Metal by Example

High-performance graphics and data-parallel programming for iOS and macOS

Linear Algebra for Graphics Programming

16 Comments / Math / September 14, 2014

Introduction

This post will cover the essential mathematics for doing 3D graphics programming. I chose to split it out into a separate post because there is quite a lot of ground to cover, and attempting to wedge all of these concepts into a tutorial post would be overwhelming. If you already have a grasp of this material, this post is largely optional, but it does establish the notational and geometric conventions I use elsewhere.

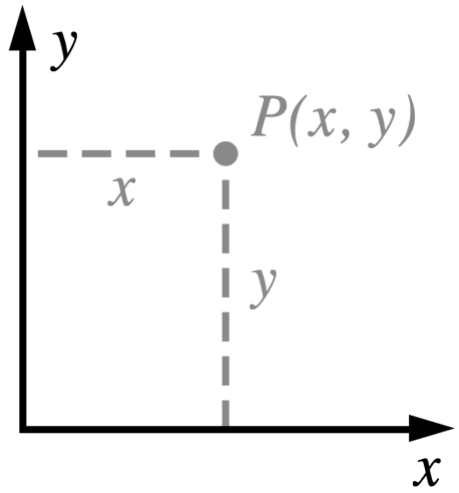
This is a living document, and I will be expanding the material substantially over the next couple of months. In particular, I'll try to include more examples and additional illustrations.

From 2D to 3D

In order to make the move to three-dimensional space, we need to introduce a few new mathematical concepts. We need to know how to represent points in 3D space, how to move points between different coordinate frames, and how to remove the third dimension when projecting points onto the screen.

The Cartesian Plane

We start with the familiar Cartesian plane from your middle school algebra class:

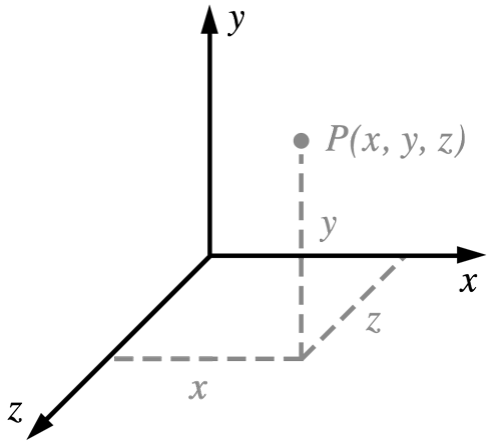


The Cartesian plane has two perpendicular axes (commonly labeled x and y). When we want to speak of things that are perpendicular, we'll call them *orthogonal*. Points are identified by specifying their extent

along each axis. For example, the point $P(3, 5)$ is 3 units to the right along the x axis and 5 units up along the y axis, relative to the origin. The origin is identified as the point having coordinates (0, 0).
We turn the Cartesian plane into a 3D coordinate space by adding another orthogonal axis, which we'll name z.

Handedness

Handedness refers to the orientation of the z-axis in a given 3D space. If the z-axis conforms to the so-called right-hand rule, $x \times y = z$, then the space is said to be *right-handed*. Alternatively, if the z-axis points in the other direction, $x \times y = -z$, the space is left-handed.



The term "handedness" comes from a mnemonic for remembering which way the third axis points. Identify your thumb as the x axis and your pointer finger as the y axis. Hold up either hand and make an 'L' with these two fingers. Now, when you extend your middle finger perpendicular to both of the other fingers, it indicates the direction of the positive z axis: the middle finger of your right hand will point toward you, and the middle finger of your left hand will point away.

The choice is arbitrary, but I choose to work in right-handed spaces when possible. However, we will give Metal our vertices in a 3D space called *clip space*, which is left-handed. As long as we make the switch from right- to left-handed at the correct place in the rendering process, everything works out okay.

