Something noteworthy

- Very very noteworthy ...
- OpenGL postmultiply each new transformation matrix

  $M = M \times M_{new}$
- Example: perform translation, then rotation

  0) M = Identity

  1) translation T(tx,ty,0) -> M = M x T(tx,ty,0)

  2) rotation R(θ) -> M = M x R(θ)

  3) Now, transform a point P -> P' = M x P

  = ~~T(tx, ty, 0) x R(θ) x P~~ Wrong!!!

Example Revisit

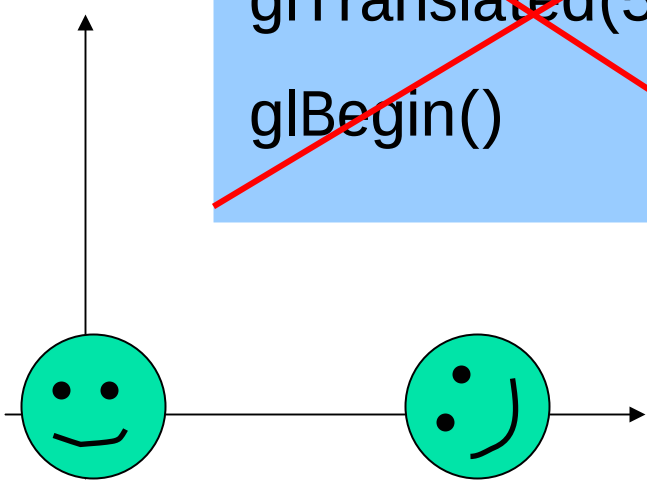
- We want rotation and then translation
- Generate wrong results if you do:

~~glRotated(60,0,0,1);~~

~~glTranslated(5,0,0);~~

~~glBegin()~~

```
glTranslated(5,0,0);
glRotate(60,0,0,1);
glBegin()
...
```



You need to specify the transformation in the opposite order!!

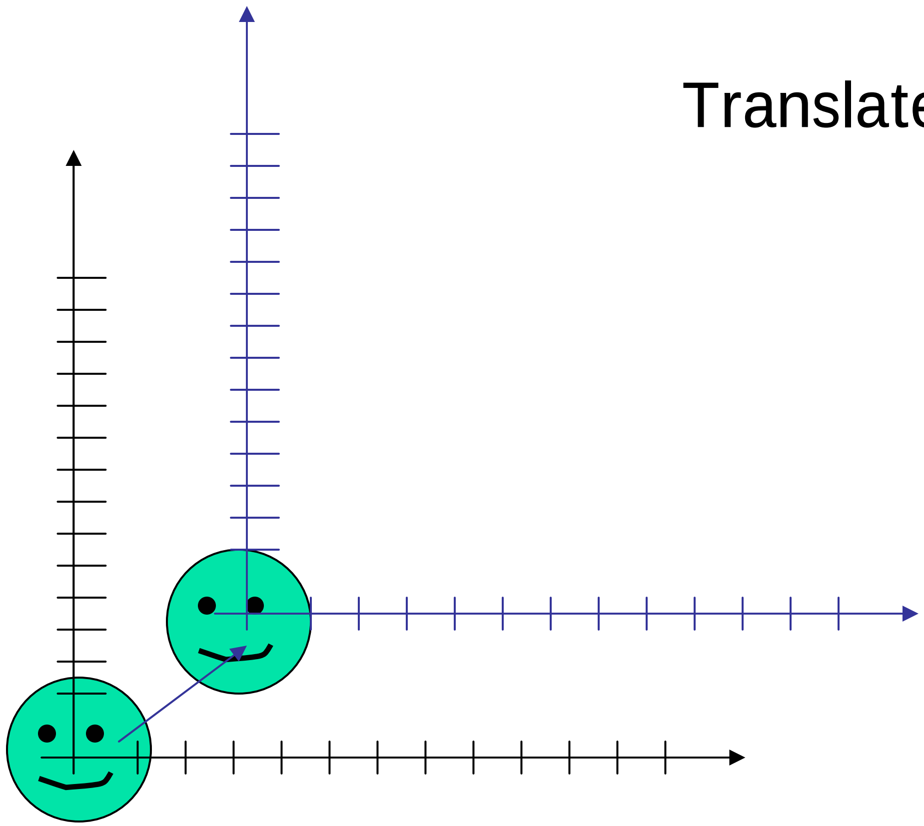
How Strange ...

- OpenGL has its reason ...
- It wants you to think of transformation in a different way
- Instead of thinking of transform the object in a fixed global coordinate system, you should think of transforming an object as moving (transforming) its local coordinate system

