3 Transformations

3.1 2D Transformations

Given a point cloud, polygon, or sampled parametric curve, we can use transformations for several purposes:

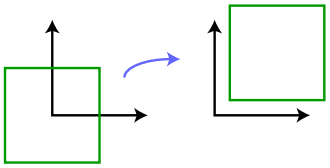
1. Change coordinate frames (world, window, viewport, device, etc).
2. Compose objects of simple parts with local scale/position/orientation of one part defined with regard to other parts. For example, for articulated objects.
3. Use deformation to create new shapes.
4. Useful for animation.

There are three basic classes of transformations:

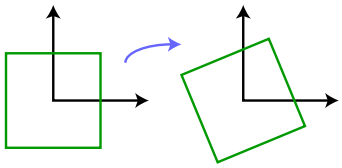
1. **Rigid body** - Preserves distance and angles.
   - Examples: translation and rotation.
2. **Conformal** - Preserves angles.
   - Examples: translation, rotation, and uniform scaling.
3. **Affine** - Preserves parallelism. Lines remain lines.
   - Examples: translation, rotation, scaling, shear, and reflection.

Examples of transformations:

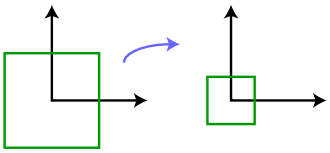
- **Translation** by vector $\vec{t}$: $\bar{p}_1 = \bar{p}_0 + \vec{t}$.



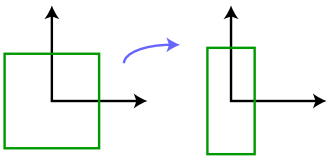
- **Rotation** counterclockwise by θ: $\bar{p}_1 = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \bar{p}_0$.



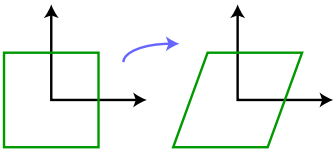
- **Uniform scaling** by scalar a: $\bar{p}_1 = \begin{bmatrix} a & 0 \\ 0 & a \end{bmatrix} \bar{p}_0$.



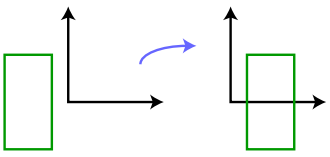
- **Nonuniform scaling** by a and b: $\bar{p}_1 = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} \bar{p}_0$.



- **Shear** by scalar h: $\bar{p}_1 = \begin{bmatrix} 1 & h \\ 0 & 1 \end{bmatrix} \bar{p}_0$.



- **Reflection** about the y-axis: $\bar{p}_1 = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \bar{p}_0$.



3.2 Affine Transformations

An **affine transformation** takes a point $\bar{p}$ to $\bar{q}$ according to $\bar{q} = F(\bar{p}) = A\bar{p} + \vec{t}$, a linear transformation followed by a translation. You should understand the following proofs.

- The inverse of an affine transformation is also affine, assuming it exists.

Proof:
Let $\bar{q} = A\bar{p} + \vec{t}$ and assume A^{-1} exists, i.e. $\det(A) \neq 0$.
Then $A\bar{p} = \bar{q} - \vec{t}$, so $\bar{p} = A^{-1}\bar{q} - A^{-1}\vec{t}$. This can be rewritten as $\bar{p} = B\bar{q} + \vec{d}$, where $B = A^{-1}$ and $\vec{d} = -A^{-1}\vec{t}$.

Note:
The inverse of a 2D linear transformation is

$$A^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}.$$

- Lines and parallelism are preserved under affine transformations.

Proof:
To prove lines are preserved, we must show that $\bar{q}(\lambda) = F(\bar{l}(\lambda))$ is a line, where $F(\bar{p}) = A\bar{p} + \vec{t}$ and $\bar{l}(\lambda) = \bar{p}_0 + \lambda\vec{d}$.

$$\begin{aligned} \bar{q}(\lambda) &= A\bar{l}(\lambda) + \vec{t} \\ &= A(\bar{p}_0 + \lambda\vec{d}) + \vec{t} \\ &= (A\bar{p}_0 + \vec{t}) + \lambda A\vec{d} \end{aligned}$$

This is a parametric form of a line through $A\bar{p}_0 + \vec{t}$ with direction $A\vec{d}$.

- Given a closed region, the area under an affine transformation $A\bar{p} + \vec{t}$ is scaled by $\det(A)$.

Note:

- Rotations and translations have $\det(A) = 1$.
- Scaling $A = \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}$ has $\det(A) = ab$.
- Singularities have $\det(A) = 0$.

Example:
The matrix $A = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$ maps all points to the x-axis, so the area of any closed region will become zero. We have $\det(A) = 0$, which verifies that any closed region's area will be scaled by zero.

