CMSC 427: Chapter 3
Geometry for Computer Graphics

Reading: Chapt 2 in our text.
Overview:
- Affine Geometry
- Coordinate systems
- 2-d and 3-d Affine transformations
- Transformations in OpenGL

Geometry Issues

Geometry: Computer graphics involves the manipulation of objects in 3-space. Basic entities include:
- **Scalars**: Real numbers for lengths, displacements, coordinates, etc.
- **Point**: a location in space.
- **(Free) Vector**: a direction and length (but no location).
- **Other geometric objects**: Lines, line segments, planes, spheres, polygons and polyhedra, curves and surfaces.
- **Operations**: Geometric objects can be combined in various ways, e.g., given two points p and q, return the vector from p to q.
- **(Affine) Transformations**: Translation, rotation, scaling, etc.
- **Representations**: How are these entities represented internally?
Geometric Systems

Affine Geometry: scalars, points, and vectors.
Affine transformations: Translation, rotation, reflection, etc.
Euclidean Geometry: Adds the notion of dot product. Allows for other useful concepts like distance, angle, area, and volume.
Projective Geometry: Augments affine geometry by adding the notion of points at infinity.
Projective Transformations: Adds perspective projection.

All Encompassing Geometric System? Unfortunately, some elements of projective geometry are inconsistent with affine geometry. Thus, we will use different systems in different contexts.

Overview

Affine Geometry
- Scalars, points, vectors
- Linear, affine, and convex combinations
Coordinate systems and Homogeneous coordinates
Euclidean Geometry
- Dot product
- Cross Product
2-d and 3-d Affine transformations
- Composing affine transformations
Transformations in OpenGL
**Affine Geometry Basics**

**Basic Entities:** scalars, points, (free) vectors.

**Points vs. vectors:** Even though they are represented the same way (as a coordinate vector) points are different from vectors.

- **Vector:** Has length and direction.
  
  Examples: Velocity, heading (direction of motion), axis of rotation, camera view direction.

- **Point:** Identifies a specific location.
  
  Examples: Vertex, object position, center of rotation, camera location.

**Naming Conventions:**

- **Scalars:** α, β, γ (Greek)
- **Points:** p, q, r (bold face)
- **Free Vectors:** u, v, w (bold face)

**Affine Geometry Operations**

**Legal Operations:**

- **Vector ops from Linear Algebra:**
  
  - scalar · vector → vector
  - vector + vector → vector

- **Point/Vector operations:**
  
  - point + vector → point
  - point - point → vector

**Illegal:** There is no "point + point", "scalar · point", nor "vector - point".
**Combinations**

**Linear combination** is a vector \( \alpha u + \beta v \), where \( u \) and \( v \) are vectors and \( \alpha \) and \( \beta \) are scalars.

**Affine combination** is a point \( \alpha p + \beta q \), where \( p \) and \( q \) are points and \( \alpha \) and \( \beta \) are scalars, where \( \alpha + \beta = 1 \).

(Equivalent to the point/vector sum: \( p + \beta(q-p) \).)

**Convex combination**: An affine combination where in addition, \( 0 \leq \alpha, \beta \leq 1 \).

\[ \frac{1}{2}p + \frac{1}{2}q \]

**Combinations and Affine Structures**

Combinations are useful for generating/representing objects.

**Line**:
- Point/Vector Sum: \( p + \alpha u \), where \(-\infty < \alpha < +\infty\).
- Affine combination: \( \alpha p + \beta q \), where \( \alpha + \beta = 1 \).

**Line Segment**:
- Point/Vector Sum: \( p + \alpha u \), where \( 0 \leq \alpha \leq 1 \).
- Convex combination: \( \alpha p + \beta q \), where \( \alpha + \beta = 1, 0 \leq \alpha, \beta \leq 1 \).
Combinations and Affine Structures

Combinations work for higher dimensional objects.