OpenGL Transformation

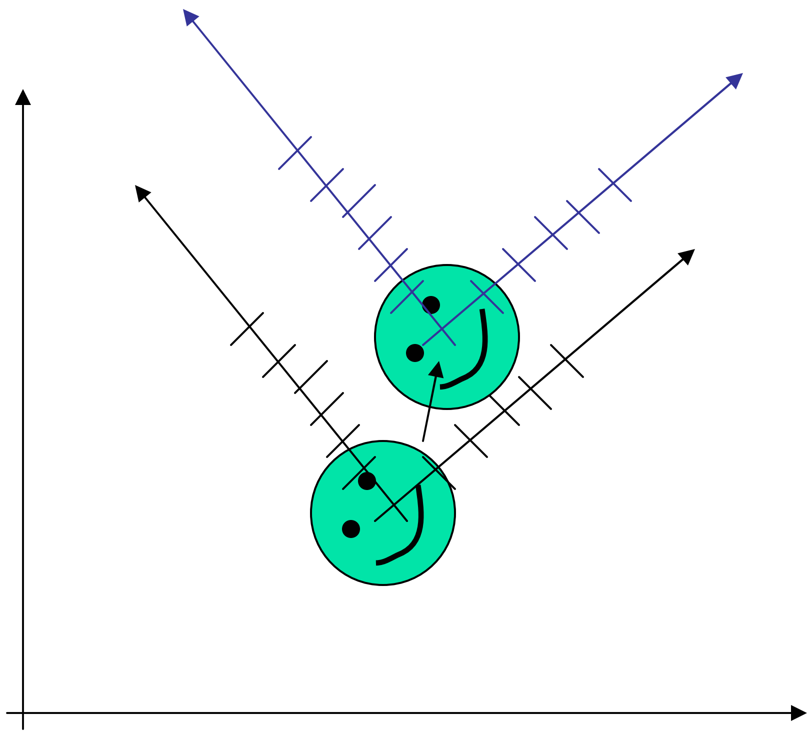
- When use OpenGL, we need to think object transformations as moving (transforming) its local coordinate frame
- All the transformations are performed relative to the current coordinate frame origin and axes

Translate Coordinate Frame



Translate (3,3)?

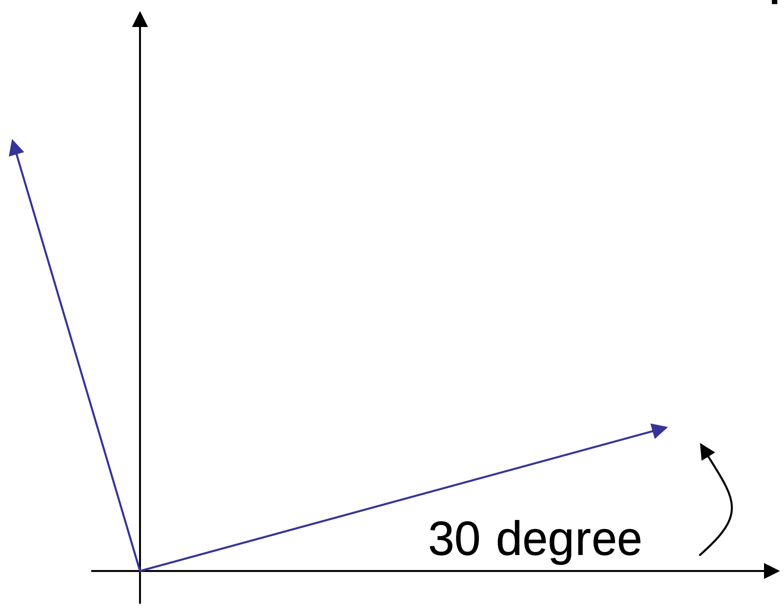
Translate Coordinate Frame (2)



Translate (3,3)?

