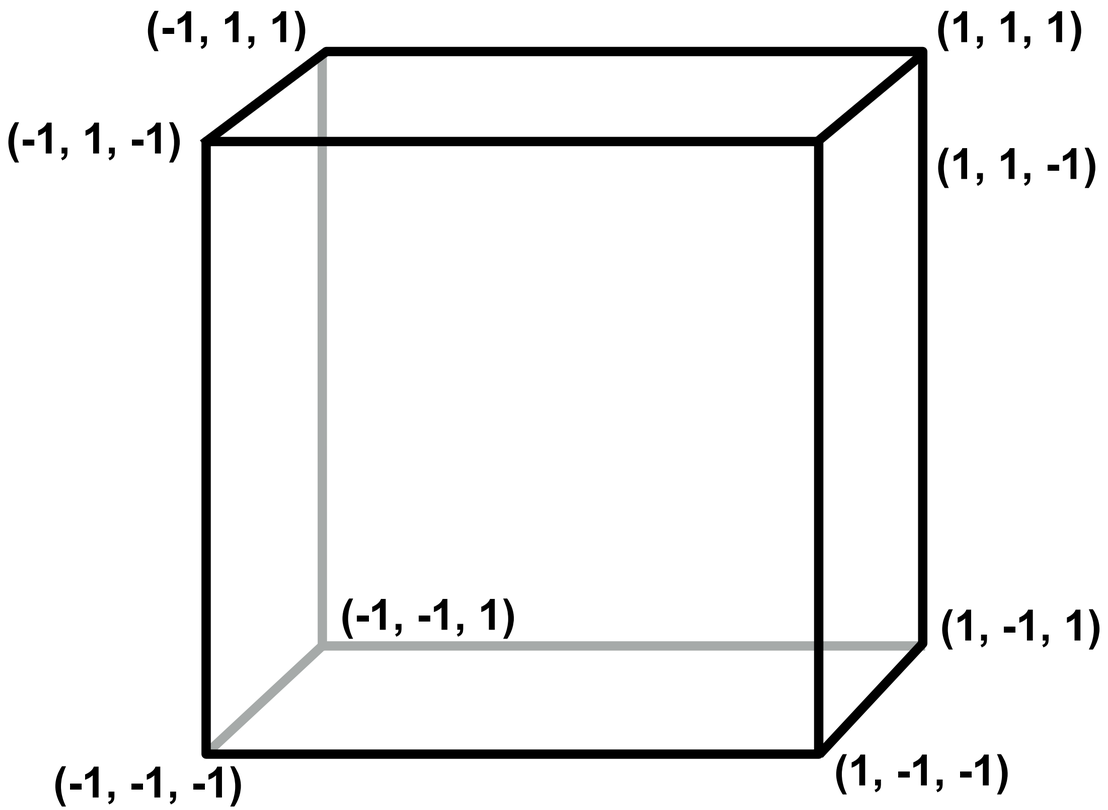
Computer Graphics

-Transformation

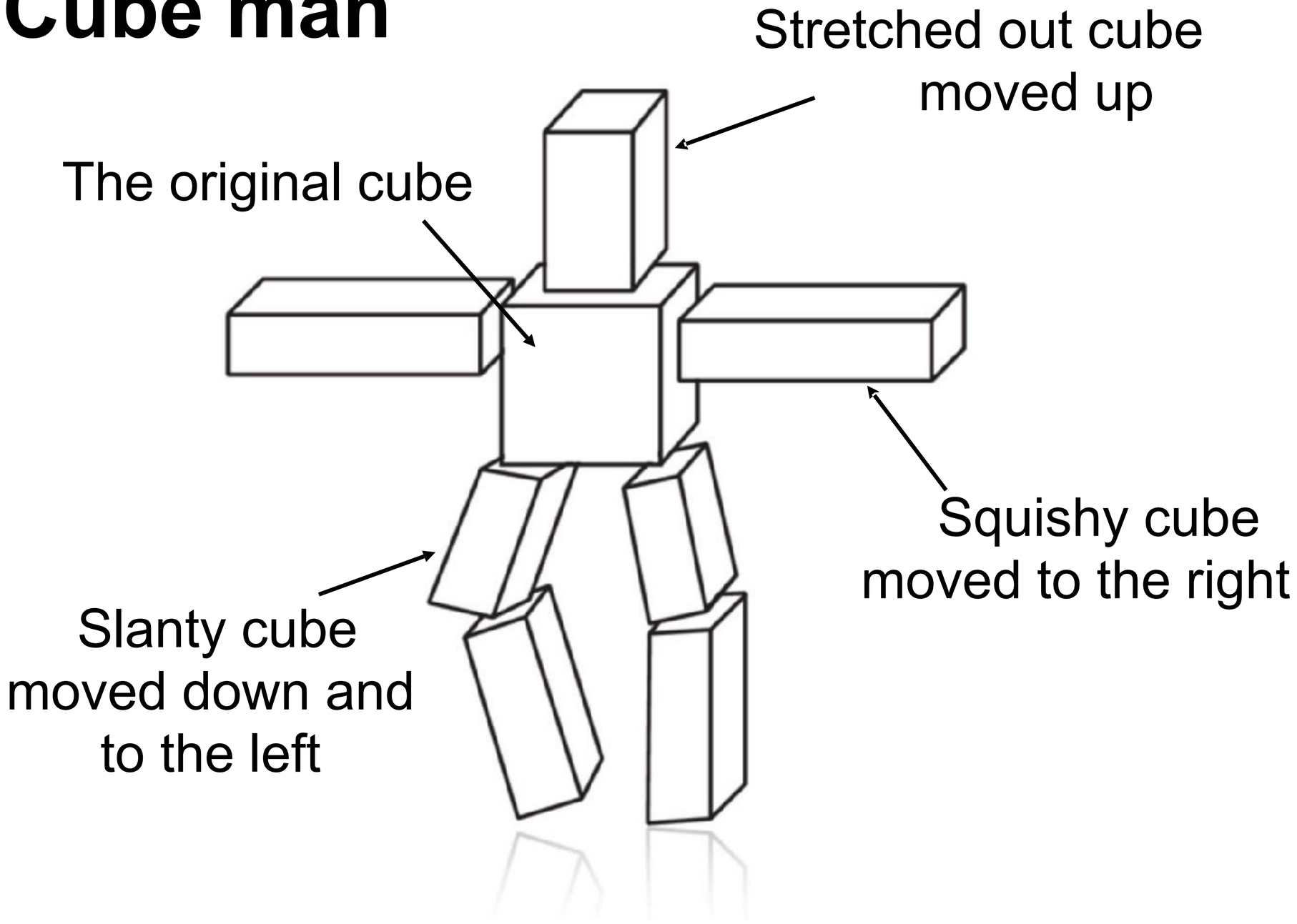
Junjie Cao @ DLUT
Spring 2019
http://jjcao.github.io/ComputerGraphics/

Pleasure may come from illusion, but happiness can come only of reality.

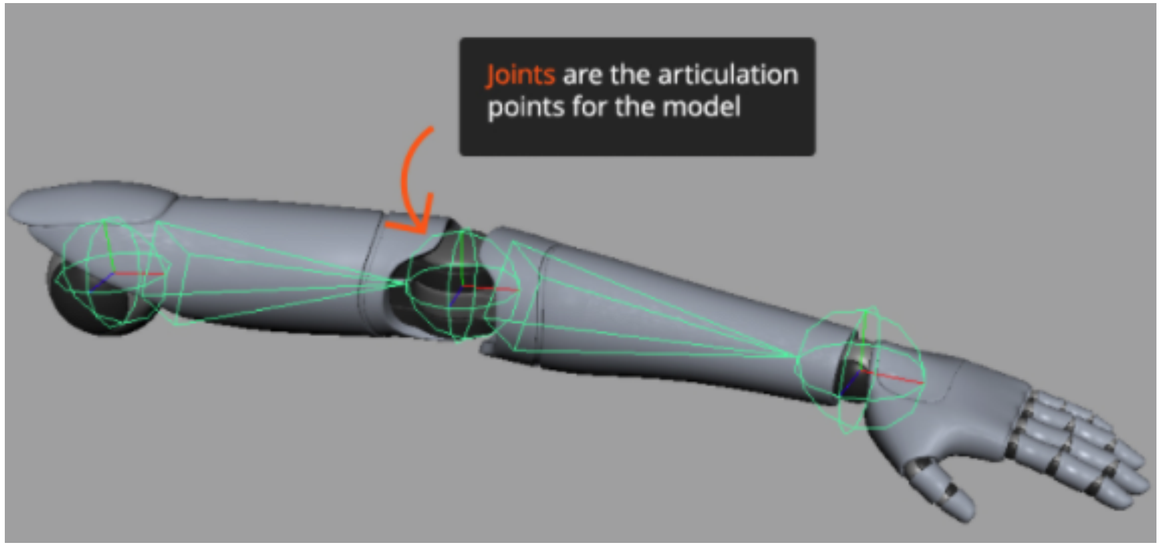
Cube

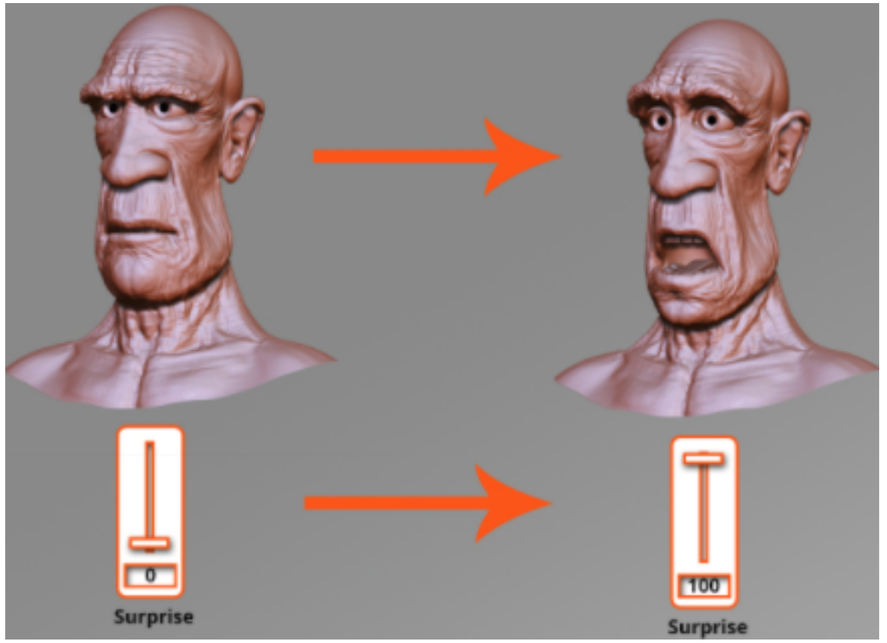


Cube man


Stretched out cube
moved up
The original cube
Squishy cube
moved to the right
Slanty cube
moved down and
to the left

Transformations in Rigging





Overview

In this box, you will find references to Eigen

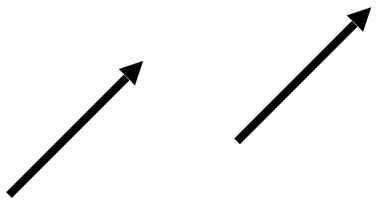
- We will briefly overview the basic linear algebra concepts that we will need in the class
- You will not be able to follow the next lectures without a clear understanding of this material

Vectors

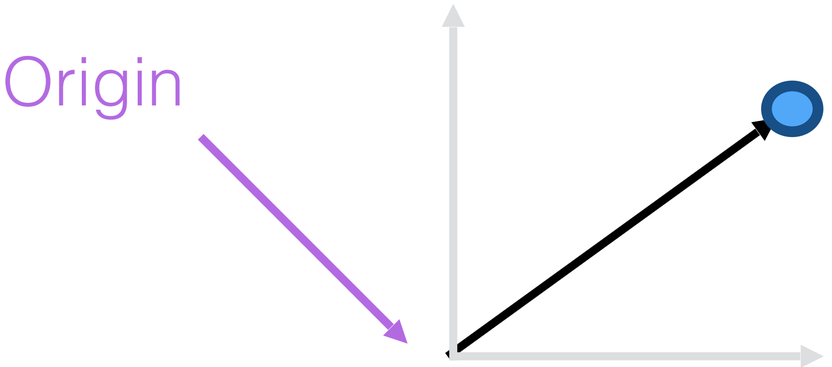
Vectors

Eigen::VectorXd

- A ***vector*** describes a direction and a length
- Do not confuse it with a location, which represent a position
- When you encode them in your program, they will both require 2 (or 3) numbers to be represented, but they are not the same object!



These two are identical!

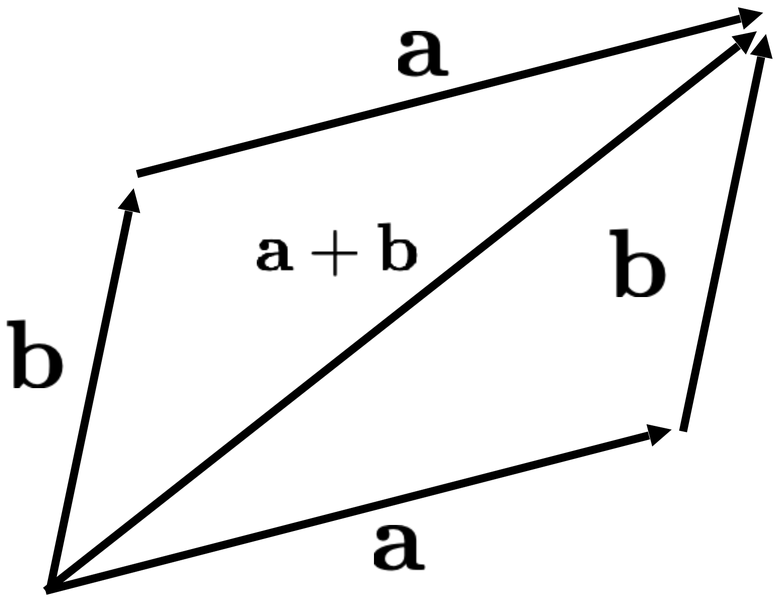


Vectors represent displacements. If you represent the displacement wrt the origin, then they ***encode*** a location.

