Universidade do Minho
Mestrado em Engenharia Informática

Visualização em Tempo Real

Non-Photorealistic Rendering

Grupo 09
Inês Ferreira - PG53879
Marta Sá - PG54084

21 de junho de 2024

Conteúdo

1 Introduction

In the realm of computer graphics, non-photorealistic rendering (NPR) has emerged as a vibrant and diverse field, distinct from the traditional goal of photorealism. NPR techniques prioritize artistic expression and stylistic representation, often inspired by various forms of traditional art. Among the many approaches to NPR, *Toon Shading*, *Gooch Shading*, *Pixelation Shading*, and *Hatch Shading* where chosen for their unique contributions to the visual storytelling toolkit.

This essay explores the principles and applications of these four NPR techniques, highlighting their individual characteristics and the creative possibilities they unlock in the world of computer graphics.

2 Toon Shading

Toon Shading is a non-photorealistic rendering technique that mimics the aesthetic of hand-drawn animation and comic books. Unlike traditional shading methods that strive for photorealism, Toon Shading simplifies the color palette and accentuates edges to create a stylized, flat appearance reminiscent of cartoons. Toon Shading is widely used in animation, video games, and digital art to achieve a playful, vibrant visual style utilizing the capabilities of 3D rendering.

2.1 Outlines

When aiming for a cartoon or hand-drawn appearance, it's common to outline the edges of a model and highlight its ridges or creases (silhouette lines) with black lines. With this in mind, we'll explore a method to achieve this effect using the geometry shader. The idea is to use this shader to create an extra geometry, to form the silhouette lines by generating small, narrow quads that align with the object's silhouette edges.

One of the key capabilities of the geometry shader is its ability to provide extra vertex information beyond the basic shapes being rendered through the use of primitive rendering modes called "adjacency"modes. These modes allow us to associate extra vertex data with each shape. In this implementation, we'll use the $GL\_TRIANGLES\_ADJACENCY$ mode to provide information about adjacent triangles in our mesh. With this mode, we provide six vertices per primitive. The diagram in Figure 1 illustrates the locations of these vertices:

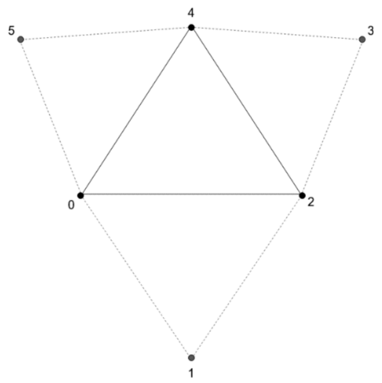

Figura 1: Triangles Adjacency.

In the preceding diagram, the solid line represents the triangle itself, and the dotted lines represent adjacent triangles. The vertices 0, 2, and 4 make up the triangle itself. On the other hand, the vertices 1, 2 and 5 make up the adjacent triangles. We can then use the adjacent triangles to identify whether an edge is part of the object's silhouette. The fundamental idea is that an edge is considered a silhouette edge if the triangle is front facing while the corresponding adjacent triangle is not. Figure 2 shows this concept in a more clear way.

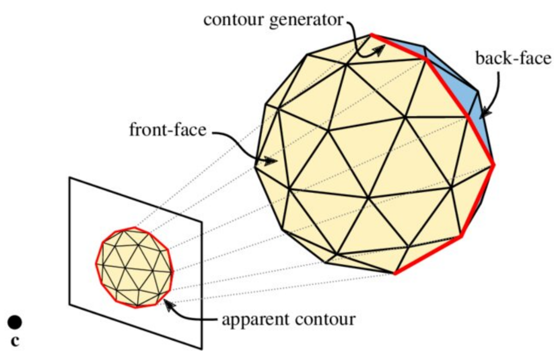

Figura 2: Front faces, back faces, and contour.

