Extracted from:

The Ray Tracer Challenge

A Test-Driven Guide to Your First 3D Renderer

This PDF file contains pages extracted from *The Ray Tracer Challenge*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2019 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

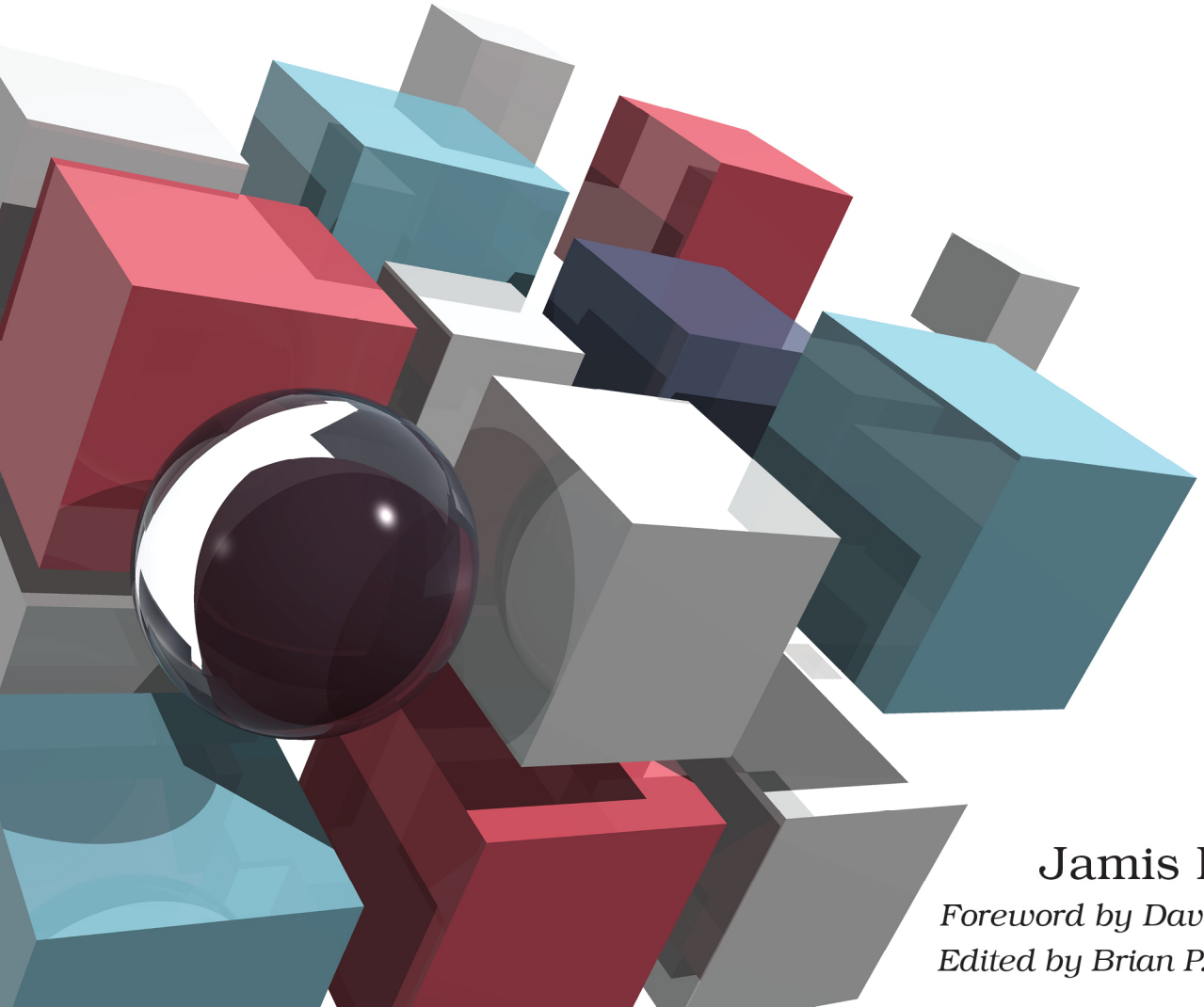
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

The Ray Tracer Challenge

A Test-Driven Guide to Your First 3D Renderer



Jamis Buck

Foreword by David Buck

Edited by Brian P. Hogan

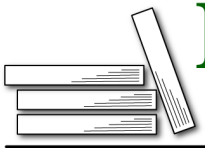
The Ray Tracer Challenge

A Test-Driven Guide to Your First 3D Renderer

Jamis Buck

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Brian P. Hogan
Copy Editor: L. Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Copyright © 2019 The Pragmatic Programmers, LLC.

