9

Graphic pipelines

In this chapter we discuss the sequence of transformations that graphics data, including vertices, control points of curves and surfaces, normal vectors, etc., must undergo in order to be rendered on the screen or displayed on some other output device. Such sequence of transformations is denoted as *graphics pipeline* [FvDFH90]. It is customary to distinguish between 2D and 3D pipelines. The graphics pipelines defined by the GKS standard for 2D graphics and by the PHIGS standard for 3D graphics, respectively, are discussed in depth. Outlines of the graphics pipelines used by Open Inventor and Java 3D are also briefly introduced. A `PLaSM` implementation of the PHIGS pipeline and some examples of automatic insertion of viewpoints in `PLaSM`-generated VRML files are finally given.

9.1 2D pipeline

In the mid-1980s the vast majority of graphics systems and applications were two-dimensional. The situation has radically changed nowadays, but important classes of 2D applications remain, including Geographical Information Systems (GIS) and 2D drafting.

9.1.1 Coordinate systems

The pipeline of transformations between different coordinate systems was introduced by the ISO graphics standard GKS in order to achieve device independence, i.e. in order to display graphics data over devices with different dimensions and pixel addressing [ISO85, EKP87]. For this purpose three coordinate systems were defined:

1. world coordinates (WC);
2. normalized device coordinates (NDC);
3. device coordinates (DC).

World coordinates are used to define graphics data (parameters of graphics primitives and geometric attributes of primitives) in some suitable reference depending on the problem at hand; *normalized device coordinates* are employed to store and transform

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

data in a device-independent way; and, finally, *device coordinates* are used to display graphics data on some suitable subset of the current output device.

World Coordinates The so-called *world coordinates* (WC) define a reference system which coincides with a standard Cartesian system of two-dimensional Euclidean space. In such a reference system the more suitable coordinate values can be used for defining graphics primitives and attributes. Only a bounded rectangular subset of this space can be displayed on the graphics device. Such a user-defined subset is called a 2D *window*.

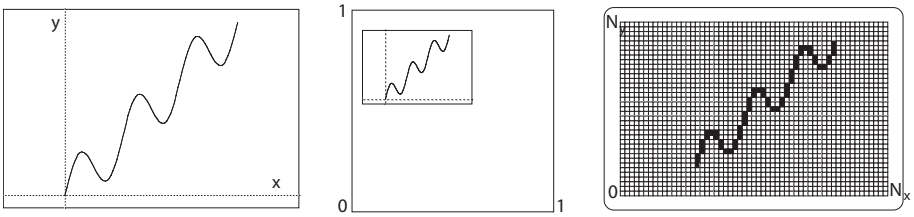

Figure 9.1 (a) Graph of the function $f : \mathbb{R} \to \mathbb{R} : x \mapsto x + 3 \sin x$, with $x \in [0, 6\pi]$, and *window* in WC (b) Normalized device coordinates and *viewport* in NDC (c) Rasterized picture in DC

Normalized Device Coordinates The standard 2D unit interval $[0, 1] \times [0, 1]$ in Euclidean space $\mathbb{E}^2$ is called the *normalized device coordinates* (NDC). This bounded subset gives a reference frame which is used to store and transform graphics data in a device-independent manner. The transformation from WC to NDC is given by defining a rectangular subset of NDC, called *viewport*, to be used as the target set where the window over graphics data must be mapped.

Device Coordinates Let us consider a device with $N_x \times N_y$ *pixels* (picture elements) which can be individually referenced in order to be, e.g., colored or displayed. The discrete 2D interval $[0, N_x - 1] \times [0, N_y - 1] \subset \mathcal{Z}^2$, corresponding to the address space of such a specific device, is called the *device coordinates* (DC). This discrete space clearly gives a "device-dependent" reference frame that can be regarded as a bijection with the set of device pixels. Notice that NDC is a square but DC is not necessarily so.

Example 9.1.1 (Function graph)
We want to generate the graph of the function $f : \mathbb{R} \to \mathbb{R} : x \mapsto x + 3 \sin x$, shown in Figure 9.1. In particular, we are interested to the graph of f restricted to the interval $[0, 6\pi]$, i.e. to the set of points

$$\{(x, f(x)) \in E^2 |\ 0 \leq x \leq 6\pi, f(x) = x + 3 \sin x\}$$

In Script 9.1.1 this function graph is approximated by a `polyline` primitive connecting 91 sampled points in the $[0, 6\pi]$ interval. We remember that the function

`SumSeqWithZero`, given in Script 7.3.1, returns the cumulative sums of the elements of the sequence it is applied to, that is:

```
SumSeqWithZero:< 1,1,1,1,1 > ≡ < 0,1,2,3,4,5 >
```

Script 9.1.1

```
( polyline
   ∼ AA:[ ID, ID + K:3 * sin ]
   ∼ SumSeqWithZero
   ∼ #:90 ): ( 6 * PI / 90 );
```

9.1.2 Normalization and device transformations

In this section we discuss how to transform a coordinate space into another coordinate space. The simple user-model of such kind of mapping, proposed by GKS, requires that two 2D intervals are defined in the domain and target spaces of the mapping, respectively. The mapping is then computed by ensuring that the first interval is affinely mapped onto the second one. The two intervals are called *window* and *viewport*, respectively. Such an approach to the computation of a coordinate transformation is called *window-viewport mapping*.

2D Extent A *2D extent*, also called *2D box* or *2D interval*, is a rectangular domain B of Euclidean space which is parallel to the reference frame. Such a box is represented by the ordered quadruple of real numbers that correspond to the coordinates of lower-left point (b_1, b_2) and upper-right point (b_3, b_4), so that:

$$B = (b_1, b_2, b_3, b_4) = [b_1, b_3] \times [b_2, b_4]$$

i.e.

$$B = \{\boldsymbol{p} = (x, y)^T |\ x \in [b_1, b_3],\ y \in [b_2, b_4]\} \subset \mathbb{E}^2$$

Normalization transformation The bijective affine mapping $\boldsymbol{T}_N$ between a *window* W_{wc} in WC and a *viewport* V_{ndc} in NDC is called the *normalization transformation*:

$$\boldsymbol{T}_N : W_{wc} \to V_{ndc},$$

where

$$W_{wc} = [w_1, w_3] \times [w_2, w_4], \qquad V_{ndc} = [v_1, v_3] \times [v_2, v_4],$$

as shown in Figure 9.2a.

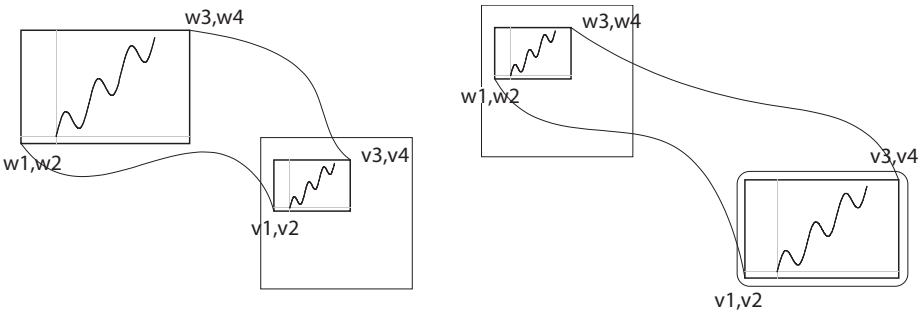

Figure 9.2 (a) Normalization transformation: WC → NDC;
(b) Device transformation: NDC → DC

Device transformation Analogously, a *device transformation* $\boldsymbol{T}_D$ is a bijective affine mapping between a *workstation window* $W_{ndc} \subset NDC$ and a *workstation viewport* $V_{dc} \subset DC$:

$$\boldsymbol{T}_D : W_{ndc} \to V_{dc},$$

where

$$W_{ndc} = [w_1, w_3] \times [w_2, w_4], \qquad V_{dc} = [v_1, v_3] \times [v_2, v_4],$$

as shown in Figure 9.2b.

9.1.3 Window-viewport mapping

A *window-viewport mapping* is, by definition, a bijective affine transformation between 2D extents $W = [w_1, w_3] \times [w_2, w_4]$ and $V = [v_1, v_3] \times [v_2, v_4]$:

$$\boldsymbol{M} : W \to V : \qquad \boldsymbol{p} \mapsto \boldsymbol{M}(\boldsymbol{p}),$$

where $\boldsymbol{M}$ is a tensor in $\mathrm{aff}\,\mathbb{E}^2$. Two methods are discussed below to compute such a transformation tensor, respectively by a direct approach and by the composition of elementary transformations.

Direct Method We use here homogeneous coordinates. The two extreme points $(w_1, w_2, 1)^T$ and $(w_3, w_4, 1)^T$ of window W are respectively mapped to the two extreme points $(v_1, v_2, 1)^T$ and $(v_3, v_4, 1)^T$ of viewport V. Also, the $\boldsymbol{M}$ tensor must transform a 2D extent, parallel to the reference frame, onto another one of the same kind. Hence, the matrix $[\boldsymbol{M}]$ will have a predictable structure, with only (unknown) coefficients of scaling and translation:

$$\begin{pmatrix} v_1 & v_3 \\ v_2 & v_4 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} a & 0 & b \\ 0 & c & d \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} w_1 & w_3 \\ w_2 & w_4 \\ 1 & 1 \end{pmatrix}$$

The previous matrix equation is equivalent to four scalar simultaneous equations in the unknown coefficients a, b, c and d:

$$\left\{\begin{array}{rcl} a\,w_1 + b &=& v_1 \\ c\,w_2 + d &=& v_2 \\ a\,w_3 + b &=& v_3 \\ c\,w_4 + d &=& v_4 \end{array}\right. , \quad \text{and hence} \quad \left\{\begin{array}{rcl} a &=& \frac{v_3 - v_1}{w_3 - w_1} \\ c &=& \frac{v_4 - v_2}{w_4 - w_2} \\ b &=& v_1 - \frac{v_3 - v_1}{w_3 - w_1} w_1 \\ d &=& v_2 - \frac{v_4 - v_2}{w_4 - w_2} w_2 \end{array}\right.$$

Composition of elementary transformations The window-viewport mapping tensor $\boldsymbol{M}$ can be derived easily by composition of some elementary transformations:

1. a translation $\boldsymbol{T}_1$ which maps the point (w_1, w_2) into the origin $\boldsymbol{o}$ of the reference system;
2. a scaling $\boldsymbol{S}_1$ of W onto the standard unit square;
3. a scaling $\boldsymbol{S}_2$ of the standard unit square onto V;
4. a translation $\boldsymbol{T}_2$ which maps $\boldsymbol{o}$ into (v_1, v_2).

