8-2014

# Data-Driven Scratch Generation for Rigid-Body Models

Ashley N. Anderson
*Clemson University*

# Data-driven Scratch Generation for Rigid-Body Models

---

A Thesis
Presented to
the Graduate School of
Clemson University

---

In Partial Fulfillment
of the Requirements for the Degree
Master of Fine Arts
Digital Production Arts

---

by
Ashley N. Anderson
August 2014

---

Accepted by:
Dr. Donald H. House, Committee Chair
Dr. Timothy A. Davis
Dr. Joshua A. Levine

# Abstract

The procedural scratch generation process for rigid-body models automates the process of locating and painting scratches on a texture map. A script computes points where a scratch is most likely to occur at each frame of an animation sequence based on whether the distance between two objects lies within a threshold. These points are then used to create a map of the scratch pattern to guide the blending of scratches onto the image. The tool may be useful to expedite the animated film production pipeline as well as predict areas of wear on machines before they are manufactured.

# Dedication

This thesis is dedicated to my family for putting up with all my long hours, complaints and tears. I could not ask for greater support. Thank you for backing me in everything I do.

Also, this thesis is dedicated to my best friend, Ali Gillespie. You have stayed by my side despite how busy or stressed I became over the past three years. You are a true friend.

# Acknowledgments

Thank you to Dr. House, my advisor, for your assistance and tempered guidance through the thesis process. You were with me every step of the way, and I could not have accomplished all that I did without your advice.

Thank you to my committee, Dr. Davis and Dr. Levine, for their guidance and support.

Thank you to Dr. Duchowski for providing those excellent images of heatmaps from his eye-tracking research.

Thank you to the Roboasis team for producing such great work to build upon.

Finally, thank you to Dr. Malloy for providing assistance with formatting.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

In computer graphics, surfacing, also called shading, means applying color, surface bumps, transparency, reflection, refraction, glint highlights, or any other material attributes to the surface of an object. The concept of surfacing closely relates to texturing, applying an image to the surface to create detail, such as labels on a robot, racing stripes on a car or veins of a leaf [5]. Shading and texturing go hand in hand to create a photo-realistic rendered image, as seen in Figure 1.1.

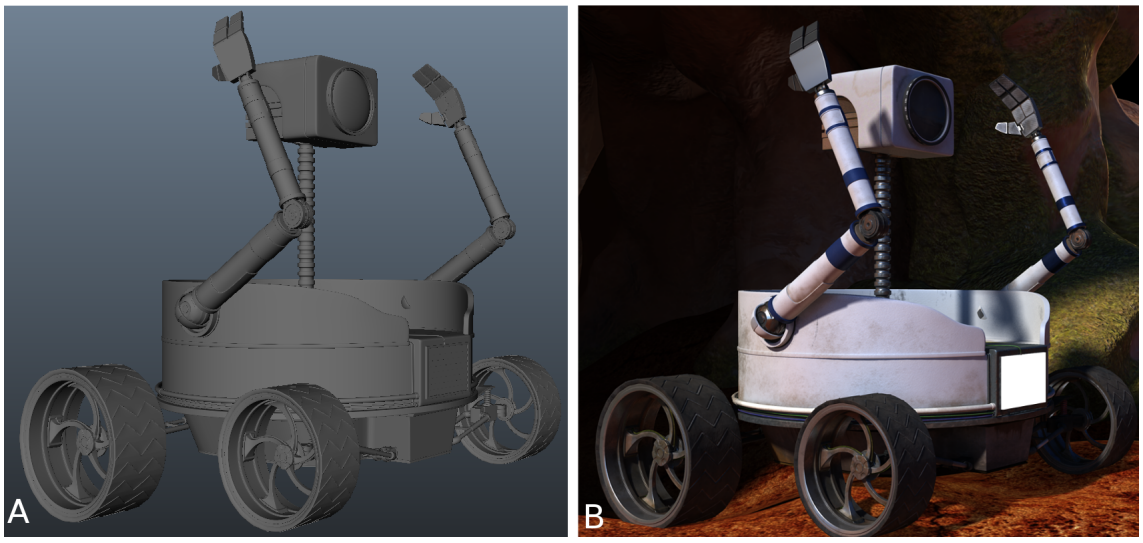



Figure 1.1: A) The model before surfacing. B) The model after surfacing.

Surfacing poses a significant challenge for the design of any computer-generated character. Artists must often ask themselves what materials will compose each part of the character and what

details those surfaces will possess. Making these decisions can be a daunting task, especially when one is unfamiliar with the object in question.

To many people, machines, especially how they function, are foreign concepts. When faced with the challenge of surfacing a robotic character, deciding whether or not to place a scratch on a surface can be one of the most difficult and confusing tasks.

This artistic challenge is the inspiration for this thesis project, which consists of a data-driven method to determine the location of scratches on a rigid body model based on the model's regular pattern of motion. The execution of a data-driven method is controlled, or "driven" by an input of data points or parameters [13]. If this project is successful, surfacing previously done by hand will become much simpler and less time consuming.

## 1.1 Project Structure

The project consists of two components. The first component is a Python script that runs within the Maya 3D software package. The script calls various Maya commands using Python and the OpenMaya API to achieve the goals of the thesis project. The function of the script is to gather data points from geometry on each frame of an animation sequence, and then pass the data to the next component of the project.

The second component is a C++ program that generates a visiualization similar to a heat map from the data gathered from the script within Maya. A heat map is a graphical representation of a data matrix in which each item in the matrix is shown as a color in the image [8]. As a general rule, color is used to indicate the density of a particular data value.

An example of a heat map is shown below in Figure 1.2. This heat map depicts eye-tracking data of subjects asked to look at the presented images. Because of the high density patten of color on the right side, we can infer that side of the image attracted the subjects attention. Heat maps of this nature are popular in marketing and advertising because they highlight what draws the subject's focus. In our work, the heat map shows where an object received the most activity, and subsequently, scratches.

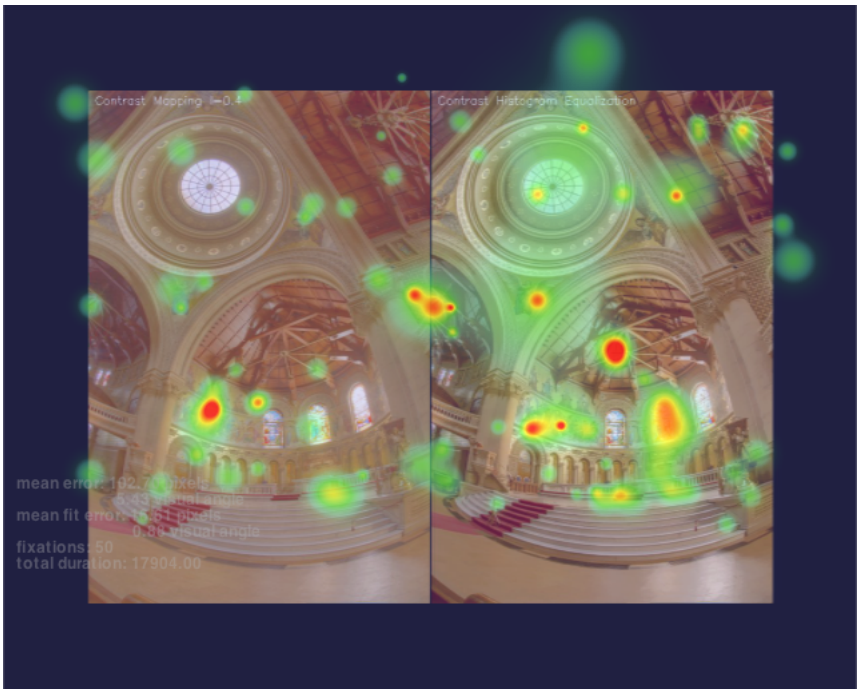



Figure 1.2: Heat Map generated by eye-tracking data. Provided by Andrew Duchowski of Clemson University. The red to green gradient correlates to data density. Red is the most dense while green is the least dense.

