Extracted from:

# The Ray Tracer Challenge

## A Test-Driven Guide to Your First 3D Renderer

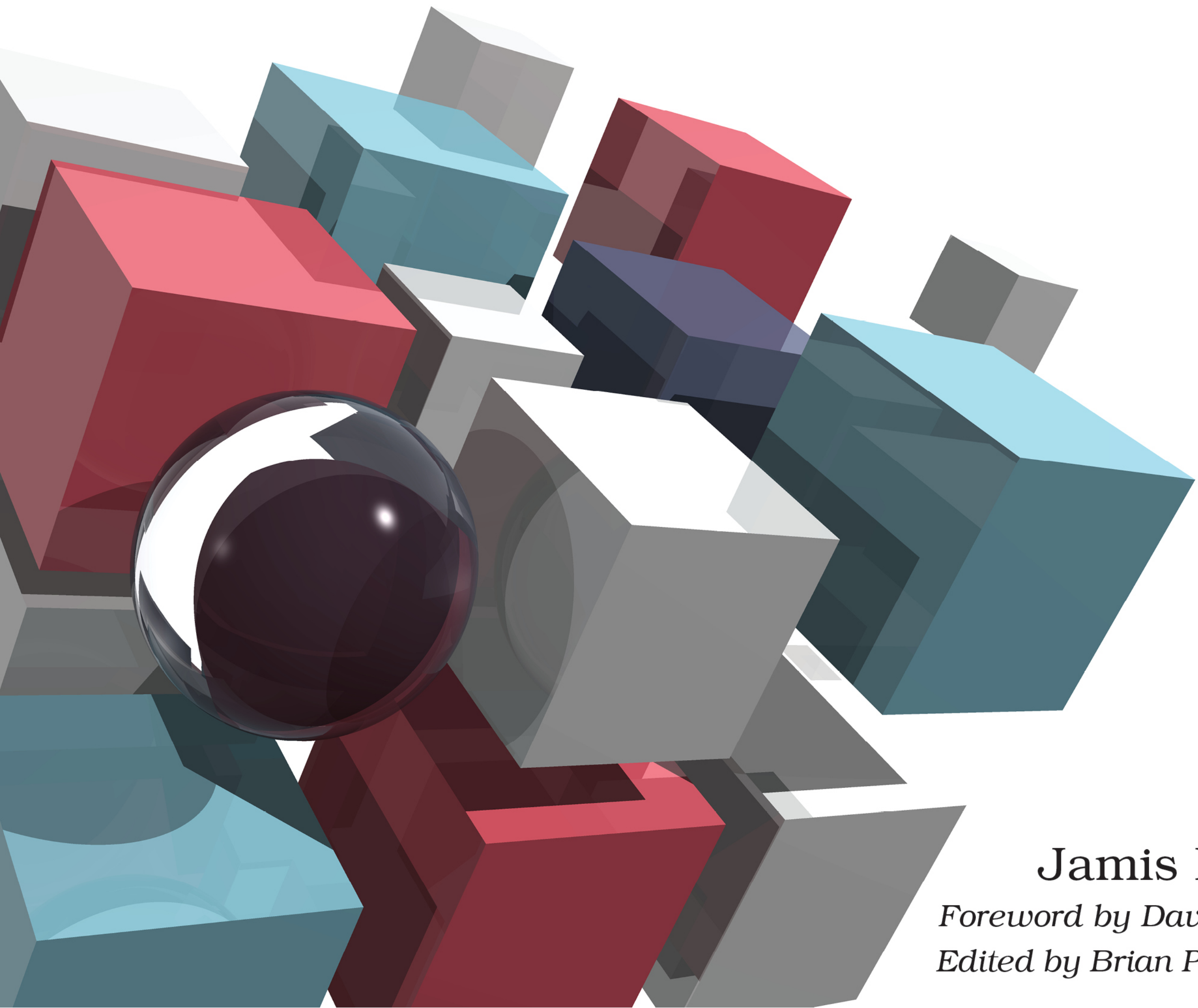
The Pragmatic Bookshelf

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

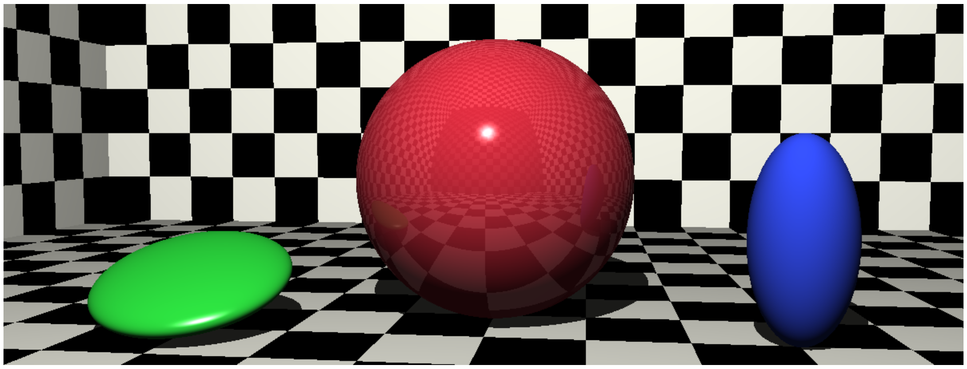
The Pragmatic Bookshelf
Raleigh, North Carolina

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

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

Hey, look. That shiny red sphere from before has company now. Its friends appear to be a cigar-looking matte blue ovoid, and a squashed green plastic thing that's tipped toward us, as if curious to see who's looking.



Would it surprise you to learn that these are all just spheres? They've been moved around, scaled, and rotated a bit too, but deep down, they're all still perfectly spherical. These transformations are all thanks to a little thing called a *matrix*.

A matrix is a grid of numbers that you can manipulate as a single unit. For example, here's a 2x2 matrix. It has two rows and two columns.

$$\begin{bmatrix} 3 & 1 \\ 2 & 7 \end{bmatrix}$$

And here's a 3x5 matrix, with three rows and five columns:

$$\begin{bmatrix} 9 & 1 & 2 & 0 & 3 \\ 0 & 0 & 2 & 3 & 1 \\ 8 & 7 & 5 & 4 & 6 \end{bmatrix}$$

For your ray tracer, you'll use primarily 4x4 matrices—those with exactly four rows and four columns, like this:

$$\begin{bmatrix} 1 & 2 & 0 & 0 \\ 0 & 1 & 4 & 1 \\ 0 & 1 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In this chapter, you'll implement a 4x4 matrix data structure and a few general matrix operations. In the chapter after this one, Chapter 4, *Matrix Transformations*, on page ?, you'll build on those operations, adding functionality to make it easier to manipulate points and vectors (and, ultimately, shapes).

Ready? Let's do this!

