A point with Cartesian coordinates \((x, y, z)\) is taken to be a column vector (3 by 1 matrix) \(p = (x \ y \ z)^T\) for purposes of applying transformations. A linear transformation \(L\) on \(\mathbb{R}^3\) is represented by a 3 by 3 matrix \(A\). In fact, there is a 1-1 correspondence between linear transformations and matrices, so that \(L\) and \(A\) are often used interchangeably. The transformed point \(p' = L(p) = Ap\) is computed as a matrix-vector product

\[
p' = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = Ap = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}.
\]

It is easily shown that \(L(\alpha p) = \alpha L(p)\) and \(L(p + q) = L(p) + L(q)\), thus defining \(L\) as linear. Conversely, to construct the matrix \(A\) associated with some linear operator \(L\), the trick is to take the columns of \(A\) to be \(L(e_j)\), where \(e_j\)'s are the standard basis vectors.
**Defn** An *affine transformation* is a linear transformation followed by a translation — a transformation of the form

\[ T(p) = Tp = Ap + t, \]

where \( A \) is a matrix, and \( t \) is a translation vector. In the case of a transformation \( T \) from \( \mathbb{R} \) to \( \mathbb{R} \), \( T \) is a linear (degree-1) polynomial (but not a linear transformation unless \( t = 0 \)).

An important property of an affine transformation, crucial for computer graphics, is that it maps lines to lines and hence planes to planes. This means that we can transform a line segment or triangle by transforming its vertices. More precisely, if \( p \) is a convex combination of vertices, then \( Tp \) is a convex combination of the transformed vertices with the same coefficients. Thus, for example, the midpoint of a line segment is the midpoint of the transformed endpoints. A projective transformation (perspective projection) also maps lines to lines but does not preserve ratios of distances.
The fact that all transformations in the OpenGL vertex pipeline map triangles to triangles is what allows fragment depths and color intensities to be computed by linear interpolation from vertex values.

The affine transformations of primary interest are translations and the following four linear operators.

1. Scaling
2. Rotation
3. Reflection
4. Orthogonal Projection
A (pure) translation is an affine transformation in which the linear part is the identity matrix \( I \):

\[
T \begin{pmatrix} x \\ y \\ z \end{pmatrix} = I \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \end{bmatrix}
\]
Scaling

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

Note that scaling an object changes not only its size, but also its distance from the origin.
Defn A *uniform scaling* operator is a scalar times the identity matrix. It scales all components by the same factor.

Problem Uniformly scale the size of a triangle by 3 without moving vertex $p_1 = (1, 1, 0)$.

Solution

1. Translate by $-p_1$
2. Scale by $3I$
3. Invert the translation: translate by $p_1$

Note that $p_1$ is not altered by the sequence of operations:

\[
p_1 \rightarrow p_1 - p_1 = 0 \rightarrow S0 = 0 \rightarrow 0 + p_1 = p_1
\]
Scaling Relative to a Point Example

Translate

Scale

R. J. Renka  
Transformation Algebra
Converting polar coordinates to Cartesian coordinates, we have

\[ x = r \cos(\phi), \quad y = r \sin(\phi) \]

so that

\[ x' = r \cos(\theta + \phi) = r(\cos \theta \cos \phi - \sin \theta \sin \phi) = \cos \theta x - \sin \theta y \]

and

\[ y' = r \sin(\theta + \phi) = r(\cos \theta \sin \phi + \sin \theta \cos \phi) = \cos \theta y + \sin \theta x. \]

Thus

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

for \( C = \cos \theta \) and \( S = \sin \theta \).
**Defn** An *orthogonal matrix* is a square matrix $R$ with orthonormal columns: $r_i^T r_j = \delta_{ij}$ for $R = [r_1 \ r_2 \ \ldots \ r_n]$; i.e., $R^T R = I$.

An orthogonal matrix $R$ preserves size and shape because it preserves inner products (and hence Euclidean norms and angles): $(Ru)^T (Rv) = u^T R^T Rv = u^T v$.

**Defn** A rotation is an orthogonal matrix with determinant 1. A planar rotation is a matrix of the form

$$R = \begin{bmatrix} C & -S \\ S & C \end{bmatrix},$$

where $C^2 + S^2 = 1$. $R$ rotates CCW through angle $\theta = \cos^{-1}(C) = \sin^{-1}(S)$.

To see that $R$ is orthogonal, note that

$$R^T R = \begin{bmatrix} C & S \\ -S & C \end{bmatrix} \begin{bmatrix} C & -S \\ S & C \end{bmatrix} = \begin{bmatrix} C^2 + S^2 & -CS + SC \\ -SC + CS & S^2 + C^2 \end{bmatrix} = I.$$
Problem Construct the affine transformation that rotates CCW through angle $\theta$ about a point $p$ in the plane.

