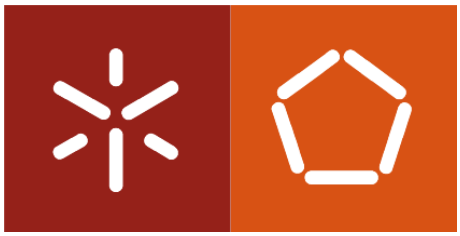

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Non-Photorealistic Rendering

Outlines

Carlos Castro (A81946)

Luís Macedo (A80494)

Introduction

Non-photorealistic rendering (**NPR**) is an area of computer graphics that, in contrast to the traditional focus on photorealism, focuses on enabling a wide variety of expressive styles for digital art. **NPR** is inspired by other artistic modes such as painting, drawing, technical illustration, and animated cartoons. It has appeared in movies and video games, usually in the form of toon shading and more technical work such as scientific visualization and architectural illustration.

Essential components of **NPR** are outlines, which are commonly paired with toon style shading but have a wide variety of uses, from highlighting important objects on the screen to increasing visual clarity in **Computer-Aided Design** (**CAD**) rendering.

The outlines can clarify the shape of complex objects or highlight essential features and allow the viewer to identify shapes, substituting the real-world subtle pieces of evidence, like shadows. An example of the importance of outlines is that cartoons rely on edge identification to convey shape information.

There are two types of outline that need to be drawn, silhouettes and creases, both indicating line discontinuity. This work focuses on implementing a silhouette extraction algorithm using the geometry shader, analyzing its pros and cons, and comparing it to common alternatives.

This document starts by introducing the concept of outlines, followed by presenting the main methods used for its detection and representation. Then, the developed implementation will be described and compared with previously mentioned methods.

Outlines

Outlines indicate some kind of perceived discontinuity and can be divided in two main classes.

Silhouettes allow the identification of discontinuities in the image space when a continuous surface ends (basically, the silhouette line separates the object from the rest of the world). When the object has many surfaces, the silhouette can appear only when the surface folds behind itself, which means that the line is present at the edge between the visible part and the non-visible part of the surface.

Like the silhouettes, the **creases** indicate discontinuities, but these discontinuities are in the normal vectors of the surface, regardless of the view direction or the camera settings.

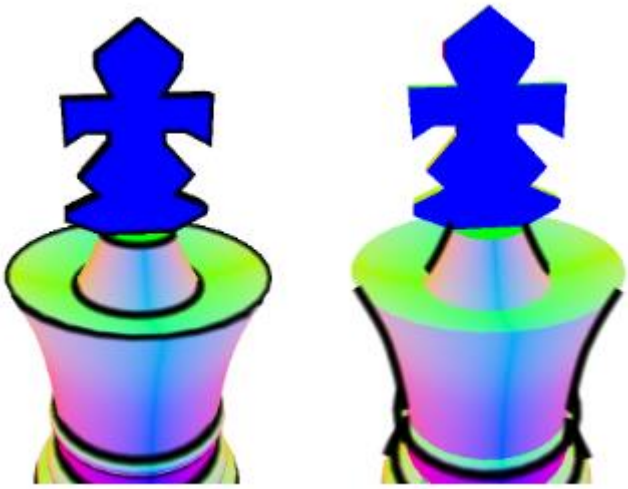

Figure 1 - Crease (left) and silhouette (right) (Hajagos, Szécsi, & Csébfalvi, 2013)

The algorithms in this section can be divided into algorithms that operate in the image space (2D) and world space (3D). The image space techniques use as an input an already rendered image or enhanced image of the scene. The object space techniques generate outlines directly using the 3D model.

Image-space Outline Detection

Image-space algorithms work by performing image processing operations on the rendered image or its associated buffers. These operations generally evaluate the first or second order differentials with finite differences or by convoluting with appropriate feature kernels (such as the Sobel kernel).

