Internal Assessment

Mathematics Analyses and Approaches HL

Modelling the Earth with Math and Technology

Introduction:

CGI, video games, social media filters, 3D modelling. What do all the following have in common? Almost all of them involve some form of 3-dimensional rendering. But how does the computer know how to render all the cool digital effects? This question popped into my mind as I had recently been getting into video game development, which heavily utilizes programs called shaders which execute math in blazing speeds to render digital effects, such as a glowing box. In this paper, I will be investigating the math under the hood of computers that drive graphics rendering.

A topic that instantly appeared in my mind was modelling planets. I was interested to see how math can be used in computer rendering to mimic nature, and at the same time I'm interested in space and rocketry, which is why I decided on modelling the Earth for this investigation.

Aim and Approach:

In this investigation, I will be applying math in programming a quad-frag shader. Typically, 3D models are defined as a set of vertices which the computer can fill to display a coherent shape. However, a quad-fragment shader only has access to the pixels across the screen. As such, simple things such as displaying a sphere become significantly more difficult. The challenge of how to color pixels on a flat plane to mimic a "3D feeling" is the crux of this investigation. Like human culture, there are many different shader languages, of which I will be using GLSL (OpenGL Shader Language). For context, it is not necessary to understand all the differences between different languages, but there are common functions and datatypes which are crucial to understanding this investigation. Just like math, there are different "types" of numbers. An integer is a positive or negative whole number, or zero (Britannica 2021). A float is simply a

fraction represented in decimal form. For example, the number 4.02 is a float. In shader code, a vector simply stores an arbitrary number of a datatype. GLSL provides a set of vectors ranging from two to four dimensions, which could be used to represent a position or direction. In this investigation, the Earth, which will be represented by vectors in 3D space, are displayed to the 2D screen through the view of a "camera." There are no actual cameras in a shader, but instead points with direction. Since a camera represents a static view of the 3D space (a picture, per se), which is displayed, one could imagine that a position vector represents the picture is being taken, and the direction vector represents where the camera is looking. The main concept used to render the scene is raycasting, which involves finding intersections between rays (origin at the camera), and geometry in the scene, such as a sphere. Rays themselves can be represented as vectors. As such, this investigation will heavily utilize vector math.

Aim:

The aim of this investigation is to create a cohesive model of the Earth as it looks like from space. To do so, I will be writing a GLSL shader program to display the Earth. In this context, the fragment shader provides access to the RGB value (red, green, blue) of each pixel on the screen, as well as a set of math functions which I can use to manipulate numerical values. This includes the pow, step, sqrt, exp, dot, and other various math functions. To properly display the Earth, I will modify the RGB value at the proper coordinates, as if painting a picture of the Earth. The functions provided by the library used in this exploration are as follows:

Function	Return value
$pow(x, n)$	x^n
$step(x, l)$	$\chi_A(x, l) = \begin{cases} 1 \text{ if } x < l \\ 0 \text{ if } x > l \end{cases}$
$\text{sqrt}(x)$	$\sqrt{x}$
$\exp(x)$	e^x
$\text{dot}(\vec{a}, \vec{b})$	$\vec{a} \cdot \vec{b}$
$\text{mix}(x, y, a)$	$x(1-a) + y(a)$
$\text{normalize}(\vec{v})$	$\lvert\vec{v}\rvert$

1. Exploration:

1.1 Intersection Derivation

How does one go about displaying 3D objects on a 2D screen? Instead of trying to transform the 2D pixels into 3D objects, it would be much easier to transform our view of the 3D scene onto a 2D canvas. To do so, a "camera" functions as a viewport, displaying the 3D scene onto the computer screen (see figure 1). A camera can be placed at any arbitrary location, represented by some position vector P_{cam}. Lastly, the center for the Earth can also be represented by a position vector P_{Earth}. Through the camera, a "ray" can be shot out for each pixel on the screen, like how photons enter humans' eyes. If a ray intersects an object, the camera knows to color that pixel using some coloring function I define. Otherwise, there is no intersection. Having no intersection is essentially the same as hitting empty space. For other situations, knowing that there's no intersection could mean coloring it as the sky, but since the Earth is in space, the color black will be colored in for points of no intersection.