Plane:
Point/Vector Sum: \( p + \alpha u + \beta v \),
where \(-\infty < \alpha, \beta < +\infty\).
Affine combination: \( \alpha p + \beta q + \gamma r \),
where \(\alpha + \beta + \gamma = 1\).

Parallelogram:
Point/Vector Sum: \( p + \alpha u + \beta v \),
where \(0 \leq \alpha, \beta \leq 1\).

Triangle:
Convex combination: \( \alpha p + \beta q + \gamma r \),
where \(\alpha + \beta + \gamma = 1, 0 \leq \alpha, \beta, \gamma \leq 1\).

In-Class Exercise

Consider the triangle \( \triangle pqr \).

(a) Express the midpoints \( p', q', r' \) as affine combinations of \( p, q, \) and \( r \).

(b) Express the vertices of the enclosing triangle \( \triangle p'q'r' \) as affine combinations of \( p, q, \) and \( r \).
Overview

Affine Geometry
- Scalars, points, vectors
- Linear, affine, and convex combinations

Coordinate systems and Homogeneous coordinates

Euclidean Geometry
- Dot product
- Cross Product

2-d and 3-d Affine transformations
- Composing affine transformations

Transformations in OpenGL

---

Parametric Representations

Parametric Representation:
Combinations induce a natural coordinate system on these structures.

Plane: The point $p + \alpha u + \beta v$ can be identified by the coordinates $(\alpha, \beta)$.

Triangle: The point $\alpha p + \beta q + \gamma r$ can be identified by the coordinates $(\alpha, \beta, \gamma)$, where $\alpha + \beta + \gamma = 1$, called the barycentric coordinates.

Examples: $(\alpha, \beta, \gamma)$

$(1, 0, 0) = p$
$(0, 1, 0) = q$
$(0, \frac{1}{2}, \frac{1}{2}) =$ midpoint of edge $qr$.
$(1/3, 1/3, 1/3) =$ centroid
(or center of mass).
Homogeneous Coordinates (for Affine Geometry)

How do we name objects?

Coordinate Frame: in 2-space consists of an origin p and 2-linearly independent basis vectors: \( F = \langle v_1, v_2; p \rangle \).
- A point can be uniquely represented as \( \alpha v_1 + \beta v_2 + p \), for some \( \alpha, \beta \).
- A vector can be uniquely represented as \( \alpha v_1 + \beta v_2 \), for some \( \alpha, \beta \).

Coordinate Axiom: Given a point \( p \), define \( 1 \cdot p = p \), and \( 0 \cdot p = \vec{0} \), where \( \vec{0} \) denotes the zero vector.
- A point can be uniquely represented as \( \alpha v_1 + \beta v_2 + 1 \cdot p \).
- A vector can be uniquely represented as \( \alpha v_1 + \beta v_2 + 0 \cdot p \).

Example: Consider the frame \( F = \langle v_1, v_2; p \rangle \).
- \( q = 3v_1 + 1v_2 + 1p \) and so \( q[F] = [3, 1, 1]^F \).
- \( r = -1v_1 + 2v_2 + 1p \) and so \( r[F] = [-1, 2, 1]^F \).
- \( u = 1v_1 + 1v_2 + 0p \) and so \( u[F] = [1, 1, 0]^F \).
- \( w = 1v_1 - 2v_2 + 0p \) and so \( w[F] = [1, -2, 0]^F \).

Homogeneous Coordinates (for Affine Geometry)

Homogeneous Coordinates: We can represent both points and vectors using a consistent notation.
- For a point: \([\alpha, \beta, 1]^F\) means \( \alpha v_1 + \beta v_2 + 1 \cdot p \).
- For a vector: \([\alpha, \beta, 0]^F\) means \( \alpha v_1 + \beta v_2 + 0 \cdot p \).

