COSC 4306

Final Report

Mathieu Carriere
Bradley McFadden
Joy McGibbon

April 4, 2022

Abstract

Non-photorealistic rendering (NPR) is a combination of computer graphics and artistic techniques used in a variety of applications to emphasize information from a photorealistic rendering or present it in a different style. This can be done either through artistic techniques that create a particular mood, or by emphasizing details of the image/model, or both.

In this project, we investigate techniques used for NPR, and apply them to our own renderings of 3D models. In particular, we attempt to reproduce the style of 2D animation with 3D models by applying cel shading, lighting effects, sampling textures, contours, and suggestive contours. For our implementation, we use the OpenGL pipeline to apply a rendering process with several stages, in order to produce our final renderings of our models.

Introduction

In the field of computer graphics, non-photorealistic rendering (NPR) is a somewhat circular term that refers to the use of rendering techniques to achieve a non-photorealistic effect. There are several situations in which a non-photorealistic rendering is advantageous.

The first field is technical illustration, as described by [2]. Technical illustrations are commonly found in manuals for products that need to be disassembled for repairs, or in training documentation for a particular device. As such, the depictions of the device are static images, and it is often the goal of the illustrators to convey as much information as possible with these static images. Often, realistic shading is omitted, and image are drawn with an isometric perspective, so the scale and shape of geometry is easier to read. In addition, outlines are commonly added. If a light source is used, sometimes a technique called "rim lighting" is applied that highlights edges of faces that is pointed away from a light source.

Another field in which NPR is commonly used, is in interactive media. Games and animation that use NPR often apply it as a design choice. For one reason or another, a realistic rendering may be seen as too visually busy, or does not provide the user with as much visual information as the designers wish. It may also be that designers wish to imitate a style closer to that of cartoons, for one reason or another. Cartoons are generally drawn in a wide variety of styles, but most have the following characteristics: Foreground objects such as characters are drawn with outlines. Most foreground objects are drawn with a single colour, and shading is shown with a darker version of this colour. This technique is referred to as cel shading, and is very common in many stylized modern games, including examples such as Sable.

The goal behind this project is to write a program that applies common NPR effects to 3D models, to create our own NPR rendering of the models. In particular, we will implement cel shading, object contours, and rim highlights to achieve our final NPR rendering of our models.

Prior and related work

Gooch et al.'s work in technical illustration demonstrates many common non-photorealistic rendering effects [2]. The authors made the observation that there was a disconnect between the style of artists and illustrators, and the style created by computer graphics programmers. Where computer graphics artists were concerned with techniques to create a realistic image, illustrators of technical manuals opt fir a style that emphasizes the geometry of an object, while removing extraenous detail. The authors note that technical illustration needs this characteristic, as the viewer does not have the ability to move around a printed image to get more information. To produce a shaded technical rendering of an object, Gooch et al. render objects with black edge lines, no shadowing, intensities far from black or white, colour indicated by surface normal, and a single light source that provides white highlights (also known as rim lighting). It is interesting to note that the authors provide a method for approximating their model using phong shading with negative light colours.

The work of Gooch is very similar to [7]. Saito and Takahashi also sought to create more clearly comprehensible renderings of 3D shapes, in particular renderings that maintained their clarity after multiples passes through a copy machine. Their work focuses on image processing techniques that occur after an existing rendering has been produced. The authors combine image enhancement techniques together to produce more clear images. They find edges, contours, discontinuities, and curved hatching together to produce images of 3D objects with enhanced edges, and more clear shape from curved hatching. Additionally, they demonstrate their results on topographical data, to produce easy to read topographical maps of terrain.

The work of DeCarlo et al. starts from a similar question [4], how can the shape of a 3D object be conveyed with just lines? The research describes two techniques to display an object's shape, namely contours and suggestive contours. Contours describe an area of an image where a surface turns away from the viewer, becoming invisible. Similarly, suggestive contours are like contours, but describe areas where the surface of an object turns away from viewer, yet remains visible. Additionally, Decarlo et al. provide mathematical definitions and algorithms that detail how contours and suggestive contours can be produced on a model.

Decaudin et al. draw upon several techniques to render 3D scenes in a cartoon style [3]. Their work presents a rendering algorithm that proceeds in four stages. Firstly, they render the scene with ambient lighting. Next, they find the outlines of each object in the scene. For each light source, they render the scene as illuminated by it, and then find the project shadows from other objects. Finally, they combine the steps to produce a cartoon style image. Notably, their ambient lighting stage renders each object with a single uniform colour, and the shadows of the scene are also a single colour, which seems like a form of cel shading.