After the visualization is generated, it is used as a heat map to blend scratches onto an existing texture map in areas of greatest data point density. A texture map is an image used to depict surface detail. The verticies of the model are mapped to coordinates within the image, called UV coordinates, which range from 0 to 1 [1].

Then the map is placed on a transparent layer over the original texture map image. The artist can either decide to end the process here and draw their own scratches within these areas, or they can also choose to blend scratches through image processing onto areas the heat map indicates. Finally, a bump map, or an image in which greyscale values add highlights and shadows to the rendered image without manipulating geometry, is generated from the resulting texture to create a more realistic look [5].

Lastly, the procedure will be demonstrated on an existing model. In order to evaluate the success of our approach, the model with procedurally placed scratches will be compared to a model with scratches placed by hand using an artist's educated guesses.

# Chapter 2

# Related Work

Other researchers have conducted studies relating to procedural wear and tear, physically-based scratch generation, procedural textures with replicated image editing, and a procedural approach to materials. The following sections will review this work.

## 2.1 Weathering

Bosch et al. [2] conducted a research study on the simulation of weathering on buildings due to flowing water. To accomplish this, they lifted data from photographs. Then, they extracted a "stain degree map" from the photographic data, which they used to apply realistic weathering patterns onto existing geometry of buildings. In addition, they extracted high-frequency details of the weathering to produce small scale variations in the final effect. This method created a natural look. Our project has a similar design structure.

Gu et al. [9] conducted another study on measuring, modeling and rendering time-varying surface appearance. Their research focused on natural phenonema such as burning, wetting, decomposition, corrosion and rust of various materials. First, they collected 26 samples of these processes and placed them into a database. Then, they used a model that included variation of temporal appearance and space-dependent textures to define the evolution of patterns caused by natural processes. They called this model the STAF model, or Space-Time Appearance Factorization model. It allowed simulation control over certain areas of the texture as well the ability to freely edit those areas.

Mérillou and Ghazanfarpour developed a physically-based simulation technique which can imitate a vast variety of weathering processes called y-ton tracing [10]. It was based off photon mapping, which is a lighting system based off physical particles of light. They traced particles called y-tons containing certain information, such as air pollution factor through the scene to create a map. When the particles struck a surface, various motions could have occured, such as reflection, bounce, flow and settling in place. They used this motion, along with the parameters for the type of weathering, to drive the changes made to texture, surface properties, and geometry.

## 2.2 Heat Maps

Heat maps are useful for indicating areas of interest in an image, text document or website. Eye-tracking heat maps show what areas of an image catch the subject's eye, such as sharp shadows or bright highlights. They are based on a histogram, which essentially creates a height map. The height map is then translated into a color gradient. In our project, the heat map is used to indicate density and location of scratches on a surface.

Duchowski et al. [6] used the GPU to render eye-tracking heat map visualizations in real time, making signifigant improvements in the speed and quality of the rendered heat map image. In Figure 2.1, this process is shown in a heat map compiled from aggregate eye-tracking data.

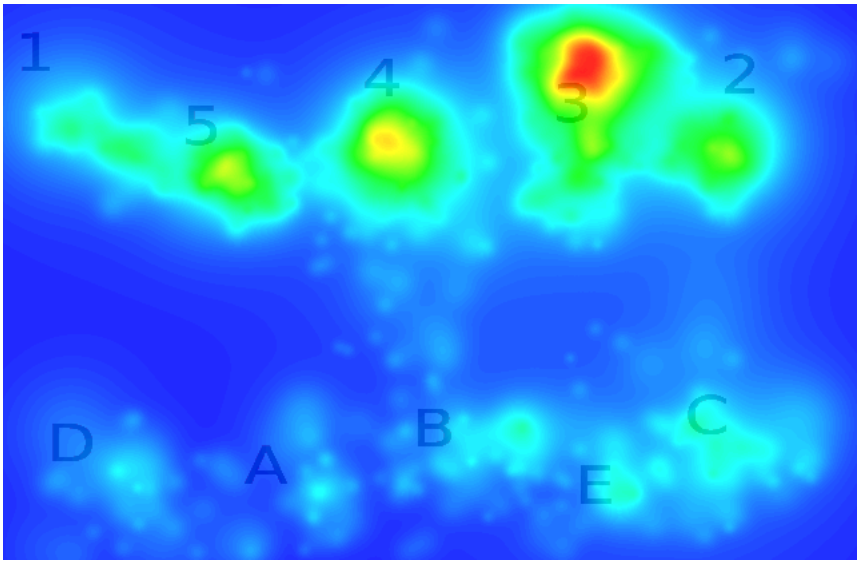



Figure 2.1: Heatmap of aggregate eye-tracking data rendered by the GPU.

## 2.3 Texture Generation

In another study, Bosch et al. [3] developed a physically-based model used to render realistic scratches. Although this project focused on the look of scratches rather than the placement, it is important to our research to understand the processes behind creating realistic scratches. They computed a different Bidirectional Reflectance Distrubution Function (BRDF) within the scratch determined by the scratch's microgeometry. A BRDF is a four-dimensional function that determines how light is reflected off of an opaque surface. Previously, they computed microgeometry of the scratch by a physically-based simulation employing parameters such as the type of scratching tool, the force behind the scratch, and the material of the object recieving the scratch [11].

Brooks and Dodgson [4] combined procedural textures with replicated image editing. Replicated image editing involves repeating editing operations globally over an image, so changes made to a pixel will affect all pixels similar to the one specified. This process used replicated painting, which alters the color of similar pixels; replicated cloning, which copys a part of one image to another; and replicated warping, which is contracting or expanding regions of a texture based on similarity to the selected pixel.

Eringis [7], of Blue Sky Studios, spoke about the completely procedural approach developed to texture surfaces in Blue Sky's computer graphics feature films. Instead of painting a texture map, layering and sculpting 3D noise created the texture images. Even though procedural texturing can be abstract and extremely technical, Blue Sky Studios streamlined the process for use by artists at any level of technical background.

# Chapter 3

# Research Design and Methods

## 3.1 Summary of Methods

The process of generating scratches based on an animation loop is composed of three steps. The first step uses a Python script integrated with Maya software to collect data points of the scratch locations. The second step uses a C++ program and the previous data points as input to generate a heatmap of the distribution pattern of scratches. The third step utilizes the heat map to blend scratches onto an existing texture map image. This process quickly computes scratch locations on a CGI model and also creates a fast, editable prototype.