Example: Consider the frame \( F = \langle v_1, v_2; p \rangle \).
- \( q = 3v_1 + 1v_2 + 1p \) and so \( q[F] = [3, 1, 1]^F \).
- \( r = -1v_1 + 2v_2 + 1p \) and so \( r[F] = [-1, 2, 1]^F \).
- \( u = 1v_1 + 1v_2 + 0p \) and so \( u[F] = [1, 1, 0]^F \).
- \( w = 1v_1 - 2v_2 + 0p \) and so \( w[F] = [1, -2, 0]^F \).
Homogeneous Coordinates in Higher Dimensions

Homogeneous coordinates work in all dimensions. Generally, in dimension $d$, we have $d+1$ coordinates.
- The first $d$ are for the basis vectors.
- The last coordinate is 1 for points and 0 for vectors.

Example:
$$ q = 3v_1 - 1v_2 + 2v_3 + 1p $$
and so $q[F] = [3, -1, 2, 1]_F$. 

Local Frames and the Standard Frame

Local Frames: It is common to use many frames.
- Each object is defined relative to a natural local frame.
- Each local frame is defined relative to a more global frame.

Example:
- Hand: relative to the shoulder frame.
- Shoulder: relative to the body frame.
- Body: relative to room frame.
- Room: relative to building frame.
- ... (where does this end?)

Standard Frame: $S = \langle u_1, u_2, \ldots, u_d; O \rangle$
- Origin point: $O$
- Bases vectors: Orthonormal unit vectors $u_1, u_2, \ldots, u_d$.
- Everything is defined relative to $S$. 

Image source: http://www.kymne.com
Properties of Homogeneous Coordinates

**Theorem**: A linear combination of points/vectors is **legal** if and only if the corresponding operation on homogeneous coordinate vectors results in a **last coordinate** of **0** (vector result) or **1** (point result).

**Legal Examples**: (Consider the last coordinate only)
- scalar · vector → vector: \( \alpha [\ldots, 0] = [\ldots, 0] \)
- vector + vector → vector: \([\ldots, 0] + [\ldots, 0] = [\ldots, 0] \)
- point + vector → point: \([\ldots, 1] + [\ldots, 0] = [\ldots, 1] \)
- point - point → vector: \([\ldots, 1] - [\ldots, 1] = [\ldots, 0] \)
- affine combination (\( \alpha p + \beta q \)): \( \alpha [\ldots, 1] + \beta [\ldots, 1] = [\ldots, \alpha + \beta], \) valid if \( \alpha + \beta = 1 \)

**Illegal Examples**:
- point + point → \( \cdot \) \([\ldots, 1] + [\ldots, 1] = [\ldots, 2] \) → Invalid!
- vector - point → \( \cdot \) \([\ldots, 0] - [\ldots, 1] = [\ldots, -1] \) → Invalid!
- scalar · point → \( \cdot \) \( \alpha [\ldots, 1] = [\ldots, \alpha] \) → Invalid!

Notational Conventions

**Row or Column Vectors?** In matrix contexts will assume that points and vectors will be written as *(homogeneous)* column vectors.

\[
\begin{pmatrix}
a_{11} & a_{12} & a_{13} \\
a_{21} & a_{22} & a_{23} \\
a_{31} & a_{32} & a_{33}
\end{pmatrix}
\begin{pmatrix}
x_1 \\
x_2 \\
x_3
\end{pmatrix}
=
\begin{pmatrix}
b_1 \\
b_2 \\
b_3
\end{pmatrix}
\]

**Notational Conventions**: To simplify notation, we will often:
- Write column vectors as row vectors: \( \begin{bmatrix} x \\ y \\ z \end{bmatrix} \) (rather than \( \begin{pmatrix} x \\ y \\ z \end{pmatrix} \))
- Use (\( \ldots \)) for *standard coordinates*.
- Use \( [\ldots] \) for *homogeneous coordinates*.
  - E.g. \( p = (1, -2, 4) \) is the same as \( p = [1, -2, 4, 1] \)
- Use (\( \ldots \)) for *other tuples*. (E.g. coordinate frames.)
Other Uses for Homogeneous Coordinates

“I heard that you divide by the last coordinate”:
You may have seen homogeneous coordinates before. They are used in two different contexts:
- **Affine geometry**: last coordinate is 0 or 1.
- **Projective geometry**: divide through by the last coordinate.
  \[ [x, y, z, w] \rightarrow [x/w, y/w, z/w, 1] \]. (We will discuss this use of homogeneous coordinates later.)

