Shining a Light on Material Appearance

Computing visibility functions using polygon intersection algorithms

Marc Ruijs

Supervisors: Ricardo Marroquim, Yang Chen

June 25, 2023

A Thesis Submitted to EEMCS Faculty Delft University of Technology, In Partial Fulfilment of the Requirements For the Bachelor of Computer Science and Engineering

Abstract

Creating photorealistic images is one of the ultimate goals of computer graphics. Previous work has shown that a material's microstructure plays a crucial role when trying to achieve photorealism. This is because a material's appearance depends on the roughness of its microstructure. Due to this dependence, effects such as masking and shadowing have to be taken into account, as these are capable of altering the effective reflectance of a material. Render engines typically use a mathematical expression, known as a visibility function, that aims to calculate the impact of these effects. However, even the best visibility function known is still an approximation; an exact solution doesn't exist. In order to evaluate the accuracy of visibility functions, an algorithm can be created that computes the correct output, such that the output of a given visibility function can be compared against it. Such an algorithm can be one of two types: approximative or exact. In this paper, we show that approximative algorithms are very capable and come close to their exact counterparts. However, there is still a non-negligible difference between them, meaning they aren't suitable for applications that demand very high levels of accuracy.

1. Introduction

"If it looks like computer graphics, it is not good computer graphics." This quote by Jeremy Birn is perhaps the best way to summarize the goal of computer graphics: to make it look so realistic that users feel like they are looking at the real world through a window, rather than a computer screen. One theory that aims to work towards this goal is called microfacet theory, which assumes surfaces to be made up of microscopically small triangles called microfacets. These microfacets each have their own normal and can point in a direction that is different from the overall surface normal. The distribution of these normals models the roughness of the surface, and therefore, influences the appearance of the material [1], as can be seen in Figure 1, which is the result of previous work [2].

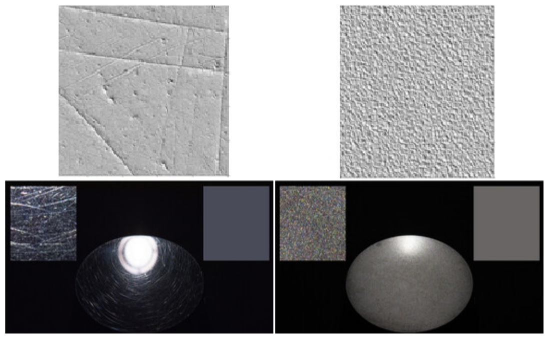

Figure 1: Aluminum disks and their surface scans. The right disk is polished only, the left disk has also been sandblasted. The difference in appearance comes solely from the difference in roughness. *Image from [2]. Licensed under CC BY-NC-ND 4.0*

The fact that the normal distribution of a surface affects the way it looks has other consequences. Imagine, for example, that we are looking at a surface such as in Figure 2. On the left, we see that our line of sight of the rightmost microfacet is obstructed; it is masked. Because of this, the normal that belongs to that rightmost microfacet no longer affects how the material appears to us. In other words, the masking effect has effectively changed the distribution of normals that are visible to us [1].

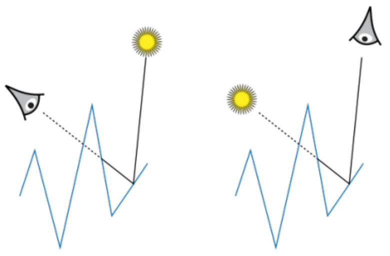

Figure 2: Left: masking prevents us from seeing the rightmost microfacet. Right: shadowing prevents the rightmost microfacet from being lit. *Image from [3]. Licensed under CC BY-NC-ND 4.0*

Similarly, the right side of Figure 2 shows a scenario where we are in fact able to see the rightmost microfacet, but in this case, our light source isn't, causing the facet to be in the shadow. This phenomenon is appropriately called shadowing, and it also influences the distribution of visible normals, and in turn, the appearance.

