Quaternions, Dual Numbers, and Dual Quaternions – An Introductory Guide and Computational Comparison with Matrices and Vectors

Abstract

I wanted to learn about dual quaternions, but the available information was sparse and very academic, so it was difficult for a young layman software engineer like me to understand them and practically use them. I spent awhile digging into the available material, experimenting, and performing a lot of equations and analysis by hand and in a basic engine that I built, and I feel much more comfortable with them now. I feel comfortable enough to write a long paper on them, how they work, some conceptual analogies that I find helpful, non-trivial example equations, and a special section on "gimbal lock" and the history of that term. I concluded that dual quaternions are very useful for managing relative transformations of objects, that they are less computationally and memorically (I made up an adverb form of "memory") expensive to create than translation and rotation matrices, but that they are significantly more expensive than matrices in transforming vectors. On top of that, GPUs are built to do all the matrix-vector maths and not dual quaternion math, so I concluded that it is most efficient to use dual quaternions with the CPU and system memory to track an object's location and orientation in 3D space, and then convert the dual quaternion to a matrix just before drawing.

My implementation can be examined at my github repo:

Purpose for Writing

I tried to learn about quaternions and dual quaternions on my own, but information on them was rather sparse at the time of writing and some of the math that was represented in the available information was obscure to me and not fleshed out with non-trivial examples that really showed them in action. I wanted to learn how to use them though and what their benefits were, so I spent several few weeks digging into complex numbers, quaternions, dual numbers, and dual quaternions in an attempt to understand the concepts, their application, their implementation in code, and their computational cost in comparison with matrices and vectors. This guide is written to help other people avoid the time-consuming incorrect paths of application and equation, get right to something that works, and satiate their curiosity if they want to understand how it works. That's the goal, at least, and I hope that this paper will grow with time as I learn new applications of dual quaternions.

My Personal Suggestion on How to Read

I tried to write this paper with linear thought so that it is a single and continuous line of thought from start to finish, but there is a lot of information here that I pulled together to explain these topics as thoroughly as I did, so I recommend against trying to understand it all in one read-through. The human brain can only absorb brand new information at a certain (rather low) rate, and it can't do it all day. Try as I might, brand new concepts and applications of those concepts sap my capacity to learn stuff very quickly and I have to take a break. I suggest reading in chunks and taking a break to let your mind chew on it.

Background on What Drove Me to Dig into This

I started to learn graphics programing and quickly became curious about the mathematical background of moving a point around in 3D space. I progressed through some openGL tutorials and was introduced matrices and vectors, and during this time I stumbled across a concept that was new to me: "quaternions". I looked into them with a guide on the website 3D Game Engine Programming[1], but quickly felt overwhelmed, so I moved passed them and went back to my openGL tutorials. Later, one of the tutorials brought up quaternions again and how they avoid the curious mathematical and engineering dark void concept referred to as "gimbal lock", so I started looking into that more closely as well.

I soon realized that I needed a graphical program in which I could experiment with these concepts, so I collected my knowledge from past tutorials and began building my own basic graphics and physics engine. I initially built it with GLM's fquat structure to represent all of my entities' orientations in 3D space, but I had difficulty getting it to work properly, particularly with the camera because source material was so sparse.

I went back to the quaternion guide on 3dgep.com and understood them better the second time, and with a little help from some people at /r/OpenGL, I learned enough to get my graphical program working and getting a camera operational. This basic engine took a couple months for me to stumble through and put together in my spare time, and during this time I stumbled across another concept in google search for "quaternion": "dual quaternion".

My first foray into dual quaternions was a paper entitled "A Beginners Guide to Dual Quaternions"[2] ("Beginners" should be "Beginner's" or "Beginners'", but oh well), by Ben Kenwright. This paper briefly reviewed quaternions and introduced me to the concept of dual quaternions, which don't supplement matrix transforms like quaternions can (quaternions can do the rotation part, but they supplement the translation matrix), but rather can completely replace them and handle both the rotation and translation components. This was curious to me. I was using a glm::vec3 for entity position and a glm::fquat for orientation, so I still made use of glm::translate(…) to move the entity around, but a dual quaternion promised to move away from matrices entirely and keep track of both the entity position and orientation in a single, inexpensive (relative to translation and rotation matrices) data structure.

In graphical programming, physics, and kinematics, it is quite common to have an entity somewhere in space on one frame, and then some forces or some other transform is applied to it, so in the next frame it must then be transformed relative to where it was. Quaternions and dual quaternions excel in relative transformation. Ben Kenwright's paper described relative transformation as being approximately 30% cheaper for dual quaternions than for matrices (see end of page 8).

Now I really wanted to learn about dual quaternions and how to use them, and fortunately, GLM's gtx library (their experimental library) had a glm::fdualquat structure. I spent a lot of time and got a lot of online assistance, again from some people at /r/OpenGL, to make them work, but I ultimately didn't understand *why* they worked. Additionally, GLM didn't have many assistant functions that would have made dual quaternions much easier to construct and use, GLM's normalization process for their

[1] http://3dgep.com/understanding-quaternions/
[2] http://wscg.zcu.cz/wscg2012/short/a29-full.pdf

fdualquat structure was not correct because the “normalized” dual quaternion multiplied by its conjugate was not 1 (the definition of a normalized multiple-component number; more in a later section), and GLM’s fdualquat structure file included glm.hpp, so it was basically including the whole world and made for a very bloated inclusion. I’ll give them some leeway though since it was their experimental library and not a release one.

So I decided to make my own quaternion and dual quaternion structures. The rest of this paper describes what I learned.

My Computational Cost Table

I think that it is important to compare the performance of dual quaternions with matrices since the former is sometimes advertised as being able to replace the latter. I will use the following table to track the number of each type of operation as well as the number of floating point values (I assume the use of floats instead of doubles) required to compute it. Constructors and binary operators can conceal a lot of temporary floats that take up a few bytes apiece and then disappear, so they are still part of the performance cost.

	Sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Operation (“complex rotor generation”, “quaternion to mat4”, etc.)	#	#	#	#	#	#	#

The sine and cosine calculations are necessary for all rotation calculations regardless of whether I am using dual quaternions or matrices. I can’t avoid them and there are not very many of any such computations, but since they are expensive, I’ll track them.

The multiplication and division are separate because they are different circuits with very different performance expectations. If it is practically possible, I avoid dividing and instead multiply by a decimal fraction. For example, instead of dividing by 2, I multiply by 0.5. Contrast normalization, which is one division that I can’t get around. In that case, the division is noted.

The addition and subtraction circuits are expected to have identical performance, so add/sub operator counts are lumped together.

The temporary floats have had me scratching my head for awhile as I try to figure out what is the most fair way to compare dual quaternions and matrices. I eventually decided to compare my implementation of quaternions and dual quaternions with GLM’s implementation of matrices. I want to make a note here about return-value optimization: if used, then I will not count the number of floats used by the return value’s constructor. If it is *not* used, then I will count them. For example, glm::rotate(…) with a glm::mat4 returns a copy of a matrix instead of constructing it in place. After looking at the function, I can understand why they did this, but it’s still a notable temporary float cost. Additionally, I will *not* include the final float cost of the data structure since that is provided by whoever

called the function. So if I am constructing a dual quaternion to perform a translation and then a rotation, I will not count the number of floats in the data structure that the function is returning to. I *will* count everything in between the call and the return though.

After I present a table, I will give a breakdown of where the numbers came from.

Mathematical Background

The Real Number

A real number is a value that represents a quantity along a continuous line. This includes whole numbers, decimals, and numbers that cannot be written as a fraction, such as π and $\sqrt{2}$ (that is, *irrational* numbers). An alternative definition is that a real number is any quantity that can be produced by the square of a lesser quantity. That is, the following equations can be solved if and only if 'a' and 'b' are real numbers:

$$a = \sqrt{b}, \text{or alternatively}, a^2 = b$$

If b is a real number, then these equations can be solved. This is important because there are values for b such that the equation cannot be solved, and therefore a would not be real. For example, suppose $b < 0$.

The Imaginary Number

An imaginary number is a number that can be written as the product of any real number and the "imaginary unit" $\boldsymbol{i}$, which is defined as follows: $\boldsymbol{i}^2 = -1$, or alternatively, $\boldsymbol{i} = \sqrt{-1}$. The following are all imaginary numbers:

$$\sqrt{-2} = \sqrt{2}\boldsymbol{i}$$

$$\sqrt{-42} = \sqrt{42}\boldsymbol{i}$$

$$\sqrt{-9001} = \sqrt{9001}\boldsymbol{i}$$

Imaginary numbers are not real numbers. Consider the equation to define a real number:

$$a = \sqrt{b}, \text{or alternatively}, a^2 = b$$

If 'b' is negative, then no real number 'a' exists to solve the equation. Alternatively, there is no real number 'a' that, when squared, is a negative number.

Real and imaginary numbers do not mix. One cannot be added to, subtracted from, multiplied by, or divided by the other.

The Operator

In mathematics, an "operation" is defined as "an action or procedure which produces a new value from one or more input values"[3]. This is a simple concept. The following are all operations:

$$a + b = c$$

[3] http://en.wikipedia.org/wiki/Operation_(mathematics)

$$a - b = c$$

$$a * b = c$$

$$\frac{a}{b} = c$$

$$\frac{(a + b) * (c - d)}{e + f} = g$$

$$-a = b$$

For the sake of this discussion, consider all the variables here to be real numbers. In each case, an action or procedure (addition, multiplication, etc.) produced a new value from one or more input values. Furthermore, these operations were executed by "operators", which are mathematical symbols that define some operation. The operators $+$, $-$, $*$, and $/$ (division) are called "binary operators" because they take in two values: a left value and a right value. Consider the first equation: the left value is 'a', the second value is 'b', and the operator symbol is '$+$', which is defined to mean that the left and right values are summed together.

Trivial, right? It depends on who or what you are communicating with. Most people understand what the symbol '$+$' means when there is a real number on the left and right side. But suppose I asked the following:

$$car + bathroom = ?$$

The '$+$' symbol is no longer defined for the left and right items unless you want to think of some silly picture.

http://lh4.ggpht.com/_7JtmuNxu--M/Sxzbv__QV6I/AAAAAAAAEx4/LLv1DRheSuQ/wm-In%20Car%20Toilet.jpg

If you are communicating with a computer program, then you will probably have to define what the '$+$' symbol means. It is already defined for integers and floats, but anything else must be defined by the programmer. Fortunately, C++ has syntax to allow the user to define operators for a class or structure like $+$, $+=$, $*$, $*=$, [ ], and others. These values must be defined for dual quaternion structures.

The last equation in the list represents a "unary operator", which is an operator that performs an operation with only one input. In this case, it negates the value of the input. A unary operator should not to be confused with a function call. For example:

Cos(x)	Function call
-x	Unary operator

The Unit Number/Vector

The concept of the unit number/vector is rather simple, but it changed the way that I thought about vectors and number lines. **Put simply, the whole purpose of a unit number is to switch between real number lines.**

In general, "unit" length means a magnitude of exactly 1, and therefore the square of a number of unit length is also exactly 1. Even the imaginary unit number ***i*** has a length of 1 because, if you square it, you get -1, which is obviously of length 1.

Suppose that we have a line of continuous real values. A unit number is a special number of that line that can be multiplied by any real number to create a point on that line of the same magnitude. The important thing is that the resulting value on that number line is the same magnitude as the real number that the unit number was multiplied by. For example, suppose we have the value 5.5. This is a value on the real number line. Let's multiply it by the imaginary unit ***i*** – we get $5.5\boldsymbol{i}$. Now we have a value of length 5.5 along the imaginary line.

Some unit numbers have special properties. ***i*** is defined such that multiplying a real number by ***i*** once brings the real number to the imaginary number line, and multiplying by ***i*** a second time returns the real number to the real number line, but now it is negative. Multiply the new real number by ***i*** two more times to bring it back around to being the original positive number on the real number line. I have seen this mathematical phenomenon sometimes referred to as an "algebraic ring" because, if you keep multiplying by them, they come back to their original point. The dual operator **ε** (covered in more detail later) is also a unit number of sorts, and it is defined such that multiplying a real number by it once brings the real number onto the dual number line, but multiplying by **ε** a second time brings it back to the real number line as 0. Therefore the dual operator is *not* a ring unit number.

This is a separation technique that is useful for separating independent sets of real values. A complex number contains two sets of independent values: the real part and the imaginary part. A dual number contains a real part and a dual part. The imaginary and dual number lines are identical in scale and value and operation to the real number line, but like orthogonal lines, they can never interact. Later, you will see that quaternions contain four independent real number lines, and dual quaternions contain eight.

In 2D or 3D (or 4D or whaterver-D) space, the axes are independent real number lines that need to be differentiated. Each axis has its own unit number that is an "axis unit vector". A vector is typically represented with a $\overrightarrow{\ }$ symbol over it, so any vector called "v" would be represented as $\vec{v}$. A unit vector has a special vector notation and is represented with a ˆ symbol over the vector name. It looks like a simple 2D hat, so a unit vector "v" is represented as $\hat{v}$ and is called "v-hat". Quick re-cap: $\vec{v}$ is general notation for a vector of any length along the axis "v", but $\hat{v}$ is a unit vector (length 1) along the axis "v".

A unit vector is composite unit number that is composed of multiples of two or more unit numbers. In 2D space a vector $\vec{v}$ is composed of the unit numbers (alternatively, "axis unit vectors") $\hat{x}$ and $\hat{y}$. In 3D space a vector $\vec{v}$ is composed of the unit numbers $\hat{x}$, $\hat{y}$, and $\hat{z}$. Finally, a unit vector, being a "unit"

something, has a magnitude of exactly one. It is composed of multiples of two or more unit numbers though, and those multiples are usually not 1 themselves, but the unit vector itself *is* of magnitude 1.

Like unit numbers, multiplying a unit vector by a real number results in a vector of that length. For example, multiplying the real number 5.5 by $\hat{v}$ results in a vector $\vec{v}$ on the "v" number line (alternatively, "v" axis") of length 5.5. 5.5 switched from the real number line to the "v" number line.

In 3D animation people will sometimes make really interesting compositions of these vectors. A texture is a 2D image, but it needs to be stretched over 3D space. To do this, they create two new unit vectors that are typically named $\hat{u}$ and $\hat{v}$ to describe a point on the texture. But a 2D vector on the $u\ v$ plane also has a point vector $\vec{p}$ in 3D space composed of multiples of the unit vectors $\hat{x}$, $\hat{y}$, and $\hat{z}$. It's quite meta.

The Conjugate

A "conjugate" is a $10 mathematical term for the result of negating the all but the first component in a multiple-component number. On a binomial (two numbers) like $a + b$, where a and b are whatever, the conjugate is $a - b$. On a longer number like $a + b + c + d$, the conjugate is $a - b - c - d$.

How is this useful? Multiplying a multiple-component number like a complex number or a quaternion or a dual quaternion by its conjugate is a roundabout way of calculating the square of the magnitude (check the math yourself and you will find that it works out to the sum of the squares of the individual coefficients), but this is only really useful if you are programming and don't want to bother squaring each term individually. Multiple-component numbers take advantage of this property in division, in which it is not possible to divide, for example $(a + b\boldsymbol{i})$ by $(c + d\boldsymbol{i})$ if because c and d are on separate number lines and therefore cannot be summed together, but it is entirely possible to multiply the numerator and denominator by the conjugate of the denominator to get $\frac{(a+b\boldsymbol{i})(c-d\boldsymbol{i})}{(c^2+(d\boldsymbol{i})^2)}$. Now the denominator is just a real number, and division by a real number is trivial. The conjugate is quite useful when trying to do some math things with multiple-component numbers.

Quaternions use their conjugate when rotating a vector, and dual quaternions use a quaternion conjugate when extracting a transformed vector from the dual part. These will be covered later.

Multiple-Component Numbers

Notation

There is a little notation that I want to clarify before moving into complex numbers, dual numbers, and quaternions.

Suppose that we have a 3D vector with 1.1 on the X axis, 2.2 on Y, and 3.3 on Z. There are multiple ways of representing this notation that appear in various tutorials that I've come across:

$$< 1.1, 2.2, 3.3 >$$

$$(1.1, 2.2, 3.3)$$

$$1.1\hat{x} + 2.2\hat{y} + 3.3\hat{z}$$

I knew the first two from math classes, but I wasn't familiar with the third. It certainly represents a 3D vector, but it uses the '+' operator to "add" three orthogonal coordinates. These values are on separate

number lines and therefore do not combine in any way, shape, or form, so the "+" operator confused me for awhile. I eventually accepted that this was simply another notation and that "+" is not being used to add anything here.

The third notation is commonly used with complex numbers, dual numbers, and quaternions, but rarely with dual quaternions because they are an 8-part number and writing all the unit numbers gets tedious

Magnitude of a Multiple-Component Number

The magnitude of a multiple-component number must satisfy the following:

$$aa^* = |a|^2$$

Where a is a multiple-component number, a^* is its conjugate, and $|a|$ is the magnitude of a (??source is some guy on the Mathematics stack exchange, but where did he get it??)

Stated another way: The square of the magnitude of a multiple-component number is equivalent to the number times its conjugate. I will address this later as it pertains to complex numbers, dual numbers, quaternions, and dual quaternions.

The Complex Number

This topic is covered in decent detail on 3D Game Engine Programming's article "Understanding Quaternions" (see references). What follows is my own attempt to explain complex numbers to myself. I hope that someone else may find it useful.

The story of quaternions begins with the definition of a complex number, which results when the operator '+' is defined to take a real number on the left and an imaginary number on the right, as follows:

Complex number: $a + b\boldsymbol{i}$ (less commonly $(a, b\boldsymbol{i})$)

where 'a' and 'b' are real numbers and ***i*** is the imaginary unit number

When we usually think of a number, we think of a whole real like 42 or a decimal real like 3.14159. But '$a + b\boldsymbol{i}$' is a perfectly legitimate number. Why? The complex number is defined for all the algebra operations that you can with a single real number. You can calculate its magnitude, you can multiply two of them together, add them, divide them, and take the square root of them if you're clever. Think of the old saying, "If it looks like a duck, walks like a duck, and quacks like a duck, then it is probably a duck." For all intents and purposes, a complex number works like a number, so it *is* a number.

Why is this number useful? That is difficult to explain in a nutshell, but the definition of ***i*** proves useful in being an alternative way to matrices to rotate a point in 2D space. Consider the following complex number multiplication:

$$(a + b\boldsymbol{i})(c + d\boldsymbol{i}) =$$

$$ac + a(d) + (b\boldsymbol{i})c + (b\boldsymbol{i})(d\boldsymbol{i}) =$$

$$ac - bd + (ad + bc)\boldsymbol{i}$$

The imaginary unit is hauled around like a coefficient of the real number and it is multiplied like a number. At the end, a new real number pops out of the multiplication of the imaginary numbers 'b' and

'd', and a new complex part is created. This real -> imaginary -> real loop is properly called an "algebraic ring" and is useful for rotating a value in 2D space.

Magnitude of a Complex Number

The magnitude of a complex number, since it is a multiple-component number, must satisfy the following equation:

$$aa^* = |a|^2$$

Where a is a multiple-component number and a^* is its conjugate

Then the magnitude squared of a complex number $a + b\boldsymbol{i}$ is as follows:

$$magnitude^2 = (a + b\boldsymbol{i})(a - b\boldsymbol{i})$$

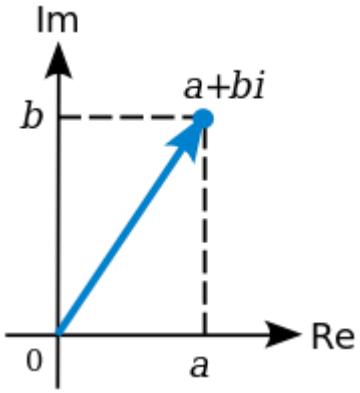
$$= a^2 - a(b\boldsymbol{i}) + b\boldsymbol{i}(a) - (b\boldsymbol{i})(b\boldsymbol{i})$$

$$= a^2 + b^2$$

Since a and b are real numbers, then the square of the magnitude is also a real number, and therefore the magnitude is the square root of this number. Quite simple.

Complex Number as a Rotor

As mentioned in the section on the unit number, the separation of the two real numbers by ***i*** allows us to consider the complex number to represent two completely independent quantities. Two completely independent lines are can be referred to as "orthogonal". Therefore, a complex number can be represented as a point in 2D space as depicted in the following picture:



http://en.wikipedia.org/wiki/File:Complex_number_illustration.svg

where "Re" stands for "real axis" and "Im" stands for "imaginary axis"

Recall my example of complex number multiplication. When two imaginary terms were multiplied together, they became a real number again. This change is represented on a 2D graph as a $+\pi/2$ radian rotation (+90° for you non-pi heathens). If I were to multiply the equation in the graph by ***i***, then I will want to change ***i*** to complex number notation as $0 + 1\boldsymbol{i}$, and then multiplying will change the original number as follows:

$$(a + b\boldsymbol{i})(0 + 1\boldsymbol{i}) =$$

$$a\boldsymbol{i} - b =$$

$$-b + a\boldsymbol{i}$$

Thus the complex number has effectively been rotated by $+\pi/2$ radians. Multiplying by ***i*** again results in another $+\pi/2$ radian rotation, and doing this two more times brings it back around to the original value.

What if I wanted to rotate by some arbitrary angle? Then I can create a rotor complex number by the following equation:

$$rotor = \cos(\theta) + \sin(\theta)\,\boldsymbol{i}$$

This will be tested in the following sections.

Matrix as a 2D Rotor (for comparison)

The following matrices will be defined column-first. So if I have a square matrix called "mat", then the notation mat[0] will refer to the first column in the matrix.

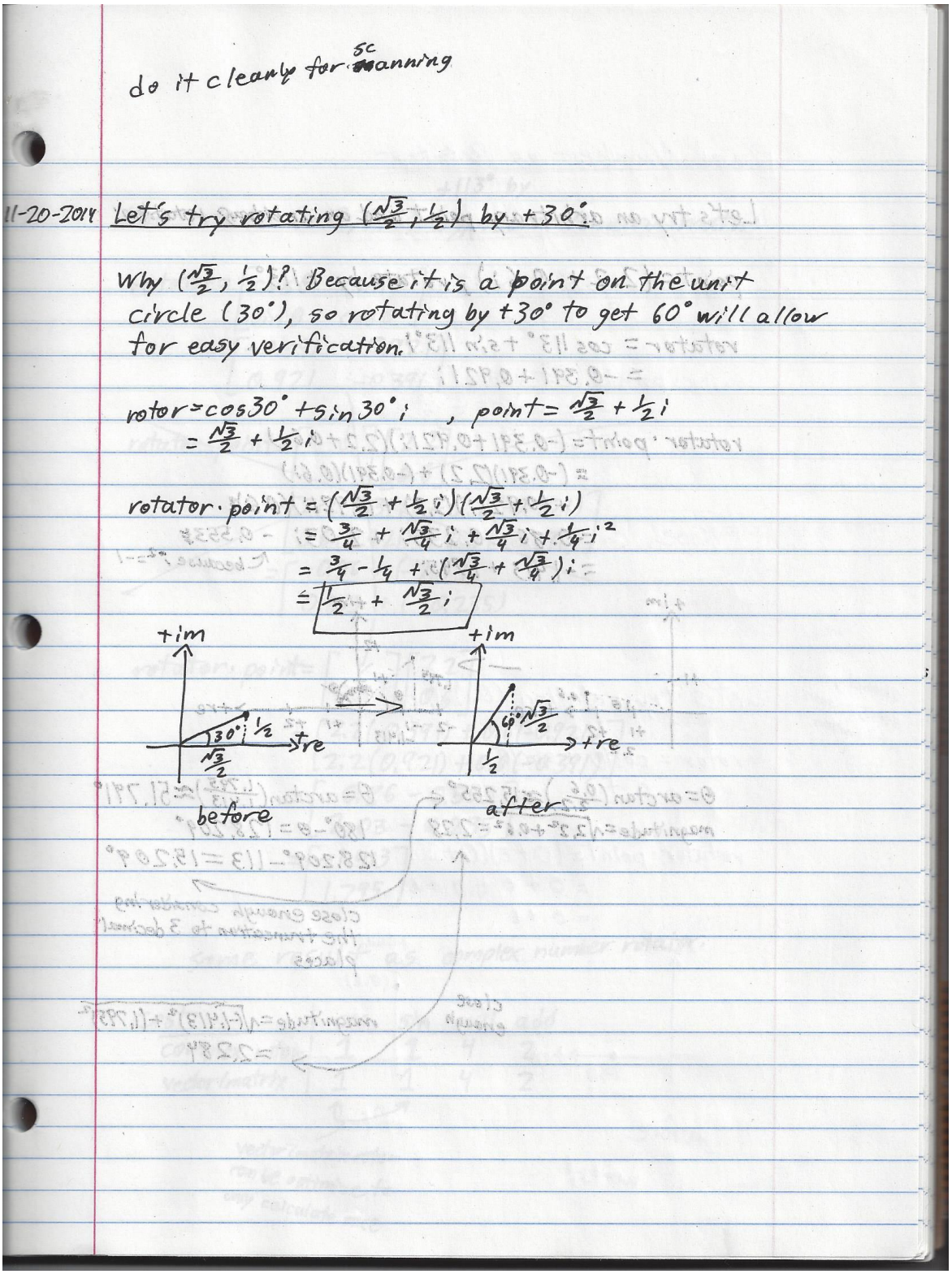
A 2D transformation matrix for rotation only is a 2x2 matrix with 4 values designated for rotation and 2 for translation. It is defined as follows:

$$\begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix}$$

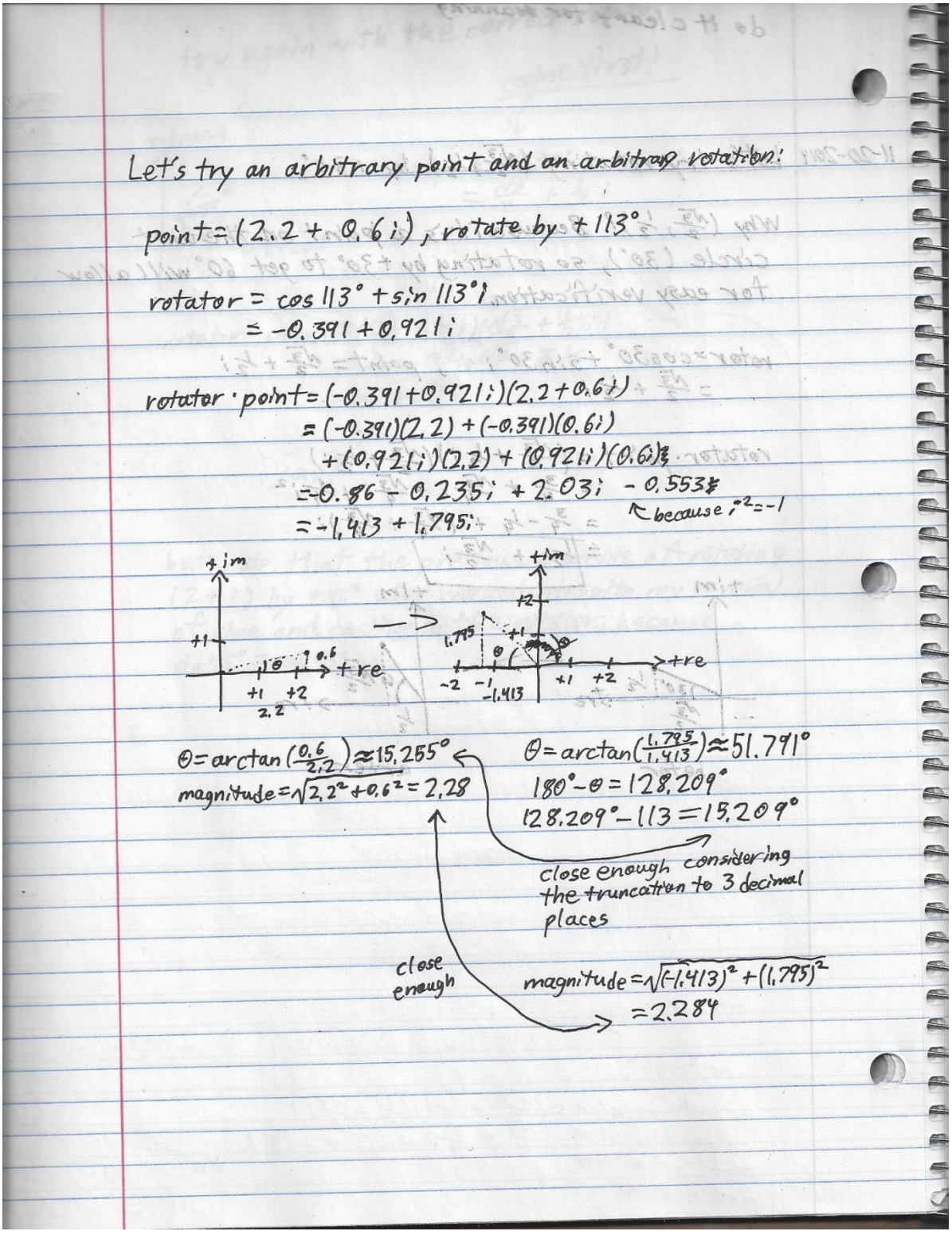
Optimization: Calculate the sin and cos once and then using the value in multiple places.

