September 30, 2011

NON-PHOTOREALISTIC RENDERING OF A HORSE

A REPORT ON THE DISPLAY OF A HORSE THAT DOES NOT LOOK LIKE A HORSE

WIM LOOMAN (*92692734*)
wgl18@uclive.ac.nz

I. INTRODUCTION

THIS REPORT details the implementation of a pair of non-photorealistic rendering algorithms using OpenGL. These rendering algorithms were then tested on a horse model to see their effectiveness.

The first algorithm was based off what is known as "*three-tone shading*," this is a technique primarily used for generating images that look like cartoon drawings.

The second algorithm was an attempt at a "*pencil shading*" algorithm. This involves rendering the 3D model as if it was hand shaded using a technique like cross-hatching.

A. Three-Tone shading

Figure 1a shows a real-life example of two-tone shading. This is basically the same as flat-shading the image (drawing each section in a single constant colour) then adding in a darker shadow section to provide a sense of depth. Three-tone shading takes this a step further and adds in a lighter highlight section that provides another clue for the depth sensing.

The actual implementation of three-tone shading is relatively simple. It is basically the diffuse term of the standard lighting model, restricted to just three tones for shadow, normal and highlights.

B. Pencil Shading

Figure 1c shows a variety of real-life pencil shading techniques. The main outcome of all these techniques is more graphite left on the darker areas, the different methods to achieve this result provide a different feeling for the images. The main technique focused on for this implementation was cross-hatching, a better image of this can be seen in Figure 1b. The cross-hatching technique uses a large number of short line segments overlapping, the darker the region the more line segments are used. This provides for an easy implementation by a real-time rendering algorithm, simply overlay more line segments on dark sections.

The method chosen to implement the cross-hatching was composed of three major sections; first a texture had to be loaded to use as the basis of the line segments, next the model had to be pre-processed to extract data required for locating the texture on the segments, finally during run-time a few calculations had to be performed to get the correct darkness.

To provide easy storage and access of the texture it was decided to use a 3D texture

object. This allowed the lines to be stored across the first and second dimensions and the differing darkness level to be stored up the third dimension. This also provided very easy access in the shader hardware to acquire the texture value without any branching.

1) Texture Creation: The texture creation used was the weakest point of the pencil shading algorithm. The implementation simply iterated down through the layers of the 3D texture, at each layer it added a few more lines to that layer and all below it. The lines added were simply complete horizontal lines across the entire texture, an example of the output can be seen in Figure 2.

2) Model Pre-processing: The model had to be pre-processed to identify where in the texture each vertex was. Because the texture was generated with lines in a single direction the angle at which the texture was applied also had to be decided. To best simulate real pencil strokes the angle to apply the texture was decided to be such that the lines align with the direction of maximum principal curvature (T_1 with associated curvature κ_1). Figure 3 shows how the two directions of principal curvature relate to the normal vector and tangent plane at a point, T_1 is the vector lying within the tangent plane and the principal curvature plane with the maximum curvature value.

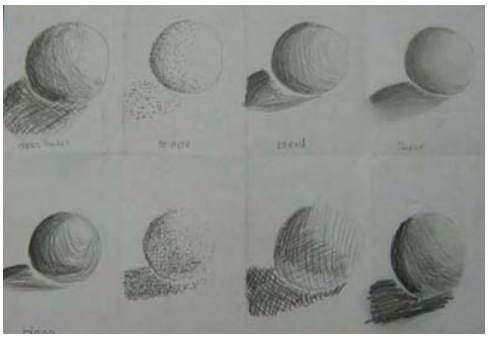
Obviously since most models passed to the program will not have a simple mathematical description this direction of maximum curvature will have to be approximated. Timothy D. Gatzke and Cindy M. Grimm performed a large test of different methods of estimating principal curvature parameters[3]. From a quick reading of this the method proposed by Taubin[4] was chosen. While this is a relatively error prone method it has a very simple implementation; in this case errors do not matter overly as any real hand-drawn image would contain a multitude of errors anyway.

