MATH 2231: Applications of Linear Algebra to Real-Time 3D Computer Graphics

Jonathan Edelman

December 13, 2019

Contents

1 Intro

Unlike many other applications of linear algebra, computer graphics are uniquely familiar in a certain respect. Video games, glossy visual effects in our favorite movies and shows, even the pie charts in PowerPoint: 3D computer graphics are quite literally a very visible, if small, part of our everyday life. However, like many other applications of linear algebra, the math at the heart of this technology isn't readily apparent to most users—nor to most mathematicians. This paper aims to explore the world of computer graphics from the perspective of the curious individual: while concrete examples (with a little code) will be provided, and specific technologies discussed, the goal is more to show the wide uses of linear algebra, rather than to act as a textbook on linear algebra with a creative framing or as an

instructional guide on how to write computer graphics. As such, this paper will assume a familiarity with linear algebra, but little to no familiarity with programming or computer graphics. Additionally, this paper will primarily focus on how to represent and efficiently operate three-dimensional space using math and modern computers, rather than the specific applications of linear algebra to more advanced topics within computer graphics. As such, this paper generally maintains a secondary focus on real-time rendering techniques, with lesser exploration of realistic and physically-based renderers. Further, this paper will take the general standpoint of seeking clarity over generality: while many of these ideas are more generally applicable, it is the opinion of the author that reasoning about specifics (and the most common perspective) is generally more digestible and therefore useful for a first-time reader. If you wish to explore these topics in more depth, it is recommended that you read the footnotes/citations (in particular *Real-Time Rendering* and *3-D Computer Graphics: A Mathematical Introduction with OpenGL*) and explore any topics you find interesting.

2 Geometry

2.1 Coordinates

Let's lay out some basic tenants for our world:

- Unambiguous: We want our representation for our world to be consistent and well defined, as well as our operations on parts of the world to be similarly well defined (e.g. the coordinates that represent an object can't be different depending on the context in which it's used.) (N.b.: From a programming perspective, this means we'll want to keep as much of our program as we can as pure functions. This will help us in breaking down, debugging, and optimizing the various problems we'll encounter along the way.)
- Performant: While the exact balance between speed and accuracy/quality will vary application to application, we probably do care about performance to some degree. We'll come back to this more later.
- Flexible: We need a system that we can tune to meet the needs of our application.

These will be guiding principals, not only in choosing our representation for the world, but throughout this project. With that in mind, let's get started!

The first thing we want to do is define our space. This, hopefully, is an intuitive process—both thanks to your understanding of linear algebra and the real world. Here, we'll be representing a three dimensional Euclidean world, much like our own. We'll call this our **worldspace**. (N.b.: While it might seem like this and two dimensional Euclidean worlds are the only reasonable options for a world, higher dimensional Euclidean space isn't too challenging to represent, and three dimensional non-Euclidean space can be represented using some other techniques.) However, we're not representing our worldspace in the same space; instead, we want to represent it in another two dimensional space so we can display it on a screen. We'll call this the **screenspace**.

Next, we need to define a coordinate system for the space. We'll start with the screenspace, since it's the simpler of the two. Since we want to represent a "normal" view of the world, it makes sense to use a Cartesian coordinate system. By convention, the screenspace is defined with an x-axis and a y-axis, with $(0, 0)$ located in the top left of the viewport (i.e. the portion of the screenspace seen by the user), positive x to the right, and positive y down. This is the same as the "standard" Cartesian system, but with the y-axis flipped. To get our worldspace, we'll simply add another axis, the z-axis, to our existing coordinate system. However, there's a bit of a snag: which way should we point our z-axis (i.e. should positive Z be facing into the screen or out of the screen)? One common way of visualizing this is the concept of "handedness", which defines this third vector as the natural direction of the middle finger of a hand once the first and second vectors are oriented properly:

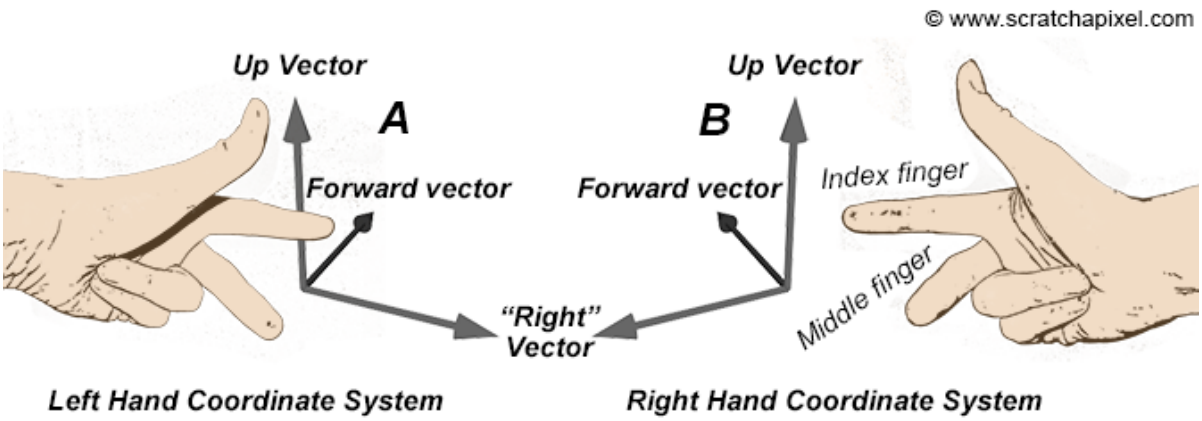


Figure 1: Coordinate systems. [4]

Note that for our purposes, the forward vector would be pointing to the right, and the direction vector into our out of the screen. Again, the hand must be oriented properly before handedness rules can by applied. By convention, many 3D graphics/CAD systems use a right-hand system by default (Blender, Fusion, Maya, OpenGL, Vulkan). However, this is not universally true, and several systems (DirectX, Renderman) use a left-handed system. However, many of the aforementioned actually support both left and right handed systems, assuming the user configures them as such.

It's important to remember that these coordinate systems are ultimately arbitrary representations. Outside of picking a Cartesian system, we haven't really conveyed any information in the choosing of our system. In fact, our definitions of up, down, left, and right are given only by the orientation of the screen itself; what we actually choose to represent is left up to us. While a world with a floor plane and a sky is intuitive, it is far from the only option. Just because it's called the y-axis doesn't mean it needs to point up (or down.) However, this also doesn't mean the systems are ambiguous; we've got enough structure. In addition, we'll be defining more behavior as we continue.

2.2 Representing points

The simplest thing to represent in any space is a single point. So far, we've been throwing around the words "vector", "coordinate", and "axis" without too much care. While these terms hopefully made sense in the context in which they were used, we're trying to be somewhat careful with out definitions. For the purposes of this article, we want to be able to represent objects defined by points within $\mathbb{R}^3$. Since we've defined our coordinate system, we now have an orthogonal basis for our space. Using unit vectors for each axis gives an orthonormal basis which we can conveniently use to define any point in our space. Following standard convention for a Cartesian coordinate system, we'll define any point within $\mathbb{R}^3$ using a vector defined along this basis, where the vector points to the location of the point in space:

$$\text{Point P} = \vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Note here that we've chosen to represent a vector in the "conventional" sense, i.e. as a 1xn matrix. However, we could also choose to represent such a vector as an nx1 matrix instead:

$$\text{Point P} = \vec{v} = \begin{pmatrix} x & y & z \end{pmatrix}$$

These are respectively called **"column-major"** and **"row-major"** order. While column-major is the far more common notation in mathematics, both are used in the world of computer graphics (OpenGL

and Blender use column-major, while DirectX and Maya use row-major.) To understand why, let's revisit matrix multiplication!

