Chapter 12

A 2D and 3D transformation library for graphics

After this chapter, you will know:

- How to convert the ideas from Chapters 6, 10, and 11 into working code.

The ideas of the previous chapters can be nicely condensed into an implementation — a collection of cooperating classes that help to maintain the point/vector distinction, the distinction between a transformation $T$ that acts on points and the associate transformations of vectors and covectors, and some of the other routine computations that are often done in graphics.

The book's website has such an implementation, in C# starting with the predefined `Point`, `Vector`, `Point3D`, and `Vector3D` WPF classes that you've already seen. You should download the implementation and look at it as you read this chapter.

The implementation depends on a matrix library — one capable of inverting matrices, solving linear systems, multiplying matrices, etc. We've chosen to import the `MathNet.Numerics.LinearAlgebra` library [?], but if you prefer another, it should be easy to substitute, as our use of the library is highly localized.

In most of the classes there are procedures that can fail under certain circumstances; for instance, if you ask for a linear transformation that sends $v_1$ to $w_1 \neq 0$, and also sends it to $2w_1$, there is no satisfactory answer. Such failures all amount to some matrix being non-invertible. We raise exceptions for such problems, and they are discussed in the code and its documentation, but not in this chapter.
The approach taken in our implementation is not the only one possible. Our approach depends on transformations as the primitive notion, but it's quite possible and reasonable to think instead of coordinate systems as the fundamental entity. Just as in Chapter ?? we discussed the interpretation of a linear transformation as a change of coordinates, one can approach much of graphics this way. You end up having coordinate-frames for vector spaces (for a two-dimensional space, you have two independent vectors; for a 3D space, you have three independent vectors), for affine spaces (typically a coordinate frame for a 2D affine space consists of three points, from which one determines barycentric coordinates, but one can also build a frame from a point and two vectors), and for projective spaces (where in 2D, a projective frame is represented by four points in “general position,” which we'll discuss shortly). This approach is taken by Mann et al. [?].

12.1 Points and Vectors

The predefined Point and Vector classes already implement the main ideas we've discussed (for two dimensions): there are operators defined so that you can add a Point and a Vector to get a new Point, but there is no operator for adding two Points, for instance. Certain convenience operations have been included, like the dot-product of Vectors.

There are idiosyncrasies in the design of the classes, however. The two coordinates of a point \( P \) are \( P.X \) and \( P.Y \); there's no way to refer to them as elements of an array of length two, nor even a predefined “cast” operation to convert to a double[2]. There is, however, a predefined CrossProduct operation for Vectors, which treats the vectors as lying in the \( xy \)-plane of 3-space, computes the 3D cross product (which always points along the \( z \)-axis), and returns the \( z \) component of the resulting vector. In deference to the convenience of having datatypes that work well with the remainder of WPF, we've ignored these idiosyncrasies, and simply used the parts of the Point and Vector classes (and their 3D analogs) that we like. We've also added some geometric functions to our LIN_ALG namespace (in which all the transformation classes reside) to do things like the two-dimensional cross-product of one vector.

12.2 Transformations

While WPF also has a class called Matrix, its peculiarities made it unsuitable for our use. Furthermore, we wanted to build a library in which the fundamental idea was that of a transformation rather than the matrices that are used to implement the transformations. We therefore defined four classes:

- MatrixTransformation2: a parent class for linear, affine, and projective transformations. Since all three can be represented by \( 3 \times 3 \)
matrices, a MatrixTransformation holds a $3 \times 3$ matrix, and provides certain support procedures to multiply and invert such matrices.

- LinearTransformation2: A transformation that takes Vectors to Vectors.
- AffineTransformation2: A transformation that can operate on both Points and Vectors.
- ProjectiveTransformation2: A transformation that operates on Points in their homogeneous representation, and includes a division-by-the-last-coordinate after the matrix multiplication.

(There are four corresponding classes for transformations in 3D.)