In other words we have:

$$\boldsymbol{M} : W \to V : \quad \boldsymbol{p} \mapsto (\boldsymbol{T}_2 \circ \boldsymbol{S}_2 \circ \boldsymbol{S}_1 \circ \boldsymbol{T}_1)(\boldsymbol{p}),$$

where

$$\begin{array}{rcl} \boldsymbol{T}_1 &=& \boldsymbol{T}(-w_1, -w_2), \\ \boldsymbol{S}_1 &=& \boldsymbol{S}\left(\dfrac{1}{w_3 - w_1}, \dfrac{1}{w_4 - w_2}\right), \\ \boldsymbol{S}_2 &=& \boldsymbol{S}\left(v_3 - v_1, v_4 - v_2\right), \\ \boldsymbol{T}_2 &=& \boldsymbol{T}(v_1, v_2). \end{array}$$

Non-isomorphic transformation The ratio A_r between the horizontal and vertical measures of a box $B = [b_1, b_3] \times [b_2, b_4]$ is called the *aspect ratio* of the box:

$$A_r(B) = \frac{b_3 - b_1}{b_4 - b_2}$$

When $A_r(W) \neq A_r(V)$, the window-viewport mapping $\boldsymbol{M} : W \to V$ is said to be *non-isomorphic*, since it does not preserve the shape of figures. In general, a non-isomorphic mapping transforms squares into rectangles and circles into ellipses.

Isomorphic transformation In order to preserve the shape of figures for every pair W and V, it is customary that the graphics system substitutes a *computed viewport* $\widehat{V}$ of maximal area, such that

$$A_r(\widehat{V}) = A_r(W),$$

to the user-defined viewport V. Hence we may have, for the two different cases shown in rightmost part of Figure 9.3:

$$A_r(V) > A_r(W) : \left\{\begin{array}{l} \widehat{v}_1 = v_1, \quad \widehat{v}_2 = v_2, \quad \widehat{v}_4 = v_4, \\ \widehat{v}_3 = v_1 + (v_4 - v_2)\, A_r(W) \end{array}\right.$$

$$A_r(V) < A_r(W) : \begin{cases} \widehat{v}_1 = v_1, \quad \widehat{v}_2 = v_2, \quad \widehat{v}_3 = v_3, \\ \widehat{v}_4 = v_2 + (v_3 - v_1)\, A_r(W) \end{cases}$$

Different strategies can also be chosen, by imposing, e.g., that the computed viewport has the same center of the user-defined one, as shown in the central part of Figure 9.3.

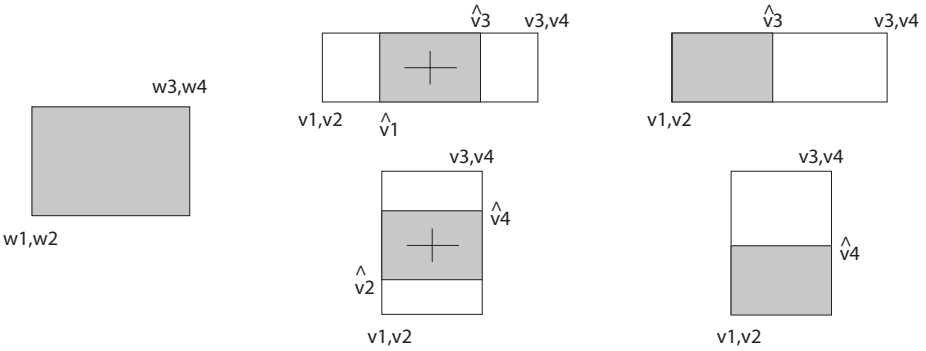

Figure 9.3 Computed viewports for preserving aspect ratio in window-viewport mapping. Two diverse strategies

Device transformation Several device transformations may be associated in GKS with the same normalization transformation. This allows simultaneous connection of several devices in one graphics application.

The simplest example of device transformation is given by mapping the NDC space onto the discrete DC space, in the hypothesis of square pixels. In this case we have, for the isomorphic mapping:

$$\boldsymbol{T}_D^{iso} : NDC \to \widehat{DC}$$

that

$$\boldsymbol{T}_D^{iso} = \boldsymbol{S}(N-1, N-1),$$

with

$$N = \min(N_x, N_y),$$

where N_x and N_y are the numbers of columns and rows of discrete device space, respectively, and

$$\widehat{DC} = [0, N-1] \times [0, N-1] \subset DC.$$

Rectangular pixel Some display devices (e.g. TV monitors) may have non-square pixels, usually with aspect ratio

$$\frac{d_x}{d_y} < 1,$$

where d_x and d_y are the pixel side measures. In this case, in order to mantain invariant the shape of figures, i.e. in order to map circles to circle, squares to squares, and so on, it is possible to define a *corrected* device transformation:

$$\boldsymbol{T}_D^{iso} : NDC \to \widehat{DC}$$

such that

$$\boldsymbol{T}_D^{iso} = \boldsymbol{S}_{pixel} \circ \boldsymbol{S}(N-1, N-1) = \boldsymbol{S}\left(\frac{d_y}{d_x}, 1\right) \circ \boldsymbol{S}(N-1, N-1)$$

where, as usual, $N = \min(N_x, N_y)$. If conversely $\frac{d_x}{d_y} > 1$, then we have $\boldsymbol{S}_{pixel} = \boldsymbol{S}\left(1, \frac{d_x}{d_y}\right)$.

Reversed y axis Some graphics monitors address the device space with encreasing y-coordinates from top to bottom of the screen. In this case the device transformation must combine a scaling tensor with a mirroring and vertical translation tensor:

$$\boldsymbol{T}_D^{iso} = \boldsymbol{T}(0, -N) \circ \boldsymbol{S}(1, -1) \circ \boldsymbol{S}_{pixel} \circ \boldsymbol{S}(N-1, N-1),$$

as shown in Figure 9.4. When displaying a graphics object on some reversed-axis device of such type, i.e. with origin of DC on its top-left point, we have the result shown in Figure 9.4b.

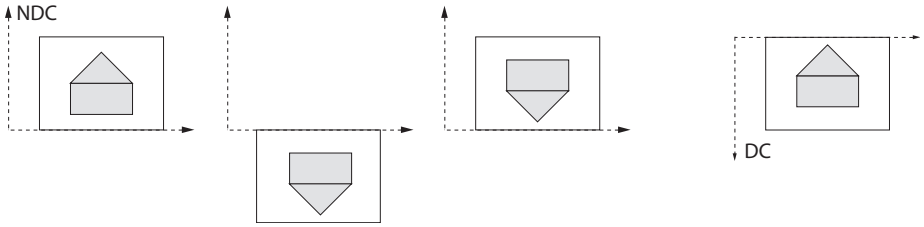

Figure 9.4 (a) Device transformation, with a reversed orientation of y axis
(b) Display on a reversed y-axis device

9.2 3D pipeline

Every 2D picture of a 3D scene is obtained by *projection*, i.e., geometrically speaking, by intersection of a plane with the bundle of straight lines which project the scene vertices from some suitable projection center. Depending on the position of this center, there are two main classes of projections: *perspective* and *parallel*, with center in a finite point or at infinity, respectively. The various types of projections are thoroughly discussed in Chapter 10. A picture of a 3D scene on the output device is produced by applying several coordinate transformations to the scene data. Such a set of coordinate transformations is often called the 3D *pipeline*, and is discussed here.

9.2.1 View model

The mechanism of projection is very similar to that of human vision, where light's rays reflected from scene reach the observer's eye and are intercepted by the retina. This is a small portion of a spherical surface, which may be thought as locally approximated by the tangent plane. The image generated by light receptors on the retina is then transmitted to the superior brain centers, where it is suitably elaborated. In particular, from the small differences between the images perceived by the two eyes, due to the small difference between the positions of "projection centers", depth information about scene points is generated.

Actually, when using computers, the projection of scene points is not computed geometrically, but is derived algebraically by applying to the scene model a linear mapping of rank two which maps a three-dimensional space onto a two-dimensional one. This approach allows production of both parallel and perspective projections by using homogeneous coordinates. To preserve depth information, needed to generate pictures with hidden parts removed, the projection mapping is always generated by the composition of a linear mapping of rank three with an ortographic projection, which removes the third coordinate while leaving invariant the remaining ones.

Camera model In the past two decades, 3D graphics systems have adopted a conceptual model of projection called the *camera model*, which is very easy to use and allows generation of both central and parallel projections with assigned geometric properties. This conceptual model of projection was developed in late 1970s by the Special Interest Group in Graphics of Association for Computing Machinery (ACM Siggraph), as a part of the more general 3D *Core system*, that became ANSI standard in those years. Subsequent graphics libraries and environments have introduced only small variations on this approach, that we are going to discuss in the following sections. In particular, we make reference to the PHIGS [ANSI87, ISO89, HHHW91, GG91, Gas92] specification of the camera model.

View parameters Four 3D vectors, defined in WC, are called *view parameters* in ANSI Core. They completely specify the picture resulting from a specific projection, also called *view*, of the scene. The picture resulting from a projection is returned in a reference system linked to the projection plane. This system is called *uvn* or *view system* in ANSI Core, and *view reference* in PHIGS. A *view model* is a set of values for view parameters. The four vector parameters which specify the projection, i.e. the *view*, are the following:

1. *Center of Projection* (COP) is the common point of projecting lines. It coincides with the observer's position. It is substituted by the vector called *Direction of Projection* (DOP) for parallel projections, where the projection center is improper, i.e. is set at infinity.
2. *View Reference Point* (VRP) is the point targeted by the observer, at the intersection of the view axis and the view plane. It is assumed as the origin of the view reference system.
3. *View Up Vector* (VUV) is a vector used to orientate the projected picture. The v axis of view system is parallel to the projection of view up vector.

4. *View Plane Normal* (VPN) is a vector normal to the view plane. It is assumed as the direction of the n axis of view reference system.

A further discussion and several examples of view parameters are given in Sections 10.1 and 10.2, where we discuss how to generate the more useful types of projections used in technical drawings.

9.2.2 Coordinate systems

In ISO graphics standard PHIGS, five different coordinate systems are used:

1. Modeling Coordinates (MC)
2. World Coordinates (WC3)
3. View Reference Coordinates (VRC)
4. Normalized Projection Coordinates (NPC)
5. Device Coordinates (DC3)

Such systems are connected by four coordinate transformations. The composition of such transformations is called the *3D pipeline*.

Each reference system, including device coordinates, in the PHIGS's 3D pipeline is fully three-dimensional. The acronyms WC3 and DC3 are used to distinguish them from the corresponding 2D coordinates of GKS.

