10

Viewing and rendering

This chapter is aimed at discussing the rendering process of a 3D scene on raster devices like display monitors and ink-jet printers. In the first part of the chapter we discuss how to choose the view models (i.e. suitable sets of parameters) in order to generate either realistic images or technical drawings of the modeled scene or object. For this purpose a detailed taxonomy of different types of projections is discussed, and several examples are given. Then the attention is shifted to the rendering process of realistic images, by discussing some main approaches to the hidden-surface removal (HSR) problem, i.e. to the removal of hidden parts of the scene. A short presentation is then given of lighting, shading and color models. Such techniques concern the computer treatment of light behavior of surfaces, and their rendering to produce realistic images. Some examples of VRML lighting, coloring and texturing PLaSM-generated models finally are discussed.

10.1 View-model

As we already know from the previous chapter, every 2D picture of a 3D scene is always obtained by *projection* from some *projection center* to some suitable *projection plane*, also called the *viewplane*.

1. The projection is said to be *perspective* when the projection center is *proper*.
2. The projection is said to be *parallel* when the projection center is *improper*, i.e. is set at infinity.

In the first case the bundle of lines that projects the scene points is a cone; in the second case it is a cylinder. In other words, the lines projecting the scene from a center at infinity are parallel.

Therefore, the first criterion used to distinguish between different types of projection is the kind, either proper or improper, of projection center. The projections of a 3D scene are then further classified depending on the position of center and on the attitude of the viewplane. The taxonomy of projections discussed in the following sections is summarized in Table 10.1.

Geometric Programming for Computer-Aided Design. Alberto Paoluzzi, Valerio Pascucci, Michele Vicentino, Claudio Baldazzi and Simone Portuesi
 ISBN: 0-471-89942-9

Table 10.1 Taxonomy of projections

Perspective	Central (1-point)	
	Accidental (2-point)	
	Oblique (3-point)	
Parallel	Orthographic	Simple
		Multiple
	Axonometric	Orthogonal
		Oblique

Perspective machines The perspective as a projection from a point, already known to the Greek painters of theatrical backdrops, was rediscovered by the artists of the Italian Renaissance. In particular, Brunelleschi and Alberti were the main theorists of the new projection techniques. Some pictures of Dürer's drawing machines, which reproduce the geometric machinery of projection, are shown in Figure 10.1. They were already known to Brunelleschi and Alberti. The books by Panofsky [Pan91, Pan71] are the authoritative source for the history of perspective as well as for most later explanations of those machines.

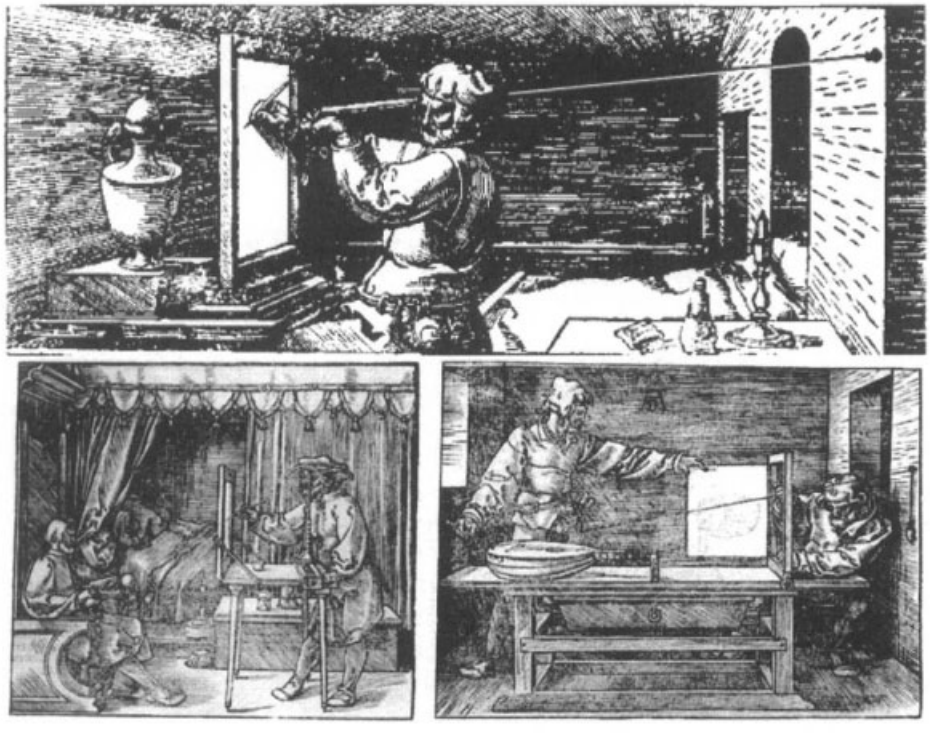


Figure 10.1 Dürer's drawing machines

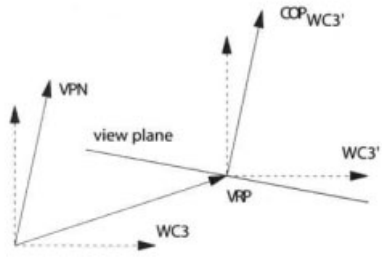


Figure 10.2 $\text{COP}_{\text{WC3}'}$ definition in a reference frame parallel to WC3, and with origin in VRP

10.1.1 View parameters

We are actually interested in computer-generated projections [CP78, Pao78, FvD82, FvDFH90] of some computer model of the scene. This section is therefore aimed at specifying how to produce some well-defined types of projection.

First we recall that the 3D scene is defined in world coordinates (WC3). The scene parts which are possibly defined in local modeling coordinate systems must be assembled in such a unique reference system before projecting.

The 2D picture of the scene is conversely generated with reference to a view reference coordinate system (VRC), called also *uvn* system, with two axes (uv) on the viewplane and the third axis (n) passing for the viewpoint.

We recall also, from Chapter 9, that such a VRC system is defined on the basis of few vector parameters given in WC3. As a matter of fact, it has:

1. its origin in *view reference point* (VRP);
2. n axis parallel to *viewplane normal* (VPN);
3. v axis (vertical, upwards) on the viewplane, and oriented as the projection of *view-up vector* (VUV);
4. u axis (horizontal, directed from left to right) normal to both v and n.

In Chapter 9 we have shown that, according to the PHIGS graphics standard, the *center of projection* (COP), for central projections, and the *direction of projection* (DOP), for parallel projections, may be derived from the *projection reference point* (PRP) given in VRC. In the present chapter, for this purpose of giving an easy way to define correct perspectives, we use a slightly different approach:

1. Define VRP in WC3, denoted as VRP_{WC3}, and use this point as the origin of a new system WC3' parallel to WC3.
2. Define $\text{COP}_{\text{WC3}'}$ in this parallel system, so that

$$\text{COP}_{\text{WC3}} = \text{VRP}_{\text{WC3}} + \text{COP}_{\text{WC3}'}.$$

3. Assume $\text{VPN}_{\text{WC3}} = \text{COP}_{\text{WC3}'}$.

This implies that the viewplane is orthogonal to the axis for VRP_{WC3} and COP_{WC3}, as shown in Figure 10.2. This also implies, as shown by several examples in the remainder of this chapter, that *central*, *accidental* or *oblique* perspectives — or 1-, 2- and 3-point perspectives, according to computer graphics literature [CP78, FvD82, FvDFH90]— are produced by $\text{COP}_{\text{WC3}'}$ points with either 1, 2 or 3 non-zero coordinates, respectively.

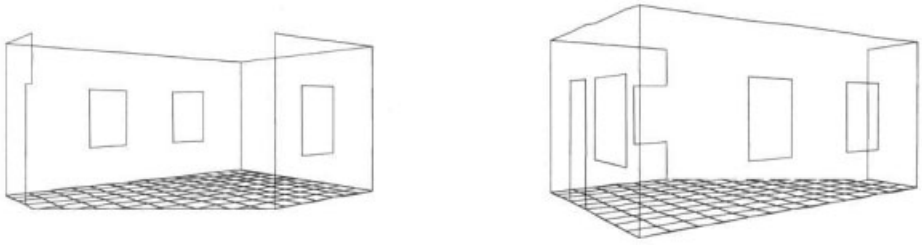


Figure 10.3 (a) Clipping at front plane (b) Clipping at back plane

Such an approach is clearly equivalent to the standard one, where the view orientation transformation is defined by giving VRP and VPN (and VUV), and where COP equates to the PRP in VRC. A set of "standard" PHIGS view models is given in Example 10.2.1, and the corresponding pictures are shown in Figure 8.11.

10.1.2 View volume

As we know from the previous chapter, the view volume is the bounded portion of WC3 which contains the subset of scene data which are seen in a specified projection. In particular, the subset of scene data which is outside the view volume must be *clipped* by the graphics system, whereas the subset of inside data must be *projected* and *rendered* on the output device.

Perspective projection The view volume of a perspective projection is a truncated *pyramid* with the apex in COP, four side planes passing for COP and for the window edges in the viewplane, and with two sides parallel to the viewplane and contained in planes with VRC equations $n = f$ and $n = b$. Such parallel planes are the *front* and *back* planes, sometimes called *hither* and *yon* planes, respectively.

Parallel projection The view volume is a *parallelepiped*, not necessarily straight, with four side edges parallel to DOP, four side faces for the window edges and front and back faces with VRC equations $n = f$ and $n = b$.

The view volume performs two different functions related to projection and clipping, respectively.

1. The shape of view volume actually specifies the *type of projection*, because the *view mapping* transformation must map this volume, as well as the inside content, onto the 3D viewport in NPC system.
2. The inside content of the view volume is projected and rendered to produce the actual picture of the scene. All the scene data that lie outside such a volume are instead clipped, i.e. not rendered, by the graphics system.

In Figure 10.3 we show the images generated by activating the clipping to the front and back planes, respectively, when the view volume intersects the scene model.

10.2 Taxonomy of projections

In this section we discuss in depth the classification of projections established in the military and engineering schools since the early times of descriptive geometry in the Napoleonic age.[1] In particular, we aim to discuss how to define the set of vector parameters that we collectively called *view model*, in order to obtain some well-specified kinds of projection.

10.2.1 Perspective

As we already said, perspective projections are classified depending on the attitude of viewplane with respect to the reference frame. In particular, the viewplane can be either parallel to some coordinate plane, or parallel to some coordinate axes, or in a general position. Correspondingly, the generated projection is said to be either 1- or 2- or 3-point perspective. Some examples of such projections from the art history are given in Figure 10.4. The corresponding perspectives of the standard cube built on the vectors of the Cartesian basis are shown in Figure 10.5.

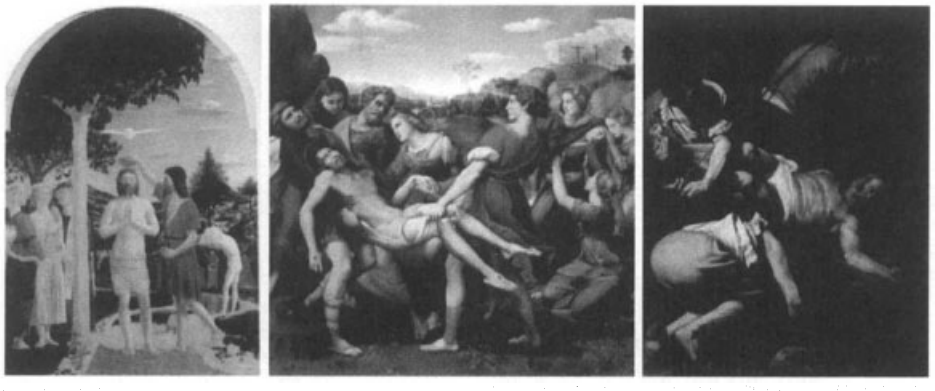


Figure 10.4 (a) Piero della Francesca, *The Baptism of Christ*, 1459, National Gallery, London (b) Raffaello, *The Carrying of Christ (Pala Baglioni)*, 1507, Borghese Gallery, Rome (c) Caravaggio, *The Crucifixion of Saint Peter*, 1600, Cerasi Chapel, Santa Maria del Popolo, Rome

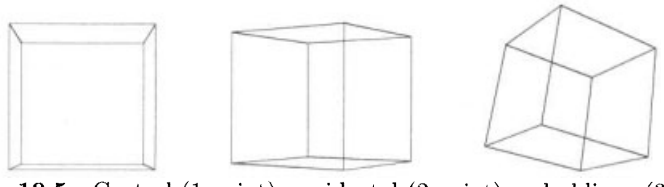


Figure 10.5 Central (1-point), accidental (2-point) and oblique (3-point) perspective

[1] Gaspard Monge (1746–1818) was a teacher at the military school of Mézières.