For reasons that will become clear in the following section, we'll need to multiply the aforementioned vectors by 3×3 matrices very frequently. In a column-major system, that looks like this:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \cdot a + x \cdot b + x \cdot c \\ y \cdot d + y \cdot e + y \cdot f \\ z \cdot g + z \cdot h + z \cdot i \end{pmatrix}$$

In a row-major system, that looks like this:

$$\begin{pmatrix} x & y & z \end{pmatrix} \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{pmatrix} x \cdot a + x \cdot d + x \cdot g & y \cdot b + y \cdot e + y \cdot h & z \cdot c + z \cdot f + z \cdot i \end{pmatrix}$$

Besides the fact that the column-major version is obviously prettier, there is a more important difference. However, it's not immediately apparent if you just look at the math. To understand what's going on, we need to talk about computers.

Generally speaking, modern computers contain a type of storage called "random access memory" (RAM.) Though it's generally neither the fastest nor slowest type of storage typically present in a computer, it is large enough and fast enough that most programs can be thought of as operating in RAM. This RAM is then allocated in continuous, linear "blocks" to individual programs, which can then choose how they utilize said memory.

Returning to the matrix described earlier, we need to encode a 3×3 matrix in linear chunk of memory. Reusing the terms from earlier (confusing, I know), we can either encode it in column-major or row-major order:[1]

Row-major	a	b	c	d	e	f	g	h	i
Column-major	a	d	g	b	e	h	c	f	i

Thus, depending on our encoding scheme for a matrix, the order in which values are naturally read changes. For performance reasons, preserving this order is a good thing; the more we can do access values in-order instead of out-of-order, the better.[2] Returning to the matrix multiplication shown earlier, we can see that the order in which we need to access values of the original 3×3 matrix to calculate the new value is different between column-major and row-major multiplication. In column-major multiplication, the access order goes $a \to b \to c \to d \to e \to f \to g \to h \to i$. In row-major multiplication, the access order goes $a \to d \to g \to b \to e \to h \to c \to f \to i$. Familiar, right?

The takeaway here is that if we're storing matrices in row-major order, performing column-major multiplication is generally better, and if we're storing matrices in column-major order, performing row-major multiplication is generally better. Though the implementation for storing matrices varies depending on the technologies used, we'll be using column major notation for vectors and multiplication, as it looks prettier and row-major matrix storage is both more common and the default implementation within the library we'll be using for our code.[3]

[1] There are also other ways of storing matrices (or, more generally, n-dimensional arrays); however, this is the most basic and intuative, and in general, the most common within the world of performant graphics. Iliffe vectors are used by a number of languages since it better optimizes for on-die cache.[5] Several alternative formats are better optimized for the space complexity of sparse matrices.[6] However, these are all outside of the scope of this project.

[2] As the name suggests, different parts of RAM can be accessed and modified in any order while maintaining the same performance. However, while the access time for any given portion of the RAM itself may not be any different, because of things like caching and pointer arithmetic, continuous access generally leads to superior performance.

[3] Though Go does include 2D arrays, the Gonum library doesn't make use of them, and instead uses a row-major format with a single input array.[8]

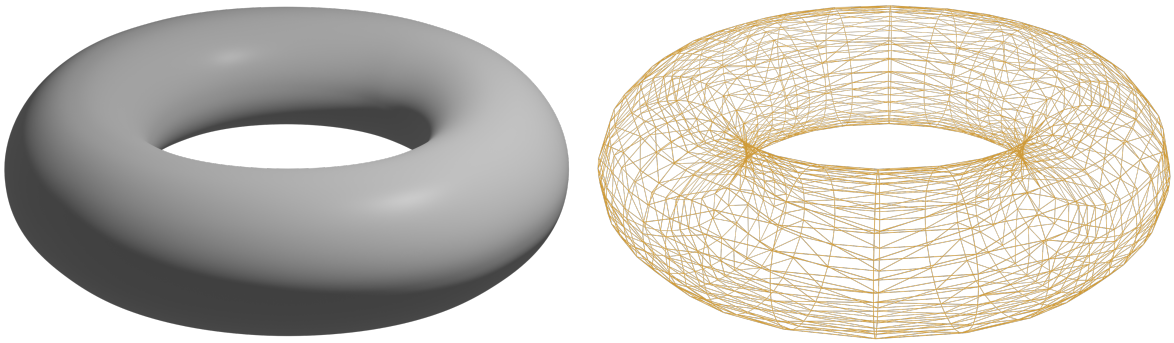
2.3 Polygons

Defining points in space is all well and good, but we want to represent objects like those in our world. Objects in the real world aren't just single points. While representing objects as just a massive collection of points—known as a point cloud—is actually a valid and frequently used technique, especially for scanned data or other digitized representations of real life objects, it's not a particularly easy format to work with from a general 3D modeling and design perspective. One approach might be to translate familiar algebraic concepts to three-dimensional space. If we can represent objects in two-dimensional space using equations, how about three-dimensional space? In fact, this is a common methodology for representing objects where the exact form of the object is important—in particular, engineering and manufacturing applications, and the associated computer-aided design software.[4] This representation offers the distinct advantage of being able to very precisely and accurately encode complex objects. However, this comes at the cost of being relatively computational expensive to operate on. So, since our main interest is in the looks of a surface, we'll take a slightly simpler approach.

To start, let's revisit our concept of what an object is. So far, I've (again) been using this term in a relatively poorly defined manner. Moving forward, let's consider an object to be defined by its surface. In order to approximate this surface, we can stitch together a series of convex polygons to form what's known as a **"polygon mesh."** Compared to our previous approach, this is much simpler to operate on. Further, since each polygon can very easily be defined by a series of points, many operations on the entire mesh can be easily done by simply operating on each point within the mesh. This will be important shortly. Now that we've chosen polygons, the next logical question is what polygons should we use? While we could allow any arbitrary polygon, this introduces a lot of operational complexity without much benefit from a 3D graphics perspective. Instead, we'll use the simplest of polygons: triangles ("tri"s.) This gives us what's known as a triangle mesh. Using triangles presents some distinct benefits: since every vertex connects to every other vertex we don't have to store any connecting information, and since any three points in $\mathbb{R}^3$ can lie on a single plane, any three points form a valid triangle.[5] For these reasons, modern real-time graphics pipelines also use triangles and triangle meshes almost exclusively.

[4] This leads into a realm of modeling and math that relies heavily on splines—in particular, a form of splines known as non-uniform rational basis splies (NURBS.) However, their math and application is outside the scope of this paper. Often, this also ties in closely with a distinct but related concept known as constructive solid geometry (CSG), though again, that is outside the scope of this paper.

[5] While quadrilaterals are commonly used for 3D modelling and in the VFX space, as they fit align more naturally with other modeling techniques and operations such as subdivision and decimation generally act better on them, we're focusing on rendering.[17]



(a) (b)

Figure 2: Torus defined two ways: (a) using precise curves and (b) a triangulated mesh

2.4 Code

Now that we've got a good idea of how we want to represent our world and the geometry in it, let's start drawing up some code!

Some things before we start: **this code is for example purposes only, and should not be used as direct reference.** There are several reasons for this. First and most importantly, no practical code should be implemented this way. The modern stack for 3D graphical programs relies on a complex interplay of several layers of software and hardware—none of which is the focus here. As such, none of the code here will make use of said stack, and will instead do everything in software. This is a horrible approach as far as performance is concerned, but makes understanding things significantly easier. While there will be performance-oriented optimizations in the code, it is not the first focus of this program. Second, this code is intended to help clarify, not operate as optimal, functional code. The purpose here is not to be "correct." In fact, much of the code needed to create a functional renderer will not be included here, both (again) for clarity's sake and for brevity. Third, because the focus of this code is relatively limited in scope, the potential for extensability is somewhat compromised, again in the hopes of improving clarity.

