Composing Transformation

- **Composing Transformation** – the process of applying several transformations in succession to form one overall transformation.

- If we apply transform a point \( P \) using \( M_1 \) matrix first, and then transform using \( M_2 \), and then \( M_3 \), then we have:

\[
(M_3 \times (M_2 \times (M_1 \times P))) = M_3 \times M_2 \times M_1 \times P
\]

(matrix multiplication is associative)

\[
M_3 \times M_2 \times M_1 = (M_3 \times M_2) \times M_1 = M_3 \times (M_2 \times M_1)
\]

- Transformation products may not be commutative: \( A \times B \neq B \times A \)

- Some cases where \( A \times B = B \times A \):
  - Translation
  - Scaling
  - Rotation
  - Uniform scaling (\( sx = sy \))

Transformation order matters!

- Example: rotation and translation are not commutative.

  - Translate (5,0) and then Rotate 60 degree
  - OR
  - Rotate 60 degree and then translate (5,0)??

How OpenGL does it?

- OpenGL’s transformation functions are meant to be used in 3D.
- No problem for 2D though – just ignore the z dimension.
- **Translation**:
  - `glTranslatef(d)(tx, ty, tz)` -> `glTranslatef(d)tx,ty,0)` for 2D.
How OpenGL does it?

- **Rotation:**
  - `glRotatef(d)(angle, vx, vy, vz) -> glRotatef(d)(angle, 0,0,1)` for 2D

  ![Rotation Diagram]

  You can imagine z is pointing out of the slide

OpenGL Transformation Composition

- A global modeling transformation matrix
  - (GL_MODELVIEW, called it M here)
  - `glMatrixMode(GL_MODELVIEW)`
  - The user is responsible to reset it if necessary
    - `glLoadIdentity()`
    - `M = 1 0 0
      0 1 0
      0 0 1`

OpenGL Transformation Composition

- Matrices for performing user-specified transformations are multiplied to the model view global matrix
- For example,
  - `glTranslated(1,1,0)`;
  - `M = M x | 1 0 1 |
    | 0 1 1 |
    | 0 0 1 |`

  - All the vertices P defined within `glBegin()` will first go through the transformation (modeling transformation)
    - `P' = M x P`

Transformation Pipeline

- Object Local Coordinates → Modeling transformation → Object World Coordinates
- ...

...
Something noteworthy

- Very very noteworthy …
- OpenGL postmultiply each new transformation matrix
  \[ M = M \times M_{\text{new}} \]
- Example: perform translation, then rotation
  0) \( M = \text{Identity} \)
  1) translation \( T(tx,ty,0) \) \( \rightarrow \) \( M = M \times T(tx,ty,0) \)
  2) rotation \( R(\theta) \) \( \rightarrow \) \( M = M \times R(\theta) \)
  3) Now, transform a point \( P \) \( \rightarrow \) \( P' = M \times P = T(tx, ty, 0) \times R(\theta) \times P \)

Wrong!!!

Example Revisit

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

Wrong!!!

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 transforming 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)?

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) …

You will be translated (5,5) relative to C2!!

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

1) Rotate (60°)
2) Translate (5,0)

(5,0)

If you think of transformations as moving the local coordinate frame

1) Translate (5,0)
2) Rotate (60°)

(5,0)

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°) - Mr
2) Translate (5,0) - Mt

P' = Mt x Mr x P is the Correct multiplication

Unfortunately, OpenGL will do Mr x Mt x P if you call glRotate() first, and then glTranslate() because of postmultiplication

So ...

If you think of transformations as moving the coordinate frame

1) Translate (5,0) - Mt
2) Rotate (60°) - Mr

(5,0)

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 Mt x Mr x 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 \( i, j \)

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

\[
P = O + x \cdot i + y \cdot j
\]

Change Coordinate System (2)

- Transform a coordinate system

We can denote the transformation of coordinate systems as

\[
C' = M \times C
\]

- \( O' = M \times O \)
- \( I' = M \times I \)
- \( J' = M \times J \)

(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

Given $P (a_3,b_3)$ in $C_3$
What is $P$'s coordinates in $C_1$?

1) Get $P$'s coordinates in $C_2$
   $P_{c2} = M_2 \times P$

2) Get $P_{c2}$'s coordinates in $C_1$
   $P_{c1} = M_1 \times P_{c2}$

$P_{c1} = M_1 \times M_2 \times P$ 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

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?
$\rightarrow (c,d)$'s coordinates in $C_1$

Look at transformation again (2)

Tell OpenGL to transform Using:

- $M_1$ (move $C_1$ to $C_2$)
- $M_2$ (move $C_2$ to $C_3$)

$P$'s final coordinates = $P$'s coordinates in $C_1$ =
$M_1 \times M_2 \times P$

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

i.e. Apply the last transformation ($M_2$) 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 \((x_1,y_1)\) or \((x_1,y_1,z_1)\) in 3D, to coordinate system C2 with the origin \((x_2,y_2)\) or \((x_2,y_2,z_1)\) 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 with C1’s.
   - Scale the unit if necessary.

2. Apply the above transformation sequence to the object in the opposite order.