Project on Cartoon-Looking Rendering of 3D-Scenes

Chenyu Wang
Oregon State University

Abstract

Instead of rendering algorithm which is trying to generate image as real as possible, the algorithm introduced in this report from [Decaudin 1996], is emphasized on how to make good cartoon looking image from a real 3-D object. Even though the cartoon looking image seems easy to approach in the first place, the author still brings the texture, specular highlight, and shadow into the image which makes the task harder than expected.

In the final project, I followed the introduction and implemented part idea of the paper announced. And I also did some interesting discussion on the edge detection setting which is not introduced in detail in author's paper.

CR Categories: I.3.4 [Computer Graphics]: Paint systems;
Keywords: cartoon-looking, outline, shadow.

1 Introduction

In most conditions, people studying computer graphics are intending to make the 3D-scenes more and more realistic by using the raytracing, Phong model, calculating the reflections and shadows. However, as a cartoon fan, I also think cartoon-looking based material is getting more and more popular in these years as a result of the widely used of smart phones. The cartoon looking games is less memory and CPU consuming than realistic looking games with the same or even more entertaining feeling. Besides, people also use cartoon looking algorithm to creating the 3D animation. Because in the past, animation is created by hand frame by frame which requires a lot of work, time and money.

Based on the situation described above, developing an algorithm which can render every frame automatically will save a lot of paper and time for us.

The algorithm was first introduced by Decaudin in 1996, [Decaudin1996] he designed the algorithm which took a 3D scene description as input and can automatically calculate the images. The image calculated by this algorithm contains not only the constant size of outlines and uniformly colored area, but also has textures, specular highlight, and shadow information. Texture is not necessary, but it will enrich and bring more details into the image. Specular highlight will provide a visual feedback of the arterial type of the objects. Shadows will provide the information that help to perceive the 3D aspect of the scene [Decaudin 1996]. The shadow that this algorithm provide will result in two different colors on the object. The parts of the object in the shadow will have a darker color while the other parts are brighter. The edge between these parts are sharp, but not as soft as most of the situation we have in poor-realism images.

The implementation this report provides is basically based on the algorithm mentioned above, but they will not be so similar. The implementation of this report uses an easier way and can also result in a fairly good image. The shadows is basically the most different part from the original algorithm. I try to use the shadow mapping technology introduced online in detail. But I cannot get the whole point especially on the texture mapping part. So even though I implemented it in the code, I didn't finally use it. But I will still discuss the shadow in the following report.

2 Algorithm

The algorithm provided by the [Decaudin 1996] is kind of complicated, but since this algorithm is the fundamental of this reports implementation, it will be fully described and explained in this report.

Figure 1 shows the working flow of the whole algorithm, and it mainly can be divided into two parts: outlines and shadows. The outlines includes two parts which are silhouettes and crease edges.

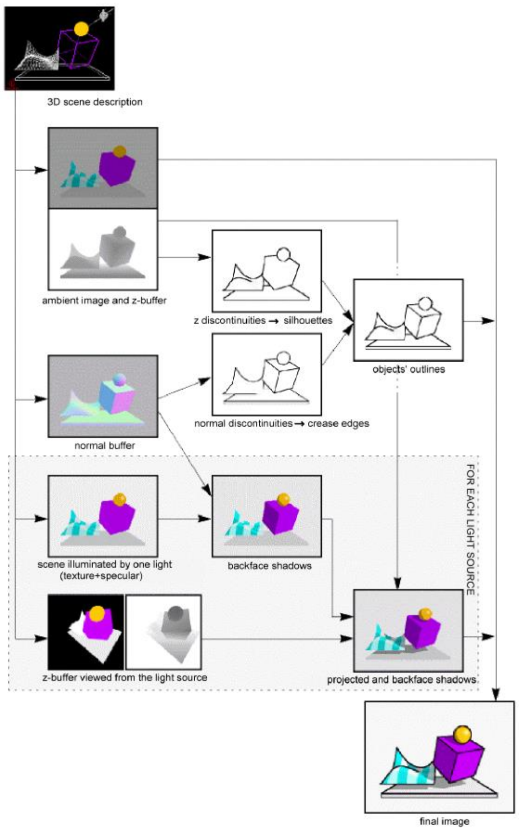
3D scene description
ambient image and z-buffer
z discontinuities → silhouettes
objects' outlines
normal buffer
normal discontinuities→ crease edges
FOR EACH LIGHT SOURCE
scene illuminated by one light (texture+specular)
backface shadows
z-buffer viewed from the light source
projected and backface shadows
final image

Figure 1: The overview of the author's algorithm, the above two create the outlines and the beanth two creates shadow and color.

