4.1

$$(x,y,z) \to (x,y,z,1) \to T(x,y,z,1) \to (x',y',z',w')$$

$$\to \left(\frac{x'}{w}, \frac{y'}{w}, \frac{z'}{w}, 1\right)$$

$$\to \left(\frac{x'}{w}, \frac{y'}{w}, \frac{z'}{w}\right)$$

4.1.1 T(points)

```
Point3f p = ...;
Transform T = ...;
Point3f pNew = T(p);
```

code

```
template <typename T> inline Point3<T>
Transform::operator()(const Point3<T> &p) const {
    T x = p.x, y = p.y, z = p.z;
    T xp = m.m[0][0]*x + m.m[0][1]*y + m.m[0][2]*z + m.m[0][3];
    T yp = m.m[1][0]*x + m.m[1][1]*y + m.m[1][2]*z + m.m[1][3];
    T zp = m.m[2][0]*x + m.m[2][1]*y + m.m[2][2]*z + m.m[2][3];
    T wp = m.m[3][0]*x + m.m[3][1]*y + m.m[3][2]*z + m.m[3][3];
    if (wp == 1) return Point3<T>(xp, yp, zp);
    else         return Point3<T>(xp, yp, zp) / wp;
}
```

4.1.2. vectors $(x,y,z,0)$ $\to T(x,y,z,0) = T(x',y',z',0)$

$$\to (x',y',z')$$

code

```
template <typename T> inline Vector3<T>
Transform::operator()(const Vector3<T> &v) const {
    T x = v.x, y = v.y, z = v.z;
    return Vector3<T>(m.m[0][0]*x + m.m[0][1]*y + m.m[0][2]*z,
                      m.m[1][0]*x + m.m[1][1]*y + m.m[1][2]*z,
                      m.m[2][0]*x + m.m[2][1]*y + m.m[2][2]*z);
}
```

4.1.3. Norm

$T(\text{vector}) \neq T(\text{norm})$

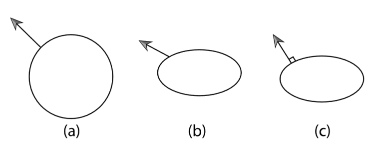

Figure 2.14: Transforming Surface Normals. (a) Original circle, with the normal at a point indicated by an arrow. (b) When scaling the circle to be half as tall in the y direction, simply treating the normal as a direction and scaling it in the same manner gives a normal that is no longer perpendicular to the surface. (c) A properly transformed normal.

$$n \cdot t = n^T \cdot t = 0$$

$$\downarrow$$

$$(Sn)^T M t = 0$$

$$n^T S^T M t = 0$$

$$\downarrow$$

$$S^T M = I$$

$$S^T = M^{-1}$$

$$S = (M^{-1})^T$$

code

<<Transform Inline Functions>>+=

```
template <typename T> inline Normal3<T>
Transform::operator()(const Normal3<T> &n) const {
    T x = n.x, y = n.y, z = n.z;
    return Normal3<T>(mInv.m[0][0]*x + mInv.m[1][0]*y + mInv.m[2][0]*z,
                      mInv.m[0][1]*x + mInv.m[1][1]*y + mInv.m[2][1]*z,
                      mInv.m[0][2]*x + mInv.m[1][2]*y + mInv.m[2][2]*z);
}
```

4.1.4. Ray

<<Transform Inline Functions>>+=

```
template <typename T> inline Normal3<T>
Transform::operator()(const Normal3<T> &n) const {
    T x = n.x, y = n.y, z = n.z;
    return Normal3<T>(mInv.m[0][0]*x + mInv.m[1][0]*y + mInv.m[2][0]*z,
                      mInv.m[0][1]*x + mInv.m[1][1]*y + mInv.m[2][1]*z,
                      mInv.m[0][2]*x + mInv.m[1][2]*y + mInv.m[2][2]*z);
}
```

4.1.5. Bounding Box

```
Bounds3f Transform::operator()(const Bounds3f &b) const {
    const Transform &M = *this;
    Bounds3f ret(M(Point3f(b.pMin.x, b.pMin.y, b.pMin.z)));
    ret = Union(ret, M(Point3f(b.pMax.x, b.pMin.y, b.pMin.z)));
    ret = Union(ret, M(Point3f(b.pMin.x, b.pMax.y, b.pMin.z)));
    ret = Union(ret, M(Point3f(b.pMin.x, b.pMin.y, b.pMax.z)));
    ret = Union(ret, M(Point3f(b.pMin.x, b.pMax.y, b.pMax.z)));
    ret = Union(ret, M(Point3f(b.pMax.x, b.pMax.y, b.pMin.z)));
    ret = Union(ret, M(Point3f(b.pMax.x, b.pMin.y, b.pMax.z)));
    ret = Union(ret, M(Point3f(b.pMax.x, b.pMax.y, b.pMax.z)));
    return ret;
}
```

tip: recall that:

It is also useful to be able to initialize a Bounds3 to enclose a single point:

<<Bounds3 Public Methods>>+=

```
Bounds3(const Point3<T> &p) : pMin(p), pMax(p) { }
```

4.1.7. Coordinate Handedness, (det[i]<0 ⇒ changed)

Fortunately, it is easy to tell if handedness is changed by a transformation: it happens only when the determinant of the transformation's upper-left 3×3 submatrix is negative.

<<Transform Method Definitions>>+=

```
bool Transform::SwapsHandedness() const {
    Float det =
        m.m[0][0] * (m.m[1][1] * m.m[2][2] - m.m[1][2] * m.m[2][1]) -
        m.m[0][1] * (m.m[1][0] * m.m[2][2] - m.m[1][2] * m.m[2][0]) +
        m.m[0][2] * (m.m[1][0] * m.m[2][1] - m.m[1][1] * m.m[2][0]);
    return det < 0;
}
```