Introduction to Transformations

A geometric transformation is a function that maps a point to another point. The most common transformations in computer graphics are translation, rotation, and scaling. In three dimensions, rotation and scaling can be represented as a multiplication of a 3×3 matrix by a 3D point. Unfortunately, translation cannot be represented in this way, but there is a formulation we'll see below that nevertheless allows us to capture all the transformations we wish to perform using matrix multiplication.

First, we'll consider the family of transformations known as *linear transformations*.

Linear Transformations

A linear transformation T must obey these two properties:

$$T(\alpha x) = \alpha T(x)$$
$$T(x + y) = T(x) + T(y)$$

In words, the first condition means that scaling the input before the transformation is the same as scaling the output after the transformation. The second condition means that the transformation of sums is equal to the sum of the transformed inputs.

Identity

The identity transformation maps every point onto itself. It is a matrix with ones along the diagonal and zeros everywhere else:

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Scale

Another common transformation is scaling. Here is a matrix that will scale points up (or down) along each axis:

$$S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

s_x, s_y, and s_z are the scale factors along the x, y, and z axes, respectively. You can see that if $s_x = s_y = s_z = 1$, the matrix is the identity matrix.

Rotation

Rotation in three dimensions is a complex topic. Fortunately, we don't need to understand all the intricacies to use them. First, we consider rotation around the Z axis, then generalize to rotation around any axis.

Rotate Around the Z Axis

Rotation around the Z axis is the simplest to visualize, since points that lie in the X-Y plane stay in the X-Y plane when rotated around the Z axis. Here is a 3×3 matrix that will rotate points about the Z axis:

$$R_z = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

It is important to note that θ is in radians, and a positive angle results in a *counterclockwise* rotation.

Matrices that represent rotation around the X or Y axis can also be formulated, and look very similar.

Rotate Around Any Axis

Suppose we want to rotate around an arbitrary axis. If u is a unit vector representing an axis in space, the following matrix will rotate points around this axis:

$$R_u = \begin{bmatrix} cos\theta + u_x^2(1-cos\theta) & u_x u_y(1-cos\theta) - u_z sin\theta & u_x u_z(1-cos\theta) + u_y sin\theta \\ u_y u_x(1-cos\theta) + u_z sin\theta & cos\theta + u_y^2(1-cos\theta) & u_y u_z(1-cos\theta) - u_x sin\theta \\ u_z u_x(1-cos\theta) - u_y sin\theta & u_z u_y(1-cos\theta) + u_x sin\theta & cos\theta + u_z^2(1-cos\theta) \end{bmatrix}$$

Once again, θ is in radians, and a positive angle results in counterclockwise rotation *when the axis is pointing at you*.

Shear

Shear is a somewhat less commonly used transformation that moves points parallel to an axis. Shearing terms arise in the off-diagonal elements of matrices.

Affine Geometry

If you work through some examples, it will become obvious that rotation, scaling, and shear are all linear transformations, but translation is not. The fact that a transformation is linear is what allows us to write it as a matrix.

Translation

The last transformation we will encounter is a translation, which moves points along a vector in space. One straightforward way to implement this is with vector addition, where we map from one point to another by adding a vector. Consider the formula below, where p is a point and t is the vector we wish to translate by.

$$\begin{bmatrix} p_x \\ p_y \\ p_y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_y \end{bmatrix} = \begin{bmatrix} p_x + t_x \\ p_y + t_y \\ p_y + t_z \end{bmatrix}$$

It is impossible to build a 3×3 matrix that can be multiplied with a point to produce the translation above. However, we can use a handy trick in the fourth dimension to handle all of our transformations in a unified way, starting with translations.

Translation as Shear in 4D

In order to express translation with a matrix, we need to add an extra coordinate ($w = 1$) to our points. Appending this coordinate lets us augment our transformation matrix with an extra row and an extra column representing the translation vector:

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_y \\ 1 \end{bmatrix} = \begin{bmatrix} p_x + t_x \\ p_y + t_y \\ p_y + t_z \\ 1 \end{bmatrix}$$

We aren't violating the assertion that translation is not a linear transformation in 3D here. The matrix above represents a type of linear transformation called *shear*. Again, it's crucial to realize that the above matrix does not represent translation of a 4D point, which would not be a linear transformation. We're exploiting a characteristic of linear transformations in 4D space that *correspond* to translations once we drop back into 3D.