In each case, we’ve defined composition of transformations by overloading the $*$ operator, and the application of a transformation to a Point or Vector by overloading the $*$ operator again. Thus to translate a point and then rotate it by $\pi/6$, we could write

```java
Point P = new Point(...);
AffineTransformation2 T = new AffineTransformation2(new Vector(3,1));
AffineTransformation2 S = AffineTransformation2.RotateXY(Math.PI/6);
Point Q = (S * T)*P;
```

Of course, if we were planning to operate on many points with this composed transformation, we’d write

```java
...  
AffineTransformation2 T2 = (S * T);
Point Q = T2*P;
```

### 12.2.1 Efficiency

Applying a transformation to a point or vector involves some memory allocation, a method-invocation, and a matrix multiplication. If you simply stored the matrix yourself, you could avoid most of this cost. And since graphics programs end up applying lots of transformations to lots of points and vectors, you might think that this is the best possible approach. If you’re writing a program that will be doing real-time graphics on a processor where computation is a real bottleneck (for example, a game that runs on a battery-powered device), it may well be the best approach. But as a student of computer graphics, you’re likely to write a lot of programs that get run just a few times. The great “cost” in your programs is your time as a developer. Using a high-level approach can help reduce bugs and even increase efficiency, as you notice ways to restructure your code that would be difficult to see if you were looking at every detail all the time. And a profiler can help you to determine exactly where in your code it’s worth the trouble of working with a low-level construct rather than a high-level one.
Nonetheless, there are places where one can get a certain amount of efficiency at no cost. For instance, the LinearTransformation2 class uses a $3 \times 3$ matrix to represent a transformation, but that matrix always has the form

$$
\begin{bmatrix}
a & b & 0 \\
c & d & 0 \\
0 & 0 & 1
\end{bmatrix},
$$

so it’s much easier to invert than a general $3 \times 3$ matrix (we just invert the upper-left $2 \times 2$ matrix); in the same way, multiplying two of these is much less work than multiplying two $3 \times 3$ matrices (we just multiply the upper-left $2 \times 2$ matrices). By overriding the MatrixTransform2 methods for inversion and multiplication, we get a large efficiency improvement.

**Exercise 12.1:** Without looking at the code, consider whether you can find a more efficient way to invert an affine transformation on $\mathbb{R}^2$ than by inverting its $3 \times 3$ matrix. Hint: the bottom row of the matrix is always $[0 \ 0 \ 1]$.

### 12.3 Specification of Transformations

For each kind of transformation, the default constructor generates the identity transformation. (The constructor for MatrixTransformation2 is protected, as only the derived classes are supposed to ever create a MatrixTransformation.) In general, though, transformations are constructed by static methods with mnemonic names. For the AffineTransformation2 class, for instance, there are eight static methods that construct transformations.

```java
public static AffineTransform2 RotateXY(double angle)
public static AffineTransform2 Translate(Vector v)
public static AffineTransform2 Translate(Point p, Point q)
public static AffineTransform2 AxisScale(double xamount, double yamount)
public static AffineTransform2 RotateAboutPoint(Point p, double angle)
public static AffineTransform2 PointsToPoints(Point p1, Point p2, Point p3, Point q1, Point q2, Point q3)
public static AffineTransform2 PointsAndVectorToPointsAndVector(Point p1, Point p2, Vector v1, Point q1, Point q2, Vector w1)
public static AffineTransform2 PointsAndVectorToPointsAndVector(Point p1, Point p2, Vector v1, Point q1, Point q2, Vector w1)
```

The naming convention is straightforward: “from” comes before “to”, so that

```
Translate(Point p, Point q)
```
creates a translation that sends \( p \) to \( q \), and within a collection of arguments, points come before vectors, so that in

```java
PointAndVectorsToPointAndVectors
```
the point \( p_1 \) is sent to the point \( q_1 \), the vector \( v_1 \) is sent to the vector \( w_1 \), and the vector \( v_2 \) is sent to the vector \( w_2 \). The method name tells you that there is one point and more than one vector; since an affine transformation of the plane is determined by its values on three points, or a point and two vectors, or two points and one vector, you know that the arguments must be one point and two vectors.