Overview

**Affine Geometry**
- Scalars, points, vectors
- Linear, affine, and convex combinations

**Coordinate systems and Homogeneous coordinates**

**Euclidean Geometry**
- Dot product
- Cross Product

2-d and 3-d Affine transformations
- Composing affine transformations

Transformations in OpenGL
Euclidean Geometry: Dot Product

Dot Product: Let \( \mathbf{u} = (u_1, \ldots, u_d) \) and \( \mathbf{v} = (v_1, \ldots, v_d) \). Recall that:
\[
(\mathbf{u} \cdot \mathbf{v}) = u_1 v_1 + u_2 v_2 + \ldots + u_d v_d.
\]
Length: \( ||\mathbf{u}|| = (\mathbf{u} \cdot \mathbf{u})^{1/2} \).
Normalizing a vector to unit length: \( \hat{\mathbf{u}} = \frac{\mathbf{u}}{||\mathbf{u}||} \).
Angle between vectors: \( \cos^{-1} \left( \frac{\mathbf{u} \cdot \mathbf{v}}{||\mathbf{u}|| \cdot ||\mathbf{v}||} \right) = \cos^{-1}(\hat{\mathbf{u}} \cdot \hat{\mathbf{v}}) \).
Orthogonality: \( \mathbf{u} \) and \( \mathbf{v} \) are orthogonal if \( (\mathbf{u} \cdot \mathbf{v}) = 0 \).
Orthogonal Decomposition: Given \( \mathbf{u} \) and \( \mathbf{v} \), express \( \mathbf{u} = \mathbf{u}_1 + \mathbf{u}_2 \), where \( \mathbf{u}_1 \) is parallel to \( \mathbf{v} \), and \( \mathbf{u}_2 \) is orthogonal to \( \mathbf{v} \).
\[
\mathbf{u}_1 = \frac{(\mathbf{u} \cdot \mathbf{v})}{(\mathbf{v} \cdot \mathbf{v})} \mathbf{v}, \quad \mathbf{u}_2 = \mathbf{u} - \mathbf{u}_1.
\]

Cross Product

Given two vectors \( \mathbf{u} \) and \( \mathbf{v} \) in 3-space, recall that their cross product is:
\[
\mathbf{u} \times \mathbf{v} = \begin{vmatrix}
\mathbf{e}_x & \mathbf{e}_y & \mathbf{e}_z \\
\mathbf{u}_x & \mathbf{u}_y & \mathbf{u}_z \\
\mathbf{v}_x & \mathbf{v}_y & \mathbf{v}_z
\end{vmatrix}
\]
Geometric interpretation:
- \( \mathbf{u} \times \mathbf{v} \) is orthogonal to \( \mathbf{u} \) and \( \mathbf{v} \), and is directed according to the right-hand rule.
- \( ||\mathbf{u} \times \mathbf{v}|| \) is the area of the parallelogram.
Skew symmetric: \( \mathbf{u} \times \mathbf{v} = -(\mathbf{v} \times \mathbf{u}) \).
Non-associative: \( \mathbf{u} \times (\mathbf{v} \times \mathbf{w}) \neq (\mathbf{u} \times \mathbf{v}) \times \mathbf{w} \).
Example: Brightness of a Triangle

**Brightness:** For a triangle $T = \triangle p_0 p_1 p_2$ in 3-space, a light source at a point $s$, and a point $q$ on $T$, the brightness of $q$ is proportional to the cosine of the angle between the $T$'s normal vector and the vector from $q$ to $s$.

