Non-Photorealistic Shading Techniques for Anime-Style Rendering

Andy Qu
University of Maryland - College Park
aqu1@terpmail.umd.edu

Shuhong Chen
University of Maryland - College Park
shuhong@cs.umd.edu

Matthias Zwicker
University of Maryland - College Park
zwicker@cs.umd.edu

Abstract

Shaders have historically played a crucial part in drawing 3D models onto a computer screen as they have been an important component in computer games, movies, and simulations. As a result, various well-known shading methods and techniques have been used to suit a multitude of purposes and environments, including Flat, Gouraud, and Phong shading. Of the many shading methods, some can be used to mimic photorealism while others have been utilized to achieve different results, including a cartoon-ish depiction. In our work, we create GLSL and Python implementations of some of the classically known shading techniques as well as other non-photorealistic shaders that aim to mimic the Anime-like style.

1. Introduction

The ability to draw certain viewpoints from within a 3D environment has always been a foundational computer graphics task. With custom shading, the possibilities of how the images will turn out are endless. In order to produce such images, there must be information about the environment, models, and viewer already provided. If we think of the viewer as a perspective camera with a location that is facing a certain direction and the environment as a collection of vertex-edge meshes for each model, we can then store the entire 3D world as data within the computer. One can then define a shader [7], or shading script, which acts as the process in which the computer will figure out which models are in view of the camera, determine where each model should be in the final picture, and how each model is to be drawn out. For photorealistic effect, some shaders take into account lighting information from the environment and integrates this into the way certain models are drawn. Lambertian lighting [4], for example, is a commonly used technique that handles lighting in that it assumes that if the direction of the light faces perpendicularly to any point on a models surface, that point should then reflect the most amount of light in all directions. Shaders [7] can also be used to achieve nonphotorealistic results. One such example of this is Cel shading [3] which achieves a cartoon-ish output by summarizing ranges of colors into single shades, allowing the resulting image to appear more ”flat” rather than in 3D.

In our work, we conduct a study that surveys a couple pre-existing classical shading techniques [5]. We then move on to explore a few shading techniques that are geared towards creating cartoon-ish results, such as Cel shading [3] and contour line drawing. We implement all such shaders using ThreeJS [9], a popular JavaScript library for creating 3D computer graphics using WebGL. Since ThreeJS includes functionality for users to write custom shaders in GLSL, the shading language for WebGL, it proved to be the perfect platform for implementing custom techniques. For some of our shaders involving multiple passes, Python, along with some of its libraries, was also used to circumnavigate some of the limitations of ThreeJS. In summary, we contribute:

- A study that surveys classical shading techniques [5], which include Flat, Gouraud, and Phong shading, all using Lambertian lighting [4].
- A study of a few shading techniques that aim to create cartoon-ish results, such as Cel shading [3], silhouette drawing, contour line drawing, and suggestive contour drawing [2].
- An implementation of all such shaders using ThreeJS [9], GLSL, and Python.

2. Related Work

There are a lot of helpful descriptions of each shading technique found on Wikipedia [5] as it describes the basic idea behind each of them. This website from CGLearn also helps illustrate some of the key differences between the classical shading methods we implemented in the work [1], including the varying ways in which each method interpolates normal vectors along the surface of a model's mesh and how each method blends the coloring from nearby faces or vertices. We also followed nonphotorealistic shading techniques that involved the drawing of suggestive contour lines in order to help convey shape as outlined by DeCarlo *et al.* [2].

3. Methodology

We provide motivation and methodology for our implemented shading methods and concepts: Lambertion lighting [4], Flat shading [8], Gouraud Shading [8], Phong Shading [8], Cel Shading [3], Silhouette or outline drawing [3], Contour line drawing [2], and Suggestive contour line drawing [2]. The resulting images from all of the implemented techniques will be shown in Sec. 4.

3.1. Lambertion Lighting

As described before, the main idea behind Lambertion lighting [4] is that it makes the assumption that if the light hits any point on the surface on the model directly perpendicularly, then those points are where the light will be reflected the most powerfully in all directions. Consider an arbitrary point on the surface of the model. If we were to define a light-direction vector, l, and another vector, n, representing the surface normal vector at that point, then the strength of reflected light can be calculated from the following expression:

$$max((n \cdot l), 0) \tag{1}$$

Assuming that both n and l are both normalized, the result of this expression would then be a scalar value that can be multiplied to the color of the light in order to obtain the final color of that point. All of the shading techniques implemented in the work will use this lighting system to base color calculations for each point on the model's surface.

3.2. Flat Shading

This is the first shading technique that introduces a method of determining normal vectors, n, for each point on the surface of a model. If a model is represented as a mesh of edges and vertices that are along the model's surface, the area between neighboring edges and vertices would form a face. Flat shading [1] simply computes the normal of each face of the mesh and assumes that every point on the same face should have the same normal vector, n, as shown in Fig. 1.

As a result, each face on a model's mesh would appear strictly the same color. The edges between neighboring faces can be clearly seen as long as both neighbors do not lie on the same plane.

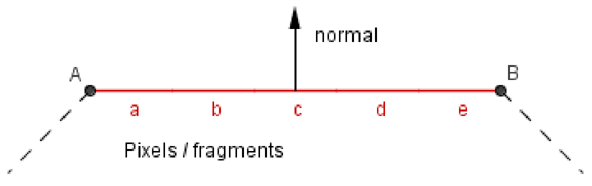


Figure 1. Example of the normal vector being the same across all points on the same face of the mesh. Taken from CGLearn [1].

GLSL shaders in ThreeJS have two parts, one running after the other while passing information in a sequential manner. The first part is the vertex shader that runs once per vertex in the mesh that is being drawn. The role of the vertex shader is to make vertex calculations based off of the information on that particular vertex, while passing the resulting information onto the next part. The second part is then the fragment shader which runs once per screen pixel the model must be displayed on. This part must take in all of the information that was passed from the vertex shader and calculate the final color of that particular pixel. Because ThreeJS uses vertex shaders, as opposed to surface shaders, the computer only knows the normal vectors of each vertex instead of each face. However, ThreeJS automatically interpolates vertex positions across the entire surface of the mesh. Therefore, we can derive the face normal from the cross product between the derivatives of interpolated positions, p, along the x and y directions of the view-space, as shown in Eq. (2).

$$\frac{\partial p}{\partial x} \times \frac{\partial p}{\partial y} \tag{2}$$

3.3. Gouraud Shading

This shading technique introduces another system of determining model normal vectors. Instead of generalizing entire faces to the same normal vector, Gouraud shading uses the normal vectors of vertices to calculate the final color of each vertex [8]. The faces that are between the vertices are then colored by linearly interpolating the colors from their surrounding vertices [1], as shown in Fig. 2.

Resulting images of Gouraud shading seem to be more "smoothly" shaded than images of Flat shading. The one disadvantage of Gouraud shading is that the mesh edges have a more distinct color than the faces. This is due to the colors of pixels lying on mesh edges only having to linearly interpolate between the two colors of the vertices the edge connects while other pixels must derive their colors from more than two vertices.

Since ThreeJS already calculates vertex normals and in-

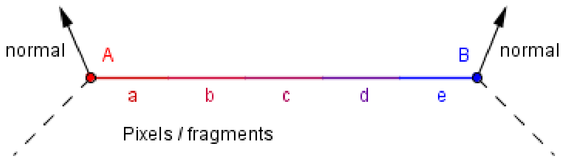


Figure 2. Example of vertices having their own color based off of vertex normals. Colors are then interpolated across the face between vertices. Taken from CGLearn [1].

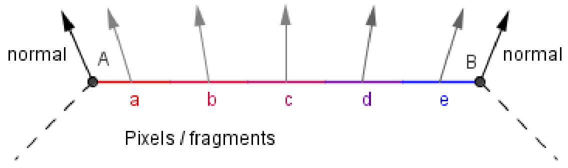


Figure 3. Example of interpolated vertex normals across the mesh surface. Taken from CGLearn [1].

terpolates colors between vertices by default, implementing Gouraud shading was simply a matter of computing the Lambertian lighting factor [4] and applying the resulting color to each vertex.

3.4. Phong Shading and Lighting

Not only does this shading technique introduce yet another system of determining surface normal vectors, it also acts as a new lighting system [1]. Phong shading aims to create more photorealistic results and improves upon Flat and Gouraud shading by first interpolating the vertex normals themselves across the surface of the mesh, as shown in Fig. 3. This means that each pixel ultimately gets its own normal vector which can then be used to calculate its own Lambertian lighting factor [4], independent from face or vertex normals. This results in an even "smoother" looking result.

