Non-photorealistic rendering and model viewer

Alejo Antonio Salvador

May 2, 2022

1 Introduction

There are several motivations for generating images that do not pretend to simulate reality. These reasons can range from aesthetics to the increased visual clarity obtained through outlines and a simpler shading. In this project I will analyze some ways to tackle this problem and describe the model viewer that is used to show these various techniques.

2 Model viewer with realistic lighting

The model viewer used as a basis for this project has a isometric projections and the rotations are expressed in Euler Angles. There is diffuse, specular, ambient and fresnel lighting.
For all of these calculations the normal used was an interpolation between the normal vectors of all the vertices of the current triangle.
Diffuse lighting is calculated as the product of the surface color and the dot product between the normal vector and the light direction.
For the specular lighting the Blinn-Phong model was used. It uses the halfway vector between the viewer and light-source vectors and it uses it to obtain the dot product between it and the normal vector. This result is then elevated to an exponent depending on the intensity of the specular reflection. This value is finally multiplied by the color of the refraction (white in this case).
The ambient light was just set as 0.3 times the diffuse color of the material.
The last component of this lighting model was the fresnel light (rim lighting). It is used to recreate the effect of light filtering through the contour of the object. It is really useful to accentuate the borders of the model when trying to create a non-realistic renderer. Its intensity is obtained as 1.0 minus the dot product of the viewer vector and the normal vector. This value was then multiplied by the diffuse component to reduce its intensity if light is not hitting that area and make sure that shadowed areas are not lighted by it.
A final but vital component of this lighting model are dynamic shadows. To achieve them I rendered the model again from the light's perspective. I used an orthographic projection since it is better to represent the parallel rays of directional light. In this way I will get a shadow map in which I will have the depth of the closest part to the light source in each pixel. This will be our shadow map which I will use in the lighting process when I render from the viewer perspective to shadow the correct areas. I will shadow any pixel where its depth from light perspective is higher than the depth of its corespondent pixel in the shadow map.
After implementing this the shadows can already be seen. However, there is still an annoying artifact called shadow acne (figure 1) that happens because there is a difference between the depth of the pixel in the shadow map and the depth obtained in the fragment shader in the final calculation. This happens because many vertexes can map to the same pixel causing a lose of precision. This gets clearly worse as the light angle become more steep since more vertexes will map to the same pixel. An easy solution for this problem would be using a bias to make sure that areas where the difference in depth is small are not shadowed. I will use the following formula $max(0.01 * (1.0 - dot(normalVector, lightDirection)), 0.001)$ to make sure that the bias gets bigger as the light angle get steeper.
Another important component of the rendering process is HDR tone mapping. As we can see the addition of all these light components can make the value of a light channel bigger than 1 and therefore I would lose color information. I don't care about oversaturating the area through specular and fresnel ligthing since saturating the channel is part of the intended look. However, I don't want the addition of the diffuse and ambient to exceed 1.0 in any color channel. To achieve this I will use Reinhard tone mapping applying the following operation on each channel: $\frac{HDRcolor(1+\frac{HDRcolor}{whiteColor^2})}{1+HDRcolor}$ where HDRcolor is the color before mapping and whiteColor is the treshold I want to set as white. An important aspect to mention is the fact that the mapping was done directly on the channel instead of doing it by using luminance. The reason for doing that is that it prevents every channel from being saturated. However, it also removes saturation from the brightest areas modifying the color slightly but I consider it a feature more than a problem. This is discussed in more detail in the case of filmic tone mapping in Jhon Hable blog at `http://filmicworlds.com/blog/filmic-tonemapping-with-piecewise-power-curves/`
One last step would be gamma correction. It is very important because all these calculations assume that color is lineal but it is actually not in a monitor. Because of this, most assets are made with the gamma of the artist's monitor

(usually around 2.2). Therefore I need linearize the textures first for calculations. After all work is done with the linearized colors I need to map the colors again to the gamma of the user's monitor. The formula to map the colors to the gamma of the monitor is $pow(linearColor, vec4(1.0/2.2))$
Let see how it looks before using various techniques to achieve the cartoon look (figure 2).

