Representing and Computing Polarized Light in a Ray Tracer

A Technical Report
in STS 4600
Presented to
the Faculty of the
School of Engineering and Applied Science
University of Virginia
in Partial Fulfillment of the Requirements for the Degree
Bachelor of Science in Computer Science

by

Jacob Welsh

April 19, 2012

On my honor as a University student, I have neither given nor received unauthorized aid
on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Signed: _____ Date: _____

Approved: _____ Date: _____
Jason Lawrence, Department of Computer Science

Representing and Computing Polarized Light in a Ray Tracer

Introduction

Ray tracing is a common technique for rendering computer-generated imagery. It works by projecting mathematical rays into a three-dimensional scene containing geometry; when an intersection is found, illumination is computed based on material properties, light sources, and new rays traced recursively (Glassner, 1989). Its advantages include the easy modeling of many physical behaviors of light; its biggest downfall is inefficiency, especially with the more realistic effects. Thus it is well suited to photo-realistic applications such as still image or movie rendering, and less so to real-time applications such as video games or 3D modeling software.

One property of light that has not been studied much in the context of ray tracing is polarization. Since light can be viewed as a transverse wave in the electromagnetic field, it has a spatial orientation, orthogonal to the direction of propagation. If the electric field vector at some fixed point is oscillating in a straight line, the light is said to be linearly polarized, with an orientation given by that line. If the field has a constant magnitude but a rotating direction, the light is said to have left or right circular polarization, depending on the direction of rotation. Or it could be a mixture of both, in which case it is described by a polarization ellipse (Hecht, 2002, pp. 325-329). In reality, a beam of light is composed of a large number of photons, each of which has some polarization state. Typically, these are incoherent and have a random distribution of states, resulting in unpolarized or partially polarized light (Hecht, 2002, p. 330). Such natural light can be statistically described by its intensity I, degree of polarization p, orientation ψ, and ellipticity angle χ, all functions of wavelength. A more useful representation is the Stokes vector, consisting of the four parameters $S_0 = I$, $S_1 = I\,p\cos2\psi\cos2\chi$, $S_2 = I\,p\sin2\psi\cos2\chi$, and $S_3 = I\,p\sin2\chi$.

The human eye is not sensitive to polarization, and most natural light sources emit unpolarized light. It does occur in the natural world though, such as in reflection off shiny surfaces and scattering in the atmosphere. This can be exploited for the purpose of glare reduction by using polarizing filters in

photography, or polarized sunglasses.

My goal in this project was to add support for polarization to a ray tracing system, and use it to model these two effects in synthetic imagery. In this paper I detail the abstract data types I used to represent polarized light and the operations that needed to be defined on them, as well as some other improvements I made to the code, and present a sample of the images that can be generated with the new framework. Also, I had hypothesized that, given the full polarization state at each pixel of an image, a virtual “ideal glare reduction” filter could be implemented by removing the polarized component of all light regardless of angle; experiments with the ray tracer showed this not to work as expected in the general case, and I explain why this is so.

Implementation

The ray tracing code base in question was developed for Prof. Jason Lawrence's introductory Computer Graphics course; I had re-implemented much of its functionality as an assignment in that course. It was written in C++ and implemented basic functionality including spheres, triangles, reflection, refraction, shadows, texture mapping with bilinear sampling, a simple Phong model for surface illumination, and reading the scene description from a text file. Points and vectors were represented with a Point3D class, colors with a Point3D representing red, green, and blue components, and rays with a Ray3D class containing one Point3D for the position and another for the direction.

The primary result of the project was to be an improved reflectance model for specular (shiny) surfaces; to demonstrate this would have required creation of a detailed scene in the ray tracer's non-standard file format. Instead, my first step was to implement support for environment maps, so that photographic imagery could be used as the background of the scene. I used the “grove” image from Paul Debevec's online Light Probe Image Gallery; that provided the requisite mapping from a 3D direction in the scene to 2D image coordinates.

To represent polarization, I added a StokesVector class. The basic operations defined on this data type are addition, scalar multiplication, and Mueller matrix multiplication (described later).