Secondly, to add even more photorealism, Phong shading sums up the colors calculated from three different light calculations. Not only does Phong shading take into account Lambertion (or diffuse) lighting, but also blends in colors from ambient and specular lighting. We define ambient light as a color that every item in the environment should be colored in unconditionally as a baseline. We then define specular light as how much the reflected light bounces directly to the viewer, as shown in Fig. 4. Consider an arbitrary point on the surface of a mesh with its own surface normal vector, n, and an incident light vector, l. If we define r as the vector representing the reflection of l on n and v as the vector from the point on the surface to the viewer

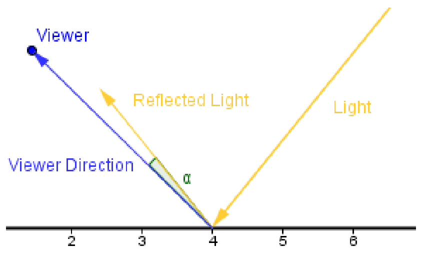


Figure 4. Example of reflected light heading towards the viewer. Taken from CGLearn [1].

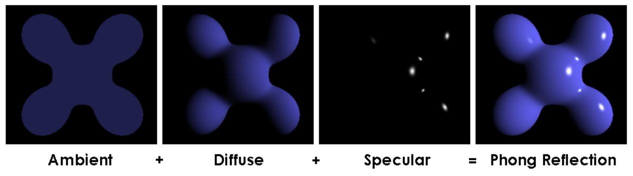


Figure 5. Example of combining the lighting components in Phong shading. Taken from Wikipedia [6].

or camera, the following expression can be used to calculate the degree of specular light:

$$\begin{cases} max(r \cdot v, 0) & \text{if } max(n \cdot l, 0) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Once the ambient, diffuse, and specular light values have been summed up, the resulting drawing of the model not only looks smooth, but also "shiny" in areas facing towards the light source, as shown in Fig. 5. The scalar result of this expression in Eq. (3) is often raised to a power to lessen the area the area that "glistens" due to specular lighting.

3.5. Cel Shading

Apart from the previously aforementioned shading techniques, Cel shading aims to create more cartoon-ish and nonphotorealistic results. We use Cel shading as our first step towards Anime-like depictions of 3D models. As mentioned before, Cel shading summarizing areas that have similar colors as single shades [3]. Our implementation of Cel shading summarizes possible values of Lambertian lighting factors into five distinct categories covering equal ranges each, resulting in five different summary values. The implementation follows the expression found below:

$$\begin{cases} 1 & \text{if } (n \cdot l) * 0.5 + 0.5 > 0.8 \\ 0.75 & \text{if } (n \cdot l) * 0.5 + 0.5 > 0.6 \\ 0.5 & \text{if } (n \cdot l) * 0.5 + 0.5 > 0.4 \\ 0.25 & \text{if } (n \cdot l) * 0.5 + 0.5 > 0.2 \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

As a result, models that are Cel shaded are only drawn with a limited palette of colors. This makes the model seem more ”flat” than 3D.

3.6. Silhouette Drawing

As we move towards the goal of drawing 3D models in a cartoon-ish Anime-like manner, the ability to draw lines onto the model become more important. We use silhouette drawing as means to mimic the overall cartoon style since we find that objects in cartoons likely show their own outlines along with its coloring. Since ThreeJS allows for the rendering of one scene on top of another, our work uses this as means to do multiple passes on the same object. Our implementation for silhouette drawing is split into two passes:

- The first pass perturbs the model’s mesh along its vertex normals and shades the model as one solid color.
- The second pass shades its own copy of the mesh using Cel shading on top of the first pass.

The first pass simply lays down a slightly bigger shape so that it can be covered by the Cel shaded second pass. The resulting render would then include both the Cel shaded model and a small outline around the area, depicting the edges of the figure in the image.

3.7. Contour Line Drawing

While we aim to have models appear less 3D, we still want to mimic an artists portrayal of a 3D object. Just as an artist would draw contour lines to convey shape, we implement our shaders to do the same. Given an arbitrary point on the surface of a model, we define the point to be a contour if the dot product between the surface normal, n, and its vector towards the viewer, v, is equal to zero, namely:

$$n \cdot v = 0 \tag{5}$$

This implies that at such points, the normal vector is directly perpendicular to the vector that is directed towards the viewer. This means that to the viewer, the point is on a crease of the surface to which the viewer cannot see the other side of. If these points are colored differently, they can potentially form a contour line as shown in Fig. 6.

The only caveat to this is that when calculating the dot products between n and v for each pixel, the pixels that were meant to be a part of the contour line did not have dot products of exactly zero. Thus, we add flexibility to the condition stated in Eq. (5), counting pixels whose dot product values fall between a certain range around zero as part of the contour as well. This results in thicker, more continuous contour lines that effectively convey the model’s shape even when it is drawn like a cartoon.

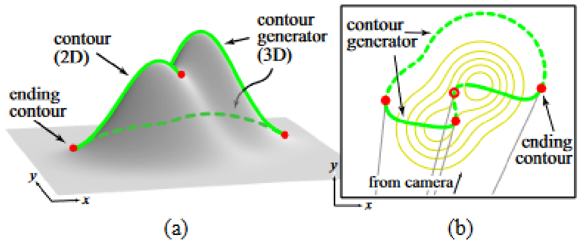


Figure 6. (a) Example of a contour line being projected to an image. (b) A topological map of the surface shown in (a) with the contour line drawn relative to the camera. Taken from DeCarlo *et al.* [2].

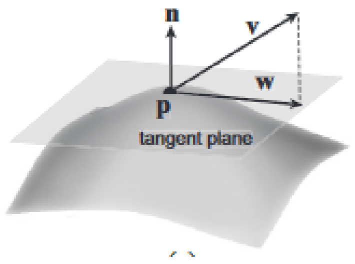


Figure 7. Example w vector from projecting v onto the tangent plane defined by n. Taken from DeCarlo *et al.* [2].

3.8. Suggestive Contour Line Drawing

The final shading technique we implement proves to be a bit more complex than the rest. In addition to contour lines, we look to also include suggestive contour lines to increase expressiveness of the original shape of the model. Suggestive contours are a bit harder to define, so we follow the definition as stated by DeCarlo *et al.* [2] as follows:

$$D_w(n \cdot v) = 0, \quad \text{and} \tag{6}$$

$$D_w(D_w(n \cdot v)) > 0 \tag{7}$$

The conditions from Eq. (6) and Eq. (7) denote the local minima of the dot product, $n \cdot v$, in the direction of a new vector, w. Given an arbitrary point on the surface of a model, let n be the surface normal of that point and v be the vector from that point to the viewer. Then w is the projection of v onto the plane defined by n, namely the tangent plane of that point on the surface as shown in Fig. 7.

Calculating $D_w(n \cdot v)$ is quite simple in GLSL since the language offers functionality to compute derivatives of certain expressions based off of finite differences between neighboring pixels in the x and y directions of the view space. We calculate w by using the following equation:

$$w = v - ((n \cdot v) \cdot n) \tag{8}$$

We then use the w from Eq. (8) to calculate $D_w(n \cdot v)$ by this following formula based on the definition of the directional derivative:

$$(\nabla(n \cdot v)) \cdot \frac{w}{||w||} \tag{9}$$

We note that even though the w vector is 3-dimensional, we can only take the derivative of $n \cdot v$ in only the x and y directions of the view-space. So, we assume that the derivative of $n \cdot v$ is zero in the z direction since there should be no difference in the dot product as we move along z. Moving along z implies jumping off of the surface of the model, therefore we choose to neglect such values in our calculations.

Taking the second degree directional derivative proved to be a bit trickier since GLSL did not have the functionality to calculate higher order derivatives. While we would have liked to implement this shading technique on GLSL because of its support for GPUs, we had to move a portion of our implementation to Python since it allows us to access information pertaining to neighboring pixels. With calculations being split across the two seperate platforms, we needed some way for the two parts to communicate. Thus, we implement another multipass approach:

- The first pass outputs a texture based on the normal vectors of the model's surface.
- The second pass outputs a texture based on the vectors from each pixel to the viewer.
- The third pass outputs a texture based on the w vectors calculated from the n and v vectors at each pixel.
- The final pass takes all the results from the previous passes and calculates which pixels should be drawn as a suggestive contour based on the finite differences between neighboring pixels.