One-point perspective

A *central* perspective is obtained when the viewplane is parallel to a coordinate plane. In such a projection only one of the improper points of coordinate axes has finite projection. In other words, the perspective has only one *accidental point*, also called a *vanishing point*, so it is often called *1-point perspective*.

Notice that in this projection the images of parallel straight lines which are also parallel to one of two coordinate axes remain parallel. The images of lines which are parallel to the coordinate axis perpendicular to the viewplane, conversely converge in the vanishing point. This effect is actually visible only when the scene to be projected is parallel to the reference frame.

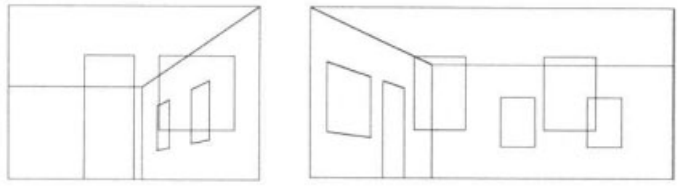


Figure 10.6 View models for one-point perspectives with center of projection set on the ground ($\text{VRP}_z = \text{COP}_z = 0$)

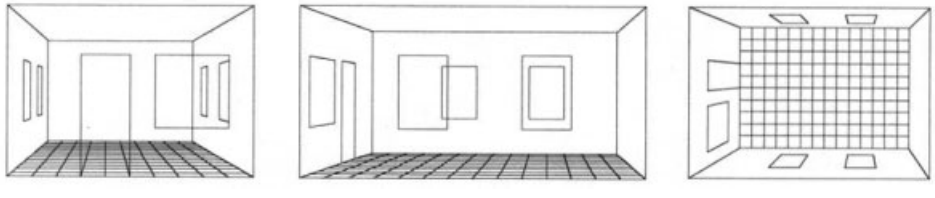


Figure 10.7 Central perspectives from the directions of coordinate axes

Two-point perspective

When the viewplane is parallel to a coordinate axis we obtain an *accidental perspective*. In such a projection two of the improper points of coordinate axes have finite perspective. In other words, the accidental perspective has two vanishing points for the coordinate axes. This projection is also called *2-point perspective*.

The parallelism of the bundle of coordinate axis parallel to view plane is conserved by the projection. The images of lines in the two bundles parallel to the other two axes, instead converge in two finite vanishing points.

Such two vanishing points define the *horizon line* on the view plane. This line contains the perspective of the improper points of every bundle of lines parallel to the ground plane $z = 0$. The horizon line is the perspective of the line at infinity of the bundle of planes parallel to the ground.

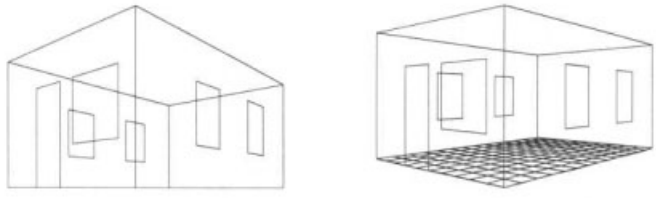


Figure 10.8 Accidental perspectives: (a) VRP on the ground (b) VRP above of the ground

Three-point perspective

The more realistic type of perspective picture is obtained by the 3-point perspective, where all improper points of coordinate axes have finite projection.

Such a projection is also called "with sloping cadre" or *oblique* perspective or *photographic* perspective, since it reproduces the behavior of the camera, that can be placed in any position and with any orientation with respect to the scene when taking a picture.

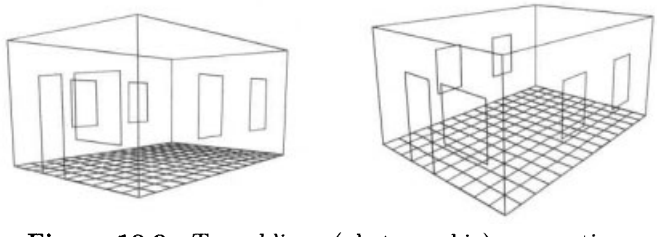


Figure 10.9 Two oblique (photographic) perspectives

Remarks In order to generate well-specified types of perspective it may be useful to take into account the following points:

1. If the viewplane is vertical, i.e. parallel to the z-axes, and $\text{VRP}_z = \text{COP}_z = 0$, then the *horizon line* coincides with the *ground line* (see Figure 10.6). The first one is the image of the line at infinity of the plane bundle parallel to $z = 0$; the second one is the intersection of the viewplane with $z = 0$.
2. To force the non-coincidence of horizon and ground lines, it is necessary to put the camera's target above the ground, i.e. to have $\text{VRP}_z \neq 0$. Usually VRP_z is set equal to the viewer's height.
3. It is very easy to specify the kind of perspective if COP is given in the WC3' system. In particular, the perspective is *central* if only one component of COP is not zero; it is *accidental* if two components are non-zero; it is *oblique* (photographic) if they are all non-zero.

Interior perspective In Figure 10.10 we show two perspective pictures from outside and inside our usual scene. In order to obtain an image of the interior of the scene without inducing some overturning of the visual cone relative to the scene part on the viewer's back, it is necessary to activate the scene clipping at the front plane, often coinciding with the viewplane, or very close to it.

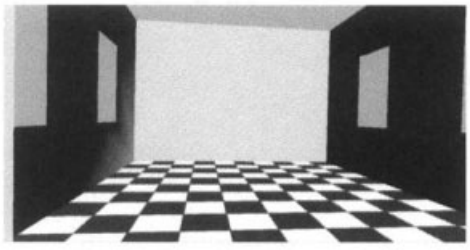


Figure 10.10 Perspective inside the scene, clipped to the front plane

10.2.2 Parallel

We remember that a projection is said to be *parallel* when the center of projection is improper. This implies that the projecting lines are a bundle of parallel lines. The parallel projections are classified as either orthographic or axonometric.

Orthographic

A parallel projection is said to be *orthographic* when (a) the projection plane is a coordinate plane, and (b) the direction of projection is parallel to the orthogonal coordinate axis.

Clearly, there are only three different orthographic projections. The projection from the direction of z-axis is called the *plan view* or planimetry. The projections from the directions of x-axis or y-axis are called either *front view* or *side view* depending on the orientation of the scene with respect to the reference frame.

In every case, the viewplane normal must be parallel to the direction of projection. The easiest choice is to set VPN = DOP.

In all sorts of parallel projections, the generated image does not depend on the choice of the VRP. In fact, because the parallelism of projecting lines, the projected image is the same on each element of a bundle of parallel planes. For the sake of simplicity we choose $\text{VRP} = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^T$.

Multiple orthographic

A multiple orthographic projection, where either two or three orthographic projections are simultaneously generated and assembled, is known as Monge's projection, after Gaspard Monge, one of the inventors of descriptive geometry at the beginning of XIX century. In his work, every 3D geometric element, i.e. points, curves and surfaces, is described by two or more corresponding orthographic projections.

The reader should notice, from the view model of Figure 10.11c, that the plan view of a Monge's projection has the image of y-axis vertical but oriented downwards. Consequently, the image of x-axis is horizontal but oriented towards left. In 2D graphics we are conversely used to see such axes upwards and from left to right, respectively. This effect is obtained simply by setting $\text{VUV} = \begin{pmatrix} 0 & -1 & 0 \end{pmatrix}^T$.

In most graphical user interface of CAD systems, such multiple orthographic projections are used together with a further view of the scene or object at hand. Usually some orthogonal axonometric projection, described in the following sections, is used for this purpose, as shown in Figure 10.11. That picture, as well most of the images of this

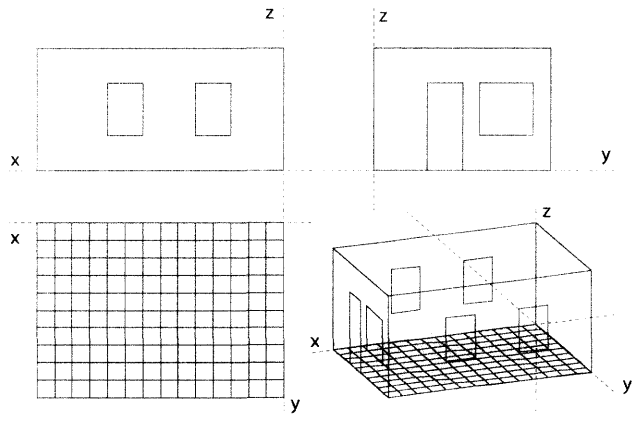
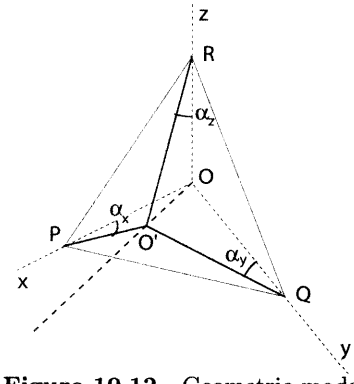


Figure 10.11 Monge's orthographic projections, and axonometric projection

section, was produced by using the solid modeler *Minerva*, developed at the University of Rome "La Sapienza" by the CAD Group in second part of 10980s [PRS89, PM89]. *Minerva* was, to the knowledge of the authors, the first solid modeler implemented on a personal computer, since it worked on the IBM PC in '86 and on the Apple Macintosh in '87.

Axonometric

All the parallel projections that use projection planes which are not normal to a coordinate axis are called *axonometric projections.* They are classified into two main classes, depending on the existence or not of a parallelism between the direction of projection and the viewplane normal. The first ones are called *orthogonal axonometric projections*; the second ones are called *oblique axonometric projections.*



$$\text{DOP} = \text{VPN} = \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix}$$

$$\begin{aligned} n_x &= \cos(\frac{\pi}{2} - \alpha_x) = \sin\alpha_x \\ n_y &= \cos(\frac{\pi}{2} - \alpha_y) = \sin\alpha_y \\ n_z &= \cos(\frac{\pi}{2} - \alpha_z) = \sin\alpha_z \end{aligned}$$

Figure 10.12 Geometric model of the orthogonal axonometric projection

Orthogonal axonometric When the direction of projection is orthogonal to the viewplane the projection is called *orthogonal axonometric projection.* In this case the DOP and VPN vectors are parallel by definition, and often are set equal each other. In

the history of graphics techniques three standard orthogonal axonometric projections are well defined. They are called respectively:

1. isometric projection
2. dimetric projection
3. trimetric projection

depending on the relationship between the sizes of the projected images of the Cartesian basis of the WC3 system, as discussed in the following.

Geometry of orthogonal projections In Figure 10.12 we give a conceptual model of geometry of orthogonal axonometric projections, where the axis OO' is directed as the DOP vector, and where the projection plane is chosen as passing for points P, Q and R. Notice that O' is the projection of the origin, and that $O'P$, $O'Q$ and $O'R$ are the images of the reference system on the viewplane.

