05: Linear Algebra

> files relating to this lecture:
>
> - assignment
> - matrix.html

Trigonometry review (30 min)

polar coordinates and cartesian coordinates

Last class we drew circles using Cartesian coordinates and the formula $x^2 + y^2 = 1$, our dots were not evenly spaced along the curve, they were evenly spaced along the X-axis. we needed a way of thinking "circularly". We needed to use polar coordinates.

Centered at the origin, the **angle** was now the thing we iterated in our loop. We used cosine and sine to convert a reguarly-incrementing angle into a drawing of a circle. We didn't have to worry about distance, our circle's radius was 1.

> Both polar (θ,d) and Cartesian (x,y) make use of two variables, and they can describe every point in 2D space. 2D requires at least two variables; 3D requires at least three...

Unit circle review

Why does a unit circle work so well? What has a length 1? (triangle hypotenuse, circle radius). What does **not** have length 1? (Sine, cosine, tangent). When possible make this **hypotenuse / radius** your entry point, multiply all values by this length and everything will scale up accordingly.

cosine and sine

- input: angle
- output: a Cartesian point, the endpoint of the radius line

Tangent, a visual explanation

review what the word *tangent* even means by drawing some wiggly curves and drawing little tangent segments along them. In trigonometry, in the unit circle, the specific tangent line we're referring to is the one that is **vertical at the far right end of the circle** (at the point [1,0]).

- input: angle
- output: the **length** of the tangent line

this requires a little drawing. extend the hypotenuse line until it hits the tangent line, *that* point, measure the length of it down to the x axis. That's what tangent means. a negative value is when we're below the x-axis.

Inverse trig functions

This is simple, they ask the opposite question, in the above examples, flip the input and output: *what angle* gives you x? In the case of tangent it's not so simple...

> two ways of writing "inverse sine": $\sin^{-1}$ and arcsine (arcsin). **Prefer arc- notation**.

atan2( ), "give me the angle for a point" (convert cartesian to polar)

I cannot understate how useful atan2 is to media artists!

Let's try out arctangent:

1. pick a length (example: 5), and draw it on the board as a vertical line.
2. Now we can fill in the rest:
    - draw an x-axis at the bottom of the line
    - draw a unit circle tangent to the line (make it radius 1 in relationship with your given length)
    - draw the y axis.
3. draw the hypotenuse and the angle! done!

Inverse tangent gives you the **angle between the x axis and the point**. It converts a length into an angle. We saw how when the length is positive it's in quadrant 1, negative: quadrant 4. But what about quadrants 2 and 3?

atan2( ) solves this by asking for more information: the original x and y point to be able to answer the question "which quadrant did it come from?"

atan2 is basically atan with a if-statement at the beginning.

> In **every other instance** x comes before y. In atan2, y is first. This is because the original programmer imagined the traditional trig notation of putting y over x (y/x) and saying y first.

code an example that draws a big right-pointing arrow at the center, rotates around its center to follow the mouse. *image search for an arrow, put the URL in the "..."*

```
// p5.js
var img;

function setup() {
  createCanvas(400, 400);
  img = loadImage("...");
```

```
  imageMode(CENTER);
}

function draw() {
  background(220);
  translate(200, 200);
  rotate(Math.atan2(mouseY - 200, mouseX - 200));
  image(img, 0, 0);
}
```

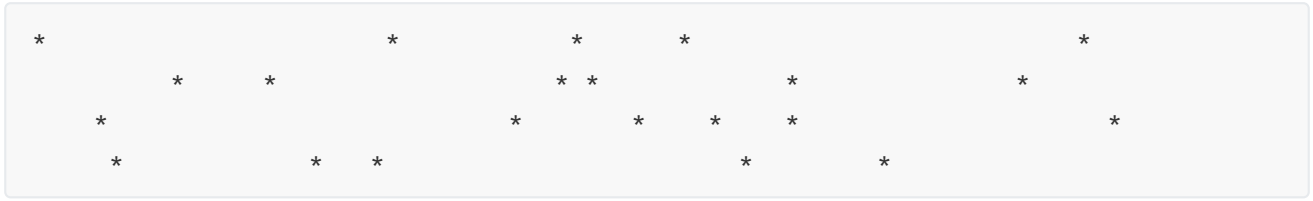
We first wrote the `image` function as `image(img, 200, 200)` but learned that the rotation was still happening at the top-left corner. we moved the 200, 200 into its own line `translate(200, 200)` so it can happen *before* the rotation. **Transformations are not communicative**, more on that later.

