DH2323 DGI Final Project Report
Silhouette Rendering with Vector Monitor Lines

Bennett Bernardoni
KTH Royal Institute of Technology
bennettb@kth.se

***Abstract*—The goal of many computer graphics applications is to generate realistic visuals. However, 3D scenes can also be effectively depicted by using non-photorealistic techniques. This work presents a method to render 3D geometry as silhouetted wireframe using lines with the style of a vector monitor in real-time. The effectiveness is then analyzed and a possible perceptual study is outlined.**

I. Introduction

At the beginning of 3D computer graphics, scenes were rendered using nothing more than a couple lines. In many cases, these were displayed using vector monitors, an old display technology where graphics are rendered via an electron beam directly drawing lines as opposed to a grid of pixels as on raster monitors. These monitors were notably used in arcade games up until the mid-1980s including Tempest[1], Star Wars[2], and Battlezone[3] all of which feature 3D graphics.

In this paper the display capabilities of vector monitors are emulated to create 3D scenes in real-time. However, as the geometry of models is much more complex than was used in the days of vector monitors, a method of determining which lines to draw is needed. An algorithm is used to detect edges that form a silhouette or crease on a model from a movable viewpoint.

The paper is organized in the following sections. Section II examines related work. Section III presents the implementation of the algorithm. Section IV evaluates the results. Finally, Section V concludes with a summary of this and future work.

II. Background

The process of extracting feature lines from a model is a popular topic in non-photorealistic rendering (NPR), a branch of computer graphics that focuses on using a variety of artistic styles to generate imagery. There are two main approaches, image space and object space[1]. Image space methods post-process rendered scenes. By detecting edges after rendering the models, geometric details are lost making drawing stylized lines harder. Therefore, this paper focuses on an object space method which directly uses the 3D geometry. Furthermore, this method can take advantage of a geometry shader and its adjacency information to enable real-time performance[2].

This work focuses on detecting two types of feature lines, silhouette (or occluding contour) lines and crease lines. As pictured in Figure 1, Silhouette lines are edges where a front facing triangle (A) meets a back facing triangle (B). Creased edges are where the angle between the normals of two front facing triangles (n_a and n_b) is greater than some threshold value ($\cos\theta$)[3]. Al-Rousan et al.[4] provides a detailed overview of these and several other types of lines. An article on Silhouette Extraction[5] has provided a rough outline of some of the steps used for the implementation presented in this paper. Additionally, it references a visibility algorithm for occluding lines from another paper[6].

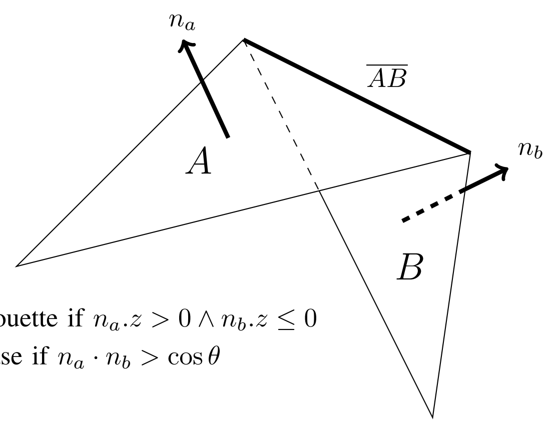
n_a
$\overline{AB}$
n_b
A
B
Silhouette if $n_a.z > 0 \land n_b.z \leq 0$
Crease if $n_a \cdot n_b > \cos\theta$

Fig. 1. The edge $\overline{AB}$ connects faces A and B with normals n_a and n_b. The mathematical definitions for silhouette and crease edges are also given.

The inspiration for this project comes from a PlayStation game called Vib-Ribbon[7]. Although Vib-Ribbon only approximates the silhouette effect using flat geometry instead of a specially rendered model. A more modern example released during the development of this project is Muffled Warfare[8], which perfectly demonstrates the intended effect.

III. Implementation

Following is a description of the implementation of the application. All the program code was written in C++ and the shader code was written in GLSL. The development process behind the implementation is recorded on the project blog[9].

A. Libraries

I was able to get the application working with the help of several libraries. Additionally, since I have not had much previous experience with these libraries, I used a couple tutorials that will be linked in the footnotes.