Methods that produce particular familiar transformations — translations, rotations, axis-aligned scales — have names indicating these.

While the names are sometimes cumbersome, they are also expressive; it’s easy to understand code that uses them.

### 12.4 Implementation

Most of the transformations are easy to implement. For instance, we first implemented the

```java
PointAndVectorsToPointAndVectors
```
method for `AffineTransform2`; once we’d done this, the `PointsToPoints` method was straightforward:

```java
public static AffineTransform2 PointsToPoints(
    Point p1, Point p2, Point p3,
    Point q1, Point q2, Point q3)
{
5        Vector v1 = p2 - p1;
        Vector v2 = p3 - p1;
        Vector w1 = q2 - q1;
        Vector w2 = q3 - q1;
        return AffineTransform2.PointAndVectorsToPointAndVectors(p1, v1, v2,
          q1, w1, w2);
10}
```

The `PointAndVectorsToPointAndVectors` code is implemented in a relatively straightforward way: we know that the vectors \( v_1 \) and \( v_2 \) must be sent to the vectors \( w_1 \) and \( w_2 \); in terms of the \( 3 \times 3 \) matrix, that means that the upper left corner must be the linear transformation of 2-space that effects this transformation. So we invoke the `LinearTransformation2` method `VectorsToVectors` to produce the transformation. Unfortunately, the resulting transformation does not send \( p_1 \) to \( q_1 \) in general. To ensure this, we precede this vector transform with a translation that takes \( p_1 \) to the origin (the translation has no effect on vectors, of course); we then perform the linear transformation; we then follow this with a transformation that takes the origin to \( q_1 \). The net result is that \( p_1 \) is sent to \( q_1 \), and the vectors are transformed as required.
To make this work relies on the `VectorsToVectors` method for `LinearTransformation2`. This, however, is straightforward: we place the vectors \( v_1 \) and \( v_2 \) in the first two columns of a \( 3 \times 3 \) matrix, with a 1 in the lower right. This transformation \( T \) sends \( e_1 \) to \( v_1 \) and \( e_2 \) to \( v_2 \). Similarly, we can use the ws to build a transformation \( S \) that sends \( e_1 \) to \( w_1 \), and \( e_2 \) to \( w_2 \). The composition \( T \circ S^{-1} \) sends \( v_1 \) to \( e_1 \) to \( w_1 \), and similarly for \( v_2 \), and hence solves our problem.

### 12.4.1 Projective transformations

The only really subtle implementation problem is the `PointsToPoints` method for `ProjectiveTransformation2`. Explaining this code requires a bit of mathematics, but it’s all mathematics that we’ve seen before in various forms.

We’re given four points \( P_1, P_2, P_3, \) and \( P_4 \) (we’ll revert to mathematical notation for the moment) in the Euclidean plane, and we are to find a projective transformation that sends them to the four points \( Q_1, Q_2, Q_3, \) and \( Q_4 \) of the Euclidean plane.

Before we go any further, we should mention a limitation. When we described the `VectorsToVectors` method of `LinearTransformation2`, we promised to send \( v_1 \) and \( v_2 \) to \( w_1 \) and \( w_2 \), but there was, in fact, a constraint. If \( v_1 = 0 \) and \( w_1 \neq 0 \), there’s no linear transformation that accomplishes this. In fact, if \( v_1 \) is a multiple of \( v_2 \), in general there’s no linear transformation solving the problem (except in the very special case where \( w_1 \) is the same multiple of \( w_2 \), in which case there are infinitely many solutions). The implicit constraint was that \( v_1 \) and \( v_2 \) must be linearly independent for our solution to work (or for the general problem to have a solution). In the case of `PointsToPoints`, we require something similar: the points \( P_i \), \( i = 1, \ldots, 4 \) must be in general position, which means that (a) no two of them can be the same, and (b) no one of them can lie on the line determined by two others (see Figure 1.1. In more familiar terms, this is equivalent to saying (a) that \( P_1, P_2, \) and \( P_3 \) form a nondegenerate triangle, and (b) that the barycentric coordinates of \( P_4 \) with respect to \( P_1, P_2, \) and \( P_3 \) are all nonzero. We’ll further require that the \( Q_i \)s are similarly in general position\(^1\).