## Creating a Matrix

First things first. You need to be able to describe a new matrix. Write a test like the following, which shows that a matrix is composed of four rows of four floating point numbers each, for a total of sixteen numbers. It should also show how to refer to the elements of the matrix.

features/matrices.feature

```
Scenario: Constructing and inspecting a 4x4 matrix
  Given the following 4x4 matrix M:
    |  1   |  2   |  3   |  4   |
    |  5.5 |  6.5 |  7.5 |  8.5 |
    |  9   | 10   | 11   | 12   |
    | 13.5 | 14.5 | 15.5 | 16.5 |
  Then M[0,0] = 1
    And M[0,3] = 4
    And M[1,0] = 5.5
    And M[1,2] = 7.5
    And M[2,2] = 11
    And M[3,0] = 13.5
    And M[3,2] = 15.5
```

The first thing to notice is when talking about the individual elements of the matrix, we specify the element's *row* first, and then its *column*. For example, element $M_{23}$ is the one at row 2, column 3. Also note in this book, row and column indices will be zero-based, so row 2 is actually the third row.

Later, in *Inverting Matrices*, on page ?, you'll need to be able to instantiate both 2x2 and 3x3 matrices in addition to 4x4 matrices, so take a moment to make sure you can create matrices of those sizes as well. Add the following tests to show that your code supports those dimensions:

features/matrices.feature

```
Scenario: A 2x2 matrix ought to be representable
  Given the following 2x2 matrix M:
    | -3 |  5 |
    |  1 | -2 |
  Then M[0,0] = -3
    And M[0,1] = 5
    And M[1,0] = 1
    And M[1,1] = -2

Scenario: A 3x3 matrix ought to be representable
  Given the following 3x3 matrix M:
    | -3 |  5 |  0 |
    |  1 | -2 | -7 |
    |  0 |  1 |  1 |
  Then M[0,0] = -3
    And M[1,1] = -2
    And M[2,2] = 1
```

Keep your matrix implementation as simple as possible. Prefer native types wherever you can, and avoid complicated abstractions. Your matrices will be doing a lot of work!

