Chapter 5

2-D Transformational Geometry

Much of computer graphics is built around manipulating geometric data. We have already seen that we can represent a 2-D shape as one or more polylines connecting vertices in 2-D. We can also represent a 3-D shape as a mesh of triangles whose vertices are assigned positions in 3-D space and whose triangles are assigned triples of these vertices. Now we can start to manipulate such a shape, by streaming through its vertices and performing operations on their positions.

We will start with two-dimensional shapes and transformations, and extend these two 3-D in the next chapter. We will also focus on three simple shape operations: scale, which changes a shape's size (and sometimes its proportions), translation, which moves a shape someplace else, and rotation, which turns a shape. These shape operations are all examples of *affine* transformations. Affine transformations have to follow two rules: (1) if three points are in a line, they remain in a line after the transformation, and (2) if one line is a fraction of the length of a second line's length, then it remains that same fraction of the second line's length after the transformation. Because of these rules, affine transformations preserve straight lines and flat planes, so they are well suited operations for 2-D polyline and 3-D polygonal mesh shapes. To transform these shapes, we only need to transform the vertices to new positions, instead of transforming every point connecting the vertices.

5.1 Canvas Coordinates

We will demonstrate 2-D transformations in the "canvas" coordinate system. This is a natural coordinate system for manipulating 2-D planar figures. The canvas coordinate system is a planar Cartesian coordinate system extending across an axis-aligned square from (-1,-1) to (1,1). This square forms the boundary of the coordinate system, such that geometry

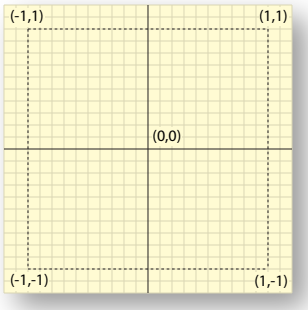

Figure 5.1: The canvas coordinate system.

that lies outside of the boundary (and also the outside portion of boundary-crossing geometry) is *clipped* (not drawn).

This canvas coordinate system is a natural coordinate system for plotting functions, though its domain may need to be extended beyond the default square, shrunk to zoom in on a portion of the domain, or moved to plot a function away from the origin. We can use 2-D transformations for these operations as this chapter later shows. The default canvas coordinate system is also a convenient region for 2-D shapes whose coordinates do not exceed a unit.

This canvas coordinate system also exists in the OpenGL vertex pipeline, where it is confusingly called "window" coordinates. Here a "window" refers to a viewing window, as opposed to a display screen window in a windowed operating system user interface, which corresponds to a rectangular array of pixels called the viewport. Hence we avoid confusion, by avoiding the overloaded term "window."

5.2 Homogeneous Coordinates

We will represent a 2-D position (x, y) in the plane as a homogeneous column vector $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$. The extra third coordinate is called the homogeneous coordinate and will simplify the representation of affine transformations. To keep these column vectors from breaking up the page, we'll write them using a transpose operator as $[x\ y\ 1]^T$ in paragraph text, but we'll continue using the untransposed format for equations.

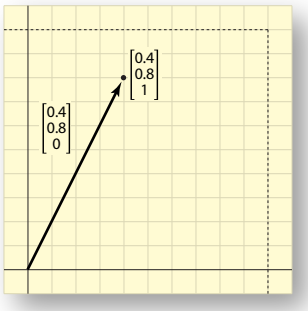

Figure 5.2: $(4, 8)$ as a point and a vector.

We can use this extra homogeneous coordinate to indicate the difference between a point (a position in the plane) and a vector (an offset from one point to another point). For example, the plane origin is the 2-D point represented as $[0\ 0\ 1]^T$. If we subtract the origin from the point position $[x\ y\ 1]^T$, we convert it into the offset vector $[x\ y\ 0]^T$ by elementwise subtraction

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}. \tag{5.1}$$

Hence $[x\ y\ 1]^T$ represents the point (x, y) in the plane, whereas $[x\ y\ 0]^T$ represents an offset vector x units horizontally and y units vertically. Similarly we can add an offset vector $[a\ b\ 0]^T$ to the point position $[x\ y\ 1]^T$ to get a new point position $[x + a\ y + b\ 1]^T$.