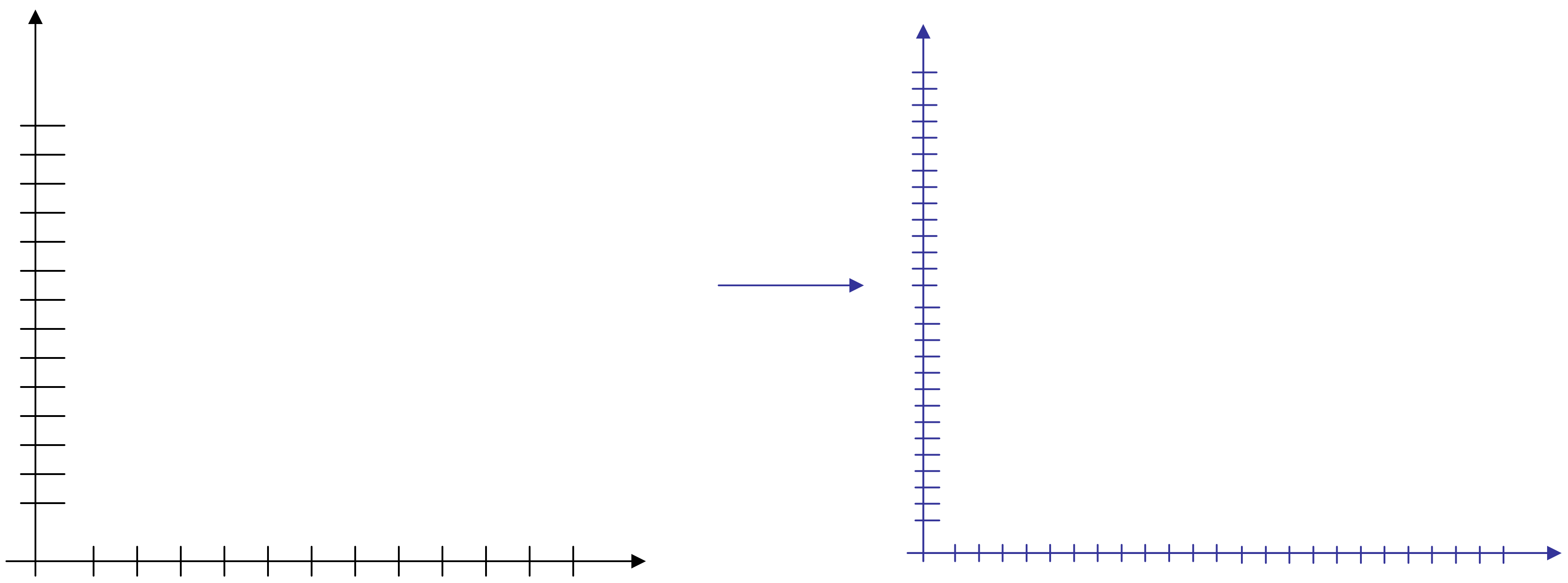
Rotate Coordinate Frame

Rotate 30 degree?

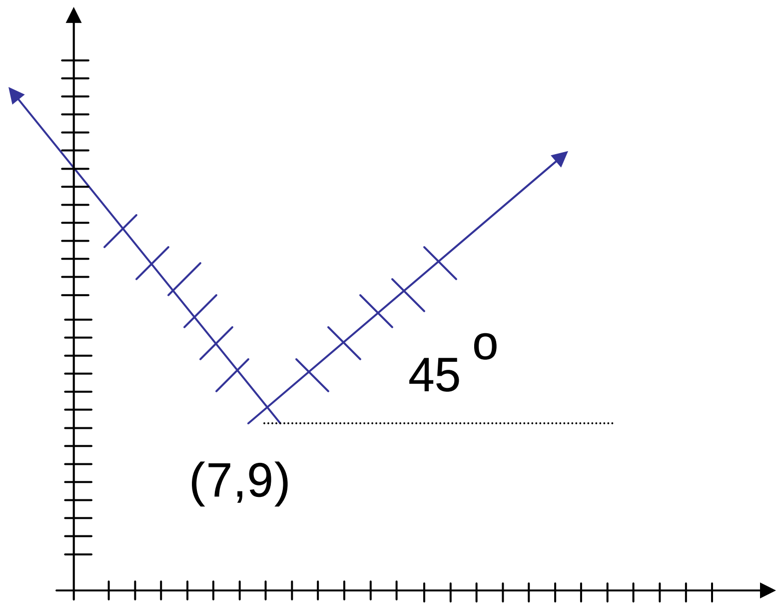


Scale Coordinate Frame

Scale (0.5,0.5)?



Compose Transformations

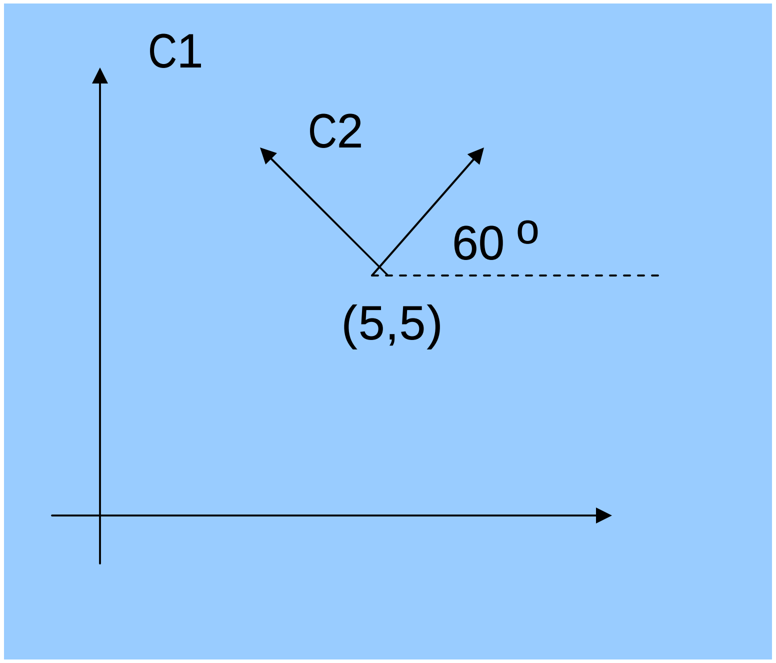


Transformations?

Answer:

1. Translate(7,9)
2. Rotate 45
3. Scale (2,2)

Another example



How do you transform from C1 to C2?

Translate (5,5) and then Rotate (60)