The shadows can be decided into two parts and they are backface shadows and projected shadows. First of all, the algorithm takes a 3D scene description as the input, and then it will render the image with only ambient light on. This operation will result in a general form of the object which stands for the scene being looked at the camera position. The color filled in the form is constant and it will be treated as a par to the main color in the final cartoon-looking result. This image will also stores the depth information even it looked not has depth at all. The depth information is a really important information that need to be used in the after steps. It will be fully described in the following sections.

2.1 Outlines

The outlines created in this algorithm can be decided into two different types. The one is silhouettes and another one is crease edges. Silhouettes correspond to places where the view direction is tangent to the object's surface [Decaudin1996]. Visible crease edges is always in the general outline of the object, in the image, it is usually in the middle of the objects, which gives the details about the edges that can be seen due to the view point.

The methods this algorithm uses to create the silhouettes and crease edges are quite similar, so only the method creating silhouettes will be clearly explained, and about the method creating crease edges, our report will only introduce the differences.

The method creating silhouettes can be divided into two parts. The gradient of the z-buffer value will be calculated first, and based on the result we have, we will use a non-linear filter to determine where the outlines should be drawn. The way we use to calculate the z-buffer value gradient is using a counter direction filter on the pixel we choose and the eight neighbor-pixels around it. The equation listed below shows the way calculating the gradient of the z-buffer value. It should be noted that A, B and C are the three pixels in the pixel x, which is the chosen pixel, above line. D and E are the two pixels beside the pixel x. The F, G and H pixel are the pixels next to x in the next line. The reason why B, D, E, and G pixel are taken into count as double value is because they are the pixels direct connect to the pixel x.

$$g = \frac{1}{8}\begin{pmatrix} |A-x| + 2|B-x| + |C-x| + 2|D-x| \\ +2|E-x| + |F-x| + 2|G-x| + |H-x| \end{pmatrix}$$

By calculating the gradient of every pixel, we can have a matrix which has the same size of the image, and every pixel position stores the z-buffer value gradient relate to that pixel. Then, the next step is to calculating the edge-limit-value, and compare it with every gradient we have in the pixel position. The way we create the edge-limit-value is by calculating the neighbors of that pixel position again using a non-linear filter. The equation showed blow is the way that algorithm uses to calculate the edge-limit-value. It should be noted that the g_{max} and g_{min} in the equation come from the gradient values around the selected pixel x and its eight neighbors.

$$\mathrm{p} = \min\left\{\left(\frac{g_{max} - g_{min}}{k_p}\right)^2, 1\right\}$$

The k_p showed in the above equation is the detection threshold in between 0 and 1. The lower k_p is the more silhouettes will be detected [Decaudin 1996]. The difference of the outlines we get by using different k_p value will be introduced in the implementation part. By following the steps mentioned above the algorithm will provide a silhouettes, as for the crease edges, we will use the same type of calculation mentioned above. However, the differences between these two types of outline is that the equations will be used on different buffers and the k_p value will be set in different ways.

The buffer that will be calculated with is no longer the z-buffer we used above, it will be that normal buffer. The way to create a normal buffer is that we will render two more images first, each image will has three light sources on. We will render two images with the color calculated based on the two equations showed below. Then in order to get the final normal buffer, we will subtracting these two renderings I_1-I_2 to get a buffer in which each component red, blue, green corresponds to n_x, n_y, n_z respectively. It also should be mentioned that the n_x, n_y, n_z mentioned above and shown in the equation is the three coordinates of the normal associated to the corresponding 3D point [Decaudin 1996].

$$\begin{aligned}\text{color} &= \text{red} * \max\{n_x, 0\} + \text{green} * \max\{n_y, 0\} \\ &\quad + \text{blue} * \max\{n_z, 0\} \\ \text{color} &= \text{red} * \max\{-n_x, 0\} + \text{green} * \max\{-n_y, 0\} \\ &\quad + \text{blue} * \max\{-n_z, 0\}\end{aligned}$$

By using the same calculation on the z-buffer entioned above on the normal buffer we just calculated, the result will shows up as the crease edges we want. Finally, by simply combine these two types of outlines, the algorithm will provide the final version of outlines as the result.