Sum

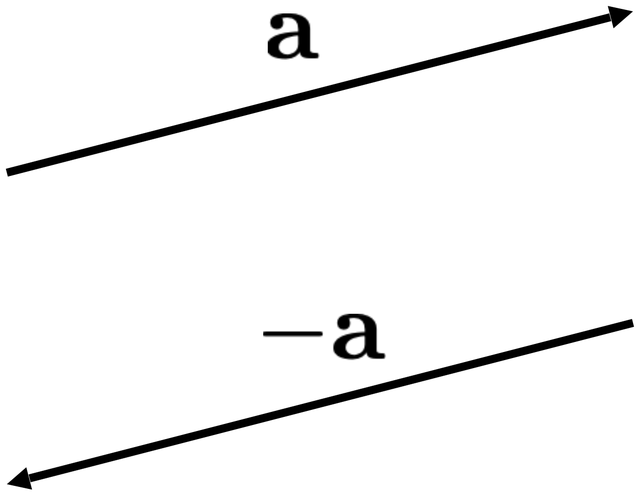
Operator +

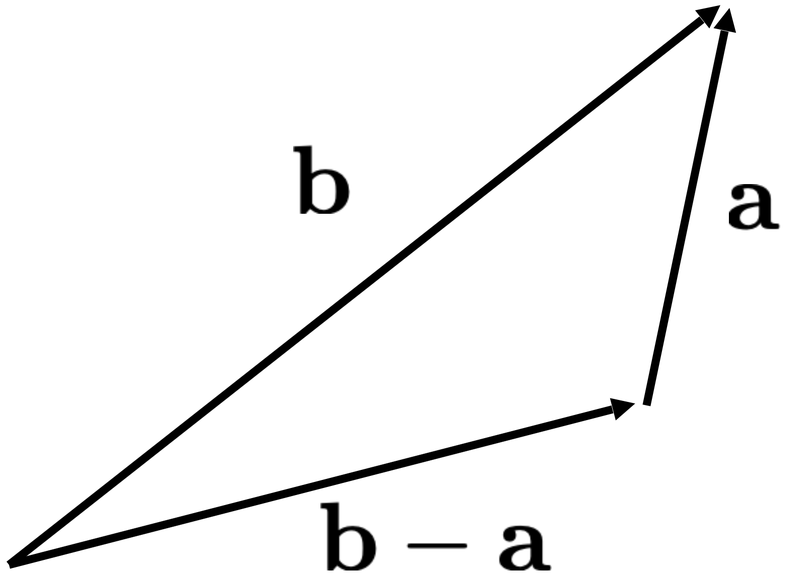
$$\mathbf{a} + \mathbf{b} = \mathbf{b} + \mathbf{a}$$



Difference

Operator -





$$\mathbf{b} - \mathbf{a} = -\mathbf{a} + \mathbf{b}$$

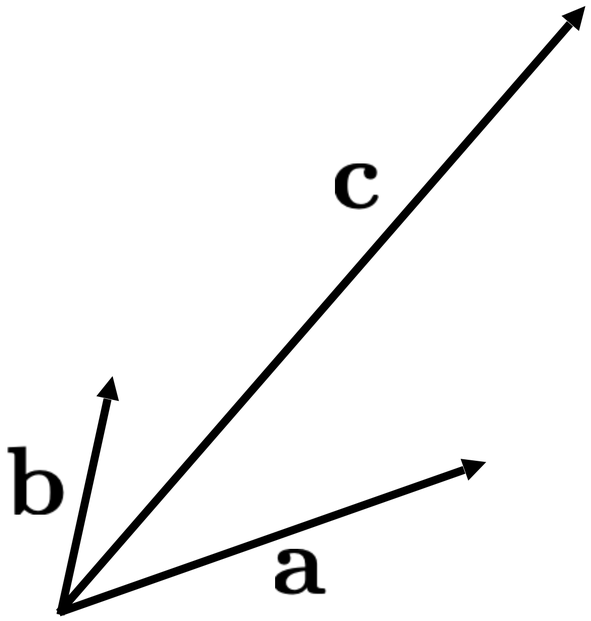
Coordinates of a Vector

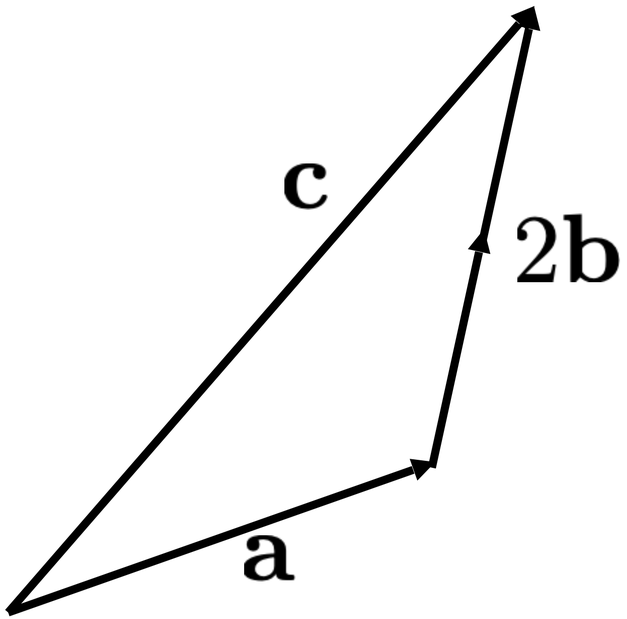
Operator []

$$\mathbf{c} = c_1\mathbf{a} + c_2\mathbf{b}$$

(c1, c2) is coordinate of c

$$\mathbf{c} = \mathbf{a} + 2\mathbf{b}$$

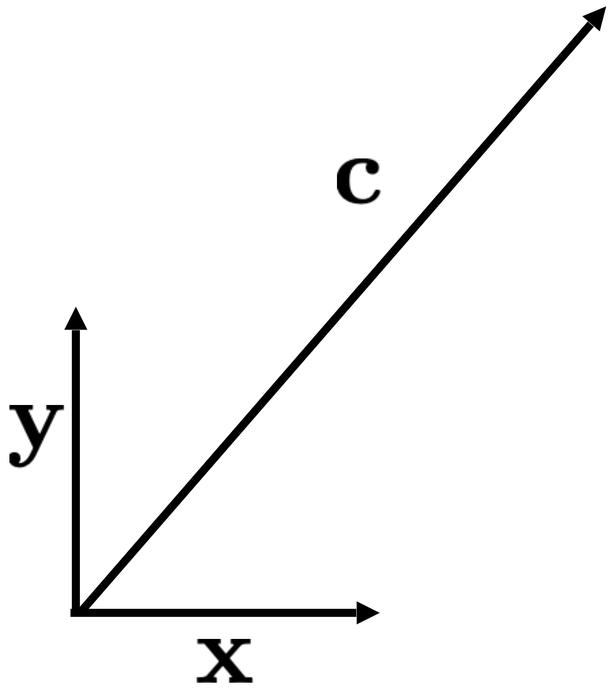




a and **b** form a 2D basis

Cartesian Coordinates of a Vector

$$\mathbf{c} = c_1\mathbf{x} + c_2\mathbf{y}$$



- **x** and **y** form a canonical, Cartesian basis

Length

- The length of a vector is denoted as ||**a**||

a.norm()

- If the vector is represented in cartesian coordinates, then it is the L2 norm of the vector:

$$||\mathbf{a}|| = \sqrt{a_1^2 + a_2^2}$$

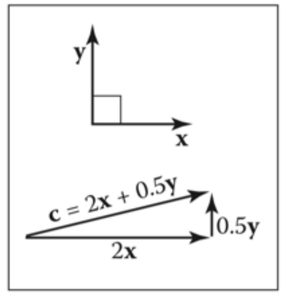


Figure 2.16. A 2D Cartesian basis for vectors.

- A vector can be normalized, to change its length to 1, without affecting the direction:

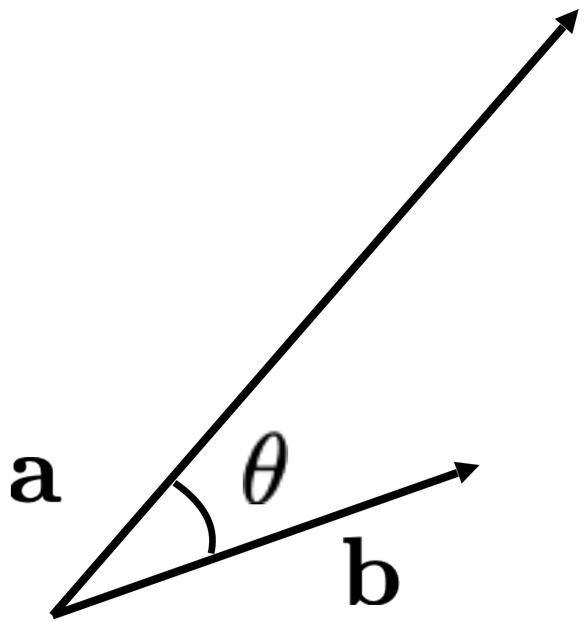
$$\mathbf{b} = \frac{\mathbf{a}}{||\mathbf{a}||}$$

CAREFUL:
b.normalize() <— in place
b.normalized() <— returns the normalized vector

Dot Product

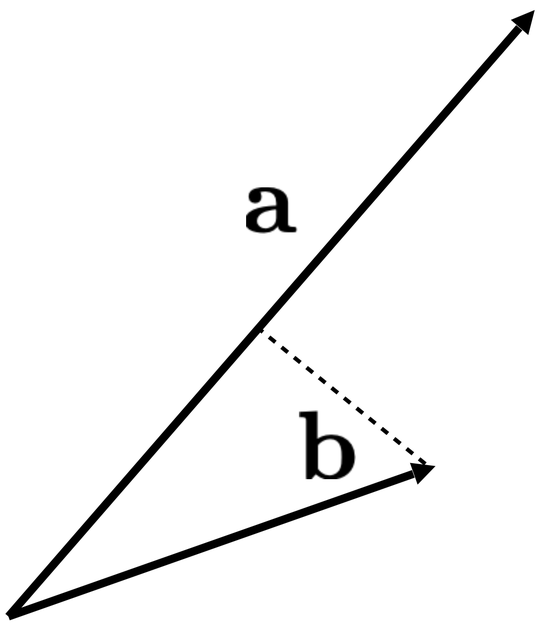
a.dot(b)
a.transpose()*b

$$\mathbf{a} \cdot \mathbf{b} = ||\mathbf{a}|| \; ||\mathbf{b}|| \cos\theta$$



- The dot product is related to the length of vector and of the angle between them
- If both are normalized, it is directly the cosine of the angle between them

Dot Product - Projection



- The length of the projection of **b** onto **a** can be computed using the dot product

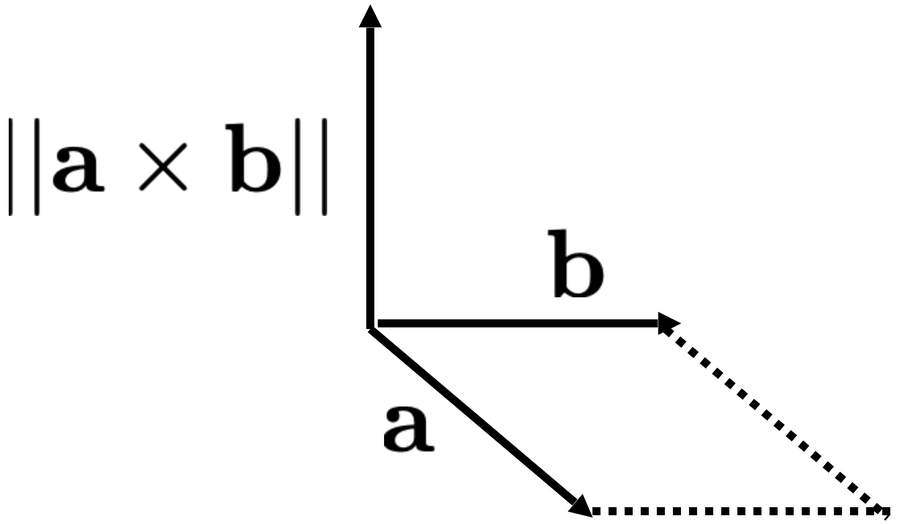
$$\mathbf{b} \rightarrow \mathbf{a} = ||\mathbf{b}|| \cos\theta = \frac{\mathbf{b} \cdot \mathbf{a}}{||\mathbf{a}||}$$

Cross Product

```
Eigen::Vector3d v(1, 2, 3);
Eigen::Vector3d w(4, 5, 6);
v.cross(w);
```

$$||\mathbf{a} \times \mathbf{b}|| = ||\mathbf{a}||\ ||\mathbf{b}|| \sin\theta$$

- Defined only for 3D vectors
- The resulting vector is perpendicular to both **a** and **b**, the direction depends on the *right hand rule*
- The magnitude is equal to the area of the parallelogram formed by **a** and **b**



Coordinate Systems

- You will often need to manipulate coordinate systems (i.e. for finding the position of the pixels in Assignment 1)
- You will always use ***orthonormal bases,*** which are formed by pairwise orthogonal unit vectors :

2D

$$||\mathbf{u}|| = ||\mathbf{v}|| = 1,$$
$$\mathbf{u} \cdot \mathbf{v} = 0$$

3D

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

Right-handed if: $\mathbf{w} = \mathbf{u} \times \mathbf{v}$

Global vs local coordinate sys

- Canonical/global/world
  - xyzo
  - never explicitly stored
- Frame of reference/local/object
  - uvwp
  - explicitly stored wrt global frame

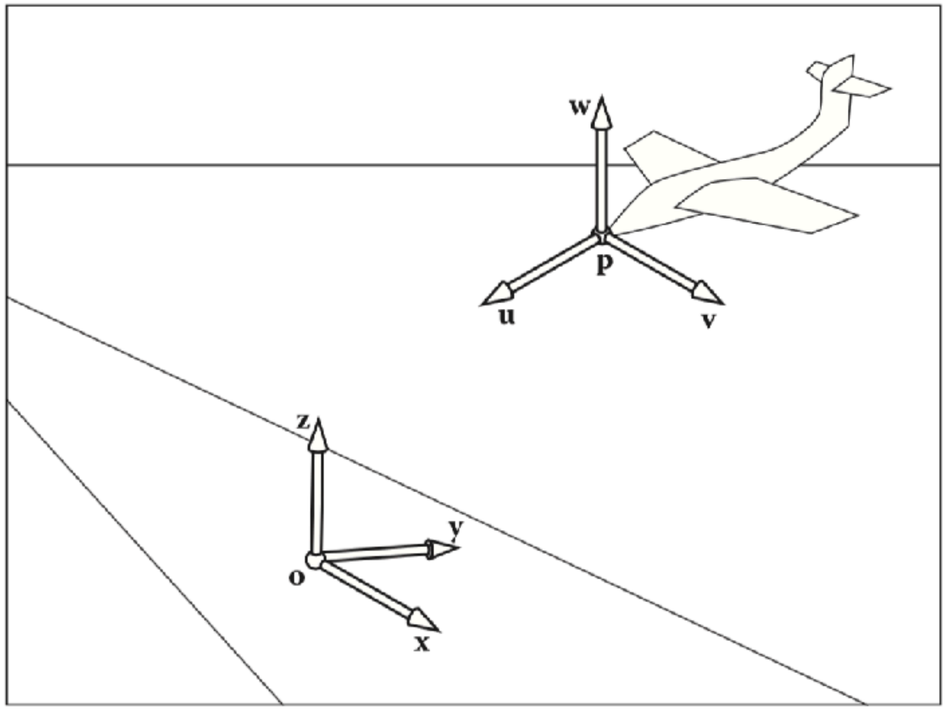
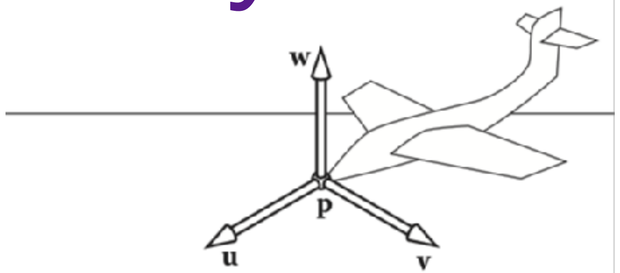


Figure 2.21. There is always a master or "canonical" coordinate system with origin **o** and orthonormal basis **x, y**, and **z**. This coordinate system is usually defined to be aligned to the global model and is thus often called the "global" or "world" coordinate system. This origin and basis vectors are never stored explicitly. All other vectors and locations are stored with coordinates that relate them to the global frame. The coordinate system associated with the plane are explicitly stored in terms of global coordinates.