It is easy to verify that a unit vector n parallel to OO' has components $\begin{pmatrix} \cos\beta_x & \cos\beta_y & \cos\beta_z \end{pmatrix}^T$, where β_x, β_y and β_z are the angles between n and the Cartesian axes. We remember from analytic geometry that such numbers are called *director cosines* of the plane. If we consider the right-angled triangles $OO'P$, $OO'Q$ and $OO'R$ and the angles $\alpha_x = \frac{\pi}{2} - \beta_x$, $\alpha_y = \frac{\pi}{2} - \beta_x$ and $\alpha_z = \frac{\pi}{2} - \beta_x$, that the projection plane makes with the axes, then we can conclude that in any orthogonal axonometric projection:

$$\text{DOP} \equiv \text{VPN} \doteq n = \begin{pmatrix} \sin\alpha_x \\ \sin\alpha_y \\ \sin\alpha_z \end{pmatrix}$$

Isometric orthogonal projection This axonometric projection get its name from the fact that all the *images* of unit basis vectors have the same size (from *iso* = same. and *metric* = measure, size). The *image* of any pair of reference axes also includes the same angle of 120°.

We could actually distinguish between eight different orthogonal isometric projections, one for each octant where to choose the direction of projection. Anyway the standard isometric projection is that with $n_x = n_y = n_z > 0$. For this projection we have:

$$\alpha_x = \alpha_y = \alpha_z = 35°20',$$

but in this case it is certainly easier to choose $\text{DOP} = \text{VPN} = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}^T$.

The orthogonal isometric projection is largely used both in mechanical and in architectural drawings. It may sometimes be a little confusing, because the projection of the standard cube appears as a regular hexagon. An example of isometric orthogonal projection is given in Figure 10.13.

Dimetric projection The name *dimetric* is used because the projections of the unit basis vectors have two different sizes. From some handbooks of drawing techniques we

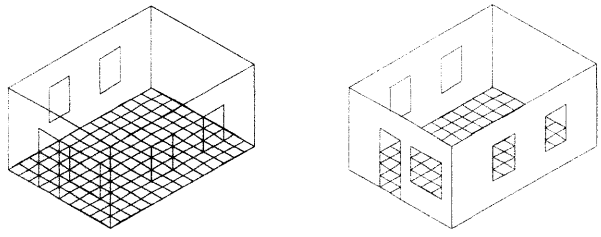


Figure 10.13 Isometric orthogonal projection

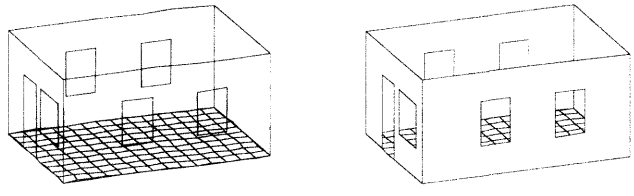


Figure 10.14 Dimetric orthogonal projection

get the values of the *director angles* of the projection plane, i.e. the angles between this plane and the coordinate axes, for the dimetric projection:

$$\alpha_x = 19°32', \quad \alpha_y = 61°50', \quad \alpha_z = 19°32'.$$

Actually there is an infinite number of different dimetric projections; the one given above should be more properly called the *standard dimetric projection.* An example is given in Figure 10.14.

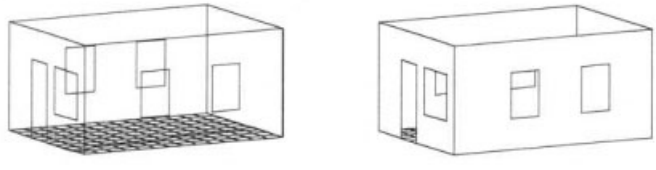


Figure 10.15 Trimetric orthogonal projection

Trimetric projection The trimetric projection is so called because the projections of the unit basis vectors have three different sizes. Clearly, there are infinite trimetric projections, each one obtained by choosing any DOP and any non-parallel VPN. The *standard trimetric projection* is obtained by the following values of director angles:

$$\alpha_x = 27°30', \quad \alpha_y = 60°30', \quad \alpha_z = 9°50'$$

The trimetric projection of our usual scene model is shown in Figure 10.15.

Oblique axonometric The axonometric projections where the DOP vector is not normal to the projection plane are called *oblique projections.* As an obvious consequence of this definition, DOP and VPN cannot be equal nor parallel. Some special types of oblique projection are often used. In particular:

1. *cavalier*[2] *projections* are oblique projections where the view plane is parallel to a coordinate plane;
2. *military cavalier projections* are isometric cavalier;
3. *cabinet projections* are dimetric cavalier.

The cavalier projections are largely used by European architects. They are easy to draw with the traditional draftsman tools, since these projections do not change the geometric figures that lie on planes parallel to a coordinate plane. The cabinet projection is also quite realistic. Last but not least, such projections may be directly used in a workshop or building yard since they represent exactly the plan or front views of the artifact.

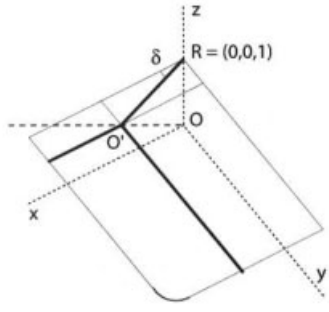


Figure 10.16 Geometric model of the cavalier projections

In every cavalier projection, the images of two Cartesian basis vectors have unit size and enclose a straight angle. This a consequence of the fact that a parallel projection on a plane of two parallel segments does not change either their sizes or their angle.

Hence, if we choose a projection plane at unit distance from the origin, then the number $|O'R|$, i.e. the size of the projection of the unit segment OR (see Figure 10.16). determines the projection type. In particular, an isometric cavalier is generated when $|O'R| = 1$, otherwise a dimetric cavalier is obtained. This is called a *cabinet projection* when $|O'R| = \frac{1}{2}$.

Also, it becomes very easy to specify the DOP vector. Looking at Figure 10.16. we may assume

$$\text{DOP} = \begin{pmatrix} |O'R| \cos\delta \\ |O'R| \sin\delta \\ 1 \end{pmatrix},$$

where δ is the angle between the projections of x and z axes.

Notice that there are ∞^2 different cavalier projections, when the orientation of the projection plane is fixed, since they depend on the angle δ and on the number $|O'R|$. Both military cavalier and cabinet projections are parametrized by the δ angle.

Military cavalier In this kind of projection the projection plane is parallel to a coordinate plane, usually the $z = 0$ plane.

---

[2] From the Italian mathematician Bonaventura Cavalieri (1599–1647), fellow of Galileo.

Here the DOP vector has an angle $\alpha_z = \frac{\pi}{4}$ to the z-axis, so that the projection of $\boldsymbol{e}_3$ basis vector results of unit size. The images of $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ have also unit size because of their parallel projection on a parallel plane. Al we already said, a military cavalier projection is isometric.

The more common military projections, called *standard* and shown in Figure 10.17, have angle δ of the image of x-axis to the vertical line equal to 30°, 45° and 60°, respectively.

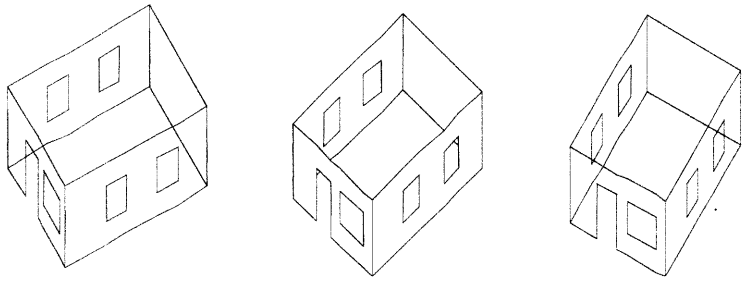


Figure 10.17 Left, centered and right standard military projections

Standard military projections are often used in architecture and interior decoration. In these applications an orientation of the picture different form the standard one may be needed. Such a requirement is easy to satisfy. The picture orientation of Figure 10.17 is given by VUV $= \boldsymbol{e}_3$; the orientations in Figure 10.18 and Figure 10.19 are generated by VUV $= -\boldsymbol{e}_2$ and VUV $= \boldsymbol{e}_2$, respectively.

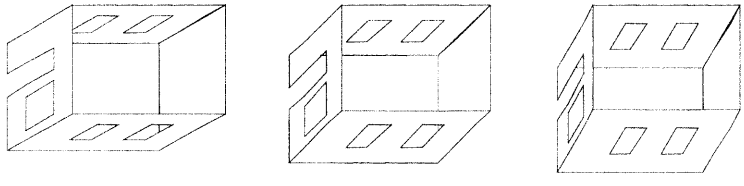


Figure 10.18 Standard military projections with VUV $= -\boldsymbol{e}_2$

Side and front cavalier Two very unusual and interesting military projections may be generated by choosing the DOP vector in a coordinate plane. The effect of such projections is a proper assembling of plan view and either front or side views, with no deformation or scaling. Such projections, shown in Figure 10.20, are sometimes used by architects, mainly in interior decoration drawings.

To understand the pictures generated by such projections we must look at the images of basis vectors. They appear as shown in Figure 10.20b. In particular, the image of $\boldsymbol{e}_3$ is parallel to the image of either $\boldsymbol{e}_1$ or $\boldsymbol{e}_2$, respectively, but has opposite orientation. The images of $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ are neither scaled nor change their angle, since the projection plane is parallel to their plane.

It is interesting to notice, by looking at Figure 10.20a, that it is possible to reconstruct all the measures of our scene model from such projections, assuming that each one of the small squares on the ground plane has unit side. The PLaSM coding of

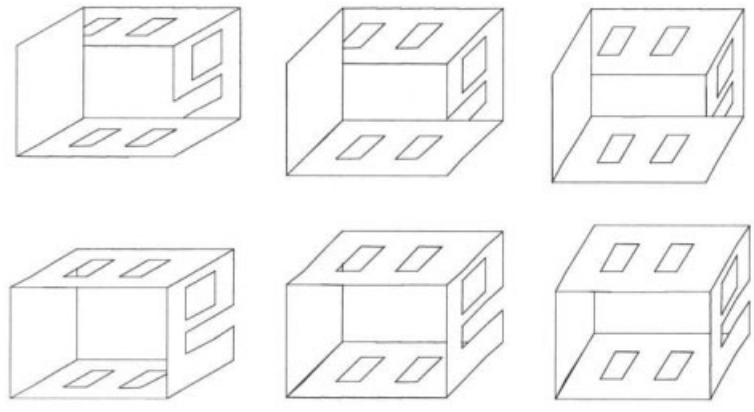


Figure 10.19 Standard military projections with VUV $= e_2$ and DOP in third and first octant, respectively

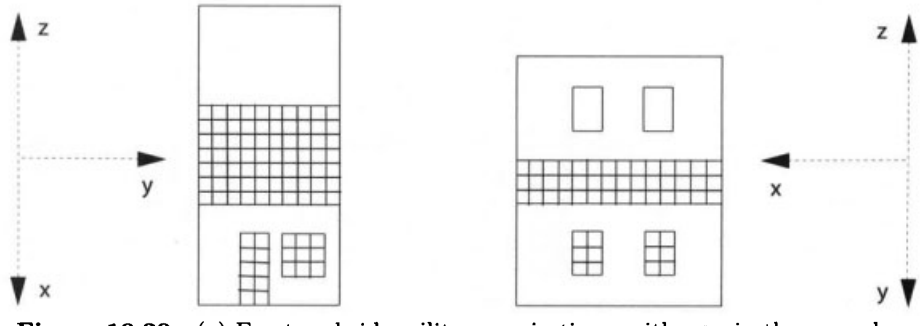


Figure 10.20 (a) Front and side military projections, with DOP in the xz and yz plane, respectively (b) Projected images of basis vectors

such a model is actually given in Section 10.8.

