v* division numbers, generating more particles in one particular direction; for a long thin piece of geometry, this is required to keep an even distribution over the entire surface. When the particles are emitted, they store the surface normal and *uv* coordinates of the surface at the point of their generation.

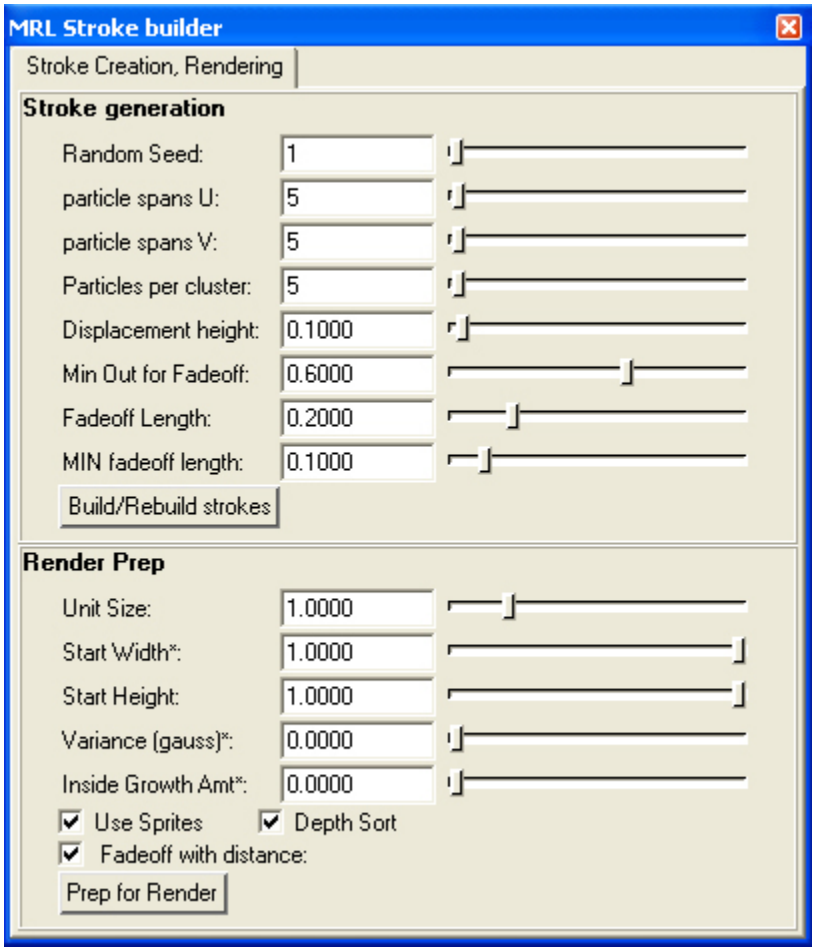

Fig. 24. GUI for the generalized stroke building tool.

The user also specifies a displacement height for the particle surface. As each particle is generated, it is displaced either up or down a random amount less than or equal to the

specified displacement height. Each particle's height is then normalized from 0 to 1, with 0 being below the surface and 1 above, and then stored for later use by the shader.

Corot often painted his grass as large low-detail blends of color punctuated by a few specific vertical brush strokes. The few, generally lighter, detail strokes are enough to give the impression that an entire hillside is covered with grass. Because of this, the generalized stroke shader is organized so that smaller detailed strokes are placed on top of darker, less-detailed strokes.

After generation, the user is able to prepare the particles for render and adjust them as necessary. The prep for render button converts the particles to particle sprites, and sizes them based on several parameters set by the user. There is a parameter for the overall size of the particle, as well as controls for adding some size randomness and making the width different than the height. Another parameter allows resizing the sprites based on their depth value; in general, it is desirable to have the deepest sprites sized much larger than the top-most sprites. This allows for fewer sprites to cover the area of a surface, and also allows for the appearance of small detail strokes on top of larger strokes.

There are also parameters that govern the effects of camera distance on the brush strokes. Brush strokes that are considered to be 'detail' strokes fade away at great distances, and one of the parameters sets the height threshold for which strokes are considered detail. Any strokes with a displacement height of greater than the threshold

are considered detail, and are eligible to fade away. Lower strokes never fade away, since if they disappeared, holes would appear in the surface. Distance thresholds are assigned randomly to the detail particles, but there is a parameter to specify the minimum distance span over which a particle may disappear.