Simple 2D Rotation with a Complex Number

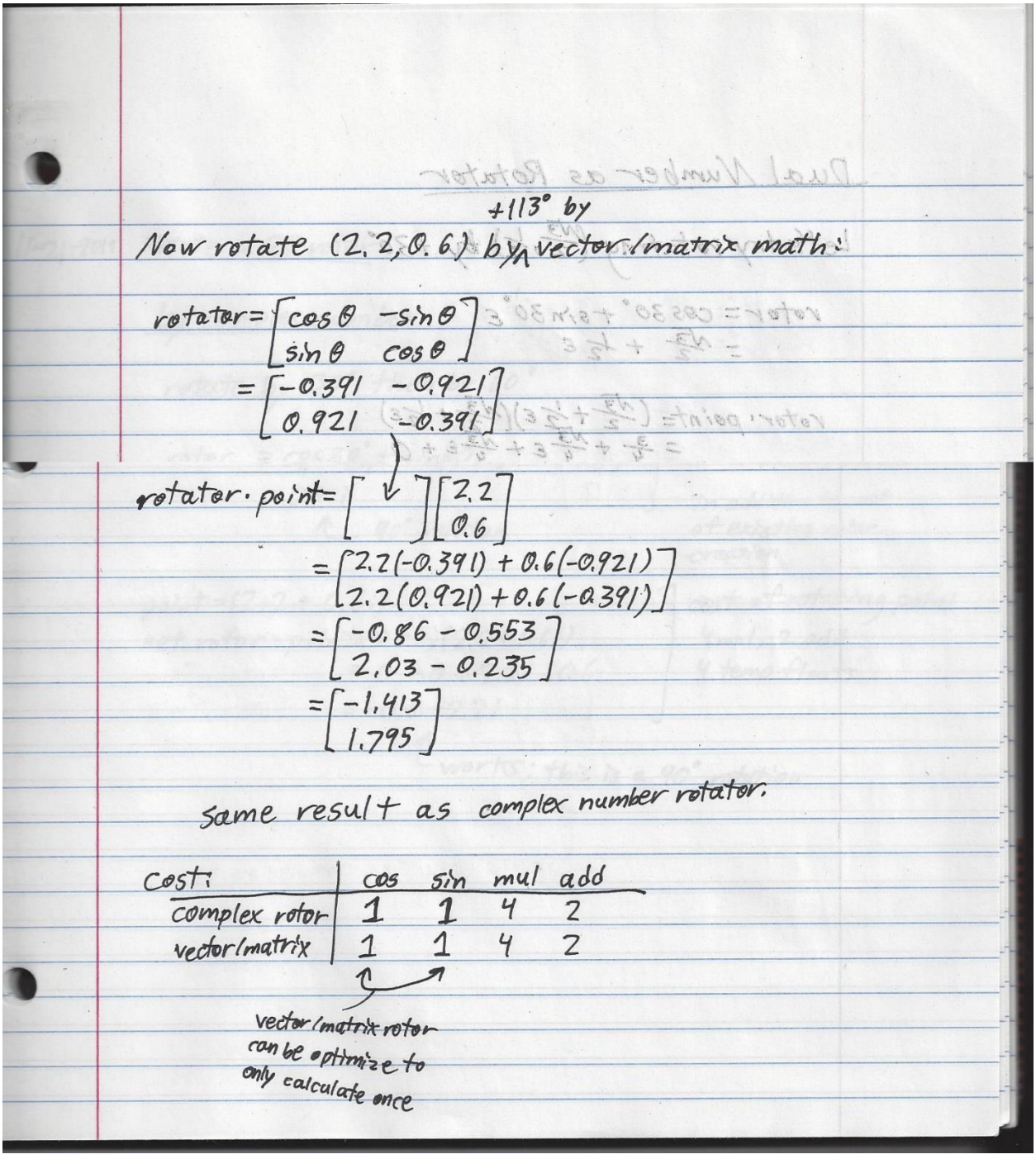
This simple rotation is just to demonstrate that it works:



Non-Trivial Rotation of a Non-Trivial Point



Non-Trivial Rotation of a Non-Trivial Point with Matrix-Vector Math



Cost of Rotating with Complex Numbers vs Matrices

Compare the computational costs of rotor generation and rotating the point. *I differentiate these operations because the difference becomes more pronounced in 3D rotations.*

Rotor generation	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Complex rotor	1	1	0	0	0	0	2
Matrix rotor	1	1	1	0	0	4	4

There's not much of a difference. The one multiplication in the matrix rotor generation comes from the negation of the sine in the second column. This may seem trivial, but I'm being picky.

The complex rotor only has two values, so it only requires two floats, whereas the matrix requires four floats. This is a trivial difference.

Rotating a point	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Complex rotor	1	1	4	0	2	0	4
Matrix rotor	1	1	4	0	2	0	4

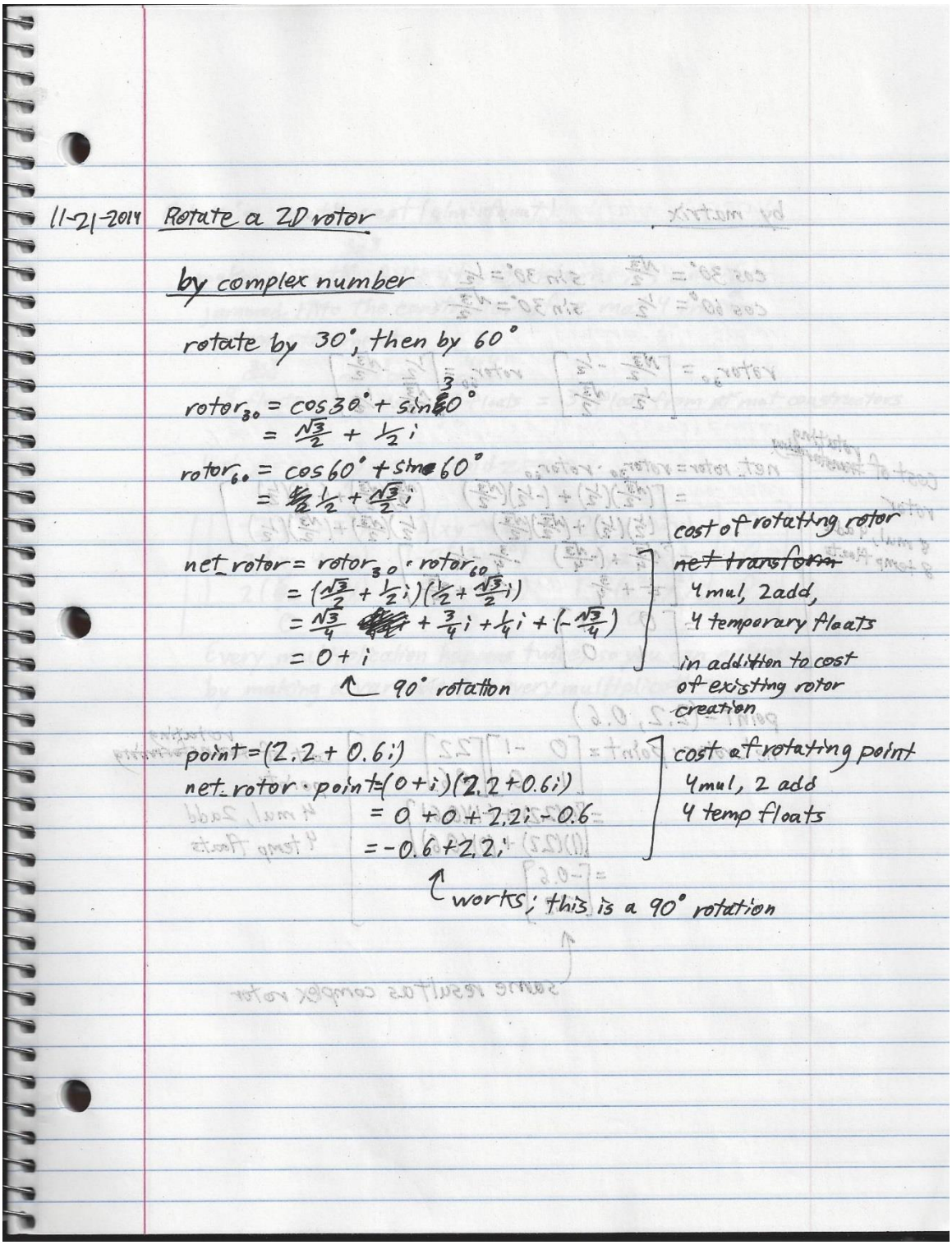
No difference.

Rotating a Complex Rotor

Suppose I have a complex rotor that will rotate a point, for example, 48°. Perhaps I have a 2D game and I have calculated a rotation for one frame, and then the next frame I want the point to rotate another 7°. I could keep track of the current rotation (48), add 7, and then calculate a new rotor, and that would certainly work.

But now consider another situation: I have a set of points on a flexible arm. As the base of arm moves, the base segment's points rotate, the next segment's points rotate relative to that, and the next segment's points rotate relative to that, etc. (I'll talk about this later in the secion on Relative Transformations). In this case, I need to calculate a new rotation for each arm segment's points and rotate that by the previous segment's rotation. This can be done by simply multiplying two rotors together. It works for matrix rotors and complex rotors.

Rotating a 2D 30° Complex Rotor by 60°



11-21-2014 Rotate a 2D rotor

by complex number

rotate by 30°, then by 60°

$rotor_{30} = \cos 30° + \sin 30°\, i$
$= \frac{\sqrt{3}}{2} + \frac{1}{2} i$

$rotor_{60} = \cos 60° + \sin 60°\, i$
$= \frac{1}{2} + \frac{\sqrt{3}}{2} i$

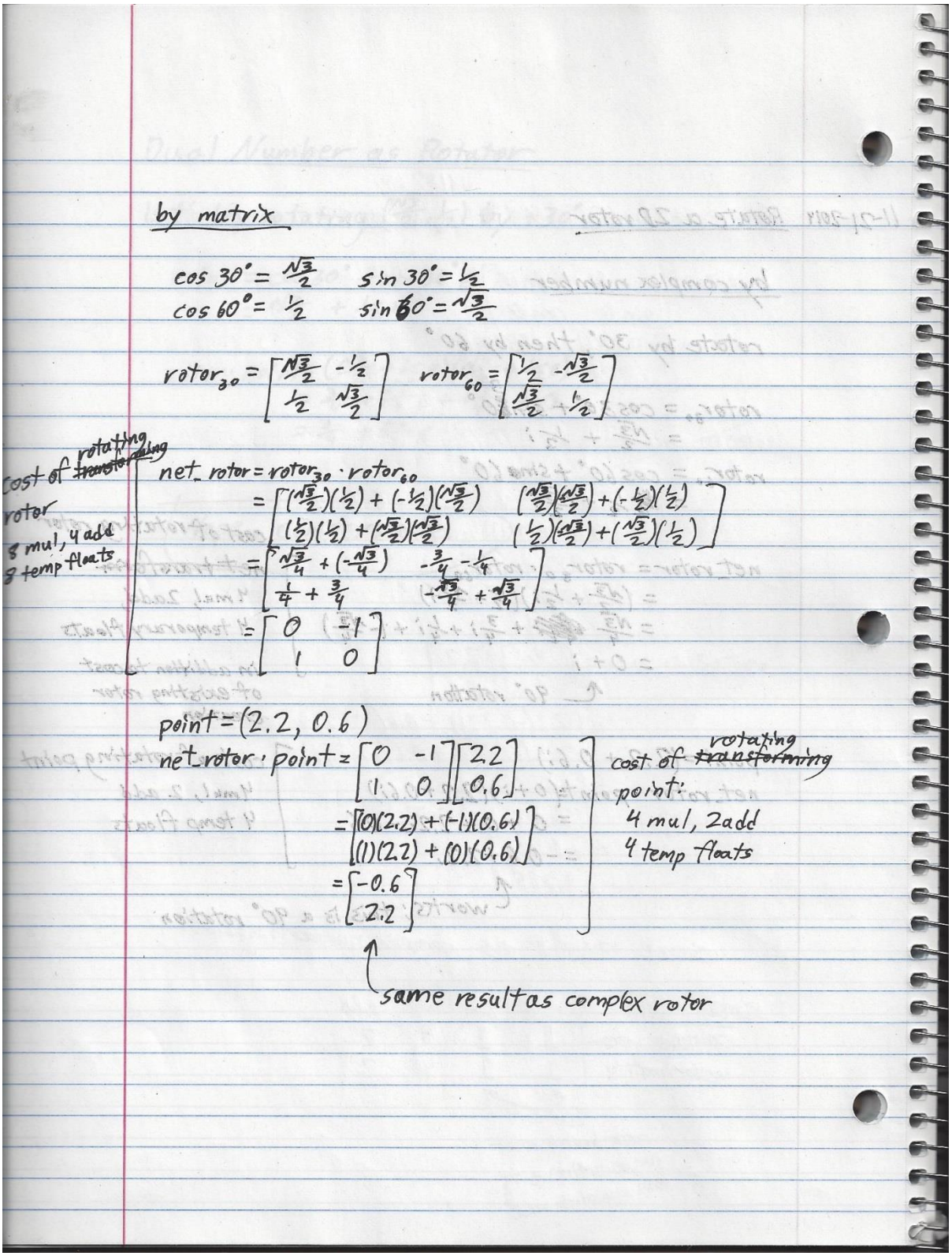
$net\_rotor = rotor_{30} \cdot rotor_{60}$
$= (\frac{\sqrt{3}}{2} + \frac{1}{2} i)(\frac{1}{2} + \frac{\sqrt{3}}{2} i)$
$= \frac{\sqrt{3}}{4} + \frac{3}{4} i + \frac{1}{4} i + (-\frac{\sqrt{3}}{4})$
$= 0 + i$
↑ 90° rotation

cost of rotating rotor
~~net transform~~
4 mul, 2 add,
4 temporary floats
in addition to cost of existing rotor creation

$point = (2.2 + 0.6i)$
$net\_rotor \cdot point = (0+i)(2.2+0.6i)$
$= 0 + 0 + 2.2i - 0.6$
$= -0.6 + 2.2i$
↑ works; this is a 90° rotation

cost of rotating point
4 mul, 2 add
4 temp floats

Rotating a 2D 30° Matrix Rotor by 60°



by matrix

$\cos 30° = \frac{\sqrt{3}}{2}$ $\sin 30° = \frac{1}{2}$

$\cos 60° = \frac{1}{2}$ $\sin 60° = \frac{\sqrt{3}}{2}$

$$rotor_{30} = \begin{bmatrix} \frac{\sqrt{3}}{2} & -\frac{1}{2} \\ \frac{1}{2} & \frac{\sqrt{3}}{2} \end{bmatrix} \quad rotor_{60} = \begin{bmatrix} \frac{1}{2} & -\frac{\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & \frac{1}{2} \end{bmatrix}$$

cost of ~~transforming~~ rotating rotor
8 mul, 4 add
3 temp floats

$$\begin{aligned} net\_rotor &= rotor_{30} \cdot rotor_{60} \\ &= \begin{bmatrix} (\frac{\sqrt{3}}{2})(\frac{1}{2}) + (-\frac{1}{2})(\frac{\sqrt{3}}{2}) & (\frac{\sqrt{3}}{2})(\frac{\sqrt{3}}{2}) + (-\frac{1}{2})(\frac{1}{2}) \\ (\frac{1}{2})(\frac{1}{2}) + (\frac{\sqrt{3}}{2})(\frac{\sqrt{3}}{2}) & (\frac{1}{2})(\frac{\sqrt{3}}{2}) + (\frac{\sqrt{3}}{2})(\frac{1}{2}) \end{bmatrix} \\ &= \begin{bmatrix} \frac{\sqrt{3}}{4} + (-\frac{\sqrt{3}}{4}) & -\frac{3}{4} - \frac{1}{4} \\ \frac{1}{4} + \frac{3}{4} & -\frac{\sqrt{3}}{4} + \frac{\sqrt{3}}{4} \end{bmatrix} \\ &= \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \end{aligned}$$

$point = (2.2, 0.6)$

$$\begin{aligned} net\_rotor \cdot point &= \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 2.2 \\ 0.6 \end{bmatrix} \\ &= \begin{bmatrix} (0)(2.2) + (-1)(0.6) \\ (1)(2.2) + (0)(0.6) \end{bmatrix} \\ &= \begin{bmatrix} -0.6 \\ 2.2 \end{bmatrix} \end{aligned}$$

cost of ~~transforming~~ rotating point:
4 mul, 2 add
4 temp floats

same results as complex rotor

Cost Comparisons for Rotating a Rotor

Rotating a 2D rotor	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Complex rotor	2	2	4	0	2	0	4
Matrix rotor	2	2	8	0	4	0	12

Note: My scanned paper of rotating the matrix rotor says that 8 temporary floats are required, which is different from the 12 that I put down in the table. This is because I was still thinking about how to catalog the cost of floats and made a mistake in my calculation. The total float cost of rotating a matrix rotor is 12: 8 temporary floats from the multiplication and 4 temporary floats from the addition that results in the final number.

Complex Number Conclusion:

In 2D space, it is computationally similar to rotate a point with a complex rotor or a matrix rotor, but rotating a matrix rotor is twice times the computational cost of rotating a complex rotor and requires three times the number of floats. That's still only 8 float multiplications and 12 floats vs 4 and 4, respectively, though, so "three times the computational cost" is nothing to panic about. The rotation is quite simple either way, but it changes a lot in 3D space.

The Dual Number

I now introduce a new unit number: the dual operator **ε** (epsilon). It is a form of a unit number, but it is commonly called an "operator" (TODO: ??why??). Like ***i,*** it changes any real number that is multiplied by it into its own number line (in this case, the dual number line), and like ***i*** the operator can still be hauled around in arithmetic like a coefficient. The operator is defined as follows:

$$\boldsymbol{\varepsilon}^2 = 0$$

Pretty exciting, isn't it?[4] A dual number is defined as follows:

$$\text{dual number} = a + b\boldsymbol{\varepsilon}$$

where a and b are real numbers

Dual Number Addition

Take two dual numbers dn1 and dn2. Add them as follows:

$$dn1 + dn2 = (a + b\boldsymbol{\varepsilon}) + (c + d\boldsymbol{\varepsilon})$$

$$= (a + c) + (b + d)\boldsymbol{\varepsilon}$$

[4] Note the sarcasm

Dual Number Multiplication

Take two dual numbers dn1 and dn2. Multiply them as follows (remember that $\boldsymbol{\varepsilon}^2 = 0$):

$$dn1 * dn2 = (a + b\boldsymbol{\varepsilon})(c + d\boldsymbol{\varepsilon})$$

$$= (a)(c) + (a)(d\boldsymbol{\varepsilon}) + (b\boldsymbol{\varepsilon})(c) + (b\boldsymbol{\varepsilon})(d\boldsymbol{\varepsilon})$$

$$= ac + (ad + bc)\boldsymbol{\varepsilon}$$

Dual Number Division

This problem has to change form in order to be calculated. We can't divide them any more then we could divide a binomial by another binomial like $\frac{x+y}{x-y}$. It has to change form until we can multiply or divide real numbers only. Do this by multiplying the top and bottom by the conjugate of the bottom. This is a useful application of the conjugate.

Given two dual numbers dn1 and dn2, divide them as follows:

$$\frac{dn1}{dn2} = \frac{a + b\boldsymbol{\varepsilon}}{c + d\boldsymbol{\varepsilon}}$$

$$= \frac{(a + b\boldsymbol{\varepsilon})(c - d\boldsymbol{\varepsilon})}{(c + d\boldsymbol{\varepsilon})(c - d\boldsymbol{\varepsilon})}$$

$$= \frac{ac - a(d\boldsymbol{\varepsilon}) + (b\boldsymbol{\varepsilon})c - (b\boldsymbol{\varepsilon})(d\boldsymbol{\varepsilon})}{c^2 - c(d\boldsymbol{\varepsilon}) + (d\boldsymbol{\varepsilon})c - (d\boldsymbol{\varepsilon})(d\boldsymbol{\varepsilon})}$$

$$= \frac{ac + (bc - ad)\boldsymbol{\varepsilon}}{c^2}$$

$$= \frac{1}{c^2} * (ac + (bc - ad)\boldsymbol{\varepsilon})$$

Scalars can be distributed throughout a dual number that they are multiplying, so this division is now a form that we can calculate.

Dual Number Magnitude

The magnitude of a dual number, since it is a multiple-component number, must satisfy the following equation:

$$aa^* = |a|^2$$

Where a is a multiple-component number and a^* is its conjugate

This is just like the complex number, but the definition of the dual operator changes the result. The magnitude of a dual number $a + b\boldsymbol{\varepsilon}$ is as follows:

$$magnitude^2 = (a + b\boldsymbol{\varepsilon})(a - b\boldsymbol{\varepsilon})$$

$$= a^2 - a(b\boldsymbol{\varepsilon}) + b\boldsymbol{\varepsilon}(a) - (b\boldsymbol{\varepsilon})(b\boldsymbol{\varepsilon})$$

$$= a^2$$

Since a is a real number, then the square of the magnitude is also a real number, and therefore the magnitude is the square root of this number. This is quite simple, although it is not intuitive. I will not need to use this calculation, but I mention it to satisfy my own curiosity.

Dual Number Normalization

Divide both real components by the magnitude, which happens to just be the square of the real part. That's it.

Dual Number Square Root

This takes a little bit of equation manipulation. The square root of a real number is defined to be a real number. I don't know if the square root of a dual number is defined, but I defined the square root operation on a dual number dn to spit out another dual number as follows:

$$\sqrt{dn} = \sqrt{a + b\boldsymbol{\varepsilon}}$$

The result of this equation will be a dual number (because I said so and I'm experimenting here), so set it equal to another dual number as follows:

$$\sqrt{a + b\boldsymbol{\varepsilon}} = c + d\boldsymbol{\varepsilon}$$

$$a + b\boldsymbol{\varepsilon} = (c + d\boldsymbol{\varepsilon})^2$$

$$a + b\boldsymbol{\varepsilon} = c^2 + (cd + dc)\boldsymbol{\varepsilon}$$

$$a + b\boldsymbol{\varepsilon} = c^2 + (2cd)\boldsymbol{\varepsilon}$$

Since c and d are real numbers, then cd and dc are identical, so I can add them together into a single vale $2cd$ (or $2dc$, but I chose the former for alphabetical order). This would not be the case if they were quaternions, in which case $cd \neq dc$, but they aren't, so I'm fine. Now I solve for c and d.

$$c = \sqrt{a}$$

$$d = \frac{b}{2c}$$

Since a is a real number, the square root is defined, and once c is defined, I can then solve for d. The result is a dual number that should, if multiplied by itself, equals $a + b\boldsymbol{\varepsilon}$.

Now to test it against the mathematical definition of a square root: a square root times itself must equal the original number. Here's the check:

$$\left(\sqrt{a} + \frac{b}{2\sqrt{a}}\boldsymbol{\varepsilon}\right)\left(\sqrt{a} + \frac{b}{2\sqrt{a}}\boldsymbol{\varepsilon}\right) = a + (\sqrt{a})\left(\frac{b}{2\sqrt{a}}\boldsymbol{\varepsilon}\right) + \left(\frac{b}{2\sqrt{a}}\boldsymbol{\varepsilon}\right)(\sqrt{a}) + \left(\frac{b}{2\sqrt{a}}\boldsymbol{\varepsilon}\right)\left(\frac{b}{2\sqrt{a}}\boldsymbol{\varepsilon}\right)$$

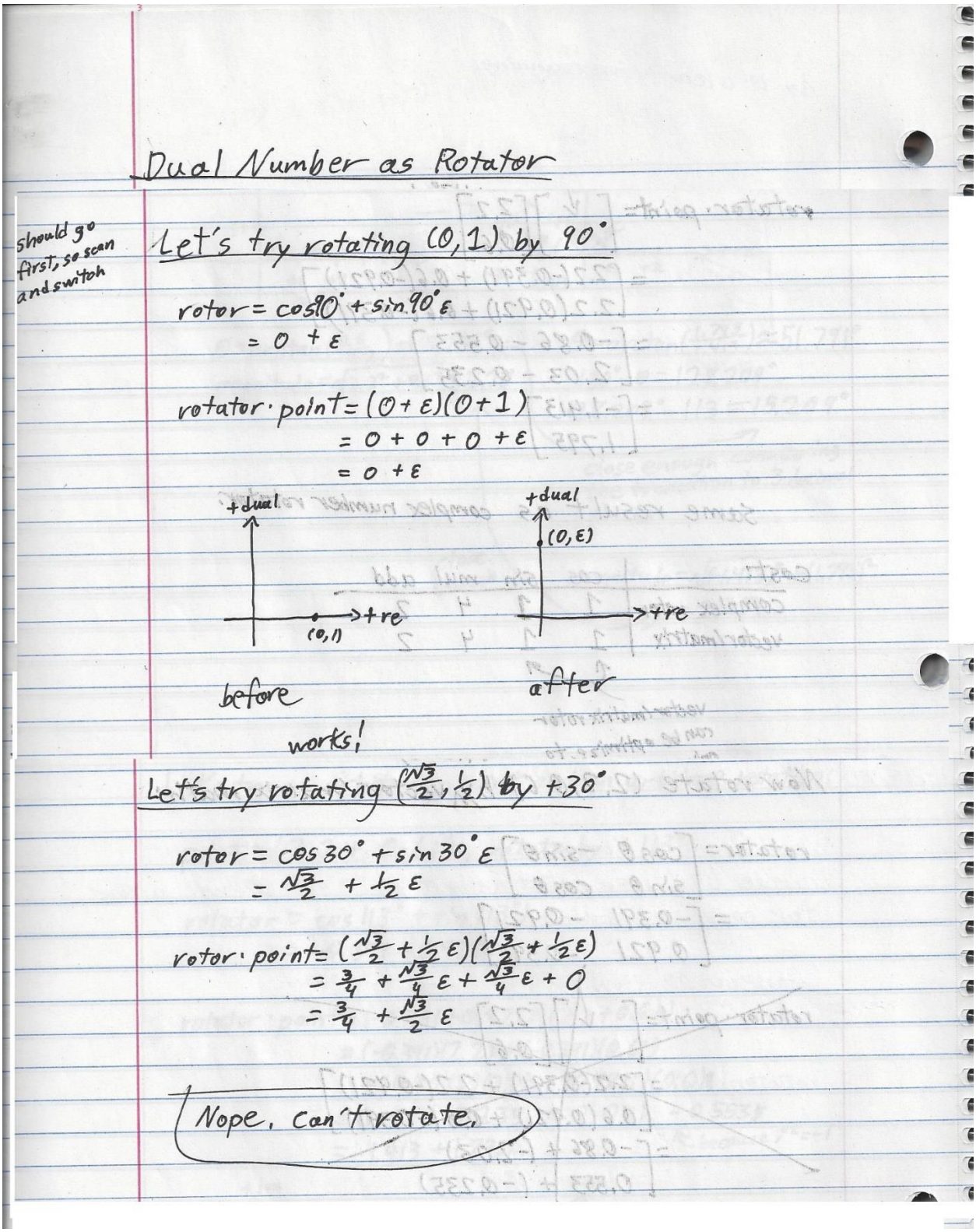
$$= a + \left(\frac{b}{2} + \frac{b}{2}\right)\boldsymbol{\varepsilon}$$

$$= a + b\boldsymbol{\varepsilon}$$

Excellent. My definition of the dual number square root satisfies the mathematical requirement of a square root.

Dual Number as a Rotor?

I'll try rotating (0,1) 90° for the trivial case. For the more general case, I'll rotate a point at the 30° mark of the unit circle by 30° to try to make a point at 60° on the unit circle.



So no. Dual numbers are not useful as rotors, but they are useful for separating two chunks of real numbers.

Movement in 3D Space

Which Direction is Positive Rotation?

