Portland State University
PDXScholar

University Honors Theses University Honors College

Spring 6-10-2023

How Photorealistic Images Are Generated

Nahom Ketema
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/honorstheses

Part of the Graphics and Human Computer Interfaces Commons
Let us know how access to this document benefits you.

Recommended Citation
Ketema, Nahom, "How Photorealistic Images Are Generated" (2023). *University Honors Theses.* Paper 1323.
https://doi.org/10.15760/honors.1352

This Thesis is brought to you for free and open access. It has been accepted for inclusion in University Honors Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

How photorealistic images are generated

by

Nahom Ketema

An undergraduate honors thesis submitted in partial fulfillment of the

requirements for the degree of

Bachelor of Science

in

University Honors

and

Computer Science

Thesis Advisor

David Ely

Portland State University

2023

Acknowledgments

A massive thank you to my advisor David Ely for dedicating a lot of time and energy to teaching me the ins and outs of computer graphics. None of this would have been possible without his help and I'm grateful for it.

Introduction

Have you ever wondered how we can see 3D objects on a 2D screen? How objects on a screen get colored? How lighting gets simulated on a computer? All of these questions and more can be answered through computer graphics.

Computer graphics is a field within computer science that deals with using computers to generate images on a screen. It has a wide range of applications ranging from video games to animations. There are different ways of drawing and visualizing graphical objects, and we will be discussing some of them in this paper.

The goal of this project is to look into different ways of drawing objects on a screen by going over the mathematics behind it, and by looking at the images it creates. Doing so will allow us to visually compare/contrast these methods.

To see how photorealistic images are generated, we will first be looking at how 3D objects are represented, and the process by which we can plot and draw the 3D object to a 2D screen. We will also be looking at how to rotate these objects to be able to view the 3D object being drawn from different angles.

We will then begin looking at how we can add lighting to the objects. Lighting is an important part of generating photorealistic images, as we will see over the course of this paper. We will be looking at a few ways with which we can add lighting to our objects. We shall also discuss the benefits and drawbacks of each method.

Lastly, we will be looking at the lighting technique that gives us photorealistic images. We will briefly discuss how it works and also look at the advantages and disadvantages of it.

The graphics library used for this thesis project is called Gkit. Gkit is a lightweight library that allows users to draw lines, shapes, and polygons. It also allows users to add coloring to these different lines or shapes using an RGB value. The screen layout of Gkit looks as follows: the bottom left of the screen has x-y coordinates (0,0) and the top right corner has coordinates (width-1, height-1). This will be important for the coming chapters.

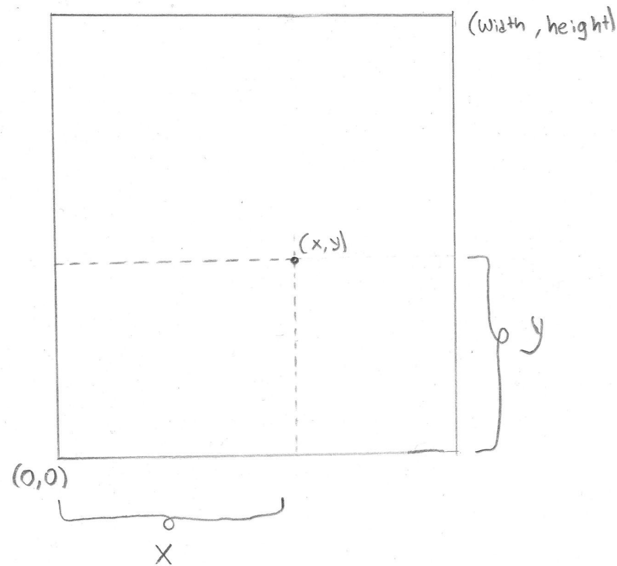

Figure 1: Gkit screen layout

Gkit uses an X11 display server to display onto a screen. In our case, we will be using xming as our display server. There are a few other alternatives users can use to get a similar result.

3D object representation

The 3D objects used for this project are stored in XYZ files. These files have all the necessary information that is needed to draw a 3D object. Below is an example of an XYZ file for a box.

```
8
     0.50000      1.00000      1.50000
    -0.50000      1.00000      1.50000
    -0.50000      1.00000     -1.50000
     0.50000      1.00000     -1.50000
     0.50000     -1.00000      1.50000
    -0.50000     -1.00000      1.50000
    -0.50000     -1.00000     -1.50000
     0.50000     -1.00000     -1.50000
6
4   0 1 2 3
4   4 7 6 5
4   0 3 7 4
4   1 5 6 2
4   2 6 7 3
4   0 4 5 1
```

Figure 2: XYZ file for a box

In order to be able to use these XYZ files, the first step is to properly understand how they work. If we look at an XYZ file, the first line has a number and that number represents the number of points used in that 3D image. In the example above, the first line is an 8 because there are 8 vertices in a box. Once we read the number of points, the next lines represent the coordinates of these points. In our example, since we have 8 points, the next 8 lines represent the XYZ coordinates for points 0 - 7. In most programming languages, indices start from a zero and that is why we have points 0 to 7 and not 1 to 8.

Once we have the points for an object, we need to be able to draw polygons to create a 3D object. This is what the remaining portion of the XYZ file represents. Notice how there is another single number right after the coordinates of the last point. This number represents the number of polygons for our object. Going back to the example above, we see that there are 6 polygons for the box. Notice how a box has 6 sides (four sides plus the top of the box and the bottom of the box). After the line that represents the number of polygons, the remaining lines show which points are connected to one another to create a polygon.

As an example, if we look at the first polygon, we see the following: “4   0 1 2 3”. Here, the 4 represents the number of points that make that particular polygon. The “0 1 2 3” represents the individual points being connected to create the polygon. In this case, point 0 is connected to point 1, point 1 is connected to point 2, point 2 is connected to point 3, and finally point 3 is connected to point 0. Using this general layout, we can draw 3D objects.

3D to 2D projection

So, we have a 3D object with x, y, and z coordinates that we want to draw onto a screen. However, a computer screen is two-dimensional and therefore we can’t just place the coordinates of 3D objects into a screen without converting them into 2D. This is what we will be looking at for this section. How do we project a 3D image onto a 2D screen?

Here is something you can try. Hold some object close to your eye. Then slowly move it further away from your eye. Notice how the object seems to shrink smaller when moving it away. This is the basic concept we need to be able to convert a 3D point to 2D.

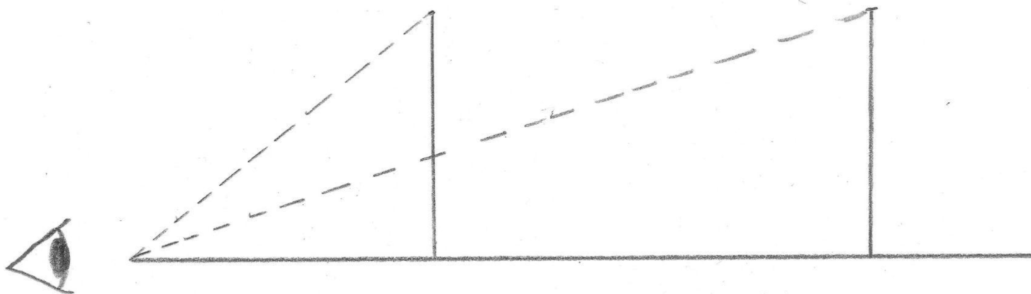

Figure 3: Two objects of the same height but different distances

In our 3D object, the x-axis represents the height and the y-axis represents the width. The z-axis represents how far the object is from us. In the image above, the object closest to the

eye would have a smaller z value than the object that is farther away. We can utilize this property to convert a 3D object to 2D.