- A composition of affine transformations is still affine.

Proof:
Let $F_1(\bar{p}) = A_1\bar{p} + \vec{t}_1$ and $F_2(\bar{p}) = A_2\bar{p} + \vec{t}_2$.
Then,

$$\begin{aligned} F(\bar{p}) &= F_2(F_1(\bar{p})) \\ &= A_2(A_1\bar{p} + \vec{t}_1) + \vec{t}_2 \\ &= A_2A_1\bar{p} + (A_2\vec{t}_1 + \vec{t}_2). \end{aligned}$$

Letting $A = A_2A_1$ and $\vec{t} = A_2\vec{t}_1 + \vec{t}_2$, we have $F(\bar{p}) = A\bar{p} + \vec{t}$, and this is an affine transformation.

3.3 Homogeneous Coordinates

Homogeneous coordinates are another way to represent points to simplify the way in which we express affine transformations. Normally, bookkeeping would become tedious when affine transformations of the form $A\bar{p} + \vec{t}$ are composed. With homogeneous coordinates, affine transformations become matrices, and composition of transformations is as simple as matrix multiplication. In future sections of the course we exploit this in much more powerful ways.

With homogeneous coordinates, a point $\bar{p}$ is augmented with a 1, to form $\hat{p} = \begin{bmatrix} \bar{p} \\ 1 \end{bmatrix}$.

All points $(\alpha\bar{p}, \alpha)$ represent the same point $\bar{p}$ for real $\alpha \neq 0$.

Given $\hat{p}$ in homogeneous coordinates, to get $\bar{p}$, we divide $\hat{p}$ by its last component and discard the last component.

Example:
The homogeneous points $(2, 4, 2)$ and $(1, 2, 1)$ both represent the Cartesian point $(1, 2)$. It's the orientation of $\hat{p}$ that matters, not its length.

Many transformations become linear in homogeneous coordinates, including affine transformations:

$$\begin{aligned} \begin{bmatrix} q_x \\ q_y \end{bmatrix} &= \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} p_x \\ p_y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \\ &= \begin{bmatrix} a & b & t_x \\ c & d & t_y \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} A & \vec{t} \end{bmatrix} \hat{p} \end{aligned}$$

To produce $\hat{q}$ rather than $\bar{q}$, we can add a row to the matrix:

$$\hat{q} = \begin{bmatrix} A & \vec{t} \\ \vec{0}^T & 1 \end{bmatrix} \hat{p} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \hat{p}.$$

This is linear! Bookkeeping becomes simple under composition.

Example:
$F_3(F_2(F_1(\bar{p})))$, where $F_i(\bar{p}) = A_i(\bar{p}) + \vec{t}_i$ becomes $M_3 M_2 M_1 \bar{p}$, where $M_i = \begin{bmatrix} A_i & \vec{t}_i \\ \vec{0}^T & 1 \end{bmatrix}$.

With homogeneous coordinates, the following properties of affine transformations become apparent:

- Affine transformations are associative.
  For affine transformations F_1, F_2, and F_3,

$$(F_3 \circ F_2) \circ F_1 = F_3 \circ (F_2 \circ F_1).$$

- Affine transformations are *not* commutative.
  For affine transformations F_1 and F_2,

$$F_2 \circ F_1 \neq F_1 \circ F_2.$$

3.4 Uses and Abuses of Homogeneous Coordinates

Homogeneous coordinates provide a different representation for Cartesian coordinates, and cannot be treated in quite the same way. For example, consider the midpoint between two points $\bar{p}_1 = (1,1)$ and $\bar{p}_2 = (5,5)$. The midpoint is $(\bar{p}_1 + \bar{p}_2)/2 = (3,3)$. We can represent these points in homogeneous coordinates as $\hat{p}_1 = (1,1,1)$ and $\hat{p}_2 = (5,5,1)$. Directly applying the same computation as above gives the same resulting point: $(3,3,1)$. However, we can *also* represent these points as $\hat{p}'_1 = (2,2,2)$ and $\hat{p}'_2 = (5,5,1)$. We then have $(\hat{p}'_1 + \hat{p}'_2)/2 = (7/2, 7/2, 3/2)$, which cooresponds to the Cartesian point $(7/3, 7/3)$. This is a different point, and illustrates that we cannot blindly apply geometric operations to homogeneous coordinates. The simplest solution is to **always convert homogeneous coordinates to Cartesian coordinates.** That said, there are several important operations that can be performed correctly in terms of homogeneous coordinates, as follows.