[2] Work by Mitchell et al. in [6] describes how a modified Phong shading model can be used to increase visual clarity of rendered models in a first-person actions game like Team Fortress 2. One technique employed is a transformation on the dot product of vector n and I in the diffuse lighting equation. The transformation prevents models from losing shape information on faces opposite a particular light source. The authors transform the Phong model to add a function that takes the scale produced by diffuse lighting, and breaks it into three regions, a dark gray end ground, a light gray start zone, and a middle ground with a slight red component. This is done from observations that artists tended to favour use light grays and dark grays instead of black and white, and preferred to mix in warm colours to their mid tones. The authors also add a dedicated rim lighting term to their Phong model in order to make upward facing surfaces more likely to be rim shaded.

Description

Lighting

The Phong lighting model is used, with specular, diffuse, and ambient light. Specular light concentrates light in one direction, providing highlights, diffuse light scatters light equally, providing shadows, and ambient light provides an equal amount of light to all parts of the surface from every direction.

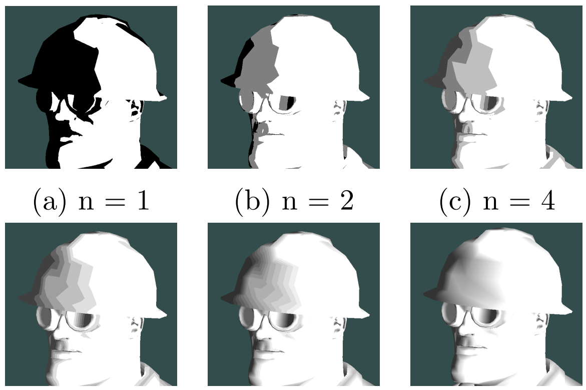
Rim lighting is also added to provide additional highlights and increase the cartoon appearance of the render. It highlights the edge of the object, and as a result improves the rendering from some angles, but appears incorrect from others.

Lighting is implemented in the fragment shader, rather than in the vertex shader, decreasing efficiency as it must be calculated for every fragment rather than the relatively small number of vertices, but providing more precise results.

Cel shading

Cel shading refers to an NPR technique where to better simulate a cartoon art style, the tones of an image are broken down into bands. The colour palette of a scene is reduced to a handful of colours. The technique simulates how an artist might draw an image, where only a few shades are used, and few separate colours are used.

To perform cel shading, a number of discrete bands n must be chosen. Higher values for n produce more bands of colour, which leads to less harsh contrast in the image, but fewer bands create more contrast, and therefore a more dramatic effect. To get the final colour of a pixel, the RGB values are divided by n+1, which returns the index of the band. Then, the index is multiplied by the size of the band, which produces the final colour. Examples of cel shading for a particular model are shown in Figure 1.

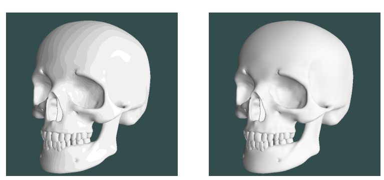


(a) n = 1 (b) n = 2 (c) n = 4

(d) n = 8 (e) n = 16 (f) None

Figure 1: Comparison of number of cel shading bands

One can notice sharp edges on the model. This is implementation specific, and happens because the colour passed to the fragment shader is interpolated between vertices by OpenGL. Normally, this creates a smooth effect and saves processing power, but when performing cel shading, it can destroy the effect, so it is turned off. Unfortunately, doing so creates jagged lines in models with a low polygon count, but the effect is not as noticeable for higher polygon count models, such as in Figure 2.



(a) n = 18 (b) None

Figure 2: Cel shading with a high polygon count model

Suggestive contours

Outlines

System overview

The described features were implemented into a real-time rendering system. The system itself was written in OpenGL3. Specifically, we used LWJGL's implementation of OpenGL. We took advantage of the vertex and fragment shaders that OpenGL3 requires to be used. The use of vertex and fragment shaders allows us to implement most algorithms on the GPU, which saves CPU cycles and increases the performance of the application.

Essentially, our system is a real-time renderer for 3D models that handles lighting, arbitrary textures, and allows shading effects to be toggled on or off for comparison. A camera is supported so that the model can be examined from a variety of viewing angles. A high-level overview of the system is presented in Figure 3.