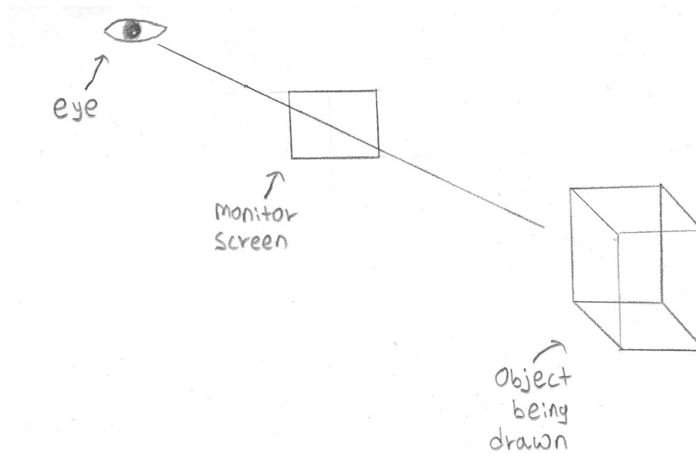

Figure 4: How 3D objects are visualized

For the image above, notice the following few things. We have a 3D object we're trying to project to 2D, we have a monitor screen which is where we will be displaying the 2D object, and we have our eyes with which we can see the object. It will be more helpful to see this same image from a different perspective, as shown below.

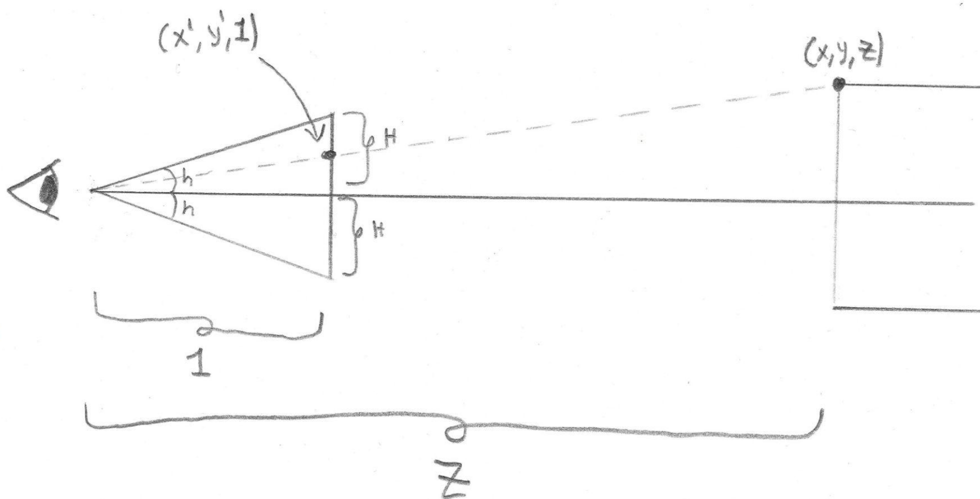

Figure 5: Side-view of Figure 4

Looking at the image above, notice how there are all sorts of geometry involved in converting a 3D object into a 2D one. The first thing to notice is the small h as well as the capital H. The small h refers to the half angle. Since our eyes can't see everything around us, we would need to limit the half angle to be of a certain value. Also, notice how the distance between the

eye and the screen is set to be 1 unit long. While this can be changed, having it be 1 unit makes it easier for computation. We will also be setting the half angle “h” to be 45°, which means that we can see up to 45° up, down, left, and right from the center. Using some trigonometry formulas, we can then compute the value of capital H. The calculation for it is below.

$$\tan\theta = \frac{opposite}{adjacent}$$

$$\tan h = \frac{H}{1}$$

$$H = \tan\frac{\pi}{4}$$

Going back to the image above, notice how there is a dotted line that goes from the eye all the way to the XYZ point. Notice how this line crosses through the monitor screen. We represent the coordinates for this point as (X’, Y’, 1). The reason the last value is a 1 is because it corresponds to the z-axis which, as mentioned before, is the distance between the eye and the monitor screen. This holds true for any point being drawn on the monitor screen, and this is the key with which we can project a 3D object into 2D.

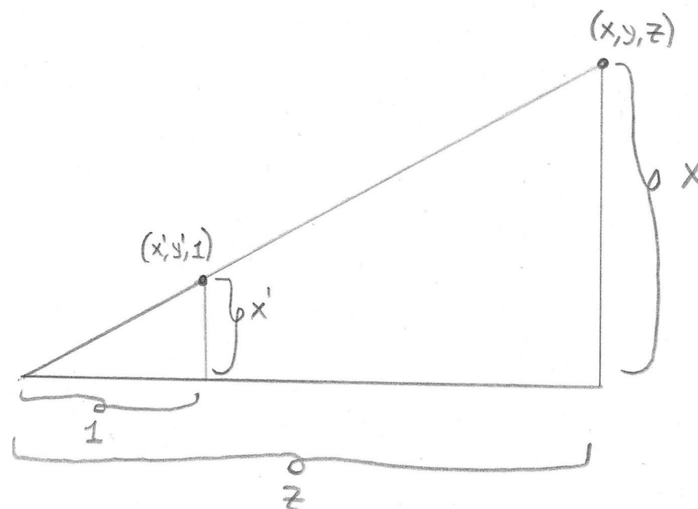

Figure 6: Image of the trigonometry behind projecting 3D objects into 2D

Using the properties of right triangles, We can compute the value of X’ as follows:

$$\frac{X}{X'} = \frac{Z}{1}$$

$$X' = \frac{X}{Z}$$

The same property also holds for Y’ as shown below:

$$\frac{Y}{Y'} = \frac{Z}{1}$$

$$Y' = \frac{Y}{Z}$$

With having to convert the 3D image into 2D out of the way, we have one more thing to do before we can plot the point over to the Gkit screen. Notice how in the screen we were using above, setting X'=0 and Y'=0 would mean the object is located at the center of the screen. However, this is not the case with Gkit. If we were to plot (0,0) into Gkit, the point would be located at the bottom left corner of the screen, which is obviously not what we want.

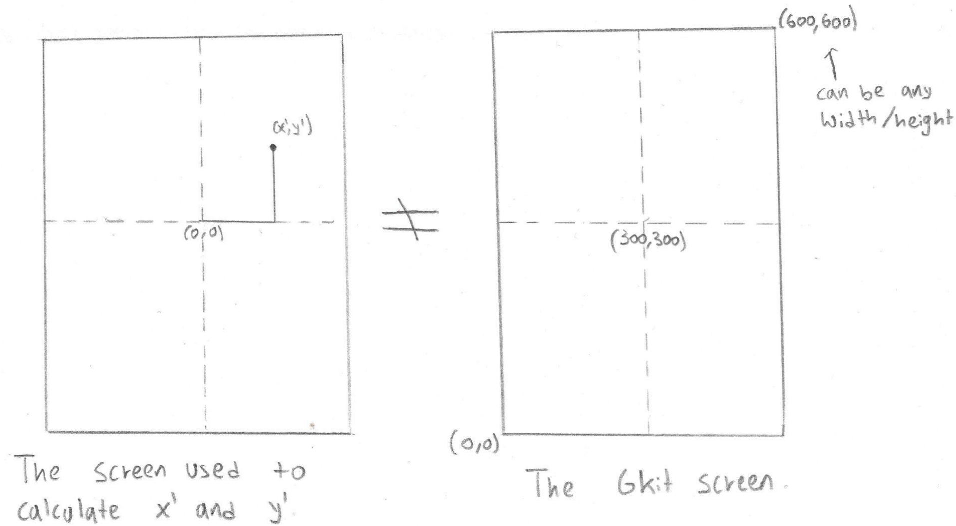

Figure 7: A brief comparison between the screen used to calculate (X', Y') and the Gkit screen.