Within the geometry shader, we can determine if a triangle is front-facing by calculating its normal vector using the cross product. In eye coordinates (or clip coordinates), a triangle is front-facing if the z coordinate of its normal vector is positive. Therefore, for a triangle with vertices A, B, and C, the z coordinate of the normal vector is given by the Equation 1:

$$n_z = (A_x B_y - B_x A_y) + (B_x C_y - C_x B_y) + (C_x A_y - A_x C_y) \tag{1}$$

After identifying the silhouette edges, the geometry shader will create thin quads that align with these edges. Together, these quads will form the desired dark lines.

2.2 Implementation

The vertex shader converts the vertex position and normal into camera coordinates and sends them as *position* and *normal* for shading in the fragment shader, being ignored by the geometry shader. The vertex position is then transformed into clip coordinates using the model-view projection matrix and assigned to built-in *gl_Position*.

The geometry shader begins by defining the input and output primitive types using the layout directive. The, an additional variable, *isEdge*, is used in order to render the mesh in a single pass with appropriate shading for the base mesh, and no shading for the silhouette lines. This variable will let the fragment shader know when its supposed to render the base mesh and when its supposed to render the silhouette edge.

Next, we check if the main triangle, defined by points 0, 2, and 4, is front-facing, using the function $isFrontFacing$ and the previously described equation, 1. If the main triangle is front-facing, we will only create a silhouette edge quad if the adjacent triangle is not front-facing. The function $emitEdgeQuad$ creates a quad that aligns with the x-y plane, i.e. facing the camera, and with an edge defined by points $e0$ and $e1$. Here's how it works:

1. It calculates the value of ext, which is the vector from $e0$ to $e1$, scaled by $PctExtend$. This scaling slightly increases the length of the edge quad to ensure coverage between neighboring quads.

2. The variable v is set to the normalized vector from $e0$ to $e1$. Then, n is computed as a vector perpendicular to v.

3. The vector n is scaled by $EdgeWidth$ to match the desired width of the quad.

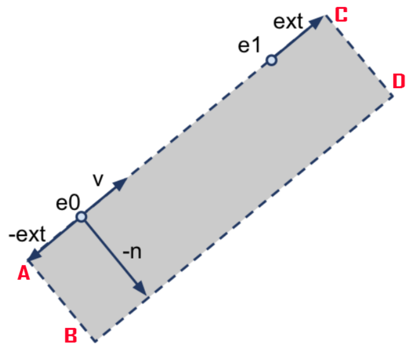

Figura 3: Edge generator.

Thus, the four corners of the quad are given by:

- A: $e0 - ext$
- B: $e0 - n - ext$
- C: $e1 + ext$
- D: $e1 - n + ext$

Where the z coordinate for A and B is the same as the z coordinate for $e0$, and the z coordinate for the vertices C and D is the z coordinate for $e1$.

Once all the silhouette quads are generated and emitted, the geometry shader will also output the original triangle unchanged, in order to be used by the fragment shader.

In the fragment shader, we decide how to color each fragment: either with toon shading or with a constant color based,as stated before, on the value of the variable *isEdge*. Toon shading achieves its effect by quantizing the light intensity into discrete levels, determined by the user, which creates the characteristic banding effect of toon shading. The final result can be observed in the Figure 4.

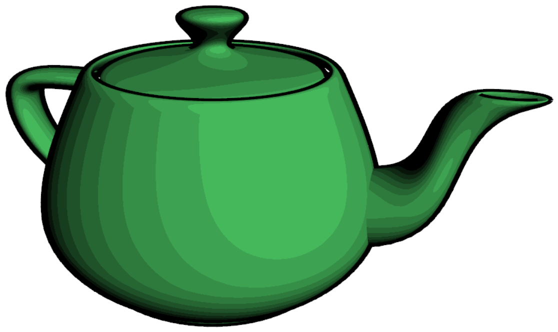

Figura 4: Feathering effect.

2.3 Issues

One issue with the previous technique is the potential for "feathering"caused by gaps between consecutive edge quads. One possible solution to this problem, would be to cover the gaps with triangles. In this case we opt to extend the length of each quad to cover these gaps, which can occasionally lead to artifacts if the extension is excessive.