Edge detection on rendered image

A simple way to generate a line drawing of a 3D scene would be to perform an edge detection algorithm on the rendered image and display the edges found. However, the image's edges do not typically correspond to the silhouette edges that should be illustrated. For instance, highly textured surfaces will generate many edges that are irrelevant to the object shape. Another issue is that no edges are detected between two overlapping objects with the same color.

Edge detection on depth map

An improvement to this would be to apply an edge detector to the depth map of the image (Figure). As the depth map is an image where the intensity of a pixel is proportional to the depth at that point in the scene, the variation in depth between adjacent pixels is usually small over a single object but large between different objects.

A problem with this method is that it does not detect the boundaries between objects that are at the same depth, nor does it detect creases. In other words, it can only detect C0 surface discontinuities.

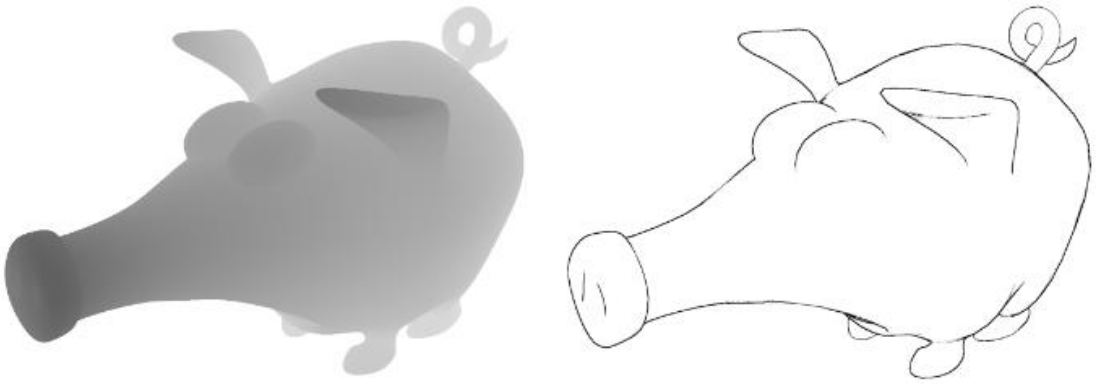

Figure 2- Depth Map (left), edges of the depth map (right) (Bénard & Hertzmann, 2019)

Edge detection on the normal map

The silhouette edges computed with the depth map can be augmented by using the surface normal as well. This is achieved by using a normal map, which is an image that represents the surface normal at each point on an object, where the (R,G,B) color components match to the (x,y,z) surface normal at that point (Figure 3).

Figure 3 - Normal map (Bénard & Hertzmann, 2019)

Computing edges on the normal map detects surface creases, which can then be combined with the edges of the depth map to produce a reasonably good silhouette image (Figure 4).

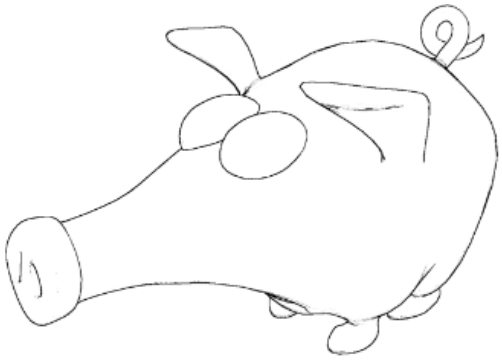

Figure 4 - Combined edges of depth map and normal map (Bénard & Hertzmann, 2019)

For many applications, the techniques described in this section are sufficient. However, they do suffer the fundamental limitation that important information about the 3D scene is discarded during rendering, and this information cannot be reconstructed from the 2D image alone.

Image-space algorithms only depend on the final image resolution, which is usually an advantage performance-wise, making this approach appealing for real-time applications. However, they have some drawbacks:

- Provide limited control over stylization, since there is no explicit curve representation, just pixels in a buffer.
- Edges can be incorrect, for example, missing contours at small discontinuities or falsely detecting them for highly detailed surfaces.
- Results may not be consistent and predictable when the resolution of the image changes.

