Final Project Report

Becky Hu (bh456)
Ruyu Yan (ry233)
Olivia Li (yl2795)
Blaire Yu (yy735)
Andrew Li (zl469)

1 Introduction

Emerging over 1000 years ago, Chinese ink painting has had a long-lasting impact on the artistic practice in East Asia. Artists typically apply varying concentrations of black ink with brushes to paint lines and shading on Xuan paper, a soft and fine-textured material that absorbs ink quickly. Such media creates visually appealing effects that are valuable in digital art, animation, and video game productions, but it is difficult to simulate with simple filtering. Although the freehand nature of Chinese ink painting deviates from the physics-based 3D rendering techniques that we have been learning in class, we found it challenging but interesting to apply 2D freehand stylization to 3D scenes in real-time.

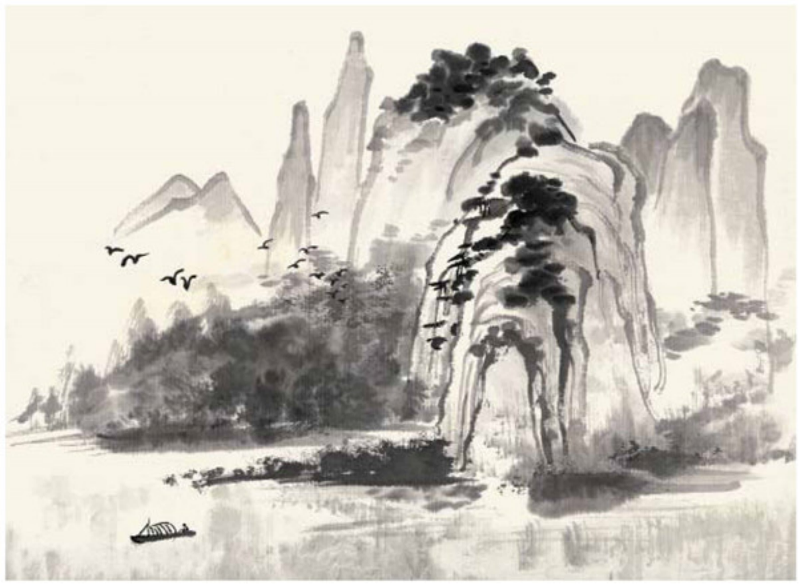


Figure 1: An example of Chinese ink painting of the mountain and water theme.

In this project, we aim to develop a deferred shading pipeline that renders a 3D scene in the style of colorful 2D Chinese ink painting. To simulate the freehand painting effect, we will render the silhouette and the interior with stroke detection in different rendering passes. To better render the classic theme of mountain and water (*shan shui*) in Chinese painting, we rendered water reflection with wave animation. Building on a stylized scene, we furthered implement repainting effects in response to camera motion and calligraphy animations according to the user input. We applied those techniques to our meticulously designed 3D scenes.

2 Rendering Pipeline

In this section we will explain how our Non-Photorealistic Rendering pipeline works.

2.1 Overview

Our approach for simulating the Chinese ink painting style ended up deviating from the reference materials that we cited in our proposal. We kept the two-step rendering for the silhouette and the interior, while we developed a general pipeline for painting strokes on a paper. Our pipeline first performs a simple shading with the geometry and material data. Then, we do a local stroke detection on patches of pixels, and apply non-maximum suppression to remove redundant strokes. Finally, we paint the strokes according to the computed data. This way, we can better approximate the brush texture and the ink diffusing effects on Chinese ink painting. We run two passes of this algorithm (shown in Figure 2) for the silhouette and the interior separately, and blend the result together.

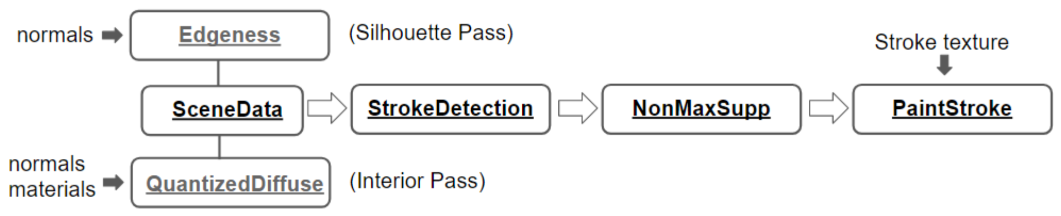


Figure 2: Diagram of the stroke simulation pipeline.

