# 6

# Affine transformations

Transformations, i.e. linear invertible automorphisms, are used to map a picture or model into another one with different size, position and orientation. We will see that each useful geometric transformation may be reduced to an invertible linear transformation of a suitable linear space. Hence, given a basis, transformations are represented by means of squared invertible matrices, called *transformation matrices*. The main aim of this chapter is to study the structure and properties of matrices of "elementary" transformations. The interpretation of matrix multiplication as a geometric operator that maps a *point locus* (i.e. either a picture or model) into another one, is one of the great old ideas of computer graphics. In this chapter we first study the transformations of the 2D Euclidean space, then we see how they generalize to 3D and to greater dimensions. We also discuss how affine transformations are implemented as dimension-independent automorphisms by `PLaSM`. Several programming examples are given in the chapter.

## 6.1 Preliminaries

Some preliminary concepts needed in basic computer graphics are discussed in this section, including the definition of linear and affine transformations, the notation and conventions used for points, vectors, angles and positive rotations, and the important ideas underlying the use of homogeneous coordinates.

### *6.1.1 Transformations*

A geometric transformation is defined as a one-to-one mapping of a point space to itself, which preserves some geometric relations of figures. For example:

1. *linear* transformations map lines *for the origin* to lines for the origin;
2. *affine* transformations map lines to lines preserving the *parallelism*;
3. uniform *scaling* transformations map point sets preserving the *similarity*;
4. translations and rotations, called *rigid* transformations, preserve the *congruence* of point sets.

*Geometric Programming for Computer-Aided Design* Alberto Paoluzzi
 ISBN 0-471-89942-9

**Linear transformations** A linear transformation is an invertible linear map of a vector space to itself. Linear transformations of a vector space $\mathcal{V}$ earned this name because the transformed $\boldsymbol{p}^*$ of each $\boldsymbol{p}$ on the line segment joining $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ belongs to the segment joining the transformed images $\boldsymbol{p}_1^*$ and $\boldsymbol{p}_2^*$. In fact, let

$$\boldsymbol{p} = \alpha_1 \boldsymbol{p}_1 + \alpha_2 \boldsymbol{p}_2 \qquad 0 \leq \alpha_1, \alpha_2, \qquad \alpha_1 + \alpha_2 = 1$$

and let $\boldsymbol{T} : \mathcal{V} \to \mathcal{V}$ be a linear transformation. By linearity of $\boldsymbol{T}$ we have

$$\boldsymbol{p}^* = \boldsymbol{T}\boldsymbol{p} = \boldsymbol{T}(\alpha_1 \boldsymbol{p}_1 + \alpha_2 \boldsymbol{p}_2) = \alpha_1 \boldsymbol{T}\boldsymbol{p}_1 + \alpha_2 \boldsymbol{T}\boldsymbol{p}_2 = \alpha_1 \boldsymbol{p}_1^* + \alpha_2 \boldsymbol{p}_2^*$$

This property is very important from a practical viewpoint, because it allows transformation of pictures or models, which are point loci with infinite elements, by just transforming a finite number of points, and in particular those called *vertices*, at the intersection of picture or model *edges*.

**Affine transformations** An *affine map* is a map between two affine spaces which preserves the affine structure, i.e. maps lines to lines and preserves the parallelisms. Let $\mathcal{A}$ and $\mathcal{B}$ be two affine spaces with underlying vector spaces $\mathcal{V}$ and $\mathcal{W}$, respectively. Then a map $\boldsymbol{A} : \mathcal{A} \to \mathcal{B}$ is affine if, for each $x \in \mathcal{A}$ and $v \in \mathcal{V}$

$$\boldsymbol{x} + \boldsymbol{v} \mapsto \boldsymbol{A}(\boldsymbol{x} + \boldsymbol{v}) = \boldsymbol{A}(\boldsymbol{x}) + \boldsymbol{T}(\boldsymbol{v}),$$

where $\boldsymbol{T}$ is linear, and is called the *linear part* of $\boldsymbol{A}$.

An *affine transformation* is an invertible affine map of an affine space to itself. An affine map is invertible only if its linear part is invertible. The affine transformations of an affine space $\mathcal{A}$ form a group with respect to function composition, called the *affine group* of $\mathcal{A}$.

**Notation** In the following sections we indifferently speak of either affine transformations — also called *tensors* in this book — and of their matrices, and normally we use the same notation. Tensor composition will occasionally be denoted by symbol concatenation, with the same notation used for matrix multiplication. An explicit denotation of functional composition is instead used within the `PLaSM` sources, where we need to distinguish between tensors, i.e. functions in the affine group, and their matrices.

### *6.1.2 Points and vectors*

According to current practice, we give the coordinate representation of elements in a point space, e.g. of point $\boldsymbol{p} \in \mathbb{E}^2$, as a *column vector* $\begin{pmatrix} x \\ y \end{pmatrix}$. The coordinate representation of the point $\boldsymbol{p}^* = \boldsymbol{A}(\boldsymbol{p})$ generated by the action of a tensor $\boldsymbol{A}$ on a point $\boldsymbol{p}$ is hence given by *left-hand* multiplication by tensor matrix $[\boldsymbol{A}]$.

Actually, for sake of simplicity, we will often use the notation $\boldsymbol{A}$ to denote both the tensor and its matrix. We shall use the full notation $[\boldsymbol{A}]$ for a tensor matrix just occasionally, with the aim of remarking its use.

So, we have:

$$\boldsymbol{p}^* = \begin{pmatrix} x^* \\ y^* \end{pmatrix} = \boldsymbol{A}\ \boldsymbol{p} = \begin{pmatrix} a & c \\ b & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + cy \\ bx + dy \end{pmatrix}.$$

The convention of representing points with column vectors and transformations with left-hand matrix multiplication was adopted in graphics quite recently. Hence, several graphics books represent points as *row vectors* and transformations as *right-hand* matrix multiplications. E.g.:

$$\boldsymbol{p}^* = \begin{pmatrix} x^* & y^* \end{pmatrix} = \boldsymbol{p}\ \boldsymbol{A}^T = \begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} ax + cy & bx + dy \end{pmatrix}$$

Notice the use of the transposed transformation matrix, and remember that the transposition of a matrix product is the product of transposed matrices, in the reverse order:

$$(\boldsymbol{S}_2\ \boldsymbol{S}_1\ \boldsymbol{p})^T = \boldsymbol{p}^T\ \boldsymbol{S}_1^T\ \boldsymbol{S}_2^T$$

### *6.1.3 Orientations and rotations*

In studying the space transformations of 3D models, we need to first of all agree on orientation of rotations and angles.

In particular, the vectors $\boldsymbol{e}_1, \boldsymbol{e}_2, \boldsymbol{e}_3$ in a Cartesian basis can be mutually oriented as the first three fingers of either the left or the right hand. The basis is called *left-hand* or *right-hand*, accordingly.

In most engineering and design disciplines, right-hand Cartesian bases are used, with the $x$ and $y$ axes contained on the image plane and the $z$ axes perpendicular and directed towards the observer. In a left-hand basis the $z$ axis has an opposite orientation, entering the image plane.

Given a basis $(\boldsymbol{e}_i)$, the rotations are considered *positive* that move the basis vectors as follows:

$$\boldsymbol{e}_1 \to \boldsymbol{e}_2, \quad \boldsymbol{e}_2 \to \boldsymbol{e}_3, \quad \boldsymbol{e}_3 \to \boldsymbol{e}_1.$$

In right-hand bases the positive rotations (and positive angles) are counter-clockwise oriented, whereas in left-hand bases the positive rotations are oriented clockwise. In order to find the sign of rotation, the observer should look at as the rotation axis, see Figure 6.1.

### *6.1.4 Homogeneous coordinates*

We know that several transformations of figures are linear. Linear transformations include scaling, rotation, reflection and shearing. They can be applied to some picture or model by matrix multiplication of picture (model) "vertices" times some suitable matrix. We also know that translations are conversely not linear, since they are applied by adding some constant vector to picture (model) vertices.

It is computationally inconvenient that the algebraic operation needed to apply different transformations is not always the same. This prevents from easy combining

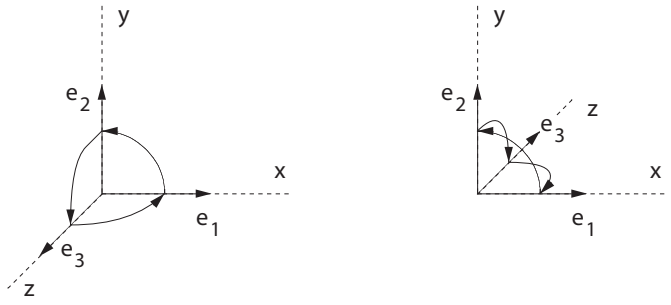


**Figure 6.1** Basis orientation and positive rotations: (a) right-hand basis (b) left-hand basis

different transformations to some object, by first multiplying the corresponding transformation matrices and by then applying the resulting product matrix to object vertices.

A solution to this problem was discovered by using *normalized homogeneous coordinates* rather than standard Cartesian coordinates. In particular, three-dimensional homogeneous coordinates are used for 2D pictures, whereas four-dimensional homogeneous coordinates are used for 3D models. More generally, $(n+1)$-dimensional coordinates are needed for $n$D models.

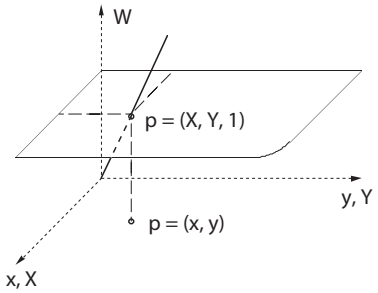


**Figure 6.2** Homogeneous coordinates of 2D plane

Homogeneous coordinates define a bijective mapping between the set of points of Cartesian plane and the set of straight lines passing for the origin $\boldsymbol{o}$ of the 3D space, see Figure 6.2. The origin is subtracted from such lines, considered as point sets, so that the mapping between plane points and space lines becomes one-to-one.

In this mapping $E^2 \rightarrow E^3$ each point $\begin{pmatrix} x \\ y \end{pmatrix} \in E^2$ is represented as a vector $\begin{pmatrix} X \\ Y \\ W \end{pmatrix} \in E^3$, with $W \neq 0$, such that $x = X/W, y = Y/W$. Notice that the same plane point is also represented by each vector $\lambda \begin{pmatrix} X \\ Y \\ W \end{pmatrix}$, where $\lambda \in \mathbb{R}$ and $\lambda \neq 0$.

The reverse mapping from lines to points is hence very simple. In order to return

from an homogeneous point (or, better, vector)

$$\boldsymbol{p}' = \begin{pmatrix} X \\ Y \\ W \end{pmatrix}$$

to its corresponding Cartesian point

$$\boldsymbol{p} = \begin{pmatrix} x \\ y \end{pmatrix},$$

two divisions by the homogeneous coordinate $W$ are needed. In order to avoid such computation it is sufficient to associate the plane point $\begin{pmatrix} x \\ y \end{pmatrix}$ to the so-called *normalized* homogeneous representation $\begin{pmatrix} X \\ Y \\ 1 \end{pmatrix}$, for which it is

$$x = X, \qquad y = Y.$$

In the remainder of this book we use only lower-case letters for coordinates, so that the generic plane point will be denoted, in homogeneous normalized coordinates, as $\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$.

## 6.2 2D Transformations

In this section we discuss the properties of Cartesian plane $(\mathbb{E}^2, \boldsymbol{o}, \{\boldsymbol{e}_1, \boldsymbol{e}_2\})$, and the structure of the *elementary* transformation matrices. In several figures the Cartesian axis are labeled both with $x, y$ and with $x^*, y^*$, in order to represent the coordinates of both the domain and range of the discussed mapping.

### *6.2.1 Translation*

A plane translation is a mapping $\boldsymbol{T} : \mathbb{E}^2 \to \mathbb{E}^2$ where a fixed vector $\boldsymbol{t} = \begin{pmatrix} m \\ n \end{pmatrix}$ is added to each point $\boldsymbol{p} = \begin{pmatrix} x \\ y \end{pmatrix}$, so that

$$\boldsymbol{p}^* = \boldsymbol{T}(\boldsymbol{p}) = \boldsymbol{p} + \boldsymbol{t} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} m \\ n \end{pmatrix} = \begin{pmatrix} x+m \\ y+n \end{pmatrix}.$$

Notice that $\boldsymbol{t}$ gives exactly the image of the origin in such a transformation. This movement of the origin implies that a translation is not a linear transformation, and hence that it cannot be directly represented by the product with some suitable matrix.