We would therefore need to do another conversion, this time from X' to X'' and from Y' to Y''. Surprisingly enough, there are a lot of similarities between having to convert X to X' and converting X' to X''. Look at the image below to see their similarities.

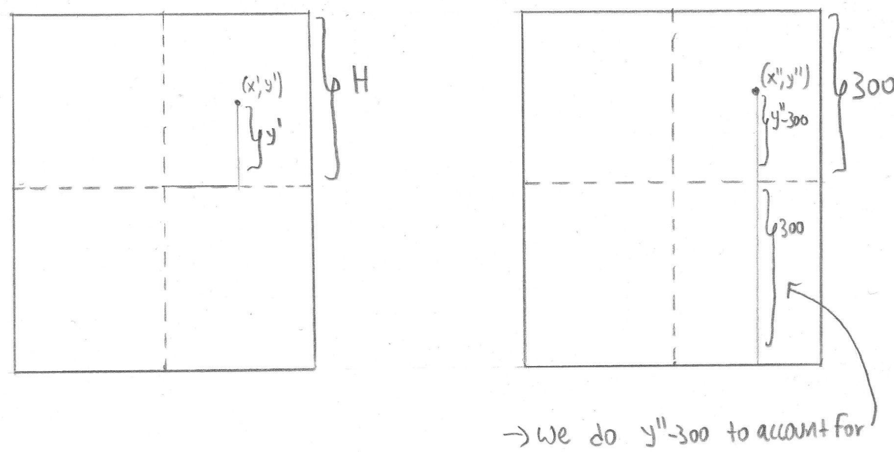

Figure 8: Comparing X' to X'' and Y' to Y''

Observe that what Y' is to H, (Y''-300) is to 300. This is going to be the basis of our calculations as we can see below.

$$\frac{Y'}{H} = \frac{Y'' - 300}{300}$$

$$Y'' = \frac{300 \cdot Y'}{H} + 300$$

The same holds true for calculating X''

$$\frac{X'}{H} = \frac{X'' - 300}{300}$$

$$X'' = \frac{300 \cdot X'}{H} + 300$$

With that being done, we can simply plot X'' and Y'' to be able to draw a 3D object into a monitor screen. The image below is an example of what we can draw using the math we went over.

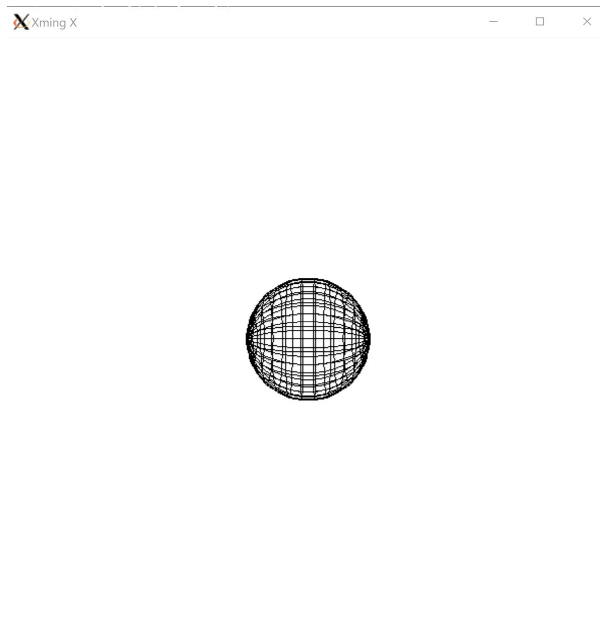

Figure 9: An image of a sphere being drawn into a monitor screen

As great as it is to be able to draw 3D objects, it would be even greater to be able to rotate the object around to view it from different angles. The good thing is, we can do that with just a little bit more math.

In order to rotate the 3D object, we would need to rotate all of the points that create the image. The first thing we need to determine is the point from which we can rotate the object. There is no right answer for this as it all depends on how you want to rotate it. For this project in particular, the point from which we will be rotating is located at the center of the image. We can determine this by figuring out the most extreme points(furthest away point vs. closest point) and averaging them out. This would give us a point that we can use to rotate. The second thing we need to figure out is the angle of rotation. While there is no right or wrong answer in this case

neither, the angle would preferably be something small. Once we have these two things, it is time to rotate the image.

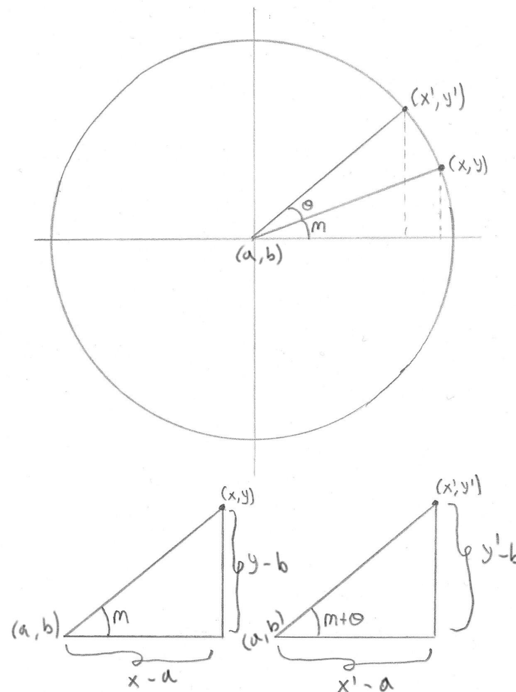

Figure 10: Rotating values (X, Y) to (X’, Y’)

The angle of rotation for the image is represented by the value θ. The value m represents the current angle of rotation. We wouldn’t need to know the value of m to be able to rotate the point, and we will see why in the calculations below. Like most of the previous calculations, (X, Y) represents the current coordinates of the point we will be moving, and (X’, Y’) represents where the point would end up. (A, B) refers to the coordinates from which we rotate the image

Looking at the image above, notice how we can isolate (X, Y) and (X’, Y’) as separate right triangles with the same hypotenuse length. Once we separate them, we can use a little bit of trigonometry to determine the values of X’ and Y’.

$$\cos(M) = \frac{X - A}{hypotenuse}$$

$$\sin(M) = \frac{Y - B}{hypotenuse}$$

$$\cos(M + \theta) = \frac{X' - A}{hypotenuse}$$

$$\sin(M + \theta) = \frac{Y' - B}{hypotenuse}$$

Now that we have the cosine and sine values for the two points, we can use the sum and difference identities for cosine and sine to find the values we need. The sum formulas for cosine and sine are shown below.

$$\cos(\alpha+\beta) = \cos\alpha\cdot\cos\beta - \sin\alpha\cdot\sin\beta$$
$$\sin(\alpha+\beta) = \sin\alpha\cdot\cos\beta + \cos\alpha\cdot\sin\beta$$

We can apply this formula to the angle created by (X’, Y’).

$$\cos(M+\theta) = \cos M\cdot\cos\theta - \sin M\cdot\sin\theta$$
$$\sin(M+\theta) = \sin M\cdot\cos\theta + \cos M\cdot\sin\theta$$

$$\cos M\cdot\cos\theta - \sin M\cdot\sin\theta = \frac{X'-A}{hypotenuse}$$
$$\sin M\cdot\cos\theta + \cos M\cdot\sin\theta = \frac{Y'-B}{hypotenuse}$$