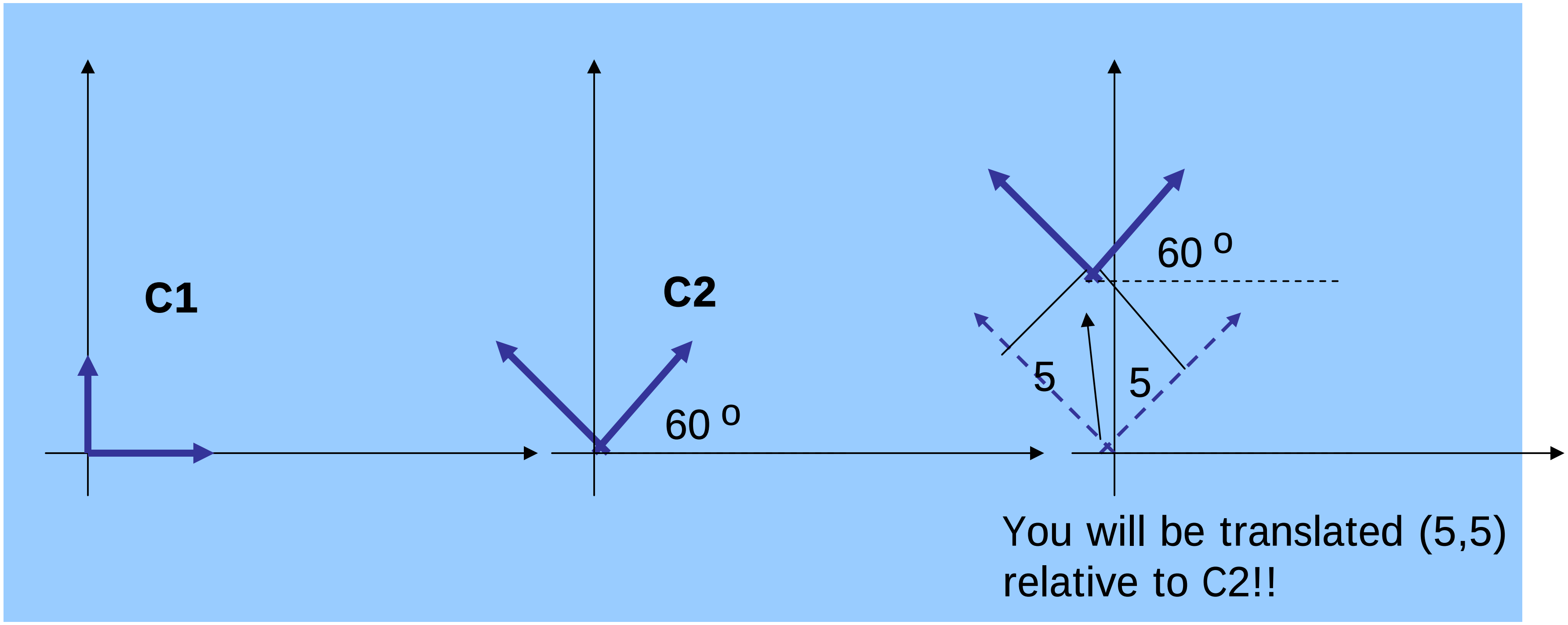
OR

Rotate (60) and then Translate (5,5) ???

Answer: Translate(5,5) and then Rotate (60)

Another example (cont'd)

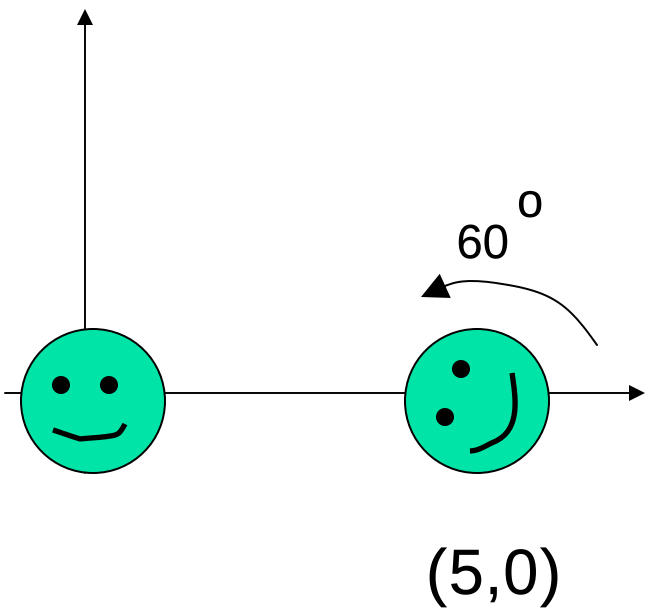
If you Rotate(60) and then Translate(5,5) ...



Transform Objects

- What does coordinate frame transformation have anything to do with object transformation?
  - You can view transformation as to tie the object to a local coordinate frame and move that coordinate frame

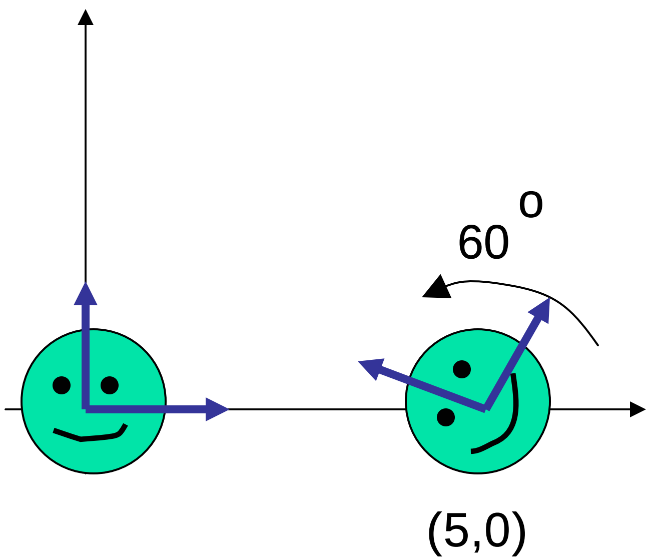
Example



Old way: Transformation as moving the object relative to the origin of a global world coordinate frame

1) Rotate (60^o)
2) Translate (5,0)

Example (cont'd)