Transformations in 3D space rotate around the three unit axes $\hat{x}, \hat{y}, \hat{z}$ and translate along them. In 2D space, positive rotation is known to be counterclockwise, but what about 3D space? In 3D space the same is true because rotations can only happen on a 2D plane, but you must know how to visual the plane so that you can understand how the rotations happen. Fortunately, the answer is quite simple.

Suppose you have some 3D vector and you want to rotate a point around it. The plane of rotation is correctly viewed by having the vector point at your face. Now imagine a plane that is orthogonal to this vector. Positive rotation in 3D space will be counterclockwise around this plane. That's all.

Try to reimagine the 2D plane with this in mind. In a right-handed coordinate system (index finger points along +X, middle finger points along +Y, thumb points along +Z), the 2D XY plane can be imagined as a 3D XYZ coordinate system in which +Z points directly at your face.

I was confused for a while with this concept when applied to the concept of an axis of rotation because an axis of rotation is two vector with the same base but pointing in opposite directions, so I couldn't figure out whether I was "looking down" the positive axis or the negative axis, which are vectors. This confusion was born out of a misunderstanding of the point of view of "looking down" a vector. Imagine you held a stick and were trying to describe to someone how to look at it a certain way. "Looking down" the stick carries no more information about orientation than looking down an axis. Either your face or the vector must be fixed for orientation to make sense. Vectors can change, so fix your face and fix the coordinate system as right-handed.

For example, you can rotate around the Y axis, but suppose you want to rotate by +87° around the Y axis. This concept doesn't make sense. You have to choose either the +Y axis (a vector) or the –Y axis (another vector pointing exactly the other way). So choose either the +Y vector or –Y vector, point it at your face, and the orthogonal plane to this vector will be the plane of rotation around which positive rotation will be counterclockwise.

Relative Transformation

Relative transformations (transforming (rotating and translating) a transform) are common in animation "bone" systems. There is a bone that is considered to be the root bone, and then all other bones are transformed relative to it. If a parent bone is transformed, then the child bone's own transform must be transformed by the parent's. It is common to make a new transform out of the previous transform and a newly calculated one, and then interpolate between them. Each step of the interpolation can be shown in a single frame, and we perceive the result as motion.

Corner-Case Problem of Shortest Path Interpolation with Matrices

In some situations, such as when a camera is looking straight up or down, the interpolation between the current camera orientation and the desired camera orientation may not result in a linear path. That is, it

doesn't take the shortest path, but instead a curve that can go noticeably out of the way and rotate undesirably. This non-linear path can result from a linear interpolation (??verify non-linearity from linear interpolation??) of matrix transforms simply because of how the math works out when transforming a matrix transform and interpolating between the old and the new. Mathematically, this effect is defined to occur when a matrix transform "loses" one or more degrees of rotation. I say "lose" because that is the common term, but I think that it is more descriptive to say that two columns of the matrix transform effectively undo each other in the matrix-vector multiplication (??verify??).

This effect is commonly called "gimbal lock", although I don't think that this term is entirely appropriate. Gimbals were used in conjunction with gyroscopes to measure orientation. Computer graphics and animation systems do not rely on gyroscopes though and are not subject to locking up in the same way. Computer graphics programs are state machines. New orientations are recalculated or interpolated every frame. Nothing locks up in matrix transforms because the matrix represents a single state of rotation and translation. The math for generating a new matrix transform still works, but interpolating between a transform that has "lost" one or more degrees of freedom and a transform that has not may result in a highly undesirably path. Therefore this problem is better called "The Problem of Shortest Paths Between Relative Transforms", but I have been overruled by tradition.

That being said, Gorilla CG has an excellent video on "gimbal lock" as it refers to 3D animation[5].

The Gimbal

"Gimbal lock" is a common topic in 3D graphics and is the primary reason that people go through the trouble of switching their transforms from matrices to quaternions. Before I talk about gimbal lock in 3D animation though, I want to describe what a "gimbal" is and what happens when it "locks up".

A gimbal is a three-axis gyroscope that used to be commonly used in Inertial Measurement Units (IMUs) for analog flying machines like aircraft, rockets, and early spacecraft, including those in the Apollo program. In the movie Apollo 13 there were several conversations about gimbals. Shortly after launch the pilot Jim Lovell stated that the gimbals were good. Shortly after the O2 tank exploded Jim reported that they were getting a lot of thruster activity, Jack Swigert stated that the way that the thrusters were firing didn't make any sense, and shortly after that an engineer on the ground that the craft was all over the place and that they kept getting close to gimbal lock. A little later they tried to correct the Apollo craft's orientation manually, and Jim reported that the craft was fighting his input, saying that they kept flirting with gimbal lock. There were a few other references in the film, and you can find them in do an online search for the Apollo 13 script and search for "gimbal".

What were these gimbals that they were talking about? If you search online for gimbals you can see animations of multiple rings spinning inside each other like a fancy gyroscope, but that isn't what was used in these older IMUs. **Put simply, "a gimbal is a pivoted support that allows the rotation of an**

[5] https://www.youtube.com/watch?v=zc8b2Jo7mno (Gorilla CG Project video on gimbal lock as it pertains to 3D animation)

object about a single axis"[6]. I'll take this thing apart for you a bit so that you can have the knowledge necessary to understand what it means for a gimbal to "lock".

Aircraft IMU for Pitch and Roll

I found a video online detailing a breakdown of an old aircraft "attitude"[7] IMU[8]. This IMU would rotate about the craft's roll axis and had a single spinning ring inside that would detect any change in pitch. This IMU appears to have been constructed with a classic gyroscope mechanism in which a single platform was mounted at the center of the IMU that would hold the spinning mass, and the other parts of the IMU responded to any change in the craft's attitude by rotating axles connected to the platform in such a way that the platform remained level at all times. The outer shell of this IMU was the outer gimbal and the support structure for the pitch-detecting spinning metal ring was the inner gimbal.

How did it detect any rotation? The breakdown in the video found a couple of magnetic field detectors made of very fine wire. When the aircraft rolled, the IMU would start to roll with it, but then a magnetic field sensor would pick up the rotating metal and cause a circuit to drive the IMU's internal motors to rotate it back. There was another magnetic field sensors for the spinning gimbal, which was used to detect changes in pitch and rotate the axles for the stationary platform such that it would remain level regardless of pitch. That's some pretty nifty old tech, but it is only in two dimensions (pitch and roll).

IMUs for 3D Space - Three-Axis Gimbal

Three-axis gimbals measure pitch, roll, and yaw. They have been around for a long time and originally came into popularity in submarines (??verify??).

Relevant side-track: Sensory deprivation is a method to deprive the body of its sense of orientation. This is performed by having a subject float in water that is the same temperature as the body in a container that is insulated from light and sounds. Subjects report loosing their sense of orientation until they move and touch the container's sides.

Back to gimbals: Navy submarines are entirely blind (visually speaking) and they float underwater with a couple miles or more between them and the ocean floor. These are large machines with no sensors that can penetrate the water to give them a sense of orientation and location without giving their position away. This is a similar situation to sensory deprivation. Solution: a three-axis gimbal. Similar to the two-axis IMU on the aircraft, the three-axis gimbal has a platform with spinning metallic masses. Instead of magnetic sensors though, it may use torque sensors instead, but the effect is the same: any change in the orientation of the IMU will induce a torque on one or more of the spinning masses, and the circuitry will drive the gimbal motors in order to keep the platform stable. Although since replaced with other sensors (??verify??), three-axis gimbals allowed submarines to track their precise location in the ocean by tracking their every change in orientation.

---

[6] http://en.wikipedia.org/wiki/Gimbal

[7] "Attitude" (*not* "altitude") in aircraft, spacecraft, and sea craft parlance refers to the craft's orientation in space.

[8] https://www.youtube.com/watch?v=AO7pn3uiWA0

The Apollo IMU

Then comes the space program, and now they need to precisely keep track of their orientation in space without access to gravity, which was useful for simplifying the gimbal (??verify??). Now they need to make a lightweight and low-power gimbal to launch into space. In typical engineer fashion when they have all the money, time, and resources in the world to tinker with things, the engineers figured it out. They decided to design an early semi-automatic guidance system into the Lunar Excursion Module (LEM, later shortened to Lunar Module (LM)). They recognized that they could run into "gimbal lock" in space since there would no longer be gravity around to keep the crew "right-side up", which becomes a very relative concept in space.

Development

I found a 1968 video in which a NASA engineer completely schooled me on this early IMU[9]. The video describes a then-prototype two-axis IMU for a spacecraft. At the 4 minute mark in the video, the narrator shows how a single-axis gyro could be used to reorient a satellite. At the 8 minute and 30 second mark he begins explaining how their prototype two-axis Control Moment Gyro worked (read, "gyro with torque-feedback and control respondent mechanisms"). At about 9 minutes and 30 seconds he begins talking about the Apollo Telescope Mount which was then yet to be built and launched.

The Apollo Telescope Mount was to be the first long-term manned spacecraft that would use a CMG. It would use three of them mounted on a rack that also supported the telescope experiment package, which in turn was pointed towards the sun. It didn't need a three-axis IMU because it would be set in orbit around the Earth in such a way that it wouldn't roll (??verify??). Initially the equipment would be required to control the telescope package to within 1/10th of a degree, but advancements in CMG technology would eventually yield control accuracies to within 1/1000th of a degree. This technology would be used to control the 85 ft-long spacecraft using the power equivalent of ~300W, or about three 100W incandescent light bulbs. That's an impressive low-power control system.

The Apollo program later developed a three-axis IMU for use in the manned lunar module. That craft needed to roll, pitch, and yaw, so it required a three-axis IMU. Three-axis IMU were known to be subject to gimbal lock if allowed to pitch, roll, and yaw in any combination in free space, and four-axis IMUs (mentioned again at the end of this section) were known to avoid gimbal lock altogether, but the three-axis IMU was lighter, less expensive, easier to build, and perfectly suitable provided that the pilot of the spacecraft didn't roll, pitch, or yaw craft in certain combinations.

I found a paper describing how the Apollo engineers decided to avoid this problem of gimbal lock[10], and in it they provide a picture and explanation of how the IMU worked. Here's the IMU schematic:

[9] https://www.youtube.com/watch?v=kKIBGY2zaQg
[10] https://www.hq.nasa.gov/alsj/e-1344.htm

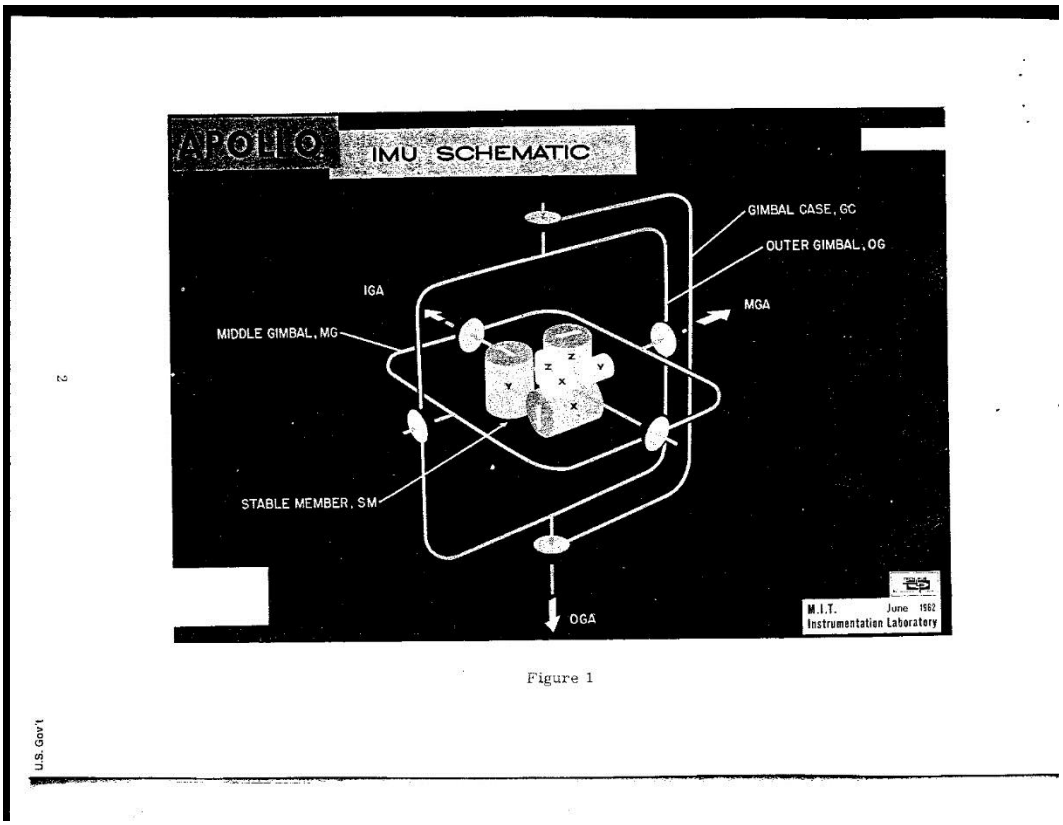


Figure 1 - https://www.hq.nasa.gov/alsj/imu-1.htm

The three gimbal axes are labeled IGA (Inner Gimbal Axis), MGA (Middle Gimbal Axis), and OGA (Outer Gimbal Axis). The six disks, two on each gimbal axis, are electric motors that will counter any rotation of the outer case (labeled in upper right of picture). The gimbal case is drawn here as a half-ring that is connected to the OGA. The gimbal case is connected to a full-ring gimbal that is called the "outer gimbal". Another full-ring gimbal is attached to it, and this one is called the "middle gimbal". There is no "inner gimbal" in this setup, but the middle gimbal does have a rotating axis that goes through the center platform (called the "stable member"), which contains the spinning masses and their torque sensors. This is the only axis that is connected to it, but the middle gimbal can rotate perpendicularly to it to counter another axis of rotation, and the outer gimbal is perpendicular to that to counter another axis. Therefore you might expect this three-axis gimbal to be able to keep the center platform stable in any orientation.

But it doesn't. Not in all situations, at least.

How It Worked

Check out this excerpt from the Apollo paper that described how the IMU (the IMU containing the gimbals and stationary platform in the picture above) was mounted, its definition of gimbal lock for this system, and how to avoid it:

The naming of the LEM body axis must be emphasized. Since the crew sits with their backbone along the thrust axis in the LEM (unlike an airplane where they look towards the thrust axis) there is possibility of confusion until the convention is firmly fixed in mind: In an airplane, in the command module, and in the LEM the roll axis is defined parallel to the thrust axis.* Thus in the stationary hover orientation, the LEM roll axis is vertical with respect to the local surface of the moon. The crew looks forward in the direction of the yaw axis.

*This definition of LEM roll axis was given to the writer by APO at MSC.

The IMU will probably be mounted with the outer gimbal axis parallel to the LEM roll axis. The inner gimbal axis will be aligned and inertially stabilized in a direction perpendicular to the landing or takeoff trajectory plane. The vehicle is at zero roll when the yaw axis is in this plane.

GIMBAL LOCK OCCURS WHEN THE LEM THRUST AXIS IS POINTED ALONG THE SPACE DIRECTION OF THE INERTIALLY STABILIZED INNER GIMBAL AXIS WHICH IS ALIGNED HORIZONTAL AND PERPENDICULAR TO THE PLANE OF THE LANDING OR TAKEOFF TRAJECTORY.

Gimbal lock is avoided by a wide margin, then, by attitudes constrained as follows:

IMU GIMBAL FREEDOM IN THE LEM PERMITS:

a. ANY ROLL ANGLE

b. ANY PITCH ANGLE AT ZERO ROLL

c. ANY YAW ANGLE AT 90' OR 270' ROLL

Well that's a mouthful, and I'll break it down in a moment, but here's the gist of it: **gimbal lock occurs when the outer gimbal's axis and the inner gimbal's axis are parallel or nearly parallel**. The middle gimbal is physically mounted to the outer gimbal, so the outer and middle gimbal axes can never be parallel. This leaves the outer and inner gimbals. The gimbals can be oriented any which way and even be on the same plane, but as long as the outer and inner gimbals' axes of rotation are not parallel or nearly parallel, then the IMU works just fine.

Now for the breakdown of the pitch, roll, and yaw axes. I'm sure that you can imagine the LEM landing on the moon because you have probably seen many pictures of it since you were a kid. If the LEM is landed on a surface, then the astronauts are sitting down relative to the landing surface. This seating position was defined to look along the yaw axis of the craft, which means that, if they were to yaw left, then they would tilt to the left, and if they were to yaw right, then they would tilt to the right. Roll is defined to be along the thrust axis of the craft, and remember that the rotation through space of positive rotation is dependent upon what vector is pointing at your face. If the crew is seated and

looking straight ahead from their seats, then the roll axis is along their backbone and their faces are perpendicular to the roll axis, so neither of the +/- roll vectors are pointed at their faces. Result: rolling the LEM would seem to an aircraft pilot like he was yawing the craft, while yawing would seem like rolling. This would take some getting used to, and it was critical that they *did* get used to it so that they would understand how to avoid locking the IMU.

The pitch axis is simpler. It is in the same plane as the yaw axis, so if they were to pitch down, then they would lean forward, and if they were to pitch up, then they would lean back. No confusion there.

The inner gimbal axis is aligned perpendicular to the landing or takeoff trajectory plane. The Apollo craft, being spacecraft, would launch up into the air from Earth, land on the moon, launch up from the moon, and splash back down in the Earth's ocean. These takeoff and landing trajectories are not vertical, but rather carefully planned to take advantage of each bodies' gravity in order to build up speed while still in the gravity well and "slingshot" the craft toward its next target. The takeoff and landing trajectories could be described in a 2D plane, and the inner gimbal axis had to be aligned perpendicular to this plane. For example, suppose that the takeoff trajectory from Earth was directly to the West. Then the inner gimbal axis would be aligned North-South.

However, while the craft's roll axis and the outer gimbal's axis are defined to be aligned, the pitch and yaw axes are *not* defined to be aligned with any axis of the craft. The inner gimbal is aligned perpendicular to a trajectory *plane*, which would change at different parts of the mission. The inner and middle gimbal axes are on the same plane, so these axes were on the same plane as the craft's pitch and yaw axes, but the axes do not necessarily align. In fact, they probably didn't (??verify??) so that pitching or yawing the craft wouldn't trigger only the middle gimbal's motors to rotate the outer gimbal's axis into being parallel with the inner gimbal's axis.

This IMU was used to keep the craft on course once set. When 150-180lb men are hopping around the spacecraft and pushing off walls, then Newton's Second law says that the spacecraft will rotate as a result. The IMU could keep the LEM on course between the Earth and the moon and vice-versa. The IMU was designed to be reset mid-flight using known star charts, and it *had* to be realigned for moon-landing missions.

Locking the Apollo IMU's Gimbals

Examine the IMU gimbal freedoms permitted and the schematic of the IMU from the previous section. The IMU casing is attached to the outer gimbal, and the outer gimbal's axis of rotation is aligned with the LEM's roll axis. **Attitude Permit #1**: Any roll angle is permitted at any yaw and any pitch. Roll only rotates the outer gimbal and does not change where its axis is pointing, so you can roll all day and you would not alter the middle or inner gimbals.

Attitude Permit #2: Any pitch angle is permitted at zero roll. Why zero roll? My guess is that it was just to be safe. Slight roll was acceptable. Zero roll was defined to be when the yaw axis was in the same plane as the inner gimbal's axis, and they likely had some instrument for this. At zero roll, then pitching the craft would likely cause both the middle and inner gimbal axes to react to keep the platform stable. The outer gimbal would therefore rotate around the pitch axis and would likely come into the same plane as the inner gimbal axis, but as long as the two axes were not parallel, then the IMU worked fine.

However, if they were not at zero roll, then the pitch axis may have been parallel with the middle gimbal axis, and pitching the craft in this scenario might make the outer gimbal axis parallel to the inner gimbal axis. Gimbal lock!

Example: Suppose that, at zero roll, the inner gimbal axis is parallel with the pitch axis (??TODO: could it have been??). At zero roll, you pitch the craft forward. The inner gimbal rotates to keep the platform stable at its original attitude, and the middle and outer gimbal axes rotate around the inner gimbal's axis. No problem here.

Example: Suppose that you are now at 90° roll. Now the craft's pitch axis is parallel to the middle gimbal's axis. Now you pitch the craft forward °. Now the outer gimbal's axis will be parallel with the inner gimbal's axis. If you keep pitching forward then the axes will no longer be parallel and you will be fine, but do *not* roll 90° and then stop and try to roll or yaw. I'll cover why in the next section.

Example: Suppose that the pitch axis is parallel to the middle gimbal axis. You're screwed. Roll the craft so that this is not the case.

Attitude Permit #3: Any yaw angle at 90° or 270° roll. Again, these designations were just to be safe. A few degrees off these two numbers was acceptable. Remember that any pitch angle is permitted at zero roll. The yaw axis is perpendicular to the pitch axis, so if you rotate 90° or 270°, then your yaw axis will be parallel to where the pitch axis would be if you were at zero roll. Now you can yaw all day and the outer and inner gimbal axes will never be parallel.

Why Are Parallel Gimbal Axes Bad?

The IMU's stable platform contains three spinning masses that are perpendicular to each other. Torque sensors detect any induced torque on theses masses caused any change in the craft's attitude and rotate the gimbal associated with the affected spinning mass or masses. If the outer and inner gimbal axes are parallel, then two of the spinning masses will now rotate the same axis and the auto-stabilization system will chaotically (read, "small change in input => big change in output") compensate for the two axes that are now trying to simultaneously stabilize the input commands from the pilot and undo the other's rotation and keep the platform stable.

This is why we say that the gimbal is "locked". The outer gimbal axis, due to the mechanical nature of the three interacting gimbal axes, is physically incapable from breaking away unless the pilot figures out the pitch-yaw combination of the craft that corresponds to the middle gimbal axis and rotates around that axis. Rotating around that axis will only trigger a reaction from the third spinning mass on the platform and not affect the inner or outer gimbals.

The easier way out of this is to reset the IMU.

What Happened to the Apollo 13 IMU?

I don't know for certain (??TODO: figure it out??), but my guess is that the explosion of the craft's O2 tank caused the craft to pitch, roll, and yaw in a particularly unfortunate way that resulted in the outer and inner gimbal axes almost becoming parallel. Recall the excerpts from the Apollo 13 movie script that I mentioned earlier. They didn't hit gimbal lock, but they did get awfully close. The craft's auto-stabilization was firing thrusters chaotically because the two gimbal axes were nearly parallel, so the spinning masses and their torque-sensing mechanisms for two of the gimbals were controlling nearly

the same axis and the gimbal motors were fighting themselves. The craft's roll axis was now nearly locked up with the IMU's idea of pitch and yaw (they are on the same plane).

The control circuits for the gimbal motors also signaled the thrusters to fire, so the IMU was firing the thrusters to try to stabilize the craft, which unfortunately just made the situation worse. The pilot Jim Lovell tried to take manual control of the craft, but his input to the thrusters didn't force the outer gimbal's axis out of being parallel to the inner gimbal axis. He reported that the craft was fighting his input, which is expected if the IMU gets roll mixed up with yaw and pitch (??TODO: how did the pilot's input work with respect to the craft's IMU and thrusters? Did he control thrusters directly or the IMU??).

Their solution? Switch off the power, turn it back on, and realign it.

Another Way Out: The Four-Axis IMU

The four-axis IMU is much like a three-axis gimbal, containing four gimbals rotating around 4 axes, each with two motors to rotate the gimbal. The four gimbals still rotate, like in a three-axis IMU, such that the center platform, which contains 4 spinning masses and 4 torque sensors, remains stable at its origin attitude. It is designed so that no two gimbal axes will ever be able to be parallel. This makes it a more complex piece of machinery than a three-axis IMU, but it avoids gimbal lock.

Gimbal Lock (Mechanically) to Gimbal Lock (Animation)

Gimbal lock is defined mechanically as when the inner and outer gimbal axes in an IMU become aligned and, due to the design of the IMU's auto-stabilizing mechanisms, is very difficult to break out of unless you turn it entirely off and reset it without power. This is where the term "lock" originally came from.

Gimbal lock is defined, mathematically, as when a map from Euler angles to rotations is not a covering map[11]. That is, in 3D space, for all possible rotation combinations around the three base axes $\hat{x}, \hat{y}, \hat{z}$, not all of those combinations can be reached.

Gimbal lock is defined in animation as not taking the shortest (read, "linear") path between two transforms when linearly interpolating between them.

Here's how these concepts are related: Mechanical gimbal lock effectively results in losing a degree of freedom, which means that a matrix transform that describes this locked state cannot always take a linear path to another transform. The locked transform, while being only a transform and therefore only covering one specific combination of Euler angles about $\hat{x}, \hat{y}, \hat{z}$, it cannot take a linear to all other possible combinations of Euler angles about $\hat{x}, \hat{y}, \hat{z}$. If you still don't understand, then shelve this problem as "math black magic" and we'll move on.

Quaternions

The website 3DGEP's article on Understanding Quaternions covers this material pretty well, but I want to cover more conceptual detail and go through the computational cost in comparison to matrices.

Conceptually, a quaternion is a mathematical representation of a four-axis gimbal. It isn't a perfect analogy because the four-axis gimbal exists in three dimensions and the quaternion does not because

[11] http://en.wikipedia.org/wiki/Gimbal_lock#Gimbal_lock_in_applied_mathematics

the quaternion contains four orthogonal number lines, so it only exists in four-dimensional space. Don't try to imagine it. You'll just hurt yourself like I did.

Quaternions are also the solution to the problem of relative transform interpolation not taking the shortest path. I honestly don't fully understand how quaternions were developed. To fully understand them, I would have to recreate a couple-year period of a dead mathematical genius by the name of William Rowan Hamilton, and I don't want to do that. They work though, they avoid the "gimbal lock" problem, and they are less expensive computationally to create and transform in comparison to matrices. Their downside is that they are much more computationally expensive than matrices when transforming a vector, and I will describe this in more detail later.

The Revelation

The story goes that William Hamilton was work on the problem of three dimensional orientation when he had a revelation:

$$\boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = -1$$

and

$$\boldsymbol{ijk} = -1$$

He was so excited about this discovery that, before he forgot it, he scratched it into the stone of a bridge that he was walking over at the time while he and his wife were on a walk. This was before the invention of the sticky note and the pencil and the ball-point pen, so I give him kudos for resourcefulness. The equation is now carved into that bridge as a monument to his discovery.

"But gee, Batman," you say, "What does this mean?" It is clever math, Robin, and not math that most mortals are gifted with understanding. You already know of the imaginary unit ***i***, but (and this blew my mind) ***j*** and ***k*** are also imaginary unit numbers on their own imaginary number lines, so ***i***, ***j***, and ***k*** are orthogonal. Hamilton's revelation was to make a 3D imaginary coordinate system, but since the square of each imaginary unit is -1, there is a real number component as well.

And what about the $\boldsymbol{ijk} = -1$ part? I'll get to that shortly.

In a 2D complex number, the real number component and the imaginary component are independent real number lines, so they are considered orthogonal. Similarly, the real, I, j, and k number lines are also orthogonal. In 3D space, ***i*** represents the X axis, ***j*** represent the Y axis, and ***k*** represents the Z axis. The multiplication of the three imaginary unit numbers is as follows:

$$ij = k, ji = -k$$

$$jk = i, kj = -i$$

$$ki = j, ik = -j$$

You do not need to memorize them. I show these multiplication relations to show that they create a simple way to get the cross product between two unit axes. You now know that ***i*** is analogous to the unit axis $\hat{x}$, ***j*** is analogous in $\hat{y}$, and ***k*** is analogous to $\hat{z}$, so compare the imaginary unit multiplications to the axis unit vector dot products in a right-handed coordinate system:

$$\hat{x} \times \hat{y} = \hat{z}, \hat{y} \times \hat{x} = -\hat{z},$$

$$\hat{y} \times \hat{z} = \hat{x}, \hat{x} \times \hat{y} = -\hat{x},$$

$$\hat{z} \times \hat{x} = \hat{y}, \hat{x} \times \hat{z} = -\hat{y},$$

And that's cool. But you don't need to memorize this.

And now about the $\boldsymbol{ijk} = -1$ part. This says that there is a real number line that is orthogonal to all three imaginary unit numbers. That is all. Multiplying any two imaginary unit numbers gives a unit number that is orthogonal to both via the right-hand rule, but multiplying all three imaginary units together gives a unit number that is orthogonal to all three, which is the real number line. We won't run into such a multiplication with quaternions or dual quaternions, but now you know what it means. It bugged me when I didn't know it.

