CS 4204 Computer Graphics

2D and 3D Transformations

Doug Bowman
Adapted from notes by Yong Cao
Virginia Tech
Transformations

What are they?
- changing something to something else via rules
- mathematics: mapping between values in a range set and domain set (function/relation)
- geometric: translate, rotate, scale, shear,…

Why are they important to graphics?
- moving objects on screen / in space
- specifying the camera’s view of a 3D scene
- mapping from model space to world space to camera space to screen space
- specifying parent/child relationships
- …
Moving an object is called a translation. We translate a point by adding to the x and y coordinates, respectively, the amount the point should be shifted in the x and y directions. We translate an object by translating each vertex in the object.

\[ x_{\text{new}} = x_{\text{old}} + t_x; \quad y_{\text{new}} = y_{\text{old}} + t_y \]
Scaling

Changing the size of an object is called a scale. We scale an object by scaling the x and y coordinates of each vertex in the object.

\[
\begin{align*}
    s_x &= \frac{w_{\text{new}}}{w_{\text{old}}} \\
    x_{\text{new}} &= s_x x_{\text{old}} \\
    s_y &= \frac{h_{\text{new}}}{h_{\text{old}}} \\
    y_{\text{new}} &= s_y y_{\text{old}}
\end{align*}
\]
Consider rotation about the origin by $\Theta$ degrees

- Radius stays the same, angle increases by $\Theta$

\[x' = r \cos (\phi + \theta)\]
\[y' = r \sin (\phi + \theta)\]
Rotation about the origin (cont.)

From the double angle formulas:

\[ \sin(A + B) = \sin A \cos B + \cos A \sin B \]
\[ \cos(A + B) = \cos A \cos B - \sin A \sin B \]

Thus,

\[ x' = x \cos \theta - y \sin \theta \]
\[ y' = x \sin \theta + y \cos \theta \]
Transformations as matrices

**Scale:**
\[
\begin{align*}
  x_{\text{new}} &= s_x x_{\text{old}} \\
  y_{\text{new}} &= s_y y_{\text{old}}
\end{align*}
\]

\[
\begin{bmatrix}
  s_x & 0 \\
  0 & s_y
\end{bmatrix}
\begin{bmatrix}
  x \\
  y
\end{bmatrix}
=
\begin{bmatrix}
  s_x \cdot x \\
  s_y \cdot y
\end{bmatrix}
\]

**Rotation:**
\[
\begin{align*}
  x_{\text{new}} &= x \cos \theta - y \sin \theta \\
  y_{\text{new}} &= x \sin \theta + y \cos \theta
\end{align*}
\]

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

**Translation:**
\[
\begin{align*}
  x_{\text{new}} &= x_{\text{old}} + t_x \\
  y_{\text{new}} &= y_{\text{old}} + t_y
\end{align*}
\]

\[
\begin{bmatrix}
  t_x \\
  t_y
\end{bmatrix}
+ 
\begin{bmatrix}
  x \\
  y
\end{bmatrix}
=
\begin{bmatrix}
  x + t_x \\
  y + t_y
\end{bmatrix}
\]
Homogeneous Coordinates

In order to represent a translation as a matrix multiplication operation we use $3 \times 3$ matrices and pad our points to become $3 \times 1$ matrices. This coordinate system (using three values to represent a 2D point) is called homogeneous coordinates.

\[
\begin{align*}
P_{(x,y)} &= \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \\
R_\theta &= \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
S_{s_x, s_y} &= \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
T_{t_x, t_y} &= \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}
\end{align*}
\]
Composite Transformations

Suppose we wished to perform multiple transformations on a point:

\[
\begin{align*}
P_2 &= T_{3,1}P_1 \\
P_3 &= S_{2,2}P_2 \\
P_4 &= R_{30}P_3 \\
M &= R_{30}S_{2,2}T_{3,1} \\
P_4 &= MP_1
\end{align*}
\]

Remember:
- Matrix multiplication is associative, not commutative!
- Transform matrices must be pre-multiplied
- The first transformation you want to perform will be at the far right, just before the point
Composite Transformations - Scaling

*Given our three basic transformations we can create other transformations.*

*A problem with the scale transformation is that it also moves the object being scaled.*

Scale a line between (2, 1) (4,1) to twice its length.
If we scale a line between (0,0) & (2,0) to twice its length, the left-hand endpoint does not move.