[1]Video Demo: https://youtu.be/AMto2HJJSSA?t=44s
[2]Video Demo: https://youtu.be/CbqFawQvdGE
[3]Video Demo: https://youtu.be/zdfKy4c7yuc?t=8m15s

SDL 2.0[4] is used to handle window creation and input. For rendering graphics OpenGL[5][6] is loaded via GLAD. GLM handles vector and matrix operations. Finally, Assimp is used to load models.

B. Generating Adjacency Data

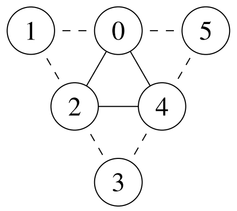

Fig. 2. GL_TRIANGLES_ADJACENCY format, numbers indicate index.

SDL, OpenGL, shader programs, and framebuffer are all initialized in a standard way. However, after importing the model with Assimp, there is no adjacency data that is required later by the geometry shader. As seen from Figure 2, for every edge of every triangle an adjacent vertex needs to be included.

At first, a naive algorithm was written that loops over every face to find the adjacent vertex for every edge. This has a time complexity of $\mathcal{O}(|V|^2)$ where $|V|$ is the number of vertices. For a simple model of 511 vertices running on my modest laptop, this takes about 4 seconds. This is slow especially considering that the test model is small, and the algorithm does not scale well.

Algorithm 1 Generate Triangle Adjacency Data

```
1: function GEN_TRI_ADJ(inIndices)
2:     halfEdgeHashTable ← empty hash table
3:     for $face \leftarrow 0$ to $(numFaces - 1)$ do
4:         for $edge \leftarrow 0$ to $2$ do
5:             $v_0 \leftarrow$ inIndices$[face][edge]$
6:             $v_1 \leftarrow$ inIndices$[face][(edge+1)\%3]$
7:             $v_2 \leftarrow$ inIndices$[face][(edge+2)\%3]$
8:             halfEdgeHashTable$[(v_0, v_1)] \leftarrow v_2$
9:     for $face \leftarrow 0$ to $(numFaces - 1)$ do
10:        for $edge \leftarrow 0$ to $2$ do
11:            $v_0 \leftarrow$ inIndices$[face][edge]$
12:            $v_1 \leftarrow$ inIndices$[face][(edge+1)\%3]$
13:            $v_{adj} \leftarrow$ halfEdgeHashTable$[(v_1, v_0)]$
14:            outIndices$[face][2 * edge] \leftarrow v_0$
15:            outIndices$[face][2 * edge + 1] \leftarrow v_{adj}$
       return outIndices
```

After some research, I found a rather simple method[7] that generates the adjacency data in $\mathcal{O}(|V|)$. It does this by first creating a hash table that maps half-edges to the third vertex of the triangle. Then, it loops through each half-edge and inputs the opposite edge into the hash table to find the adjacent vertex. The full procedure is given in pseudocode in Algorithm 1. Generating data for the same model with this method took 100 milliseconds on my laptop, which is 40 times faster. Additionally, the algorithm will scale better for more complex scenes.

C. Render Loop

Now that the initialization code has been described, following is an overview of the render loop. First, the view matrix is recalculated based on user input of the position and rotation of the camera. Then, the model is rendered to a depth texture. After a vertex shader transforms the vertices, a geometry shader detects and emits the vector lines. Next, a fragment shader draws these lines to a framebuffer if they pass a custom depth test. The screen is then blurred with a two-pass Gaussian blur. This blurred framebuffer and the raw vector lines framebuffer are combined and drawn to the screen to perform a bloom effect.

1) Render to Depth Texture: To render the stokes in the vector lines fragment shader, a complete depth buffer is needed. Therefore, an early Z pass is performed. The depth texture can easily be created by turning off the color mask and binding a framebuffer with only a depth attachment. Then, the models are drawn using a vertex shader that only multiplies by the Model-View-Projection matrix and no fragment shader.

2) Setup Vector Lines Rendering Pass: After rendering the depth texture, the color mask is re-enabled, the standard depth test is disabled, the depth framebuffer is unbound, and the depth texture is bound to texture 0. Next, the models are drawn again, this time using the vector lines shader. The vertex shader simply transforms the vertices as well as passing through the object space coordinates for perspective independent calculations.

3) Vector Lines Geometry Shader: In the geometry shader, the given triangle is first verified to be front facing. Then each of the three edges are checked to determine if they form a silhouette or crease. If they do, the line is emitted as a triangle strip of four points.