Definition of a Quaternion

Note: I will often abbreviate quaternions as "q" or "quat" in my equations.

A quaternion q is defined by the following:

$$q = w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$$

Where w, x, y, and z are numbers and ***i***, ***j***, and ***k*** are the three orthogonal imaginary unit numbers.

When w, x, y, and z are real numbers, then "q" is called a "real quaternion". This is what most people mean when they say "quaternion", and it is what I mean when I say it. This notation differentiates it from the "dual quaternion", which I will discuss later.

The real number "w" is on the real number line and is commonly called "the scalar". It will always start as either a 1 or a 0, but it will change as the quaternion is multiplied by other quaternions. It represents nothing in 3D space and exists for mathematical book keeping reasons, namely that the square of any unit number results in a new number on the real number axis, and that number can't disappear. Fortunately, the math all works out, so don't fret over exactly what it means. The math works out, and that's the main thing that matters.

Notice that x, y, and z do not have the hat or vector symbols over them. These are scalars to the unit vectors ***i***, ***j***, and ***k***, much like a 3D vector's X, Y, and Z values are scalar multiples of the unit vectors $\hat{x}$, $\hat{y}$, and $\hat{z}$.

$$vector = < x, y, z > = x\hat{x} + y\hat{y} + z\hat{z}$$

$$similarly, q = 0 + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$$

The imaginary components of the quaternion are together identical in scale and function to vectors, so a quaternion is often represented as a scalar and a vector as follows:

$$q = w + \vec{v}$$

Or more commonly:

$$q = [w, \vec{v}] \text{ or } q = (w, \vec{v})$$

Both of these are much nicer and more manageable notation than the additive notation.

Could the Quaternion Concept Be Defined with Three Values Only?

Why use four numbers to represent a rotation (or a point) in 3D space? Why not have something like the following?

$$trinernian = x + y\boldsymbol{i} + z\boldsymbol{j}$$

Where *trinernian* is a made-up word based on the "quat" part of "quaternion", x, y, and z are real numbers and ***i*** and ***j*** are imaginary units orthogonal to the real number line and to each other.

This definition is a simple expansion of the complex number and only adds the ***j*** term. Why not try this?

I'm sure that Hamilton did, and he found that it didn't work. In 3D space you need to be able to find orthogonal vectors via a cross-product or cross-product-like operation. The vectors that these operations spit out depend on the order of the input vectors. So just as $\hat{x} \times \hat{y} = \hat{z}$, so also $\hat{y} \times \hat{x} = -\hat{z}$.

Let's try to get the + and – unit axes by defining ***i*** and ***j*** to be orthogonal to each other and to the real number line, and remember that the unit numbers for the real number line are simply 1 and -1.

$$ij = 1, ji = -1$$

$$i1 = i, 1i = i$$

$$j1 = j, 1j = j$$

Well…that failed. We can't get + or – ***i*** and ***j*** this way, so this is no way to represent a rotation (or a point) in 3D space. We need ***k***, and that realization was Hamilton's great discovery that lead to quaternions.

Special Quaternion Names: Real and Pure

A quaternion is commonly thought of as having a scalar component and a vector component. The number is technically four real numbers of four orthogonal number lines, but three of them behave like the X, Y, and Z vectors in 3D space, so they are collectively labeled as the vector component.

There are special names when one of these components are zero.

$$real\ quaternion = (w, 0)$$

$$pure\ quaternion: (0, \vec{v})$$

The real quaternion is equivalent to 1. The vector component, while 0, is still written to satisfy the notation of the quaternion, but its value is merely 1. Multiplying anything by a real quaternion gives the original value because it is merely 1. I don't know why this has a special name, but it does.

The pure quaternion is more useful and is used to represent a point vector. This is covered just before the Quaternion as a Rotor section.

Quaternion Addition

Two quats q1 and q2 are added as follows:

$$q_2 + q_2 = \left(q_{1w} + q_{1x}\boldsymbol{i} + q_{1y}\boldsymbol{j} + q_{1z}\boldsymbol{k}\right)$$

$$+\left(q_{2w} + q_{2x}\boldsymbol{i} + q_{2y}\boldsymbol{j} + q_{2z}\boldsymbol{k}\right)$$

$$= (q_{1w} + q_{2w}) + q_{1\vec{v}} + q_{2\vec{v}}$$

You can only add two numbers that are one the same number line, so the real numbers are added and the vector components are added to make a new scalar and a new vector.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Quaternion Addition	0	0	0	0	4	0	4

Breakdown:

- Add/Sub: hopefully obvious
- Temporary floats
  - 1 temporary float for each of operator (4 total)
  - Return-value optimization in use, so constructor cost is discounted

Quaternion Multiplication

Two quats q1 and q2 are multiplied as follows:

$$q_2 * q_2 = \left(q_{1w} + q_{1x}\boldsymbol{i} + q_{1y}\boldsymbol{j} + q_{1z}\boldsymbol{k}\right) * \left(q_{2w} + q_{2x}\boldsymbol{i} + q_{2y}\boldsymbol{j} + q_{2z}\boldsymbol{k}\right)$$

The first multiplication of everything will create 16 items, and I don't want to go through all those subscripts, so I will instead show you the pattern that results when all the imaginary unit multiplication is done:

$$q_2 * q_2 = (q_{1w}q_{2w} - dot(q_{1\vec{v}}, q_{2\vec{v}}), q_{1w}q_{2\vec{v}} + q_{2w}q_{1\vec{v}} + cross(q_{1\vec{v}}, q_{2\vec{v}})$$

This is nice because the dot and cross product functions are readily available in the GLM library.

I encourage you to work this out yourself on a scratch pad. I found it a bit enlightening to see that the results of the 16 multiplications and resolving the axis switching stemming from the multiplications of $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$ with each other resulted in the above equation. It isn't magic. It's just some tedious math ☺.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Multiplication	0	0	16	0	13	0	29

Breakdown:

- Multiplication
  - 1 from scalar multiplication
  - 3 from dot product
  - 6 from scalar * vector

    - 6 from cross product
- Add/Sub
    - 1 from scalar + dot product
    - 3 from dot product
    - 3 from vector addition of $q_{1w}q_{2\vec{v}} + q_{2w}q_{1\vec{v}}$
    - 3 from vector addition of adding $q_{1w}q_{2\vec{v}} + q_{2w}q_{1\vec{v}}$ to the result of the cross product
    - 3 from cross product
- Temporary floats
    - 1 from each operator (29 total)
    - Return-value optimization in use, so constructor cost is discounted

Note: I was confused for a little bit why there were only 13 additions, but then I remembered that there are four linearly independent real numbers lines in the result, so some of the terms will not add together.

Quaternion Division

3DGEP's article "Understanding Quaternions" covers this, so I defer to them.

Quaternion Conjugate

Suppose we have a generic quaternion q as follows:

$$q = w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$$

Then the conjugate is often represented with the notation q* ("q star") and is defined as follows:

$$q^* = w - x\boldsymbol{i} - y\boldsymbol{j} - z\boldsymbol{k}$$

The conjugate can be used in calculating the square of the magnitude of the quaternion and when rotating a point.

Multiplying a quaternion by its conjugate will yield the sum of the squares of w, x, y, and z. You can tell, just from looking at q and q* that you will get a w^2, but notice that you will also get $(x\boldsymbol{i})(-x\boldsymbol{i})$, and since $\boldsymbol{i}^2 = -1$, then the result is positive x^2. Ditto for y and z. All other components will cancel out.

This is a convenient programming shortcut to calculating the square of the magnitude.

Optimization: Manually square each component.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Conjugate	0	0	3	0	0	0	3

Breakdown:

- Multiplication: multiplying the three imaginary coefficients by -1
- Temporary Floats:
    - 1 for each operator (3 total)

- Return-value optimization in use, so the quaternion constructor cost is discounted

Quaternion Magnitude

Magnitude Squared

The magnitude of a quaternion, since it is a multiple-component number, must satisfy the equation:

$$aa^* = |a|^2$$

Where a is a multiple-component number and a^* is its conjugate

The magnitude of a quaternion q works out as follows:

$$magnitude^2 = qq^*$$

$$= (w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k})(w - x\boldsymbol{i} - y\boldsymbol{j} - z\boldsymbol{k})$$

I won't multiply this out here because quaternion multiplication done the long way results in 16 multiplications, but I will show you the result:

$$magnitude^2 = (w^2 + x^2 + y^2 + z^2)$$

This is a real number, so the magnitude is the square root of this number.

Optimization: When programming, skip the multiplication of the quaternion by its conjugate and write a non-static class method to square the components individually before taking the square root.

I want to mention that the magnitude of a quaternion is only on the real number line. Just as squaring the components of 2D r 3D vector or a complex number for the magnitude calculation makes no mention of the axis unit vectors, so the quaternion's magnitude calculation makes no mention of the imaginary unit numbers. This is one calculation in which the unit numbers don't move about with their coefficients.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Magnitude (quat * $quat_{conjugate}$)	0	0	19	0	13	0	36
Magnitude (optimized)	0	0	4	0	3	0	7

Breakdown (quat * $quat_{conjugate}$):

- Multiplication
  - 3 from conjugate creation
  - 16 from quaternion multiplication
- Add/Sub
  - 13 from quaternion multiplication
- Temporary Floats
  - 3 from conjugate operators
  - 4 from constructor of temporary conjugate quaternion

    - 29 from quaternion multiplication operators

Breakdown (optimized):

- Multiplication: 1 for each component multiplied by itself (4 total)
- Add/Sub: 4 multiplication results added together (3 total)
- Temporary Floats:
    - 1 for each operator (7 total)
    - Return-value optimization in use, so the quaternion constructor cost is discounted

Quaternion Magnitude

The magnitude squared is a real number, so take its square root to get the magnitude.

Quaternion Normalization

Divide each real component by the quaternion's magnitude.

Optimization: Avoid unnecessary and expensive division by diving 1 by the magnitude, then multiply each real component by the inverse magnitude. The resulting 1 division and 4 multiplications are computationally cheaper than 4 divisions.

Note: I am assuming that the optimized magnitude-squared calculation is in use.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Normalization	-	-	8	1	3	1	13

Breakdown:

- Multiplication:
    - 4 from quaternion magnitude squared
    - 4 from multiplying through the inverse magnitude
- Division:
    - 1 from calculating inverse magnitude
- Add/Sub:
    - 3 from quaternion magnitude squared
- Square Root:
    - 1 from quaternion magnitude
- Temporary Floats:
    - 1 for each operator (13 total)

Quaternion Dot Product

3DGEP's article "Understanding Quaternions" covers this, so I defer to them. It is useful for finding the cosine of the angle of the shortest path between two normalized quaternions, just like the dot product between two normalized 3D vectors finds the cosine of the angle of the shortest path between them.

I don't use it though (not yet, at least), so I'm not covering it.

Disguising a Point Vector as a Quaternion

If I want to transform a point vec3 with a quaternion, then I will have to convert the point into a quaternion first. Here's how it's done:

Suppose you have a point $< 1, 2, 3 >$. Then the quaternion form of the point would be a pure quaternion as follows:

$$point\ (in\ Euclidean\ space) = 1\hat{x} + 2\hat{y} + 3\hat{z}$$

$$point\ (in\ imaginary\ space) = \ 1\boldsymbol{i} + 2\boldsymbol{j} + 3\boldsymbol{k}$$

$$= 0 + \ 1\boldsymbol{i} + 2\boldsymbol{j} + 3\boldsymbol{k}$$

$$(0, < 1, 2, 3 >)$$

That's all. Now you can multiply a quaternion rotator by this point to rotate it.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Creating a Point	0	0	0	0	0	0	0

No temporary floats required. Utilize return-value optimization and just shove the vector values into their corresponding places in the quaternion's imaginary components.

Generating a 3D Rotor: Quaternion Approach

Disclaimer: I don't know why the following works, but the math works out. I suspect that it was partially figured out by tinkering with the equation and messing with values until it worked. Or maybe some brilliant dead math dude had an epiphany and it worked. I have no conceptual explanation.

Recall that a complex number rotor is calculated as follows:

$$complex\ rotor = \cos(\theta) + \sin(\theta)\,\boldsymbol{i}$$

A quaternion is calculated in a similar, but not exact way:

$$quat\ rotor = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}\right)x\boldsymbol{i} + \sin\left(\frac{\theta}{2}\right)y\boldsymbol{j} + \sin\left(\frac{\theta}{2}\right)z\boldsymbol{k}$$

$$= (\cos\left(\frac{\theta}{2}\right), sin\left(\frac{\theta}{2}\right)\hat{v})$$

In this equation, x, y, and z describe the *normalized vector* of rotation, hence the $\hat{v}$ instead of $\vec{v}$. **It is important for the calculations that the rotation vector is normalized.**

This is 3D space, and although it is conceptually acceptable to say, "rotate around an axis", it is *not* acceptable to put an rotation axis into the quaternion for two reasons: (1) An axis cannot be normalized because, by definition, it extends infinitely in two directions and therefore has no magnitude to begin with, and (2) positive rotation is only defined for rotation around a vector because, again, an axis points in two directions and therefore positive rotation is not defined.

If the rotation axis (read, “either of the two vector directions that describe the axis”) is perpendicular to the point vector being rotated, then you don’t have to divide the angle by two and you can multiply the rotor by the point. Then you just pluck the resulting values out of the point. As described previously, the point is disguised as a quaternion by dumping the X, Y, and Z components of the vector into the quaternion as coefficients of ***i***, ***j***, and ***k***, respectively, and setting the scalar to zero. After the rotation, the resulting X, Y, and Z components are plucked out of these coefficients.

Optimization: Instead of dividing θ by 2 before calculating the sinθ or cosθ, multiply it by 0.5 once and use the same value in both.

Optimization: When normalizing, divide 1 by the rotation vector magnitude, then multiply each vector component by the inverse magnitude. This saves division costs.

Optimization: When performing the first multiplication (rotor * point quat), the scalar of the point quat is 0, so a couple multiplications can be eliminated. This only eliminates a few float multiplications though and would require jumping through some hoops in a program just to handle this special case, so **I will ignore this optimization**.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Quaternion Rotor Generation	1	1	11	1	2	1	17

Breakdown:

- Sin: 1 from rotor creation
- Cos: 1 from rotor creation
- Multiplication:
    - 3 from normalizing the rotation vector (magnitude squared calculation)
    - 3 from normalizing the rotation vector (multiplying through inverse magnitude)
    - 3 from multiplying normalized rotation vector by $\sin(angle)$
    - 2 from multiplying θ by 0.5 (cheaper than dividing by 2)
- Division:
    - 1 from normalizing the rotation vector (calculating inverse magnitude)
- Add/Sub:
    - 2 from normalizing the rotation vector (magnitude squared calculation)
- Square Root:
    - 1 from normalizing the rotation vector (square root of magnitude squared)
- Temporary Floats:
    - 10 from normalization of rotation vector (1 for each operator)
    - 7 from each other operator

Generating a 3D Rotor: Matrix Approach

I want to compare the computational cost of generating a rotor with matrices, so I will now provide a baseline by analyzing glm::rotate(…), which takes a reference to a starting glm::mat4, a rotation angle in radians, and a vector to rotate around. It will not alter the argument mat4x4, but rather uses it as a starting point. This function is optimized to rotate an existing matrix transform. Rather than try to replicate the matrix generation and explain it, I'll just paste the code. It's an impressive computation:

```
   template <typename T, precision P>
   GLM_FUNC_QUALIFIER detail::tmat4x4<T, P> rotate
      (
      detail::tmat4x4<T, P> const & m,
      T const & angle,
      detail::tvec3<T, P> const & v
      )
   {
#ifdef GLM_FORCE_RADIANS
      T a = angle;
#else
#            pragma message("GLM: rotate function taking degrees as a parameter is
deprecated. #define GLM_FORCE_RADIANS before including GLM headers to remove this
message.")
      T a = radians(angle);
#endif
      T c = cos(a);
      T s = sin(a);

      detail::tvec3<T, P> axis(normalize(v));
      detail::tvec3<T, P> temp((T(1) - c) * axis);

      detail::tmat4x4<T, P> Rotate(detail::tmat4x4<T, P>::_null);
      Rotate[0][0] = c + temp[0] * axis[0];
      Rotate[0][1] = 0 + temp[0] * axis[1] + s * axis[2];
      Rotate[0][2] = 0 + temp[0] * axis[2] - s * axis[1];

      Rotate[1][0] = 0 + temp[1] * axis[0] - s * axis[2];
      Rotate[1][1] = c + temp[1] * axis[1];
      Rotate[1][2] = 0 + temp[1] * axis[2] + s * axis[0];

      Rotate[2][0] = 0 + temp[2] * axis[0] + s * axis[1];
      Rotate[2][1] = 0 + temp[2] * axis[1] - s * axis[0];
      Rotate[2][2] = c + temp[2] * axis[2];

      detail::tmat4x4<T, P> Result(detail::tmat4x4<T, P>::_null);
      Result[0] = m[0] * Rotate[0][0] + m[1] * Rotate[0][1] + m[2] * Rotate[0][2];
      Result[1] = m[0] * Rotate[1][0] + m[1] * Rotate[1][1] + m[2] * Rotate[1][2];
      Result[2] = m[0] * Rotate[2][0] + m[1] * Rotate[2][1] + m[2] * Rotate[2][2];
      Result[3] = m[3];
      return Result;
   }
```

Whoa. Remember that this is column-first notation, so Rotate[index] or Result[index] accesses a column, not a row.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Matrix Rotor Generation	1	1	58	1	42	1	162

Breakdown:

- Sin: 1 from rotor creation
- Cos: 1 from rotor creation
- Multiplication
    - 3 from rotation vector normalization (magnitude squared calculation)
    - 3 from rotation vector normalization (multiplying inverse magnitude through)
    - 1 from `((T(1) - c) * axis`
    - 15 from filling out Rotate mat4
    - 36 from filling out Result mat4 - nine vec4 * float multiplications (remember that argument “m” is a 4x4 matrix, so m[index] is a vec4 column)
- Division
    - 1 from rotation vector normalization (calculating inverse magnitude)
- Add/Sub
    - 2 from rotation vector normalization (magnitude squared calculation)
    - 1 from `T(1) - c`
    - 15 from filling out Rotate mat4
    - 24 from filling out Result mat4 - six “vec4 + vec4” additions (m[index] is a vec4 column)
- Square Root
    - 1 from rotation vector normalization (square root of magnitude squared)
- Temporary Floats
    - 10 from rotation vector normalization operation
    - 1 for each operator (104 total)
    - 16 from Rotate mat4 constructor
    - 16 from Result mat4 constructor
    - 16 from returning Result with a mat4 copy constructor, not optimized

Generating a 3D Rotor: Comparing Matrices and Quaternions

I’ll just paste the results of the last two sections here for easy comparison:

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Quaternion Rotor Generation	1	1	11	1	2	1	17
Matrix Rotor Generation	1	1	58	1	42	1	162

Hot dang! Rotation matrices are expensive to make in comparison! It's a different story though with using a quaternion to rotate a point.

Rotating a Point: Quaternion Approach

A quaternion can be used to rotate a vec3 point around an arbitrary axis. I will first show to it can done ~~wrong~~ with a simple approach that only works in a handful of cases, and then I will demonstrate how to make a quaternion rotor for an arbitrary case.

Curiosity: Rotating a Point around an Orthogonal Vector (Works, but Only in Corner Cases)

If I want to rotate a point vector around a vector that is orthogonal to the point vector, then I can use a simplified rotor, as follows:

$$quat\ rotor = (\cos(\theta), sin(\theta)\hat{v})$$

$$rotated\ point\ p' = rotor * (0, \vec{p})$$

I won't give a cost breakdown because this isn't a general case, but I will *give* an example to demonstrate that it works. This was just a curiosity to me because I was using overly simplistic rotation examples when I was working through this stuff by hand, and I wanted to know why I shouldn't take this approach. If you're not interested, go ahead and skip this and the next section.

1-3-2014 Create a quaternion rotor to rotate a point vector around an orthogonal rotation vector

Rotate $\langle 1, 0, 0 \rangle$ around $\langle 0, 1, 0 \rangle$ by 45°
(x y z) (x y z)

↑ already normalized, so no need to normalize ourselves

similar to ~~Like~~ a complex rotor in 2D, rotor = $\cos\theta \cdot w + \sin\theta \cdot \vec{v}$

$\cos 45° = \frac{\sqrt{2}}{2}$

$\sin 45° = \frac{\sqrt{2}}{2}$

for a rotor, w always ~~= 1~~ (quat always starts at 1); $\vec{v}$ = rotation vector

rotor = $\frac{\sqrt{2}}{2} + 0i + \frac{\sqrt{2}}{2}j + 0k$

$= (\frac{\sqrt{2}}{2}, \langle 0, \frac{\sqrt{2}}{2}, 0 \rangle)$

disguise the point vector as a quaternion:

point = $\langle 1, 0, 0 \rangle$

point quat = $(0, \langle 1, 0, 0 \rangle)$ w = 0 for a point quat

compare, notice the difference; it is very simple (yay!)

rotated point = rotor · point quat

$= (\frac{\sqrt{2}}{2}, \langle 0, \frac{\sqrt{2}}{2}, 0 \rangle)(0, \langle 1, 0, 0 \rangle)$

recall $q_1 \cdot q_2 = ((q_1 w)(q_2 w) - dot(q_1 \vec{v}, q_2 \vec{v}), q_1(q_2 \vec{v}) + q_2(q_1 \vec{v}) + cross(q_1 \vec{v}, q_2 \vec{v}))$

$= ((\frac{\sqrt{2}}{2})(0) - dot(\langle 0, \frac{\sqrt{2}}{2}, 0 \rangle, \langle 1, 0, 0 \rangle), \frac{\sqrt{2}}{2}\langle 1, 0, 0 \rangle + 0(\langle 0, \frac{\sqrt{2}}{2}, 0 \rangle + cross(\langle 0, \frac{\sqrt{2}}{2}, 0 \rangle, \langle 1, 0, 0 \rangle))$

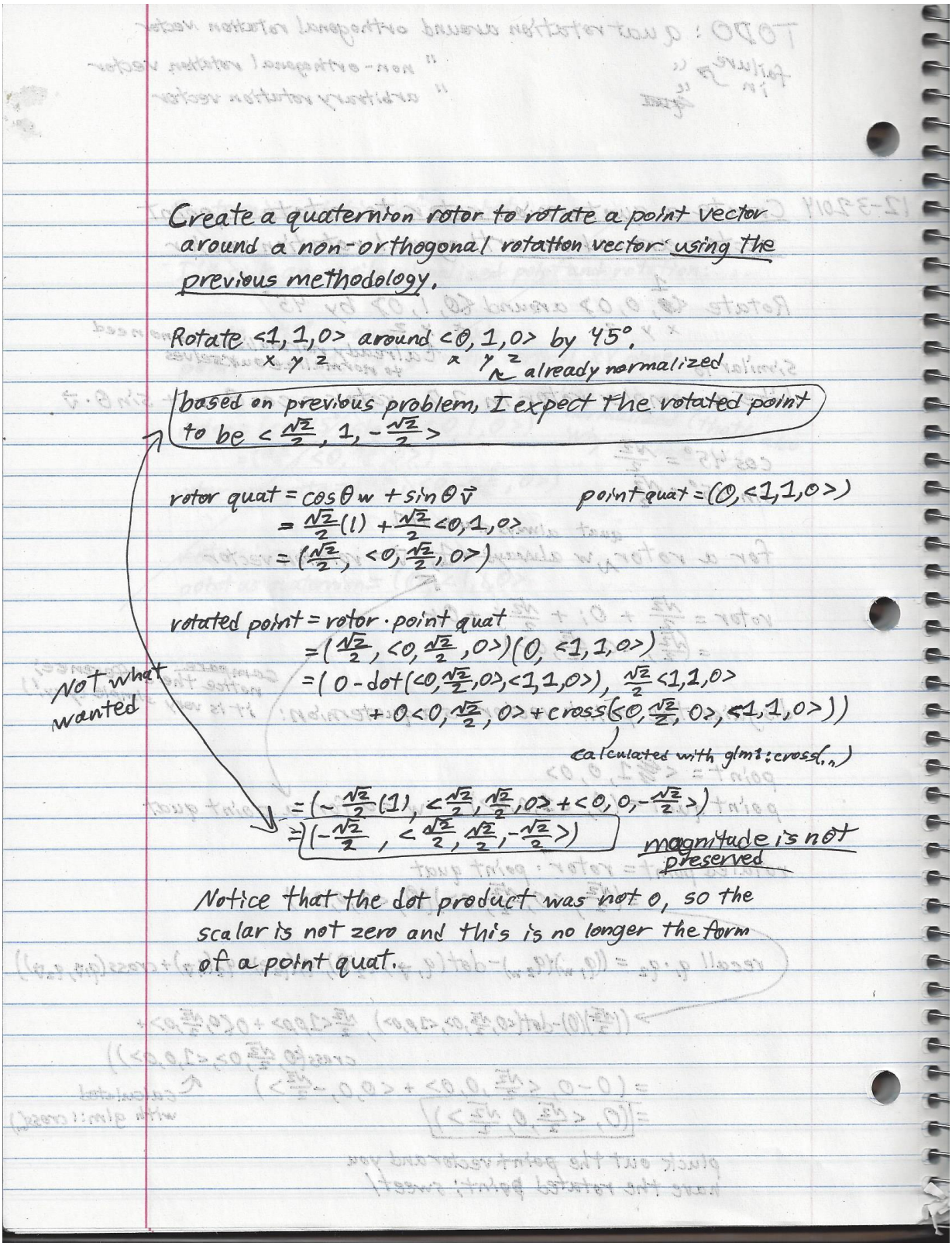
$= (0 - 0, \langle \frac{\sqrt{2}}{2}, 0, 0 \rangle + \langle 0, 0, -\frac{\sqrt{2}}{2} \rangle)$ ← calculated with glm::cross(...)

$= \boxed{(0, \langle \frac{\sqrt{2}}{2}, 0, \frac{\sqrt{2}}{2} \rangle)}$

pluck out the ~~point~~ vector and you have the rotated point; sweet!

Rotating a Point around a Non-Orthogonal Vector with the Same Approach (Fails)

Will this same approach work in the general case (hint: “no”)?



Create a quaternion rotor to rotate a point vector around a non-orthogonal rotation vector using the previous methodology.

Rotate $\langle 1,1,0\rangle$ around $\langle 0,1,0\rangle$ by 45°.
$x\ y\ z$ $x\ y\ z$ already normalized

based on previous problem, I expect the rotated point to be $\langle \frac{\sqrt{2}}{2}, 1, -\frac{\sqrt{2}}{2}\rangle$

rotor quat $= \cos\theta\, w + \sin\theta\, \vec{v}$ point quat $= (0, \langle 1,1,0\rangle)$

$= \frac{\sqrt{2}}{2}(1) + \frac{\sqrt{2}}{2}\langle 0,1,0\rangle$

$= (\frac{\sqrt{2}}{2}, \langle 0, \frac{\sqrt{2}}{2}, 0\rangle)$

rotated point = rotor · point quat

$= (\frac{\sqrt{2}}{2}, \langle 0, \frac{\sqrt{2}}{2}, 0\rangle)(0, \langle 1,1,0\rangle)$

$= (0 - \text{dot}(\langle 0, \frac{\sqrt{2}}{2}, 0\rangle, \langle 1,1,0\rangle), \frac{\sqrt{2}}{2}\langle 1,1,0\rangle + 0\langle 0, \frac{\sqrt{2}}{2}, 0\rangle + \text{cross}(\langle 0, \frac{\sqrt{2}}{2}, 0\rangle, \langle 1,1,0\rangle))$

calculated with glm::cross(,,)

Not what wanted

$= (-\frac{\sqrt{2}}{2}(1), \langle \frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0\rangle + \langle 0, 0, -\frac{\sqrt{2}}{2}\rangle)$

$= (-\frac{\sqrt{2}}{2}, \langle \frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}\rangle)$

magnitude is not preserved

Notice that the dot product was not 0, so the scalar is not zero and this is no longer the form of a point quat.