## 3.2 Data Point Generation Script

A Python script uses the placement of geometry within an animation loop to determine locations of scratches on the surface of an object. This script is detailed in Figure 3.1. In this algorithm we check the distance between a vertex and the point closest to its position on the surface of another mesh against a threshold. This check determines whether a scratch would be made at that location on that frame of animation. Later, we will explain this algorithm in further detail.

Inside Maya, we select the object to receive the scratches, called the scratchee, then select the object to administer the scratches, called the scratcher, as shown in Figure 3.2. To select multiple objects in Maya, hold down the shift key, then click the desired objects. Then, the *ls* Maya command retrieves the names of the selected objects, which are placed into respective variables.

```
get number of vertexes on scratcher
For(frames in frame range of animation sequence){
   For(each vertex on scratcher){
      find vertex position
      place vertex position into list
   }
   For(each vertex on scratcher){
      find closest point to vertex on surface of scratchee
      place closest point into list
   }
   For(each vertex on scratcher){
      find distance between vertex and its closest point
      If(distance is greater than threshold){
          delete point from closest point list
      }
   }
   For(each closest point){
      find UV coordinate
      place UV coordinate into list
   }
}
print list of UV coordinates to file
```

Figure 3.1: Determine scratch locations based on animation loop.

Any namespace tacked onto the name of the objects is then removed to avoid confusion. Namespace in Maya is a string added to the front of an object name when it is referenced into a scene.

We inject randomness into the animation to better test the process by connecting a Maya noise node, usually used to generate texture, into an axis of rotation of a joint. For additional variation, the *frequency* and *implode* components of the Maya noise node are animated over time. Figure 3.3 shows the layout and set up of the noise node within Maya as well as the animation curve within the graph editor.

The script loops through the animation sequence frame by frame. At the top of the main loop, the script increments the current time. The Maya command *currentTime* takes a frame number as input and uses it to update the animation.

Next, the Maya command *polyEvaluate* is called on the scratcher. This function returns information about a given mesh. In this case, we use the "v" flag to identify the number of vertexes on the scratcher. This number allows us to loop through the vertexes to find their positions in world space using the Maya command *pointPosition*. The positions found by this command are appended to a list of vertexes.

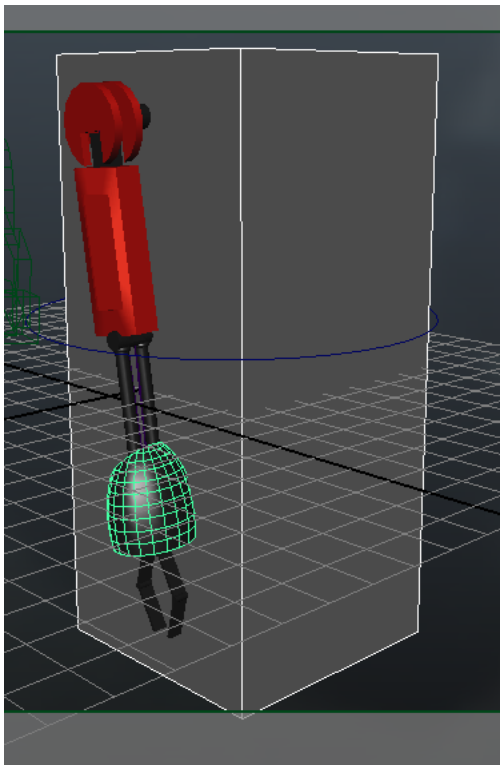
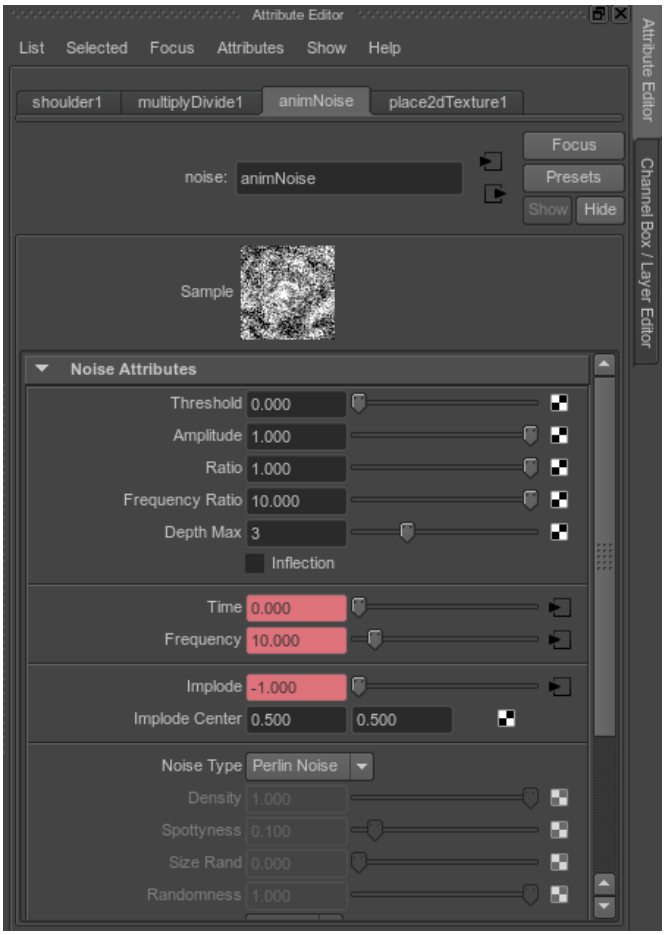


Figure 3.2: Selecting two objects in Maya.

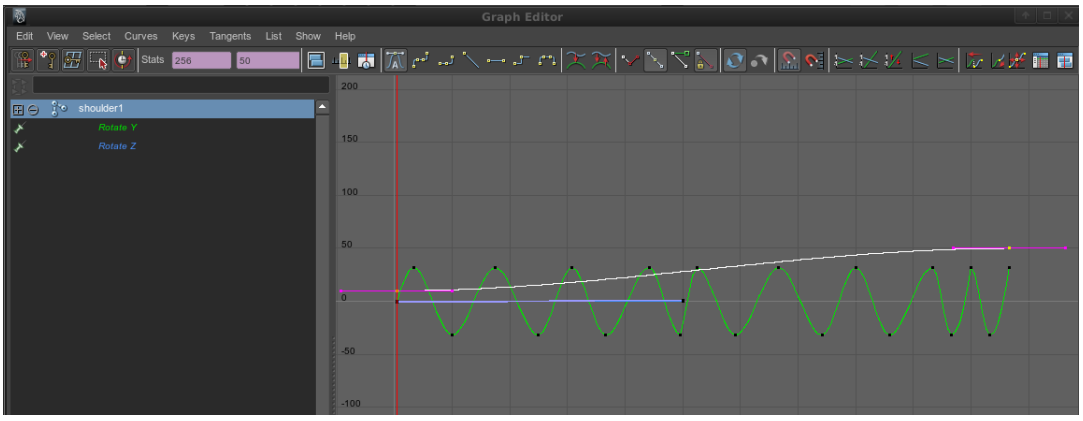
Then, the script loops through the list of vertex positions. Using the *nearestPointOnMesh* node, included in a default Maya plugin of the same name, we find the closest point to each vertex in world space on the scratchee mesh surface. The node must first be created within the script. Then, the world mesh of the scratchee is connected to the *inMesh* of the node. This procedure is necessary to ensure the *nearestPointOnMesh* node returns point values in world space instead of object space. The interface of the *nearestPointOnMesh* node is shown in Figure 3.4. The node consists of an input field called *In Position* and output fields called *Position*, *UV*, *Normal*, *Face Index* and *Vertex Index*. We input the vertex position into the In Position field, and the value of the closest point on the scratchee mesh is returned. We then place this value onto a list of closest points.