Solution

1. Translate by $-p$
2. Rotate through angle $\theta$ about the origin
3. Translate by $p$

The fixed point of the operator is $p$:

\[
p \quad \rightarrow \quad p - p = 0 \quad \rightarrow \quad R0 = 0 \quad \rightarrow \quad 0 + p = p\]

Defn A *rigid body transformation* is an affine transformation in which the matrix is orthogonal. It preserves shape and size, but not location or orientation.
Rotations in $\mathbb{R}^3$

There are simple formulas for the axis rotations. To rotate about the $x$ axis, we leave the $x$-component unaltered, and apply a planar rotation to the $(y,z)$ pair:

$$ R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & C & -S \\ 0 & S & C \end{bmatrix} \implies R_x \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ Cy - Sz \\ Sy + Cz \end{bmatrix}, $$

where $C^2 + S^2 = 1$. Similarly,

$$ R_y = \begin{bmatrix} C & 0 & S \\ 0 & 1 & 0 \\ -S & 0 & C \end{bmatrix}, \quad R_z = \begin{bmatrix} C & -S & 0 \\ S & C & 0 \\ 0 & 0 & 1 \end{bmatrix} $$

Note that the patterns are consistent with the cyclical ordering of components, corresponding to pairs $(y,z)$, $(z,x)$, and $(x,y)$.
Rotations in $\mathbb{R}^3$ continued

We have used the convention that the matrices rotate CCW as viewed from the positive end of the axis toward the origin. In order to verify this, try a simple example like a 90-degree rotation applied to one of the standard basis vectors (axis-aligned unit vectors).

For $\theta = \pi/2$ we have $C = \cos \theta = 0$ and $S = \sin \theta = 1$ so that

$$R_y = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}, \quad \Rightarrow \quad R_y e_3 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = e_1$$

corresponding to a CCW rotation in a right-handed coordinate system. The inverse matrix $R_y^T$ maps $e_3$ to $-e_1$. Note that $A e_j$ is the $j^{th}$ column of $A$.

It is easily verified that the coordinate-axis rotations are orthogonal, and that the product of orthogonal matrices is orthogonal. In fact, by Euler’s theorem, any product of rotations is a rotation about some Euler pole.
Defn A reflection \( R \) is a symmetric orthogonal matrix, and hence involutory (equal to its inverse): \( R^2 = I \). As an operator on \( \mathbb{R}^n \), it reflects about an \((n-1)\)-dimensional subspace (hyperplane).

The reflection about the orthogonal complement of unit vector \( u \) is \( R = I - 2uu^T \). Note that \( R^2 = I - 4uu^T + 4uu^T uu^T = I \).

The reflection in the plane or line whose normal is \( u = e_j \) is \( R = I - 2e_je_j^T \) (the identity with -1 as the \( j^{th} \) diagonal entry). It reverses the sign of the \( j^{th} \) component of the vector to which it is applied. This is a special case of a scaling operator. Reflection about the y axis, for example, is as follows.

\[
R = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \Rightarrow R \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -x \\ y \end{bmatrix}.
\]

Note that reflecting twice restores the original point \( (R^2 = I) \).
**Defn** An *orthogonal projection* (operator) is a symmetric idempotent matrix $P$: $P^T = P$ and $P^2 = P$.

The operator that projects onto the orthogonal complement of unit vector $u$ (a linear subspace) is $P = I - uu^T$. Projection onto the x-y plane (complement of $e_3$), for example, involves zeroing $z$ components:

$$P = I - e_3e_3^T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Note that projecting twice is the same as projecting once ($P^2 = P$). The projected point $Pq$ is the closest point in the range of $P$ (the projection plane) to $q$. 
Reflection and Projection: Geometry

\[ u, \|u\| = 1 \]

\[ u(u^T q) \]

\[ Pq = q - u(u^T q) \]

\[ Rq = q - 2u(u^T q) \]
Inverse Transformations

To invert translation by $\mathbf{t}$, translate by $-\mathbf{t}$. The composition of the operators is the identity:

$$T_{-\mathbf{t}}(T_{\mathbf{t}}(\mathbf{q})) = (\mathbf{q} + \mathbf{t}) - \mathbf{t} = \mathbf{q} \quad \text{for all } \mathbf{q} \quad \Rightarrow \quad T_{-\mathbf{t}} \circ T_{\mathbf{t}} = I.$$