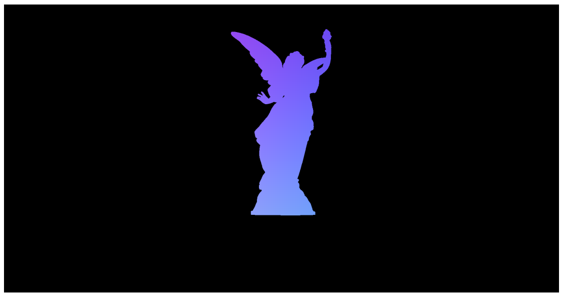
The only pass that must be computed in Python is the fourth and final pass, while all other passes can be done by the GLSL side. This final pass calculates the first order directional derivative by finding the following forward finite difference of $n \cdot v$:

$$D_w(n \cdot v) = \begin{pmatrix} n_{x+1} \cdot v_{x+1} - n \cdot v \\ n_{y+1} \cdot v_{y+1} - n \cdot v \end{pmatrix} \cdot \frac{w}{||w||} \tag{10}$$

We can then define the second order derivative as the following backward finite difference of $D_w(n \cdot v)$:

$$D_w^2(n \cdot v) = \begin{pmatrix} D_w(n \cdot v) - D_w(n_{x-1} \cdot v_{x-1}) \\ D_w(n \cdot v) - D_w(n_{y-1} \cdot v_{y-1}) \end{pmatrix} \cdot w \tag{11}$$

(a) Example output from first pass, with normal vectors shown.


(b) Example output from second pass, with view vectors shown.

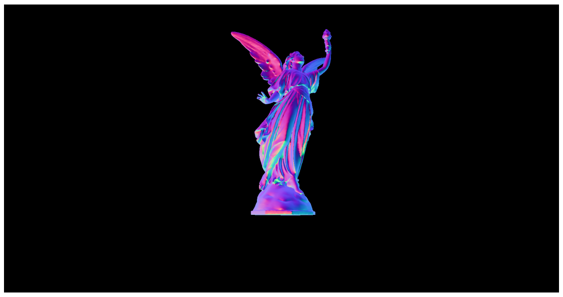

(c) Example output from third pass, with w vectors shown.

Figure 8. Outputs from the intermediate passes for calculating suggestive contours. These passes were run on the Stanford Lucy model.

By using the expressions found in Eq. (10) and Eq. (11), we only need to access directly adjacent neighbors surrounding the pixel we are calculating derivatives of. We run this calculation on all pixels that are not on the edge of the image, taking into account the w, v, and n vectors for each pixel, in order to determine which pixels should be drawn as a suggestive contour. Just like in drawing regular contour lines, some derivative values do not fit the exact conditions from Eq. (6) and Eq. (7). Thus, we set a range around the correct values to allow for flexibility, resulting in thicker, more continuous suggestive contour lines. We show an example of outputs of the three intermediary passes of our implementation of suggestive contour shading on the Stanford Lucy in Fig. 8.

With the implementations of all the previously mentioned shading techniques complete, we combine all of the nonphotorealistic shaders together to achieve our goal of depicting 3D models in an Anime-like style.

4. Results

We first show the results of our classical shaders on more basic 3D shapes that are provided to us by default from the ThreeJS library. Since ThreeJS also includes its own proprietary classical shaders, we show a comparison between the results from our implementation and their implementations. We then move on to show the effects of each of our nonphotorealisitic shaders individually before combining all of them together. We decided to show our nonphotorealistic shaders on more complex 3D models since they are more detailed than the simple shapes ThreeJS includes by default, making them more likely to have contours and suggestive contours on any given angle of view. We also wanted these 3D models to have as high of a polygon count as possible so that our shader thresholds would work better for them.

As demonstrated in Fig. 9, Fig. 13, and Fig. 17, flat shading only shades the 3D shape based on the normals of the faces in its mesh, making all of the pixels in each face of the model share the same color. This results in flat shading giving the models the simplest look. In Fig. 10, Fig. 14, and Fig. 18, we show that because its dependence on vertex normals, the results of Gouraud shading display the vertices and edges of a mesh faintly different from the rest of the model's surface. However, this issue is lessened if the polygon resolution of the model is increased, since there would be more edges and vertices to blend colors in with. Even with its disadvantages, Gouraud shading still gives a much smoother look to the model than flat shading. We further improve this smoothness with Phong-interpolated normals, as seen in Fig. 11, Fig. 15, and Fig. 19. With interpolated vertex normals, each pixel no longer needs to depend on the color of its neighboring vertices since each pixel will have its own normal vector to calculate its own color. This rids of the disadvantages of only using vertex normals as Gouraud shading does while under the same Lambertion lighting system. We show that we can add another flavor to the 3D models by using Phong shading, as displayed in Fig. 12, Fig. 16, and Fig. 20. With Phong lighting, shininess of objects can be simulated by adding in both specular and ambient light calculations, making the models look even more photorealistic. When comparing our implementations against the ThreeJS library, we show that were able to accurately reproduce the results that ThreeJS achieves with its shaders with the exception of our Phong shader. Though we were not able to mimic the exact "shine" of the resulting renders of the ThreeJS implementation, we were still able to replicate a shine that remains loyal to the purpose of the Phong reflection model [6].

In Fig. 21 and Fig. 22, we show the effects of each nonphotorealistic shader we implemented. We then combine all of their respective effects together to try to replicate the model's appearance in a cartoon-ish context. We finally test out our nonphotorealistic strategy to more complex models, including models inspired by popular Anime culture, to see whether our results replicate an Anime-like style. Since final result of including suggestive contours draws extra lines on the character's face, we show the difference between including and excluding suggestive contours, as seen in Fig. 23, Fig. 24, Fig. 25, and Fig. 26. We also test our nonphotorealistic shading techniques on a model of Amber from the game Genshin Impact and of Pekora Usada, a Hololive virtual YouTuber. Since both of these models are based off of Anime-styled characters, we tried our shaders on these characters to see if we have successfully mimicked the Anime-style 2D depiction.

5. Conclusion & Future Work

While we have shown implementations for nonphotorealistic shading in ThreeJS, GLSL, and Python, we could have improved our implementations even further. We were not able to completely implement our suggestive contour shader [2] on the ThreeJS platform due to limited access to data on neighboring pixels. Because of our decision to move to Python to support this shader, we lost the advantage of GPU parallelism for certain parts of the implementation. In order to load the results from the previous passes to Python for it to calculate suggestive contours, we import the images one by one, pixel by pixel. Even the calculations that are done in Python are also computed pixel by pixel. This process proves to be highly inefficient and not ideal. We look to the Python library, NumPy, to optimize the storing of image information since the data structures included in that library are highly optimized for large amounts of data that are of the same type. We can look toward further improving our implementation by vectorizing the calculations done on the pixel data stored in the NumPy structures since NumPy is heavily optimized for such operations. Future efforts could also be used to discover an implementation for suggestive contours that is based entirely on GLSL.

In conclusion, we provide a study on both classical and nonphotorealistic shading techniques and show each of their effects on various 3D models. We effectively use Cel shading [3] along with silhouette, contour line, and suggestive contour line drawing [2] to mimic the Anime-style depiction and display the combined effects on various models, including a few Anime characters. As a result, we are able to produce images that accurately imitates a cartoonish 2D look akin to the overall Anime-style. We have made our shader implementations publicly available at https://github.com/ShuhongChen/anime-shaders

References

[1] CGLearn. Computer graphics learning - shading and lighting. https://cglearn.codelight.eu/pub/

computer-graphics/shading-and-lighting. 2, 3

[2] Doug DeCarlo, Adam Finkelstein, Szymon Rusinkiewicz, and Anthony Santella. Suggestive contours for conveying shape. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 22(3):848–855, July 2003. 1, 2, 4, 6

[3] Wikipedia. Cel shading — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Cel%20shading&oldid=1082784221, 2022. [Online; accessed 05-May-2022]. 1, 2, 3, 6

[4] Wikipedia. Lambertian reflectance — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Lambertian%20reflectance&oldid=1071808627, 2022. [Online; accessed 05-May-2022]. 1, 2, 3

[5] Wikipedia. List of common shading algorithms — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=List%20of%20common%20shading%20algorithms&oldid=1077139001, 2022. [Online; accessed 05-May-2022]. 1

[6] Wikipedia. Phong shading — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Phong%20shading&oldid=1078655348, 2022. [Online; accessed 06-May-2022]. 3, 6