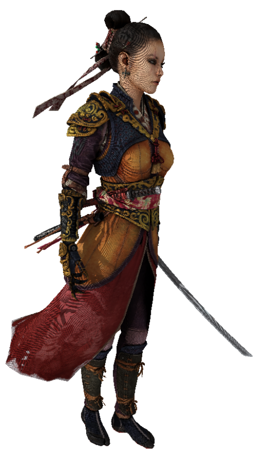
Figure 1: Shadow acne

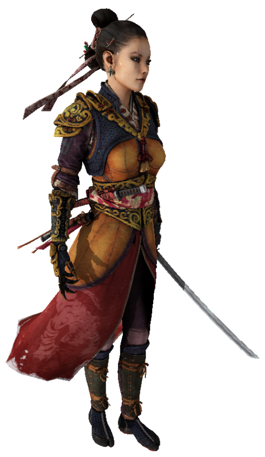
Figure 2: A model without using the non realistic shading techniques

3 Non-realistic rendering

3.1 Cel-shading

Cel-shading is the act of tresholding the color transition in such a way that there is a limited number of lighting levels on the image. Since this is the way in which most pictures are hand drawn, it is really useful to achieve a feeling of a flat hand drawn image. It is achieved though setting a threshold for every type of lighting previously mentioned in such a way that the area is lit with a given intensity if the original intensity is greater than a given value. I made the number of splits used in cel-shading for diffuse and specular lighting configurable. Ambient light was just set as the color for areas not lighted and a base to which diffuse and specular lighting is added.
Diffuse was implemented trough tresholding the cosine of the angle between the normal vector and the light source since this is what determines the intensity of light. By tresholding this angle instead of the resulting diffuse color this makes sure that the lighting level of the area in cel-shading only depends on the original lighting level and not the color of the texture on the surface.
Specular lighting was implemented in a similar way but the angle used is the one between the normal vector and the halfway vector between the viewer and light source vectors.
Finally rim (Fresnel) light intensity was calculated by just tresholding it before multiplying it with the diffuse lighting since that is the moment where the value stored is its intensity.