Modeling Coordinates (MC) are coordinates which are *local* to each structure in the *structure network*. It is very useful and natural that each component substructure in a hierarchical model can be modeled by using a local coordinate frame. The local coordinates of each structure are called *modeling coordinates*.

A *traversal* algorithm, that we know (from Section 8.3) to be a *Depth First Search* (DFS), is used to linearize the structure network and transform all component substructures, i.e. the graphics primitives there contained, to the same coordinate frame, say, to world coordinates.

World Coordinates (WC3) are the global coordinates of the structure posted to a workstation, in order to be displayed or interactively modified. Often, world coordinates coincide with the local coordinates of the root (i.e. the initial structure) of some hierarchical structure network.

World coordinates are used as the common reference frame for all the graphics primitives (graphics data) contained in every component of a 3D scene. Such a reference frame is also used to define the camera position and orientation in a view model.

View Reference Coordinates (VRC) are used to establish the position of the observer in the scene, and the orientation of the view. With respect to the camera analogy, this view reference coordinates system is uniquely determined by the position and orientation of the camera.

The *view reference coordinate system*, or *uvn* system, has its origin in the view reference point (VRP), n axis parallel to the view plane normal (VPN), and v axis

parallel to the projection of view-up vector (VUV), all given in WC3. The u axis is then uniquely determined. The view plane normally coincides with the $n = 0$ subspace

The *projection reference point* (PRP) given in VRC, and the *type of projection* (either parallel or perspective) completely specify the projection. The 2D *window limits* on view plane in VRC and the *front, and back plane distances* (given as n values in VRC) specify the *view volume* used to clip the scene to the desired portion.

Normalized Projection Coordinates (NPC) are used to describe the type of projection desired. In particular this is fixed by specifying the relative position to the observer and view plane, as well as some additional parameters which define what portion of the scene must be rendered on the output device.

The view mapping transformation first transforms the view volume given in VRC into a *canonical volume in NPC*, then into a 3D viewport contained in the 3D standard interval $[0, 1]^3$, where the transformed and clipped data are mantained in a device-independent manner.

Normalized projection coordinates are also used to compose different pictures. For example, a Monge projection (which is a composite orthographic projection where different views are composed simultaneously — see Section 10.2.2) is obtained by connecting different VRC systems to the same NPC. Different views can be composed in NPC by specifying different *projection viewports*.

The third coordinate of NPC system is the *perspective depth* of scene points and is used to compute the relative occlusion between scene parts. The actual projection of the scene is simply obtained by eliminating this coordinate, both in perspective and parallel cases.

Device Coordinates (DC3) are discrete 3D coordinates depending on the device. Such coordinates are three-dimensional in PHIGS, where advanced graphics devices are considered fully 3D.

Sometimes the device address space is bjiectively mapped onto the set of *voxels* (volume elements), really 3D, that allow for dynamic volume visualizations. More often a two-dimensional array of reals, called a *z-buffer*, is closely coupled to a 2D *frame buffer*, which accommodates a color index for each point displayed on the raster device. The z-buffer algorithm, which drives the rasterization of graphics primitives producing a hidden-surface removed picture of the scene, is discussed in Section 10.3.6.

9.2.3 Transformations of coordinates

A pipeline of four transformations of coordinates is associated with the five reference systems of PHIGS systems:

1. Structure Network Traversal
2. View Orientation
3. View Mapping
4. Workstation Transformation.

The Structure Network Traversal was already discussed in Section 8.3 and is only briefly recalled here; the other transformations are discussed in detail in the following subsections.

Structure Network Traversal is a composite algorithmic transformation from modeling coordinates that are local to the various hierarchical structures, to the global world coordinates:

$$MC \rightarrow WC3.$$

As we already know from Section 8.3, this algorithm clips the primitives certainly outside the view volume, while transforming the remaining ones to WC3 coordinates. The mapping from MC to WC3 is performed by traversing the structure network with a DFS, and multiplying each encountered primitive times the Current Transformation Matrix (CTM). The traversal algorithm returns the set of clipped primitives in the coordinates of the posted root stucture, assumed as WC3.

View Orientation is the mapping from world coordinates to view reference coordinates:

$$WC3 \rightarrow VRC.$$

This transformation is a rigid (possibly improper) transformation i.e. is composed by a translation, a rotation and possibly by an elementary reflection, to be considered only when WC3 and VRC systems have different orientation. In recent years world coordinates and view reference coordinates are both right-handed, so that the reflection coincides with the identity mapping.

View Mapping is the mapping from view reference coordinates to normalized projection coordinates:

$$VRC \rightarrow NPC.$$

This transformation maps the view volume in VRC onto a canonical volume $[-1, 1] \times [-1, 1] \times [-1, 0]$, then onto some NPC viewport. The first step is accomplished by composing a translation, a shearing, a scaling and possibly a perspective transformation (mathematically an affine homology). Such a transformation allows unification of the treatment of both parallel and perspective projections. A 3D window-viewport mapping, analogous to the transformation discussed in Section 9.1, is finally applied to transform the canonical volume onto some NPC viewport.

Workstation Transformation is the mapping from normalized projection coordinates to discrete 3D device coordinates:

$$NPC \rightarrow DC3.$$

It is used to transform a 3D workstation-window in NPC into a 3D workstation-viewport in DC3, both defined as 3D extents parallel to the coordinate frames. As in the 2D case, it is composed by translations and scaling transformations.

9.2.4 View orientation

The view-orientation transformation is a translation followed by a rotation, possibly improper, that moves the VRP to the origin, the VPN to the z axis, the projection of VUV to the y axis, and the cross-vector of the first two to the x axis.

This roto-translation may be followed by a z-reflection if the WC3 system and the VRC system have different orientations, e.g. if the first is right-handed whereas the second one is left-handed, or vice versa.

Notice that the view-orientation transformation is exactly the same for both the parallel and the perspective case.

$$\boldsymbol{VO} = \boldsymbol{R}(VPN, VUV) \circ \boldsymbol{T}(-VRP)$$

where

$$\boldsymbol{T}(-VRP) = \begin{pmatrix} 1 & 0 & 0 & -vrp_x \\ 0 & 1 & 0 & -vrp_y \\ 0 & 0 & 1 & -vrp_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\boldsymbol{R}(VPN, VUV) = \begin{pmatrix} r_{ux} & r_{vx} & r_{nx} & 0 \\ r_{uy} & r_{vy} & r_{ny} & 0 \\ r_{uz} & r_{vz} & r_{nz} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

with

$$\boldsymbol{r}_n = VPN / \|VPN\|$$

$$\boldsymbol{r}_u = VUV \times \boldsymbol{r}_n / \|VUV \times \boldsymbol{r}_n\|$$

$$\boldsymbol{r}_v = \boldsymbol{r}_n \times \boldsymbol{r}_u$$

The transformation can also be seen as change of coordinates. The three vectors of the new basis, i.e., respectively, $\boldsymbol{VO}(\boldsymbol{r}_u)$, $\boldsymbol{VO}(\boldsymbol{r}_v)$ and $\boldsymbol{VO}(\boldsymbol{r}_n)$, denoted here as x, y and z, are traditionally named u, v and n in graphics systems. Since we cannot invent new names for the basis of each coordinate system in the 3D pipeline, we will continue to use the standard names x, y and z for basis vectors of each pipelined reference frame.

View Model We show here the great simplicity of the specification of a view model in a PHIGS-like graphics system. It may be useful to remember that VRP is the *target* point in the camera analogy, and that VPN is the normal to the view plane, so that it determines the *orientation* of the camera axis. Conversely, the VUV vector defines the vertical direction of the projected picture, i.e. the *rotation* of the camera about its axis. Let us remember also that PRP, the *window* limits and the *front* and *back* planes are given in VRC.

Script 9.2.1 (View model)

```
DEF vrp = < 0, 2.0, 2>;
DEF vpn = < 1, 0, 0 >;
DEF vuv = < 0, 1, 1 >;
DEF prp = < 0, 0, 25 >;
DEF front = 10;
DEF back = -1;
DEF window = < -4, -3, 4, 3 >;
```

Example 9.2.1 (View model definition)
A set of values for the parameters described above will be called a *view model* in this book. An example of view model is given in Script 9.2.1.

Implementation As we already know, the `ViewOrientation` tensor is composed of a translation followed by a rotation. The first one moves the origin to the PRP; the second one moves three unit normal vectors `Ru`, `Rv` and `Rn`, depending on VPN and VUV, in the unit vectors of reference frame. The implementation given in Script 9.2.2 directly translates the transformation formulas discussed in Section 9.2.4.

Script 9.2.2

```
DEF ViewOrientation = RotVRC ∼ TranslVRC;
DEF TranslVRC = T:<1,2,3>:(AA:-:vrp);
DEF RotVRC = (MAT ∼ TRANS): <<1,0,0,0>,AL:<0,Ru>,AL:<0,Rv>,AL:<0,Rn>>
WHERE
   Ru = UnitVect:(vuv VectProd Rn),
   Rv = Rn VectProd Ru,
   Rn = UnitVect:vpn
END;
```

In Figure 9.5 we show the world positions of the house model and the view volume previously given.

In Figure 9.6 we show the result of application of tensor `ViewOrientation` to the `WCscene` model defined by Script 9.4.4. The generating expression of the model shown in Figure 9.6 is

```
ViewOrientation: WCscene;
```

Let us note that all the pictures of this section use both a Monge's and a dimetric projection. The direction of the z axis in the dimetric projection is always vertical in the pictures. The Monge's images instead mantain the axis orientation typical of this kind of composite projection.