Global vs local coordinate sys

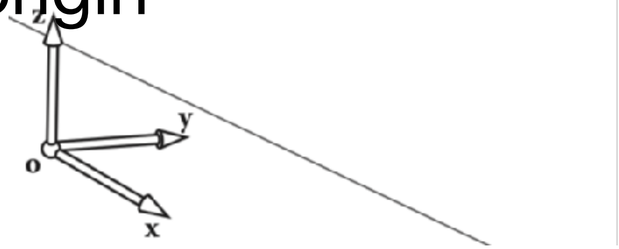


- local frame stored in canonical frame

$$\mathbf{u} = x_u\mathbf{x} + y_u\mathbf{y} + z_u\mathbf{z}.$$

- A location implicitly includes an offset from canonical origin

$$\mathbf{p} = \mathbf{o} + x_p\mathbf{x} + y_p\mathbf{y} + z_p\mathbf{z},$$



- store a vector a with respect to the u-v-w frame
  - The result is already in canonical frame

$$\mathbf{a} = u_a\mathbf{u} + v_a\mathbf{v} + w_a\mathbf{w}.$$

- To get the u-v-w coordinates of a vector b stored in the canonical coordinate

$$u_b = \mathbf{u} \cdot \mathbf{b}; \quad v_b = \mathbf{v} \cdot \mathbf{b}; \quad w_b = \mathbf{w} \cdot \mathbf{b}.$$

References

Fundamentals of Computer Graphics, Fourth Edition
4th Edition **by Steve Marschner, Peter Shirley**

Chapter 2

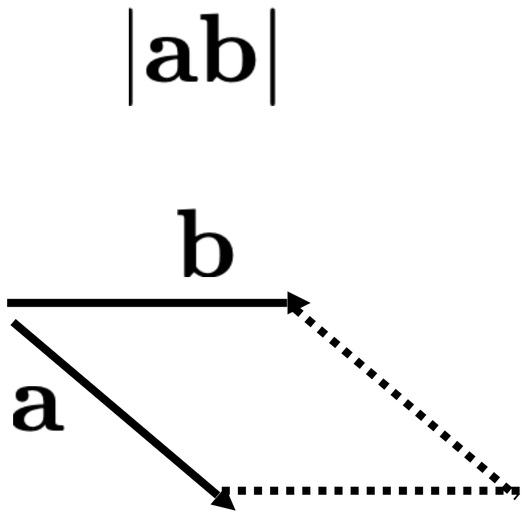
Matrices

Overview

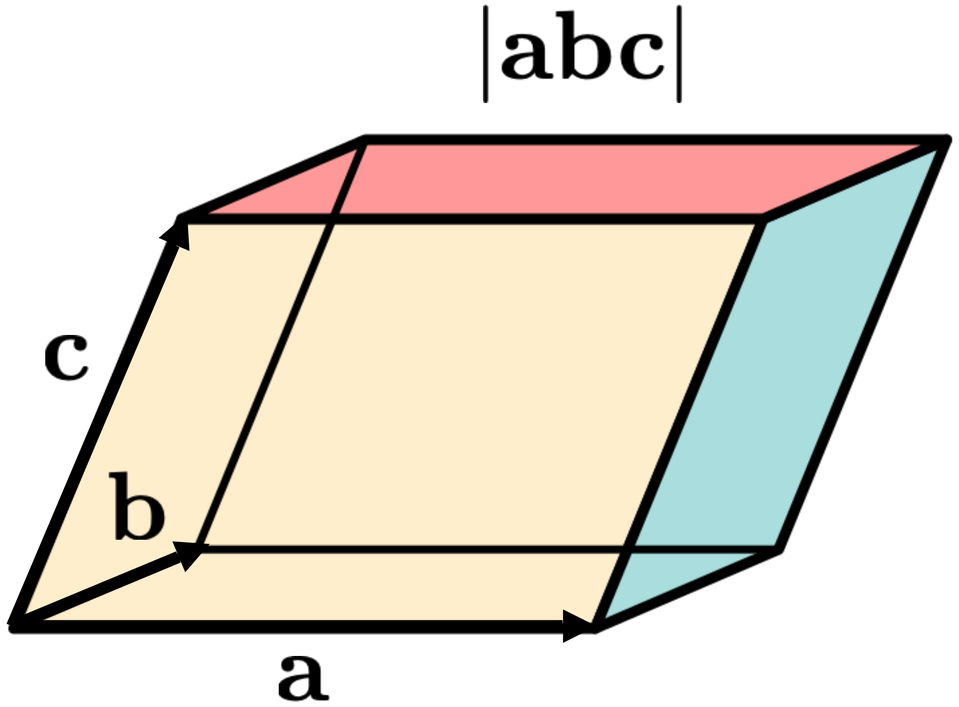
- Matrices will allow us to conveniently represent and ally transformations on vectors, such as translation, scaling and rotation
- Similarly to what we did for vectors, we will briefly overview their basic operations

Determinants

- Think of a determinant as an operation between vectors.



Area of the parallelogram



Volume of the parallelepiped
(positive since abc is a right-handed basis)

By Startswithj - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=29922624

Matrices

Eigen::MatrixXd A(2,2)

- A matrix is an array of numeric elements

$$\begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix}$$

Sum $$\begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} + \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix} = \begin{bmatrix} x_{11} + y_{11} & x_{12} + y_{12} \\ x_{21} + y_{21} & x_{22} + y_{22} \end{bmatrix}$$

A.array() + B.array()

Scalar Product $$y * \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} = \begin{bmatrix} yx_{11} & yx_{12} \\ yx_{21} & yx_{22} \end{bmatrix}$$

A.array() * y

Transpose

```
B = A.transpose();
A.transposeInPlace();
```

- The transpose of a matrix is a new matrix whose entries are reflected over the diagonal

$$\begin{bmatrix} 1 & 2 \end{bmatrix}^T = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

- The transpose of a product is the product of the transposed, in reverse order

$$(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$$

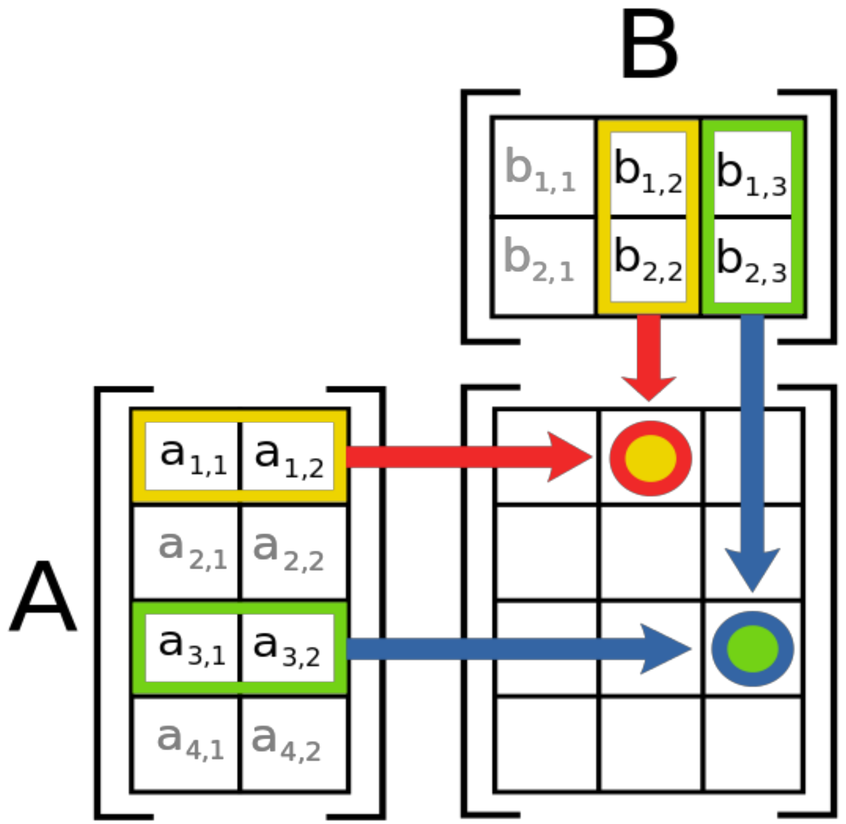
Matrix Product

- The entry i,j is given by multiplying the entries on the i-th row of A with the entries of the j-th column of B and summing up the results

- It is NOT commutative (in general):

$$\mathbf{AB} \neq \mathbf{BA}$$

```
Eigen::MatrixXd A(4,2);
Eigen::MatrixXd B(2,3);
A*B;
```



Intuition

$$\begin{bmatrix} | \\ \mathbf{y} \\ | \end{bmatrix} = \begin{bmatrix} -\mathbf{r_1}- \\ -\mathbf{r_2}- \\ -\mathbf{r_3}- \end{bmatrix} \begin{bmatrix} | \\ \mathbf{x} \\ | \end{bmatrix}$$

$$y_i = \mathbf{r_i} \cdot \mathbf{x}$$

Dot product on each row

$$\begin{bmatrix} | \\ \mathbf{y} \\ | \end{bmatrix} = \begin{bmatrix} | & | & | \\ \mathbf{c_1} & \mathbf{c_2} & \mathbf{c_3} \\ | & | & | \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$\mathbf{y} = x_1\mathbf{c_1} + x_2\mathbf{c_2} + x_3\mathbf{c_3}$$

Weighted sum of the columns

Inverse Matrix

Eigen::MatrixXd A(4,4);
A.inverse() <— do not use this to solve a linear system!

- The inverse of a matrix $\mathbf{A}$ is the matrix $\mathbf{A}^{-1}$ such that $\mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$

where **I** is the *identity matrix* $\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

- The inverse of a product is the product of the inverse in opposite order:

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}$$

Diagonal Matrices

```
Eigen::Vector3d v(1,2,3);

A = v.asDiagonal()
```

- They are zero everywhere except the diagonal:

$$\mathbf{D} = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix}$$

- Useful properties:

$$\mathbf{D}^{-1} = \begin{bmatrix} a^{-}1 & 0 & 0 \\ 0 & b^{-}1 & 0 \\ 0 & 0 & c^{-1} \end{bmatrix}$$

$$\mathbf{D} = \mathbf{D}^T$$

Orthogonal Matrices

- An orthogonal matrix is a matrix where
  - each column is a vector of length 1
  - each column is orthogonal to all the others
- A useful property of orthogonal matrices that their inverse corresponds to their transpose:

$$(\mathbf{R}^T\mathbf{R}) = \mathbf{I} = (\mathbf{R}\mathbf{R}^T)$$

Linear Systems

- We will often encounter in this class linear systems with ***n*** linear equations that depend on ***n*** variables.
- For example:

$$\begin{aligned} 5x + 3y - 7z &= 4 \\ -3x + 5y + 12z &= 9 \\ 9x - 2y - 2z &= -3 \end{aligned}$$

$$\begin{bmatrix} 5 & 3 & -7 \\ -3 & 5 & 12 \\ 9 & -2 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 9 \\ -3 \end{bmatrix}$$

- To find ***x,y,z*** you have to "solve" the linear system. Do not use an inverse, but rely on a direct solver:

```
Matrix3f A;
Vector3f b;
A << 5,3,-7,  -3,5,12,  9,-2,-2;
b << 4, 9, -3;
cout << "Here is the matrix A:\n" << A << endl;
cout << "Here is the vector b:\n" << b << endl;
Vector3f x = A.colPivHouseholderQr().solve(b);
cout << "The solution is:\n" << x << endl;
```

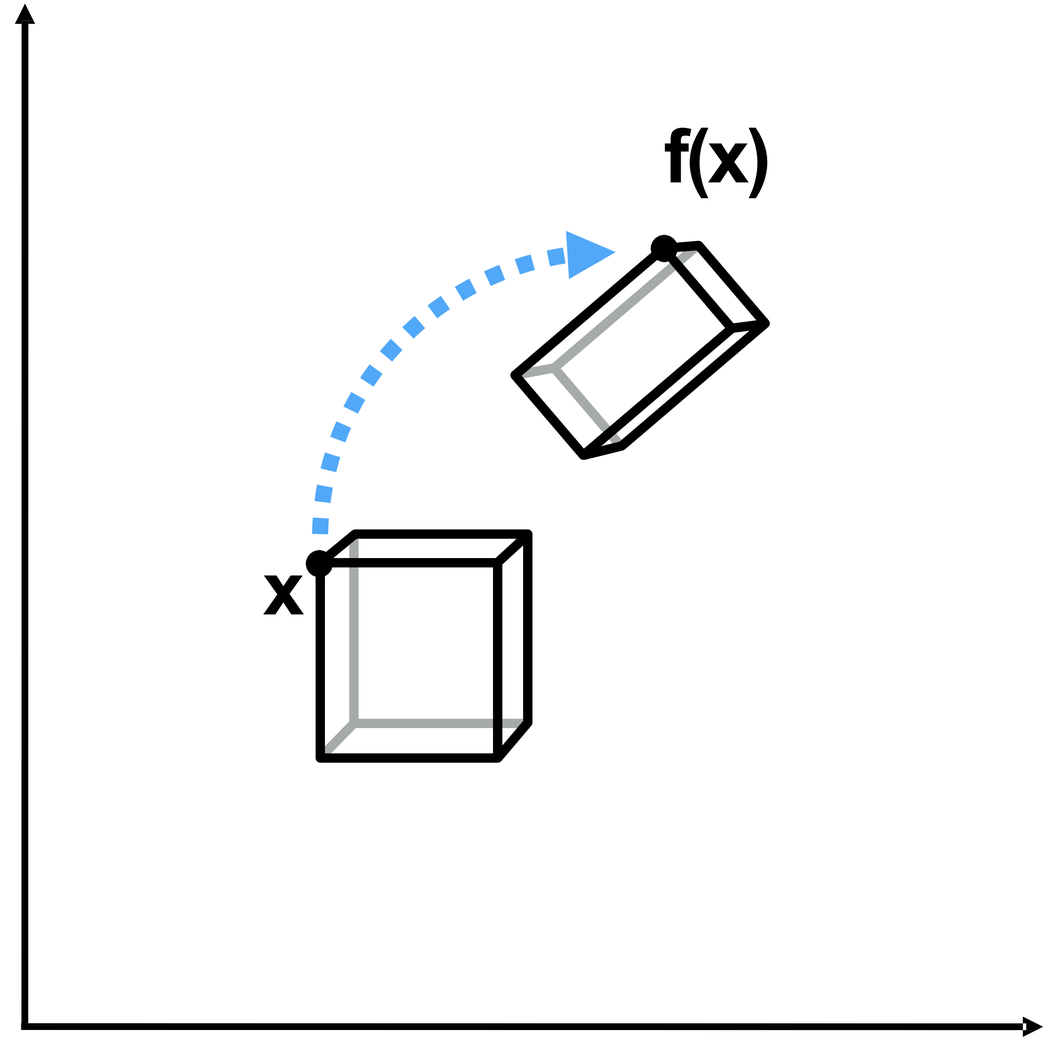
References

Fundamentals of Computer Graphics, Fourth Edition
4th Edition **by Steve Marschner, Peter Shirley**

Chapter 5

Linear Transformations

Basic idea: f transforms x to f(x)

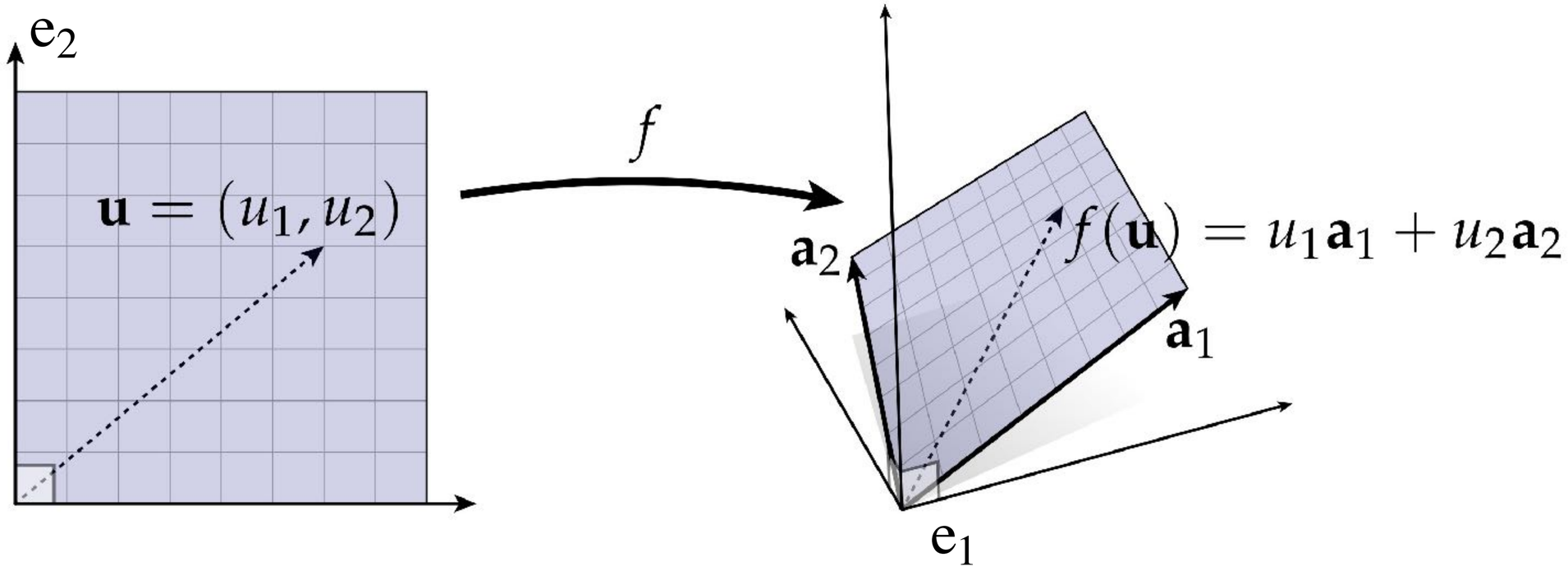


And what is our favorite type of transformation?

What can we do with linear transformations?