Note that we already know what the cosine and sine values for M are. Therefore, we can replace those values.

$$\frac{X-A}{hypotenuse}\cdot\cos\theta - \frac{Y-B}{hypotenuse}\cdot\sin\theta = \frac{X'-A}{hypotenuse}$$
$$\frac{Y-B}{hypotenuse}\cdot\cos\theta + \frac{X-A}{hypotenuse}\cdot\sin\theta = \frac{Y'-B}{hypotenuse}$$

Finally, we can cancel out the hypotenuse from both sides and we get the equation to calculate for X’ and Y’.

$$X'-A = (X-A)\cdot\cos\theta - (Y-B)\cdot\sin\theta$$
$$X' = (X-A)\cdot\cos\theta - (Y-B)\cdot\sin\theta + A$$

$$Y'-B = (Y-B)\cdot\cos\theta + (X-A)\cdot\sin\theta$$
$$Y' = (Y-B)\cdot\cos\theta + (X-A)\cdot\sin\theta + B$$

Notice how there are only two variables below but we are using a 3D object. The reason behind this is that we are rotating the object across an axis. We can rotate objects along three axes. The X axis, the Y axis, and the Z axis.

Using the formula above allows us to view images from different angles. The image below is an example of how we can rotate the sphere from Figure 9 to see it from another angle.

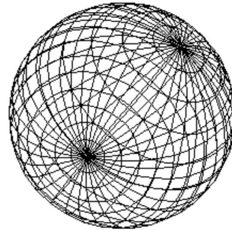

Figure 11: Image of the sphere from Figure 9 viewed from another angle

Looking at the sphere we looked at, notice how we can see all of the polygons that make up that object. While this is great to be able to see all of the points and polygons, in reality, we wouldn't be able to see the rear end of the sphere. Therefore, our next step would be to ensure that only the polygons directly visible to our eyes can be seen in the image. There are a few ways of doing this, but we will be focusing on two different methods, and we will be looking at the advantages and disadvantages of each method.

Back-face elimination

In order to properly understand how this method works, we would need to quickly look at some mathematical concepts.

If we look at the sphere in Figure 11, we can see that the sphere is made up of many different polygons. These polygons are flat surfaces that are small in size and it is the combination of these polygons that gives us the image. Having more polygons allows us to create more realistic objects but with the increase in the number of polygons comes an increase in the amount of computations we need. Therefore, there is a tradeoff between performance and realism. This is why many old video games looked more “blocky” compared to newer ones. The devices at that time were nowhere near as powerful as the devices today and therefore couldn’t afford to add too many polygons without sacrificing performance.

For this method, we will be focusing on a single polygon within the object. This polygon would have what is called a normal vector. In order to explain what a normal vector is, we need to quickly explain what a vector is. A vector is a quantity that gives us a magnitude and direction. We can build vectors using two points, as we will see later on. The normal vector of an object is a vector that is perpendicular to the polygon. Since the polygons that make up the

sphere are oriented in different directions, their normal vectors would also be different from one another.

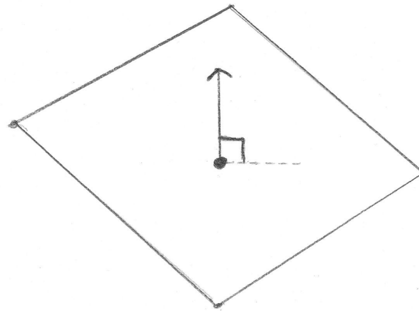

Figure 12: Image of the normal vector of a surface

The core concept behind this method is that, to be able to see a polygon, the angle between the vector created by our eye and the normal vector has to be less than 90°. We therefore can look at the angle and only proceed to draw the polygon if the angle is less than 90°. In doing so, we get an image as follows.

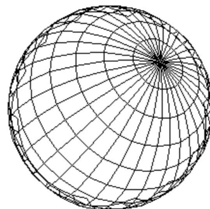

Figure 13: Image of the sphere using back-face elimination

As great as this method is, it comes with a major drawback. Consider a scenario where there is another sphere right behind the sphere shown. If we were using this method, the front half of the second sphere would also be shown. However, that is not exactly what we want to happen when drawing the images, and that leads us to our next method which deals with this problem.

Painter’s algorithm

The Painter’s algorithm gives us another perspective with which we can address this issue. Instead of having to calculate the normal vector and find the angle between it and our eye, the painter’s algorithm sorts the polygons from the furthest polygon from our eye to the closest one and then colors the polygons accordingly. Using this method, the polygons furthest away from our eye get colored first, followed by the polygons closer to our eye. Therefore, if the polygons on the rear of the object were not meant to be viewed, they get covered off by the polygons in the front. While this method is simple to explain, it does come with a drawback, as we can see in the image below.

Figure 14: Image of the sphere using Painter’s algorithm

The image above uses Painter’s algorithm to draw the sphere. We use the color gray to draw the object. Since all of the polygons are colored using the same color, the final image looks more like a circle than it does a sphere. This is an issue because we can’t really see the depth of the objects. We can fix this by utilizing what is called light models.

Light Models

One of the reasons we can view depth in real life is due to the way surfaces are lighted. Objects close to a light source tend to be brighter than those further away. We can use this to be able to color polygons accordingly and create objects that look more realistic.

One of the ways we can light objects is by using light models. For light models, there are three basic forms of lighting that can be used to color polygons. They are called Ambient lighting, Diffuse lighting, and Specular lighting.

Ambient lighting is lighting that is uniform to all polygons. If we were only using ambient lighting to color our object, we would get images similar to the sphere shown for Painter's algorithm. We, therefore, need to add other forms of lighting. Diffuse lighting considers the direction of the light source to color the objects. Therefore, polygons facing the light source tend to be brighter than those that aren't. Specular lighting on the other hand refers to the very bright spot made by the light source bouncing off the object into our eyes. One big difference between Diffuse lighting and Specular lighting is that Specular lighting is dependent on where we are looking at the object from. Specular lighting moves as we move our eyes. Using these three factors, we can determine the intensity of the light and we can paint the polygons using this intensity.

Before we get to the math behind how we light objects, we need to discuss the way we color objects. Every color can be described as a combination of the colors red, green, and blue(RGB). Therefore, to color an object we can pass values for red, green, and blue. For our program, these values range from 0 to 1, where 0 represents the lack of color and 1 represents the maximum amount of that color. An RGB value of (0,0,0) refers to the color black and the RGB value of (1,1,1) refers to the color white. As an example, the color grey has an equal amount of red, green, and blue. We can increase or decrease the amount of these variables to either make the gray color darker or lighter. This is important because we will initially be using grayscale to color our objects. The way we do so is by calculating the intensity of the light, which is a value from 0 to 1. We can then color the polygon using this intensity by passing an RGB value of (Intensity, Intensity, Intensity), which would give us the lighting we need in grayscale. Once we determine how to calculate the intensity, we can change the color of our image by adding a minor modification.

The first thing however is to calculate the intensity. Since the intensity is the sum of ambient, diffuse, and specular, we need to figure out what the maximum value is for ambient, diffuse, and specular. We don't want the sum of these to be over 1 and therefore we need to limit the maximum value they can be. In our case, we will be setting the maximum ambient value to 0.2, and the maximum diffused value to 0.4, giving us a maximum specular value of 0.4. Therefore, added up they can't be greater than 1.