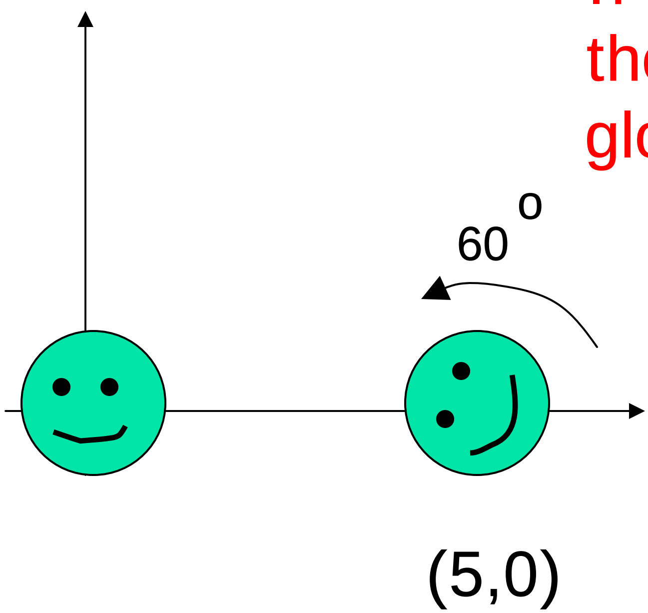
If you think of transformations as moving the local coordinate frame

1) Translate (5,0)
2) Rotate (60^o)

Exact the opposite order compared to the previous slide!!

So …

If you think of transformations as moving the object relative to the origin of a global world coordinate frame

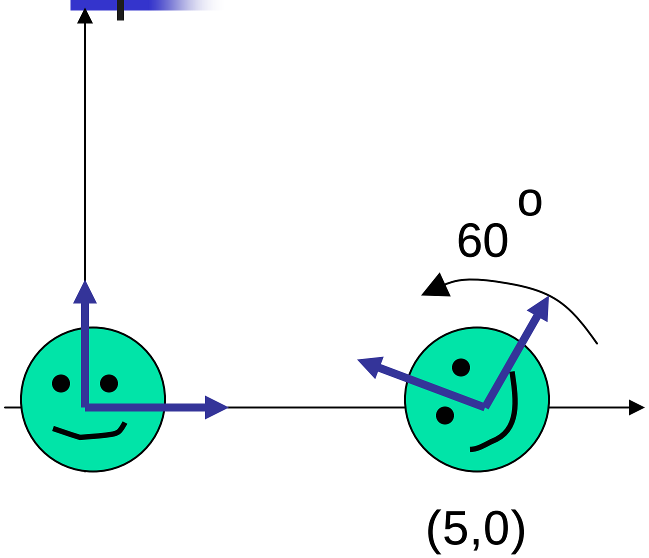


1) Rotate (60^{o}) - M_R
2) Translate (5,0) - M_T

$P' = M_T \times M_R \times P$ is the Correct multiplication

However, OpenGL will do $M_R \times M_T \times P$ if you call glRotate() first, and then glTranslate() because of postmultiplication

So … (cont'd)



If you think of transformations as moving the coordinate frame

1) Translate (5,0) - M_T
2) Rotate (60^o) - M_R

So if you think in terms of moving coordinate frames, you will want to perform Translate first, and then Rotate (I.e., call glTranslate() first and then glRotate())

OpenGL will do $M_T \times M_R \times P$ -> The correct multiplication order!!!

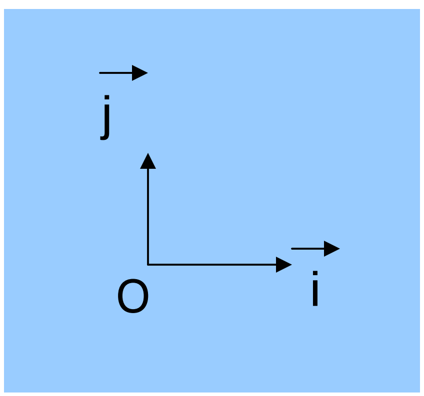
Put it all together

When you use OpenGL ...

- Think of transformation as moving coordinate frames
- Call OpenGL transformation functions in that order
- OpenGL will actually perform the transformations in the reverse order
- Everything will be just right!!!

Change Coordinate System (1)

- What constitutes a coordinate system?
- Origin O
- Basis vector $\vec{i}, \vec{j}$