2.2 Stroke Detection

In order to render strokes to mimic Chinese ink painting styles, we need to determine the places where we place the strokes. We define a data structure called stroke stamp which includes the following information:

- **Stroke radius**: the radius of the stroke stamp in the unit of the screen-size texture.
- **Stroke orientation**: the orientation that the positive x-axis of the stroke stamp points to.
- **Stroke alpha**: the transparency value applying to the entire stroke stamp.
- **Stroke texture**: the brush texture used for each stroke stamp (see Figure 3 as an example).

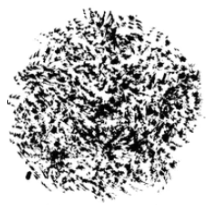


Figure 3: Stroke texture used for the silhouette.

By overlapping stroke stamps in desired patterns, we can simulate different brush painting effects. For the silhouette, we want to apply thin and smooth strokes along the edge of meshes with decreasing stroke radius at the tip. For the interior, we want to apply large strokes with less variation in intensity. To optimize towards different goals, we came up with two separate algorithms for stroke detection.

2.2.1 Silhouette

The silhouette is defined by pixels whose normal is perpendicular to the view direction. Therefore, we can easily detect the silhouette by checking whether the dot product between the normal and the view direction is zero. We can then find all the silhouette pixels and color them black.

However, this clear-cut rule does not consider the neighboring pixels and may lead to noisy results. More importantly, this strict threshold - only pixels with normals perpendicular to view directions are colored black - does not fully simulate the ink bleeding effects in the Chinese ink painting. It also creates a constant stroke width which does not simulate the dynamics and randomness in art creations.

We then came up with a two-pass system to resolve all the issues listed above. The idea is to create a continuous function that takes in a patch of neighboring pixels to compute the **stroke radius** and **stroke alpha**. In the first pass, we compute the absolute value of the dot product between the normal direction and the view direction at every pixel. We call this value "edgeness" (which is actually "non-edgeness" because the larger the edgeness, the less perpendicular its normal is to the view direction). We write the edgeness value onto a frame buffer. In the second pass, for each pixel, we average edgeness over a patch of their neighboring pixels. We then compute the stroke data so that the larger the average edgeness, the smaller the alpha value and stroke radius.

Lastly, in the second pass, we also compute the **stroke orientation** from the gradient orientation of the local patch. The stroke stamps along one stroke will be well-aligned if they are all perpendicular to the edge. For computational efficiency, we approximate the 2D gradient orientation by summing the gradient along the x and the y direction. To avoid having a thick stroke with overlapping stroke stamps, we look up the nearby pixels along the gradient orientation and only keep the stroke stamp at the location with edgness smaller than the neighbors. Finally, along with the stroke radius and the stroke alpha, we output a four-channel stroke stamp map which we will use for drawing strokes in the next step.

2.2.2 Interior

The rule for detecting the interior strokes is less clear than the silhouette. We initially hoped to achieve stroke-like textures by simply quantizing the interior colors under the diffuse model. We do so by mapping the RGB values to the LAB space, quantizing the L value, and then mapping back to the RGB space.

However, this has led to issues similar to directly using dot products to decide the silhouette above. Namely, we ended up with blocks of colors with minimal transitions between them. Even though we tried to smoothen the drastic changes by placing a box filter on regions with a high standard deviation, there were still hardly any variations within each color block which hardly resembled the vibrant ink painting strokes.

As a result, similar to computing the silhouette, we came up with a continuous function to compute the stroke radius, stroke alpha value, and stroke direction. Here we used a three-pass program. In the first pass, we quantized the RGB values as described above. In the second pass, we look up the quantized RGB values for a patch of neighboring pixels to compute the average color on the left, right, top, and bottom of each pixel. We then compute the gradient direction the same way we computed that for the silhouette. We make it so that the pixels with larger gradient magnitude have a smaller stroke alpha value so that places with larger color change can have a more gentle stroke and therefore a smoother transition. Additionally, pixels with larger depth will have a smaller alpha value. The stroke radius is proportional to the stroke alpha value.