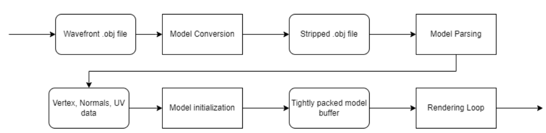


Figure 3: An overview of stages in the system

The first stage of our system takes in .obj files and reformats parts of the file in order to make it easier to parse. This was implemented as a separate Python program, since it needs only be done once per model. The second stage of the system parses a .obj file to produce a prototype of a Model that our system is later able to render. In doing so, the vertices, vertex normals, and texture coordinates from the file are packed into the prototype model. Additionally, the indices that use this data to define faces of the model are also read. At this stage, textures are also read in for the model from the .mtl file referenced by the .obj file. One caveat is that imported models should be triangulated, as our system has no procedure in place to triangulate polygons. The next stage of the system packs the Model data into a format that the vertex shader can understand. An strided array of nine floating point values per vertex is created. The first three values each stride represent the vertex data, the next three are the vertex normal, and the final three contain texture coordinates. These values are passed as attributes to the vertex shader. The buffer only ever has to be calculated once per model, so this stage does not repeat. Every frame, each model is rendered. The render proceeds by passing light and camera parameters to the vertex shader, along with the vertex, vertex normal and texture coordinates from the previous stage. Two rendering stages happen for our program, the outline render loop, and the model render loop. These stages are shown visually in Figure 4.

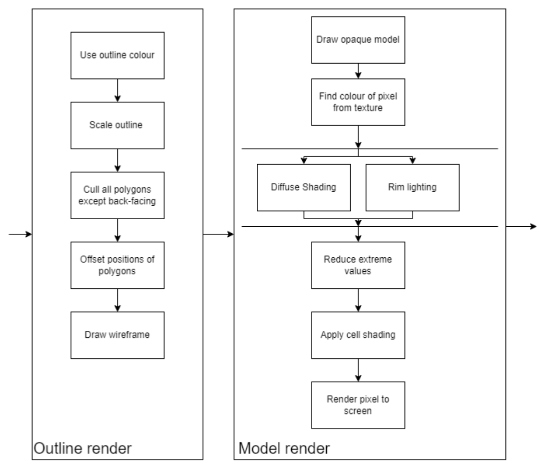


Figure 4: An overview of stages in the rendering step

The outline render loop draws only the back faces of the model in a black wireframe. Edges are drawn at a size of 5 pixels instead of the default 1 pixel, so they stick out from the model. Furthermore, the polygons of the model are draw slightly offset to further accentuate the outlines. One issue with this approach to drawing outlines is that the size of the outlines is constant no matter the distance to the model, so far away objects end up with much more noticeable outlines than closer models.

The model render loop handles the lighting and the cel shading of the object, and the its work is done in the vertex and frament shaders. The steps are detailed below.

The vertex shader is the entry point for the OpenGL pipeline, so its main role is to forward the parameters that the fragment shader uses, and to specify the position of each vertex sent to it. Additionally, the vertex shader calculates the lighting normal.

The fragment shader does the bulk of the work of our application. In the fragment shader, two main procedures happen. First, the colour of the lighting for the fragment is found. This is a multi-step process that involves applying diffuse lighting, rim lighting, and reducing extreme light and dark values in order to preserve the colour of the model. Afterward, cel shading is applied to discretize the colours of the model.

Figure 5 shows the rendering process at three stages, drawing the outline, drawing the inner model, and the combined final result. One drawback to this approach of drawing outlines is that the rendering pipeline is used twice per model. However, it may be not quite as intensive since only back faces are drawn, and the wireframe of the model is not filled.

(a) Outline (b) Unshaded (c) Shading (d) Final

Figure 5: Comparison of output at each rendering stage

Experimental results

Conclusions

Bibliography

[1] Angel, E., Shreiner, D. (2012). *Interactive Computer Graphics: A top-down approach using OpenGL*. Pearson.

[2] Gooch, A., Gooch, B., Shirley, P., Cohen, E. (1998). *A non-photorealistic lighting model for automatic technical illustration*. Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, 447–452.

[3] Decaudin, P. (1996). *Cartoon-looking rendering of 3D scenes*

[4] DeCarlo, D., Finkelstein, A., Rusinkiewicz, S., Santella, A. (2003). *Suggestive contours for conveying shape*. In ACM SIGGRAPH 2003 Papers (pp. 848–855).

[5] Graf, C. (2002). *NPR-Shading*.

[6] Mitchell, J., Francke, M., Eng, D. (2007). *Illustrative rendering in Team Fortress 2*. Proceedings of the 5th International Symposium on Non-Photorealistic Animation and Rendering, 71–76.

[7] Saito, T., Takahashi, T. (1990). *Comprehensible rendering of 3-D shapes*. Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques, 197–206. https://doi.org/10.1145/97879.97901

[8] Phong, B. T. (1975). *Illumination for computer generated pictures*. Communications of the ACM, 18(6), 311–317.