This is an important distinction because affine transformations can have different effects on points than on vectors. For example, moving a point changes its coordinates, but moving a vector does not. In this form, one can also add and subtract any number of offset vectors, but can only add or subtract these offset vectors from at most one point position. Two point

positions can be only subtracted to form an offset vector, whereas other operations, such as addition of two point positions, do not make sense. For now, this extra homogeneous element will always be zero or one, but in the next chapter when we talk about perspective, the homogeneous element can take on other values and these rules regarding homogeneous point and vector arithmetic no longer apply.

Recall that we represent a shape using a list of N (shared) vertex positions $\{v_0, v_1, \ldots, v_{N-1}\}$ and a list of how these vertices are connected to form polygons. These vertices become a list of column vectors, so for the vertices of a polyline shape in 2-D we have

$$\{v_i\} = \left\{ \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix}, \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}, \ldots, \begin{bmatrix} x_{N-1} \\ y_{N-1} \\ 1 \end{bmatrix} \right\}.$$

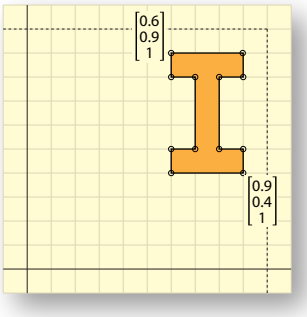

Figure 5.3: A 2-D shape described by a closed polygon with vertices: {(0.6,0.9), (0.6,0.8), (0.7,0.8), (0.7,0.5), (0.6,0.5), (0.6,0.4), (0.9,0.4), (0.9,0.5), (0.8,0.5), (0.8,0.8), (0.9,0.8), (0.9,0.9)}.

When reading in vertices from a model, the extra homogeneous coordinate is usually one, so we don't need to explicitly store it with the model, but when we load each point in the model for transformation, we will need to add this extra homogeneous coordinate to convert the point into this homogeneous column vector format.

We will apply affine transformations to these column vectors using a matrix-vector product. The matrix will always be square, so that mutliplying a transformation matrix times a column vector produces a new column vector with the same number of elements. So a 3×3 transformation matrix applied to a 3-element homogeneous column vector representing a 2-D point will generate a new 3-element homogeneous column vector representing the transformed 2-D point, in general

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + c \\ dx + ey + f \\ 1 \end{bmatrix}. \tag{5.2}$$

By properly selecting the matrix values $a, b, \ldots, f$ these matrices can be configured to apply any affine transformation, to change the size, proportions, position or orientation of a shape.

5.3 Scale

If we want to scale (expand or contract) a polygonal shape around a point, then we need to change the distance from that point to each of its vertices by some factor. A (uniform) scale transformation using a scaling factor s changes the distance from every vertex to the origin by the same factor s, by multiplying each of the vertex coordinates by that factor. If $s > 1$ then the shape is enlarged, or if $s < 1$ then the shape shrinks. We implement

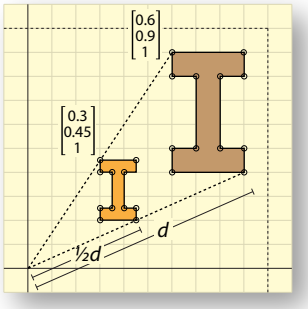

Figure 5.4: Shape scaled by $s = \frac{1}{2}$.

this transformation with a 3×3 diagonal scale matrix as

$$\begin{bmatrix} s & & \\ & s & \\ & & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} sx \\ sy \\ 1 \end{bmatrix}. \tag{5.3}$$

(To avoid visual clutter, we omit matrix elements that are zero.)

We want to keep the homogeneous element of the column vector the same (set to one), so the bottom row of the transformation matrix ignores the spatial coordinates of the input and simply reproduces the homogeneous element.

We can generalize scale transformation into a stretch or squash operation that changes the proportions, specifically the aspect ratio, of the shape. For such a non-uniform scale in the plane, we need separate scaling factors: h for the horizontal proportion of the scale, and v for the vertical proportion. If $h = 1$ and $v > 1$ then the shape remains the same width and is stretched vertically. If $h = 1$ and $v < 1$ then the width remains constant but the shape is squashed, as in Figure 5.5. This general scale is implemented by the diagonal matrix

