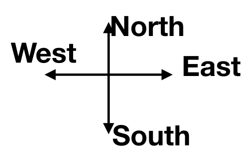
CSC 433/533
Computer Graphics

Rotations - more perspectives
Transforming from one coordinate system to another

- From Linear algebra: A **basis** $\{\vec{v}_1, \vec{v}_2 \dots \vec{v}_d\}$ is a set of vectors such that every point p in a space(plane/space…) could be expressed as a **linear combination,** $p = \alpha_1 \cdot \vec{v}_1 + \alpha_2 \vec{v}_2 + \dots + \alpha_d \cdot \vec{v}_d$ … and the scalars $\{\alpha_1 \dots \alpha_d\}$ are unique.
- The space is **spanned** by this basis.
- Multiplication by a matrix M is a linear operation: That is
  - $M \cdot \vec{0} = \vec{0}$
  - $M \cdot (\vec{u} + \vec{v}) = M\vec{u} + M\vec{v}$
  - $M(\alpha \vec{u}) = \alpha(M\vec{u})$

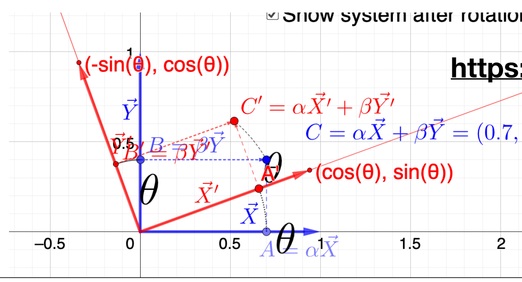


- We are all very familiar with the basis $\vec{X} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ $\quad and \quad$ $\vec{Y} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$
- For example, if $v = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$ then $v = 2\vec{X} + 3\vec{Y}$, could be understood as

Start from Origin, walk 2 meters in the $\vec{X}$ direction, followed by 3 meters in the $\vec{Y}$ direction.

We can express rotation by creating a new basis of $\mathbb{R}^2$

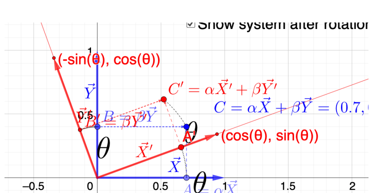
- **To specify a rotation, it is sufficient to create a new coordinates system, and specify what is the correspondence between the old and new basis.**
- **To be precise, create M, such that the it's column of M is the $i'th$ vector (represented as a linear combination of the basis)**
- The text above is probably very cryptic without multiple examples
- "Tricky" way to find rotation matrix. If $\vec{X}, \vec{Y}$ are unit vectors in the old coordinate system, then we could think about the ro as rotating the coordinates systems as well, and after the rotation we expect
- $\vec{X} \xrightarrow[by\ \theta]{Rotation} \vec{X'}$ **and** $\vec{Y} \xrightarrow[by\ \theta]{Rotation} \vec{Y'}$ **.** Let R_θ be the rotation matrix. This means: $R_\theta \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \vec{X'}$ **and** $R_\theta \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \vec{Y'}$**.**
- **But** $R_\theta \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \vec{X'}$ **is just the first column of** R_θ **. And** $R_\theta \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ **is the second column.**
- **Lets try: Write** $R_\theta = \begin{bmatrix} \vdots & \vdots \\ \vec{X}' & \vec{Y}' \\ \vdots & \vdots \end{bmatrix}$**.**
- **then for every data point** $C = (\alpha, \beta)$ **, we could write (in a somehow obnoxious way)** $C = \alpha \vec{X} + \beta \vec{Y}$**.**
- $R_\theta \cdot C = \begin{bmatrix} \vdots & \vdots \\ \vec{X}' & \vec{Y}' \\ \vdots & \vdots \end{bmatrix} \left(\alpha \vec{X} + \beta \vec{Y} \right) \overset{\text{linearity}}{=} \alpha \begin{bmatrix} \vdots & \vdots \\ \vec{X}' & \vec{Y}' \\ \vdots & \vdots \end{bmatrix} \vec{X} + \beta \begin{bmatrix} \vdots & \vdots \\ \vec{X}' & \vec{Y}' \\ \vdots & \vdots \end{bmatrix} \vec{Y} = \alpha \vec{X}' + \beta \vec{C}'$