9.2.5 View mapping

The view mapping transforms the *view volume* in VRC onto the *canonical volume* in an intermediary coordinate system, and then maps this volume onto the *3D viewport* in NPC. The canonical volume is defined as the 3D interval $[-1,1]\times[-1,1]\times[-1,0]$ in the parallel case, and the truncated pyramid with vertex in the origin, squared basis

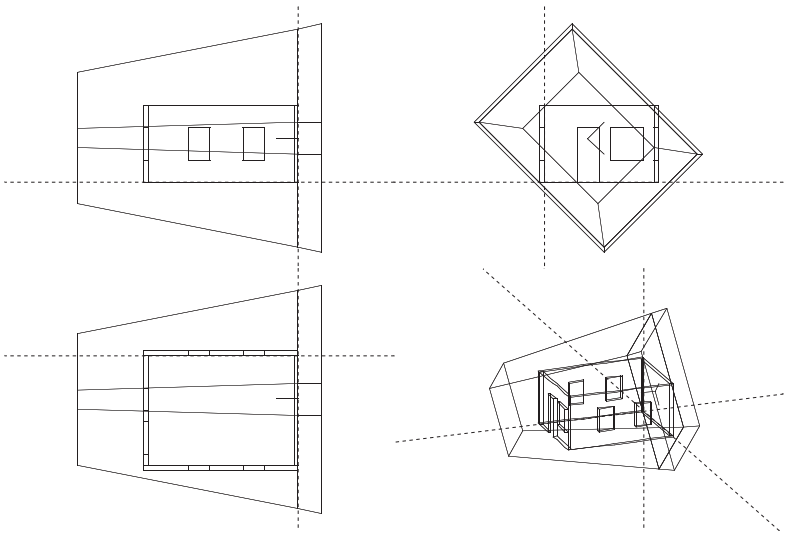

Figure 9.5 House model and perspective view volume in world coordinates

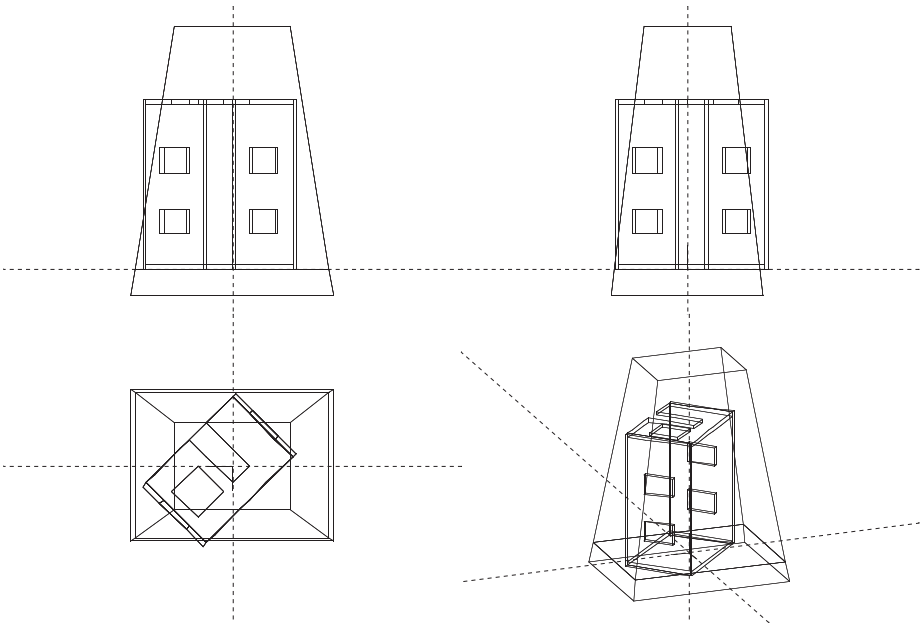

Figure 9.6 House model in VRC according to the view orientation transformation

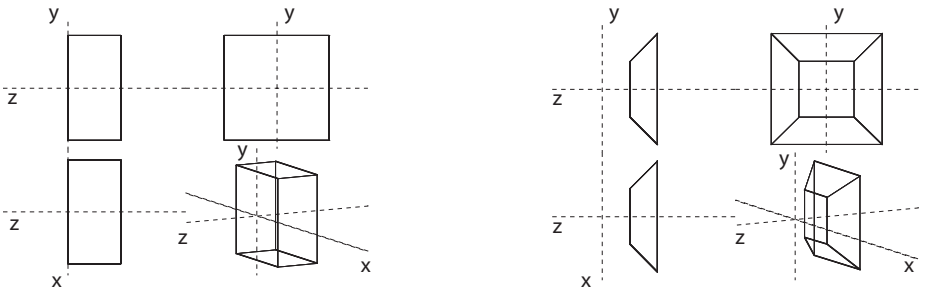

Figure 9.7 Canonical view volume: (a) parallel case (b) perspective case

in the plane $z = -1$ and side faces into the planes with unit slope, in the perspective case.

In perspective case an affine homology[1] is applied to the pyramidal canonical volume, which is transformed to the parallelepiped canonical volume, then mapped to the 3D viewport. In such canonical volume, two points aligned with the observer belong to the line of equations $x = a, y = b$, and differ only for their *perspective depth*, which coincides with the z coordinate.

In the following we distinguish between the view mapping $\boldsymbol{VM}_{per}$ in the perspective case, from the view mapping $\boldsymbol{VM}_{par}$ in the parallel case.

Parallel case The view mapping tensor $\boldsymbol{VM}_{par}$ is the composition of a shearing, a scaling and a translation:

$$\boldsymbol{VM}_{par} = \boldsymbol{T}_{par} \circ \boldsymbol{S}_{par} \circ \boldsymbol{H}_z.$$

The shearing $\boldsymbol{H}_z$ must shear the Direction Of Projection (DOP) vector in $(0, 0, dop_z, 1)^T$, thus shearing the possibly oblique view volume into a straight one. The DOP vector in VRC is defined in PHIGS as difference between the Center of Window (CW) and the Projection Reference Point (PRP):

$$DOP = CW - PRP = \begin{pmatrix} (u_{max} + u_{min})/2 \\ (v_{max} + v_{min})/2 \\ 0 \\ 1 \end{pmatrix} - \begin{pmatrix} prp_u \\ prp_v \\ prp_n \\ 1 \end{pmatrix}$$

So, it must be

$$\begin{pmatrix} 0 \\ 0 \\ dop_z \\ 1 \end{pmatrix} = \boldsymbol{H}_z \begin{pmatrix} dop_x \\ dop_y \\ dop_z \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & sh_x & 0 \\ 0 & 1 & sh_y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} dop_x \\ dop_y \\ dop_z \\ 1 \end{pmatrix},$$

and hence

$$sh_x = -\frac{dop_x}{dop_z}, \quad sh_y = -\frac{dop_y}{dop_z}.$$

[1] A bijective mapping of lines to lines and planes to planes that preserves the incidence relationship.

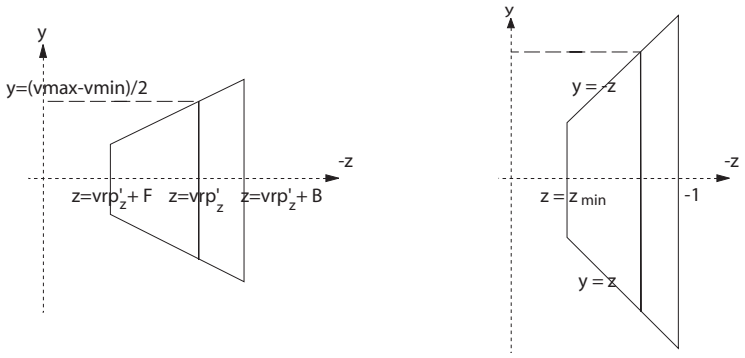

Figure 9.8 Scaling to canonical view volume of parallel case

After the action of such tensor, the bounds of view volume are

$$u_{min} \leq x \leq u_{max}, \quad v_{min} \leq y \leq v_{max}, \quad B \leq z \leq F$$

to be scaled and translated to the canonical volume

$$-1 \leq x \leq 1, \quad -1 \leq y \leq 1, \quad -1 \leq z \leq 0,$$

so that

$$\boldsymbol{T}_{par} = \boldsymbol{T}\left(-\frac{u_{min}+u_{max}}{2}, -\frac{v_{min}+v_{max}}{2}, -F\right),$$

$$\boldsymbol{S}_{par} = \boldsymbol{T}\left(\frac{2}{u_{max}-u_{min}}, \frac{2}{v_{max}-v_{min}}, \frac{1}{F-B}\right).$$

Perspective case The view mapping tensor $\boldsymbol{VM}_{per}$ is the composition of a translation of PRP, that coincides with COP in this case, to the origin, followed by a shearing to make straight the view pyramid, and by a composite scaling to map the result into the canonical volume:

$$\boldsymbol{VM}_{per} = \boldsymbol{S}_{per} \circ \boldsymbol{H}_z \circ \boldsymbol{T}(-PRP),$$

where:

1. $\boldsymbol{T}(-PRP)$ moves the center of projection to the origin;
2. the shearing $\boldsymbol{H}_z$ tensor coincides with the one of parallel case;
3. the scaling tensor can be decomposed as: $\boldsymbol{S}_{per} = \boldsymbol{S}_2 \circ \boldsymbol{S}_1$.

where $\boldsymbol{S}_1$ maps the straight view pyramid onto a unit slope pyramid:

$$\boldsymbol{S}_1 = \boldsymbol{S}(\frac{-2\ vrp'_z}{u_{max}-u_{min}}, \frac{-2\ vrp'_z}{v_{max}-v_{min}}, 1)$$

and where $\boldsymbol{S}_2$ uniformly scales the three-space to move the $z = B$ plane (the `Back plane`) to the $z = -1$ plane:

$$\boldsymbol{S}_2 = \boldsymbol{S}(\frac{-1}{vrp'_z+B}, \frac{-1}{vrp'_z+B}, \frac{-1}{vrp'_z+B}^{\ T})$$

Notice that vrp'_z is obtained by mapping the VRC origin by the translation to PRP and by the subsequent shearing:

$$VRP' = (\boldsymbol{H}_z \circ \boldsymbol{T}(-PRP))(0,0,0,1)^T$$

Implementation The `ViewMapping` tensor maps the view volume from VRC to NPC. The NPC view volume must coincide, in the perspective case, with the pyramid centered in the origin and with squared basis $[-1,1] \times [-1,1]$ in the plane of equation $z = -1$.

As seen in Section 9.2.5, the view mapping tensor $\boldsymbol{VM}_{per}$ is composed of a translation tensor `T_per`, by a tensor shearing H_z and by a scaling tensor `S_per`. Also in this case, the code given in Script 9.2.3 implement very directly the formulas given in Section 9.2.5.

Notice that `umin`, `vmin`, `umax` and `vmax` are generated by selecting the first, second third and fourth component of 2D `window`, and that the z component of VRP is obtained by opening the polyhedral data structure and by extracting the origin of VRC exposed to the action of `T_per` and `SH_per`. The `MK` operator that transforms a point into a 0-dimensional polyhedron, so that tensors can apply to it, is given in Script 3.3.15.