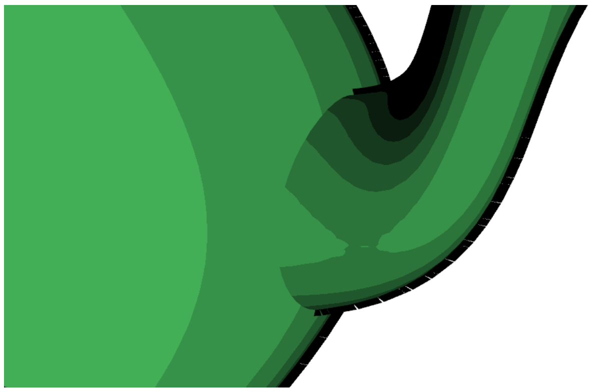

Figura 5: Feathering effect.

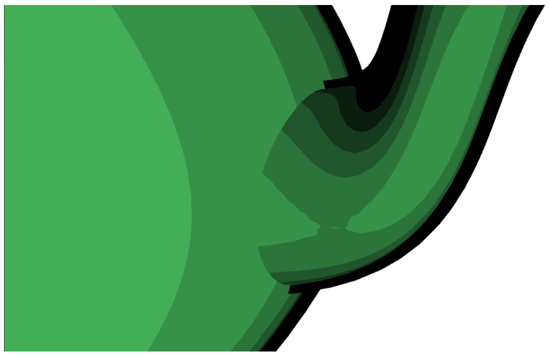

Figura 6: Extended quads.

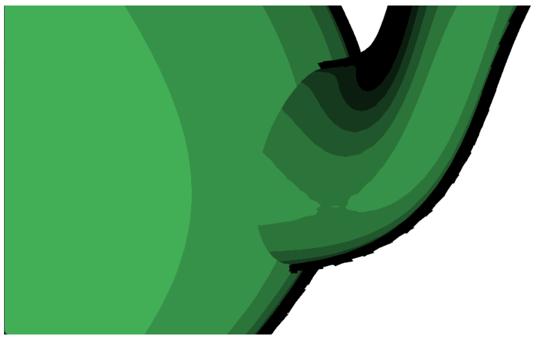

Figura 7: Artifacts due to excessive extension.

A second issue is related to depth testing. If an edge polygon extends into another area of the mesh, it can be clipped due to the depth test. The edge polygon should extend vertically throughout the middle of the preceding figure, but is clipped because it falls behind the part of the mesh that is nearby. This issue can be solved by using custom depth testing when rendering the silhouette edges.

3 Gooch Shading

Gooch Shading is a non-photorealistic rendering technique designed to enhance the perception of shape and features without relying on the conventional shading approaches that mimic real-world lighting. Gooch Shading achieves this by using warm and cool colors to represent light and shadow, respectively. This color dichotomy not only enhances the visibility of object details but also provides a more illustrative and technical appearance, making it especially useful in fields such as technical illustration, education, and any application where clarity and comprehension are paramount.

To implement this technique we started by implementing the vertex shader where we simply made the necessary transformations to pass the variables that we will need in the fragment shader, namely the normals, the light direction and the vertexes positions, all of them in the camera space. With regard to the fragment shader, given that we needed to represent the areas illuminated with warm colors and the areas not illuminated with cold colors, while avoiding noticeable and sudden transitions from one area to the other, we used the following equation:

$$I = \left(\frac{1 + \hat{I} \cdot \hat{n}}{2}\right) k_{\text{cool}} + \left(\frac{1 - \hat{I} \cdot \hat{n}}{2}\right) k_{\text{warm}} \tag{2}$$

The sense of depth can be communicated at least partially by a hue shift. However, to maximize the quality of the results, it is ideal to have a strong cool to warm hue shift using natural colors.