> I often think of the translate as happening *after* the rotate, even though it's before, because I think of coordinate spaces from an object-perspective, whereas OpenGL thinks of them from a world-perspective. Translations kind of happen in reverse... Honestly I still find it confusing to this day.

Why do we need linear algebra?

can you calculate the average of two angles? *average the 2 numbers.*

What are some inconsistencies with this method? sometimes it calculates the "interior" angle, sometimes not. If the two angles are near the 0 where it flips +360 suddenly...



let's talk about distance searching, like the question "what point is nearest?". Given a field of points:

Q: what is the highest point (closest to the top)?

A: simply compare the y axis values.

Q: given a point, what are all the points to the right and left? Or in-front-of and behind?

A: if the person is facing 90degrees, the answer is easy, comparing the x axis and y axis values.

What happens when the axis becomes *anything other* than the 90 degrees of the X and Y axis? both of these shortcuts fail.

Linear Algebra

this lecture is heavily based on Grant Saunderson's Essence of Linear Algebra. huge debt of gratitutde!

Vectors

a vector is two things at once, a list of numbers and an arrow with a length and a direction.

A drawing of the vector is always assuming it **starts at the origin**. I can't draw two vectors spaced apart from one another, if I'm going to *move a vector from the origin* we would need more data, we're not bothering with this yet.

draw two vectors and write them out in component form. I'm going to write them as the first example from here on.

$$\mathbf{v} = \begin{bmatrix} 3 \\ 1 \end{bmatrix} \ or \ \ \mathbf{v} = (3, 1)$$

vector addition

a vector plus a vector results in a vector. adding two vectors is placing one at the end of another, and drawing a new line to the last tip. its the same as when we were introduced to addition as children, by moving steps (two numbers of steps) on the number line. step through two vectors by moving x1, y1, then x2, y2. then do all the Xs at once and then all the Ys. it's the same as adding each component first, in the list form.

show an example of *visually* adding two vectors. Then translate these visual drawings into components, let's say they look like (3,1) and (-1,2). Show how the **components can be added**.

$$\begin{bmatrix} 3 \\ 1 \end{bmatrix} + \begin{bmatrix} -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 - 1 \\ 1 + 2 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

The vector (2,3) should match with our drawing!

Vector addition is communicative. I could have added one vector to the other or reversed it.

scalar multiplication

a number times a vector results in a vector. the number "scales" the vector, stretching or squishing it along the same direction. 2v stretches the vector by a factor of 2. 1/3 shrinks it. -3 reverses the direction and scales it by 3. We can also perform the operation numerically:

$$2 \begin{bmatrix} 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \cdot 3 \\ 2 \cdot 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 2 \end{bmatrix}$$

why do we need to normalize?

```
vec.normalize()
```

a normalized vector removes the concept of distance, but retains the "angle" idea. if we add two vectors that are normalized, we solve the bisect two angles problem! but they need to be normalized.

epsilon

when does the example above fail-adding two vectors to get the bisection? *when they are 180° apart.*

the answer becomes close to zero (but not zero, exactly), we need to make use of **epsilon comparison**.

epsilon is a tiny number, basically zero, but large enough that we can call anything above it to be "not zero". checking if something is almost zero in code needs to consider the negative number case. It can be done a couple ways

```
if (value < EPSILON && value > -EPSILON)
```

```
if (Math.abs(value) < EPSILON)
```

basis vectors

let me introduce two special vectors: **î** and **ĵ** (*eye hat* and *jay hat*), they are each unit vectors, length of 1, and they each run along the two axes in the positive direction. **î** is [1,0] and **ĵ** is [0,1].

for a moment, think of vectors in a new way: each component **scales the corresponding basis vector**. take the vector [2,3] for example. imagine each component is a scalar, the first scales î and the second scales ĵ. so a [2,3] vector is 2 times the X unit vector, and 3 times the Y unit vector. the advantage of doing this is that we can start to think about *changing the unit vectors*.

$$\begin{bmatrix} 2 \\ 3 \end{bmatrix} = 2 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + 3 \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

span

imagine new unit vectors. pick two random vectors and don't worry about number coordinates, just draw them. Consider: adding together *scaled* versions of these new î and ĵ, can we hit every point in the plane?

this is what **span** is. the span of these two vectors is all of 2D space.