Affine Transformations

The type of transformation we built above has a name: it is an *affine transformation*. An affine transformation is a linear transformation composed with a translation. Perhaps you noticed that the upper 3×3 matrix in the translation matrix was the identity matrix.

In order to create a general affine transformation that represents a rotation, scale, or shear, **and** a translation, we will place this upper 3×3 matrix with the corresponding linear transformation, and place the translation vector in the last column as before. We might write this compactly as a partitioned matrix:

$$\left[\begin{array}{c|c} \mathbf{R} & \vec{t} \\ \hline \vec{0}^T & 1 \end{array}\right]$$

Above, $\mathbf{R}$ is a 3×3 linear transformation, such as a rotation. $\vec{t}$ is the 3×1 column vector containing the translation, and $\vec{0}^T$ is the 1×3 row vector containing all zeroes.

Compositing Transformations

Because of the associative property of matrix multiplication, transformations can be combined by matrix multiplication:

$$(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$$

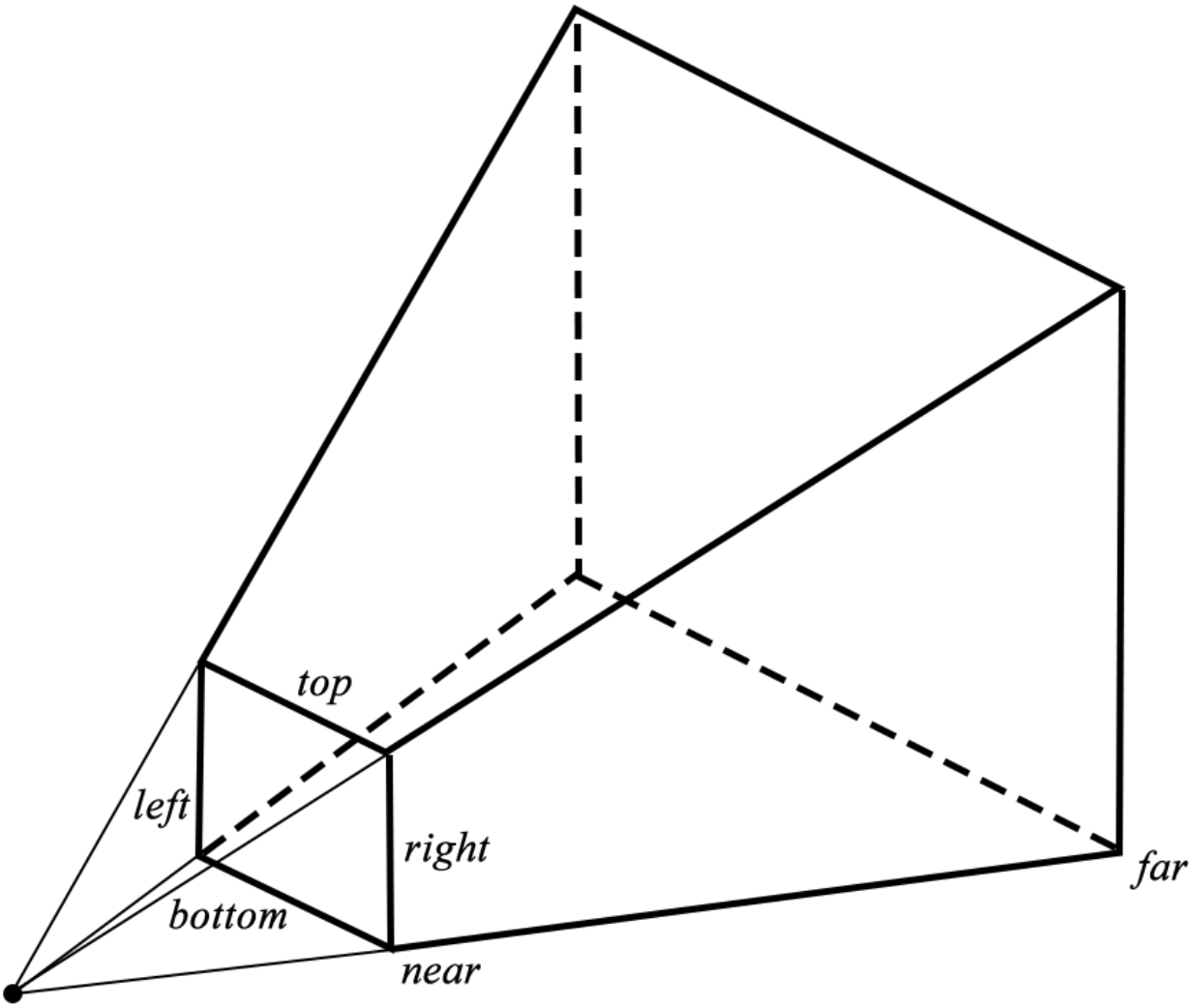
As before, this matrix will be applied with a vector on the right, and therefore the transformations will be applied *right-to-left*. So, if we had a sequence of transformations to apply scaling ($\mathbf{S}$), rotation ($\mathbf{R}$), and translation ($\mathbf{T}$) to an object, in that order, we would compose the transformation as $\mathbf{TRS}$. This transformation first scales the object, then rotates it, then positions it in space, with each transformation occurring relative to the origin of the previous transform's local coordinate space.