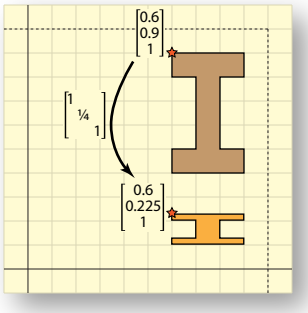

Figure 5.5: Shape scaled by $v = \frac{1}{4}$ vertically.

$$\begin{bmatrix} h & & \\ & v & \\ & & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} hx \\ vy \\ 1 \end{bmatrix}. \tag{5.4}$$

5.4 Translation

If we want to move a shape, we can add an offset vector to its vertices. This is called *translation,* and a translation by (a, b) means that each vertex (x, y) is moved to a new position $(x', y') = (x + a, y + b)$. Ordinarily one cannot add a constant value to a vector element using the matrix-vector product, but the homogeneous coordinate allows us to do this. Hence we implement translation as a matrix and apply it using the matrix vector product

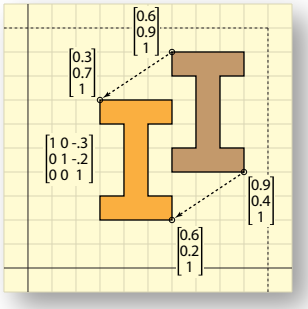

Figure 5.6: Shape translated by the offset vector $(-3, -2)$.

$$\begin{bmatrix} 1 & & a \\ & 1 & b \\ & & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + a \\ y + b \\ 1 \end{bmatrix}. \tag{5.5}$$

The translation coefficients a and b are placed in the third column of the translation matrix, and these coefficients are multiplied by the homogeneous coordinate of the column vector and added to the appropriate coordinate of the column vector.

5.5 Relative Scale

Representing both of these transformations as a matrix makes it easier and more efficient to compose multiple transformations. Suppose we want to

uniformly scale a shape around some other point, say (a, b), instead of the origin, as in Figure 5.7. We do this by first translating the shape (and the space around the shape) such that the point (a, b) is moved to the origin, then scale by a factor of s, and then translate the result such that the origin is moved back to the point (a, b), as illustrated in Figure 5.8.

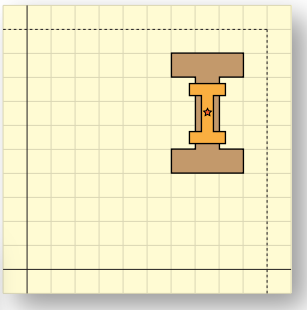

Figure 5.7: Scaling a shape about the point (7.5,6.5).

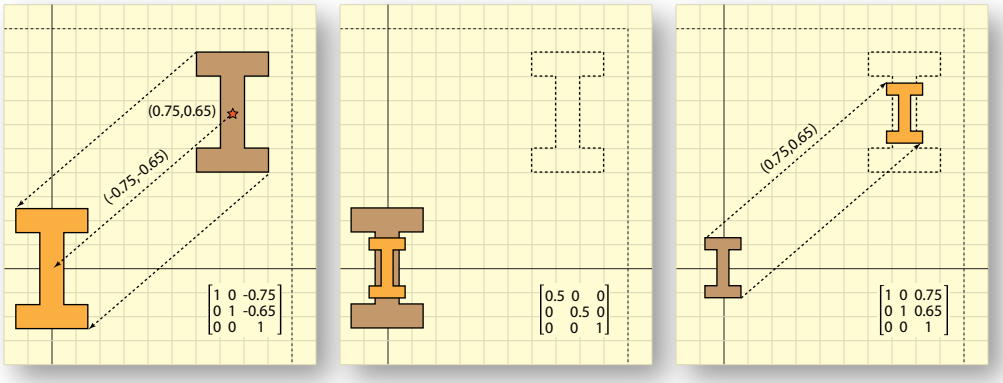

Figure 5.8: Three steps to scale a shape about a center point: translate the center point (7.5,6.5) to the origin, scale by 1/2 about the origin, then translate the origin (and the shape) back to the centerpoint (7.5,6.5).