Object Space Outline Detection

This section will describe methods for finding silhouettes of models in 3D. These methods are more complex than the image space methods but can produce outlines with much higher precision. These outlines are also suitable for additional processing, for example, they can be rendered with natural brush strokes.

The developed solution, described later in this section, utilizes the geometry shader to efficiently detect and draw the contour. Additionally, some other common methods for object space outline detection will be presented.

Two-pass rendering

The basic idea of these approaches is to render the geometry twice: first, a standard rendering pass to fill the depth buffer, and then second, the geometry is enlarged and only back-faces are rendered, to make the contours emerge from the rasterization. Using the first generation of programmable hardware, a similar effect is achievable in one pass by enlarging all back-facing polygons in the Vertex Shader.

This method provides some basic control over the stylization of the outline, such as width and color (Figure), but nothing beyond that. Some of its drawbacks are that it only addresses contour edges,

not internal edges, or creases, and that it can be slow for large models, as it requires two rendering passes. Another aspect to consider is that objects further away will have smaller outlines than those nearby. To get constant width outlines, the outline needs to be scaled by the distance from the camera. Of course, this is only a drawback if constant outlines are required.

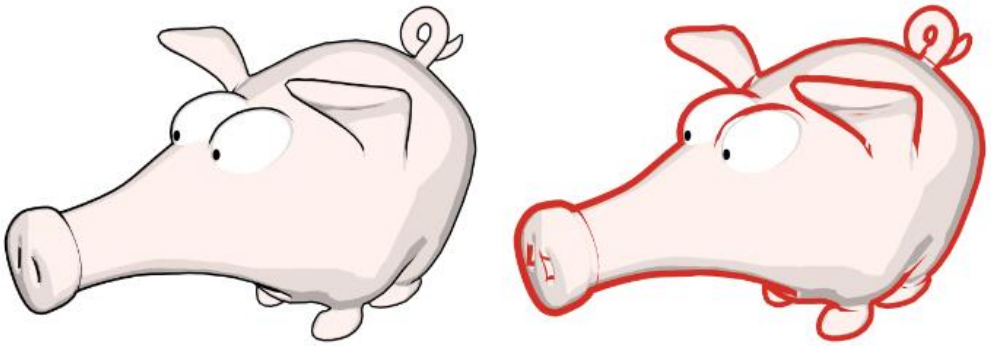

Figure 5 – Outlines rendered using the two-pass method with black edges on the left, and wider red edges on the right. (Bénard & Hertzmann, 2019)

Dot product between the direction to the viewer and the normal vector

In the case of smooth surfaces, points on the surface at silhouettes are characterized by normal vectors parallel to the viewing plane and therefore orthogonal to the viewer's direction. By calculating the direction to the viewer and the normal vector and testing whether they are (almost) orthogonal to each other, we can therefore test whether a point is (almost) on the silhouette.

More specifically, if **V** is the normalized direction to the viewer and **N** is the normalized surface normal vector, then the two vectors are orthogonal if the dot product is 0: $\boldsymbol{V} \cdot \boldsymbol{N} = 0$. This can be done in the fragment shader, where the normal of the fragment being shaded will be needed. If it is very close to perpendicular (dot product $\boldsymbol{V} \cdot \boldsymbol{N}$ is close to 0), then its color can be set to the outline color.

To better convey unlit areas, the outline's thickness can be adjusted considering the diffuse reflection term of the fragment. This could be achieved by defining two outline thicknesses, one for fully lit areas and one for unlit areas. The thickness parameter would then be interpolated between these two values according to the diffuse reflection term, allowing for a continuous change in thickness. This can be seen in the donkey in Figure 6, where the outlines at its belly and in the ears are considerably thicker than other outlines.