The basic outline of the method is to take a variety of data points from the one-ring neighourhood of the vertex in question (mainly consisting of normal approximations at the points along with vectors from the point in question to each point), process this data to arrive at a 3x3 matrix with eigenvalues and eigenvectors obeying a fixed homogenous linear relationship with the required valeus. This matrix can then be constrained to the tangent plane with a Householder transformation and the resulting 2x2 sub-matrix can easily be diagonalised with a Givens rotation. Once the matrix has been diagonalised computing the eigenvalues and eigenvectors is simple and can be used to directly recover κ_1 and T_1.

Once these values are found the final step is to use these to relate each vertex (x, y, z) to

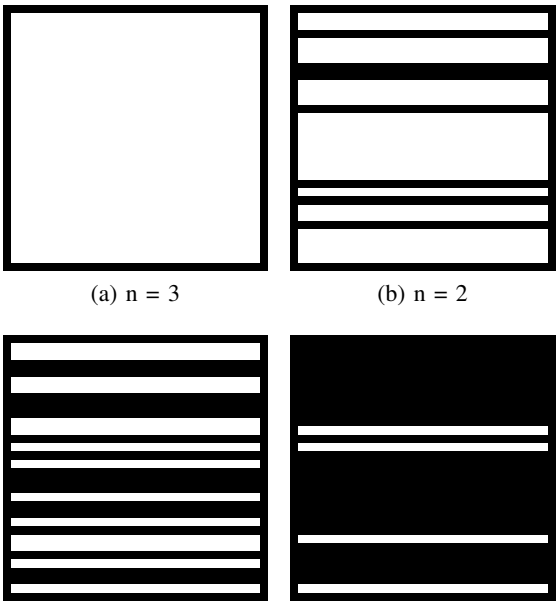
(a) Two-tone shading.

(b) Pencil hatching. [1]



(c) Different pencil shading techniques. [2]

Fig. 1: Real-life examples of non-photorealistic renderings.



(a) n = 3 (b) n = 2

(c) n = 1 (d) n = 0

Fig. 2: A small example texture (a black border has been added to the layers).

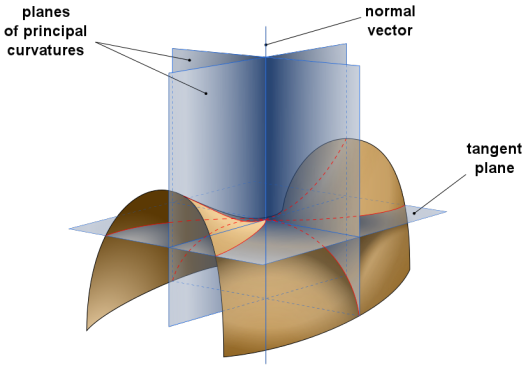


Fig. 3: Saddle surface with normal planes in directions of principal curvatures. [5]

it's corresponding texture co-ordinate (s, t). Because the model is kept as a triangle mesh each face will have three T_1 values associated with it, one at each vertex. To provide a simulation of multiple crossing lines at different angles these are used to generate three different sets of texture co-ordinates for each face.

To compute the s and t values one of the vertices is chosen to be the source vertex (a), this is given co-ordinates $(s, t) = (0.0, 0.0)$. First the direction vectors to each of the other vertices (v_{ab}, v_{ac}) is projected on to the tangent plane at the source vertex using the normal (n_a):

$$\bar{v_{ab}} = v_{ab} - (v_{ab} \cdot n_a)\, n_a \quad (1)$$

$$\bar{v_{ac}} = v_{ac} - (v_{ac} \cdot n_a)\, n_a \quad (2)$$

These tangent direction vectors are then projected on to the directions of maximum and minimum principal curvature to determine their s and t values:

$$(s_b, t_b) = (\bar{v_{ab}} \cdot T_1, \bar{v_{ab}} \cdot T_2) \quad (3)$$

$$(s_c, t_c) = (\bar{v_{ac}} \cdot T_1, \bar{v_{ac}} \cdot T_2) \quad (4)$$

This is then repeated with each of the vertices set as vertex a. This results in 3 sets of texture co-ordinates for each vertex that are passed through to the shaders.

3) Run-time Processing: During run-time the only calculation that needs to be performed is determining the final texture value. This is based off the diffuse term of the standard lighting model, ($p = \max(n \cdot l, 0.0)$ where n is the normal at the point and l is the vector towards the light source). Once this has been determined the 3D texture is simply accessed at (s, t, p) with linear interpolation.