A silhouette line is detected if the adjacent triangle is back facing. The direction of the face is determined by examining the sign of the Z component of the face normal. Note that the normal is computed from the clip space vertices after the perspective transformation. Contrarily, crease edges are determined by taking the dot product of the object space normals. This ensures that crease edges are view independent. This value, which is the cosine of the angle between normals, is then compared to some threshold value. The current implementation uses a threshold angle of $60°$.

After an edge has been detected, it must be clipped to prevent a negative homogeneous component from affecting later computations. Clipping is performed via the Cohen-Sutherland algorithm on the six standard homogeneous clipping planes.

Finally, the output variables for the fragment shader are calculated. The first is the position of the four corners of the rendered rectangle. The line is extruded by a constant pixel

[4]http://lazyfoo.net/tutorials/SDL/index.php
[5]http://www.opengl-tutorial.org/
[6]https://learnopengl.com/
[7]https://gamedev.stackexchange.com/questions/62097/building-triangle-adjacency-data

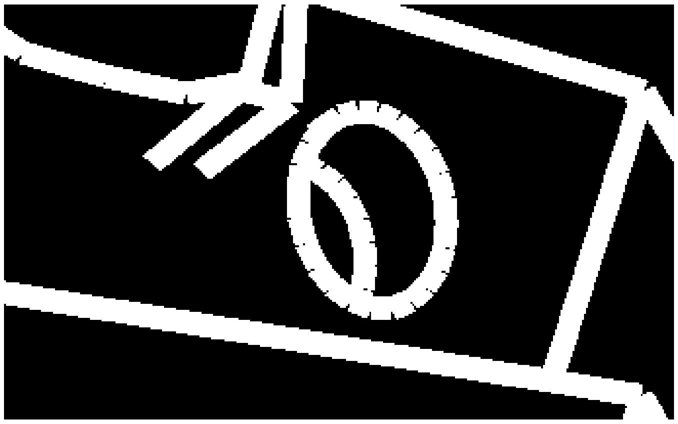

Fig. 3. Example of gaps at vertices. Lines are rendered with a flat color for clarity.

distance perpendicular to the spine in both directions (four pixels in current implementation). The lines are also extended by this distance to ensure that no gaps appear when two lines meet at an angle as shown in Figure 3. Finally, a Z-bias is added to reduce Z-fighting as explained at greater detail in the next section. All these offsets are added to the original points while taking care to preserve the homogeneous component to not disrupt rasterization.

The texture coordinates, length of the line, and spine position are also computed for use in the fragment shader. The texture coordinates are qualified as `noperspective` to be interpolated correctly in screen space.

4) Vector Lines Fragment Shader: Next, the fragment shader tests line depth using a custom depth test and calculates the pixel brightness to emulate the look of a vector monitor.

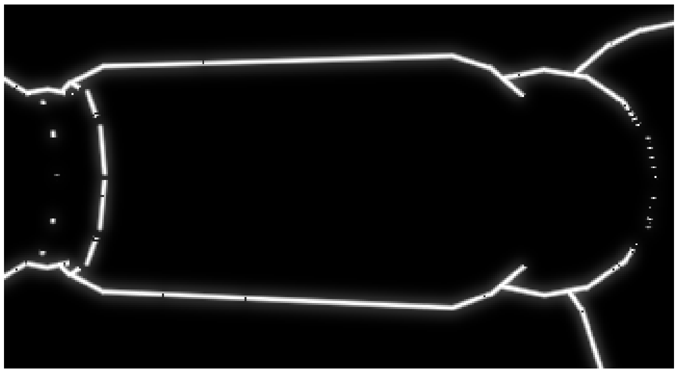

Fig. 4. Example of Z-fighting causing broken lines.

A depth test is necessary to properly occlude lines that are hidden behind the model. A standard depth test would not work as the extruded part of the line might extend into the model, failing the depth test even though it should be visible. This is very common in the case of valley creases and for the model side of silhouette edges. One solution is to perform the depth test along the spine of the edge. This method is called the spine test[6]. To perform this test, the depth buffer needs to be fully filled. This is why a depth texture was rendered in an earlier step. Now, the Z component of the fragment simply needs to be compared to the position of the spine point in the depth texture. However, due to the precision of floating point values some Z-fighting still occurs even after adding a Z-bias in the geometry shader. The result of this is shown in Figure 4. To reduce Z-fighting even more, the eight adjacent pixels are also sampled to achieve better accuracy. If the depth test is failed, the fragment is discarded, otherwise the pixel brightness is calculated.