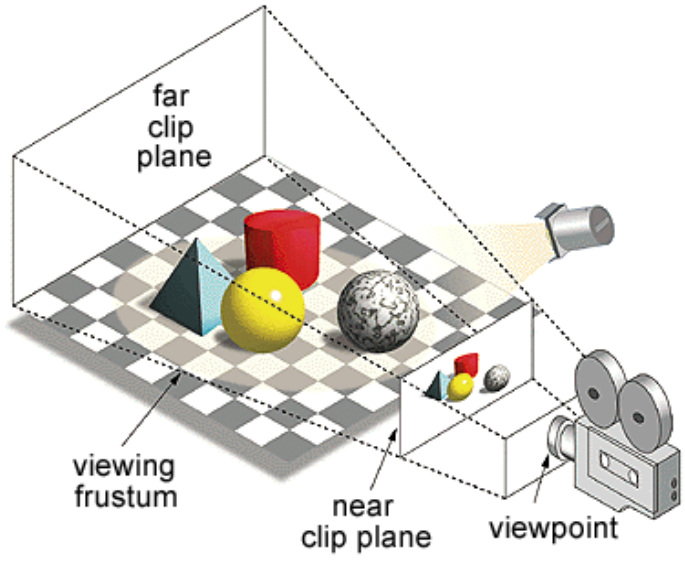

Figure 1 camera (Tang, 2018)

Each individual ray shot out from the camera can be thought of as a directional vector originating from the camera. As such, it follows that the intersection point of a ray with an object can be defined as $x = x_0 + \vec{r}_d t$, where $x_o$is the origin of the ray (i.e., camera position), $\vec{r}_d$is the direction of the ray, and t is the distance between x_0 and the surface of the object. Only intersections with the surface of an object are checked, since the Earth is opaque, and so only the surface will be seen. In the equation $x = x_0 + \vec{r}_d t, t$ is still unknown. However, given that the equation for a circle is $|P - C|^2 = r^2$ where P is a point on the surface of the sphere and C represents the center of the sphere, the equation for x can be substituted in for P, resulting in the equation $(x_0 + \vec{r}_d t - C)^2 = r^2$. As $x_0 + \vec{r}_d t - C$ results in a vector, squaring it is the same as taking the dot product. Given that the general form of a quadratic is $ax^2 + bx + c$, multiplying out the left side results in a quadratic polynomial, where $\vec{r}_d, C, r \; and \; x_0$ are known. A quadratic makes sense, since like a parabola, there are 3 cases: One, there are two zeroes, meaning that the distance from the ray to the sphere is 0, and thus intersects at an exit point. Two, there is one

zero, meaning that there is only one point at which the ray intersects the sphere. Three, there are no real zeros, meaning that the ray doesn't intersect the sphere.

$$\vec{r}_d t^2 + 2x_0\vec{r}_d - 2C\vec{r}_d t + x_0^2 - 2x_0 C + C^2 - r^2 = 0$$

Simplifying, the equation becomes:

$$\vec{r}_d t^2 + 2\vec{r}_d t(x_0 - C) + x_0^2 - 2x_0 C + C^2 - r^2 = 0$$

However, one may observe that $x_0^2 - 2x_0 C + C^2$ in the c term of the quadratic is simply a difference of perfect squares, and can be simplified further:

$$\vec{r}_d^2 t + 2\vec{r}_d t(x_0 - C) + (x_0 - C)^2 - r^2 = 0$$

According to the equation for vector dot product, $A \cdot B = |A||B| \cos\theta$. Since $\vec{r}_d$ is a direction vector (and thus normalized), we know that when we square the terms, that the a term of the quadratic is simply $1 * 1 = 1$, which eliminates the coefficient for the t^2 term, resulting in the final equation of

$$t^2 + 2\vec{r}_d t(x_0 - C) + (x_0 - C)^2 - r^2 = 0.$$

Given the equation for the reduced quadratic equation as $x_{\frac{1}{2}} = -\frac{p}{2} \pm \sqrt{\left(\frac{p}{2}\right)^2 - q}$ where