Projection

Since our ultimate aim when programming 3D graphics is to produce a 2D picture, we need a way to squash the third dimension down while creating the illusion of perspective. This is achieved through the use of a *perspective projection* transformation. The projection transform is applied in the vertex shader.

The View Frustum

You can think of the portion of the world that is visible through a virtual camera as a pyramid with the top chopped off. This shape is called a *frustum*.



The view frustum, with clipping planes labeled

By choosing an aspect ratio (the ratio between the width and height of the viewport) and a field of view, we implicitly determine the *clipping planes* that make up the sides of the view frustum.
The pyramidal shape of the frustum is a natural outcome of our demand for perspective. The perspective projection scales points relative to their distance from the virtual camera, which in turn causes the sloped sides of the view frustum to become parallel as it undergoes this transformation. Points far away are squeezed closer to the axis of view, which causes the phenomenon of *foreshortening*, an important depth cue.

Clip Space and Normalized Device Coordinates

We need to make sure that the points produced by our projection transform are in the coordinate space it expects. Everything that is to be visible on the screen must be scaled down into a box that ranges from -1 to 1 in x, -1 to 1 in y, and 0 to 1 in z. This coordinate system is called *clip space*, and it's where the hardware determines if triangles are completely visible, partially visible, or completely invisible. The edges of triangles

that are partially visible are *clipped* against the planes of clip space. This clipping process may turn a triangle into a polygon, which is then re-triangulated to produce the geometry that gets fed to the fragment shader.

The Projection Matrix

Now that we know where we're starting from (the view frustum) and where we're going (the clip space volume), we can construct a matrix that appropriately scales points from one to the other:

$$\mathbf{P}_{\mathbf{perspective}} = \begin{bmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{-f}{f-n} & \frac{-fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

In this matrix, r, t, n, and f refer to the clipping plane offsets illustrated in the figure of the view frustum: right, top, near, and far planes, respectively.

The construction of this matrix is beyond the scope of this article, but you can view a very well-illustrated derivation at Song Ho Ahn's site. Note that OpenGL uses a different clip-space than Metal: OpenGL's z axis runs from -1 to 1, rather than 0 to 1. This accounts for the slightly different form of the above matrix, as compared to the matrix derived in the reference.