The pixel brightness is set as inversely proportional to the distance from the spine. There is an edge case if a fragment is in an end cap, the distance to the endpoint is used instead to give rounded ends. This can be seen at the end of some of the broken lines in Figure 4. The line color is then multiplied by the brightness to determine the output color sent to the framebuffer. The lines are blended using `GL_MAX` as the blend equation. This prevents overlapping endpoints from being too bright.

5) Gaussian Blur and Bloom: Finally, a bloom effect is achieved by blurring the output from the previous step with a Gaussian blur and adding this back to the raw vector lines texture.

The blur is performed as a two-pass Gaussian blur to improve efficiency. If a 9×9 kernel were used in a one-pass blur it would take 81 samples. Instead, a 1D blur is applied in the horizontal direction and then in the vertical direction requiring $9+9 = 18$ samples per iteration. At first, to increase the blur strength this process was repeated four times. However, this was later found out to be nearly equivalent to a 17-tap blur applied once. Then, even further improvement came from the algorithm presented in an article[10]. This method takes advantage of the GPU's texture filtering by sampling between texels to calculate two weights at a time. Now, only a 9-tap filter is needed to achieve the same results.

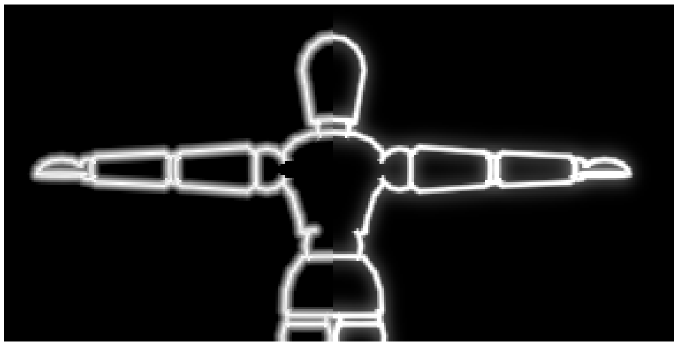

Fig. 5. Comparison of an 8 pixel half-width rendering without bloom (left) to a 4 pixel half-width rendering with bloom (right).

Lastly, bloom is applied by adding the original vector lines output back to the blurred version. There is also an option to scale the blurred component by a fixed factor. However, a factor of 1 achieved the best results. A comparison with and without bloom is given in Figure 5.

IV. Analysis

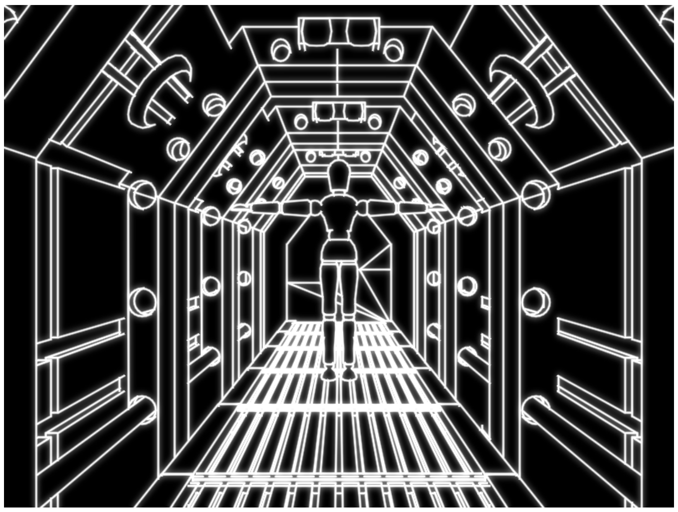

Fig. 6. Final implementation rendering a test scene of a dummy in a sci-fi hallway.

My goal for this project was to write an application that could render arbitrary models to look like they are being displayed on a vector monitor in real-time. As evidenced by Figure 6, this goal has been achieved to my satisfaction. However, there are a couple other, less subjective ways to measure the success of this project.

The final application meets many of the objectives laid out in the project specification. The outer edges and the distinguishing feature lines are drawn by detecting silhouette and crease edges. Lines that fall behind other parts of the scene are properly occluded. Also, the edges do in fact emulate lines drawn by a vector display with their inversely proportional brightness and bloom effect.