In the third pass, we performed a non-maximum suppression so that we only keep pixels that have the largest stroke radius in their neighborhood. That way we don't create redundant strokes.

2.3 Stroke Painting

With stroke stamp data for both the silhouette and the interior, we developed a one-pass shader for painting the scene with simulated strokes. We approximate brush strokes by stamping transformed stroke textures on a plane and interpolating between them.

The inputs of the painting shaders are:

- **Search-area radius**: To identify the stroke stamps that cover a pixel, we need to look up stroke stamp centers around a neighborhood. The ideal search-area radius should be equal to the largest possible stroke stamp radius, while a large search region can be a performance bottleneck of our pipeline. In our implementation, we used roughly half of the largest stroke stamp radius as the search-area radius.
- **Stroke stamp map**: The four channels of the stroke stamp map encode the radius, orientation, and alpha value of each stroke stamp. The size of the stroke stamp map also matches the screen size, so a pixel in the stroke stamp map with a radius value greater than 0 indicates that we detected a stroke stamp at that location.
- **Quantized color texture**: (only for interior): In the previous step, we shaded the scene with simple diffuse shading under a directional light, and quantized the color into several levels to simulate layers of ink.

To compute the color of a pixel, we first search for stroke stamps within a neighborhood (demonstrated in Figure 4). For each nearby stroke stamp center, we transform our pixel coordinate to the local coordinate system of the stroke stamp according to its radius and orientation. Then, we look up the intensity value in the stroke texture at the transformed coordinate and multiply it by the stroke alpha. The resulting intensity will be used as the alpha channel of the output color computed for this stamp. For the silhouette, we simply use black color; for the interior, we sample the quantized color texture at the current pixel location. We repeat the operation with all the nearby pixels and accumulate the color texture value, so we can simulate multiple stamps applying on a single pixel. Figure 5 shows examples highlighting our stroke painting algorithm applied on the silhouette and the interior.

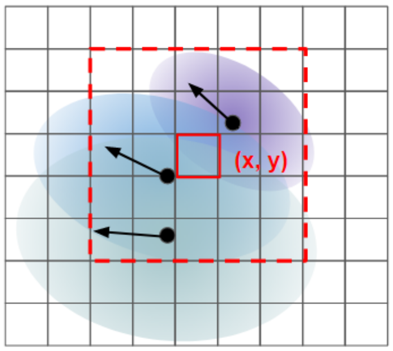


Figure 4: We look up stroke stamps in the neighborhood and blend the color sampled from the stroke texture with transformed texture coordinates.

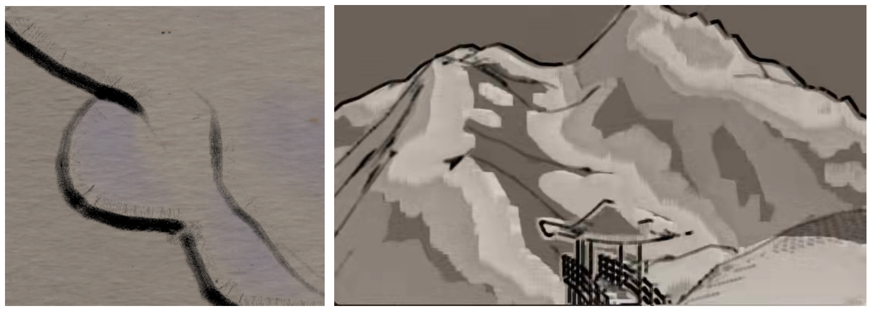


Figure 5: Examples of stroke painting highlighting the silhouette (left) and the interior (right).

3 Extensions

Building upon the aforementioned rendering pipeline, we added a few extra features and functionalities to our work.

3.1 Repainting

To add some extra fanciness, we implemented an animation for drawing the silhouette, which we refer to as *repainting*. Whenever the user releases the camera control, they will see the silhouette appearing from the bottom to the top of the screen.

