2011

Ray-traced simulation of radiation pressure for optical lift

Timothy Peterson

Ray-Traced Simulation of Radiation Pressure for Optical Lift

by

Timothy J. Peterson

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of Master of Science
in Computer Science

Supervised by

Associate Professor Dr. Grover Swartzlander
Center for Imaging Science, Department of Physics
Rochester Institute of Technology
Rochester, New York
November 8, 2011

Approved by:

Dr. Joe Geigel, Associate Professor
Thesis Advisor, Department of Computer Science

Dr. Grover Swartzlander, Associate Professor
Reader, Center for Imaging Science, Department of Physics

Dr. Hans-Peter Bischof, Professor
Observer, Department of Computer Science

Abstract

**Ray-Traced Simulation of
Radiation Pressure for Optical Lift**

Timothy J. Peterson

Advisor: Dr. Joe Geigel

When light refracts at a surface, it changes the direction and intensity of the light rays. By Newton's 3rd law, this process imparts a small momentum to the object. This effect can be exploited to manipulate very small objects, by carefully selecting the shape and optical properties of the object. This thesis describes a computational model based on the laws of physics to determine the forces and torques resulting from light interacting with a hemicylindrical particle. This model has been successfully used to predict for the first time a phenomenon called optical lift.

Acknowledgments

I would like to thank Grover Swartzlander for overseeing the optical lift project, and Alexandra Artusio-Glimpse and Alan Raisanen for their work in conducting the physical experiments. I would also like to thank Joe Geigel for serving as my thesis advisor.

Thank you to Sandia National Labs for financial support for this project.

Contents

Abstract . . . . . **ii**

Acknowledgments . . . . . **iii**

1 Introduction 1

2 Background 2

2.1 Optics 2

2.2 Ray tracing 4

3 Analysis 7

4 Requirements 10

5 Design 12

6 Results 18

7 Conclusion 24

Bibliography **27**

A Supplementary media **29**

B User's guide **30**

1 Overview 30

2 Installation 30

3 Configuration 31

4 Usage 39

List of Figures

1 Ray diagrams (ignoring reflections) for a sphere in the single-beam optical trap, showing the net restoring force for displacements (a) toward and (b) away from the light source. Green arrows show forces on the object due to refraction of light rays, and blue arrows show the net force. 3

2 A simple ray tracing scenario, with a camera, a light source, and three objects. Point A is illuminated, due to the unobstructed shadow ray to the light source. Point B is in not illuminated, because the shadow ray hits an opaque object before it reaches the light source. Point C's illumination cannot be determined by ray tracing due to the refracting sphere nearby that modifies the light before it reaches the plane. 5

3 Ray intersection with reflected and refracted rays, showing the angles used for Snell's Law. 5

4 Photon mapping applied to the ray tracing scenario in Figure 2. Light rays are traced from the light source through the refracting object, and the illumination is recorded where each ray hits the plane. 7

5 The three constructive solid geometry operators available for constructing objects: (a) union, (b) intersection, and (c) difference. 10

6 Two types of light sources, (a) collimated and (b) focused. 11

7 Example trace of a single light ray passing through a transparent refracting object. 13

8 The overall design of the system. Python components were created for this simulation, while the C components were pre-existing (although POV-Ray was modified). 14

9 Starting locations of parallel rays fired from a planar light source in refinement levels 0 through 3. The pattern is repeated for all unit squares on the light source, until the object has been completely covered. 16

10 Cross-section of glass hemicylinder rod in water at 18 degrees from horizontal. Light rays are incident from the left and are colored red (high intensity) through gray (low intensity). Momenta applied to the object by each ray in the Minkowski model are shown as green vectors. The arrow from the center indicates the net momentum applied to the object by the light, and the horizontal arrow suggests the magnitude and direction of the net torque. 19
11 Glass hemicylinder in water rotated into two stable orientations, (a) 13.5° and (b) 32.9° from horizontal. 20
12 Torque on hemicylinder with respect to rotation angle, with stable orientations marked. Orientations range from -90° (flat face away from light) to $+90^\circ$ (flat face toward light). 20
13 Glass hemicylinder moving freely in water in a left-to-right light beam. The path of the center of mass is traced, with orientation shown at equal time intervals. 21
14 Torque on hemicylinder vs. angle of attack, under varying refinement levels. 22
15 Path of center of mass of the hemicylinder under five different time steps. . 23
16 (a) Experimental setup for observing optical lift. (b) Time-lapse composite image showing the particle's motion from left to right due to optical lift. . . 24
17 Ray diagram for optical tweezers with spherical particle off-center. Focused light is incident from the bottom and focused at the origin. 25
18 Optical tweezers force map displayed with (a) vector field and (b) color-mapped magnitude (both with magnitude on a logarithmic scale). Light enters the system from the bottom and is focused at the origin, with the trapping point slightly above. 25

1 Introduction

There is ongoing research in the world of optics that involves using light to manipulate microscopic objects without making physical contact. This is made possible by the fact that light has momentum, albeit a very small amount per photon. The effects of this momentum are rarely visible in human-scale objects on Earth, but very small or lightweight objects can be affected significantly. For instance, the solar sail is a device used in space to harness the radiation pressure from sunlight [15]. A large expanse of a thin, reflective film deployed from a spacecraft can provide a small amount of propulsion and steering, owing to momentum transferred from the reflected sunlight.

In addition to absorption and reflection, light's momentum can also be harnessed using refraction through transparent objects. When light passes through a surface into a material of different refractive index, it changes direction and hence imparts momentum. One current application of this effect is known as optical tweezers, where a focused beam of light can trap a transparent particle at the focal point, allowing it to be moved around without physical contact [4].

A new concept currently being explored is called optical lift [14]. In this scenario, light incident upon an object would produce a sustained force perpendicular to the direction of the light, reminiscent of aerodynamic lift in which horizontal motion of a wing through air results in a vertical lift force. While there may be many ways to implement this, the approach we investigate here is to use microscopic transparent particles suspended in water. The particle shape we will initially investigate is the hemicylinder, *i.e.* a cylinder sliced in half lengthwise. While this shape does resemble an aerodynamic wing, the mechanism producing lift is completely different.

This thesis will describe a simulation of the motion-inducing effects of light upon these hemicylindrical particles, which was followed by a physical experiment conducted by other members of the team that demonstrated the validity of the simulations. Additionally, the simulation was tested in an optical tweezers scenario using focused light, although this is not the primary goal.

2 Background

2.1 Optics

Radiation pressure is very small, easily overwhelmed by the motion of surrounding gas molecules. It was with some difficulty that experiments were devised to isolate and measure radiation pressure, which eventually confirmed Maxwell's equation predicting the force [11]. Forces due to reflection of light are easy to compute based on the momentum of the incoming light, but the momentum of light inside a refracting medium is a controversial matter.

Abraham [1] proposed that the momentum of a photon is p_0/n, while Minkowski [10] proposed $p_0 n$, where p_0 is the momentum of a photon in free space and n is the refractive index of the medium. On the surface, these two expressions seem irreconcilable. Burt and Peierls [6] explain the dilemma and give a thought experiment that concurs with Abraham. However, it has been recently proposed that the two are in fact equivalent [5], being expressions of different types of momenta. Since this issue has not yet been resolved, the simulation constructed for this thesis will permit both models to be tested.

While optical lift is a new concept, optical trapping has been investigated since 1970. Ashkin [3] gives a history of optical trapping through 2000. Initial experiments by Ashkin found that micrometer-sized particles could be pushed through a fluid along a laser beam. In addition, particles in the low-intensity fringes of the beam are pulled toward the brightest area of the beam. These forces are known as the scattering force, parallel to the beam, and the gradient force, toward the highest intensity area of the beam.

The first optical trap consisted of two focused laser beams pointed in opposite directions toward each other, focused on two closely-spaced points. Particles are trapped in the area between the foci, since the scattering forces push the particles away from the foci, and the gradient force keeps them along the beam axis. The viscosity of the fluid damps the motion of the particles upon arrival within the trap, so they settle down at the trapping point rather than oscillating.

Later refinements of the technique allow the use of a trap made up of a single focused laser [4]. In this case, the gradient force pushes the particle toward the focus from all

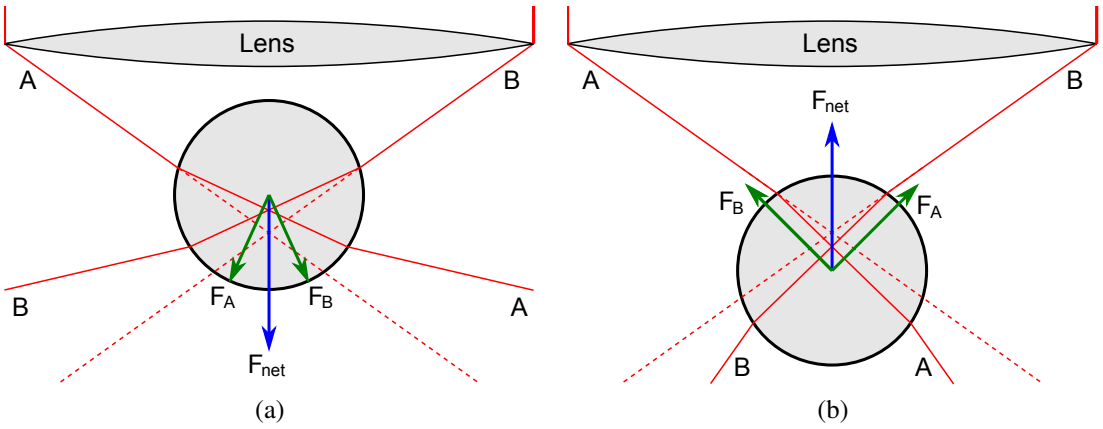


Figure 1: Ray diagrams (ignoring reflections) for a sphere in the single-beam optical trap, showing the net restoring force for displacements (a) toward and (b) away from the light source. Green arrows show forces on the object due to refraction of light rays, and blue arrows show the net force.

directions. If the focusing angle is large enough (up to 70° off axis in some cases), the gradient force can be larger than the scattering force even when the particle is displaced beyond the focus, resulting in a net force back toward the focus. This is illustrated in Figure 1.