The following code is written in a language known as Go. It's a relatively new language, first released in 2009, but (largely thanks to its backing by Google) has gained enough popularity and support in that time to be sufficiently mature to work with for such a project. It also has several features that make it nice for this application: it's compiled, giving us enough performance overhead to actually run our code, yet has memory safety and garbage collection, meaning we won't have to do much memory management; it is statically typed, but has type inference, making code shorter and generally easier to read. However, most importantly, I already knew it.

If you don't know Go, don't worry too much; in general, the code should be fairly easy to read (I've previously described it as C with the prose of Python and standard lib/memory management of Java.) If all that means nothing to you, don't worry; again, the idea here is that it's a supporting component, not the main idea.

Right. On to the code. Let's start by making our world, using those constraints from earlier.

```
1 package main
```

```
2
3 type Scene struct {}
```

scene.go

Hm. That's not terribly interesting—besides that our constraints from before become assumptions that we don't need to declare in our code. Our scene also isn't going to be terribly interesting without any objects, so let's add those. Remember that we're using a triangle mesh to represent our objects and vectors to represent points in space.

It's at this point that it's necessary to introduce our representation for matrices. We'll be using the Gonum library—mostly just so we don't have to deal with our own types (we'll still do most of the rest on our own.)

```
1  package objects
2
3  import "gonum.org/v1/gonum/mat"
4
5  type Triangle struct {
6      // Representing each vertex as a vector (column) in a dense 3x3
       matrix
7      RawVertices [3]*mat.Dense
8  }
9
10 type VertexObject struct {
11     // Can have as many faces as we want
12     Faces []*Triangle
13 }
```

objects/vert_object.go

```
1  package main
2
3  type Scene struct {
4      objects []objects.VertexObject
5  }
```

scene.go

Notice that we're using pointers for the triangles in our objects and the vertices of our triangles. This second point in particular is important, as it allows us to define multiple triangles in the mesh using the same points—something that comes up very frequently in real meshes, since continuous surfaces will use many contiguous triangles.[6]

Since we're not actually generating the meshes ourselves, that's it! Amazing how simple it is, right? Sure, there's always more code behind the scenes, but the actual core is right there. That's part of the beauty of this representation: **it integrates extremely cleanly with how computers naturally operate.**

[6] For good reason, this also aligns very, very closely with the common Wavefront OBJ 3D file format. This is how we'll be loading in models to actually use in our program. However, the syntax for how we do that is really just an exercise in text parsing, so we'll skip it here.

3 Spatial transformations

3.1 Intro

Up until this point, our use of vectors to to represent points in space has appeared to be a largely arbitrary one. Sure, they correspond well to chunks of memory on a computer, but at that point, why not just use individual variables or groups of variables?[7] The reason is transformations.

Transformations and their associated transformation matrices are the true building block for 3D graphics. The more we can do as a transformation, the more we can represent as a matrix multiplication rather than any other kind of operation, the better. To understand why, let's revisit matrix multiplication—more specifically, its associative property. Recall that for matrix multiplication, $A(BC) = (AB)C$—or, since we're using column-major representation for our vectors and therefore post-order multiplication, $(AB...Z)\vec{v} = (A(B...(Z\vec{v})))$. This means that to apply a series of transformations to a group of vertices, rather than multiplying each vertex by each of the transformations, we can premultiply all the transformations a single time, cache that matrix, then multiply that matrix by each of the vectors. This offers a huge performance benefit: rather than our execution time being a function of both our vertices and transformations, it can be a linear time function of the number of vertices. Given how expensive matrix multiplication is[8] and the thousands or even millions of vertices present in a scene, this is a huge savings. Of course, not every object will have exactly the same transformations applied to it, but since large numbers of vertices typically share transformations, the savings are still quite significant.

For these reasons, transformations crop up in many different places in what may seem like unrelated use cases. Modeling a complex object? Use transformations to represent changes. Moving the character around the world? Transformations. Rendering the scene? Transformations.

3.2 Linearity and affinity

In order to manipulate meshes, we need some format for building transformation matrices that we can use to represent the transformations we want to do. Typically, transformations are introduced in a context of linear transformations. Recall the properties of a linear transformation:

1. Closure under addition, i.e. $T(\vec{x} + \vec{y}) = T(\vec{x}) + T(\vec{y})$
2. Closure under scalar multiplication, i.e. $T(k\vec{x}) = kT(\vec{x})$

These properties are useful in many contexts, but they also mean that **there's no way to represent a translation as a linear transformation** since it doesn't leave the origin fixed, i.e. is not closed under scalar addition. If we were to represent translations as a separated concept with their own math, we lose the core benefit of using matrices in the first place, since we can no longer represent our translations as the combination of several matrices. Instead, we need a new form to represent these transformations as matrices.

The solution here is quite actually quite simple: combine the linear transformation and translation into a new transformation that perform a linear transformation then a translation, known as an **affine transformation**. In other words, a transformation is affine iff $A(x) = T(x) + \vec{t}$. We can represent this transformation in a 4×4 matrix, with vectors notated in "homogeneous notation" $\vec{v} = (v_x \ v_y \ v_z \ w)^T$, where the w represents the homogeneous coordinate.

[7] In the most general sense, this will continue to be true; "representing" means nothing more than just thinking about from a certain perspective. On a technical level, we are the same thing, just with some nice wrappers to help us better conceptualize these ideas. The same goes for any representation and application of the concepts of linear algebra.

[8] There are sub-cubic implementations, but since we're using small matrices, it generally makes sense to stick to the cubic algorithm to minimize overhead.

$$\begin{bmatrix} a & b & c & t_x \\ d & e & f & t_y \\ g & h & i & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

By extension, we can convert between homogeneous coordinates and Cartesian coordinates by dividing each coordinate by the homogenous coordinate, and vice versa by simply setting 1 as the homogenous coordinate:

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} \frac{x}{w} \\ \frac{y}{w} \\ \frac{z}{w} \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

The need for this w term will become more readily apparent later.

As it turns out, this is enough to represent all the major transformations we'll need. Further, this also preserves several important properties:

1. Collinearity, i.e. points on a line remain on a line
2. Proportionality, i.e. the ratio between distances is preserved
3. Parallelism, i.e. parallel lines remain parallel

Angles and lengths meanwhile are not preserved.

With that out of the way, let's get started! The following is a summary of the transformations we'll be covering; use it as a reference for notation if you so desire.

Notation	Name	Characteristics
$S(\vec{s})$	scaling	Scales along one of the basis vectors. Affine, linear transformation only.
$T(\vec{t})$	translation	Moves a point. Affine, translation only.
R	rotation	Rotates (manner varies on context.) Orthogonal and affine, linear transformation only.
$P_o(s)$	orthographic projection	Parallel projection. Affine. Linear transformation and translation.
$P_p(s)$	perspective projection	Projection with perspective. Linear transformation and translation.

[16]

3.3 Code: Framework

Let's implement a corresponding interface.

```go
package transformations

import "gonum.org/v1/gonum/mat"

type Transformation interface {
  GetTransformation() *mat.Dense
}
```

objects/transformations/transformation_interface.go

Again, our interface is quite simplistic; again, this corresponds with the relative simplicity of the core idea. At their heart transformations are really just matrices, and to a certain extent, that's all we really need to represent a transformation. However, it's still often useful to add additional functionality, helpers, and encapsulation to allow for more natural interfacing with our transformations (from a user perspective.) [9]

3.4 Scaling

As introduced in a standard linear algebra course, scaling transformations are quite simple. Here's no different: since there's no need to perform any translations, the affine scaling transformation is simply a linear transformation with no translation. Our corresponding matrix is equally simple:

$$S(\vec{s}) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

One trick we do have though: rather than blocking out a large chunk of memory for each element in the matrix, since we know it will be a diagonal matrix (that is only a few elements can change), we can simply store those elements rather than every entry in the matrix.

```go
package transformations

import "gonum.org/v1/gonum/mat"

type Scaling struct {
  X, Y, Z float64
}

func (scaling *Scaling) GetTransformation() *mat.Dense {
  return mat.NewDense(4, 4, []float64{
    scaling.X, 0, 0, 0,
    0, scaling.Y, 0, 0,
    0, 0, scaling.Z, 0,
    0, 0, 0, 1,
  })
}
```

objects/transformations/scaling.go

[9] Go isn't object oriented, yet it retains methods and interfaces (although methods are only functions with a special receiver argument.) In Go, interfaces are implemented implicitly, i.e. any methods that happen to conform to the interface will be inferred to implement the interface. This is why the code for the following transformations doesn't contain specific keywords relating to the inheritance.