Conventions

The SIMD Library

With iOS 8 and OS X Yosemite, Apple introduced a library called `simd` that implements SIMD (single-instruction, multiple-data) arithmetic for scalars, vectors, and matrices. This library provides exactly the same types and operations that the Metal shading language does, which means that there's an opportunity for reuse between your Objective-C code and Metal shaders. I use SIMD types and operations whenever possible. For more information, view the WWDC 2014 session, *What's New in the Accelerate Framework*.

Matrix Storage

There are two options when storing two-dimensional matrices in memory: row-major and column-major. With row-major storage, the rows of the matrix are written contiguously to memory, while with column-major storage, the columns are written contiguously. Because the SIMD library adopts column-major storage, I will do so as well.

When working with SIMD matrix types in Objective-C code, you cannot use two-dimensional array syntax to access elements. Instead, you must first index into the `columns` array, then index into the returned array by

specifying the row element. For example, here's how you would get the element in the first row, third column:

```
float element = matrix.columns[2][0];
```

When working with matrices in shader code, you can use the more compact syntax `matrix[2][0]`, but note that the subscript order is still *column, row* rather than the more typical *row, column* as you'd expect in C.

← Previous Post

Next Post →

16 thoughts on "Linear Algebra for Graphics Programming"

AOAKENFO
SEPTEMBER 30, 2014 AT 10:40 AM

This is really the meat of graphics programming isn't it? But it's skimmed over in this post. I have yet to read a treatise that really cements this knowledge in the mind of a newcomer. I suppose it's like the difference between looking at code to generate a sine wave:
http://music.columbia.edu/cmc/musicandcomputers/popups/chapter4/xbit_4_1.php

and looking at this:
http://global.oup.com/us/companion.websites/fdscontent/uscompanion/us/static/companion.websites/9780199922963/images/SineAnimation.gif

Perhaps an interactive .playground where users could play with the effects of vectors and matrices would be helpful?

Reply

WARRENM
SEPTEMBER 30, 2014 AT 4:45 PM

Ash, I couldn't agree more. I definitely hope to flesh out this content much further in the future. My original intent with this site was to write content that would help intermediate graphics programmers move to Metal. I've found myself writing a lot of supporting material in an effort to give context to the discussion of Metal's feature set.

But you're completely right; the math doesn't deserve short shrift.

Reply

MATTHEW LINTLOP
JANUARY 16, 2015 AT 7:53 PM

Hi Warren,

You are an exceptional teacher of Metal keep it up!

Keep up the good work. I also "departed Apple several times ; can Metal do this easily???"
http://m.youtube.com/watch?v=vBcjReIDMQs

Reply

WARRENM
JANUARY 17, 2015 AT 11:46 AM

Looks pretty straightforward. I assume you're using pre-baked lighting textures for the Cornell box, and using a third-party physics library? For manipulating the cubes, you presumably use picking (with a ray cast into the scene) and create a short-lived spring system to generate the drag effect. Are you planning to implement real-time shadows?

Reply

MATTHEW LINTLOP
JANUARY 18, 2015 AT 1:07 AM

Actually I was using another OpenGL ES Engine Called SIO2. It uses Bullet physics. That will be easy to match with Metal because it's just bullet objects that Metal has to catch up to right 😉
Real time shadow math is beyond my educational level, but I ca imagine I plugin FX architecture – shaders & & even compute kernels – would crank with Metal devices for the future…
But I can imagine major improvements in real time 3D graphics across devices with Metal…

Reply

MATTHEW LINTLOP
JANUARY 18, 2015 AT 1:12 AM

And implementing Apple's SKActions with transformations seems some to me. For example: move to destination in 3.0 seconds, dissolve, etc.

Reply

MATTHEW LINTLOP
JANUARY 18, 2015 AT 7:28 PM

By the way...why doesn't the latest iPod touch 5 Generation support Metal? I haven't event been able to run Metal on a real device yet....but I will soon with a new iPad. Cheers!
(and even the Mac OS X Simulator doesn't do Metal???)

Reply

WARRENM
JANUARY 19, 2015 AT 8:39 AM

The latest iPod touch doesn't support Metal because it still has an A5 processor, which is downright ancient compared to last year's A8 processor.