We see here that the translation becomes linear when using homogeneous coordinates. In fact, the translation which maps a point $\boldsymbol{p}$ in

$$\boldsymbol{p}^* = \boldsymbol{p} + \boldsymbol{t},$$

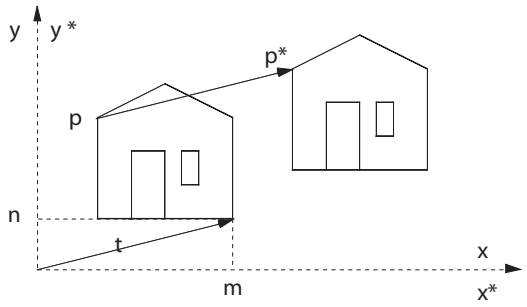


**Figure 6.3** 2D translation

with $\boldsymbol{t} = \begin{pmatrix} m & n \end{pmatrix}^T$, can be expressed by using homogeneous coordinates as:

$$\boldsymbol{p}^* = \boldsymbol{T}\,\boldsymbol{p} = \begin{pmatrix} 1 & 0 & m \\ 0 & 1 & n \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x+m \\ y+n \\ 1 \end{pmatrix}$$

It is easy to see, looking at Figure 6.2 and to the structure of matrix $\boldsymbol{T}$, that by using three-dimensional homogeneous coordinates a translation of $\mathbb{E}^2$ is reduced to an elementary shearing (see Section 6.2.5) of $\mathbb{E}^3$.

**`PLaSM` representation of translation** The elementary geometric transformations, i.e. translation, scaling, rotation and shearing, are defined in `PLaSM` by means of *higher level functions*, which require a double application to integer and real parameters to return the desired *transformation tensor*, which must be in turn applied to some geometric object to return the transformed object.

In particular, the application of a third-order transformation operator (say `T`, `S` or `R`) to either one or more integers, which *specify the coordinates* to be affected by the transformation, returns a more specific function, which must be applied to a suitable number of real parameters in order to return a tensor. The integer numbers are called *specificators*. The real numbers are called (transformation) *parameters*. Translation tensors are generated by `PLaSM` expressions:

```
T:specificators:parameters
```

where *specificators* are either one integer or a sequence of integers and *parameters* are either one real or a sequence of reals, accordingly. Notice that *specificators* and *parameters* sequences of translations must have the same length. The token `T` is a `PLaSM` keyword, and cannot be redefined by the user.

According to their mathematical definition, transformation tensors are functions. Hence they can be composed with other functions, or applied to language expression that generate geometric objects, i.e. polyhedral complexes.

The translation tensors generated by such language expressions can be either composed by standard composition operator or applied to some language expression which evaluates to a polyhedral complex. E.g., in order to apply a translation with vector $\boldsymbol{t} = \begin{pmatrix} 10.1 & 11.2 \end{pmatrix}^T$ to the polyhedron `pol`, we write:

```
T:<1,2>:<10.1,11.2>:pol
```

In order to gain a full understanding, remember that multiple applications associate to left, so that the previous expression is evaluated as:

```
((T:<1,2>):<10.1,11.2>):pol;
```

As a further example, the translations $\boldsymbol{T}(l,m), \boldsymbol{T}(l,0), \boldsymbol{T}(0,m) \in Aff\ \mathbb{R}^2$ can be defined and applied as either:

```
T:<1,2>:<l,m>:pol      or      (T:1:l ~ T:2:m):pol;
```

**Example 6.2.1 (2D house)**
A very simple 2D polyhedral complex `House2D`, shown in Figure 6.4, is defined in Script 6.2.1. It is generated by aggregation of three 2D polyhedra respectively denoted `wall`, `door` and `window`. Each part is defined in local coordinates, with the local origin positioned in its lower left point. Two suitable 2D translation tensors are applied to `door` and `window` respectively, in order to relocate them in the Cartesian frame of `wall`.

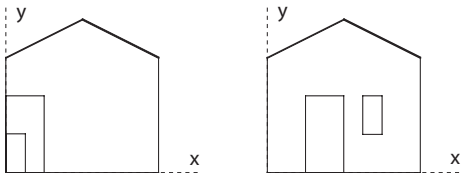


**Figure 6.4** 2D house: (a) without translations of parts (b) with translations

**Script 6.2.1**

```
DEF House2D = STRUCT:< wall, T:1:2:door, T:<1,2>:<5,2>:window >
WHERE
   wall = MKPOL:<<<0,0>,<8,0>,<0,6>,<8,6>,<4,8>>,<1..5>,<<1>>>,
   door = CUBOID:<2,4>,
   window = CUBOID:<1,2>
END;
```

Two translation tensors are also applied to the second and third instances of the formal parameter `Object`, of polyhedral type, in the definition of function `triplet` given in Script 6.2.2. The value generated by evaluation of expression `triplet:House2D` is shown in Figure 6.5. The code is written with the aim of noting that the value generated by the evaluation of the expression `T:1:12` is a transformation tensor, i.e. a *function*.

The `triplet` function could also be written more compactly, by using a proper `FL` style, as:

```
DEF triplet = STRUCT ~ [ID, T:1:12, T:1:12 ~ T:1:12];
```

The `triplet` definition in Script 6.2.2 is said to use a $\lambda$-style, where formal parameters are given in the function *head* and referenced in the function *body*. The proper `FL`

**Script 6.2.2**

```
DEF triplet (Object::IsPol) = STRUCT:
   < Object, transl:Object, (transl ∼ transl):Object >
WHERE
   transl = T:1:12
END;

triplet:House2D;
```

style is instead without parameters, as discussed in Section 2.1. The two definitions are equivalent. Clearly, to choose a programming style is not only a stylistic matter. There are good reasons for both choices, but a discussion of this topic is beyond the scope of this book.

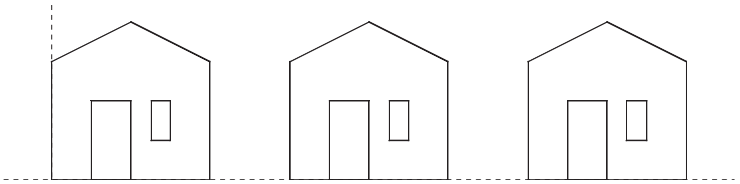

**Figure 6.5** Aggregation of three 2D houses

### *6.2.2 Scaling*

A *scaling* $\boldsymbol{S}$ is a transformation tensor represented by a diagonal matrix with positive coefficients, so that we have:

$$\boldsymbol{p}^* = \boldsymbol{S}\ \boldsymbol{p} = \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax \\ by \end{pmatrix}, \qquad a, b > 0$$

If $a, b > 1$, then $\boldsymbol{S}$ produces a *dilation* of pictures; conversely, if $a, b < 1$ then it produces a *compression* of pictures. A scaling with $a = b = 1$ is an *identity* mapping. Examples of 2D scaling on the first and second coordinate directions are given in Figures 6.6a and 6.6b, respectively.

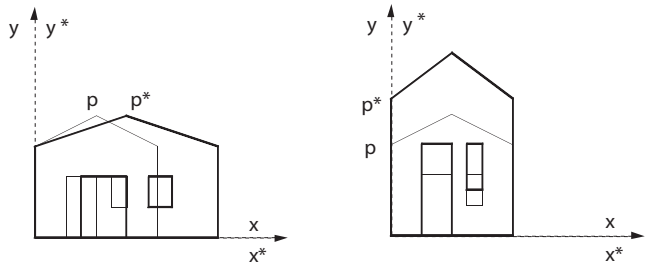


**Figure 6.6** 2D scaling: (a) about the $x$ coordinate (b) about the $y$ coordinate

The matrices of *uni-directional* scaling tensors, named $\boldsymbol{S}_x$ and $\boldsymbol{S}_y$, differ from the

identity matrix for just one of the diagonal coefficients:

$$\boldsymbol{p}^* = \boldsymbol{S}_x\ \boldsymbol{p} = \begin{pmatrix} a & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax \\ y \end{pmatrix}$$

$$\boldsymbol{p}^* = \boldsymbol{S}_y\ \boldsymbol{p} = \begin{pmatrix} 1 & 0 \\ 0 & b \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ by \end{pmatrix}.$$

A generic scaling transformation (matrix) can be obtained by composition (product) of uni-directional scaling transformations (matrices):

$$\begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix} = \begin{pmatrix} a & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & b \end{pmatrix}.$$

When $a = b$ the scaling is said to be *uniform*. The action of an uniform scaling is shown in Figure 6.7. In particular, the picture shows the action of a scaling matrix that doubles both the coordinates of picture points. We note that, in this transformation:

1. the size of all line segments is doubled;
2. the image $\boldsymbol{p}^*$ of each point $\boldsymbol{p}$ is located on the line joining $\boldsymbol{p}$ to the origin;
3. the transformed picture is not only enlarged, but also moved away from the origin.

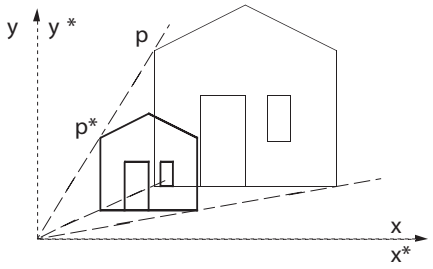


**Figure 6.7** The action of a uniform scaling tensor

The normalized homogeneous matrix $\boldsymbol{S}' \in \mathbb{R}^3_3$ of the 2D scaling tensor can easily be derived from the corresponding non-homogeneous matrix $\boldsymbol{S} \in \mathbb{R}^2_2$, by adding a unit row and a unit column:

$$\boldsymbol{p}^* = \boldsymbol{S}'\boldsymbol{p} = \begin{pmatrix} \boldsymbol{S} & \boldsymbol{0} \\ \boldsymbol{0} & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} ax \\ by \\ 1 \end{pmatrix}$$

**`PLaSM` representation of scaling** Scaling tensors are generated by the evaluation of `PLaSM` expressions

`S:`*`specificators`*`:`*`parameters`*

where *specificators* and *parameters* have the same type we discussed for translation tensors. The token `S` is a `PLaSM` keyword, and cannot be redefined by the user.

**Example 6.2.2 (2D trees)**
Example 6.2.1 is continued here, by adding a very schematic tree to the 2D house there developed. For this purpose we use the function `Circle` given in Script 1.6.1. This function generates a polygonal approximation of the circle of given radius centered in the origin.

Then the circle is translated and aggregated to a thin rectangle to generate an idealized tree shape, generated by `MyTree`. This one is a `PLaSM` function, parametrized on the shape height `h`, which generates a specific tree model when applied to an actual parameter value. Such a tree generator is instanced twice, using a translation, to give the `Trees` object.

Finally, the function `triplet` given in Script 6.2.2 is used to generate the model shown in Figure 6.8.

**Script 6.2.3**

```
DEF Leaves (radius::IsReal) = Circle:radius:<18,1>;

DEF MyTree (h::IsReal) = STRUCT:
   < T:1:(-:h/48):(CUBOID:<h/24,h/3>), T:2:(2*h/3):(Leaves:(h/3)) >;

DEF Trees = STRUCT:<MyTree:9, T:1:2:(MyTree:11) >;
DEF HouseTrees = STRUCT:< House2D, T:1:-0.75:Trees >;

triplet:HouseTrees;
```

Note that we could not use the token `TREE`, since this one is a reserved `PLaSM` keyword, which denotes a primitive operator with a semantics similar to that of `INSR` and `INSR` operators. See Section 2.1.2.

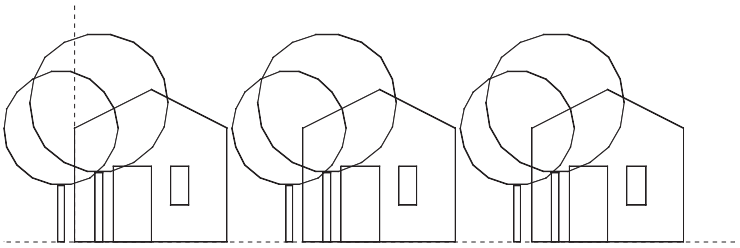

**Figure 6.8** Geometric value generated by the evaluation of the `PLaSM` expression `triplet:HouseTrees`

### *6.2.3 Reflection*

The *reflection* — sometimes called *mirroring* in graphics — about a coordinate axis, is a linear transformation defined by a matrix generated by setting to $-1$ one of the diagonal coefficients of the identity matrix. Hence a reflection strongly resembles a scaling transformation. Two elementary reflections $\boldsymbol{M}_x$ and $\boldsymbol{M}_y$ can be given in the

$\mathbb{E}^2$ plane:

$$\boldsymbol{M}_x = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}, \qquad \boldsymbol{M}_y = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

As always, the normalized homogeneous representation of such transformations is obtained by adding a unit row and column to their matrices:

$$\boldsymbol{M}'_x = \begin{pmatrix} \boldsymbol{M}_x & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix}, \qquad \boldsymbol{M}'_y = \begin{pmatrix} \boldsymbol{M}_y & \mathbf{0} \\ \mathbf{0} & 1 \end{pmatrix}.$$

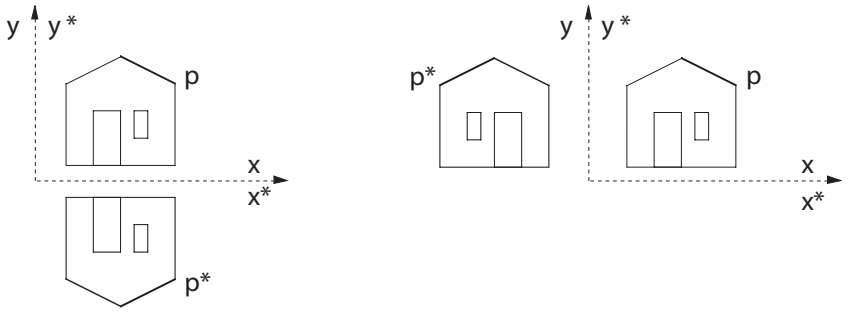


**Figure 6.9** Elementary reflections of plane: (a) about the $x$ axis (b) about the $y$ axis

Clearly, the effect of a reflection tensor is to change the sign of just one coordinate of points. In Figure 6.9 is shown the action of both elementary reflections of the plane.