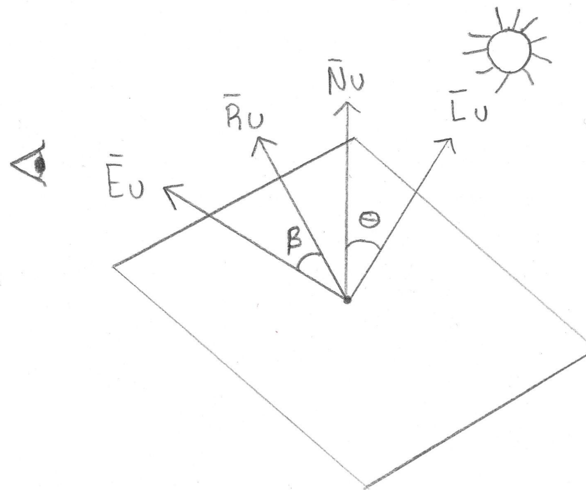

Figure 15: Image of all the vectors involved in creating light models

Since the ambient value gets applied to every polygon similarly, there isn't really any computation that needs to be done. All we would need to do is add the maximum ambient value for all surfaces. As for calculating the diffuse and specular lighting, there are a few vectors that we would need in order to find their respective values. The first vector that we need is the normal vector(represented as Nu). This is a vector that creates a right angle with the polygon. The other vector that we need is the light vector(represented as Lu). This vector points towards the source of light and will be useful to compute the diffuse lighting. The third vector that we need is the reflection vector(represented as Ru). This vector is essentially the reflection of the light vector and will be used to calculate the specular lighting value. Finally, we also have the eye vector(represented as Eu). This vector points towards the eye and will be important to calculate the specular lighting. Notice how all of the representations of these vectors have a "u" at the end. The reason behind this is that these vectors will need to be converted into a unit vector for our calculations. A unit vector is a vector that has a magnitude of 1. We will discuss how to convert vectors to unit vectors and the reason we do so later on.

Figure 16: Image of a vector that starts at point (X, Y, Z) and ends at point (X', Y', Z')

Now that we know the vectors that we need, we need to find out how we can compute the vectors. Vectors have direction and magnitude. The direction shows where a vector is pointing to and the magnitude shows how large or small a vector is. All we need to find the values of a vector are two points, the starting point and the ending point. Once we have the coordinates for those points we can compute the value for vector V as shown below.

$$start = (X, Y, Z)$$

$$end = (X', Y', Z')$$

$$\vec{V} =< (X' - X), (Y' - Y), (Z' - Z) >$$

As was mentioned earlier, vectors have varying magnitudes. In order to convert a vector to a unit vector, we would need to know the magnitude of the vector. For a vector v that has components Vx, Vy, and Vz, we can determine the magnitude of the vector as follows.

$$\vec{V} =< Vx, Vy, Vz >$$

$$|\vec{v}| = \sqrt{Vx^2 + Vy^2 + Vz^2}$$

Now all that is left is to divide the X, Y, and Z components of the vector by the magnitude as shown below.

$$\vec{Vu} =< \frac{Vx}{|\vec{v}|}, \frac{Vy}{|\vec{v}|}, \frac{Vz}{|\vec{v}|} >$$

With that, we are now ready to start creating these vectors. The first and arguably the most difficult vector to calculate is the normal vector. In order to do so, we would need at least two vectors. This shouldn't be too difficult since every polygon contains a minimum of three points, from which we can create two vectors. Once we get these vectors, we can compute what is called a “cross product”. The cross product of two vectors gives a vector that is perpendicular to both vectors, which is what we’re looking for in a normal vector. The cross product of two vectors A and B is represented by A×B. We can compute it as follows.

$$\vec{A} = < Ax, Ay, Az >$$

$$\vec{B} = < Bx, By, Bz >$$

$$\vec{A} \times \vec{B} = < (AyBz - AzBy), (AzBx - AxBz), (AxBy - AyBx) >$$

If we have three points, namely P1, P2, and P3, we can create vector A starting at P1 and ending at P2, and we can create vector B which starts at P2 and ends at P3. Using these vectors, we can use the formula above to determine the normal vector.

We can now start calculating the light vector and eye vector. To do so we would need to find the coordinates of the light source, as well as the coordinates of the eye. For our drawings, we have been using the coordinates (0, 0, 0) for our eye, which we will keep on using. For the light coordinates, we can set it to be any reasonable number. We can then pick any point within the polygon from which we can create these two vectors. While we can use any point within the polygon, it would be best to be consistent with the point we choose.

Lastly, we need to determine the reflection vector. The first step is to determine the height h of the light vector. If we look at the image below, we can see that the light vector makes an angle with the normal vector, which we represent as θ. We can use some trigonometry to determine the value of h as follows. Note that since Lu is a unit vector, its magnitude is 1.

$$cos\theta = \frac{h}{Lu}$$

$$cos\theta = \frac{h}{1}$$

$$h = cos\theta$$

We will be looking at how we can find the value of θ in the next section. For now, notice how the height of the reflection vector is equal to the height of the light vector. We can use that to our advantage and place the light vector on top of the reflection vector, as we can see in the image below. These two vectors added together have a magnitude double the size of h and a direction similar to Nu. We can use that to determine the value of Ru as follows.

$$Ru + Lu = (2h)Nu$$

$$Ru = (2h)Nu - Lu$$

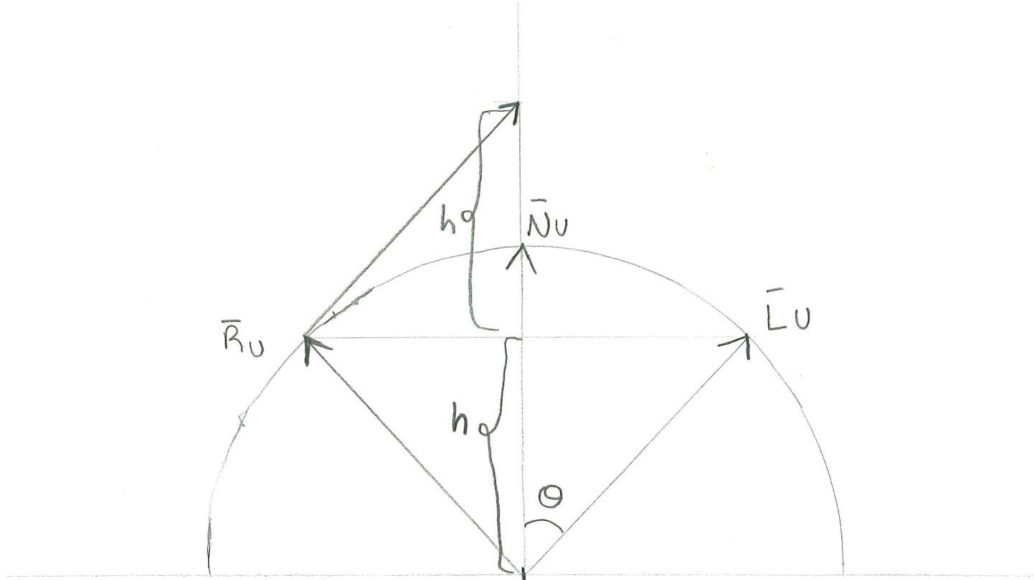

Figure 17: Image showing the correlation between Lu, Nu, and Ru

Now that we have all of the vectors we need, it is time to start calculating the angles we need to be able to color the polygons. The way we do so is by calculating the “dot product”. The dot product of two vectors A and B is represented by A▪B. It is worth noting that, unlike cross-products, dot-products give us a numeric value and not a vector. We can compute the dot product as follows.