(0,0) is known as a fixed point for the basic scaling transformation. We can use composite transformations to create a scale transformation with different fixed points.
Fixed Point Scaling

Scale by 2 with fixed point = (2, 1)

Translate the point (2, 1) to the origin

Scale by 2

Translate origin to point (2, 1)

\[
\begin{array}{cccc}
1 & 0 & 2 & 2 \\
0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 \\
\end{array}
\begin{array}{cccc}
2 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 \\
\end{array}
\begin{array}{c}
10 -2 \\
01 -1 \\
00 1 \\
\end{array}
\begin{array}{c}
20 -2 \\
01 0 \\
00 1 \\
\end{array}
\begin{array}{c}
T_{2,3} \\
S_{2,1} \\
T_{2,-1} \\
C \\
\end{array}
\]

Before

After
Example of 2D transformation

*Rotate around an arbitrary point A:*
Rotate around an arbitrary point
Rotate around an arbitrary point

We know how to rotate around the origin

\[
\begin{pmatrix}
Q_x \\
Q_y \\
1
\end{pmatrix} =
\begin{pmatrix}
\cos \theta & -\sin \theta & 0 \\
\sin \theta & \cos \theta & 0 \\
0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
P_x \\
P_y \\
1
\end{pmatrix}
\]
Rotate around an arbitrary point

\[ \text{but that is not what we want to do!} \]
So what do we do?
Transform it to a known case

\[ \text{Translate}(-A_x, -A_y) \]
Second step: Rotation

\[ \text{Translate}(-A_x, -A_y) \]
\[ \text{Rotate}(-90) \]

Diagram showing the transformation of point \( A \) to \( P' \) and then to \( P'' \) with the given transformations.
Final: Put everything back

\[ \text{Translate}(-A_x, -A_y) \]
\[ \text{Rotate}(-90) \]
\[ \text{Translate}(A_x, A_y) \]
Rotation about arbitrary point

**IMPORTANT!**: Order

\[ M = T(A_x, A_y) \ R(-90) \ T(-A_x, -A_y) \]
Rotation about arbitrary point

Rotation of \( \theta \) degrees about point (x,y)

Translate (x,y) to origin

Rotate

Translate origin to (x,y)

\[
C = \begin{bmatrix}
1 & 0 & x \\
0 & 1 & y \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\cos \theta & -\sin \theta & 0 \\
\sin \theta & \cos \theta & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & -x \\
0 & 1 & -y \\
0 & 0 & 1
\end{bmatrix}
\]

\( T_{x,y} \)  \( R_\theta \)  \( T_{-x,-y} \)
Shears

Original Data → y Shear → x Shear

\[
\begin{bmatrix}
1 & 0 & 0 \\
\alpha & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\quad \quad
\begin{bmatrix}
1 & b & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]
Reflections

Reflection about the y-axis

\[
\begin{pmatrix}
-1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{pmatrix}
\]

Reflection about the x-axis

\[
\begin{pmatrix}
1 & 0 & 0 \\
0 & -1 & 0 \\
0 & 0 & 1 \\
\end{pmatrix}
\]
More Reflections

Reflection about the origin

\[
\begin{pmatrix}
-1 & 0 & 0 \\
0 & -1 & 0 \\
0 & 0 & 1
\end{pmatrix}
\]

Reflection about the line \(y = x\)

?
Affine Transformations in 3D

General form

\[
\begin{pmatrix}
Q_x \\
Q_y \\
Q_z \\
1
\end{pmatrix} =
\begin{pmatrix}
m_{11} & m_{12} & m_{13} & m_{14} \\
m_{21} & m_{22} & m_{23} & m_{24} \\
m_{31} & m_{32} & m_{33} & m_{34} \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
P_x \\
P_y \\
P_z \\
1
\end{pmatrix}
\]
Elementary 3D Affine Transformations

