

CSCI-580: 3D GRAPHICS AND RENDERING



Non-Photorealistic Rendering: Cel-Shading

Chetan Ahuja

Keyur Bhulani

Ricardo Chavarria

Arzin Chibber

12/06/2007

TABLE OF CONTENTS

<i>Abstract</i>	<i>3</i>
<i>Introduction</i>	<i>4</i>
<i>Phase 1: Shading</i>	<i>5</i>
<i>Shading Implementation</i>	<i>5</i>
<i>Phase 2: Outlining</i>	<i>7</i>
<i>Depth Map Implementation</i>	<i>7</i>
Drawing the Depth Map	<i>7</i>
<i>Normal Map Implementation</i>	<i>8</i>
Two Rendering passes	<i>8</i>
One Rendering pass	<i>8</i>
Alternative Normal Map:	<i>8</i>
<i>Sobel Operator</i>	<i>9</i>
<i>Contour Detection Filter</i>	<i>10</i>
<i>Conclusion</i>	<i>12</i>
<i>References</i>	<i>13</i>

ABSTRACT

Cel-Shading is a method of rendering 3D objects to make them look like a 2D drawn images. This paper will focus on explaining the existing techniques we implemented to achieve Cel-Shaded rendering on the CSCI-580 rendering API.

INTRODUCTION

Cel-Shading has its origins on celluloid 2D animations like the ones created by Disney and many other animation companies. It is also called “toon-shading” because its results mimic cartoon and comic drawings. In simple terms, outlines are drawn between areas of smooth color to get the simulated 2d drawn effect. (Möller)

Our approach to Cel-Shading was very hands-on. Surface shading was easily implemented as a shader that runs on a per-pixel base and computes the correct shade for the pixel. For outline detection, we started by implementing Object Space detection using back-face culling techniques and then moved on to use image processing techniques, in Image Space, to achieve great results.

There are many methods of implementing Cel-Shading, but most of the time two phases are involved: shading and outlining.

The first phase is to shade the object using one color, taken from the model’s data or from the model’s texture. This phase may also include adding specular highlights.

The second phase is to find the outlines of the object, both internal and external, and then draw them. Outlines are what makes cartoons particular because they define boundaries between areas of different colors. In real life, we have no outlines, but cartoons need them to define differences between entities.

PHASE I: SHADING

Shading is the process of giving color to a surface depending on its orientation towards the camera, the lights that affect it and the color data encoded into the object or its texture. In Cel-Shading, it is done either using a solid flat color, or using a multi-tone approach (Möller).

When doing multi-tone shading, if the pixel faces a light, the color is displayed in full intensity. Pixels which are partially facing the light display varying degrees of the color, but the color changes are very sharp. Specular highlights are also calculated for each pixel if the material currently being rendered requires it. To achieve a Cel-Shaded effect, the specular highlight has to be clamped to either 0 or 1.

SHADING IMPLEMENTATION

To implement shading, we use a texture lookup function. The texture lookup function uses either a 1D texture map with a grayscale, or a 1D array of floating points ranging between 0 and 1.

The Diffuse Shading dot product ($N \cdot L$) is used to calculate the texture map index (Möller). After calculating the index, the texture lookup function is called with the index as a parameter. The texture lookup returns a shade (between 0 and 1) that is then multiplied with the surface's color (either a texture or a solid color) to get the correct color for the pixel.

In the case of Specular highlights, they are calculated for each pixel using the following equation:

$$\mathit{specularColor} = (N \cdot E)^{\mathit{specpower}}$$

If this value is above a certain threshold, the pixel is output in white. Otherwise, the specular component is zero and the shading color will be the output. This allows for full specular highlight or non at all.

In case the object has a texture instead of a solid color, it is only a matter of using either of our filters (Sobel or Contour, explained in the next section). A grayscale value has to be calculated from the color components (r, g, b) values using the formula:

$$I_{\mathit{grayscale}} = 0.3 * I_{\mathit{red}} + 0.59 * I_{\mathit{green}} + 0.11 * I_{\mathit{blue}} \text{ (Saurabh)}$$

Instead of applying it on the Normal map color value, the filters are applied on the texture color value. Quite simply, when this calculated grayscale value of the texture color is put through the filter, it gives us texture discontinuities. The final result will be that abrupt changes in the colors of the texture will be outlined. (Wasilewski)