The distance values of the generalized particles are computed in a very similar way to the tree distances. When the scene management tool is used to update scene distances, it calculates the distance of each corner of the NURBS surface, relative to the camera viewing direction and the distance circles. Individual particle distances are then interpolated from the corner distances using the particle's stored uv coordinates.

The shader for generalized particle strokes is conceptually very similar to the leaf shader; it computes a lightness value, manipulates that value based on particle data, and then uses it as a lookup to a color palette. The opacity mask of each sprite is then created with a thresholded noise algorithm that is very similar to the one used by the leaf shader. There are, however, slight differences at nearly every step.

The initial lighting calculation uses the concept of aggregate normals, as suggested by Peterson and Lee [21]. In addition to the normal of of the surface where it was generated, called the aggregate normal, each particle contains a random-facing normal. The normal vector used for lighting calculations is a blend of the aggregate normal and the random normal, based on a user controlled blending weight. Allowing the random-

normal to blend with the aggregate normal introduces some random variation on the surface of the object. Allowing too much influence from the random normal, however, risks losing the overall shape of the surface.

After the initial diffuse lighting calculation, the `lightness` of the particles is modified based on their displacement height. Particles that are deeper are artificially darkened based on a user-adjusted parameter. Without this feature, detail brush strokes might receive the same illumination, and therefore the same color as strokes beneath them, and would be invisible. Since the lower strokes are darkened slightly, the detail brush strokes show up on top.

The generalized particle strokes receive shadows the same way as the leaf particles, with the same bias function used to manipulate the overall strength of the shadows. Unlike the leaf particles, the generalized particles themselves do not cast shadows. The surface that is used to create the particles is made invisible to the primary render pass, but is set to cast shadows. Since it is roughly the same shape as the particles created from it, it gives off proper shadows.

Color values are not computed inside the generalized stroke shader. Instead, the `lightness` value, as well as the user-adjusted substroke frequency and lightness level thresholds is sent to a material definition function that is stored in the same file as the color definitions. The shader also passes the material function a user-specified string

that names the desired material. The material function takes the name of the material and the lightness values, and determines an output color using the same color palette spline methods as the leaf shader. The reason for separating the color calculations into a separate material function is that multiple shaders can then easily share the same material and color palette. Also, an individual shader can potentially have multiple materials. For example, a hillside shader could use a noise function or texture map to determine if a specific particle was in a grassy section or a dirt section, and then determine the particle's color by passing the lighting information and the name of the desired material to the material function.

The generalized particle shader allows more detailed control over the frequencies used for the stroke-generation noise. There are parameters for a base noise, and an amount to increase that noise for the detailed particles. There are also separate frequency modifiers for the *s* and *t* directions for both the base strokes and the detail strokes. The grass shader uses much higher frequencies in the *s* direction than the *t* direction of detail strokes in order to give the detail strokes the thin, vertical appearance of tufts of grass. The rock shader uses frequencies that are the same in both directions in order to keep circular, non-directional brush strokes.

The edge masks for the particle strokes are computed in a nearly identical method to those of the leaf strokes. One of the problems with using the particle system to cover a piece of landscape is trying to generate enough particles to completely cover the surface

with 'paint', without having so many particles that render times are too slowed. Particle sprites in areas that have normals parallel to the view direction need to be large to maximize coverage, but sprites at the portions of a surface with normals perpendicular to the view direction, such as at the crests of the hills in Fig. 25, need to have smaller heights so that they do not extend too far above the surface of the hill. The shader handles this by adjusting the vertical portion of the edge mask gain function according to the dot product of the surface normal and the viewing direction; regular sprites are rendered at their full size, while sprites at the crest of a hill are drastically shorted by the increased mask size. This effect is visible in the left hill of Fig. 25, which is missing the blurry circular strokes visible in the top right of the right hill.