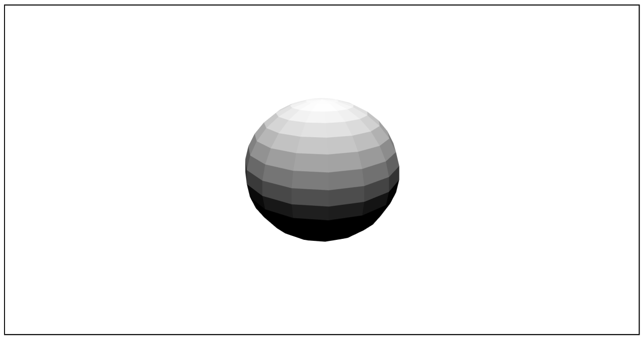
[7] Wikipedia. Shader — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Shader&oldid=1084249312, 2022. [Online; accessed 05-May-2022]. 1

[8] Wikipedia. Shading — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Shading&oldid=1057623637, 2022. [Online; accessed 06-May-2022]. 2

[9] Wikipedia. Three.js — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Three.js&oldid=1085408421, 2022. [Online; accessed 05-May-2022]. 1

A. Artist Attributions

- ajax.stl: https://cults3d.com/en/3d-model/art/bust-of-ajax
- amber.stl: https://hub.vroid.com/en/characters/3742574954744824945/models/3661281045858685259
- lucy.stl: https://www.thingiverse.com/thing:41939
- usada_pekora.stl: https://3d.nicovideo.jp/works/td67414
- Utah_teapot.stl: https://cults3d.com/en/3d-model/art/utah-teapot-solid


(a) Flat shaded sphere using our implementation.

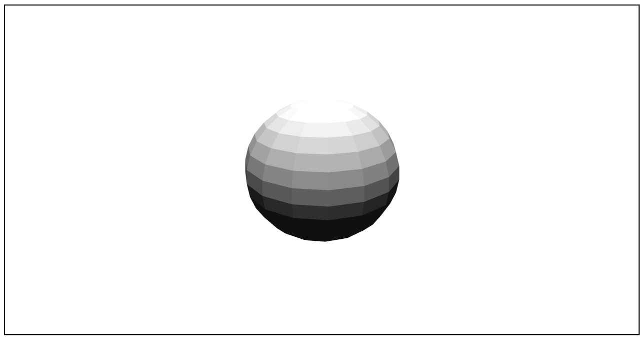
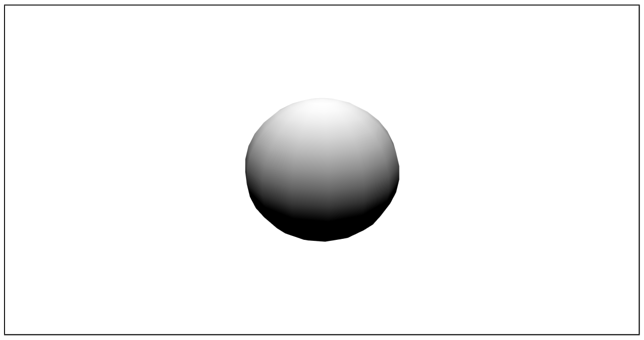

(b) Flat shaded sphere using ThreeJS implementation.

Figure 9. Example of Flat shading on a basic sphere mesh. Because flat shading only uses face normals, the faces of the mesh can be differentiated from each other.


(a) Gouraud shaded sphere using our implementation.

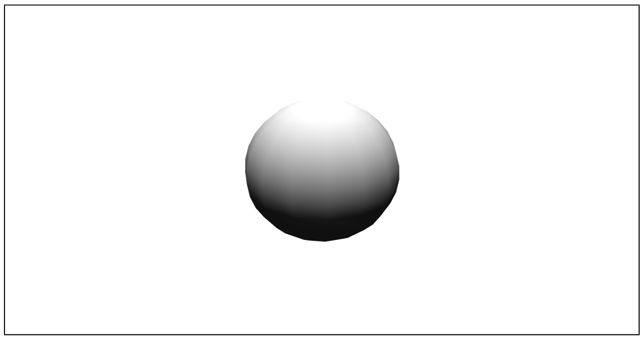
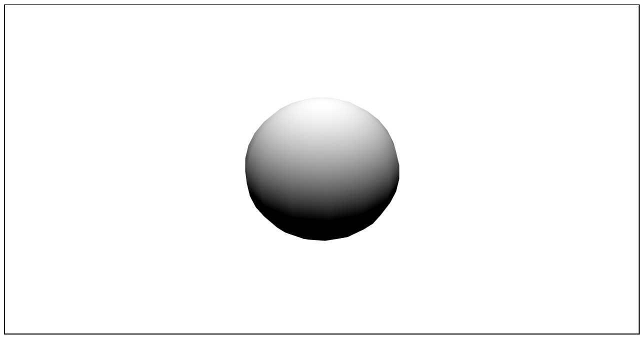

(b) Gouraud shaded sphere using ThreeJS implementation.

Figure 10. Example of Gouraud shading on a basic sphere mesh. Because Gouraud shading only uses vertex normals, the sphere looks ”smoother.” However, faint white lines along the mesh edges are still apparent since colors are interpolated between vertices. Edge pixels only interpolate between two vertices while face pixels must interpolate between three or more.


(a) Lambert shaded sphere using our implementation.

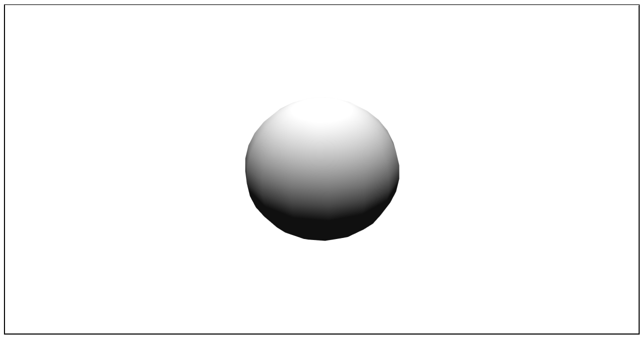
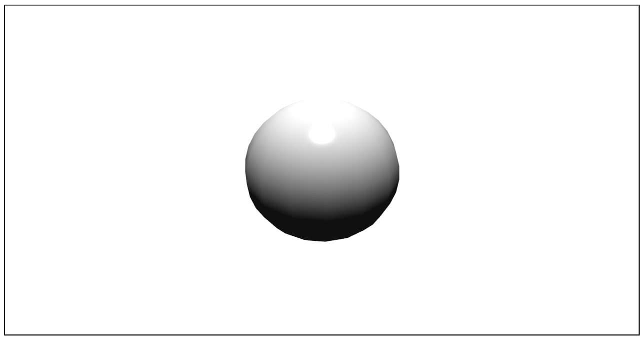

(b) Lambert shaded sphere using ThreeJS implementation.

Figure 11. Example of Lambert shading on a basic sphere mesh with Phong normals. Since this shader now uses interpolated normal vectors from vertex normals, the color of each pixel no longer relies on the color of neighboring vertices, giving the shape an even ”smoother” appearence.


(a) Phong shaded sphere using our implementation.

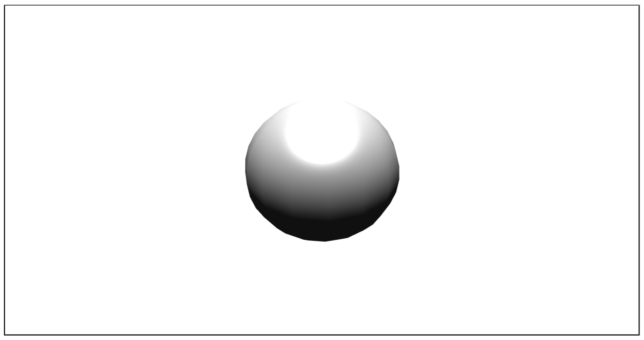
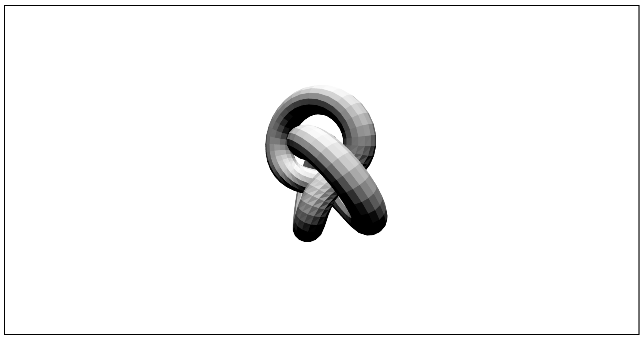

(b) Phong shaded sphere using ThreeJS implementation.