$p = \frac{b}{a}$, and $q = \frac{c}{a}$, we can then use the reduced quadratic equation to solve for t, since $\vec{r}_d, x_0, C,$ and r are given variables, and the a term is just 1.

$$t_{\frac{1}{2}} = -\frac{p}{2} \pm \sqrt{\left(\frac{p}{2}\right)^2 - q} \qquad p = 2\vec{r}_d t(x_0 - C), q = (x_0 - C)^2 - r^2$$

$$t_{\frac{1}{2}} = -\frac{2\vec{r}_d(x_0 - C)}{2} \pm \sqrt{\left(\frac{2\vec{r}_d(x_0 - C)}{2}\right)^2 - ((x_0 - C)^2 - r^2)}$$

$$t_{\frac{1}{2}} = -\vec{r}_d(x_0 - C) \pm \sqrt{\left(\vec{r}_d(x_0 - C)\right)^2 - (x_0 - C)^2 + r^2}$$

From the discriminant, we can determine whether the sphere has been hit. If the discriminant is negative, then there was no intersection between the ray and the sphere. If the discriminant equals zero, then there is one intersection point (see figure 2). If the discriminant is positive, then the ray intersects the sphere at two points: an entry, and an exit point.

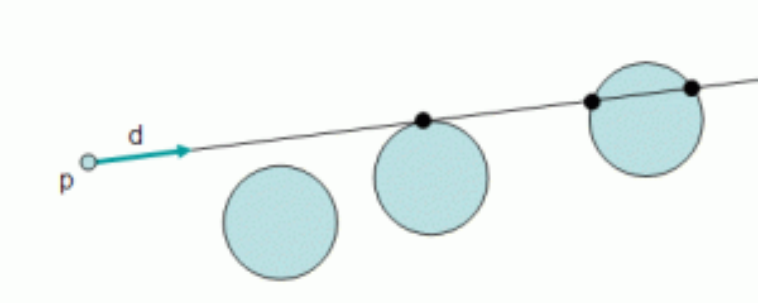

Figure 2 intersections (Ray-sphere intersection, 2011)

However, a major aspect has not been addressed. Where does the camera point? For context, the coordinate system in a fragment shader is unlike a traditional cartesian plane. Coordinates are normalized, meaning that the bottom left corner has the coordinate point $(0, 0)$ and the top right corner is the point $(1, 1)$. These axes are commonly denoted by "u" and "v" instead of "x" and "y." Since the camera position is simply a position vector, it holds no information about the camera direction (ray direction). Yet, the screen is the viewport of the camera, meaning that the camera must be looking at *something*. By convention, the camera faces into the screen, $z = -1$. As such, the default ray direction is $(u, v, z) = (u, v, -1)$. Nevertheless, this does not necessarily mean that the camera will be facing at the Earth. This can be remedied with a transformation. As

our camera is in three-dimensional space, there are also three axes that need to be defined, the x, y, and z axes. The z axis can be found by subtracting the target position (in this case, the earth) and the camera position, as shown in figure 2. Since the cross product of two vectors produces a vector which is perpendicular to both, the x axis can be found as the cross product between the z axis and the up vector $(0, 1, 0)$. The same principle can be applied to find the y axis, by taking the cross product of the x and z axis (see figure 4). Creating the transformation matrix as $\begin{bmatrix} x_x & y_x & z_x \\ x_y & y_y & z_y \\ x_z & y_z & z_z \end{bmatrix}$ where each column represents the vector of an axis, and then multiplying it by the default direction of the camera $(u, v, -1)$ results in the correct camera orientation.