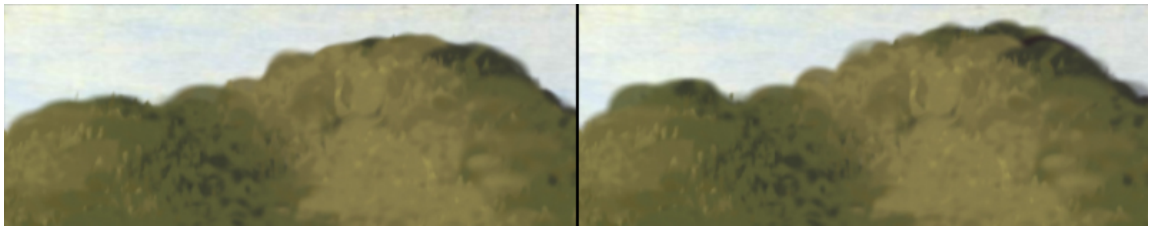

Fig. 25. Dynamic resizing of edge masks, based on viewing angle.

The generalized particle shader handles the effects of atmospheric perspective in the same way as the leaf shader, with identical controls for the variation of hue, value and saturation with distance.

CHAPTER V

DISCUSSION OF RESULTS

V.1. Overview

The overall result is a temporally coherent system that achieves a painterly look for 3D rendered animations. Fig. 26 through Fig. 29 show some still frames from animations produced with the system. Light and color are handled in a manner that adds to the painterly aesthetic of the final imagery. The system is able to convey form and depth, while still appearing to be painted in a 2-dimensional image plane. The images are rendered in a style that is clearly influenced by Corot, while not specifically recreating any of his paintings. In the judgment of several viewers, the system achieves a great deal of success in producing a painterly look; several have even mistaken rendered images for photographs of paintings.

Fig. 26. Painterly rendered windmill scene with clearly visible atmospheric perspective.

Fig. 27. Painterly rendered rocks, grass and trees.

Fig. 28. A demonstration of shadows and moving light sources.

Fig. 29. A painterly rendered image with many trees.

The use of procedural strokes is fairly successful at creating a painterly look. The shader parameters offer enough control over the brush strokes to effectively convey different surface types, such as grass or rock, and to depict these surfaces at varying distances. Many systems use predefined brush stroke textures, but the procedural nature of this system's brush strokes allow for a lot of dynamic manipulation and variance that would probably be harder to achieve using predefined stroke textures. Even in large areas covered with one shader, such as the grassy hills from Fig. 29, the randomly created procedural brush strokes offer enough variance that no pattern is discernible pattern.

One of the main goals of the system was to represent depth in a painterly way, both in terms of brushwork and color. All of the coloring, depth effects and brush stroke stylization visible in the above images was done at render-time - all of the images were rendered in one pass, and have had no post processing applied to them. The grass shader used on the hills in the windmill painting clearly demonstrates the system's effectiveness at conveying depth. The shader is the same on all three hills, and all visible changes are caused only by distance to the camera. The hill to the right of the windmill appears much farther away due to color changes and the use of fewer, larger brush strokes. These depth effects are all dynamically controlled by the shader, and can be animated to smoothly change over the course of an animation.

Animations created with the system are temporally coherent, and do not exhibit either the shower door problem or the gift-wrapped look. The brush strokes look as though

they were painted in the image plane, but the three-dimensional form of objects is not lost. Complete camera freedom is allowed, and objects smoothly transition between different levels of detail. Dynamic changes in atmospheric perspective and brush stroke stylization add a temporal painterly element to animated sequences.

V.2. Usability

The tree creation tools are very easy to use. A tree can be created in seconds, with only a few mouse clicks. The scripts successfully accomplish their primary goal, with is to provide an interface for the quick creation of generalized tree shapes. However, attempting to create very specific shapes can be counterintuitive and sometimes impossible. Although they do not function as an artist tool for creating art directable scenes, as a testing tool, the tree generation scripts work quite well. Due to the large amount of randomness in the script, it will occasionally generate completely inappropriate shapes; however, it only takes one mouse click to discard these shapes and rebuild a new one.

One feature designed to improve usability is a set of functions for saving and loading parameter settings. If a user calibrates the tree generation parameters to produce a desirable type of trees, they may save those settings as a named preset and then reload them for later use.