Addition represents two separate, incoherent light sources being combined into one ray; one advantage of using the Stokes vector representation is that this is simply implemented as component-wise addition. Scaling the vector by a number between zero and one represents attenuating the light, as would happen in a partial reflection or transmission where polarization is not affected. Subtraction and multiplication by a negative are undefined, as light cannot have a negative intensity. There are also functions to convert a Stokes vector to and from the (I, p, ψ, χ) "angles" form. To support color images, I added a RayColor class, containing a StokesVector for each of the red, green, and blue wavelengths, and replaced the parts of the code that stored color in a Point3D with a RayColor.

One more detail was needed to make these representations useful: a way to determine the direction in space relative to which the polarization angle is measured. I considered using some global constant, such as the y-axis or the camera's "up" axis, but this would result in a singularity for rays parallel to that direction (which could happen, for instance, with reflected rays). Instead, I extended the Ray3D class into an OrientedRay3D by adding an orientation vector which, together with the ray's direction, defines a reference frame for the Stokes parameters for each ray. To facilitate computation of angles, these vectors are required to be orthogonal and of unit length. The choice of orientation for a ray is arbitrary, so this can be achieved for the initial camera rays by orthogonalizing the camera's "up" vector with respect to the direction. Conveniently, this results in the final polarization angles being relative to the axes of the image plane.

Another function defined on the StokesVector for convenience is reorientation. Sometimes the polarization can be most easily computed in one reference frame, but must be returned in another. This function takes in the ray's direction, old, and new orientation vectors, and computes the new Stokes parameters. Since the vectors are required to have unit length, and the orientations to be orthogonal to the direction, the angle to rotate ψ can be found with a few vector products. If the direction is D, old orientation A, and new orientation B, then $\cos(\Delta\psi) = B \cdot A$ and $\sin(\Delta\psi) = (B \times A) \cdot D$. If an angle-based representation were used instead of a Stokes vector, this reorientation would be computationally

cheaper (at the expense of addition), since here ψ must be computed from S_1 and S_2, the rotation added, then converted back.

The next step was to implement the Fresnel equations. These describe the degree of transmission and reflection when light reaches a boundary between media of differing optical density, such as air and water or glass, and depend on index of refraction, angle, and polarization (Fig. 1). Specifically, there is one pair of equations for light polarized parallel to the plane of incidence, and a different pair for perpendicular polarization (Hecht, 2002, pp. 113-115). To handle this, I assume that the incoming light is unpolarized, and thus can be decomposed into fully polarized components in the parallel and perpendicular directions. This is done for both the reflected and transmitted rays, and a new StokesVector is built by mixing these with coefficients given by the equations.

I also implemented a "thin surface" model, to approximate the effect of two nearby parallel surfaces, as in a pane of glass or a hollow sphere. In the original ray tracer this was achieved by specifying 1 as the index of refraction, but with the Fresnel model that would correctly produce an invisible surface. In the thin approximation, the transmission coefficients are squared from the single-surface case, reflection coefficients adjusted based on conservation of energy, and the "refracted" angle equals the incident angle. This is valid as long as the surface is thin enough that the multiple reflections cannot be distinguished, but not so thin that wave interference effects occur. It produces a particularly nice contrast with the original constant reflectance model (Fig. 2, top spheres).

Once there was polarized light in the scene, I needed a way to show it. The natural way to do this was with a polarizing filter. Filters, as well as other optical elements, can be modeled with a Mueller matrix: a 4x4 matrix that operates on Stokes vectors (Hecht, 2002, p. 378). Since the recursive ray tracing function returns Stokes vectors, and at the top level (looping through image pixels) these are all oriented with respect to a camera axis, all that was needed was to add a matrix multiply to the loop, prior to discarding the S_1 through S_3 components to form an intensity image (Fig. 3).

A related research problem is that of shape from polarization: reasoning about the shape and

orientation of a specular surface based on polarized photography (Rahmann & Canterakis, 2001). The idea is that, since the reflected component is strongly polarized perpendicular to the plane of incidence at many angles, the angle and degree of polarization reaching the camera can be used to reconstruct the surface normal at each pixel. The difficulty is that the transmitted component is also polarized, albeit more weakly, and perpendicular to the reflected one (Fig. 5, 6). Thus, for transparent surfaces under arbitrary lighting conditions, the polarization data alone is not sufficient to determine the orientation of the surface, even when complete Stokes vectors are available, as in this synthetic imagery. Working around this is an active area of research (Miyazaki et al, 2004), but it means that my "ideal glare removal" filter cannot work as planned. However, it can work if the filter has knowledge of the surface normals: the projection of the normal onto the image plane is used as the axis of a linear polarizer, and much of the reflection is eliminated (Fig. 4).