The inverse of a linear transformation is the matrix inverse, where the inverse of $A$, if it exists, is the matrix $A^{-1}$ such that $A^{-1}A = AA^{-1} = I$. There are many other (equivalent) definitions of an invertible (nonsingular) matrix: nonzero determinant, linearly independent columns, or linearly independent rows, for example. In general it is not easy to decide if an arbitrary matrix is invertible, and computationally, the best we can do is determine if it is close to singular. The matrices that arise in graphics, however, are easily categorized.
Inverse Linear Transformations

- To invert a scaling matrix, scale by the reciprocals of the scale factors, assuming they are all nonzero.

\[
\begin{bmatrix}
  s_x & 0 \\
  0 & s_y \\
\end{bmatrix}^{-1} = \begin{bmatrix}
  1/s_x & 0 \\
  0 & 1/s_y \\
\end{bmatrix}.
\]

- To invert a rotation through angle \( \theta \), rotate through angle \(-\theta\) (about the same axis in the case of \( \mathbb{R}^3 \)). Equivalently, transpose the matrix:

\[
R_{-\theta} = \begin{bmatrix}
  \cos(-\theta) & -\sin(-\theta) \\
  \sin(-\theta) & \cos(-\theta) \\
\end{bmatrix} = \begin{bmatrix}
  \cos(\theta) & \sin(\theta) \\
  -\sin(\theta) & \cos(\theta) \\
\end{bmatrix} = R_{\theta}^T.
\]

- A reflection is its own inverse.

- A projection is not invertible. Recall that the 'projection operator' used by OpenGL is not actually a projection. Depths are retained.
The composition of two linear transformations is represented by the product of the corresponding matrices. The OpenGL modelview transformation is usually a composition of several affine transformations. It is more efficient to compute and store the composition as a single matrix and apply it to the vertices than to apply each operator individually to all the vertices. This raises the question of how to represent a translation operator as a matrix. The solution lies in the use of *homogeneous coordinates*, which also allow projective transformations to be represented by matrices.
The point with Cartesian coordinates \((x, y, z)\) has homogeneous coordinates \((x, y, z, 1)\), or more generally, \(\alpha \ast (x, y, z, 1)\) for any nonzero scalar \(\alpha\) — the equivalence class of all points in \(\mathbb{R}^4\) that project to \((x, y, z, 1)\) (with center of projection at the origin and projection plane \(w = 1\)). Reversing the mapping, the Cartesian coordinates of the point with homogeneous coordinates \((x, y, z, w)\) are \((x/w, y/w, z/w)\) obtained by scaling by \(1/w\) and dropping the fourth component \(1\). Note that as \(w \to 0\), the point approaches \(\infty\) in the direction \((x, y, z)\). Thus, \((x, y, z, 0)\) is referred to as the point at infinity. It’s not a valid point location, but \((x, y, z)\) is a valid direction, and OpenGL stores normal vectors and light source directions with \(w = 0\).
Affine Transformations in Homogeneous Coordinates

For the purpose of representing affine transformations, we always have \( w = 1 \). Then the matrix representation of \( T \) for \( T\mathbf{p} = A\mathbf{p} + \mathbf{t} \) is

\[
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} & t_1 \\
  a_{21} & a_{22} & a_{23} & t_2 \\
  a_{31} & a_{32} & a_{33} & t_3 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]

The blocks of the matrix are as follows.

- Upper left 3 by 3 block: linear operator \( A \)
- Upper right 3 by 1 block: translation vector \( \mathbf{t} \)
- Lower left 1 by 3 block: zero row vector \( \mathbf{0}^T \)
- Lower right 1 by 1 block: scalar with value 1

For a pure translation, we take \( A = I \) so that \( T\mathbf{p} = \mathbf{p} + \mathbf{t} \), and for a linear operator, we have \( \mathbf{t} = \mathbf{0} \) so that \( T\mathbf{p} = A\mathbf{p} \).
We measure the cost of forming composite transformations (matrix-matrix products) and of transforming vertices (matrix-vertex products) by counting multiplies. The number of adds is about the same. In general, a matrix-vector product requires $n^2$ multiplies for an order-$n$ matrix. In the case of homogeneous coordinates, however, the matrix has a special structure. There is no need to multiply by zero or one.

The matrix-vector product requires computing three components, each requiring three multiplies. The matrix-matrix product requires computing 12 components, each requiring three multiplies.