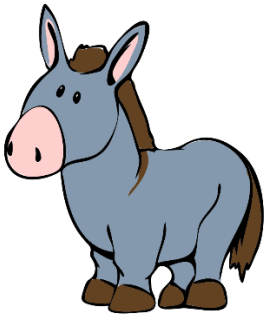

Figure 6 - A cartoonish donkey (Villarreal, 2020)

The major drawback of this method is that it only works for smooth edges. It also generates outlines of varying thickness (which is a plus or a minus depending on the visual style). However, the overall thickness of the outlines can be controllable by a shader property. This approach "consumes" a little bit of the model to do the outline, which again, can be desirable depending on the visual style.

Outline detection with the geometry shader

For a mesh surface, and a given viewpoint, the silhouette (or contour) is a curve on the surface that delineates the frontier between what is locally visible and invisible, that is, between front and back-facing surface regions (Figure 7).

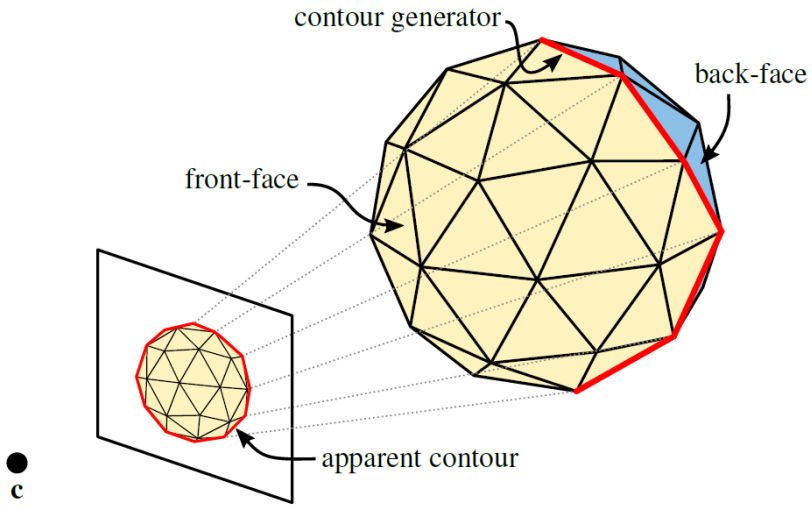

Figure 7 - Front faces, back faces, and contour (Bénard & Hertzmann, 2019)

The front faces are shown in yellow and are visible to the camera. The back faces are in blue and are not visible to the camera. The collection of all mesh edges that connect front-faces to back-faces are together called the contour generator. The apparent contour is the visible projection of this curve onto the image plane.

A common way to detect a contour edge is to compute the normals of its two adjacent faces and checking whether their dot-products with the view direction have opposite signs. An early solution used to detect the set of contour edges on a mesh was by iterating over every mesh edge and performing this check. The iteration over the mesh edges must be performed every time the camera or object position changes, which is very expensive for complex models. This technique was later improved and parallelized by using the geometry shader.

The geometry shader can be used to draw silhouette lines by producing the additional geometry for the outlines. The shader will approximate these lines by generating small, thin quads, aligned with the edges, that make up the silhouette of the object. This is accomplished since this shader can provide additional vertex information related to the nearby primitives within a mesh, using the Layout Qualifier `GL_TRIANGLES_ADJACENCY`. The additional information is presented in Figure

8, where the points 0, 2 and 4 consist of the original triangle, and the points 1, 3 and 5 are the points of the adjacent triangles.

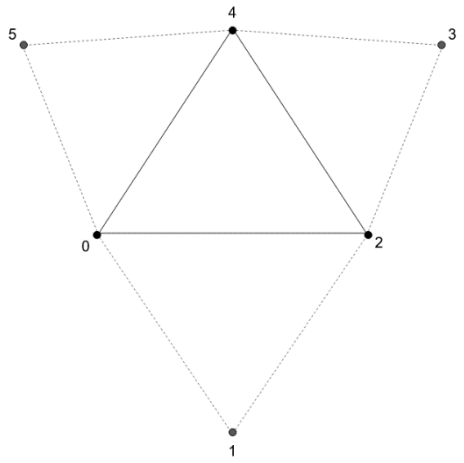

Figure 8 - Triangles adjacency

The extra points information is used to determine whether a triangle edge is part of the silhouette of the object. The basic assumption is that an edge is a silhouette if the triangle is front-facing and the corresponding adjacent triangle is not front-facing.