Figure 12. Example of Phong shading on a basic sphere mesh, using both Phong normals and Phong lighting. With both specular and ambient light added to the diffuse light calculated from Lambert techniques, the sphere now appears as if it had a ”shine.” While we did not exactly mimic the way in which the sphere ”shines” the way it does in the ThreeJS implementation, we decided that our implementation had a good enough accent of light, which already achieves the purpose of Phong shading.


(a) Flat shaded torus knot using our implementation.

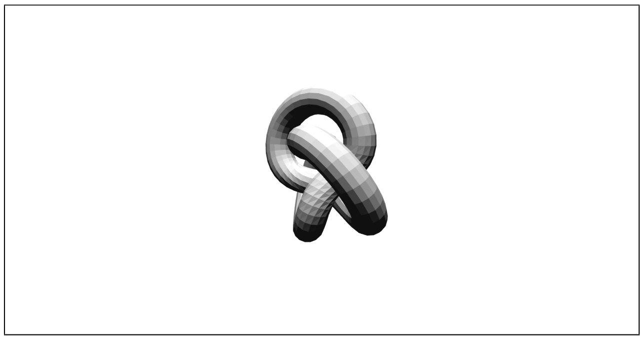
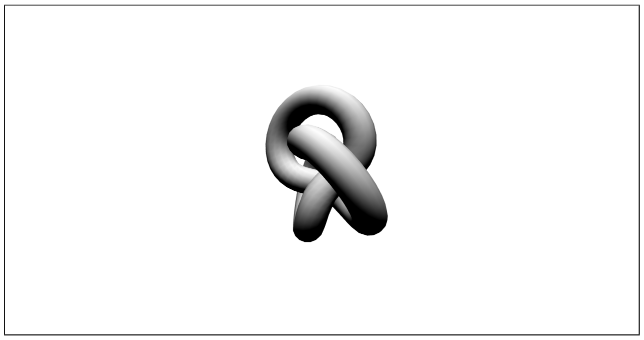

(b) Flat shaded torus knot using ThreeJS implementation.

Figure 13. Example of Flat shading on a basic torus knot mesh. Again, the faces of the mesh are more apparent under flat shading.


(a) Gouraud shaded torus knot using our implementation.

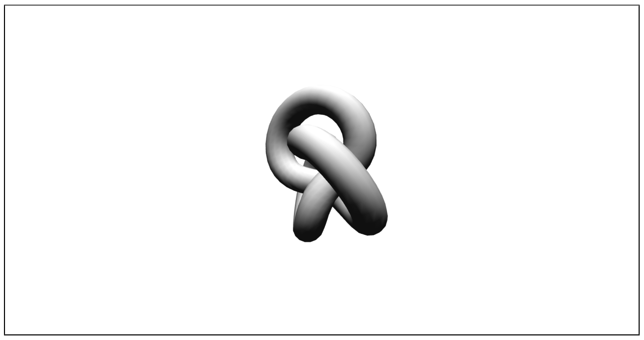
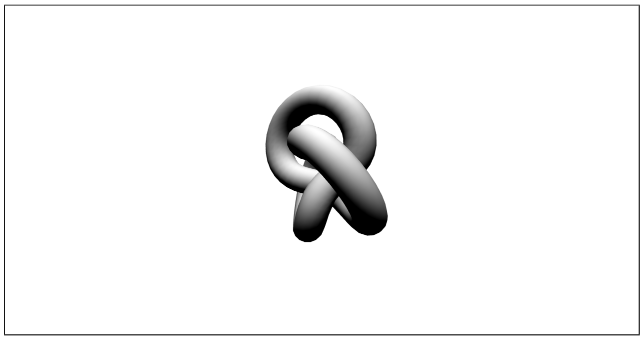

(b) Gouraud shaded torus knot using ThreeJS implementation.

Figure 14. Example of Gouraud shading on a basic torus knot mesh. Again, the edges of the mesh are more differentiable from the rest of the surface.


(a) Lambert shaded torus knot using our implementation.

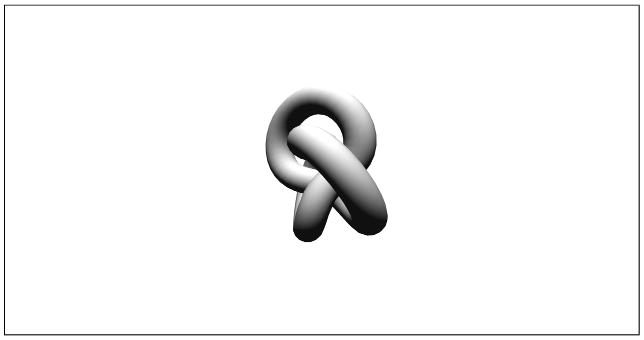
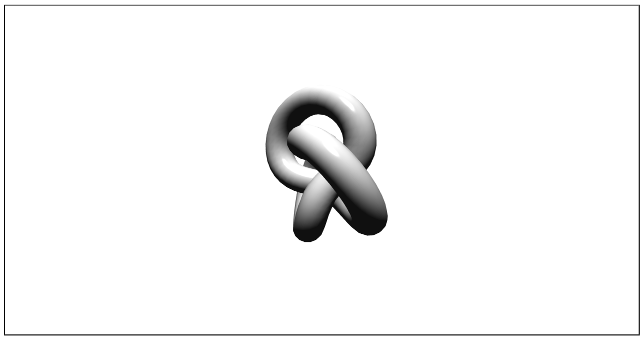

(b) Lambert shaded torus knot using ThreeJS implementation.

Figure 15. Example of Lambert shading on a basic torus knot mesh with Phong normals. Again, the interpolated normals in the shader gives the shape an ”smoother” appearence.


(a) Phong shaded torus knot using our implementation.

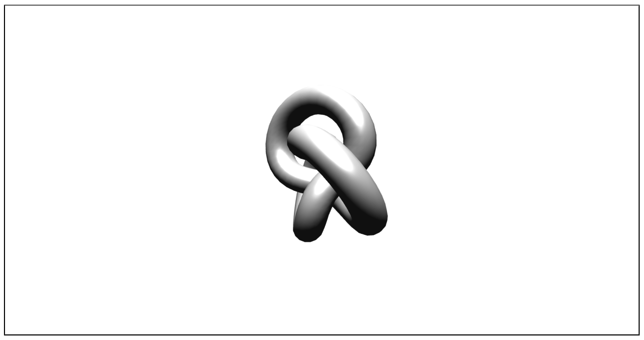
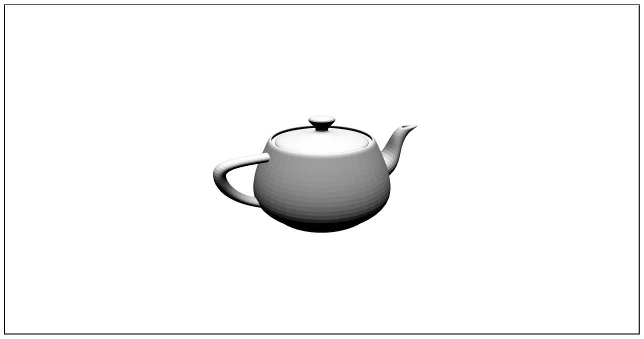

(b) Phong shaded torus knot using ThreeJS implementation.

Figure 16. Example of Phong shading on a basic torus knot mesh, using both Phong normals and Phong lighting. Again, the added layers of light gives the shape a ”shinier” appearence.


(a) Flat shaded teapot using our implementation.