Finally, I implemented a "synthetic sky" environment map to give a rough model of color and polarization in the sky due to Rayleigh scattering (Fig. 7-10). Polarization is zero at the position of the sun and directly across from it, and maximized 90° from it.

In addition to the polarization features, I made various other improvements to the code. It uses a hand-written parser for reading scene files, to which I added support for comments. The file format consists of just keywords and numbers, so this made it easier to keep track of different materials and to turn things on or off when editing the test scenes. Also, I took advantage of the highly parallel nature of ray tracing to implement multi-threaded rendering: the image is divided into small tiles, which are dynamically assigned to a number of threads as they complete their work. I added *const*-correctness to much of the code, in part to help verify that the threads do not unsafely modify shared data structures. The multi-threading yielded a 5.2x speedup on a 6-core workstation.

Conclusion

In this project I aimed to design a framework for dealing with polarized light in a ray tracer, and use it to implement some physically-based effects. I achieved this goal and was able to model Fresnel

reflectance, polarization aspects of atmospheric scattering, and polarizing camera filters. Future work that could be done with this code includes extending the Fresnel model to allow for polarized incoming light, or to allow non-transparent surfaces, using a combination of Fresnel equations for surface reflection and some other model for body reflection. It could also be useful as a source of images of surfaces with known geometry, in the testing of shape from polarization techniques.

References

Glassner, A.S. (1989). *An Introduction to ray tracing.* San Francisco, CA: Morgan Kauffman.

Hecht, E. (2002). *Optics* (4th ed.). San Francisco, CA: Addison Wesley.

Miyazaki, D., Kagesawa, M., & Ikeuchi, K. (2004). Transparent surface modeling from a pair of polarization images. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 26*(1), 73-82.

Rahmann, S., & Canterakis, N. (2001). Reconstruction of specular surfaces using polarization imaging. *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 1*, I-149 - I-155.

Result images

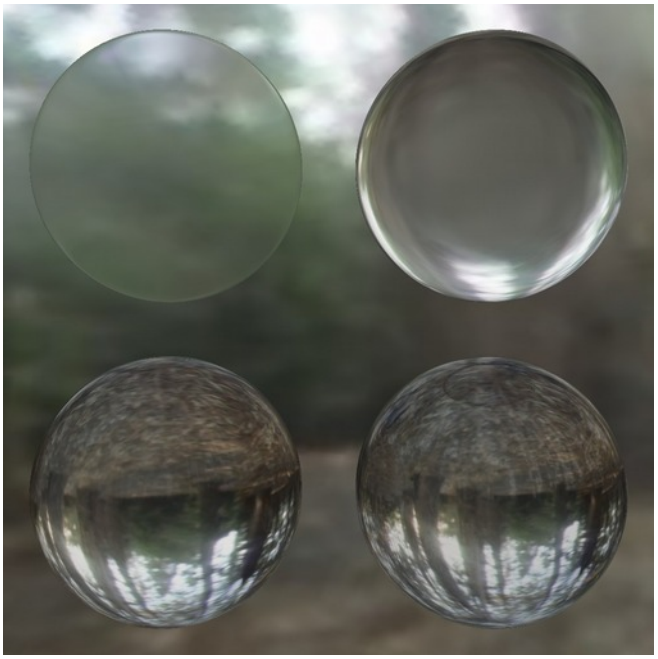

Figure 1. Fresnel reflection for solid spheres of varying index of refraction. As the index increases, the reflected trees become more visible over the refracted (upside down) ones.

Figure 2. Glass spheres under different shading models. Left: constant reflection; right: Fresnel reflection; top: hollow; bottom: solid.

Figure 3. Hollow glass sphere viewed through a vertical linear polarizer (brightened to match original intensity). Reflection is reduced at top and bottom; transmission is reduced at left and right.

Figure 4. What an “ideal” glare filter would look like, removing as much of the reflected component as possible based on polarization and surface orientation.

Figure 5. Hollow sphere with black and white circles for contrast.