Returning to the main problem of sending the \( P \)s to the \( Q \)s: When we express the points \( P_i \) and \( Q_i \) as elements of 3-space, we add a 1 to each of them to make it a vector whose tip lies in the \( w = 1 \) plane. We’ll call these vectors \( p_1, p_2, \) etc. Our problem can then be expressed by saying that we

\(^1\)This latter condition is overly stringent, but simplifies the analysis somewhat.
seek a $3 \times 3$ matrix $M$ with the property that
\[
M_{p_1} = \alpha q_1 \\
M_{p_2} = \beta q_2 \\
M_{p_3} = \gamma q_3 \\
M_{p_4} = \delta q_1
\]
for some four nonzero numbers $\alpha, \beta, \gamma,$ and $\delta$ (because, for instance, $\alpha q_1$, when we divide through by the last coordinate, will become $q_1$). The problem is that we do not know the values of the multipliers.

This problem, as stated, is too messy. If there were no multipliers, we'd be looking for a $3 \times 3$ matrix with $M_{p_i} = q_i$ ($i = 1, \ldots, 4$). We can only solve such problems for three vectors at a time, not four. So the multipliers are essential — without them, there'd be no solution at all. But they also complicate matters: we're looking for the nine entries of the matrix, and the four multipliers, which makes 13 unknowns. But we have four equations, each of which has three components, so we have 12 equations and 13 unknowns, a large underdetermined system. It's easy to see why the system is underdetermined, though: if we found a solution $(M, \alpha, \beta, \gamma, \delta)$, then we could double everything and get another equally good solution $(2M, 2\alpha, 2\beta, 2\gamma, 2\delta)$ to Equations 1.1–1.4.

**Inline Exercise 12.1:** Verify this claim.

So our first step is to make the solution unique by declaring that we're looking for a solution with $\delta = 1$. This gives 13 equations in 13 unknowns; we could simply solve this linear system. But there's a simpler approach that involves much less computation in this $3 \times 3$ case, and even greater savings in the $4 \times 4$ case.

We'll follow a familiar pattern in simplifying the problem. To send the $p$s to the $q$s, we'll instead find a way to send four standard vectors to the $q$s, and the same four vectors to the $p$s, and then compose one of these transformations with the inverse of the other. The four standard vectors we'll use are $e_1, e_2, e_3,$ and $u = e_1 + e_2 + e_3$. We'll start by finding a transformation that sends these to multiples of $q_1, q_2, q_3,$ and $q_4$, respectively.

**Step 1:** The matrix whose columns are $q_1, q_2,$ and $q_3$, sends $e_1$ to $q_1$, $e_2$ to $q_2$, and $e_3$ to $q_3$. Unfortunately, it does not necessarily send $u$ to $q_4$; instead, it sends $u = e_1 + e_2 + e_3$ to $q_1 + q_2 + q_3$, i.e., the sum of the columns. If we scale up each of the columns by a different factor, the resulting matrix will still send $e_i$ to a multiple of $q_i$ for $i = 1, 2, 3$, but will send $u$ to a sum of multiples of the $q$s. We therefore, as a first step, write $q_4$ as a linear combination of $q_1, q_2, q_3$:
\[
q_4 = \alpha q_1 + \beta q_2 + \gamma q_3, \tag{12.5}
\]
which is exactly the same thing as writing $Q_4$ in barycentric coordinates
with respect to $Q_1, Q_2, \text{and } Q_3$. Note that because of the general position assumption, $\alpha, \beta,$ and $\gamma$ are all nonzero.

