modelling power, they’re easy to implement, and they lead to computational problems that are easy to solve.

Links

[ Further discussion about least squares problems on Wikipedia ]
https://en.wikipedia.org/wiki/Linear_regression
https://en.wikipedia.org/wiki/Linear_least_squares_(mathematics)

[ More about the Moore–Penrose pseudoinverse ]
https://en.wikipedia.org/wiki/Moore–Penrose_pseudoinverse

Exercises

E7.10 You want to determine whether a coin is fair, so you toss it repeatedly and record the number of times it lands heads. On the first trial, you flip the coin eight times and obtain four heads, which is a heads-to-flips ratio of $\frac{2}{3}$. On subsequent trials you obtain heads- to-flips ratios of $\frac{9}{16}$, $\frac{23}{32}$, $\frac{17}{32}$, and $\frac{20}{40}$. Find the best-fitting linear model $h(x) = mx$ to describe the number of heads in a trial with $x$ flips.

E7.11 Find the best-fitting affine model $y = b + mx$ to the $(x, y)$ data points $(0, 3.9)$, $(1, 3.2)$, and $(2, 1.9)$. Perform all the calculations by hand.

Hint: Find the Moore–Penrose pseudoinverse.

E7.12 Calculate the total squared error $S(m^*) = \|Am^* - \bar{b}\|^2$ of the best-fit linear model obtained in Example 1 (page 361). Use the SymPy calculation at bit.ly/leastsq_ex1 as your starting point.

Hint: The Matrix method .norm() might come in handy.

E7.13 Revisit Example 2 (page 363) and find the total squared error of the best-fit affine model $S(m'^*) = \|Am'^* - \bar{b}\|^2$. You can start from the calculation provided here bit.ly/leastsq_ex2 and extend it.

7.8 Computer graphics

Linear algebra is the mathematical language of computer graphics. Whether you’re building a simple two-dimensional game with stick figures, or a fancy three-dimensional visualization, knowing linear algebra will help you understand the graphics operations that draw pixels on the screen.
In this section, we'll discuss some basic computer graphics concepts. In particular, we'll introduce homogeneous coordinates, a representation for vectors and matrices that uses an extra dimension. Instead of representing a three-dimensional vector as a triple of Cartesian coordinates \((x, y, z)_c\), we'll use a four-dimensional homogeneous coordinates vector \((x, y, z, 1)_h\). Homogeneous coordinates allow us to represent any computer graphics transformation as a matrix-vector product. We've already seen that scalings, rotations, reflections, and orthogonal projections can be represented as matrix-vector products. With homogeneous coordinates, we can represent translations and perspective projections as matrix products, too. That's very convenient, since it enables us to understand all computer graphics operations in terms of matrix multiplications.

Computer graphics is a vast subject, far more extensive than the pages allotted in this book. To keep things simple, we'll focus on two-dimensional graphics, and only briefly touch upon three-dimensional graphics. The goal is not to teach you the commands of computer graphics APIs like OpenGL and WebGL, but to give you the basic math tools for understanding what's happening under the hood.

**Affine transformations**

In Chapter 5 we studied various linear transformations and their representations as matrices. We also briefly discussed the class of affine transformations, which consist of a linear transformation followed by a translation:

\[
\vec{w} = T(\vec{v}) + \vec{d}.
\]

In the above equation, the input vector \(\vec{v}\) is first acted upon by a linear transformation \(T\), and then the output of \(T\) is translated by the displacement vector \(\vec{d}\) to produce the output vector \(\vec{w}\).

In this section, we'll use homogeneous coordinates for vectors and transformations, which allow us to express affine transformations as matrix-vector products in a larger vector space:

\[
\vec{w} = T(\vec{v}) + \vec{d} \iff \vec{W} = A\vec{V}.
\]

If \(\vec{v}\) is an \(n\)-dimensional vector, then its representation in homogeneous coordinates \(\vec{V}\) is an \((n+1)\)-dimensional vector. The \((n+1) \times (n+1)\) matrix \(A\) contains the combined information about both the linear transformation \(T\) and the translation \(\vec{d}\).