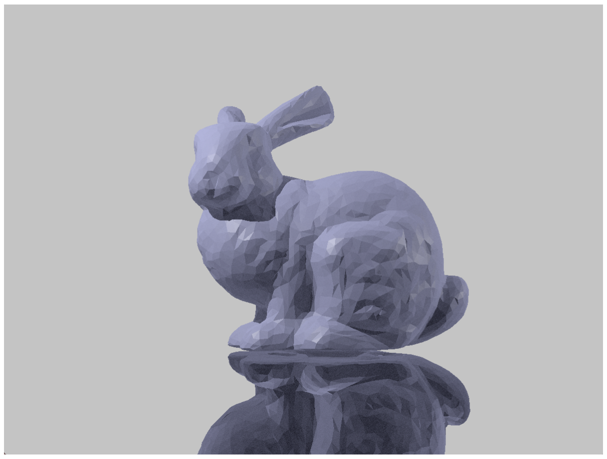
3.2 Water Reflection and Waves

In our project, we implemented some relatively simple water reflection simulation, as elements such as lakes and rivers usually make up an important part of traditional Chinese ink landscape paintings.

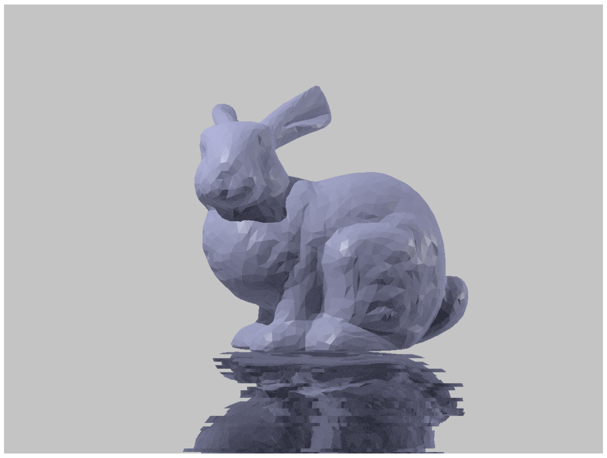
Water reflection can largely be seen as a type of mirror reflection, except that the presence of water waves might distort the reflected images of some objects. Therefore, we decided to simulate water reflection effects using the steps as follows:

- We shift the positions of all the objects in the scene, so that the bottom of all the objects coincide with the plane $y = 0$. For scenes that contains a cube that serves as the floor, we take out the floor from the scene to achieve the best visual effect. This way, the bunny scene that we normally use for testing contains a single bunny that sits on the plane $y = 0$.

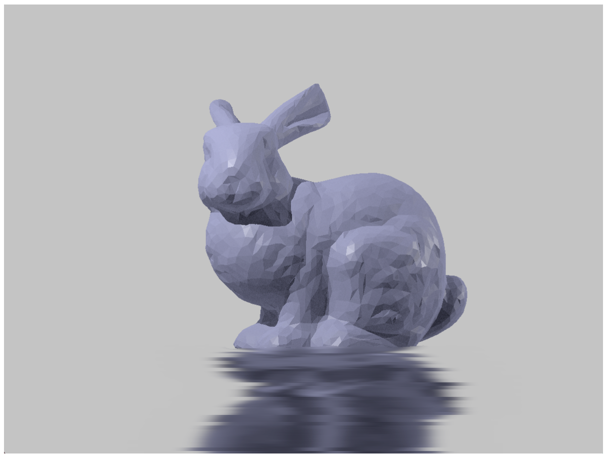
- We create a new mesh corresponding to each original mesh in the scene. We do this by adding a new vertex shader that takes the original mesh configurations as input and flips y components of all the vertex positions and normals in the mesh. This way, the shader program draws a counterpart for each mesh in the scene, which is symmetrical to the original mesh about the plane $y = 0$.