The system also allows the user to label trees as they are generated with names such as “poplar” or “tall tree”, and then perform universal actions on these groups of trees. The scene management tools allow a user to select groups of trees by name, or subparts of trees by name, or even specifically by name and type. This allows the user to, for example, use one mouse click to select all leaf particle shapes from the “poplar” trees in a scene and then apply a shader or perform some other action to only these objects.

The scene management also makes handling the distance calculations easy. When it is time to prepare an animation for rendering, the user may click on a button, and the script will parse the animation frame by frame, setting all distance values properly.

Both the leaf scripts and the generalized stroke scripts allow a user to modify particles after they are created. For example, the user may select a tree and load the leaf generation script. Instead of re-generating the leaf particles, the user may choose to only adjust one or more of the size parameters. Also, if a user wishes to generate particles with slightly different values, they may select a tree and run the leaf generation script. When it load the GUI, it will already have the parameters pre-set to the values that were used to create the leaves, instead of the default values; this allows the user to make a few slight changes to the parameters and regenerate the particles, instead of trying to remember the original settings that were used. The generalized particle script operates the same way.

Another useful byproduct of the way the generalized particle generation script works is that model perfection is unnecessary. Landscapes may be created by arranging NURBS surfaces so that they roughly correspond to each other – it is not necessary to carefully stitch the surfaces together, as it would be if they were being rendered normally. This allows for 'sketchy' modeling, and speeds up the process of modeling a landscape.

The shader is designed to emulate a relatively loose, nonuniform style, but does allow some control over the brush strokes. Changing the relative frequency and direction of brush strokes can make them appear to represent different surfaces, such as grass, leaves or rock. Changing other parameters can change the aesthetic of the entire image. Fig. 30 shows how changing the noise smooth-step thresholds and opacity clamps can create a fair amount of variation in the appearance of the strokes.

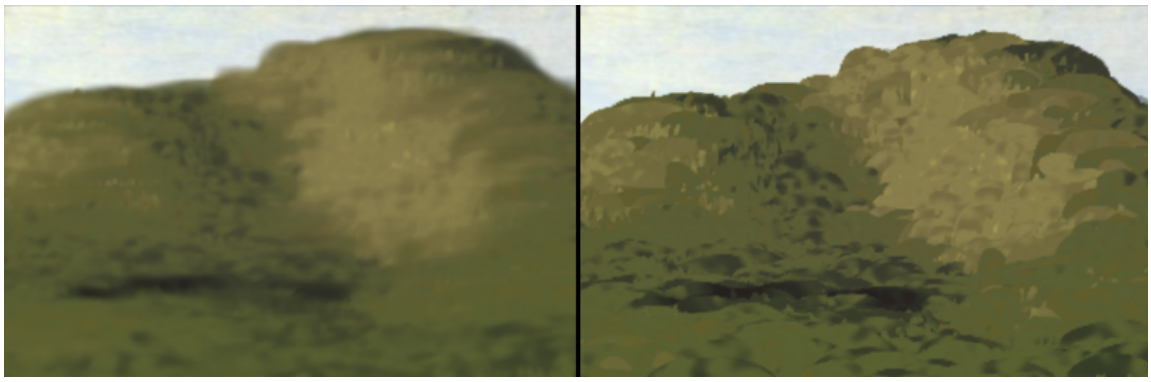

Fig. 30. The effects of changing brush stroke opacity and feathering.

The scene management tools are quite accessible and greatly speed up the process of building and animating a scene. Both the tree generation MEL scripts and the shaders were designed with control placed over ease of use, and have an intimidating number of parameters for someone unfamiliar with the system. While they are easy to use for someone who does know the system, they would need to be streamlined and better documented for use by other artists.