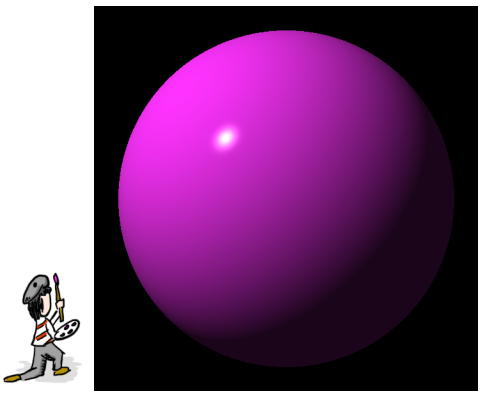
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 978-1-68050-271-8
Book version: P1.0—February 2019

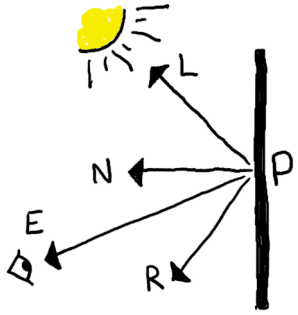
Hot diggity! You are unstoppable. You just drew the silhouette of a three-dimensional sphere with nothing but some code and math! That's, like, level-10 wizard stuff.

Still—sad, but true!—the results are not quite what most people think of as "3D rendered." Time to fix that.

In this chapter, you'll implement a model to simulate the reflection of light from a surface, which will finally allow you to draw that sphere and make it look three dimensional. In fact, by the end of the chapter, you'll have rendered an image very much like this one:



To do this, you'll add a source of light, and then implement a shading algorithm to approximate how brightly that light illuminates the surfaces it shines on. It might sound complicated, but it's not. The truth is that most ray tracers favor approximations over physically accurate simulations, so that to shade any point, you only need to know four vectors. These are illustrated in the figure on page 5.



If `P` is where your ray intersects an object, these four vectors are defined as:

- `E` is the *eye vector*, pointing from `P` to the origin of the ray (usually, where the eye exists that is looking at the scene).

• Click HERE to purchase this book now. discuss

- `L` is the *light vector*, pointing from `P` to the position of the light source.
- `N` is the *surface normal*, a vector that is perpendicular to the surface at `P`.
- `R` is the *reflection vector*, pointing in the direction that incoming light would bounce, or reflect.

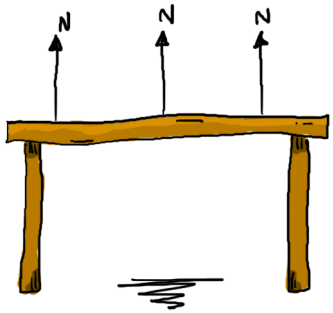
You already have the tools to compute the first two vectors:

- To find `E`, you can negate the ray's direction vector, turning it around to point back at its origin.
- To find `L`, you subtract `P` from the position of the light source, giving you the vector pointing toward the light.

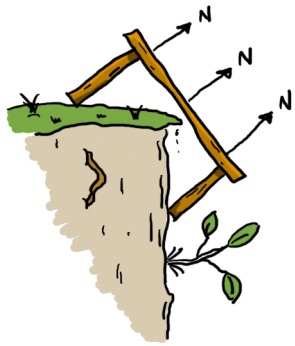
The surface normal and reflection vector, though…those are new. Before you can use those, we need to pause and talk about how to compute them.

Surface Normals

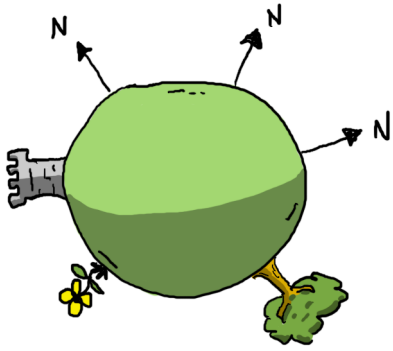
A *surface normal* (or just *normal*) is a vector that points perpendicular to a surface at a given point. Consider a table, as shown in the following figure.