In order to take masking and shadowing into account, physically based render engines often make use of a mathematical expression, known as the *visibility function* [1], which aims to calculate the percentage of a surface area that is visible from a certain viewpoint, given that the light is coming from a certain direction. Multiple proposals of visibility functions have been made, most notably the Smith visibility function [4], which is still used in render engines today [1]. However, it isn't perfect. In deriving his function, Smith assumed there to be no correlation between the height of a microfacet, and the heights of its neighbors. That is, he assumed the microfacets to be structured randomly, causing the height of a microfacet to be uncorrelated to its normal [4]. However, this leads to the Smith function being less accurate for materials that do have a correlation between microfacet height and normal, as is illustrated in Figure 3. Here, the appearance of the material changes depending on the angle at which the light hits it, due to the structure of the material.

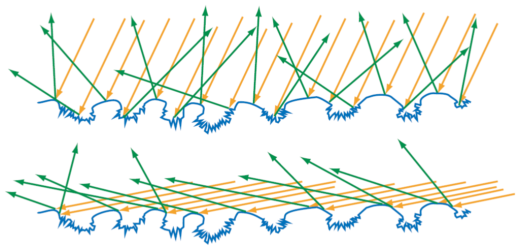
Figure 3: A structured material where higher parts are smooth, and lower parts are rough. In the top image, light is able to get into the rougher parts of the material, causing it to appear rough. In the bottom image, the shallow angle of the light causes it to reflect off of the smooth peaks of the surface, making it unable to reach the rough parts, which results in the material looking smooth.
Image from [1]. Licensed under CC BY-NC-ND 4.0

In order to evaluate the accuracy of a proposed visibility function, an algorithm can be written that is able to calculate the correct output, given a microsurface, view direction, and light direction. That output can then be compared against the output of the visibility function we wish to evaluate. By comparing the visibility function against the algorithm for a wide variety of scenarios, we can gain insight into the general accuracy of the visibility function. We will be able to see what scenarios it thrives in, and where it falls short.

Creating such an algorithm with the purpose of comparing and evaluating visibility functions can be done in multiple ways. The first option is to calculate the percentage of visible surface area in an exact way. While this sounds like an obvious choice, such an algorithm would be extremely slow, especially compared to the second option, which is an approximative algorithm. While these algorithms are inherently inaccurate, they are significantly faster. What is to be determined, is the magnitude of the inaccuracies. Should the approximated result be very close to the exact result, one could justify using an approximated algorithm to evaluate visibility functions. If there is a significant difference between the two algorithms, we cannot make this justification, and the exact method should be used whenever accuracy is of the essence. Answering this question of whether or not approximative algorithms are accurate enough is exactly the goal of this paper. We will present an exact algorithm and compare it against a previously implemented approximative algorithm.

2. Methodology

In order to calculate the percentage of visible surface area, the task has to be divided into smaller pieces. First, the assumption is made that shadowing and masking are independent. While this is not true in reality, it allows us to handle masking and shadowing separately. That is, we can start by only implementing masking.

Next, we are able to start by also ignoring the masking function, and solely focussing on filtering out back-facing triangles. This is done by taking the dot product between the view direction and the normal of the triangle, which can be calculated using the triangle's vertices.

Now that the triangles that face away from us are no longer being considered, we turn to implement the masking function. First, the microsurface is projected onto the camera's view plane, resulting in a 2D projection of all triangles. Then we have to determine the total area covered by these triangles. This process requires three steps: combining all projected triangles into one polygon, triangulating that polygon, and finally, calculating the total area using the result of the previous step.

2.1. Combining triangles

The process of combining all triangles into one polygon is an iterative one. We start by combining the first two triangles into a polygon. The third triangle is then combined with the result of the previous step. This process repeats until all triangles have been considered.

2.1.1. General case

When combining two triangles, or, more generally, two polygons, the algorithm has to be able to deal with multiple different types of intersections. An algorithm [5] is used to categorize the intersection between two edges, with four possible outcomes: vertex-vertex, vertex-edge, edge-edge, or no intersection. The most general case consists only of edge-edge intersections, of which an example is shown in Figure 4, where △ABC and △PQR intersect at points I_1 and I_2.