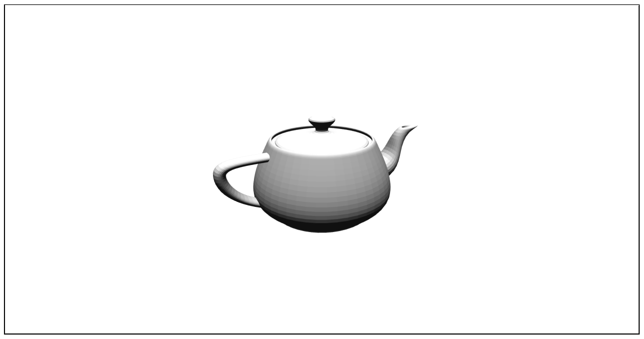
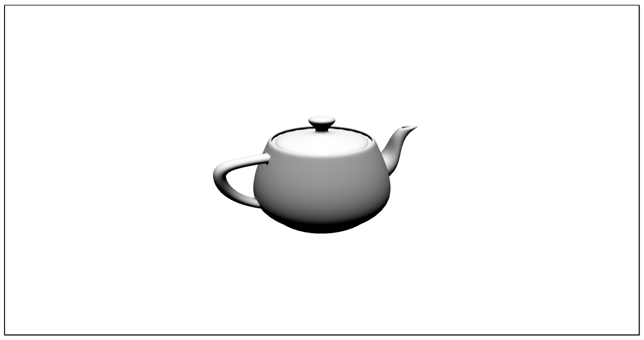

(b) Flat shaded teapot using ThreeJS implementation.

Figure 17. Example of Flat shading on the Utah Teapot mesh. Since this particular mesh has more polygons, the result shows a lot more faces, allowing the final rendering to appear more round.


(a) Gouraud shaded teapot using our implementation.

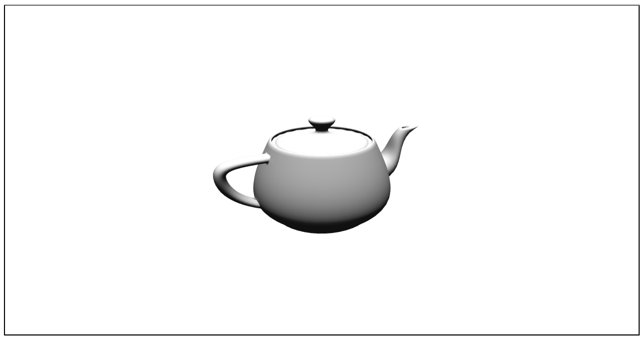
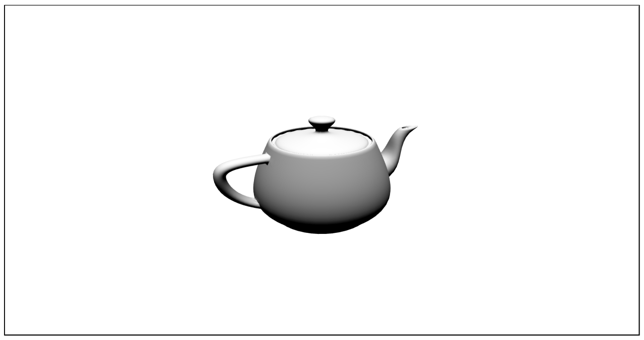

(b) Gouraud shaded teapot using ThreeJS implementation.

Figure 18. Example of Gouraud shading on the Utah Teapot mesh. Because this mesh has more polygons, edges are shorter and more frequent, allowing their distict color to be barely noticable. This changes the general color of the teapot allowing the main disadvantage of Gouraud shading to be somewhat bypassed.


(a) Lambert shaded teapot using our implementation.

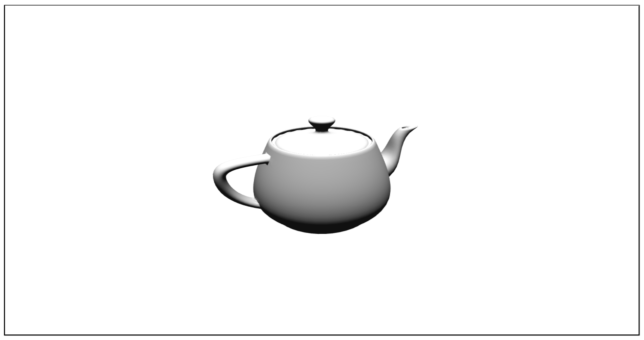
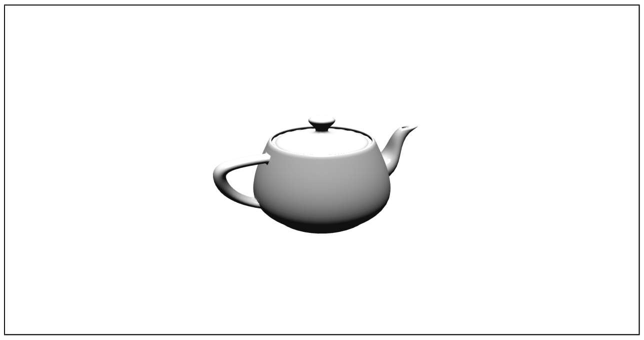

(b) Lambert shaded teapot using ThreeJS implementation.

Figure 19. Example of Lambert shading on the Utah Teapot with Phong normals.


(a) Phong shaded teapot using our implementation.

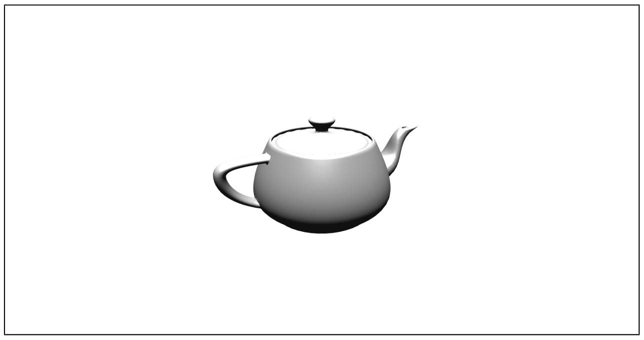
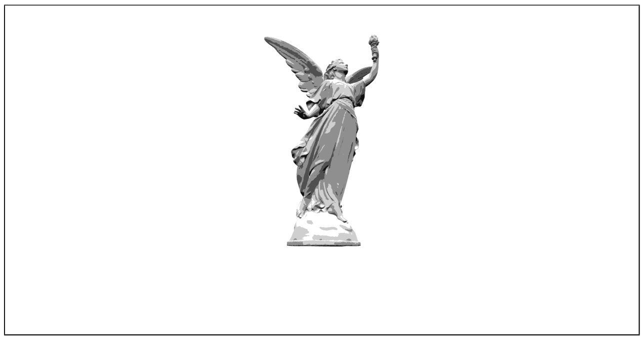
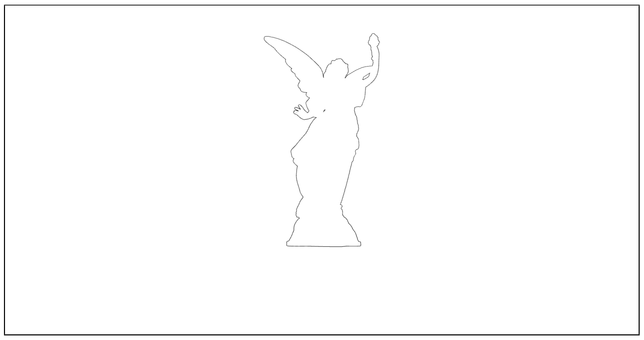
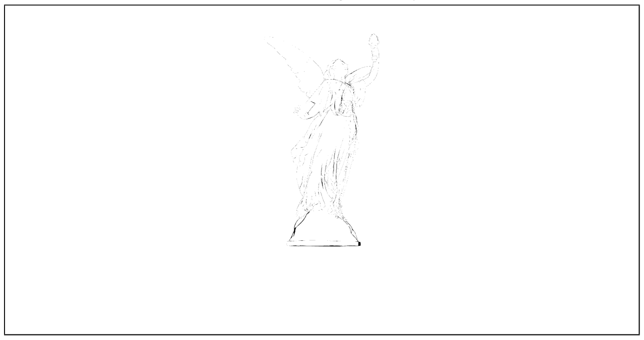
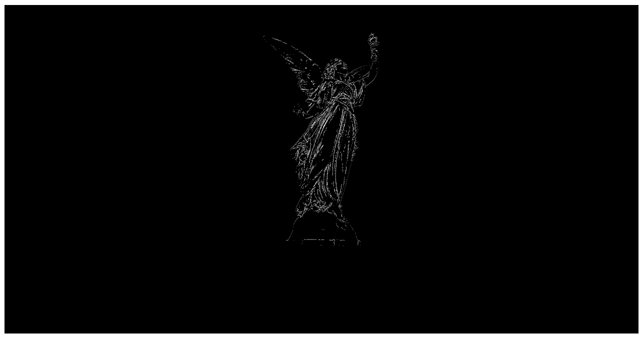

(b) Phong shaded teapot using ThreeJS implementation.