A flat surface like a table will have the same normal at every point on its surface, as shown by the vectors labeled `N`. If the table is level, the normals will be the same as "up," but even if we tilt the table, they'll still be perpendicular to the table's surface, like the following figure shows.



Things get a little trickier when we start talking about nonplanar surfaces (those that aren't uniformly flat). Take the planetoid in the following figure for example.



The three normal vectors certainly aren't all pointing the same direction! But each is perpendicular to the surface of the sphere at the point where it lives.

Let's look at how to actually compute those normal vectors.

Computing the Normal on a Sphere

Start by writing the following tests to demonstrate computing the normal at various points on a sphere. Introduce a new function, `normal_at(sphere, point)`, which will return the normal on the given sphere, at the given point. You may assume that the point will always be on the surface of the sphere.

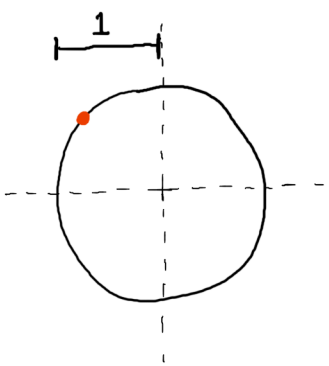
• Click HERE to purchase this book now. discuss

features/spheres.feature
```
Scenario: The normal on a sphere at a point on the x axis
  Given s ← sphere()
  When n ← normal_at(s, point(1, 0, 0))
  Then n = vector(1, 0, 0)

Scenario: The normal on a sphere at a point on the y axis
  Given s ← sphere()
  When n ← normal_at(s, point(0, 1, 0))
  Then n = vector(0, 1, 0)

Scenario: The normal on a sphere at a point on the z axis
  Given s ← sphere()
  When n ← normal_at(s, point(0, 0, 1))
  Then n = vector(0, 0, 1)

Scenario: The normal on a sphere at a nonaxial point
  Given s ← sphere()
  When n ← normal_at(s, point(√3/3, √3/3, √3/3))
  Then n = vector(√3/3, √3/3, √3/3)
```

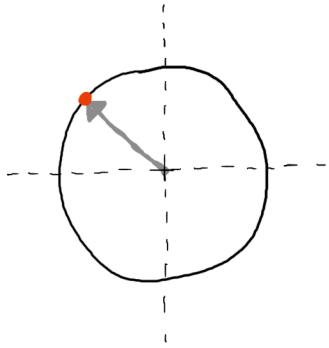
One other feature of these normal vectors is hiding in plain sight: they're *normalized*. Add the following test to your suite, which shows that a surface normal should always be normalized.

features/spheres.feature
```
Scenario: The normal is a normalized vector
  Given s ← sphere()
  When n ← normal_at(s, point(√3/3, √3/3, √3/3))
  Then n = normalize(n)
```

Now, let's make those tests pass by implementing that `normal_at()` function. To understand how it will work its magic, take a look at the unit circle in the following figure. It's centered on the origin, and a point (presumably a point of intersection) has been highlighted on its circumference.



Let's say you want to find the normal at that highlighted point. Draw an arrow from the origin of the circle to that point, as in the following figure.



It turns out that this arrow—this vector!—is perpendicular to the surface of the circle at the point where it intersects. It's the normal! Algorithmically speaking, you find the normal by taking the point in question and subtracting the origin of the sphere ((0,0,0) in your case). Here it is in pseudocode:

```
function normal_at(sphere, p)
  return normalize(p - point(0, 0, 0))
end function
```

(Note that, because this is a unit sphere, the vector will be normalized by default for any point on its surface, so it's not strictly necessary to explicitly normalize it here.)

If only that were all there were to it! Sadly, the sphere's transformation matrix is going to throw a (small) wrench into how the normal is computed. Let's take a look at what needs to happen for the normal calculation to compensate for a transformation matrix.

• Click HERE to purchase this book now. discuss