Extracted from:

The Ray Tracer Challenge

A Test-Driven Guide to Your First 3D Renderer

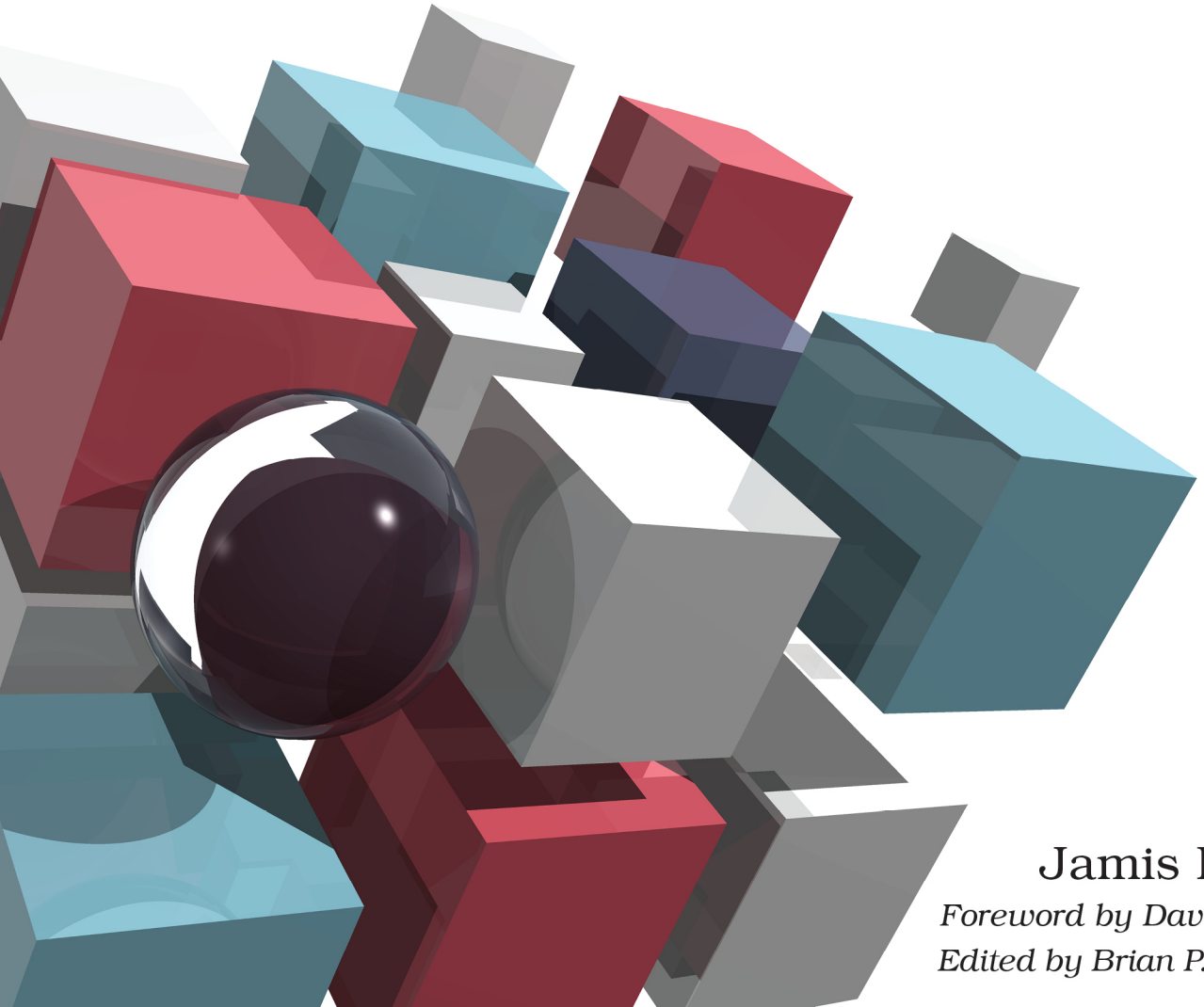
The Pragmatic Bookshelf

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

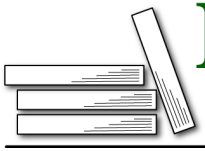
The Ray Tracer Challenge

A Test-Driven Guide to Your First 3D Renderer

Jamis Buck

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

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*.

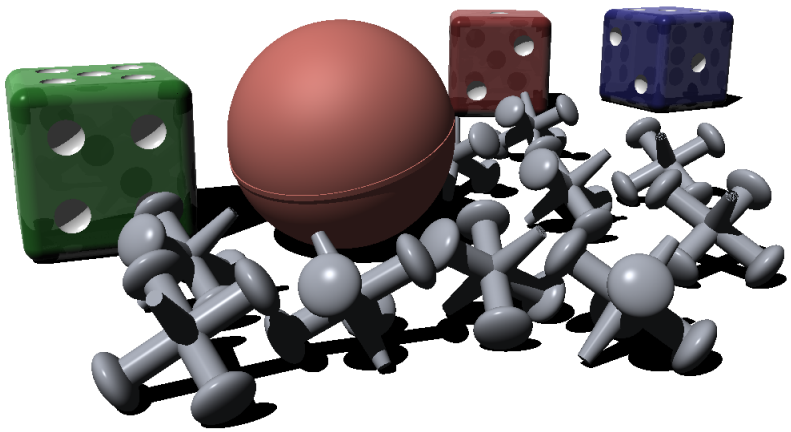
For international rights, please contact *rights@pragprog.com*.