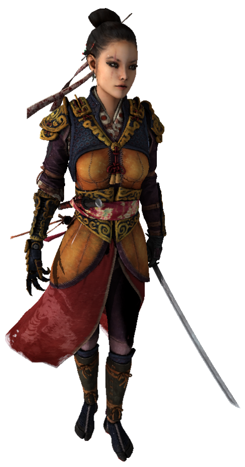
Figure 3: Without cel-shading

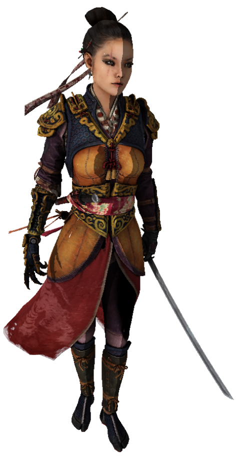
Figure 4: With cel-shading

Comparasion of cel-shading character

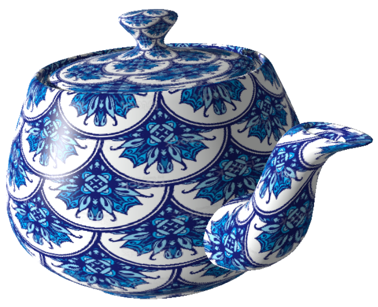
Figure 5: Without cel-shading

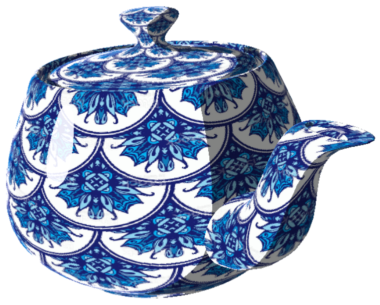
Figure 6: With cel-shading

Comparasion of cel-shading for specular light

3.2 Drawing the outline

There are many ways to draw the outline of a 3d model. They can be either achieved in object-space or in screen space. One of the most used object space methods is the inverted hull one but it has the problem that it is not too effective for models not specifically made for it since it does not draw some interior lines and the width of the lines are inconsistent. Other object space solutions I found didn't achieve the look I was searching for. Therefore, I decided to implement 2 different screen space solutions.
To implement these 2 solutions I need to get a depth and normal vector buffers during the rendering of the image to utilize later on the calculations. It is very important to mention than I need to linearize the depth because the depth stored in the buffer is not the real one but instead the one that is obtained through the matrix projection. To do so, I first to need to remap the depth to the range [-1,1] since webGL automatically maps it to the range [0,1] when passing the depth from the vertex shader to the fragment shader. With that done we apply the formula
$linearizedDepth = \frac{2*nearPlane*farPlane}{farPlane+nearPlane-depth*(farPlane-nearPlane)}$
With that done lets see how to use that information to detect the borders. The first one consisted on identifying the areas were the depth change suddenly. However, this solution had the problem that it was too sensible in areas were the objects surface are close to parallel to the view vector. This is because if that happens there will always be a huge variation in depth between two adjacent pixels.
To fix this problem I implemented another solution that took into account both normal vectors, depth and the position to find the planes tangent to the surface. In this way a pixel will be part of the outline if the distance between the position of the vertex represented there and the planes that correspond to the adjacent pixels is greater than a fixed threshold value.
In the two methods I used the discrete Laplace operator since it achieved a look extremely similar to the sobel operator with one less pass required. One problem usually present for the Laplace operator is that it is extremely sensible to noise (this is usually fixed through Gaussian sound) but since I'm working with depth and normal vectors the curves should be relatively smooth except for the cases where I effectively want to draw an outline.
Both of this methods present some problems in low-poly models since there is a relatively steep difference between the normal vectors and depth of 2 adjacent vertex. However, this can be easily fixed by setting the correct threshold depending on the size of the polygons in the model.
There are some cases where thicker outlines are wanted. The solution used to achieve this is just saving the outline in a different buffer by itself and then placing in each pixel the darkest color between all of the neighbours. Therefore each pass of this process increase the width of the outlines by two.
Finally, it is important to mention that the method relying on normal vectors allows me to set a smaller threshold without seeing annoying artifacts since it is less sensible to the viewing angle. The difference between both of this methods can be seen in figures 4, 5, 6 and 7.

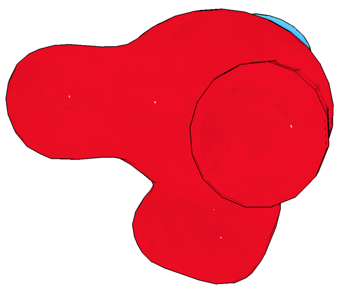
Figure 7: Depth difference

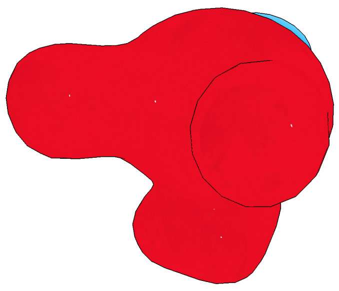
Figure 8: Tangent Planes distance

Huge difference in the artifacts shown in the leg of the model

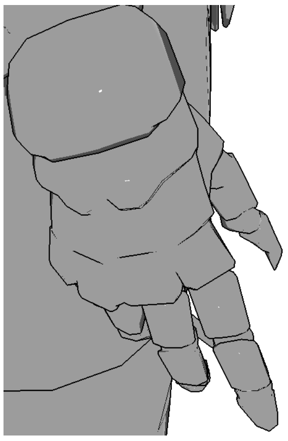
Figure 9: Depth difference

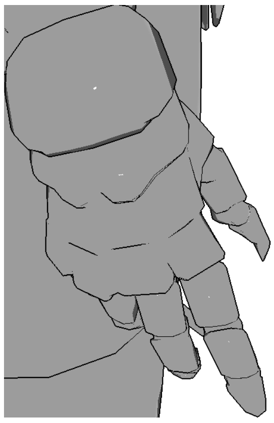
Figure 10: Tangent Planes distance

Level of precision for small details

3.3 Applying a paper texture

One thing useful to convey the feeling of this being a hand drawn image is using a grey-scale texture to convey the inherent rugosity of canvas paper. This is achieved though an image space transformation. To do so the color of the pixels corresponding to the 3d model are multiplied by the color of the paper texture by using alpha multiplication and setting the alpha coefficient with a paper transparency parameter. In areas where there are not pixels corresponding to the model the paper texture is shown directly.
The image shown in figure 11 will not have a color texture applied to make the paper texture effect more noticeable