To automate the hue shift technique and introduce luminance variations in tone usage, we can explore two extreme methods for generating color scales: transitioning from blue to yellow tones and using scaled object-color shades. Our ultimate model combines these methods linearly. Blue and yellow tones are selected to ensure a transition from cool to warm colors, independent of the object's diffuse color.

As we know the color blue and yellow have the following RGB composition:

$$k_{\text{blue}} = (0, 0, b), \quad b \in [0, 1] \tag{3}$$

$$k_{\text{yellow}} = (0, y, y), \quad y \in [0, 1] \tag{4}$$

However, the blue-to-yellow tones range from a fully saturated blue to fully saturated yellow which results in a very sculpted but unnatural image, and is independent of the object's diffuse reflectance k_d. The extreme tone related to k_d is a variation of diffuse shading where k_{cool} is pure black and $k_{warm} = k$, what would look like the traditional diffuse shading. What we implemented is a compromise between these strategies with a transitions that is the result of a combination of tone scaled object-color and a cool-to-warm undertone. To do that we used the formula bellow to calculate the k_{cool} and the k_{warm}.

$$k_{\text{cool}} = k_{\text{blue}} + \alpha k_d \tag{5}$$

$$k_{\text{warm}} = k_{\text{yellow}} + \beta k_d \tag{6}$$

Consequently the values for b and y will determine the strength of the overall temperature shift, and the values of α and β will determine the prominence of the object color and the strength of the luminance shift. This combination should look something like what we can observe in the Figure 8:

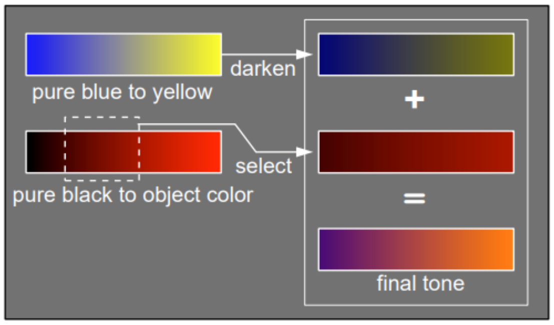

Figura 8: How the tone is created for a pure red object by summing a blue-to-yellow and a dark-red-to-red tone.

To conclude we simply calculate the specular component and added it to the intensity, I, previously calculated, obtaining the results presented in Figure 9.

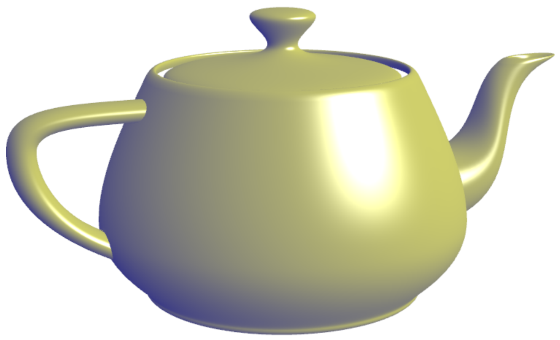

Figura 9: Gooch shading applied to a teapot model.

4 Hatch Shading

Hatch Shading is a technique that emulates the traditional artistic method of hatching, where closely spaced lines are used to create tonal or shading effects.By varying the density, orientation, and thickness of the hatching lines, artists can convey depth, form, and surface details in a highly stylized yet informative manner. In the digital realm, Hatch Shading leverages this technique to represent light and shadow in a manner that evokes hand-drawn illustrations, being particularly valuable in applications such as technical illustration, animation, and digital art, where the goal is to achieve a balance between visual appeal and the clear communication of information.

The implementation of this technique started by having a vertex shader responsible for transforming the vertex positions and normals, passing the necessary data to the fragment shader, including the textures coordinates. The fragment shader then uses this textures coordinates to sample the six textures exposed in the Figure 10.