- Matrix-matrix product: 36 multiplies
- Matrix-vector product: 9 multiplies
Inverse Transformations in Homogeneous Coordinates

Note that $T$ maps a point $p$ in homogeneous coordinates to the transformed point in homogeneous coordinates.

\[
\begin{bmatrix}
T \mathbf{p} \\
1
\end{bmatrix} = \begin{bmatrix}
A & \mathbf{t} \\
0^T & 1
\end{bmatrix} \begin{bmatrix}
\mathbf{p} \\
1
\end{bmatrix} = \begin{bmatrix}
A\mathbf{p} + \mathbf{t} \\
1
\end{bmatrix}.
\]

The sequence of operations represented by $T$ is (1) apply the linear operator $A$, and then (2) translate by $\mathbf{t}$. We invert this by first translating by $-\mathbf{t}$, and then applying $A^{-1}$ to obtain $A^{-1}(\mathbf{p} - \mathbf{t})$ when applied to $\mathbf{p}$. This is equivalent to applying $A^{-1}$ and then translating by $-A^{-1}\mathbf{t}$. Hence the inverse operator is

\[
\begin{bmatrix}
A^{-1} & -A^{-1}\mathbf{t} \\
0^T & 1
\end{bmatrix}.
\]

We can verify this as follows.

\[
\begin{bmatrix}
A^{-1} & -A^{-1}\mathbf{t} \\
0^T & 1
\end{bmatrix} \begin{bmatrix}
A & \mathbf{t} \\
0^T & 1
\end{bmatrix} = \begin{bmatrix}
I & 0 \\
0^T & 1
\end{bmatrix}.
\]
**Composite Transformation Example 1**

**Problem** Construct the transformation that rotates a square in the x-y plane through angle $\theta$ about its center $c = (c_x, c_y)$.

**Solution** We will simplify the notation by omitting z components; i.e., we use homogeneous coordinates for points in $\mathbb{R}^2$.

1. **Translate by $-c$:** $T_{-c} = \begin{bmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & 1 \end{bmatrix}$

2. **Rotate through angle $\theta$:** $R_{\theta} = \begin{bmatrix} C & -S & 0 \\ S & C & 0 \\ 0 & 0 & 1 \end{bmatrix}$, where $C = \cos \theta$ and $S = \sin \theta$

3. **Translate by $c$:** $T_c = \begin{bmatrix} 1 & 0 & c_x \\ 0 & 1 & c_y \\ 0 & 0 & 1 \end{bmatrix}$
The composite transformation is

\[
T = T_c R_\theta T_{-c} = \begin{bmatrix}
1 & 0 & c_x \\
0 & 1 & c_y \\
0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
C & -S & 0 \\
S & C & 0 \\
0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
1 & 0 & -c_x \\
0 & 1 & -c_y \\
0 & 0 & 1
\end{bmatrix}
\]

\[
= \begin{bmatrix}
C & -S & c_x \\
S & C & c_y \\
0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
1 & 0 & -c_x \\
0 & 1 & -c_y \\
0 & 0 & 1
\end{bmatrix}
\]

\[
= \begin{bmatrix}
C & -S & -Cc_x + Sc_y + c_x \\
S & C & -Sc_x - Cc_y + c_y \\
0 & 0 & 1
\end{bmatrix}
\]

\[
\Rightarrow T(p) = R(p) - R(c) + c = R(p - c) + c.
\]
Let $\theta = \pi/2$ so that $C = 0$ and $S = 1$, and let $c_x = c_y = 2$. Then

$$T = \begin{bmatrix} 0 & -1 & 4 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \Rightarrow T \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 1 \end{bmatrix}, \quad T \begin{bmatrix} 3 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \\ 1 \end{bmatrix},$$

$$T \begin{bmatrix} 3 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix}, \quad T \begin{bmatrix} 1 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$
**Problem** Consider a unit square (side length 1) centered at \( c = (10, 10) \) with sides parallel to the axes. Construct a matrix \( T \) that rotates the square through \( \theta = 20 \) degrees clockwise about its center, and scales the side lengths uniformly by \( \alpha \) without changing the location of the center. Let \( C = \cos(20^\circ) \), \( S = \sin(20^\circ) \).

**Solution** \[ T = T_{-c}^{-1} S\alpha R_{\theta}^{-1} T_{-c} = \]

\[
\begin{bmatrix}
1 & 0 & 10 \\
0 & 1 & 10 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\alpha & 0 & 0 \\
0 & \alpha & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
C & S & 0 \\
-S & C & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 0 & -10 \\
0 & 1 & -10 \\
0 & 0 & 1
\end{bmatrix}
= \]

\[
\begin{bmatrix}
\alpha C & \alpha S & 10\left(1 - \alpha C - \alpha S\right) \\
-\alpha S & \alpha C & 10\left(1 - \alpha C + \alpha S\right) \\
0 & 0 & 1
\end{bmatrix}.
\]
Problem Describe the effect of applying 36 successive transformations by $T$ to the square by specifying the resulting shape, size, orientation, and location.