Note that we're still returning a dense matrix rather than an alternative sparse matrix representation since we still have to multiply it with other matrices and, in general, the small, temporary increase in space complexity is better than having to special case a different type of matrix multiplication.

3.5 Translation

Translations are similarly simple to represent. Unsurprisingly, the translation matrix is made by simply using the translation portion of the translation matrix, while keeping the linear transformation portion the same using the identity:

$$T(t) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And again, we'll use the same trick for representing the transformation:

```go
package transformations

import "gonum.org/v1/gonum/mat"

type Translation struct {
  X, Y, Z float64
}

func (translation *Translation) GetTransformation() *mat.Dense {
  return mat.NewDense(4, 4, []float64{
    1, 0, 0, translation.X,
    0, 1, 0, translation.Y,
    0, 0, 1, translation.Z,
    0, 0, 0, 1,
  })
}
```

objects/transformations/translation.go

3.6 Rotation

Recall that in two dimensional space, for the definition of a rotation as a linear transform described by rotating by a given angle θ about the origin, the following matrix corresponds to the given transformation:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

This definition of a rotation works quite well when all vectors are centered on the origin, since any rotation occurs both about the world's origin and any given vector's origin.

However, once we move from two dimensions to three dimensions, this representation starts to run into trouble. With the addition of an extra dimension, we gain two new two-dimensional subspaces that we can perform rotation on. Initially, this may not seem like a huge deal: simply add two new angles, ϕ and ψ, find a corresponding transformation matrix, and move on. This definition quickly starts to run into trouble though.[11]

First, let's consider the need to compose several transformations together, all of which happen to be the same type of transformation. While it would be possible to simply perform matrix multiplication with each of the corresponding transformation matrices, this isn't optimal. Remember, matrix multiplication isn't cheap, so the less we can do it, the better.[10] Instead, we can make use of our knowledge that each transformation is the same to reduce the number of calculations:

$$S(\vec{s_1}) \cdot S(\vec{s_2}) = S(\vec{s_1}\vec{s_2})$$
$$T(\vec{t_1}) \cdot T(\vec{t_2}) = T(\vec{t_1} + \vec{t_2})$$

And, in our code:[11]

```
package transformations

import "gonum.org/v1/gonum/mat"

type Scaling struct {
  X, Y, Z float64
}

...

func (scaling *Scaling) Multiply(other *Scaling) *Scaling {
  return &Scaling{
    X: scaling.X * other.X,
    Y: scaling.Y * other.Y,
    Z: scaling.Z * other.Z,
  }
}
```

objects/transformations/scaling.go

```
package transformations

import "gonum.org/v1/gonum/mat"

type Translation struct {
  X, Y, Z float64
}

...

func (translation *Translation) Multiply(other *Translation) *
    Translation {
  return &Translation{
```

[10] While we will be discussing this technique, the working implementation of the code does not utilize it, as demonstration only uses a few simple transformations and the additional complexity and overhead doesn't make up for the potential reduction in time complexity. However, these techniques are important for larger-scale projects.

[11] Note that this is a simplified version that doesn't handle the abstraction you might actually want when implementing such code

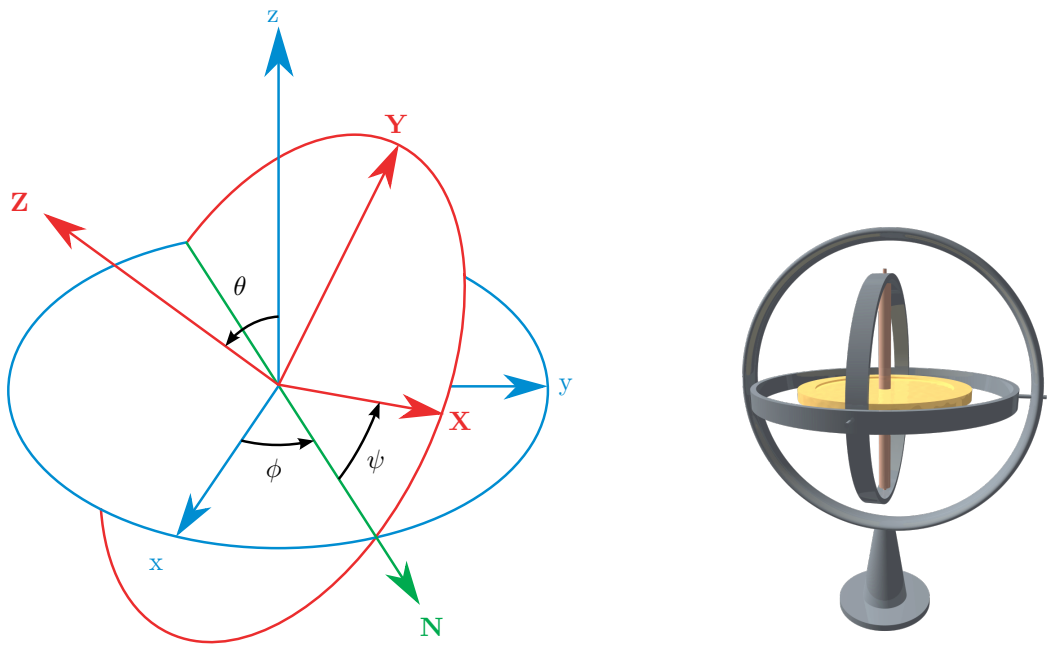
```
13      X: translation.X + other.X,
14      Y: translation.Y + other.Y,
15      Z: translation.Z + other.Z,
16    }
17  }
```

objects/transformations/translation.go

However, doing the same with the aforementioned representation of a rotation doesn't work, since each new rotation relies on the full result of the previous rotations. In other words, the multiplication of rotations in this form is both non-commutative and lacks "simple" representation (even if we define a consistent order in which ϕ, θ, and ψ are applied since out-of-order rotations can't easily be re-ordered.) Similarly, while we could describe our rotation using the rotation matrix itself to describe the mapping of each of our basis vectors, this is to be avoided for the same reasons.

Second, moving between orientations—i.e. interpolating intermediate steps in a rotation—is extremely difficult with this representation for all rotations that cannot be accomplished by rotating around a single axis. Even if strictly speaking this doesn't impact the speed of our rendering pipeline, the need to perform such interpolation for things such as animations is quite frequent; as such, having a representation that aligns well with this need is useful both for computational purposes and for our own reasoning.

One solution is to use what are known as Euler angles. Rather than rely on the global Cartesian coordinate system, Euler angles work by defining a local coordinate system and reference frame to describe orientation. Similar to our earlier definition, Euler angles rely on three rotational angles generally notated with the same symbols ϕ, θ, and ψ (sometimes α, β, and γ.) Unlike our previous definition however, each angle measures the rotation relative to the results of the other rotations (or, in the case of the first angle, relative to a default direction vector.) This is commonly visualized using a three-axis gimbal, where the axes of rotation are represented by three nested rings, connected along orthogonal axes around which they can freely pivot.