Optical traps can be used for atoms and small inanimate particles, as well as live bacteria and other cells. This can allow manipulation of these organisms more easily than with any mechanical means. In fact, internal surgery can be performed on individual cells, by using optical tweezers to drag organelles and other internal components around within the cell. Additionally, laser pulses can be used to make cuts in the walls of cells, in order to extract or implant items. Current research includes work on shaping light beams in more exotic ways in order to better control the trapped objects, *e.g.* angular momentum, non-diffracting beams, and computer-driven holography [7].

Ashkin [2] uses a ray optics model to analyze the forces exerted by optical tweezers on a perfectly refracting sphere. He accounts for rays reflecting indefinitely inside the sphere, and the resulting scattered rays exiting the sphere. The net forces are numerically integrated with the help of a computer, to plot the gradient and scattering forces that create the optical trap. While many particles used with optical trapping, such as bacteria, are not spherical, the same principles apply.

Tweezers can be used to grab single particles or gather clusters of particles and move them around as desired. Multiple tweezers can be used in concert to bring components together for assembly. Also, some aspherical particles such as bacteria can be rotated by using two tweezers to pull on opposite ends. However, manipulating n particles independently requires n instances of the tweezer apparatus.

Optical lift provides a different method of manipulating particles, which can operate on many particles simultaneously. The difference is that while optical tweezers shapes the light beam and uses ordinary-shaped particles, optical lift takes the converse approach and shapes the particles while using an ordinary (unfocused) light beam. The techniques and applications of optical lift are still topics to be explored.

2.2 Ray tracing

Light can act like either particles or waves, depending on the circumstances. The wave nature of light comes into play when dealing with very small features of objects, but the objects used for optical lift are large enough (micrometers in size) to be simulated correctly using only the particle nature of light. The standard method of simulating photons is through the well-known computer graphics technique known as ray tracing [16].

In the real world, light is emitted from light sources, which then interacts with objects in the scene, and some of it enters either an eye or a camera. It would be impractical to simulate every photon emitted from the light source, since very few of them reach the camera. Ray tracing alleviates this inefficiency by reversing the process. For each pixel desired in the output image, we assume that a ray of light landed on that pixel in the camera from the appropriate direction, and find out where it must have come from. This is known as reverse ray tracing, since the rays are traced backward from the actual direction of the light.

Given a camera at a certain location with a defined image plane, we iterate over all pixels on that image plane and determine the direction of the light making up that pixel. This is used to create a ray equation corresponding to the light that entered that pixel. All of the objects in the scene are tested for intersection with that ray, and the object intersection

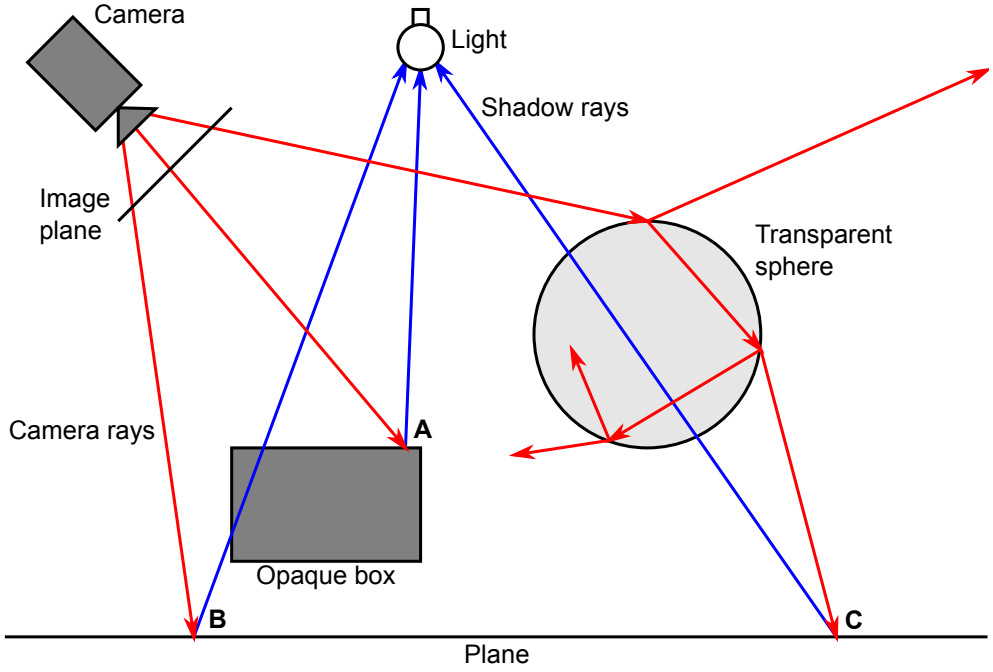


Figure 2: A simple ray tracing scenario, with a camera, a light source, and three objects. Point A is illuminated, due to the unobstructed shadow ray to the light source. Point B is in not illuminated, because the shadow ray hits an opaque object before it reaches the light source. Point C's illumination cannot be determined by ray tracing due to the refracting sphere nearby that modifies the light before it reaches the plane.

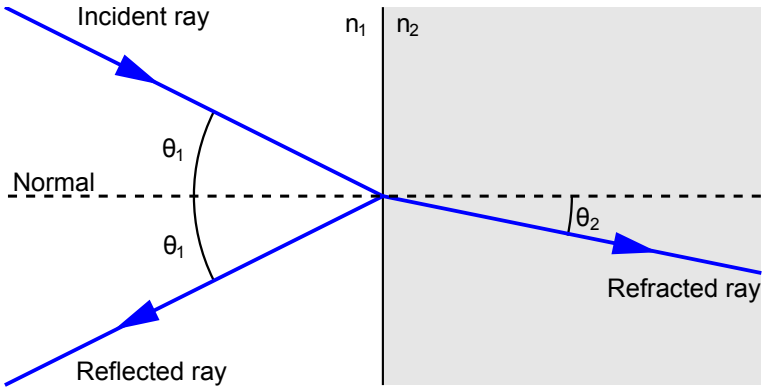


Figure 3: Ray intersection with reflected and refracted rays, showing the angles used for Snell's Law.

found closest to the ray's origin is taken. If the object is opaque, then all light for that pixel must have come from light illuminating that point on the object. This is determined by firing shadow rays from that point toward all of the light sources, to determine whether the point is illuminated or in shadow. However, if the object is transparent or reflective, one or more new rays must be traced recursively from the intersection point in order to determine the final color of that point, as shown in Figure 3. For refractive objects, Snell's law is used to determine the direction of the transmitted ray. Snell's law states that

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1} \tag{1}$$

where θ_1 is the angle of incidence from the normal, θ_2 is the angle of the refracted ray, and n_1 and n_2 are the indices of refraction of the materials containing the incident and refracted rays, respectively. Fresnel's equation is then used to determine the intensity weighting between the reflected and refracted rays. R_s is the fraction of light reflected at the interface for s-polarization (electric field perpendicular to the plane of incidence), while R_p is the same for p-polarization. The remainder T of the light is transmitted ($R + T = 1$).

$$R_\mathrm{s} = \left(\frac{n_1\cos\theta_1 - n_2\cos\theta_2}{n_1\cos\theta_1 + n_2\cos\theta_2}\right)^2 \tag{2}$$

$$R_\mathrm{p} = \left(\frac{n_1\cos\theta_2 - n_2\cos\theta_1}{n_1\cos\theta_2 + n_2\cos\theta_1}\right)^2 \tag{3}$$

Standard (camera-based) ray tracing correctly renders transparent and reflective objects, but cannot handle caustics, where light from a light source is modified or redirected by a transparent or reflective object before illuminating a surface. Photon mapping solves this problem by using forward (light-based) ray tracing to determine the illumination of the scene, followed by camera-based ray tracing to generate an image based on that illumination. [9] The photon mapping pass begins at each light source, firing packets of energy as rays that uniformly sample every object in the scene that could produce caustics, as in Figure 4. The recursive ray tracing algorithm follows each packet of light through as many bounces as necessary until an opaque object is reached that absorbs the light. Every packet

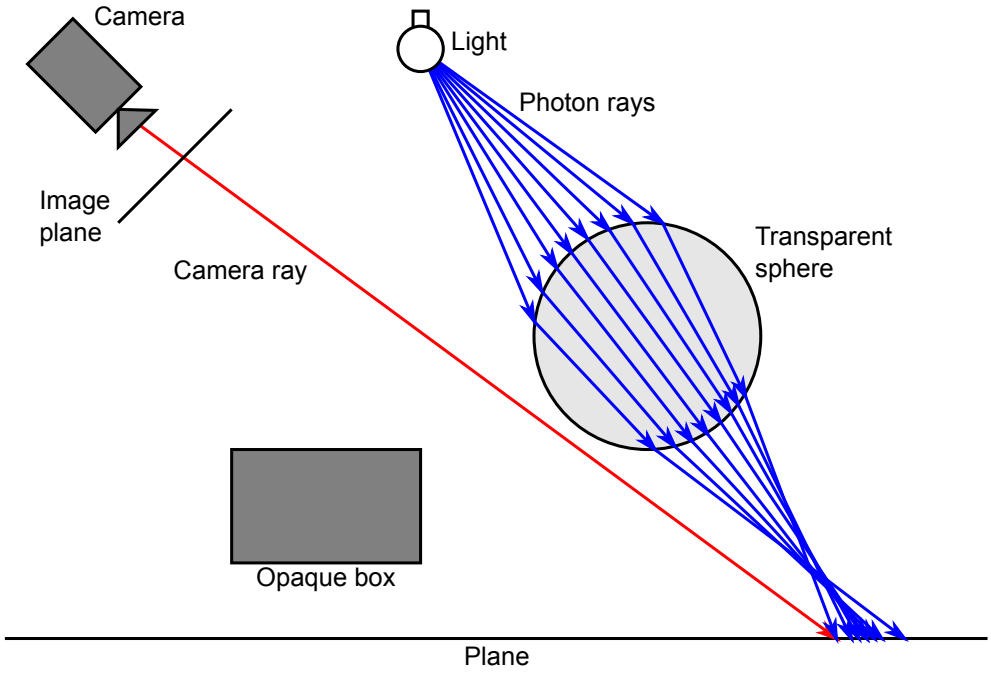


Figure 4: Photon mapping applied to the ray tracing scenario in Figure 2. Light rays are traced from the light source through the refracting object, and the illumination is recorded where each ray hits the plane.

of light with its intensity and direction is recorded at the point on the surface that it illuminates. When the camera-based ray tracing pass needs to determine illumination at a point, it examines the photon map to account for indirect illumination, in addition to testing for direct illumination.