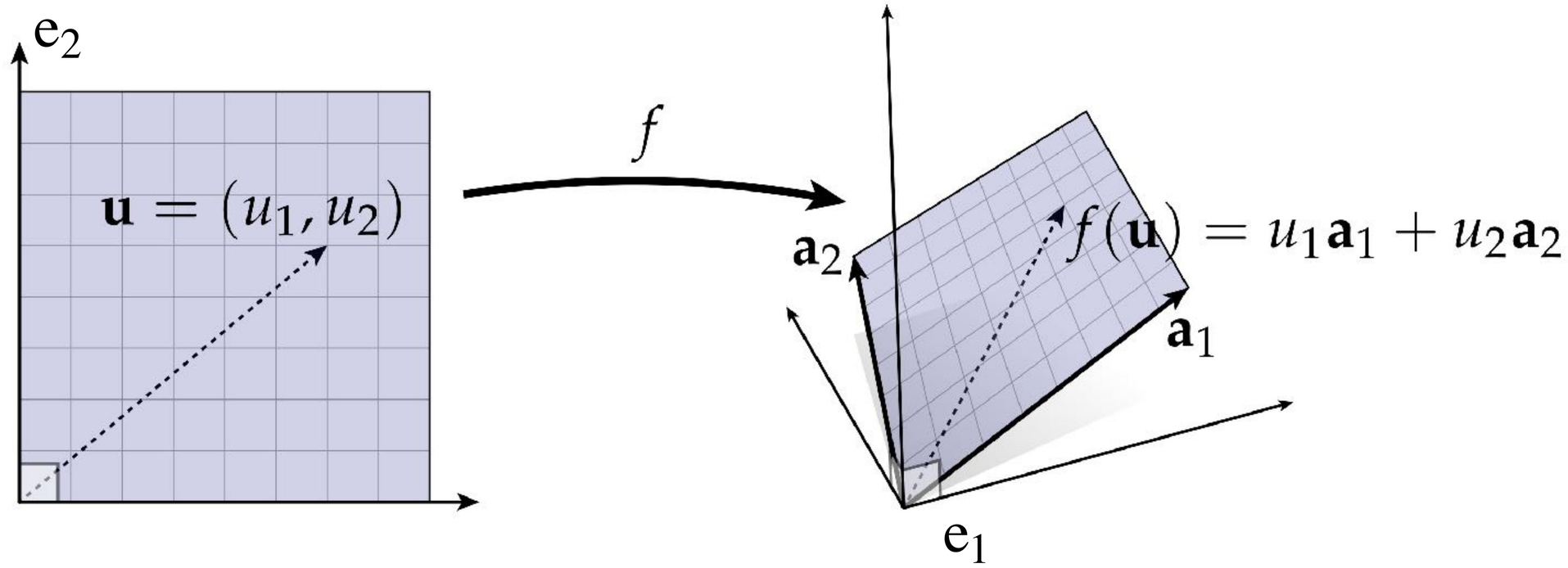
- What did linear mean?
  - $f(u + v) = f(u) + f(v)$
  - $f(au) = af(u)$
- Cheap to compute
- Composition of linear transformations is linear
  - Leads to uniform representation of transformations
  - E.g., in graphics card (GPU) or graphics APIs

Linear transforms



- **Do you know…**
  - **what u_1 and u_2 are?**
  - **what a_1 and a_2 are?**

Linear transforms



- ***u*** **is a linear combination of** e_1 **and** e_2
- $f(u)$ **is that same linear combination of** a_1 **and** a_2
  - a_1 **and** a_2 **are** $f(e_1)$ **and** $f(e_2)$
- **by knowing what** e_1 **and** e_2 **map to, you know how to map the entire space!**

2D Linear Transformations

- Each 2D linear map can be represented by a unique 2×2 matrix

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

- Concatenation of mappings corresponds to multiplication of matrices

$$L_2(L_1(\mathbf{x})) = \mathbf{L}_2 \mathbf{L}_1 \mathbf{x}$$

```
L2 * L1 * x;
```

- Linear transformations are very common in computer graphics!

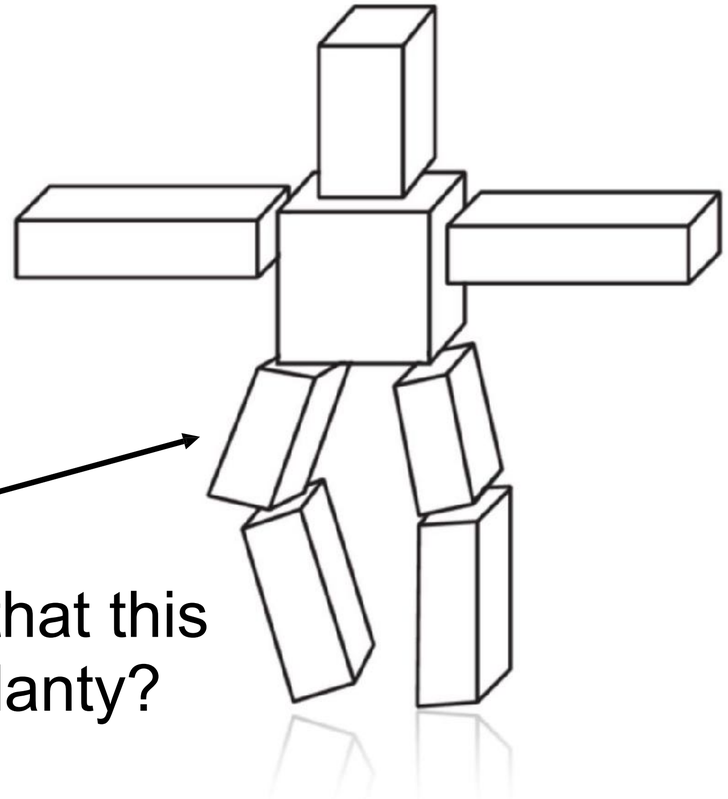
Linear transforms

If a map can be expressed as

$$\mathbf{f(u)} = \sigma_{i=1}^{m} u_i a_i$$

with fixed vectors a_i, then it is linear

Let’s look at some transforms that are important in graphics…



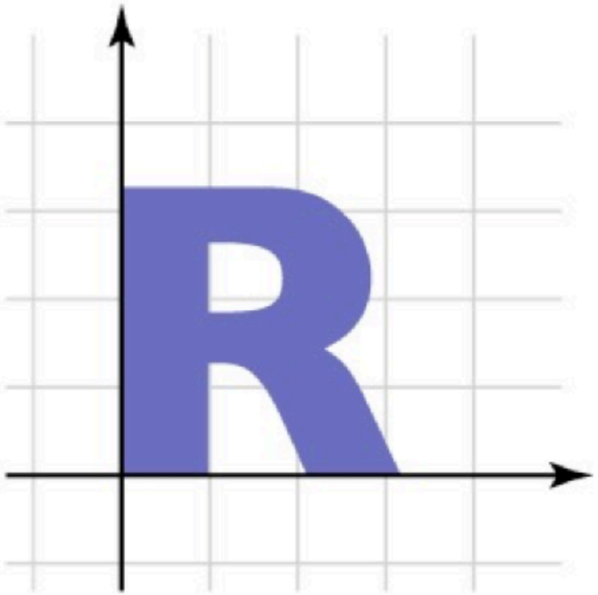
How do you formally tell a computer that this cube should be squished and slanty?

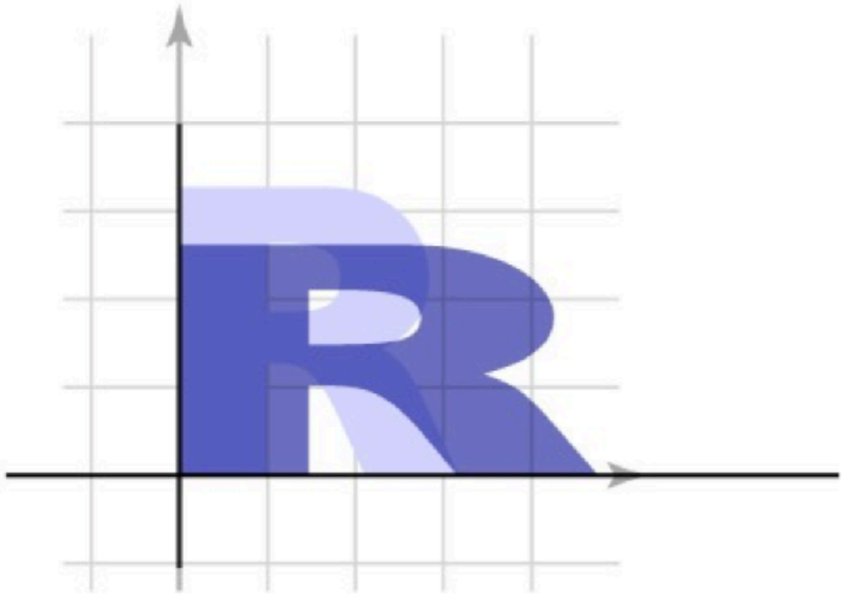
Linear transformation gallery

- **Nonuniform scale**

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix}$$

$$\begin{bmatrix} 1.5 & 0 \\ 0 & 0.8 \end{bmatrix}$$



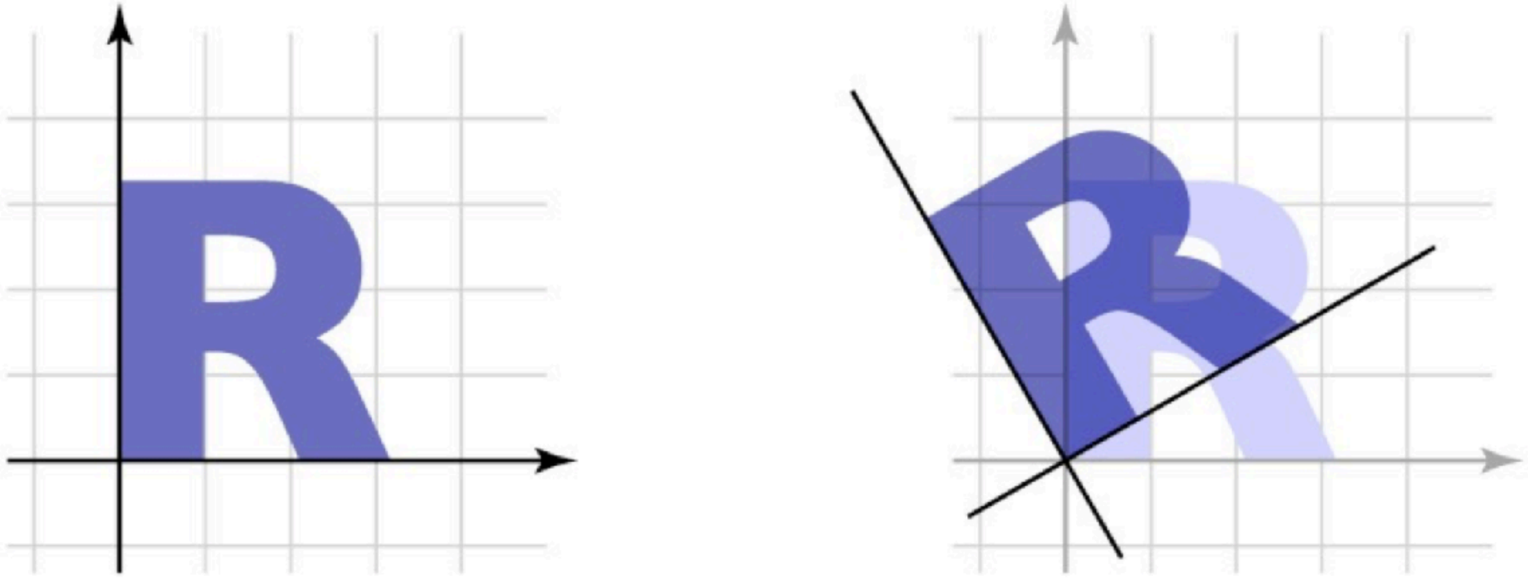


Linear transformation gallery

- **Rotation**

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}$$

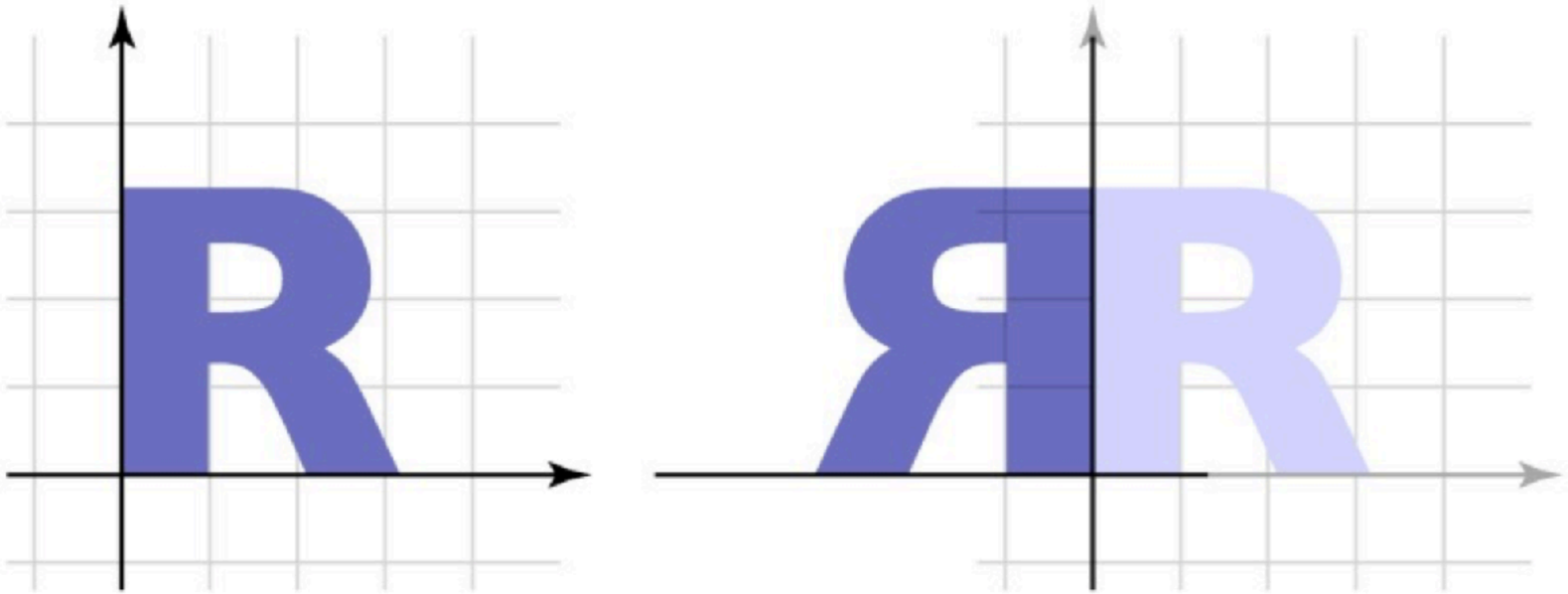
$$\begin{bmatrix} 0.866 & -0.5 \\ 0.5 & 0.866 \end{bmatrix}$$



Linear transformation gallery

- **Reflection**
  - can consider it a special case of nonuniform scale

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$

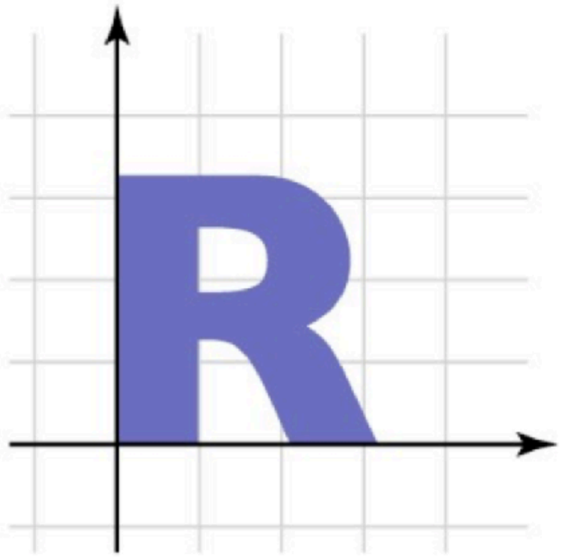


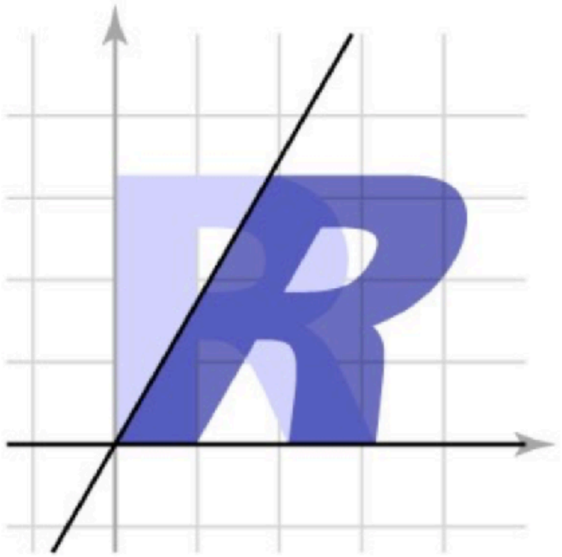
Linear transformation gallery

- **Shear**
  - can also build these from rotations and nonuniform scales

$$\begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x + ay \\ y \end{bmatrix}$$