Another target of this project was to have the application function in real-time. This can be measured by testing frame rate. Using my relatively lower powered laptop with integrated graphics, the algorithm runs at 241.0 frames per second to render two test model of 2956 (dummy) and 6595 (hallway) vertices in Figure 6. This equates to 4.15 milliseconds per frame where 1.99 milliseconds come from the bloom step which is constant for a given screen resolution. Another timing test shows that one, two, and three dummies take 3.154, 3.294, and 3.457 milliseconds per frame (= 317.1, 303.5, and 289.2 fps) respectively to render. This suggests that the method scales linearly with scene complexity. Even with a more complex scene comprised of more lines, the algorithm still should be able to run smoothly in real-time especially with better hardware.

Next, let us look at any flaws or graphical artifacts present in the current implementation. While Z-biasing and supersampling the depth buffer have greatly reduced Z-fighting, it still occasionally occurs. Another problem is the quality of the vector line representation is dependent on the model used. If the model has very smooth curves, crease edges will not be present leading to a shapeless blob of silhouette edges. Additionally, some objects tend to be more recognizable using few lines while others are too complex to be represented well.

A. Perceptual Study

Another approach towards assessing this project is using a perceptual study. While most perceptual studies focus on the realistic aspect of computer graphics, some studies have been conducted in the field of NPR. A survey of user evaluation methods of NPR[11] has categorized several techniques according to their targeted application. These include psychological, architectural, medical, learning, and natural phenomenon applications, as well as, perception of space, and texture-based depictions. The specific studies range from evaluating the emotional response of NPR[12] to the effectiveness of NPR techniques in scientific illustrations[13].

As the goal of this project was to create convincing representations for 3D geometry, a perceptual study should evaluate the effectiveness of the vector line representations. One possible method would be to test participants on their ability to identify objects while varying the rendering algorithm. This would provide incite as to how recognizable vector line images are.

1) Related Studies: The study introduced above pertains to visualization comprehension of NPR imagery. A comprehensive study by Kim et al.[14], tested participants in the effectiveness of texture patterns at conveying 3D shape information. The results suggest that accuracy was affected by the principal direction texture pattern used. Velez et al.[15] investigates the difficulties of understanding visualizations by testing the differences in spatial ability. It found that high spatial ability correlates with accuracy on their computer-based 3D visualization test, but not with response time.

2) Design: First, the participants will be presented a scene filled with several object for a short time. Then, they will be questioned on what objects they saw. Both the scene and each individual object will be scored by accuracy. The rendering method of the scene will be changed to test the effectiveness of each method. Possible methods include wireframe, solid silhouette, cel shading, realistic shading, and the vector lines representation presented in this paper. No texturing or coloring will be applied to control their effects. Instead this study will be focused on the effects of shading and line representations. The renderer and stimuli will be rearranged according to a latin square design.

3) Stimuli: This study is interested in the effects of the rendering methods on two axes of stimuli. First, the models used should cover a wide array of distinguishability. This can range from easily recognizable things like a bike to hard objects like a golf ball. Second, the scene should be composed of a variable number of objects of the same distinguishability. Combined, the results will tell us how the effectiveness of a given renderer at representing complex objects and large scenes.

4) Expectations: The vector line representations tend to be more simplistic and cleaner than other methods which may improve recognition speeds. However, certain objects are better suited to be distinguished by the algorithm than other objects. I believe it will perform better on larger scenes but worse on complex objects.

V. Conclusion

A lot of research has been conducted in the field of NPR focusing on rendering techniques for stylized lines. The work presented here has used these techniques and expounded upon them to emulate the look of a vector monitor. The final application generates adjacency information, computes a depth texture, runs a shader program to detect and render the vector-styled lines, and finally applies a bloom effect. Then, this implementation was analyzed on its objectives and flaws. Finally, a possible perceptual study was suggested.

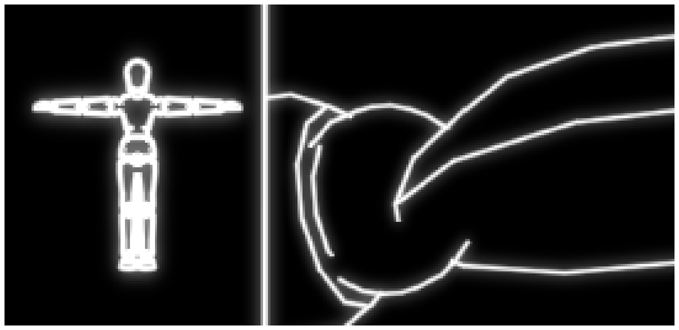

Fig. 7. Improvable cases of a faraway model (left) and intersecting geometry (right).