Script 9.2.3

```
DEF ViewMapping = S_per ∼ SH_per ∼ T_per;
DEF T_per = T:<1,2,3>:(AA:-:prp);
DEF SH_per = MAT:
   << 1, 0, 0, 0 >,
    < 0, 1, 0, dopx / dopz >,
    < 0, 0, 1, dopy / dopz >,
    < 0, 0, 0, 1 >>
WHERE
   dopx = (umin + umax)/2 - s1:prp,
   dopy = (vmin + vmax)/2 - s2:prp,
   dopz = 0 - s3:prp
END;
DEF S_per = S:<1,2,3>:<sx,sy,sz>
WHERE
   sx = (2 * vrp_z)/((umax - umin)*(vrp_z + back)),
   sy = (2 * vrp_z)/((vmax - vmin)*(vrp_z + back)),
   sz = -1/(vrp_z + back)
END;

DEF umin = S1:window; DEF vmin = S2:window;
DEF umax = S3:window; DEF vmax = S4:window;
DEF vrp_z = (s3 ∼ s1 ∼ s1 ∼ UKPOL ∼ SH_per ∼ T_per ∼ MK): <0,0,0>;
```

In Figure 9.9 we show the result of application of `ViewMapping` tensor to the model generated by the previous step. The generating expression of the model shown is in this case:

```
(ViewMapping ∼ ViewOrientation): WCscene;
```

9.2.6 Perspective transformation

The so-called *perspective transformation* [FvDFH90], mathematically an affine homology, maps the canonical pyramid volume of central projections onto the canonical parallelepiped volume of parallel projections.

Such a transformation moves the origin to the improper point of z axis and the front plane to the plane $z = 0$, while keeping invariant the back plane, of current equation $z = -1$. Such a perspective tensor is associated with a matrix

$$\boldsymbol{P} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{1+z_{min}} & \frac{-z_{min}}{1+z_{min}} \\ 0 & 0 & -1 & 0 \end{pmatrix}, \quad z_{min} \neq -1 \tag{9.1}$$

Example 9.2.2 (Perspective transformation)
Let us consider the vertices

$$\boldsymbol{r} = \begin{pmatrix} -z_{min} \\ -z_{min} \\ z_{min} \\ 1 \end{pmatrix}^T \quad \text{and} \quad \boldsymbol{s} = \begin{pmatrix} 1 \\ 1 \\ -1 \\ 1 \end{pmatrix}^T$$

of canical view volume of Figure 9.7b, where $\boldsymbol{r}$ is at the intersection of planes $x = -z$, $y = -z$ and $z = z_{min}$, and $\boldsymbol{s}$ is at the intersection of planes $x = -z$, $y = -z$ and $z = -1$. They are respectively mapped to

$$\boldsymbol{P}(\boldsymbol{r}) = \begin{pmatrix} -z_{min} \\ -z_{min} \\ 0 \\ -z_{min} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} \quad \text{and to} \quad \boldsymbol{P}(\boldsymbol{s}) = \begin{pmatrix} 1 \\ 1 \\ -1 \\ 1 \end{pmatrix}$$

Perspective Transformation The `perspTransf` tensor, given in Script 9.2.4, maps the canonical view volume of central projections onto the canonical view volume of parallel projections, the 3D extent $[-1, 1] \times [-1, 1] \times [-1, 0]$.

In order to implement such a tensor it is necessary to remember that `PLaSM`, for the purpose of allowing for easy dimensional-independence of geometric operations, has conventionally chosen the first coordinate as the homogeneous one. Hence the matrix (9.1) of affine homology discussed in Section 9.2.6 must be accordingly modified. The desired result may be obtained by applying the cyclic permutation

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix}$$

to the matrix rows and columns. The `perspTransf` tensor is hence defined as in the following Script.

Script 9.2.4

```
DEF perspTransf = (MAT ~ INV):<
  <      0,           0, 0,      -1>,
  <      0,           1, 0,       0>,
  <      0,           0, 1,       0>,
  <-:z_min/(1+z_min), 0, 0, 1/(1+z_min)> >
WHERE
  z_min = -:(vrp_z + front)/(vrp_z + back)
END;
```

The result of application of the `perspTransf` tensor to the model generated at the previous step is shown in Figure 9.10. The PLaSM expression which generates the model represented in such figure is

```
(perspTransf ~ ViewMapping ~ ViewOrientation): WCscene;
```

9.2.7 Workstation transformation

The workstation transformation maps a 3D workstation window in NPC onto a 3D workstation viewport in DC3. This mapping is similar to the 2D one defined by GKS and discussed in Section 9.1.2. It is composed of a translation that moves the NPC point of minimum coordinates to the origin, then by a scaling of the 3D extent to the size of the viewport and by a final translation of the origin to the DC3 viewport point of minimum coordinates. The device transformation is applied to the geometric data of primitives, including the control points of curves and surfaces (see Section 11.2). Such primitives are then rasterized in 3D, often using some variation of the z-buffer approximated algorithm for removing the hidden parts. Exact algorithms for hidden-surface removal would have already been applied in NPC coordinates.

So, if $W = [w_1, w_4] \times [w_2, w_5] \times [w_3, w_6] \subset NPC$, and $V = [v_1, v_4] \times [v_2, v_5] \times [v_3, v_6] \subset DC3$, then we have

$$\boldsymbol{T}_D : NPC \to DC3$$

such that

$$\boldsymbol{T}_D = \boldsymbol{T}(v_1, v_2, v_3) \circ \boldsymbol{S}(\frac{v_4 - v_1}{w_4 - w_1}, \frac{v_5 - v_2}{w_5 - w_2}, \frac{v_6 - v_3}{w_6 - w_3}) \circ \boldsymbol{T}(-w_1, -w_2, -w_3)$$

View volume clipping It is very convenient to perform in NPC the detail clipping of geometric primitives to the boundaries of the view volume. Remember that a fast culling of a hierarchical scene graph can already be performed in WC3 by pruning its covering tree at traversal time, on the basis of an intersection test between the containment box of current node (root of substructure) and the containment box of the view volume.

A clipping in NPC after perspective transformation is numerically convenient, because the intersection of primitives with a 3D extent parallel to the reference frame allows the use of very simple inequalities for boundary half-spaces, i.e.:

$$-x \leq 1, \qquad x \leq 1,$$

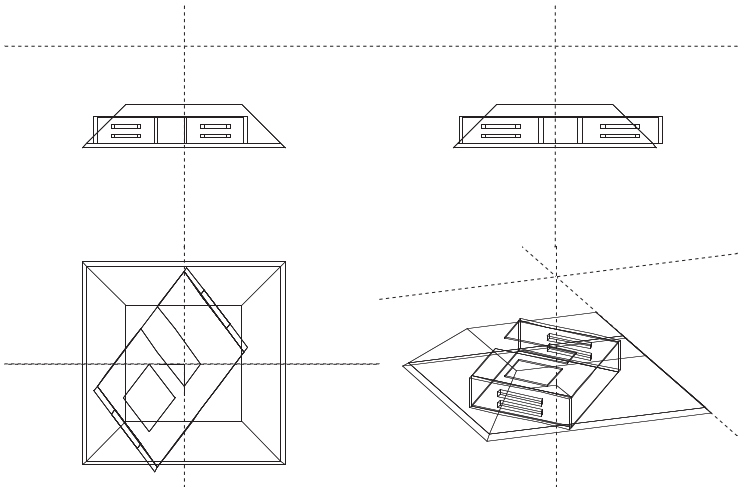

Figure 9.9 House model (in NPC) after the orientation and the view mapping transformation

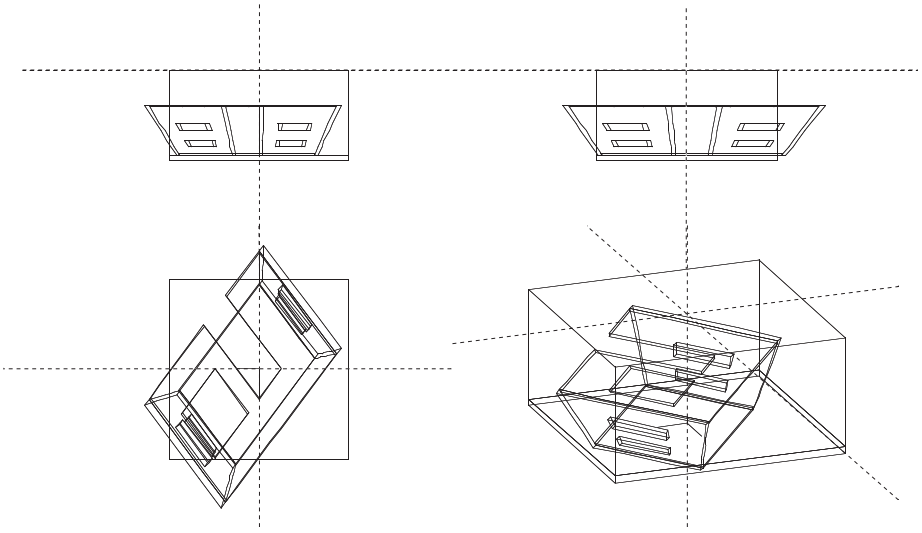

Figure 9.10 Canonical volume after perspective transformation

$$-y \leq 1, \qquad y \leq 1,$$

$$-z \leq 1, \qquad z \leq 0.$$

Such a clipping is easily microcoded on the graphics boards. The result of applying in NPC a clipping operation to the result of perspective transformation is shown in Figure 9.11. A possible PLaSM generating expression is given in Script 9.2.5.

Script 9.2.5

```
DEF WCscene = < WCvolume, house >;
DEF clipping = <1,2,3> && <1,2,3>;
DEF perspPipeline = perspTransf ~ ViewMapping ~ ViewOrientation;

(clipping ~ AA:perspPipeline): WCscene;
```

Some comments on the code in Script 9.2.5 are probably needed. First, notice that the `clipping` function is just an alias for the operator of intersection of polyhedral complexes of full dimensionality in 3D. The meaning of the `perspPipeline` function is straightforward.

Window-viewport mapping Two window-viewport mappings are supported by PHIGS, between VRC and NPC as well as between NPC and DC3. The first one is slightly simpler, since the mapping domain is the canonical volume $[-1, 1] \times [-1, 1] \times [-1, 0]$, which is mapped onto a NPC viewport. As we know, this kind of mapping is a composition of a translation, a scaling and a further translation.

To finally obtain some realistic image from our example, we only need to scale our result to the size of VRC window. A further mapping to some NPC viewport should subsequently apply:

```
(S:<1,2>:<(umax-umin)/2,(vmax-vmin)/2> ~ clipping
~ AA:perspPipeline): WCscene;
```

The geometric result of the previous expression is shown in Figure 9.12. The final 2D image algebraically generated by eliminating the third coordinate, is shown in Figure 9.13, where both a *wire-frame* image of projected model and a *hidden-surface removed* image are reported. We discuss in Section 10.3 how the insertion of perspective transformation in the 3D pipeline allows for a more efficient solution to the problem of removing the hidden parts of the scene.