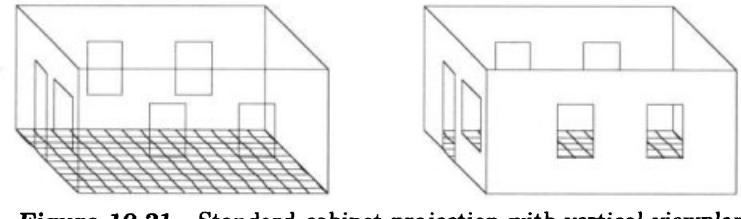


Figure 10.21 Standard cabinet projection with vertical viewplane

Standard cabinet As we know, in every cavalier projection the viewplane is parallel to a coordinate plane. Hence the images of the two basis vectors spanning this plane have unit size. In a cabinet projection, the DOP vector is chosen in such a way that the image of the basis vector parallel to VPN gets size $\frac{1}{2}$.

For each given attitude of the viewplane there is an infinite number of different cabinet projections. The so-called *standard cabinet* has VPN $= e_2$, size $\frac{1}{2}$ of the e_2 image and angle 45° from this one to the horizontal line. A standard cabinet projection

is shown in Figure 10.21.

Example 10.2.1 (Standard view models)
A detailed definition of all significant standard viewmodels is given in Script 10.2.1, according to the PHIGS specification, i.e., with `vrp`, `vpn`, `vuv` given in WC3, and `prp`, `window`, `front` and `back` given in VRC.

Notice that the `arc` function produces an argument conversion from degrees to radians. The `onepoint`, `twopoints` and `threepoints` definitions produce perspective view models, whereas all the other definitions clearly produce parallel view models.

Use examples of such models and the display of produced projections, exported as Flash files, are given in Figure 8.11 and Script 8.5.22, respectively.

Script 10.2.1 (Standard view models)

```
DEF ViewModel (vrp, vpn, vuv, prp, window::IsSeq; front, back::IsReal)
   = < vrp, vpn, vuv, prp, window, front, back >;

DEF arc (degrees, cents::IsReal) = (degrees + cents/60) * PI / 180;

DEF onepoint = ViewModel: <<0,5,4>, <1,0,0>, <0,0,1>, <0,0,35>,
   <-1,-1,1,1>, 0, -1 >;
DEF twopoints = ViewModel: <<0,0,4>,<3,2,0>,<0,0,1>,<0,0,35>,
   <-1,-1,1,1>, 0, -1 >;
DEF threepoints = ViewModel: <<0,0,4>,<3,2,1>,<0,0,1>,<0,0,35>,
   <-1,-1,1,1>, 0, -1 >;
DEF orthox = ViewModel: <<0,0,0>,<1,0,0>,<0,0,1>,
   <0,0,1>,<-1,-1,1,1>,1,-1 >;
DEF orthoy = ViewModel: <<0,0,0>,<0,1,0>,<0,0,1>,
   <0,0,1>,<-1,-1,1,1>,1,-1 >;
DEF orthoz = ViewModel: <<0,0,0>,<0,0,1>,<0,-1,0>,
   <0,0,1>,<-1,-1,1,1>,1,-1 >;
DEF isometric = ViewModel: <<0,0,0>, AA:(SIN arc):<<35,20>,<35,20>,
   <35,20>>, <0,0,1>, <0,0,25>,<-1,-1,1,1>,1,-1 >;
DEF dimetric = ViewModel: <<0,0,0>, AA:(SIN arc):<<19,32>,<61,50>,
   <19,32>>, <0,0,1>,<0,0,25>,<-1,-1,1,1>,1,-1 >;
DEF trimetric = ViewModel: <<0,0,0>, AA:(SIN arc):<<27,30>,<60,30>,
   <9,50>>, <0,0,1>, <0,0,25>,<-1,-1,1,1>,1,-1 >;
DEF centralCavalier = ViewModel: <<0,0,0>,<0,0,1>,<-1,-1,0><0,-1,1>,
   <-1,-1,1,1>,1,-1 >;
DEF leftCavalier = ViewModel: <<0,0,0>,<0,0,1>,<-1/2,SQRT:3/-2,0>,
   <0,-1,1>,<-1,-1,1,1>,1,-1 >;
DEF rightCavalier = ViewModel: <<0,0,0>,<0,0,1>,<SQRT:3/-2,-1/2,0>,
   <0,-1,1>,<-1,-1,1,1>,1,-1 >;
DEF cabinet = ViewModel: <<0,0,0>,<0,1,0>,<0,0,1>,
   <SQRT:2/-4,SQRT:2/4,1>,<-1,-1,1,1>,1,-1 >;
DEF xCavalier = ViewModel: <<0,0,0>,<0,0,1>,<0,-1,0>,
   <0,-1,1>,<-1,-1,1,1>,1,-1 >;
DEF yCavalier = ViewModel: <<0,0,0>,<0,0,1>,<-1,0,0>,
   <0,-1,1>,<-1,-1,1,1>,1,-1 >;
```

10.3 Hidden-surface removal

The photorealistic rendering of 3D scenes requires (a) the *hidden-surface removal* from the scene; (b) the *color shading* of visible surfaces depending on position, intensity, color, shape and orientation of lighting sources as well as on reflectance properties of surface materials; and (c) the *texture mapping* on 3D surfaces of a proper projective image of some 2D picture of material they are made. All such three points greatly enhance the realism of computer rendering of the modeled scene or object. Methods and algorithms for hidden-surface removal, illumination, color shading and texture mapping are the subject of this section and the subsequent sections of this chapter. Without loss of generality we suppose the geometry of the scene is described as a set of 3D polygons.

10.3.1 Introduction

The *hidden-surface removal* (HSR) is a major step in realistic graphics rendering of 3D scenes. It is usually assumed the the objects in the scene are *dull*, so that they cannot be traversed by light rays. As a consequence, not all parts of an object, as well as not all objects in the scene, are visible to the viewer. It is customary to distinguish between *direct* (or internal) and *indirect* (or external) *visibility*.

In recent years, when speaking of hidden-parts removal, we usually mean the removal of *hidden-surfaces*, more than the removal of *hidden lines*, i.e. of the portions of the boundary edges which are not seen by the viewer, depending on the greater practical importance of raster graphics with respect to vector graphics. Anyway, some algorithms used for removal of hidden-surfaces may be also used to remove their boundary edges.

Even the computation of *shades* generated by a given assignment of light sources may be reduced to the solution of a set of HSR problems.

Taxonomy of algorithms The algorithms for HSR are usually classified, following the early survey [SSS74] by Sutherland, Sproull and Schumacker, into two classes:[3]

1. *object space* (or *exact*) algorithms, where the problem is solved using real coordinates, usually in NPC;
2. *image space* (or *approximated*) algorithms, where the computation is done in integer coordinates, usually in DC3, while rasterizing the picture with a resolution dependent on the available quantity of video RAM.

The first approach was used when using vector devices like pen plotters, which today have disappeared from the market. The second kind of approach is the standard nowadays with raster devices, and is usually implemented in firmware on graphics cards.

It is easy to see that a set of n polygons may produce, in the worst case, a hidden-surface removed scene with $O(n^2)$ visible polygon parts. An optimal algorithm with quadratic complexity is given in [McK87].

---

[3] Foley *and others* [FvDFH90] refer to such two classes as *object-precision* and *image-precision* algorithms, respectively.

10.3.2 Pre-processing

Interactive visualization of large geometric data sets with high depth-complexity has long been a major problem within computer graphics. In particular, to efficiently solve the HSR problem always requires some pre-processing, including

- hierarchical graph culling;
- perspective transformation;
- view volume clipping;
- back-face culling;
- occlusion culling.

The word *culling* stands for something picked out from others, especially something rejected because of inferior quality. The root is from Latin *colligere*.

The aim of the above computations is to reduce the dimension of input, i.e. the number of processed polygons, while at the same time reducing the complexity of fundamental tests performed on each projected polygon.

Graph culling and view-volume clipping If the viewpoint is close or inside the scene, so that only a data subset is actually visible, then it becomes highly useful to discard the scene portion outside the view-volume. This clipping is mandatory when the viewer is positioned *inside* the scene data, in order to prevent the projection of polygons on the viewpoint's back.

It is often possible to distinguish between a gross and a detailed clipping to the view volume.

A gross clipping is performed when traversing the hierarchical scene graph (or structure network — see Section 8.2), and is usually called *hierarchical graph culling*. It is implemented as an *intersection* between the *containment boxes* of the current subgraph and the view volume. If such boxes have an empty intersection, then traversal of the subgraph stops. Otherwise, recursive traversal continues. In order to perform a graph culling the containment box of subgraph data must be stored and maintained in each graph node.

The view-volume clipping is executed only on the data which passed the graph culling previously described. Such detailed clipping is performed using either NPC or DC3 coordinates, depending on the architecture of the whole rendering pipeline.

It may be useful to notice that the performance of a 3D browser of textscvrml data is strongly improved when the "world" description is enriched with information regarding the bounding boxes of world elements. Also, depending on the graph culling, the animation of a "walk-through" in a complex architectural scene may be much faster than the presentation of the whole scene from an external viewpoint.

Perspective transformation To remove the hidden-surfaces of a scene requires discovering the polygons or their parts which are unseen from the viewpoint position. A *visibility test* on pairs of points must be often performed for this purpose. This may be defined as follows:

1. given two points $\boldsymbol{p}$ and $\boldsymbol{q}$, discover if they are aligned with the viewpoint $\boldsymbol{o}$;
2. given three aligned points $\boldsymbol{p}$, $\boldsymbol{q}$ and $\boldsymbol{o}$, determine if either $\boldsymbol{p}$ or $\boldsymbol{q}$ is closer to $\boldsymbol{o}$.

A simple geometric approach to visibility test requires computing if

$$(\boldsymbol{p} - \boldsymbol{o}) \times (\boldsymbol{q} - \boldsymbol{o}) = \boldsymbol{0}, \tag{10.1}$$

i.e. verifying that three 2×2 determinants are zero, and then testing if $\boldsymbol{p}$ is closer to $\boldsymbol{o}$ than $\boldsymbol{q}$:

$$|\boldsymbol{p} - \boldsymbol{o}| < |\boldsymbol{q} - \boldsymbol{o}| \tag{10.2}$$

i.e., in components:

$$(x_p - x_o)^2 + (y_p - y_o)^2 + (z_p - z_o)^2 < (x_q - x_o)^2 + (y_q - y_o)^2 + (z_q - z_o)^2.$$

It is very useful, from a computational viewpoint, to perform the visibility test using NPC coordinates where the *perspective transformation*, already discussed in Section 9.2.6, has been included in the 3D pipeline.

Using such coordinates, the visibility test is algebraically reduced to the following much simpler formulation, where the condition

$$x_p = x_q \quad \text{and} \quad y_p = y_q$$

is tested rather than equation (10.1), and the condition

$$z_p < z_q,$$

is tested rather than equation (10.2).

Clearly, only two numeric comparisons are executed for the first point, rather than the computation of three determinants, and just one comparison is executed for the second point rather than 6 products, 6 differences and four additions.

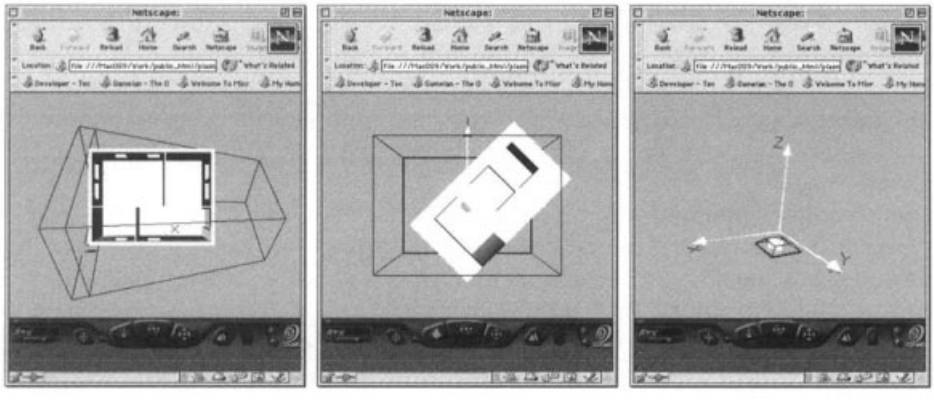


Figure 10.22 Viewing pipeline: (a) scene and view volume in world coordinates (WC3); (b) in view reference coordinates (VRC); (c) in normalized projection coordinates (NPC) before perspective transformation

Hence, when using NPC or DC3 coordinates, the z coordinate may correctly be called *perspective depth* of points and used to directly compare the relative position of two points respective to the viewer when they have the same x and y coordinates. Last but not least, this approach allows for a unified treatment of perspective and parallel projections when removing the hidden-surfaces of the scene. Some images from the viewing pipelines are given in Figure 10.22.