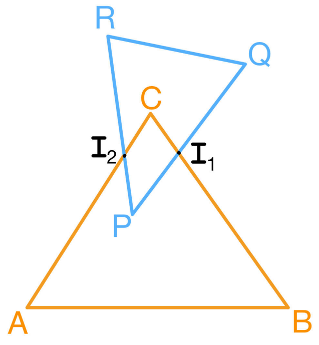

Figure 4: A common case of two triangles intersecting. △ABC intersects with △PQR at points I_1 and I_2.

The algorithm processes a case like this as follows. It starts by finding a vertex that is guaranteed to be part of the resulting polygon so that we can use that as a starting point. For example, the vertex C does not satisfy that criterion, since it lies inside △PQR, and is thus not going to be part of the polygon. In the case of this example, we can simply start at A. Now we analyze the edge that goes from A to the next vertex, which is B. Clearly, no edge intersects with AB, and so B must also be part of the polygon. Now, when we look at the edge from B to C, we do encounter an intersection, namely with the edge going from P to Q. Since this is an edge-edge intersection, we can simply follow the intersecting edge to its endpoint; that is, we add the intersection point I_1 to the polygon, and go to Q. Since there are no edges intersecting with the line from I_1 to Q, we add Q to the polygon as well. Note that we cannot always blindly follow an intersecting edge, as we will show in a later example. Analyzing the edge QR, we again see no intersections, so we add R. When trying to go from R to P, we encounter an intersection with CA. Since this is also an edge-edge intersection, we can add the intersection point to the polygon, and go to the endpoint of the intersecting edge, which is A. Now we are back at our starting point. We could technically go to B again, but there is no point in doing that since we have already visited that vertex. Therefore, the algorithm is done, and it returns the polygon that consists of the points A, B, I_1, Q, R, I_2.

2.1.2. Edge cases

Now we will look at a scenario where the steps taken in the previous section won't suffice. Looking at Figure 5, we see that we now have three intersection points: P lies on AB, Q lies on BC, and BC intersects with RP.

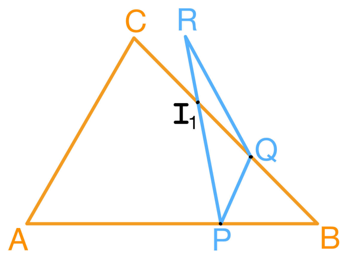

Figure 5: △ABC intersects with △PQR such that P and Q are intersection points.

Instead of giving a complete overview of how the algorithm processes this situation, we will highlight only the key aspects. Recall that, in the case of an edge-edge intersection, we can simply add the intersection point to the polygon, and follow the intersecting edge to its endpoint. In Figure 5, we encounter a type of intersection for which that rule does not generally work: the vertex-edge intersection. Consider point P, which lies on AB. Note how "follow the intersecting edge to its endpoint" does not apply here, since AB is not intersecting with an edge of △PQR, but rather, a vertex. Instead, we have to look at where the next edge of △PQR, starting at P, is going. In this case, the next vertex is Q, so we consider the edge PQ, which goes inside of △ABC, and we, therefore, should not follow it, but go to B instead. In other words, we completely ignore the intersection. Now consider point Q, which lies on BC. The next edge that starts at Q is QR, which goes outside of △ABC, so this time we should follow it, and go to R. We need a way to distinguish these two scenarios. It might be tempting to look at whether or not the endpoint of the edge is inside △ABC. R is outside of △ABC, while Q lies on BC, which we could count as being inside △ABC. However, this approach generally fails, since it could happen that an edge of △PQR first goes through △ABC, but has its endpoint outside of it. Imagine R being to the left of AC, for example. Then QR goes through △ABC, so we should not follow it, but since R is outside of △ABC, we would still end up doing just that.

We are on the right track though. Instead of using the endpoint of the edge, we want to stay at the starting point and take a very small step in the direction of the endpoint of the edge. Then we check if that point is inside △ABC or not. By doing it this way, we practically eliminate the issue of the previously proposed method, even though it is technically not completely solved; if △ABC is very small we could still end up testing a point that lies outside of it even though our edge goes through it first. However, if our step toward the endpoint is small enough, we will run into floating-point rounding errors before this becomes an issue.