Solution

$$T^k = (T^{-1}_c S_{\alpha} R_{\theta}^{-1} T_c)^k = T^{-1}_c (S_{\alpha} R^{T}_{\theta})^k T_c$$

$$= T^{-1}_c S_{\alpha} R^{-k\theta} T_c$$ since $S_{\alpha} R^{T}_{\theta} = R^{T}_{\theta} S_{\alpha}$

$$= T^{-1}_c S_{\alpha^k} R_{-k\theta} T_c$$

Thus, $T$ scales the sides by $\alpha^k$ and rotates the square through angle $k\theta = 720^\circ$ clockwise about the center. The shape is square; side lengths are $\alpha^{36}$; the sides remain parallel to the axes, and the center remains at (10,10). Note that the scaling and rotation operators commute because the scaling is uniform ($S_{\alpha} = \alpha I$).
**Problem** Construct the (unique) *affine transformation* \( T \) that maps \([x_1, x_2]\) to \([y_1, y_2]\); i.e., \( y = T(x) = ax + b \) with the scale factor (slope) \( a \) and translation factor (y-intercept) \( b \) chosen so that

\[
\begin{align*}
  y_1 &= T(x_1) = ax_1 + b \quad \text{and} \\
  y_2 &= T(x_2) = ax_2 + b.
\end{align*}
\]

**Solution 1** Subtract (1) from (2):

\[
y_2 - y_1 = a(x_2 - x_1) \Rightarrow a = \frac{y_2 - y_1}{x_2 - x_1}.
\]

Solve (1) for \( b \):

\[
b = y_1 - ax_1.
\]

Hence

\[
T(x) = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1.
\]
Solution 2  Preserve ratios

\[
\frac{y - y_1}{y_2 - y_1} = \frac{x - x_1}{x_2 - x_1} \quad \Rightarrow \quad y - y_1 = \frac{y_2 - y_1}{x_2 - x_1} (x - x_1) \quad \Rightarrow \\

y = T(x) = \frac{y_2 - y_1}{x_2 - x_1} (x - x_1) + y_1.
\]
Problem Construct the 'projection' operator produced by
\texttt{glOrtho(l, r, b, t, n, f)}; i.e., the affine transformation \( T \)
that maps \([l, r] \times [b, t] \times [-f, -n]\) into \([-1, 1]^3\) with \((l, b, -n)\)
mapped to \((-1, -1, -1)\) and \((r, t, -f)\) mapped to \((1, 1, 1)\).

Solution A point \( p \) in the view volume \([l, r] \times [b, t] \times [-f, -n]\) is
represented by a column vector \([x \ y \ z]^T\), and \( T \) is defined by a
scaling matrix \( A = \text{diag}(s_x, s_y, s_z) \) and translation vector
\( b = [b_x \ b_y \ b_z]^T \): \( T(p) = Tp = \)

\[
Ap + b = \begin{bmatrix}
  s_x & 0 & 0 \\
  0 & s_y & 0 \\
  0 & 0 & s_z \\
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  z \\
\end{bmatrix}
+ \begin{bmatrix}
  b_x \\
  b_y \\
  b_z \\
\end{bmatrix}
= \begin{bmatrix}
  s_x x + b_x \\
  s_y y + b_y \\
  s_z z + b_z \\
\end{bmatrix};
\]

i.e., the problem reduces to three instances of the univariate affine
transformation problem (linear interpolation).
x component: map \([l, r]\) to \([-1, 1]\)

\[
s_x = \frac{2}{r - l}, \quad b_x = -1 - s_x l = -\frac{(r - l) - 2l}{r - l} = -\frac{r + l}{r - l}.
\]

y component: map \([b, t]\) to \([-1, 1]\)

\[
s_y = \frac{2}{t - b}, \quad b_y = -\frac{t + b}{t - b}.
\]

z component: map \([-n, -f]\) to \([-1, 1]\)

\[
s_z = -\frac{2}{f - n}, \quad b_z = -\frac{f + n}{f - n}.
\]

This transformation (in homogeneous coordinates) is displayed at the end of Appendix F. Note, however, that the inverse transformation is incorrectly specified in the Redbook.
The call `glOrtho(l, r, b, t, n, f)` generates $R$, where

$$R = \begin{bmatrix}
\frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\
0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\
0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\
0 & 0 & 0 & 1
\end{bmatrix}, \quad R^{-1} = \begin{bmatrix}
\frac{r-l}{2} & 0 & 0 & \frac{r+l}{2} \\
0 & \frac{t-b}{2} & 0 & \frac{t+b}{2} \\
0 & 0 & \frac{f-n}{-2} & -\frac{n+f}{2} \\
0 & 0 & 0 & 1
\end{bmatrix}$$