Back-face culling This operation removes the surfaces of solids which cannot be seen because they reflect or diffuse the light rays in directions that cannot reach the viewer's eye. Such polygons are called *back-faces*. Their external normal gives an angle greater than $\frac{\pi}{2}$ to the DOP or COP vectors.

Let us consider that the viewpoint is mapped, by the perspective transformation, to the point at infinity of z-axis. Assuming a right-handed NPC system, the back-faces are easily recognized depending on the negative sign of the z-component of their *external* normal. This is a further computational benefit of perspective transformation.

The HSR problem is hence easy to solve for *single* and *convex* objects in NPC or DC3 coordinates. In such case it is sufficient to cull away the polygon faces whose external normal $\boldsymbol{n}$ gives an angle greater than $\frac{\pi}{2}$ to the view direction, i.e. to the z-axes. It is easy to see that this condition is satisfied when $n_z \leq 0$.

Therefore, given a data set of complexity measured by the number n of uniformly distributed faces, it is possible to assert that an average number $\frac{n}{2}$ of polygons is removed by back-face culling. If the scene contains only one convex object, then the back-face culling completely solves the visibility problem. Some examples of this fact are given in Figure 10.23.

Example 10.3.1 (Back-face culling)
An example is given here of a PLaSM program generating a convex solid by intersection of three rotated instances of another convex solid. The generating function is called `Jewel` in Script 10.3.1. The three intersected instances are subject to a rotation of $\alpha = \frac{\pi}{4}$ about an axis of the reference frame. The `Jewel` operator is used twice, so generating the `convex2` and `convex3` objects, starting from the translated unit cube.

Script 10.3.1 (Back-face culling)

```
DEF Jewel (arg::IsPol) = arg1 & arg2 & arg3
WHERE
   arg1 = R:<2,3>:(PI/4):arg,
   arg2 = R:<3,1>:(PI/4):arg,
   arg3 = R:<1,2>:(PI/4):arg
END;

DEF convex1 = T:<1,2,3>:<-0.5,-0.5,-0.5>:(CUBOID:<1,1,1>);
DEF convex2 = Jewel:convex1;
DEF convex3 = Jewel:convex2;
```

The pictures resulting from hidden-surface removal by back-face culling of the geometric values obtained by evaluating the symbols `convex1`, `convex2` and `convex3` are shown in Figure 10.23.

Occlusion culling Although graphics hardware has improved greatly in recent years, advances have not been able to keep up with the rapid growth of model complexity. *Occlusion culling* is a popular technique for reducing the number of polygons to be processed by the rendering engine by culling away those portions of the geometry which are hidden from the viewer by other geometry.

Figure 10.23 Hidden-surface removal via back-face culling

An interesting and simple approach to this problem was recently presented by Bernardini and others [BKES00]. In a pre-processing stage, they approximate the input model with a hierarchical data structure and compute simple view-dependent polygonal occluders to replace the complex input geometry in subsequent visibility queries. When the user is inspecting and visualizing the input model, the computed occluders are used to avoid rendering geometry which cannot be seen.

Their Directional Discretized Occluders (DDOs) approach, as most of techniques to accelerate the rendering, involves a pre-processing phase where an object-space hierarchical data structure — an octree in their current implementation, see Section 13.2.2 — is generated. The actual scene geometry is stored with the leaf nodes of the octree, and nodes are recursively subdivided so that a bounded number of primitives per leaf node is obtained. Each square, axis-aligned face is a view-dependent polygonal occluder that can be used in place of the original geometry in subsequent visibility queries.

When the user is inspecting and visualizing the input model, the rendering algorithm visits the octree data structure in a top-down, front-to-back order. Valid occluders found during the traversal are projected and added to a two-dimensional data structure. Each octree node is first tested against the current collection of projected occluders: if the node is not visible, traversal of its subtree stops. Otherwise, recursion continues and if a visible leaf node is reached, its geometry is rendered.

This method has several advantages which allow it to perform conservative visibility queries efficiently and it does not require any special graphics hardware.

10.3.3 Scene coherence

Before analyzing a few paradigmatic HSR algorithms in some detail, it is worth discussing the *scene coherence* relationship used to increase the efficiency of most object-space algorithms.

For this purpose, let consider a simple scene with two sole objects, i.e. two sole space polygons. We want to understand under what conditions the viewpoint may see both polygons, or better, under what conditions none of them may cover the other making it (or some its parts) non-visible from the viewpoint. The aim is to find some formal conditions allowing dividing the HSR problem into subproblems that can be solved *independently.*

Visibility relation Given a pair of 3D polygons, every plane such that the polygons lie in different halfspaces is called *separation plane* of such polygons. We define as follows a binary *visibility* relation $\lhd$ on the set of 3D polygons:

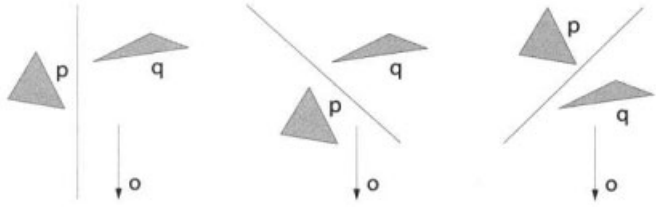


Figure 10.24 Three possible arrangements of two polygons and a separating plane

Definition The notation $a \lhd b$, where a and b are polygons, means that "a cannot cover b" with respect to viewpoint. We state that

$$a \lhd b$$

if there exists a separation plane that either (a) contains the viewpoint or (b) has a and the viewpoint in opposite halfspaces.

We can read the expression $a \lhd b$ as "*a not covers b*", or better as "*a beyond b*" — clearly with respect to viewpoint. First of all, let remember that in NPC the viewpoint is coincident with the point at the infinity of the z-axis. Then consider that, given two polygons p, q and a separation plane Π, there are three possible cases, as shown in Figure 10.24:

1. the plane Π contains the viewpoint, i.e. is parallel to the z-axis;
2. the viewpoint lies in the same halfspace of p;
3. the viewpoint lies in the same halfspace of q.

Hence, in case (1) we have both $p \lhd q$, and $q \lhd p$; in case (2) we have $q \lhd p$; in case (3) we have $p \lhd q$.

Visibility graph Let a scene description as a set of space polygons be given, and also suppose that a viewpoint position has been fixed. The graph of the visibility relation on a set of space polygons is called the *visibility graph*. Unfortunately the visibility relation *is not a partial order*. In fact, in general it is neither antisymmetric nor transitive. Anyway, the object-space algorithms must normally compute some partial ordering of polygons, according to such relation. To make the relation antisymmetric is easy. For any double pair $a \lhd b$ and $b \lhd a$ it is just sufficient to choose one pair. It is more difficult to make the relation transitive. For this purpose it is necessary to "open the cycles" of the kind $a \lhd b \lhd c \lhd a$. In this case it is sufficient to fragment a polygon within the visibility cycle, e.g. as shown by Figure 10.25.

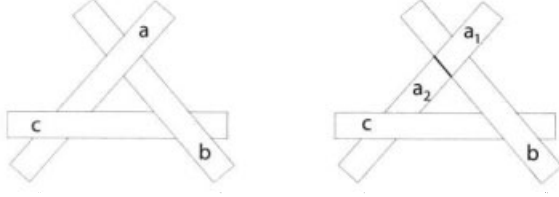


Figure 10.25 Opening a cycle in the *beyond* relation by fragmenting a polygon

A total ordering derived from the visibility graph is called *depth ordering*, and the

relative constructive algorithm is called *depth sort*. It often constitutes an important point of exact solutions to HSR problem.

Depth-sort

HSR algorithms in object-space are strongly based on the visibility relation, also called object coherence [FvDFH90], that is used to *depth-sort* [NNS72] or to *depth-merge* [SPGR93] the scene polygons, and on the subsequent rendering the polygons either in *back-to-front* or in *front-to-back* order, as discussed by the following subsections.

The so-called *depth-sort*, also known as *Newell's algorithm*, is actually due to Newell, Newell and Sacha [NNS72]. It is one of best known approaches to HSR problem.

On the set of polygons which results from pre-processing steps seen in Section 10.3.2, this algorithm computes a depth ordering and then generates the resulting picture by back-to-front rasterization.

Such a depth ordering is not exactly a sort on z, because each polygon is a set with infinite points, usually with variable z. More precisely, it is a total ordering compatible with the visibility relation previously discussed, so that it takes the name of *depth-sort*.

A black-box description of depth-sort follows:

1. (*Input*) culled, clipped and NPC-mapped but unordered scene polygons;
2. (*Output*) depth-ordered scene polygons.

Such an algorithm is based on a set of nested tests used to check the pairs of a preliminary total ordering of polygons produced by a z-sort. It may be described as constituted by two main steps, called *pre-sort* and *depth-sort*, respectively.

Let us denote with s_x, s_y, s_z the x-, y- and z-extensions of the s polygon, i.e. the polygon projections upon the coordinate axes, and with h_s the plane of s.

Sorting on z The polygons of the scene are preliminary ordered on the values of their minimal z coordinate. Any kind of sorting algorithm may be used for this aim.

Depth-Sorting The algorithm is a variation of a *bubble-sort*, where a quite complex test is used to compare if the generic pair of polygons is either already ordered or not.

Let p be the polygon currently ordered, i.e. whose space position is currently compared against those of polygons in the already ordered subset, according to the usual strategy of bubble-sort. Let us also assume that p must be currently compared against the q polygon.

We formally describe in Script 10.3.2 the comparison test on the pair (p, q) by using some pseudo-code.

The intersection tests on the coordinate projections of polygons are easily reduced to number comparisons. For example,

$$(p_z \cap q_z = \emptyset) \equiv (z_{max}(q) < z_{min}(p)) \vee (z_{max}(p) < z_{min}(q)), \tag{10.3}$$

and so on for the other coordinates. When using a z-sort as pre-processing, the above test may be simplified to the rightmost clause.

Script 10.3.2 (Depth-Test)
Algorithm DepthTest $(p, q :: IsPolDim :< 2, 3 >)$
{
if $p_z \cap q_z = \emptyset$, **then** $p \lhd q$
else if $p_x \cap q_x = \emptyset$ **then** $p \lhd q$
else if $p_y \cap q_y = \emptyset$ **then** $p \lhd q$
else if $\boldsymbol{o}$ and p are into *opposite* halfspaces of h_q **then** $p \lhd q$
else if $\boldsymbol{o}$ and q are into the *same* halfspace of h_p **then** $p \lhd q$
else if If $p_{xy} \cap q_{xy} = \emptyset$ **then** $p \lhd q$
else $Swap(p, q)$
}

The notation p_{xy} and q_{xy} stands for the projection of p and q in the xy plane, respectively. To test if geometric objects $\boldsymbol{o}$ and p live in opposite halfspaces of the plane h_q, we may substitute a point into the plane, test for sign and multiply the signs generated by two substitutions. If the result is 1, then the two objects live in the same halfspace; if the result is instead -1, then they live in opposite halfspaces.

If the *DepthTest* also fails with respect to the converse pair (q, p), then none of the two pairs may belong to the output ordering, because there is a cycle in the visibility relation. So, it is necessary to fragment one of the two polygons, e.g. by using the plane of the other for the cut.

Implementation

The depth-sort algorithm based on the Newell's visibility test is implemented here. As we know, the `PLaSM` language does its best when used in a declarative fashion. But it can be also usefully employed to quickly prototype complex geometric algorithms, as we show in this section.