Now our algorithm is able to correctly handle this type of edge case. There exist other types of edge cases, as shown in Figure 6, which we will briefly go over. (a) shows one more example of a vertex-edge intersection, but here, the endpoint of our current edge, AB, intersects with an edge of △PQR. We will add the intersection point, B, to our polygon, and go to the endpoint of the edge we intersect with. That is, we will go to P. The remaining edge cases are all vertex-vertex. The first, shown in (b), is a simple one. Points A and P overlap, so we have to choose between going to B and Q. Since Q is inside △ABC, we will simply go to B. Looking at (c), it again might be tempting to simply say that Q is outside of △ABC, so we should go to it. While we indeed should in this case, (d) shows why the reasoning fails, as we definitely don't want to go to Q there. As described earlier, we determine whether or not to go to Q by testing if the point that is very close to P and in the direction of Q is inside △ABC or not. Finally, we see two cases where AB and PQ are collinear. In (e), Q lies on AB, while in (f), it extends past AB. This difference is exactly how the algorithm determines its next action. If Q is on AB, it goes to B; if B is on PQ, it goes to Q.

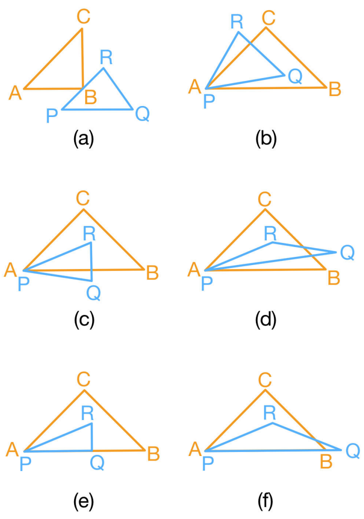

Figure 6: Remaining edge cases that our algorithm needs to be able to handle.

Implementing these edge cases, combined with being able to handle the general case, the algorithm is now able to combine any two polygons into one, which can be triangulated in the next step.

2.2. Triangulation

Triangulation algorithms are able to split a polygon into triangles. Our implementation is based on one described in previous works [6][7]. Before describing how the algorithm works, we will explain what it means for a vertex to be an *eartip*. The definition is as follows. Take any vertex of the polygon. Then, draw a line from the vertex that precedes the one you chose, to the vertex that succeeds it. If this edge is entirely inside the polygon, that means that those three vertices, the one you chose, the one that proceeds it, and the one that succeeds it, form an *ear*, with the vertex you chose being the *eartip*. If the edge from the preceding vertex to the succeeding vertex at some point intersects with any of the polygon edges, the vertices don't form an ear. We will look at examples while explaining the algorithm. The convenient property of eartips is that they can be removed from the polygon without increasing its complexity, which allows us to simplify it systematically by removing one vertex at a time until we are left with a single triangle.

Now we are ready to describe the algorithm, using Figure 7 as an example. Given a polygon with more than three vertices, it starts by searching for an ear. In order to check if A is an eartip, the line from E, which precedes A, to B, which succeeds A, is drawn in Figure 7(b). Since it intersects with polygon edge CD, A is not an eartip, so we check the next vertex. B is checked by drawing the line AC, which shows that B isn't an eartip either. The next vertex, C, is an eartip since BD is entirely inside the polygon. Now that we have found an ear, we first store the triangle that is formed by the ear, △BCD. Then we remove from the polygon the eartip and the edges that go to it and come from it. Obviously, we cannot have a gap in our polygon, so we connect the vertex that preceded the eartip with the vertex that succeeded the eartip, as shown in Figure 7(e).

Since we have just ripped a vertex out of our polygon, we want to check if the neighboring vertices have been affected. So, we will check both the succeeding and preceding vertices, which, in this example, are D and B, respectively. In the case that neither of these are eartips, the algorithm goes back to checking all vertices in the polygon, as it did in the first step. Figure 7(f) shows that D is not an eartip, since BE leaves the polygon. AD does stay entirely inside ABDE, so B is an eartip, and the same procedure as last time is repeated: the triangle that makes up the ear, △ABD is stored, and B is removed from the polygon. At this point, the algorithm sees that there are only three vertices remaining, so it stores these as the last triangle, and returns a list of all triangles it has stored. In this case, those triangles are △BCD, △ABD, and △ ADE. It is easily verified that these triangles combined exactly make up the original polygon.