**Inline Exercise 12.2:** Explain this last statement.

In terms of code, to find $\alpha, \beta,$ and $\gamma$, we build a matrix $Q = \begin{bmatrix} q_1 & q_2 & q_3 \end{bmatrix}$ and let

$$\begin{bmatrix} \alpha & \beta & \gamma \end{bmatrix} = Q^{-1}q_4.$$ \hfill (12.6)

**Inline Exercise 12.3:** Explain why the solution of Equation 1.6 is in fact a solution to Equation 1.5.

Now consider the matrix

$$A = \begin{bmatrix} \alpha q_1 & \beta q_2 & \gamma q_3 \end{bmatrix}.$$  

It’s straightforward to verify that it sends $e_i$ to a multiple of $q_i$ for $i = 1, 2, 3$, and it sends $u$ to the sum of its columns, which is, by Equation 1.6, exactly $q_4$.

Repeating this process, we can find a matrix transformation with matrix $B$ sending $e_1, e_2, e_3, u$ to multiples of $p_1, p_2, p_3, p_4$. The matrix $AB^{-1}$ then sends the $p$s to multiples of the corresponding $q$s.

Note that in solving this problem we solved a $3 \times 3$ system of equations and inverted a $3 \times 3$ matrix — far less computation than solving a $13 \times 13$ system of equations.

**Inline Exercise 12.4:** Explain why the matrix $B = \begin{bmatrix} \alpha' p_1 & \beta' p_2 & \gamma' p_3 \end{bmatrix}$, where $\alpha', \beta', \gamma'$ are the barycentric coordinates of $P_4$ with respect to $P_1, P_2, \text{and } P_3$, is invertible. Hint: write $B = PS$, where $S$ is diagonal and $P$ has $p_1, p_2, p_3$ as columns. Now apply the general-position assumption about the points $P_i (i = 1, \ldots, 4)$.

### 12.5 Three dimensions

The three-dimensional portion of the library is completely analogous to the 2D one, except that rotations are somewhat more complicated; to implement rotation about an arbitrary vector, we use Rodrigues’ formula; the implement rotation about an arbitrary ray (specified by a point and direction), we translate the point to the origin, rotate about the vector, and translate back. The projective `PointsToPoints` method uses the same general approach that we used in two dimensions, replacing the solution of 21 simultaneous equations with a solution of 4 simultaneous equations and a $4 \times 4$ matrix inversion.
12.6 Associated transformations

When we have an affine transformation $T$ on Euclidean space, we've said that we can either transform points or vectors, and we've incorporated this into our code. For an affine transformation, we've also seen how to transform covectors; in the code, we've defined a `Covector` structure (which bears some strong similarity to the `Vector` structure, in the sense of storing two doubles). And for affine transformations, there's an associated transformation, $T.\text{NormalMap}$, of covectors (we've bowed to convention here in calling this the “normal map” rather than “covector map,” since its use in graphics is almost entirely restricted to normal (co)vectors to triangles).

We'll now take a larger view of this situation. A typical affine map of the plane is

$$T : \mathbb{E}^2 \to \mathbb{E}^2 : (x,y) \mapsto (2x + 3y - 1, x - y + 6).$$

The “translation” part of the map is $\begin{bmatrix} -1 \\ 6 \end{bmatrix}$; the linear-transformation part is $\begin{bmatrix} 2 & 3 \\ 1 & -1 \end{bmatrix}$. We've generally expressed this map using homogeneous coordinates and a $3 \times 3$ matrix, but this is merely a choice of representation. At its core, it is simply a nice map from $\mathbb{R}^2$ to $\mathbb{R}^2$. As such, it has a derivative, $DT$. Before discussing its derivative, however, let's write down a second map from $\mathbb{E}^2$ to $\mathbb{E}^2$, a more complicated one:

$$S : \mathbb{E}^2 \to \mathbb{E}^2 : (x,y) \mapsto (x^2 - 3y, 2xy - \sin(\pi y) + 6).$$