when does this not work? when vectors lie on top of one another, right? this is a broken system (these vectors are *linearly dependent*). the span of these two vectors is just a line. we definitely prefer our basis vectors to be *linearly independent*.

matrices as transformations

transform is a special term for function. in math, functions take in something, like a number and spits out another number, in linear algebra, a function (transformations) takes in a vector and spit out another vector.

transforms are things you're familiar with, rotation, scale, shear (we're going to see why translation is a special case, if you're curious, in our conversation from day 2 on transformations, there was a special kind where the *origin never changed*).

a transformation is said to be **a linear transformation** as long as the grid lines stay parallel, straight, evenly-spaced, and the origin stays in the same place.

parallel and evenly-spaced

when you stretch and rotate space, you move the vector to its new location, but you also move the unit X and Y vectors. and this is what makes linear transformations so cool, you can describe the entire transformation, for all vectors, by just writing where the unit X and Y vectors end up. this is because all of our vectors can be described as linear combinations of those unit vectors.

if we define a transform as simply "where the basis vectors end up" then to transform a vector is simply to multiply the x component by the X-basis vector and the y component by the Y.

$$\mathbf{v} = x \begin{bmatrix} \hat{\imath}_x \\ \hat{\imath}_y \end{bmatrix}, y \begin{bmatrix} \hat{\jmath}_x \\ \hat{\jmath}_y \end{bmatrix}$$

the convention is to store the two transformed $\hat{\imath}$ and $\hat{\jmath}$ in a matrix each as a column.

$$\begin{bmatrix} \hat{\imath}_x & \hat{\jmath}_x \\ \hat{\imath}_y & \hat{\jmath}_y \end{bmatrix}$$

Matrix vector multiplication laid out looks like:

$$\begin{bmatrix} a & c \\ b & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = x \begin{bmatrix} a \\ b \end{bmatrix} + y \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} ax + cy \\ bx + dy \end{bmatrix}$$

(oh no. we have to talk about matrix column/row order)

Important! When you are looking at a matrix, in all the mess of the numbers you are looking at column vectors.

$$\begin{bmatrix} \begin{pmatrix} a \\ b \end{pmatrix} \begin{pmatrix} c \\ d \end{pmatrix} \end{bmatrix}$$

we can now multiply vectors by matrices! (what about multiplying matrices by matrices)

run through some examples multiplying a vector by a matrix

- rotation (90 degree)
- scale

- shear

Translation

until this point, translation has been entirely left out of the conversation.

Line representations: (point, vector) (point, point) (vector, scalar-distance-to-center)

Classwork

scale a polygon in SVG only using a matrix. Let's do this by creating an SVG from scratch. start out with:

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 100 100">
  <circle cx="50" cy="50" r="10" />
</svg>
```

modify this by adding another circle *behind* the first circle (before), give it a gray fill and a transformation. We are going to edit the matrix components directly.

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg" viewBox="0 0 100 100">
  <circle cx="0" cy="0" r="10" fill="gray" transform="matrix(1 0 0 1 0 0)" />
  <circle cx="50" cy="50" r="10" />
</svg>
```

This matrix is the identity matrix. Change the `matrix()` entry to create a skew effect, the shadow falls of into the distance creating an illusion of depth.

> anytime you add affine transforms to a shape, you **must set it's coordinates to (0, 0)**

Edit the shadow (first circle):

1. set cx and cy to 0, move these translation components into the matrix (final column)
2. modify the matrix to create a skew effect

```
matrix(1 0 -1 0.5 60 55)
```

Create a rotation matrix

apply our knowledge from trig to create a matrix that rotates at ANY degree.

Make a drawing showing the basis vectors rotating around 30°, 60°, see how they are tracing the outline of a circle? We know how to define these points!

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

Combining transformations

the key to multiplying matrices is that *inside* each matrix are simply vectors. the basis vectors. we can multiply *those* by matrices, to get the columns of the resulting matrix.

matrix multiplication goes from **right to left**.

are transforms **communicative**? can you rotate then scale? same as scale then rotate?

associative? yes!

a matrix that is a product of two or more matrices is called a **composition matrix**.

Homework

Make a "fire-escape" type of a map with instructions as a sequence of matrices.

Make a sign that gives instructions for how to get from one place to another place in ITP. Present the matrices in two ways: a sequence of transform matrices, and one composition matrix that represents it all. The units are in human-steps.

References

- Measurement by Paul Lockhart, section 2 Time and Space.
- Essence of linear algebra by Grant Saunderson
- Paul Bourke, Geometry