$\vec{A} = < Ax, Ay, Az >$

$\vec{B} = < Bx, By, Bz >$

$\vec{A} \cdot \vec{B} = AxBx + AyBy + AzBz$

Here is how we can use the dot product of two vectors to determine the angle between them.

$\vec{A} \cdot \vec{B} = \left|\vec{A}\right| \left|\vec{B}\right| \cos\theta$

In the equation above, the value θ represents the angle between the two vectors. Therefore, the dot product of two vectors is equal to the product of their magnitudes multiplied by the cosine of the angle between the vectors. Since the vectors that we are working with have been converted to a unit vector, the magnitudes of the vectors are all equal to 1. Therefore, the angle θ is the arccosine of the dot product of our vectors.

Since we know how to calculate the angle between two vectors, we can determine the values for angle θ(using unit vectors Nu and Lu) and angle □(using unit vectors Ru and Eu). The angle θ is used to calculate the diffuse component. If the angle θ is zero, it means that the polygon is directly facing the source of light and hence, the diffuse value should be its maximum. If the angle is greater than 90°, it means that the source of light is in the opposite direction of the polygon and therefore there would be no diffuse lighting. We can use cosine to determine the amount of diffuse lighting we need since the value of cosine 0° is equal to 1 and

the value of cosine 90° is 0. However, we would need to ensure the angles greater than 90° are set to 0 and not some negative number.

As for the angle □, the core concept is very similar to what we did but there is a slight change that we would need to add. What makes specular lighting different is that it is a powerful beam of light we can see due to the way it reflects off a surface. While it is true that □ equalling 0° gives us the highest specular value, adding a few degrees would significantly drop the amount of specular value. This results in the image having a very bright spot which would tell us where the source of light is located. To be able to implement this, all we would need to do is raise the cosine of □ to the power of a large number(50 in our case). We can use a calculator to see why this works. If we raise the value 1 to the power of 50, we still get a 1. On the other hand, if we raise the value of 0.99 to the power of 50, we get the value of 0.605. The dropoff only gets worse the lower we go. If we raise 0.9 to the power of 50, we get a value of 0.005. Notice how this would mean the cosine of beta would have to be very close to 1 if we want to see any specular lighting.

Using all of these, we can get an intensity variable that we can compute as follows.

$$Intensity = Ambient_{max} + (Diffuse_{max} \cdot \cos\theta) + (Specular_{max} \cdot \cos^{50}\beta)$$

Although we now have the intensity value, it might be worthwhile to discuss a few edge cases. The first edge case we look at is if the normal vector is facing the opposite direction of the eye vector and the light vector. All we have to do in this case is negate the normal vector to make it face the same direction as the eye vector and the light vector. The second edge case is if the eye vector is facing the opposite side of the light vector and the normal vector. This would mean that we’re looking at the back side of the object, and hence the intensity would be equal to the ambient value. Lastly, if the light vector is facing the opposite direction of the normal vector and the eye vector, that would also mean that we are looking at the back side of the object and so we will only be using the ambient value to calculate the intensity.

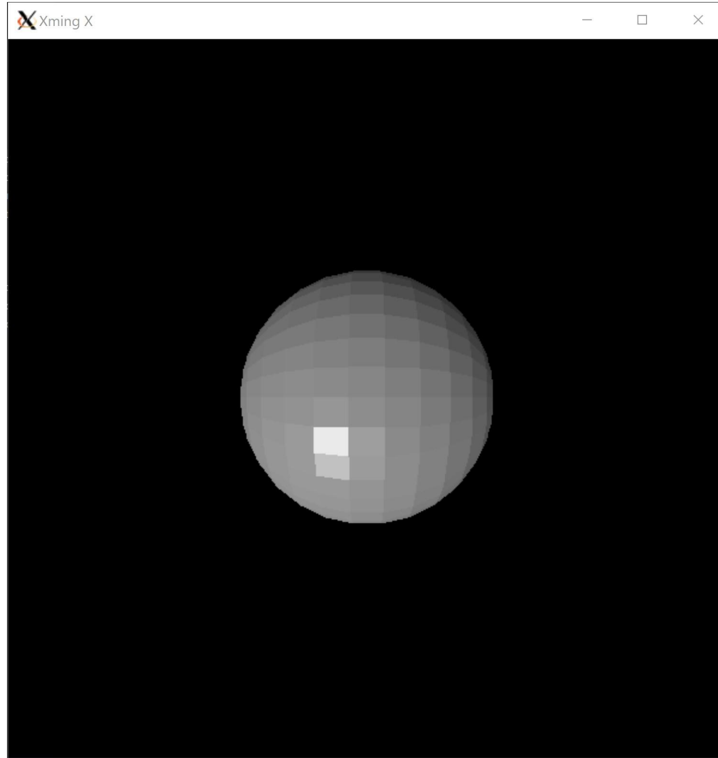

Figure 18: Image of the sphere using light models

Once we are done with our calculations, we can color the polygons to get something that looks like the image above. Notice the bright white light made because of the specular lighting. We can also notice that polygons located at the bottom left are brighter than those at the top right. This is due to the diffuse lighting. It’s hard to see but there is also ambient lighting for all of the polygons.

So far, we have been able to use some math to be able to add some lighting to our objects. Since we have been using an intensity variable to color the polygons, our image has been a form of grayscale. What if we wanted our sphere to be a shade of green? All we have is a sphere that looks gray. The good news is that with a little more math, we can color the sphere to be any color we want.

The first thing we have to determine is the color that we want to paint the object to. Of course, this color will need to have an RGB value. Once we have this, all we need to do is move the RGB value closer to white or black. The way we do this is by using the intensity that we calculated. We first separate the RGB value into its three components, red, green, and blue. Using the red component and the intensity, we can compute the value for the new red component. The same holds for the blue component and the green component.

For now, let's just focus on the red component. When determining the value of the new red component, there are three cases we need to look at; the intensity value being greater than, less than, or equal to the sum of the maximum ambient and diffuse values. As was mentioned earlier, we are using 0.2 for our maximum ambient value and 0.4 for our maximum diffuse value, which gives us a sum of 0.6. Of course, this is not something set in stone, it is just the values we have been using to calculate the intensity.

If the intensity value is less than the sum of maximum ambient and diffuse values(0.6), we are going to be shifting the color to black. The extent to which we shift the color to black depends on how close or far the intensity is to 0.6. If the intensity is close to 0.6, the color doesn't change much from its original value. If the intensity is something close to zero, the color will become dark. The image below gives some context on this. The first thing we need to do is figure out how close the intensity is to 0.6 as a percentage. 100% would mean the intensity is equal to 0.6 and 0% would mean that the intensity is equal to 0. Dividing I by 0.6 can give us a value from 0 to 1 that we can use as a percentage to determine the new color. All we have to do to find the new value is to multiply the old value by the percentage we just found. The formula is shown below.

$$newRGB = oldRGB \cdot (\frac{Intensity}{(Ambient_{max} + Diffuse_{max})})$$

$$newRGB = oldRGB \cdot (\frac{Intensity}{0.6})$$

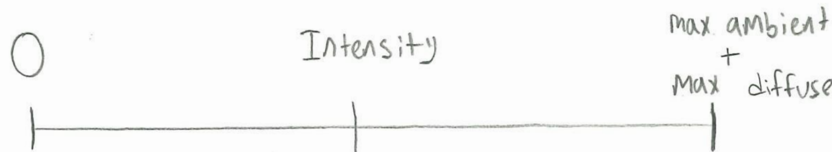

Figure 19: An intensity value less than the sum of maximum ambient and diffuse values