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

Getting Started

Okay. You are officially awesome. You're one of *those* programmers, the ones who actively seek out new ways to apply their craft and tackle challenges for the thrill of it. You're in good company!

With this book, you're going to build a 3D renderer from scratch. Specifically, you'll build a *ray tracer*, casting rays of light backward into a scene and following their paths as they bounce around toward a light source. It's generally not a very fast technique (and so isn't well-suited for real-time rendering) but it can produce very realistic results. By the end of this book, you'll be able to render scenes like this one:



And you don't have to be a mathematician or computer scientist to do it!

Beginning at the bottom, you'll build a foundation of basic routines and tools. You'll use those to bootstrap other routines, making light rays, shapes, and functions to predict how they'll interact. Then things start moving quickly, and within a few chapters you'll be producing realistic images of 3D spheres. You'll add shadows and visual effects like geometric patterns, mirror reflections, and glass. Other shapes follow—planes, cubes, cylinders, and more.

By the end of the book, you'll be taking these primitive shapes and combining them in complex ways using set operations. There'll be no stopping you!

The specific algorithm you'll implement is called *Whitted ray tracing*,[1] named for Turner Whitted, the researcher who described it in 1979. It's often referred to as *recursive* ray tracing, because it works by recursively spawning *rays* (lines representing rays of light) and bouncing them around the scene to discover what color each pixel of the final image should be. In a nutshell, the algorithm works like this for each of the image's pixels:

1. Cast a ray into the scene, and find where it strikes a surface.
2. Cast a ray from that point toward each light source to determine which lights illuminate that point.
3. If the surface is reflective, cast a new ray in the direction of reflection and recursively determine what color is reflected there.
4. If the surface is transparent, do the same thing in the direction of refraction.
5. Combine all colors that contribute to the point (the color of the surface, the reflection, and refraction) and return that as the color of the pixel.

Over the course of this book, you'll implement each of those steps, learning how to compute reflection vectors, how to approximate refraction, how to intersect rays with various primitive shapes, and more. Sooner than you might think, you'll be rendering awesome 3D scenes!

Who This Book Is For

Ultimately, this book is for anyone who loves writing code, but you'll get the most out of it if:

- You have prior experience writing software (perhaps a year or more).
- You've written unit tests before.
- You like tinkering and experimenting with code and algorithms.

It really doesn't matter what programming environment or operating system you prefer. The only code in this book is pseudocode. Admittedly, the explanations *do* tend toward imperative, procedural, and object-oriented languages, but the concepts and tests themselves are translatable to any environment you wish.

1. en.wikipedia.org/wiki/Ray_tracing_(graphics)#Recursive_ray_tracing_algorithm

How to Read This Book

Each chapter is presented as a series of tests covering a small piece of the overall ray tracer. Since each one builds on previous chapters, you'll be most successful if you read them in sequence.

You'll implement your ray tracer in test-first style, writing a few tests at a time and making them pass by implementing the corresponding functions and features in code. The first half of the book is structured to take you smoothly from test to test, but as you get into the second half of the book, the pace picks up. With greater experience comes greater responsibility! You'll still be given the tests, but there will be less hand-holding, and the tests will be presented in a more linear fashion, almost like a checklist.

Each chapter introduces one or more new features, discusses how the feature works at a high level, and then walks you through the tests and how to make them pass. The tests are posed as Cucumber scenarios,[2] but it is absolutely *not* necessary to use Cucumber to implement them. Please feel free to use whatever system you prefer to write your tests!

Typically, Cucumber is used to describe high-level interactions between a user and an application, but the tests in this book use it differently. Here, you'll see it used to describe lower-level interactions, like how various inputs to a specific function might affect the function's output. This lets the book walk you through the construction of an API, step by step, rather than just showing you the high-level behavior that you need to try to emulate. For example, consider the following hypothetical specification which describes the behavior of concatenating two arrays.

```
Scenario: Concatenating two arrays should create a new array
  Given a ← array(1, 2, 3)
    And b ← array(3, 4, 5)
  When c ← a + b
  Then c = array(1, 2, 3, 3, 4, 5)
```

It's structured like any Cucumber scenario, but describes low-level API interactions:

- It begins with two assumptions ("`Given…And`"), which must be true to start. These use left arrows (←) to assign two arrays to two variables, `a` and `b`.

2. Technically, the tests are written in Gherkin, which is the language in which Cucumber specs are written. See cucumber.io.