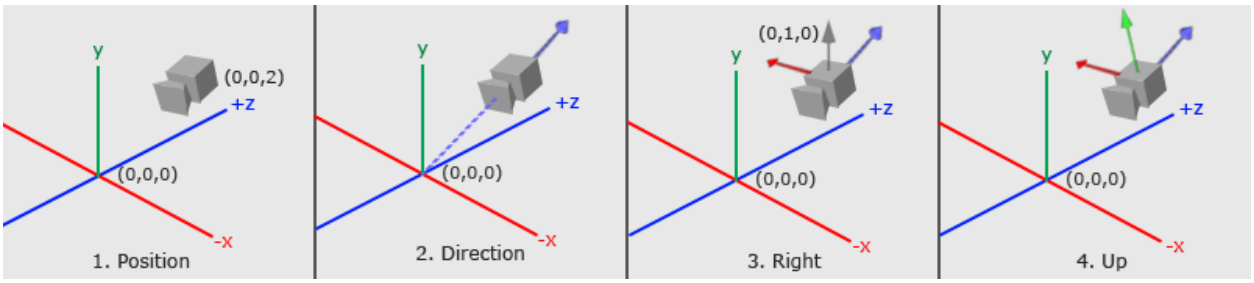

Figure 3 camera orientation (Vaughn, 2021)

1.2 Coloring and Shading

The program can then run the sphere intersection equation for every pixel on the screen to tell if it intersects the sphere, to which an arbitrary color can be assigned to each intersection. However, if the program were to run, the result would be a purely black screen. This is because despite knowing the distances, the computer has no way to quantify whether to color the pixel. Therefore, we can write a step function which returns 1 if the camera ray intersects the sphere, and 0 if there is no intersection. This can be represented by the function

$\chi_A(x) = \begin{cases} 1 \text{ if } x < 1000 \\ 0 \text{ if } x > 1000 \end{cases}$. Since colors can be represented as three component vectors, multiplying $\chi_A(x)$ by some RGB color C will render the intersection point with the color C, while non-intersections are $0 * C = (0 \quad 0 \quad 0)$, or the color black (see figure 4).

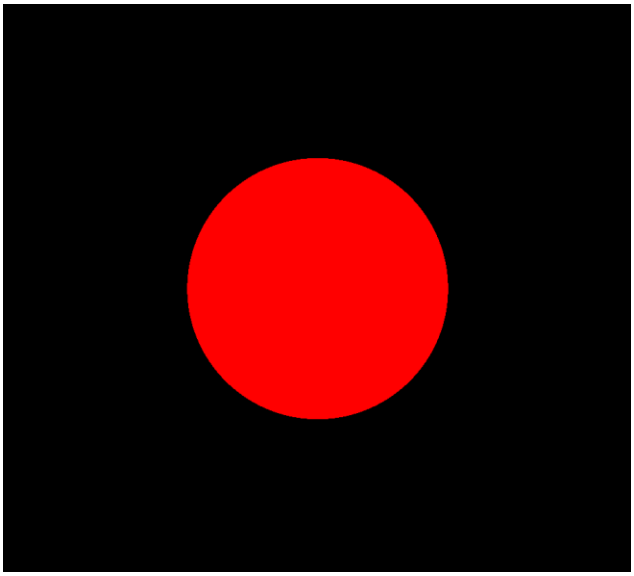

Figure 4 sphere intersection colored red

As shown in figure 4, the rendered object isn't so much of a "sphere" rather than a circle. This is because the entire sphere is colored uniformly, unlike a 3d object which has shadows, reflections, etc. A quick and easy way to shade a 3D object is by a technique known as diffuse shading. The question then rises of how to relate the light source vector to the sphere. By the definition of the dot product, the dot product of two vectors equates to a scalar representing the amount one vector goes in the direction of another, given by the equation $\vec{a} \cdot \vec{b} = |\vec{a}| \left|\vec{b}\right| \cos\theta$. Through this equation, it becomes apparent that values of θ in the first and fourth quadrant will yield positive cosine values, while values in the other half of the unit circle will yield negative values. This concept can be applied to the "sphere" drawn on the screen. The front hemisphere facing the light source should be bright (positive values), while the backside should be dark