V.3. Known Problems

Although the system achieves most of the goals originally specified, there are a few areas in which improvements could be made. The most noticeable problem with the final images is the lack of foreground detail. The highest detail allowed by the system generally works well for most objects until they are in the extreme foreground. Both grass and trees look blurry with brush strokes that are too large when they get too near the camera. This could probably be fixed by extending the LOD system to include a method for handling objects in the extreme foreground. In many of Corot's paintings, trees that are very close to the viewer are depicted as very sparse, separated brush strokes. Perhaps as trees get very close to the camera, the interior leaves could disappear, allowing the viewer to see through the foliage. The process for doing so could be added using many of the same methods that allow the exterior leaves to disappear as the tree recedes into the distance. Some of the lack of detail in foreground objects also comes from the way that distances are computed on the grass particles. For a large patch of

grass, it is fairly inaccurate to base the distances only on the relationship between the camera and the corners of the NURBS patch; this seems to occasionally assign incorrect distance values to foreground particles. This method was chosen for speed and ease, but should probably be replaced with a more accurate system.

Another problem with the images is that some areas appear too procedural and lack the variation and detail of brush stroke styles that would be present in a real painting. The brush strokes are controllable to a degree, but they lack the finesse and variety of styles that Corot employed in his paintings. The colors are also not as varied and rich as they are in Corot's paintings. Some of this is because of the limitations of RGB color mixing, but some of it is also because of the difficulties in choosing a proper palette; it is very difficult to match Corot's eye for color.

One problem with the animations is that the particle sprites do sometimes pop in and out of view as a camera moves. The reason for this is that the particles are really just single points in space, and the sprites rotate around the particles to match the camera viewing angle. It is possible for the camera angle to change such that one sprite will suddenly pop in front of another sprite. Fig. 31 demonstrates this problem, showing how the sprites react as a camera rotates around a scene.

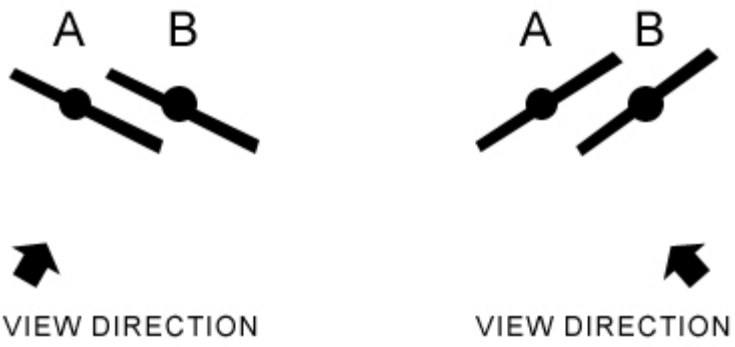

Fig. 31. Sprite popping problem. Because the sprites are oriented towards the camera, a change in viewing angle will cause one sprite to move in front of the other.

This problem is fairly fundamental to the way sprites are used by the system. Nevertheless, since the popping seems to occur fairly regularly over the entire image, it is not nearly as distracting as if it happened only in one particular spot. Since it is just general visual noise, it is fairly easy to tune out. This problem could probably be fixed with some sort of localized frame-blending or other post-processing technique. Expanding the popping to be a smooth change over the course of three to five frames would probably make it unnoticeable.

Another problem with animations is that it is difficult to properly calibrate the system that adds and removes leaf particles with distance. When it is not calibrated correctly, the silhouettes fluctuate in size, and some parts of the tree appear to change too suddenly. In a complex scene, however, this is not particularly noticeable.

V.4. Future Work

The generalized particle stroke generator is a big step towards making the system useful as a general rendering system, but it would benefit from additional control over the brush stroke styles. The current controls allows for roundish shapes and either dominantly vertical or horizontal strokes, but does not allow much major control beyond that. One potentially simple improvement would be to allow control over the particle sprites' "twist" property, which allows the sprite to rotate around the camera's view axis. This would introduce more variance in the strokes and allow greater control over their primary direction, which would add to the believability of the resulting images.

Also, there is currently no support for animating surface deformations with the system. Once particles are created, they have no direct tie to the surfaces they were generated from. Perhaps the simplest possible solution would be to use Maya's built in deformation system; the version of Maya used for this project has a documented software bug that does not preserve custom particle attributes when they are animated with deformers; since the particle's custom data is absolutely crucial for this rendering system, the use of deformers was not an option. The problem may well have been fixed in more recent versions of Maya, and then the use of wrap deformers could be a quick way to add a lot of functionality to the system. However it is accomplished, the system would be more useful if it allowed the animation of more than just the camera.