9.3 Other implementations

A brief introduction to different models of 2D and 3D graphics approaches and pipelines is given in this section. Our aim is to help the reader to make connections and look at the similarities and differences between the different implementations of the same concept. According to the ancient Romans, we believe that history, even in the short run of two/three decades, is the best *magistra*.

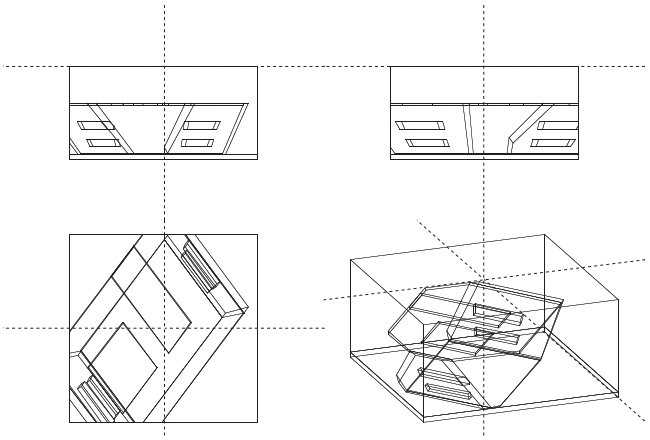

Figure 9.11 Canonical volume after perspective transformation and clipping

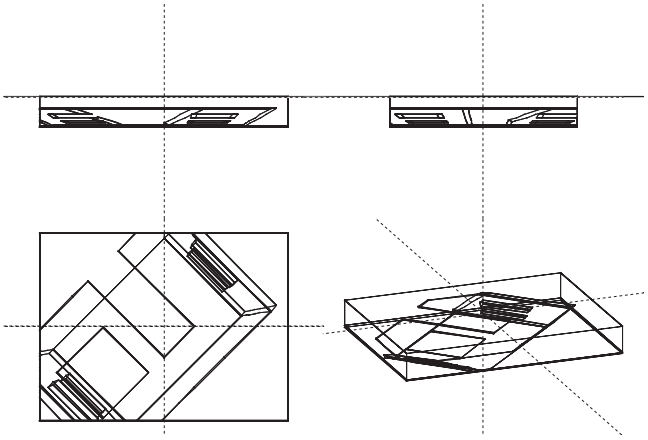

Figure 9.12 Canonical volume after perspective transformation, clipping and scaling to the window 2D

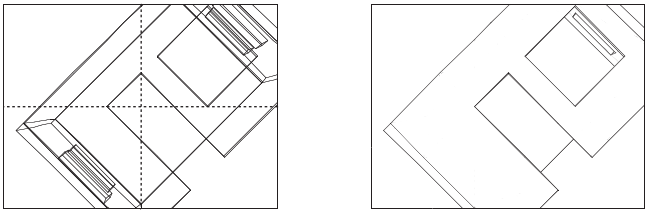

Figure 9.13 Final projected image: with (b) and without (a) removal of hidden lines

9.3.1 Scalable vector graphics (SVG)

SVG is a recent standard for vector 2D graphics on the web defined by the *W3 Consortium* [Svg02]. When using the SVG format, a 2D graphics at vector precision will be created by the browser based on plain text instructions contained in a SVG file or directly embedded in a HTML document. No image files are necessary. A working draft specification was released on 2001 by the World-Wide Web Consortium, partly inspired by two earlier specifications by Microsoft, Macromedia, and others, and by Adobe, Netscape, Sun, and others.

When looking at the SVG specification document [Svg02], the authors were very impressed by the similarities with the 2D standard graphics codified by GKS, as a further signal of how pervasive and influential was the 1980s' movement for device-, platform-, application- and language-independent standardization of graphics methods.

The first thing we should note is that SVG is *plain text*. The code can live within an HTML document with no other files involved. This one is the main difference from Flash graphics, that is a binary format, that requires either ad hoc interactive tools or a dedicated API to create, i.e. much more than a plain text editor. The second thing to be aware of is that SVG is written in XML, that is a powerful and simple way to present structured information on the web.

Primitives and attributes SVG offers several predefined primitives, and gives the user full control of their appearance, in particular by controlling the *Fill* and *Stroke* attributes. *Fill* means painting the interior of the shape by specifying the color and even making the inside of a shape partially transparent. *Stroke* means painting along the shape outline. Several built-in primitives are available, including:

1. rectangles (with optional rounded corners)
2. circles
3. ellipses
4. pie slices
5. polygons
6. paths

For example, `circle` may be defined as:

```
<circle style="fill: red; stroke: yellow"/>
```

The `<path>` element allows a combination of sequentially straight lines, cubic Bézier curves, elliptic or circular arcs, so that shapes can be made by any combination. And the user can have control over the color, width, antialiasing, and opacity of the stroke as well as how outlines end or come together. Also, it is possible to fill any shape with a GIF or JPEG image or make the image define a *pattern* tile to fill the space. And it is possible to create a pattern that would cover the stroke of a shape.

Graphic text Graphic text can easily be inserted into a drawing. `<text>` is a new element for defining what your text is and what styling information you want to apply to it. The `x` and `y` attributes of a text elements, in particular, allow absolute positioning

of text string on a web page. The text elements may be positioned along a `<path>`, as you can do, say, in *Adobe Illustrator*. Imagine drawing a curving path and then having the base line of your text follow that curve.

Grouping and naming Multiple graphics elements can be grouped hierarchically and considered together enclosed in a `<g>` tag. A group or individual graphics element can be *named* and *instanced* several times in different positions and orientations and also stored for later use. Similarly, it is possible to define a set of characteristics in one part of the document and then apply those characteristics somewhere else, very much as is done with classes in CSS.

Each drawing can be positioned anywhere on a web page. SVG relies on Cascading Stylesheets (CSS) to take charge of positioning on the page as well as other visual parameters. Several graphics layers can be positioned over each other by any desired ordering, making use of the CSS property called `z-index`.

Transformations and effects Furthermore, graphics elements, i.e. vector shapes, images, and text, can be subject to the following effects:

1. Clipping paths
2. Masks
3. Gradients
4. Visibility
5. Opacity
6. Transformations
7. Filter effects
8. Animation
9. Scripting

When a *clipping path* is applied to a region, only the area within that path will be visible. It is even possible to use a `<text>` as a clipping path. Is also possible to use any other graphic as an *alpha mask*, getting close to *Photoshop*-style techniques on the Web. Linear or radial *gradients* allow smooth transitions from one color to another within any shape. A *visibility* or *opacity* property for a single graphics element or a group may be set. *Transformations* include rotation, shearing, scaling, and translation. *Animations* are possible, because the language allows for JavaScript *scripting*, used to manipulate SVG graphics. In particular, any graphic, grouping, path, image, or text can be assigned any of the standard HTML *event handlers* (onclick, onmouseover, onmouseout, onload, and so on). Last but not least, it is possible to type in some code that would apply *filters* such as a *Gaussian blur* or *diffuse lighting* effects to SVG graphics or text.

SVG exporting Some limited exporting to SVG files is possible for 2D `PLaSM` geometric objects. For this purpose the built-in `SVG` primitive is used. An example of exported SVG file is shown in Figure 9.14. The exporting syntax is given below.

```
SVG: object: width_cm: 'filename.svg'
```

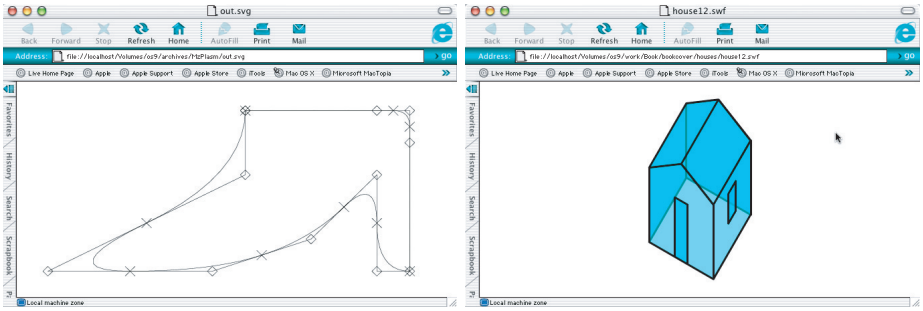

Figure 9.14 Rendering in a web browser of vector graphics exported by `PLaSM`: (a) SVG graphics (b) Flash graphics

9.3.2 Open Inventor camera model

The conceptual model used by Open Inventor [WO94] to specify the view is no different from the camera model adopted by ANSI Core and ISO PHIGS standards.

A camera node may be inserted anywhere in an Open Inventor scene graph. Such a node generates a picture of every object situated after it in the graph. The camera orientation is affected by the current geometric transformation. A node `SoCamera` is provided at this purpose, with attributes

1. `viewportMapping`, associated with the type of treatment to apply in non-isomorphic camera-viewport mapping;
2. `position`, which is the location of viewport in local coordinates. It is affected by current geometric transformation;
3. `orientation`, of the camera viewing direction. Together with the current geometric transformation, this specifies the orientation of the camera in world coordinates;
4. `aspectRatio`, i.e. ratio of the camera width to height;
5. `neardistance`, `farDistance`, `focalDistance`, specify in VRC the distance of camera viewpoint from front and back clipping planes as well as from the point of focus, i.e. from the view plane.

Two subclasses `SoPerspectiveCamera` and `SoOrthographicCamera` are derived from the `SoCamera` class. A new field `heightAngle` is added to `SoPerspectiveCamera` in order to specify the vertical angle in radians of the camera view volume, i.e. of the truncated pyramid volume specified by the camera. The horizontal angle of the view volume is determined by `heightAngle` and by the camera's `aspectRatio`.

The new field `height` of `SoOrthographicCamera` derived class specifies the height of the camera's parallelepiped view volume for parallel projections. A switch node (of kind blinker, e.g.) may be used to choose between different predefined cameras of a given scene.

9.3.3 Java 3D viewing model

Java 3D, the vendor-neutral 3D API based on Java platform, gives full support to the creation of virtual worlds and to multiple interaction with them by using a plenty of gadgetry. It hence must support the interaction of virtual and physical realities,

needing a much more complex viewing model that the standard camera's one.

Our main sources for the material in this section were [SN99, SRD00] by the chief architect of the Java 3D API and others.

In the Java 3D approach, a *VirtualUniverse* holds everything within one or more *Locales*. A Locale positions in a universe one or more *BranchGroups*, where each BranchGroup holds a *scene graph*. Scene graphs in Java 3D are typically divided into two types of branch graphs, called *Content branch* and *View branch*; the former contain scene modeling, including shapes, lights, and other content; the latter contain viewing information.