Rotating a Point around an Arbitrary Vector (General Case)

I already gave the general case a few sections back for generating a quaternion rotor, but I'll recap it here.

$$quat\ rotor = \left(\cos\left(\frac{\theta}{2}\right), sin\left(\frac{\theta}{2}\right)\hat{v}\right)$$

$$rotated\ point\ p' = rotor * (0, \vec{p}) * rotor_{conjugate}$$

For the computational cost analysis, I will assume that the rotor was already created.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Quaternion Point Rotation	0	0	37	0	26	0	69

Breakdown:

- Multiplication:
  - 2 from multiplying angle by 0.5 (cheaper than dividing it by 2)
  - 3 from rotor conjugate construction (negating the vector term of the rotor)
  - 16 from first quaternion multiplication
  - 16 from second quaternion multiplication
- Add/Sub:
  - 13 from first quaternion multiplication
  - 13 from second quaternion multiplication
- Temporary Floats:
  - 3 from rotor conjugate: negating the vector term
  - 4 from rotor conjugate quaternion constructor (it's temporary)
  - 4 from point quaternion construction (it's temporary)
  - 29 from first quaternion multiplication
  - 29 from second quaternion multiplication
  - Return-value optimization for returning the rotated vec3 point, so it's constructor cost is discounted

Rotating a Point: Matrix Approach

This is almost like comparing apples and oranges, but I'll try to set a stage for comparison.

I assumed that, in the quaternion case, there is a function for point rotation that takes references to a quaternion rotor and a vec3 point to be rotated, and it returns a rotated point as a vec3. The glm::rotate(…) function generates a glm::mat4, so I will assume that there is a similar rotation function for matrices that takes references to a glm::mat4 and a vec3 point to be rotated, and it also returns a rotated point as a vec3.

Then the matrix will be used to rotate the vector as follows:

$$vec3_{rotated} = vec3(mat4_{rotate} * vec4(vec3_{point}))$$

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Matrix Point Rotation	0	0	16	0	12	0	32

Breakdown:

- Multiplication:
  - 16 (4 multiplications per matrix row * 4 rows)
- Add/Sub:
  - 12 (3 multiplications per matrix row * 4 rows)
- Temporary Floats:
  - 1 for each operator (28 total)
  - 4 for casting the vec3 point to a vec4 prior to multiplication
  - Return-value optimization in use, so the out-vec3 constructor's float cost is discounted

Rotating a Point: Comparing Matrices and Quaternions

I'll copy the computational costs here so that they can be easily compared:

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Quaternion Point Rotation	0	0	37	0	26	0	69
Matrix Point Rotation	0	0	16	0	12	0	32

Well hot dang again! Quaternions are twice as expensive computationally to rotate a point. This comparison tells me that while quaternion rotors are significantly less expensive than matrices to generate, they are significantly *more* expensive at rotating points. This conclusion now begs the question: is it possible to combine the two approaches so as to get the most out both worlds?

Yes it is. For my next trick, I will cast a quaternion rotor to a 4x4 matrix.

Quaternion to a 4x4 Matrix

I do not use quaternions for my orientations (I use dual quaternions), so I have no need to cast a quaternion to a 4x4 matrix. Someone might want to know how to cast the quaternion to a matrix themselves though, so I'll show how GLM does it.

Technically, a rotation in 3D space could be described by 3x3 matrix, but translations are also used in 3D space, and those are part of a 4x4 matrix, so I'll cast the quaternion to a 4x4. The glm::mat4 structure is very useful here. The GLM matrix is column-first, so mat[0] accesses the first column, *not* the first row, and therefore the translation column (the last column) is accessed by mat[3].

The following is an excerpt from glm::mat4_cast(const glm::fquat &), which takes a reference to a quaternion and puts it into a 4x4 matrix. Technically, mat4_cast(…) calls mat3_cast, which creates a mat3x3, fills it out, and returns a copy into mat4_cast's return value. This can be optimized for your application by defining your own mat4x4, filling that out, and returning a copy. The computational cost difference is only due to the extra floats from the extra constructor call.

Filling out the Matrix

Whatever your approach to generating the 4x4 matrix, it is filled out as follows:

$$q = w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$$

$$mat = \begin{bmatrix} 1-2(y^2+z^2) & 2(xy-wz) & 2(xz+wy) & 0 \\ 2(xy+wz) & 1-2(x^2+z^2) & 2(yz-wx) & 0 \\ 2(xz-wy) & 2(yz+wx) & 1-2(x^2+y^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note: I apologize for it not being perfectly aligned. Microsoft Word only natively supports a 2x2 matrix, so I put 2x2 matrices into each matrix component, but the spacing isn't nice and pretty like I want.

Optimization: Each multiplication (x^2, wz, xy, etc.) in the matrix is performed twice, so you can make a variable for each multiplication, which will cut your x, y, and z multiplications from 18 down to 9. GLM does this in glm::mat4_cast(…).

Optimization: Each of the 9 calculated values can be stored in a single float and then passed into a glm::mat4 constructor in the return value. This gets rid of one of the constructors, but GLM doesn't do this.

Cost of Quaternion to Matrix

The following is the cost of GLM's mat4_cats(…), which calls mat3_ cats(…) to do its job.

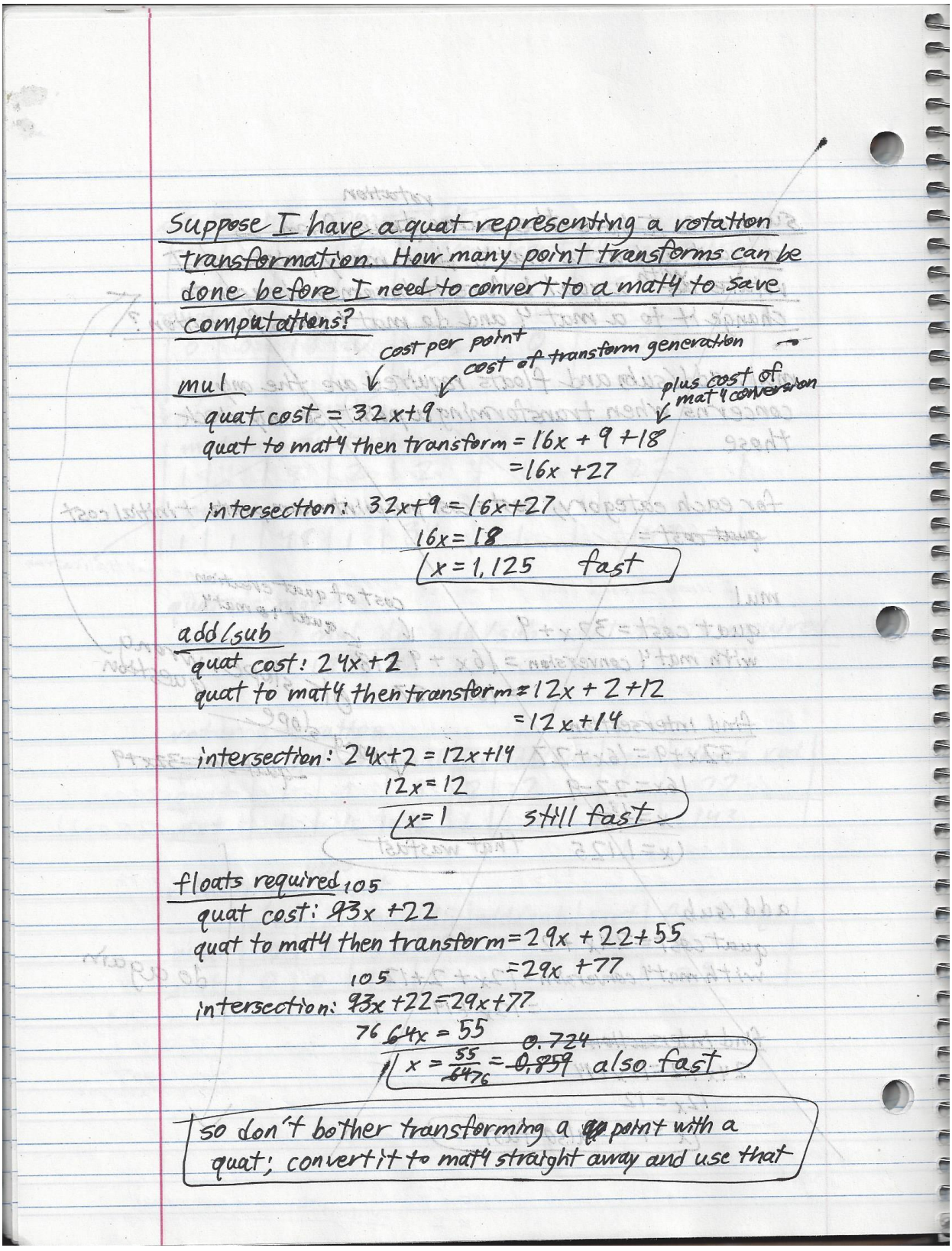
Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
glm::mat4_cast(…)	0	0	18	0	12	0	49

Breakdown:

- Multiplication:
  - 9 from the temporary ww, wx, wy etc. that each occur twice
  - 9 from the multiplications by 2
- Add/Sub
  - 9 from addition of the ww, wx, wy terms to each other
  - 3 from subtractions from 1
- Temporary Floats:
  - 1 for each operator (31 total)
  - 9 for mat3 constructor (a temporary mat3 is declared in the function mat3_cast(…))
  - 9 for mat3 copy constructor (the mat3 is copy-returned to mat4_cast(…))

- Return-value optimization in mat4_cast(…), so the mat4 constructor cost is discounted

How many rotations would you need to do with quaternions before converting the quaternion to a mat4x4 becomes more beneficial?



Suppose I have a quat representing a rotation transformation. How many point transforms can be done before I need to convert to a mat4 to save computations?

cost per point ↓ cost of transform generation → plus cost of mat4 conversion ↓

mul

quat cost = $32x+9$

quat to mat4 then transform = $16x + 9 + 18$

$= 16x + 27$

intersection: $32x+9 = 16x+27$

$16x = 18$

$x = 1,125$ fast

add/sub

quat cost: $24x+2$

quat to mat4 then transform = $12x + 2 + 12$

$= 12x + 14$

intersection: $24x+2 = 12x+14$

$12x = 12$

$x = 1$ still fast

floats required 105

quat cost: $93x + 22$

quat to mat4 then transform = $29x + 22 + 55$

$= 29x + 77$

intersection: $93x + 22 = 29x + 77$

$64x = 55$

$x = \frac{55}{64} = 0.859$ also fast

so don't bother transforming a point with a quat; convert it to mat4 straight away and use that

Rotating a Rotor: Quaternion Approach

Suppose I have a quaternion rotor on some frame N of an animation that rotates a vector point $\vec{v}_{point}$ around a rotation vector $\vec{v}_{rotation1}$ by θ_1 degrees. Then on the next frame (frame N + 1)I want to rotate it relative to where it is by θ_2 around a rotation vector $\vec{v}_{rotation2}$. This is accomplished by generating a rotor for frame N and multiplying the existing rotor by the new one. Repeat for frame N + 2, and N + 3, and so on. This is similar to how the complex rotor for 30° was rotated 60° and the resultant rotor did a 90° rotation.

This is represented as follows:

$$q_{rotor\ net} = q_{rotor\ existing} * q_{rotor\ new}$$

The computational cost is identical to quaternion multiplication. I also assume that the two rotors exist and do not need to be created.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Rotating a Quaternion Rotor	0	0	16	0	13	0	29

I will now show an example.

Rotating a rotor: quat-style

Do something easy and simple: Rotate a 30° rotor by 60°. Compare the result to a built-from-scratch 90° rotor. Rotate only around $\langle 0,1,0\rangle$ to keep things consistent.

$q_{30°} = (\cos(\frac{30°}{2}), \sin(\frac{30°}{2})\langle 0,1,0\rangle)$
$= (0.966, \langle 0, 0.259, 0\rangle)$

$q_{60°} = (\cos(\frac{60°}{2}), \sin(\frac{60°}{2})\langle 0,1,0\rangle)$
$= (0.866, \langle 0, 0.5, 0\rangle$

more specifically ~~$\frac{\sqrt{3}}{2}$~~, but keep it in decimal for consistency

$(q_{30°})(q_{60°}) = ((0.966)(0.866) - dot(\langle 0,0.259,0\rangle, \langle 0,0.5,0\rangle),$
$0.966\langle 0, 0.5, 0\rangle + 0.866\langle 0, 0.259, 0\rangle + cross(v_1, v_2)$
$= (0.837 - 0.130, \ll$
$\langle 0, 0.483, 0\rangle + \langle 0, 0.224\rangle + \langle 0, 0, 0\rangle$

shorthand

↑ cross product of parallel vectors

$= (0.707, \langle 0, 0.707, 0\rangle)$

compare with $q_{90°}$

$q_{90°} = (\cos(\frac{90°}{2}), \sin(\frac{90°}{2})\langle 0,1,0\rangle)$
$= (\frac{\sqrt{2}}{2}, \langle 0, \frac{\sqrt{2}}{2}, 0\rangle)$
$\approx (0.707, \langle 0, 0.707, 0\rangle)$

match!

Rotating a Rotor: Matrix Approach

Matrices use the same approach:

$$rotor_{net} = rotor_{existing} * rotor\_new$$

The cost is identical to matrix multiplication. As usual, I am using 4x4 matrices. I again assume that the two rotors exist and do not need to be created.

Operation Cost	sin	cos	mul	div	add/sub	Sqrt	Temporary floats required
Rotating a Matrix Rotor	0	0	64	0	48	0	112

Breakdown:

- Multiplication:
  - 64 (4 multiplications per cell * 16 cells)
- Add/Sub:
  - 48 (3 additions per cell * 16 cells)
- Temporary Floats:
  - 1 for each operator (112 total)
  - Return-value optimization in use, so the 4x4 matrix constructor cost is discounted

Rotating a Rotor: Comparing Matrices and Quaternions

I will copy the computational cost tables here for easy comparison.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Rotating a Quaternion Rotor	0	0	16	0	13	0	29
Rotating a Matrix Rotor	0	0	64	0	48	0	112

Again, quaternions are the computational champions.

Conclusion

Quaternions are great for rotating transforms without risking the shortest path problem (i.e., “gimbal lock”), and as a bonus they are much less expensive to rotate than matrices. They are, however, *much more* expensive when rotating a vec3 point vector, so keep track of rotations with quaternions, then convert them to matrices to transform a point.

Dual Quaternions

If the quaternion is the conceptual equivalent of a four-axis gimbal gyroscope, then the dual quaternion is the conceptual equivalent of a four-axis gimbal gyroscope plus three-axis accelerometer, which allows it to completely track orientation and location changes in 3D space. This ability to encode both rotation and translation around any axis gave it the nickname "screw transform" (??is this nickname unique to the dual quaternion or does it also apply to the rotation + translation matrix??), but I'll just call it a dual quaternion.

I'll give you a heads up: it is less computationally expensive to generate and transform them compared to 4x4 matrices, but it is much more expensive to use them to transform a single point. Their best use is in keeping track of transformations, then casting them to 4x4 matrices to transform your point vectors.

Dual Quaternion: Dual Number with Quaternions or Quaternion with Dual Numbers?

I learned a lot about dual quaternion construction from a dual quaternion article on Simon's Tech blog[12]. If you start with a quaternion and insert dual numbers instead of real numbers, you will get the following:

$$quaternion\ q = w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$$

$$??= (a + b\varepsilon) + (c + d\varepsilon)\boldsymbol{i} + (e + f\varepsilon)\boldsymbol{j} + (g + h\varepsilon)\boldsymbol{k}$$

Multiply the coefficients through to get the follow:

$$??= a + b\varepsilon + c\boldsymbol{i} + d\varepsilon\boldsymbol{i} + e\boldsymbol{j} + f\varepsilon\boldsymbol{j} + g\boldsymbol{k} + h\varepsilon\boldsymbol{k}$$

Rearrange this equation to group together terms with and without the dual operator as follows:

$$??= (a + c\boldsymbol{i} + e\boldsymbol{j} + g\boldsymbol{k}) + (b + d\boldsymbol{i} + f\boldsymbol{j} + h\boldsymbol{k})\varepsilon$$

The values are now arranged to form two quaternions. One has only real values, and the other has only real numbers on the dual number line. We therefore call this construct a "dual quaternion" and abbreviate the two quaternions as follows:

$$dq = q_{real} + q_{dual}\varepsilon$$

Note: You would get the same result if you started with a dual number and jammed quaternions (quad numbers) into the place of the real numbers.

Is this even legal? Can you just stick anything claiming to be a number into any other place that takes a number? Yes you can! This is one of the beauties of math. The result may be entirely useless if the operators are not defined, such as if you tried to represent an angle with an irrational number and tried to calculate the cosine of that angle (cosine is defined for use with real numbers and nothing else), but it is certainly legal. Fortunately for the curious newcomer, the dual number arithmetic operations arise from simple extensions of dual number and quaternion arithmetic, which in turn arose from real number arithmetic.

I don't have anything else to say conceptually about dual quaternions, so I'll dig right into the math.

---

[12] http://simonstechblog.blogspot.com/2011/11/dual-quaternion.html

Dual Quaternion Addition/Subtraction

Add two dual quaternions $dq_1 = q_{1r} + q_{1d}\boldsymbol{\varepsilon}$ and $d1_2 = q_{2r} + q_{2d}\boldsymbol{\varepsilon}$ (or subtraction; it's the same process and the same computational cost) as follows:

$$dual\ quat\ result = dq_1 + d1_2$$

$$= (q_{1r} + q_{2r}) + (q_{1d} + q_{2d})\boldsymbol{\varepsilon}$$

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Dual Quat Add/Sub	0	0	0	0	8	0	16

Breakdown:

- Add/sub
  - 8 from 2 quaternion additions at 4 additions per
- Temporary floats required
  - 1 for each operation (8 total)
  - 8 from 2 temporary quaternions (from quat additions)
  - Return-value optimization in use, so the float cost of the dual quaternion constructor is discounted

Dual Quaternion Multiplication

This one is a bit of a beast. It looks deceptively simple, but it can blow up really big when you try to do it by hand. Transforming a point vector or another dual quaternion transform requires dual quaternion multiplication, and I will show by-hand examples later.

Multiply two dual quaternions $dq_1 = q_{1r} + q_{1d}\boldsymbol{\varepsilon}$ and $d1_2 = q_{2r} + q_{2d}\boldsymbol{\varepsilon}$ as follows:

$$dual\ quat\ result = dq_1 * d1_2$$

$$= (q_{1r})(q_{2r}) + (q_{1r})(q_{2d}\boldsymbol{\varepsilon}) + (q_{1d}\boldsymbol{\varepsilon})(q_{2r}) + (q_{1d}\boldsymbol{\varepsilon})(q_{2d}\boldsymbol{\varepsilon})$$

The last term results in an $\boldsymbol{\varepsilon}^2$ term, so that cancels out and I am left with the following:

$$= (q_{1r})(q_{2r}) + ((q_{1r})(q_{2d}) + (q_{1d})(q_{2r}))\boldsymbol{\varepsilon}$$

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Dual Quat Multiplication	0	0	48	0	43	0	91

Breakdown:

- Mutliplication:
  - 48 from 3 quaternion multiplications (16 per)

- Add/Sub:
  - 39 from 3 quaternion multiplications (13 per)
  - 4 from quaternion addition in the dual part
- Temporary Floats:
  - 1 for each operator (
  - I use embedded return-value optimization when multiplying and adding the quaternions in the dual part, so their constructor costs are discounted
  - Return-value optimization in use, so the float cost of the dual quaternion is discounted

Dual Quaternion Division

No idea. ??remove this section altogether? I;ve never used dual quaternion division??

Dual Quaternion Conjugate

This was a difficult subject for me. I read in Ben Kenwright's paper, "A Beginners Guide to Dual Quaternions", that the conjugate of a dual quaternion $dq = q_r + q_d\boldsymbol{\varepsilon}$ was $dq^* = q_r^* + q_d^*\boldsymbol{\varepsilon}$. This seemed odd because it didn't jive with the pattern that I understood of the other definitions of conjugation.

I knew that the conjugate of a complex number $cn = a + b\boldsymbol{i}$ is $cn^* = a - b\boldsymbol{i}$.

I knew that the conjugate of a quaternion $q = w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$ is $q^* = w - x\boldsymbol{i} - y\boldsymbol{j} - z\boldsymbol{k}$.

I knew that the conjugate of a dual number $dn = a + b\boldsymbol{\varepsilon}$ is $dn^* = a - b\boldsymbol{\varepsilon}$.

So it looked like I should just negate all non-real parts. But dual quaternions have two sets of non-real unit numbers: the complex unit vectors $\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}$ and the dual operator $\boldsymbol{\varepsilon}$, so I had to think about it and experiment as follows:

There are three ways (in my opinion) to think of a dual quaternion:

1. $dq = q_r + q_d\boldsymbol{\varepsilon}$
2. $dq = (w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}) + (w_0 + x_0\boldsymbol{i} + y_0\boldsymbol{j} + z_0\boldsymbol{k})\boldsymbol{\varepsilon}$
3. $dq = (w + w_0\boldsymbol{\varepsilon}) + (x + x_0\boldsymbol{\varepsilon})\boldsymbol{i} + (y + y_0\boldsymbol{\varepsilon})\boldsymbol{j} + (z + z_0\boldsymbol{\varepsilon})\boldsymbol{k}$

Therefore I can also think of three possible conjugations:

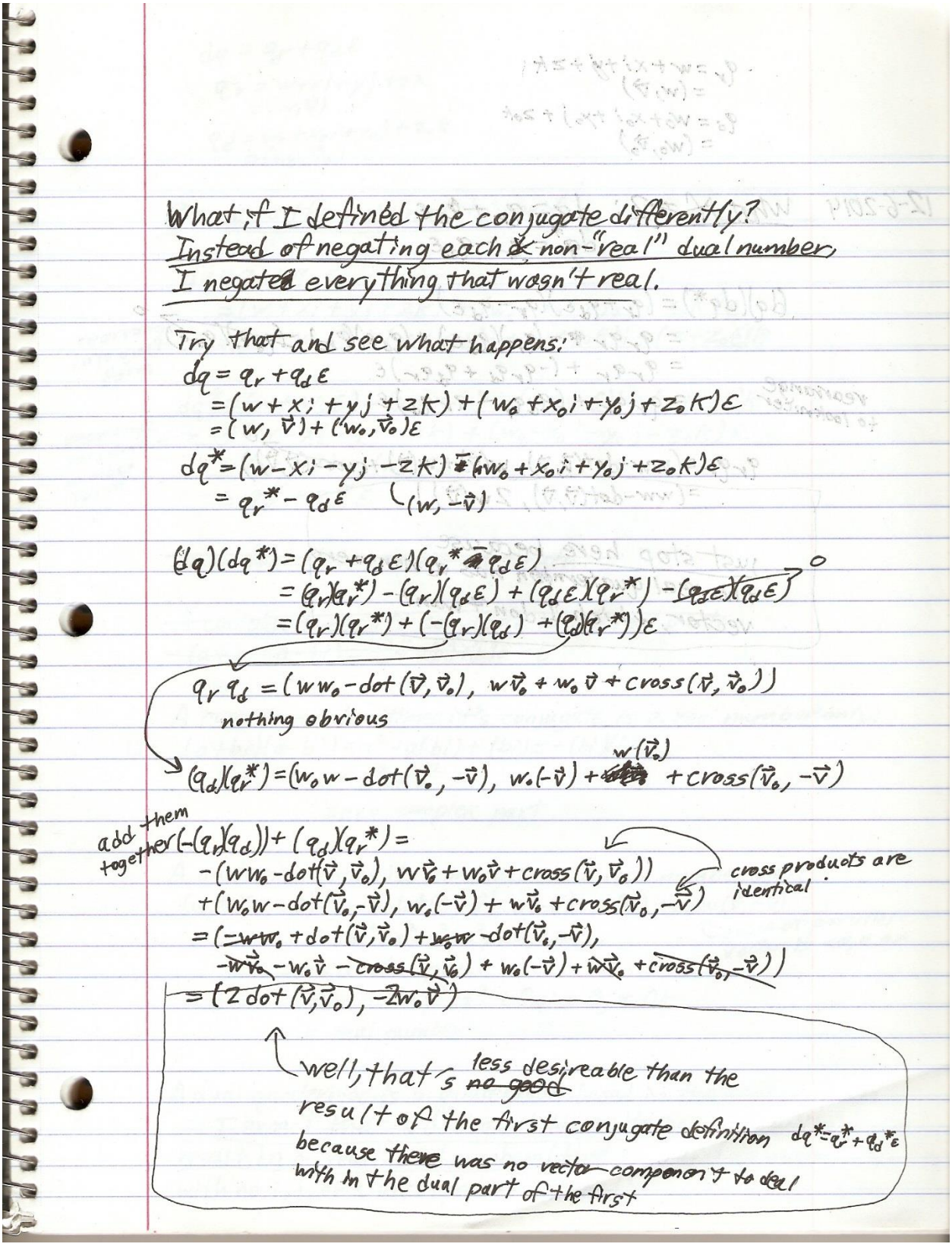
1. Negate all non-real components: $dq = (w - x\boldsymbol{i} - y\boldsymbol{j} - z\boldsymbol{k}) + (-w_0 - x_0\boldsymbol{i} - y_0\boldsymbol{j} - z_0\boldsymbol{k})\boldsymbol{\varepsilon}$
2. Negate the non-real quaternion: $dq = q_r - q_d\boldsymbol{\varepsilon}$
3. Negate the non-real dual numbers: $dq = (w + w_0\boldsymbol{\varepsilon}) - (x + x_0\boldsymbol{\varepsilon})\boldsymbol{i} - (y + y_0\boldsymbol{\varepsilon})\boldsymbol{j} - (z + z_0\boldsymbol{\varepsilon})\boldsymbol{k}$

Which is right? Could multiple options be correct?

The only way that I found to judge between them is to see if a dual quaternion times its conjugate is a "real" dual number. That is, it is a dual number with only real values (no complex operators ***i***, ***j***, or ***k***).

??I think that the definition of a conjugate depends on the kind of number you are trying to get out of it. Some conversation I had on math stackexchange??

Conjugate Approach 1 (doesn't work)



Conjugate Approach 2 (doesn't work)

$q_r = w + xi + yj + zk$
$= (w, \vec{v})$
$q_d = w_0 + x_0 i + y_0 j + z_0 k$
$= (w_0, \vec{v}_0)$

12-6-2014 What if #2: $dq = q_r + q_d\varepsilon$
$dq^* = q_r - q_d\varepsilon$

$(dq)(dq^*) = (q_r + q_d\varepsilon)(q_r - q_d\varepsilon)$
$= q_r q_r - (q_r)(q_d\varepsilon) + (q_d\varepsilon)(q_r) - (q_d\varepsilon)(q_d\varepsilon)$ → 0
$= q_r q_r + (-q_r q_d + q_d q_r)\varepsilon$
rearrange to look nicer
$= q_r q_r + (q_d q_r - q_r q_d)\varepsilon$

$q_r q_r = (ww - dot(\vec{v},\vec{v}),\ w(\vec{v}) + w(\vec{v}) + cross(\vec{v},\vec{v}))$ → 0
$= (ww - dot(\vec{v},\vec{v}),\ 2w(\vec{v}))$

just stop here because the real quaternion has a non-zero vector, which I don't want

Conjugate Approach 3 (works!)