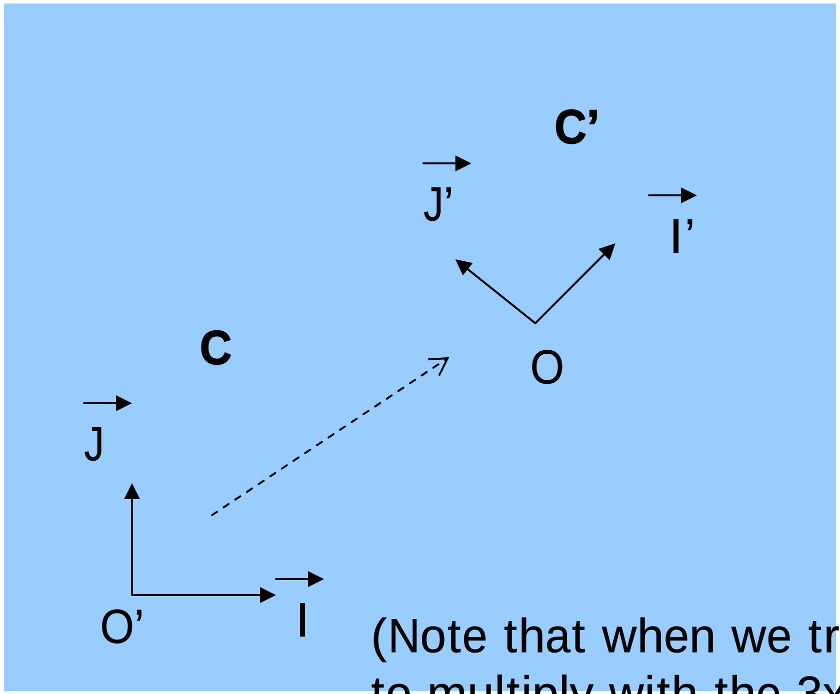


Any point P (x,y) in the coordinate system can be represented:

$$P = O + x * \vec{i} + y * \vec{j}$$

Change Coordinate System (2)

- Transform a coordinate system



We can denote the transformation of coordinate systems as

C' = M x C

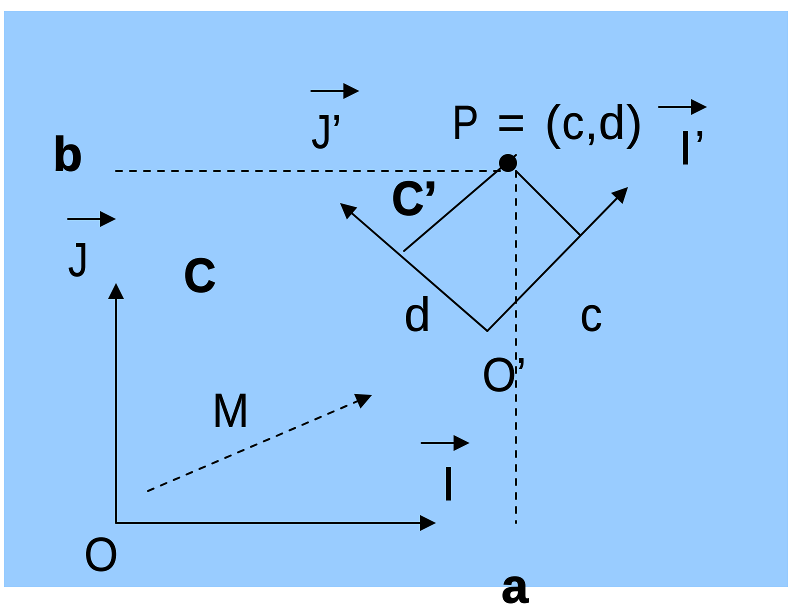
- O' = M x O
- **I'** = M x **I**
- **J'** = M x **J**

(Note that when we transform a vector (a,b), we use (a,b,0) to multiply with the 3x3 matrix M (as opposed to (a,b,1) like we do for points

Change Coordinate System (3)