3 Analysis

The photon mapping component of ray tracing provides the inspiration for a solution to our optical lift problem. Here we require just the light-based photon mapping pass, and not the camera-based image rendering pass. Instead of tracing photons to surfaces and recording only where they land, we wish to track their movements through the object, and for every surface that changes the photon's momentum we apply an equal and opposite momentum to that point on the object.

At each ray-object intersection, the packet of light represented by the ray experiences a change in momentum. We compute this by taking the total momentum of the one or

two outgoing rays and subtracting the momentum of the incoming ray. For Abraham's and Minkowski's formulations, respectively, the momentum of a single ray is given by

$$\vec{p}_A = \frac{U}{c} n \vec{d} \tag{4}$$

$$\vec{p}_M = \frac{U}{cn} \vec{d} \tag{5}$$

where U is the energy of the ray, c is the speed of light, n is the index of refraction, and $\vec{d}$ is the direction of the ray. For a ray intersection that splits the ray into reflected and transmitted components by fractions R and T (where $R+T=1$), the change in momentum is given by:

$$\Delta\vec{p}_A = \frac{U}{c}\left(R\frac{\vec{d}_R}{n_1} + T\frac{\vec{d}_T}{n_2} - \frac{\vec{d}}{n_1}\right) \tag{6}$$

$$\Delta\vec{p}_M = \frac{U}{c}\left(Rn_1\vec{d}_R + Tn_2\vec{d}_T - n_1\vec{d}\right) \tag{7}$$

For the purposes of this simulation, we are not concerned with the exact magnitude of the total momentum, only with the overall properties of the object in the light beam. As a result, we can replace the scale factors from the energy and speed of light $\frac{U}{c}$ with an arbitrary intensity I. $\Delta\vec{p}$ is the change in momentum of the light, so to simulate the object's motion we apply an equal and opposite momentum of $-\Delta\vec{p}$ to the object at every intersection point.

Given all of the individual ray impacts on the surface, the next step is to simulate the rigid body motion of the object. In the physical experiment, the object is very small and is suspended in water. We consider this system to be highly damped, in that the object's velocity will drop to zero soon after an impact, owing to the viscosity of the medium compared to the small mass of the object.

Conservation of momentum states that given initial and final velocities $\vec{v}_i$ and $\vec{v}_f$, we have $m\vec{v}_f - m\vec{v}_i = \Delta\vec{p}$. If the velocity drops to zero soon after an impact, then prior to each time step in the simulation we have $\vec{v}_i = 0$. Therefore, $\vec{v}_f = \Delta\vec{p}/m$. For simplicity, this velocity is then applied as a constant over the whole time step, so $\Delta x = \vec{v}_f \Delta t$. For an accurate simulation we need $\vec{v}_f \Delta t$ to be small, so the time step can be reduced if the results

are too coarse.

The same rationale applies in computing rotation, where we again consider angular velocity to be heavily damped. Each ray intersection i has a momentum $\Delta\vec{p_i}$ and a lever arm $\vec{r_i}$ relative to the object's center of mass. We take $L = \sum \vec{r_i} \times \Delta\vec{p_i}$ to obtain the net torque L, and convert it to a total rotation at the end of the time step via $\Delta\theta = \frac{L}{I}\Delta t$. The moment of inertia I must be computed according to the shape of the object.

The simulation process can be summarized as follows:

$\vec{p}_{net} \leftarrow 0$

$\vec{\tau}_{net} \leftarrow 0$

for each ray i with children r and/or t **do**

{Compute the momentum of the incoming and outgoing rays independently}

$\vec{p_i} \leftarrow I_i \cdot \vec{d_i} \cdot n_i$

$\vec{p_r} \leftarrow I_r \cdot \vec{d_r} \cdot n_r$

$\vec{p_t} \leftarrow I_t \cdot \vec{d_t} \cdot n_t$

{Compute the net momentum change on the object}

$\Delta\vec{p_i} \leftarrow \vec{p_i} - (\vec{p_r} + \vec{p_t})$

{r_i is the vector from the center of mass to the point of intersection}

$\vec{\tau_i} \leftarrow \vec{r_i} \times \Delta\vec{p_i}$

{Sum the momentum change on the object}

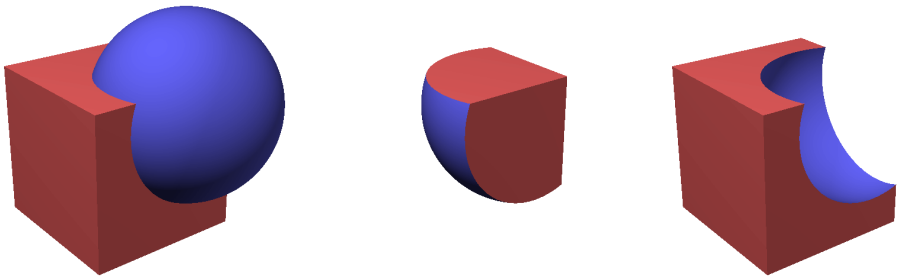
$\vec{p}_{net} \leftarrow \vec{p}_{net} + \Delta\vec{p_i}$

{Sum the torques on the object}

$\vec{\tau}_{net} \leftarrow \vec{\tau}_{net} + \vec{\tau_i}$

end for

The simulation proceeds through time by repeating this process for each time step. Since we are assuming a heavily damped system, no velocity is carried forward between time steps, and each step stands on its own based upon the initial position and orientation of the object.



(a) (b) (c)

Figure 5: The three constructive solid geometry operators available for constructing objects: (a) union, (b) intersection, and (c) difference.

4 Requirements

This section will describe the requirements of the software that will carry out the simulations as described.

The first step in running a simulation is to create the scene. There will be one or more objects to be simulated in the scene. These objects will be composed of geometric primitives that can be described mathematically, so that the ray tracer can work with them easily. Some necessary primitives include the sphere, cylinder, and prism (forming a box, for instance). Additionally, these primitives can be combined via Boolean operations, known as constructive solid geometry, as shown in Figure 5. For instance, a hemicylinder can be created by taking the intersection of a cylinder with a plane (or a box).

Every object in the scene has an index of refraction (*e.g.* 1.5 for glass), as does the medium in which the objects are suspended (1.33 for water). Also, it can be useful to add inclusions to the objects, *i.e.* an area of differing refractive index within the object. For instance, a particle of glass could have a bubble of air within it. This can be done by taking the union of the particle and the inclusion.

The object to be simulated will start at some position in space, typically the origin, while a light source will illuminate it from another position in space. There will be two types of lighting available, shown in Figure 6: collimated and focused. Collimated light in this simulation will be emitted from a plane with a given orientation in space, typically along the z axis. Focused light will be focused on the origin around the z axis, with a maximum

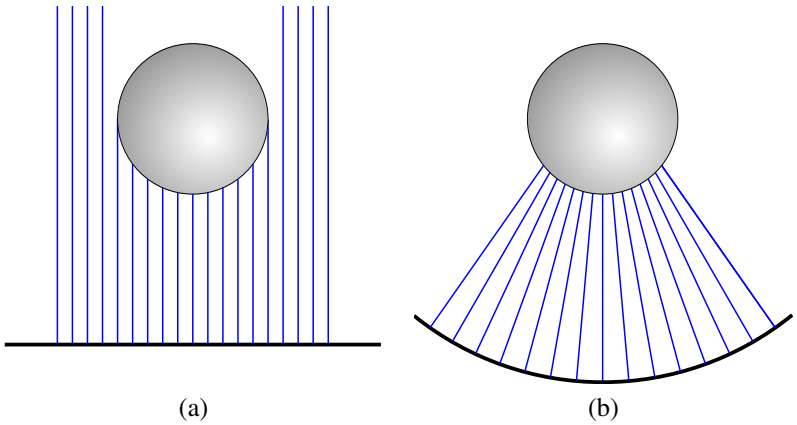


Figure 6: Two types of light sources, (a) collimated and (b) focused.

angle from the axis determined by the given numerical aperture of the light source.

Given a scene definition, simulation begins by tracing light rays starting from the light source and passing through the object. The ray tracer will have configuration parameters that determine how dense the rays will be, to allow the user to balance between accuracy and CPU time required. Each ray will carry a packet of energy with an intensity corresponding inversely to the density of the rays, such that the total energy entering the system is independent of the ray density. Another configuration parameter will determine to what recursion depth the rays will be traced, so that rays reflecting inside an object do not continue bouncing indefinitely.

Once the rays have been traced, the momentum change will be computed for each ray, as well as the total momentum and torque on the object. There will be two modes of simulation. In the first mode, the light's effect on the object will be computed independently for a set of initial conditions, for instance different positions or orientations relative to the light source. In the second mode, the object will be moved between frames according to the laws of physics, so that the object's progression over time can be seen.

The simplest form of output desired is a tabular text file indicating such things as the object's position and orientation at each frame in the simulation. For testing and verification purposes, it will also be possible to obtain a human-readable listing of each ray that was traced, including the start and end points, the intensity, and the index of refraction.