The *ViewPlatform* is a leaf node in a view branch of the scene graph which defines a viewpoint within the scene, by giving a frame of reference for the user's position and orientation in the virtual world. There can be many *ViewPlatforms* in a scene graph. Each such platform can be transformed by a *TransformGroup* parent node. User interface and animation features may modify such nodes to move the platforms under application control, like "magic carpets" [SN99] flying on the scene.

Many additional classes control how that scene is rendered, using either a *perspective* or a *parallel* projection. Support for *room-mounted* and *head-mounted displays* is provided, as well as for user's *head tracking*.

Virtual vs physical worlds Shapes, branch groups, locales, and the virtual universe define the *virtual world*. A user co-exists in both this virtual world and in the *physical world*. In particular, s/he has a position and orientation in both worlds. The Java 3D view model handles *co-existence mapping* between virtual and physical worlds. A chain of relationships controls several mappings. In particular they map: the eye locations relative to the user's head; the head location relative to a head tracker; the head tracker relative to the tracker base; the tracker base relative to display (image plate), and so on.

A so-called *view policy* selects one of two constraint systems, associated with either *room-mounted displays*, whose locations are fixed, like CRTs, video projectors, multi-screen walls and portals, or to *head-mounted displays* (HMDs), whose locations change as the user moves.

When using room-mounted displays and head tracking, the constraint system uses the eye location relative to the image plate to compute a view volume (*view frustum*), where the eyepoint locations are computed automatically. To map from eye to image plate, the constraint system uses a chain of coordinate system mappings, linking *Eyes* to *Head* to *Head tracker* to *Tracker base* to *Image plate*. In particular, the constraint system uses the left and right eye locations relative to the left and right image plates to compute two view volumes.

Physical to virtual mappings are needed to allow the user to interact with the virtual scene. Recall that the user co-exists in the virtual and physical worlds. For this purpose consider that the user has both a physical and a virtual position and orientation. We have seen that room- and head-mounted display view policies handle mapping from the user's physical body to a tracker base and image plates. To map from this physical world to the virtual world, it is necessary to add to the constraint chain: a tracker base to coexistence mapping; a coexistence to view platform mapping; a view platform to locale mapping, and finally a locale to virtual universe mapping.

Viewing model Summing up, the Java 3D viewing model is composed of: a *view policy* to choose a room- or head-mounted constraint system; a set of *physical body*, *physical environment*, and *screen configuration* parameters; a set of policies to guide the chosen constraint system, including the *view attach policy*.

The *view attach policy* establishes how the view platform origin is placed relative to the user (i.e., how it is attached to the user's view). Three such policies are possible:

1. *Nominal head* policy places the view platform origin at the user's head. It is convenient for arrangement of content around the user's head for a heads-up display. It is very similar to "older" view models.
2. *Nominal feet* policy places the view platform origin at the user's feet, at the ground plane. It is convenient for walk-throughs, where the user's feet should touch the virtual ground.
3. *Nominal screen* policy places the view platform origin at the screen center. It enables the user to view objects from an optimal viewpoint.

Implementation The Java 3D viewing model is implemented through several classes. A `VirtualUniverse` defines the universe coordinate system. A `Locale` places a scene graph branch within that universe. A `ViewPlatform` (and a `Transform3D` above it) define a viewpoint within that locale. It defines a frame of reference for the user's position and orientation in the virtual world. A `View` is the virtual user standing on a `ViewPlatform`. There can be many views on the same view platform. A `PhysicalBody` describes the user's dimensions for use by a `View`. There is always one `PhysicalBody` for a `View`. A `PhysicalEnvironment` describes the user's environment for use by a `View`. There is always one `PhysicalEnvironment` for a `View`. A `Canvas3D` selects a screen area on which to draw a `View`. Every View has one or more Canvas3Ds. A `Screen3D` describes the physical display device (image plate) drawn on by a `Canvas3D`.

In conclusion: the 3D transformation pipeline in Java 3D is quite complex because it has to map a virtual world to several physical worlds, but it is quite easy to use because each kind of object has reasonable defaults. In the virtual world the `ViewPlatform` controls the user's virtual position and orientation, whereas a `View` sets a view policy. In the physical world a `PhysicalBody` describes the user, whereas the `PhysicalEnvironment` describes the user's environment. Finally, a `Canvas3D` selects a region to draw into, whereas a `Screen3D` describes the screen device.

9.4 Examples

9.4.1 PHIGS pipeline display

In this section we develop some `PLaSM` functions which help to graphically display our implementation of the 3D pipeline for the central projection case. As already shown in Example 9.2.1, some functions without parameters are used to specify the view model.

The tensor $\boldsymbol{VO}$, $\boldsymbol{VM}_{per}$ and $\boldsymbol{P}$ of view orientation, view mapping and perspective transformations are implemented by `PLaSM` tensors denoted as `viewOrientation`, `viewMapping` and `perspTransf`, in Sections 9.2.4, 9.2.5 and 9.2.6, respectively. Notice that such tensors implicitly depend on the view model parameters, i.e. they are implicit functions of such arguments.

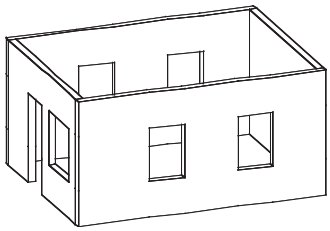

Figure 9.15 The model of the scene we have projected.

In particular, we generate here a geometric model of *view volume* starting from the parameters in view model. A geometric model of the *scene*, the *view volume* and a simplified model of the *view reference* system will be shown in world coordinates and then mapped by the component tensors of 3D pipeline in each intermediary reference frame.

Geometric model of scene A sufficiently realistic 3D house model is given here to illustrate the various steps of 3D pipeline. Such a `house` model is quite simplified, and in particular is open at the top. The source code generating the model is given in Script 9.4.1; some projections of it are shown in Figure 9.15.

Script 9.4.1

```
DEF mesh = INSL:* ∼ AA:QUOTE;
DEF house = T:2:-0.2:(walls - windows - door)
WHERE
   walls = STRUCT:<xWalls, yWalls>,
   xWalls = mesh:<<7>,<0.2,-5,0.2>,<3.5>>,
   yWalls = mesh:<<0.2,-6.6,0.2>,<-0.2,5>,<3.5>>,
   windows = STRUCT:
      < mesh:<<-1.5,1,-1.5,1>,<0.2,-5,0.2>,<-1,1.5>>,
      mesh:<<-6.8,0.2>,<-3,1.5>,<-1,1.5>> >,
   door = mesh:<<-6.8,0.2>,<-1.5,1>,<2.5>>
END;
```

Computation of View Volume in WC

In this section we show the computation of the view volume in WC3 starting from a given view model. The simplest method to generate such a volume probably consists in building the very simple pyramidal volume in NPC before of perspective projection, and then in getting its WC3 counterimage by applying the inverse 3D pipeline to it.

Canonical View Volume in NPC The canonical volume before perspective transformation is a truncated pyramid with side faces $z = x, z = -x, z = y, z = -y$, back face $z = -1$, and front face $z = z_{min}$. The view plane is defined by the equation $z = z_{proj}$.

Therefore, in `PLaSM` the canonical volume in normalized projection coordinates may

be generated by a MKPOL function which defines a polyhedral complex with 2 convex cells, each one given as convex combination of 8 vertices. In total, we need to explicitly give 12 vertices, situated by groups of 4 on the planes $z = z_{min}, z = z_{proj}$ and $z = -1$, as described by Script 9.4.2.

Script 9.4.2

```
DEF NPCvolume = MKPOL:< verts, cells, pols>
WHERE
   verts = < <z_min, z_min, z_min>, <z_min, -:z_min, z_min>,
      <-:z_min, z_min, z_min>, <-:z_min, -:z_min, z_min>,
      <z_proj, z_proj, z_proj>, <z_proj, -:z_proj, z_proj>,
      <-:z_proj, z_proj, z_proj>, <-:z_proj, -:z_proj, z_proj>,
      <1, 1, -1>, <1, -1, -1>, <-1, 1, -1>, <-1, -1, -1> >,
   cells = < <8, 7, 6, 5, 4, 3, 2, 1>, <12,11,10, 9, 8, 7, 6, 5> >,
   pols  = < <1, 2> >;

DEF z_min  = -:(vrp_z + front) / (vrp_z + back);
DEF z_proj = -:vrp_z / (vrp_z + back);
```

View Volume in VRC In order to generate the geometric model of view volume in view reference coordinates, called VRCvolume in Script 9.4.3, it is sufficient to apply the inverse view mapping to the NPCvolume given in Script 9.4.2.

We also give as a polyhedral complex of dimension $(1, 3)$, a triplet of orthogonal segments of unit length, called ReferenceFrame, to be used as an image of the VRC system in the set of pictures dedicated to the discussion of the 3D perspective pipeline.

View Volume in WC Finally, the view volume in world coordinates, called WCvolume, is obtained from the view model by applying the inverse view orientation mapping to the polyhedral complex VRCvolume given in Script 9.4.3. The scene used to produce several pictures in this example is defined as a structure that contains the WCvolume, the house model and the uvnSystem_in_WC.

9.4.2 VRML camera implementation

In this section we implement some simple operators to automatically generate viewpoints from the directions of the reference axes when producing a VRML output for PLaSM-generated models. For this purpose we will define, in Script 9.4.8, two functions called AxialCameras and CenteredCameras, respectively. The AxialCameras operator, when applied to some polyhedral complex, will insert in the output VRM hierarchical graph, three viewpoint nodes looking at the origin from the point on the x, y and z axis, respectively. The three viewpoints are labeled with strings 'x view', 'y view' and 'z view', and are searchable from the VRML interface of common web browsers. The CenteredCameras operator does something similar, but also it centers the viewpoints with respect to the orthogonal extent of the polyhedral scene or model it is applied to, so centering the produced views in the browsing

Script 9.4.3

```
DEF VRCvolume = (INV_T_per ∼ INV_SH_per ∼ INV_S_per): NPCvolume;
DEF INV_T_per = T:<1,2,3>:(prp);
DEF INV_S_per = S:<1,2,3>:<sx,sy,sz>
WHERE
   sx = ((umax - umin)*(vrp_z + back))/(2 * vrp_z),
   sy = ((vmax - vmin)*(vrp_z + back))/(2 * vrp_z),
   sz = -:(vrp_z + back)
END;
DEF INV_SH_per = MAT:
   <<1,0,0,0 >,
   <0,1,0,-:dopx / dopz>,
   <0,0,1,-:dopy / dopz>,
   <0,0,0,1 >>
WHERE
   dopx = (umin + umax)/2 - s1:prp,
   dopy = (vmin + vmax)/2 - s2:prp,
   dopz = 0 - s3:prp
END;
DEF uvnSystem = MKPOL:<
   < <0,0,0>,<1,0,0>,<0,1,0>,<0,0,1> >,
   < <1,2>,<1,3>,<1,4> >,
   < <1,2,3> > >;
```

viewport, as shown by Figures 9.18a, 9.18b and 9.18c.