- Assuming P (c, d) in C', and C' is obtained by transforming C using M, i.e.,

$$C' = M \times C$$

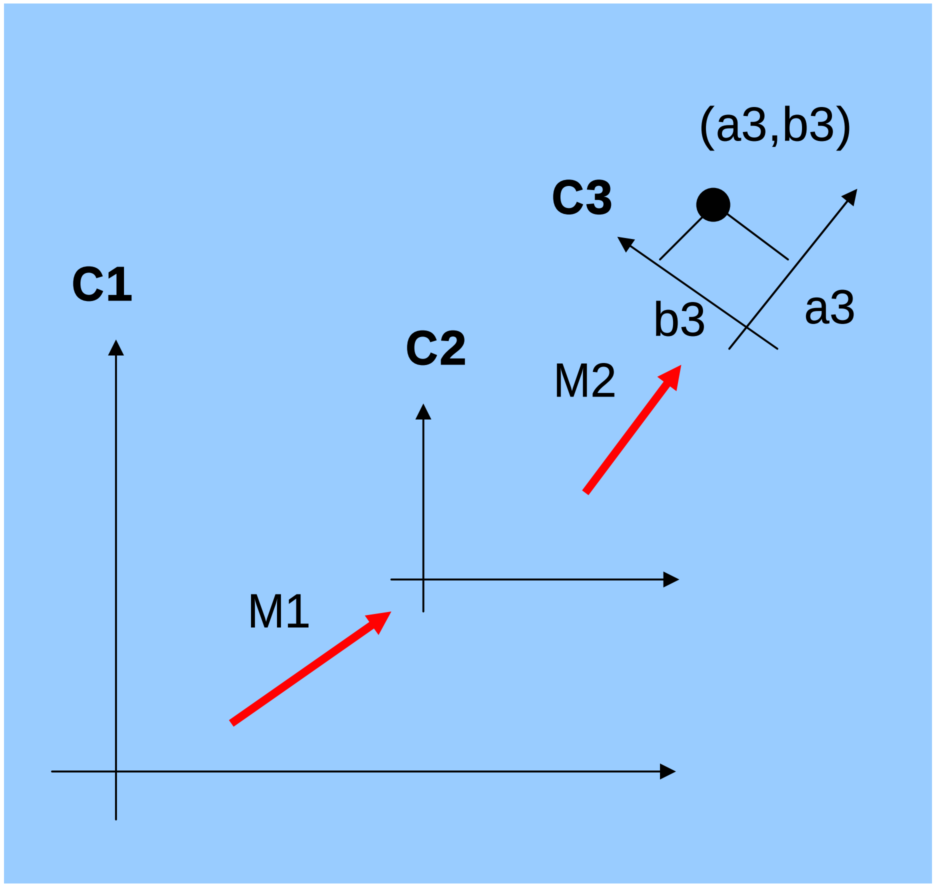


Then the coordinates for P in C is

$$P' = M \times P$$

$$(a,b,1) = M \times (c,d,1)$$

Successive Coordinate Changes



C1 —M1→ C2 —M2→ C3

Given P (a3,b3) in C3
What is P's coordinates in C1?

1) Get P's coordinates in C2
   P_c2 = M2 x P

2) Get P_c2's coordinates in C1
   P_c1 = M1 x P_c2

P_c1 = M1 x M2 x P the answer!!

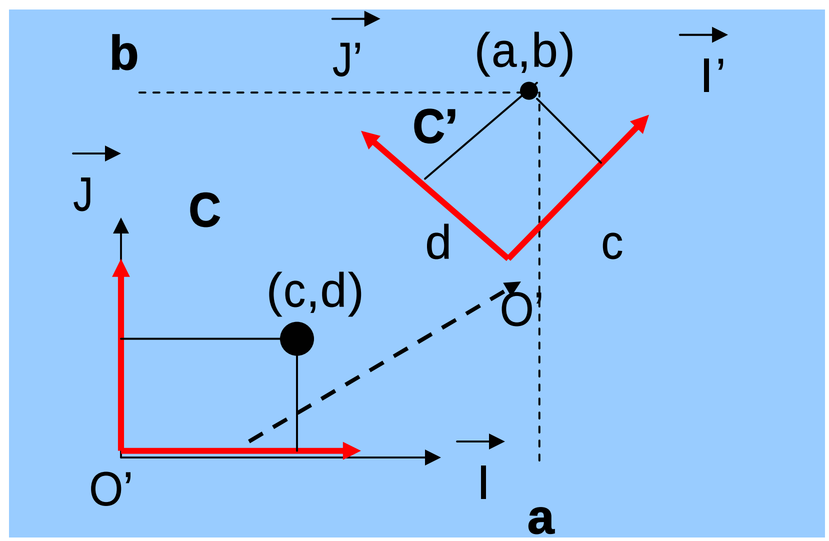
Change Coordinate System (4)

- What does it have anything to do with object transformation?
- We can view transformation as moving the coordinate system (reference frame) and tie the object with that frame

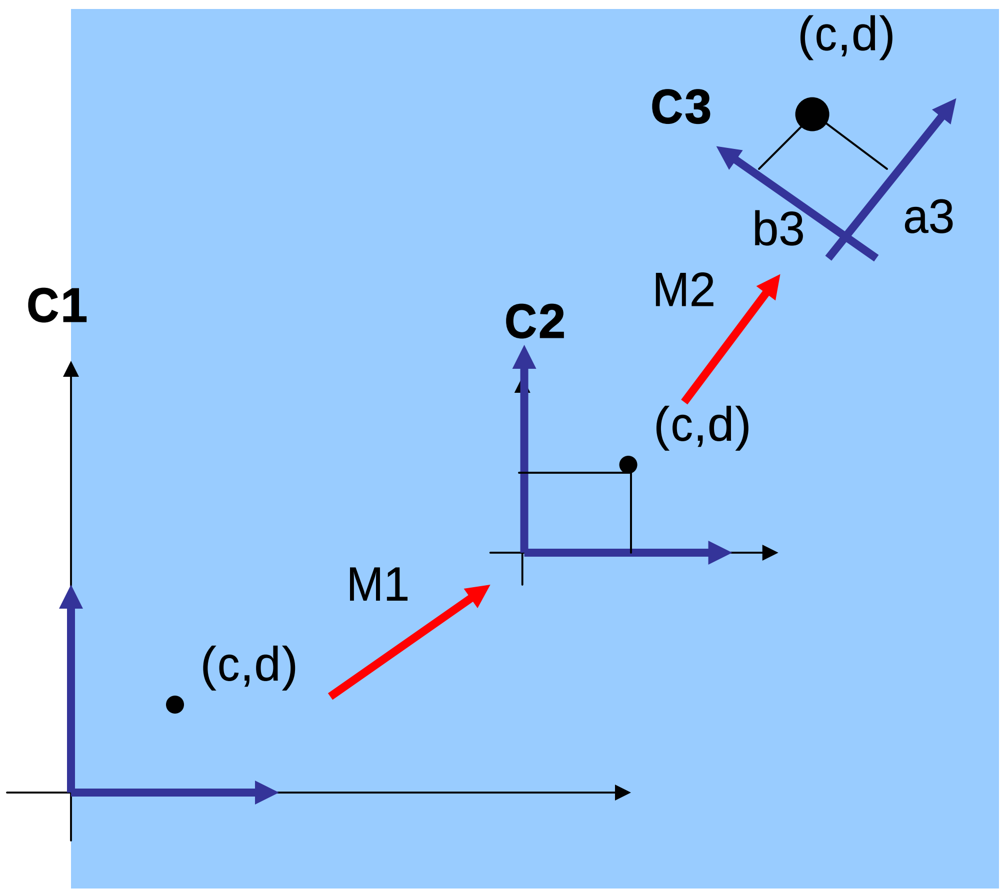
$$\begin{vmatrix} a \\ b \\ 1 \end{vmatrix} = M \times \begin{vmatrix} c \\ d \\ 1 \end{vmatrix}$$

What is (a,b)? The coordinates Of the point P (c,d) in C after the coordinate system change

i.e, the new coordinates after transforming (c,d)



Look at transformation again...