(negative values). For every point on the circle, there is a corresponding normal vector. By comparing the angle of the normal to the direction of light, we can then determine how bright to shade that point. However, it is worth noting that since the goal is to find the amount of light reflected by the sphere, the direction vector of light is not coming from the light source, but inversed, so that the light vector points into the light source. Take for example, a point facing directly into the light source. The angle between the normal and light vectors would be 0, meaning that the dot product would not be scaled and thus stay at maximum intensity. For a point on the opposite side, facing away from the light source, the angle would then be π. Of course, $\cos\pi = -1$, so the dot product would be the most intense dark spot on the sphere. As θ goes from 0 to π, the intensity goes from 1 to -1. Using the dot product, I can achieve a proper three-dimensional portrayal of the sphere, with a shadow.

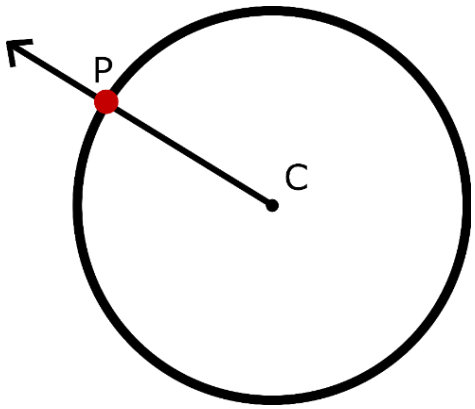

Figure 5 calculating normal

For brightness b at a point $P, b_p = \vec{l} \cdot \vec{n}_p$, where $\vec{l}$ is the direction that light comes from, and $\vec{n}$ is the normal vector at a point P. In other shapes, the normal could require complex calculus to evaluate. However, all points on a sphere are equidistant from the center, meaning that the normal at point P is simply the vector $\overrightarrow{CP}, or\ n_p = P - C$ (see figure 5). P is the same as the intersection point between the ray and the sphere, which means that the normal can be calculated

as

$n_p = x_o + \vec{r}_d \cdot \left(-\vec{r}_d(x_0 - C) - \sqrt{(\vec{r}_d(x_0 - C)^2 - (x_0 - C)^2 - r^2}\right) - C$. However, the square root in the equation yields two solutions, one positive and one negative. This is because the ray, like an arrow piercing an apple, may enter and exit the apple at two distinct points (see figure 2). Nevertheless, only the negative root needs to be considered, since that intersection point represents the entry point, while the other root does not need to be considered.

However, a caveat with this shading method is that the extreme areas, points which are on the opposite side of the light, will have negative or zero dot products, creating extremely dark zones, which seem unrealistic (see figure 6).

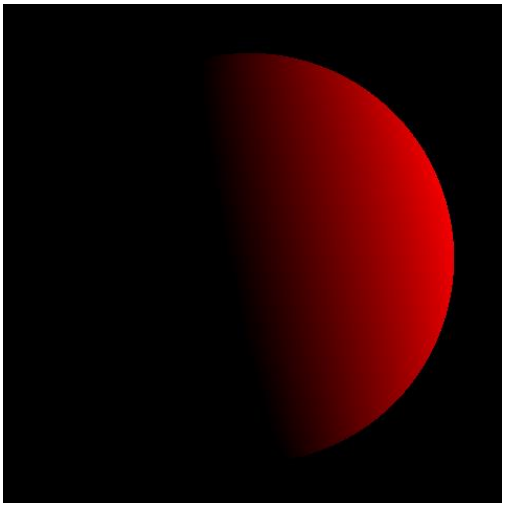

Figure 6 unmodified diffusion shading

A solution is to shift the domain of the dot product. Unmodified, the dot product is at least -1, and at most, 1. This domain is different from my expectation since in real life, light cannot have negative intensity. Therefore, the domain of the brightness function should be shifted to $[0, 1]$ by multiplying the dot product by 0.5, and then adding 0.5. Applying those transformations on the lowest value -1 results in 0, the new minimum domain value. Although, shifting the domain

illustrates an unnatural amount of ambient light for the purpose of this investigation, shown in figure 7.