**Homogeneous coordinates**

Instead of using a triple of Cartesian coordinates to represent points \(p = (x, y, z)_c \in \mathbb{R}^3\), we’ll use the quadruple \(P = (x, y, z, 1)_h \in \mathbb{R}^4\),
which is a representation of the same point in homogeneous coordinates. Similarly, the vector \( \vec{v} = (v_x, v_y, v_z)_c \in \mathbb{R}^3 \) in Cartesian coordinates corresponds to the vector \( \vec{V} = (v_x, v_y, v_z, 1)_h \in \mathbb{R}^4 \) in homogeneous coordinates. Though there is no mathematical difference between points and vectors, we’ll stick to the language of points as it is more natural for graphics problems.

An interesting property of homogeneous coordinates is that they’re not unique: the vector \( \vec{v} = (v_x, v_y, v_z)_c \) corresponds to a whole set of points in homogeneous coordinates, \( \vec{V} = \{(av_x, av_y, av_z, a)\}_h \), for \( a \in \mathbb{R} \). This makes homogeneous coordinates invariant to scaling:

\[
\begin{bmatrix}
  a \\
  b \\
  c
\end{bmatrix}_c \quad \Leftrightarrow \quad \begin{bmatrix}
  a \\
  b \\
  c \\
  1
\end{bmatrix}_h = \begin{bmatrix}
  5a \\
  5b \\
  5c \\
  5
\end{bmatrix}_h = \begin{bmatrix}
  500a \\
  500b \\
  500c \\
  500
\end{bmatrix}_h.
\]

This is kind of weird, but this extra freedom to rescale vectors arbitrarily leads to many useful applications.

To convert from homogeneous coordinates \( (X, Y, Z, W)_h = (a, b, c, d)_h \) to Cartesian coordinates, we divide each component by the \( W \)-component to obtain the equivalent vector \( (X, Y, Z, W)_h = (\frac{a}{d}, \frac{b}{d}, \frac{c}{d}, 1)_h \), which corresponds to the point \( (x, y, z)_c = \left( \frac{a}{d}, \frac{b}{d}, \frac{c}{d} \right)_c \in \mathbb{R}^3 \).

In the case when the underlying Cartesian space is two-dimensional, the point \( p = (x, y)_c \in \mathbb{R}^2 \) is written as \( P = (X, Y, W)_h = (x, y, 1)_h \) in homogeneous coordinates. The homogeneous coordinates \( (X, Y, W)_h = (a, b, d)_h \) (where \( d \neq 0 \)) represent the point \( (x, y)_c = \left( \frac{a}{d}, \frac{b}{d} \right)_c \in \mathbb{R}^2 \).

This conversion between homogeneous coordinates and Cartesian coordinates can also be understood geometrically, as illustrated in Figure 7.7. The point \( (X, Y, W)_h \) in homogeneous coordinates corresponds to an infinite line in the three-dimensional \( XYZ \)-coordinate space. To obtain the Cartesian coordinates for this point, find the intersection of the infinite line with the plane \( W = 1 \).

To distinguish Cartesian vectors from homogeneous vectors, we’ll use capital letters like \( P, A, B, \ldots \) for points and vectors in homogeneous coordinates, and lowercase letters like \( p, d, b, \ldots \) when referring to points and vectors in Cartesian coordinates.

**Affine transformations in homogeneous coordinates**

Consider the affine transformation that consists of the transformation \( T : \mathbb{R}^2 \to \mathbb{R}^2 \) followed by a translation by \( \vec{d} = (d_1, d_2) \). If \( T \) is represented by the matrix \( M_T = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \), then the affine transfor-
Figure 7.7: A two-dimensional Cartesian point \( (x, y) \), corresponds to an infinite line in a three-dimensional homogeneous coordinates space \( \{(ax, ay, a)_h, a \in \mathbb{R}\} \). To convert a point from homogeneous coordinates to Cartesian coordinates, we must find where the infinite line intersects the plane \( W = 1 \). The homogeneous coordinates of the intersection point will have the form \( (x, y, 1)_h \). The first two components of this point are the Cartesian coordinates \( (x, y) \).

