

CSCI 580 - Final Project

Loose and Sketchy Animation

Team "Hodor Hodor!!!"

Gulsheen Ahuja

Lanchen Zheng

Nitin Havanoor

Vipul Garg

Yuwen Lian

# **Introduction**

# NPR is the area of computer graphics that enables creativity and expression. Its counterpart, the photorealistic rendering, as its name suggests, is the exact replication of the real scene. This type of rendering would leave very less room for the creator to improvise on his creativity on describing his work. NPR is the technique, which on the other hand, sets the creator free, if you can say. The creator is free to describe his characters as he feels appropriate and in a way that will exemplify his work. Our idea here is to develop a filter, which would allow the user to set parameters and render "loose" sketches for his 3D scenes.

The "loose and sketchy" filter automatically draws the visible silhouette edges of a 3-D model using image processing and a stochastic, physically-based particle system and converts a 3-D animation into a sequence of gestural sketches for creating finished animations in a non-photorealistic style. It takes as input the depth map of a 3D modeled scene and creates an image that looks like a sketchy line drawing of the silhouette edges of the scene. Compared to other computer generated drawing styles, the sketchy, gestural quality of the image created using this technique is considered by some to be the most expressive.

As part of our project, we have implemented the full Loose and Sketchy Rendering Algorithm. As part of our tool, we have also given the user an option to change the lights, rotate the object, increase or decrease several parameters to give the user flexibility in designing the scene.

# **Implementation**

We have used the application framework which was provided to us via the Homework. We have stripped some of its features and implemented new ones which are summarized at the end of the section. We have used all the codes which we wrote during different homework assignment to capture the final 3D image and the depth values.

Given a .obj file, we parse it using an object parser and then pass it through the code written as part of our homework as it is to generate the framebuffer image of the 3d model.

# **Blurred Depth Map:**

Once we have converted the image, the display object contains the perspective corrected depth values for the z-buffer. We use GzGetDisplay to get the z values for each pixel in the image. We also keep track of the minimum and maximum z value present in the image. Once we obtain the z value for each pixel, we normalize it within the range to get all the values between [0, 1] and then scale it back to [0, 255] for display.

Once we have got the depth map, we blur the depth map using a 3X3 tent filter as given below:

1 2 1

2 4 2

1 2 1

Before applying the filtering we extend the image with zeros on each side so that we are able to apply the filter on the boundary pixels as well.

The results of the above are as follows:

Figure: Blurred Depth Map for pawn.obj in different orientations

Figure: Blurred Depth Map for rhino.obj in different orientations

Figure: Blurred Depth Map for car.obj in different orientations

# **Edge Map:**

For each pixel (i,j) of the depth map image, which have the same values for R, G and B, calculate its Row and Column Gradient (Gr and Gc) respectively using the following matrices:

Gr =

-1 -2 -1

0 0 0

1 2 1

Gc =

1 0 -1

2 0 -2

1 0 -1

Calculate Edge Magnitude Gm =

The Edge Magnitude Gm will be fractional values and will have decimal places, take the integer value of each edge magnitude to quantize the values.

For each pixel (i,j) in the image, get the binary image Ib with a given threshold T.

Ib(i, j) = 0 if Gm(i, j) < T

255 if Gm(i, j) >= T

Here 0 represents non-edge pixel and255 represents edge pixels. Ib is also called the edge map of the image. Here are the results of the above:

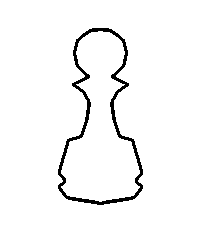
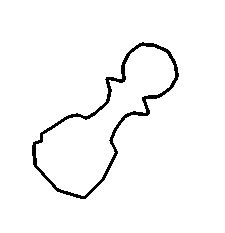
 

Figure: Edge Map for pawn.obj in different orientations

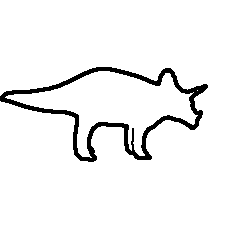
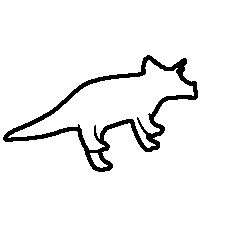
 

Figure: Edge Map for rhino.obj in different orientations

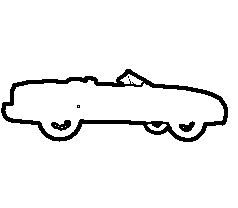
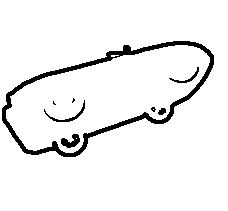
 

Figure: Edge Map for car.obj in different orientations

# **Generate Loose Strokes:**

To generate loose strokes, we first need to find the forcefield vectors at each pixel in the image. These forcefield vectors basically provide a direction in which to move a stroke. It is a 2D vector and it has two components:

1. The first component is found out by rotating the gradient vectors of the blurred depth map by 90 degrees. This first component serves to push the particles along the edges of the map. It is achieved in the following way:

ff1x = -1.0 \* gy

ff1y = 1.0 \* gx

1. The second component of the forcefield is found out by first blurring the edge magnitude of the blurred depth map. Next, the gradient at each pixel is found and then rotated by 90 degrees to find the second component. This second component forces the particle back on the edge if it starts to stray. This is done in the following way:
2. Store the edge magnitude as a separate image, Ib in the following way:

Ib(i, j) = 255 if Gm(i, j) > 255

Gm(i,j) o.w.