An important part of the system will be graphical output. First, for each frame a 2D or 3D plot can be generated showing the object and some or all of the rays interacting with it. Also shown will be vectors indicating the momentum contribution by each ray interaction with the object, as well as the net momentum and torque. These plots can be built into a video file showing the object's motion in animated form.

In addition to per-frame plots, there will also be graphs showing the object's behavior over time. For instance, one might graph the net torque versus the angle of rotation of the object, to find stable orientations in the optical lift scenario. Another graphing application could be a vector field plot showing the force on the object at various positions relative to a focused light beam for optical tweezers.

5 Design

POV-Ray [13] is a well-known, free, open-source software package for creating photo-realistic images using ray tracing, and as such it has implemented Snell's and Fresnel's equations for physically accurate handling of transparent objects. Its primary technique for rendering images is through reverse ray tracing (camera to objects), but it can also do forward ray tracing (light source to objects), known as photon mapping. This photon mapping process is what we want to harness for this project. The results of this process are normally used to compute lighting in the scene, but here we want to do completely different processing, so the source code has been modified in order to store a log of every ray tracing step made during the photon mapping process. The binary record format used to store this data is shown in Table 1. Each ray is described by one such record, and the ray trace file consists solely of these records, with no header data. Figure 7 shows an example set of rays traversing an object. Those rays could be described as follows:

- Ray #1 with intensity 1.00 went from point A on the light source to point B on the object. It spawned ray #2 by refraction and ray #3 by reflection.
- Ray #2 with intensity 0.96 went from point B on the surface of the object to another point C on the surface. It then spawned ray #4 by total internal reflection.

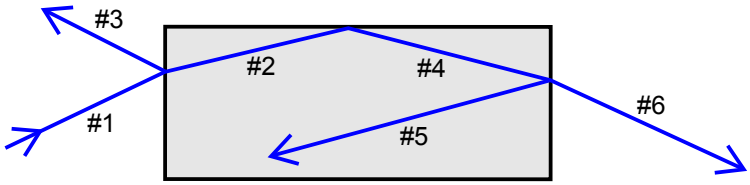


Figure 7: Example trace of a single light ray passing through a transparent refracting object.

Field	Type	Bytes	Notes
ID	int	8	
Parent ID	int	8	
Start	float	3×8	Light source or end of parent ray
Direction	float	3×8	(end − start) normalized
Color	float	3×4	In RGB color space
IOR	float	4	Index of refraction
End	float	3×8	Undefined if ray hit no objects
Normal	float	3×8	Zero if ray hit no objects

Table 1: Binary file format used to convey ray trace data between customized POV-Ray and Python software. Data type sizes are architecture-dependent; these examples were taken from a 32-bit x86 machine.

- Ray #3 with intensity 0.04 didn't hit anything, and went off to infinity in some direction.
- Ray #4 with intensity 0.96 went from point C on one side of the object to point D on another side. It spawned ray #5 by refraction and ray #6 by reflection.
- *(many more rays follow)*

Apart from POV-Ray, which is written in C, the remainder of the system is written in Python, a well-known scripting language. One advantage of this type of language is that in the absence of a GUI the user can edit simulation scenarios as Python source code, which can then be easily ingested by the Python interpreter to configure the system. Two major third-party libraries were used. First, NumPy provides facilities for doing computations on arrays of numbers more efficiently than in pure Python code, mitigating some of the speed difference between C and Python [12]. Second, Matplotlib is a graphing library that produces high-quality plots, both on-screen and written to disk [8]. Figure 8 shows how these components fit together.

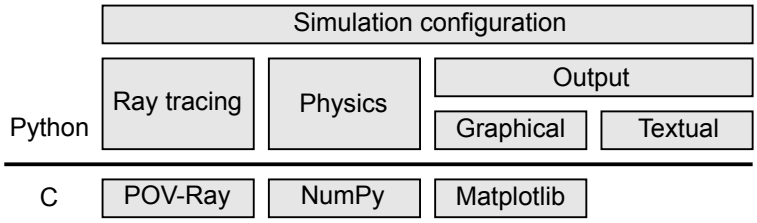


Figure 8: The overall design of the system. Python components were created for this simulation, while the C components were pre-existing (although POV-Ray was modified).

There are three primary steps in executing a simulation: configuration, ray tracing, and output generation. The configuration step is initiated by the user creating a Python source file that describes all aspects of the simulation, as follows.

The object under simulation has a shape and a composition. Under the limitations of this system, the object is perfectly smooth and transparent, and so the composition is defined simply by the index of refraction of the material. The shape is defined in terms of the geometric primitives that POV-Ray supports, including spheres, cylinders, boxes, and more complex mathematical shapes. These primitives are augmented by the constructive solid geometry operators, union, intersection, and difference, which permit multiple primitives to be combined using Boolean operators. This construction style seems to be sufficient for current experiments with shaping objects.

The second aspect of the scene configuration is the light source. POV-Ray supports two types of light sources. Point sources emit light uniformly in all directions from a single point in space. Parallel light sources emit light from all points on an infinite plane, directed along the normal. For testing the concept of optical lift, we are not concerned with effects at the edges of the collimated light beam, so such an unbounded light source is acceptable. POV-Ray targets specific objects with light during its photon mapping process and stops when the object has been completely covered, so the infinite nature of the light source is not a computational concern. Additionally, in order to support the optical tweezers scenario we require focused light. Since POV-Ray does not natively support focused light sources, this will be done by using a parallel light source facing away from the object, reflecting off a paraboloid surface constructed so as to focus light on the origin. POV-Ray is able to do

this through a primitive capable of representing any quadric surface.

For computational expedience, we should trace enough rays to produce accurate results, but not past a point of diminishing returns. We control this using two parameters: the trace depth and ray density. The trace depth is the number of ray intersections to follow before giving up on a particular ray. This is needed because otherwise light rays could bounce around inside the object indefinitely. At each bounce these rays would typically spawn a transmitted ray exiting the object, but also another reflecting ray inside. After a number of bounces the intensity contribution becomes insignificant, but POV-Ray is capable of tracing to significant depths if desired.

The ray tracing density controls how many rays are emitted from the light source per unit area. The greater the density, the more accurate the simulation, because more of the object's surface is covered. This parameter is particularly important in objects that have small features (including sharp edges), so the density must be set high enough to accurately sample these areas. One significant example of this is when the flat side of a hemicylinder is close to parallel with the incoming rays. The exposed area as seen by the light source is small, but rays reflected at a shallow angle can create large momentum transfers.

There are two factors affecting the placement of sampling rays. First, as mentioned, the ray density must be sufficient for an accurate simulation. Second, an important part of the graphical output of the system is the plotting of rays traversing the object, which typically involves displaying a cross-section of the 3D simulation space, and an evenly-spaced set of rays is useful to plot. While the simulation requires thousands of rays to be sufficiently accurate, the plotted output requires few rays, perhaps 10-20, in order to be easily understood.

These two issues are resolved here by tracing rays in groups called refinement levels, demonstrated in Figure 9. Each refinement level n defines an evenly-spaced square grid of ray starting points with spacing 2^{-n} on the light source plane. Refinement level 0 fires rays with a spacing of $2^0 = 1$ unit. Refinement level 1 fires rays with a spacing of $2^{-1} = 1/2$ unit, which is a superset of level 0 and contains four times as many rays. Likewise, level 2 is spaced at $2^{-2} = 1/4$ units, again with four times as many rays as the previous. Each ray

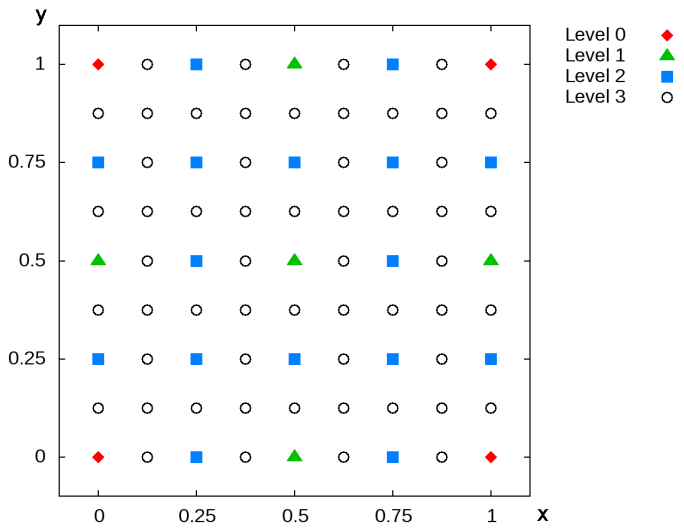


Figure 9: Starting locations of parallel rays fired from a planar light source in refinement levels 0 through 3. The pattern is repeated for all unit squares on the light source, until the object has been completely covered.

carries the number of the lowest refinement level it belongs to, so that a small subset of rays can be easily extracted for plotting regardless of the maximum refinement level generated by the ray tracer. Also, this paves the way for a future version of the software to support progressive refinement, where an additional refinement level can be ray-traced on demand to augment an existing data set.

The stock POV-Ray code samples objects in a circular pattern. It starts by firing rays toward the center of the object, and covers concentric circles until all of the rays in one circular pass miss the object, indicating that the object has been completely covered. This creates spacing between rays that is mostly uniform yet not on a grid, which makes it unsuitable for the refinement level concept. To solve this problem, POV-Ray was modified to fire rays on a rectangular grid of a given spacing rather than a circular pattern, so as to be compatible with the concept of refinement levels. It retains the strategy of ray-tracing concentric circles by using concentric squares on the grid instead.

The simulation will normally proceed in discrete time steps, and consequently will require a clock that takes the simulation from some start time to some end time, ticking by a given time step. Based on the progress of the simulation, the user will need to adjust these

parameters to obtain a balance between accuracy and speed. In addition to this time-based mode, it is also possible to ignore the clock and instead manipulate the object's location parameters in a systematic fashion to analyze its instantaneous behavior. When simulating optical lift it is instructive to determine the momentum and torque of the object across a range of angles of attack relative to the light source, in order to find stable orientations.