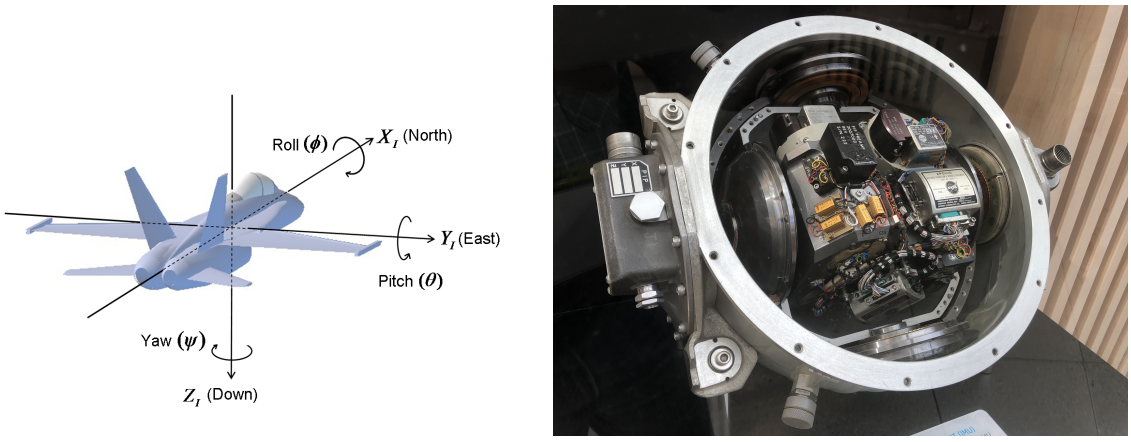


(a) (b)

Figure 3: Euler angle representations [1] [12]

Note that notation can vary, as the symbols used and order in which these transformations are performed is arbitrary, and must be specified. Typically these follow the right-hand rule with naturally ordered axes and variables ($X \to Y \to Z$, $\phi \to \theta \to \psi$), but any order can be chosen.

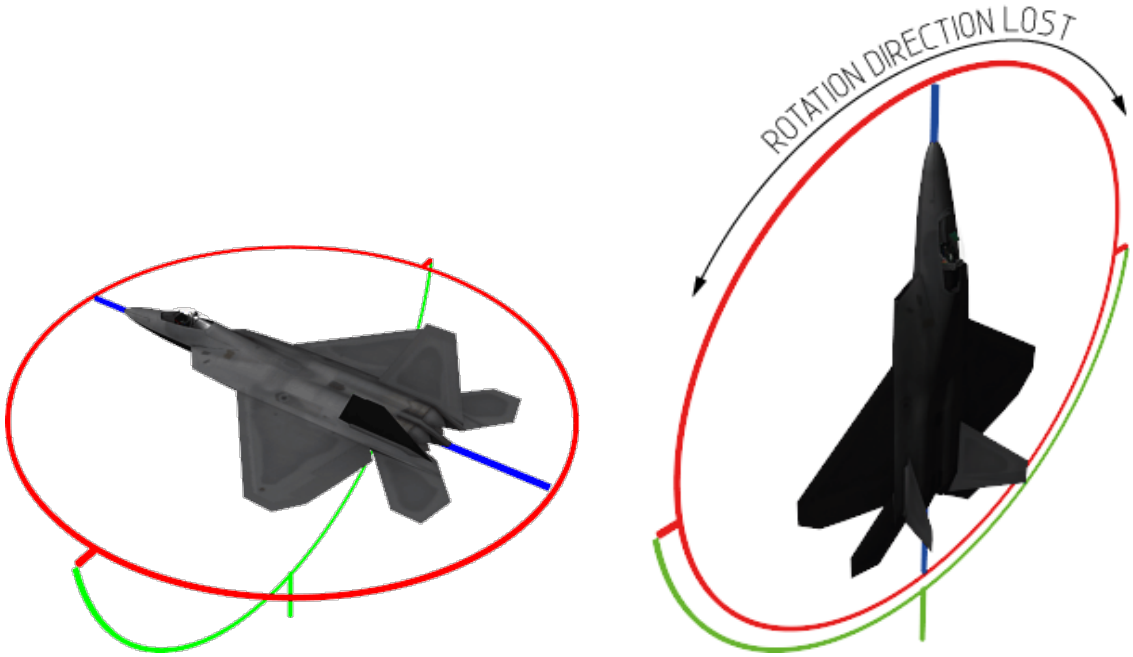
Though this may sound slightly confusing, Euler angles are probably the most intuitive system for describing rotation available. As such, this representation is frequently used in applications such as aviation, often with the rotations named "roll", "pitch", and "yaw". This framework can even be seen in the mechanical framework of early inertial measurement units:



(a) (b)

Figure 4: (a) Plane showing Euler angles [7] (b) Apollo IMU [14]

So long as each of the individual angle transformations is applied in the same order each time, it's now trivial to combine rotations. Further, smooth rotation is generally quite simple, since it's possible to just align one of the rotation vectors using the others, then rotate along that vector. This solves both problems from earlier—mostly. This representation suffers from what's known as **gimbal lock.** When two or more of the axes are aligned, the system doesn't have enough freedom to rotate in all available dimensions.



(a) (b)

Figure 5: (a) Gimbal in a normal state (b) Gimbal in a locked state [10]

This can be a somewhat hard concept to visualize in a static context; if you're having trouble, I recommend you watch this animation. While it is possible to "get out" of gimbal lock before performing any necessary transformations, doing so is nontrivial computationally and is fairly difficult to implement. Thus, despite their relative ease of use, Euler angles are generally not desirable for use within 3D graphics.

Thus we come to our next representation: axis-angle. The idea here is that instead of describing our rotation as a series of simpler rotations, we can instead describe it as a single rotation around an arbitrary axis. This immediately solves the gimbal lock problem, as well as the interpolation issue, since interpolating a rotation now simply involves interpolating a single angle. Compositing rotations about the same axis is also quite easy. However, it's still difficult to composite rotations about different axes.

To make this possible, we need to introduce the idea of a quaternion—and how to represent rotation with them. Quaternions are a type of complex numbers, with a real component and three imaginary components, notated $a + b\hat{i} + c\hat{j} + d\hat{k}$. Multiplication for quaternions behaves quite similarly to that of complex numbers:

$\times$	1	$\hat{i}$	$\hat{j}$	$\hat{k}$
1	1	$\hat{i}$	$\hat{j}$	$\hat{k}$
$\hat{i}$	$\hat{i}$	-1	$\hat{k}$	$-\hat{j}$
$\hat{j}$	$\hat{j}$	$-\hat{k}$	-1	$\hat{i}$
$\hat{k}$	$\hat{k}$	$\hat{j}$	$-\hat{i}$	-1

Quaternions—and higher dimensional complex numbers as a whole—are an extremely interesting and surprisingly useful branch of mathematics. However, this paper does not have then space to discuss them in full.[12] Unfortunately, quaternions—due largely to their 4-dimensional nature—are rather difficult to reason about, and cannot be so easily and consisely explained. Thus, in order to keep the focus on linear algebra and not quaternions, it will unfortunately have to suffice to say that unit quaternions can represent the vectors of a unit 4D hypersphere, and when considered within the framework of quaternion conjugation, translates well to rotation in 3D space (in particular, our axis angle representation.) The main takeaway here is that performing quaternion conjugation performs a more complicated movement in 4D space that allows us to cleanly represent rotations in 3D space. In fact, we can actually cleanly convert from the axis-angle representation to a quaternion representation:

$$\begin{aligned} R(\vec{v}, \theta) &= R(q) \\ &= R(cos(\theta) + sin(\theta)(\vec{v}_i + \vec{v}_j + \vec{v}_k)) \end{aligned}$$