**Normal:** The normal vector $\mathbf{n}$ for any point on $T$ is orthogonal to the vectors $\mathbf{v}_1 = p_1 - p_0$ and $\mathbf{v}_2 = p_2 - p_0$:

$$\mathbf{n} = \mathbf{v}_1 \times \mathbf{v}_2.$$  

**Light Vector:** The vector from $q$ to $s$ is:

$$\mathbf{l} = s - q.$$  

**Cosine:** The brightness is proportional to:

$$\frac{\mathbf{n} \cdot \mathbf{l}}{||\mathbf{n}|| \cdot ||\mathbf{l}||} = \frac{\mathbf{n} \cdot \mathbf{l}}{\sqrt{(\mathbf{n} \cdot \mathbf{n})(\mathbf{l} \cdot \mathbf{l})}}.$$  

Example: Normal of a Vertex in a Mesh

**Application:** We have a mesh of triangles. At each vertex we want to estimate the normal vector by averaging the normals of the incident triangles.

$$\mathbf{n} = \frac{1}{k} (\mathbf{n}_1 + \mathbf{n}_2 + \cdots + \mathbf{n}_k) = \frac{1}{k} \sum_{i=1}^{k} \mathbf{n}_i.$$  

**Weighted Average:** (Better) weight faces according to their areas. Let $a_i$ = area of triangle $i$, and let $A = \text{total area}$.

$$\mathbf{n} = \frac{1}{A} (a_1 \mathbf{n}_1 + a_2 \mathbf{n}_2 + \cdots + a_k \mathbf{n}_k) = \frac{1}{A} \sum_{i=1}^{k} a_i \mathbf{n}_i.$$
Overview

Affine Geometry
- Scalars, points, vectors
- Linear, affine, and convex combinations

Coordinate systems and Homogeneous coordinates

Euclidean Geometry
- Dot product
- Cross Product

2-d and 3-d Affine transformations
- Composing affine transformations

Transformations in OpenGL

Transformations

Need transformations in graphics for:
- Constructing scenes: hierarchical, articulated joints, animation.
- Viewing: perspective, object to world, world to screen, ...
- Navigating: pan, zoom, tilt, walk, ...
- Manipulating: picking, dragging, placing, ...

Types of common transformations in graphics:
- Rigid transformations: preserve lengths and angles.
- Affine transformations: preserve parallel lines.
Affine Transformations

**Affine Transformation**: A transformation that preserves affine combinations:

\[ T(\alpha p + \beta q) = \alpha T(p) + \beta T(q). \]

**Examples**:
- **Translation**:
- **Rotation**:
- **(Uniform) Scaling**:
- **(Non-uniform) Scaling**:
- **Reflection**:
- **Shearing**:

---

Transformation Matrices in 2-Space

**Theorem**: Let \( S = (u_1, u_2; O) \) be the standard coordinate frame in 2-space. Consider a 2-d affine transformation that maps \( u_i \) to \( v_i \) and maps \( O \) to \( p \). Then the corresponding **transformation matrix** is:

\[
\begin{bmatrix}
    v_{11} & v_{12} & p_1 \\
    v_{21} & v_{22} & p_2 \\
    0 & 0 & 1
\end{bmatrix}
\]

**Example**: Consider the **translation transformation** that maps the origin to the point \( t = (t_x, t_y) \). The standard basis vectors are unchanged. So the matrix is:

\[
\begin{bmatrix}
    1 & 0 & t_x \\
    0 & 1 & t_y \\
    0 & 0 & 1
\end{bmatrix}
\]
Transformation Matrices in 2-Space

Rotation: by angle \( \theta \) counterclockwise about the origin.
- The x-basis vector is mapped to \((\cos \theta, \sin \theta)\).
- The y-basis vector is mapped to \((-\sin \theta, \cos \theta)\).
- The origin is unchanged.