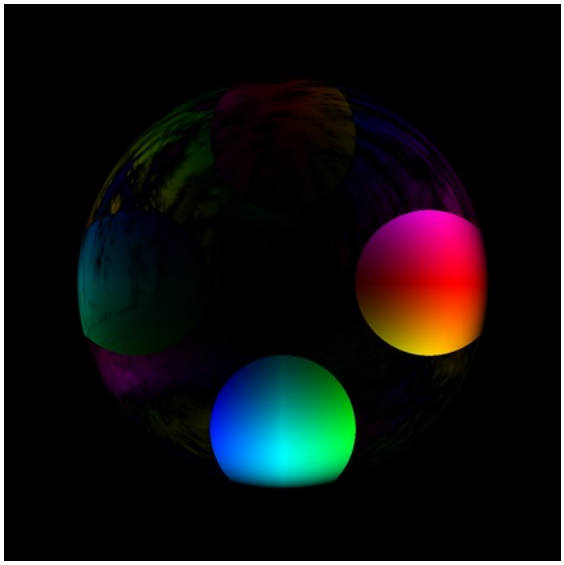

Figure 6. Visualization of polarization in Fig. 5. Hue represents angle. When there is a strong transmitted component (white), polarization is weak and perpendicular to when there is no transmission (black).

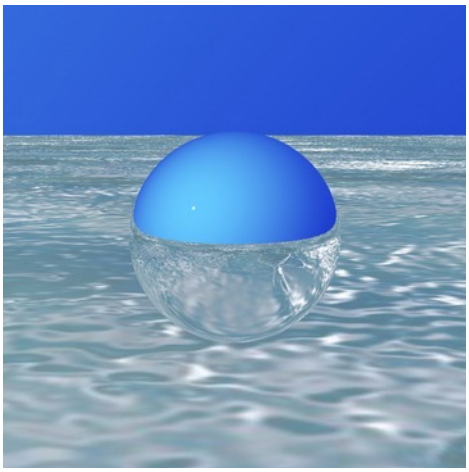

Figure 7. Sky over water, reflected in a sphere so as to see a wide range of angles from the sun. (water texture: http://borysses.deviantart.com/)

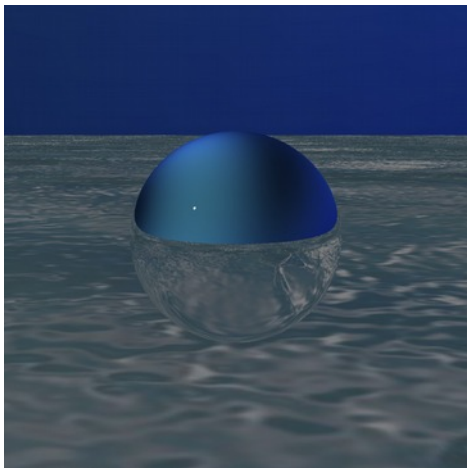

Figure 8. The scene from Fig. 7 seen through a linear polarizer. The sky appears darkest at 90° from the sun because it is the most polarized.

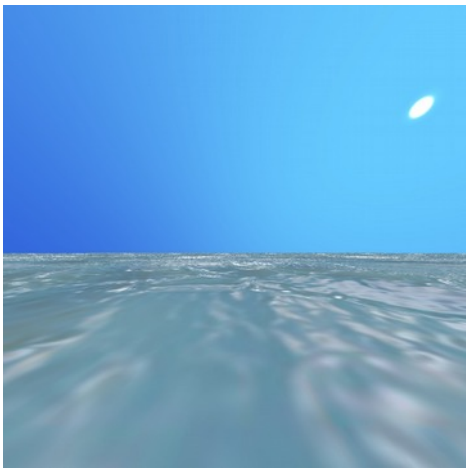

Figure 9. Sky over water (sun appears distorted due to the wide field of view).

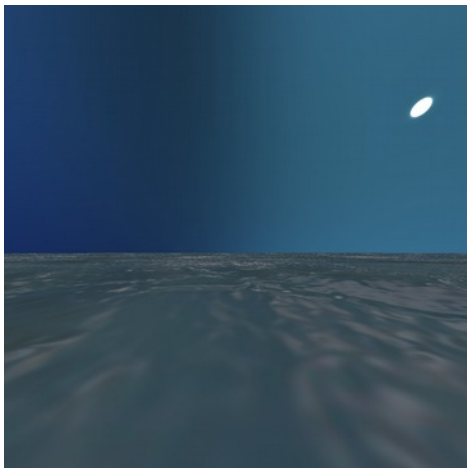

Figure 10. With polarizer.