There is a major difference between our implementation and the original formulation of the algorithm previously discussed. Instead of using the above variation of the *bubble-sort*, we make use of the `SORT` function already implemented in Script 2.1.30, which is a proper applicative implementation of the standard *merge-sort*.

It is well-known [AU92] that a bubble-sort has a worst case complexity $O(n^2)$, when its input is in reverse order. Conversely, the best behavior is reached when the input is already ordered. The pre-sort in the z-direction is hence performed with the aim of giving a quasi-ordered input to the proper depth-sort. Such pre-processing is not needed when a merge-sort algorithm is used, whose performance is $O(n \log n)$ for any possible input.

Toolbox The geometric functions needed are given in Script 10.3.3:

1. `mixedProd` computes the mixed product $\boldsymbol{a} \times \boldsymbol{b} \cdot \boldsymbol{c}$ of vectors $\boldsymbol{a}, \boldsymbol{b}$ and $\boldsymbol{c}$;
2. `box_int:`i`:<p,q>` predicate computes the test of equation (10.3) between the projections of polygons p and q along the i-th coordinate axis;
3. `three_points` returns the first three points from the first convex cell of (the unique polyhedral cell of) its input polygon. Such points are used to make geometric tests against the affine hull of the polygon;

4. `onPlane`, given three points p_1, p_2, p_3 and a further point q, returns the signed value of the mixed product of vectors

$$p_2 - p_1,\ p_3 - p_1,\ q - p_1.$$

   This value is zero when q belongs to the affine hull of p_1, p_2, p_3; otherwise its sign characterizes the set-membership of q to the affine halfspaces;
5. `plane_vs_points` returns the product of signs of numbers generated when checking two points for set-membership with a plane.

Few vector functions are also used, which are not collected here, including `vectDiff`, `vectProd` and `innerProd`. They were discussed in Chapter 3. The `SORT` function is defined in Script 2.1.30.

Script 10.3.3 (Toolbox)

```
DEF mixedProd (a,b,c::IsVect) = a vectProd b innerProd c;

DEF box_int (coord::IsInt) (p,q::IsPol) = OR:<
   GT:(MAX:coord:p):(MIN:coord:q),
   GT:(MAX:coord:q):(MIN:coord:p) >;

DEF three_points (polygon::IsPol) =
   (APPLY ~ [CONS ~ AA:SEL ~ [S1,S2,S3] ~ S1 ~ S2,S1] ~ UKPOL):polygon;

DEF onPlane (p1,p2,p3::IsPoint)(q::IsPoint) =
   (mixedProd ~ AA:vectDiff ~ DISTR): <<p2,p3,q>, p1>;

DEF plane_vs_points (a,b::IsFun;p::IsPoint) =
   * ~ AA:(SIGN ~ APPLY) ~ DISTL ~ [onPlane ~ a,[S1 ~ b, K:p]];
```

Newell's pre-sort The pre-processing step is not mandatory in our implementation, but it is given anyway, to stress the pattern of use of the `SORT` operator with different predicates. In this approach the input `polygons` are paired with their minimum z values, and the pairs are then sorted using the `z_pred` predicate to check if the generic couple of pairs is either already ordered or not.

Script 10.3.4 (Pre-sort)

```
DEF z_pred (a,b::ispair) = GT:(S2:a):(S2:b);

DEF presort (polygons::IsSeqOf:(IsPolDim:<2,3>)) = (S1
   ~ TRANS
   ~ SORT:z_pred
   ~ TRANS
   ~ [ID,AA:(MIN:3)] ): polygons;
```

Visibility test The `depth_test` predicate given in Script 10.3.5 is a direct translation of the Newell's visibility test on 3D polygon pairs already discussed. It is written as a sequence of five cascaded tests, and works assuming that there are no cycles in the visibility relation. We also assume that there are no intersections between polygons. This assumption is certainly true when they came from the boundary of non-intersecting solid bodies.

For sake of numerical robustness we enforce the precondition that pairs of polygons cannot share any portion of their boundary. The satisfaction of this requirement is guaranteed from the application of an explosion transformation, with parameters slightly greater than 1, before the depth-sort. See the `depth-sort` definition in Script 10.3.6.

Script 10.3.5 (Visibility test)

```
DEF depth_test = IF:< step1, K:true, K:<> >
WHERE
   step1 = IF:< box_int:3, K:true, step2 >,
   step2 = IF:< box_int:1, K:true, step3 >,
   step3 = IF:< box_int:2, K:true, step4 >,
   step4 = IF:< (C:EQ:-1):(plane_vs_points:<q,p,o>), K:true, step5 >,
   step5 = IF:< (C:EQ: 1):(plane_vs_points:<p,q,o>), K:true, K:<> >,
   p = three_points ~ S1,
   q = three_points ~ S2,
   o = < 0, 0, 1E6 >
END;
```

Depth-Sort A set of 3D polygons is extracted from the polyhedral `scene` by the `extract_polygons` function, defined in Script 10.8.4. Then, before depth ordering, such polygons are mutually disconnected by the function `explode:< 1.001, 1.001, 1.001 >` in order to put them in general position with respect to each other. The inverse explosion transformation is applied after the ordering has been produced by the `depth_sort` function.

The computational pipeline previously described is coded in Script 10.3.6 by the function called `depth_sort`, which is applied to the input `scene`. Notice that the pre-processing step is not mandatory in this approach. The `explode` function is given in Script 10.8.6.

Script 10.3.6 (Depth-Sort)

```
DEF depth_sort (scene::IsPol) =
   ( explode:< 1/1.001, 1/1.001, 1/1.001 >
   ~ SORT:depth_test
   ~ presort
   ~ explode:< 1.001, 1.001, 1.001 >
   ~ extract_polygons
   ): scene;
```

The algorithm has been written as above for sake of readability and compactness. A strong optimization of performance could be obtained, after the polygons extraction from the scene and their separation through the slight explosion, by pairing each polygon with its containment box and with the triplet of points used to construct its affine hull.

10.3.4 Back-to-front and front-to-back

The algorithms that are depth-sort based may be classified depending on the order of presentation of polygons on the output device. Both total orders generated from depth-sort, from furthest to nearest to the viewpoint, and its reverse, are used.

Back-to-front presentation When the input polygons have been depth-ordered, the Newell's algorithm generates the hidden parts removed picture by rasterizing the furthest polygon first. Then the second one in decreasing distance from the viewpoint is rasterized, where this one may cover some parts of the previous, and so on, until the polygon nearest to the viewpoint is presented on the device. When the presentation of depth-ordered polygons is finished, the only scene portions seen from the viewpoint are displayed. This strategy is called *back-to-front* presentation. Notice that a raster device is needed to execute this algorithm. Such a strategy, requiring drawing some picture areas more and more times, cannot be used on a vector device.

This method has some drawbacks. First, it can be used only with raster devices. Second, although the rasterization is very quick, when storing or moving a picture of a complex scene across a computer network, e.g. converted in *Postscript*, the size of data may become a problem, especially when we consider that most of polygons cannot be seen from the viewpoint.

Front-to-back presentation This name is given to the opposite strategy, where the picture of the scene is generated by drawing the depth-sorted polygons starting with the nearest polygon and ending with the furthest one.

In this case the whole first polygon is drawn and assumed as initial value of the so-called *apparent boundary* of the scene, i.e. the current boundary of the picture. Such an "occluder" is then subtracted from each following polygon in the depth-ordered sequence. Each non-empty result of such subtractions is drawn, since it is visible from the viewpoint. At the same time, the picture boundary is modified by union with each drawn polygon fragment. This procedure, repeated for each element in the *input* sequence, is described by using pseudo-code in Script 10.3.7.

This approach, which is perfect for vector devices, may also be used for presentations on raster devices. When used before a *Postscript* conversion of a vectorized scene picture, it may reduce some orders of magnitude the size of graphics data.

10.3.5 Binary Space Partition

The BSP algorithm removes the hidden parts of the scene by working in object space and does not require any specialized hardware. It was often used to generate animations in computer games (e.g. in the famous DOOM) as well as in real-time

Script 10.3.7 (Front-to-back)
Algorithm Front-to-back (*input* :: *IsSeqOf*: (*IsPolDim* :< 2, 3 >))
{
reverse the *input* sequence;
$B := \emptyset$; (initialize the apparent boundary)
for each polygon $p \in$ *input*:
{
$p := p - B$;
draw or rasterize p;
$B := B \cup p$;
}
}

walks-through architectural models.

This algorithm builds in a pre-processing step a specialized data structure, which is a sort of spatial index. This binary tree is called *Binary Space Partition tree*, or simply *BSP-tree*. Using this data structure, for each possible viewpoint position a depth-ordering is generated in linear time with the number of polygons. Due to this performance, the algorithm is sufficiently fast to allow the animation of quite complex scenes using non-specialized graphics hardware.

It is worth noting — if the scene is static and only the observer is moving — that the same BSP-tree may be used for ordering the polygons in all the possible views. Furthermore, a new depth-order must actually be computed only when the viewpoint leaves the current cell of the space partition induced by the BSP-tree. Various subsequent frames of an animation can thus be rendered without computing a new polygon ordering. Such a major property of computer animations was discovered in the 1970s under the name of *scene coherence*.

Approach It may be useful to distinguish three main phases with the BSP-algorithm:

1. *construction* of the BSP-tree;
2. BSP-tree *traversal*, with generation of a depth-ordering;
3. *rendering* of the sorted list of polygons.

So, the BSP-tree construction step is needed only once when the scene is static. Conversely, a traversal of the BSP-tree is necessary for each space cell crossed by the viewpoint trajectory, if any. If such a trajectory is predefined, then all the tree traversals can be computed in advance with respect to the proper animation. The rendering of the scene is clearly demanded by the 3D pipeline, and is not considered part of the realm of HSR algorithms, because it is necessary to compute for each frame the position of data in device space.

The BSP algorithm can be described in a dimension-independent way. We choose this approach because it will allow us to discuss BSP-trees with reference to various applications, including HSR, solid modeling, and representation of polygons, polyhedra and geometric manifolds of higher dimensions.

Definitions

Given a set of hyperplanes in Euclidean space E^d, a *BSP-tree* defined on such hyperplanes establishes a hierarchical partitioning of the E^d space.

A node ν of such a binary tree represents a convex and possibly unbounded region of E^d denoted as R_ν. The two sons of an internal node ν are denoted as $below(\nu)$ and $above(\nu)$, respectively. Leaves correspond to unpartitioned regions, which are called either *empty* (OUT) or *full* (IN) *cells*. Each internal node ν of the tree is associated with a *partitioning hyperplane* h_ν, which intersects the interior of R_ν. The hyperplane h_ν partitionates R_ν into three subsets:

1. the subregion $R_\nu^0 = R_\nu \cap h_\nu$ of dimension $d-1$;
2. the subregion $R_\nu^- = R_\nu \cap h_\nu^-$ where h_ν^- is the negative halfspace of h_ν. The halfspace h_ν^- is associated with the tree edge $(\nu, below(\nu))$. The region R_ν^- is associated with the *below* subtree, i.e. $R_\nu^- = R_{below(\nu)}$;
3. the subregion $R_\nu^+ = R_\nu \cap h_\nu^+$ where h_ν^+ is the positive halfspace of h_ν. The halfspace h_ν^+ is associated with the tree edge $(\nu, above(\nu))$. The region R_ν^+ is associated with the *above* subtree, i.e. $R_\nu^+ = R_{above(\nu)}$;

For any node ν in a BSP tree, the region R_ν is the intersection of the closed halfspaces on the path from the root to ν. The region described by any node ν is:

$$R_\nu = \bigcap_{e \in E(\nu)} h_e$$

where $E(\nu)$ is the edge set on the path from the root to ν and h_e is the halfspace associated with the edge e.

Tree construction

The BSP pre-processing consists of the scene tree construction. Such construction is executed in $O(n^3)$ in the worst case. Actually some heuristics are used to reach a near $O(n^2)$ complexity, to the cost of some sub-optimal increasing of scene storage.