For each simulation frame, the Python code converts the configured scene (object and light source) into POV-Ray's scene description language. POV-Ray is then called to process the scene, and the custom modifications made to POV-Ray will log every ray tracing operation into a file. Upon completion, Python code reads this file and computes the momentum of each ray and the momentum change at each ray-object intersection. All of the individual momenta are combined to determine the overall momentum and torque experienced by the object.

Output is generated through any number of configurable views, which are ways of representing the state of the simulation. Views can operate in two modes, either per-frame or summary. Per-frame views generate one file per simulation frame, containing output pertaining to that frame, while summary views aggregate the data from all of the frames, for example to graph torque on the object over time.

There are two primary types of output: text and images. The textual output is in a tab-separated text format. The columns of values emitted are configurable and can come from any aspect of the simulation that changes over time, for example object position and orientation, momentum and torque, ray count, etc.

Image outputs are generally plots in either 2D or 3D. The content of each plot is determined by which plotting components the user enables. For instance, one component plots the path of each ray that was traced, and another component plots the momentum imparted on the object by each ray. A third component plots the shape of the object itself. Another set of components is available to plot summary data, for instance extracting the torque on the object in each frame and plotting it versus object rotation angle.

All of these plotting components natively plot data in three dimensions. For 3D plots, this data can be plotted directly by projecting into 2D. For 2D plots, the user must select

the desired axes, *i.e.* xy, yz, or zx. The third dimension is then removed from the data (flattened) for plotting. The user can also configure the data range for the plot, or allow the range to automatically fit the data.

Since ray processing is a core feature of the system, each plot can be configured to filter the set of rays it will plot, prior to processing by the plotting components. The primary method is by refinement level, where a lower-density subset of rays is displayed in lieu of the entire high-density collection. Rays can also be filtered by depth, for instance to display only rays up through the first or second surface interaction. Finally, the rays can be sliced to display only the rays that originated in the plane being plotted, so an xy plot shows only those rays that began at $z = 0$ on the light source.

Once a frame has been computed and outputted, the object is moved and rotated according to the momentum and torque, and the loop repeats until all frames have been computed. At the end, any summary views are also finalized and rendered, resulting in one output file each for the whole simulation.

6 Results

To determine the validity of the output data from the software, we ran simple simulations of rays interacting with a slab of glass, and manually verified the angles, intensities, and momenta that resulted. Additionally, we created an external validation program that processes the ray trace file and checks that Snell's law and Fresnel's equations have been applied properly using an independent implementation.

Figure 10 shows a cross-section of a glass hemicylindrical rod in water, which is the primary focus of optical lift experimentation. We begin with an analysis of the 2D behavior of a semicircular cross-section. The rays emitted from the light source are parallel as they impact the hemicylinder. In this orientation, all of the incident rays first hit the curved side of the object, throwing off weak reflected rays as well as transferring momentum to the object, indicated by the green vectors. The transmitted rays, still containing most of the beam power, hit the other side of the object, either the flat side or the curved side once more. The flat side is at such an angle that the rays experience total internal reflection, which

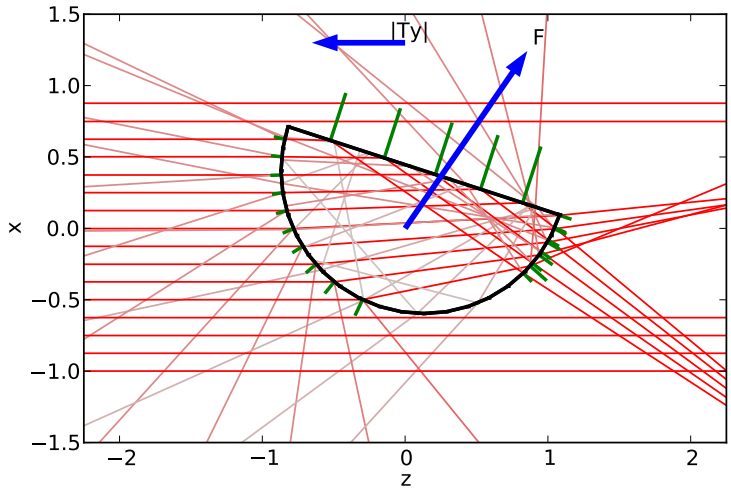
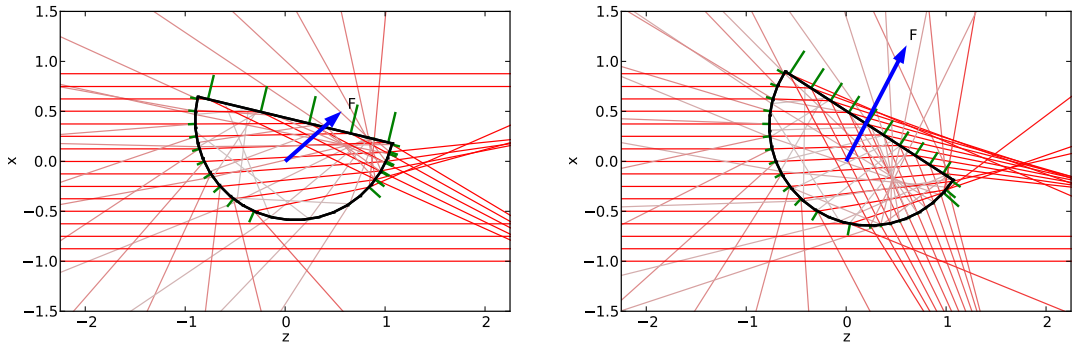


Figure 10: Cross-section of glass hemicylinder rod in water at 18 degrees from horizontal. Light rays are incident from the left and are colored red (high intensity) through gray (low intensity). Momenta applied to the object by each ray in the Minkowski model are shown as green vectors. The arrow from the center indicates the net momentum applied to the object by the light, and the horizontal arrow suggests the magnitude and direction of the net torque.

creates a significantly larger momentum kick than the transmitted rays. The primary beam then exits the object as two slightly-focused beams in different directions. Also, every ray exiting the object produces an additional reflected ray inside, yielding many further low-intensity rays both inside and outside the object. The rays will continue bouncing around inside the object until the trace depth limit is reached.

In this orientation, the net momentum vector is at a significant angle from the direction of the incoming light, which indicates a significant amount of lift. However, there is a non-zero torque, meaning that the object will rotate (counter-clockwise in this case) if left to its own devices. Figure 11(a) shows the orientation toward which it will rotate. At this angle, the torque is zero and stable, because any rotation out of this orientation will result in a torque back toward the stable angle. Compared to the previous orientation, the net momentum is now smaller and at a shallower angle, but will remain constant indefinitely, providing stable optical lift. From other initial orientations, the object will stabilize in position (b).

These stable equilibrium positions were determined using Figure 12, which shows how



(a) (b)

Figure 11: Glass hemicylinder in water rotated into two stable orientations, (a) $13.5°$ and (b) $32.9°$ from horizontal.

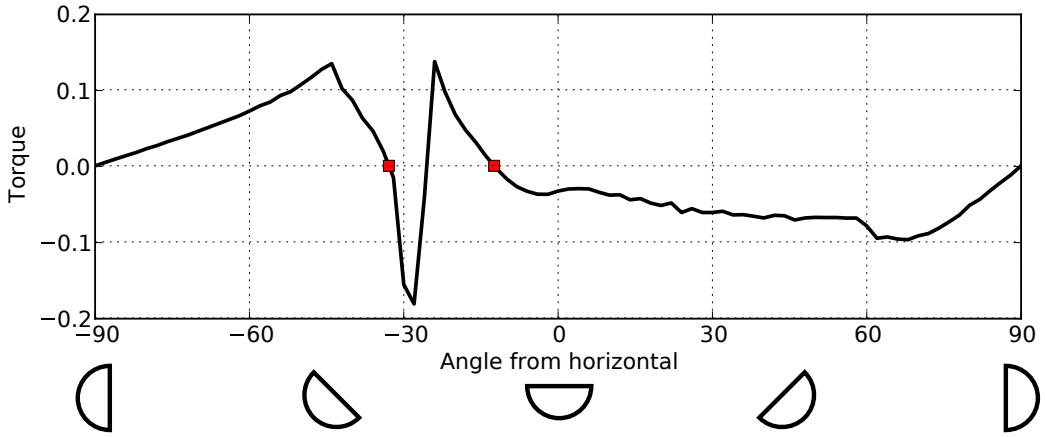


Figure 12: Torque on hemicylinder with respect to rotation angle, with stable orientations marked. Orientations range from $-90°$ (flat face away from light) to $+90°$ (flat face toward light).

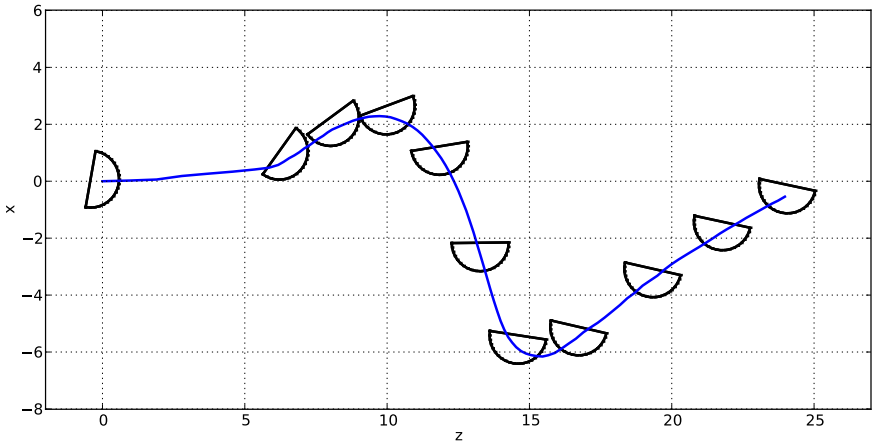


Figure 13: Glass hemicylinder moving freely in water in a left-to-right light beam. The path of the center of mass is traced, with orientation shown at equal time intervals.

the torque on the object varies as a function of the angle of attack. Since the semicircle has bilateral symmetry, we can plot through one half turn of rotation to see the complete behavior. We find that the hemicylinder has two stable and two unstable equilibria. Therefore, when launched from any orientation, the object will rotate until it reaches a stable position and then exhibit stable lift.