2.2 Shadows

In order to implement shadows, we will use only one light source a time due to the situation that multiple light sources might be added into the 3D description.

Two types of shadows will be calculated in this algorithm, one is backface shadows and another one is projected shadows. However, before calculating the shadows, we need to render the image first.
The basic idea to rendering the image is to remove the dot product of the object's normal and the light vector from the Phong shading equation. If the object is specular, then the specular highlight will be kept because it provides information about object's material type [Decaudin 1996].

In order to achieve the goal, the algorithm will first copy the diffuse color into ambient, then set the diffuse color to black, and also keep the specular. By doing in this way, it will result in an image where each object is uniformly colored by its diffuse color. Also by doing this, the texture and the specular highlight information will be stored in the image. In order to create the backface shadows, the algorithm will just simply examine the dot product of the selected point's normal and the its light vector. Because of the fact that the backface shadows are just the object areas opposed to the light source [Decaudin 1996], if the dot product results in a negative value, then the selected point is in the shadow and its color needs to be darken. Calculating the projected shadows is more complicated than implementing backface shadows, the algorithm uses a technique called "shadow mapping".

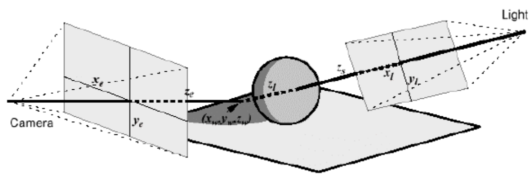

Figure 2: The shadow map technique to calculate projected shadows.

As the Figure 2 shows above, this technique can be divided into several steps. The first step is to calculate every pixie's coordinate in the image rendered from camera position int the real 3D world coordinate. The pixel coordinate (x_e, y_e, z_e) is relate to the position this pixel in the image pixel matrix, the z_e value is from the z-buffer value in this camera position. The calculated world coordinate (x_w, y_w, z_w) will be taken into next step's calculation. Then we will set the camera to the lighted light position, and rendering the image again in order to get the z-buffer value from the new camera position. The world coordinate (x_w, y_w, z_w) will be calculated into a new pixel coordinate in the image rendered from the light position, and they will be stored in to (x_l, y_l, z_l). By comparing the new z_l value from the new image with the new z-buffer value of the point z_s, the algorithm will determine whether the point is in the shadow or not. If $z_l < z_s$, then the point is not "seen" by the light, in another words, it is in the shadow. Except this kind of condition, then the point will be lighted up.

By combining these two types of shadows with the image rendered at the beginning of the shadow step, we will get an image with detailed texture and specular highlight information alone with both backface shadows and projected shadows.

2.3 Final result

To create the final result, the algorithm will take the advantage of the previous image rendered with only ambient light on, the final result of outline calculating.

By using the equation mentioned below, the algorithm will provide the final cartoon-looking image. It should be mentioned that the reason why it uses outline in (1-outline) way is because outlines are just black pixels, so we want those pixels in the outline area be colored into black. In order to do this, we will set the original outline pixels' value 1 into 0, and set other pixels' value from 0 back to 1.

3 Implementation

The algorithm from [Decaudin 1996] has been fully explained in the previous section. However, the whole algorithm is quite complicated and not easy to implement. The method this report using to implement the cartoon-looking image is easier to understand and implement. This method also covers the content size outline, the uniformly colored area. As the extra result of implementing outlines, the method to create a sketch-looking outline is also introduced in the following sections. The implementation will mainly be introduced in three parts, and they are color, outline and (partially) shadow. As showed in the Figure 3, the example image used in this section is more complicated than a simple teapot which I also make the teapot span to simulate the animation condition. But instead of rendering the image offline, my span teapot is online rendering which requires more efficiency than offline. Besides, in order to test the shadows, a cube is added at the bottom of the teapot showed in Figure 3.