Though it appears daunting (and, indeed, its derivation relies on a more complicated understanding of quaternions than can be conveyed here), the rotation matrix doesn't contain anything too complicated:

$$R(q) = \begin{bmatrix} 1 - 2(q_j^2 + q_k^2) & 2(q_i q_j - q_k q_w) & 2(q_i q_k + q_j q_w) \\ 2(q_i q_j + q_k q_w) & 1 - 2(q_i^2 + q_k^2) & 2(q_j q_k - q_i q_w) \\ 2(q_i q_k - q_j q_w) & 2(q_j q_k + q_i q_w) & 1 - 2s(q_i^2 + q_j^2) \end{bmatrix}$$

And, as previously mentioned, we can perform quaternion conjugation to concatenate multiple rotations:

$$(R(p)R(q))\vec{v} = p(q\vec{v}q^{-1})p^{-1}$$

At this point, we can go ahead and implement this in our code:

```
package transformations

import (
  "gonum.org/v1/gonum/mat"
  "math"
)

type Rotation struct {
  I, J, K, W float64
}

func RotationFromAxisAngle(axis *mat.VecDense, theta float64) *Rotation {
  return &Rotation{
    axis.At(0, 0) * math.Sin(theta/2),
    axis.At(1, 0) * math.Sin(theta/2),
    axis.At(2, 0) * math.Sin(theta/2),
```

[12] If you want to gain a better conceptual foundation, I recommend you view this series.

```
    math.Cos(theta / 2),
  }
}

func (r *Rotation) GetTransformation() *mat.Dense {
  return mat.NewDense(4, 4, []float64{
    1 - 2*(r.J*r.J+r.K*r.K), 0 + 2*(r.I*r.J-r.K*r.W), 0 + 2*(r.I*r.K+r.J*r.W), 0,
    0 + 2*(r.I*r.J+r.K*r.W), 1 - 2*(r.I*r.I+r.K*r.K), 0 + 2*(r.J*r.K-r.I*r.W), 0,
    0 + 2*(r.I*r.K-r.J*r.W), 0 + 2*(r.J*r.K+r.I*r.W), 1 - 2*(r.I*r.I+r.J*r.J), 0,
    0, 0, 0, 1,
  })
}
```

objects/transformations/rotation.go

4 Viewing and projection

4.1 Seeing is believing

As alluded to by the name, the viewing transformation, and corresponding projection matrix, handles the projection of worldspace onto the screenspace, roughly speaking. In other words, the viewing transformation handles making our 3D world look like a 3D world on our 2D screen—even if that image isn't exactly what we're used to.

It's important to note that the viewing transformation is a graphical projection, not a projection as typically considered in linear algebra. Though the projection matrix is not necessarily affine, and can distort all dimensions, it is still a projection from $\mathbb{R}^3 \to \mathbb{R}^3$, i.e. it does not discard the Z axis. This is important, as the Z-depth information is important for handling a number of calculations, most notable visibility (though we won't explore those here.)

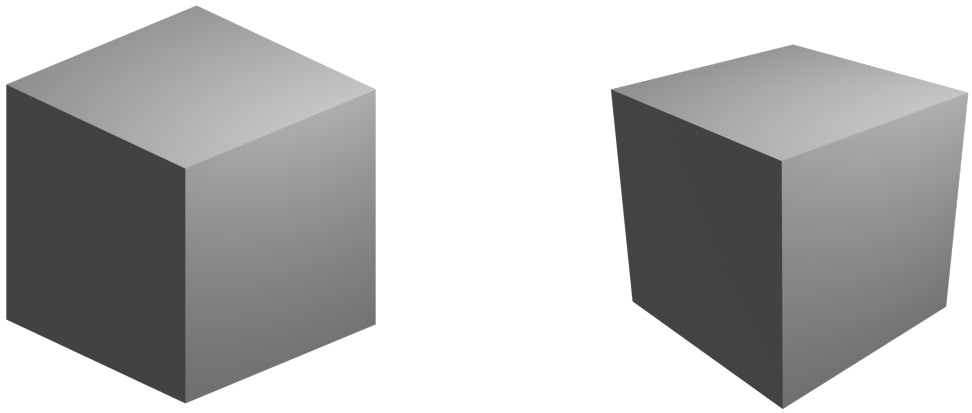
Handling viewing usually starts with considering a camera in space, looking out into our world. The important thing to recognize here is that the camera is not stationary, since we want to be able to change our view. However, working with a non-stationary camera would mean adjusting our projection math to constantly handle all the translations and transformations of the camera itself. Instead, since all motion is relative anyways, we can instead simply transform our world in the opposite manner, allowing us to keep our camera fixed at the origin while giving the same appearance of movement. These transformations can be stored separately, then applied to the objects in the scene.

4.2 Types of projection

As shown in the table in section 3.2, for our purposes, there are two major types of projections: the orthographic projection and the perspective projection.[13] The main distinction of the orthographic projection, as alluded to by the name, is that it is a parallel projection, i.e. the "projection lines" are parallel to each other and orthogonal to the viewing plane of the screenspace. Thus, the orthographic projection is an affine transformation. By contrast, the perspective transformation works using normal perspective, where the projection lines are instead case outward from a single point at various angles relative to the viewing plane (dependent on the field of view of the viewing plane.) This often discussed

[13] While there are others, most notable the oblique projection, these two are the most common and useful.

in conjunction with the concept of the "pinhole camera", where the pinhole is the point that all the projection lines travel through. This is similar to how cameras and our eyes perceive the world, and thus, is generally the most familiar projection.



(a) (b)

Figure 6: A cube viewed with (a) an orthographic projection (b) a perspective projection

4.3 Orthographic projection matrix

Hopefully based on our description of the orthographic projection, you can guess what the corresponding matrix should be.

$$P_o = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Since all that's happening is an affine projection of the existent geometry onto the viewing plane and we want to maintain the Z values for other purposes, **no changes are necessary** to get the most basic form of the orthographic projection matrix. However, this doesn't make the projection matrix redundant, as we'll see shortly.

4.4 Perspective projection matrix

Again, the perspective projection matrix is surprisingly simple. Setting the viewing plane n distance from the camera at the origin:

$$P_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & n^{-1} & 1 \end{bmatrix}$$

To understand what this is doing, let's revisit the notation of the w term. Recall that $(x\ y\ z\ w)^T = w^{-1}(x\ y\ z)^T$. This means that this is effectively scaling the third vector in the linear transformation portion of the affine transformation by the reciprocal of the distance, i.e. making the portion of the linear transformation that handles the Z component of any given vector smaller. In other words, the further a vector is from the origin along its Z-axis, the more it'll be squeezed towards zero.

If you've ever done drawing in an art or architecture class using point perspective, this concept should

hopefully be familiar: the further along the basis axes you travel towards the point, the closer to the axis you get.