Another critical part of your matrix implementation is *matrix comparison*. You'll be comparing matrices a lot, especially in this chapter and the next, so it's important to get it right. The following two tests are not exhaustive but ought to point you in the right direction. For example, you'll want to make sure that very similar numbers are handled correctly when comparing matrices, as described in *Comparing Floating Point Numbers*, on page ?.

features/matrices.feature

```
Scenario: Matrix equality with identical matrices
  Given the following matrix A:
      | 1 | 2 | 3 | 4 |
      | 5 | 6 | 7 | 8 |
      | 9 | 8 | 7 | 6 |
      | 5 | 4 | 3 | 2 |
    And the following matrix B:
      | 1 | 2 | 3 | 4 |
      | 5 | 6 | 7 | 8 |
      | 9 | 8 | 7 | 6 |
      | 5 | 4 | 3 | 2 |
  Then A = B

Scenario: Matrix equality with different matrices
  Given the following matrix A:
      | 1 | 2 | 3 | 4 |
      | 5 | 6 | 7 | 8 |
      | 9 | 8 | 7 | 6 |
      | 5 | 4 | 3 | 2 |
    And the following matrix B:
      | 2 | 3 | 4 | 5 |
      | 6 | 7 | 8 | 9 |
      | 8 | 7 | 6 | 5 |
      | 4 | 3 | 2 | 1 |
  Then A != B
```

Once you've got the basic matrix data structure working, linear algebra is your oyster. We're going to do some wild things with matrices, but we'll start small; let's talk about multiplying them together.

## Multiplying Matrices

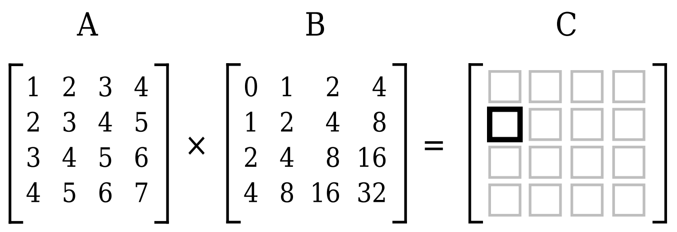
Multiplication is the tool you'll use to perform transformations like scaling, rotation, and translation. It's certainly possible to apply them one at a time, sequentially, but in practice you'll often want to apply several transformations at once. Multiplying them together is how you make that happen, as you'll see when you get to Chapter 4, *Matrix Transformations*, on page ?.

So let's talk about matrix multiplication. It takes two matrices and produces another matrix by multiplying their component elements together in a specific way. You'll see how that works shortly, but start first by writing a test that describes what you expect to happen when you multiply two 4x4 matrices together. Don't worry about 2x2 or 3x3 matrices here; your ray tracer won't need to multiply those at all.

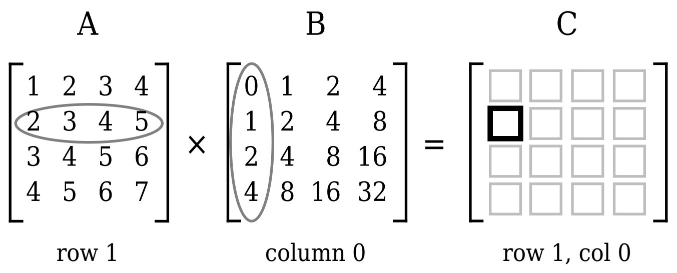
**features/matrices.feature**

```
Scenario: Multiplying two matrices
  Given the following matrix A:
      | 1 | 2 | 3 | 4 |
      | 5 | 6 | 7 | 8 |
      | 9 | 8 | 7 | 6 |
      | 5 | 4 | 3 | 2 |
    And the following matrix B:
      | -2 | 1 | 2 |  3 |
      |  3 | 2 | 1 | -1 |
      |  4 | 3 | 6 |  5 |
      |  1 | 2 | 7 |  8 |
  Then A * B is the following 4x4 matrix:
      | 20|  22 |  50 |  48 |
      | 44|  54 | 114 | 108 |
      | 40|  58 | 110 | 102 |
      | 16|  26 |  46 |  42 |
```

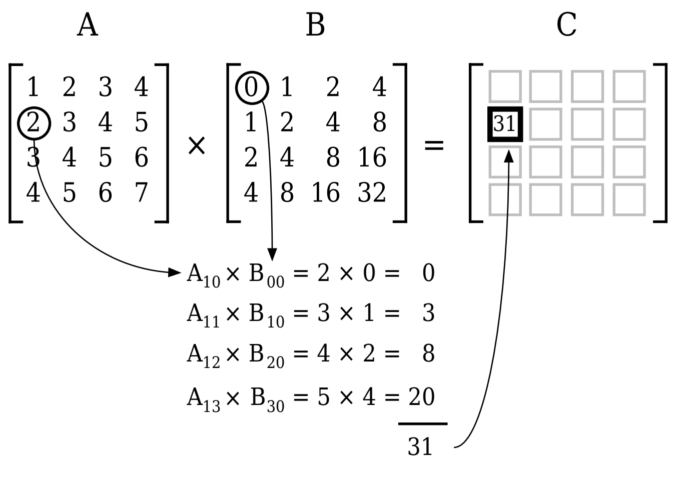
Let's look at how this is done for a single element of a matrix, going step-by-step to find the product for element $C_{10}$, highlighted in the figure on page 9.