It would also be useful to add shaders for generating procedural skies and water. Both of these could probably be achieved with relatively few modifications to the generalized stroke shader. It could also be interesting to address the interaction of the paint and the surface of the canvas. Corot's paintings are occasionally painted thinly enough to allow the canvas surface to show through, which is an element of his paintings not addressed in the rendering system.

CHAPTER VI

CONCLUSION

The primary goal of this research project was to develop techniques for creating 3D animations with a visual aesthetic similar to the work of French painter Camille Corot. Instead of developing an isolated system, the development process emphasized the creative expansion of existing software, which allowed for more robust results and quicker development times.

Many efforts in NPR oil research have taken the broad approach of attempting to replicate oil painting as a medium, but the vast array of styles and effects possible with physical oil paint makes such an endeavor far beyond the scope of a master's thesis. The choice to limit the system to replicating one particular artist's style allowed for placing more emphasis on visual quality than range. Narrowing the primary focus further still to just trees allowed more time to be spent exploring the possibilities offered by time-based oil painting.

The resulting system provides a flexible interface with powerful tools for quickly generating painterly renderings of 3D scenes. Images created with the system clearly exhibit Corot's influence, and allow for CG light and form to be communicated with painterly language. The power of the automated brush stroke system comes at the cost of

the per-brush-stroke complete artistic control that a real painter would have, but the shaders and tools allow a large degree of high level artistic control.

Animations created with the system combine film language and painting technique to bring a new aesthetic to time-based work. The results of this research suggest ideas that could be explored at a much greater depth in the future, but, as it is now, the system already provides a new tool for artistic expression.

REFERENCES

[1] Apodaca, A. and Gritz, L. *Advanced Renderman: Creating CGI for Motion Pictures.* San Diego, CA: Academic Press, 2000.

[2] Baxter, W., Wendt, J., and Lin, M. C. "IMPaSTo: a realistic, interactive model for paint." In *Proceedings of the 3rd International Symposium on Non-Photorealistic Animation and Rendering* (Annecy, France, June 07 - 09, 2004). S. N. Spencer, ed. NPAR '04. New York, NY: ACM Press, pp. 45-148. 2004.

[3] Baxter, B., Scheib, V., Lin, M. C., and Manocha, D. "DAB: interactive haptic painting with 3D virtual brushes." In *Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques* SIGGRAPH '01. New York, NY: ACM Press, pp. 461-468. 2001.

[4] Bousseau, A., Neyret, F., Thollot, J., and Salesin, D. Video watercolorization using bidirectional texture advection. *ACM Trans. Graph.* Vol. 26, no. 3, article 104. July 2007.

[5] Calahan, S. "Lighting on *Ratatouille*". Lecture, Texas A&M University, Feb 7, 2008.

[6] 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* International Conference on Computer Graphics and Interactive Techniques. New York, NY: ACM Press/Addison-Wesley Publishing Co., pp. 421-430. 1997.

[7] Davis, A., Johnson, S. R. "Impressionist paint 2008." *Proceedings of Computational Aesthetics in Graphics, Visualization and Imaging* 2008, to appear.

[8] Fabbri, F, ed. *Corot: I Maestri Del Colore.* Milan, Italy: A. & G. Marco, 1965.

[9] Gooch, B. and Gooch, A. *Non-photorealistic Rendering.* Natick, MA: A K Peters, Ltd., 2001.

[10] Gooch, A., Gooch, B., Shirley, P., and Cohen, E. "A non-photorealistic lighting model for automatic technical illustration." In *Proceedings of the 25th Annual Conference on Computer Graphics and interactive Techniques* SIGGRAPH '98. , New York, NY: ACM Press, pp. 447-452. 1998.

[11] Gould, D. *Complete Maya Programming: An Extensive Guide to MEL and C++ API.* San Francisco, CA: Morgan Kauffman, 2002.

[12] Haeberli, P. “Paint by numbers: abstract image representations.” *SIGGRAPH Comput. Graph.* Vol. 24, no. 4, pp. 207-214. Sep 1990.

[13] Haller, M. and Sperl, D. “Real-time painterly rendering for MR applications.” In *Proceedings of the 2nd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia* (Singapore, June 15 - 18, 2004). S. N. Spencer, ed. GRAPHITE '04. New York, NY: ACM Press, pp. 30-38. 2004.