As for Simulator support, yeah, that's disappointing. Seems like it wouldn't be too terribly difficult, but it wasn't enough of a priority for the initial release. Despite the inconvenience, though, GPU performance in the Simulator is radically different from the onboard GPU, which makes on-device testing paramount.

Reply

MATTHEW LINTLOP
JANUARY 21, 2015 AT 6:31 PM

How do you think Metal does compare with SceneKit for small development teams who don;t want to use Unity but are tempted to build their own engines? There's some pretty amazing technology there?

Scenekit does some pretty amazing graphics with much fewer typing than Metal. 😉

Reply

WARRENM
JANUARY 22, 2015 AT 11:52 AM

Writing a renderer, much less an engine, is a major undertaking. I think a small team would be foolish to reinvent the wheel when Unity is so incredibly powerful and comes with support for multiple platforms out of the box. If your game is a commercial venture and not merely a hobby, why reinvent the wheel and simultaneously lock yourself into a set of devices that only comprise 50% of the total number of iOS devices in the wild?

Of course developing with SceneKit is faster than with Metal. SceneKit is expressly designed to be a high-level abstraction layer. It has the same type of limitation as other abstraction layers: when you try to scale it and you hit the wall, you have nowhere to go except down the abstraction stack.

Reply

MATTHEW LINTLP
JANUARY 31, 2015 AT 9:47 PM

Do you thin the folks at Apple will ever use Metal under the hood of SceneKit? Making a 3D Box,, or Pyramid in 1 line of code is very cool….Let Apple do the hard work as usual;}

Reply

WARRENM
JANUARY 31, 2015 AT 9:52 PM

I suppose it’s possible, but given how OpenGL-centric it is (e.g., custom drawing and shading are done with gl functions and GLSL shaders), it would require some changes to make it sufficiently API-agnostic or add Metal as a supported implementation target.

Reply

MATTHEW LINTLOP
FEBRUARY 1, 2015 AT 2:09 PM

I can’t wait for your book to unlock the secrets of future device in 3D. Let’s have a beer in SF & talk about taking over the world n 3D.

Reply

ALEX K.
DECEMBER 23, 2015 AT 2:58 AM

Just wanted to let you know that the projection matrix you state is in fact incorrect! I've fallen into the same trap of just using the OpenGL projection matrix but Metal uses a different normalized device space (NDC, 2x2x2 for Opengl and 2x2x1 for Metal) and therefore you are effectively cutting off the projection! I've written a detailed explanation in my blog, hope it helps:
http://blog.athenstean.com/post/135771439196/from-opengl-to-metal-the-projection-matrix

Reply

WARREN MOORE
OCTOBER 23, 2016 AT 3:36 PM

Thanks for the note, Alex, and sorry it took me so long to reply. You're absolutely correct. I've fixed the matrix above and hope to update the sample code that uses the incorrect projection math before long.

Reply

Pingback: Using Matrix Transformations in 3D Printing – Linear Algebra Applications S19

Leave a Comment

Your email address will not be published. Required fields are marked *

Type here..

Name*

Email*

Website

Notify me of follow-up comments by email.

Notify me of new posts by email.

Post Comment »

This site uses Akismet to reduce spam. Learn how your comment data is processed.

Search …

Recent Posts

Using Basis Universal Texture Compression with Metal

What's New in Metal (2019)

Vertex Data and Vertex Descriptors

Picking and Hit-Testing in Metal

Rendering 3D Text with Core Text and libtess2

Rendering Physically-Based ModelIO Materials

Writing a Modern Metal App from Scratch: Part 2

Writing a Modern Metal App from Scratch: Part 1

The Metal by Example Book is Now Available!

Metal Performance Shaders in Swift

First Look at MetalKit

Compressed Texture Formats in Metal

Rendering Text in Metal with Signed-Distance Fields

Translucency and Transparency in Metal

Video: An Introduction to 3D Graphics with Metal in Swift

Archives

Select Month