After we find the closest points on the surface of the scratchee to the vertexes of the scratcher, we loop through these points and calculate the distance $d = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2 + (Z_2 - Z_1)^2}$ between them.

Then, each point is checked against a threshold of world space units, which can be changed to produce various ranges of accuracy. The threshold is a small value, and is used to decide if a




(a) Noise node setup




(b) Animation curve

Figure 3.3: Randomness injected into animation using noise node

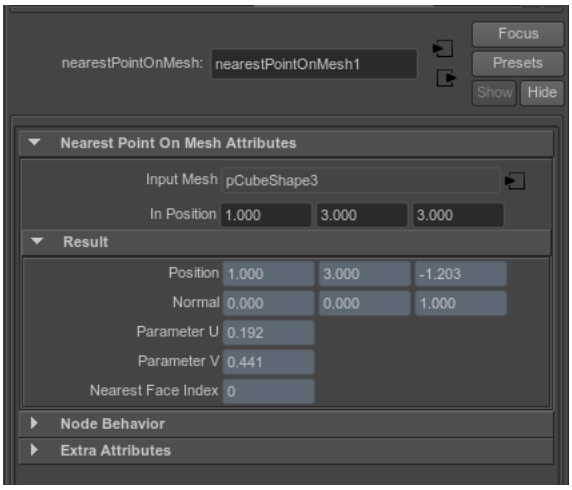



Figure 3.4: Interface of nearestPointOnMesh

scratch could realistically exist at that point on the surface of the scratchee. If a point lies outside of the threshold, then it is removed from the closest points list.

After this, the script loops through the remaining points on the closest points list. The *nearestPointOnMesh* node is utilized once more, this time to return the corresponding UV texture coordinate of each point. These UV coordinates are placed on a list, which is printed to a file once the animation sequence ends. This data will be used in the following part of the process.

## 3.3 2D Histogram and Heat Map Generation

We use the data points generated from the animation loop in Maya to create a heat map, which shows locations of scratches by being drawn on top of an existing texture map. Figure 3.5 compares the texture map with and without the heat map.

Each pixel in the image corresponds to a bin of the 2D histogram. A bin holds an integer value which corresponds to the number of data points that lie at a certain value or within a certain range of values. The program determines which pixel each data point corresponds to, then increments the value of that bin by one. We show how this is accomplished below in Figure 3.6. This process results in a grainy image, which causes the blended scratches to appear grainy as well. So, to remedy this effect, we run the data through a box filter to create a smoother resulting image.

We then create a heat map from the 2D histogram, as shown in Figure 3.7. First, we compute the range of the bin values. Then, each pixel is assigned a color, and we determine the

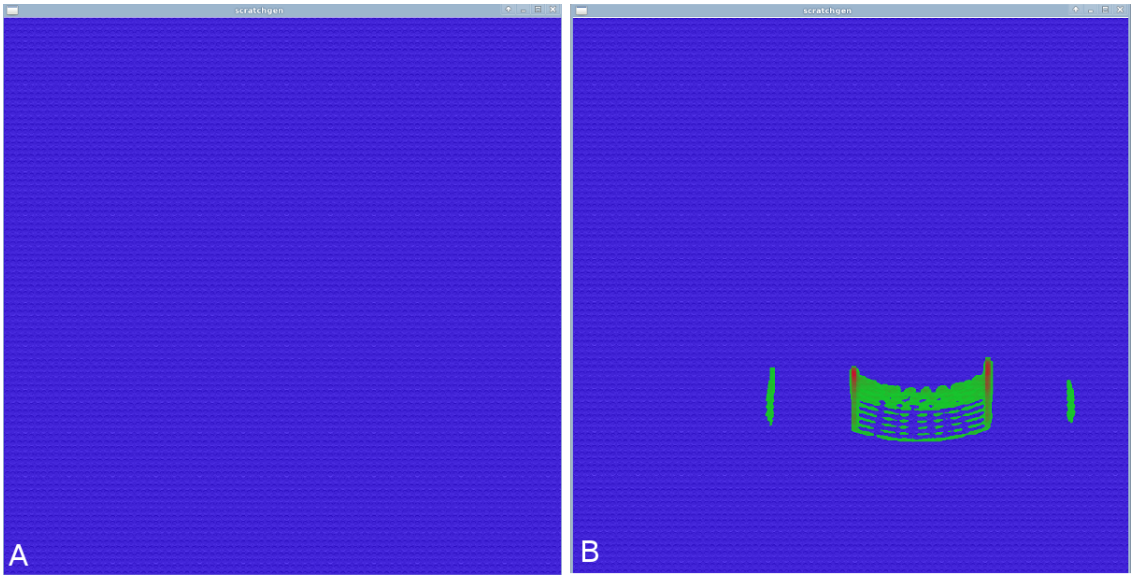



Figure 3.5: A) Texture map before image processing. For this example, the image is blue to make the red and green of the heat map more visible. B) Heatmap superimposed over the texture map.

```
For(each point in data point list){
   find pixel value of data point
   If(the value is within the bounds of the pixmap){
      increment the bin for that value by one
   }
}
```

Figure 3.6: Create a 2D histogram.

interpolation of the color between red and green by the location of the value within the range. Values closer to the maximum will be more red, while values closer to the minimum will be more green. Any bin that holds a value of zero is assigned no color and a transparent alpha value. The resulting heat map is shown previously in Figure 3.5.

## 3.4 Converting Heat Map to Scratches

Scratches are blended onto the texture based on the pattern of the heat map. The scratches are more visible in the red areas of the heat map and fade out toward the green areas of the heat map. This occurs because the red areas represent a greater density of data points.

An alpha value is assigned to each pixel of the scratched image based on the red value of the heat map at that pixel. Pixels with a high red value will be more opaque, while pixels with a

```
For(each bin in the histogram){
   If(the value is greater than max){
      max = value
   }
For(each pixel in image){
   red value = histogram value/max
   green value = 1.0 - red value
   blue values = 0.0
   assign rgb values to pixel
   place pixel into pixmap
}
```

Figure 3.7: Convert a 2D histogram into a heat map.

low red value will be more transparent. This value can be adjusted with a padding to control the desired level of opacity to create a subtle or dramatic effect. Then, the scratched image is layered over the original image using the over operation $C_o = C_a\alpha_a + C_b\alpha_b\,(1 - \alpha_a)$ [12]. $C_o$ is the output color channel, $C_a$ is one input color channel and $C_b$ is another input color channel. $\alpha_a$ and $\alpha_b$ are the transparency channels of the inputs. Figure 3.8 illustrates how this process works in code.

```
For(each pixel){
   value = red + padding
   If(value > than max value for pixel){
      value = max
   }
   alpha = value
   If(red and green = 0){
      alpha = 0
   }
}
```

Figure 3.8: Blend scratches into an image.