Algorithm The planes used are those which contain the scene polygons. The binary tree is built inductively. The root plane ν is suitably chosen and is associated with its polygon. Such a root polygon $p(\nu)$ is thus eliminated from the list of scene polygons. This list is then split into two sublists associated with the *above* and *below* subtrees. The first one contains the polygons in R_ν^+; the second one contains the polygons in R_ν^-. Polygons which are crossed by ν are split and their fragments are associated to the proper subtree. Such process is repeated on each subtree until each sublist contains just one element.

Analysis It is essential for pre-processing efficiency that each root plane is chosen in such a way that (a) it crosses and splits the least possible number of polygons, so that their total number grows as small as possible, and that (b) the generated *above* and *below* subsets of polygons are relatively balanced, so that the subdivision process may

continue over subsets of near half cardinality. If both such constraints are satisfied at each step, then the algorithm works efficiently.

When the number of output polygons equates the number n of input polygons, such pre-processing has a cost variable between $O(n^3)$, for a completely unbalanced tree, and $O(n^2 \log n)$, for a perfectly balanced tree.

Actually, the total number of polygons may increase at each step depending on the choice of the root polygon, so that some satisfying compromise between the storage and time costs is heuristically searched.

Usually, each root plane is chosen in linear time, by comparing the size of the *above* and *below* sublists associated with some fixed number, usually between 4 and 8, of randomly chosen planes. Experimentally, this approach gives a pre-processing time between $O(n^2)$ and $O(n \log n)$, and a storage cost ranging between 1 and 4 times the size of input data.

The paper [PY90] by Paterson and Yao gives an algorithm for efficient binary space partitioning. It is efficient in the sense that, for an input space of n polygons, a naïve partioning scheme will result in a BSP tree of size $O(n^3)$ while this algorithm yields paritions of size $O(n^2)$.

Tree traversal

Some kind of *inorder* traversal [AHU83] is used to traverse a BSP tree. Remember that, if a binary tree is a single node, then the node is added to the output list; otherwise one of its subtrees is first traversed in inorder, then the root is processed, then the other subtree is traversed in inorder.

The *node processing* consists in determining if the viewpoint is either in the above or below subspace of the current plane. Such a processing is done in constant time, by substituting the viewpoint coordinates in the plane equation, and looking for the sign of the resulting number. Since the node processing requires a constant time, the whole tree traversal requires a linear time.

Script 10.3.8 (Traverse back-to-front)

```
Algorithm TraverseB2F (ν : BSPtree) {
    if IsLeaf (ν) then Output( ν )
    else {
        if o ∈ R_ν^+ then {
            TraverseB2F (below(ν))
            Output( ν )
            TraverseB2F (above(ν)) }
        else {
            TraverseB2F (above(ν))
            Output( ν )
            TraverseB2F (below(ν)) }
    }
}
```

The descent ordering in the two subtrees may vary in each node, depending on the

type of depth ordering to compute, which may be either *back-to-front*, say going from the plane which is further from the viewpoint to the closest plane, or *front-to-back*, say, going from the nearest to the more distant plane.

Back-to-front ordering According to the inorder traversal, if the node is a leaf, then the associated polygon is put in the output list. Otherwise: (a) the viewpoint position with respect to the current plane is evaluated; (b) the subtree associated with the subspace which does not contain the viewpoint is traversed in inorder; (c) the node is put in the output list, and (d) the subtree corresponding to the subspace which contains the viewpoint is finally traversed in preorder.

The geometric idea used here concerns the visibility relation. As we already know, no one polygon in the subspace which does not contain the viewpoint may cover the polygons in the viewpoint subspace. The back-to-front traversal algorithm is given in Script 10.3.8.

Front-to-back ordering If the node is a leaf, the polygon is put in the output list.

Otherwise: (a) the viewpoint position with respect to the current plane is evaluated; (b) the subtree associated with the subspace which contains the viewpoint is traversed in inorder; (c) the node is put in the output list, and (d) the subtree corresponding to the subspace which does not contain the viewpoint is traversed in preorder. The front-to-back traversal algorithm is described using pseudo-code in Script 10.3.9.

Script 10.3.9 (Traverse front-to-back)

```
Algorithm TraverseF2B (ν : BSPtree) {
    if IsLeaf (ν) then Output( ν )
    else {
        if o ∈ R_ν^+ then {
            TraverseF2B (above(ν))
            Output( ν )
            TraverseF2B (below(ν)) }
        else {
            TraverseF2B (below(ν))
            Output( ν )
            TraverseF2B (above(ν)) }
    }
}
```

10.3.6 HSR algorithms in image-space

Image-space algorithms are traditionally classified according to the type of raster subset they focus on. Three main classes are usually considered, respectively with pixel-based algorithms (z-buffer and ray-casting), line-based algorithms (including various scan-line algorithms), and area-based algorithms (including Warnock's algorithm and others). In the following of this section we only discuss the z-buffer

algorithm, because some variation of this algorithm is adopted today by the totality of 3D graphics accelerators.

z-buffer algorithm

A main aspect of this algorithm is that it does not need any preliminary ordering of the scene polygons. The algorithm efficiency is strongly affected by this fact.

Input/output The input to the algorithm is the unordered set of 3D polygons of the scene, already transformed in DC3 coordinates. The output is constituted by the final contents of the frame buffer, which contains a rasterized picture of the scene parts visible from the viewpoint.

Storage structures Two storage arrays are used, respectively denoted as *frame-buffer* and *z-buffer*. Such arrays have the same number of rows and columns, and their elements are associated to the pixels of the viewport on the display device. The frame-buffer contains a color value, usually either a composite RGB value or an index to some color look-up table. The z-buffer contains the discrete value of the z coordinate associated to the centroid of the polygon portion represented on a given pixel. Corresponding elements on the frame-buffer and on the z-buffer, say on the same row and column, are associated to the same polygon portion.

Notice that the z-buffer elements hold integer numbers, in a range of values depending on the number of bits per element, often equal to 24. Of course, the depth resolution of the algorithm depends on the quantity of RAM available for the z-buffer. This resolution may strongly affect the output quality when rastering 3D scenes of high geometric complexity.

Algorithm The algorithm initialization can be distinguished from the algorithm core. In the initialization step the frame-buffer is set to the background color, and the z-buffer is set to the minimum representable value, in the hypothesis that a right-handed frame is used for the DC3 coordinates. The view direction is that of positive z axis.

In the algorithm core the scene polygons are rasterized in any order. Let us consider the processing of the generic polygon. A covering with a discrete set of pixels at integer DC3 coordinates is produced, and the discrete z for each pixel is also computed. Such set of pixels give a minimal covering of the polygon image, but only a pixel subset is usually stored, according to their current visibility from the viewpoint.

Each polygon is rasterized by rows. For each generated pixel we need to decide if either it must be stored in the frame buffer and in the z-buffer, or not. Such a decision is positive if the pixel already stored in memory at the same xy address (i.e. at the same <column, row> pair) is further from the observer than the current pixel or vice versa. It is considered further when the integer value stored in the z-buffer is smaller than the z of the current pixel.

When all the scene polygons have been rasterized, the frame buffer will contain the scene picture really perceived from the viewpoint.

The generic rasterization step requires the computation of the z coordinate of the

current pixel starting from the known z of the previous pixel.

Let $\boldsymbol{p}_k = (x_k, y_k)$ be the current pixel, belonging to the y_k row and to the x_k column of the device space. The z coordinate in $\boldsymbol{p}_k$ and $\boldsymbol{p}_{k+1}$ pixels may be computed by substituting the ordinate and abscissa of the two points into the plane equation of their polygon:

$$\begin{aligned} ax_k + by_k + cz_k + d &= 0, \\ ax_{k+1} + by_{k+1} + cz_{k+1} + d &= 0. \end{aligned}$$

In other terms:

$$\begin{aligned} z_k &= -\frac{1}{c}(d + ax_k + by_k), \\ z_{k+1} &= -\frac{1}{c}(d + ax_{k+1} + by_{k+1}), \end{aligned}$$

and hence

$$\Delta z = z_{k+1} - z_k = -\frac{a}{c}(x_{k+1} - x_k) - \frac{b}{c}(y_{k+1} - y_k).$$

But $y_{k+1} - y_k = 0$ and $x_{k+1} - x_k = 1$, so that

$$z_{k+1} = z_k + \Delta z = z_k - \frac{a}{c}. \tag{10.4}$$

By comparing z_{k+1} with the value already stored in position (x_{k+1}, y_{k+1}) of z-buffer, it is possible to decide whether to store the pixel depth z_{k+1} or not. If the z value is stored, then the pixel color is also computed and stored. For this purpose other computations may be required, including normal interpolation and/or color shading, as discussed in the following sections.

Analysis It may be interesting to note that, as shown by equation (10.4), the computation of the z increment from the previous pixel only requires a subtraction by a constant number. This algorithm, which is extremely simple, is easily implemented in firmware on 3D graphics accelerators. Notice that the generation time of the hidden-surface removed picture depends linearly on the average area of polygons and on their number.

10.4 Illumination models

To generate realistic images by computer it is necessary both to remove the hidden-surfaces of the scene and to render the visible ones by taking into account the physical principles which regulate the diffusion and reflectance of light rays that incise on the external surfaces of bodies in the scene to be rendered. In particular, we need to take into account not only the geometry of the scene, but also some optical and physiologic aspects related to the diffusion, reflection and perception of light.

In this section we aim to discuss how the surfaces reflect and diffuse the incident light radiation. Let us first remember from optics that the light intensity incident on the exterior of a body may be

1. absorbed;
2. transmitted;
3. reflected, both diffusely and specularly,

in percentages that depend both on the body material and on the nature of the fixture of its exterior surface.

Lighting models The terms "lighting" or "illumination models" are used to denote the choice of some particular equation to compute the light intensity in a space point, as a function of the intensity of the incident radiation and the geometric and physical properties of the surface the point belongs to. The goal when using lighting models is to compute the light intensity on the surfaces of the scene, given some assigned distribution and configuration of light sources and some assigned material properties of the surfaces.

Sources When we model a scene, we have to give both its geometry and some suitable light sources. Such sources may be either set at infinity or at some finite position. As a consequence, light rays will be either parallel or divergent, respectively. Light sources may be geometrically shaped either as point sources or as distributed, i.e. extended, sources. Depending on requirements, we may have scenes with only one light source or with several sources. Also, we may imagine that the emitted light has the same intensity in all the directions, or that a source shines in a preferred direction. Such sources are called directional light sources.

Absorbed radiation The portion of incident radiation which is absorbed by a body is transformed into heat and produces an increase in temperature. Notice that if the body had absorbed all the incident light, then it would be not visible. Such perfectly absorbent body is called "black body". The absorbed radiation is not interesting for our purposes, but it may be useful to note that different bodies do not absorb the different wave lengths in the same way. For this reason the different materials look as if they have different colors when exposed to the daylight.

Transmitted radiation The incident radiation is partly transmitted inside the body when this is transparent. This case will not be considered in the remainder of the text. On this point we will only discuss how to set the percentage of transparency when stating the material properties in VRML.

Diffusely reflected radiation The diffuse radiation can be thought to be due to the portion of incident radiation which is absorbed by the external layer of the body, and then is uttered in all directions allowed by the position and orientation of the external surfaces. We will see that, at least in computing the diffuse intensity using the simplest diffusion model, it is not necessary to consider the viewpoint position, since the body, in the case of perfect light diffusion, behaves exactly as an emitter body.

Specularly reflected radiation We may imagine that the specularly reflected portion of the incident radiation does not traverse the external layer of the body, but is directly reflected in the plane defined by the direction of the normal to the surface in the considered point and by the direction of the incident light. In this case we need to compute how much reflected intensity may geometrically reach the viewpoint position.