The second scenario is when the intensity is greater than the sum of maximum ambient and diffuse values. In this scenario, we are going to be shifting the color to white. As we have done before, we will need to calculate a percentage. The way by which we calculate the percentage is a little different from what we had done before. The reason behind this is that instead of having an intensity somewhere between 0 and 0.6, we now have an intensity somewhere between 0.6 and 1. Therefore, in this scenario, if we have a 0%, it means that our color stays the same and if we have a 100%, our color turns white. The formula to calculate the percentage is shown below

$$percentage = \frac{intensity - (ambient + diffuse)}{1 - (ambient + diffuse)}$$

Essentially, all we're doing to calculate the percentage is determining the distance between the current intensity and the sum of maximum ambient and diffuse values, and dividing it by the distance between 1 and that sum. It might look like we're using entirely different formulas but the reality is that they are pretty similar to one another. If you look closely, if we replaced the sum of maximum ambient and diffuse values with 0 and we replaced 1 with the sum of maximum ambient and diffuse values, the percentage we get is similar to what we had gotten for the first scenario.

Lastly, once we have the percentage, we can calculate the value for the new RGB. The way we use this percentage is a little different. Unlike the first scenario, if we have a 0% we

want it to retain the color it already had, and if we have a 100%, we want the color to turn white irrespective of what the original color was. To achieve this we can use the formula below. The percentage is the variable we calculated above.
$newRGB = oldRGB \cdot (1 - percentage) + percentage$

Now that we know how to calculate the RGB values when the intensity is greater than or less than the sum of maximum ambient and diffuse values, we need to find out how to calculate the RGB value where the intensity is equal to the sum of maximum ambient and diffuse values. The best thing about this case is that we don't need to do any more math to determine the new RGB value. We can use either of the scenarios above to find the new color. If we use the first scenario, the intensity divided by the sum of maximum ambient and diffuse values gives us a 1, and multiplying any number by 1 gives us the same number. If we use the second scenario, we first calculate the percentage, which comes out to be 0%. Plugging this into the equation also gives us the same color as we originally had. Ultimately, if the intensity is equal to the sum of maximum ambient and diffuse values, we don't change it.

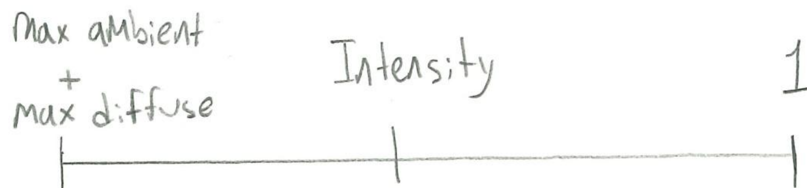

Figure 20: An intensity value greater than the sum of maximum ambient and diffuse values

Since we now have all of the formulas we need, we can use these formulas to compute the red, green, and blue values. Using those values allows us to color the objects to be whatever we want. The figure below shows us an image of a green sphere made using the computations above.

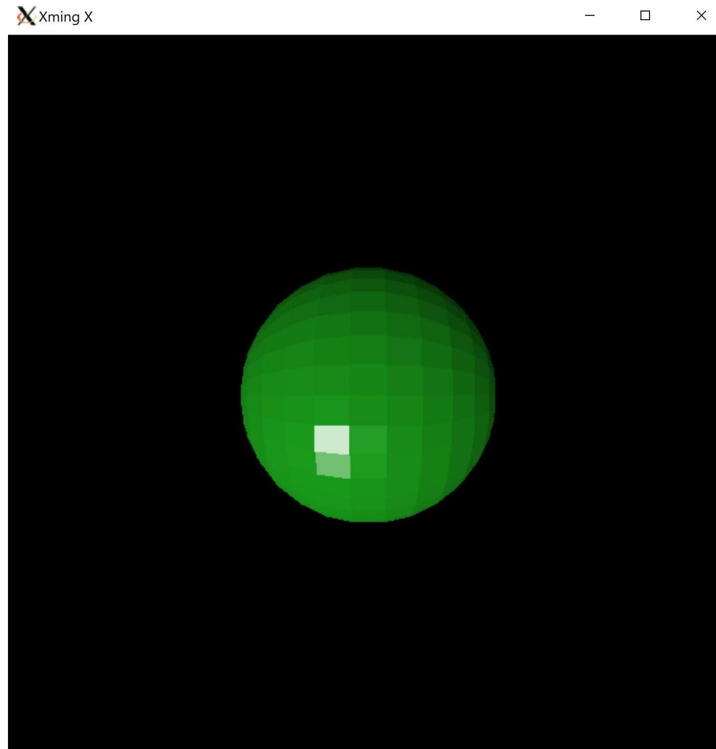

Figure 21: Image of a green sphere using light models

Ray Tracing

As we have seen above, light models are a great way of approximating lighting. However, they come with some limitations that we need to address if we want to create the most photorealistic images possible. Notice how we are using polygons to color the images. In doing so, a polygon gets colored uniformly. This is a disadvantage because it doesn't give us the most realistic images or the most accurate lighting. Looking at our sphere, for instance, the lighting we get makes it look more like a golf ball than a sphere. An easy way of getting past this is by increasing the number of polygons. Adding more polygons makes each polygon smaller, and hence gives a more realistic image. However, there is an even better way to generate photorealistic images, and it is called ray tracing.

When doing ray tracing, we will need to reimagine how we draw images into a screen. Instead of passing in points with XYZ coordinates and drawing polygons using them, we pass in equations of objects that our computer can use to ray trace. For instance, the equation below defines a sphere of radius R whose center is located in coordinates (X_0, Y_0, Z_0).

$$(X - X_0)^2 + (Y - Y_0)^2 + (Z - Z_0)^2 = R^2$$

Unlike what we have been doing so far, ray tracing works on a pixel-by-pixel basis. For every pixel in a screen, we shoot a ray that starts off at our eye, passes through that pixel, and intersects zero or more objects. If the ray intersects no objects, it means that there are no objects to be intersected at that pixel and hence we will give it the background color. If a ray intersects exactly one point, we calculate the lighting for that one point and color the pixel

accordingly. If a ray intersects more than one point, we will need to determine the point that is closest to the pixel and calculate the lighting for it.

It is worth noting that most modern ray tracers use sophisticated algorithms to create accurate reflections/refractions of objects. Due to the limited scope of this project, we will only be looking at the basics of how we can apply light models for every pixel to make a simple ray tracer. After completion, our ray tracer will be able to ray trace a sphere. Since the equation of a sphere differs from equations of other objects, our ray tracer will only be able to ray trace spheres.