The call `glFrustum(l, r, b, t, n, f)` generates $R$, where

$$R = \begin{bmatrix}
\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\
0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\
0 & 0 & -(f+n) & \frac{-2fn}{f-n} \\
0 & 0 & -1 & 0
\end{bmatrix}, \quad R^{-1} = \begin{bmatrix}
\frac{r-l}{2n} & 0 & 0 & \frac{r+l}{2n} \\
0 & \frac{t-b}{2n} & 0 & \frac{t+b}{2n} \\
0 & 0 & \frac{-(f-n)}{2fn} & \frac{f+n}{2fn} \\
0 & 0 & 0 & -1
\end{bmatrix}$$
Perspective Projection

\[
\tan \theta = \frac{y'}{n} = \frac{y}{-z}
\]

\[
z = -f
\]

\[
z = -n
\]
Given a frustum defined by \( l < r, b < t, \) and \( 0 < n < f, \) and a point in the frustum with eye coordinates \((x, y, z)\), the projected point is \((-nx/z, -ny/z, -n)\). This is easily derived from the picture on the previous page. If we retained the depth, the transformed point would be \((-nx/z, -ny/z, z)\). That transformation is problematic as illustrated on the next page. We need to alter the depth in such a way that the transformation maps lines to lines but retains relative depths. The solution is to map \( z \) to \( z' = -fn/z - (f + n) \). This leaves \( z \) in the interval \([-f, -n]\) and retains relative depth:

\[
\begin{align*}
z'_1 < z'_2 & \iff -fn/z_1 < -fn/z_2 \\
& \iff -fnz_2 < -fnz_1 \quad (\text{since } z_1z_2 > 0) \\
& \iff -z_2 < -z_1 \quad (\text{since } fn > 0) \iff z_1 < z_2.
\end{align*}
\]
The figure illustrates the problem with the nonlinear depth-perspective transformation $T(x, y, z) = (-nx/z, -ny/z, z)$. The line containing solid circles is mapped to the curve containing open circles.
**Theorem** Define the nonlinear depth-perspective transformation by
\[ T(x, y, z) = \left( -\frac{nx}{z}, -\frac{ny}{z}, -\frac{fn}{z} - (f + n) \right) \] for
\[-f \leq z \leq -n < 0. \] Then \( T \) maps lines to lines, and hence planes to planes. More precisely, for \( p_1 = (x_1, y_1, z_1) \) and \( p_2 = (x_2, y_2, z_2) \),
\[ T(p_1 + t(p_2 - p_1)) = T(p_1) + s(T(p_2) - T(p_1)), \quad (*) \]
where
\[ s(t) = \frac{z_2 t}{z_1 + t(z_2 - z_1)} \in [0, 1], \quad s'(t) = \frac{z_1 z_2}{[z_1 + t(z_2 - z_1)]^2} > 0, \]
so that
\[ T\{p_1 + t(p_2 - p_1) : t \in [0, 1]\} = \{ T(p_1) + s(T(p_2) - T(p_1)) : s \in [0, 1]\}. \]
proof It suffices to verify (*). The first component is

\[
\frac{-n[x_1 + t(x_2 - x_1)]}{z_1 + t(z_2 - z_1)} = -\frac{nx_1}{z_1} + s \left( -\frac{nx_2}{z_2} + \frac{nx_1}{z_1} \right).
\]

Solving for \(s\) gives \(s(t) = \frac{z_2 t}{z_1 + t(z_2 - z_1)}\). The second component of (*) is identical to the first with \(y_i\) in place of \(x_i\) (\(i=1,2\)), and the third component is the same as the first with \(f\) in place of both \(x_1\) and \(x_2\). \(\square\)
Perspective Projection continued

Following the depth-perspective transformation we map the point to normalized device coordinates using the operator previously derived.

\[
\begin{bmatrix}
\frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\
0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\
0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
-\frac{nx}{z} \\
-\frac{ny}{z} \\
-\frac{fn}{z} - (f + n) \\
1
\end{bmatrix}
= 
\begin{bmatrix}
-\frac{2nx}{(r-l)z} & -\frac{r+l}{r-l} \\
-\frac{2ny}{(t-b)z} & -\frac{t+b}{t-b} \\
-\frac{2fn}{(f-n)z} & -\frac{f+n}{f-n} \\
1
\end{bmatrix}
\cdot \begin{bmatrix}
-\frac{nx}{z} \\
-\frac{ny}{z} \\
-\frac{fn}{z} - (f + n) \\
1
\end{bmatrix}
\]