- We modify the appearance of meshes that represent the reflected images; these modifications are performed directly in the screen space. To make sure we only distort the reflected images of objects, we added a color channel in our `GBuffer`, which serves as an indicator that tells which fragments in the rendered image correspond to the reflected meshes.
- We slice the reflected images of objects into horizontal stripes in screen space, and apply some random horizontal shift to each stripe. This process is implemented in our fragment shader `shift.fs`, which takes the rendered image that contains pairs of meshes corresponding to all objects in the scene as input, and outputs an image where all the reflected meshes are distorted by horizontal shifts. The shifts can be done by assigning the pixel color of one fragment to another fragment with a slightly different geometry texture coordinate–for horizontal shifts, we can assign the pixel color at (x_0, y_0) to the fragment at $(x_0 + \Delta x, y_0)$.



- Our last step in simulating the appearance of water waves is performing some horizontal blurring on the disturbed reflected meshes. Our previous fragment shader `shift.fs` also outputs a mask that indicates which part of the processes image needs horizontal blurring, so that we do not accidentally blur the section of the image where the original meshes are drawn. This way, we can use the shader `blur.fs` to perform horizontal blurring on each shifted horizontal stripes from the previous step.



Moreover, we found that we can easily animate the water waves by generating multiple sets of random shifts, and interpolate between these different sets of shifts based on the timing. We have some demonstrations of this effect in our video.

3.3 Calligraphy

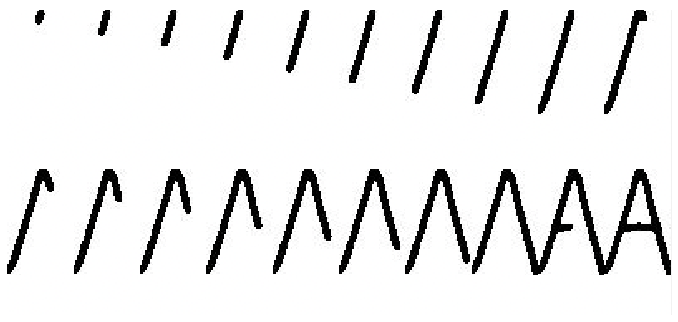
Another feature that we added into our scenes is calligraphy. Specifically, we added English letters and words into our scenes and managed to have these words painted with strokes appearing in order.

To achieve this effect without slowing down our program, we precomputed a stack of 2D textures corresponding to each English letter. The pipeline for generating a stack of 2D textures is as follows:

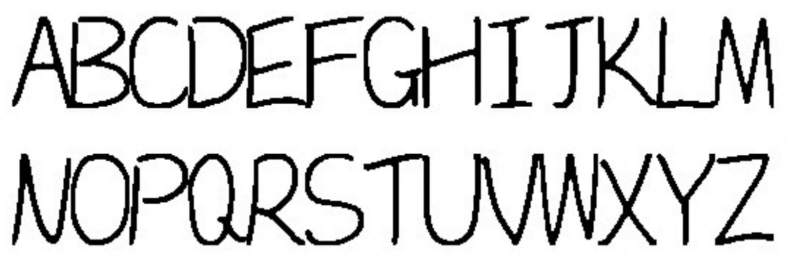
- Our input corresponding to the stroke shapes comes from some individual's handwriting–we simply have them write an letter on a piece of paper and scan the handwriting.

u u v v v L M N
v w x y y z O P Q
y y y R S T
A B B C C U U V
C D E F G W X Y
H I J J K Z

- We create a stack of *procedural subimages* based on the original picture of the handwriting. We load the picture into some image editor, and erase the original image step by step, saving one copy of each intermediate image. The sequence of images, when ordered correctly, demonstrate the procedure during which the letter is written.
- We process the sequence of images in `MATLAB`, using the `Image Processing Toolbox`. Important steps include Gaussian filtering to adjust the width of the strokes, creating binary masks of the strokes, and downsampling the images into the size of 40×96.



- We save a stack of 40 × 96 images for each letter in English, and load them into our rendering program as 2D textures. These 2D textures are selected, loaded, and bound to appropriate shader programs at appropriate time points when we intend to paint letters in our scene.



Acknowledgements

We would like to sincerely appreciate all the help we received from Prof. Marschner who has shed great lights on our ideation and implementation. We would like to thank Yuhang Wu (yw974) for providing his handwriting samples as inputs to our calligraphy feature.