Figure 13 shows what is predicted to happen when a hemicylindrical rod is released in water, starting with the flat face nearly perpendicular to the incoming light. The object immediately begins rotating toward a stable orientation, but along the way the net momentum vector swings back and forth, and hence the object changes lift directions while still moving along with the light. Once it reaches one of the stable orientations, it begins lifting in a straight line, and will continue indefinitely after that.

All of the ray plots shown have used the Minkowski momentum model, primarily for aesthetics because the Minkowski expression has the property that the momentum vectors are always normal to the surface and pointing away from the higher refractive index. In comparing simulation runs that differ only in which momentum model they use, we find that the overall results for Abraham and Minkowski differ significantly only when an insufficient number of rays are traced. Increasing the ray count causes the two results to

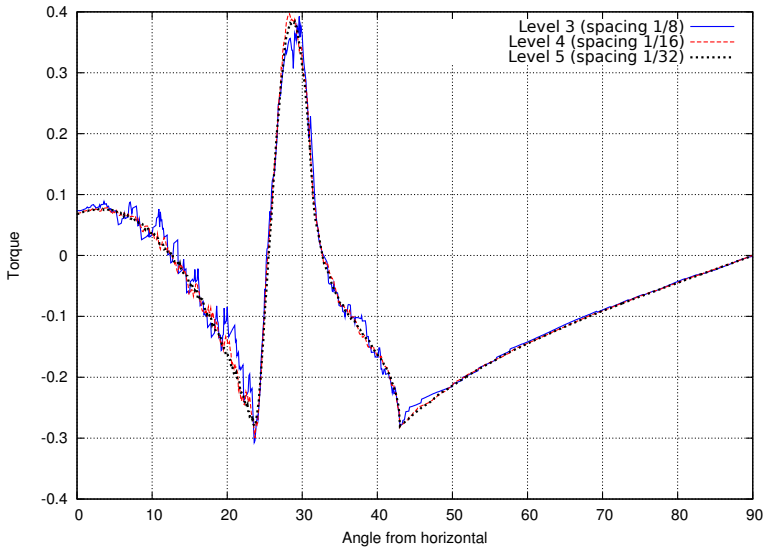


Figure 14: Torque on hemicylinder vs. angle of attack, under varying refinement levels.

converge, the difference between the two providing a measure of the accuracy of the results.

Figure 14 augments the torque vs. angle plot from Figure 12 with a comparison of the results under multiple refinement levels. The refinement level 3 curve appears very noisy at shallow angles, as does level 4 to a lesser extent. This is likely due to the fact that at these shallow angles there are relatively few rays hitting the flat side of the object, and at the same time these rays are experiencing total internal reflection and thus creating large momenta. The combination of a small number of rays having a disproportionately large contribution to the total exacerbates the sampling error in this area. Refinement level 5, with a density of 2^{10} rays per unit area, appears to be a minimum acceptable level for accurate simulations. Note that this conclusion depends on the fact that the hemicylinder under consideration had a radius of 2 units, so for differently-sized objects the refinement level must be adjusted accordingly.

As with most time-based scenarios, the simulation proceeds over a series of time steps. While there exist numerical integration techniques for computer animation that reduce motion error when forces are computed frame-by-frame, we have not utilized these because the exact motion path is not as important as the overall behavior of the object. Figure 15

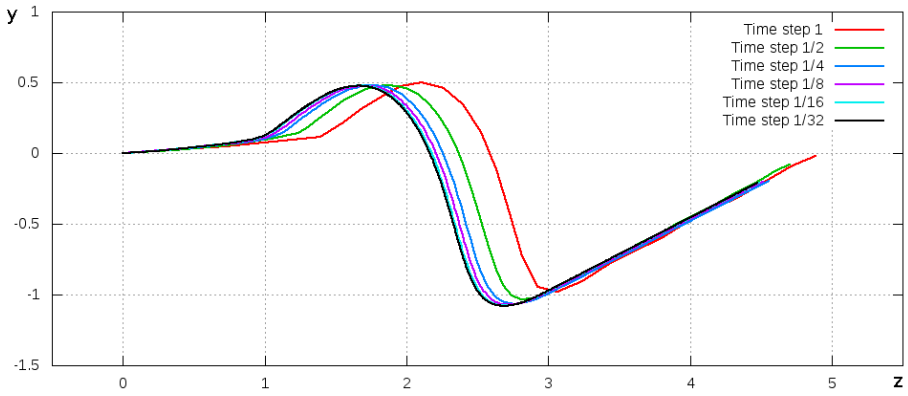
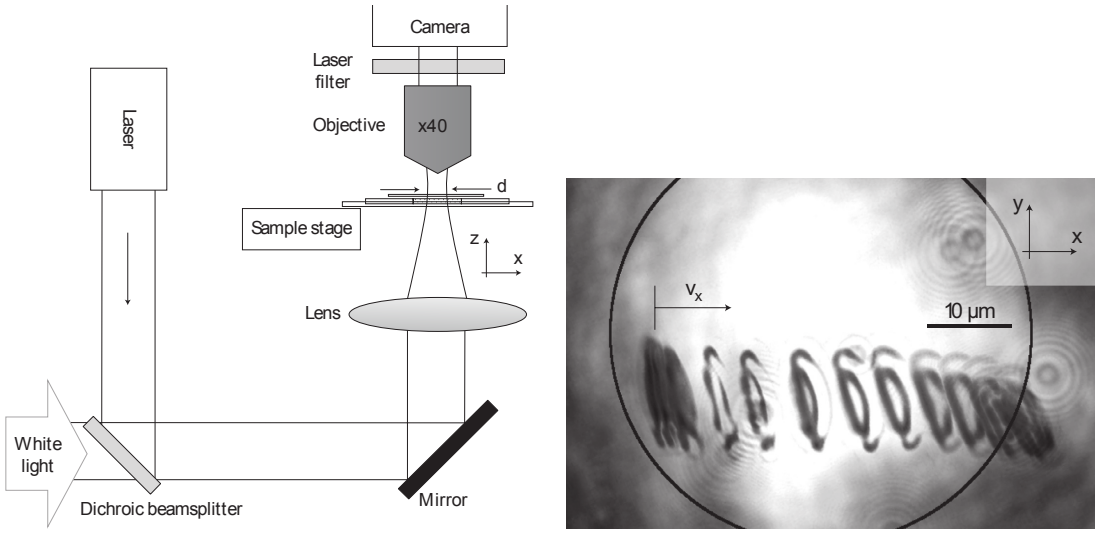


Figure 15: Path of center of mass of the hemicylinder under five different time steps.

shows the hemicylinder's path from Figure 13 when simulated under five different time steps. This demonstrates that the path of the object converges as the time step is decreased, indicating that for this scenario a time step of 1/32 appears to be sufficient.

The preceding analysis has assumed that the axis of the hemicylinder remains perpendicular to the light beam, essentially simulating the behavior of a semicircle in two dimensions. This will clearly not be the case in the physical experiment, as the particle is free to rotate into any orientation. As soon as the particle rotates slightly off-perpendicular, the semicircular flat end caps come into play. In simulation, this results in the particle tumbling end-over-end, rather than stabilizing in a perpendicular orientation, as is desired.

The simulation software was only part of the entire effort toward developing optical lift. After simulations determined that the hemicylindrical particle was viable, a physical experiment was carried out by other members of the team, as described in [14]. In this experiment, microscopic particles were fabricated by melting a rectangle of transparent material on a flat surface. Surface tension resulted in a curved cross-section with a flat base, as desired, but also resulted in the ends of the particle being curved, rather than flat. The particles were 14μm long and 6μm wide. As shown in Figure 16(a), a laser beam was directed vertically upward into a glass cell containing these particles. As predicted by the simulation, the particles rotated to a stable equilibrium orientation, and then experienced a stable lift force perpendicular to the light (horizontally). Figure 16(b) shows the particle's motion over time.



(a) (b)

Figure 16: (a) Experimental setup for observing optical lift. (b) Time-lapse composite image showing the particle's motion from left to right due to optical lift.

As a further demonstration of the capabilities of this software, we also simulated the optical tweezers scenario, using highly-focused light and a spherical particle. Figure 17 is a ray diagram showing how the sphere is pulled toward the trapping point close to the focus. Running the simulation for a grid of particle locations results in the vector field of Figure 18(a), in which we see that the particle will indeed move to the trapping point from anywhere near the focus. The same data can also be visualized as in Figure 18(b), which emphasizes the magnitude of the forces rather than the direction.

7 Conclusion

We have demonstrated an implementation of ray-tracing specialized for the simulation of radiation pressure in refracting objects, a tactic which had not been explored to date. This software was instrumental in preparing the first experimental demonstration of optical lift [14].