Recall that the homogeneous coordinates may be scaled by any nonzero value without altering the equivalence class. We can eliminate the nonlinearity in the expression for the normalized device coordinates by scaling with \(-z\). This gives

\[
\begin{bmatrix}
\frac{2nx + (r + l)z}{r-l} \\
\frac{2ny + (t + b)z}{t-b} \\
\frac{-(f + n)z - 2fn}{f-n} \\
- z
\end{bmatrix}^T.
\]
The above expression is precisely the clip coordinates obtained by applying $R$ to the eye coordinates, where $R$ is the affine transformation generated by $\text{glFrustum}(l, r, b, t, n, f)$:

$$
\begin{bmatrix}
\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\
0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\
0 & 0 & \frac{-(f+n)}{f-n} & -\frac{2fn}{f-n} \\
0 & 0 & -1 & 0 \\
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1 \\
\end{bmatrix}
= 
\begin{bmatrix}
\frac{2nx+(r+l)z}{r-l} \\
\frac{2ny+(t+b)z}{t-b} \\
-(f+n)z-2fn \\
f-n \\
\end{bmatrix} - z
$$

We have thus shown that the transformation generated by $\text{glFrustum}$ followed by perspective division (division by the $w$-component) produces the normalized device coordinates associated with a depth-perspective transformed point.
**Viewport Transformation**

**Problem** Construct the viewport mapping produced by `glViewport(x0, y0, w, h)`; i.e., the affine transformation $T$ that maps normalized device coordinates $[-1, 1]^3$ to window coordinates plus depth $[x_0, x_0 + w] \times [y_0, y_0 + h] \times [0, 1]$.

**Solution** As in the case of the orthographic projection mapping, the components can be treated separately.

$$
x' = \frac{w}{2}(x + 1) + x_0
$$

$$
y' = \frac{h}{2}(y + 1) + y_0
$$

$$
z' = \frac{1}{2}(z + 1).
$$
The *aspect ratio* of a rectangle is the ratio width/height. In graphics the rectangles of interest include pixels, the display area of the screen, the window, the viewport, and the view volume (intersected with the near plane).

**Problem** Given display area aspect ratio 16/10 (typical of LCD monitors) and screen resolution $n_x$ by $n_y$, compute the *pixel ratio*.

**Solution** Let the pixel ratio be $p_x/p_y$. Then the screen width is $n_x p_x$, and the height is $n_y p_y$. Hence

$$\frac{n_x p_x}{n_y p_y} = \frac{8}{5} \Rightarrow \frac{p_x}{p_y} = \frac{8}{5} \frac{n_x}{n_y} = \frac{8n_y}{5n_x}.$$  

Square pixels are obtained with resolutions 1280 X 800, 1440 X 900, 1680 X 1050, and 1920 X 1200. A monitor with display area aspect ratio 5/4 should use resolutions such as 960 X 768, 1280 X 1024, and 1600 X 1280.
**Problem** Given a rectangle \([X_{min}, X_{max}] \times [Y_{min}, Y_{max}]\), construct a view volume and viewport such that the rectangle is centered in the window, completely visible, and has its aspect ratio preserved by the projection and viewport transformations.

Since we cannot reliably query the window system for the screen resolution, we must assume that pixels are square — usually a valid assumption. Then a simple solution uses a square view volume and square viewport. Compute a center and radius for the view volume as follows: \(x_c = (X_{min}+X_{max})/2, y_c = (Y_{min}+Y_{max})/2, r = \sqrt{(X_{max}-X_{min})^2 + (Y_{max}-Y_{min})^2}/2\). The code is on the following page.

Is the viewport as large as possible? What portion of the window does the rectangle occupy when both the window and rectangle have aspect ratio 2? The above code is not a good solution when the rectangle’s aspect ratio differs significantly from 1.
void reshape(int w, int h)
{
    if (w <= h) {
        glViewport(0, (GLint)(h-w)/2, (GLint)w, (GLint)w);
    } else {
        glViewport((GLint)(w-h)/2, 0, (GLint)h, (GLint)h);
    }
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity;
    gluOrtho2D(xc-r, xc+r, yc-r, yc+r);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity;
    return;
}
Zoom and pan are achieved by changing the view volume size and center, respectively. By expanding the size of the view volume (in the x and y directions) with a fixed viewport, we are effectively zooming out and rendering a larger portion of the scene. Conversely, by mapping a smaller portion of the scene into the fixed-size viewport, we are zooming in.

