EECS 351 : Introduction to Computer Graphics

3D Geometric Operations

Jack Tumblin, ver. 1.9

RECALL we defined a (geometrical) 'vector' as the result of *subtraction* of 2 points; it has direction and length, but no position. What happens when we apply addition, multiplication, and division, averaging, weighted sums (or "LERPing"), and other math operators to points and vectors? Which combinations of operators and operands offer sensible tools for 3D shapes, and which ones have no geometric meaning?

--(Point,Point): Binary: ~~Add?~~ Subtract(makes a vector), Average, weighted sum, ~~multiply?~~ ~~divide?~~
Unary: ~~Scale?~~ ~~Sign-change~~?(BAD! Results vary with choice of origin point)

--(Point,Vector) Binary: Add, Subtract(point minus vector makes a point), ~~average~~, ~~weighted sum~~, ~~multiply~~, ~~divide~~?

--(Vector,Vector)? Binary: Add, Subtract (makes a vector), Average, Weighted sum, multiply (two ways—see below), ~~divide~~?
Unary: Scale? Sign-change?

3-D Operators for Points vs. Vectors

Example:

--Write two 4-element vectors E and F as columns: $[ex\ ey\ ez\ 0]^T$ and $[fx\ fy\ fz\ 0]^T$.

Length of a vector E: $||\mathbf{E}|| = \text{sqrt}(ex^2 + ey^2 + ez^2)$

--JARGON: '**Normalize**' a vector == Scale the vector to make its length =1.0:

norm(E) = (ex/||E||, ey/||E||, ez/||E||)

--JARGON: a '**unit vector**' is a vector whose length is 1.0. Normalizing always creates a unit vector.

--Puzzle: does it make any sense geometrically to 'normalize' points?

(**ANS:** No, it doesn't. By definition, a point has neither 'length' nor 'magnitude': it's a location!)

--Puzzle: What should happen to 'w' in a 4-tuple when you normalize it?

(**ANS:** nothing! if you normalize a vector, w==0 before and after. Normalize a point?!? don't!!)

--Puzzle: Suppose you define a 'unit cube' centered at the origin whose 8 corner points have (x,y,z,w) coordinates given by (+/-1, +/-1, +/-1, 1). What is the distance between opposite corners of the cube?

ANS: We can choose any 2 opposing corner points, but I choose
P1 = (1,1,1,1) and P0 = (-1,-1,-1,1). The vector that reaches from P0 to P1 is given by
V01 = (P1 – P0) = (1,1,1,1) – (-1,-1,-1,1) = (1,1,1,1) + (1,1,1,-1) = (2,2,2,0).
The length of this vector is $\text{sqrt}(2^2 + 2^2 + 2^2 + 0^2) = \text{sqrt}(3 \cdot 4) = 2 \cdot \text{sqrt}(3)$.

--Puzzle: Suppose you define a cube whose 8 corner points have (x,y,z,w) coordinates given by (+/- L, +/- L, +/- L, 1). We could then write 8 vectors that reach from the origin to each cube corner as (+/- L, +/- L, +/- L, 0). What is the length of each vector? ANS: length = $L \cdot \text{sqrt}(3 \cdot L^2) = L \cdot \text{sqrt}(3)$.
What distance separates cube corners that share an edge? ANS: 2L: (separation of adjacent corners)

--Puzzle: Suppose we chose fixed A, B, C values to define 8 vectors (+/-A, +/-B, +/-C, 0). We use these to define the corners of a rectangular solid by adding these 8 vectors to the origin point (0,0,0,1). If we 'normalize' these 8 vectors, will it change the shape of the solid? Does it yield a cube?
ANS: no; all 8 vectors have the same length (length = $\text{sqrt}(A^2 + B^2 + C^2 + 0^2)$), and thus normalizing them scales each of their magnitudes by 1/length, all magnitudes==1, but the directions of these vectors don't change. This centered rectangular solid keeps its shape; non-cubes do not become cubes.

--Puzzle: For a unit cube centered at point (a,b,c,1) what 8 vectors reach from the origin to the corners of the cube? ANS: $[a +/-1,\ b +/-1,\ c +/-1,\ 0]^T$.
--Puzzle: If we normalize those 8 vectors, what happens to the shape they describe? Can any vector have zero length (can't be normalized)? ANS: shape depends strongly on a, b and c values; if any are +/-1, the cube has 1 or more vertices at the origin. Normalizing the vectors will move each corner towards or away from the origin to a distance of 1.0 (onto a sphere of radius 1) without changing vector direction. Sketch it on paper, or in WebGL software: what happens when a,b,c are all small (<1)? large (>>1)?