II. Implementation

A. Three Tone Shading

The three tone shading was simply implemented as a vertex and fragment shader pair. The vertex shader transforms the normal vector by the normal matrix and records the light direction vector from the current vertex to the 0th light source. The fragment shader then takes the interpolated vectors, re-normalizes them and finds their dot product. If the result of this is below 0.15 (determined experimentally) the colour used is 0.6 times the materials diffuse colour, if it's above 0.95 (again experimentally determined) then the colour used is 1.2 times the materials diffuse, otherwise the materials diffuse is used unchanged.

B. Pencil Shading

1) Texture generation: The texture generation was simply implemented as stated. The inbuilt random function was used to determine which rows to put the lines on. Each layer down the texture there were simply an extra 32 lines times the number of layers passed added, so the top layer got no lines, the next got 32, the next got 64 + the original 32, the next 96 + the original 96, etc.

2) Model Pre-processing: The pre-processing of the model was implemented mostly in the `Vert` class of the half-edge structure. The lowest level operations were a part of the general geometry library, but the rest of the high level operations were left to be implemented directly in the `Vert#calculate_normals` method as they were not going to be re-used elsewhere.

The `Face` class also contained a few parts of the pre-processing for converting the maximum primary curvature found in the Vert class to the required texture co-ordinates.

3) Run-time Processing: The run-time processing was implemented very similarly to the three tone shading. A vertex shader computed the transformed normal vector and light direction at each vertex and passed the (s, t) texture co-ordinates through to the fragment shader. These were interpolated by the OpenGL pipeline then the fragment shader calculated the dot product of the normal and light direction to get the third component of each texture co-ordinate. The texture was then accessed three times and the average of the colours at the point was used to colour the fragment.

III. Results

A. Three-Tone Shading

The results of the three tone shading can be seen in Figures 4 and 5. Looking at these the technique is definitely working effectively. Likely the biggest flaw is the lack of crease lines to identify areas of sharp model curves. The silhouette edges have been drawn showing where parts of the model overlap other parts, most noticeable around the body-leg joins and the edge of the horse head.

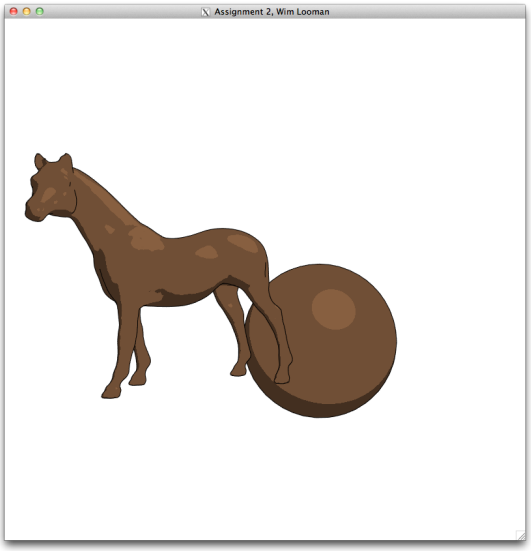


Fig. 4: Three tone-shading of a horse.

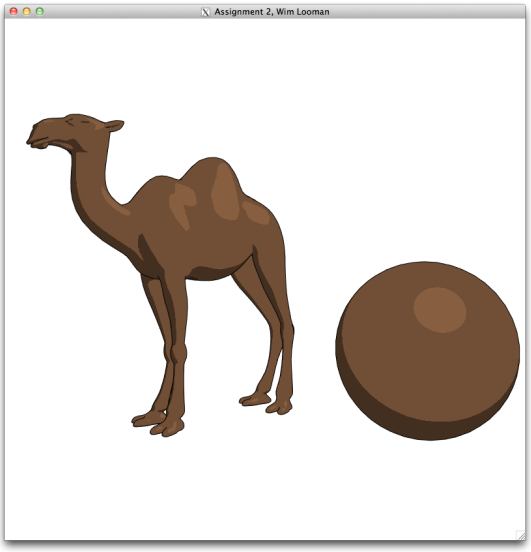


Fig. 5: Three tone-shading of a camel.