VRML viewpoint implementation The PLaSM language and its underlying representation were recently extended by adding properties to nodes of the Hierarchical Polyhedral Complex (HPC) data structure. In such a way, graphics concepts such as appearance, lights, and viewpoints, that do not strictly depend on the geometry, can be inserted in the hierarchy in a simple and non-invasive manner. Furthermore, such properties are consistently retained after the evaluation of every PLaSM operators, including e.g. mapping, skeleton extraction, product and so on, and without any change in the implementation of the predefined language operators.

A camera is defined in PLaSM by making use of the semantics of the VRML viewpoint node. A viewpoint applies to the scene subgraph rooted on it, and is affected by the current value of transformation matrix. In PLaSM, a camera is joined to a polyhedral complex by an expression of this kind:

```
CAMERA:< pol_complex, camera >

camera ≡
< position, orientation, fieldOfView, focalDistance, description >
```

where position and orientation are a triplet and a quadruple of numeric expressions, respectively, according to the VRML semantics. We notice that a VRML orientation has the coordinates of a vector parallel to the orientation axis in the first three components, and an angle (in radiants) in the fourth component.

Script 9.4.4

```
DEF WCvolume = invViewOrientation:VRCvolume;
DEF invViewOrientation (volume::IsPol) = (invTransl ∼ invRot): volume
WHERE
   invTransl = T:<1,2,3>:vrp,
   invRot = MAT:< <1,0,0,0>, AL:<0,Ru>, AL:<0,Rv>, AL:<0,Rn> >,
   Ru = UnitVect:(vuv VectProd Rn),
   Rv = Rn VectProd Ru,
   Rn = UnitVect:vpn
END;

DEF uvnSystem_in_WC = invViewOrientation: uvnSystem;
DEF WCscene = STRUCT:< WCvolume, house, uvnSystem_in_WC >;
```

Toolbox First a small toolboox of auxiliary functions is given. It contains a non-raised version `IsGT` of the `GT` predefined operator, the `greater` selector of two argument numbers, which returns the greater of them. The `MK` function, used to transform a sequence of coordinates into a 0-dimensional polyhedron, is given in Script 3.3.15.

Script 9.4.5 (Toolbox)

```
DEF IsGT (a,b::IsReal) = GT:a:b;
DEF bigger  (a,b::IsReal) = IF:<IsGT,s2,s1>:<a,b>;
```

Axial Camera Actually, several VRML browsers do not implement the full viewpoint semantics. In particular they implement the viewpoint `position`, and the angle value in the `orientation` field, as well the `fieldOfView` given in radians, but the result of using an `orientation` axis different from `0,0,1` is unpredictable, whereas the `focalDistance` field is unused.

Such browsers' implementation limits underlie our design choices of Script 9.4.6, where the `MyCamera` operator applies to a $(0,3)$-dimensional polyhedron, which coincides with the origin of 3D space, a camera with a variable `prp` and a variable description `string`. Any `axialCamera:`i expression, with $i \in \{1,2,3\}$, will return a suitably oriented $(0,3)$-dimensional polyhedron, with attached camera in its local coordinate space, which may be rooted to the scene to display.

Let us remember that each binary `PLaSM` operator can be applied also in infix form, often increasing the code readability. In Figure 9.16 we show the display from the viewpoints in the VRML file generated by the `STRUCT` expression above, where `MKframe` is the 3D object described in Script 6.5.3. The last expression, where the `AxialCameras` operator given in Script 9.4.8 is used, produces exactly the same output.

Centered Camera A useful camera operator, called `centeredCamera:`, is given is Script 9.4.7. The expression `centeredCamera:i:obj` puts a centered viewpoint on the x_i direction, $i \in \{1,2,3\}$, in the polyhedral scene `obj`, at a distance suitably chosen, in such a way that the whole scene is gracefully accommodated into the VRML browser viewport, when the VRML file exported by `PLaSM` is loaded in the memory.

Script 9.4.6 (Axial Camera)

```
DEF prp = < 0,0,5 >;
DEF MyCamera (prp::Isseq)(string::IsString) = MK:< 0,0,0 >
   ASSIGNCAMERA < prp, < 0,0,1,0 >, <PI/4>, <>, string >;

DEF axialCamera  (i::IsIntPos) =
   IF:< C:EQ:3, K:(MyCamera:prp:'z view'), testxy >:i
WHERE
   testxy = IF:< C:EQ:1,
      K:((R:<2,3>:(PI/2) ~ R:<1,3>:(PI/2)):(MyCamera:prp:'x view')) ,
      K:((R:<1,2>:PI ~ R:<2,3>:(PI/2)):(MyCamera:prp:'y view')) >,
END;

STRUCT:< axialCamera:1, axialCamera:2, axialCamera:3, Mkframe >;
AxialCameras: MKframe;
```

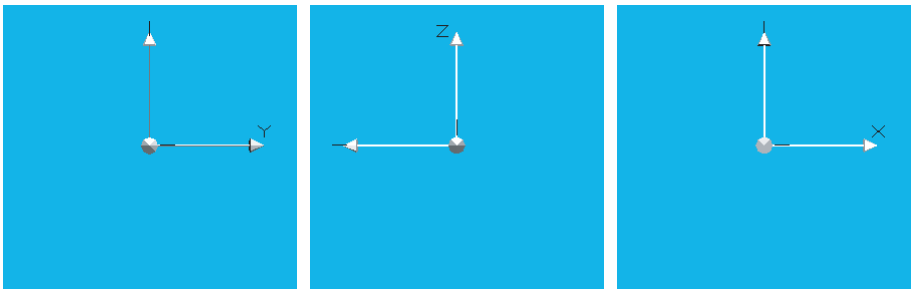

Figure 9.16 Scene display from viewpoints generated on the x, y and z axes, respectively

Set of cameras Three axial cameras or three centered cameras on the directions of the reference axes are generated by the functions `AxialCameras` and `CenteredCameras` given in Script 9.4.8. An example of use of the last operator in displaying a quite complex 3D scene is shown in Figure 9.18. The inverse ordering `<3,2,1>` of axis indices is used in order to get the 'z view' as the first viewpoint on opening the VRML browser, as usual.

Script 9.4.7 (Centered Camera)

```
DEF centeredCamera (i::IsIntPos) (obj::IsPol) =
   (T:i:(trParam - S3:PRP) ~ T:<1,2,3>:objCenter):(axialCamera:i)
WHERE
   objCenter = MED:<1,2,3>:obj,
   trParam = 1.25 * bigger:(SIZE:(pair:i):obj) + (SIZE:i:obj)/2,
   pair = IF:< C:EQ:3, K:<1,2>, IF:< C:EQ:1, K:<2,3>, K:<1,3> > >
END;
```

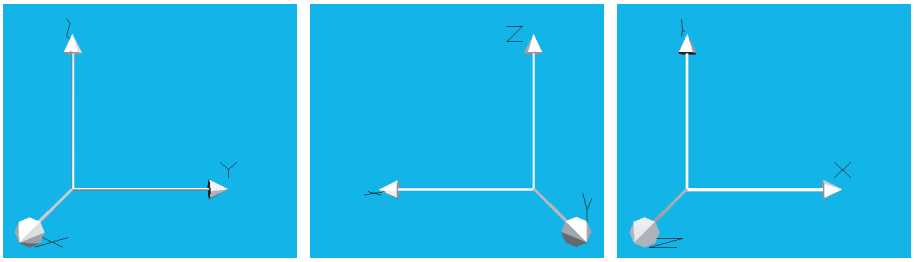

Figure 9.17 Scene display from viewpoints generated by `centeredCameras` operator

Script 9.4.8 (Set of cameras)

```
DEF AxialCameras  (obj::IsPol) =
   STRUCT: (obj AL AA:AxialCamera:<3,2,1>);

DEF CenteredCameras  (obj::IsPol) =
   STRUCT: (obj AL (CONS ∼ AA:centeredCamera):<3,2,1>:obj);
```

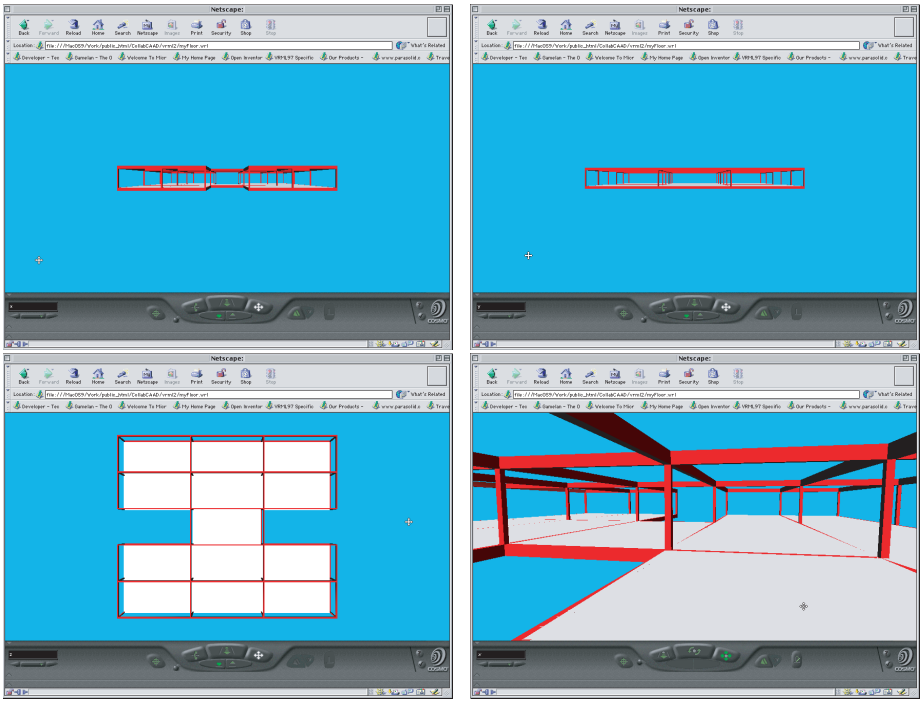

Figure 9.18 Different viewpoints in PLaSM-generated VRML model of beams and pillars in a building fabric