This will serve to provide a contrast to $T$. The derivative of $S$ at the point $(x,y)$ is the linear transformation whose matrix is

$$\begin{bmatrix} 2x & -3 \\ 2y & 2x - \pi \cos(\pi y) \end{bmatrix},$$

which is commonly called the Jacobian matrix at $(x,y)$. To make this concrete, that means that Jacobian matrix at $(x,y) = (1,2)$ is

$$\begin{bmatrix} 2 & -3 \\ 4 & (2 - \pi) \end{bmatrix}.$$

This is the matrix for the linear transformation

$$\begin{bmatrix} u \\ v \end{bmatrix} \mapsto \begin{bmatrix} 2 & -3 \\ 4 & (2 - \pi) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix},$$

which we call $DS(1,2)$. You may be asking why we refer to the derivative at a point as a linear transformation. After all, when we look at the function $f : \mathbb{R} \to \mathbb{R} : x \mapsto x^2$, the derivative is $f'(x) = 2x$, and the derivative at, say, $x = 3$ is simply 6, which is a number rather than a linear transformation.
The answer is that in one dimension, the number 6 and the linear transformation “multiply by six” are easy to conflate. But in higher dimensions, the notion of the derivative as linear transformation is far more natural. Our most frequent use of the derivative is captured in the first few terms of Taylor’s theorem, which says, in one dimension, that (under some mild conditions)

\[ f(x + h) = f(x) + f'(x)h + o(h^2), \]

which can be read in the following way: if we pick a point \( x \), we can evaluate \( f(x) \) there. If we want to evaluate \( f \) at a point near \( x \), say \( x + h \) (where \( h \) is small), we can simply take the value we already computed for \( f(x) \) and add to it a multiple of \( h \) (namely \( f'(x)h \)); the result will be approximately correct; indeed, the error in the result is no more than a constant times \( h^2 \), so when \( h \) is small, the approximation is very good.

As an instance of this, we can consider the function \( g(x) = \sqrt{x} \). It’s easy to evaluate \( g(100) = 10 \). We can use this to evaluate \( \sqrt{102} \) as well: according to Taylor’s theorem,

\[ g(102) = g(100 + 2) = g(100) + g'(100) \cdot 2 + ... \]

Since \( g'(100) = 1/(2\sqrt{100}) = 1/20 \), we get

\[
\begin{align*}
g(102) &= g(100) + g'(100) \cdot 2 + ... \\
&= 10 + (1/20) \cdot 2 + ... \\
&= 10 + (1/10) + ... \\
&\approx 10.1.
\end{align*}
\]

The precise value is about 10.0995.

We can rewrite Taylor’s formula like this:

\[ f(x + h) - f(x) \approx f'(x)h. \]

In this form, is says that if we move a small amount \( h \) away from \( x \), the value of \( f \) changes by \( f'(x)h \), i.e., a constant multiple of \( x \) (where we are thinking of \( x \) as fixed, and \( h \) as varying). We can think of this as transforming a small change in \( x \) into a small change in \( f(x) \); the point of Taylor’s theorem (in this form) is that this transformation from a small change in the domain to a small change in the codomain is approximately “multiply by a constant,” i.e., is approximately a linear transformation.

The corresponding statement for our function \( S \) from \( \mathbb{R}^2 \) to \( \mathbb{R}^2 \) is that

\[ S((x, y) + \begin{bmatrix} h \\ k \end{bmatrix}) - S(x, y) \approx J S(x, y) \begin{bmatrix} h \\ k \end{bmatrix}, \]

where \( J S(x, y) \) is the Jacobian matrix of \( S \) at the point \((x, y)\). Again, we can read this as saying that a small change in \((x, y)\) — moving by an amount \( \begin{bmatrix} h \\ k \end{bmatrix} \)
— changes \( f \) by \( JS(x, y) \begin{bmatrix} h \\ k \end{bmatrix} \), i.e., by multiplication by a constant (which is a \( 2 \times 2 \) matrix in this case). The derivative of \( S \) at \((x, y)\) is this linear transformation. So the derivative is a function that takes a point (namely \((x, y)\)) and produces a linear transformation.