Using vector algebra, this operation would be

$$s((x, y) + (-a, -b)) + (a, b) = (sx + (1 - s)a, sy + (1 - s)b). \qquad (5.6)$$

Using transformation matrices, this becomes

$$T_{a,b}\ S_s\ T_{-a,-b} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & & a \\ & 1 & b \\ & & 1 \end{bmatrix} \begin{bmatrix} s & & \\ & s & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & -a \\ & 1 & -b \\ & & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} s & & (1-s)a \\ & s & (1-s)b \\ & & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \qquad (5.7)$$

Consider the left-hand side of this equation first. The order of these matrices might be the opposite of ones intuition. The first operation $T_{-a,-b}$, the translation by $(-a, -b)$, is the third matrix of the product, whereas operation $T_{a,b}$, translate by (a, b), is the first matrix in the product but the last operation to be applied. It is easier to think of these matrices as functions, and to think of the the left-hand side as the composed function $T_{a,b}(S_s(T_{-a,-b}(x, y)))$, where the innermost (rightmost) function $T_{-a,-b}$ gets applied first. But unlike functions, we eliminate the nested parentheses with the understanding that the rightmost operation is applied first. These "functions" that transform homogeneous vectors by matrix-vector

multiplication are formally known as *linear operators.* The application of linear operators likewise avoids the use of parentheses, resembling the matrix representation at the beginning of (5.7).

When applied to a single point, the matrix representation (5.7) is more complicated and less efficient than the simpler vector algebra (5.6). However the associative property of matrix multiplication allows us to multiply the three matrices together first and then perform a matrix-vector product of the resulting matrix with our column position vector. While this does not save any operations for a single column vector, recall that we are performing this operation on every vertex in our shape. Videogame shapes can have tens of thousands of vertices, and scanned models of real-world object, such as Michelangelo's statues in Florence, can have upwards of 100 billion vertices. We can first multiply all of these transformation matrices together to form a single matrix product, and then apply the transformation as a single matrix-vector product to each vertex in the shape.

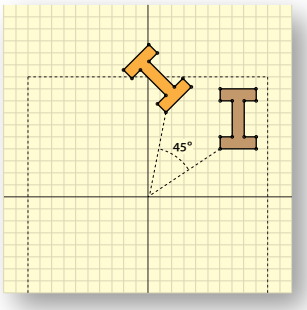

Figure 5.9: Shape rotated by 45° about the origin.

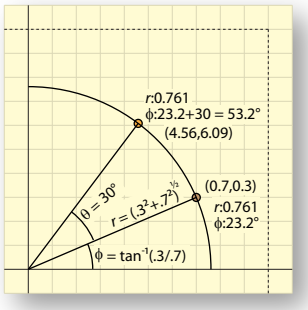

Figure 5.10: The point $(7, 3)$ rotated 30° to the point $(4.56, 6.09)$ computed using polar coordinates.

5.6 Rotation

Another important affine transformation useful for modeling and animation is rotation. Given a point (x, y) in the plane, we want to find its position (x', y') after it has been rotated by an angle of θ counterclockwise about the origin. The 3×3 homogeneous transformation matrix that performs this rotation is

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & \\ \sin\theta & \cos\theta & \\ & & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{5.8}$$

We can derive this transformation by first converting the point (x, y) into polar coordinates (r, ϕ) where r is the distance from the origin and ϕ is the counterclockwise angle the point makes about the origin with respect to the positive x axis. We can convert between cartesian coordinates (x, y) and polar coordinates (r, ϕ) as

$$r = ||(x, y)|| = \sqrt{x^2 + y^2}, \tag{5.9}$$

$$\phi = \arctan \frac{y}{x} \tag{5.10}$$

$$x = r\cos\phi \tag{5.11}$$

$$y = r\sin\phi. \tag{5.12}$$

Rotating the polar coordinates point (r, ϕ) by an angle θ about the origin yields a point with polar coordinates $(r, \phi + \theta)$ and Cartesian coordinates

$$x' = r\cos(\phi + \theta), \tag{5.13}$$

$$y' = r\sin(\phi + \theta). \tag{5.14}$$

We can rotate a point directly in Cartesian coordinates, avoiding the conversion to polar coordinates, by using a convenient trigonometric identity

$$\cos(\phi+\theta) = \cos\phi\cos\theta - \sin\phi\sin\theta, \quad (5.15)$$

$$\sin(\phi+\theta) = \sin\phi\cos\theta + \cos\phi\sin\theta, \quad (5.16)$$

that most of us forgot shortly after we first learned it. Plugging these angle sum formulae into the rotation result (5.13,5.14) gives us the form found in the aforementioned rotation matrix (5.8)

$$\begin{aligned} x' &= r(\cos\phi\cos\theta - \sin\phi\sin\theta) \\ &= x\cos\theta - y\sin\theta, \end{aligned} \quad (5.17)$$

$$\begin{aligned} y' &= r(\sin\phi\cos\theta + \cos\phi\sin\theta), \\ &= y\cos\theta + x\sin\theta. \end{aligned} \quad (5.18)$$

