1 INTRODUCTION

Non-Photorealistic Rendering (NPR) aims for expressive, stylized rendering over realistic rendering. It aims to both communicate information using images and represent images in new styles outside of the traditional "life-like" style found in Computer Graphics [4]. In popular media today, NPR techniques have captured a lot of attention through movies and video games. There’s multiple sides to the appeal: for viewers, it’s a refreshing alternative from traditional media, which aims to capture reality as much as possible. For artists, it allows for an easier time making stylized artwork: artists get the flexibility of 3D rendering but with a similar expressiveness to 2D styles.

Much of the research regarding Non-Photorealistic rendering attempts to make 3D stylized effects as accurate as their 3D counterparts. This has lead to the creation of numerous techniques for NPR, including stylized shading and edge drawing.

Many 3D applications now have Non-Photorealistic rendering capabilities, though they’re often secondary to the photorealistic rendering facilities of that software.

The goal for this project was to take some of the NPR techniques outlined in prior work and implement them in a real-time rendering system, in which you could import any mesh and “try on” a variety of stylized looks for its rendering.

2 BACKGROUND

The two aspects of NPR that this project focused on were stylized shading and object edge outlining. Prior work in these areas is outlined by Sayeed and Howard [4].

2.1 Stylized Shading


2.1.1 Tone Shading. Tone shading is a method of mapping light intensity across a surface to arbitrary detailing both tone shading [2]. The technique was first focused on creating a “warm/cool” effect when rendering with an object [2]. To render a matte object with no specular reflection with an infinitely far light source, we get the following equation [2]:

\[ I = \left( \frac{1 + \hat{l} \cdot \hat{n}}{2} \right) k_{cool} + \left( 1 - \frac{1 + \hat{l} \cdot \hat{n}}{2} \right) k_{warm} \]  

where \( \hat{l} \) and \( \hat{n} \) are the light and normal directions and their dot product is used to interpolate linearly between a cool and warm color. Here, the warm color is shown when an object face is oriented away from the light. Thus, \( k_{cool} \) can be thought of as the “highlight” color, while \( k_{warm} \) is thought of as the shadow color.

2.1.2 Cartoon Shading. Cartoon shading (also called “toon shading”) is a method of rendering which involves creating wide bands of constant color around an object, removing detail [4].

Similar to tone shading, cartoon shading can also be thought of as interpolation between two colors, though the interpolation is done through a step function instead of a linear function. There are many algorithms for implementing toon shading, though Lake et al. [3] perform diffuse toon shading by calculating a shadow and illuminated color and choosing between them based on the value of \( \hat{l} \cdot \hat{n} \).

2.2 Object Outlining

In NPR, objects can either be outlined using image-based methods or geometry-based methods. Image-based methods treat outlining more as a post-processing effect after the initial scene image is rendered, while geometry-based methods use existing object edge geometry to create outlines. This project focuses on using geometry-based methods for object outlining.

In doing geometry-based object outlining, there are three main types of edges that are used for rendering outlines: silhouette edges, crease edges, and border edges [3, 4].

2.2.1 Silhouette Edges. Silhouette edges mark the general bounding outline of an object. They occur whenever a front and back-facing face meet [3, 4]. For an edge that borders two faces, its silhouette status can be calculated with the following equation [3]:

\[ (\text{faceNormal}_1 \cdot \text{eyeVect}) \ast (\text{faceNormal}_2 \ast \text{eyeVect}) \leq 0 \]  (2)

Here, the \( \text{eyeVect} \) is eye vector of the camera being used to render the scene.

2.2.2 Crease Edges. Crease edges mark sharp changes in the curvature of an object, and are generally used as “accent strokes” that highlight the form of an object along with silhouette edges [3, 4].

Crease edges are rendered when the angle between two neighboring faces of an edge surpasses a threshold. The angle between two faces can be taken by calculating the angle between their normal vectors.