We show the resulting image in Figure 3.9. Here, the red and green areas of the heat map are replaced with scratches. From this starting point, the artist can edit the image as they see fit to achieve a final look. Also, we can create a bump (normal) map from the blended image, again seen in Figure 3.9. To create a bump map, we bring the image into an editor, desaturate it, then adjust the contrast until a desired result is achieved. A bump map creates a realistic rendered image because the scratches appear to be actually gouged out of the surface of the object. In a bump map, dark areas appear to recede into the surface while light areas appear to pop out of the surface.

However, the actual geometry remains constant in these areas because only the surface

shading manipulated. Note, that if a bump map was already applied to the geometry, this process will not react as if the geometry has those bumps and dips. Geometry displacements must be applied before the process begins. We can also create normal maps and displacement maps from the result.

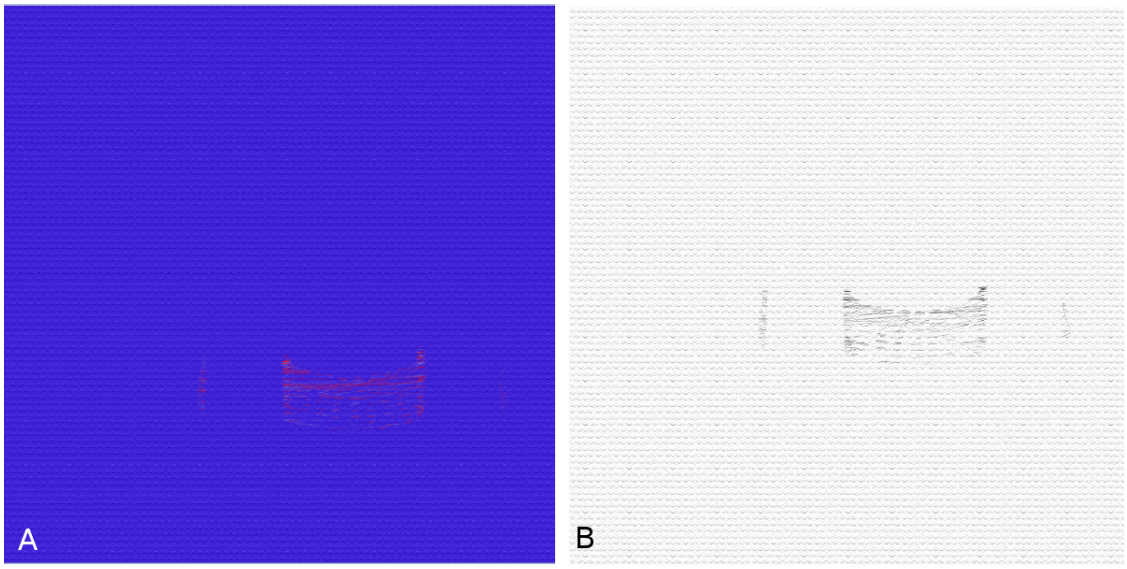



Figure 3.9: A) Scratched blended into the texture map. B) Bump map generated from the texture map.

We return to Maya to apply the resulting scratched texture map to the object. From there, we proceed to create a rendered image of the object, which is now scratched based on its animation loop. A shader node is created inside Hypershade, and the material qualities, such as shininess and reflectivity, are set. Then, the scratched texture map and the scratched bump map are applied to this shader node. Figure 3.10 shows an example of the result on the object shown throughout the description of the process.

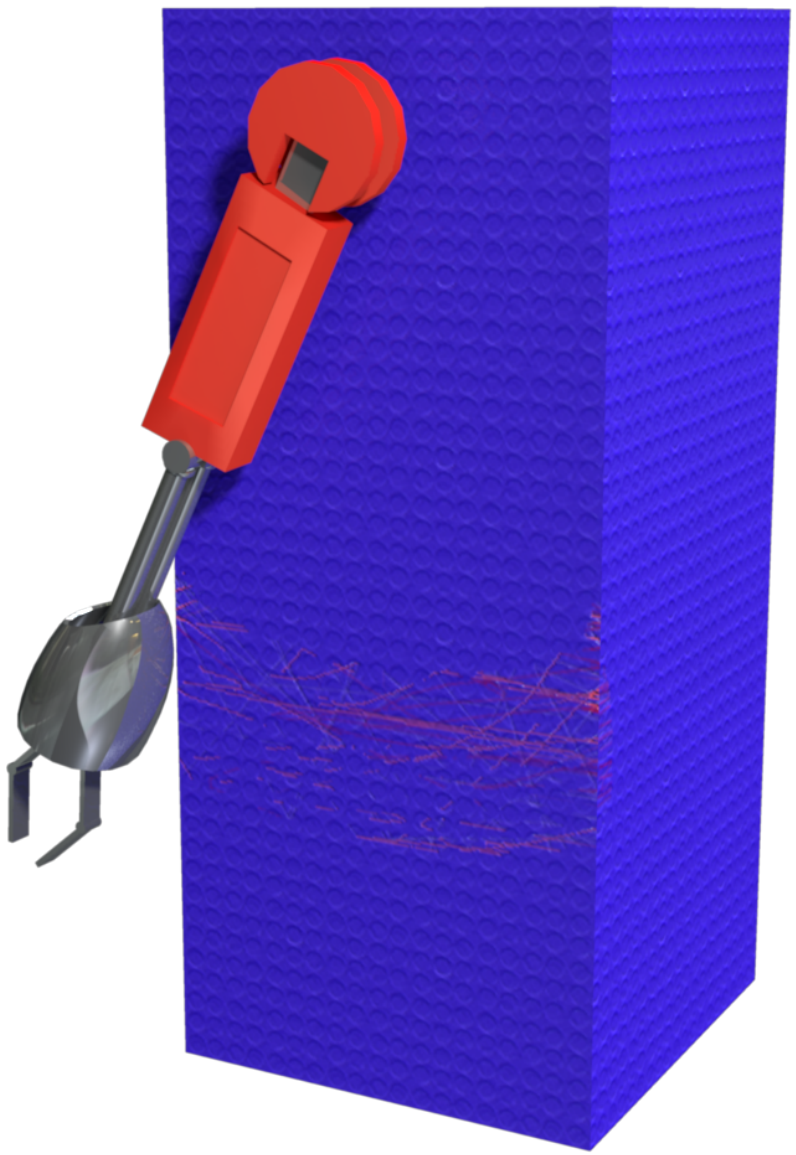


Figure 3.10: Final result of the scratch generation process rendered from Maya.

# Chapter 4

# Results

## 4.1 Analysis

In out test cases the method produced the expected results. Importantly, the scratch locations determined by the process appear to be physically correct. They follow the motion of the scratcher object's animation, as seen in Figure 4.1. The object moves in an arc like a pendulum, and the map of the scratches is in the shape of a circular arc. Also, due to the randomness injected into the animation, the motion of the object becomes erratic. It simulates skipping along the surface of the scratchee. These skips appear in the map as voids in the scratch pattern, which is further indication of the ability of the process to detect the subtleties of motion.