https://www.geogebra.org/m/upxwfnuc

We can express rotation by creating a new basis of $\mathbb{R}^2$

- This is going to be extremely useful when discussing rotations in 3D
- To specify a rotation, it is sufficient to create a new coordinates system, and specify what is the correspondence between the old and new basis.
- To be precise, create M, such that the it's **column** of M is the $i'th$ vector (represented as a linear combination of the old basis)
- The text above is probably very cryptic without multiple examples
- "Tricky" way to find rotation matrix. If $\vec{X}, \vec{Y}$ are unit vectors in the old coordinate system, then we could think about the rotation as rotating the coordinates systems as well, and after the rotation we expect
- $\vec{X} \xrightarrow[by\ \theta]{Rotation} \vec{X'}$ and $\vec{Y} \xrightarrow[by\ \theta]{Rotation} \vec{Y'}$. Let R_θ be the rotation matrix. This means $R_\theta \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \vec{X'}$ and $R_\theta \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \vec{Y'}$.
- But $R_\theta \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \vec{X'}$ is just the first column of R_θ . And $R_\theta \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ is the second column.
- Lets try: Write $R_\theta = \begin{bmatrix} \vdots & \vdots \\ \vec{X}' & \vec{Y}' \\ \vdots & \vdots \end{bmatrix}$.
- then for every data point $C = (\alpha, \beta)$, we could write (in a somehow obnoxious way) $C = \alpha \vec{X} + \beta \vec{Y}$.
- $R_\theta \cdot C = \begin{bmatrix} \vdots & \vdots \\ \vec{X}' & \vec{Y}' \\ \vdots & \vdots \end{bmatrix} \left(\alpha \vec{X} + \beta \vec{Y} \right) \overset{\text{linearity}}{=} \alpha \begin{bmatrix} \vdots & \vdots \\ \vec{X}' & \vec{Y}' \\ \vdots & \vdots \end{bmatrix} \vec{X} + \beta \begin{bmatrix} \vdots & \vdots \\ \vec{X}' & \vec{Y}' \\ \vdots & \vdots \end{bmatrix} \vec{Y} = \alpha \vec{X}' + \beta \vec{C}' = C'$
- Important take home message: To find the rotation matrix, just create a matrix where each column is one of the new basis vector (written using coordinates of the old coordinate system)



Operations on Images

Slides inspired from Fredo Durand

- Point (Range) Operations:

- Affect only the range of the image (e.g. brightness)
- Each pixel is processed separately, only depending on the color

Operations on Images

Slides inspired from Fredo Durand

- Domain Operations:


- Only move the pixels around

Operations on Images

Slides inspired from Fredo Durand

- Neighborhood operations:


- Combine domain and range
- Each pixel evaluated by working with other pixels nearby

Concept for the Day: Pixels are Samples of Image Functions

Jim Blinn's Corner

http://www.research.microsoft.com/~blinn/

What Is a Pixel?

James F. Blinn

Microsoft Research

I am not a major sports fan. But there is one story from the folklore of football that has always intrigued me. The story goes that legendary football coach Vince Lombardi was observing his new players, recruited from the best college teams and all presumably excellent players. He was, however, not pleased with their performance. So he called them all to a meeting, which he began by holding up the essential tool of their trade and saying, "This is a football." I was sufficiently impressed by this back-to-basics attitude that, when I taught computer graphics rendering classes, I used to start the first lecture of the term by going to the blackboard (boards were black then, not white) and drawing a little dot and saying, "This is a pixel."

But was I right? Most computer graphicists would agree that the pixel is the fundamental atomic element of imaging. But what is a pixel really? As I have played with various aspects of pixel mashing, it has occurred to me that the concept of the pixel is really multifaceted

that spirit I am going to list some possible meanings for a pixel that I will expand on in some later columns.

A pixel is a little square

Early 2D windowing systems considered a pixel a little square. This is perhaps the simplest possible definition, but it only works if you are mostly drawing horizontal and vertical lines and rectangles that are integral numbers of pixels in size. Anything at fractional pixel size or at an angle yields jaggies and other forms of aliasing.