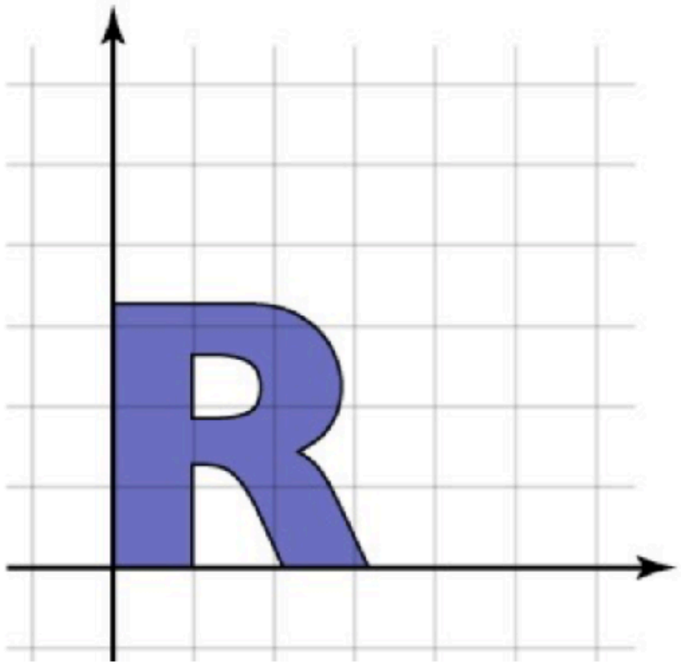
$$\begin{bmatrix} 1 & 0.5 \\ 0 & 1 \end{bmatrix}$$

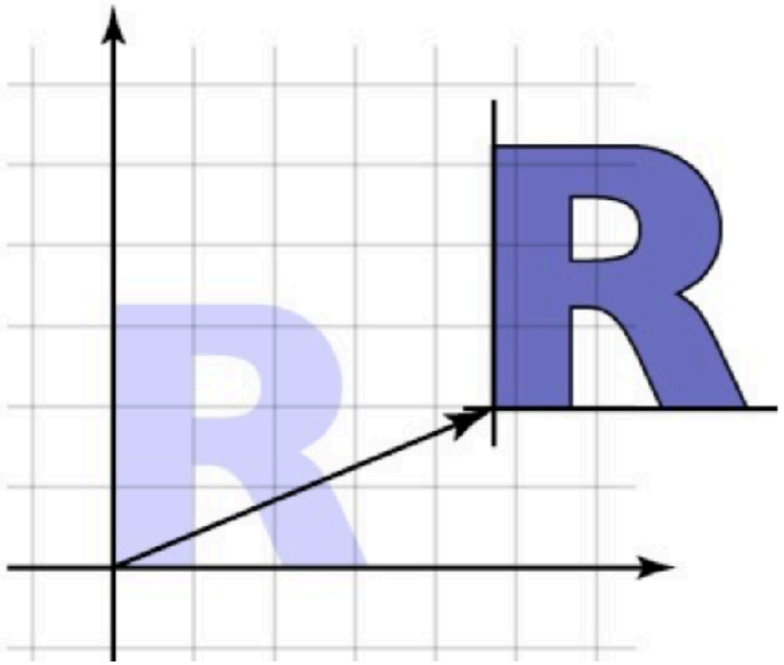




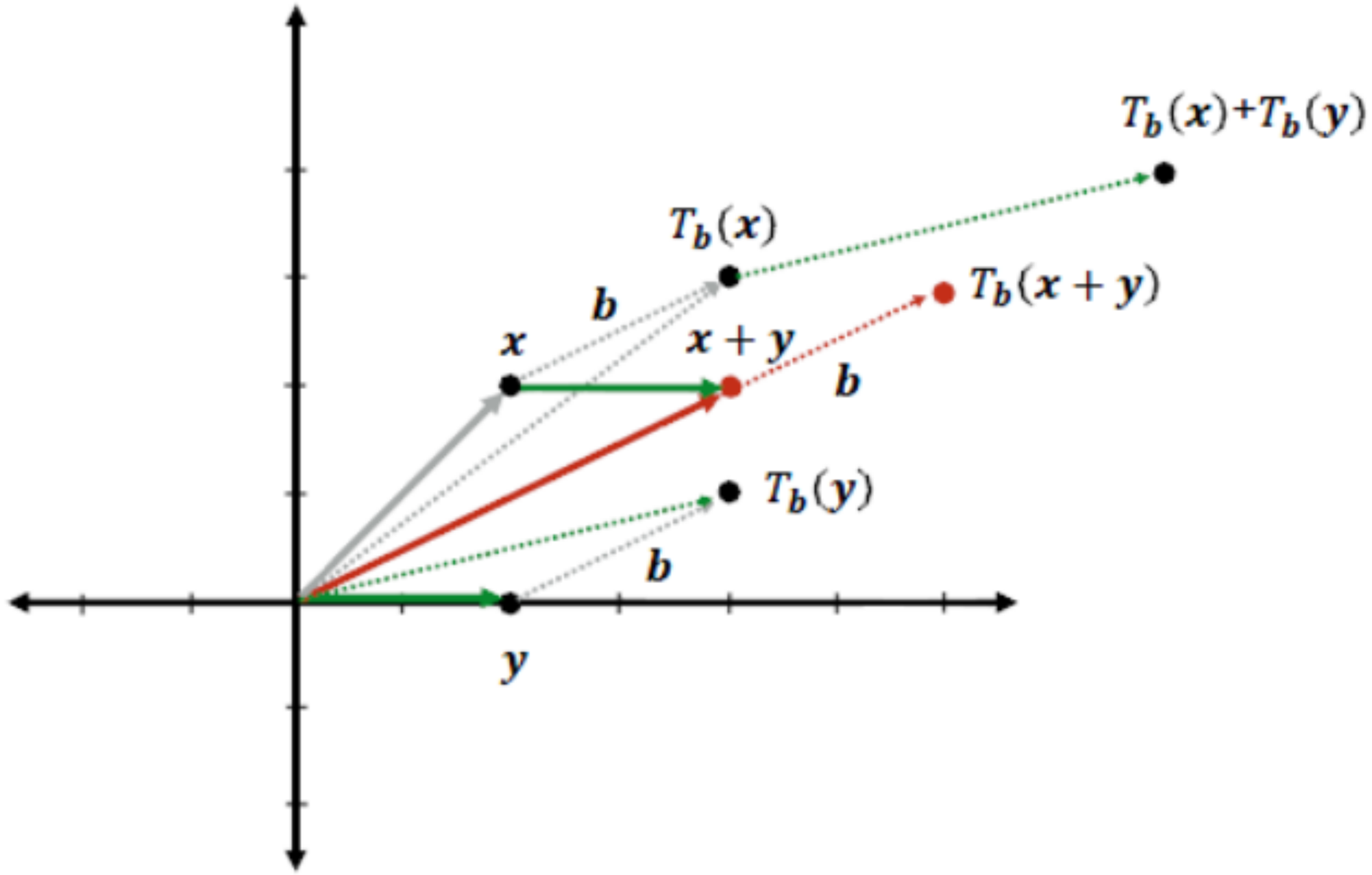
Translation

- **Simplest transformation:** $T(\mathbf{v}) = \mathbf{v} + \mathbf{u}$
- **Inverse:** $T^{-1}(\mathbf{v}) = \mathbf{v} - \mathbf{u}$
- **Example of transforming circle**





Is translation linear?



No. Translation is affine.

Composing transformations

- **Want to move an object, then move it some more**
  - $\mathbf{p} \rightarrow T(\mathbf{p}) \rightarrow S(T(\mathbf{p})) = (S \circ T)(\mathbf{p})$
- **We need to represent S o T ("S compose T")**
  - and would like to use the same representation as for S and T
- **Translation easy**
  - $T(\mathbf{p}) = \mathbf{p} + \mathbf{u}_T; S(\mathbf{p}) = \mathbf{p} + \mathbf{u}_S$

    $(S \circ T)(\mathbf{p}) = \mathbf{p} + (\mathbf{u}_T + \mathbf{u}_S)$
- **Translation by uT then by uS is translation by uT + uS**
  - commutative!

Composing transformations

- **Linear transformations also straightforward**
  - $$T(\mathbf{p}) = M_T\mathbf{p}; S(\mathbf{p}) = M_S\mathbf{p}$$
    $$(S \circ T)(\mathbf{p}) = M_S M_T \mathbf{p}$$

- **Transforming first by M_T then by M_S is the same as transforming by $M_S M_T$**
  - only sometimes commutative
    - e.g. rotations & uniform scales
    - e.g. non-uniform scales w/o rotation
  - Note $M_S M_T$, or $S \circ T$, is T first, then S

Combining linear with translation

- **Need to use both in single framework**
- **Can represent arbitrary seq. as** $T(\mathbf{p}) = M\mathbf{p} + \mathbf{u}$
  - $T(\mathbf{p}) = M_T\mathbf{p} + \mathbf{u}_T$
  - $S(\mathbf{p}) = M_S\mathbf{p} + \mathbf{u}_S$
  - $(S \circ T)(\mathbf{p}) = M_S(M_T\mathbf{p} + \mathbf{u}_T) + \mathbf{u}_S$

    $= (M_S M_T)\mathbf{p} + (M_S\mathbf{u}_T + \mathbf{u}_S)$
  - e.g. $S(T(0)) = S(\mathbf{u}_T)$
- **Transforming by M_T and $\mathbf{u}_T$, then by M_S and $\mathbf{u}_S$, is the same as transforming by $M_S M_T$ and $\mathbf{u}_S + M_S\mathbf{u}_T$**
  - This will work but is a little awkward

Homogeneous coordinates

- **A trick for representing the foregoing more elegantly**
- **Extra component *w* for vectors, extra row/column for matrices**
  - for affine, can always keep $w = 1$
- **Represent linear transformations with dummy extra row and column**

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \\ 1 \end{bmatrix}$$

Homogeneous coordinates

- **Represent translation using the extra column**

$$\begin{bmatrix} 1 & 0 & t \\ 0 & 1 & s \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x+t \\ y+s \\ 1 \end{bmatrix}$$

Homogeneous coordinates

- **Composition just works, by 3x3 matrix multiplication**

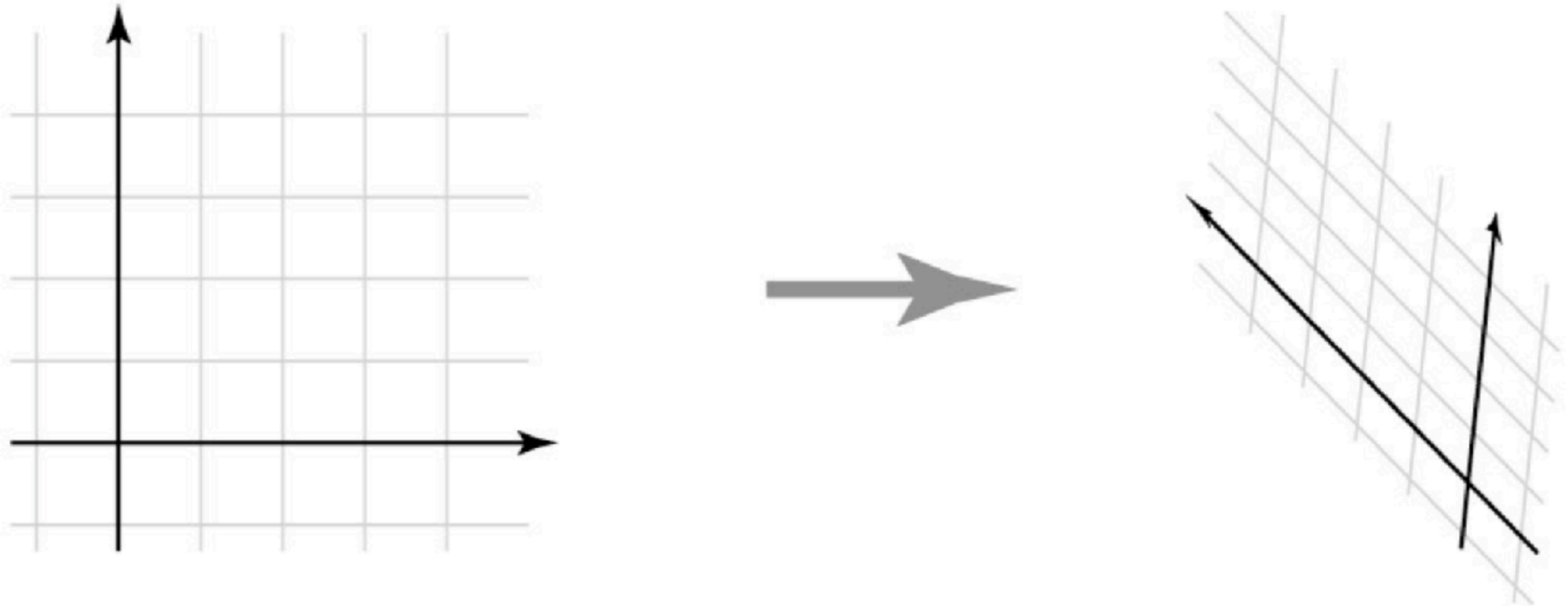
$$\begin{bmatrix} M_S & \mathbf{u}_S \\ 0 & 1 \end{bmatrix} \begin{bmatrix} M_T & \mathbf{u}_T \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} (M_S M_T)\mathbf{p} + (M_S \mathbf{u}_T + \mathbf{u}_S) \\ 1 \end{bmatrix}$$

- **This is exactly the same as carrying around *M* and u**
  - but cleaner
  - and generalizes in useful ways as we'll see later

Affine transformations

- **The set of transformations we have been looking at is known as the "affine" transformations**
  - straight lines preserved; parallel lines preserved
  - ratios of lengths along lines preserved (midpoints preserved)



3D Affine transformation gallery

Represent 3D transforms as 3x3 matrices and 3D-H transforms as 4x4 matrices

Scale:

3D

$$\mathbf{S_s} = \begin{bmatrix} \mathbf{S}_x & 0 & 0 \\ 0 & \mathbf{S}_y & 0 \\ 0 & 0 & \mathbf{S}_z \end{bmatrix}$$

3D-H

$$\mathbf{S_s} = \begin{bmatrix} \mathbf{S}_x & 0 & 0 & 0 \\ 0 & \mathbf{S}_y & 0 & 0 \\ 0 & 0 & \mathbf{S}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Shear (in x, based on y,z position):

$$\mathbf{H}_{x,\mathbf{d}} = \begin{bmatrix} 1 & \mathbf{d}_y & \mathbf{d}_z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \mathbf{H}_{x,\mathbf{d}} = \begin{bmatrix} 1 & \mathbf{d}_y & \mathbf{d}_z & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Reflection

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Translate:

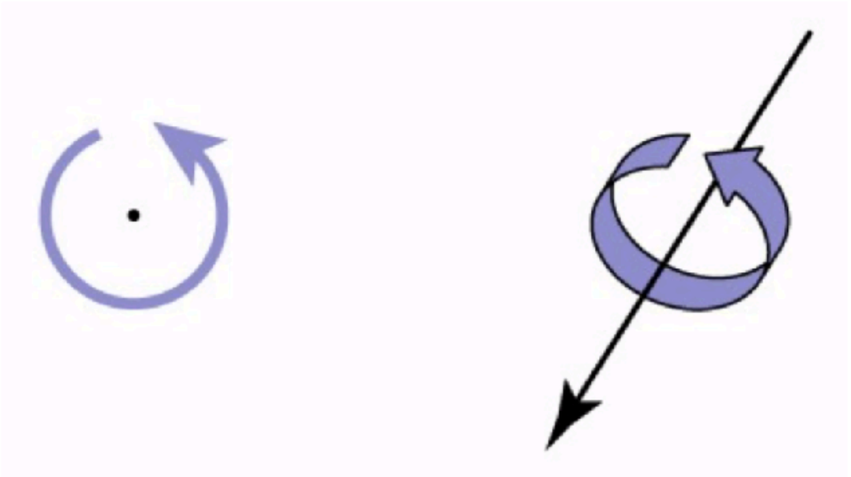
3D-H

$$\mathbf{T_b} = \begin{bmatrix} 1 & 0 & 0 & \mathbf{b}_x \\ 0 & 1 & 0 & \mathbf{b}_y \\ 0 & 0 & 1 & \mathbf{b}_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