A triangle can be determined to be front-facing by computing the triangle's normal vector, using the cross product. When working with clip coordinates, the z coordinate of the normal vector will be positive for front-facing triangles and negative otherwise.

For a triangle with vertices A, B and C, the z coordinate of the normal vector can be obtained with the following equation:

$$n_z = \left(A_x B_y - B_x A_y\right) + \left(B_x C_y - C_x B_y\right) + \left(C_x A_y - A_x C_y\right)$$

In addition to checking whether the triangle is front-facing, it is checked whether the triangle is present on the screen, that is, whether the coordinates (in clip space) of the triangle points are between -2 and 2.

Once the silhouette edges are determined, the geometry shader will produce additional thin quads aligned with the edges and with the x-y plane (facing the camera). These quads, taken together, are what make up the desired silhouette lines.

The Figure represents the method to calculate the quads points, where:

- **e0** e **e1** are the points that represent the edge of the original triangle
- vector **ext** is the distance to overdraw the outline after the edge points
- vector **n** is the thickness of the outline
- **A**, **B**, **C** and **D** are the points of the quad

In addition to this, **edgeOverdraw** and **edgeWidth** are user defined parameters that will affect the quad creation.

The function that calculates **A**, **B**, **C** and **D** receives **e0** and **e1** as input. It begins by calculating **ext,** scaling the vector from **e0** to **e1** by **edgeOverdraw** (percentage to overdraw the outline). A normalized vector from **e0** to **e1** is assign to the variable **v.** Next, it is time to compute **n**, a vector perpendicular to **v**, that is calculated by scaling the vector (**-v.y**, **v.x**) with **edgeWidth** (outlines width). The last part is to calculate the coordinates of the quad points and emit them. This calculation is as follows:

- **A**: $(e0.xy - ext, e0.z)$
- **B**: $(e0.xy - n - ext, e0.z)$
- **C**: $(e1.xy - ext, e1.z)$
- **D**: $(e1.xy - n - ext, e1.z)$

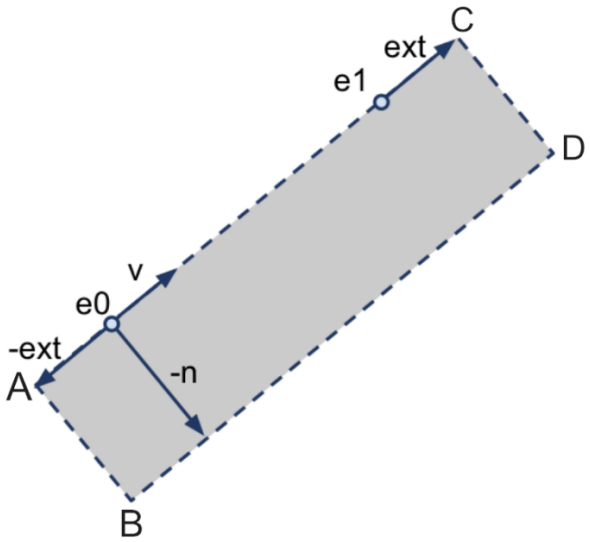

Figure 9 - Outline quad (Wolff, 2011)

The quads are extended further than the original edge points because of a problem usually called feathering. What happens is that on curved surfaces there would be gaps between consecutive edge quads, as in Figure .