2.2.3 Border Edges. Border edges mark the faces of a mesh that lie on the edge of a single polygon or are shared between two polygons with different materials [3]. They serve as additional accent edges along with crease and silhouette edges.
3 APPROACH

The Non-Photorealistic renderer was built in C++ using GLOO, an object-oriented library for interacting with OpenGL that provided by MIT's 6.4400 Course Staff. The main components of the render were doing stylized shading and outlining. The implementation details are for both aspects are described below.

3.1 Shading

For this project, only one shader was applied uniformly to all object geometry at a time to allow for more coherent scene rendering. Both tone shading and toon shading were implemented with OpenGL fragment shaders. The general logic for both shaders was the same.

3.1.1 Tone Shading. For directional lights, calculating the final color outputted in tone shading involves a straightforward application of the tone mapping equation detailed earlier. Here, however, we replace the cool and warm colors with our desired shadow and illuminated colors for our mesh, giving us the following equation:

\[ I = (1 + \frac{\hat{l} \cdot \hat{n}}{2})k_{\text{illuminated}} + \left(1 - \frac{\hat{l} \cdot \hat{n}}{2}\right)k_{\text{shadow}} \]  

(3)

Here, \( k_{\text{illuminated}} \) and \( k_{\text{shadow}} \) are (r, g, b) vectors of size 3, and are stored as uniform properties of the shader. This was done to give an easy way to modify how a scene is shaded.

One challenge for point lights however, is to ensure that \( k_{\text{illuminated}} \) and \( k_{\text{shadow}} \) are still the only colors that are interpolated between darker the farther it is from a point light. One challenge for point lights however, is to ensure that \( k_{\text{illuminated}} \) and \( k_{\text{shadow}} \) are still the only colors that are interpolated between darker the farther it is from a point light. One challenge for point lights however, is to ensure that \( k_{\text{illuminated}} \) and \( k_{\text{shadow}} \) are still the only colors that are interpolated between darker the farther it is from a point light. One challenge for point lights however, is to ensure that \( k_{\text{illuminated}} \) and \( k_{\text{shadow}} \) are still the only colors that are interpolated between darker the farther it is from a point light. One challenge for point lights however, is to ensure that \( k_{\text{illuminated}} \) and \( k_{\text{shadow}} \) are still the only colors that are interpolated between darker the farther it is from a point light. One challenge for point lights however, is to ensure that \( k_{\text{illuminated}} \) and \( k_{\text{shadow}} \) are still the only colors that are interpolated between darker the farther it is from a point light. One challenge for point lights however, is to ensure that \( k_{\text{illuminated}} \) and \( k_{\text{shadow}} \) are still the only colors that are interpolated between darker the farther it is from a point light.

Using the calculated attenuation for a point light, tone shading can also be done for a point light using a modified version of Equation 3:

\[ I = (a + \frac{\hat{l} \cdot \hat{n}}{2})k_{\text{illuminated}} + \left(1 - a \frac{\hat{l} \cdot \hat{n}}{2}\right)k_{\text{shadow}} \]  

(4)

where \( a \) represents attenuation, which is a single number. \( a \) is calculated based on the distance from the light and an additional attenuation parameter, similarly to other shading models.

3.1.2 Toon Shading. Toon shading involves roughly the same process as tone shading, but instead uses a step function for interpolating between the shadow and light color:

\[ \text{def step}(\text{val}, \text{threshold}): \]

\[ \text{if} (\text{val} >= \text{threshold}), \text{return} 1 \]
\[ \text{if} (\text{val} < \text{threshold}), \text{return} 0 \]

The equation for rendering a directional light using toon shading is as follows:

\[ I = \left(\text{step}\left(\frac{\hat{l} \cdot \hat{n}}{2}, t\right)\right)k_{\text{illuminated}} + \left(1 - \text{step}\left(\frac{\hat{l} \cdot \hat{n}}{2}, t\right)\right)k_{\text{shadow}} \]  