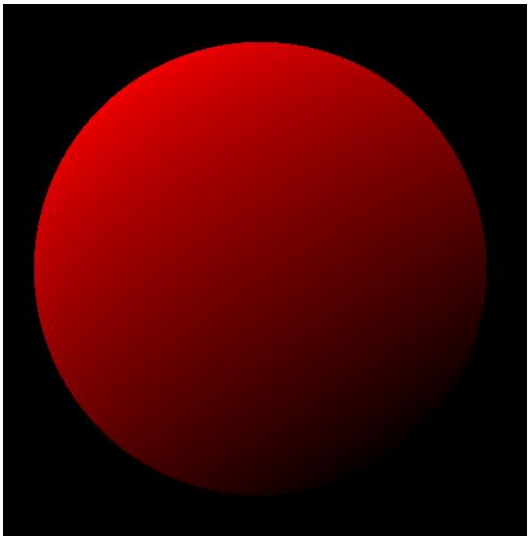

Figure 7 domain shifted brightness is unrepresentative of the Earth's brightness

What I decided to do was take a blend between the two styles of diffuse shading by linearly interpolating between the values produced by both functions, given a weight a (mix, n.d.). The resulting function is:

$b_p(a) = (\vec{l} \cdot \vec{n}_p)(1 - a) + \left((\vec{l} \cdot \vec{n}_p) \cdot 0.5 + 0.5\right) a$. With $a = 0.2$, the resulting shadows are softer than that of figure 5, but harder than figure 6 (see figure 8). This shading function most closely replicates the brightness of the Earth in each hemisphere.

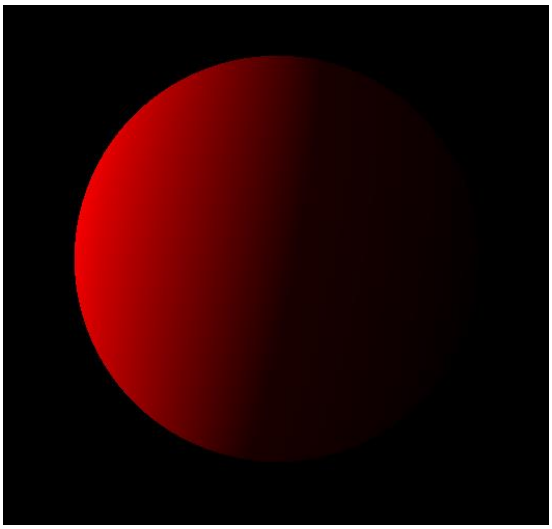

Figure 8 mixed diffuse shading

Looking at 3D objects, one might observe that most 3D objects have a "shiny spot," also referred to as a specular highlight; the same can be said for the Earth. The specular highlight can be calculated as $k_{spec} = \max\left(0,\ (\vec{R} \cdot \vec{V})\right)^n$, given $\vec{R}$, the reflection of light off the surface of the object, $\vec{V}$, the normal vector at point P, and n, a smoothness constant (Lyon, 1993). Overall,

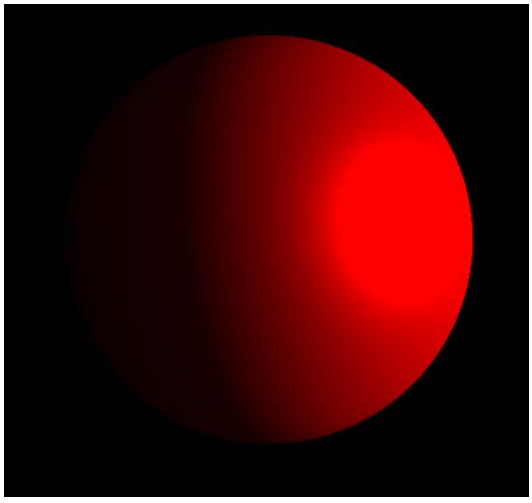

Figure 9 phong specular shading with n = 16

while the phong illumination model (diffusion, ambient and specular shading) can provide decent, recognizable 3D shading, it is neither the best method, nor the most realistic. The reason