A pixel is a point sample of a continuous function

A more enlightened signal processing approach thinks of a pixel as a point sample of a continuous function (see the dots in Figure 1). (This approach was actually taken by the rendering community long before pixel displays were used for user interfaces and windowing

Image Samples

- Each pixel is a sample of what?
  - One interpretation: a pixel represents the intensity of light at a single (infinitely small point in space)
- The sample is displayed in such a way as to spread the point out across some spatial area (drawing a square of color)

Continuous vs. Discrete

- Key Idea: An image represents data in either (both?) of
  - Continuous domain: where light intensity is defined at every (infinitesimally small) point in some projection
  - Discrete domain, where intensity is defined only at a discretely sampled set of points.

Converting Between Image Domains

- When an image is acquired, an image is **sampled** from some continuous domain to a discrete domain.
- **Reconstruction** converts digital back to continuous.
- The reconstructed image can then be **resampled** and **quantized** back to the discrete domain.

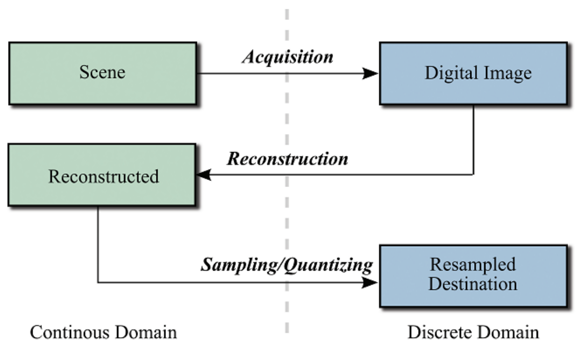


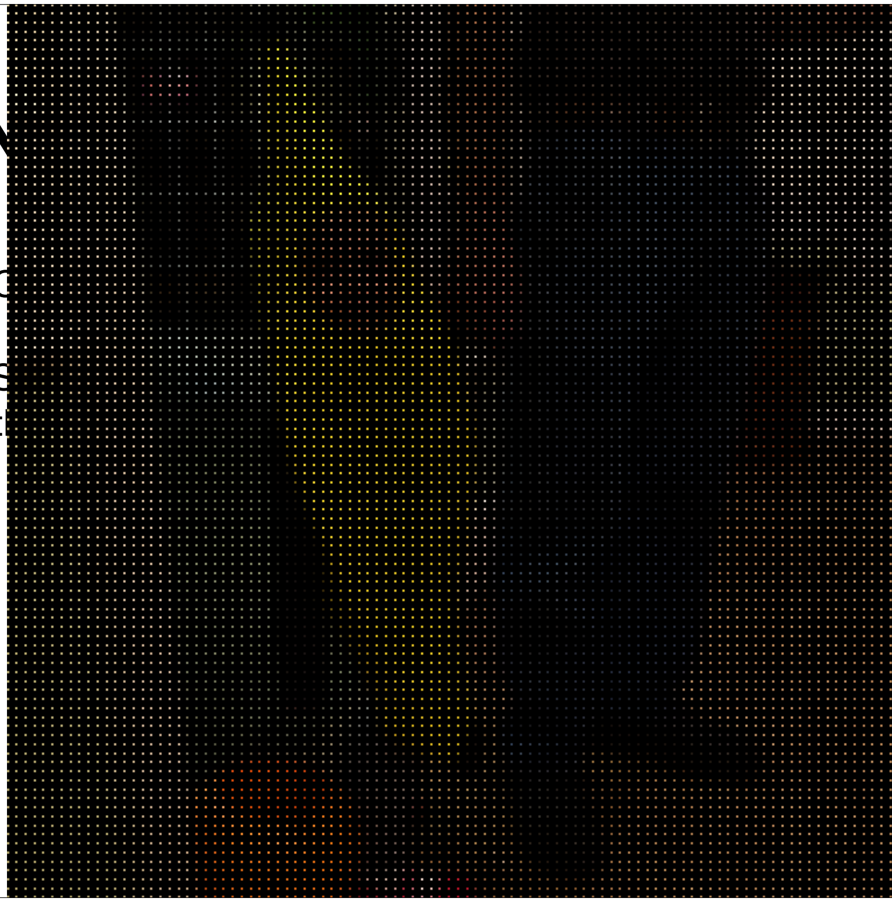
Figure 7.7. Resampling.