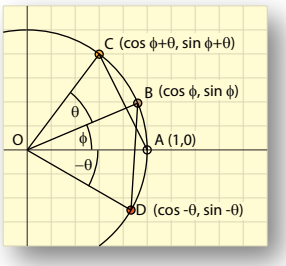

Figure 5.11: Quick proof of the sum of angles formula for cosines.

$||AC||^2 = (\cos\phi+\theta-1)^2+\sin^2\phi+\theta$
$= 2-2\cos\phi+\theta.$

$||BD||^2 = (\cos\phi-\cos-\theta)^2+$
$(\sin\phi-\sin-\theta)^2$
$= 2-2\cos\phi\cos\theta+2\sin\phi\sin\theta.$

$||AC|| = ||BD||$ □.

5.7 Inverse Transformations

Most of the transformations we have described have opposite transformations, such that the transformation followed by its opposite will return a shape to its original state (e.g. proportions, position and orientation). When it exists, the opposite of a transformation with matrix M is given by the matrix inverse M^{-1}. Hence a transformation followed by its opposite (inverse) yields no transformation at all, $M^{-1}M = I$. The identity matrix

$$I = \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} \quad (5.19)$$

is a diagonal matrix consisting of ones, and its application to a shape leaves the shape unchanged. It is a uniform scale by factor one, a translation by zero units, a rotation by zero degrees.

A translation matrix can always be inverted, and results in a translation in the opposite direction

$$T^{-1} = \begin{bmatrix} 1 & & a \\ & 1 & b \\ & & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & & -a \\ & 1 & -b \\ & & 1 \end{bmatrix}, \quad (5.20)$$

and $T \times T^{-1} = I$.

A rotation matrix can also always be inverted, and results in a rotation in the opposite direction. Because of the structure of a rotation matrix, its inverse is also its transpose. Hence if R is a rotation matrix with angle

θ then

$$\begin{aligned} R^{-1} &= \begin{bmatrix} \cos\theta & -\sin\theta & \\ \sin\theta & \cos\theta & \\ & & 1 \end{bmatrix}^{-1} \\ &= \begin{bmatrix} \cos -\theta & -\sin -\theta & \\ \sin -\theta & \cos -\theta & \\ & & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta & \sin\theta & \\ -\sin\theta & \cos\theta & \\ & & 1 \end{bmatrix} \\ &= R^T, \end{aligned} \tag{5.21}$$

since $\cos -\theta = \cos\theta$ and $\sin -\theta = -\sin\theta$.

A scale matrix S might not be invertible. If S is invertible then its inverse S^{-1} is also a scale matrix whose scaling factors are reciprocals of the original scale factors

$$S^{-1} = \begin{bmatrix} h & & \\ & v & \\ & & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \frac{1}{h} & & \\ & \frac{1}{v} & \\ & & 1 \end{bmatrix}. \tag{5.22}$$

If S is not invertible, then it is a *projection.* For example a non-uniform scale by $h = 1$ in the x direction and $v = 0$ in the y direction flattens a shape onto the x-axis. The inverse (5.22) of such a matrix would result in a divide by zero and so is undefined. In other words, we cannot unflatten a shape. If such a non-invertible matrix is multiplied by others to form a composite transformation, then the resulting matrix product will also be non-invertible.

If an arbitrary 3×3 transformation matrix M is invertible, then its inverse is

$$\begin{aligned} M^{-1} &= \begin{bmatrix} a & b & e \\ c & d & f \\ 0 & 0 & 1 \end{bmatrix}^{-1} \\ &= \frac{1}{ad-bc} \begin{bmatrix} d & -b & bf-de \\ -c & a & ce-af \\ 0 & 0 & ad-bc \end{bmatrix} \\ &= \frac{1}{\det M} \operatorname{adj} M. \end{aligned} \tag{5.23}$$

Hence the inverse of a matrix M is the *adjugate* of M scaled by the reciprocal of the determinant of M. The adjugate always exists, but if the determinant is zero, then its reciprocal is undefined and the inverse of M does not exist. For a 3×3 matrix M representing a 2-D affine transformation, we can confirm M is invertible by ensuring $ad \neq bc$.