In the homogeneous coordinates representation, the input has an additional component that contains the constant value one, and the matrix has an additional column that operates on this constant component. The result of the matrix-vector product is to add the constants \( d_1 \) and \( d_2 \) to the first and second coordinates:

\[
\begin{bmatrix}
    x' \\
    y'
\end{bmatrix} = \begin{bmatrix}
    m_{11} & m_{12} \\
    m_{21} & m_{22}
\end{bmatrix} \begin{bmatrix}
    x \\
    y
\end{bmatrix} + \begin{bmatrix}
    d_1 \\
    d_2
\end{bmatrix} \iff
\begin{bmatrix}
    x' \\
    y' \\
    1
\end{bmatrix} = \begin{bmatrix}
    m_{11} & m_{12} & d_1 \\
    m_{21} & m_{22} & d_2 \\
    0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
    x \\
    y \\
    1
\end{bmatrix}.
\]

In the homogeneous coordinates representation, the input has an additional component that contains the constant value one, and the matrix has an additional column that operates on this constant component. The result of the matrix-vector product is to add the constants \( d_1 \) and \( d_2 \) to the first and second coordinates:

\[
x' = m_{11}x + m_{12}y + d_1, \quad y' = m_{21}x + m_{22}y + d_2,
\]

which is exactly what translation by \( \vec{d} \) is all about. As you can see, there's nothing fancy about homogeneous coordinates. Make sure you understand the above matrix-vector products and that you're convinced there is no new math here—just the good old matrix-vector product you're familiar with.

**Graphics transformations in 2D**

When points and vectors are represented in homogeneous coordinates, we can express most useful geometric transformations of the form \( T : \mathbb{R}^2 \to \mathbb{R}^2 \) as \( 3 \times 3 \) matrices in homogeneous coordinates. In addition to affine transformations we discussed above, homogeneous coordinates also allow us to perform perspective transformations, which are of central importance in computer graphics. The
most general transformation we can perform using homogeneous coordinates is

\[
\begin{bmatrix}
    x' \\
    y' \\
    w'
\end{bmatrix}
= \begin{bmatrix}
    m_{11} & m_{12} & d_1 \\
    m_{21} & m_{22} & d_2 \\
    p_1 & p_2 & 1
\end{bmatrix}
\begin{bmatrix}
    x \\
    y \\
    w
\end{bmatrix},
\]

where \( M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \) corresponds to any linear transformation, \( \vec{d} = (d_1, d_2) \) corresponds to a translation, and \( (p_1, p_2) \) is a perspective transformation.

**Linear transformations**

Let \( R_\theta \) be the clockwise rotation by the angle \( \theta \) of all points in \( \mathbb{R}^2 \). In homogeneous coordinates, this rotation is represented as

\[
p' = R_\theta(p) \quad \Leftrightarrow \quad P' = M_{R_\theta}P,
\]

where \( M_{R_\theta} \) is the following \( 3 \times 3 \) matrix:

\[
\begin{bmatrix}
    \cos \theta & \sin \theta & 0 \\
    -\sin \theta & \cos \theta & 0 \\
    0 & 0 & 1
\end{bmatrix}.
\]

The \( 2 \times 2 \) submatrix in the top-left corner of matrices in homogeneous coordinates can represent any linear transformation: projections, reflections, scalings, and shear transformations. The following equation shows the homogeneous-coordinates matrix representations of three linear transformations: the reflection through the \( x \)-axis \( M_{R_x} \), an arbitrary scaling \( M_S \), and a shear along the \( x \)-axis \( M_{SH_x} \).

\[
M_{R_x} = \begin{bmatrix}
1 & 0 & 0 \\
0 & -1 & 0 \\
0 & 0 & 1
\end{bmatrix} \quad M_S = \begin{bmatrix}
s_x & 0 & 0 \\
0 & s_y & 0 \\
0 & 0 & 1
\end{bmatrix} \quad M_{SH_x} = \begin{bmatrix}
1 & a & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