[14] Kowalski, M. A., Hughes, J. F., Rubin, C. B., and Ohya, J.“User-guided composition effects for art-based rendering.” In *Proceedings of the 2001 Symposium on Interactive 3D Graphics* I3D '01. New York, NY: ACM, pp. 99-102. 2001.

[15] Leymarie, J. *Corot.* New York, NY: Rizzoli International Publications, Inc., 1979.

[16] Litwinowicz, P. “Processing images and video for an impressionist effect.” In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques.* New York, NY: ACM Press/Addison-Wesley Publishing Co., pp. 407-414. 1997.

[17] Luft, T. and Deussen, O. “Real-time watercolor illustrations of plants using a blurred depth test.” In *Proceedings of the 4th International Symposium on Non-Photorealistic Animation and Rendering* (Annecy, France, June 05 - 07, 2006). NPAR '06. New York, NY: ACM Press, pp.11-20. 2006.

[18] Meier, B. J. “Painterly rendering for animation.” In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques* SIGGRAPH '96. New York, NY: ACM Press, pp. 477-484. 1996.

[19] Mettais, V. *Louvre - 7 Centuries of Painting.* Versailles, France: Art Lys, 2000.

[20] Park, YS and Yoon, KH. “Adaptive brush stroke generation for painterly rendering.” In *Proceedings of Eurographics 2004, Short Presentations,* EUROGRAPHICS, Aire-la-Ville, Switzerland, pp. 65--68. 2004.

[21] Peterson, S. and Lee, L. “Simplified tree lighting using aggregate normals.” In *ACM SIGGRAPH 2006 Sketches* (Boston, Massachusetts, July 30 - August 03, 2006). SIGGRAPH '06. New York, NY: ACM Press, pp. 47. 2006.

[22] Petrides, G. A. *Peterson Field Guides: A Field Guide to Trees and Shrubs.* Boston, MA: Houghton Mifflin Company, 1958.

[23] Prusinkiewicz, P. and Lindenmayer, A. *The Algorithmic Beauty of Plants.* New York, NY: Spinger-Verlag, 1990.

[24] Reeves, W. T. and Blau, R. "Approximate and probabilistic algorithms for shading and rendering structured particle systems." In *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques* SIGGRAPH '85. New York, NY: ACM Press, pp. 313-322. 1985.

[25] Rines, F. M. *Design and Construction in Tree Drawing.* Pelham, NY: Bridgman Publishers, Inc. 1936.

[26] Rost, R. J. *OpenGL Shading Language*. Boston, MA: Addison-Wesley, 2004

[27] Strothotte, T. and Schlechtweg, S. *Non-photorealistic Computer Graphics.* San Francisco, CA: Morgan Kauffman, 2002.

[28] Schwarz, M., Isenberg, T., Mason, K., and Carpendale, S. "Modeling with rendering primitives: an interactive non-photorealistic canvas." In *Proceedings of the 5th International Symposium on Non-Photorealistic Animation and Rendering* (San Diego, California, August 04 - 05, 2007). NPAR '07. New York, NY: ACM Press, pp. 15-22. 2007.

[29] The Elder Scrolls IV: Oblivion. Bethesda Softworks, 2006.

[30] The Yorck Project: 10,000 Meisterwerke der Malerei. http://en.wikipedia.org/wiki/Jean-Baptiste-Camille_Corot.

APPENDIX A

SUPPLEMENTAL MOVIE FILES

Several supplemental Quicktime movies are available for download. These animations demonstrate the system's temporally coherent response to varying speeds of forward and rotational camera motion; as distance from the camera changes, the shape, size and color of brush strokes are automatically modified. Also shown are the effects of moving camera distance markers independently from the camera, and animating the position of the light source.

VITA

Name: Michael Losure

Address: Texas A&M University
C418 Langford Center
3137 TAMU
College Station, TX 77843-3137

Email Address: losurem@gmail.com

Education: B.A., Computer Science and Visual Arts, Union College, 2004
M.S., Visualization Sciences, Texas A&M University, 2008