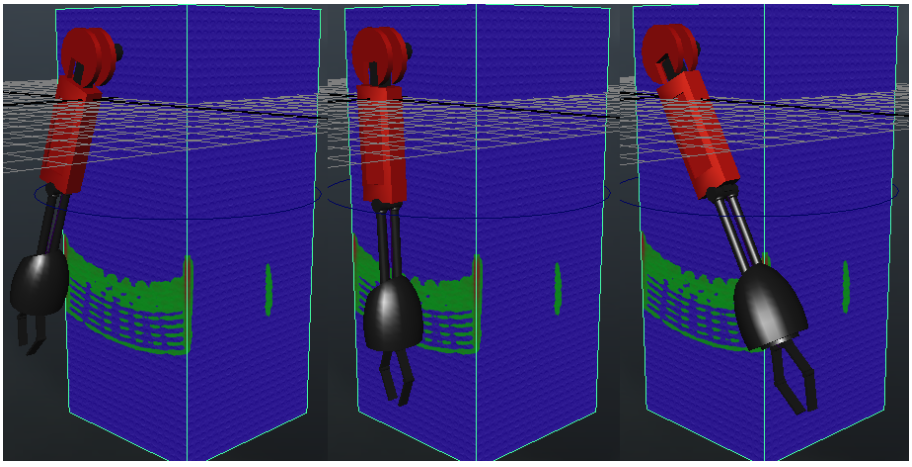


Figure 4.1: Results at different stages of the animation loop. The scratches follow the motion of the animation.

The time frame of the process is quick from start to finish. Data points can be obtained from the python script within Maya in a reasonable amount of time. The script ran and completed on the example model with a vertex count of 436 and frame count of 256 in less than a minute. The image processing code also ran under a minute. This was true even when the program was tested with images up to a 4k (4096x4096) resolution.

When we manipulated the threshold, or the maximum distance between objects where a point is considered a scratch, it produced predictable results. At lower values, the results were less dense, but more precise. At greater values, the results were more dense, but less precise. Figure 4.2 illustrates the effect of changing the threshold gradually from a very low value producing virtually no points to a high threshold producing many points. To achieve a high number of data points with a low threshold, the animation loop must be run for an extended amount of time.

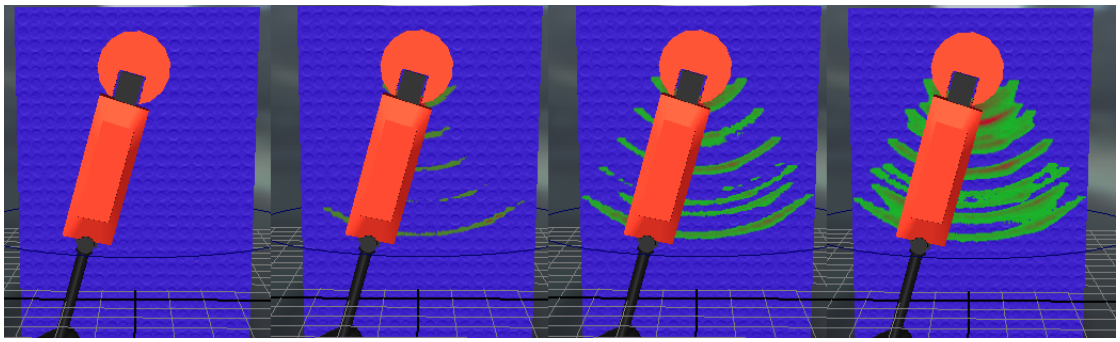


Figure 4.2: Results at different thresholds. Values from left to right: 0.5, 1.0, 1.5, 2.0. 226 samples per frame for 256 frames.

Also, the number of vertexes on the scratcher object caused variation in the result. As seen in Figure 4.3, we tested the process at a threshold of 1.5 using a rectangular prism with only 8 vertexes. In this case, the scratches only appeared on the edges where the vertexes existed. Then we subdivided the geometry such that it contained 226 vertexes. The subdivision caused the scratches to appear to be more spread out throughout the length of the object. Thus, geometry should be subdivided and smoothed before beginning the process.

The number of frames the process runs for also has a variable effect on the data collected. Running the process for an increased number of frames results in increased density and spread of the data points. This is true for all thresholds, even though a threshold of 0.5 did not produce data able to create a visible heat map within the tested time range. The results did not plateau within the tested time range. The number of data points collected increased at each interval.

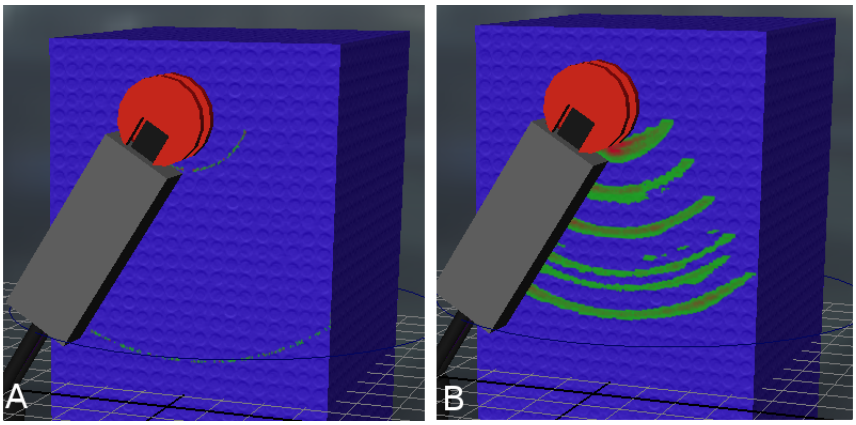



Figure 4.3: A) Result with object containing 8 vertexes. B) Result with object containing 226 vertexes.

