
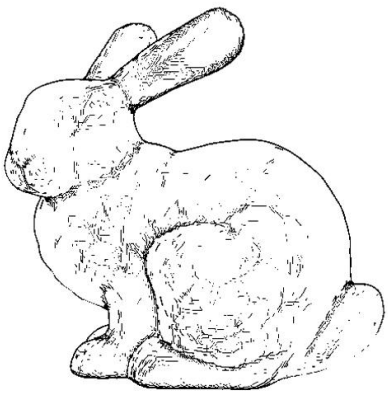


Object-Space Toon Shading

Lore Cox Mary DeVarney

Abstract

Non-photorealistic rendering provides ways to take complex objects and show them in often simplified, but artistic ways. This paper presents a toon shader that works within real time, and allows the user to show the model in multiple styles. Our renderer finds the silhouette edges, based on the camera position, along with the border and crease edges of the model. We then highlight these edges by creating new geometry to outline them. Adding in our shader, we are able to create and experiment with multiple artistic styles.

1 Introduction

Photorealistic rendering is not always the best look for a project. Stylization is important for being able to get across the proper feel, and emphasize the parts which should be at the forefront. A game heavily rooted in Japanese mythology such as Okami is much better served by a rendering style emulating ink paintings than it would be by a gritty realistic setting. Non-Photorealistic rendering also has the advantage of being better able to avoid the uncanny valley, a place in which things are so close to real that all people can see are the differences. When there is less information put forward, there are fewer things to notice that could go wrong. It also allows for more emphasis and exaggeration when animating, as the models will not be constrained to what people know as the typical limitations.

2 Related Works

The area of Non Photorealistic Rendering has developed many different techniques, but the one common theme is that the results are usually artistic in some way. One of the most common goals is to take an image or model and produce an image that looks as if it had been done by an artist. Georges Winkenbach and David H. Salesin's *Computer-Generate Pen-and-Ink Illustration* discusses how to render a pen-and-ink illustration using an automated rendering system. Michael P. Salisbury et al.'s *Interactive Pen-and-Ink Illustration* produces similar results; however they instead use a model and a library of stored stroke textures.

Lee Markosian et al.'s paper on *Real-Time Nonphotorealistic Rendering* is concerned with rendering simplified pictures of complex objects. They manage to make it real-time by focusing on speed at the expense of accuracy and detail. Their results are basic outlines of the objects, but they are able to add different styles and textures to the lines, giving each of them a different type of feeling.

Amy Gooch et al's *A Non-Photorealistic Lighting Model For Automatic Technical Illustration* takes objects and highlights the edges by adding lines, and expands on traditional shading techniques so that the results are a technical illustration you could find with any instruction manual. Their shaders can go from metal to phong with impressive results.

Although different parts of these works are similar, this paper will discuss how to create a toon shader by finding and creating new outlines by adding geometry to the model.

3 Object/Image Space Algorithms

It is important the difference between an object space algorithm and an image space algorithm. An image space algorithm uses the coordinate system of the image, usually using a system that is x pixels wide by y pixels high. The result is often some form of pixel map, and is found by determining which object is hit first by the projector when looking through each pixel.

An objects space algorithm uses the coordinate system of the objects and its environment. Unlike with the image algorithm, the object algorithm's results include the collection or objects and polygons that make up the scene. The outcome is determined by which parts of the objects are unobstructed by other objects or even itself. The correct color is then assigned. For this project, the object algorithm is used.

4 Silhouette Edges

Silhouette edge determination is carried out by analyzing each edge and its respective triangles with respect to the camera position. A silhouette edge has one adjacent face pointed towards the camera, while the other faces away. This can be obtained by analyzing the dot product of the face normal with the direction to the camera, producing the cosine of the angle between them. When one result is positive and the other negative, it is a silhouette edge.
Each edge stores two Booleans to serve this; one to mark whether it is a silhouette edge, and one to mark whether the edge has been hit for analysis yet. Each pair of half-edges is analyzed in order to determine whether the edge is a silhouette, but each half-edge is stored individually. In order to prevent more analysis than necessary, both edges in the pair are marked as hit once they are analyzed so that when the second edge pops up while scanning through the hash table, things don't get calculated an extra time.
In order to obtain extra speed, the mesh stores the last known camera position, which is what gets used for silhouette edge determination. By comparing that to the current position and only calculating edges anew when there is a change, the idle state is simply sleeping when the camera is static, as opposed to carrying out constant calculations which always have the same result.