It is convenient for this purpose to define the view volume by a center \( c = (x_c, y_c) \) and radius \( r \), along with a zoom factor \( s \) with default value 1:

\[
\text{glOrtho}(x_c-s*r, x_c+s*r, y_c-s*r, y_c+s*r, n, f);
\]

Then panning is achieved by altering \( c \), and zooming is achieved by perhaps doubling or halving \( s \) in response to user input.
**Problem** Given eye position (camera position or center of projection) \( E = (Ex, Ey, Ez) \), center (object point or reference point) \( C = (Cx, Cy, Cz) \), and view-up direction (up vector) \( v = (Vx, Vy, Vz) \), construct the viewing part of the modelview matrix; i.e., implement

\[
\text{gluLookAt}(Ex, Ey, Ez, Cx, Cy, Cz, Vx, Vy, Vz); 
\]

**Solution** Define projection vector \( n \) as the unit normal to the projection plane: \( n = (E - C)/\|E - C\| \). Then construct the *rigid body transformation* as follows.

1. Translate by \(-C\)
2. Rotate by \( R \), where \( Rn = e_3 = (0, 0, 1) \)
3. Rotate by \( R_z \), where \( R_z(u/\|u\|) = e_2 = (0, 1, 0) \) for \( u = RPv \), where \( P = I - nn^T \) is the orthogonal projection onto the orthogonal complement of \( n \)
4. Translate by \(-\|E - C\|e_3\).
Recall that glRotate requires an angle and pole (axis direction). We may associate unit direction vectors with points on the unit sphere. Given a pair of points, \( \mathbf{n} \) and \( \mathbf{e}_3 \), separated by less than 180 degrees, there is a unique Euler pole defining the rotation \( R \) from \( \mathbf{n} \) to \( \mathbf{e}_3 \) along the shorter great circle arc — the unit normal to the plane of \( \mathbf{n} \) and \( \mathbf{e}_3 \) with the right sense. It may be computed as \( \mathbf{n} \times \mathbf{e}_3 = (n_2, -n_1, 0) \) normalized to a unit vector, for \( \mathbf{n} = (n_1, n_2, n_3) \). The angle \( \theta \) can be computed from the dot product: \( \mathbf{n} \cdot \mathbf{e}_3 = ||\mathbf{n}|| ||\mathbf{e}_3|| \cos(\theta) = \cos(\theta) = n_3 \). The angle associated with \( R_z \) is computed similarly.
Viewing Transformations: animation

Note that the distance $\|E - C\|$ is only relevant for a perspective projection, in which case, it must not be too small. We zoom in or out by changing the extent of the view volume in the near plane rather than changing the distance $n$ to the near plane.

A simpler solution, which can be shown to be equivalent, is as follows.

1. Translate by $-E$
2. Rotate by $R$, where $Rn = e_3$
3. Rotate by $R_z$, where $R_z(u/\|u\|) = e_2$ for $u = Rv$ projected onto the x-y plane (by zeroing out the z component).

If we construct a sequence of frames associated with small changes in $E$, but fixed values of $C$ and $\|E - C\|$, we are effectively rotating $E$ about fixed $C$. This is equivalent to rotating the object or scene about $C$ in the opposite direction. By allowing an interactive user to rotate about two of the coordinate axes, the scene can be viewed from an arbitrary perspective.
A similar problem is to simulate motion in which the object point $C$ changes. In this case, we alter $C$ with fixed $E$ and $\|E - C\|$. In general we wish to allow an interactive user to effectively rotate the scene about the coordinate axes in eye coordinates, thus changing Euler angles ($pitch$, $yaw$, and $roll$):

- horizontal axis (pitch): Up/Down arrows or X/x
- vertical axis (yaw or heading): Left/Right arrows or Y/y
- z axis (roll or bank): Z/z

In order to change the pitch, for example, we can easily construct a matrix $R_x$ that rotates about the x axis, but then we must change the modelview matrix from $M$ to $R_x M$. Unfortunately, this requires left-multiplying, and a call to `glRotate*` right-multiplies. Another problem is that accumulating a product of a large number of rotations will eventually fail to be a rotation due to roundoff error.
Instead of computing products of rotation matrices, it is preferable to represent the rotation with a single axis and angle or the quaternion equivalent. Composition of rotations can be computed robustly as products of quaternions.

A viewing transformation that allows an interactive user to alter both the eye position (viewer location relative to the scene) and the orientation (viewing direction or Euler angles) is as follows.

1. Translate by $-C$
2. Rotate through location_angle about location_axis
3. Translate by $-3r\mathbf{e}_3$, where $r$ is the radius of a bounding sphere
4. Rotate through orientation_angle about orientation_axis

The second translation was somewhat arbitrarily chosen to make $n = 2r$. 
The code for the above procedure is as follows:

```c
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glRotated(orientation_angle, orientation_axis[0], orientation_axis[1], orientation_axis[2]);
glTranslated(0.0, 0.0, -3.0*r);
glRotated(location_angle, location_axis[0], location_axis[1], location_axis[2]);
glTranslated(-Cx, -Cy, -Cz);
```