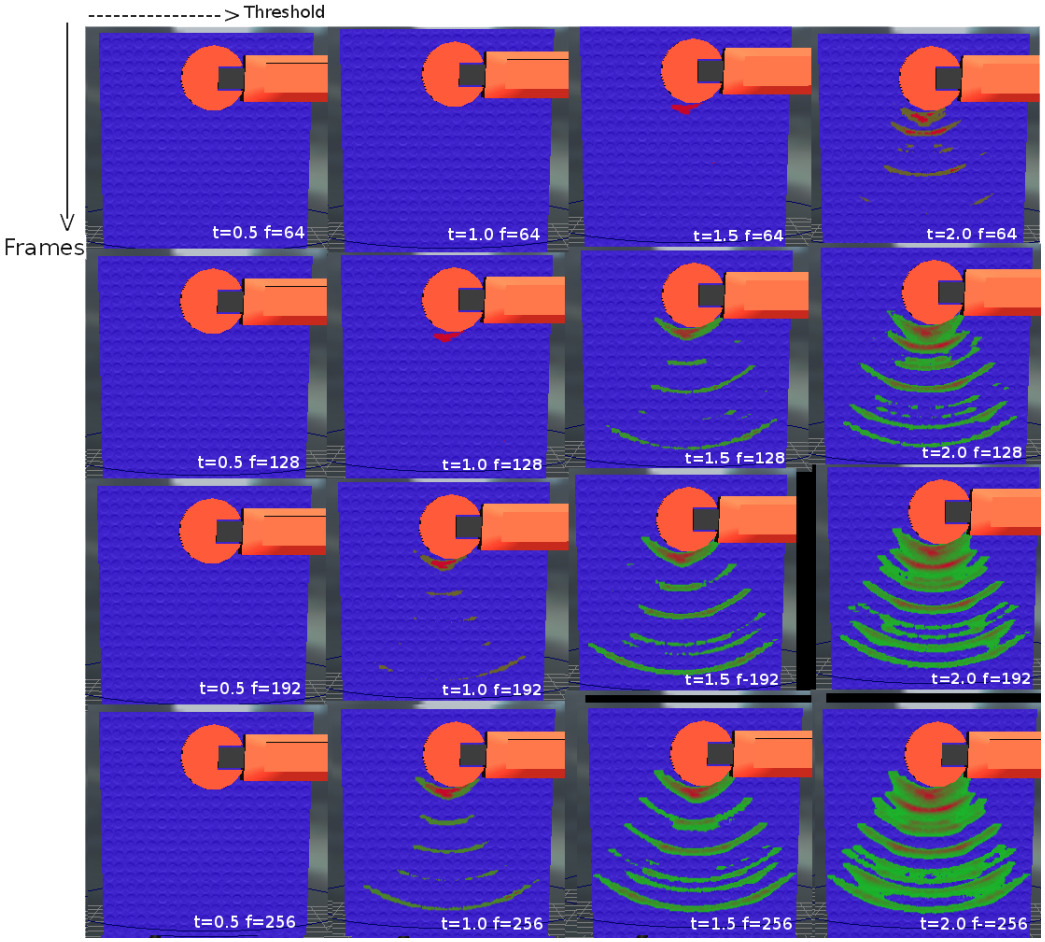



Figure 4.4: Table of results at different thresholds and times. t = threshold value f = frames.

The process was tested at different threshold values and frame numbers. We tested threshold values of 0.5, 1.0, 1.5, and 2.0 at frame intervals of 64, 128, 192 and 256. Figure 4.4 shows these results in a table format, with thresholds along the x-axis and frame numbers along the y-axis.

## 4.2 Evaluation of Methods

To evaluate our method, we constructed a comparison between the results of our method and those produced by an artist. The robot model, environment model, rig and texture maps are from a previously completed animation called Roboasis. The scratches on the texture maps were drawn by hand by artists on the Roboasis team. Then, we applied our process to these texture maps. We compared the results of the process to the original texture map created by the artist.

We animated the robot model based on the motion it exhibited in the animated short. Using this animation, we created a heat map of the scratches caused by the arm on the body. Then, we placed the heat map over the original texture. We display the original texture and the same texture with the superimposed heat map side by side in Figure 4.5. Our process placed scratches in the same locations as the artist. However, it also placed scratches in locations that the artist left unmarked. The shape of the scratches created by our process also differs. Our scratches follow an organic curve while the ones drawn by the artist are laid out in a straight line.

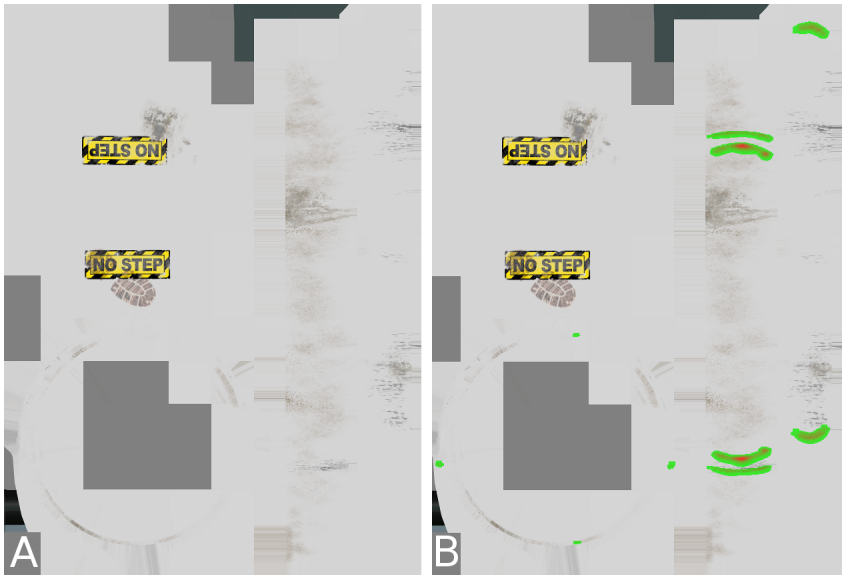



Figure 4.5: Heat map comparison. A) was painted by an artist. B) shows the generated heat map.

In this example, we used the heat map as a guide to draw additional scratches onto the texture map by hand. This technique demonstrates the versatility of our scratch location generation process because once the heat map is generated the artist can then take multiple pathways to the final image. The rendered images appear in Figure 4.6, placed side by side for comparison. The red box in each image highlights the areas of the image affected by the process.

The effect of the process is both subtle and believable. The new scratches are realistic because they follow the natural movement of the robot. Also, their shape is organic, which lends additional variation and interest to the surface. However, the process did indicate scratches to appear on the other side of the considered object because the object's thin nature caused its other side to be triggered within the set threshold. In this case, the artist could choose to lower the threshold, or the could use their best judgment when editing the scratches, since the automated nature of the process is not foolproof.