5 Preprocess Edges

Along with the silhouette edges, adding lines to show the border s and creases within the mesh is also important, and give more detail to the final output. To know which one should be marked, all the edges go through preprocessing to check whether or not they are a border or crease edge.

With the Half Edge data structure, it is simple to tell which of the edges are border edges. It edge is first checked to see whether or not it has an opposite, and if it doesn't it is then marked as a border edge.

Figuring out which edges are crease edges is a bit more work. The best way to determine whether or not an edge is a crease edge is to find the angle between the two planes on either side of it. To do this, one first calculates the dot product of the two surface normals. The dot product is calculated by multiplying each corresponding variable and adding them all together, so every x value is multiplied together, and every y value, and then they are all added together. The resulting value is the magnitude of the two vectors, multiplied by the cosine of the angle between them, as shown below:

$$n1 \cdot n2 = ||n1|| \; ||n2|| \cos\theta$$

Since it is only the angle that is needed, the next step is to first calculate the magnitude each vector by squaring each variable and adding them together. Finally the angle can be found by taking the inverse cosign of the dot product divided by the magnitudes. Then to get it in degrees, multiply it by 180 over pi:

$$\theta = \cos^{-1}(n1 \cdot n2 \,/\, ||n1|| \; ||n2||) * 180/\pi$$

After that, the only thing left to be determined is the specific bounds for what qualifies as a crease edge. In this case, it starts at any angle less than 140 degrees, but can be shifted up and down to increase or decrease the number of edges and detail.

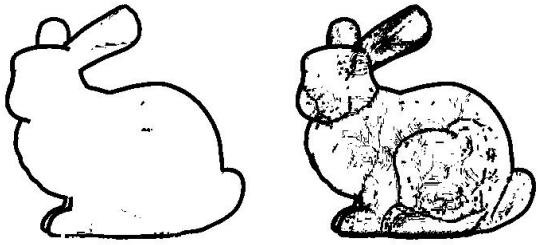

Figure 1: Right to Left a) Model with very little detail b) Model with a lot of detail

6 Shader

The shader function is more of a 1-D texture due to a lack of occlusion detection. To calculate the proper shade for each triangle, the face normal is compared via dot product to the (initial) light position, and the resulting cosine is scaled into a 0-1 space, then compared to the values read in from the given text file. All values from one threshold up until the next are counted as being of that color value, resulting in linear interpolation. As there are no actual rules for the light affecting the colors, the rules may be broken, and the shades do not have to move from lightest to darkest. Each face stores the index of its correct value for the shade array, allowing for quick lookup when drawing the mesh.

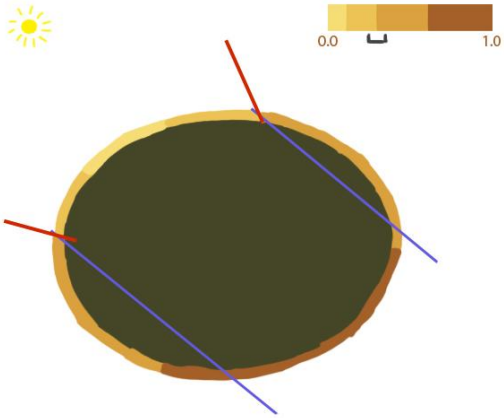

Figure 2: Displays problems with using the stencil buffer for each individual shade

Occlusion testing is not implemented due to the difficulty of doing so with arbitrary shader arrays. For two shades with the threshold between them at 0.5, implementation would be exactly identical to that of shadow polygons, drawing everything in the first color and then using the stencil buffer to redraw everything as the second. However, this becomes more complex as more levels are added, especially ones with thresholds between 0 and 0.5. When the threshold is less than 0.5, drawing the lines between the current and next level will not capture everything which should be shaded. Figure **2** captures this, showing the lines drawn in the direction from the light source at the border between shades two and three in blue. The outside of the ellipsoid mesh is colored with what the shades should be at those places. It can clearly be seen that merely redrawing what is inside of the blue lines would miss the outsides of the mesh which should be shade 3, and they would end up improperly colored. A possible solution to this would be to begin by drawing everything in the shade at 0.5, and doing the algorithm in each direction of the slider, heading outwards in the “lighter” direction first in order to make sure everything turns “darker” properly when blocked from the light source’s view. These are the red lines of the same figure. However, this runs the risk of accidentally causing parts of the mesh closer to the camera to become lighter than they should