**`PLaSM` representation of reflection** As we said, reflections resemble scaling. So it is easy to implement such tensors by using the predefined operator `S`:

$\boldsymbol{M}_x \equiv$ `S:1:-1`
$\boldsymbol{M}_y \equiv$ `S:2:-1`

The same implementation may be used for elementary reflections of 3D space.

**Example 6.2.3**
The previous example is continued here, by adding symmetry to the scene. This is done by reflecting the object `HouseTrees` and aggregating the original and the reflected object instances. The result of the evaluation of Script 6.2.4 is shown in Figure 6.10. A new implementation of the `triplet` function is given, where the $x$ size of the formal parameter `Obj` is used to compute the translation tensor.

### *6.2.4 Rotation*

A rotation of the plane about the origin is a linear mapping that moves each point $\boldsymbol{p} \in \mathbb{E}^2$ to a point $\boldsymbol{p}^* = \boldsymbol{R}(\boldsymbol{p})$ along an arc of circumference centered in the origin and with constant angle $\alpha$.

**Script 6.2.4**

```
DEF Mirror (d::IsIntPos)(Obj::IsPol) = (STRUCT ~ [S:d:-1, ID]):Obj;
DEF triplet (Obj::IsPol) = (STRUCT ~ [ID, T:1:x, (T:1:x ~ T:1:x)]):Obj
WHERE
   x = SIZE:1:Obj
END;

triplet:(Mirror:1:HouseTrees)
```

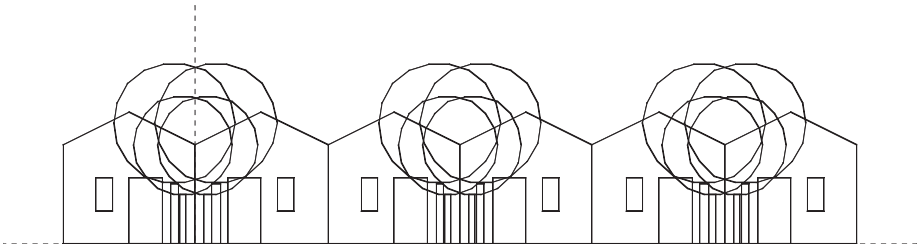

**Figure 6.10** triplet of models generated by the expression `triplet:(Mirror:1:HouseTrees)`

The matrix of a rotation tensor is easily computed by considering the mapping of basis vectors $\{\boldsymbol{e}_i\}$. In particular, we can write that $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ are mapped to $\boldsymbol{e}_1^*$ and $\boldsymbol{e}_2^*$, respectively, where $\boldsymbol{R}$ is the unknown rotation matrix:

$$\begin{pmatrix} \boldsymbol{e}_1^* & \boldsymbol{e}_2^* \end{pmatrix} = \boldsymbol{R} \begin{pmatrix} \boldsymbol{e}_1 & \boldsymbol{e}_2 \end{pmatrix}.$$

Looking at Figure 6.11 we can write, more explicitly:

$$\begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} = \boldsymbol{R} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},$$

from which $\boldsymbol{R}$ is trivially derived.

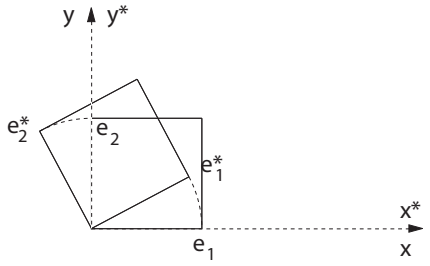


**Figure 6.11** Rotation of the unit square built on the standard basis vectors

We note that the fixed point of this mapping is the origin, and that the rotation matrix depends on the rotation angle $\alpha$. The normalized homogeneous matrix $\boldsymbol{R}' \in Lin\ \mathbb{R}^3$ of a plane rotation is obtained from the non-homogeneous matrix $\boldsymbol{R} \in Lin\ \mathbb{R}^2$ in the standard way:

$$\boldsymbol{p}^* = \boldsymbol{R}'\boldsymbol{p} = \begin{pmatrix} \boldsymbol{R} & \boldsymbol{0} \\ \boldsymbol{0} & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x\ \cos\alpha + y\ \sin\alpha \\ -x\ \sin\alpha + y\ \cos\alpha \\ 1 \end{pmatrix}$$

**`PLaSM` representation of rotation** A rotation is called *elementary* when the set of fixed points of this transformation is a coordinate subspace. Tensors of elementary rotations are generated by the evaluation of `PLaSM` expressions such as

```
R:<i,j>:α
```

where `<i,j>` is a pair of integers that denotes the coordinate pair affected by the transformation, and the real $\alpha$ is the parameter of the transformation, i.e. the rotation angle, given in radians units.

The same syntax is used to specify elementary rotation tensors both in $E^2$ and in higher-dimensional spaces. Such specification of rotation is truly dimension-independent. Consider, in fact, that a rotation in $\mathbb{E}^n$, e.g. $\boldsymbol{R}_{ij}(\alpha)$, is defined as one of the $\binom{n}{2}$ isometries of $E^n$ which keep fixed a coordinate subspace of dimension $n-2$ (and co-dimension 2), with equations

$$x_k = 0, \qquad 1 \leq k \leq n,\ k \neq i, j.$$

The two non-constant coordinates are changed by the tensor matrix according to the pattern of *sin* and *cos* functions previously seen in the 2D case.

Hence in $E^2$ we have the unique operator

```
R:<1,2> : IR → Lin IR^2
```

such that, for each $\alpha \in \mathbb{R}$, the rotation tensor `R:<1,2>:`$\alpha$ is returned. Conversely, in $E^3$ there are $\binom{3}{2} = 3$ different operators

```
R:<1,2> : IR → Lin IR^3,
R:<2,3> : IR → Lin IR^3,
R:<1,3> : IR → Lin IR^3,
```

which return a rotation tensor when applied to a real number, which is interpreted as the rotation angle. Analogously, in $E^4$ we have $\binom{4}{2} = 6$ different elementary rotation operators of this kind. Non-elementary rotations can be obtained by composition of suitable elementary rotations, as discussed in Section 6.3.2 for the 3D case, where the set of fixed points is an arbitrary axis.

**Example 6.2.4 (2D car)**
In this example we generate a group of simplified 2D cars. We also rotate such a model with a variable angle. The single car model is generated by the `PLaSM` object named `car` in Script 6.2.5 and shown in Figure 6.12. The used `Circle` generating function is given in Script 2.2.5.

A row of integer length `n` of cars is generated by the function `carQueue`. Also a function `rotatedCarQueue` is given in Script 6.2.6, which generates a row of `n` cars on a "hillside" of any slope, specified by the formal parameter `degrees`. This function

**Script 6.2.5 (Car model)**

```
DEF car = (T:2:0.5 ∼ STRUCT):
   < body, T:1:1.5:wheel, T:1:6:wheel >
WHERE
   body = MKPOL:<verts, cells, pols>,
   verts = <<0,0>,<3,0>,<7,0>,<6,2>,<4,2>,<3,1>,<1,1>>,
   cells = <<1,2,6,7>,<2,3,4,5,6>>,
   pols = <<1,2>>,
   wheel = S:<1,2>:<0.5,0.5>:(Circle:1:<18,1>)
END;
```

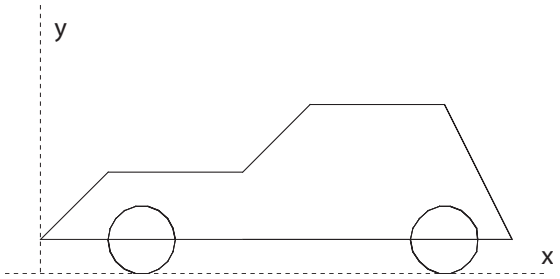


**Figure 6.12** 2D model of a simplified car

rotates the row of cars according to the hillside angle `alpha`. The scene generated by the expression `rotatedCarQueue:5:8` is shown in Figure 6.13.

A function `InclinedTriple`, where the mirrored pairs of houses given in previous examples are here translated also in the $y$ direction, is given in Script 6.2.7. The 2D models produced by this function are easily combined with those produced by the function `rotatedCarQueue`. The resulting scene is shown in Figure 6.14.

**Script 6.2.6 (Rotated car row)**

```
DEF carQueue (n::IsInt) = (STRUCT ∼ ##:n):< car, T:1:(1.2*SIZE:1:car) >;

DEF rotatedCarQueue (n::IsInt)(degrees::IsReal) =
   STRUCT:< basis, R:<1,2>:alpha:(carQueue:n) >
WHERE
   basis = MKPOL:<<<0,0>,<x,0>,<x,y>>,<<1,2,3>>,<<1>>>,
   x = (SIZE:1:(carQueue:n)) * (COS:alpha),
   y = (SIZE:1:(carQueue:n)) * (SIN:alpha),
   alpha = degrees * PI/180
END;

rotatedCarQueue:5:8
```

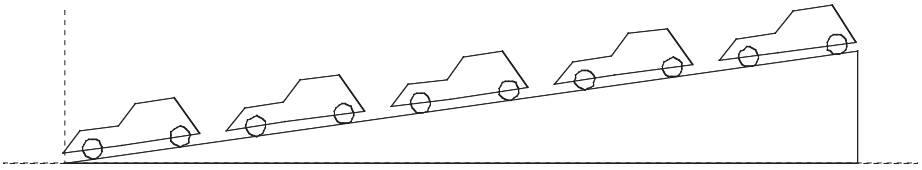

**Figure 6.13** Car row on the hillside

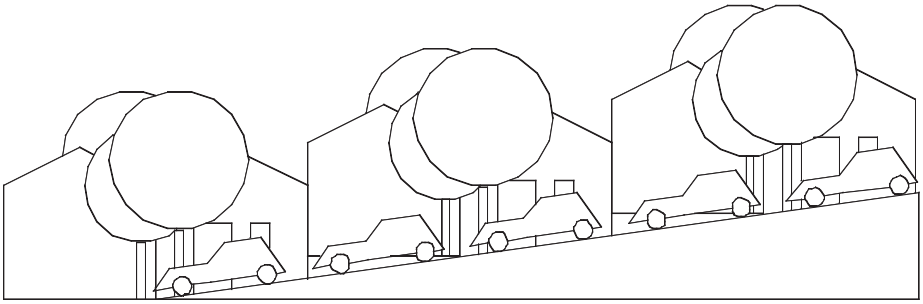

**Figure 6.14** 2D model generated by the expression `STRUCT:< InclinedTriple:8:(Mirror:1:HouseTrees), rotatedCarQueue:5:8 >;`

**Script 6.2.7 (2D scene)**

```
DEF InclinedTriple (degrees::IsReal)(Object::IsPol) = STRUCT:
   < Object, transf:Object, (transf ~ transf):Object >
WHERE
   transf = T:<1,2>:<x, x * TAN:(PI*degrees/180)>,
   x = SIZE:1:Object
END;

STRUCT:<
   InclinedTriple:8:(Mirror:1:HouseTrees),
   rotatedCarQueue:5:8
>;
```

*6.2.5 Shearing*

Let us consider the plane as a bundle of straight lines perpendicular to a coordinate axis. An elementary 2D shearing is a tensor which maps line points to other points of the same line, in such a way that:

1. all points of a line translate by the same vector, i.e. the line translates by the vector;
2. only one line (the coordinate axis belonging to the bundle) is identically mapped, i.e. is not translated;
3. the translation of each line is proportional to its distance from the fixed line.