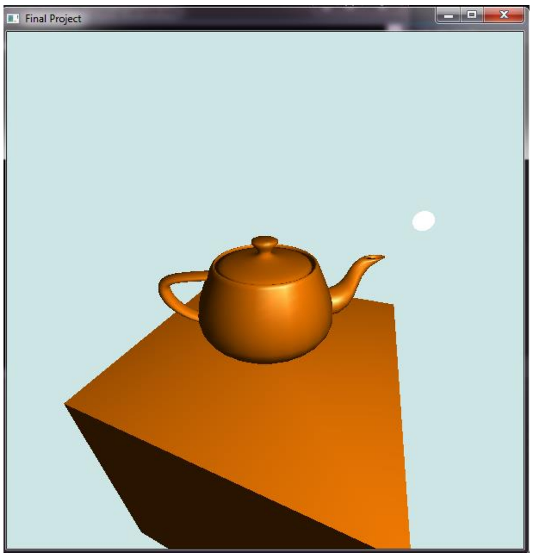

Figure 3: The realistic teapot with the white sphere as the light source but no shadow in it.

3.1 Color

In the original algorithm, the color comes from the image only rendered with ambient light on and the images rendered with only one light source on. The after one also provides the texture and specular light information. However, rendering the images with every light source on could be a pain in the neck. To implement these pictures, we need to change the setting of the light several times (number of light source times) and render one image every time. Also, to render these images, we should doing a lot other operations, for example copying the diffuse color into ambient and keeping the specular light information.

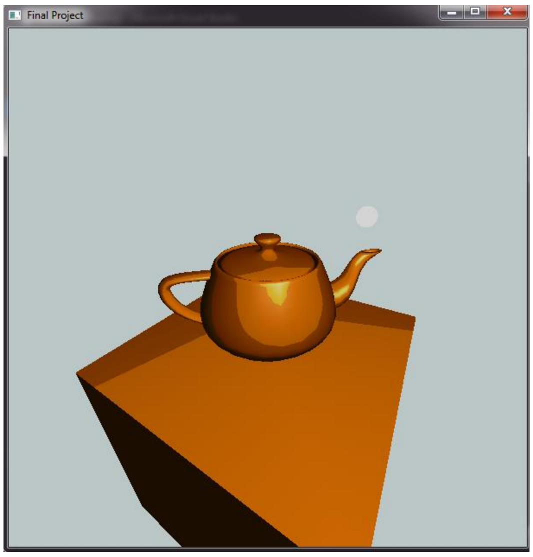

Figure 4: The same teapot with light source on right, after implemented the uniform color changing introduced in this section.

The method this report using is far easier than the original method.

By using only a few lines of code, the OpenGL will render the 3D description into a normal photo-realism image. By manipulating the existing photo-realism image, setting a range of the value of RGB colors, we will have the result that the object be filled up with several uniformly color area, just like images showed above.

To implement the uniformly colored area, this report is using the method to separate the range of RGB color into three parts, which is from 0 to 0.33;0.34 to 0.66;and 0.67 to 1 in the normalized condition. By setting the pixel color information in the different range with different constant value plus its original value divided by a constant number, we will result in several uniformly colored area. The detailed equation is showed below.

$$\text{color} = \begin{cases} \frac{color}{1.5} & if\ color < 0.33 \\ 0.33 + \frac{color-0.33}{2} & if\ 0.34 \leq color \leq 0.66 \\ 0.66 + \frac{color-0.66}{2} & if\ 0.67 \leq color \end{cases}$$

3.2 Outline

In the original algorithm, the outline is divided into two different types, the silhouette and the crease edge. However, implementing crease edges requires the normal buffer calculating. I don’t know whether I can read the normal from the glReadPixel or other similar formula. But to calculating the normal buffer we need calculate the normal of every point related to every pixel in the image. This relates to a bunch of work and also hard calculation formulas. Even though I understand the whole process, it’s still a pain to implement the whole process. So I made a hard decision to give up calculating the normal buffer which is stated in this report.