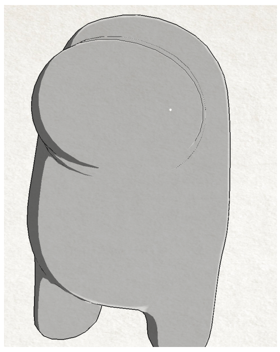
Figure 11: Paper texture applied

3.4 Simulating pencil stroke

There are many ways to simulate the strokes of pencils or pens. One way to do so is through a sequence of mip-mapped hatch images corresponding to different tones, collectively called a tonal art map (SIGGRAPH2001). However I decided against using it since I wanted to go for a look closer to the one used in TV animation or some game like Valkyria Chronicles characterized by using parallel lines in shadowed areas. I also used another kind of parallel lines on the rest of the surface to give it look more like a sketch. Both of this techniques were used in the game previously mentioned. However, I made one change from the game since the game presents problem called "shower-door effect" which is the illusion that the user is viewing the scene through a sheet of semi-transmissive glass in which the strokes are embedded. To fix this problem I decided to move the background image together with the translation of the object. I did not care about rotations since this problem is not as evident in them. It is important to take into account the depth of the object since the displacement in screen space does not correlate directly with the displacement in object space in the perspective projection. Instead it depends on the depth of that specific part of the object. The formula used to do this correction is $trans/linearDepth * 0.5$ where trans is trans is the XY translation and linearDepth is the depth linearized.
The problem with this correction is that, as you translate the object, new areas on the side of the object that were not previously seen can now be seen because of perspective. For this reason, this areas do not adjust correctly with the formula used and therefore deform lines slightly and they stop being straight. However, I consider this problem to be less noticeable than the shower-door effect.
Let's see how the model looks with the pencil stroked applied to it. The first image will show how the model looks with this effect applied without translation. The second one will show the problem present when the model is translated enough to notice the deformation of the stroke lines.

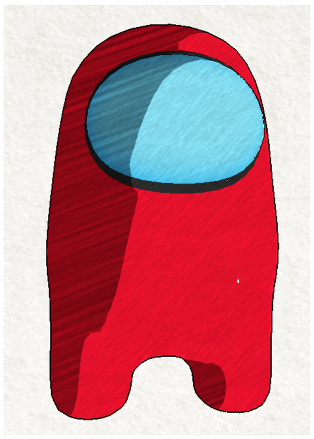
Figure 12: without deformation of translation

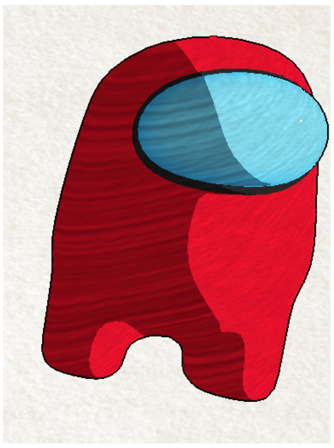
Figure 13: with translation deformation

deformation of pencil lines when translating

4 Resulting render and future work

When combining all the different rendering stages used for achieving this sketch book look the resulting image is very different than before and it conveys the look of something hand drawn way better as it can be seen in the next images.

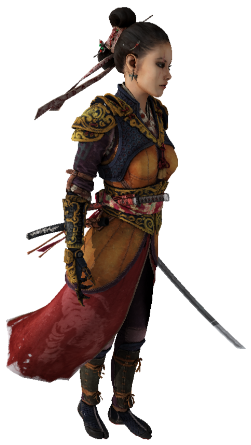
Figure 14: basic "realistic" rendering

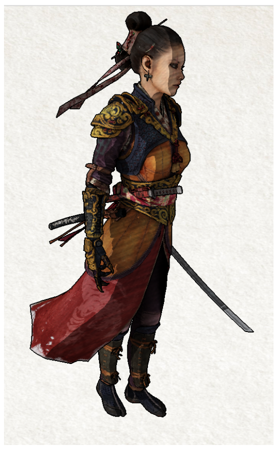
Figure 15: non-realistic rendering

Some things to improve in this work would be fixing the deformation of the lines because of translation or implementing the cross hatching presented in SIGGRAPH2001 or NPAR2002. It would also be nice to implement a model loader able to load models with different materials since this change would improve greatly the images presented here.

References

[1] Symposium on Non-Photorealistic Animation and Rendering (NPAR) 2002, 53-58.

[2] Common Techniques to Improve Shadow Depth Maps. `https://docs.microsoft.com/en-us/windows/win32/dxtecharts/common-techniques-to-improve-shadow-depth-maps`

[3] Filmic Tonemapping with Piecewise Power Curves. John Hable `http://filmicworlds.com/blog/filmic-tonemapping-with-piecewise-power-curves/`

[4] Implementing a “sketch” style of rendering in webGL. `https://medium.com/cbrebuild/implementing-a-sketch-style-of-rendering-in-webgl-d6f0e4685a17`

[5] Emil Praun, Hugues Hoppe, Matthew Webb, Adam Finkelstein. ACM SIGGRAPH 2001 Proceedings, 581-586.