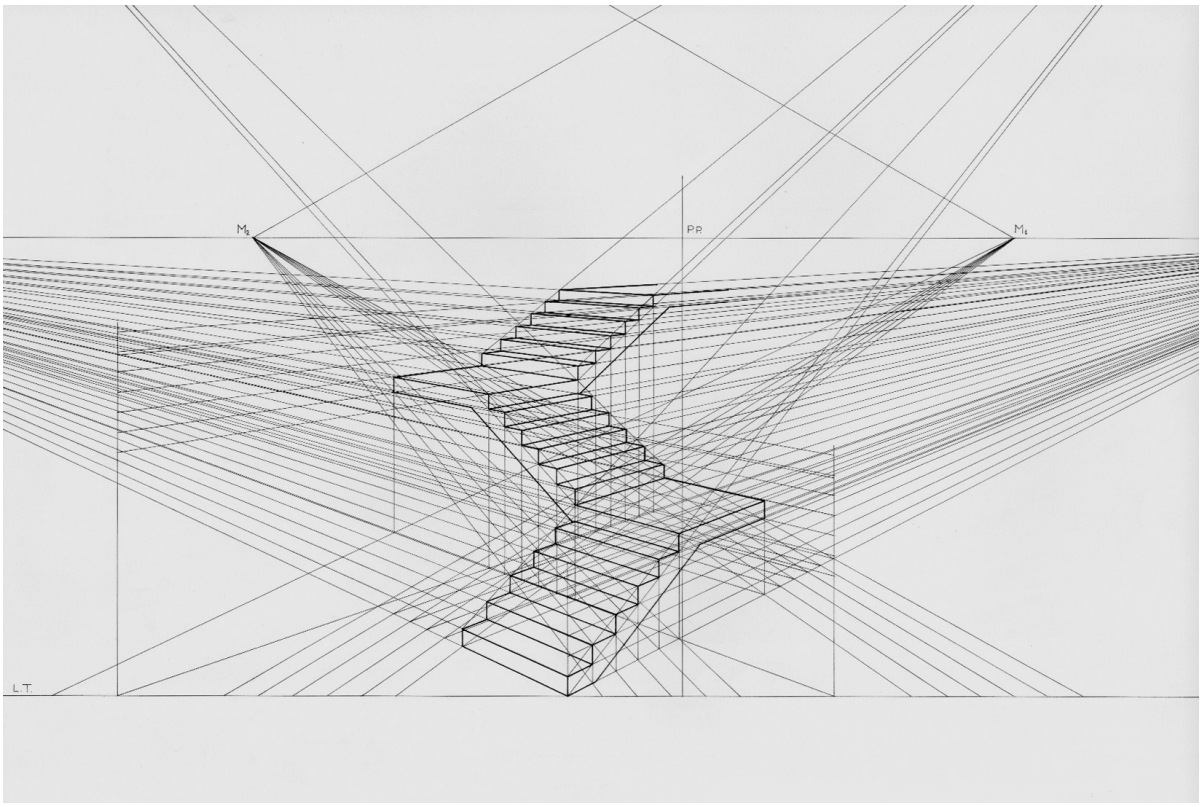


Figure 7: [15]

4.5 Normalization and conversion

While these basic transformation matrices do suffice, neither of these matrices are in their ideal usable form. For good reason, these matrices are not used in their simplest form by any major libraries (OpenGL, DirectX) or hardware vendors[14].

Again, recall that we need the Z value for other calculations (most notable visibility.) However, the nature of these calculations means that the absolute Z values are unimportant; instead, the values relative to each other are important, as is the precision and accuracy of said values—especially close to the camera.[15] These Z values are stored as floating point numbers, where non-integer, non-fractional numbers are represented as a combination of a sign, a "significand" with the significant digits of the number, and an exponent (in our case, since we're using 64-bit IEEE 754 double-precision floating points, the sign is one bit, the exponent is 11 bits, and the significand is 52 bits.) Thus, these values can (practically) represent all of our $\mathbb{R}$ space, albeit at varying levels of precision depending on the exact value. However, this is not entirely desirable for our application, as we want to finer control the use of the significand. In general, for our problems, we know that points closer to the viewing plane must be rendered with more precision, while ones that are further away can be handled with less precision (or even discarded

[14]Many GPUs now handle projection onboard; this is now the canonical way of doing things, rather than specifying a particular projection matrix

[15]Again, the particulars of why are outside the scope of this paper, but if you want a search term, "z fighting" would be a good place to start.

entirely.) Thus, we can normalize our Z values to a given range, choosing how we want to distribute our precision:

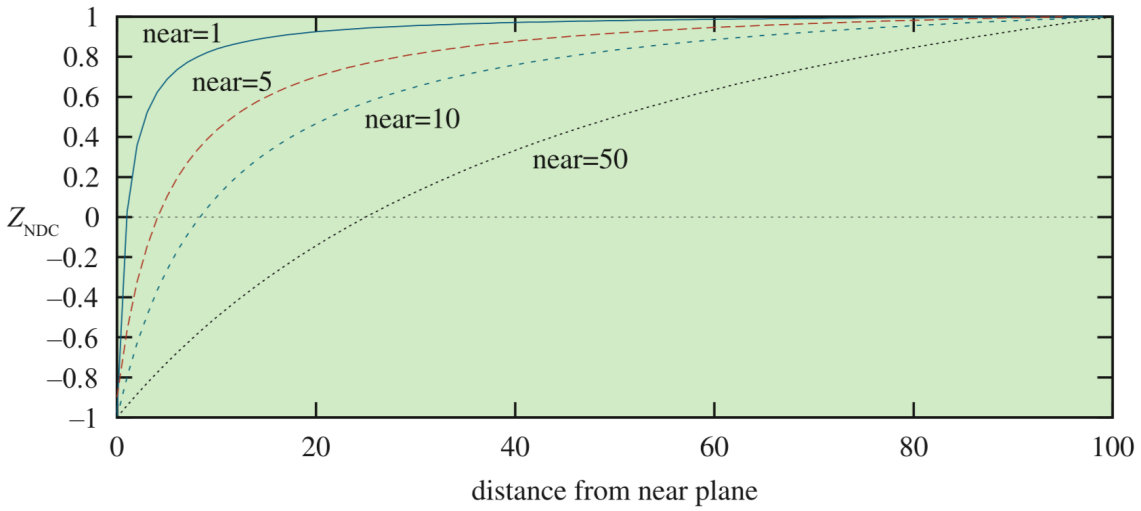


Figure 8: The effect of varying the distance of the near plane from the origin. [16]

Note that this graph shows a normalization from $(-1, 1)$; this is not always the case.[16]

Our final step is to handle the coordinate transformations. As discussed in section 2.1, our screenspace and worldspace use different conventions for defining coordinates. So, we'll convert between there here. While it would be possible to simply convert every value when actually updating the raster image, by incorporating it into the projection matrix, we can decrease overhead. First, we need to define the left, right, bottom, and top bounds of our screenspace, as well as a near-clipping and far-clipping plane beyond which no vectors are visible. We can notate these as l, r, b, t, n, and f respectively. Applying these two properties gives the following transformation matrices:

$$P_o = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$P_p = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{1}{\tan(\frac{\theta}{2}) \cdot \frac{r-l}{t-b}} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{\theta}{2})} & 0 & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

[16] $(-1,\ 1)$ is more frequently used on GPUs and OpenGL, where $(0, 1)$ is used with DirectX; we will stick with the former.

where θ optionally is the vertical field of view. Hopefully, especially with the orthographic projection, the normalization (values along the diagonal of the linear transformation) and transformation to screenspace coordinates (the translation) are clear.

5 Putting it together

5.1 Code

At this point, we have enough to assemble our core code. Remember that this isn't the full code (see section 2.4.)
Let's start by updating our scene with a projection matrix, a view type, and an updater for that matrix, as well as adding a updater function to handle the actual application of the various matricies, and a list of the functions we want to use to render out our projected triangles:

```
package main

type Scene struct {
    objects                                []objects.VertexObject
    projection                             *mat.Dense
    viewType                               ViewType
    nearClippingPlane, farClippingPlane float64
    updater                                func(width, height, fov float64
   )
    shaders                                []shaders.Shader
}

type ViewType uint8

const (
  ORTHOGRAPHIC = 0
  PERSPECTIVE  = 1
)

func (s Scene) PerspectiveTransformUpdate(width, height, fov float64) {
  top := math.Tan((fov/360*2*math.Pi)/2) * s.nearClippingPlane
  bottom := -top
  right := top
  left := -top

  s.projection.Set(0, 0, (2*s.nearClippingPlane)/(right-left))
  s.projection.Set(0, 2, (right+left)/(right-left))
  s.projection.Set(1, 1, (2*s.nearClippingPlane)/(top-bottom))
  s.projection.Set(1, 2, (top+bottom)/(top-bottom))
  s.projection.Set(2, 2, -(s.farClippingPlane+s.nearClippingPlane)/(s.
    farClippingPlane-s.nearClippingPlane))
  s.projection.Set(2, 3, -(2*s.farClippingPlane*s.nearClippingPlane)/(s
    .farClippingPlane-s.nearClippingPlane))
}

func (s Scene) drawObjects(canvas shaders.Canvas) {
    for _, object := range s.objects {
```

```
35
36          ...
37
38          for _, transformationMat := range object.Transformations {
39              // Calculating and concatenating the transformations for
    each object
40              ...
41          }
42
43          // Concatenating scene-wide transformations with object-
    specific transformations
44          ...
45
46          for _, face := range object.Faces {
47              for i, vertex := range face.RawVertices {
48                  // Applying transformations to each vertex
49                  ...
50                  // Normalizing each vector
51                  ...
52              }
53          }
54
55          for _, shader := range s.shaders {
56              shader.Shade(object, canvas)
57          }
58      }
59  }
```