**Translation**

\[
\begin{pmatrix}
Q_x \\
Q_y \\
Q_z \\
1
\end{pmatrix}
= 
\begin{pmatrix}
1 & 0 & 0 & T_x \\
0 & 1 & 0 & T_y \\
0 & 0 & 1 & T_z \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
P_x \\
P_y \\
P_z \\
1
\end{pmatrix}
\]
Scaling Around the Origin

\[
\begin{pmatrix}
Q_x \\
Q_y \\
Q_z \\
1
\end{pmatrix} =
\begin{pmatrix}
s_x & 0 & 0 & 0 \\
0 & s_y & 0 & 0 \\
0 & 0 & s_z & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
P_x \\
P_y \\
P_z \\
1
\end{pmatrix}
\]
Shear around the origin

Along x-axis

\[
\begin{pmatrix}
Q_x \\
Q_y \\
Q_z \\
1
\end{pmatrix}
= \begin{pmatrix}
1 & a & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
P_x \\
P_y \\
P_z \\
1
\end{pmatrix}
\]
### 3D Rotation

**Various representations**

*Decomposition into axis rotations* (x-roll, y-roll, z-roll)

**CCW positive assumption**
Reminder: 2D rotation (z-roll)

\[ Q_x = \cos \theta P_x - \sin \theta P_y \]
\[ Q_y = \sin \theta P_x + \cos \theta P_y \]

\[
\begin{pmatrix}
Q_x \\
Q_y \\
1
\end{pmatrix} =
\begin{pmatrix}
\cos \theta & -\sin \theta & 0 \\
\sin \theta & \cos \theta & 0 \\
0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
P_x \\
P_y \\
1
\end{pmatrix}
\]
Three axes to rotate around
Z-roll

\[ Q_x = \cos \theta P_x - \sin \theta P_y \]
\[ Q_y = \sin \theta P_x + \cos \theta P_y \]
\[ Q_z = P_z \]

\[ R_z(\theta) = \begin{pmatrix}
\cos(\theta) & -\sin(\theta) & 0 & 0 \\
\sin(\theta) & \cos(\theta) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1
\end{pmatrix} \]
X-roll

**Cyclic indexing**

\[ x \rightarrow y \rightarrow z \rightarrow x \rightarrow y \]

\[ Q_y = \cos \theta P_y - \sin \theta P_z \]
\[ Q_z = \sin \theta P_y + \cos \theta P_z \]
\[ Q_x = P_x \]

\[
R_x(\theta) = \begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & \cos(\theta) & -\sin(\theta) & 0 \\
0 & \sin(\theta) & \cos(\theta) & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
Y-roll

\[ Q_z = \cos \theta P_z - \sin \theta P_x \]
\[ Q_x = \sin \theta P_z + \cos \theta P_x \]
\[ Q_y = P_y \]

\[ R_y(\theta) = \begin{pmatrix}
\cos(\theta) & 0 & \sin(\theta) & 0 \\
0 & 1 & 0 & 0 \\
-\sin(\theta) & 0 & \cos(\theta) & 0 \\
0 & 0 & 1 & 0
\end{pmatrix} \]
Rigid body transformations

Translations and rotations
Preserve lines, angles and distances
Inversion of transformations

Translation: \( T^{-1}(a,b,c) = T(-a,-b,-c) \)

Rotation: \( R^{-1}_{\text{axis}}(b) = R_{\text{axis}}(-b) \)

Scaling: \( S^{-1}(sx, sy, sz) = S(1/sx, 1/sy, 1/sz) \)

Shearing: \( Sh^{-1}(a) = Sh(-a) \)
Rotation around an arbitrary axis

*Euler’s theorem:* Any rotation or sequence of rotations around a point is equivalent to a single rotation around an axis that passes through the point.