so the transformation matrix is:
\[
\begin{bmatrix}
\cos \theta & -\sin \theta & 0 \\
\sin \theta & \cos \theta & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

Scale: x-axis by \( s_x \) and y-axis by \( s_y \)
- The x-basis vector is mapped to \((s_x, 0)\).
- The y-basis vector is mapped to \((0, s_y)\).
- The origin is unchanged.

so the transformation matrix is:
\[
\begin{bmatrix}
s_x & 0 & 0 \\
0 & s_y & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

Composing Transformations

Transformations can be composed through matrix multiplication.

Efficient: allows us to perform a series of transformations with a single matrix/vector multiplication.

Order: Because we post-multiply, the order of evaluation is from right to left:
\[
M_3 \cdot M_2 \cdot M_1 \cdot p \quad \rightarrow \quad (M_3 \cdot (M_2 \cdot (M_1 \cdot p)))
\]

Remember: Matrix multiplication is associative \(((AB)C = A(BC))\), but not commutative \((AB \neq BA)\). So the order in which matrices are listed does matter.
Composing Transformations

Example: Rotate the plane counterclockwise by angle \( \theta \) about the point \( t = (t_x, t_y) \).

Step 1: Translate \( t \) to origin.
\[
M_1 = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix}
\]

Step 2: Rotate by \( \theta \) about origin.
\[
M_2 = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}
\]

Step 3: Translate origin back to \( t \).
\[
M_3 = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}
\]

Final Transformation: \( M_3 \cdot M_2 \cdot M_1 \rightarrow M \).
\[
M = \begin{bmatrix} \cos \theta & -\sin \theta & t_x(1-\cos \theta) + t_y\sin \theta \\ \sin \theta & \cos \theta & t_y(1-\cos \theta) + t_x\sin \theta \\ 0 & 0 & 1 \end{bmatrix}
\]
Transformation Matrices in 3-Space

Theorem: Let $S = \langle u_1, u_2, u_3; O \rangle$ be the standard coordinate frame in 3-space. Consider a 3-d affine transformation that maps $u_i$ to $v_i$ and maps $O$ to $p$. Then the corresponding transformation matrix is:

$$
\begin{bmatrix}
v_{11} & v_{21} & v_{31} & p_1 \\
v_{12} & v_{22} & v_{32} & p_2 \\
v_{13} & v_{23} & v_{33} & p_3 \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

Example: Consider the translation transformation that maps the origin to the point $t = (t_x, t_y, t_z)$. The standard basis vectors are unchanged. So the matrix is:

$$
\begin{bmatrix}
1 & 0 & 0 & t_x \\
0 & 1 & 0 & t_y \\
0 & 0 & 1 & t_z \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

Transformation Matrices in 3-Space

Rotation: by angle $\theta$ counterclockwise about:

- **x-axis**
  $$
  \begin{bmatrix}
  1 & 0 & 0 & 0 \\
  0 & \cos \theta & -\sin \theta & 0 \\
  0 & \sin \theta & \cos \theta & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  $$

- **y-axis**
  $$
  \begin{bmatrix}
  \cos \theta & 0 & \sin \theta & 0 \\
  0 & 1 & 0 & 0 \\
  -\sin \theta & 0 & \cos \theta & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  $$

- **z-axis**
  $$
  \begin{bmatrix}
  \cos \theta & -\sin \theta & 0 & 0 \\
  \sin \theta & \cos \theta & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 0 & 1
  \end{bmatrix}
  $$

Scale: the x, y, and z axes by factors $s_x, s_y, s_z$, respectively:

$$
\begin{bmatrix}
s_x & 0 & 0 & 0 \\
0 & s_y & 0 & 0 \\
0 & 0 & s_z & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
$$

Reflection: is a special case of scaling by a negative scale factor.
Transformation Matrices in 3-Space

Shearing: Consider a shear that fixes the x,y-plane and maps the unit z-basis vector to \( h = (h_x, h_y, 1) \):

\[
\begin{bmatrix}
1 & 0 & h_x \\
0 & 1 & h_y \\
0 & 0 & 1
\end{bmatrix}
\]

Composing Transformations

Example: Derive a transformation that scales points by a factor of 2, but with a center of expansion at the point \( p = (1, 1, 0) \).