These help a lot in identify the depth of the model, but adding in crease lines along areas such as the edge of the leg muscles on the horse's body and down the camel's body between the humps would greatly enhance the differentiability of these attributes.

B. Pencil Shading

Figures 6, 7, 8 and 9 show the pencil shading result on a horse and camel model, along with close ups of a part of each.

Close up the pencil shading doesn't look too bad, the largest flaw is probably the fact that the lines don't meet up very well. At

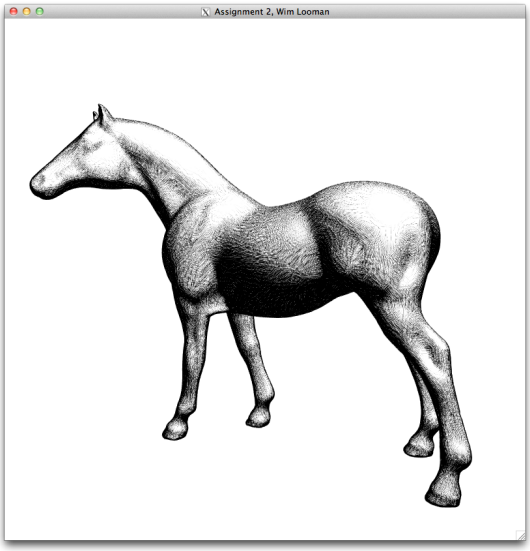


Fig. 6: Pencil shading of horse model.

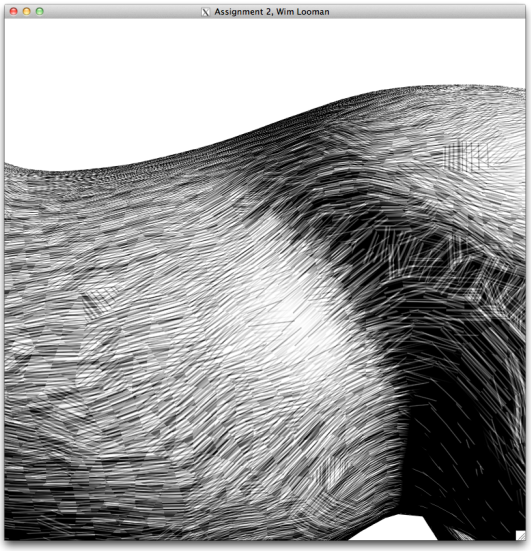


Fig. 7: Close up of horse model.

distance the lines sort of become a mess however, this is mainly because of the lack of mipmapping. If the texture was mipmapped then the view further out would be greatly enhanced. However the very simple single pixel line texture used is not conducive to automatic mipmapping.

So really the greatest enhancement would be provided by having a better pencil texture either generated or pre-drawn and loaded at run-time. For example the stroke generation by Mario Costa Sousa and John W. Buchanan [6] shown in Figure 10 looks much better, using the method they propose to generate the textures would likely make the pencil shading a lot more lifelike.

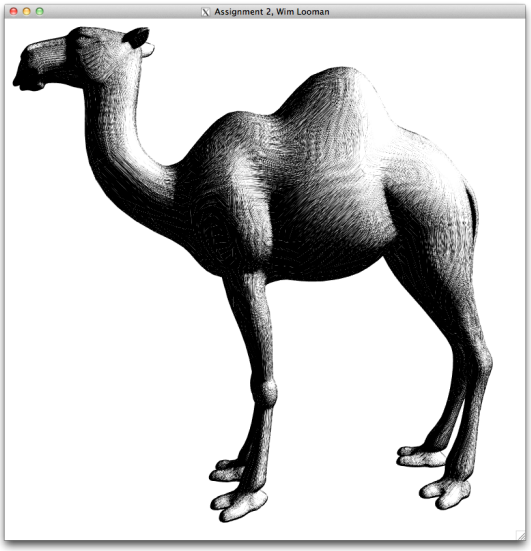


Fig. 8: Pencil shading of camel model.

IV. Code Description

A. C Modules

1) `geom`*:* This is a rewrite of the 3D python geometry library I used last year for the second COSC363 assignment. Just simple 3D geometry stuff to make working with points and vectors easier. Now also includes simple 3x3 matrix computations as well.