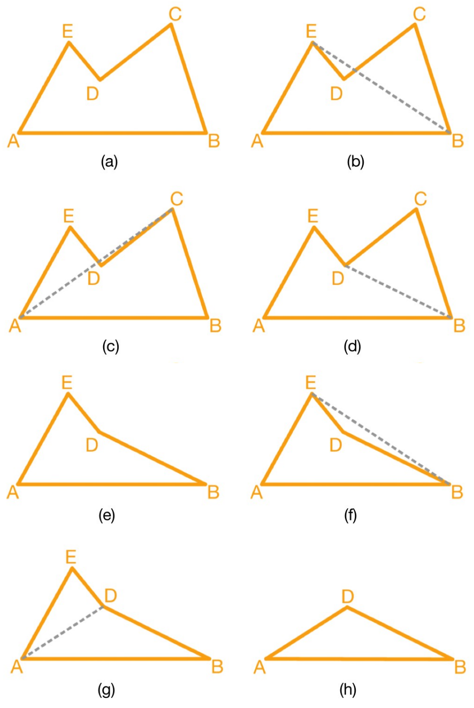

Figure 7: Triangulation steps for polygon ABCDE

2.3. Calculating the area

This final part of the algorithm is by far the simplest. We iterate over all triangles that we obtained using triangulation, and calculate their areas using the well-known formula for the area of a triangle: Area = height $\times$ base $\times \frac{1}{2}$. Adding up all these areas leads to the total area of our polygon, and therefore, the total area of the surface that is visible to us.

3. Responsible Research

In this research, ethical issues are not to be worried about. The only data that was used while conducting the research comes in the form of 3D scans of sandblasted aluminum disks, which were created in previous research [2]. Theoretically, the researchers of that study could have fabricated their results, but even if that is the case, it doesn't influence the integrity of this paper. This is because their data is not used as ground truth. Rather, it is solely used to verify the behaviour of our algorithm, and to determine how capable it is when calculating the area of a surface. Whether that surface is real or fabricated is not of the essence for our research. Regarding the results of this paper, all steps our algorithm takes in order to get to a result are described in the paper, and anyone who wishes to reproduce them is very much welcome to do so. The code is available at https://github.com/MJARuijs/ResearchProject. The algorithms can be found in the src/math/shapes/Intersection.h and src/math/shapes/Polygon.cpp files.

4. Results

In order to ascertain the degree of accuracy of approximative methods, our algorithm is compared against such a method which has been implemented earlier, and will be available at https://gitlab.tudelft.nl/ychen32/vgonio-mirror once finished at a later date. The methods are compared using the surface as seen in Figure 8. The surface is viewed with varying zenith angles, ranging from 90° to 10°, and an azimuth of 0°. One of these cases is shown on the bottom half of the figure, where the zenith is 45°. The results are shown in Table 1, and will be discussed in the next section.

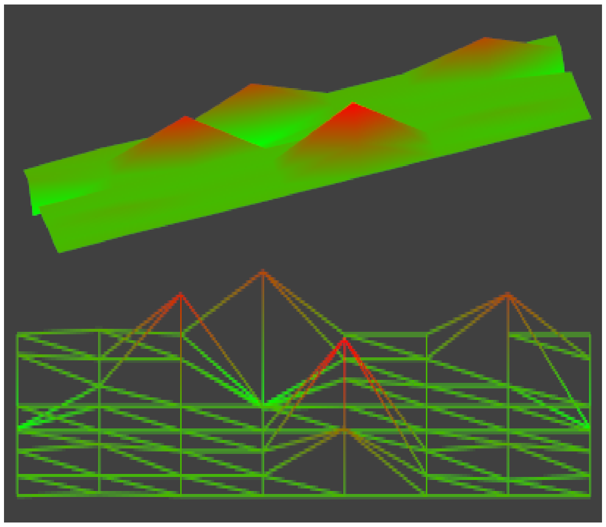

Figure 8: The top half shows the surface used for testing. The bottom half is one of the tested viewpoints, where the zenith angle is 45°

Zenith	Our algorithm	Approximated method
90°	100%	99.1%
80°	100%	99.2%
70°	95.1%	94.7%
60°	86.8%	86.9%
50°	79.2%	79.5%
40°	38.5%	71.8%
30°	62.9%	63.9%
20°	55.4%	55.6%
10°	40.2%	49.3%

Table 1: The percentages of visible surface area according to our algorithm and an approximation method, for varying zenith angles.