$dq = q_r + q_d\varepsilon$

$q_r = w + xi + yj + zk$
$= (w, \vec{v})$

$q_d = w_0 + x_0 i + y_0 j + z_0 k$
$= (w_0, \vec{v}_0)$

Back to the original to verify

$dq = q_r + q_d\varepsilon$
$= (w + xi + yj + zk) + (w_0 + x_0 i + y_0 j + z_0 k)\varepsilon$

rearrange into quat form → $= (w + w_0\varepsilon) + (x + x_0\varepsilon)i + (y + y_0\varepsilon)j + (z + z_0\varepsilon)k$

$dq^* = (w + w_0\varepsilon) - (x + x_0\varepsilon)i - (y + y_0\varepsilon)j - (z + z_0\varepsilon)k$

rearrange into dual number form → $= (w - xi - yj - zk) + (w_0 - x_0 i - y_0 j - z_0 k)\varepsilon$
$= (w, -\vec{v}) + (w_0, -\vec{v}_0)\varepsilon$
$= q_r^* + q_d^*\varepsilon$

~~Now for the moment of truth:~~

~~A complex times its conjugate is the real part:~~
~~$(a+bi)(a-bi) = a^2 - abi + bia - b$~~

A complex number times it's conjugate is a real number only:
$(a+bi)(a-bi) = a^2 - a(bi) + (bi)a - (bi)(bi)$
$= a^2 + b^2$
zero complex part

A quaternion times its conjugate is a real number only:
$(w, \vec{v})(w, -\vec{v}) = (w^2 - \text{dot}(\vec{v}, -\vec{v}),\ w(-\vec{v}) + w(\vec{v}) + \text{cross}(\vec{v}, -\vec{v}))$
$= (w^2 + x^2 + y^2 + z^2, \langle 0,0,0\rangle)$
↑ cross of parallel vectors is $\langle 0,0,0\rangle$
zero vector part
$= w^2 + x^2 + y^2 + z^2 + 0i + 0j + 0k$
= real number

A dual quaternion is a quaternion based in the dual number, so I expect that a dual quat times its conjugate will result in a "real" dual number (that is, a dual number with no i, j, or k unit numbers attached).

$$(dq)(dq^*) = (q_r + q_d\varepsilon)(q_r^* + q_d^*\varepsilon)$$
$$= q_r(q_r^*) + q_r(q_d^*\varepsilon) + (q_d\varepsilon)(q_r^*) + \cancel{(q_d\varepsilon)(q_d\varepsilon)} \to 0$$
$$= q_r(q_r^*) + (q_r(q_d^*) + q_d(q_r^*))\varepsilon$$

$$q_r(q_r^*) = w^2 + x^2 + y^2 + z^2 + 0i + 0j + 0k$$

$$q_r(q_d^*) = (w w_0 - dot(\vec{v}, -\vec{v}_0),\ w(\vec{v}_0) + w_0(\vec{v}) + cross(\vec{v}, -\vec{v}_0)$$
$$q_d(q_r^*) = (w_0 w - dot(\vec{v}_0, -\vec{v}),\ w_0(-\vec{v}) + w(\vec{v}_0) + cross(\vec{v}_0, -\vec{v})$$

$$cross(\vec{v}, -\vec{v}_0) \equiv cross(\vec{v}_0, \vec{v})$$
$$cross(\vec{v}_0, -\vec{v}) \equiv cross(\vec{v}, \vec{v}_0)$$

opposites, so adding their results together will be $\langle 0,0,0\rangle$

extract negative from dot products

$$q_r(q_d^*) + q_d(q_r^*) = (w w_0 + dot(\vec{v}, \vec{v}_0) + w_0 w + dot(\vec{v}_0, \vec{v}),$$
$$w(-\vec{v}_0) + w_0(\vec{v}) + cross(\vec{v}, -\vec{v}_0) + w_0(-\vec{v}) + w(\vec{v}_0) + cross(\vec{v}_0, -\vec{v}))$$
$$= (2ww_0 + 2dot(\vec{v}, \vec{v}_0),\ \langle 0,0,0\rangle)$$

conclusion:

$$(dq)(dq^*) = (w^2 + x^2 + y^2 + z^2, \langle 0,0,0\rangle) + (2ww_0 + 2dot(\vec{v}, \vec{v}_0), \langle 0,0,0\rangle)\varepsilon$$

This is a "real" dual number ~~with~~ since it has no complex parts (i, j, or k). How nice.

This is also the square of the magnitude of the dual number. Take the square root of this dual number to get the magnitude.

And that is why the dual quaternion conjugate is the way it is. It is the only one that generates a pure dual number when multiplied by it's conjugate.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Dual Quat Conjugate	0	0	6	0	0	0	6

Breakdown:

- Multiplication:
  - 3 from negating the vector component of the real part
  - 3 from negating the vector component of the dual part
- Temporary Floats:
  - 1 from each operator (6 total)
  - I use embedded return-value optimization, so the constructor costs of negating the vectors all the way to generating the dual quaternion itself are discounted

Dual Quaternion Magnitude

Magnitude Squared

The magnitude of a dual quaternion, since it is a multiple-component number, must satisfy the following equation:

$$aa^* = |a|^2$$

Where a is a multiple-component number and a^* is its conjugate

The Wikipedia entry on dual quaternion magnitude[13] states the following for a normalized dual quaternion $\hat{A}$ (the hat notating a unit dual quaternion):

$$\hat{A}\hat{A}^* = (A, B)(A^*, B^*)$$

$$= (AA^*, AB^* + BA^*)$$

$$= (1,0)$$

Where A and B are real quaternions, B has an implied dual operator applied to it, and A^* and B^* are their conjugates

While true, this is not particularly helpful. It is saying that a normalized dual quaternion times its conjugate is 1. I already knew that. What about a non-normalized dual quaternion? I'll follow the equation for calculating the square of the magnitude of a multiple-component number.

The magnitude of a dual quaternion dq works out as follows:

$$magnitude^2 = (dq)(dq^*)$$

[13] http://en.wikipedia.org/wiki/Dual_quaternion#Norm

$$= (q_{real} + q_{dual}\boldsymbol{\varepsilon})(q^*_{real} + q^*_{dual}\boldsymbol{\varepsilon})$$

$$= [(w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}) + (w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k})\boldsymbol{\varepsilon}][(w - x\boldsymbol{i} - y\boldsymbol{j} - z\boldsymbol{k}) + (w - x\boldsymbol{i} - y\boldsymbol{j} - z\boldsymbol{k})\boldsymbol{\varepsilon}]$$

I am definitely not going to write all this out here, but I will show you the result. It looks like a mess, and it is if you are doing it by hand, and it simplifies to the following:

$$magnitude^2 = ((q_{real}.w)^2 + dot(q_{real}.\vec{v}, q_{real}.\vec{v}), < 0,0,0 >)$$

$$+(2(q_{real}.w)(q_{dual}.w) + dot(q_{real}.\vec{v}, q_{dual}.\vec{v}), < 0,0,0 >)\boldsymbol{\varepsilon}$$

That's a handful. But it *is* a dual number because the vector components of both quaternions are 0. This makes sense to me because a dual quaternion is based on a dual number, but I'll have to jump through some hoops to normalize it.

Optimization: Rather than taking the easy way out by just multiplying a dual quaternion by its conjugate, compute the equation above.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Dual Quat Magnitude Squared (naive)	0	0	54	0	43	0	97
Magnitude Squared (optimized)	0	0	9	0	6	0	15

Breakdown (naïve):

- Multiplication:
  - 6 from generating the dual quaternion conjugate
  - 48 from dual quaternion multiplication
- Add/Sub:
  - 43 from dual quaternion multiplication
- Temporary Floats:
  - 6 from generating the dual quaternion conjugate
  - 91 from dual quaternion multiplication

Breakdown (optimized):

- Multiplication:
  - 3 from regular scalar multiplications
  - 6 from dot products
- Add/Sub:
  - 2 from regular scalar addition
  - 4 from dot products
- Temporary Floats:
  - 1 for each operator (15 total)

Dual Number Square Root Re-Visit

Make use of the dual number square root equation here. Recall from much earlier in the paper that I solved for the square root of a dual number dn as follows:

$$dn = a + b\boldsymbol{\varepsilon}$$

$$\sqrt{dn} = \sqrt{a + b\boldsymbol{\varepsilon}}$$

Set the square root of $a + b\boldsymbol{\varepsilon}$ equal to another (unknown) dual number.

$$\sqrt{a + b\boldsymbol{\varepsilon}} = c + d\boldsymbol{\varepsilon}$$

$$a + b\boldsymbol{\varepsilon} = (c + d\boldsymbol{\varepsilon})^2$$

$$a + b\boldsymbol{\varepsilon} = c^2 + (cd + dc)\boldsymbol{\varepsilon}$$

$$a + b\boldsymbol{\varepsilon} = c^2 + (2cd)\boldsymbol{\varepsilon}$$

The final, simplified equation only requires real numbers, which makes it quite cheap.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Dual Number Square Root	0	0	3	0	0	0	3

Breakdown:

- Multiplication:
  - 3
- Temporary Floats:
  - 1 for each operator (3 total)

Magnitude (Combining the Two)

The total cost of computing the magnitude of a dual quaternion is therefore the cost of computing the magnitude squared plus the cost of computing the square root.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Dual Number Magnitude	0	0	12	0	6	0	18

Breakdown:

- Multiplication:
  - 9 from optimized magnitude squared
  - 3 from dual number square root
- Add/Sub:

    - 6 from optimized magnitude squared
- Temporary Float:
    - 15 from optimized magnitude squared
    - 3 from dual number square root

That's really cheap and **requires no square root** to boot!

Dividing a Dual Quaternion by a Dual Number

Normalization will require that a dual quaternion be divided by its magnitude, which has already been established as a dual number. Is this process defined?

Recall how I defined dual number division:

Given two dual numbers dn1 and dn2, divide them as follows:

$$\frac{dn1}{dn2} = \frac{a + b\boldsymbol{\varepsilon}}{c + d\boldsymbol{\varepsilon}}$$

$$= \frac{(a + b\boldsymbol{\varepsilon})(c - d\boldsymbol{\varepsilon})}{(c + d\boldsymbol{\varepsilon})(c - d\boldsymbol{\varepsilon})}$$

$$= \frac{ac - a(d\boldsymbol{\varepsilon}) + (b\boldsymbol{\varepsilon})c - (b\boldsymbol{\varepsilon})(d\boldsymbol{\varepsilon})}{c^2 - c(d\boldsymbol{\varepsilon}) + (d\boldsymbol{\varepsilon})c - (d\boldsymbol{\varepsilon})(d\boldsymbol{\varepsilon})}$$

$$= \frac{ac + (bc - ad)\boldsymbol{\varepsilon}}{c^2}$$

$$= \frac{1}{c^2} * (ac + (bc - ad)\boldsymbol{\varepsilon})$$

Now suppose that the numerator is a dual quaternion instead of a dual number. This problem would then change to a dual number being divided by a dual number. Does this equation still hold? Yes it does! It holds because c and d are real numbers (scalars), so even though a and b are quaternions, they are only multiplied by scalars and remain in the numerator. The final equation shows there will multiple quaternion * scalar computations and one quaternion addition in the dual part, so it looks pretty cheap.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Divide Dual Quat by Dual Number	0	0	26	1	4	0	50

Breakdown:

- Multiplication:
    - 4 from ac (quaternion * scalar)
    - 4 from bc (quaternion * scalar)
    - 4 from ad (quaternion * scalar)
    - 1 from c^2

    - 8 from multiplying through $\frac{1}{c^2}$
- Division
    - 1 from $\frac{1}{c^2}$
- Add/Sub:
    - 4 from $bc - ad$ (quaternion subtraction)
- Temporary Floats:
    - 1 from each operator (26 total)
    - 4 from temporary quaternion ac
    - 4 from temporary quaternion bc
    - 4 from temporary quaternion ad
    - 4 from temporary quaternion $bc - ad$
    - 8 for temporary dual quaternion $ac + (bc - ad)\boldsymbol{\varepsilon}$

Dual Quaternion Normalization

This is not as easy as I have read it to be. I will address the error in GLM (as of 0.9.5.3), and then I will show you how to calculate it correctly (read, "satisfies the requirement that a multiple-component number times its conjugate = the real number 1 only without any imaginary or dual components").

GLM's fdualquat Normalization Error

Error: When GLM (as of 0.9.5.2) normalizes a dual quaternion (glm::normalize(const glm::fdualquat &)), it seems to assume (this is my own thought) that the square of the dual number magnitude, which resulted in a real number with no dual part, also held for dual quaternions. That is, since the square of the magnitude of the dual number $a + b\boldsymbol{\varepsilon}$ was a^2 and therefore the magnitude was simply the absolute value of a (if a was negative, then its square would be positive, and the square root of that would simply be the absolute value), then the square of the magnitude of a dual quaternion $q_{real} + q_{dual}\boldsymbol{\varepsilon}$ was equivalent to $q_{real}.magnitude\_squared()$, and therefore the magnitude would simply $q_{real}.magnitude()$. This assumption is in error. As shown in the section on dual quaternion magnitude, the number*conjugate requirement results in a non-zero dual component.

The magnitude of a normalized multiple-component number is simply the real number 1, and therefore the square of the magnitude will also be 1. It is therefore easy to check if your dual quaternion is normalized: multiply it by its conjugate and check if the resulting real quaternion's scalar is 1 and the 7 other values are 0. **GLM's approach fails this check.**

12-12-2014 GLM's normalization of dual quats

Suppose I have a dual quat $dq = q_r + q_d\varepsilon$, where
$q_r = (1 + 2i + 3j + 4k)$ and $q_d = (5 + 6i + 7j + 8k)\varepsilon$.
$= (1, \langle 2,3,4\rangle)$ $= (5, \langle 6,7,8\rangle)\varepsilon$

in dual_quaternion.inl, lines 271-278:

```
dq normalize(dq)
{
  return dq / length(dq.real);
}
```

The length(...) function is the magnitude. This value will be as follows

$$\text{magnitude}(q_r) = \sqrt{1^2+2^2+3^2+4^2} = \sqrt{30} \approx 5.477$$

yes, this implemented with 8 divisions

$$\frac{dq}{5.477} = \left(\frac{1}{5.477} + \frac{2i}{5.477} + \frac{3j}{5.477} + \frac{4k}{5.477}\right) + \left(\frac{5}{5.477} + \frac{6i}{5.477} + \frac{7j}{5.477} + \frac{8k}{5.477}\right)\varepsilon$$
$$\approx (0.183, \langle 0.365, 0.548, 0.730\rangle) + (0.913, \langle 1.085, 1.278, 1.461\rangle)\varepsilon$$

This should be normalized now, so I'll call it dq_n

If dq_n really is normalized, then $(dq_n)(dq_n^*)$ should $= 1$.
That is the real portion's vector and the whole dual part should be 0 and the real portion's scalar exactly 1.

I'll try it.

for now, let q_r, q_r^*, q_d, and q_d^* refer to the normalized version of dq, which is dq_n

$$dq_n^* = (0.183, \langle -0.365, -0.548, -0.730\rangle) + (0.913, \langle -1.095, -1.278, -1.461\rangle)\varepsilon$$

$$(dq_n)(dq_n^*) = q_r q_r^* + q_r(q_d^*\varepsilon) + (q_d\varepsilon)q_r^* + (q_d\varepsilon)(q_d^*\varepsilon)^{\to 0}$$
$$= q_r q_r^* + (q_r q_d^* + q_d q_r^*)\varepsilon$$

$$q_r q_r^* = (0.183(0.183) - \text{dot}(\vec{v}, -\vec{v}),\ 0.183(-\vec{v}) + 0.183(\vec{v}) + \text{cross}(\vec{v}, -\vec{v})$$
$$= (0.0333 + 0.133 + 0.300 + 0.533, \langle 0,0,0\rangle)$$
$$= (1.000, \langle 0,0,0\rangle)$$

cross of parallel vectors is 0

↑ looks promising

$$q_r q_d^* = (0.183(0.913) - \text{dot}(\vec{v}, -\vec{v}_0),\ 0.183(-\vec{v}_0) + 0.913(\vec{v}) + \text{cross}(\vec{v}, -\vec{v}_0)$$
$$= (0.183(0.913) + 0.365(-1.095) + 0.548(-1.278) + 0.730(-1.461),\ \text{vector stuff})$$
$$= (0.167 + 0.400 + 0.700 + 1.067,\ \text{vector stuff})$$
$$= (2.334,\ \text{vector stuff})$$

I'm not solving for the vector stuff yet because I want to see if the $\vec{v}$ and $\vec{v}_0$ terms in the next multiplication cancel this one's.

$$q_d q_r^* = (0.913(0.183) - \text{dot}(\vec{v}_0, -\vec{v}),\ 0.913(-\vec{v}) + 0.183(\vec{v}_0) + \text{cross}(\vec{v}_0, -\vec{v})$$
$$= (0.913(0.183) + 1.095(-0.365) + 1.278(-0.548) + 1.461(-0.730),\ \text{vector stuff})$$
$$= (0.167 + 0.400 + 0.700 + 1.067,\ \text{vector stuff})$$
$$= (2.334,\ \text{vector stuff})$$

addition of dual parts

$$q_r q_d^* + q_d q_r^* = (2.334 + 2.334,$$
$$0.183(-\vec{v}_0) + 0.913(\vec{v}) + \text{cross}(\vec{v}, -\vec{v}_0) \quad (= -\text{cross}(\vec{v}_0, \vec{v}))$$
$$+ 0.913(-\vec{v}) + 0.183(\vec{v}_0) + \text{cross}(\vec{v}_0, -\vec{v}) \quad (= \text{cross}(\vec{v}, \vec{v}_0))$$
$$= (4.668, \langle 0,0,0\rangle)$$

opposites

finally so $(dq_n)(dq_n^*) = (1.000, \langle 0,0,0\rangle) + (4.668, \langle 0,0,0\rangle)\varepsilon$

NOT $= 1$, so this normalization approach failed

My Own Confused (and in error) Reasoning (??remove this??)

I also thought that it might be possible to take the dual number that was the dual quaternion's square magnitude and calculate the magnitude of that (how meta) to get a real number, take the square root of that, then divide all eight of the dual quaternion's components by it (I'm still a bit confused trying to understand my own reasoning here). I would do this for a dual quaternion dq as follows:

$$dual\ number\ magnitude^2\ dn = (dq)(dq^*)$$

$$real\ number\ magnitude^2\ r\_squared = (dn)(dn^*)$$

$$magnitude\ of\ dq = \sqrt{r\_squared}$$

$$normalized\ dq = \frac{dq}{r}$$

But this didn't work. I'm not entirely sure that what I was doing was mathematically legal, but I tried to do this to avoid dividing a dual quaternion by its dual number magnitude. My efforts were in vain.

Correct Normalization (Dual Quat Times Conjugate = 1)

Here's how you do it right. Suppose you have a non-normal dual quaternion dq. Calculate its magnitude, then divide the original dual quaternion dq by its magnitude. These operations have already been described, so I will simply sum their costs.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Dual Quat Normalization	0	0	28	1	10	0	68

Breakdown:

- Multiplication:
  - 12 from dual number magnitude
  - 26 from dividing a dual quaternion by a dual number
- Division
  - 1 from dividing a dual quaternion by a dual number
- Add/Sub:
  - 6 from dual number magnitude
  - 4 from dividing a dual quaternion by a dual number
- Temporary Floats:
  - 18 from dual number magnitude
  - 50 from dividing a dual quaternion by a dual number

Here's an example (be warned; it's a bit lengthy because I did it by hand and stretched out the calculations to show my work, but it simplifies to a dual quaternion division by a dual number and a few quaternion-scalar multiplications):

Normalizing a dual quat

Suppose you have a dual quat $dq = q_r + q_d\varepsilon$ and its dual number magnitude $dn = a + b\varepsilon$. Then the normalized (or "unit") $\hat{dq} = \frac{dq}{dn}$.

~~If, if~~

$$\hat{dq} = \frac{dq}{dn} = \frac{q_r + q_d\varepsilon}{a + b\varepsilon}$$, where q_r and q_d are quaternions and a and b are real numbers

Can we divide a dual quat by a dual number? The numbers don't line up conceptually?! One has quaternions and the other reals!

Solution: Dual number division is ~~computed~~ started by multiplying the top and bottom by the bottom's conjugate.

$$\frac{q_r + q_d\varepsilon}{a + b\varepsilon} = \frac{(q_r + q_d\varepsilon)(a - b\varepsilon)}{(a + b\varepsilon)(a - b\varepsilon)} = \frac{q_r a - q_r(b\varepsilon) + (q_d\varepsilon)a - (q_d\varepsilon)(b\varepsilon)}{a^2 - a(b\varepsilon) + (b\varepsilon)a - (b\varepsilon)(b\varepsilon)} \to 0$$

$$= \frac{q_r a + (-q_r b + q_d a)\varepsilon}{a^2}$$

$$= \frac{1}{a^2}(q_r a + (q_d a - q_r b)\varepsilon)$$

↑ This is a dual quaternion.

It is possible to optimize this a bit by multiplying the real q_r by $\frac{1}{a}$ and the dual q_d by $\frac{1}{a}$, but that requires an extra variable and only saves you two floats. I won't bother doing this.

I'll try normalizing again, properly this time.

$$dq = q_r + q_d\varepsilon$$
$$= q(1+2i+3j+4k) + (5+6i+7j+8k)\varepsilon$$
$$= (1, \langle 2,3,4\rangle) + (5, \langle 6,7,8\rangle)\varepsilon$$
$$dq^* = q_r^* + q_d^*$$
$$= (1, \langle -2,-3,-4\rangle) + (5, \langle -6,-7,-8\rangle)\varepsilon$$

$$\text{magnitude}^2 = (dq)(dq^*)$$
$$= q_r q_r^* + q_r(q_d^*\varepsilon) + (q_d\varepsilon)q_r^* + (q_d\varepsilon)(q_d^*\varepsilon)^0$$
$$= q_r q_r^* + (q_r q_d^* + q_d q_r^*)\varepsilon$$

I know that $q_r q_r^*$ will result in a zero vector component, and I know that $q_r q_d^*$ has the same scalar but an opposite vector to $q_d q_r^*$, so $q_r q_d^* + q_d q_r^*$ results in 2 scalar and a zero vector. I can therefore simplify the calculation considerably by only calculating the scalars.

$$q_r q_r^* = (1(1) - \text{dot}(\vec{v}, -\vec{v}), \langle 0,0,0\rangle)$$
$$= (1^2 + 2^2 + 3^2 + 4^2, \langle 0,0,0\rangle)$$
$$= (30, \langle 0,0,0\rangle)$$

$$q_r q_d^* = (1(5) - \text{dot}(\vec{v}, -\vec{v}_0), \text{whatever vector})$$
$$q_d q_r^* = (\underbrace{5(1) - \text{dot}(\vec{v}_0, -\vec{v})}_{\text{pick one and double it (they're identical)}}, \underbrace{-\text{whatever vector}}_{\text{sum to } \langle 0,0,0\rangle})$$

$$q_r q_d^* + q_d q_r^* = (\overbrace{5 + 12 + 21 + 32}^{2x}, \langle 0,0,0\rangle)$$
$$= (70, \langle 0,0,0\rangle)$$
$$140$$
oops

resume computation

$$\frac{dq}{\text{magnitude}} = \frac{q_r + q_d\varepsilon}{c + d\varepsilon} = \frac{1}{c^2}(q_r c + (q_d c - q_r d)\varepsilon)$$

$q_r c = (1, \langle 2,3,4\rangle)\ 5.477$
$= (5.477, \langle 10.954, 16.431, 21.908\rangle)$

$q_d c = (5, \langle 6,7,8\rangle)\ 5.477$
$= (27.385, \langle 32.862, 38.339, 43.816\rangle)$

$q_r d = (1, \langle 2,3,4\rangle)\ 12.78$
$= (12.78, \langle 25.56, 38.34, 51.12\rangle)$

$q_d c - q_r d = (27.385 - 12.78,$
$\langle 32.862 - 25.56,$
$38.339 - 38.34,$
$43.816 - 51.12\rangle)$
$= (14.605, \langle 7.302, -0.001, -7.304\rangle)$

now apply the $\frac{1}{c^2}$ multiplication

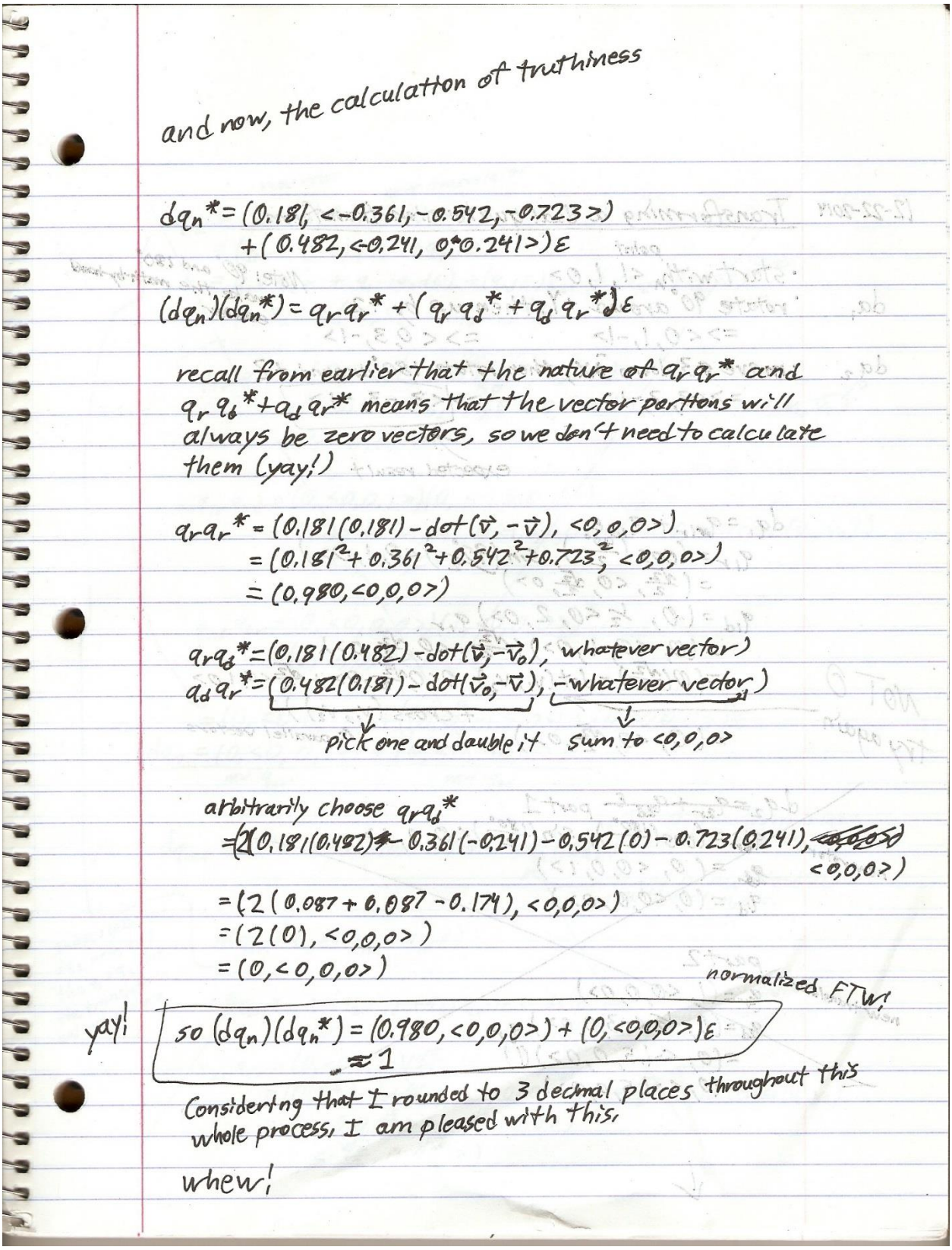
$\frac{1}{c^2} = \frac{1}{(\sqrt{30})^2} = \frac{1}{30} = 0.033$

$\frac{1}{c^2}(q_r c) = 0.033\,(5.477, \langle 10.954, 16.431, 21.908\rangle)$
$= (0.181, \langle 0.361, 0.542, 0.723\rangle)$

$\frac{1}{c^2}(q_d c - q_r d) = 0.033(14.605, \langle 7.302, -0.001, -7.304\rangle)$
$= (0.482, \langle 0.241, 0.000, -0.241\rangle)$

so normalized $dq = dq_n = (0.181, \langle 0.361, 0.542, 0.723\rangle)$
$+ (0.482, \langle 0.241, 0.000, -0.241\rangle)\,\varepsilon$
0

↓ but does $(dq_n)(dq_n^*) = 1$?



And that's how you do it!

Justification of Analysis Method

GLM's rotator and translator generation functions provide a convenient way to construct a new matrix or to rotate/translate an existing one, all with the same function. Dual quaternions have no such convenience. The only way to transform a dual quaternion is to multiply it by another. So how is it appropriate to compare GLM's double-duty functions with my dual quaternions, which are optimized to only generate the transforms? Put simply, I have no other basis for comparison (??is this true??) that I am aware of.

If you want to generate a new rotation or translation matrix, you have to suffer the cost of the whole glm::rotate(…) or glm::translate(…). This is why I am comparing them.

The full transform comparisons, in which both a translation and rotation are encapsulated in the same transform, be they dual quaternion or matrix, are more even comparisons of purpose.

Matrix Translation Cost Analysis

I already covered the generation cost of a matrix rotor with glm::rotate(…). I am about to get into one of the dual quaternion strengths: creating a combination translation and rotation relatively (compared to matrices) inexpensively. To do this, I will quickly analyze the cost of glm::translate(…) (the one that takes a matrix reference as an argument so that any existing matrix can be passed in), and then I will recap the cost of generating a matrix rotor through glm::rotate(…). Then I will have a basis for comparison.