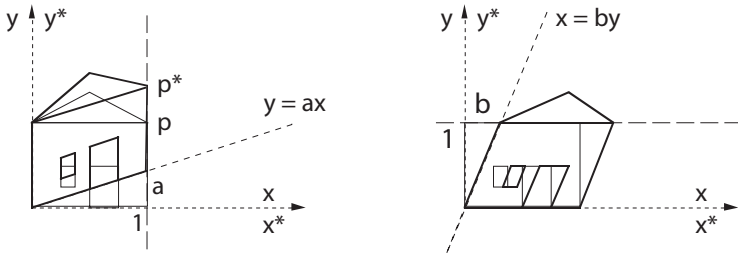


**Figure 6.15** (a) Action of a shearing $\boldsymbol{H}_x$ normal to $x$ axis (b) Action of a shearing $\boldsymbol{H}_y$ normal to $y$ axis

In other words, a shearing tensor does not change a coordinate, whereas the other one is changed linearly with the fixed coordinate. Algebraically we can write:

$$\boldsymbol{p}^* = \boldsymbol{H}_x\, \boldsymbol{p} = \begin{pmatrix} 1 & 0 \\ a & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ y + ax \end{pmatrix}$$

$$\boldsymbol{p}^* = \boldsymbol{H}_y\, \boldsymbol{p} = \begin{pmatrix} 1 & b \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x + by \\ y \end{pmatrix}$$

In particular, notice in Figure 6.15, that the line at unit distance from the axis translates of $a$ (respectively, of $b$). In other words the parameters $a$ and $b$ respectively represent the translations of lines $x = 1$ and $y = 1$.

**`PLaSM` representation of shearing** A predefined shearing tensor does not currently exist in `PLaSM`. Anyway, this operator is very easy to give as a user-defined function, by using the standard language mechanism for tensor definition, i.e. the operator `MAT`, which accepts as input a normalized homogeneous invertible matrix and returns the corresponding tensor:

$$\texttt{MAT} : \mathbb{R}^3_3 \rightarrow \text{lin}^3$$

**Example 6.2.5 (Running car)**
In order to define a parametrized tensor $\boldsymbol{H}_y(b)$ we can proceed as in Script 6.2.8, where the function `MAT` $\sim$ `MatHom` is applied to the non-homogeneous tensor matrix.

Then, two shearing tensor values — `Hy:1` and `Hy:-1`, respectively — are applied to the 2D `car` model previously defined, in order to get three key-frames of a very simple animation storyboard.

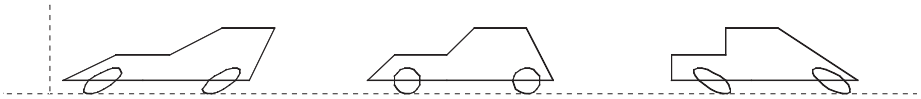

**Figure 6.16** Three key-frames of a simple storyboard

In particular, the model in Figure 6.16 is obtained by evaluating the symbol `story`, whereas the model in Figure 6.17 is obtained by evaluating the symbol `story3D`. Remember, from Section 3.3.6, that in `PLaSM` the homogeneous coordinate is the first one, so that we have:

```
MatHom:<<1, 5>, <0, 1>> ≡ <<1, 0, 0>, <0, 1, 5>, <0, 0, 1>>
```

**Script 6.2.8**

```
DEF Hy (b::IsReal) = (MAT ~ MatHom):<<1,b>,<0,1>>

DEF story = STRUCT:< Hy:1:car, T:1:12:car, (T:1:24 ~ Hy:-1):car >
DEF story3D = R:<2,3>:(PI/2):(Story * QUOTE:<3.5>);
```

The storyboard scenes can be easily transformed in 3D scenes by extruding the 2D images (see also Section 14.4) The extrusion operation is implemented in `PLaSM` as a Cartesian product with some 1D polyhedron. Also, we have to apply a rotation tensor of angle $\frac{\pi}{2}$ around the $x$ axis to the resulting object, in order to put the wheels upon the $xy$ plane.

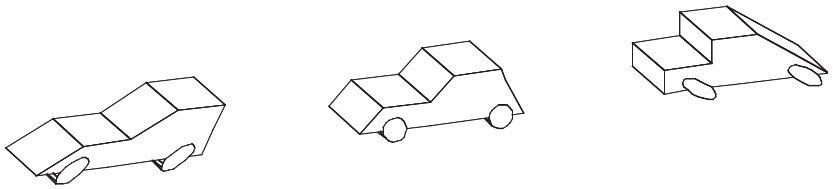

**Figure 6.17** Three key-frames of the 3D storyboard ("My wife's car").

### *6.2.6 Generic transformation*

We consider here the action of a generic tensor matrix $\boldsymbol{Q}$ on the unit square built on the basis vectors of a Cartesian frame $(\boldsymbol{o}, \boldsymbol{e}_i)$, with

$$\boldsymbol{Q} = \begin{pmatrix} a & c \\ b & d \end{pmatrix}.$$

Let $\boldsymbol{o}^*, \boldsymbol{a}^*, \boldsymbol{b}^*, \boldsymbol{c}^*$ be the images under $\boldsymbol{Q}$ of points $\boldsymbol{o}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$, respectively, with $\boldsymbol{a} = \boldsymbol{o} + \mathbf{e}_1$, $\boldsymbol{b} = \boldsymbol{o} + \mathbf{e}_2$ and $\boldsymbol{c} = \boldsymbol{o} + \mathbf{e}_1 + \mathbf{e}_2$. We can either write

$$\begin{pmatrix} \boldsymbol{o}^* & \boldsymbol{a}^* & \boldsymbol{b}^* & \boldsymbol{c}^* \end{pmatrix} = \boldsymbol{Q} \begin{pmatrix} \boldsymbol{o} & \boldsymbol{a} & \boldsymbol{b} & \boldsymbol{c} \end{pmatrix},$$

or, by using coordinates:

$$\begin{pmatrix} 0 & a & c & a+c \\ 0 & b & d & b+d \end{pmatrix} = \begin{pmatrix} a & c \\ b & d \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

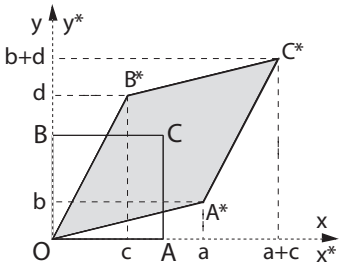


**Figure 6.18** Action of a generic tensor on the standard unit square

Looking at Figure 6.18, it is easy to note that:

1. a linear mapping does not move the origin;
2. the parallelisms of lines is conserved by the mapping, i.e. mapped parallel lines are parallel;
3. the size of areas is, in general, not conserved.

### *6.2.7 Tensor properties*

**Functional notation** We remember, from previous sections, that plane rotations depend on one real parameter, whereas plane translation and scaling depend on two real parameters, and so on. Hence we write, with homogeneous coordinates and by using a mathematical notation to denote plane transformation tensors:

$$\begin{aligned} \boldsymbol{R}_{xy} &: \mathbb{R} \to \mathrm{lin}^3 : \alpha \mapsto \boldsymbol{R}_{xy}(\alpha) \\ \boldsymbol{T}_{xy} &: \mathbb{R}^2 \to \mathrm{lin}^3 : (m, n) \mapsto \boldsymbol{T}_{xy}(m, n) \\ \boldsymbol{S}_{xy} &: \mathbb{R}^2 \to \mathrm{lin}^3 : (a, b) \mapsto \boldsymbol{S}_{xy}(a, b) \end{aligned}$$

The above notation gives a further explanation of the design choice for the syntax of `PLaSM` tensors, where rotation, translation and scaling are respectively denoted as `R:<1,2>:`$\alpha$, `T:<1,2>:<m,n>` and `R:<1,2>:<a,b>`. In higher-dimensional spaces, the set of indices may clearly vary.

**Composition or product** When a succession of tensors $\boldsymbol{Q}_1, \boldsymbol{Q}_2, \ldots, \boldsymbol{Q}_n$ is applied to a point $\boldsymbol{p}$, we can either write

$$\boldsymbol{p}^* = (\boldsymbol{Q}_n \circ \cdots \circ \boldsymbol{Q}_2 \circ \boldsymbol{Q}_1)(\boldsymbol{p}), \quad \text{or}$$

$$\boldsymbol{p}^* = \boldsymbol{Q}_n \ \cdots \ \boldsymbol{Q}_2 \ \boldsymbol{Q}_1 \ \boldsymbol{p},$$

depending on the meaning (either tensor or tensor matrix) of the symbol $\boldsymbol{Q}_i$.

**Associativity** In the following expressions parentheses are not needed, since both tensor composition and product of matrices are associative operations. In fact:

$$\boldsymbol{Q}_1 \circ \boldsymbol{Q}_2 \circ \boldsymbol{Q}_3 = (\boldsymbol{Q}_1 \circ \boldsymbol{Q}_2) \circ \boldsymbol{Q}_3 = \boldsymbol{Q}_1 \circ (\boldsymbol{Q}_2 \circ \boldsymbol{Q}_3)$$

$$\boldsymbol{Q}_1\boldsymbol{Q}_2\boldsymbol{Q}_3 = (\boldsymbol{Q}_1\boldsymbol{Q}_2)\ \boldsymbol{Q}_3 = \boldsymbol{Q}_1\ (\boldsymbol{Q}_2\boldsymbol{Q}_3)$$

**Commutativity** In general, tensor composition and matrix product are not commutative:

$$\boldsymbol{Q}_1 \circ \boldsymbol{Q}_2 \neq \boldsymbol{Q}_2 \circ \boldsymbol{Q}_1 \quad \text{and} \quad \boldsymbol{Q}_1\boldsymbol{Q}_2 \neq \boldsymbol{Q}_2\boldsymbol{Q}_1.$$

There are some important exceptions to this rule. The list is not exhaustive:

1. composition (product) of rotations about the same axis is commutative;
2. composition (product) of translations is commutative;
3. composition (product) of scaling is commutative;
4. composition (product) of rotations and uniform scaling is commutative.

A proof scheme of such statements is easier for their matrix versions. Write a matrix multiplication explicitly, and compute the resulting product matrix. The parameters of the compound mapping are expressed as either sum or product of parameters of component transformations.

**Product of scaling** The statement follows from commutativity of number product.

$$\boldsymbol{S}_2\ \boldsymbol{S}_1 = \begin{pmatrix} a_2 & 0 & 0 \\ 0 & b_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} a_1 & 0 & 0 \\ 0 & b_1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} a_1a_2 & 0 & 0 \\ 0 & b_1b_2 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \boldsymbol{S}_1\ \boldsymbol{S}_2$$

**Product of translations** The statement follows from commutativity of number summation.

$$\boldsymbol{T}_2\ \boldsymbol{T}_1 = \begin{pmatrix} 1 & 0 & m_2 \\ 0 & 1 & n_2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & m_1 \\ 0 & 1 & n_1 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & m_1 + m_2 \\ 0 & 1 & n_1 + n_2 \\ 0 & 0 & 1 \end{pmatrix} = \boldsymbol{T}_1\ \boldsymbol{T}_2$$

**Product of rotations** Trigonometric formulas of addition are used, together with commutativity of number summation.

$$\begin{aligned}
\boldsymbol{R}(\beta)\ \boldsymbol{R}(\alpha) &= \begin{pmatrix} \cos\beta & -\sin\beta & 0 \\ \sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} \cos\alpha\cos\beta - \sin\alpha\sin\beta & -(\cos\alpha\sin\beta + \sin\alpha\cos\beta) & 0 \\ \cos\alpha\sin\beta + \sin\alpha\cos\beta & \cos\alpha\cos\beta - \sin\alpha\sin\beta & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
&= \begin{pmatrix} \cos(\alpha+\beta) & -\sin(\alpha+\beta) & 0 \\ \sin(\alpha+\beta) & \cos(\alpha+\beta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
&= \boldsymbol{R}(\alpha)\ \boldsymbol{R}(\beta)
\end{aligned}$$

**Composition and inverse** From the statements proved above, one can conclude that:

1. Rotation and translation tensors have an *additive componibility*:

$$\boldsymbol{T}_{xy}(m_1, n_1) \circ \boldsymbol{T}_{xy}(m_2, n_2) = \boldsymbol{T}_{xy}(m_1 + m_2, n_1 + n_2),$$

$$\boldsymbol{R}_{xy}(\alpha_1) \circ \boldsymbol{R}_{xy}(\alpha_2) = \boldsymbol{R}_{xy}(\alpha_1 + \alpha_2)$$

2. Conversely, scaling tensors have a *multiplicative componibility*:

$$\boldsymbol{S}_{xy}(a_1, b_1) \circ \boldsymbol{S}_{xy}(a_2, b_2) = \boldsymbol{S}_{xy}(a_1 a_2, b_1 b_2)$$

3. Hence, it immediately follows for the inverse mappings that:

$$(\boldsymbol{T}_{xy}(m, n))^{-1} = \boldsymbol{T}_{xy}(-m, -n)$$

$$(\boldsymbol{R}_{xy}(\alpha))^{-1} = \boldsymbol{R}_{xy}(-\alpha)$$

$$(\boldsymbol{S}_{xy}(a, b))^{-1} = \boldsymbol{S}\left(\frac{1}{a}, \frac{1}{b}\right)$$

### *6.2.8 Fixed point transformations*

Each invertible linear transformation $\boldsymbol{Q}$ of the plane has the origin of the Cartesian frame as the only fixed point, i.e. $\boldsymbol{Q}(\boldsymbol{o}) = \boldsymbol{o}$. To have a fixed point different from the origin, we need to compose three mappings, that respectively:

1. move $\boldsymbol{q}$ to the origin;
2. apply the desired transformation;
3. move the origin back to $\boldsymbol{q}$.