**Orthogonal projections**

Projections can also be represented as \( 3 \times 3 \) matrices. The projection onto the \( x \)-axis corresponds to the following representation in homogeneous coordinates:

\[
p' = \Pi_x(p) \quad \Leftrightarrow \quad \begin{bmatrix}
    x' \\
    0 \\
    1
\end{bmatrix} = \begin{bmatrix}
    1 & 0 & 0 \\
    0 & 0 & 0 \\
    0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
    x \\
    y \\
    1
\end{bmatrix}.
\]
Translation

The translation by the displacement vector $\vec{d} = (d_x, d_y)$ corresponds to the matrix

$$M_{T_d} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix}.$$ 

Note the top-left submatrix of $M_{T_d}$ contains the identity matrix; we perform the identity linear transformation and a translation by $\vec{d}$.

Figure 7.8: Illustration of the different transformations on a sample shape. Source: wikipedia File:2D_affine_transformation_matrix.svg

Rotations, reflections, shear transformations, and translations can all be represented as multiplications by $3 \times 3$ matrices. Figure 7.8 shows examples of transformations we can perform using the matrix-vector product in homogeneous coordinates. I hope by now you’re convinced that this idea of adding an extra “constant” dimension to vectors is useful. But wait, there’s more! We haven’t yet seen what happens when we put coefficients in the last row of the matrix.
Perspective projections

The notion of perspective is very important in the world of visual art. For a painting to look realistic, the relative size of the objects pictured must convey their relative distance from the viewer. Distant objects in the scene are painted smaller than objects in the foreground. To transform a three-dimensional scene into a realistic two-dimensional painting, we need to apply a perspective transformation. Rather than give you the general formula for perspective transformations, we’ll derive the matrix representation for perspective transformations from scratch. Trust me, it will be a lot more interesting.

We can understand perspective transformations by tracing imaginary light rays that originate from each corner of an object and travel to the eye of an observer $O$ in a straight line (see Figure 7.9). The image of the perspective projections is created along the path of the light ray, where the ray intersects the projective plane, which represents the screen where the image will form. Since we’re working in $\mathbb{R}^2$, we can draw a picture of what’s going on.

![Figure 7.9: The perspective projection of a square onto a screen. Note how the side of the square that is closer to the observer appears longer than the farther side.](image)

Let’s assume the observer is placed at the origin, $O = (0, 0)$, and let’s compute the perspective transformation that projects points onto the line with equation $y = d$. Under this perspective projection, every point $p = (x, y)$ in the plane maps to a point $p’ = (x’, y’)$ on the line $y = d$. Figure 7.10 illustrates the situation.

The only math prerequisite you need to remember is the general principle for similar triangles: if a triangle with sides $a$, $b$, $c$ is similar to a triangle with sides $a’, b’, c’$, then the ratios of the lengths of the triangles’ sides will be equal:

$$\frac{a'}{a} = \frac{b'}{b} = \frac{c'}{c}.$$

Since the two triangles in Figure 7.10 are similar, we know $\frac{x'}{x} = \frac{d}{y'}$. 
and therefore we directly obtain the expression for \((x', y')\) as follows:

\[
x' = \frac{d}{y} x,
\]
\[
y' = \frac{d}{y} y = d.
\]

This doesn’t look very promising, however, since the expression for \(x'\) contains a division by \(y\). The set of equations is not linear in \(y\), and therefore cannot be expressed as a matrix-product. If only there were some way to represent vectors and transformations that also allowed division by coefficients too.