(5)

where \( t \) is an arbitrary threshold between \([0, 1]\), usually set to \( t = 0.5 \) for balanced rendering. The equation for rendering a point light using toon shading follows a similar pattern:

\[ I = \left(\text{step}\left(\frac{\hat{\text{point}} \cdot \hat{n}}{2}, t\right)\right)k_{\text{illuminated}} + \left(1 - \text{step}\left(\frac{\hat{\text{point}} \cdot \hat{n}}{2}, t\right)\right)k_{\text{shadow}} \]  

(6)

3.2 Outlining

Outlining involves first getting a map from all edges of a mesh to their adjacent faces. This is done with the following algorithm:

\[ \text{def buildEdgeFaceMap}: \]

\[ \text{for each face:} \]
\[ \text{get the two edges belonging to it} \]
\[ \text{for each edge in the face edges:} \]
\[ \text{edge_map[edge].add(face)} \]

Note that we represent edges in C++ as a Pair between two vertex indices. This requires implementation of a custom hashing function for edges, since Pair’s don’t hash naturally. When implementing the hash function, the important thing is for two edges who have the same vertex indices but in different orders to have the same hash value.

After building our map from edges to their adjacent faces, edges can be tested for their relevant edge conditions. Each of the three conditions can be tested as follows:

\[ \text{def IsSilhouette(edge, camera):} \]
\[ \text{if edge only borders 1 face, return false} \]
\[ \text{get two faces neighboring edge, face1 and face2} \]
\[ \text{if dot(face1.normal, camera.direction) * dot(face2.normal, camera.direction) <= 0: return true} \]
\[ \text{else return false} \]

\[ \text{def IsCrease(edge, threshold):} \]
\[ \text{if edge only borders 1 face, return false} \]
\[ \text{get two faces neighboring edge, face1 and face2} \]
\[ \text{if angle between face1 and face2 >= threshold: return true} \]
\[ \text{else return false} \]

\[ \text{def IsBorder(edge, threshold):} \]
\[ \text{if edge only borders 1 face, return true} \]
\[ \text{else return false} \]

3.3 General Rendering Pipeline

The general rendering pipeline for our renderer is as follows:

1. Import Mesh
2. Process Mesh Edges
3. Mark Crease and Border edges
4. On Each Frame:
   a. Shade mesh using either tone or toon shading based on current light
   b. Calculate silhouette edges using current camera position and direction
   c. Draw Crease, Border, and Silhouette edges as lines in OpenGL

Note that we only need to compute the crease and border edges once as soon as we’ve imported our model.
3.4 Modularity
The renderer supports some modular features while rendering. The user has the option to toggle the light between a point and sun light, and also has the ability to turn off any of the Crease, Border, or Silhouette edges. The user can also switch between two predefined shaders: one for tone shading and another for tone mapping.

The user can also change the threshold for marking crease edges. This requires all crease edges to be recalculated after.

Furthermore, the user has the ability to change the background of the scene.

The user can also choose the model they want to render when starting the renderer.

4 RESULTS
Some results with the renderer can be seen in the paper’s figures.

5 CONCLUSION
This project is a Non-Photorealistic rendering system that uses some of the standard techniques found in the industry to provide users with the ability to quickly preview models in a non-traditional way.

There’s a few extensions that could be made to the renderer. One possible extension is to further increase the renderer’s modularity by giving users the ability to specify specific shading colors for each of the tone shader and toon shader, as well as specify the object’s edge colors.
The renderer could also be more detailed in how it renders, and account for the predefined material colors of an object when rendering, so there could be color variation in the rendered scene. Furthermore, the renderer could account for the specular components of materials when rendering, instead of only rendering everything in a diffuse way.

Furthermore, the renderer could also be faster. Currently, calculating silhouette lines is a large slowdown for the renderer, and this could be improved by making use of an edge buffer as detailed in Buchanan and Sousa [1].

REFERENCES