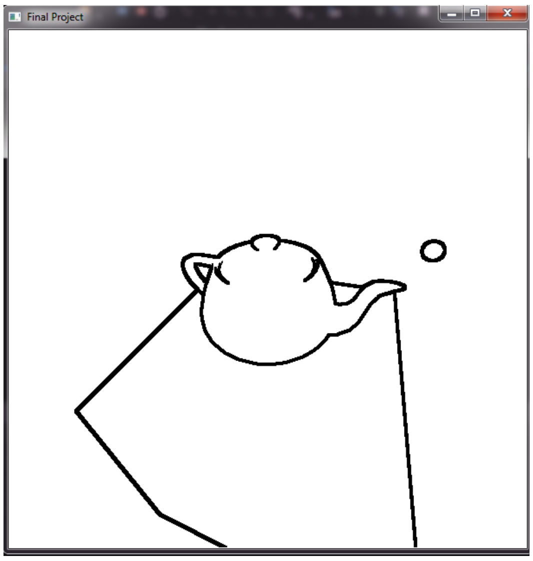

Figure 5: The frame of out teapot and cube by using only silhouette method kp=0.0007 edgelim=0.7. No crease edge because of missing normal buffer. But it still looks good.

So in implementation, silhouette is the only method of calculating outlines used in this report. The Figure 5 gives us the flavor of only using silhouette outline method without giving the color and texture to the teapot. And in the following paragraph, I will set different value of edge-limit-value and k_p in previous equation, it also can achieve the crease edges, even though the teapot contains so many unnecessary outlines.

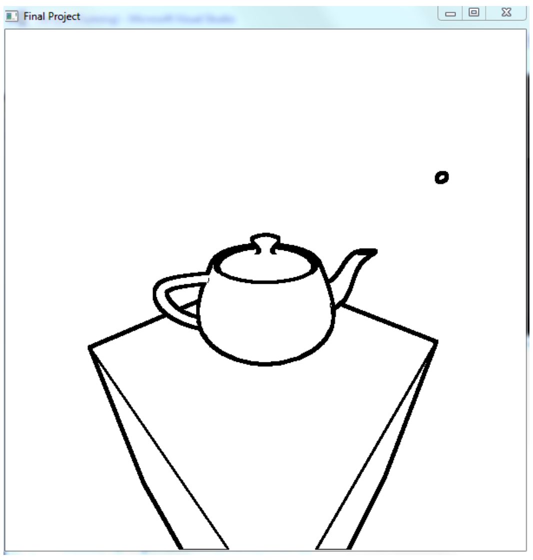

Figure 6: The frame of out teapot and cube by using only silhouette method kp=0.0003 edgelim=0.2. Crease edge is also detected because of the low edge limitation.

To rendering the cartoon-looking image, we only need a simple outline such as Figure 6. However, during the implementation, it is found that some outlines looked like sketch is also pretty. By comparing the images using different edge-limit-value and k_p value, it is easy to find out that the lower k_p is, the more silhouettes will be detected [Decaudin 1996]. Also, the lower edge-limit-value is, the wider outline is.

Figure 7 shows a lot of scratches and unnecessary edges which has k_p=0.00005 and edge limitation= 0.7. After practicing by many times, I will use k_p=0.0003 and edge limitation= 0.2 as our default value because it gives both the outline edge and the crease edges.

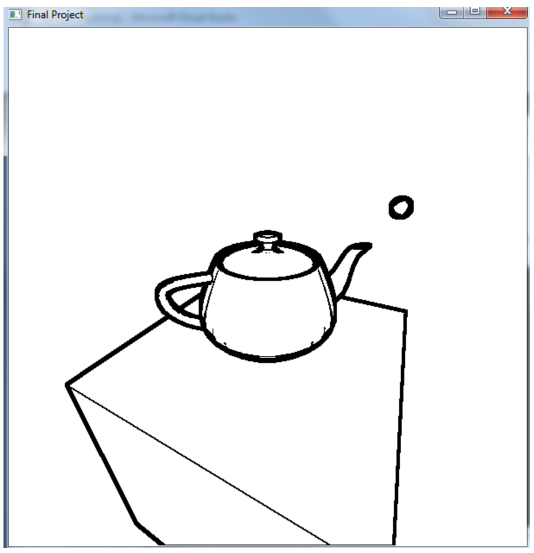

Figure 7: The frame of out teapot and cube by using only silhouette method kp=0.00005 edgelim=0.7. More unnecessary edges are detected.