Here's GLM's code:

```
template <typename T, precision P>
GLM_FUNC_QUALIFIER detail::tmat4x4<T, P> translate
(
   detail::tmat4x4<T, P> const & m,
   detail::tvec3<T, P> const & v
)
{
   detail::tmat4x4<T, P> Result(m);
   Result[3] = m[0] * v[0] + m[1] * v[1] + m[2] * v[2] + m[3];
   return Result;
}
```

It looks like it creates a temporary 4x4 matrix, multiplies some of its vec4 columns by scalars, and then copy-returns the result.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Matrix Translation Generation	0	0	12	0	12	0	56

Breakdown:

- Multiplication:
    - 12 from three column (a vec4) * scalar multiplications

- Add/Sub:
  - 12 from three column + column additions
- Temporary Floats:
  - 1 from each operator (24 total)
  - 16 from Result (a temporary 4x4 matrix)
  - 16 from the copy-return constructor of Result

Pure Rotor – Dual Quaternion Approach

Construct a purely rotational dual quaternion as follows:

$$dual\ quat\ transform\ dq_t = q_r + q_d\boldsymbol{\varepsilon}$$

$$q_r = (\cos\left(\frac{\theta}{2}\right), \sin\left(\frac{\theta}{2}\right) * normalize(vector_{rotation}))$$

$$q_d = (0, < 0,0,0 >)$$

It's quite simple. Generate a rotor the usual way, then make the dual part entirely 0.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Dual Quat Pure Rotor	1	1	8	1	2	1	18

Breakdown:

- Sin: 1
- Cos: 1
- Multiplication:
  - 3 for rotation vector normalization (computing magnitude squared)
  - 3 for rotation vector normalization (multiplying through by inverse magnitude)
  - 2 for multiplying θ by 0.5 (cheaper than dividing each element by 2)
- Division:
  - 1 for rotation vector normalization (computing inverse magnitude)
- Add/Sub:
  - 2 for rotation vector normalization (computing magnitude squared)
- Square Root:
  - 1 for rotation vector normalization (compute magnitude from magnitude squared)
- Temporary Floats:
  - 1 for each operator (14 total)
  - 4 for temporary quaternion real part
  - Return-value optimization in use, so constructor cost of the quaternion dual part (assumed hard coded) and the full dual quaternion is discounted:

Pure Rotor – Comparing Matrices and Dual Quaternions

I'll copy the computation cost tables here for easy comparison:

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Dual Quat Pure Rotor	1	1	8	1	2	1	18
Matrix Rotor Generation	1	1	58	1	42	1	162

A purely rotational dual quaternion is far cheaper to produce than a matrix rotor.

Pure Translator – Dual Quaternion Approach

Construct a purely rotational dual quaternion as follows:

$$dual\ quat\ transform\ dq_t = q_r + q_d\boldsymbol{\varepsilon}$$

$$q_r = (1, < 0,0,0 >)$$

$$q_d = \left(0, \frac{1}{2} vector_{translate}\right)(q_r)$$

$$= \left(0, \frac{1}{2} vector_{translate}\right)(1)$$

$$= \left(0, \frac{1}{2} vector_{translate}\right)$$

Notice that the dual part requires a quaternion multiplication. This is normal, but when generating a pure translator, the real part is simply 1, so I ignore this multiplication, and as a result, generating the dual portion becomes easier.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Dual Quat Pure Translator	0	0	3	0	0	0	3

Breakdown:

- Multiplication:
  - 3 from multiplying translate vector by 0.5 (cheaper than dividing all three elements by 2)
- Temporary Floats:
  - 1 for each operator (3 total)
  - Most of this can be hard-coded, so I will assume that there are no temporary quaternions holding the real and dual parts
  - Return-value optimization in use, so the float cost of the dual quaternion will be discounted

Pure Translator – Comparing Matrices and Dual Quaternions

I will copy the computation cost tables here for easy comparison.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Dual Quat Pure Translator	0	0	3	0	0	0	3
Matrix Translation Generation	0	0	12	0	12	0	56

There isn't a significant difference from an absolute perspective (12 multiplications vs 3, 12 additions vs 0), but the dual quaternion wins out.

Full Transform - Rotate Then Translate – Dual Quaternion Approach (a.k.a., "Screw Transform")

Given a rotation vector, a rotation angle, and a translation angle, I can construct a "screw transform", so called because it describes a rotation and a displacement, much like the surface of a screw (I don't know why a translate-then-rotate doesn't have this nickname). It is created as follows:

$$dual\ quat\ transform\ dq_t = q_r + q_d\boldsymbol{\varepsilon}$$

Construct the real portion as follows:

$$q_r = (\cos\left(\frac{\theta}{2}\right), \sin\left(\frac{\theta}{2}\right) * normalize(vector_{rotation}))$$

Just like with the quaternion rotor, it is important that the rotation vector be normalized!

Construct the dual portion as follows:

$$q_d = \left(0, \frac{1}{2} vector_{translate}\right) * q_r$$

Note: Technically, the ½ of the dual portion is multiplied as follows:

$$q_d = \frac{1}{2} * (0, vector_{translate}) * q_r$$

The real portion of the quaternion is 0 though, so I applied the ½ where it is applied most easily, which is to the vector component prior to multiplication.

The construction of this dual part is not trivial, but it is vital that it be constructed this way: starting with a 0 scalar, halving the point vector, multiplying by the previously constructed real quaternion, and multiplying the whole thing by 2.

Note: Due to the way that the dual portion is constructed, a transformed point will have to be extracted by undoing some of this. I tried to figure out a way around multiplying by 2 and the real portion so that I could avoid the extraction expense, but it simply doesn't work out. I'll cover later, when I transform a point, how to extract the point from the dual part.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Dual Quat Rotate Then Translate	1	1	27	1	15	1	54

Breakdown:

- Sin: 1
- Cos: 1
- Multiplication:
  - 3 for rotation vector normalization (computing magnitude squared)
  - 3 for rotation vector normalization (multiplying through by inverse magnitude)
  - 2 for multiplying θ by 0.5 (cheaper than dividing each element by 2)
  - 3 for multiplying translate vector by 0.5 (cheaper than dividing each element by 2)
  - 16 from quaternion multiplication (constructing dual part)
- Division:
  - 1 for rotation vector normalization (computing inverse magnitude)
- Add/Sub:
  - 2 for rotation vector normalization (computing magnitude squared)
  - 13 from quaternion multiplication (constructing dual part)
- Square Root:
  - 1 for rotation vector normalization (compute magnitude from magnitude squared)
- Temporary Floats:
  - 1 for each operator other than the quaternion multiplication (17 total)
  - 29 from quaternion multiplication (constructing dual part)
  - 4 for temporary quaternion real part (my implementation)
  - 4 for temporary quaternion dual part (my implementation)
  - Return-value optimization in use, so constructor cost of dual quaternion is discounted

Full Transform - Rotate Then Translate – Matrix Approach

This approach is fairly simple because GLM has provided convenient libraries. I have already described the cost of glm::rotate(…) and glm::translate(…), both of take a const reference to a glm::mat4 as a starting matrix. I can therefore create a matrix with glm::rotate(…) and then pass the result into glm::translate(…). This means two function calls, and while it would be possible to utilize return-value optimization by embedding the glm::translate(…) call into the matrix reference argument of glm::rotate(…), I will assume that the return value of glm::rotate(…) is assigned to a temporary glm::mat4 variable, as any sane programmer would do. This also means that I will count the number of floats required by this temporary matrix. I *will*, however, assume that glm::translate(…) is called by this fictional generate_rotate_then_translate(…) function's return statement, so return-value optimization is in use and I will *not* count the temporary float cost of this second matrix.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required

Matrix Rotate Then Translate	1	1	70	1	54	1	234

Breakdown:

- Sin: 1 from rotor generation
- Cos: 1 from rotor generation
- Multiplication:
  - 58 from rotor generation
  - 12 from translation generation
- Division:
  - 1 from rotor generation
- Add/Sub:
  - 42 from rotor generation
  - 12 from translation generation
- Square Root:
  - 1 from rotor generation
- Temporary Floats:
  - 162 from rotor generation
  - 16 from temporary matrix (result of rotor generation)
  - 56 from translation generation
  - Return-value optimization in use, so the float cost of the resulting 4x4 matrix is discounted

Full Transform – Rotate Then Translate – Comparing Matrices and Dual Quaternions

I will copy the cost tables for each operation here for easy comparison.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Dual Quat Rotate Then Translate	1	1	27	1	15	1	54
Matrix Rotate Then Translate	1	1	70	1	54	1	234

The dual quaternion wins out again.

Full Transform - Translate Then Rotate – Dual Quaternion Approach

I can also generate a transformation that will translate the point and *then* rotate it. I have imagined this to be useful in modeling an object that became dynamically attached to another object and then gets swung around, such two space vessels fighting, then one fires a space harpoon gun (because every land-originating thing in space must be preceded by “space”) into the other, thus locking them together.

I construct this transformation with a purely rotational dual quaternion and a purely translational dual quaternion as follows:

$$dq_{rotor} = generate\ pure\ rotor$$

$$dq_{translator} = generate\ pure\ translate$$

$$dq_{transform} = dq_{rotor} * dq_{translator}$$

And that's it.

Optimization: This isn't as expensive as you might think because, despite the addition of the dual quaternion multiplication, the pure translator construction is cheap, and I can also take advantage of the form of the pure rotor and pure translator to significantly optimize the multiplication, as follows:

$$dq_{transform} = dq_{rotor} * dq_{translator}$$

$$= (rotor_{real} + rotor_{dual}\boldsymbol{\varepsilon})(translator_{real} + translator_{dual}\boldsymbol{\varepsilon})$$

$$= (rotor_{real})(translator_{real}) + (rotor_{real})(translator_{dual}\boldsymbol{\varepsilon}) + (rotor_{dual}\boldsymbol{\varepsilon})(translator_{real}) + (rotor_{dual}\boldsymbol{\varepsilon})(translator_{dual}\boldsymbol{\varepsilon})$$

$$= (rotor_{real})(1) + \big((rotor_{real})(translator_{dual}) + (0)(translator_{real})\big)\boldsymbol{\varepsilon} + (whatever)\boldsymbol{\varepsilon}^2$$

$$= rotor_{real} + ((rotor_{real})(translator_{dual}))\boldsymbol{\varepsilon}$$

Now I only have to generate the pure translator and the pure rotor, then do a single a quaternion multiplication, not unlike the screw transform (rotate-then-translate). In my implementation, I utilize the quaternion's rotor generation for the real part and I generate a pure quaternion (zero (0) scalar) with half the vector for the dual part, then I jam them into a dual quaternion constructor.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Dual Quat Translate Then Rotate	1	1	30	1	15	1	55

Breakdown:

- Sin: 1 from rotor generation
- Cos: 1 from rotor generation
- Multiplication:
  - 11 from rotor generation
  - 3 from multiplying through translation vector by 0.5 (cheaper than dividing all three components by 2)
  - 16 from quaternion multiplication
- Division:
  - 1 from rotor generation
- Add/Sub:
  - 2 from rotor generation
  - 13 from quaternion multiplication
- Square Root:

    - 1 from rotor generation
- Temporary Floats:
    - 17 from rotor generation
    - 29 from quaternion multiplication
    - 1 for each other operator (3)
    - 4 for temporary real part (my implementation)
    - 4 for temporary dual part (my implementation)
    - Return-value optimization in use, so the float cost of the dual quaternion is discounted

The cost is very close to that of a rotate-then-translate dual quaternion.

Full Transform – Translate Then Rotate – Matrix Approach

To perform the same feat with a matrix, I would have to call glm::translate(…) and pass the resulting matrix into glm::rotate(…). **This will be the same cost as calling glm::rotate(…) first and then glm::translate(…).**

Full Transform – Translate Then Rotate – Comparing Dual Quaternions and Matrices

I'll copy the cost computation tables here for easy comparison.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Dual Quat Translate Then Rotate	1	1	30	1	15	1	55
Matrix Translate Then Translate	1	1	70	1	54	1	234

The dual quaternion is still better.

Point Transformation – Dual Quaternion Approach

Suppose I have a dual quaternion transform dq_t and a point vector p. I want to translate the point, so I begin by putting the point into a purely translator dual quat shell that I will call dq_p.

$$dq_p = (1, < 0,0,0 >) + (0, \frac{1}{2} * < P_x, P_y, P_z >)\boldsymbol{\varepsilon}$$

I then translate the point in one of two ways.

Approach 1 (Slightly More Expensive)

!!verify that it works because I haven't gotten it to yet!!

I transform the point as described by Ben Kenwright in his paper, "A Beginners Guide to Dual-Quaternions":

$$dq_{result} = (dq_t)(dq_p)(dq_t^*)$$

This requires two dual quaternion multiplications, and is therefore quite expensive.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Ben Kenwright point transformation							

insert scan here

Approach 2 (Slightly Less Expensive; The Way I Do It)

I transform the point by right-multiplying the point dual quaternion against the transform as follows:

$$dq_{result} = (dq_t)(dq_p)$$

The result is another dual quaternion, but I need a point vector back. I extract the point from the result by multiplying the dual portion by the conjugate of the real portion, then multiplying the whole thing by 2, as follows:

$$q_{translate} = 2 * (dq_{result}.q_{dual} * dq_{result}.q^*_{real})$$

$$translated\ point\ p_{translated} = q_{translate}.vector$$

Notice the conjugate of the real part! This is an important part of the calculation, and that asterisk is an easy thing to miss in all variables and superscripts and subscripts of the equation.

Note: $q_{translate}$ *will* have a 0 scalar. It will be a pure quaternion, which is equivalent to just the vector portion. This is one way that I can be confident that the math is correct.

Cost of Extraction Alone

Transforming a point (disguised as a dual quaternion) does not result in a vector that I can readily use. The equations show that it must be extracted (personal experimentation has also proved this to me). This extraction isn't free.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Point vec3 extraction	0	0	23	0	13	0	41

Breakdown:

- Multiplication:
  - 3 from generating the conjugate of the real part
  - 16 from quaternion multiplication
  - 4 from multiplying through quaternion by 2
- Add/Sub:
  - 13 from quaternion multiplication
- Temporary Floats:
  - 4 from temporary quaternion (conjugate of real part)

- 29 from quaternion multiplication
- 4 from temporary quaternion (result of quaternion multiplication)
- 4 from temporary quaternion (result of scalar * quaternion)
- Return-value optimization in use, so the float cost of the vec3 is discounted

Total Cost of Transforming a Point

Similar to my quaternion rotor example, I will assume that the transform already exists and that the point already exists as a vec3. The point must now be jammed into a dual quaternion, multiplied with the transform, then extracted.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
My point transformation	0	0	74	0	56	0	155

Breakdown:

- Multiplication:
  - 3 from generating point dual quaternion (requires multiplying through point vec3 by 0.5)
  - 48 from dual quaternion multiplication
  - 23 from extraction
- Add/Sub:
  - 43 from dual quaternion multiplication
  - 13 from extraction
- Temporary Floats:
  - 8 from temporary dual quaternion (jamming the point into a dual quaternion)
  - 91 from dual quaternion multiplication
  - 8 from temporary dual quaternion (result of transform * point)
  - 41 from extraction
  - 1 for each other operator (7 total)
  - Return-value optimization in use, so the float cost of the returned vec3 is ignored

Well that's expensive. But enough of that for now. Here's an example:

expected resulting point: $\langle -\frac{\sqrt{2}}{2}, 3.5, -\frac{\sqrt{2}}{2} \rangle$

$= \langle -0.707, 3.5, -0.707 \rangle$

12-10-2014 **Use a dual quat to rotate $\langle 1,1,0 \rangle$ by 135° around $\langle 0,1,0 \rangle$ and move it up by 2.5.**

generate the transform

$dq_t = q_r + q_d \varepsilon$

already normalized, but would cost 6 mul, 2 add, 1 sqrt, 1 div, 1 vec3

convert to rad for my sake

$q_r = (\cos(\frac{135°}{2}), \sin(\frac{135°}{2}) \langle 0,1,0 \rangle)$

$= (\cos(\frac{3\pi}{8}), \sin(\frac{3\pi}{8}) \langle 0,1,0 \rangle)$

$= (0.383, \langle 0, 0.924, 0 \rangle)$

cost: 1 div, 1 cos, 1 sin, 3 mul, 1 vec3s, 1 quat (includes a vec3)

$q_d = (0, \frac{1}{2} \langle 0, 2.5, 0 \rangle) q_r$

$= (0, \langle 0, 1.25, 0 \rangle)(0.383, \langle 0, 0.924, 0 \rangle)$

$= (0(0.383) - dot(\vec{v}_1, \vec{v}_2), 0\langle 0, 0.924, 0 \rangle + 0.383\langle 0, 1.25, 0 \rangle$

$+ cross(\vec{v}_1, \vec{v}_2)$ → cross of parallel vectors is 0

$= (-1.155, \langle 0, 0.479, 0 \rangle)$

cost: 3 mul + quat multiplication

16 mul, 12 add, 47 floats (5 vec3s, 1 quat, 1 for each operator)

generate the point as a dual quat so that we can perform dual quat multiplication:

$dq_p = q_r + q_d \varepsilon$

$q_r = (1, \langle 0,0,0 \rangle)$

$q_d = (0, \frac{1}{2} \langle 1,1,0 \rangle) q_r$ → ignore because it is equivalent to multiplying by 1

$= (0, \langle 0.5, 0.5, 0 \rangle)$

cost: 3 mul

transform:

$(dq_t)(dq_p) = (dq_{t\cdot}q_r + dq_{t\cdot}q_d\,\varepsilon)(dq_{p\cdot}q_r + dq_{p\cdot}q_d\varepsilon)$

$= (dq_{t\cdot}q_r)(dq_{p\cdot}q_r) + (dq_{t\cdot}q_r)(dq_{p\cdot}q_d\,\varepsilon)$

$+ (dq_{t\cdot}q_d\,\varepsilon)(dq_{p\cdot}q_r) + \cancel{(dq_{t\cdot}q_d\varepsilon)(dq_{p\cdot}q_d\varepsilon)}$ → 0

~~$= (0.383, \langle 0, 0.924, 0\rangle)(1, \langle 0,0,0\rangle)$~~

~~$+ (-1.155, \langle 0, 0.479, 0\rangle)(0, \langle 0.5, 0.5, 0\rangle)$~~ sorry; typo

simplify

$= (0.383, \langle 0, 0.924, 0\rangle)(1, \langle 0,0,0\rangle)$

$+ (0.383, \langle 0, 0.924, 0\rangle)(0, \langle 0.5, 0.5, 0\rangle)\,\varepsilon$

$+ (-1.155, \langle 0, 0.479, 0\rangle)(1, \langle 0,0,0\rangle)\,\varepsilon$

suggested optimization: the q_r of the point dual quat will always be 1, so don't bother with the first and third ~~quat~~ quat multiplications

compute

$= (0.383, \langle 0, 0.924, 0\rangle)$

$+ (0.383, \langle 0, 0.924, 0\rangle)(0, \langle 0.5, 0.5, 0\rangle)\,\varepsilon$

$+ (-1.155, \langle 0, 0.479, 0\rangle)\,\varepsilon$

point dual quat. dual.w is always 0, so possible for further optimization

$= (0.383(0) - dot(\vec{v}_1, \vec{v}_2),\ 0.383\langle 0.5, 0.5, 0\rangle$

$+ 0\langle 0, 0.924, 0\rangle + cross(\vec{v}_1, \vec{v}_2)$

$= (0 - 0.462, \langle 0.192, 0.192, 0\rangle + \langle 0,0,0\rangle + \langle 0, 0, -0.462\rangle)$

$= (-0.462, \langle 0.192, 0.192, -0.462\rangle)$

~~$(0.383 \quad 0.924$~~

$= (0.383, \langle 0, 0.924, 0\rangle)$

$+ (-0.462 - 1.155, \langle 0.192, 0.192, -0.462\rangle + \langle 0, 0.479, 0\rangle)\,\varepsilon$

result $= (0.383, \langle 0, 0.924, 0\rangle) + (-1.617, \langle 0.192, 0.671, -0.462\rangle)\,\varepsilon$

Whoo... looks like a mess, but there is one more step

extract the translated point vector from the dual portion of the multiplication result

$point\ quat = 2(result.q_d)(result.q_r^*)$

$result.q_r^* = (0.383, \langle 0, -0.924, 0 \rangle)$

$(result.q_d)(result.q_r^*) = (-1.617, \langle 0.192, 0.671, -0.462 \rangle)(0.383, \langle 0, -0.924, 0 \rangle)$

$= (\underbrace{(-1.617)(0.383)}_{-0.619} - dot(\vec{v}_1, \vec{v}_2), \underbrace{(-1.617)\langle 0, -0.924, 0 \rangle}_{\langle 0, 1.494, 0 \rangle})$

$+ 0.383 \langle 0.192, 0.671, -0.462 \rangle + cross(\vec{v}_1, \vec{v}_2)$

$= (-0.619 - (-0.620), \langle 0, 1.494, 0 \rangle + \langle \underbrace{0.071}, 0.257, -0.177 \rangle$

$+ \langle -0.427, 0, -0.177 \rangle$

$= (0.001, \langle -0.356, 1.751, -0.354 \rangle)$

$2 \cdot (all\ this) = (0.002, \langle 0.712, 3.502, -0.708 \rangle)$

transformed point

recall that the expected point was

$\langle -\frac{\sqrt{2}}{2}, 3.5, -\frac{\sqrt{2}}{2} \rangle \approx \langle -0.707, 3.5, -0.707 \rangle$

considering that I rounded to 3 decimal places and not more, I'm content with this by-hand result.

Point Transformation – Comparing Matrices with Dual Quaternions

I already analyzed the computation cost of transforming a vector with a matrix way back in the section on quaternions. I'll copy both computation cost tables here for easy comparison.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
My point transformation	0	0	74	0	56	0	155
Matrix Point Rotation	0	0	16	0	12	0	32

Well that's a less rosy picture for the dual quaternion. The basic lesson here is that a point *can* be transformed with a dual quaternion, but it's a really poor choice of CPU cycles to do it this way. A much better way would be to convert to it to a 4x4 matrix, which brings me to the next section.

Dual Quaternion: To Mat4x4

I have nothing to explain conceptually here, so I'll just throw the math at you. I got the formulas from Ben Kenwright's paper, "A Beginners Guide to Dual-Quaternions". Here is the class method that I use to cast my dual quaternions to a glm::mat4. Recall that the glm::mat4 is ordered column-first, so mat[0] gets the first column in the matrix.

```
glm::mat4 F_Dual_Quat::to_mat4() const
{
   // get an identity matrix
   glm::mat4 mat;

   // extract rotational information
   float scalar = this->m_real.m_scalar;
   float x = this->m_real.m_vector.x;
   float y = this->m_real.m_vector.y;
   float z = this->m_real.m_vector.z;

   mat[0][0] = (scalar * scalar) + (x * x) - (y * y) - (z * z);
   mat[0][1] = (2 * x * y) + (2 * scalar * z);
   mat[0][2] = (2 * x * z) - (2 * scalar * y);

   mat[1][0] = (2 * x * y) - (2 * scalar * z);
   mat[1][1] = (scalar * scalar) + (y * y) - (x * x) - (z * z);
   mat[1][2] = (2 * y * z) + (2 * scalar * x);

   mat[2][0] = (2 * x * z) + (2 * scalar * y);
   mat[2][1] = (2 * y * z) - (2 * scalar * x);
   mat[2][2] = (scalar * scalar) + (z * z) - (x * x) - (y * y);

   // extract translational information
   Math::F_Quat trans = (this->m_dual * 2.0f) * this->m_real.conjugate();
   mat[3][0] = trans.m_vector.x;
   mat[3][1] = trans.m_vector.y;
   mat[3][2] = trans.m_vector.z;
```

```
        // the last row remains untouched; that row is the realm of clip space and
perspective division

        return mat;

    }
```

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Dual Quat Cast to Mat4	0	0	59	0	28	0	128

Breakdown:

- Multiplication:
  - 36 from computing the first three rows and columns (4 per cell * 9 cells)
  - 23 from extracting translation component
- Add/Sub:
  - 13 from extracting translation component
  - 15 from computing the first three rows and columns (5 per column * 3 columns)
- Temporary Floats:
  - 16 from the temporary matrix
  - 4 for the temporary floats (scalar, x, y, z)
  - 1 for each other operator (51)
  - 41 from extracting translation component
  - 16 from copy-returning the matrix (no return-value optimization in use)

Ouch. Point transformation performance was terrible, and now this. Dual quaternions are looking less advantageous. But wait, there's more!

Transforming a Transform – Dual Quaternion Approach

Just like complex rotors, quaternion rotors, and matrix transforms, a dual quaternion transform is transformed by multiplying them together, as follows:

$$new\ transform = new\ transform * old\ transform$$

And that's it. If I assume that both transforms exist, then **the cost of transforming a dual quaternion transform is simply the cost of multiplying two dual quaternions**.

Transforming a Transform – Matrix Approach

There are actually two ways I can do this because glm::rotate(…) and glm::translate(…) take a const reference to a glm::mat4 and then rotate it or translate it, respectively. I assume that the old transform exists, and that I want to do a combination of rotation and translation.:

(1) Create the new transform by calling glm::rotate(…) and glm::translate(…) in whatever order you need, passing the result of the first as an argument to the second. Then multiply the new matrix transform by the old one to get the net transform. This is how I learned to do it in tutorials, so I anticipate that this is how a lot of people still do it.

(2) Take advantage of the ability to feed a matrix into the glm::rotate(…) or glm::translate(…). Pass the old transform into these two functions in whichever order you need, passing the result of the first as an argument to the second. **This is the same computation cost as generating a brand new rotate-then-translate or translate-then-rotate transform**.

I'll analyze both.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Transforming Matrix Transform (naïve)	1	1	134	1	102	1	362
Transforming Matrix Transform (optimized)	1	1	70	1	54	1	234

Breakdown (naïve):

- Sin: 1 from generating new transform
- Cos: 1 from generating new transform
- Multiplication:
  - 70 from generating new transform
  - 64 from 4x4 matrix multiplication
- Division:
  - 1 from generating new transform
- Add/Sub:
  - 54 from generating new transform
  - 48 from 4x4 matrix multiplication
- Square Root:
  - 1 from generating new transform
- Temporary Floats:
  - 234 from generating new transform
  - 16 for temporary 4x4 matrix (the new transform is temporary)
  - 112 from 4x4 matrix multiplication

Oh dear goodness. That's 1448 bytes for a naïve matrix transformation, and 936 bytes for an optimized one. That's expensive for a single operation.

Transforming a Transform – Comparing Matrices and Dual Quaternions

I'll copy the computation cost tables here for easy comparison. I'll be thorough and show both approaches to matrices here.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required

Transforming a Dual Quaternion Transform	0	0	48	0	43	0	91
Transforming Matrix Transform (naïve)	1	1	134	1	102	1	362
Transforming Matrix Transform (optimized)	1	1	70	1	54	1	234

After seeing the point transformation and cast-to-mat4 costs of a dual quaternion, it's nice to see the dual quaternion win again.

Finale – Six Transforms and then Transforming a Point with Dual Quaternions

Note: I won't do any computation cost analysis here. This is just a finale example to demonstrate the transformation principles that I have shown so far in a non-trivial example.

I don't want to perform some pithy little example of rotating by 60°, then 30° like I did with the complex number rotor and the quaternion rotor because the dual quaternion can also translate things. I'll try something a bit crazy, but not so crazy that I can't verify it in my head.

Remember, to rotate around a vector, imagine the basic 3D vector space: Y is up, X and Z are in the horizontal plane, and +Z is defined as the cross product of +X and +Y. Don't do what I did and confuse this basic 3D vector space with "+X cross +Z = +Y", because that's not right.

I'll start with $< 1,1,0 >$, then:

(1) Rotate around +Y by 90° (goes to $< 0,1,-1 >$), then move +2 in Y (goes to $< 0,3,-1 >$).
(2) Rotate around +Z by 90° (goes to $< -3,0,-1 >$), then move +5 in X (goes to $< 2,0,-1 >$).
(3) Move -4 in Y (goes to $< 2,-4,-1 >$), then rotate 180° around –Z (goes to $< -2,4,-1 >$).

Regardless of whether I use dual quaternion or matrix transforms T, the net transform T_net is formed as follows:

$$T_{net} = T_3 * T_2 * T_1$$

This follows the general transformation order shown above: newest transforms on the left, oldest on the right.

I'll start with $< 1,1,0 >$. I'll use my first transform to rotate it around +Y by 90° and move it up by 2, which should put it at $< 0,3,-1 >$. I'll use my second transform to translate it by -3 in X (putting it at $< -3,3,-1 >$, and then I'll rotate it around +Z by 180°, which should put the point at $< 3,-3,-1 >$. I therefore have a rotation, then a translation, then another translation, and lastly another rotation. This example will hopefully demonstrate the transformational power of the dual quaternion. This could certainly be done with matrices, but I want to show that the dual quaternion can do it for cheaper.

Doing This Transform with Dual Quaternions

There are 5 pages here. I checked my work step-by-step with my implementation of quaternions and dual quaternions just to make sure that everything was kosher. And it works!