scene.go

After that, we just need a main to handle the initial setup and some basic communication:

```
1  package main
2
3  const fov = 30
4
5  func main() {
6      // 2D graphics
7      imageChannel := make(chan *image.RGBA, 60)
8      myDisplay := display.CreateDisplay(imageChannel)
9
10     myScene := Scene{
11     projection: mat.NewDense(4, 4, []float64{
12       1, 0, 0, 0,
13       0, 1, 0, 0,
14       0, 0, 1, 0,
15       0, 0, -1, 0,
16     }),
17     viewType: PERSPECTIVE,
18     nearClippingPlane: -1, farClippingPlane: -10,
19     myScene.updater = myScene.SceneUpdater
20     shaders: []shaders.Shader{
21         // Draws the triangles with (randomly chosen) shaded in colors
```

```
      shaders.ShadeFacesInt{Colors: []*color.RGBA{
        {43, 142, 198, 255},
        {41, 35, 114, 255},
        {181, 23, 37, 255},
        {33, 48, 52, 255},
        {163, 183, 190, 255},
      }},
    },
  }

  // Reading from a file the describes the geometry of a model
  obj := objects.ReadFromObj("bmw2.obj")
  myScene.objects = append(myScene.objects, obj)

  // Handling display functionality in a separate thead and
   communicating via a channel
  go func() {
      ...
      w, h := myDisplay.GetDimensions()
      frame := genBlankCanvas(w, h)
      ...
      myScene.updater(float64(w), float64(h), fov)
  myScene.drawObjects(shaders.Canvas{
    Image:      frame,
    ZBuffer:    zBuffer,
    UseZBuffer: true,
  })
  imageChannel <- frame
  ...
  }()
  myDisplay.Start()
}

func genBlankCanvas(w, h int) *image.RGBA {
  canvas := image.NewRGBA(image.Rect(0, 0, w, h))
  for i := range canvas.Pix {
    canvas.Pix[i] = 255
  }
  return canvas
}
```

main.go

And with that (plus some more code), we've got a working rasterizing rendering program!

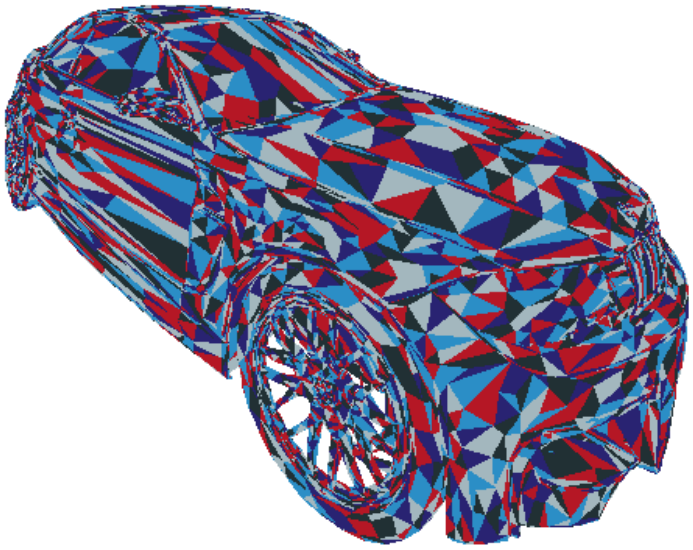


Figure 9: Image of a car rendered using the described program

5.2 Conclusion

The field of 3D computer graphics is a large one, and unfortunately, there's only so much I can cover. From texture mapping to backface culling, ray tracing to Phong shading and shadow mapping (very similar to projection!), there are dozens of applications of linear algebra to the field. However, they all build off this base. Though there's more technical and more foundational knowledge to learn, my hope is that this paper has provided a sufficiently digestible overview of the foundations of computer graphics that it allows you to explore the field in more depth—up or down the technology stack—if you so desire.

References

[1] Lionel Brits. *Euler angles*. Jan. 9, 2008. URL: https://en.wikipedia.org/wiki/File:Eulerangles.svg.

[2] Samuel R. Buss. *3D Computer Graphics: A Mathematical Introduction With OpenGL*. Cambridge University Press, 2003.

[3] Juan David Gonzáles Cobas. "Mathematics of 3D Graphics". Blender Conference. 2004. URL: https://www.cs.trinity.edu/~jhowland/class.files.cs357.html/blender/blender-stuff/m3d.pdf.

[4] *Computer Graphics from Scratch*. URL: https://www.scratchapixel.com/index.php?redirect (visited on 11/15/2019).

[5] Jeffery Doe. *Loop order and Iliffe vector*. Aug. 25, 2017. URL: https://massivetechinterview.blogspot.com/2017/08/loop-order-and-iliffe-vector.html.

[6] Jack Dongarra. *Survey of Sparse Matrix Storage Formats*. Nov. 20, 1995. URL: http://netlib.org/linalg/html_templates/node90.html.

[7] *Figure 1 - The Intertial Frame*. URL: http://www.chrobotics.com/library/understanding-euler-angles (visited on 12/10/2019).

[8] Gonum. *Gonum*. Version v0.6.1. Nov. 19, 2019. URL: https://www.gonum.org.

[9] John F. Hughes et al. *Computer Graphics: Principles and Practice*. 3rd. Addison-Wesley, 2014.

[10] joojaa. *How to achieve gimbal lock with Euler angles?* Dec. 30, 2016. URL: https://computergraphics.stackexchange.com/a/4438.

[11] Kai. *Why do people use quaternions?* Feb. 8, 2012. URL: https://gamedev.stackexchange.com/a/23543.

[12] LucasVB. *A 3D gyroscope rendered in POV-Ray*. Oct. 4, 2006. URL: https://commons.wikimedia.org/wiki/File:3D_Gyroscope-no_text.png.

[13] *OpenGL Tutorial*. URL: http://www.opengl-tutorial.org/miscellaneous/ (visited on 11/15/2019).

[14] Arnold Reinhold. *Apollo program Inertial Measurement Unit on display at the Draper Labs 2019 "Hack the Moon" exhibit, held in Cambridge, Massachusetts, in honor of the 50th anniversary of the Apollo 11 moon landing*. Sept. 13, 2019. URL: https://commons.wikimedia.org/wiki/File:Apollo_IMU_at_Draper_Hack_the_Moon_exhibit.agr.jpg.

[15] Luciano Testoni. *Prospettiva accidentale di una scala a tre rampe, eseguita con il metodo dei punti misuratori. Il file è stato creato scandendo il disegno con uno scanner*. Mar. 15, 1995. URL: https://commons.wikimedia.org/wiki/File:Staircase_perspective.jpg.

[16] Möller Tomas et al. *Real-Time Rendering*. 4th. CRC Press, 2018.

[17] Noah Witherspoon. *Why are quads used in filmmaking and triangle in gaming?* Aug. 4, 2017. URL: https://computergraphics.stackexchange.com/a/5466.