**Scaling with fixed point** A scaling tensor with fixed point $\boldsymbol{q} \neq \boldsymbol{o}$, with $\boldsymbol{q} = \begin{pmatrix} m & n \end{pmatrix}^T$, is given by:

$$\boldsymbol{S}_{\boldsymbol{q}}(m, n, a, b) = \boldsymbol{T}_{xy}(m, n) \circ \boldsymbol{S}_{xy}(a, b) \circ \boldsymbol{T}_{xy}(-m, -n)$$

The succession of elementary transformations whose composition has the desired action, is graphically shown in Figure 6.19.

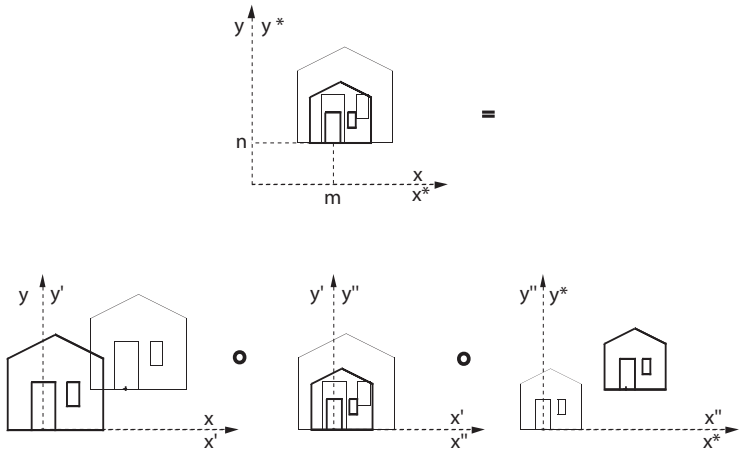


**Figure 6.19** Decomposition of a scaling with fixed point into a product of elementary transformations

**Rotation with fixed point** Analogously, for the rotation about a fixed point $\boldsymbol{q} \neq \boldsymbol{o}$, with $\boldsymbol{q} = \begin{pmatrix} m & n \end{pmatrix}^T$, we have:

$$\boldsymbol{R}_{\boldsymbol{q}}(m, n, \alpha) = \boldsymbol{T}_{xy}(m, n) \circ \boldsymbol{R}_{xy}(\alpha) \circ \boldsymbol{T}_{xy}(-m, -n)$$

The succession of elementary transformations, which give a rotation of angle $\alpha$ about a fixed point $\boldsymbol{q}$, is graphically shown in Figure 6.20.

## 6.3 3D Transformations

### *6.3.1 Elementary transformations*

The extension of transformations already discussed for the 2D case is very easy. Some more care is just needed for 3D rotation and shearing, so that we reserve most of this section to them. In the remainder, with the aim of unifying the management of both linear and affine transformations and of using the matrix product as the only geometric operator, we use normalized homogeneous coordinates and tensors in lin $\mathbb{R}^4$.

**Translation and scaling** The *translation* tensor $\boldsymbol{T}_{xyz}(l, m, n)$ with parameters $l, m, n$ and the *scaling* tensor $\boldsymbol{S}_{xyz}(a, b, c)$ with parameters $a, b, c$ are represented, respectively, by matrices

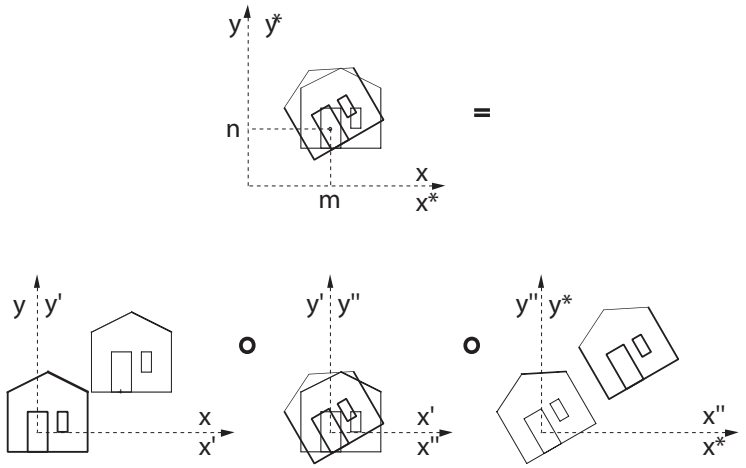


**Figure 6.20** Decomposition of a rotation with fixed point into a product of elementary transformations

$$\boldsymbol{T}_{xyz}(l,m,n) = \begin{pmatrix} 1 & 0 & 0 & l \\ 0 & 1 & 0 & m \\ 0 & 0 & 1 & n \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \text{and} \quad \boldsymbol{S}_{xyz}(a,b,c) = \begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

**Shearing** An elementary shearing of the 3D space is a tensor which does not change a coordinate and changes the other ones as linear functions of the non-transformed coordinate. We hence distinguish three elementary shearing tensors $\boldsymbol{H}_{yz}(a,b)$, $\boldsymbol{H}_{xz}(a,b)$ and $\boldsymbol{H}_{xy}(a,b)$, whose matrices differ from the identity matrix just along the elements of one column. Such tensor matrices are easier to remember if denoted with the index of the invariant coordinate:

$$\boldsymbol{H}_x(a,b) \equiv \boldsymbol{H}_{yz}(a,b) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ a & 1 & 0 & 0 \\ b & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\boldsymbol{H}_y(a,b) \equiv \boldsymbol{H}_{xz}(a,b) = \begin{pmatrix} 1 & a & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & b & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\boldsymbol{H}_z(a,b) \equiv \boldsymbol{H}_{xy}(a,b) = \begin{pmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In fact we have, respectively:

$$\begin{aligned}
\boldsymbol{p}^* &= \boldsymbol{H}_x(a,b)\,\boldsymbol{p} = \begin{pmatrix} x & y+ax & z+bx & 1 \end{pmatrix}^T \\
\boldsymbol{p}^* &= \boldsymbol{H}_y(a,b)\,\boldsymbol{p} = \begin{pmatrix} x+ay & y & z+by & 1 \end{pmatrix}^T \\
\boldsymbol{p}^* &= \boldsymbol{H}_z(a,b)\,\boldsymbol{p} = \begin{pmatrix} x+az & y+bz & z & 1 \end{pmatrix}^T
\end{aligned}$$

To visualize the action of such tensors, the 3D space should be considered as a bundle of planes parallel to a coordinate plane. The coordinate plane is invariant in this mapping; the other planes are moved by a translation on themselves. The translation of each plane is a linear function of its distance from the coordinate plane.

Consider, e.g., the tensor $\boldsymbol{H}_z = \boldsymbol{H}_{xy}(a,b)$. In this case:

1. the plane $z = 0$ is invariant;
2. the plane $z = 1$ translates by a translation vector $\boldsymbol{t} = \begin{pmatrix} a & b & 0 \end{pmatrix}^T$.
3. each plane $z = c$ translates by $\boldsymbol{t} = \begin{pmatrix} ac & bc & 0 \end{pmatrix}^T$;

**Elementary rotations** Given a Cartesian frame in $\mathbb{E}^3$, we call *elementary rotations* $\boldsymbol{R}_x$, $\boldsymbol{R}_y$ and $\boldsymbol{R}_z$, also denoted as $\boldsymbol{R}_{yz}$, $\boldsymbol{R}_{xz}$ and $\boldsymbol{R}_{xy}$, respectively, three functions from reals to tensors in $\mathrm{lin}^4$, which return, for any given angle, the rotation tensor about the corresponding coordinate axis. We use equivalently either the notation $\boldsymbol{R}_z$ or $\boldsymbol{R}_{xy}$ to denote either the invariant coordinate or the varying coordinates. E.g.:

$$\boldsymbol{R}_x : \mathbb{R} \to \mathrm{lin}^4 : \alpha \mapsto \boldsymbol{R}_{yz}(\alpha)$$

The matrices of elementary rotation tensors defined above are obtained by suitably embedding the rotation matrix of 2D plane into the $4 \times 4$ identity matrix:

$$\boldsymbol{R}_x(\alpha) \equiv \boldsymbol{R}_{yz}(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

$$\boldsymbol{R}_y(\beta) \equiv \boldsymbol{R}_{xz}(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

$$\boldsymbol{R}_z(\gamma) \equiv \boldsymbol{R}_{xy}(\gamma) = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

**Example 6.3.1 (Stack of rotated elements)**
An assembly of rotated parallelepipeds is produced in this example. First a parallelepiped element is defined in Script 6.3.1, and is translated in $x, y$ by a tensor

`T:<1,2>:<-5,-5>`, in order to align the object center with the $z$ axis. Then the `pair` object is defined as an assembly of the untransformed element with a second element instance rotated about the $z$ axis (by tensor `R:<1,2>:(PI/8)`) and translated (by tensor `T:3:2`). The geometric value generated by evaluation of the `pair` symbol is shown in Figure 6.21a.

**Script 6.3.1**

```
DEF element = (T:<1,2>:<-5,-5> ~ CUBOID):<10,10,2>;
DEF pair    = STRUCT:< element, (T:3:2 ~ R:<1,2>:(PI/8)):element >;
DEF column  = (STRUCT ~ ##:17): < element, T:3:2, R:<1,2>:(PI/8) >;
```

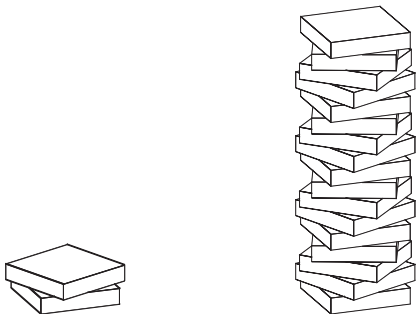

**Figure 6.21** Rotated 3D elements: (a) `pair` (b) `column`

A fully equivalent but more elegant definition of the `pair` object can be given in proper `FL` style as follows:

```
DEF pair = (STRUCT ~ [ID, T:3:2 ~ R:<1,2>:(PI/8)]): element
```

To produce the `column` object shown in Figure 6.21b, both the combinatorial power of `FL` and the semantics of *hierarchical structures* (also called *hierarchical graphs* in graphics) are exploited. A wider discussion of hierarchical structures is given in Chapter 8.

### *6.3.2 Rotations*

A 3D space rotation is a linear orthogonal transformation with a set of fixed points (called *autospace* in linear algebra) of dimension 1, known as the *rotation axis*. In such a transformation, every space point (outside the rotation axis) is mapped to the second extreme of a circle segment of constant angle with its center on the rotation axis, and belonging to the orthogonal plane passing for the point.

To compute the matrix of a rotation tensor $\boldsymbol{R}_{xyz}(\boldsymbol{n}, \alpha)$, with

$$\boldsymbol{R}_{xyz} : \mathbb{R}^3 \times \mathbb{R} \rightarrow \text{lin}^4 : (\boldsymbol{n}, \alpha) \mapsto \boldsymbol{R}_{xyz}(\boldsymbol{n}, \alpha),$$

where the rotation axis is parallel to the vector $\boldsymbol{n}$ and $\alpha$ is the rotation angle, we can proceed in several ways, two of which are discussed below.

**Composition of elementary rotations** A non-elementary 3D space rotation $\boldsymbol{R}_{xyz}(\boldsymbol{n}, \alpha)$ can be reduced to the composition of a suitable succession of elementary rotations.