Naive Image Rescaling Code

```
//scale factor
let k = 4;

//create an output greyscale image that is both
//k times as wide and k times as tall
Uint8Array output = new Uint8Array((k*W)*(k*H));

//copy the pixels over
for (let row = 0, row < H; row++) {
   for (let col = 0; col < W; col++) {
      let index = row*W + col;
      let index2 = (k*row)*W + (k*col);
      output[index2] = input[index];
   }
}
```

N g

- C
- S
  t



What’s the Problem?

- The output image has gaps!
- Why: we skip a many of the pixels in the output.
- Why don’t we fix this by changing the code to at least put some color at each pixel of the output?

Naive Image Rescaling Code

```
//scale factor
let k = 4;

//create an output greyscale image that is both
//k times as wide and k times as tall
Uint8Array output = new Uint8Array((k*W)*(k*H));

//copy the pixels over
for (let row = 0, row < H; row++) {
   for (let col = 0; col < W; col++) {
      let index = row*W + col;
      let index2 = (k*row)*W + (k*col);
      output[index2] = input[index];
   }
}
```

"Inverse" Image Rescaling Code

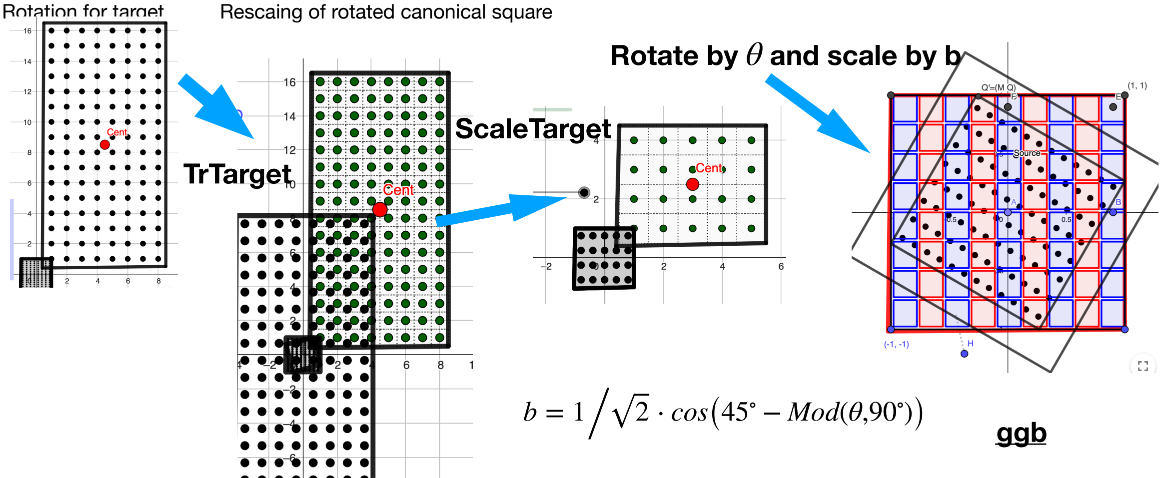
```
//scale factor
let k = 4;

//create an output greyscale image that is both
//k times as wide and k times as tall
Uint8Array output = new Uint8Array((k*W)*(k*H));

//Loop over each output pixel instead.
for (let row = 0, row < k*H; row++) {
   for (let col = 0; col < k*W; col++) {
      let index = (row/k)*W + (col/k);
      let index2 = row*k*W + col;
      output[index2] = input[index];
   }
}
```