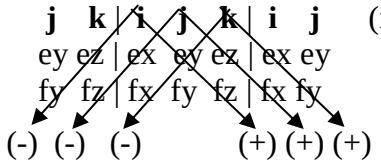
The vector 'add' and 'subtract' operators are obvious, but how can we 'multiply' two vectors? Math provides (at least) 2 ways:

1) Dot product of vectors E and F (written **E·F, or "E dot F"), finds a scalar value**:

--the 3D inner product, a *projection:*
it means "find how much of E is in the F direction; multiply that by the length of F"
-- More formally: **E·F** = ex·fx + ey·fy + ez·fz, and therefore it commutes: **E·F = F·E**
-- **E·F** = ||E|| ||F|| cos θ, where θ is the angle between the vectors (if connected at one vertex)
-- Dot products can find angles, too; easiest if we make ||E|| ||F|| = 1 by normalizing **E** and **F** first.
ANS: if ||E|| ||F|| = 1, then **E·F** = cos θ; θ= arccos(**E·F**)
if E is perpendicular to F, **E·F = ?** **ans: 0**, because none of vector E is in the F direction.
if E is parallel to F, **E·F = ?** **ans: ||E|| ||F||**
if E is perpendicular to the plane Π,
E·F > 0 if E and F aim towards the *same side* of the plane;
E·F < 0 if E and F aim towards opposite plane sides, and =0 if F is parallel to plane.

2) **Cross product** of vectors E,F (written **E x F, or "E cross F") finds a vector value,**

-the 3D outer product; finds the vector '**N**' that's perpendicular to E an perpendicular to F.
N = E x F
--The N vector points in the direction perpendicular to the plane formed by E and F.
--Within that plane, E and F define 2 sides of a parallelogram.
The area of that parallelogram is the length of the vector **N**.
--How do we compute a cross product?
E x F = [ey·fz – ez·fy, ez·fx – ex·fz, ex·fy – ey·fx]
Nobody can remember that! I find it easier to remember as a 3x3 matrix determinant:



(i,j,k are unit vectors in the x,y,z directions)

--**ANTI**-COMMUTATIVE! E x F = **-**(F x E); vector of same magnitude, but opposite direction.
--Like dot products, cross-products can also find angles between 3D vectors E,F:
|| **E x F** || = ||**E**|| ||**F**|| sin B
if E is perpendicular to F, ||**E x F**|| = ||**E**||·||**F**|| **,** the area of a rectangle with E,F as edges
or if E is parallel to F, ||**E x F**|| = **0**

---Puzzles: if we can 'multiply' vectors in two ways, can you find a way to reverse those processes two ways, by defining two kinds of 'vector-divide' operators? What procedure would you devise to undo the result of a dot-product? Would it require you to know one or both of the unknown vectors' lengths?

Or their directions? What procedure would you devise to undo the result of a cross-product? Would it require you to know one or both of the unknown vectors' lengths? Or their directions?

For yet another concise yet thorough review of dot-products and cross-products, complete with many helpful solved problems, see "Vector Analysis" – Murray R. Spiegel, Schaum's Outline Series in Mathematics, McGraw-Hill Book Co. For more details, consult a good textbook such as "An Introduction to Linear Algebra" by Gilbert Strang, or "*Mathematics for 3D Game Programming and Computer Graphics*" Eric Lengyel, (2013). Chapters 2,3,4.

ASIDE: This course, WebGL, OpenGL, and most books on graphics use column-vector notation. However, the authors of the Windows' DirectX library chose to use row-vector notation instead. This simple linear algebra theorem converts between column-style (OpenGL) and row-style(DirectX) vectors and matrices. Recall that if we multiply a column vector v_0 by matrix M we will get a new column vector v_1; and we can write it as $v_1 = Mv_0$.
Recall that a row-vector is just the transpose of a column vector:

$$v_0 == \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \quad v_0^T == [x\ y\ z\ w].$$

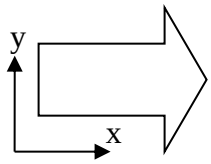
Theorem: if $v_1 = Mv_0$, then $v_1^T = v_0^T M^T$

Why? Remember, 'vectors' are just skinny matrices; use
Theorem: For matrices AB, $(AB)^T = B^T A^T$

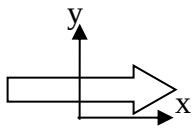
3D TRANSFORMATIONS

How can you translate (slide up/down/left/right), rotate, zoom, and shear a drawing?
How does order of applying these matrix operations affect the geometric results?
Given all the vertices of the 2D shape on the left, how can you make these shapes?

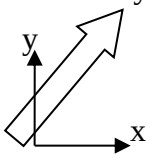
Start with this:

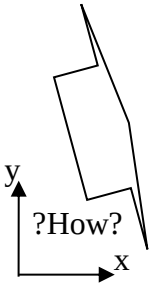


Scale in y,
then Translate in x:



Then Rotate around origin,
then Translate in y:





-Define the 'canvas' as part of a plane in 3D, measured in x,y,z (for convenience, think z=0 for now).
-Define a 3D 'vertex' as a point on that x,y,z plane; write it as a 4-element column vector $[x,y,z,1]^T$.
We'll soon explain more about the '1' (after we show how it enables us to make translation matrices).
-Almost all useful 3D manipulations for all parts of shapes defined by 3D vertices
consist of a sequence of 'rigid body' transformations; three basic operations in some useful sequence.