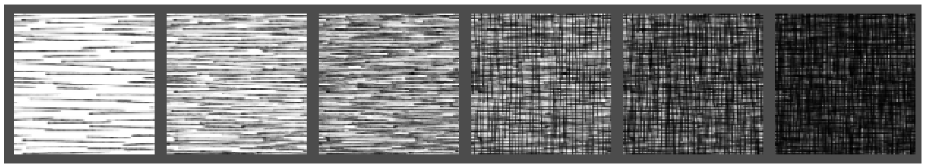

Figura 10: Textures chosen to implement the Hatch Shading.

Furthermore, it computes the light intensity and selects the appropriate hatch textures to blend, producing the final shaded appearance. In this process the light intensity is divided into six levels, *shades*, corresponding to the number of textures used. Higher values of intensity correspond to a less dense texture and, on the other hand, lower values of intensity correspond to a more dense texture. Based on this intensity, the fragment shader blends between adjacent hatch textures using a custom interpolation function f. In order to achieve this smoothness between textures, we used the predefined function *mix* that performs a linear interpolation between two values using a function to weight between them. For instances, lets consider the blending of the last two textures in the Figure 10. The *mix* function in this case would perform the steps in the Equation 7:

$$mix(texture5, texture4, f) = texture5 \cdot (1 - f) + texture4 \cdot f \tag{7}$$

Note that each intensity value is evaluated based on specific thresholds defined as multiples of the variable *shades*. Since each threshold blends two textures together, its important to use the second texture argument of the *mix* function in a certain threshold as the first texture argument of the *mix* function in threshold immediately after, assuring a smooth transition from a threshold to another. Thus, the f function uses the intensity and shade level to return the intensity value in a range between 0 and 1, ensuring the existence of smooth transitions between different hatch textures in the same threshold. Below, we can observe the implementation of the function f and its use in each threshold:

```
float f(float intensity, float i, float shades){
    return (6. * (intensity - (i * shades)));
}
```

```
float shades = 1./6.;
if(intensity <= shades){
    c = mix(h5,h4,f(intensity,0.,shades));
} else if(intensity <= 2*shades){
    c = mix(h4,h3,f(intensity,1.,shades));
} else if(intensity <= 3*shades){
    c = mix(h3,h2,f(intensity,2.,shades));
} else if(intensity <= 4*shades){
    c = mix(h2,h1,f(intensity,3.,shades));
} else if(intensity <= 5*shades){
    c = mix(h1,h0,f(intensity,4.,shades));
}else if(intensity <= 6*shades){
    c = mix(h0,vec4(1.),f(intensity,5.,shades));
}
```

The results can be observed in Figure 11.

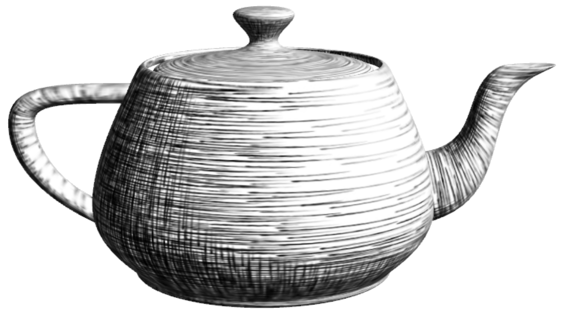

Figura 11: Hatch shading applied to a teapot model.

5 Pixelation Shading

Pixelation Shading brings a retro, pixel-art aesthetic to modern computer graphics. This rendering approach intentionally reduces the resolution of rendered images, mimicking the look of early video games and digital art, by converting complex visuals into a grid of larger pixels. This technique emphasizes simplicity and clarity, often highlighting essential details and shapes without the distractions of intricate textures and gradients. Pixelation Shading is widely used in video games, digital art, and animation, where the goal is to evoke a sense of nostalgia or to achieve a minimalist, stylized look.

Regarding the implementation of pixelation, we start by creating a vertex shader that only has the responsibility of carrying out the necessary transformations to the variables, in order to pass them to the fragment shader.