What does the matrix look like?
Rotation around an arbitrary axis

Axis: \( \mathbf{u} \)
Point: \( P \)
Angle: \( \beta \)
Method:

1. Two rotations to align \( \mathbf{u} \) with \( x \)-axis
2. Do \( x \)-roll by \( \beta \)
3. Undo the alignment
Derivation

1. $R_z(-\phi)R_y(\theta)$
2. $R_x(\beta)$
3. $R_y(-\theta)R_z(\phi)$

Altogether:

$R_y(-\theta)R_z(\phi)R_x(\beta)R_z(-\phi)R_y(\theta)$

We can add translation too if the axis is not through the origin.
## Properties of affine transformations

- *Preservation of affine combinations of points.*
- *Preservation of lines and planes.*
- *Preservation of parallelism of lines and planes.*
- *Relative ratios are preserved*
- *Affine transformations are composed of elementary ones.*
General form

Rotation, Scaling, Shear

Translation

\[
\begin{pmatrix}
m_{11} & m_{12} & m_{13} & m_{14} \\
m_{21} & m_{22} & m_{23} & m_{24} \\
m_{31} & m_{32} & m_{33} & m_{34} \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
Transformations as a change in coordinate system

All transformations we have looked at involve transforming points in a fixed coordinate system (CS).

Can also think of them as a transformation of the CS itself
Transforming the CS - examples

Translate(4,4)

Rotate(180°)
Why transform the CS?

Objects often defined in a “natural” or “convenient” CS

To draw objects transformed by T, we could:

- Transform each vertex by T, then draw
- Or, draw vertices in a transformed CS
Drawing in transformed CS

Tell system once how to draw the object, then draw in a transformed CS to transform the object.

House drawn in a CS that’s been translated, rotated, and scaled.
Mapping between systems

Given:

- The vertices of an object in CS₂
- A transformation matrix M that transforms CS₁ to CS₂

What are the coordinates of the object’s vertices in CS₁?
Mapping example

Point P is at (0,0) in the transformed CS (CS₂). Where is it in CS₁?

Answer: (4,4)

*Note: (4,4) = T_{4,4} P
Mapping rule

*In general, if CS\(_1\) is transformed by a matrix \(M\) to form CS\(_2\), a point P in CS\(_2\) is represented by MP in CS\(_1\)*
Another example

Translate(4,4), then Scale(0.5, 0.5)

Where is P in CS\(_3\)? (2,2)
Where is P in CS\(_2\)? \(S_{0.5,0.5}(2,2) = (1,1)\)
Where is P in CS\(_1\)? \(T_{4,4}(1,1) = (5,5)\)

*Note: to go directly from CS\(_3\) to CS\(_1\) we can calculate \(T_{4,4} S_{0.5,0.5}(2,2) = (5,5)\)*
<table>
<thead>
<tr>
<th>General mapping rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>If CS$<em>1$ is transformed consecutively by $M_1$, $M_2$, ..., $M_n$ to form CS$</em>{n+1}$, then a point P in CS$_{n+1}$ is represented by</strong></td>
</tr>
<tr>
<td>$M_1 M_2 ... M_n P$ in CS$_1$.</td>
</tr>
</tbody>
</table>

**To form the composite transformation between CSs, you postmultiply each successive transformation matrix.**
OpenGL Transformations

Learn how to carry out transformations in OpenGL

• Rotation
• Translation
• Scaling

Introduce OpenGL matrix modes

• Model-view
• Projection
OpenGL Matrices

In OpenGL matrices are part of the state

**Multiple types**

- Model-View (GL_MODELVIEW)
- Projection (GL_PROJECTION)
- Texture (GL_TEXTURE) (ignore for now)
- Color(GL_COLOR) (ignore for now)

**Single set of functions for manipulation**

**Select which to manipulated by**

- glMatrixMode(GL_MODELVIEW);
- glMatrixMode(GL_PROJECTION);
**Current Transformation Matrix (CTM)**

*Conceptually there is a 4 x 4 homogeneous coordinate matrix, the current transformation matrix (CTM) that is part of the state and is applied to all vertices that pass down the pipeline.*

*The CTM is defined in the user program and loaded into a transformation unit.*

\[
\begin{align*}
\text{vertices} & \rightarrow \text{CTM} \rightarrow \text{vertices} \\
\text{p} & \downarrow \quad \text{C} \\
\text{p'} = \text{Cp} & 
\end{align*}
\]
CTM operations