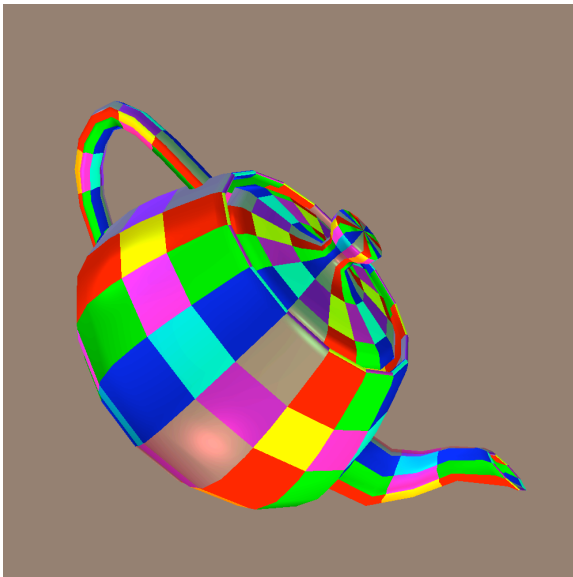
The following image shows the results of the shader:



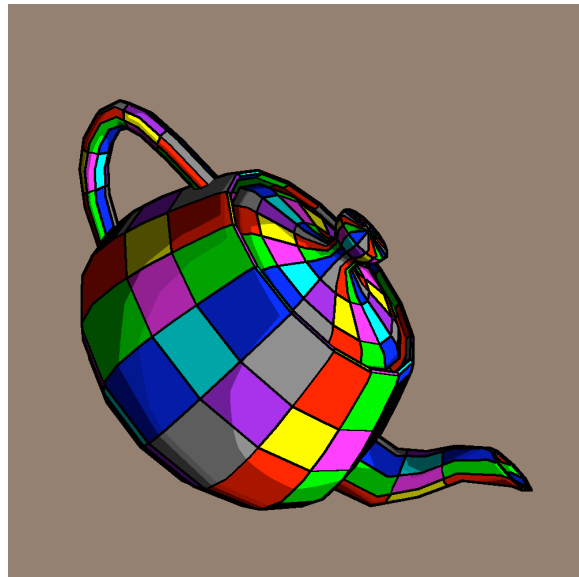
Solid-color Cel-Shaded Rendering with Specular Highlights

The Cel-Shaded output bellow has a Contour Detection filter passing over it in 2 phases. Each produces black outlines. One phase will outline silhouette edges and the other will outline abrupt changes in the texture color.

Phong Shaded Rendering



Cel-Shaded Rendering



The outlining process is explained in more detail in the next section.

PHASE 2: OUTLINING

Outlines can be implemented in two different stages of the rendering process. The first is in the Model (Object) space and the second is the Image space. Our implementation detects outlines in the Image Space. Detection in Image space can be done using two methods – Depth Map and Normal map. A combination of both methods will also yield higher quality because both methods will find edges that the other can't find.

DEPTH MAP IMPLEMENTATION

A Depth Map, also known as Z-buffer or Depth buffer, can be represented as a 2D array in which the X, Y coordinates of the pixels are the array indices and the value holds the Z value. Hence, for an image that is 256 pixels by 256 pixels, we will have a 256×256 array, where the content of the array is the Z value for each pixel.

DRAWING THE DEPTH MAP

The following image is a visual representation of a Depth Map. Here the color of the Depth Map image corresponds to the Z-values of the 3D model. It is important to note that the Depth Map is relative to the camera position. Hence, the Depth Map image is sensitive to changes in camera position and object transformations. In the above gray scale image, the Depth Map shows the minimum value of Z as white, and the maximum value of Z as black. The objective is to find large Z discontinuities of the Depth Map which are going to be the outlines of the image. (Hertzmann) (Decaudin)



Depth Map Image

The implementation in our API entailed using the display buffer (*display->fbuf*) which contained the x, y, z, r, g, b, a (Position, Color and Alpha) values. We take the Z value and scale it between 0 and 1, where 0 represents the minimum value of Z and 1 represents the maximum value of Z for the Teapot model. The Z value replaces the values of r, g, b to generate the above image, but we map the values (0, 1) to (255, 0).

NORMAL MAP IMPLEMENTATION

The Normal map is an image which represents the surface normal at each pixel on an object by giving it a color. The values of the (r, g, b) color components of a pixel in the Normal map correspond to the (N_x , N_y , N_z) surface normal at that pixel. Computing a Normal map can be done in one of two ways:

TWO RENDERING PASSES

On the first rendering pass (I_1) we place three directional lights on the scene. A red light (1,0,0) lying on the positive X axis, a green light (0,1,0) lying on the positive Y axis, and a blue light (0,0,1) lying on the positive Z axis. The ambient coefficient (K_a) and the specular coefficient (K_s) are set as 0, while the diffuse coefficient (K_d) is set to 1. The calculation of the color for the normals (N_x , N_y , N_z) is as follows:

$$\mathbf{color} = \mathbf{red} \max\{N_x, 0\} + \mathbf{green} \max\{N_y, 0\} + \mathbf{blue} \max\{N_z, 0\}$$