Element $C_{10}$ is in row 1, column 0, so you need to look at row 1 of the `A` matrix, and column 0 of the `B` matrix, as shown in the following figure.



Then, you multiply corresponding pairs of elements together ($A_{10}$ and $B_{00}$, $A_{11}$ and $B_{10}$, $A_{12}$ and $B_{20}$, and $A_{13}$ and $B_{30}$), and add the products. The following figure shows how this comes together.


A
B
C
31
$A_{10} \times B_{00} = 2 \times 0 = 0$
$A_{11} \times B_{10} = 3 \times 1 = 3$
$A_{12} \times B_{20} = 4 \times 2 = 8$
$A_{13} \times B_{30} = 5 \times 4 = 20$
31

The result, here, is `31`, and to find the other elements, you perform this same process for each row-column combination of the two matrices.

Stated as an algorithm, the multiplication of two 4x4 matrices looks like this:

1. Let `A` and `B` be the matrices to be multiplied, and let `M` be the result.

2. For every row `r` in `A`, and every column `c` in `B`:
3. Let $M_{rc} = A_{r0} * B_{0c} + A_{r1} * B_{1c} + A_{r2} * B_{2c} + A_{r3} * B_{3c}$

As pseudocode, the algorithm might look like this:

```
function matrix_multiply(A, B)
  M ← matrix()

  for row ← 0 to 3
    for col ← 0 to 3
      M[row, col] ← A[row, 0] * B[0, col] +
                     A[row, 1] * B[1, col] +
                     A[row, 2] * B[2, col] +
                     A[row, 3] * B[3, col]
    end for
  end for

  return M
end function
```

If this all feels kind of familiar, it might be because you've already implemented something very similar—the dot product of two vectors on page ?. Yes, it's true. Matrix multiplication computes the dot product of every row-column combination in the two matrices! Pretty cool.

Now, we're not done yet. Matrices can actually be multiplied by *tuples*, in addition to other matrices. Multiplying a matrix by a tuple produces another tuple. Start with a test again, like the following, to express what you expect to happen when multiplying a matrix and a tuple.

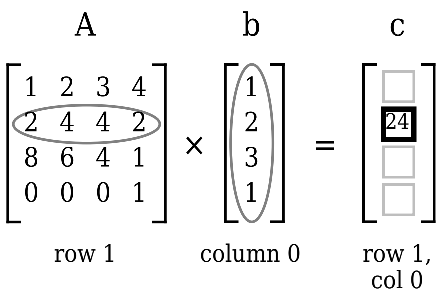
features/matrices.feature
```
Scenario: A matrix multiplied by a tuple
  Given the following matrix A:
      | 1 | 2 | 3 | 4 |
      | 2 | 4 | 4 | 2 |
      | 8 | 6 | 4 | 1 |
      | 0 | 0 | 0 | 1 |
    And b ← tuple(1, 2, 3, 1)
  Then A * b = tuple(18, 24, 33, 1)
```

How does it work? The trick begins by treating the tuple as a really skinny (one column!) matrix, like this:

$$(1, 2, 3, 1) \Rightarrow \begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \end{bmatrix}$$

Four rows. One column.

It comes together just as it did when multiplying two 4x4 matrices together, but now you're only dealing with a single column in the second "matrix." The following figure illustrates this, highlighting the row and column used when computing the value of $c_{10}$.



To compute the value of $c_{10}$, you consider only row 1 of matrix `A`, and column 0 (the *only* column!) of tuple `b`. If you think of that row of the matrix as a tuple, then the answer is found by taking the dot product of that row and the other tuple:

$$2 \times 1 + 4 \times 2 + 4 \times 3 + 2 \times 1 = 24$$

The other elements of `c` are computed similarly. It really is the exact same algorithm used for multiplying two matrices, with the sole difference being the number of columns in the second "matrix."

If you're feeling uncomfortable with how much magic there is in these algorithms, check out "An Intuitive Guide to Linear Algebra"[1] on BetterExplained.com. It does a good job of making sense of this stuff!

Pause here to make the tests pass that you've written so far. Once you have them working, carry on! We're going to look at a very special matrix, and we'll use multiplication to understand some of what makes it so special.

1. betterexplained.com/articles/linear-algebra-guide