Ir g

- It is nice to transform the Source image Isou and the target image Itar, both, so all pixels are in the “canonical square” $x \in [-1,1],\ y \in [-1,1]$.
- In general the image Is file has pixels Isou(i,j), where $i = 1 \ldots Ny$ (number of rows), and $j = 1 \ldots Nx$.
- Sift so center in (0,0) : $\mathbf{T}rTar = \begin{pmatrix} 1 & 0 & -\frac{NxTarget-1}{2} \\ 0 & 1 & -\frac{NyTarget-1}{2} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} j \\ i \\ 1 \end{pmatrix}.$ $\quad \mathbf{S}caTarget = \begin{pmatrix} -\frac{2}{NxTarget} & 0 & 0 \\ 0 & \frac{2}{NyTarget} & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} j \\ i \\ 1 \end{pmatrix}$

Rotation by β: $R = \begin{pmatrix} cos\beta & -sin\beta & 0 \\ \sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{pmatrix}$

- Rotation for target Rescaing of rotated canonical square



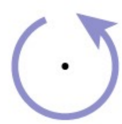
$$b = 1 \big/ \sqrt{2} \cdot cos\left(45^\circ - Mod(\theta, 90^\circ)\right)$$

$$q' = TrSource^{-1} \cdot ScalSource^{-1} \cdot Scale3 \cdot R \cdot ScaTarget \cdot TrTarget \cdot q$$

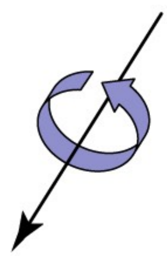
Rotations in 3D

- In 2D, a rotation is about a point
- In 3D, a rotation is about an axis

convention: positive rotation is CCW



2D



3D

convention: positive rotation is CCW when axis vector is pointing at you

Rotations about 3D Axes

- In 3D, we need to pick an axis to rotate about

$$\text{rotate-z}(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

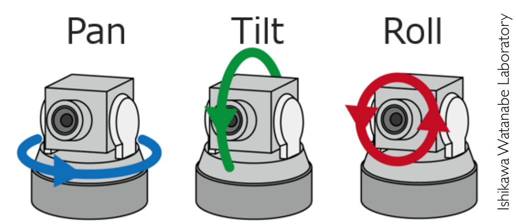
- And we can pick any of the three axes

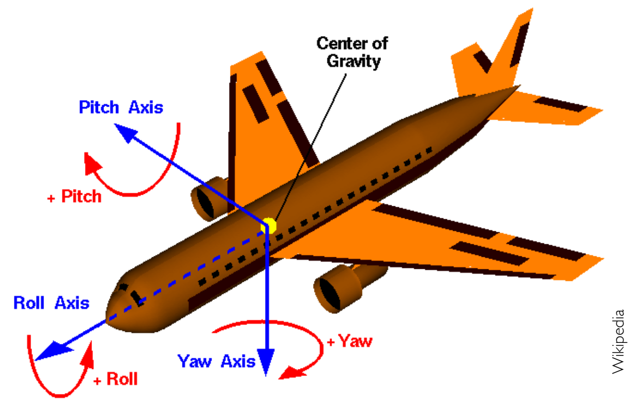
$$\text{rotate-x}(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix}$$

$$\text{rotate-y}(\phi) = \begin{bmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{bmatrix}$$

Building Complex Rotations from Axis-Aligned Rotations

- Rotations about **x**, **y**, **z** are sometimes called **Euler angles**
- Build a combined rotation using matrix composition





Arbitrary Rotations

- To rotate about any axis: we change the coordinate space we are working in, using orthogonal matrices.
- Consider orthogonal matrix R_{uvw}, form by taking three orthogonal vectors $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$:

Property of orthogonal vectors:

$$\mathbf{u} \cdot \mathbf{u} = \mathbf{v} \cdot \mathbf{v} = \mathbf{w} \cdot \mathbf{w} = 1$$
$$\mathbf{u} \cdot \mathbf{v} = \mathbf{v} \cdot \mathbf{w} = \mathbf{w} \cdot \mathbf{u} = 0$$

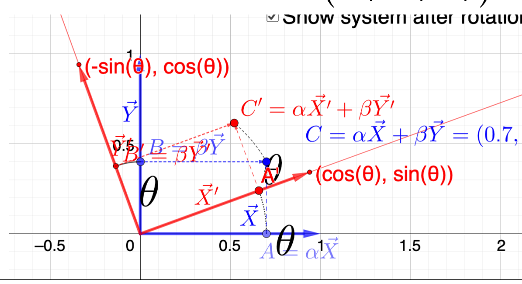
$$\mathrm{R}_{uvw} = \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \\ \mathbf{w} \end{bmatrix}$$