First, a compound rotation $\boldsymbol{R}_y(\gamma) \circ \boldsymbol{R}_x(\beta)$ about the $x$-axis and $y$-axis is applied, with the aim of cancelling two components of the mapped $\boldsymbol{n}$-axis, which is thus transformed onto the $z$-axis. A $z$-rotation tensor $\boldsymbol{R}_z(\alpha)$ of angle $\alpha$ is applied at this point. Finally, the inverse rotation $(\boldsymbol{R}_y(\gamma) \circ \boldsymbol{R}_x(\beta))^{-1}$ is applied, with the aim of mapping the $\boldsymbol{n}$-axis back to its original position. Hence we write:

$$\begin{aligned}
\boldsymbol{R}_{xyz}(\boldsymbol{n}, \alpha) &= (\boldsymbol{R}_y(\gamma) \circ \boldsymbol{R}_x(\beta))^{-1} \circ \boldsymbol{R}_z(\alpha) \circ (\boldsymbol{R}_y(\gamma) \circ \boldsymbol{R}_x(\beta)) \\
&= \boldsymbol{R}_x(\beta)^{-1} \circ \boldsymbol{R}_y(\gamma)^{-1} \circ \boldsymbol{R}_z(\alpha) \circ \boldsymbol{R}_y(\gamma) \circ \boldsymbol{R}_x(\beta) \\
&= \boldsymbol{R}_x(-\beta) \circ \boldsymbol{R}_y(-\gamma) \circ \boldsymbol{R}_z(\alpha) \circ \boldsymbol{R}_y(\gamma) \circ \boldsymbol{R}_x(\beta)
\end{aligned}$$

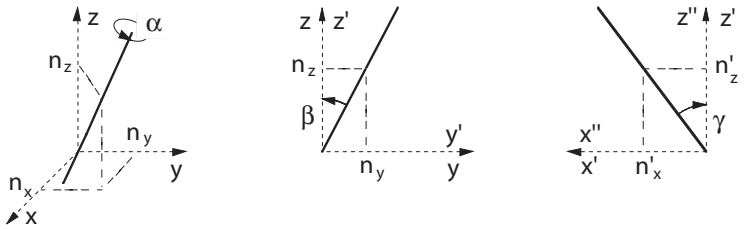


**Figure 6.22** Decomposition of a general rotation into elementary rotations: (a) the $\boldsymbol{n}$-axis (b) the $x$-rotation (c) the $y$-rotation

We must finally compute the angles $\beta$ and $\gamma$, to draw out the elementary rotation tensors $\boldsymbol{R}_x(\beta)$ and $\boldsymbol{R}_y(\gamma)$. Looking at Figure 6.22 we can write:

$$\beta = \arctan\left(\frac{n_y}{n_z}\right) \qquad \gamma = -\arctan\left(\frac{n'_x}{n'_z}\right)$$

where $\boldsymbol{n}' = \boldsymbol{R}_x(\beta)\ \boldsymbol{n}$.

**Transformation of coordinates** A rotation tensor $\boldsymbol{R}_{xyz}(\boldsymbol{n}, \alpha)$ can be derived very easily by composition of:

1. a coordinate transformation $\boldsymbol{Q}(\boldsymbol{n})$ which maps the unit vector $\frac{\boldsymbol{n}}{|\boldsymbol{n}|}$ and two orthogonal unit vectors into the elements of a new basis;
2. a rotation $\boldsymbol{R}_z(\alpha)$ about the $z$-axis of this new basis;
3. the inverse coordinate transformation $\boldsymbol{Q}^{-1}(\boldsymbol{n})$.

Hence we write:

$$\boldsymbol{R}_{xyz}(\boldsymbol{n}, \alpha) = \boldsymbol{Q}^{-1}(\boldsymbol{n}) \circ \boldsymbol{R}_z(\alpha) \circ \boldsymbol{Q}(\boldsymbol{n}). \tag{6.1}$$

To compute $\boldsymbol{Q}(\boldsymbol{n})$ we choose an orthonormal vector triplet with an element directed as the rotation axis. Let $\boldsymbol{q}_x, \boldsymbol{q}_y, \boldsymbol{q}_z$ be such a triplet, given in coordinates relative to the old basis $\{\boldsymbol{e}_i\}$. We see they are transformed in a new basis $\{\hat{\boldsymbol{e}}_i\}$ by the unknown matrix $\boldsymbol{Q}(\boldsymbol{n})$:

$$\begin{pmatrix} \hat{\boldsymbol{e}}_1 & \hat{\boldsymbol{e}}_2 & \hat{\boldsymbol{e}}_3 \end{pmatrix} = \boldsymbol{Q}(\boldsymbol{n}) \begin{pmatrix} \boldsymbol{q}_x & \boldsymbol{q}_y & \boldsymbol{q}_z \end{pmatrix}.$$

But the left-hand side is the identity matrix, and hence:

$$\boldsymbol{Q}(\boldsymbol{n}) = \begin{pmatrix} \boldsymbol{q}_x & \boldsymbol{q}_y & \boldsymbol{q}_z \end{pmatrix}^{-1} = \begin{pmatrix} \boldsymbol{q}_x & \boldsymbol{q}_y & \boldsymbol{q}_z \end{pmatrix}^{T} = \begin{pmatrix} \boldsymbol{q}_x^T \\ \boldsymbol{q}_y^T \\ \boldsymbol{q}_z^T \end{pmatrix}$$

where we set $\boldsymbol{Q}^{-1}(\boldsymbol{n}) = \boldsymbol{Q}^T(\boldsymbol{n})$ since $\boldsymbol{Q}(\boldsymbol{n})$, which maps an orthogonal unit triplet into an orthogonal unit triplet, is an orthogonal transformation.

Finally, we have to define the unit vectors $\boldsymbol{q}_x$, $\boldsymbol{q}_y$ and $\boldsymbol{q}_z$. First we set

$$\boldsymbol{q}_z = \frac{\boldsymbol{n}}{\|\boldsymbol{n}\|},$$

and also, provided that $\boldsymbol{n} \neq \boldsymbol{e}_3$, which would imply the trivial case $\boldsymbol{R}(\boldsymbol{n}, \alpha) = \boldsymbol{R}_z(\alpha)$, we can write

$$\boldsymbol{q}_x = \frac{\boldsymbol{e}_3 \times \boldsymbol{n}}{\|\boldsymbol{e}_3 \times \boldsymbol{n}\|}, \quad \text{and} \quad \boldsymbol{q}_y = \boldsymbol{q}_z \times \boldsymbol{q}_x.$$

**Example 6.3.2**

In this example we discuss the `PLaSM` implementation of rotation tensor $\boldsymbol{R}_{xyz}(\boldsymbol{n}, \alpha)$. For this purpose we first define a function `Rot_xyz` with a real `alpha` and a vector `n` as formal parameters. The function body, i.e. the expression used to compute its values, is a direct `PLaSM` translation of Formula (6.1).

The `MatHom` function, of signature $\mathcal{M}_n^n \to \mathbb{R}_{n+1}^{n+1}$, is given in Script 3.3.13; the functions `UnitVect` and `VectProd` are given in Scripts 3.2.4 and 3.1.3, respectively. Such functions homogenize a squared matrix by adding a unit row and column, normalize a vector and compute a vector product, respectively.

The higher-level function to compute the rotation tensor is called `Rot_xyz`. It is actually a filter which invokes either `Rot_n` or the predefined operator `R:<1,2>`, depending on the orientation of the `n` vector parameter. Two predicates `IsZero` and `IsUp` are defined for this purpose.

Notice that the tensor `MAT:Q` is defined by homogenizing the `<qx, qy, qz>` matrix, where the orthonormal vectors are directly accumulated by *row*, according to the discussed method. Remember in fact that a `PLaSM` matrix representation is a sequence of matrix rows.

The models of Figure 6.23 are respectively generated as:

```
STRUCT:<
   CUBOID:< 1, 1, 0.2 >,
   MKPOL:< <<0,0,0>, <1,1,0>>, <<1,2>>, <<1>> >
>;
```

**Script 6.3.2 (3D rotation about the $n$-axis)**

```
DEF Rot_n (alpha::IsReal; n::IsVect) =
   (MAT∼TRANS):Q ∼ R:<1,2>:alpha ∼ MAT:Q
WHERE
   Q  = MatHom:<qx, qy, qz>,
   qx = UnitVect:(<0,0,1> VectProd n),
   qy = qz VectProd qx,
   qz = UnitVect:n
END;

DEF IsZero = AND ∼ AA:(C:EQ:0);
DEF IsUp = AND ∼ [C:EQ:0∼s1, C:EQ:0∼s2, NOT∼C:EQ:0∼s3];

DEF Rot_xyz = IF:< OR∼[IsUp,IsZero]∼s2, R:<1,2>∼s1, Rot_n >;
```

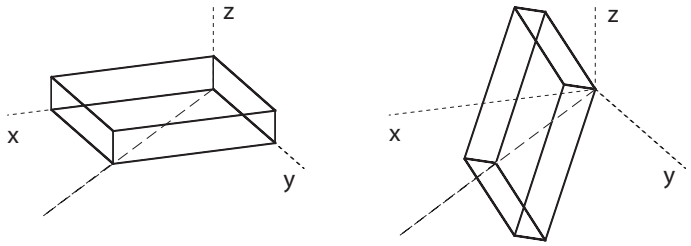


**Figure 6.23** Action of a rotation tensor `Rot_xyz:<PI/2,<1,1,0>>` of angle $\pi/2$ about axis $\boldsymbol{n} = [1\ 1\ 0]^T$

```
STRUCT:<
   (Rot_xyz:<PI/2,<1,1,0>> ∼ CUBOID):< 1, 1, 0.2 >,
   MKPOL:< <<0,0,0>, <1,1,0>>, <<1,2>>, <<1>> >
>;
```

### *6.3.3 Rotations about affine axes*

A more general rotation tensor of $E^3$, with axis an *affine* subspace of dimension 1, i.e. a straight line in general not passing for the origin, is obtained by composition of transformations in lin $\mathbb{R}^4$:

$$\boldsymbol{R}^*_{xyz}(\boldsymbol{n}, \boldsymbol{p}, \alpha) = \boldsymbol{T}_{xyz}(\boldsymbol{p} - \boldsymbol{o}) \circ \boldsymbol{R}_{xyz}(\boldsymbol{n}, \alpha) \circ \boldsymbol{T}_{xyz}(\boldsymbol{o} - \boldsymbol{p})$$

where $\boldsymbol{R}^*_{xyz}(\boldsymbol{n}, \boldsymbol{p}, \alpha)$ denotes the *rotation about the $\boldsymbol{n}$-axis passing for point $\boldsymbol{p}$*, and $\boldsymbol{o}$ is the origin of a Cartesian frame for $E^3$.

We note that the parameters of function $\boldsymbol{T}_{xyz}$ are vectors, since, from Section 3.2, the difference of points is a vector.

### *6.3.4 Algebraic properties of rotations*

A rotation tensor $\boldsymbol{R} \in \text{lin}\,\mathcal{V}$ does not change the angles between vectors. This property can be written algebraically using the inner product.

For each $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{V}$:

$$\boldsymbol{u} \cdot \boldsymbol{v} = \boldsymbol{R}(\boldsymbol{u}) \cdot \boldsymbol{R}(\boldsymbol{v})$$

We can write for the corresponding matrices:

$$[\boldsymbol{u}]^T[\boldsymbol{v}] = [\boldsymbol{R}\boldsymbol{u}]^T[\boldsymbol{R}\boldsymbol{v}] = [\boldsymbol{u}]^T[\boldsymbol{R}]^T[\boldsymbol{R}][\boldsymbol{v}]$$

so, we get

$$[\boldsymbol{R}]^T[\boldsymbol{R}] = [\boldsymbol{R}^T][\boldsymbol{R}] = [I],$$

and hence

$$\boldsymbol{R}^T\boldsymbol{R} = I = \boldsymbol{R}^{-1}\boldsymbol{R}.$$

We can conclude that

$$\boldsymbol{R}^T = \boldsymbol{R}^{-1}$$

i.e. that a rotation is an *orthogonal* tensor.

Some further properties of rotations are listed here:

1. In order to verify if a matrix is a rotation, it is sufficient to show that:
$$[\boldsymbol{Q}][\boldsymbol{Q}]^T = [\boldsymbol{Q}]^T[\boldsymbol{Q}] = [\boldsymbol{I}]$$
2. Since $[\boldsymbol{R}]$ is an orthogonal matrix, *det* $\boldsymbol{R}$ is either 1 or $-1$; a rotation with determinant 1 is called *proper*; with determinant $-1$ is called *improper*.
3. An improper rotation is the product of a proper rotation times a reflection.
4. The angle $\alpha$ of a rotation $\boldsymbol{Q}$ can be computed from the relation
$$tr\ \boldsymbol{Q} = 1 + 2\cos\alpha$$
5. The axis of a 3D rotation is given by the eigenvector associated with the real eigenvalue of its matrix.

**Reflections and improper rotations** A *reflection* or *mirroring* is a tensor which reverses each vector of an axis for the origin and maintains fixed each vector of the orthogonal linear subspace.

For a reflection tensor $\boldsymbol{M}$ we have:

$$\boldsymbol{M}^2 = \boldsymbol{I}$$

$$\boldsymbol{M} = \boldsymbol{M}^T = \boldsymbol{M}^{-1}$$

The reflections which fix the coordinate planes are called *elementary* reflections or mirroring. They can be implemented as scaling with one coefficient equal to $-1$:

$$\begin{aligned} \boldsymbol{M}_x &= \boldsymbol{S}_{xyz}(-1,1,1) \\ \boldsymbol{M}_y &= \boldsymbol{S}_{xyz}(1,-1,1) \\ \boldsymbol{M}_z &= \boldsymbol{S}_{xyz}(1,1,-1) \end{aligned}$$

Tensors $\boldsymbol{Q}$ with matrix determinant $-1$ are called either *improper rotations* or *rotational reflections* . They are given by the composition of a rotation about some axis and by the reflection with respect to the plane orthogonal to such axis.

Angle and axis of an improper rotation $\boldsymbol{Q}$ are obtained from trace and eigenvector of its matrix, as for proper rotations. Exchanging $\boldsymbol{Q}$ with $-\boldsymbol{Q}$ exchanges the proper with the improper, and changes the sign of the rotation angle, but does not change the axis.