5. Discussion

In this work, we provided an algorithm that is able to exactly calculate the percentage of surface area that is visible from a certain viewing angle. We compared it against an algorithm that approximates the answer, in an effort to determine whether or not using an exact method is required for applications that require a high level of accuracy.

The results show that the approximative algorithm is very capable, and produces results that are very close to the exact method. It still isn't perfect, however, as is immediately noticeable when analyzing the results for a zenith angle of 90°. In this scenario, the surface is viewed from straight above, meaning that all surface triangles are completely visible, and the result should always be 100%. The fact that the approximative method gives a result of 99.1% is due to the inaccuracies that are inherent to approximative algorithms.

There are two outliers in the data, namely for zenith angles of 40° and 10°. The reason for these outliers will be addressed in the next section.

Based on the results, the research question will be answered as follows. Whether or not using an approximative method is justified depends on one's use case. If the highest possible level of accuracy is required, it is best to use an exact method, as the approximative method can be off by as much as 1 percent point. However, for applications where a delta of 1 percent point is acceptable, approximative methods are a potent alternative, with the added benefit of running substantially faster.

5.1. Limitations and Future Work

The biggest limiting factor of our algorithm is floating point rounding errors. These can cause our algorithm to make mistakes when combining polygons, leading to an incorrect polygon of which the area is calculated. It is therefore recommended to visualize and inspect the resulting polygon after running the algorithm, in order to verify that its output is correct.

Nonetheless, the majority of the results are consistent. The algorithm we provided can also be used to calculate the shadowing function. In future works, combining the output of the masking function and shadowing function could be investigated, which could lead to an algorithm that is able to calculate the output of the entire visibility function in an exact way. At that point, we will be able to evaluate and compare any proposed visibility functions.

References

[1] T. Akenine-Mller, E. Haines, and N. Hoffman, *Real-Time Rendering, Fourth Edition*, 4th, USA: A. K. Peters, Ltd., 2018, pp. 327–338.

[2] O. Clausen, Y. Chen, A. Fuhrmann, and R. Marroquim, "Investigation and simulation of diffraction on rough surfaces," *Comput. Graph. Forum*, vol. 42, no. 1, pp. 245–260, 2023, doi: https://doi.org/10.1111/cgf.14717. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14717

[3] M. Pharr, W. Jakob, and G. Humphreys, "Physically based rendering: from theory to implementation," 3rd, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016. [Online]. Available: https://pbr-book.org/3ed-2018/Reflection_Models/Microfacet_Models

[4] E. Heitz, "Understanding the masking-shadowing function in microfacet-based brdfs," *J. Comput. Graph. Techn. (Jcgt)*, vol. 3, no. 2, pp. 68–70, Jun. 30, 2014. [Online]. Available: http://jcgt.org/published/0003/02/03/

[5] C. L. Sabharwal, and J. L. Leopold, "A generic design for implementing intersection between triangles in computer vision and spatial reasoning," in *Innovative Res. Attention Model. Comput. Vision Appl.*, R. Pal, Ed., Hershey, PA, USA: IGI Global, pp. 195–235. [Online]. Available: https://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/978-1-4666-8723-3.ch008

[6] P. Schneider, and D. Eberly, *Geometric Tools for Computer Graphics*, Elsevier Science, 2003, pp. 772–775. [Online]. Available: https://books.google.nl/books?id=82kntxqd1BoC

[7] T. Akenine-Mller, E. Haines, and N. Hoffman, *Real-Time Rendering, Fourth Edition*, 4th, USA: A. K. Peters, Ltd., 2018, p. 685.