be (or darker, if the “lighter” shades are instead

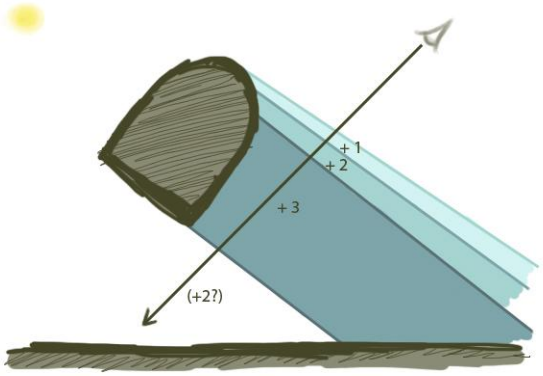

Figure 3: Displays incorrect extra shading caused by skipping levels

processed from 0.0 down to 0.5). A way to get around this would be to cheat, drawing all polygons shaded as stored up until 0.5, and using the stencil buffer for all values 0.5-1, but that may not be the desirable look depending on the shading gradient in the “lighter” region. There is also the problem of skipping levels. Depending on the complexity of the shader and topography of the model, there will not always be a clear and complete ring between each shade. One face may be at level 2 while its neighbor is at level 5. Attempting to create a ring between 2 and 3 would fail due to a gap at this edge. Simply drawing every division below the current shade will also fail, as illustrated in figure **3.**The stencil buffer may get confused when drawing all the extra polygons, incrementing three times for divisions between (1,2), (2,3) and (3,4), but only decrementing once due to a division between (1,4). The ground plane in the example shown would end up being shaded darker for the given viewpoint due to the stencil buffer results despite the fact it is supposed to be fully lit. The solution would be that after drawing the model at shade 1, the buffer would then need to be run for all shades 2-n. For each shade k, a complete ring would have to be constructed by marking all edges fulfilling the condition of being a border between a shade 1-(k-1) and a shade k-n. Regardless of the added complexity involved in solving extra problems with the stencil buffer, using the shader with the buffer for occlusion testing and shadows would result in a runtime of O(T * S), where T is the number of triangles in the model and S is the number of levels in the 1D shader. This is because the entire mesh needs to be re-rendered for each new level of the shader to be added in properly. Given that not all of the possible shaders follow light-to-dark rules and therefore would not make sense with shadows, and that looking up the shade for each triangle is O(1) if it is already being loaded to get drawn anyway, this has not been implemented.

7 Rotation

The rotation functionality was added in to help deal with lines that are pointed straight at the camera so that they are hardly visible. Ideally what should happen is that once found, these lines are rotated until they are perpendicular and can be seen perfectly fine by the camera.

To find the lines that are pointed straight at the camera, take the dot product of the vertex normal and the camera. The dot product of two vectors that are facing in opposite directions is 180 degrees, so once the dot product is found, those vertices that are closest to 180 are the ones pointing at the camera. These vertices are then marked.

The next step is to find a vector that is perpendicular to the normal by using the cross product. The first vector used is the vertex normal, but the second one can change depending on which direction the vector is going to be perpendicular to. To figure this out, first calculate which way the edge is going, by finding the difference between the position of the starting vertex and the ending vertex. If the biggest difference is in the x-direction, then it uses a vector like (0,1,0) or (0,-1,0) to have the results be perpendicular in the y-direction. Similar tests are also used for the y-direction and the z-direction.

Once the correct direction is found, it is time to calculate the cross product. To get the cross product, for each variable, you take the other variables and multiply them by the

opposite from each vector and take the difference:

$$\mathbf{v1 \times v2 = (a_2b_3 - a_3b_2)x + (a_3b_1 - a_1b_3)y + (a_1b_2 - a_2b_1)z}$$

After finding the cross product, the new vector is stored at the vertex. If there is already a vector stored, the average between the two is found, and then doubled to make up for the difference.

Currently, this code works well on simple meshes, like the cube. It doesn't work as well on more complex edges. For many of these edges, the angle between the two surface faces is too small, so that when the perpendicular normal is used, it disappears within the mesh. To try and make up for this, there is a check for these meshes, so when the angle is too small, it is left alone. Even with this however, many more checks and test would need to be run on each line to get the ideal output.

8 Triangulation

As the lines are being drawn as rectangles rather than using OpenGL's line function, it would have been aesthetically pleasing to get the lines to taper off at the edges, such around the bunny's leg, rather than existing as bricks. However, this has proved to be much more difficult than anticipated, and remains in a state not functioning as intended.