Our first step when ray tracing is to determine where pixels are located in an XYZ coordinate system. For a 600 by 600 screen, an XY value of (0,0) would be the center of the screen. An X value of -300 would put us at the leftmost point of the screen and an X value of 300 puts us at the rightmost point on the screen. A Y value of -300 would put us at the bottom of the screen and a Y value of 300 would put us at the top of the screen. Using that we can determine the X and Y values for every pixel. Finally, we need to determine the Z value for each pixel. The great thing is that the Z value is the same for all pixels. Since we know that the screen goes 300 units up from the center, and we also know our half-angle, we can use some trigonometry to determine the Z value.

$$tan(halfangle) = \frac{300}{Z}$$

$$Z = \frac{300}{tan(halfangle)}$$

We now need to determine the location of our eye. As we have been doing for most of this paper, our eye will be located at (0, 0, 0). Using the location of our eye and the location of the pixel we are raytracing, we can create a ray. Once we have the ray, we can easily determine whether a certain point falls on it. The first step however is to find this ray. All we have to do is find the change in X, Y, and Z. Since our eye is located at (0,0,0) in our case, the change in X is essentially the X value of the pixel. The same holds for the Y and Z variables. Once we have this, all we need is the origin of the ray. Again, we know that the ray starts at our eye so we can use that to determine the formula which we can see below.

$$\begin{bmatrix} X_{ray} \\ Y_{ray} \\ Z_{ray} \end{bmatrix} = t \cdot \begin{bmatrix} pixel_x - eye_x \\ pixel_y - eye_y \\ pixel_z - eye_z \end{bmatrix} + \begin{bmatrix} eye_x \\ eye_y \\ eye_z \end{bmatrix}$$

In this formula, t can be any non-negative number and it will give us a value (X_{ray}, Y_{ray}, Z_{ray}) that the ray will cross through. We can use this to determine if the ray intersects a point in the sphere.

Looking at the equation for a sphere that was provided earlier, we know that the center of the sphere and its radius was given to us. Therefore, any X, Y, and Z values that solve the equation are valid points in the sphere. Notice how we can plug in the X_{ray}, Y_{ray}, and Z_{ray} in terms of X, Y, and Z. Ultimately, plugging these variables into the sphere gives us a quadratic equation in the form below, where A, B, and C are some numbers.

$$A \cdot t^2 + B \cdot t + c = 0$$

We can solve for t using the quadratic formula. Doing so will give us 0, 1, or 2 solutions. If we get no solutions, it means that the ray does not intersect the sphere. If we get one solution,

it means that the ray only intersects the sphere at one point. If we get two solutions, it means that the ray intersects the sphere at two points. If we get back two points when solving the quadratic equation, we will use the smaller t value, as it will be the point closest to our eyes. If we have more than one sphere in our scene, we do the same thing for every sphere and keep track of the point closest to our eye.

With that done, we are now very close to completing our ray tracer. All we need to do now is figure out the lighting for the point we are looking at. If our ray tracer was unable to find a point that intersects the sphere, we give it the background color. If we have a point in a sphere, all we need to do is apply a light model for that point. If you remember, light models need a normal vector to operate. This can normally be a little tricky and would require some calculus knowledge. Thankfully in our case, the vector that starts at the center of the sphere and ends at the point we're looking at is essentially the normal vector. We can then apply light models and color the pixel accordingly.

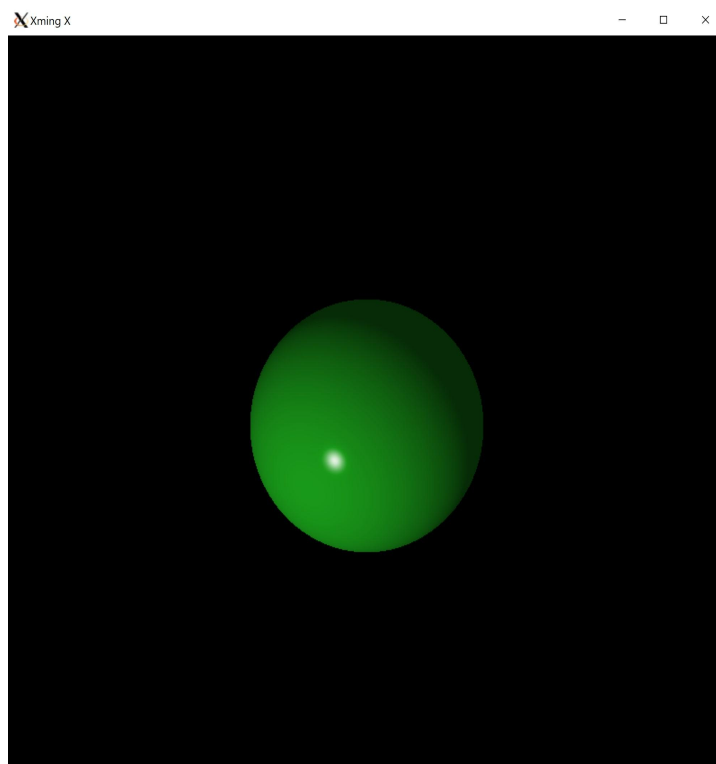

Figure 22: Image of a green sphere using ray tracing

Congratulations! We have created a very simple ray tracer. Notice how the sphere above looks a lot smoother and closer to a real-life sphere than what we have been doing before. Now that we know how ray tracing generally works, it might be worth discussing some of the advantages and drawbacks of ray tracing.

Advantages of Ray Tracing

Ray tracing offers the most photorealistic images compared to other methods. In our sphere, for instance, we have moved from using polygons to try and accurately model a sphere to having an actual equation of a sphere that we can use to draw on a screen.

Of course not every object in real life is a sphere, we can still have objects or other kinds of surfaces. Since ray tracing goes over every pixel on a screen to determine the color it should be, ray tracing would still offer the most realistic form of lighting.

Disadvantages of Ray Tracing

As realistic as ray tracing is, there is a reason why it is not used everywhere. The biggest drawback of ray tracing is that it can be computationally expensive. Even for our simple ray tracer, we have to shoot rays to every pixel of the screen to determine what color our pixel will be. As you can see, this can take a very long time. If we wanted to add more spheres to our scene, it would take even longer since we have to check if our ray intersects any of the spheres. This time complexity only gets worse with advanced ray tracers since they would have to do more computations to determine the most accurate color for every pixel.

Due to this issue, ray tracing has not been widely adopted in some fields until recently. For instance, in the gaming industry, performance is crucial. Games need to refresh the screen multiple times every second. This makes ray tracing a less-than-ideal choice. On a positive note, advances in computer hardware and software are allowing for ray tracing to be possible in modern graphics cards.

In the animation industry, ray tracing is more common since there is a greater focus on realism than performance. Many animation studios have what are called “render farms” which are powerful computers that can render animations. These computers render the animations on a frame-by-frame basis, which get stitched together to create the movie.

Another minor disadvantage of ray tracing is that since different objects have different equations, the process of ray tracing is different for every object. Notice how when building our ray tracer, we had to use the equation of a sphere to calculate the point of intersection. While the core concept of ray tracing remains the same for every object, we can't use our ray tracer to ray trace a cylinder or some different object.

Conclusion

Over the course of this paper, we have gone from being able to draw points and simple polygons on a screen to being able to use ray tracing to accurately model a sphere. Of course, it was a long process that required a lot of math but hopefully, you’ve gained some knowledge from this paper.

As a quick recap, we started this paper off by introducing the graphics library Gkit which we have been using to create all of those cool-looking images. We then quickly defined xyz files and how we could use them to draw points and polygons in 3D space. After doing so we looked at the process behind converting those points in 3D to 2D so that we could draw them onto a computer screen. The next thing we did was add rotation to our objects. That allowed us to view our objects from different angles.

Once we had that done, our next step was to add lighting to our objects. We used vectors to calculate the lighting of every polygon for our objects. Initially, the process behind this gave us lighting that was in grayscale. We added a little bit more math to be able to add colors

to our objects, which was what we did. Lastly, we looked into a different system of drawing objects, and that system is referred to as ray tracing. Ray tracing allowed us to create realistic objects.

As was mentioned before, most commercially used ray tracers are a lot more advanced than the ray tracer we have made. They usually have accurate reflections, where light can bounce off a surface and onto another object. They can also model refractions, that can help simulate some objects. All of these things added together make ray tracing an ideal method with which we can generate photorealistic images.