**Global scaling** A uniform scaling tensor $\boldsymbol{S}_{xyz}(a,a,a)$ can also be represented by a matrix $(s_{ij}) \in \mathbb{R}^4_4$, which differs from the identity just for the coefficient $s_{44}$:

$$\boldsymbol{S}_{xyz}(a,a,a) \equiv \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{a} \end{pmatrix}$$

In fact, it is easy to verify that:

$$p^* = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{a} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ \frac{1}{a} \end{pmatrix} = \begin{pmatrix} ax \\ ay \\ az \\ 1 \end{pmatrix}$$

**Structure of matrices** Resuming what we said about the affine transformations of $E^3$, which are represented in homogeneous coordinates by $4 \times 4$ real matrices, we note they always have the structure:

$$\boldsymbol{Z} = \begin{pmatrix} \boldsymbol{Q} & \boldsymbol{m} \\ \boldsymbol{0}^T & a \end{pmatrix}$$

where $\boldsymbol{Q}$ is an invertible $3 \times 3$ matrix. If $\boldsymbol{m} \neq \boldsymbol{0}$, then $\boldsymbol{Z}$ contains a translational component. If $a \neq 1$, then we say that $\boldsymbol{Z}$ is *not normalized*. In this case it contains a global scaling component with parameter $\frac{1}{a}$.

**Tensor action on covectors** In both graphics and modeling applications it is sometimes necessary to apply a space transformation to face equations instead of to vertex coordinates. This often happens within the geometric kernel of `PLaSM`, where the linear equations of faces of polyhedral models are explicitly stored in memory.

Let, e.g.,

$$ax + by + cz + d = 0$$

be the Cartesian equation of a plane, and remember that this one is a point set mapped to zero by a linear function $\phi : \mathbb{R}^4 \to \mathbb{R}$, called *covector*, and represented in coordinates as the *row* vector

$$\boldsymbol{q} = \begin{pmatrix} a & b & c & d \end{pmatrix}$$

of coefficients of the plane. We often need to compute the covector $\boldsymbol{q}^*$ associated with the plane transformed by the action of a tensor $\boldsymbol{M}$ on $E^3$.

If $\boldsymbol{p} = \begin{pmatrix} x & y & z & 1 \end{pmatrix}^T$ is the homogeneous representation of $E^3$ points, then the points of the plane will satisfy the relation:

$$\boldsymbol{q}\ \boldsymbol{p} = 0. \tag{6.2}$$

Clearly, if we apply a tensor $\boldsymbol{M}$ to $\mathbb{E}^3$, we also have:

$$\boldsymbol{q}^*\ \boldsymbol{p}^* = \boldsymbol{q}\ \boldsymbol{T}\ \boldsymbol{M}\ \boldsymbol{p} = 0, \tag{6.3}$$

where $\boldsymbol{T}$ is the unknown tensor for the dual space, to be applied to $\boldsymbol{q}$ covector. Since both identities (6.2) and (6.3) must hold, it is:

$$\boldsymbol{T}\ \boldsymbol{M} = \boldsymbol{I} \qquad \text{and hence} \qquad \boldsymbol{T} = \boldsymbol{M}^{-1}$$

Therefore, in order to apply a tensor $\boldsymbol{M}$ to a point space, we need to apply a left-hand matrix multiplication times $\boldsymbol{M}$ to the underlying vectors, as well as a right-hand matrix multiplication times $\boldsymbol{M}^{-1}$ to the underlying covectors.

## 6.4 PLaSM implementation

### *6.4.1 Pre-defined affine tensors*

Some predefined higher-level `PLaSM` functions return either translation, scaling or rotation tensors, when applied to suitable parameters. Such functions are respectively denoted as `T`, `S` and `R`.

As is well known at this point, one major `PLaSM` characteristics is *dimensional independence*. This property allows definition of functions and evaluating expressions which are able to generate geometric objects of any dimension. A `PLaSM` object, seen as a point set, may in fact belong to spaces described by bases with any number of elements. In other words, object points may have any number of coordinates.

Affine transformation functions are accordingly designed to generate transformation tensors able to work on any user-specified subset of object coordinates.

**Translation** Translation tensors are generated by the `PLaSM` operator denoted by symbol `T`. Hence, this symbol cannot be re-defined by the user, without generating an error at interpretation time. The signature of `T` operator is:

$$\texttt{T} : Z^d \to \mathbb{R}^d \to \text{lin}\ \mathbb{R}^*, \qquad 1 \leq d$$

The dimension of the resulting tensor is not specified, because it is only determined at run-time — and not at interpretation time — depending on the dimension of the polyhedral expression it is actually applied to. For example, we can equally write:

```
(T:2:10.5 ~ T:4:3):(CUBOID:<1,1,1,1,1>)  or
(T:<2,4>:<10.5,3>):(CUBOID:<1,1,1,1,1>)  or
T:<2,4>:<10.5,3>:(CUBOID:<1,1,1,1,1>)  or
(T:<2,4>:<10.5,3> ~ CUBOID):<1,1,1,1,1>
```

where the evaluation of each such expression gives an instance of the 5-dimensional unit hypercube, translated on the second and fourth coordinates.

**Example 6.4.1**
Two very simple polyhedra are defined in Script 6.4.1, and designated as `basis` and `pyramid`. Then the assembly `pair` is given by using the `STRUCT` function, whose semantics is described in Chapter 8.

**Script 6.4.1**

```
DEF basis = CUBOID:<10,10,1>;
DEF pyramid = MKPOL:<
   <<0,0,0>,<8,0,0>,<8,8,0>,<0,8,0>,<4,4,4>>,<1..5>,<<1>> >;
DEF pair = STRUCT:<basis, T:<1,2,3>:<1,1,1>:pyramid>
```

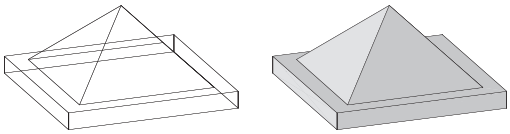

**Figure 6.24** `pair` assembly with pyramid translation

In Figure 6.24 the object generated by evaluation of `pair` symbol is shown. A similar example is given in Script 6.4.2 where a `transl` tensor is defined and instanced more times. The `triplet` value is shown in Figure 6.25.

**Script 6.4.2**

```
DEF transl = T:<1,2,3>:<1,1,1>;
DEF pair   = STRUCT:<basis, transl:pyramid>;
DEF triplet = STRUCT: <basis, transl:basis, (transl ~ transl):pyramid>;
```

**Scaling** Scaling tensors are generated by the higher-level `PLaSM` operator named `S`. The signature of this operator is:

$$\mathtt{S} : Z^d \to \mathbb{R}^d \to \text{lin } \mathbb{R}^*, \qquad 1 \leq d$$

Also in this case the dimension of the resulting tensor is not specified, and is determined at run-time, as for translation tensors.

**Rotation** The higher-level operator `R` generates elementary rotation tensors, i.e. space isometries which change two coordinates according to the pattern of plane rotations, and leave the other coordinates invariant. The `R` signature is:

$$\mathtt{R} : Z^2 \to \mathbb{R} \to \text{lin } \mathbb{R}^*.$$

Once more, the dimension of the generated tensors is not specified, and is determined at run-time, as for translation and scaling. E.g., the tensor `R:<1,2>:PI` can be either applied to a 2D or to a 4D polyhedron.

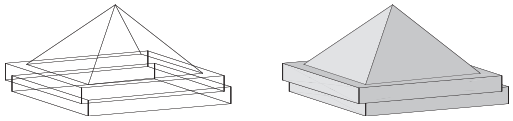

**Figure 6.25** More complex `triplet` assembly

**Example 6.4.2 (2D and 3D Clock)**
We define here two PLaSM functions, named `clock2D` and `clock3D`, that generate simplified clock models with time set as specified by their integers parameters, corresponding to hours and minutes, respectively. The implemented code is given in Scripts 6.4.3 and 6.4.4. Some generated models are shown in Figures 6.26 and 6.27.

A 2-dimensional clock model is first given, by defining a circular `background`, the 12 hour `ticks`, and the `hour` and `minute` hands, each one given in a local coordinate frame. The `Circle` function is given in Script 1.6.1, and returns a polygonal approximation of circle of given radius. In this case it returns a 2D regular polygon with 24 sides.

**Script 6.4.3 (2D Clock)**

```
DEF background = Circle:0.8:<24,1>;
DEF minute = (T:<1,2>:<-0.05,-0.05> ~ CUBOID):<0.9,0.1>;
DEF hour = (T:<1,2>:<-0.1,-0.1> ~ CUBOID):<0.7,0.2>;
DEF ticks = (STRUCT ~ ##:12):< tick, R:<1,2>:(PI/6) >;
DEF tick = (T:<1,2>:<-0.025,0.55> ~ CUBOID):<0.05,0.2>;

DEF clock2D (h,m::IsInt) = STRUCT:<
   background,
   ticks,
   R:<1,2>:( PI/2 - (h + m/60)*PI/6 ):hour,
   R:<1,2>:( PI/2 - m*PI/30 ):minute
>;
```

The interesting part of Script 6.4.3 is constituted by the `clock2D` function, with integer parameters `h` and `m`, for hours and minutes to set, respectively. An $xy$ rotation of angle `PI/2 - (h+m/60)*PI/6` is there applied to the `hour` hand. The positive constant $\frac{\pi}{2}$ is summed to move the origin of angles on the vertical line. The negative term takes into account both the angles induced by integer parameters `h` and `m`. Such a term is negative, because the clock hand movements are (quite obviously!) *clockwise*, whereas positive angles are counter-clockwise. The definition of the `Q` function is discussed in Section 1.6.2. It is used in Script 6.4.4 as a short-hand for `QUOTE ~ LIST`.

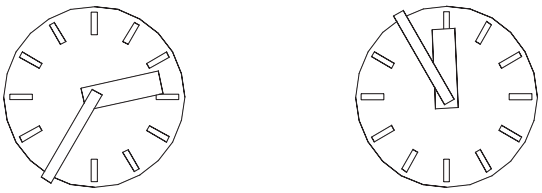

**Figure 6.26** Geometric values generated by evaluation of `clock2D:<2,35>` and `clock2D:<11,55>`

The `clock3D` function differs from `clock2D` by addition widths to objects in the $z$ direction, and by coloring red the solid `background`. The adding of VRML-like colors and textures to geometric models is discussed in Chapter 10. The generated 3D clock model is displayed in Figure 6.27.

Notice that both the Cartesian product of polyhedra * and the `COLOR` binary function are used infix in the 3D `background` transforming expression. They are evaluated correctly by the interpreter, because infix operators are left-associative in `PLaSM`, i.e.:

$$\texttt{A}\ op_1\ \texttt{B}\ op_2\ \texttt{C} \equiv \texttt{(A}\ op_1\ \texttt{B)}\ op_2\ \texttt{C}$$

Notice also that each tensor inserted as isolated element in the `STRUCT` sequence, e.g. as `T:3:0.2`, is applied to each geometric value which follows it, according to the semantics of hierarchical structures (Chapter 8). The `Q` operator, which is a shortcut for `QUOTE` working both on number sequences and on single numbers, is given in Script 1.5.5.

**Script 6.4.4 (3D Clock)**

```
DEF clock3D (h,m::IsInt) = STRUCT:<
   background * Q:0.2 COLOR RGB:<1,0,0>,
   T:3:0.2:(ticks * Q:0.01), T:3:0.2,
   R:<1,2>:( PI/2 - (h + m/60)*PI/6 ):(hour * Q:0.03), T:3:0.03,
   R:<1,2>:( PI/2 - m*PI/30 ):(minute * Q:0.03)
>;
```

**Figure 6.27** 3D VRML model generated by evaluation of `clock3D:<2,35>`

### *6.4.2 User-defined affine tensors*

It may sometimes be useful to apply tensors not to polyhedra but to single points, i.e. directly to a set of coordinates (see, e.g. Script 3.3.15). This may be done by implementing the desired tensors as user-defined `PLaSM` functions. In the following the elementary rotations, translations and scaling in $E^3$, which depend on only one real parameter, are given.