Figure 20. Example of Phong shading on the Utah Teapot, using both Phong normals and Phong lighting. The difference in the amount of shine between implementations becomes more important here. With more complex meshes, different degrees of shine start to dictate which parts of the mesh will shine from certain angles.


(a) Cel shading on Lucy.


(b) Silhouette shading on Lucy.


(c) Contour shading on Lucy.


(d) Final pass of the suggestive contour shader applied on Lucy.

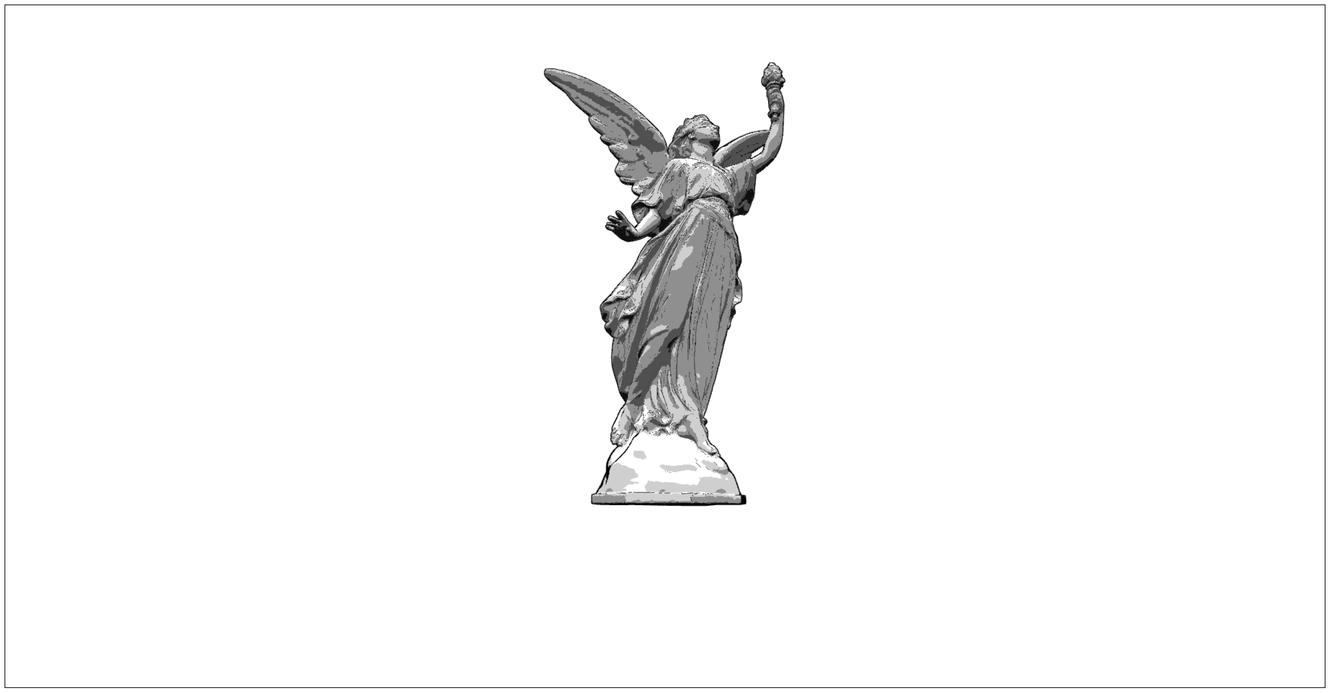
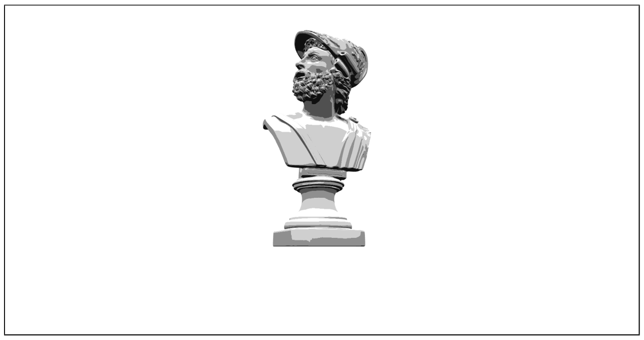
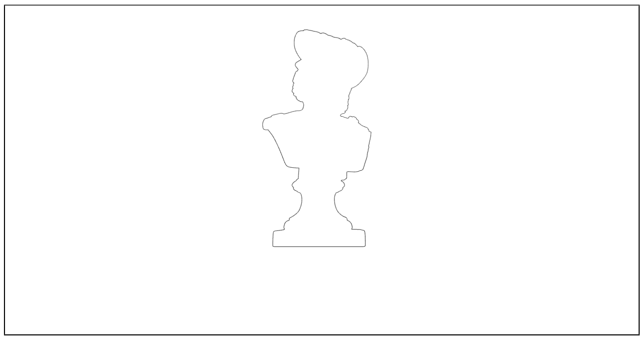
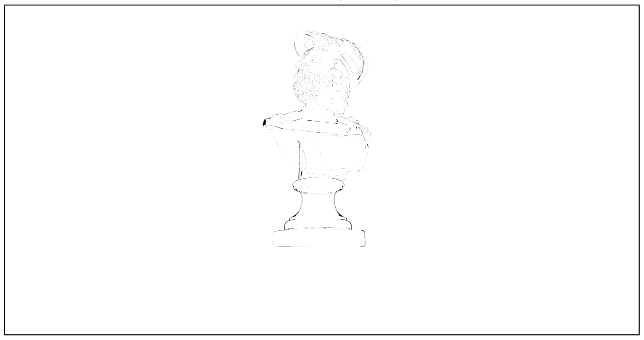
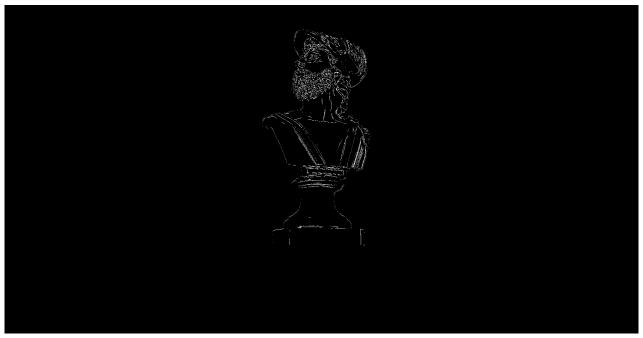

(e) All four of the nonphotorealistic shading techniques being applied to Lucy simultaneously.

Figure 21. Example of the nonphotorealistic shaders being applied to the Stanford Lucy model both individually and simultaneously. As you can see, the image seems to have a lot of extra lines drawn especially on the face of the figure


(a) Cel shading on Ajax.


(b) Silhouette shading on Ajax.


(c) Contour shading on Ajax.


(d) Final pass of the suggestive contour shader applied on Ajax.