--Rigid-Body transformations:

Translate (shift position of the shape by some desired direction and magnitude)
Rotate (turn the object around some specified axis by some specified angle)
Scale (shrink or enlarge the shape, scaling all coordinates by the same 'scale factor')

--We can write each one of these transformations as matrix,

--We can use ordinary matrix multiplication to combine or 'concatenate' them;
We can combine *any* transformation sequence, no matter how complex, into *just one matrix*.
If you write a vertex location point $(x_0,y_0,z_0,1)$ as a 4-element column vector v_0 and multiply one or more of the matrices below, you can make a new, transformed vertex position v_1:

Scale Matrix:
$$\mathbf{S} = \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
(action: x *= sx; y *= sy; z *= sz; and w coordinate does not change)

Translate Matrix :
$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
(action: x += tx; y += ty; z+= tz; and w coordinate does not change)

Rotate$_z$ Matrix:
$$\mathbf{R_z} = \begin{bmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
(action: rotate CCW around the z axis; and w, z coordinates do not change. c, s are the sines, cosines of angle theta)

Graphics conventions:
--We write vectors in lower case, matrices in upper case; point v, matrix M.
--We always write 3D vertices, 3D points, and 3D vectors as 4-element 'column' vectors;
--when applied to a matrices *precede* vectors: $v_1 = \mathbf{M}v_0$
v' = **TRS**v changes vector v into vector v' by applying **S**, then **R**, then **T**.
--Theorem: matrix multiplies are associative, but ***not*** commutative; **TRS**v != **SRT**v:
--Theorem: commutation requires arranged transposes. For A,B matrices: $\mathbf{(AB)^T = B^T A^T}$

Sanity Check: MUST we transform every point on every line of every drawing? NO.
IF we transform the vertices only, then use LERPs to 'connect the dots',
will we get the same drawing? YES!
--In OpenGL/WebGL we specify all drawing primitives as a sequence of vertices:
edges as point-pairs, line-strip (or 'paths' in PhotoShop) and triangles as a list of vertices, and meshes as a sequence of triangles. Only the triangles matter, not their sequence: the same triangles drawin in a different sequence yields the same drawing.
--In OpenGL/WebGL, we transform ONLY the vertices,
then draw onscreen the lines or surfaces between the *transformed* vertices.

?What guarantees that 'vertex-only' transforms work correctly for all shapes?
Ans: By 'linear invariance' (and later 'projective invariance'): every point in every original edge specifies a unique point in the transformed edge, and any transformed straight-line edge always gives a straight line result. (You can prove this yourself – try it! Express an edge as a LERP, let parameter t vary from 0 to 1: 0<=t<=1. Given two vertices v0, v1, every point p along the edge between v0 and v1 is p(t) = v0 + (v1-v0)*t. What happens when you linearly transform p(t)? ANS: lines transform to lines for any 4x4 matrix).

Geometrically, the order we apply these T,R, and S matrices is crucial!
Results from translate-then-rotate are dramatically different from rotate-then-translate!
How do we know which to use?

Demo: How to 'Compose' a sequence of 2D transforms:
https://www.youtube.com/watch?v=CmgSye6-Ack (from Eric Haines short-course)
How can we do this mathematically? What is the ordering of the matrices?

Geometric Duality:
Matrix multiplies have only one meaning mathematically, but TWO meanings geometrically. This is a true geometric duality: both are valid, consistent, provably correct description of the process, yet they are not equivalent. You need to know BOTH of them, as OpenGL uses both, but in different ways.

Confusing the two or swapping them indiscriminately is a sure path to grief. Remember the two as:

Method 1--a **transformation of vertices** that
---keeps the drawing axes unchanged;
(Same coordinate system: same origin, same coordinate vectors **i,j,k** both *before* and *after* the transformation), but
---moves the vertices; it changes their numbers, their xyz coordinate values:
(Different x,y,z,w values before and after transformation)

Method 2--a **transformation of drawing axes** that
---moves the drawing axes; changes the coordinate system:
(Different origin, different coordinate vectors **i,j,k** *after* the transformation), but
---keeps the vertices unchanged;
(Same coordinate values, same x,y,z,w values before and after transformation).

==========

SURPRISE!
The OpenGL /WebGL specifications use *Method 2* to describe all transformations:
gl.translate(), gl.rotate(), gl.scale(), etc.
(…and for a very good reason…)
We will examine this 'duality' very carefully and thoroughly. Our textbook ignores it, mostly, as do many introductory computer graphics books.

==========

If you don't gain complete mastery of this 'duality', then you'll find that OpenGL/WebGL "just never seems to works right" for you. Your robot's legs sometimes walk away from its torso: your camera aiming controls work 'backwards' and often look away from your intended direction of gaze, and your lights never stay where you put them, and "You're Gonna Have a Bad Time!"