- After everything has been initialized, an action occurs ("When"). The result of this action is what is to be tested. Note that this also uses the left arrow, assigning the result of concatenating `a` and `b` to another variable, `c`.
- Finally, an assertion is made ("Then"), which must be true. This uses the equals operator (`=`) to assert that the variable `c` is equal to the given array.

Your job as the reader is to implement each test, and then make each pass. You're welcome to do so in Cucumber if you like—in fact, the Cucumber tests may be downloaded from the publisher,[3] to save you the effort of keying them all in by hand. But if Cucumber isn't your thing, you can be just as successful by translating the Cucumber scenarios into whatever testing system you prefer. Honestly, part of the puzzle—part of the fun!—is translating each specification into a working unit test. The scenario tells you what the behavior should be. *You* get to decide how to make it happen.

While working through this book, you're going to discover that an implementation that worked for one test might not work well (or at all) for a later test. You'll need to be flexible and willing to refactor as you discover new requirements. That, or read the entire book through before beginning your implementation so you know what's coming up.

Also, be aware that I've made many of the architectural decisions in this book with the goal of being easy to explain. Often, there will be more efficient ways to implement a function, or to architect a feature. You may disagree with the book at times, and that's okay! This book is a roadmap, describing just one of many possible ways to get to the goal. Follow your own aesthetic sense. Make your code your own.

Lastly, at the end of each chapter is a section called "Putting It Together." This is where you'll find a description of something that builds on the code you wrote for that chapter and gives you a chance to play and experiment with your new code. Sometimes it will be a small project, and other times a list of possible things to try or directions to explore. It's certainly possible to skip those sections if you're in a hurry, but if you do you'll be missing one of the most enjoyable parts of the journey.

Things to Watch Out For

A ray tracer is math-heavy. There's no getting around it. It works its magic by crunching numbers, finding intersections between lines and shapes, computing reflections and refractions, and blending colors. So, yes, there will

3. pragprog.com/book/jbtracer/the-ray-tracer-challenge

be a great deal of math here, but I will mostly give it to you, ready to implement. You'll find little or no focus on where the math comes from, no derivations of formulas, no explanations of why an equation does what it does. You'll see the formulas and, where necessary, walk through how to implement them, but you won't wade through proofs and derivations. If the proofs and derivations are what you particularly enjoy, you can always find a great deal of information about them online.

Also, number-crunching tends to be fairly CPU-intensive. A ray tracer offers a lot of opportunities to optimize code, but that's not the focus of this book. If you follow along and implement just what is described, your code will probably not be very efficient or very fast—but it *will* work. Think of optimization as a bonus exercise!

Other things to watch out for, in no particular order, are these:

Comparing floating-point numbers

Especially in tests, you'll need to be able to compare two floating-point numbers to determine if they are approximately equal. The specifications in the book represent this loose comparison with a simple equals sign. In practice, you'll need to be more explicit and test that the two numbers are within an error value that the book refers to as *EPSILON*, something like this: $|a - b| < EPSILON$. In practice, using a tiny value like 0.0001 for `EPSILON` is generally fine.

Comparing data structures

As with comparing numbers, it's also assumed that you'll need to compare data structures to see if they are equal. For example, you'll need to be able to see whether two points are the same. These comparison routines aren't explicitly described in the book, but you'll need to implement them all the same. It wouldn't hurt to add tests for these routines, too, despite them not being given in the book.

Representing infinity

In later chapters, like Chapter 12, *Cubes*, on page ?, and Chapter 13, *Cylinders*, on page ?, you'll need to be able to compare numbers with infinity. If your programming language can represent infinity natively, that's great! Otherwise, you can usually fake it by using a very large number instead. (Something like 1×10^{10} is usually plenty. In many programming languages, you can write that as `1e10`.)

Use your own names and architecture!

The names of functions and variables given in the book are just recommendations. The functions are designed so that the first argument is the

"responsible party," or the entity with responsibility for the domain in question. In object-oriented terms, the first argument would be the self object. But don't let this stop you from reassigning those responsibilities if you prefer. You should always feel free to choose names more appropriate to your own architecture.

Also, the ray tracer will be described imperatively, but you should look for ways to adapt these descriptions to the strengths and idioms of your programming environment. Embrace your classes, modules, namespaces, actors, and monads, and make this ray tracer your own!

A lot of work has gone into making sure everything in this book is accurate and error-free, but nobody's perfect. If you happen to find a mistake somewhere, please let me know about it. You can report errata on the book's web site.[4] And be sure to visit the book's discussion forum,[5] where you can ask questions, share tips and tricks, and post eye candy you've rendered with your ray tracer. This forum is purely my own and is not affiliated with the Pragmatic Bookshelf in any way.

With all that out of the way, brace yourself—we're going to jump right in and get started. This is going to be fun!

4. pragprog.com/book/jbtracer/the-ray-tracer-challenge
5. forum.raytracerchallenge.com