We’ll let \( L(\mathbb{R}^2, \mathbb{R}^2) \) denote the set of all linear transformations from \( \mathbb{R}^2 \) to \( \mathbb{R}^2 \); this notation, for the transformation \( S \), the derivative is \( DS: \mathbb{R}^2 \to L(\mathbb{R}^2, \mathbb{R}^2): (x, y) \mapsto \begin{pmatrix} h & k \\ 2x & 2y \\ 2x - \pi \cos(\pi y) \end{pmatrix} \).

For any particular \((x, y)\), this tells how small changes in the domain (near \((x, y)\)) transform to small changes in the codomain (near \(S(x, y)\)). Notice the the linear transformation depends on \((x, y)\): the rule for transforming changes near \((x, y) = (1, 1)\) is different from the rule near \((x, y) = (3, 0)\).

**Inline Exercise 12.5:** Write out \( DS(1, 1) \) and \( DS(3, 0) \); each of these should be a linear transformation from \( \mathbb{R}^2 \) to \( \mathbb{R}^2 \); verify that the transformations are in fact different.

For our original affine transformation \( T \), the result is qualitatively different; the derivative of \( T \) is

\[
DT: \mathbb{R}^2 \to L(\mathbb{R}^2, \mathbb{R}^2): (x, y) \mapsto \begin{pmatrix} h & k \\ 2 & 3 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} h \\ k \end{pmatrix}.
\]

In this case, the derivatives at \((1, 1)\) and at \((3, 0)\) are exactly the same; both are simply

\[
[h, k] \mapsto \begin{pmatrix} 2 & 3 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} h \\ k \end{pmatrix}.
\]

This says that no matter where you are in \( \mathbb{E}^2 \), if you move in the direction \([1 \ 0]\), then \( T \) will change by \([2 \ 3]\). The derivative is a constant; its value at every point is the same linear map. (The linear map is *not* constant, however!)

The derivative is more than just a constant; its constant value is the transformation that we’ve been calling “the way that \( T \) transforms vectors.”

In summary: for an affine transformation \( T \), the associated vector transform is simply the derivative of \( T \) at any point; to be concrete, the associated vector transformation is \( DT(0, 0) \).

Similarly, the associated covector transformation is the one whose matrix is the inverse-transpose of the matrix for \( DT(0, 0) \).

For projective maps, things are not so simple. The derivative of a projective map is no longer constant, so we can no longer treat displacements at one point as being “the same as” displacements at another point. For example, for the projective transform given by

\[
P(x, y) = \left( \frac{2x}{y + 1}, \frac{x - y}{y + 1} \right)
\]
the derivative of $P$ is

$$DP : \mathbb{E}^2 \to L(\mathbb{R}^2, \mathbb{R}^2) : (x, y) \mapsto \begin{bmatrix} \frac{2}{(y+1)} & -\frac{2x}{(y+1)^2} \\ \frac{1}{(y+1)} & -(x+1)/(y+1)^2 \end{bmatrix} \begin{bmatrix} h \\ k \end{bmatrix}.$$ 

The derivative at $(x, y) = (1, 1)$ is the map

$$\begin{bmatrix} h \\ k \end{bmatrix} \mapsto \begin{bmatrix} 1 \\ 1/2 \end{bmatrix} \begin{bmatrix} 1 \\ -1/2 \end{bmatrix} \begin{bmatrix} h \\ k \end{bmatrix},$$

while the derivative at $(x, y) = (2, 0)$ is the map

$$\begin{bmatrix} h \\ k \end{bmatrix} \mapsto \begin{bmatrix} 2 \\ 1 \end{bmatrix} \begin{bmatrix} 2 \\ -2 \\ 1 \\ -3 \end{bmatrix} \begin{bmatrix} h \\ k \end{bmatrix}.$$ 