Let’s analyze the perspective transformation in terms of homogeneous coordinates, and see if we find anything useful:

\[
\begin{bmatrix}
    x' \\
    y' \\
    1
\end{bmatrix} = \begin{bmatrix}
    \frac{d}{y} x \\
    d \\
    1
\end{bmatrix} = \begin{bmatrix}
    x \\
    y \\
    \frac{y}{d}
\end{bmatrix}.
\]

The second equality holds because vectors in homogeneous coordinates are invariant to scalings: \((a, b, c)_h = a(a, b, c)_h\) for all \(a\). We can shift the factor \(\frac{d}{y}\) as we please: \((\frac{d}{y} x, d, 1) = \frac{d}{y}(x, y, \frac{y}{d}) = (x, y, \frac{y}{d})\). In the alternate homogeneous coordinates expression, we’re no longer dividing by \(y\). This means we can represent the perspective transformation as a matrix-vector product:

\[
\begin{bmatrix}
    x' \\
    y' \\
    1
\end{bmatrix} = \begin{bmatrix}
    x \\
    y \\
    \frac{y}{d}
\end{bmatrix} = \begin{bmatrix}
    1 & 0 & 0 \\
    0 & 1 & 0 \\
    0 & \frac{1}{d} & 0
\end{bmatrix} \begin{bmatrix}
    x \\
    y \\
    1
\end{bmatrix}.
\]

Now that’s interesting. By preparing the vector \((X', Y', W')\) with a third component \(W' \neq 1\), we can force each coefficient to be scaled
by \( \frac{1}{w'} \), which is exactly what we need for perspective transformations. Depending on how the coefficient \( W' \) is constructed, different perspective transformations can be obtained.

A perspective projection transformation is a perspective transformation followed by an orthogonal projection that removes some of the vector’s components. The perspective projection onto the line with equation \( y = d \) is the composition of a perspective transformation \( P : \mathbb{R}^3 \rightarrow \mathbb{R}^3 \) followed by an orthogonal projection \( \Pi_x : \mathbb{R}^3 \rightarrow \mathbb{R}^2 \) which simply discards the \( y' \) coordinate:

\[
\begin{bmatrix}
x' \\
1
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & \frac{1}{d} & 0
\end{bmatrix} \begin{bmatrix}
x \\
y \\
1
\end{bmatrix} \Rightarrow \begin{bmatrix}
x' \\
1
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 \\
0 & \frac{1}{d} & 0
\end{bmatrix} \begin{bmatrix}
x \\
y \\
1
\end{bmatrix}.
\]

Note that only the \( x' \) coordinate remains after the projection. This is the desired result since we want only the “local” coordinates of the projection plane \( y = d \) to remain after the projection.

Certain textbooks on computer graphics discuss only the combined perspective-plus-projection transformation \( \Pi_x P \), as described on the right side of the above equation. I prefer to treat the perspective transformation separately from the projection, since it makes the math easier to understand. Also, by including the “depth information” (the \( y \)-coordinates) of the objects we’re projecting we can determine which objects appear in front of others.

**General perspective transformation**

Let’s now look at the general case of a perspective transformation that projects arbitrary points \( p = (x, y) \) onto the line \( ax + by = d \). Again, we assume the observer is located at the origin \( O = (0, 0) \). We want to calculate the coordinates of the projected point \( p' = (x', y') \), as illustrated in Figure 7.11.

To obtain the general perspective transformation, we’ll follow a logical reasoning similar to the special case we considered above based on the properties of similar triangles. Define \( a \) to be the projection of the point \( p \) onto the line with direction vector \( \vec{n} = (a, b) \) passing through the origin. Using the general formula for distances (see Section 4.1), we can obtain the length \( \ell \) from \( O \) to \( a \):

\[
\ell = d(O, a) = \frac{\vec{n} \cdot p}{\|\vec{n}\|} = \frac{ax + by}{\|\vec{n}\|}.
\]
Figure 7.11: The point \( p' = (x', y') \) is the projection of the point \( p = (x, y) \) onto the line with equation \( ax + by = d \). We define points \( a' \) and \( a \) in the direction of line’s normal vector \( \vec{n} = (a, b) \). The distances from the origin to these points are \( \ell' \) and \( \ell \) respectively. We have \( \ell'/\ell = x'/x = y'/y \).