The second rendering pass (I_2) places the three directional lights of the first pass in the corresponding negative axes. The calculation of color for the normal (N_x , N_y , N_z) is as follows:

$$\mathbf{color} = \mathbf{red} \max\{-N_x, 0\} + \mathbf{green} \max\{-N_y, 0\} + \mathbf{blue} \max\{-N_z, 0\}$$

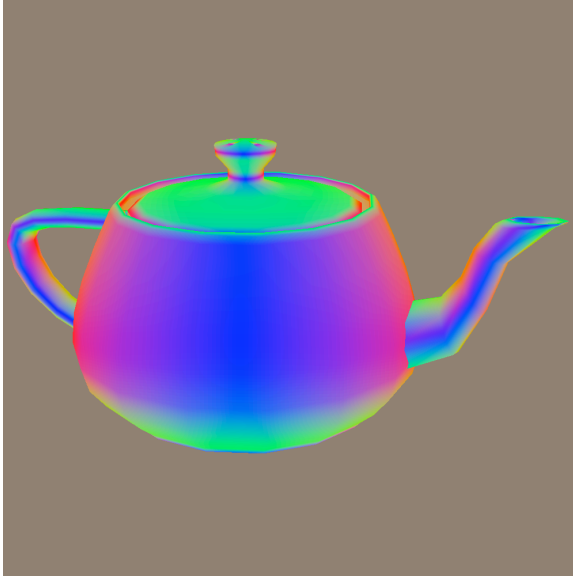
Finally, subtracting the two renderings, $I_1 - I_2$, gives a buffer in which each component red, blue and green correspond to N_x , N_y and N_z respectively. (Decaudin)

ONE RENDERING PASS

In order to calculate the Normal map on a single rendering pass, we must put lights on the positive and negative axes at the same time, if the renderer allows it. The red light is placed on the positive and negative X axis, the green light is placed on the positive and negative Y axis, and the blue light is placed on the positive and negative Z axis. (Decaudin)

ALTERNATIVE NORMAL MAP:

An alternative Normal map can be implemented using the above techniques but using negative lights (instead of positive lights) on the negative axes. i.e.. having a negative red light coming from the negative X axis, a negative green light coming from the negative Y axis and a negative blue light coming from the negative Z axis into the model. (Hertzmann)



Normal Map



Normal Map (negative lights)

SOBEL OPERATOR

A Sobel operator is one that calculates a gradient based on the Z-value (Depth map) and surface normal (Normal map) at each pixel. The results of the operator will be a gradient which will be filtered to detect abrupt changes.

On our API, we implemented the Sobel operator by creating a Depth map and a Normal map with colors and coordinates. Afterwards, a 3x3 Sobel operator is applied on the Z-values or surface normals of a 3x3 pixel neighborhood. Two passes are performed, one for G_x (horizontal component), one pass for G_y (vertical component).¹

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

The magnitude is approximated using $G = |G_x| + |G_y|$. A suitable threshold, which decides if an outline should be present, is applied on G. (Strothotte) (Wikipedia)

A 5x5 Sobel operator was used instead of using the 3x3 operator because it improved the quality of the output (less noise).

¹ A is the source image



Outlines using 5x5 Sobel operator on Normal map

CONTOUR DETECTION FILTER

The first step is to apply the following differential operator of order 1 and size 3x3

$$g = 0.125 * (|A-x| + 2|B-x| + |C-x| + 2|D-x| + 2|E-x| + |F-x| + 2|G-x| + |H-x|)$$

where A,B, ... ,H are the 8 neighboring pixels of x:

A	B	C
D	x	E
F	G	H

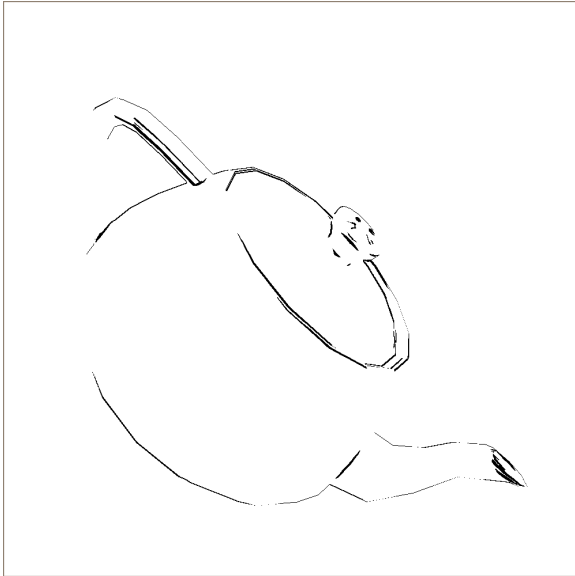
This operator produces a gradient from the Depth map or the Normal map. A threshold has to be applied on the gradient to obtain outlines. The following 3x3 nonlinear filter is used:

$$p = ((g_{\max} - g_{\min}) / K_p)^2$$

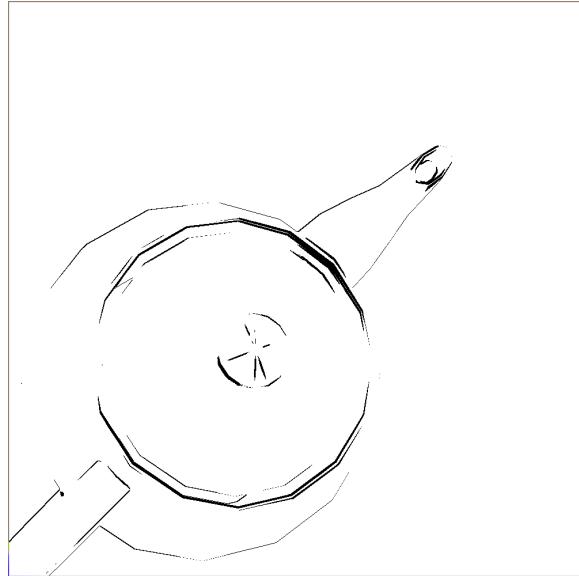
Here g_{\max} and g_{\min} are the maximum and minimum values of the gradient in the 3x3 neighborhood, and K_p is the detection threshold between 0 and 1. (Decaudin) (Saito) For our Normal map, $K_p = 0.2$. The Contour Detection filter for a Normal map should be applied on a grayscale image. We used the following formula to composite the colors and get a grayscale image:

$$I_{\text{grayscale}} = 0.3 * I_{\text{red}} + 0.59 * I_{\text{green}} + 0.11 * I_{\text{blue}} \text{ (Saurabh)}$$

After applying the filter to $I_{\text{grayscale}}$, the following Normal map edges are found:



Contour Detection on Normal map 1



Contour Detection on Normal map 2

CONCLUSION

While researching and implementing Cel-Shading, we were exposed to many methods that showed us how to implement image processing. We found that there are now definite methods to implement Cel-Shading and that it depends, mostly, on the result we want to achieve and the computing power we are going to dedicate to the rendering.

One of the most challenging aspects of the project was finding an appropriate threshold for the outline detection algorithms to work correctly.

In the end, we feel that our results are compelling and produce a high quality output on the flexible CSCI-580 API.

REFERENCES

- Möller, Thomas, and Eric Haines. Real-Time Rendering. 2nd edition. Wellesley, Ma: A K Peters, 2002.
- Saurabh, Aditya and Michael Suwandi. “Non-Photorealistic Rendering for Technical Color Illustration.” November 2007. <http://www.cs.utexas.edu/%07Emsuwandi/cs384g/CS384G_Final_Project.html>
- Wasilewski, Michael. “Toon Shading and Silhouette Rendering.” November 2007. <<http://www.postulate.org/silhouette.php>>
- “Cel-Shaded animation.” *Wikipedia, The Free Encyclopedia*. November 2007. <http://en.wikipedia.org/wiki/Cel-shaded_animation>
- “Sobel operator.” *Wikipedia, The Free Encyclopedia*. November 2007. <http://en.wikipedia.org/wiki/Sobel_operator>
- Ishaya, Vishvananda. “Real-Time Cartoon Rendering with Direct-X 8.0 Hardware.” <<http://www.gamedev.net/reference/articles/article2021.asp>>
- Decaudin, Philippe. “Cartoon-Looking Rendering of 3D-Scenes” Research Report INRIA #2919 (1996)
- Saito, Takafumi, and Tokiichiro Takahashi. “Comprehensible Rendering of 3-D Shapes.” Computer Graphics, Volume 24, Number 4, (1990)
- Hertzmann, Aaron, “Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines.” Media Research Laboratory <<http://mrl.nyu.edu/~hertzman/hertzmann-intro3d.pdf>>
- Strothotte, Thomas, and Stefan Schlechtweg. Non-Photorealistic Computer Graphics: Modeling, Rendering and Animation. 1st edition. San Francisco, Ca: Morgan Kaufmann Publishers, 2002.