General Rotation Matrices

- **A rotation in 2D is around a point**
- **A rotation in 3D is around an axis**
  - so 3D rotation is w.r.t a line, not just a point
  - there are many more 3D rotations than 2D
    - a 3D space around a given point, not just 1D

convention: positive rotation is CCW



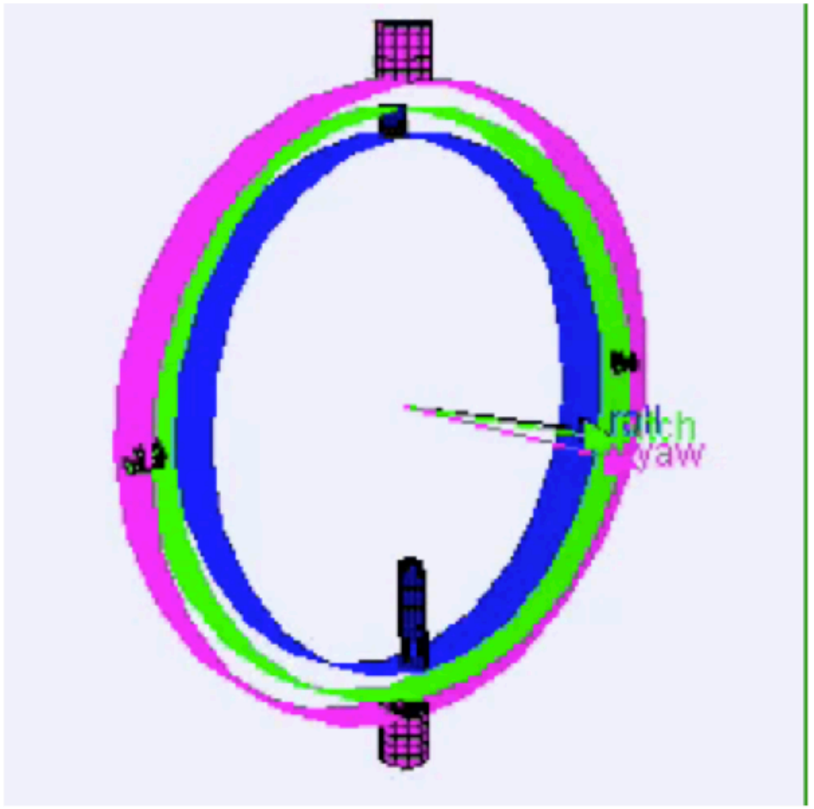
2D

3D

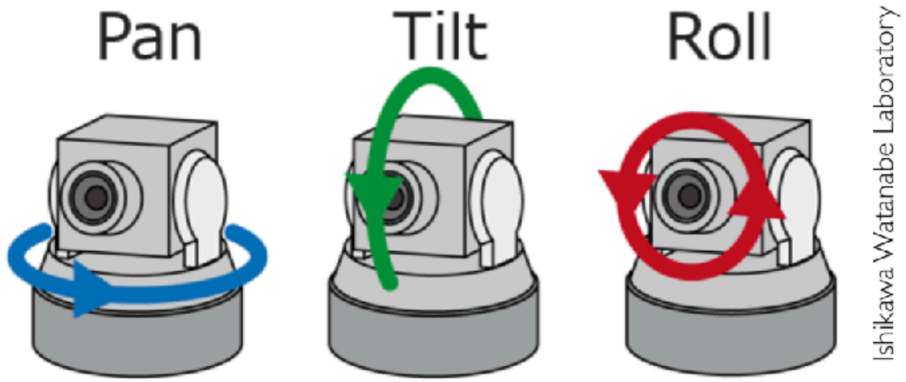
convention: positive rotation is CCW when axis vector is pointing at you

Euler angles

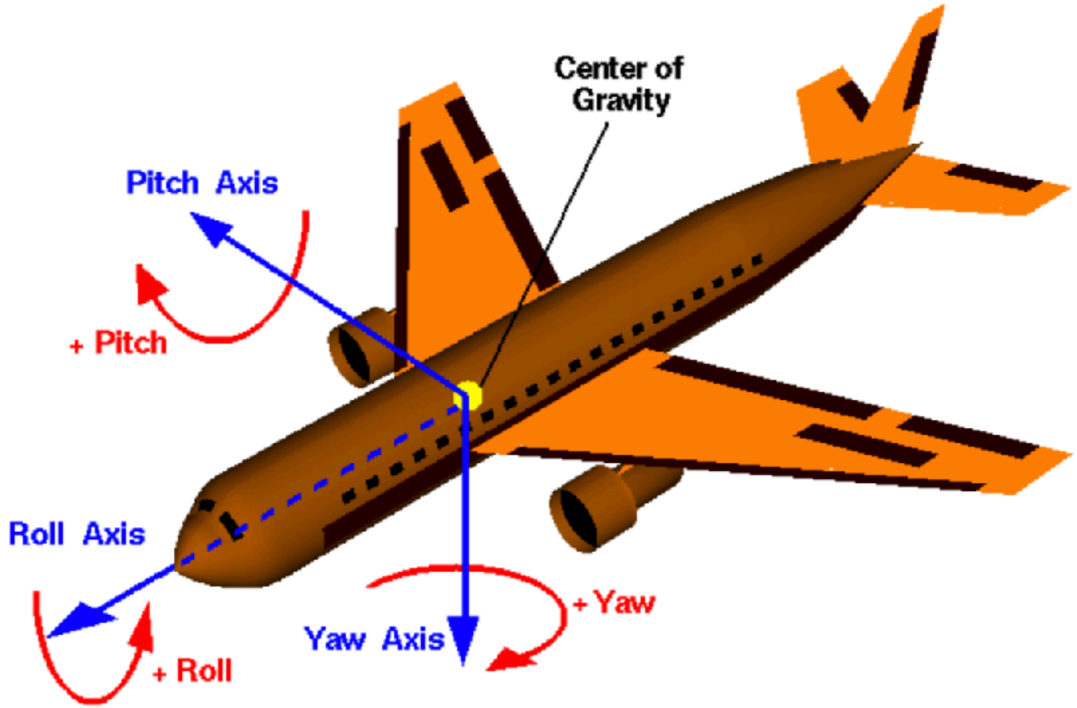
- **An object can be oriented arbitrarily**
- **Euler angles: simply compose three coord. axis rotations**
  - e.g. x, then y, then z: $R(\theta_x, \theta_y, \theta_z) = R_z(\theta_z)R_y(\theta_y)R_x(\theta_x)$
  - "heading, attitude, bank" (common for airplanes)
  - "roll, pitch, yaw" (common for vehicles)
  - "pan, tilt, roll" (common for cameras)



Euler angles in applications



Ishikawa Watanabe Laboratory

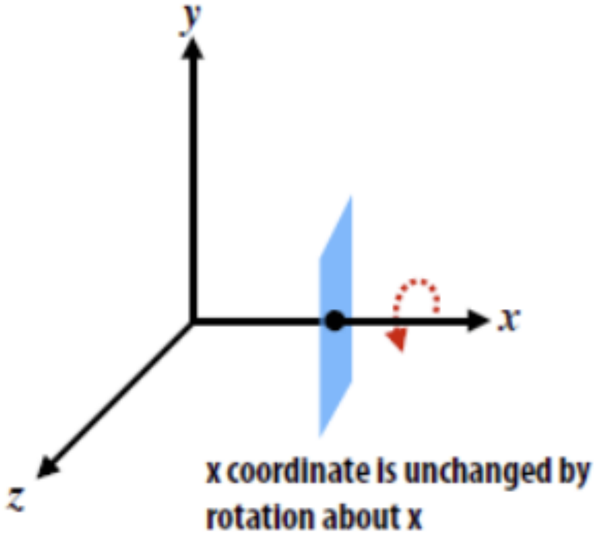


Wikipedia

Representing Rotations in 3D: Euler Angles

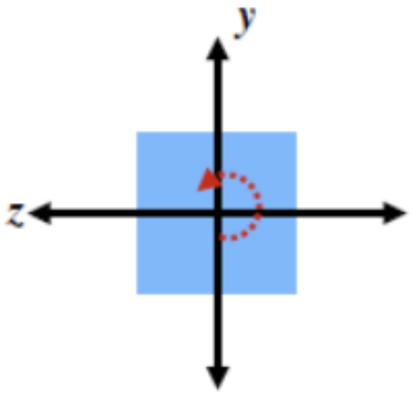
Rotation about x axis:

$$\mathbf{R}_{x,\theta} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$



x coordinate is unchanged by rotation about x

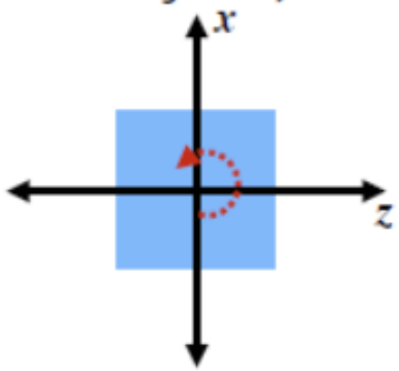
View looking down -x axis:



Rotation about y axis:

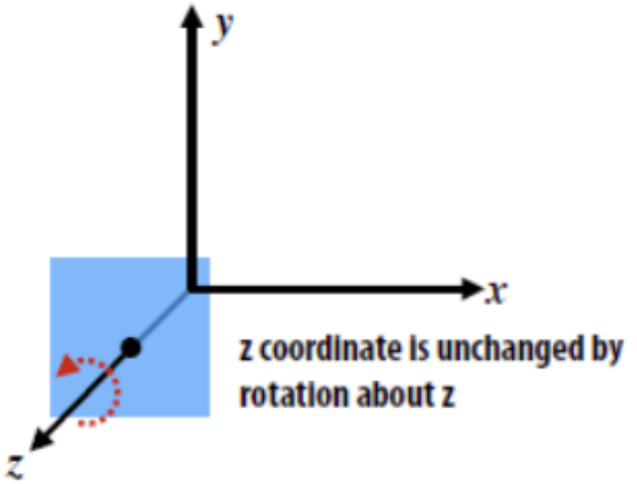
$$\mathbf{R}_{y,\theta} = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

View looking down -y axis:



Rotation about z axis:

$$\mathbf{R}_{z,\theta} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



z coordinate is unchanged by rotation about z

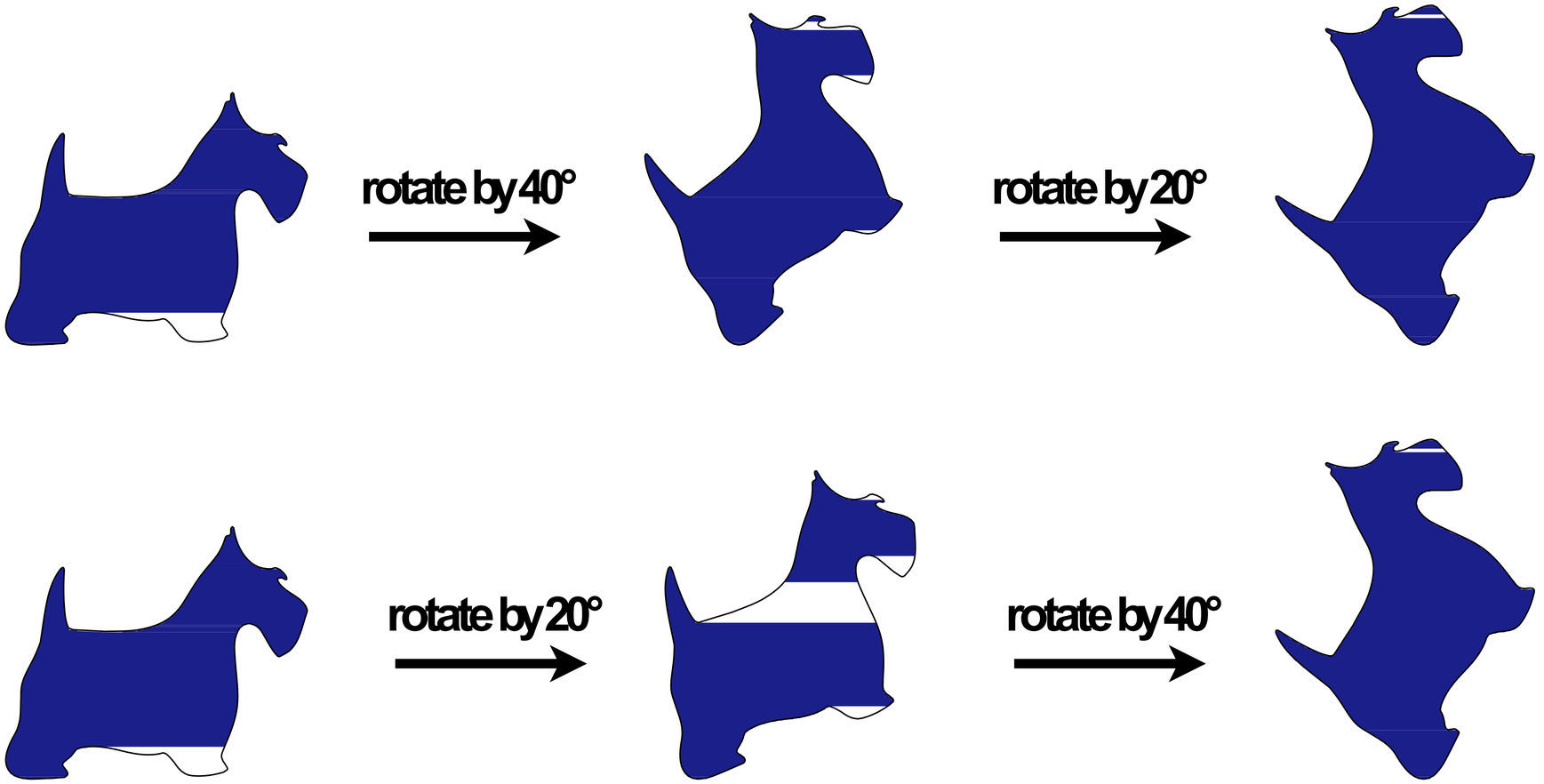
General affine transformations

- The previous slides showed “canonical” examples of the types of affine transformations
- Generally, transformations contain elements of multiple types
  - often define them as products of canonical transforms
  - sometimes work with their properties more directly

Composite affine transformations

Commutativity of Rotations—2D

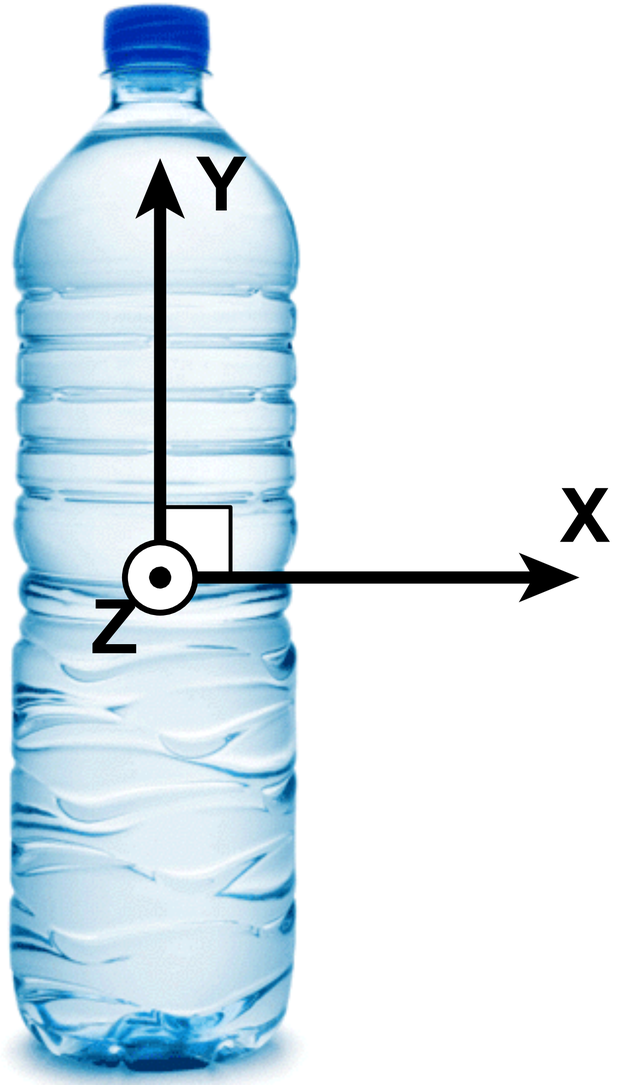
- In 2D, order of rotations doesn't matter:



Same result! 2D rotations commute

Commutativity of Rotations—3D

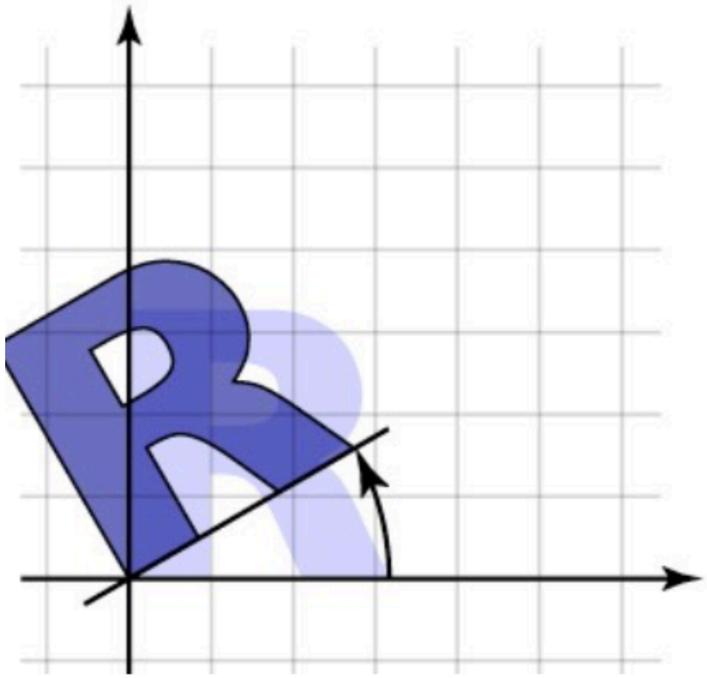
- What about in 3D?
- IN-CLASS ACTIVITY:
  - Rotate 90°around Y, then 90°around Z, then 90°around X
  - Rotate 90° around Z, then 90° around Y, then 90° around X
  - (Was there any difference?)



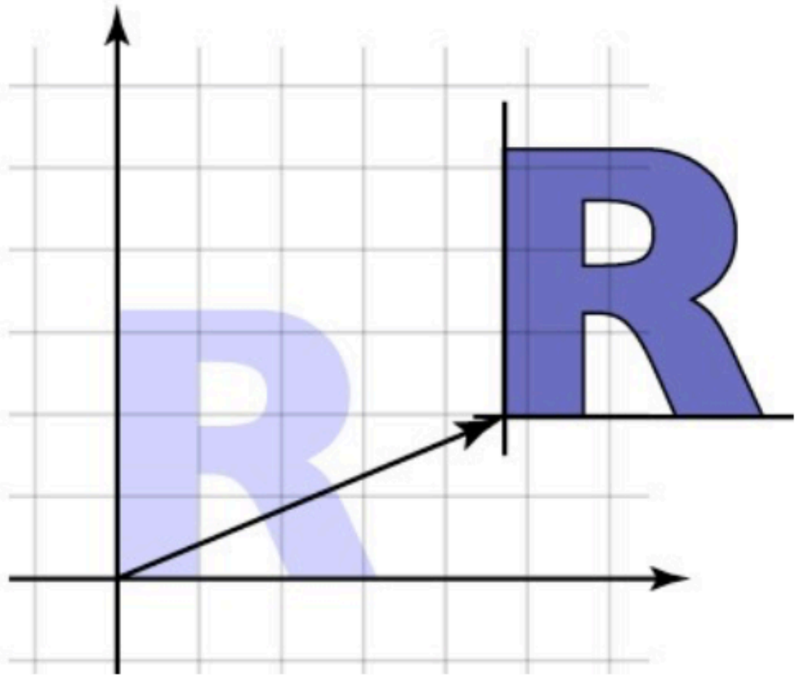
CONCLUSION: bad things can happen if we're not careful about the order in which we apply rotations!

Composite affine transformations

- In general not commutative: order matters!



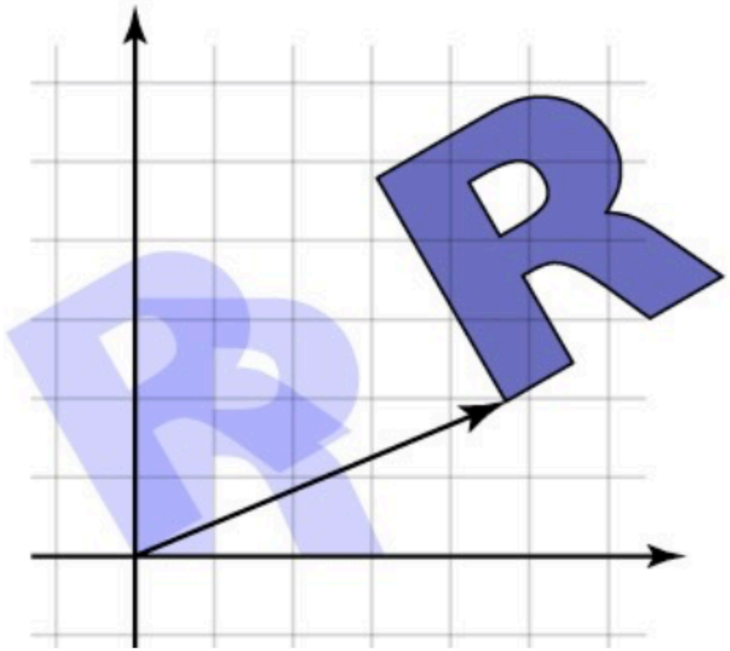
rotate, then translate



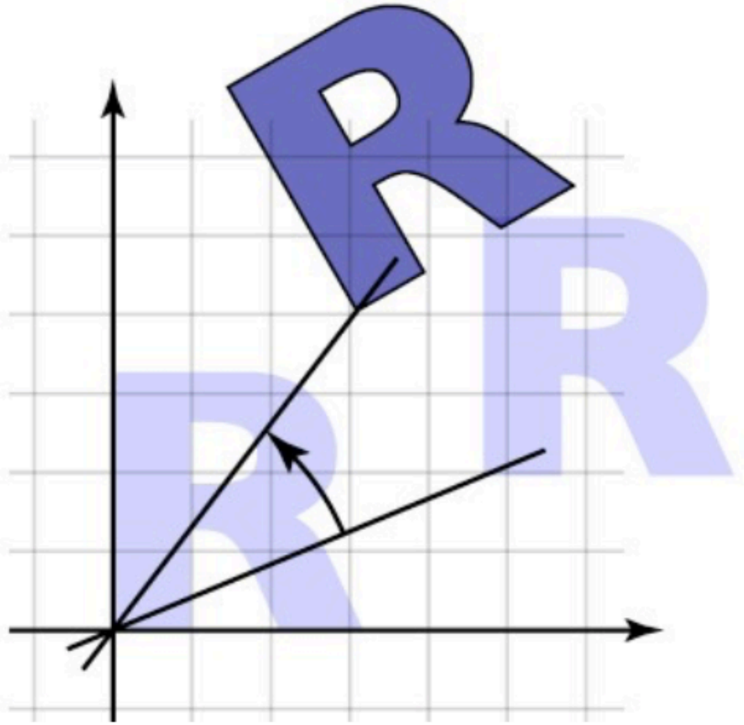
translate, then rotate

Composite affine transformations

- In general not commutative: order matters!



rotate, then translate



translate, then rotate

Rigid motions

- **A transform made up of only translation and rotation is a *rigid motion* or a *rigid body transformation***
- **The linear part is an orthonormal matrix**

$$R = \begin{bmatrix} Q & \mathbf{u} \\ 0 & 1 \end{bmatrix}$$

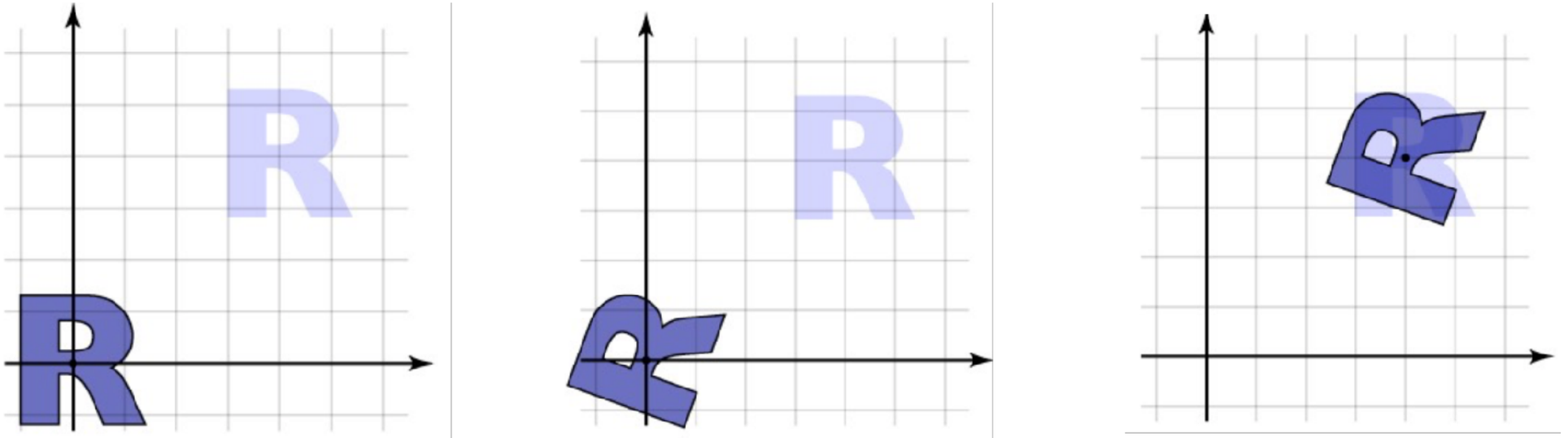
- **Inverse of orthonormal matrix is transpose**
  - so inverse of rigid motion is easy:

$$R^{-1}R = \begin{bmatrix} Q^T & -Q^T\mathbf{u} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} Q & \mathbf{u} \\ 0 & 1 \end{bmatrix}$$

Composing to change axes

- Want to rotate about a particular point
  - could work out formulas directly...
- Know how to rotate about the origin
  - so translate that point to the origin

$$M = T^{-1}RT$$



Transforming points and vectors

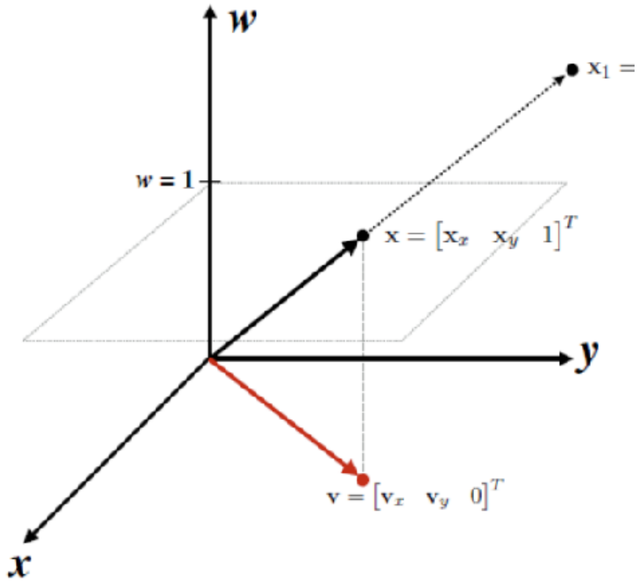
- **Recall distinction points vs. vectors**
  - vectors are just offsets (differences between points)
  - points have a location
    - represented by vector offset from a fixed origin
- **Points and vectors transform differently**
  - points respond to translation; vectors do not

$$\mathbf{v} = \mathbf{p} - \mathbf{q}$$

$$T(\mathbf{x}) = M\mathbf{x} + \mathbf{t}$$

$$\begin{aligned} T(\mathbf{p} - \mathbf{q}) &= M\mathbf{p} + \mathbf{t} - (M\mathbf{q} + \mathbf{t}) \\ &= M(\mathbf{p} - \mathbf{q}) + (\mathbf{t} - \mathbf{t}) = M\mathbf{v} \end{aligned}$$

Transforming points and vectors



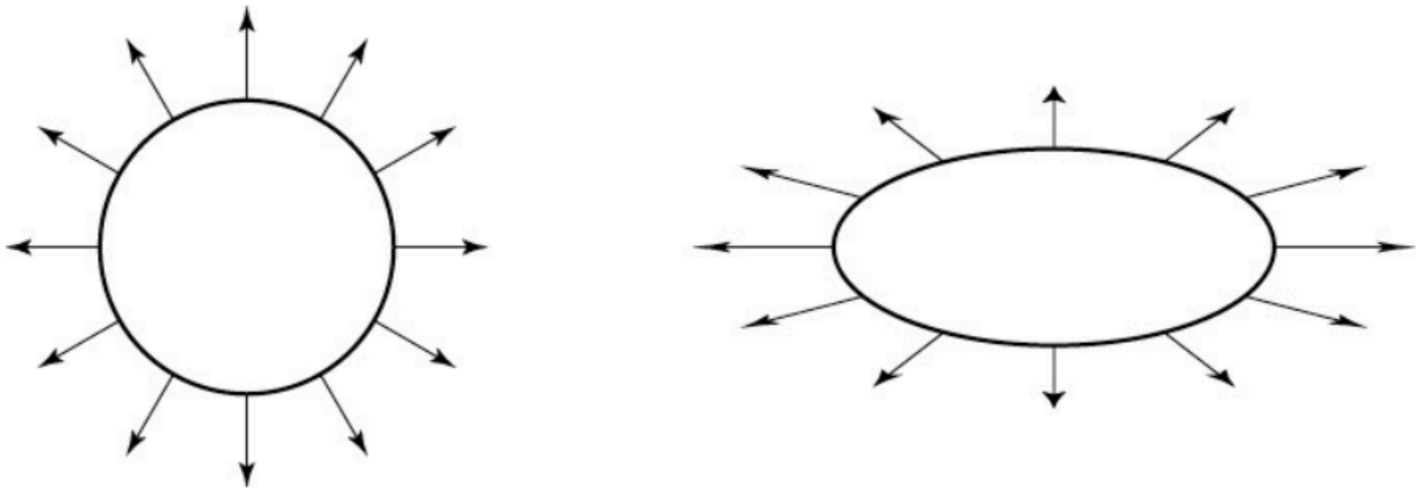
- **Homogeneous coords. let us exclude translation**
  - just put 0 rather than 1 in the last place

$$\begin{bmatrix} M & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} M\mathbf{p}+\mathbf{t} \\ 1 \end{bmatrix} \quad \begin{bmatrix} M & \mathbf{t} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix} = \begin{bmatrix} M\mathbf{v} \\ 0 \end{bmatrix}$$

  - and note that subtracting two points cancels the extra coordinate, resulting in a vector!
- **Preview: projective transformations**
  - what's really going on with this last coordinate?
  - think of R^2 embedded in R^3: all affine xfs. preserve $z=1$ plane
  - could have other transforms; project back to $z=1$