While this software is functional for its intended use thus far, there are many improvements that could be made in the future. For continued progress in optical lift, new particle shapes and properties must be tested, which will require enhancements in scene description

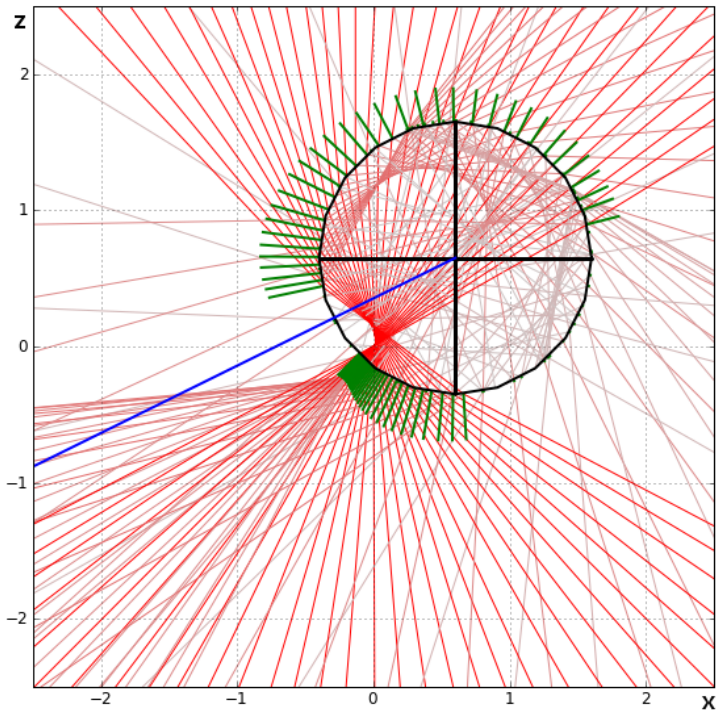
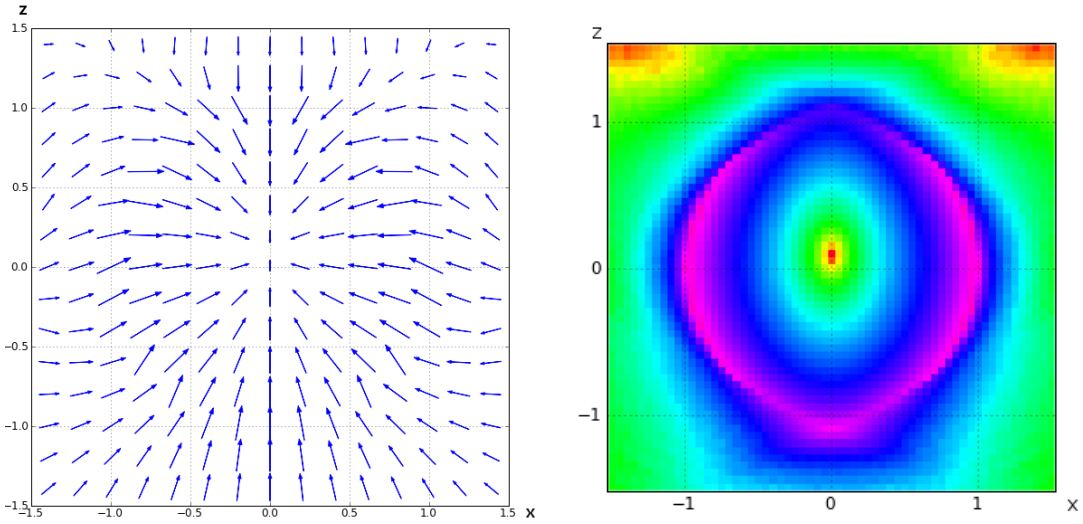


Figure 17: Ray diagram for optical tweezers with spherical particle off-center. Focused light is incident from the bottom and focused at the origin.



(a) (b)

Figure 18: Optical tweezers force map displayed with (a) vector field and (b) color-mapped magnitude (both with magnitude on a logarithmic scale). Light enters the system from the bottom and is focused at the origin, with the trapping point slightly above.

and ray tracing capabilities. Currently only perfectly transparent objects are supported, whereas real materials exhibit some degree of absorption. Different surface treatments would also be useful, so that parts of the object can be made black, white, or mirrored.

A graphical interface for designing particles and controlling the simulation will be beneficial, given that the current system operates solely with a command line interface with text files as configuration. Visually designing the simulation scenario would be more intuitive than entering coordinates in a file. The ray tracing data could also be analyzed graphically, so the user can select individual rays or groups of rays and examine their properties.

The current ray tracer is based on POV-Ray, which introduces unnecessary limitations to the system. A more flexible solution would be to create a new purpose-built ray tracer that fits the needs of this simulation specifically. These enhancements include better control over light source shapes, including polarization in the momentum computations, and simulating multiple objects at once.

Bibliography

[1] Max Abraham. Zur elektrodynamik bewegter körper. *Rendiconti del Circolo Matematico di Palermo (1884 - 1940)*, 28:1–28, 1909. 10.1007/BF03018208.

[2] A. Ashkin. Forces of a single-beam gradient laser trap on a dielectric sphere in the ray optics regime. *Biophysical Journal*, 61(2):569 – 582, 1992.

[3] A. Ashkin. History of optical trapping and manipulation of small-neutral particle, atoms, and molecules. *Selected Topics in Quantum Electronics, IEEE Journal of*, 6(6):841 –856, Nov/Dec 2000.

[4] A. Ashkin, J. M. Dziedzic, J. E. Bjorkholm, and Steven Chu. Observation of a single-beam gradient force optical trap for dielectric particles. *Opt. Lett.*, 11(5):288–290, May 1986.

[5] Stephen M. Barnett. Resolution of the Abraham-Minkowski dilemma. *Phys. Rev. Lett.*, 104:070401, Feb 2010.

[6] M. G. Burt and R. Peierls. The Momentum of a Light Wave in a Refracting Medium. *Royal Society of London Proceedings Series A*, 333:149–156, May 1973.

[7] K Dholakia and T Cizmar. 5(6):335–342, 2011.

[8] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, May-Jun 2007.

[9] Henrik Wann Jensen. Global illumination using photon maps. In *Rendering Techniques '96*, pages 21–30. Springer-Verlag, 1996.

[10] H. Minkowski. Die grundgleichungen für die elektromagnetischen vorgange in bewegten körpern. In *Nachr. Ges. Wiss. Gottingen*, pages 53–111, 1908.

[11] E. F. Nichols and G. F. Hull. The Pressure due to Radiation. *Astrophys. J.*, 17:315–+, June 1903.

[12] Travis E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007.

[13] POV-Ray – Persistence of Vision Raytracer. `http://www.povray.org/`.

[14] Grover A. Swartzlander, Timothy J. Peterson, Alexandra B. Artusio-Glimpse, and Alan D. Raisanen. Stable optical lift. *Nature Photonics*, 5(1):48–51, 2011.

[15] G. Vulpetti, L. Johnson, and G.L. Matloff. *Solar Sails: A Novel Approach to Interplanetary Travel*. Springer, 2010.

[16] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23:343–349, June 1980.

Appendix A

Supplementary media

The accompanying CD contains the source code for the software, as well as three videos, as follows.

`hemicyl_rotate.avi` Shows a hemicylinder rotated through 360 degrees in a parallel light beam.

`hemicyl_motion.avi` Shows the predicted flight path of a hemicylinder, as in Figure 13.

`particle_motion.mov` Shows lift motion of a physical hemicylinder, as in Figure 16(b).

These videos are best viewed with a capable video player such as VLC.

Appendix B

User's guide

1 Overview

The software is stored in a folder called `optics`, and the main entry point is through a script called `run.py`. The folders in the distribution are as follows:

`config` Simulation configuration files.

`output` Contains output files from simulation runs.

`povray` POV-Ray include files referenced by generated scene files.

`python` Root of Python source tree, containing Python `optics` package.

`scripts` Stand-alone utility programs.

2 Installation

This software was developed on Linux and tested on Mac, but has not been ported to Windows. Most Linux distributions have the appropriate packages available, so the system's package manager can be used to install Python, NumPy, and Matplotlib. Mac OS comes with Python pre-installed, but it has proven troublesome to install NumPy and Matplotlib into the stock Python, so it is recommended to install a separate copy directly from the web site at `python.org`, and then install NumPy and Matplotlib into that installation. The 64-bit versions of these packages have not been tested, so using the 32-bit versions is recommended.

Since a custom version of POV-Ray is required, it must be compiled manually. Download the Unix source package (non-GUI version) from `povray.org`. The patch was

created against version 3.6.1, so using future versions may require manual intervention. The patch to the source code is found in the `optics/povray` directory. Unpack the POV-Ray package and apply the patch from within the `povray-3.6.1` source directory using this command:

```
patch -p1 <povray-ray-logging.patch
```

(with the appropriate path to the patch file included).

In order to compile POV-Ray, it must be configured with a text string indicating who compiled it. This is done with a command like:

```
./configure COMPILED_BY="Name <email>"
```

For compilation on Mac, force 32-bit mode by adding these parameters to the configure command:

```
CFLAGS="-m32" CXXFLAGS="-m32"
```

Compile POV-Ray by running `make`, and optionally install it with:

```
sudo make install
```

Installation is only required if the stock POV-Ray is not already installed, because the standard library files must be installed in the standard location.

The file `optics/python/optics/settings.py` contains the configuration for where to find the custom POV-Ray binary. If it has been installed in the standard location, the default of "`povray`" will be sufficient. Otherwise, enter the absolute path to the `povray` executable.

For 64-bit Macs, it is also necessary to instruct Python to use only 32-bit libraries. This is done by editing `optics/run.py` so that the first line reads:

```
#!/usr/bin/env arch -i386 python
```

3 Configuration

All configuration of the simulation system is done using Python files in the `optics/config` directory. Several examples are included, as follows:

hemicyl_torque.py Hemicylinder sampled at a range of angles of attack, generating a torque vs. angle graph (Figure 12).

hemicyl_motion.py Glass hemicylinder in parallel light moving freely in water and reaching a stable angle of attack.

hemicyl_trace.py Same as `hemicyl_motion`, but with a single output plot showing the motion path and periodic orientation markers (Figure 13).

cylinder_3d.py Glass cylinder perpendicular to parallel light, moving freely and plotted in 3D space.

trap_sphere.py Optical tweezers demonstration (Figures 17 and 18).

All data relating to a single simulation run is stored in a Config object. The Config object is always found in the variable `config` in the context of a saved configuration file. We will now walk through the contents of a typical configuration file.