Arbitrary Rotations

- In multiple cases, we are have calculated the camera parameters, $\vec{u}, \vec{v}, \vec{w}$ and Up vectors , and want to rotate the camera, so perform a transformation a transformation taking $\vec{u} \to \vec{x}, \quad \vec{v} \to \vec{y}, \quad \vec{w} \to \vec{z}.$ (Assume camera =(0,0,0))
- Technique one: compute by how much we need to rotation around each axis - very tedious.
- Technique two: Use the matrix R_{uvw}, where

$$R_{uvw} = \begin{pmatrix} - & \vec{u} & - \\ - & \vec{v} & - \\ - & \vec{w} & - \end{pmatrix}$$

$$(R_{uvw})^{-1} = \textbf{transpose}(R_{uvw}) = \begin{pmatrix} | & | & | \\ \vec{u} & \vec{v} & \vec{w} \\ | & | & | \end{pmatrix}$$



Arbitrary Rotations

- What happens when we apply R_{uvw} to any of the basis vectors, e.g.:

$$\mathbf{R}_{uvw}\mathbf{u} = \begin{bmatrix} \mathbf{u}\cdot\mathbf{u} \\ \mathbf{v}\cdot\mathbf{u} \\ \mathbf{w}\cdot\mathbf{u} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \mathbf{x}$$

- But this means that if we apply R_{uvw}^{T} (the transpose of R_{uvw}) to the Cartesian coordinate vectors, e.g.:

$$\mathbf{R}_{uvw}^{\mathrm{T}}\mathbf{y} = \begin{bmatrix} x_u & x_v & x_w \\ y_u & y_v & y_w \\ z_u & z_v & z_w \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} x_v \\ y_v \\ z_v \end{bmatrix} = \mathbf{v}$$

Recall: Vector Multiplication

- Given two 3D vectors:

$$\mathbf{a} = (x_a, y_a, z_a) \quad \mathbf{b} = (x_b, y_b, z_b)$$

- So far, we've learned two forms for "multiplication":
  - Dot (inner) product (2 vectors in, 1 scalar out)

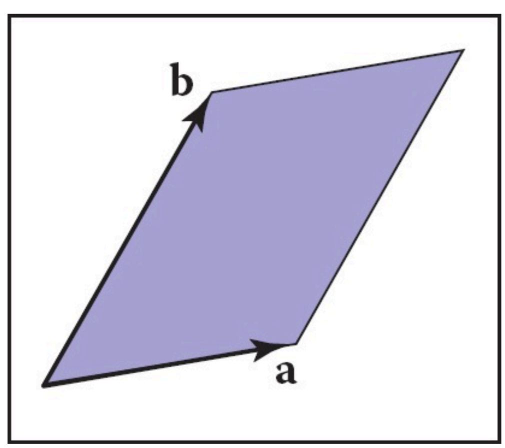
$$\mathbf{a} \cdot \mathbf{b} = x_a x_b + y_a y_b + z_a z_b$$

  - Cross product (2 vectors in, 1 vector out)

$$\mathbf{a} \times \mathbf{b} = (y_a z_b - z_a y_b, z_a x_b - x_a z_b, x_a y_b - y_a x_b)$$

Determinants as Vector Multiplication

- Usually thought of as an operation on a matrix (similar to vector norms) that produces a scalar, but they can also be considered a multiplication of vectors:
- For 2d vectors **a** and **b**, the **determinant**, |**ab**|, is equal to the *signed* area of the parallelogram formed by **a** and **b**
  - **Signed** here means that |**ab**| = -|**ba**|
  - Related: ||**a** × **b**||



Determinants as Vector Multiplication

- For 3d vectors **a**, **b**, and **c**, the determinant, |**abc**|, is the *signed* volume of the parallelepiped formed by **a**, **b**, and **c**

- Sign refers to left-handed or right-handed coordinate system