Approach: Compose the following transformations:
- Translate \( p \) to the origin (by \((-1, -1, 0))\).
- Scale by factor 2 about origin.
- Translate origin by to \( p \) (by \((1, 1, 0))\).
Composing Transformations

**Step 1:** Translate p to origin (by (-1, -1, 0)).

\[
M_1 = \begin{bmatrix}
1 & 0 & 0 & -1 \\
0 & 1 & 0 & -1 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

**Step 2:** Scale by factor 2 about origin.

\[
M_b = \begin{bmatrix}
2 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 \\
0 & 0 & 2 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

**Step 3:** Translate origin back to p (by (1, 1, 0)).

\[
M_3 = \begin{bmatrix}
1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

**Final Matrix:** \( M_3 \cdot M_2 \cdot M_1 \rightarrow M \) (Remember: right to left.)

\[
\begin{bmatrix}
1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
2 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 \\
0 & 0 & 2 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

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

**Verify:** Under this transformation, the center of expansion should be unchanged and the origin should be mapped to (-1,-1,0).

\[
\begin{bmatrix}
2 & 0 & 0 & -1 \\
0 & 2 & 0 & -1 \\
0 & 0 & 2 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
= \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
2 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 \\
0 & 0 & 2 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 & 0 & -1 \\
0 & 1 & 0 & -1 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
= \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Overview

Affine Geometry
- Scalars, points, vectors
- Linear, affine, and convex combinations

Coordinate systems and Homogeneous coordinates

Euclidean Geometry
- Dot product
- Cross Product

2-d and 3-d Affine transformations
- Composing affine transformations

Transformations in OpenGL

Transformations in OpenGL

OpenGL provides support for transformations. There are a number of contexts in which transformations are used.

Modelview Mode (GL_MODELVIEW): Used for
- transforming objects in the scene and
- changing the coordinates into a form that is easier for OpenGL to deal with.

Projection Mode (GL_PROJECTION): Used for projecting objects onto the 2d image plane.

Texture Mode (GL_TEXTURE): Used for transforming (wrapping) textures onto surfaces of your objects.
Transformations in OpenGL

OpenGL has three matrix stacks. Matrix operations apply to the current stack.

To specify which stack you want to manipulate, use:

```
glMatrixMode ( mode );
```

where (mode) is either:
- GL_MODELVIEW
- GL_PROJECTION
- GL_TEXTURE.

Modelview mode is the most common (and the default), so it is common to switch back to Modelview mode.

```
glMatrixMode ( GL_PROJECTION );
// ... do something in Projection mode ...
glMatrixMode ( GL_MODELVIEW );
```

OpenGL Matrix Stack Operations

Matrix Stack: Each matrix mode has a stack of matrices. All operations apply to the active matrix at the top of the stack.

**glLoadIdentity ()**
- Set the active matrix to identity.

**glLoadMatrixf ( GLfloat m )**
**glLoadMatrixd ( GLdouble* m )**
- Set the 16 values of the active matrix to those specified by m.

**glmMultMatrixf ( GLfloat m )**
**glmMultMatrixd ( GLdouble* m )**
- Multiplies the active matrix by m.

Beware: OpenGL assumes column-major order, whereas C++ assumes row-major order.
OpenGL Matrix Stack Operations

**Stack Operations**: It is possible to save the current matrix state by pushing and popping.

- **glPushMatrix()**
  - Make a copy of the active matrix and push it on the stack.