3.3 Shadow

The original algorithm introduced in this report covers two types of shadows. One type is backface shadows and another type is projected shadows. In the program, I didn't implement the shadow successfully. But I still did some try on attempting to make it happen.

Firstly, I want to generate the texture information from the light positon and collect all the position information of the object in the image. By multiply a transformer matrix, we can map all the points from 2-D image to 3-D object. And we do the same thing from the camera position. If from both the two view point, we can see the same coordinate, I will light on the pixel otherwise we do nothing. The hard part here is to get the coordinate of the object which is in our case is the teapot. I don't know how to collect the 3-D position information.

The second way to do shadow is to use shader. The idea is basically to map all the 2-D position to the 3-D object position. Then I will calculate the distance from the 3-D position to the light source. After that, use the distance to compare with the zbuffer viewed from the light source. If the distance is larger than zbuffer, that means this position is not accessible to the light which shouldn't be light on, vice versa.

3.4 other implement

In the program, I try to use double pointer instead of double array to store the ambient and zbuffer information. The benefits of using pointer is that I can malloc specific amount of memory when I reshape the windows to a different size other than 600*600 because once we claim an array it should has fixed size. But in real, the glReadPixels function will give memory violating problem if I want to use double pointer to replace the double array. The reason of failure is not clear and verified. So I have to quit this idea in the end.

4 Conclusion

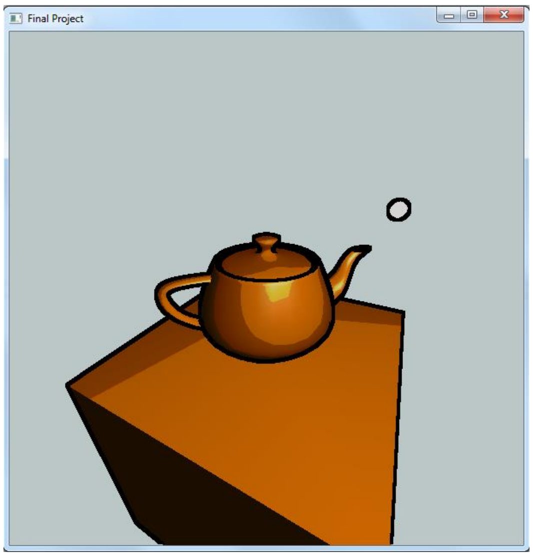

Figure 8: The final result of implementation.

This report mainly does two things. It introduces an algorithm published by Philippe Decaudin in 1996 [Decaudin 1996], which can render a 3D scene description into cartoon-like images.

This report fully explains the algorithm and also comes up with an original implementation to do almost the same thing. The implementation introduced in this report is doing the similar thing as the original algorithm does, but it create the result in an easier way, which simplify the original algorithm. The implementation mentioned in this report not only render the cartoon-looking image with constant size outline, uniformly colored area, and shadows, but also provides a simple way to create a sketch-like outline which could be treated individually. The final result of the implementation is not perfect.

The shadow it draws is still in a wrong form. However, it also mentions the reason why this problem happens, so more future work is needed.

5 Further work

The further work of this implementation is obviously. Correcting the projected shadow will be the first thing that needs to be added.

There also exists some other jobs that could be added into this implementation. As the original algorithm shows, it is necessary for the rendering to handle both backface and projected shadows if it is used to render a 3D animation has both static and moving objects.

Also, adding the normal buffer into the implementation will bring more details into the image, so it could also be done. The goal is not to completely follow the original algorithm but finding a more simple way to implement the cartoon-looking rendering.

References

DECAUDIN, P. 1996. Cartoon looking rendering of 3D scenes.
Research Report 2919, INRIA, June.