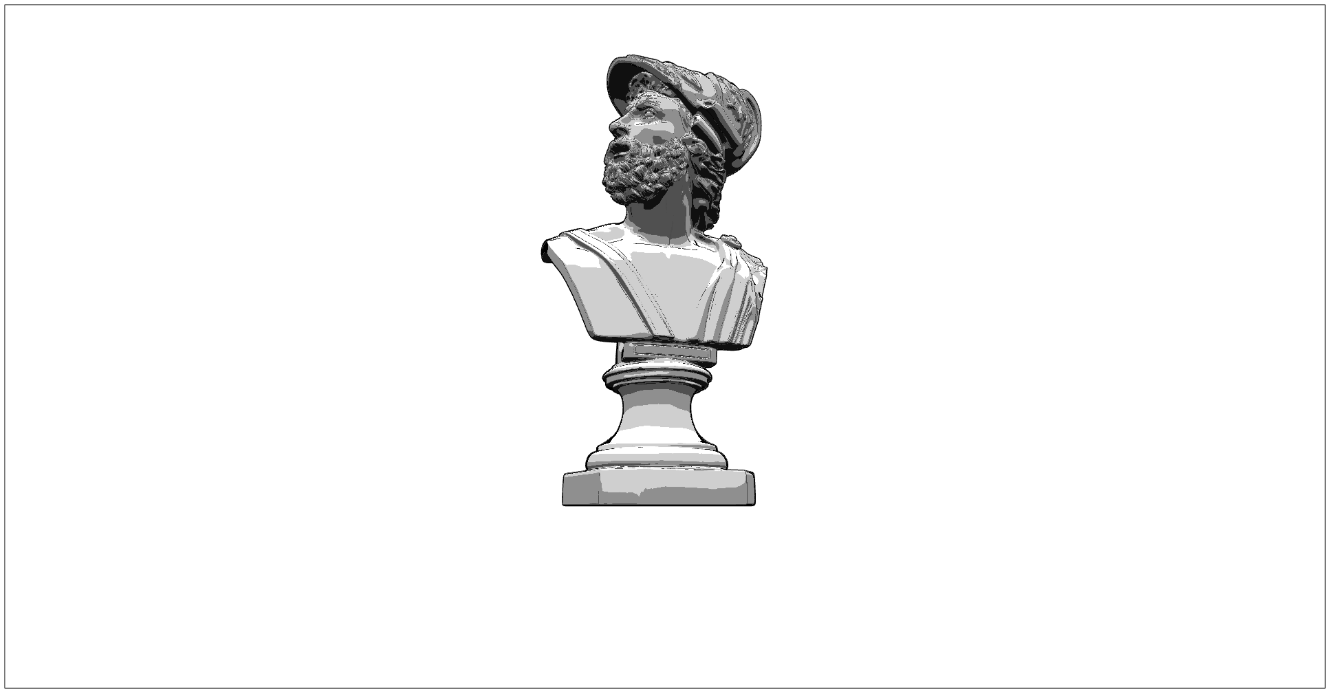
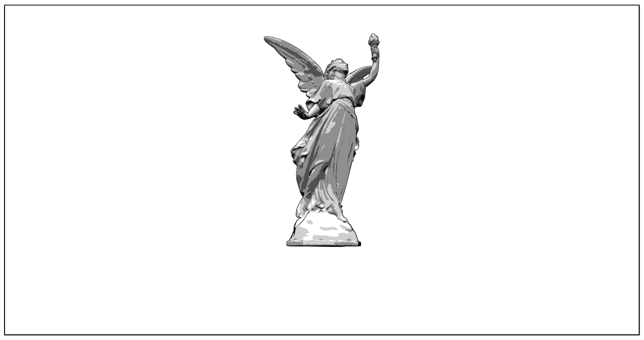

(e) All four of the nonphotorealistic shading techniques being applied to Ajax simultaneously.

Figure 22. Example of the nonphotorealistic shaders being applied to the Ajax bust model both individually and simultaneously. As you can see, the image seems to have a lot of extra lines drawn especially on the face of the figure.


(a) Lucy drawn without suggestive contours.

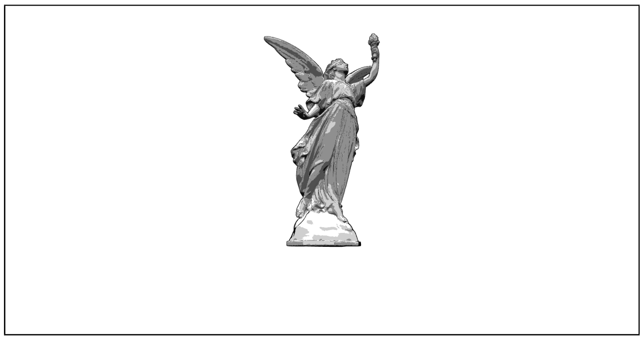
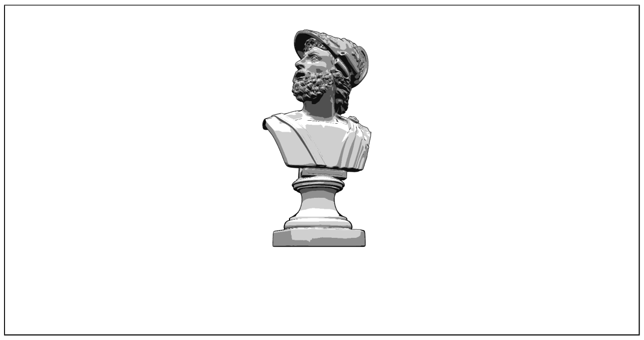

(b) Lucy drawn with suggestive contours.

Figure 23. Comparing Lucy drawn with all nonphotorealistic shading techniques with and without suggestive contours to see which result creates a more cortoon-ish image.


(a) Ajax drawn without suggestive contours.

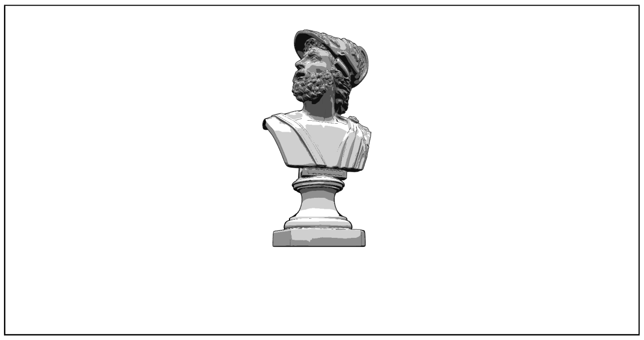
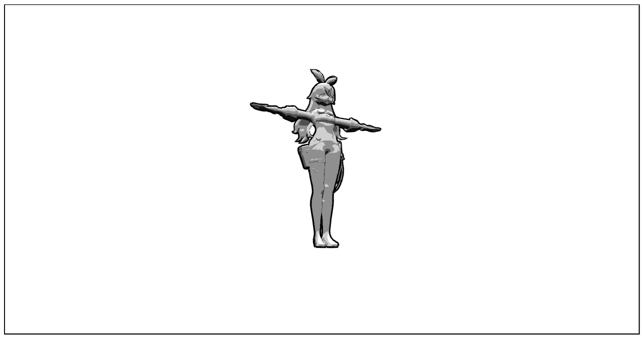

(b) Ajax drawn with suggestive contours.

Figure 24. Comparing Ajax drawn with all nonphotorealistic shading techniques with and without suggestive contours to see which result creates a more cortoon-ish image.


(a) Amber drawn without suggestive contours.

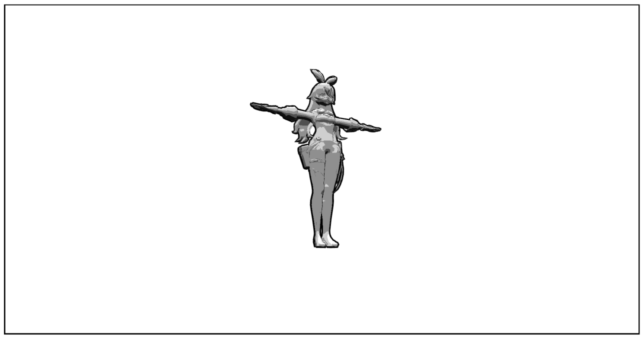
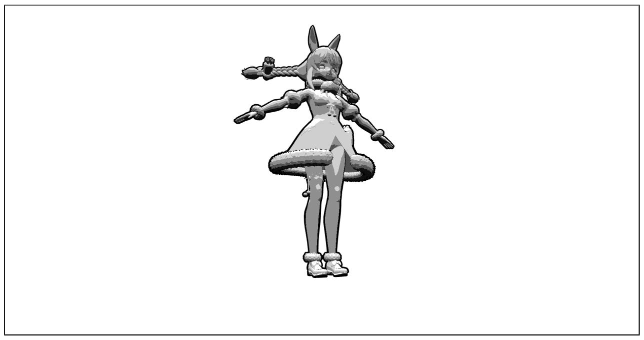
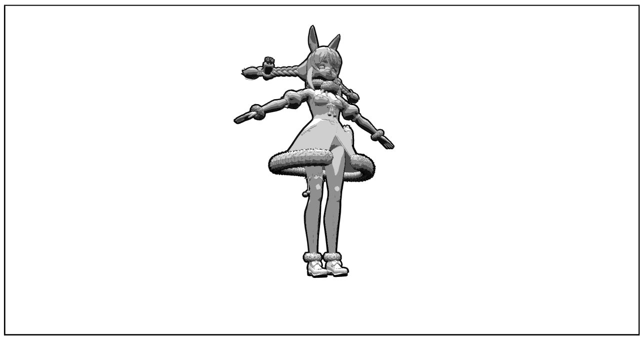

(b) Amber drawn with suggestive contours.

Figure 25. Comparing Amber drawn with all nonphotorealistic shading techniques with and without suggestive contours to see which result creates a more cortoon-ish image.



(a) Pekora drawn without suggestive contours.



(b) Pekora drawn with suggestive contours.

Figure 26. Comparing Pekora drawn with all nonphotorealistic shading techniques with and without suggestive contours to see which result creates a more cortoon-ish image.