Testing whether there are any other edges around each vertex of a line (and tapering if there are none) may work for pre-processed edges, but will always fail for silhouette edges as silhouettes are always present in complete rings. Additionally, facing direction of the edges cannot be checked because silhouettes are always marked consistently in regards to which face is not visible. For the silhouette outlining the model, this does not matter so much, but for inner silhouettes such as the folds of the bunny's ear, tapering the lines would be desirable. Other methods which would not work include checking which edge is closer to the camera, as the full outline may have similar situations and triangulate in undesirable locations, and checking the difference in angle from the center of each edge to the camera, because a sufficiently complex mesh such as the 40k bunny will also fail this test for the full outline.

The mesh does currently have some triangulating behavior, although the exact cause given the current code remains undetermined. The full outline always triangulates in the same direction, even when the edge marked as being a silhouette is flipped to the other half-edge in the pair. This results in a very odd look, but it can be interesting, such as on the 1k bunny when pre-processed sensitivity is turned all the way up.

9 Success/Failure Cases

There are a small number of failure cases aside from various algorithmic drawbacks discussed above. Meshes such as spheres soft or rounded edges tend not to produce perfectly smooth outlines, despite that being the nature of the original shape. Because the silhouette edges are determined based on the direction from the center of the face to the camera position, bringing the camera in very close can cause mistakes as the silhouette pulls inwards from the actual visible extremes of the mesh. Back faces are handled improperly by both line and shader; lines are drawn outwards from the normals at each vertex, causing back faces to have no visible lines, and shade is determined by the face normal, so the back of a face is exactly the same shade as the front. Finally, as an object-space outline shader, popping becomes noticeable in the silhouette edges as the mesh is moved, especially with thicker lines. The algorithm is most successful for objects with clear creases due to how pre-processed lines are calculated, and for high-poly meshes such as the 40k bunny, where thinner lines and fairly high sensitivity can begin to look like a pen sketch.

10 Extras

There is a debug mode available in the program to check which edges are being identified as what. It does the calculations for the current camera angle, and then can be rotated and seen from multiple perspectives without the silhouette edges getting recalculated.
Silhouette edges are marked in red, while border and crease edges are noted in yellow.
Whiteout mode renders all triangles as unlit white, and is useful for checking how successful a particular outline is, in addition to just looking good in some renders.
There is also a line color randomizer function; when pressed, the line color the mesh is drawn with changes. It can be reset to the usual black.
Line width can be incremented and decremented, as can the sensitivity of what counts as a pre-processed edge.

11 References

[1] *Real-Time Nonphotorealistic Rendering*
Lee Markosian et al., 1997

[2] *Computer-Generated Pen-and-Ink Illustration*
Georges Winkenbach and David H. Salesi, 1994

[3] *Stylized Rendering Techniques for Scalable*
Real-Time 3D Animation
Adam Lake, et al.

[4] *X-Toon: An Extended Toon Shader*
Pascal Barla, Joeelle Thollot, Lee Markosian

[5] *Non-Photorealistic Rendering*
Anna Vilanova

[6] *A Non-Photorealistic Lighting Model For Automatic Technical Illustration*
Amy Gooch et. al.

[7] *Interactive Pen-and-Ink Illustration*
Michael P. Salisbury et al.

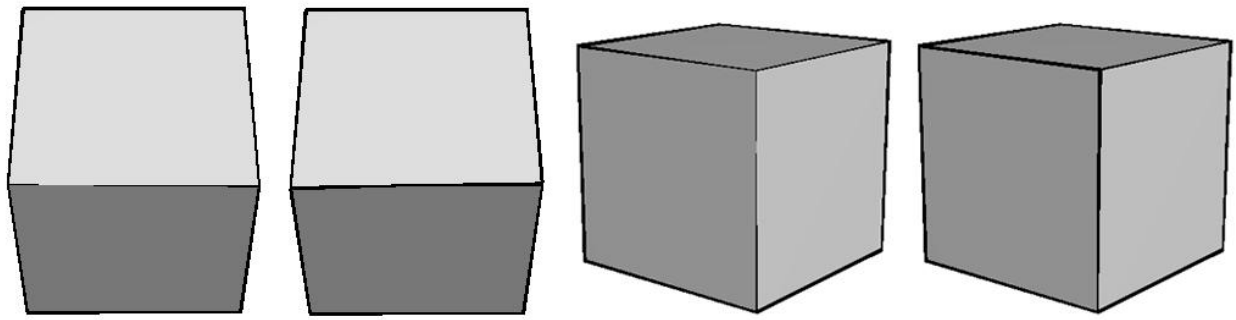

Figure 4: Two examples of Rotation working: The left side is without rotation and the right side is with rotation