Transforming a transform

I'll pick some angles that are easy to do by hand.
Start with point $\langle 1, 1, 0 \rangle$, then:

(1) rotate around +Y by 90°, then move +2 in Y
$\Rightarrow \langle 0, 1, -1 \rangle$ $\Rightarrow \langle 0, 3, -1 \rangle$

(2) rotate around +Z by 90°, then move +5 in X
$\Rightarrow \langle -3, 0, -1 \rangle$ $\Rightarrow \langle 2, 0, -1 \rangle$

(3) move −4 in Y, then rotate 180° around −Z
$\Rightarrow \langle 2, -4, -1 \rangle$ $\Rightarrow \langle -2, 4, -1 \rangle$

↑ expected result

$dq_1 = q_{1r} + q_{1d}\,\varepsilon$

$q_{1r} = (\cos(\frac{90°}{2}), \sin(\frac{90°}{2})\langle 0, 1, 0 \rangle)$
$= (\frac{\sqrt{2}}{2}, \langle 0, \frac{\sqrt{2}}{2}, 0 \rangle)$

$q_{1d} = (0, \frac{1}{2}\langle 0, 2, 0 \rangle)\, q_{1r}$
$= (0, \langle 0, 1, 0 \rangle)(\frac{\sqrt{2}}{2}, \langle 0, \frac{\sqrt{2}}{2}, 0 \rangle)$ parallel vectors ⇒ 0
$= (0(\frac{\sqrt{2}}{2}) - \text{dot}(\vec{v}_1, \vec{v}_2),\ 0(\vec{v}_2) + \frac{\sqrt{2}}{2}(\vec{v}_1) + \text{cross}(\vec{v}_1, \vec{v}_2))$ ~~cross~~ ⇒ 0
$= (-\frac{\sqrt{2}}{2}, \langle 0, \frac{\sqrt{2}}{2}, 0 \rangle)$

$dq_1 = (\frac{\sqrt{2}}{2}, \langle 0, \frac{\sqrt{2}}{2}, 0 \rangle) + (-\frac{\sqrt{2}}{2}, \langle 0, \frac{\sqrt{2}}{2}, 0 \rangle)\,\varepsilon$

$dq_2 = q_{2r} + q_{2d}\,\varepsilon$

$q_{2r} = (\cos(\frac{90°}{2}), \sin(\frac{90°}{2})\langle 0, 0, 1 \rangle)$
$= (\frac{\sqrt{2}}{2}, \langle 0, 0, \frac{\sqrt{2}}{2} \rangle)$

$q_{2d} = (0, \frac{1}{2}\langle 5, 0, 0 \rangle)\, q_{2r}$
$= (0, \langle \frac{5}{2}, 0, 0 \rangle)(\frac{\sqrt{2}}{2}, \langle 0, 0, \frac{\sqrt{2}}{2} \rangle)$
$= (0(\frac{\sqrt{2}}{2}) - \text{dot}(\vec{v}_1, \vec{v}_2),\ 0(\vec{v}_2) + \frac{\sqrt{2}}{2}(\vec{v}_1) + \text{cross}(\vec{v}_1, \vec{v}_2))$
$= (0, \langle \frac{5\sqrt{2}}{4}, 0, 0 \rangle + \langle 0, -\frac{5\sqrt{2}}{4}, 0 \rangle)$
$= (0, \langle \frac{5\sqrt{2}}{4}, -\frac{5\sqrt{2}}{4}, 0 \rangle)$

$dq_2 = (\frac{\sqrt{2}}{2}, \langle 0, 0, \frac{\sqrt{2}}{2} \rangle) + (0, \langle \frac{5\sqrt{2}}{4}, -\frac{5\sqrt{2}}{4}, 0 \rangle)\,\varepsilon$

$dq_3 = q_{3r} + q_{3d}\varepsilon$ translate then rotate, so use the optimized construction to save time (and wrist)

$q_{3r} = q_{rotor\_r}$
$= (\cos(\frac{180°}{2}), \sin(\frac{180°}{2})\langle 0, 0, -1\rangle)$
$= (0, \langle 0, 0, -1\rangle)$

$q_{3d} = (q_{rotor\_r})(q_{trans\_d})$
$= (0, \langle 0, 0, -1\rangle)(0, \frac{1}{2}\langle 0, -4, 0\rangle)$
$= (0, \langle 0, 0, -1\rangle)(0, \langle 0, -2, 0\rangle)$
$= (0(0) - dot(\vec{v}_1, \vec{v}_2), 0(\vec{v}_2) + 0(\vec{v}_1) + cross(\vec{v}_1, \vec{v}_2))$
$= (0, \langle 0, -2, 0, 0\rangle)$

$dq_3 = (0, \langle 0, 0, -1\rangle) + (0, \langle -2, 0, 0\rangle)$

Now we need dq_{net}, which will transform the original point to the expected final position.

$dq_{net} = (dq_3)(dq_2)(dq_1)$

transformations happen later ← → transformations happen earlier

split into $dq_3((dq_2)(dq_1))$

$dq_{temp} = (dq_2)(dq_1)$
$= q_{2r}q_{1r} + (q_{2r}q_{1d} + q_{2d}q_{1r})\varepsilon$

$q_{2r}q_{1r} = (\frac{\sqrt{2}}{2}, \langle 0, 0, \frac{\sqrt{2}}{2}\rangle)(\frac{\sqrt{2}}{2}, \langle 0, \frac{\sqrt{2}}{2}, 0\rangle)$
$= (\frac{\sqrt{2}}{2}(\frac{\sqrt{2}}{2}) - dot(\vec{v}_1, \vec{v}_2), \frac{\sqrt{2}}{2}(\vec{v}_2) + \frac{\sqrt{2}}{2}(\vec{v}_1) + cross(\vec{v}_1, \vec{v}_2))$
$= (\frac{1}{2}, \langle 0, \frac{1}{2}, 0\rangle + \langle 0, 0, \frac{1}{2}\rangle + \langle -\frac{1}{2}, 0, 0\rangle)$
$= (\frac{1}{2}, \langle -\frac{1}{2}, \frac{1}{2}, \frac{1}{2}\rangle)$

↓

$$q_{2r}q_{1d} = \left(\tfrac{\sqrt{2}}{2}, \langle 0, 0, \tfrac{\sqrt{2}}{2}\rangle\right)\left(-\tfrac{\sqrt{2}}{2}, \langle 0, \tfrac{\sqrt{2}}{2}, 0\rangle\right)$$
$$= \left(\tfrac{\sqrt{2}}{2}\left(-\tfrac{\sqrt{2}}{2}\right) - dot(\vec{v}_1, \vec{v}_2), \tfrac{\sqrt{2}}{2}(\vec{v}_2) + \left(-\tfrac{\sqrt{2}}{2}\right)(\vec{v}_1) + cross(\vec{v}_1, \vec{v}_2)\right)$$
$$= \left(-\tfrac{1}{2}, \langle 0, \tfrac{1}{2}, 0\rangle + \langle 0, 0, -\tfrac{1}{2}\rangle + \langle -\tfrac{1}{2}, 0, 0\rangle\right)$$
$$= \left(-\tfrac{1}{2}, \langle -\tfrac{1}{2}, \tfrac{1}{2}, -\tfrac{1}{2}\rangle\right)$$

$$q_{2d}q_{1r} = \left(0, \langle \tfrac{5\sqrt{2}}{4}, -\tfrac{5\sqrt{2}}{4}, 0\rangle\right)\left(\tfrac{\sqrt{2}}{2}, \langle 0, \tfrac{\sqrt{2}}{2}, 0\rangle\right)$$
$$= \left(0\left(\tfrac{\sqrt{2}}{2}\right) - dot(\vec{v}_1, \vec{v}_2), 0(\vec{v}_2) + \tfrac{\sqrt{2}}{2}(\vec{v}_1) + cross(\vec{v}_1, \vec{v}_2)\right)$$
$$= \left(-\left(\tfrac{5\cdot 2}{4\cdot 2}\right), \langle \tfrac{5\cdot 2}{4\cdot 2}, -\tfrac{5\cdot 2}{4\cdot 2}, 0\rangle + \langle 0, 0, \tfrac{5}{4}\rangle\right)$$
$$= \left(+\tfrac{5}{4}, \langle \tfrac{5}{4}, -\tfrac{5}{4}, \tfrac{5}{4}\rangle\right)$$

for dual part

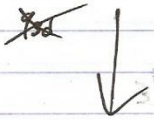
$$q_{2r}q_{1d} + q_{2d}q_{1r} = \left(-\tfrac{1}{2}, \langle -\tfrac{1}{2}, \tfrac{1}{2}, -\tfrac{1}{2}\rangle\right) + \left(+\tfrac{5}{4}, \langle \tfrac{5}{4}, -\tfrac{5}{4}, \tfrac{5}{4}\rangle\right)$$
$$= \left(\tfrac{3}{4}, \langle \tfrac{3}{4}, -\tfrac{3}{4}, \tfrac{3}{4}\rangle\right)$$

$$dq_{temp} = \left(\tfrac{1}{2}, \langle -\tfrac{1}{2}, \tfrac{1}{2}, \tfrac{1}{2}\rangle\right) + \left(\tfrac{3}{4}, \langle \tfrac{3}{4}, -\tfrac{3}{4}, \tfrac{3}{4}\rangle\right)\varepsilon$$

$$dq_{net} = (dq_3)(dq_{temp})$$
$$= q_{3r}q_{+r} + (q_{3r}q_{+d} + q_{3d}q_{+r})\varepsilon$$

$$q_{3r}q_{+r} = \left(0, \langle 0, 0, -1\rangle\right)\left(\tfrac{1}{2}, \langle -\tfrac{1}{2}, \tfrac{1}{2}, \tfrac{1}{2}\rangle\right)$$
$$= \left(0\left(\tfrac{1}{2}\right) - dot(\vec{v}_1, \vec{v}_2), 0(\vec{v}_2) + \tfrac{1}{2}(\vec{v}_1) + cross(\vec{v}_1, \vec{v}_2)\right)$$
$$= \left(-\left(0 - 1\left(\tfrac{1}{2}\right)\right), \langle 0, 0, -\tfrac{1}{2}\rangle + \langle \tfrac{1}{2}, \tfrac{1}{2}, 0\rangle\right)$$
$$= \left(\tfrac{1}{2}, \langle \tfrac{1}{2}, \tfrac{1}{2}, -\tfrac{1}{2}\rangle\right)$$

$$q_{3r}q_{+d} = \left(0, \langle 0, 0, -1\rangle\right)\left(\tfrac{3}{4}, \langle \tfrac{3}{4}, -\tfrac{3}{4}, \tfrac{3}{4}\rangle\right)$$
$$= \left(0\left(\tfrac{3}{4}\right) - dot(\vec{v}_1, \vec{v}_2), 0(\vec{v}_2) + \tfrac{3}{4}(\vec{v}_1) + cross(\vec{v}_1, \vec{v}_2)\right)$$
$$= \left(-\left(-1\left(\tfrac{3}{4}\right)\right), \langle 0, 0, -\tfrac{3}{4}\rangle + \langle -\tfrac{3}{4}, -\tfrac{3}{4}, 0\rangle\right)$$
$$= \left(\tfrac{3}{4}, \langle -\tfrac{3}{4}, -\tfrac{3}{4}, -\tfrac{3}{4}\rangle\right)$$



$q_{3d}\, q_{tr} = (0, \langle -2, 0, 0 \rangle)\left(\frac{1}{2}, \langle -\frac{1}{2}, \frac{1}{2}, \frac{1}{2} \rangle\right)$

$= \left(0\left(\frac{1}{2}\right) - \text{dot}(\vec{v}_1, \vec{v}_2),\ 0(\vec{v}_2) + \frac{1}{2}(\vec{v}_1) + \text{cross}(\vec{v}_1, \vec{v}_2)\right)$

$= \left(-(-2(-\tfrac{1}{2})), \langle -\tfrac{1}{2}, 0, 0 \rangle + \tfrac{1}{2}\langle 0, 1, -1 \rangle\right)$

$= \left(-1, \langle -\tfrac{1}{2}, 1, -1 \rangle\right)$

for dual part

$q_{3r} q_{td} + q_{3d} q_{tr} = \left(\frac{3}{4}, \langle -\frac{3}{4}, -\frac{3}{4}, -\frac{3}{4} \rangle\right) + \left(-1, \langle -\tfrac{1}{2}, 1, -1 \rangle\right)$

$= \left(-\frac{1}{4}, \langle -\frac{7}{4}, \frac{1}{4}, -\frac{7}{4} \rangle\right)$

$dq_{net} = \left(\frac{1}{2}, \langle \frac{1}{2}, \frac{1}{2}, -\frac{1}{2} \rangle\right) + \left(-\frac{1}{4}, \langle -\frac{7}{4}, \frac{1}{4}, -\frac{7}{4} \rangle\right)\varepsilon$

Whewww. Now I'll transform the point.

12-23-2014 Use the optimization suggested when transforming a point.

dual part of point-as-dual quat: $(0, \frac{1}{2}\langle 1, 1, 0 \rangle)$ ← original point

$\Rightarrow \text{point}_{\text{quat dual}} = (0, \langle \frac{1}{2}, \frac{1}{2}, 0 \rangle)$

transformed point as dual quat =

$dq_{net}.\text{real} + (dq_{net}.\text{real} \cdot \text{point}_{\text{dual}} + dq_{net}.\text{dual})\varepsilon$

$\hookrightarrow = \left(\frac{1}{2}, \langle \frac{1}{2}, \frac{1}{2}, -\frac{1}{2} \rangle\right)(0, \langle \frac{1}{2}, \frac{1}{2}, 0 \rangle)$

$= \left(\frac{1}{2}(0) - \text{dot}(\vec{v}_1, \vec{v}_2),\ \frac{1}{2}(\vec{v}_2) + 0(\vec{v}_1) + \text{cross}(\vec{v}_1, \vec{v}_2)\right)$

$= \left(-\left(\frac{1}{2}\left(\frac{1}{2}\right) + \frac{1}{2}\left(\frac{1}{2}\right) + \left(-\frac{1}{2}\right)(0)\right), \langle \frac{1}{4}, \frac{1}{4}, 0 \rangle + \langle \frac{1}{4}, -\frac{1}{4}, 0 \rangle\right)$

$= \left(-\frac{1}{2}, \langle \frac{1}{2}, 0, 0 \rangle\right)$

for dual point

$dq_{net}.\text{real} \cdot \text{point}_{\text{dual}} + dq_{net}.\text{dual} =$

$\left(-\frac{1}{2}, \langle \frac{1}{2}, 0, 0 \rangle\right) + \left(-\frac{1}{4}, \langle -\frac{7}{4}, \frac{1}{4}, -\frac{7}{4} \rangle\right) = \left(-\frac{3}{4}, \langle -\frac{5}{4}, \frac{1}{4}, -\frac{7}{4} \rangle\right)$

transformed point as dual quat =

$\left(\frac{1}{2}, \langle \frac{1}{2}, \frac{1}{2}, -\frac{1}{2} \rangle\right) + \left(-\frac{3}{4}, \langle -\frac{5}{4}, \frac{1}{4}, -\frac{7}{4} \rangle\right)\varepsilon$

Finally, extract the point

dual and real of ~~pt~~ transformed point dual quat

$$\text{extracted point}_{quat} = 2 \cdot dual \cdot real^*$$

$$real = \left(\tfrac{1}{2}, \langle \tfrac{1}{2}, \tfrac{1}{2}, -\tfrac{1}{2}\rangle\right)$$
$$real^* = \left(\tfrac{1}{2}, \langle -\tfrac{1}{2}, -\tfrac{1}{2}, \tfrac{1}{2}\rangle\right)$$

$$dual \cdot real = \left(-\tfrac{3}{4}, \langle -\tfrac{5}{4}, \tfrac{1}{4}, -\tfrac{7}{4}\rangle\right)\left(\tfrac{1}{2}, \langle -\tfrac{1}{2}, -\tfrac{1}{2}, \tfrac{1}{2}\rangle\right)$$
$$= \left(-\tfrac{3}{4}\left(\tfrac{1}{2}\right) - dot(\vec{v}_1, \vec{v}_2),\ -\tfrac{3}{4}\langle -\tfrac{1}{2}, -\tfrac{1}{2}, \tfrac{1}{2}\rangle + \tfrac{1}{2}\langle -\tfrac{5}{4}, \tfrac{1}{4}, -\tfrac{7}{4}\rangle + cross(\vec{v}_1, \vec{v}_2)\right)$$
$$= \left(-\tfrac{3}{8} - \left(-\tfrac{5}{4}\left(-\tfrac{1}{2}\right) + \tfrac{1}{4}\left(-\tfrac{1}{2}\right) + \left(-\tfrac{7}{4}\right)\left(\tfrac{1}{2}\right)\right),\ \langle \tfrac{3}{8}, \tfrac{3}{8}, -\tfrac{3}{8}\rangle + \langle -\tfrac{5}{8}, \tfrac{1}{8}, -\tfrac{7}{8}\rangle + \langle -\tfrac{3}{4}, \tfrac{3}{2}, \tfrac{3}{4}\rangle\right)$$
$$= \left(-\tfrac{3}{8} - \tfrac{5}{8} + \tfrac{1}{8} + \tfrac{7}{8},\ \langle \tfrac{3}{8} - \tfrac{5}{8} - \tfrac{3}{8} - \tfrac{3}{4},\ \tfrac{3}{8} + \tfrac{1}{8} + \tfrac{3}{2},\ -\tfrac{3}{8} - \tfrac{7}{8} + \tfrac{3}{4}\rangle\right)$$
$$= \left(0, \langle -\tfrac{8}{8}, \tfrac{16}{8}, -\tfrac{4}{8}\rangle\right) \quad (-\tfrac{6}{8},\ \tfrac{12}{8},\ -\tfrac{6}{8})$$
$$= \left(0, \langle -1, 2, -\tfrac{1}{2}\rangle\right)$$

Typo here in the addition, so I added the negative signes to teh X and Z components in MS Paint.

$$\text{extracted point}_{quat} = 2\left(0, \langle -1, 2, -\tfrac{1}{2}\rangle\right)$$
$$= \left(0, \langle -2, 4, -1\rangle\right)$$

transformed point = <2, 4, -1>

yay! It is what ~~I use~~ I expected!

And that's the way that dual quaternions crumble[14].

Full Frame Analysis

I have already established that the dual quaternion approach to generating rotate-then-translate and translate-then-rotate transforms are significantly cheaper than their matrix counterparts, but there's a couple wrinkle in the analysis: Vectors are significantly cheaper to transform than with dual quaternions, and GPU hardware is designed to rip through matrix math, not dual quaternions. So the dual quaternion transform must be converted to a 4x4 matrix before being practically useful. **I now want to take this into account.**

I will assume that I am using my own object transform process, which is to transform objects relative to where they are. I do not keep track of absolute object locations in 3D space, but rather keep track of their transform. On each frame, I move the object relative to where it was. For example, suppose I have a cube somewhere. On the first frame, I want it to move left and rotate counterclockwise. On the second frame, I want it to move up and rotate fast counterclockwise. Each frame moves the object relative to where it was.

A full frame of computations therefore requires generating the transform for an object, creating a net transform by transforming the existing object's transform with the new one, and then sending the net transform result off to the vertex shader so that the GPU can transform each vertex. Dual quaternions require the added step of converting to a 4x4 matrix.

I will now compare the following processes:

- Dual quaternion:
  - Generate transform
  - Transform existing transform
  - Convert to matrix
- Matrix:
  - Generate transform
  - Transform existing transform

For dual quaternions, rotate-then-translate and translate-then-rotate transforms differ in cost slightly, so I'll analyze them both.

As I did previously, I will assume that the current object's transform already exists. This cost analysis compares everything between there and spitting out a net transform.

Dual Quaternion: Rotate Then Translate

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Full Frame Analysis – Dual Quat R-then-T	1	1	134	1	86	1	273

[14] I felt I had to put something clever here after all this time. This was the best that I could come up with.

Breakdown:

- Sin: 1 from transform generation
- Cos: 1 from transform generation
- Multiplication:
    - 27 from transform generation
    - 48 from transforming the transform (dual quaternion multiplication)
    - 59 from casting to mat4
- Division:
    - 1 from transform generation
- Add/Sub:
    - 15 from transform generation
    - 43 from transforming the transform
    - 28 from casting to mat4
- Square Root:
    - 1 from transform generation
- Temporary Floats:
    - 54 from transform generation
    - 91 from transforming the transform
    - 128 from casting to mat4

Dual Quaternion: Translate Then Rotate

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Full Frame Analysis – Dual Quat T-then-R	1	1	137	1	86	1	274

Breakdown:

- Sin: 1 from transform generation
- Cost: 1 from transform generation
- Multiplication:
    - 30 from transform generation
    - 48 from transforming the transform
    - 59 from casting to mat4
- Division:
    - 1 from transform generation
- Add/Sub:
    - 15 from transform generation
    - 43 from transforming the transform
    - 28 from casting to mat4
- Square Root:
    - 1 from transform generation
- Temporary Floats

- 55 from transform generation
- 91 from transforming the transform
- 128 from casting to mat4

Matrix

Thanks to the const reference to a glm::mat4 in glm::rotate(…) and glm::translate(…), generating the new transform and transforming the existing transform can be rolled into one step. I will again be thorough and copy both the naïve and optimized computation costs for transforming the transform.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Transforming Matrix Transform (naïve)	1	1	134	1	102	1	362
Transforming Matrix Transform (optimized)	1	1	70	1	54	1	234

Comparing Matrices and Dual Quaternions

I'll copy the computation costs of the full frame analysis of both matrices and dual quaternions here for easy comparison.

Operation Cost	sin	cos	mul	div	add/sub	sqrt	Temporary floats required
Full Frame Analysis – Dual Quat R-then-T	1	1	134	1	86	1	273
Full Frame Analysis – Dual Quat T-then-R	1	1	137	1	86	1	274
Transforming Matrix Transform (naïve)	1	1	134	1	102	1	362
Transforming Matrix Transform (optimized)	1	1	70	1	54	1	234

Well, that's a sobering picture. Despite the promises of dual quaternions to be cheaper than matrices when being generated and transforming transforms, the lack of ability to combine the generation and transforming the transform and the requirement to cast it to a mat4 before rendering nearly kills it's competitiveness with matrices.

Nearly.

Conclusion

Dual quaternions offer a novel way to combine both the translation and rotational aspects of a transformation. While cheaper to construct and transform than matrices, they are significantly more expensive to use when transforming a point. This computation expensive plus the fact that GPU

hardware is designed to work with matrices and not dual quaternions demands that a dual quaternion transform be converted to a matrix prior to rendering, and this incurs its own cost. In every frame, it is highly likely that only one transform will be created for each object, and that this new transform will be used to modify the existing one to create a net transform for that object. Due to the requirement to cast to a mat4, dual quaternions loose the computation competitive edge to matrices by a long shot.

But dual quaternions have two notable advantages over matrices: They utilize quaternions to encode orientation, thus avoiding gimbal lock, and they are much easier (??verify??) to normalize than matrices.

My conclusion on the matter is this: **Matrices have the performance edge over dual quaternions, but in the corner cases when gimbal lock is a threat and the transform looses normalization due to float approximations, dual quaternions can save your bacon. Yum.**

Dual Quaternion Application

The Code

See my code repo (??insert link here??). It's a public repo because I don't want to pay the github fee, but it's also my private experimental repo, so please be nice and don't push anything to it ☺.

Storing Full Transforms of Objects in a Scene

Storing Both Translation and Orientation

Before I switched to dual quaternions, I was using a glm::vec3(…) for entity position in world space and a quaternion for orientation. The dual quaternion transform stores both pieces of information.

The Destination Vector Space of the Dual Quaternion is Not World Space

The transform is applied to model (or "object", depending on the terminology you like to use) space, but the result does not appear to be entirely in world space. The dual quaternion is a map from model space to its own vector space, and all my computations work out to put the pixels in the right space, but I couldn't figure out exactly where an object was in world space given only its dual quaternion transform. I could extract the translational component from the dual part, but the translational component was only due to direct translational movement, such as going up or down, forward and back, and strafing left and right with my keyboard controls.

Testing: I tilted an object so that it was not level with respect to the X-Z plane, moved it in a circle using the strafe and yaw controls (example: strafe left and yaw right), and printed out the translation vector of the dual quaternion to the console (recall that the translation component $= (2.0f * dq_{dual} * dq^{*}_{real}).\vec{v}$).

Findings: I expected it to therefore move up and down in Y as it moved around the circle, but it didn't. Only the X and Z components changed.

Conclusion: As far as the dual quaternion transform was concerned, the object was still level.

This fact suggested to me that the object might be transformed in its own model space, but I am hesitant to believe that because the dual quaternion transform performs a rigid transformation, and any rigid transformation moves the source vertices to a difference vector space. It certainly appears similar to model space, but whatever this destination vector space is, it is not world space (**Note**: In my code, I

am still calling the matrix produced by the dual quaternion model-to-world matrix, which, as I've tried to explain here, is not correct) (??TODO: remove this note when you correct your code??).

??TODO: verify results with another object; you only performed these computations with the camera's entity??

Lighting Must Be Done in Camera Space

Before I started doing specular lighting in my Sandbox_Game_2 project, I was doing diffuse lighting in world space, and all was well. Then I started implementing specular lighting, and performing these computations required a vertex-to-camera vector. That meant that I needed to know exactly where the camera was in world space, but as I described in the previous paragraph, I couldn't figure out where it was.

I spent a few hours trying to figure this out, but eventually I decided to follow the example of the Arcsynthesis tutorial and implement the fragment shader (this is where I am doing my lighting equations) in camera space, in which the camera's position in implicitly the origin (<0,0,0>).

Preparing Vertices for Camera Space

Since the fragment shader requires camera space locations, I had to prepare my light positions, which are in world space, and my matrix uniforms for my vertex positions and vertex normal.

I got the world-to-camera matrix (again, this is not quite correct; a more correct term would be "dual-quaternion-space-to-camera" matrix) (??TODO: remove this note when you correct your code??) from my camera (see the Camera section).

Uniform Vectors: Lights: Light positions were put into camera space by multiplying the world-to-camera matrix (!!) by the light vector (a glm::vec4 with the w component being 1.0f to enable translation). I could have done this in the vertex shader, but the light position is independent of any geometry, so I put the light positions into model space before I began looping through and drawing my list of renderables and put the result into a uniform that the fragment shader can grab.

Uniform Matrix 1: I got the camera-to-clip matrix from the glm::perspective(…) function.

Uniform Matrix 2: The model-to-camera matrix was unique for each entity and resulted from the following matrix multiplication:

```
glm::mat4 model_to_camera = camera_mat * r.m_model_to_world_mat;
```

Where "r" is a "renderable" class that stores the transform for a particular geometry and a pointer to that geometry class. On each frame, the renderable for each entity updates its matrix by casting the entity's dual quaternion to a glm::mat4. Please note that this is my own code, but I hope that the generic application is apparent.

Vertex Shader: Preparing Incoming Vertices for Camera Space

My vertex shader receives position, normal, and color vectors as vec3s with each vertex. The position and normal are in model space. The model-to-camera matrix and camera-to-clip matrix are uniforms in this shader. Outputs to the fragment shader are the position in camera space, the normal in camera space, and the color, all as vec3s (this is a personal preference because I like to work with just vec3s in the fragment shader).

The color is simply passed straight through without modification.

A temporary position vector is created by casting the input position vector in model space to a vec4 with a w component of 1.0f to enable translation, then transforming it with the model-to-camera matrix. The output position in camera space is form by casting this temporary position vector back to a vec3. The gl_Position value is computed by transforming the temporary position vec4 with the camera-to-clip matrix.

The output normal is created by casting the input normal vector to a vec4 with a w component of 0.0f to disable translation, transforming it with the model-to-camera matrix, and then casting it back down to a vec3. I don't mind this double casting, but if it bugs you, then you'll have to create a third matrix uniform, a glm::mat3 created by jamming the model-to-camera matrix into a glm::mat3 constructor, effectively chopping off the fourth row and column. Going this route requires generating another matrix for every renderable, which isn't very expensive, but I just preferred to only use one matrix and explicitly cast the vertex normal to a vec4 and disable translation, transform it, and explicitly cast it back down to a vec3.

Camera

The world-to-camera matrix is quite simple. The following code is how I do it:

```
glm::mat4 Camera::get_world_to_view_matrix() const
{
   Math::F_Dual_Quat dq = m_where_and_which_way;
   dq.m_dual *= -1.0f;

   return dq.to_mat4();
}
```

??"Look at" algorithm??