In the fragment shader to achieve a pixelation effect, we need to modify the texture coordinates such that multiple pixels in the original texture are mapped to the same pixel in the output. This involves snapping the texture coordinates to a regular grid. The following steps explain how the Equation 8 accomplishes this.

$$pixelCoord = floor(pixelCoord/pixelSize) \cdot pixelSize \tag{8}$$

As we know the texture coordinates ($texCoord$) provided to the fragment shader are normalized, meaning that they range from 0 to 1. To work with pixel-level operations, we first convert these normalized coordinates to absolute pixel coordinates. To do that we first use the function $textureSize$ that, given a texture, it returns a $vec2$ in the form $(width, height)$ and we store the result in a variable $texSize$. After that, we multiply the $texCoord$ by the $texSize$ to calculate the pixel coordinates.

To create a pixelated effect, we need to map multiple pixels in the texture to a single pixel in the output. We define the size of these grid with the variable $pixelSize$. This is done by dividing the $pixelCoord$ by the pixel size.

Next, we snap the scaled coordinates to the nearest lower integer value using the floor function. This step effectively ensures that any coordinates within the same $pixelSize$ grid are mapped to the top-left corner of that grid.

Furthermore, we scale the snapped coordinates back up to the original scale by multiplying by $pixelSize$. This converts the coordinates back to absolute pixel coordinates, but now they are aligned to the $pixelSize$ grid. The Equation 8 is a combination of all this steps. This equation ensures that any original pixel coordinate is mapped to a coordinate that corresponds to the top-left corner of a $pixelSize$ x $pixelSize$ grid in the texture. This effectively groups multiple pixels together, creating a pixelated effect when the texture is sampled at these modified coordinates.

After snapping the coordinates to the pixelation grid, we need to convert these pixel coordinates back to normalized texture coordinates, which range from 0 to 1. To do that we divide one by the the $texSize$ and that gives us the size of a single pixel in the texture in normalized coordinates that we store in $texelSize$. After that, to obtain the normalized pixel coordinates, we multiply the non normalized pixel coordinates by $texelSize$.

Finally, we sample the texture at the pixelated texture coordinates to get the color:

```
colorOut = texture(tex, pixelatedTexCoord);
```

The results of the pixelation shading can be observed in Figures 12 and 13.

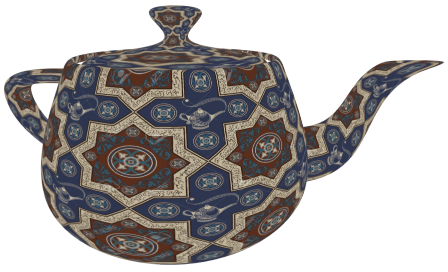
Figura 12: Teapot model.

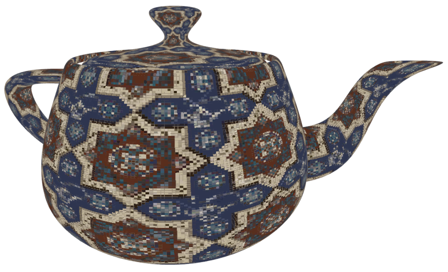
Figura 13: Pixelation shading.

6 Conclusion

In the realm of computer graphics, non-photorealistic rendering (NPR) represents a departure from traditional photorealism, emphasizing artistic expression and stylistic representation inspired by various forms of traditional art. Techniques like Toon Shading, Gooch Shading, Pixelation Shading, and Hatch Shading offer distinct visual styles that enrich the toolkit for visual storytelling in digital media.

Throughout this essay, we explored the principles and applications of these NPR techniques, highlighting their individual characteristics and creative possibilities. The developed implementations of these techniques have shown promising results for rendering smooth models, with customizable some features. Challenges such as feathering have been addressed to some extent. For instance, the dependency of the geometry shader on adjacency data can lead to inaccuracies in outlining for certain models constructed differently. Further enhancements, such as incorporating textures for stylizing outlines, could improve the overall visual fidelity.

Testing primarily focused on simple and smooth models has validated the suitability of these techniques for such cases. However, for comprehensive analysis, more complex models should be incorporated into future studies to evaluate their performance across varied geometries and scenarios.