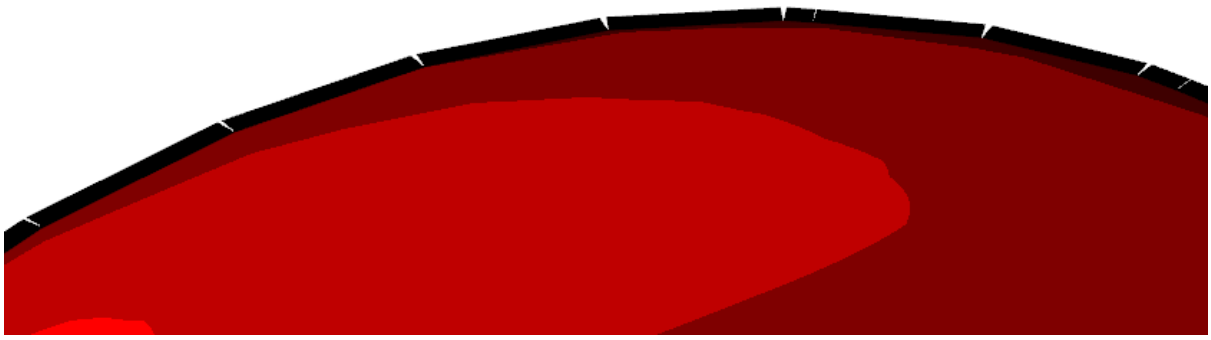

Figure 10 - Outline feathering

This problem is fixed by slightly extending the quads to fill the gap (Figure). This can cause artifacts (Figure) if extended too far but works well for most cases. Another more complex solution would be to fill the gaps with additional geometry such as triangles.

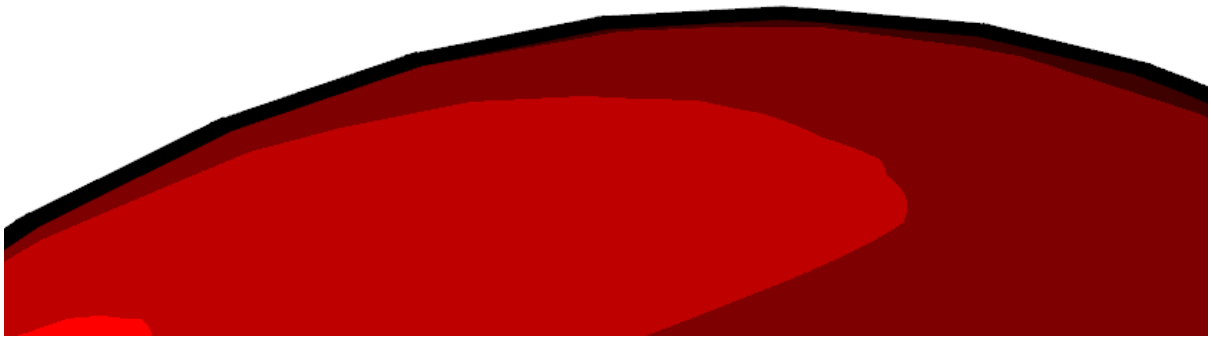

Figure 11 - Extended quads

Figure 12 - Artifact created by the exaggeration of the quads extension

After generating all the silhouette quads, the geometry shader will output the original triangle (3 vertex), and at most, a quad for each edge (4 vertex).

Lastly, to render the mesh in a single pass with appropriate shading for the base mesh, and no shading for the silhouette lines, an additional output variable is passed to the fragment shader indicating if it is rendering the base mesh or the silhouette edge. The fragment shader shades the fragment depending on if it is an edge. If it is and edge, a fixed color is used, if not, a regular shading technique is used such as toon shading which was implemented in the developed solution.

The pipeline can be divided in the following steps:

1. The vertex shader is a simple pass-through shader that outputs the vertex position, normal in camera coordinates and light direction, in addition to the standard position.
2. The geometry shader:
    a. Receives as input the output of the vertex shader and the additional information about the adjacency triangles (points not shared with the original triangle).
    b. Determines if the main triangle is front-facing, and if so, check if the adjacency triangles are back facing, one at a time.
    c. If any of the adjacency triangles are back facing, produce the outline quad for the shared edge with the original triangle.
    d. Outputs the vertex data from the original triangle and, if applicable, the vertex data from the silhouette quads.
3. The fragment shader outputs a fixed color if it is on an edge or uses toon shading if it is not.