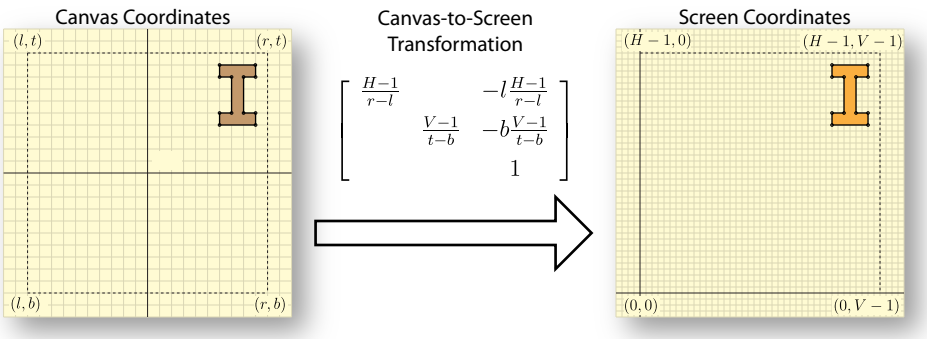

Figure 5.12: The canvas-to-screen transformation maps vertices from their positions in the canvas to the corresponding locations of pixels on the screen.

The adjugate is informally also known as the "adjoint" which is often denoted $M^\dagger$. The adjugate is sometimes called the "pseudoinverse" because it is a uniform scale factor away from the inverse, and does not require any division. In some cases we need the opposite of a transformation but do not care if the opposite includes an arbitrary uniform scale. In such cases we compute the adjugate and forget the determinant, which is cheaper and more robust.

5.8 The Canvas to Screen Transformation

The final stage of the vertex pipeline is a transformation from canvas coordinates to screen coordinates. In the vertex pipeline, the canvas coordinates usually extend from $(-1,-1)$ to $(1,1)$ but they can be set to extend to any region of the plane, say from (l,b) to (r,t), to delineate the axis-aligned rectangular domain for whatever is being plotted, as shown in Figure 5.12.

The screen coordinate system sets the coordinates of individual pixels at integer (x,y) locations. When the vertices of a triangle are converted from canvas coordinates to screen coordinates, we get the positions of the three corners of the triangle (which may or may not lie precisely on pixel positions). The rasterizer will fill in the triangle pixels based on these three corner positions of its vertices in screen coordinates.

The general canvas-to-screen transformation consists of a translation that maps the left-bottom corner (l,b) to the origin, followed by a scale that transforms the canvas width $r-l$ into the screen horizontal resolution $H-1$ and the canvas height $t-b$ into the screen vertical resolution $V-1$.

$$C2S = \begin{bmatrix} \frac{H-1}{r-l} & & \\ & \frac{V-1}{t-b} & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & & -l \\ & 1 & -b \\ & & 1 \end{bmatrix} = \begin{bmatrix} \frac{H-1}{r-l} & & -l\frac{H-1}{r-l} \\ & \frac{V-1}{t-b} & -b\frac{V-1}{t-b} \\ & & 1 \end{bmatrix} \tag{5.24}$$

In the vertex pipeline, the canvas is set to extend from $(-1,-1)$ to $(1,1)$. The canvas-to-screen transformation can be specialized to this fixed canvas size, and simplifies to

$$C2S_{-1,1} = \begin{bmatrix} \frac{H-1}{2} & & \frac{H-1}{2} \\ & \frac{V-1}{2} & \frac{V-1}{2} \\ & & 1 \end{bmatrix}. \tag{5.25}$$

5.9 Plotting Fields

In order to plot a planar field of values over a rectangular domain we first need to iterate over all of the pixels in screen coordinates and convert each pixel location into its corresponding position in canvas coordinates. We can accomplish this by inverting the canvas-to-screen transformation matrix $C2S$ to find the screen-to-canvas transformation matrix $S2C$.

While we can invert $C2S$ directly to find the matrix $S2C = C2S^{-1}$, it is easier to decompompose $C2S = S \times T$ into its product of a scale S and a translation T. Then the inverse

$$S2C = C2S^{-1} = (S \times T)^{-1} = T^{-1} \times S^{-1}. \tag{5.26}$$