- **glPopMatrix()**
  - Pop the active matrix off the stack.

**Example**:

1. `glLoadIdentity();`
2. `glLoadMatrixf(A);`
3. `glPushMatrix();`
4. `glMultMatrixf(B);`
5. `glPushMatrix();`
6. `glPopMatrix();`

OpenGL Transformation Operations

**Standard Transformations**: Rather than specifying your own matrix, it is more common to use one of the standard transformations. Below "GLtype" is either "GLfloat" or "GLdouble".

- **glTranslate(fd)** (GLtype x, GLtype y, GLtype z)
  - Multiply the active matrix by the translation matrix that translates by (x, y, z).

- **glRotate(fd)** (GLtype angle, GLtype x, GLtype y, GLtype z)
  - Multiply the active matrix by the 3-d rotation matrix that rotates CCW by angle degrees about the axis from (0,0,0) to (x, y, z).

- **glScale(fd)** (GLtype sx, GLtype sy, GLtype sz)
  - Multiply the active matrix by the scale matrix that scales x by sx, y by sy, and z by sz.
Applying Transformations

The active transformation matrix is **automatically** applied to all drawing. The typical order is:
- save the current matrix state (push)
- apply the desired transformation matrix to active matrix
- draw your object(s)
- restore the matrix state (pop)

**Example:** Draw a 2x2 rectangle centered at the origin, rotated about the origin by 20 degrees CCW.

```gl
glPushMatrix();
glRotatef(20, 0, 0, 1);
glRectf(-1, -1, 1, 1);
glPopMatrix();
```

Remember that OpenGL works 3-d. Rotation in 2-d is achieved by rotation about the z-axis.

Order of Transformations

**Order of Evaluation:**
- Vertices are transformed by the current model-view matrix, you must specify transformations before drawing.
- Matrices are **post-multiplied**, so the transformations are applied to in the reverse of the order in which they are given.

**Example:**
- Given a procedure `drawFlag()`, which draws a flag at the origin. We want to (1) **rotate** the flag by 30 degrees CW about the origin and (2) **translate** to (3,2).

```gl
glPushMatrix();
glTranslatef(3, 2, 0);
glRotatef(-30, 0, 0, 1);
drawFlag();
glPopMatrix();
```

Note reversal
2D Projection and Viewport Transformation

**Projection Transformation**: Maps points from your idealized drawing area to a rectangular **viewport**.

**Viewport Transformation**: Maps points from the viewport a region of your graphics window. (Usually all of it, but you can specify any portion you like.)

---

![Diagrams showing projection and viewport transformations]

---

**2D Projection Transformation**

**Projection Transformation**: is set up in Projection mode.

```c
glMatrixMode ( GL_PROJECTION );
gLoadIdentity ( );
gluOrtho2D ( left, right, bottom, top );
glMatrixMode ( GL_MODELVIEW );
```

---

![Diagrams showing projection and viewport transformations]

---

Chapter 3, Slide: 47

Chapter 3, Slide: 48

Copyright © D. M. Mount and A. Varshney
Viewport Transformation

Viewport Transformation: is set up using glViewport.

\[ \text{glViewport}(x, y, \text{wid}, \text{hgt}); \]
- \((x, y)\) are the lower left corner of the viewport (in pixels).
- \((\text{wid}, \text{hgt})\) are the width and height of the viewport (in pixels).
- Use \text{glViewport}(0, 0, \text{winWid}, \text{winHgt}) to use full window.
- A good place to put this is in your reshape callback.

Your drawing area (world coordinates)

Clipped

Viewport (screen coordinates)

Summary

Summary:
- Affine Geometry
  - Scalars, points, vectors
  - Linear, affine, and convex combinations
- Coordinate systems
- 2-d and 3-d Affine transformations
  - Composing affine transformations
- Transformations in OpenGL

What's Next?
- Scan conversion