It is interesting to note that, by using the standard composition of the language, such functions may be first partially specified, then freely composed, and only later applied

**Script 6.4.5 (User-defined 3D tensors)**

```
DEF Tx (a::IsReal)(x,y,z::IsReal) = < x + a, y, z >;
DEF Ty (a::IsReal)(x,y,z::IsReal) = < x, y + a, z >;
DEF Tz (a::IsReal)(x,y,z::IsReal) = < x, y, z + a >;

DEF Sx (a::IsReal)(x,y,z::IsReal) = < x * a, y, z >;
DEF Sy (a::IsReal)(x,y,z::IsReal) = < x, y * a, z >;
DEF Sz (a::IsReal)(x,y,z::IsReal) = < x, y, z * a >;

DEF Rx (a::IsReal)(x,y,z::IsReal) =
   < x, cos:a * y - sin:a * z, sin:a * y + cos:a * z >;
DEF Ry (a::IsReal)(x,y,z::IsReal) =
   < cos:a * x + sin:a * z, y, (- sin):a * x + cos:a * z >;
DEF Rz (a::IsReal)(x,y,z::IsReal) =
   < cos:a * x - sin:a * y, sin:a * x + cos:a * y, z >;
```

to the target points. For example, a rotation of angle $\alpha$ of the point $\boldsymbol{p} = (p_x, p_y, p_z)$ around a line parallel to the $y$-axis and passing for the point $\boldsymbol{q} = (0, h, k)$, i.e.

$$\boldsymbol{p}^* = \boldsymbol{R}_{xyz}(\boldsymbol{e}_2, \boldsymbol{q}, \alpha)(\boldsymbol{p})$$

can be directly computed in PLaSM as:

```
DEF p_star =
   (Tz:k ~ Ty:h ~ Ry:alpha ~ Ty:(-:h) ~ Tz:(-:k)):< px,py,pz >
```

## 6.5 Examples

In this section we discuss some simple but quite realistic examples of geometric programming with affine transformations, whose implementation requires combining tensors in several ways with other PLaSM operators. The more interesting example, is probably the first one, where some functions MKframe and MKvector are given, to generate the geometric model of a Cartesian reference frame and of any applied vector.

### *6.5.1 Modeling applied 3D vectors and reference frames*

Two useful functions MKframe and MKvector are given to generate the geometric model of the Cartesian frame, as well as of any *applied vector* in 3D. Several functions, including type predicates, vector and matrix functions, the rotation transformation around a given axis for the origin, and more, are given in previous Scripts.

In Script 6.5.1 the symbol MKversork generates a geometric model of the unit vector $e_3$ of the $z$ axis, as made by a cylinder of radius 1/100 and height 7/8 and by a cone of radius 1/16 and height 1/8.

**Script 6.5.1 (geometric model of the unit vector <0,0,1>)**

```
DEF MKversork =
   CYLINDER:<1/100,7/8>:3 TOP (Cone:<1/16,1/8>:8);
```

The MKvector function in Script 6.5.2 generates the geometric model of an applied vector with first extreme in point p1 and second extreme in point p2. The arrow model of the actual vector is produced by first scaling the object generated by MKversork to the proper size, then by rotating to make it parallel to the final model and then by translating the model to the final position. Finally, the optimize operator is applied, in order to make possible the VRML exporting of a polyhedral complex with a general transformation matrix inserted in the data structure. As already discussed elsewhere, the composite operator MKPOL ∼ UKPOL has the effect of flattening the hierarchical data structure of a polyhedral complex, by applying all the stored transformation matrices to the vertex data.

**Script 6.5.2 (geometric model of the applied vector p1− >p2)**

```
DEF optimize = MKPOL ∼ UKPOL

DEF MKvector (p1::IsPoint)(p2::IsPoint) =
   (optimize ∼ Tr ∼ Rot ∼ Sc):MKversork
WHERE
   Tr = T:<1,2,3>:p1,
   Rot = Rotn:< alpha, n >,
   Sc = S:<1,2,3>:<b,b,b>,
   b = VectNorm:u,
   u = p2 VectDiff p1,
   alpha = ACOS:(<0,0,1> innerProd UnitVect:u),
   n = <0,0,1> VectProd u
END;
```

In Script 6.5.3 the MKframe operator is given, which produces a model of the standard Cartesian frame, with labeling of axis, as shown in Figure 6.28.

**Script 6.5.3 (geometric model of 3D reference frame)**

```
DEF MKframe = STRUCT:<
   MKvector:<0,0,0>:<1,0,0>,
   MKvector:<0,0,0>:<0,1,0>,
   MKvector:<0,0,0>:<0,0,1>,
   (T:<1,2>:<1,1/8> ∼ S:<1,2>:<1/20,1/20>):XX,
   (T:<2,3>:<1,1/8> ∼ S:<2,3>:<1/20,1/20>):YY,
   (T:<3,1>:<1,1/8> ∼ S:<3,1>:<1/20,1/20>):ZZ >
WHERE
   XX = MKPOL:<<<-1,1,0>,<1,-1,0>,<1,1,0>,<-1,-1,0>>,
      <<1,2>,<3,4>>,<<1,2>>>,
   YY = MKPOL:<<<0,0,0>,<0,-1,1>,<0,1,1>,<0,0,-1>>,
      <<1,2>,<1,3>,<1,4>>,<<1,2,3>>>,
   ZZ = MKPOL:<<<-1,0,1>,<1,0,1>,<-1,0,-1>,<1,0,-1>>,
      <<1,2>,<1,4>,<3,4>>,<<1,2,3>>>,
END;
```

The following example shows the sum of a set of applied vectors in 3D, each given

as an ordered pair of points. The VRML file exported by the language is displayed in Figure 6.28. Notice that the last vector, shown in red in Figure 6.28, is the sum of the previous three.

**Script 6.5.4 (Applied vectors modeling)**

```
STRUCT:< MKframe,
   MKvector:<0.5,-0.1,0>:<1.1,-0.7,0.3>,
   MKvector:<1.1,-0.7,0.3>:<2.1,-0.2,-0.3>,
   MKvector:<2.1,-0.2,-0.3>:<1.4,0.3,-0.5>,

   MKvector:<0.5,-0.1,0>:<1.4,0.3,-0.3> COLOR RED >;
```

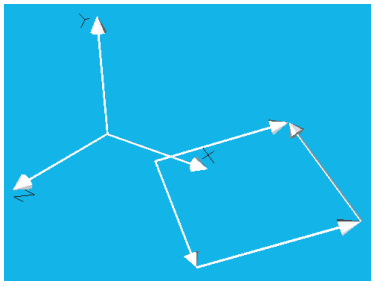

**Figure 6.28** Geometric modeling of Cartesian frame and some applied vectors

**Example 6.5.1 (Linear ramp)**
First of all, we define a function `Ramp` in Script 6.5.5, which generates a parametrized linear stair as a complex of 2D polygons in $\mathbb{E}^3$.

In particular, this function accepts as input the three dimensions of a step, i.e. the *width*, *depth* and *height*, denoted as `x,y,z`, respectively, according to the Cartesian axes they are parallel to. The partial function generated by actual values of such parameters is then applied to an integer `n`, which specifies the number of steps of the ramp.

**Script 6.5.5 (Linear ramp)**

```
DEF Ramp (x,y,z::IsReal)(n::IsInt) = (STRUCT~##:n):<step, T:<2,3>:<y,z>>
WHERE
   step = STRUCT:<
      (T:3:z ~ EMBED:1):step_foot,
      (R:<2,3>:(PI/2) ~ EMBED:1):step_rise >,
   step_foot = CUBOID:<x,y>,
   step_rise = CUBOID:<x,z>
END;
```

We note the use of the `PLaSM` operator `EMBED:`$n$ which, when applied to a polyhedral

complex of $E^m$, embeds it in the coordinate subspace of $E^{m+n}$ with equations

$$x_{m+1} = x_{m+2} = \cdots = x_{m+n} = 0.$$

The `EMBED:1` function is hence used here to embed some $E^2$ rectangles into the $x_3 = 0$ subspace of $E^3$.

In Figure 6.29 we show the result of the evaluation of expression

```
Ramp:<0.9,0.28,0.16>:10;
```

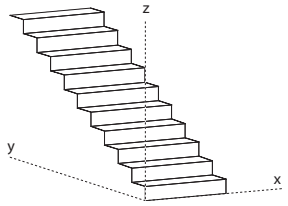


**Figure 6.29** Model generated by function `Ramp` with 10 steps

**Example 6.5.2 (Multiple linear ramp)**
The function `Ramp` is then used to generate an object `theRamp` which is multiply instanced and transformed, to generate the quite complex aggregation of stairs and landings shown in Figure 6.30.

The `Landings` body of Script 6.5.6 is a geometric expression which uses the `Q` operator, introduced in Script 1.5.5, to transform both numbers and sequences of numbers into 1D polyhedral complexes, and the Cartesian product `*` of polyhedral complexes (See Section 12).

The formal parameter `n`, in both `MultipleStair` and `Landings` generating functions, clearly denotes the number of floors.

---

**Script 6.5.6 (StairCase)**

```
DEF doubleRamp = STRUCT:< theRamp, (T:2:Y ~ S:<1,2>:<-1,-1>):theRamp > ;
DEF MultipleStair (n::IsInt) = (STRUCT ~ ##:n): < doubleRamp, T:3:Z > ;
DEF X = SIZE:1:theRamp;
DEF Y = SIZE:2:theRamp;
DEF Z = SIZE:3:theRamp;

DEF Landings (n::IsInt) = T:<1,2>:<-:X,-:X>:
   (Q:(2*X) * Q:<X,-:Y,X> * (@0 ~ Q ~ #:n):Z);

DEF StairCase = (STRUCT ~ [ MultipleStair, Landings ]):(n_floors - 1)
```

---

Two quite different values of the `StairCase` object are shown in Figure 6.30. In particular, the `StairCase` model associated with the definitions

```
DEF theRamp = Ramp:<3, 0.28, 0.16>: 8;
```

```
DEF n_floors = 2
```

is shown is Figure 6.30a. The one generated by changing such definitions to

```
DEF theRamp = Ramp:<0.95, 0.30, 0.15>: 10;
DEF n_floors = 5
```

is given in Figure 6.30b.

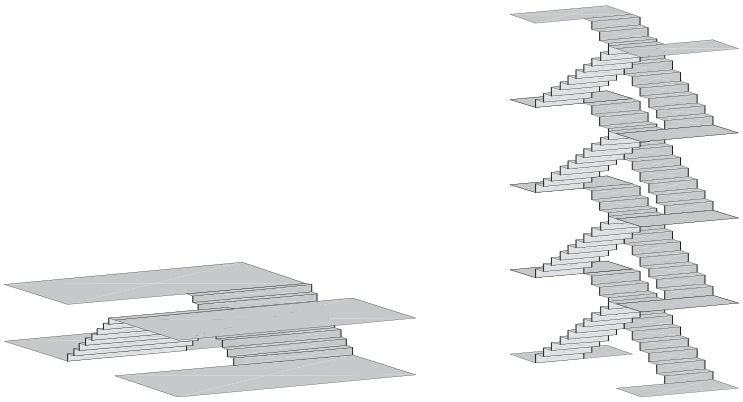

**Figure 6.30** Two instances of the implicitly parametrized `stairCase` object

**Example 6.5.3 (Helix ramp)**
The construction of a spiral stair ramp with squared basis and triangular steps is described here. Each step pair gives a turn of $\frac{\pi}{2}$ to the helix-centered angle, as shown by Figure 6.31.

In particular, the function `L_Ramp` of Script 6.5.7 depends on two positive reals `x, z`, and on a positive integer `n_turns`. The first two parameters correspond respectively to the half size of squared staircase and to the step height; the second one gives the number of $\frac{\pi}{2}$ turns, so that the number of steps is $2 \times$ `n_turns`.

---

**Script 6.5.7 (Parametric helix ramp)**

```
DEF L_Ramp (x,z::IsRealPos)(n_turns::IsIntPos) = (STRUCT ~ ##:n_turns):
   < step1, T:3:z, step2, T:3:z, R:<1,2>:(PI/2) >
WHERE
   step1 = STRUCT:<
      (S:1:-1 ~ T:1:(-:x) ~ T:3:z):step_foot,
      (R:<2,3>:(PI/2)) :step_rise >,
   step2 = STRUCT:<
      (S:2:-1 ~ T:2:(-:x) ~ T:3:z):step_foot,
      (R:<1,2>:(PI/4) ~ R:<2,3>:(PI/2)):step_rise >,
   step_foot = (EMBED:1 ~ S:<1,2>:<x,x> ~ SIMPLEX):2,
   step_rise = (EMBED:1 ~ S:<1,2>:<2**0.5,2**0.5> ~ CUBOID):<x,z>
END;
```

---

Notice that the primitive `PLaSM` function `SIMPLEX`, when applied to the integer 2, returns the 2-dimensional unit simplex of the plane, i.e. the triangle built on the basis vectors $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$.

As in several previous examples, the body of `L_Ramp` function relies on the semantics of structures, where each tensor is applied to each following complex in the sequence order.

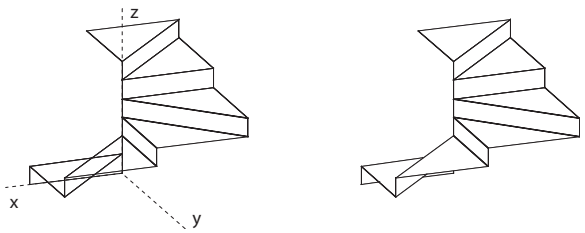


**Figure 6.31** Squared helix ramp with turn of $\frac{3}{2}\pi$ angle