Similarly, we define \( \ell' \) to be the length of the projection of \( p' \) onto \( \vec{n} \):

\[
\ell' = d(O, a') = \frac{\vec{n} \cdot p'}{||\vec{n}||} = \frac{ax' + by'}{||\vec{n}||} = \frac{d}{||\vec{n}||}.
\]

The last equation holds true because the point \( p' \) is located on the line with equation \( ax + by = d \).

By the similarity of triangles, we know the ratio of lengths \( x'/x \) and \( y'/y \) must equal the ratio of orthogonal distances \( \ell'/\ell \):

\[
\frac{x'}{x} = \frac{y'}{y} = \frac{\ell'}{\ell} = \frac{d}{ax + by}.
\]

We can use this fact to express the coordinates \( x' \) and \( y' \) in terms of the original \( x \) and \( y \) coordinates. As in the previous case, expressing points in homogeneous coordinates allows us to arbitrarily shift scaling factors:

\[
\begin{bmatrix}
  x' \\
  y' \\
  1
\end{bmatrix} = \begin{bmatrix}
  \ell'x \\
  \ell'y \\
  1
\end{bmatrix} = \begin{bmatrix}
  x \\
  y \\
  1
\end{bmatrix} = \begin{bmatrix}
  x \\
  y \\
  ax + by
\end{bmatrix}.
\]

The last expression is linear in the variables \( x \) and \( y \), therefore it has a matrix representation:

\[
\begin{bmatrix}
  x' \\
  y' \\
  1
\end{bmatrix} = \begin{bmatrix}
  X' & Y' & W'
\end{bmatrix} \begin{bmatrix}
  1 & 0 & 0 \\
  0 & 1 & 0 \\
  a & b & 0
\end{bmatrix} \begin{bmatrix}
  x \\
  y \\
  1
\end{bmatrix}.
\]

This is the most general example of a perspective transformation. The “scaling factor” \( W' \) is a linear combination of the input coordi-
nates: \( W' = \frac{a}{d} x + \frac{b}{d} y \). Observe that by setting \( a = 0 \) and \( b = 1 \), we recover the perspective transformation for the projection onto the line \( y = d \).

**Graphics transformations in 3D**

Everything we saw in the previous section about two-dimensional transformations also applies to three-dimensional transformations. A three-dimensional Cartesian coordinate triple \( (x, y, z) \in \mathbb{R}^3 \) is represented as \( (x, y, z, 1) \in \mathbb{R}^4 \) in homogeneous coordinates. Transformations in homogeneous coordinates are represented by \( 4 \times 4 \) matrices. An affine transformation is represented by the matrix

\[
A = \begin{bmatrix}
    m_{11} & m_{12} & m_{13} & d_1 \\
    m_{21} & m_{22} & m_{23} & d_2 \\
    m_{31} & m_{32} & m_{33} & d_3 \\
    0 & 0 & 0 & 1 \\
\end{bmatrix},
\]

and the perspective transformation onto the line \( ax + by + cz = d \) with the observer at the origin is expressed as the matrix

\[
P = \begin{bmatrix}
    1 & 0 & 0 & 0 \\
    0 & 1 & 0 & 0 \\
    0 & 0 & 1 & 0 \\
    \frac{a}{d} & \frac{b}{d} & \frac{c}{d} & 0 \\
\end{bmatrix}.
\]

These should look somewhat familiar to you. Affine transformations, projections, and perspective transformations are the same in 3D as in 2D, except we’re now working in a four-dimensional homogeneous coordinates space.

The best part about being able to apply the homogeneous coordinates representation to all transformations is that we can compose transformations together, in a way that’s similar to building with LEGO. We’ll learn about this in the next section.

**3D graphics programming**

Inside every modern computer is a special-purpose processor, called the **graphics processing unit (GPU)**, which is dedicated to computer graphics operations. Modern GPUs can have thousands of individual graphics processing units called shaders, and each shader can perform millions of linear algebra operations per second. Think about it; thousands of processors working in parallel to calculate matrix-vector products for you—that’s a lot of linear algebra calculating power!
The reason we need so much processing power is because 3D models are made of thousands of little polygons. Drawing a 3D scene (also known as rendering) involves performing linear algebra manipulations on all these polygons. This is where the GPU comes in. The job of the GPU is to translate, rotate, and scale the polygons of the 3D models by placing them into the scene, and then computing what the scene looks like when projected to the two-dimensional window (the screen) through which you're observing the virtual world. This transformation—from the model coordinates to world coordinates, and then to screen coordinates (pixels)—is carried out in a graphics processing pipeline.