Despite meeting its targets, this work still has several possible extensions to create a better implementation. As the lines are constant width, a model that is far away will seem cramped (Figure 7, left). The model could be displayed more sparsely by selecting edges more restrictively.

The test models used for this paper were conducive towards the application of the algorithm. However, other models will not have as many distinguishing lines. This could be solved by implementing some of the other lines described by Al-Rousan et al.[4]. Edges can also be drawn at lines of intersecting geometry as exhibited where the hand meets the wrist in Figure 7. The solution unfortunately is outside the scope of this project as it would require advanced intersection tests to efficiently calculate the intersection paths of the meshes.

Since the edges are taken directly from the underlying models, curved lines appear jagged. A method for smoothing these paths like presented in Wang et al.[16] could be applied.

For the most part the overdraw method hides the gaps between edges. But, at T-intersections of lines the overdrawn part will stick out as seen in the top of the wrist in Figure 7.

Finally, in the current implementation all the lines are pure white. Instead the scene could be made more interesting by varying color and brightness. Even basic shading would be possible.

References

[1] R. Sayeed and T. Howard, “State of the Art Non-Photorealistic Rendering (NPR) Techniques,” in *Theory and Practice of Computer Graphics 2006*, L. M. Lever and M. McDerby, Eds. The Eurographics Association, 2006.

[2] P. Hermosilla and P. P. Vázquez Alcocer, “Single pass gpu stylized edges,” in *IV Iberoamerican Symposium in Computer Graphics*, 2009, pp. 47–54.

[3] J. Doss. (2008, Aug) Inking the cube: Edge detection with direct3d 10. [Online]. Available: http://www.gamasutra.com/view/feature/1644/sponsored_feature_inking_the_.php?print=1

[4] R. Al-Rousan, M. S. Sunar, and H. Kolivand, “Stylized line drawings for shape depiction,” in *2015 4th International Conference on Interactive Digital Media (ICIDM)*, Dec 2015, pp. 1–5.

[5] P. Rideout. (2010, Oct) Silhouette extraction. [Online]. Available: http://prideout.net/blog/?p=54

[6] F. Cole and A. Finkelstein, “Two fast methods for high-quality line visibility,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 5, pp. 707–717, Sept 2010.

[7] NanaOn-Sha, “Vib-ribbon,” PlayStation, Dec 1999, video demo: https://youtu.be/HhYa1WyeVsE.

[8] Gattai Games, “Muffled warfare,” PC, May 2018, https://www.muffledwarfare.com/.

[9] B. Bernardoni. (2018, May) Project blog. [Online]. Available: http://bbernardoni.com/DGIProject/

[10] D. Rákos. (2010, Sept) Efficient gaussian blur with linear sampling. [Online]. Available: http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/

[11] C. Gatzidis, S. Papakonstantinou, V. Brujic-Okretic, and S. Baker, “Recent advances in the user evaluation methods and studies of non-photorealistic visualization and rendering techniques,” in *2008 12th International Conference Information Visualisation*, July 2008, pp. 475–480.

[12] R. L. Mandryk, D. Mould, and H. Li, “Evaluation of emotional response to non-photorealistic images,” in *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Non-Photorealistic Animation and Rendering*, ser. NPAR ’11. New York, NY, USA: ACM, 2011, pp. 7–16. [Online]. Available: https://dl.acm.org/citation.cfm?id=2024678

[13] T. Isenberg, P. Neumann, S. Carpendale, M. C. Sousa, and J. A. Jorge, “Non-photorealistic rendering in context: An observational study,” in *Proceedings of the 4th International Symposium on Non-photorealistic Animation and Rendering*, ser. NPAR ’06. New York, NY, USA: ACM, 2006, pp. 115–126. [Online]. Available: https://dl.acm.org/citation.cfm?id=1124747

[14] S. Kim, H. Hagh-Shenas, and V. Interrante, “Conveying shape with texture: experimental investigations of texture’s effects on shape categorization judgments,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 10, no. 4, pp. 471–483, July 2004.

[15] M. C. Velez, D. Silver, and M. Tremaine, “Understanding visualization through spatial ability differences,” in *VIS 05. IEEE Visualization, 2005.*, Oct 2005, pp. 511–518.

[16] L. Wang, C. Tu, W. Wang, X. Meng, B. Chan, and D. Yan, “Silhouette smoothing for real-time rendering of mesh surfaces,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 3, pp. 640–652, May 2008.