The CTM can be altered either by loading a new CTM or by postmultiplication

- Load an identity matrix: \( C \leftarrow I \)
- Load an arbitrary matrix: \( C \leftarrow M \)
- Load a translation matrix: \( C \leftarrow T \)
- Load a rotation matrix: \( C \leftarrow R \)
- Load a scaling matrix: \( C \leftarrow S \)

- Postmultiply by an arbitrary matrix: \( C \leftarrow CM \)
- Postmultiply by a translation matrix: \( C \leftarrow CT \)
- Postmultiply by a rotation matrix: \( C \leftarrow CR \)
- Postmultiply by a scaling matrix: \( C \leftarrow CS \)
Rotation about a Fixed Point

Start with identity matrix: $C \leftarrow I$

Move fixed point to origin: $C \leftarrow CT$

Rotate: $C \leftarrow CR$

Move fixed point back: $C \leftarrow CT^{-1}$

Result: $C = TRT^{-1}$ which is backwards.

This result is a consequence of doing postmultiplications.
Let's try again.
Reversing the Order

We want $C = T^{-1} R T$
so we must do the operations in the following order

$C \leftarrow I$
$C \leftarrow CT^{-1}$
$C \leftarrow CR$
$C \leftarrow CT$

Each operation corresponds to one function call in the program.

Note that the last operation specified is the first executed in the program.
CTM in OpenGL

OpenGL has a model-view and a projection matrix in the pipeline which are concatenated together to form the CTM.

Can manipulate each by first setting the correct matrix mode.
**Rotation, Translation, Scaling**

Load an identity matrix:

```gl
glLoadIdentity()
```

Multiply on right:

```gl
    glRotatef(\text{theta}, \text{vx}, \text{vy}, \text{vz})
    \text{theta} \text{ in degrees, (vx, vy, vz) define axis of rotation}
    glTranslatef(\text{dx}, \text{dy}, \text{dz})
    glScalef(\text{sx}, \text{sy}, \text{sz})
```

Each has a float (f) and double (d) format (glScaled)
Example

Rotation about z axis by 30 degrees with a fixed point of (1.0, 2.0, 3.0)

```c
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(1.0, 2.0, 3.0);
glRotatef(30.0, 0.0, 0.0, 1.0);
glTranslatef(-1.0, -2.0, -3.0);
```

Remember that last matrix specified in the program is the first applied
Arbitrary Matrices

Can load and multiply by matrices defined in the application program

```c
glLoadMatrixf(m)
glMultMatrixf(m)
```

The matrix \( m \) is a one dimension array of 16 elements which are the components of the desired 4 x 4 matrix stored by columns.

In `glMultMatrixf`, \( m \) multiplies the existing matrix on the right.
Transformations in OpenGL

*OpenGL makes it easy to do transformations to the CS, not the object*

**Sequence of operations:**

- Set up a routine to draw the object in its “base” CS
- Call transformation routines to transform the CS
- Object drawn in transformed CS
OpenGL transformation example

```c
void drawHouse()
{
    glBegin(GL_LINE_LOOP);
    glVertex2i(0,0);
    glVertex2i(0,2);
    ...  
    glEnd();
}
```

```c
void drawTransformedHouse()
{
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(4.0, 4.0, 0.0);
    glScalef(0.5, 0.5, 1.0);
    drawHouse();
}
```

Draws basic house

Draws transformed house
Matrix Stacks

In many situations we want to save transformation matrices for use later

- Traversing hierarchical data structures
- Avoiding state changes when executing display lists

OpenGL maintains stacks for each type of matrix

- Access present type (as set by `glMatrixMode`) by

  ```
  glPushMatrix()
  glPopMatrix()
  ```
OpenGL matrix stack example

glLoadIdentity();
glPushMatrix();
glMultMatrixf(m1);
glPushMatrix();
glMultMatrixf(m2);
render chair2;
glPopMatrix();
glPushMatrix();
glMultMatrixf(m3);
render chair1;
glPopMatrix();
glPushMatrix();
render table;
glPopMatrix();
glPushMatrix();
render rug;
glPopMatrix();
render room;

M0

M1

M2

M3

M4

chair1

chair2

room

table

rug