Thus moving a one unit to the right from $(x, y) = (1, 1)$ would cause $P$ to change by about $\begin{bmatrix} 1 \\ 1/2 \end{bmatrix}$, while moving one unit to the right from $(2, 0)$ would cause $P$ to change by $\begin{bmatrix} 2 \\ 1 \end{bmatrix}$.

In this context, it no longer makes sense to treat vectors as displacements on the entire Euclidean space; instead, we need to consider displacements at each point separately. We speak of a tangent vector as a pair consisting of both the vector its basepoint, the point from which it originates. Hence for a projective transformation $P$, the associated “vector transformation” has to act on individual tangent vectors, i.e., point-vector pairs. We therefore have a method $\text{TangentVectorTransform}($Point $p$, $\text{Vector}$ $v$) for transforming tangent vectors. (If C# supported multiple-value return, i.e., functions that return two values at the same time, which can then be assigned to two variables, we would have had $\text{TangentVectorTransform}$ return a Point-Vector pair.)

Transforming covectors is similar: the covector transformation varies from point to point. In this case, we've used the name $\text{CovectorTransform}$. 

### 12.7 Other structures

Depending on how you plan to use the linear algebra library, it might make sense to create classes to represent other common geometric entities like rays, lines, planes in 3-space, ellipses and ellipsoids (which are transformed to ellipses and ellipsoids, respectively, by nondegenerate linear and affine maps). A ray, for instance, might be represented by a Point and a direction Vector. It's then natural to define either

```csharp
public static Ray operator*(AffineTransformation2 T, Ray r)
```

or

```csharp
public Ray RayTransform(Ray r)
```
in the AffineTransformation2 class; a good implementation would transform the ray’s Point by \( T \), and would transform its direction Vector and normalize the result, because many computations on rays are easier when their directions are expressed as unit vectors.

**Exercise 12.2:** Create a Ray class to represent a ray in the plane, and build the associate ray-transformation in the AffineTransformation2 class. Do the same for a Line. What constructors should the Line class have? What about a Segment class? What methods should Segment have that Line lacks? Can you do the same thing for making rays, lines, and segments cooperate with the ProjectiveTransformation class? Think about what happens when a ray crosses the line on which a projective transformation is undefined.

**Exercise 12.3:** (Mathematical). General position of the points \( P_i, (i = 1, \ldots, 4) \) was needed to invert the matrix \( B \) in the construction of the PointsToPoints method for projective maps. We also assumed that the points \( Q_i, (i = 1, \ldots, 4) \) were in general position, but this assumption was stronger than necessary. What is the weakest geometric condition on the \( Q_i \) that allows the PointsToPoints transformation to be built?

**Exercise 12.4:** Explain why the two characterizations of general position for four points in the plane — that (a) no one lies on a line passing through another pair and (b) the first three form a nondegenerate triangle, and the fourth is not on the extensions of any of the sides of this triangle — are equivalent. Pay particular attention to the failure cases, i.e., show that if four points fail to satisfy condition (a), they also fail to satisfy condition (b), and vice-versa.

**Exercise 12.5:** Enhance the library by defining one-dimensional transformations as well (LinearTransformation1, AffineTransformation1, ProjectiveTransformation1). The first two classes will be almost trivial. The third is more interesting; include a constructor `ProjectiveTransform1(double p, double q, double r)` that builds a projective map sending 0 to \( p \), 1 to \( q \), and \( \infty \) to \( r \) (i.e., \( \lim_{x \to \infty} T(x) = r \)). From such a constructor it’s easy to build a PointsToPoints transformation.