![Diagram](image.png)

**Figure 7.12:** A graphics processing pipeline for drawing 3D objects on the screen. A 3D model is composed of polygons expressed with respect to a coordinate system centred on the object. The model matrix positions the object in the scene, the view matrix positions the camera in the scene, and finally the projection matrix computes what should appear on the screen.

We can understand the graphics processing pipeline as a sequence of matrix transformations: the model matrix $M$, the view matrix $V$, and the projection matrix $\Pi_s$. The GPU applies this sequence of operations to each of the object's vertices, $(x, y, z, 1)_s$, to obtain the pixel coordinates, $(x', y')_s$, of the vertices on the screen:

$$
\begin{bmatrix}
x' \\
y' \\
z \\
1
\end{bmatrix}_s = \Pi_s VM
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}_m \Rightarrow (x, y, z, 1)_m M^T V^T \Pi_s^T = (x', y')_s.
$$

In the context of computer graphics, it is customary to represent the graphics processing pipeline in the "transpose picture," so that vertex data flows from left to right as in Figure 7.12. Instead of representing vertices as column vectors multiplied by matrices $M$, $V$, and $\Pi_s$ from the left, we represent vertices as row vectors multiplied by matrices $M^T$, $V^T$, and $\Pi_s^T$ from the right. All the reasoning remains the same, and all the transformation matrices described above still work; you just might need to transpose them if you're using them in a program.

Finally, a comment on efficiency. It is not necessary to compute the matrix-vector products with $M^T$, $V^T$, and $\Pi_s^T$ for each vertex. It's much more efficient to pre-compute a combined transformation
matrix, $C^T = M^T V^T \Pi_s^T$, and apply $C^T$ to each of the coordinates of the 3D objects. Similarly, when adding another object to the scene, only the model matrix needs to be modified, while the view and projection matrices remain the same.

**Practical considerations**

We discussed homogeneous coordinates and the linear algebra transformations used for computer graphics. This is the essential theory you’ll need in order to get started with computer graphics programming. The graphics pipelines used in modern 3D software involve a few more steps than the simplified version shown in Figure 7.12. We’ll now discuss some points and practical considerations for programming computer graphics on real GPUs:

- There are actually *two* graphics pipelines at work in the GPU. The *geometry pipeline* handles the transformation of polygons that form the 3D objects. A separate *texture pipeline* controls the graphics patterns that fill the polygons. The final step in the rendering process combines the outputs of the two pipelines.

- The model and view matrices can be combined to form a single model-view matrix that converts object coordinates to camera coordinates.

- Not all objects in the scene need to be rendered. We don’t need to render objects that fall outside the camera’s viewing angle. We can also skip objects that are closer than the *near plane* or farther than the *far plane* of the camera. Though the scene could extend infinitely, we’re only interested in rendering the subset of the scene that we want displayed on the screen, which we call the *view frustum*. See the illustration in Figure 7.13.

![Figure 7.13](image.png)

*Figure 7.13:* The perspective transformation that projects a three-dimensional model onto a two-dimensional screen surface. Note only a subset of the scene (the *view frustum*) is drawn, which lies between the *near plane* and the *far plane* within the *field of view*.

I encourage you to pursue the subject of computer graphics on your own. There are excellent free resources for learning OpenGL and WebGL on the web.
Discussion

Homogeneous coordinates and projective geometry are powerful mathematical techniques with deep connections to many advanced math subjects. For the purpose of this section, we explored the subject from an engineering perspective—as a handy trick for working with computer graphics using matrix-vector products. Homogeneous coordinates have many other applications that we did not have time to discuss. We’ll briefly mention some of these additional applications here, with the hope of inspiring you to research the subject further.