The visibility of the outlines is determined by the standard depth test which can lead to some issues. If an edge extends into another area of the mesh, it can be clipped. Simply disabling the depth test for outlines would not work as creases would be visible from the opposite side of the model. The solution is to use custom depth testing, testing along the edge that originated the quad rather than at the current fragment. This method is called **spine testing**.

Another implementation was developed that uses spine testing and extrudes the quad in both directions instead of only outwards, with the edge being in the middle. As this implementation uses spine testing instead of regular depth testing, both sides of the quad can be seen, even if half of it is inside the mesh. For example, in Figure 13 quads in edges A and B would be partially occluded by the mesh, spine testing fix this by testing visibility at the red dot.

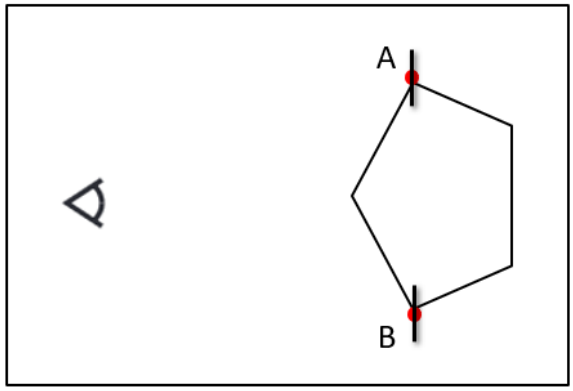

Figure 13 - Top view

This new implementation is now divided in 3 passes instead of the single pass of the first implementation. The first pass is an early Z pass that stores the depths in a G-buffer, the second renders the toon shading, and the last renders the outlines.

The main differences are now on the last pass. The geometry shader outputs a quad that extends to both sides of the edge, instead of only outwards (as can be seen in Figure 14) and adds information relative to the edge points. This information is then used on the fragment shader to perform the depth test on the edge points that originated the fragment. This depth test is performed using the depth values obtained in the first pass, while the regular depth test is disabled.