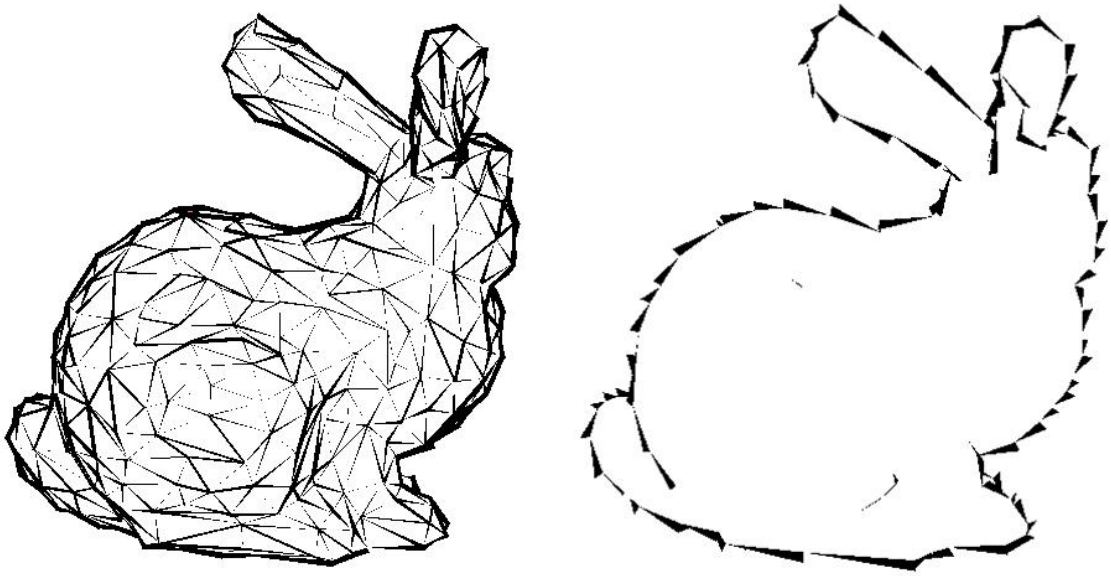

Figure 5: Triangulation: Left to Right a) Shows how interesting triangulation can look in certain cases despite being wrong b) Shows why triangulation is not working

Figure 6: Left to Right a) 8 level light to dark shader b) Breaking typical lighting conventions; light to dark is not required c) Shader with many close levels that makes a mesh look more complex than it actually is

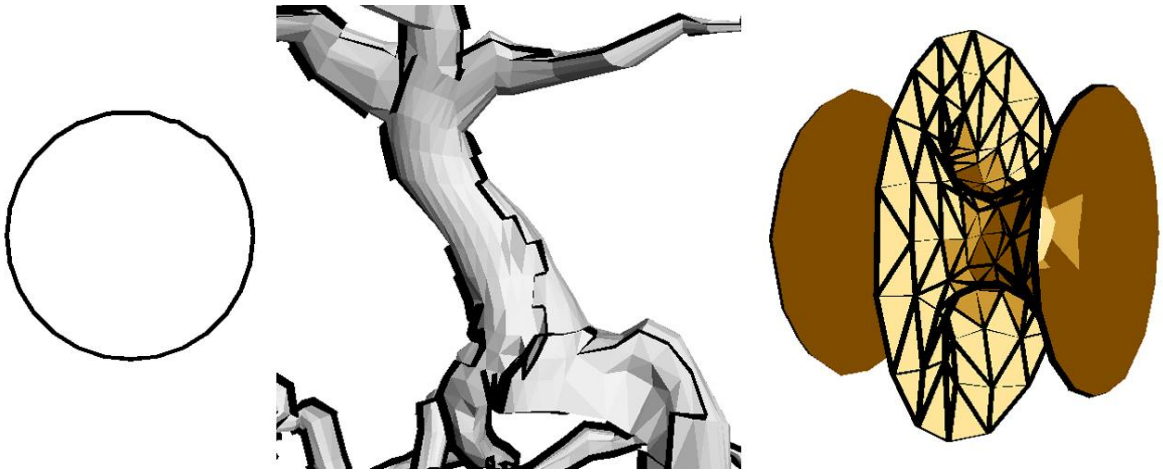

Figure 7: Failure Cases Left to Right: a) Supposedly smooth meshes do not always produce smooth edges b) Moving the camera too close causes errors as the silhouette moves in from the actual visible edge 3) Back Faces do not have outlines drawn and cause shading problems because face normals are what the shading is based on

Figure 8: Extras Left to Right a) High poly meshes with low line width kind of look like pen and ink sketches b) High line width and full preprocessing sensitivity can create an oil-dipped shiny look c) Debug Mode