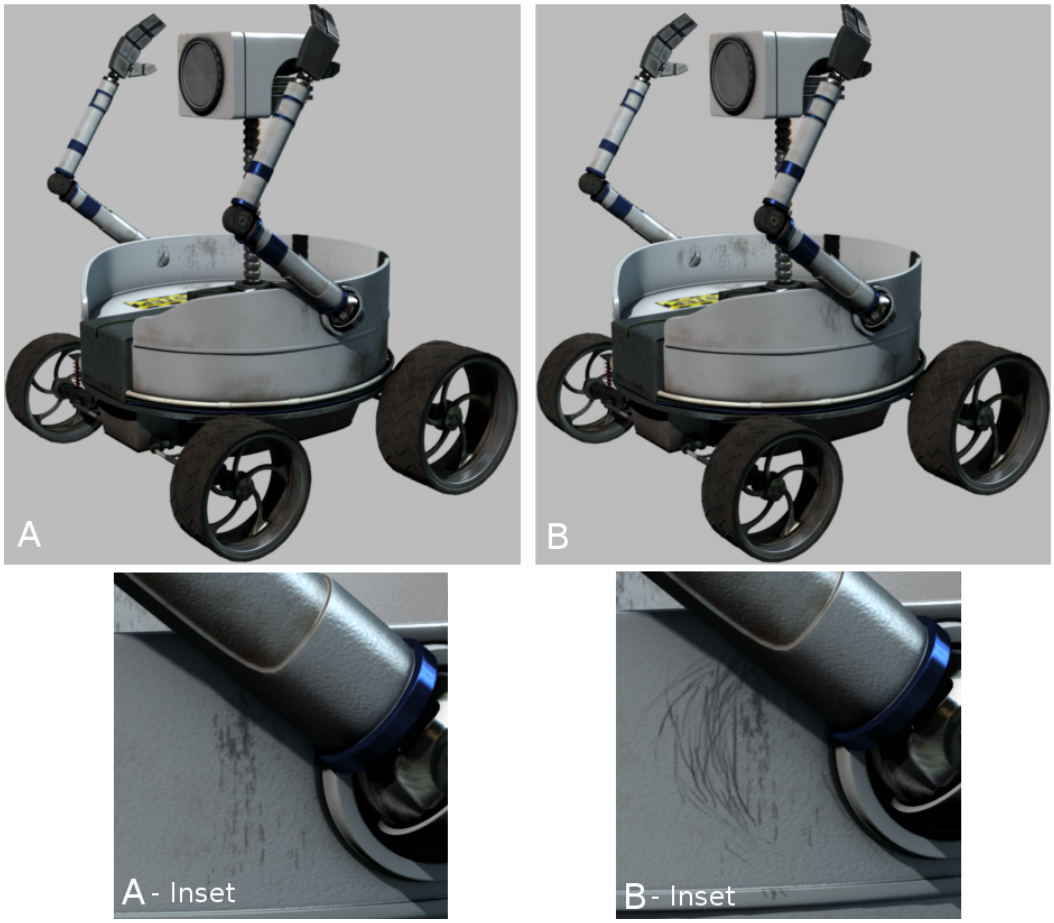



Figure 4.6: Comparison of rendered images. A) original artist's texture and B) texture with the process applied. Below: close-up of detail.

# Chapter 5

# Conclusion

## 5.1 Conclusions of Results

The results are an indicator of practical value of our process in a production environment. The process turns the typical order of the production pipeline on its head because it assumes some animation has been completed when surfacing begins. Usually, surfacing is nearly finished before animation begins. However, speed of the process could be useful for the production of quick proof of concept images. Also, the process has a quick turnaround, so the user will be able to go through several revisions of their texture map a day using this tool. This may lead to improved surfacing quality due to the number of revisions allowed by this tool.

The process placed scratches in areas that were left blank by the artist. Also, the scratch pattern differed from the scratch pattern in the artist's texture map. Even though the process is a hybrid of simulation and statistics, its use can significantly increase the realism and accuracy of scratch placement. The removal of the guesswork involved in placing scratches would be especially evident in work by an artist who lacks a background in mechanics.

The animation sequence length has an effect on the results. As the length of the sequence increases, the total number of data points increases as well. This result leads to the conclusion that the process will yield the most accurate and complete scratch patterns when run over a large number of frames using a lower threshold.

## 5.2 Practical Applications

The practical applications of the procedural scratch generation process span from production, to industrial and to medical applications. The project in its current state can be extended in many different directions.

In production, the process can be used to expedite a proof of concept of scratches on a robotic character or a mechanical model. It defines the locations and pattern of scratches quickly and automatically. Then, the artist has a few options. They could draw scratches over the generated heat map, or use the blend function to produce generic scratches. Then they could edit those scratches until a desired look is achieved.

In manufacturing, industrial design and mechanical engineering, the process could be used to predict areas of concentrated wear on machines before they are put into production. An engineer could model a design in a 3-D graphics package, then animate the model cycling through its intended motion. Then they could run our script to determine which areas may have the most possible collisions in the physical world, and in turn the most wear due to scraping. Adjustments driven by these results could then be made to minimize the impact of wear.

The same principle could be applied to prosthetic joints. Such devices go under continued repetitive motion, which leads to inevitable wear. Locating the regions of concentrated wear can lead to its reduction. This can lead to improved motion and prolonged joint life, so patients do not have to return to surgery for replacement.

## 5.3 Recommendations for Further Research

The current state of the process allows for extensions, including ones for improved functionality within Maya as well as for image processing. Further research could be made into any of these categories.

Work could be done to implement the script into a fully functional plug-in for Maya. Specifically, we want to add functionality running calculations in the background. This would allow the artist to work on other tasks while data collection is in process, especially in cases where the process is incrementing through many frames of animation. They could access the script by clicking on a button as part of the shelf instead of having to use the script editor. The ability to select a list of objects to scratch and then run the process on all of them automatically would also be a

helpful improvement. In addition, automatic generation of specular and normal maps would further streamline the process.

More research could be conducted on adding functionality to the code. This functionality would calculate the directionality of the scratches in addition to their location and patterning, perhaps utilizing a vector pass to obtain that data. This would improve the realism of the final result. Also, more work could be done to further improvements in accuracy. We suggest this because at times the process will detect scratches on the far side of a thin object.

Finally, a useful addition to the project could be functionality to adjust the look of the scratches. This adjustment would be based on the material of the scratcher and the material of the object receiving the scratches. We would modify the scratches procedurally using automation BRDF adjustment based on material attributes such as hardness, type and shininess. This addition would add another level of automation to the process, which in turn allows the artist more time to work on painting finer details.

# Bibliography


[1] D. Blythe and T. McReynolds. Lighting and shading techniques for interactive applications. In *SIGGRAPH '99 Course 12*, August 1999.

[2] C. Bosch, P. Laffont, H. Rushmeier, Holly E., J. Dorsey, and G. Drettakis. Image-guided weathering: A new approach applied to flow phenomena. *ACM Trans. Graph.*, 30(3):20, 2011.

[3] C. Bosch, X. Pueyo, S. Mérillou, and D. Ghazanfarpour. A physically-based model for rendering realistic scratches. *Computer Graphics Forum*, 23(3):361–370, 2004.

[4] S. Brooks and N. A. Dodgson. Integrating procedural textures with replicated image editing. In *Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*, GRAPHITE '05, pages 277–280, New York, NY, USA, 2005. ACM.

[5] D. Derakhshani. *Introducing Maya 2009*. Wiley Publishing, Inc., 2009.

[6] A. Duchowski, T. Price, M. M., M. Meyer, and P. Orero. Aggregate gaze visualization with real-time heatmaps, in eye tracking research & applications. In *ETRA '12 Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 13–20. ACM, March 2012.

[7] Michael Eringis. A completely procedural approach to materials. In *SIGGRAPH '07 Sketches*, August 2007.

[8] M. Friendly. The history of the cluster heat map. *The American Statistician*, 2009.

[9] J. Gu, C. Tu, R. Ramamoorthi, P. Belhumeur W. and Matusik, and S. Nayar. Time-varying surface appearance: Acquisition, modeling and rendering. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 762–771, New York, NY, USA, 2006. ACM.

[10] S. Mérillou and D. Ghazanfarpour. A survey of aging and weathering phenomena in computer graphics. *Computers & Graphics*, 32(2):159 – 174, 2008.

[11] Fred E. Nicodemus. Directional reflectance and emissivity of an opaque surface. *Applied Optics*, 4(7):767–775, July 1965.

[12] Thomas Porter and Tom Duff. Compositing digital images. *SIGGRAPH Comput. Graph.*, 18(3):253–259, January 1984.

[13] Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins. Data-driven and demand-driven computer architecture. *ACM Comput. Surv.*, 14(1):93–143, March 1982.