Affine transformations. An important case in the previous section is applying an affine transformation to a point in homogeneous coordinates:

$$\bar{q} = F(\bar{p}) = A\bar{p} + \vec{t} \quad (1)$$
$$\hat{q} = \hat{A}\hat{p} = (x', y', 1)^T \quad (2)$$

It is easy to see that this operation is correct, since rescaling $\hat{p}$ does not change the result:

$$\hat{A}(\alpha\hat{p}) = \alpha(\hat{A}\hat{p}) = \alpha\hat{q} = (\alpha x', \alpha y', \alpha)^T \quad (3)$$

which is the same geometric point as $\hat{q} = (x', y', 1)^T$

Vectors. We can represent a vector $\vec{v} = (x, y)$ in homogeneous coordinates by setting the last element of the vector to be zero: $\hat{v} = (x, y, 0)$. However, when adding a vector to a point, the point must have the third component be 1.

$$\hat{q} = \hat{p} + \hat{v} \quad (4)$$
$$(x', y', 1)^T = (x_p, y_p, 1) + (x, y, 0) \quad (5)$$

The result is clearly incorrect if the third component of the vector is not 0.

Aside:
Homogeneous coordinates are a representation of points in **projective geometry.**

3.5 Hierarchical Transformations

It is often convenient to model objects as hierarchically connected parts. For example, a robot arm might be made up of an upper arm, forearm, palm, and fingers. Rotating at the shoulder on the upper arm would affect all of the rest of the arm, but rotating the forearm at the elbow would affect the palm and fingers, but not the upper arm. A reasonable hierarchy, then, would have the upper arm at the root, with the forearm as its only child, which in turn connects only to the palm, and the palm would be the parent to all of the fingers.

Each part in the hierarchy can be modeled in its own local coordinates, independent of the other parts. For a robot, a simple square might be used to model each of the upper arm, forearm, and so on. Rigid body transformations are then applied to each part relative to its parent to achieve the proper alignment and pose of the object. For example, the fingers are positioned to be in the appropriate places in the palm coordinates, the fingers and palm together are positioned in forearm coordinates, and the process continues up the hierarchy. Then a transformation applied to upper arm coordinates is also applied to all parts down the hierarchy.

3.6 Transformations in OpenGL

OpenGL manages two 4×4 transformation matrices: the *modelview matrix*, and the *projection matrix*. Whenever you specify geometry (using `glVertex`), the vertices are transformed by the current modelview matrix and then the current projection matrix. Hence, you don't have to perform these transformations yourself. You can modify the entries of these matrices at any time. OpenGL provides several utilities for modifying these matrices. The modelview matrix is normally used to represent geometric transformations of objects; the projection matrix is normally used to store the camera transformation. For now, we'll focus just on the modelview matrix, and discuss the camera transformation later.

To modify the current matrix, first specify which matrix is going to be manipulated: use `glMatrixMode(GL_MODE` to modify the modelview matrix. The modelview matrix can then be initialized to the identity with `glLoadIdentity()`. The matrix can be manipulated by directly filling its values, multiplying it by an arbitrary matrix, or using the functions OpenGL provides to multiply the matrix by specific transformation matrices (`glRotate`, `glTranslate`, and `glScale`). Note that these transformations **right-multiply** the current matrix; this can be confusing since it means that you specify transformations in the reverse of the obvious order. Exercise: why does OpenGL right-multiply the current matrix?

OpenGL provides a **stacks** to assist with hierarchical transformations. There is one stack for the modelview matrix and one for the projection matrix. OpenGL provides routines for pushing and popping matrices on the stack.

The following example draws an upper arm and forearm with shoulder and elbow joints. The current modelview matrix is pushed onto the stack and popped at the end of the rendering, so, for example, another arm could be rendered without the transformations from rendering this arm affecting its modelview matrix. Since each OpenGL transformation is applied by multiplying a matrix on the right-hand side of the modelview matrix, the transformations occur in reverse order. Here, the upper arm is translated so that its shoulder position is at the origin, then it is rotated, and finally it is translated so that the shoulder is in its appropriate world-space position. Similarly, the forearm is translated to rotate about its elbow position, then it is translated so that the elbow matches its position in upper arm coordinates.

```
glPushMatrix();

glTranslatef(worldShoulderX, worldShoulderY, 0.0f);
drawShoulderJoint();
glRotatef(shoulderRotation, 0.0f, 0.0f, 1.0f);
glTranslatef(-upperArmShoulderX, -upperArmShoulderY, 0.0f);
drawUpperArmShape();

glTranslatef(upperArmElbowX, upperArmElbowY, 0.0f);
```



```
drawElbowJoint();
glRotatef(elbowRotation, 0.0f, 0.0f, 1.0f);
glTranslatef(-forearmElbowX, -forearmElbowY, 0.0f);
drawForearmShape();

glPopMatrix();
```