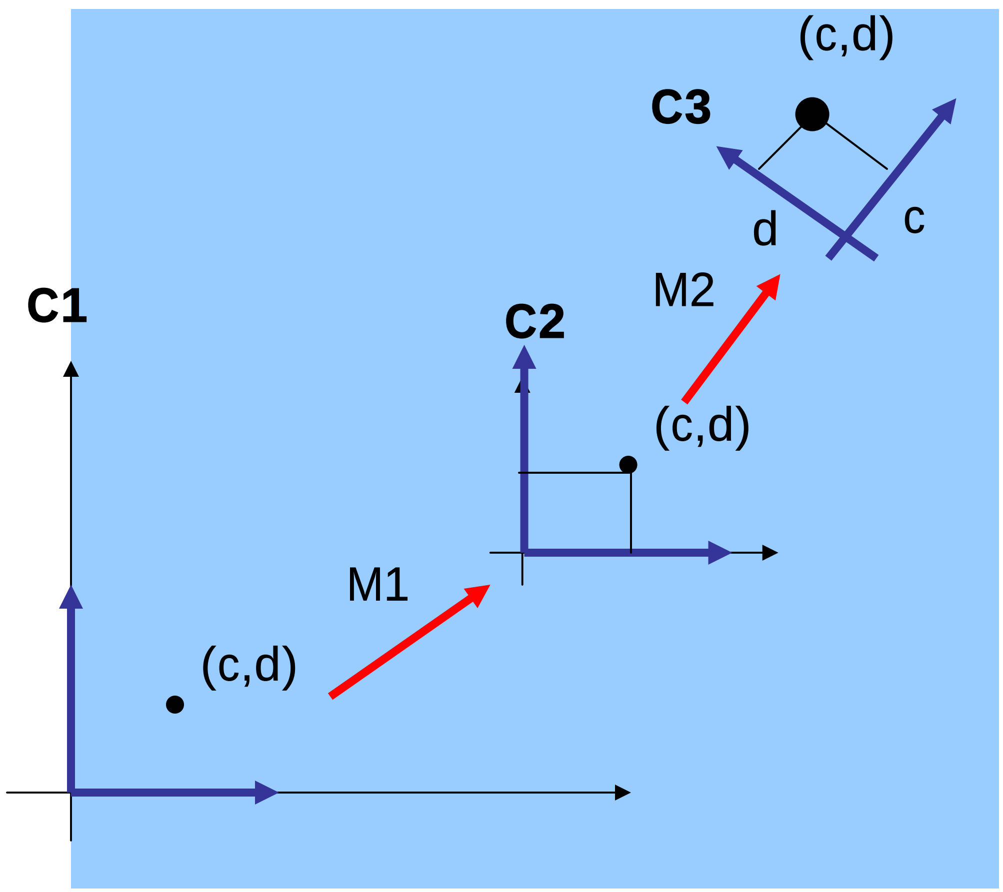


Think transformation of point P (c,d) as a sequence of coordinate frame change

P (c,d) is always tied to the (local) coordinate frame

P's final position after the Transformations?
-> (c,d)'s coordinates in C1

Look at transformation again (2)



Tell OpenGL to transform Using:

M1 (move C1 to C2)
M2 (move C2 to C3)

P's final coordinates =
P's coordinates in C1 =
M1 x M2 x P

This is what we want, and exactly what OpenGL does!!

i.e. Apply the last transformation (M2) to the point first

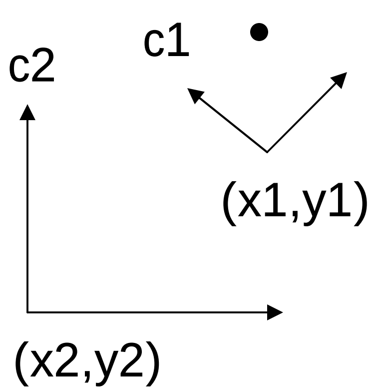
Look at transformation again (3)

- In other words: If you think of transformations as changing coordinate frames, the order that you specify the transformations (for the frames) will be exactly opposite to the order that the transformations are actually applied (i.e. matrix- multiplied) to the object

Put it all together

Coordinate system transformation

- Transform an object from coordinate system C1 with the origin at (x1,y1) or (x1,y1,z1) in 3D, to coordinate system C2 with the origin (x2,y2) or (x2,y2,z1) in 3D



1. Find the transformation sequence to move C2 to C1 (so C2 will align with C1)
   - Move the origin of C2 to coincide with the origin of C1
   - Rotate the basis vectors of C2 so that they coincide wih C1's.
   - Scale the unit if necessary
2. Apply the above transformation sequence to the object in the opposite order