Homogeneous coordinates are convenient for representing planes in $\mathbb{R}^3$. The plane with general equation $ax + by + cz = d$ is represented by the vector $\mathbf{N} = (a, b, c, -d)$ in homogeneous coordinates. This is a natural way to express planes. Instead of describing the plane’s normal vector $\mathbf{n}$ separately from the constant $d$, we can represent the plane by an “enhanced” normal vector $\mathbf{N} \in \mathbb{R}^4$:

$$\text{Plane with } \mathbf{n} = (a, b, c) \text{ and constant } d \quad \Rightarrow \quad \mathbf{N} = (a, b, c, -d).$$

This is really cool because we can now use the same representation for both vectors and planes, and perform vector operations between them. Yet again, we find a continuation of the “everything is a vector” theme we’ve encountered throughout this book.

What good is representing planes in homogeneous coordinates? For starters, it allows us to verify whether any point $\mathbf{p}$ lies in the plane $\mathbf{N}$ by computing the dot product $\mathbf{N} \cdot \mathbf{p}$. The point $\mathbf{p}$ lies inside the plane with normal vector $\mathbf{N}$ if and only if $\mathbf{N} \cdot \mathbf{p} = 0$. Consider the point $\mathbf{P}_1 = (0, 0, \frac{d}{c}, 1)$ that lies in the plane $ax + by + cz = d$; we can verify that

$$\mathbf{N} \cdot \mathbf{P}_1 = (a, b, c, -d) \cdot (0, 0, \frac{d}{c}, 1) = 0.$$ 

It’s also possible to easily obtain the homogeneous coordinates of the plane $\mathbf{N}$ that passes through any three points, $\mathbf{P}$, $\mathbf{Q}$, and $\mathbf{R}$. We’re looking for a vector $\mathbf{N}$ that is perpendicular to all three points. We can obtain $\mathbf{N}$ using a generalization of the cross product, computed using a four-dimensional determinant:

$$\mathbf{N} = \begin{vmatrix} \hat{e}_1 & \hat{e}_2 & \hat{e}_3 & \hat{e}_4 \\ p_x & p_y & p_z & p_w \\ q_x & q_y & q_z & q_w \\ r_x & r_y & r_z & r_w \end{vmatrix},$$

where $\{\hat{e}_1, \hat{e}_2, \hat{e}_3, \hat{e}_4\}$ is the standard basis for $\mathbb{R}^4$. I bet you haven’t seen a four-dimensional cross product before—this stuff is wild! See the links below if you want to learn more about homogeneous coordinates, projective spaces, or computer graphics.
7.9 Cryptography

Cryptography is the study of secure communication. The two main tasks that cryptographers aim to achieve are private communication (no eavesdroppers) and authenticated communication (no impersonators). Using algebraic operations over finite fields $\mathbb{F}_q$, it’s possible to achieve both of these goals. Math is the weapon for privacy!

The need for private communication between people has existed long before the development of modern mathematics. Thanks to modern mathematical techniques, we can now perform cryptographic operations with greater ease, and build cryptosystems with security guaranteed by mathematical proofs. In this section, we’ll discuss the famous one-time pad encryption technique invented by Gilbert Vernam. One-time pad encryption is important because Claude Shannon proved it is absolutely secure. In order to understand what that means precisely, we’ll first need some context about the concepts studied in the field of cryptography.

Context

The secure communication scenarios we’ll discuss in this section involve three parties:

- Alice is the message sender
- Bob is the message receiver
- Eve is the eavesdropper

Alice wants to send a private message to Bob, but Eve has the ability to see all communication between Alice and Bob. You can think of Eve as a Facebook administrator, or an employee of the Orwellian, privacy-invading web application *du jour*. To defend against Eve, Alice will *encrypt* her messages before sending them to Bob, using a secret key only Alice and Bob have access to. Eve will be able to capture the encrypted messages (called *ciphertexts*) but they will be unintelligible to her because of the encryption. Assuming Bob receives the messages from Alice, he’ll be able to *decrypt* them using