Transforming normal vectors

- **Transforming surface normals**
  - differences of points (and therefore tangents) transform OK
  - normals do not; therefore use inverse transpose matrix



have: $\mathbf{t} \cdot \mathbf{n} = \mathbf{t}^T \mathbf{n} = 0$

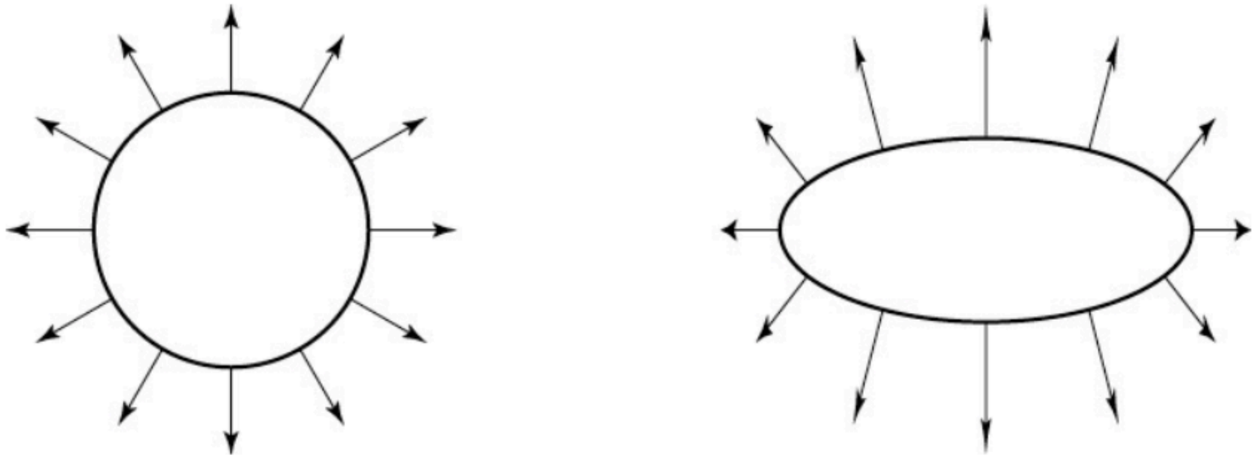
want: $M\mathbf{t} \cdot X\mathbf{n} = \mathbf{t}^T M^T X \mathbf{n} = 0$

so set $X = (M^T)^{-1}$

then: $M\mathbf{t} \cdot X\mathbf{n} = \mathbf{t}^T M^T (M^T)^{-1} \mathbf{n} = \mathbf{t}^T \mathbf{n} = 0$

Transforming normal vectors

- **Transforming surface normals**
  - differences of points (and therefore tangents) transform OK
  - normals do not; therefore use inverse transpose matrix



have: $\mathbf{t} \cdot \mathbf{n} = \mathbf{t}^T \mathbf{n} = 0$

want: $M\mathbf{t} \cdot X\mathbf{n} = \mathbf{t}^T M^T X \mathbf{n} = 0$

so set $X = (M^T)^{-1}$

then: $M\mathbf{t} \cdot X\mathbf{n} = \mathbf{t}^T M^T (M^T)^{-1} \mathbf{n} = \mathbf{t}^T \mathbf{n} = 0$

Summary of basic transforms

Linear:

$$f(\mathbf{x}+\mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y})$$
$$f(a\mathbf{x}) = af(\mathbf{x})$$

Scale
Rotation
Reflection
Shear

Not linear:

Translation

Affine:

Composition of linear transform + translation
(all examples on previous two slides)

$$f(\mathbf{x}) = g(\mathbf{x}) + \mathbf{b}$$

Not affine: perspective projection (will discuss later)

Euclidean: (Isometries)

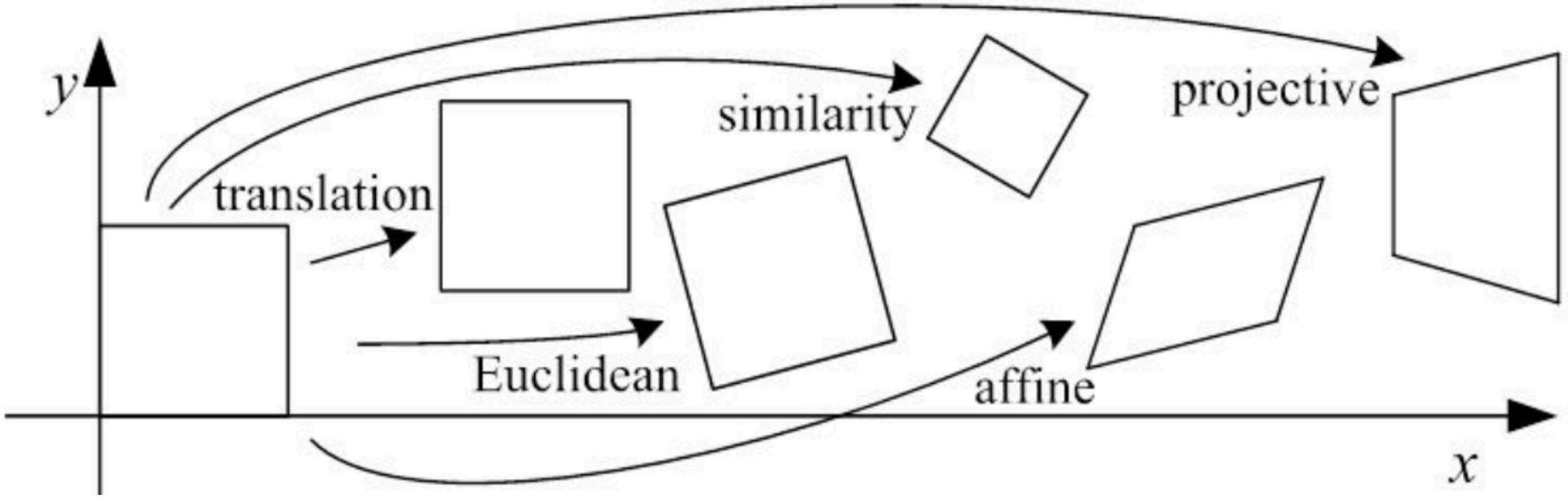
Preserve distance between points (preserves length)

$$|f(\mathbf{x}) - f(\mathbf{y})| = |\mathbf{x} - \mathbf{y}|$$

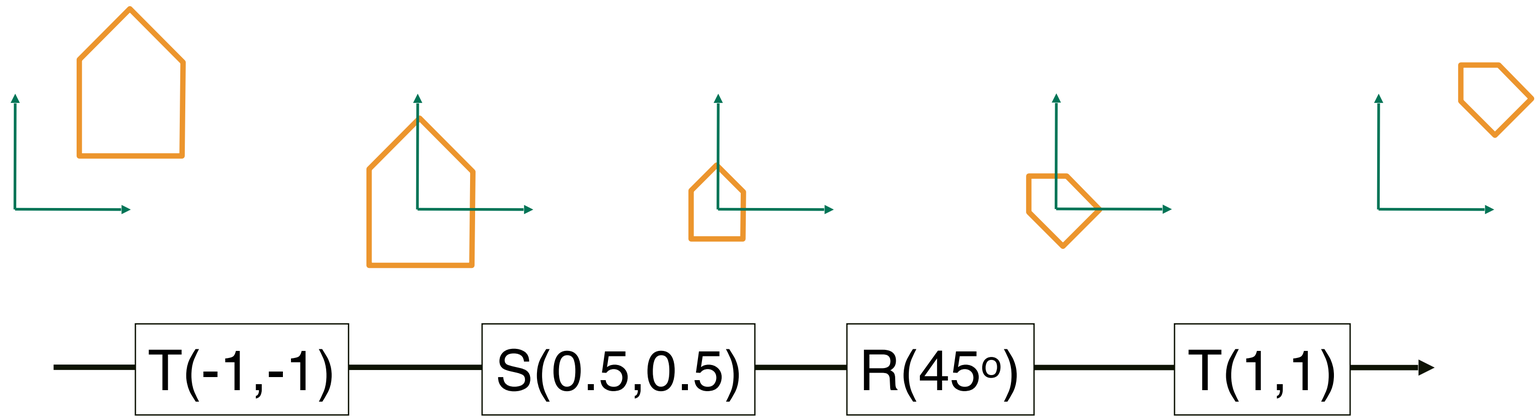
Translation
Rotation
Reflection

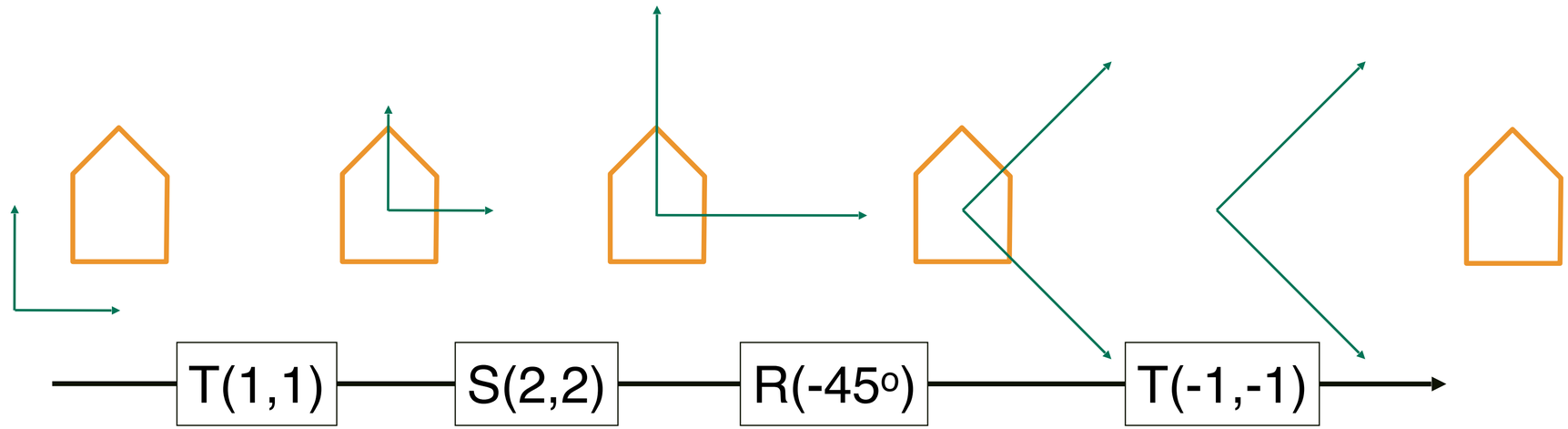
"Rigid body" transforms are Euclidean transforms that also preserve "winding" (does not include reflection)

2D Geometric Transformations



Transform Object or Camera?

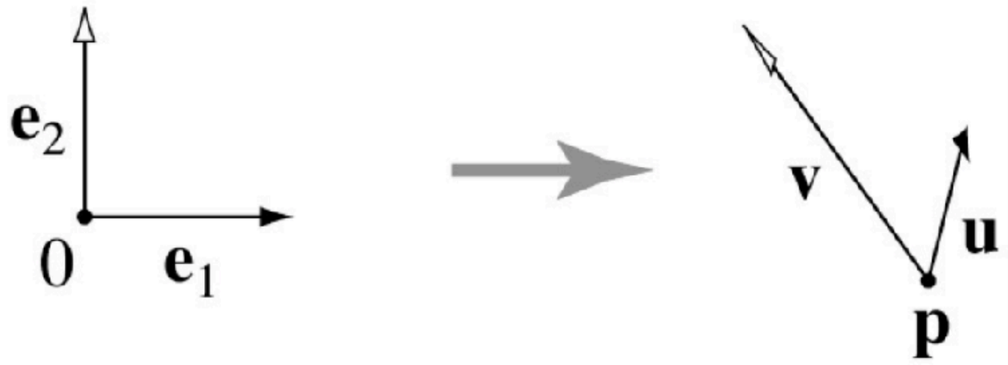




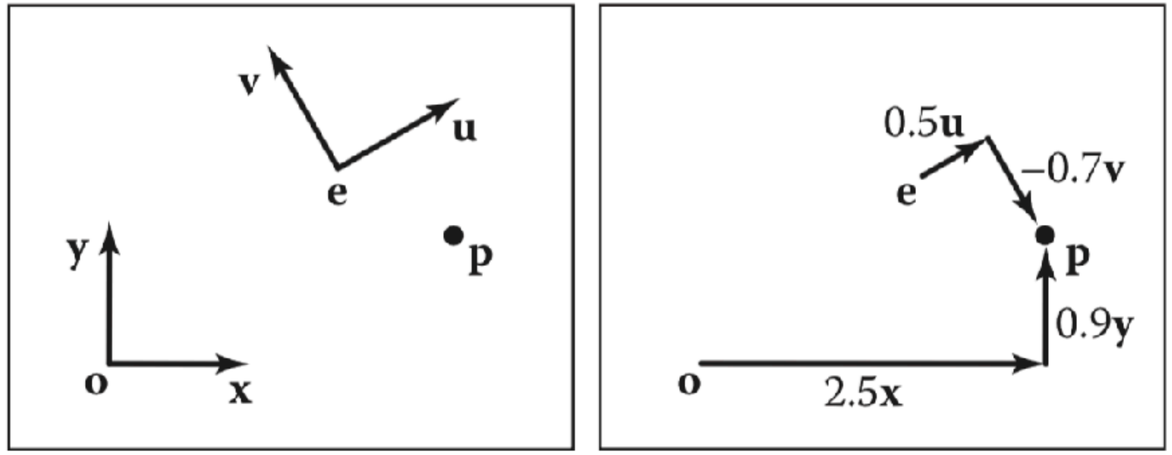
Affine change of coordinates

- Coordinate frame: point plus basis
- Interpretation: transformation changes representation of point from one basis to another
- “Frame to canonical” matrix has frame in columns
  - takes points represented in frame
  - expresents them in canonical basis
  - e.g.[0 0],[1 0],[0 1]
- Seems backward but bears thinking about

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{p} \\ 0 & 0 & 1 \end{bmatrix}$$



frame-to-canonical



The point **p** can be represented in terms of either coordinate system.

$$\mathbf{p} = (u_p, v_p) \equiv \mathbf{e} + u_p\mathbf{u} + v_p\mathbf{v}.$$

$$\mathbf{p} = (x_p, y_p) \equiv \mathbf{o} + x_p\mathbf{x} + y_p\mathbf{y}.$$

$$\begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_e \\ 0 & 1 & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & 0 \\ y_u & y_v & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & x_v & x_e \\ y_u & y_v & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix}$$

$$\mathbf{p}_{xy} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{e} \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p}_{uv}.$$

Canonical-to-frame

$$\begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & x_e \\ 0 & 1 & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_u & x_v & 0 \\ y_u & y_v & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & x_v & x_e \\ y_u & y_v & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & y_u & 0 \\ x_v & y_v & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -x_e \\ 0 & 1 & -y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix}$$

$$\mathbf{p}_{xy} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{e} \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p}_{uv}.$$

$$\mathbf{p}_{uv} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{e} \\ 0 & 0 & 1 \end{bmatrix}^{-1} \mathbf{p}_{xy}. \quad \mathbf{p}_{uv} = \begin{bmatrix} \mathbf{x}_{uv} & \mathbf{y}_{uv} & \mathbf{o}_{uv} \\ 0 & 0 & 1 \end{bmatrix} \mathbf{p}_{xy}.$$

Affine change of coordinates

- **When we move an object to the canonical frame to apply a transformation, we are changing coordinates**
  - the transformation is easy to express in object's frame
  - so define it there and transform it

$$T_e = F T_F F^{-1}$$

  - T_e is the transformation expressed wrt. $\{\mathbf{e}_1, \mathbf{e}_2\}$
  - T_F is the transformation expressed in natural frame
  - F is the frame-to-canonical matrix $[\boldsymbol{u}\ \boldsymbol{v}\ \boldsymbol{p}]$
- **This is a *similarity transformation***

Building general rotations

- **Using elementary transforms you need three**
  - translate axis to pass through origin
  - rotate about ***y*** to get into ***x-y*** plane
  - rotate about ***z*** to align with ***x*** axis

- **Alternative: construct frame and change coordinates**
  - choose ***p***, ***u***, ***v***, ***w*** to be orthonormal frame with ***p*** and ***u*** matching the rotation axis
  - apply similarity transform $T = F\, R_x(\theta)\, F^{-1}$

Coordinate frame summary

- **Frame = point plus basis**
- **Frame matrix (frame-to-canonical) is**

$$F = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{p} \\ 0 & 0 & 1 \end{bmatrix}$$

- **Move points to and from frame by multiplying with *F***

$$p_e = F p_F \quad p_F = F^{-1} p_e$$

- **Move transformations using similarity transforms**

$$T_e = F T_F F^{-1} \quad T_F = F^{-1} T_e F$$

References

Fundamentals of Computer Graphics, Fourth Edition
4th Edition **by Steve Marschner, Peter Shirley**

Chapter 6