being that lighting is much more complex. In reality, the Earth is not the only celestial body floating in space, nor is the sun the only light source. Millions of other stars also act as light sources. A common topic mentioned in physics is raytracing. In essence, raytracing manually calculates the reflections off other surfaces, providing much more accurate lighting and reflections than the phong model. A tradeoff is that raytracing is exponentially more computationally expensive. One could reason that the computer would need tens to thousands of calculations to simply color one pixel, out of the millions of pixels on the screen. In this investigation, I made the judgement of using the phong illumination model, since far away light sources such as stars provide negligible light. By using phong shading, the computation time is also exponentially faster than raytracing and can thus be used in real time computer graphics and video games, a major interest of this paper. However, a purely red sphere doesn't look like the Earth at all. Although, by sampling a cubemap of the Earth, the result looks convincing (see figure 10).

Figure 10 cubemapped Earth

The principle behind cubemapping is the same as how the Earth scene is rendered by a camera. An equirectangular photo of the Earth can be split into six faces of a cube. A “camera” can be placed in the middle of the cube, shooting rays in all directions. As each ray intersects the cube, the color of the pixel at the intersection of the cube is the same as the point on the sphere.

2. Conclusion

Figure 9 result, after applying Rayleigh Scattering

Overall, the final Earth shader achieved the goal of the investigation. The output created by the program, while not completely true to reality, looks similar enough to the way light bounces off the real Earth, and could pass for something found in a video game. I was able to apply core ideas of vector math that I learned in class in a completely different environment, improving my ability to apply my mathematical knowledge in different areas of inquiry. However, there are limitations to this model. Namely, the texturing of the sphere will never be truly accurate as there is no way to continuously map a texture to a sphere without a discontinuity, using regular uv mapping (Burns & Gidea, 2005). Additionally, while the model does not illustrate every detail in

the Earth's geometry (i.e., mountain ranges, ocean depths), I think that it properly fulfills its role as a model, a simplified representation of the Earth.

I learned about how math can be used between different dimensions to create seemingly impossible creations, and the beauty of math that allows humans to mimic nature. The math explored in this investigation has many applications outside of this exploration. While vector math is useful in computer graphics, the concepts utilized also lend themselves heavily to physics, and other real-life applications.

In the future, I could consider improving on my model of the Earth by adding moving clouds (trigonometry), 3D terrain offsets, and mathematically procedurally generating stars in the background. This would not only serve as a fun project but help expand my knowledge and application of math.

All in all, this exploration has inspired me to find new ways of representing nature through math, especially as my interest in game development and computer graphics increases.

References

Britannica, T. Editors of Encyclopaedia (2021, April 6). integer. Encyclopedia Britannica. https://www.britannica.com/science/integer

Burns, K., & Gidea, M. (2005). *Differential geometry and topology: With a view to dynamical systems*. CRC Press.

Lyon, R.F. (1993). *Phong Shading Reformulation for Hardware Renderer Simplification*.

Jakob. (2020, December 18). *Ray-sphere intersection - from math to code*. Fifty Lines of Code. Retrieved January 31, 2022, from https://fiftylinesofcode.com/ray-sphere-intersection/

mix - OpenGL 4 Reference Pages. (n.d.). Retrieved January 31, 2022, from https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/mix.xhtml

Ray-sphere intersection. Lighthouse3d.com. (2011, June 3). Retrieved February 15, 2022, from https://www.lighthouse3d.com/tutorials/maths/ray-sphere-intersection/

Tang, V. (2018, December 19). *Intro to 3D with Three.js*. Medium. Retrieved January 31, 2022, from https://tangiblecodes.medium.com/intro-to-3d-with-three-js-c826891f452

Vaughn, N. (2021, April 26). *Build an alternative camera model that looks at a target!* inspirnathan. Retrieved February 15, 2022, from https://inspirnathan.com/posts/56-shadertoy-tutorial-part-10/