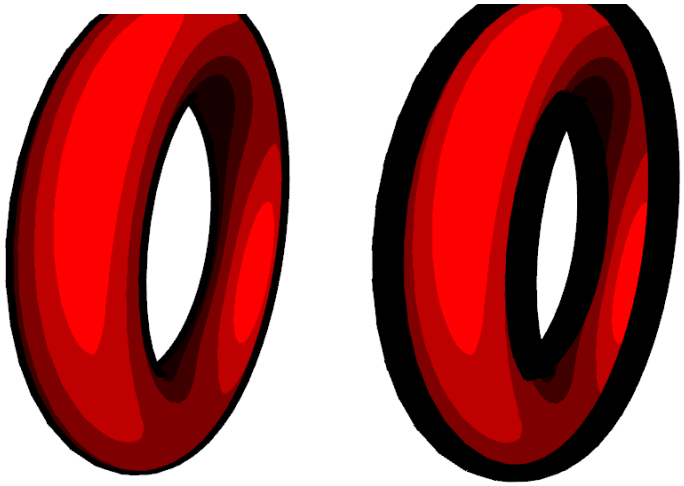

Figure 14 - Different outline widths with quads centered at the edge

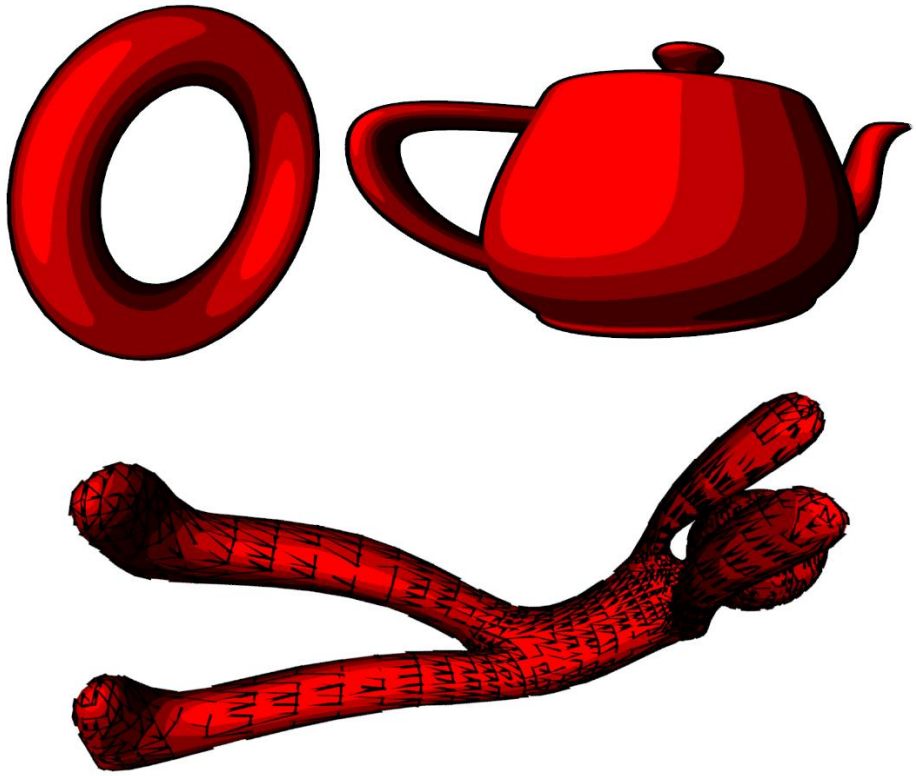

Figure 15 - Algorithm results

The presented algorithm has some advantages and disadvantages. In the group of disadvantages, there is the incompatibility with specific 3D models, due to the way they are built (for example, the man in Figure 13), as there may be a bad connection between triangles, or the algorithm detects falsely outlines where it should not be, because of the direction of the normal being barely front-facing. The outline in sharp corners does not work well in this algorithm because they do not get fully connected, and if they do, appears artifacts, like the one in Figure .

In terms of advantages, this algorithm is relatively simple to design, which means that it can be easily added to any rendering system and can replace costly edge detection filters with a more flexible, geometry aware method in real time. Both the width, color, and amount of overdraw can be easily customizable. Furthermore, since the geometry shader creates new geometries (the quads), it is possible to apply effects to the outlines, like applying a pencil stroke texture, and besides that, the width of the lines and the percentage to overdraw the outline can be modifiable.

Conclusions

The developed implementation provides good results for smooth models, while also providing some customization for the outline. The problems of feathering and depth testing were addressed but there are still some issues that restrict its use. Due to the geometry shader's dependency on adjacency data, some models have incorrect outlines due to the way they were built. The stylization of the outlines could also be further improved, for instance, by adding textures.

The implementation was tested mainly with simple and smooth models which verified its suitability for these cases, but more complex models should be used for further analysis.

References

Bénard, P., & Hertzmann, A. (2019). Line Drawings from 3D Models: A Tutorial. *Foundations and Trends® in Computer Graphics and Vision*.

Hajagos, B., Szécsi, L., & Csébfalvi, B. (2013). Fast silhouette and crease edge synthesis with geometry shaders. *Proceedings of the 28th Spring Conference on Computer Graphics*, (pp. 71-76). New York.

Villarreal, M. R. (2020, April 16). *GLSL Programming/Unity/Toon Shading.* Retrieved from wikibooks: https://en.wikibooks.org/wiki/GLSL_Programming/Unity/Toon_Shading

Wolff, D. (2011). *OpenGL 4 Shading Language Cookbook.* Packt Publishing.