2) `view`*:* This module mainly takes care of the display and reshape functions and setting up what display type is used for the models along with all the OpenGL initialisation.

3) `player`*:* This module simply represents the cameras viewpoint and handles moving this around when the movement keys or mouse are used.

4) `controller`*:* This module takes care of calling all the other initialise functions then takes the input and converts it to the correct function calls.

5) `lights`*:* This module takes care of the light initialisation and display.

6) `shaders`*:* This module handles the loading and changing of shaders.

7) `sphere`*:* This module uses subdivision of an icosahedron as

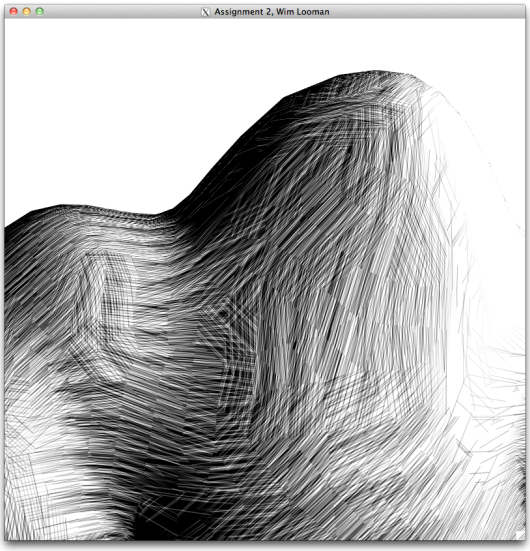


Fig. 9: Close up of camel model.

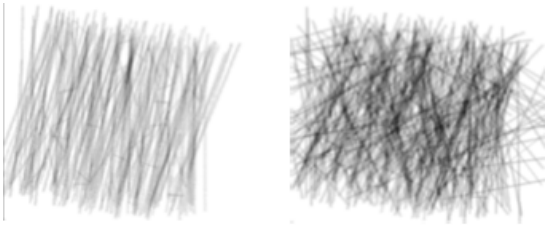


Fig. 10: Pencil strokes generated by Sousa and Buchanan's method.

provided by halma from http://www.3dbuzz.com/vbforum/showthread.php?118279-Quick-solution-for-making-a-sphere-in-OpenGL

8) `texture`*:* This module handles the generation of the pencil shading texture.

9) `model`*:* This folder contains all the modules related to the half-edge data structure and the model loading and displaying.

10) `time`*:* This module provided both POSIX and Windows implementations of a high resolution timer to keep the movement constant.

B. Shaders

All the shaders are contained in the `shaders` folder. These are all simply named with what they do.

C. Controls

All keys listed below must be lowercase when pushed.

1) Looking: Hold down the left mouse button and use the mouse to look around.

2) Navigation: Use *A*, *W*, *S*, *D*, *Z* and *the spacebar* to move the viewpoint left, forwards, back, right, up and down respectively.

3) Other:

Q	Toggle rotation of the model.
E	Toggle showing the FPS in the console.
R	Switch to the next shader model.

REFERENCES

[1] Manuela W1. *Example: cross hatch* used under (CC BY-NC-SA 2.0) license. Available at: http://www.flickr.com/photos/manuela_w/5063203640

[2] Manuela W1. *Values: Different types of shading* used under (CC BY-NC-SA 2.0) license. Available at: http://www.flickr.com/photos/manuela_w/5063203636

[3] Timothy D. Gatzke and Cindy M. Grimm (2006). *Estimating Curvature on Triangular Meshes*. Available at: http://www.worldscinet.com/ijsm/mkt/free/S0218654306000810.pdf

[4] Gabriel Taubin (1995). *Estimating the Tensor of Curvature of a Surface From a Polyhedral Approximation*. Available at: http://mesh.caltech.edu/taubin/publications/taubin-iccv95b.pdf

[5] Eric Gaba (Sting). *View of the planes establishing the main curvatures on a minimal surface* used under (CC BY-SA 2.0) license. Available at: http://en.wikipedia.org/wiki/File:Minimal_surface_curvature_planes-en.svg

[6] Mario Costa Sousa and John W. Buchanan (1999). *Computer-Generated Graphite Pencil Rendering of 3D Polygonal Models*. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.86.3948&rep=rep1&type=pdf