1. Blur the edge magnitude image using 3X3 tent filter as told in the first subsection.
2. Find the edge gradient of the blurred image as told in the previous sub section, store them as gx and gy.
3. Rotate them by 90 degrees to get the second component of the forcefield vector.

ff2x = -1.0 \* gy

ff2y = 1.0 \* gx

Finally the forcefield vector is found out by subtracting the second component from the first.

ffx = ff1x - ff2x

ffy = ff1y - ff2y

Next we use a physically-based modeling technique to generate the loose strokes. We first generate a set number of particles depending on the number of pixels colored in the edge map. For each particle, we generate a random position in the image. If the position lies on the edge map, we draw a stroke from the starting position in the direction of the forcefield vector with a pre decided distance. Once we reach the end position, we again draw another stroke in the direction of the forcefield vector at that new position with a specified length, This process is restricted to a maximum of 5 times. We also make note of the pixels already covered by the loose strokes and if another particle lands on the same pixel we ignore it. This process is repeated until all the particles have been processed.

The results for the above are as follows:

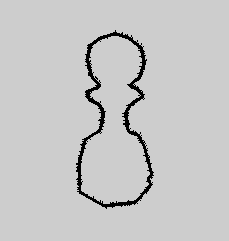
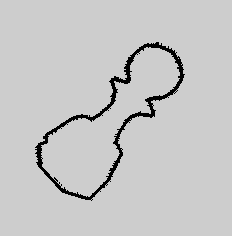
 

Figure: Loose strokes for pawn.obj in different orientations

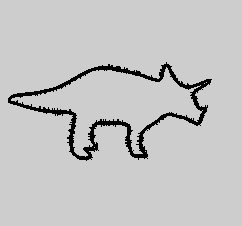
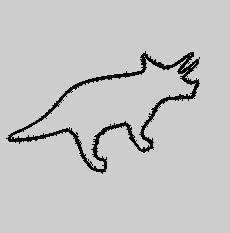
 

Figure: Loose strokes for rhino.obj in different orientations

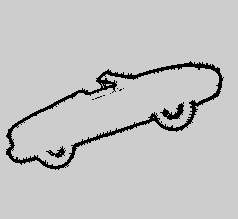
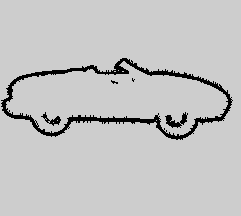
 

Figure: Loose strokes for car.obj in different orientations

**Features of the Tool**

**Lighting**

The tool provides the user with the choice of adjusting the lights for a particular scene by giving full control over the two directional lights and ambient light present for each object file.

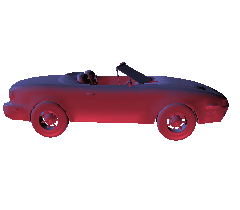
** **

Figure: car.obj for different lighting conditions

**Edge Thickness**

The user has an option to increase/decrease the edge thickness of the object.

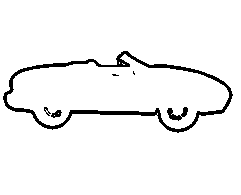
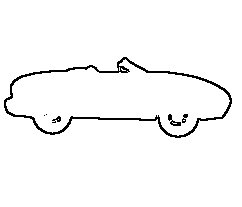
 

Figure: car.obj for different edge thickness

**Edge Randomness**

The user has an option to increase/decrease the edge randomness of the object.

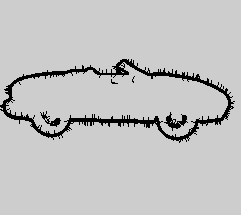
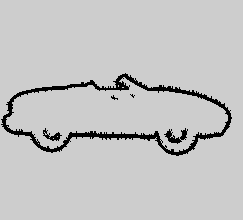
 

Figure: car.obj for different edge randomness

**Blurriness**

The user has an option to increase/decrease the blurriness of the object.

Figure: car.obj for different blurriness

Different Steps:

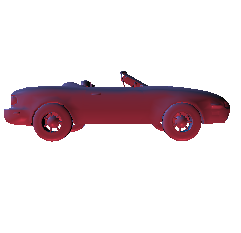


Figure: car.obj rendered as a simple 3d model



Figure: car.obj rendered as a depth map

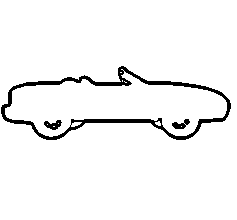


Figure: car.obj rendered as a edgemap



Figure: car.obj rendered as a blurred 3d model



Figure: car.obj rendered as procedural texture applied on blurred 3d model



Figure: car.obj rendered as image texture applied on blurred 3d model

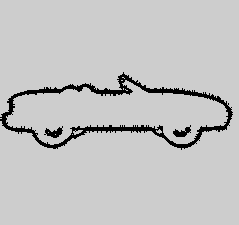


Figure: car.obj rendered as loose strokes



Figure: car.obj rendered as a loose and sketchy rendered image using image texture

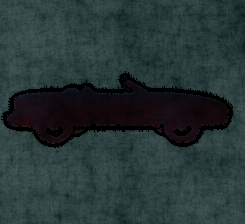


Figure: car.obj rendered as a loose and sketchy rendered image using procedural texture