The controller object controls the flow of time during the simulation. The type of controller and the settings provided to it determine how many frames will be simulated and how much time passes between frames. The most basic controller implementation provides for linear time progression, as follows:

```
config.setController(TimeController(
    start = 0.0,        # Time value of first frame
    end   = 20.0,       # Time value of last frame
    step  = 0.5,        # Time step between frames
))
```

Another controller can be substituted that does not increment the time value between frames, but instead changes the orientation of the object for each frame. The following controller configuration would sample the object at rotations of $-90, -85, \ldots 85, 90$ degrees around the X axis.

```
config.setController(RotateController(
    start = -90,        # Angle of first frame
    end   = 90,         # Angle of last frame
    step  = 5,          # Angle between frames
    axis  = X,          # Axis of rotation
))
```

The third type of controller generates frames where the object is sampled by translating to points on a grid. The grid is anchored at a specified center point which will always be sampled, and the grid expands out from that point until it reaches the edges of the bounding box (inclusive). This is so that the sampled grid can be easily aligned with a stable trapping point.

```
config.setController(GridController(
    center  = (0, 0, 0),         # Grid anchor point
    spacing = 0.1,               # Grid spacing
    start   = (-1.5, 0, -1.5),   # Negative corner of bounds
    end     = (1.5, 0, 1.5),     # Positive corner of bounds
))
```

Next we configure the ray tracing engine. Currently only POV-Ray is supported, but others could be plugged in using this mechanism. The `traceLevel` parameter specifies the maximum level of ray bounces to trace. Level 1 goes from the light source to the first object surface, level 2 is all the rays spawned from level 1 rays, and so on. The `maxRefinement` parameter specifies the number of ray refinement levels to generate. Rays are fired in a grid pattern, or an equivalent when using non-parallel light rays. Refinement level 0 is a unit-sized grid, level 1 is an 0.5-unit grid, and so on for each level.

```
config.setRaytracer(POVRay(
    traceLevel = 5,
    maxRefinement = 5,
))
```

The target object(s) are suspended in some medium, the properties of which are specified here. Three materials are predefined: Air, Water, and Glass. Other materials are specified by the index of refraction, *e.g.* `Material(1.25)`.

```
config.scene.medium = Water
```

Both the Abraham or Minkowski light momentum models are supported, and one must be selected here.

```
optics.physics.momentumModel = Minkowski
```

Each addObject() call adds an independent target object to the scene, although the POV-Ray implementation currently supports only one object. Each object can have a unique

name. The object is always created with its coordinate system origin at the object's center of mass. The initial position and orientation in the simulation are optional. The orientation is a 3x3 rotation matrix that can be created using utility functions such as `RotateXMatrix`.

```
config.addObject(Sphere(radius = 1, material = Glass))

config.addObject(
    Cylinder(
        name = 'target',  # Optional unique object name
        axis = X,         # Axis of cylinder
        radius = 1,       # Radius in X-Y plane
        length = 5,       # Length in Z direction
        material = Glass, # Material
    ),
)

config.addObject(
    HemiCylinder(
        name = 'target',  # Optional unique object name
        radius = 1,       # Radius in X-Y plane
        length = 2,       # Length in Z direction
        material = Glass, # Material
    ),
    position = (0, 0, 0), # Initial position (optional)
    orientation =         # Initial orientation
        # Rotation around X axis
        RotateXMatrix(math.radians(0)),
)

config.addObject(
    PolygonPrism(
        name = 'target',  # Optional unique object name
                          # Polygon points in X-Z plane
        points = ((-1, -1), (1, -1), (1, 1), (-1, 1)),
        length = 1,       # Extrusion length in Y direction
        material = Glass, # Material
    ),
)
```

A parallel light source is specified by a point on the plane and the direction of the light rays (also the plane normal). This example creates light rays in the z direction emitted from every point on the $z = -5$ plane.

```
config.addLight(ParallelLight(
    position = (0, 0, -5),
    direction = (0, 0, 1),
))
```

A focused light in POV-Ray is composed of a parallel light facing away from the target and a parabolic reflector. The result is that all of the light rays converge at the origin. The maximum angle of these rays from the z axis can be specified using either a half-angle or a numerical aperture. The distance parameter specifies how far along the $-z$ axis the light source will be, in order to provide enough free space for the target object.

```
config.addLight(FocusedLight(
   distance = 4,
   aperture = 1.2,
#   angle = 70,
))
```

The simulator object creates motion between frames. MotionSimulator is the standard implementation, which applies the light-imparted momentum in 3D. Also, it can optionally apply only the rotational motion and not the translational motion.

```
config.setSimulator(MotionSimulator(
    pinObjects=False,   # Prevent linear motion
))
```

The results of the simulation are delivered to the user using one or more views. Each view extracts information from the frame object and displays or stores it as specified.

The numerical state of each frame is typically stored in a text file for later analysis. The TextOutput view is configured using a list of TextComponent objects which each specify one or more columns of data to be written to the file. Each frame results in one line in the file containing the output from each component in turn. The first line of the file contains the labels for each column.

```
config.addView(TextOutput(
    name='summary',
    filename='summary.csv',
    components=[
        # Timestamp of the frame
        FrameTime(),
        # Sequence number of the frame
        FrameNumber(),
        # Number of rays emitted from the light source that
        # hit the target object
        NumInitialRays(),
        # XYZ position of the center of mass of the object
        ObjectPosition('target'),
        # XYZ vector of momentum imparted to the object
        ObjectMomentum('target'),
        # XYZ vector of torque imparted to the object
        ObjectTorque('target'),
    ]
))
```

Output can be generated in the form of an image file for each frame using a FileImageView. There can be more than one such view in order to create, for instance, multiple perspectives of the object. A view in summary mode generates one image for the entire simulation, as opposed to normal frame-by-frame mode.

```
config.addView(FileImageView(
    # The 'name' parameter is used to create the filename
    # for this view's image for each frame, for example
    # "f00001.plot.png"
    name = 'plot',
    # Enable/disable entire view
    enabled = True,
    # Enable/disable summary mode
    perFrame = False,
    # Output file format (typically png or pdf)
    fileformat = 'png',
    # Size of plot in inches
    size = (10, 5),

    # The appearance of the image is determined by the
```

```
# rendering style selected. Plots can be made in 2D or
# 3D using the Matplotlib plotting library. Choose one
# style per image view.

# 2D plotting style
style = Matplotlib2D(
    # Plot data in the YZ plane, flattening the X axis
    plane = 'yz'
    # Data range to plot (y1, z1, y2, z2)
    range = (-2, -2, 2, 2),
    # If true, 'range' is ignored and set to fit the data
    autoAxes = False
    # If true, X and Y units will be equal lengths
    aspectEqual = True
),

# --OR--

# 3D plotting style
style = Matplotlib3D(
    # Data range (x1, y1, z1, x2, y2, z2)
    range = (-3, -3, -3, 3, 3, 3),
    # Horizontal angle from which to view the 3D space
    azimuth = -30,
    # Vertical angle
    elevation = 30,
),

# Typically many more rays are computed than would be
# useful to plot. The 'rays' parameter is a function
# that selects which rays to plot out of the entire
# collection. The filterRays function can filter by
# either a maximum ray depth or a maximum refinement
# level. The sliceRays function selects only rays
# originating on a given axis plane (0=X, 1=Y, 2=Z).
# The 'chain' parameter allows multiple filters to be
# applied.
rays = filterRays(depth=None, refinement=3,
                  chain=sliceRays(axis=0)),
```

```
    # Any number of different items can be plotted at once.
    components = [
        # Draw a line tracing each ray, with graduated colors
        PlotRays(width=1, color=rayColorGammaRange(
            gamma  = 0.2,
            bright = (1.0, 0.0, 0.0),
            dark   = (0.8, 0.8, 0.8))),
        # Draw line showing momentum at each ray intersection
        PlotRayMomentum(scale=1, color='green', width=2),
        # Draw the outline of the target object(s)
        PlotObjectOutline(color='black', width=2),
        # Draw a line showing the net momentum on the object
        PlotObjectMomentum(scale=1, color='blue', width=2),
    ]
))
```

Plotting components are Python objects that convert the simulation data into plotted shapes. Some of the frequently-used components for per-frame plots are as follows:

PlotObjectOutline Draws an outline of the object.

PlotRays Draws a line corresponding to each ray, with color varying according to intensity.

PlotRayMomentum Draws a vector at the end of each ray, showing the direction and magnitude of the momentum transferred from this ray and its children to the object.

PlotObjectMomentum Draws a single vector from the object's center of mass indicating the net force experienced by the object.

PlotObjectTorque Draws a single vector indicating the net torque experienced by the object.

A different set of plotting components is used for summary plots, typically when evaluating the object at a grid of points in space:

PlotPointPerFrame Plots one point per frame and lines between them, where the x and y coordinates are extracted from the frame as the user desires.

PlotVectorField Plots a vector at the center of mass of the object for each frame, typically showing the force on the object at that point.

PlotColorGrid Similar to PlotVectorField, except that the results are squares of color corresponding to the magnitude of the force.

Examples of the usage of these components can be found in the various demonstration configurations.

4 Usage

To run a simulation, first choose a file from the `config` directory, for example `hemicyl_motion.py`. In a shell, go to the `optics` directory and enter:

```
./run.py hemicyl_motion.py
```

Status information will be displayed as the simulation progresses, including the frame number, simulation time value, elapsed time, and estimated time remaining. The output files will be found in the `output/hemicyl_motion` directory.

If the Matplotlib library is able to access a windowing system, it is possible to see the output plots in real time using interactive mode. To enable this, append `interactive=1` to the command line. One window will open for each plot, and they will update after each frame is rendered.

Internally, each command line argument after the configuration filename is treated as a Python statement executed in the context of the Config object. Therefore, passing `interactive=1` is equivalent to including `config.interactive=1` in the config file. One other useful option accessible in this way is `keepTempFiles=1`, which prevents the temporary output files from being deleted from the output directory after each frame. This includes the POV-Ray scene file `f00001.pov`, the POV-Ray error output `f00001.povout`, and the ray log file `f00001.trace`. This can be useful for debugging purposes if a simulation fails to complete.