10.4.1 Diffuse light

We know that the incident light on a opaque body may be partly diffused, partly reflected and partly absorbed by the body. Let us consider the simplest lighting model, where only the diffused light intensity is taken into account. The involved vector quantities are shown in Figure 10.26, where we have

1. $\boldsymbol{p}$ = point on the body surface;
2. $\boldsymbol{n}$ = normal unit vector to the surface in $\boldsymbol{p}$;
3. $\boldsymbol{\ell}$ = direction of incident light, given by the difference between the source position, supposed point-shaped, and $\boldsymbol{p}$;
4. θ = angle between vectors $\boldsymbol{\ell}$ and $\boldsymbol{n}$.

Lambert's diffuse intensity The physical phenomenon is regulated by the following *Lambert's cosine law.* If a body is a perfect diffuser, then the diffused intensity $I_d(\boldsymbol{p})$ from a point $\boldsymbol{p}$ of its surface is proportional to the incident light intensity and to the cosine of the angle θ:

$$I_d(\boldsymbol{p}) = K_d I_\ell \cos\theta, \tag{10.5}$$

where I_ℓ is the light intensity incident in $\boldsymbol{p}$. The factor K_d is called *diffusion constant.* The θ angle must necessarily vary between 0 and $\frac{\pi}{2}$.

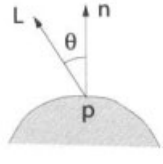


Figure 10.26 Lambert's cosine model of diffused reflection

The model described by equation 10.5 is the simplest lighting model, with a single light source which illuminates a single opaque body. For θ greater than $\frac{\pi}{2}$, we have $I_d(\boldsymbol{p}) = 0$, where no light would be available since the $\boldsymbol{p}$ point would not be visible from the source.

Such a single assumption is too strong, because it would render as *black* all the surface portions where $\frac{\pi}{2} \leq \theta$. In other words, when using this simple model, we should render as black all the scene polygons which are not presently visible from both the viewpoint and the light source. Such a result would be actually unacceptable.

Ambient diffuse intensity The problem of black surfaces discussed above may be easily solved by considering that in any scene there is some diffused lighting due to

surrounding environment. For this purpose it is sufficient to assume that on each body surface there is also some, usually small, quantity of light not directly received from a light source but coming from the ambient surroundings of the scene. In other terms, it is assumed that each body will diffuse a constant portion of the ambient lighting.

An additive *ambient* term $K_a I_a$ is hence introduced in the diffusion equation 10.5, depending on the average *ambient lighting* I_a, characteristic of the ambient of the scene, and on the *ambient diffusion* constant K_a, characteristic of the body material:

$$I_d(\boldsymbol{p}) = K_d I_\ell \cos\theta + K_a I_a \tag{10.6}$$

Notice that black surfaces may appear again, as soon as either K_a or I_a are equal to zero. But also this simple model has some important drawbacks. Consider in fact two parallel polygons made with the same material and located at different distances from the viewpoint. By using the above equation they would have the some light intensity in each point (and hence exactly the same color), so that, in case of partial visual occlusion with respect to the viewpoint, it would be actually impossible for the observer to distinguish the first surface from the second surface.

Depth attenuation of intensity To get more realistic results it is necessary to take into account the intensity attenuation as a function of depth. In particular, we should consider as appropriate physical parameter the *intensity flow*, defined as the energy which crosses the unit surface. This parameter would be measured at distance d by the intensity diffused in $\boldsymbol{p}$, over the area $4\pi d^2$ of the spherical surface centered in $\boldsymbol{p}$. The intensity attenuation should be hence made proportional to the inverse of the squared distance.

But dividing the first term of equation (10.6) by the square of the viewpoint distance gives results which do not experimentally match the user experience. In particular it is possible to notice that by varying the intensity (i.e. the color) with the inverse of the square of the viewpoint distance produces a too strong variation between points belonging to surfaces located at relatively small distances from each other.

Conversely, some good visual results are given by the equation:

$$I_d(\boldsymbol{p}) = \frac{K_d}{d+K} I_\ell \cos\theta + K_a I_a \tag{10.7}$$

where d is the viewpoint distance from $\boldsymbol{p}$ and K is an additive constant which gives a "fine-tuning" of the lighting behavior of surfaces, until to make them look realistic. Notice that d cannot be interpreted as the distance from viewpoint if this one is at infinity. In such a case d is assumed as the z difference between $\boldsymbol{p}$ and the scene point with maximum z.

10.4.2 Specular reflection

If a body is perfectly reflective, say, a perfect mirror, then all the radiation incident on $\boldsymbol{p}$, which is not absorbed or transmitted, is reflected along the symmetric direction of $\boldsymbol{\ell}$ with respect to $\boldsymbol{n}$, i.e. along the reflection vector $\boldsymbol{r}$. See Figure 10.27.

With a perfectly reflective body, the observer would see the considered point only if located along the reflection direction. Actually, real bodies do not behave like perfect

ones, and do not reflect the light only along the reflection direction. Conversely, the reflected intensity is spatially distributed around $\boldsymbol{r}$. Hence the viewer, located on the view direction $\boldsymbol{v}$, may in any case get a perception of some portion of the reflected light, and can see $\boldsymbol{p}$ when $\theta + \alpha < \frac{\pi}{2}$.

In particular, the reflected light intensity perceived along $\boldsymbol{v}$ is a function of the angle α between $\boldsymbol{v}$ and $\boldsymbol{r}$, as shown in Figure 10.27. Notice that the view direction $\boldsymbol{v}$ is not necessarily coplanar with the $\boldsymbol{\ell}$, $\boldsymbol{n}$ and $\boldsymbol{r}$ vectors.

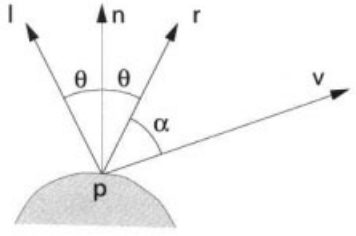


Figure 10.27 Specular reflection from a point source

The quantity of specularly reflected energy strongly depends in a quite complex way both on the wavelength λ of the incident light and on the angle α between the reflection and view directions.

Phong's reflection model A simplified description of the phenomenon is due to Phong Bui-Tuong [Pho75]. In this model it is assumed that the *specularly reflected intensity* $L_s(\boldsymbol{p})$ is proportional to (a) the incident radiation I_ℓ as well as to (b) the n-th power of cosine of angle α between the reflection and the view directions, through a function $w(\theta, \lambda)$ which depends on the incidence angle θ and on the wavelength λ.

$$L_s(\boldsymbol{p}) = w(\theta, \lambda) I_\ell \cos^n \alpha$$

The n parameter depends on the body material. In Figure 10.28 we show the graph of $\cos^n \alpha$ function, as generated by Script `PLaSM` 10.4.1.

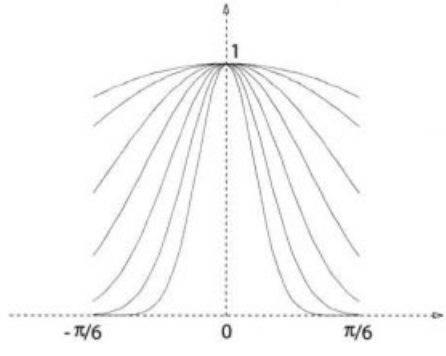


Figure 10.28 Graph of function $\cos^n \alpha$, for $-\frac{\pi}{6} \leq \alpha \leq \frac{\pi}{6}$, and with samples of n between 1 and 80

In Figure 10.28 are shown the effects of various values of n on $\cos^n \alpha$ and hence on

the spatial distribution of energy. The exponent n is low with no reflective material, e.g. paper, and can reach values between 50 and 100 with very reflective materials, such as metals with surface polished as a mirror.

Script 10.4.1 (Graphs of $\cos^n \alpha$)

```
DEF graph(n::IsIntpos) = MAP:[s1, ** ~ [cos ~ S1, K:n]]:
   ((T:1:(-:PI/6) ~ intervals:(pi/3):40);
DEF COSnGraphs = (STRUCT ~ AA:Graph):< 1, 2, 5, 10, 20, 40, 80 >;
```

Simplified model of specular reflection The w function of Phong's model, which takes into account both the incidence angle θ and the incident wavelength λ, is actually very complex, and can only be given empirically. It is usually substituted by a constant K_s, called the *specular reflection constant*, so that we have the simplified reflection model:

$$L_s(\boldsymbol{p}) = K_s I_\ell \cos^n \alpha.$$

The material properties are thus incorporated in the parameter n, whereas the geometric (incidence angle) and physical (wavelength) properties of the incident light are summarized by the K_s constant. Since the aspects already discussed for the diffused intensity continue to hold, the following better expression is usually adopted for the specularly reflected intensity:

$$I_s(\boldsymbol{p}) = \frac{K_s}{d+K} I_\ell \cos^n \alpha + K_a I_a. \tag{10.8}$$

Aggregated reflection models In summary, the expressions already seen for diffuse and specularly reflected intensities can be aggregated, since common bodies usually behave both as light diffusers and as mirrors. The aggregated model for the reflected light intensity is

$$I(\boldsymbol{p}) = \frac{I_\ell}{d+K}(K_d \cos\theta + K_s \cos^n \alpha) + K_a I_a \tag{10.9}$$

where K_d, K_s and n are material properties, I_ℓ depends on the light source, and α, θ, K and d are characteristics of the geometric configuration of the surface. The geometric parameters of the illumination model are better "highlighted" (!) by using inner products of unit vectors rather than cosines of angles:

$$I(\boldsymbol{p}) = \frac{I_\ell}{d+K}\left(K_d\, \boldsymbol{\ell} \cdot \boldsymbol{n} + K_s\, (\boldsymbol{r} \cdot \boldsymbol{v})^n\right) + K_a I_a. \tag{10.10}$$

More in general, several light sources may appear on the stage, with different incident light intensities and properties. Hence, for the reflected intensity $I(\boldsymbol{p})$ in a point illuminated by various sources we can write

$$I(\boldsymbol{p}) = \sum_i \left[\frac{I_{\ell_i}}{d + K_i} (K_d \, \boldsymbol{\ell}_i \cdot \boldsymbol{n} + K_s \, (\boldsymbol{r}_i \cdot \boldsymbol{v})^n) \right] + K_a I_a. \tag{10.11}$$

Geometry of reflection models Vectors $\boldsymbol{\ell}, \boldsymbol{n}$ and $\boldsymbol{v}$ are independent variables of the geometric problem. In particular, $\boldsymbol{\ell}$ can be obtained by normalizing the point difference between the source location and the considered point $\boldsymbol{p}$. The $\boldsymbol{v}$ vector is computed analogously, by normalizing the point difference between the viewpoint and $\boldsymbol{p}$.

The normal vector $\boldsymbol{n}$ is constant on each planar surface. In a polygon it can be computed by the vector product of two consecutive edges with an internal angle less than $\frac{\pi}{2}$. Such a convex internal angle always exists, for both convex and concave polygons, where the common vertex of two edges is extremum with respect to one coordinate, e.g. is the point with either maximal or minimal z, provided that the polygon does not lie on a plane with equation $z = c$.

The computation of the direction of reflection $\boldsymbol{r}$ is a bit more complex. In this case it is necessary to compute a rototranslation tensor $\boldsymbol{Q}$ which maps $\boldsymbol{p}$ in the origin and the normal unit vector $\boldsymbol{n}$ in the basis vector $\boldsymbol{e}_3$ of the z axis. So we have, in homogeneous coordinates:

$$\boldsymbol{r} = \boldsymbol{Q}^{-1} \boldsymbol{S}_{xy}(-1, -1) \, \boldsymbol{Q} \, \boldsymbol{\ell},$$

where $\boldsymbol{S}_{xy}(-1, -1)$ is the scaling tensor that reverses the sign of the x and y coordinates.

10.5 Color models

In this section we quickly present the more important concepts about color as individual perception of colored lights. We also discuss the additive and subtractive color theories, needed to understand the production of colored pictures done by computer monitors and printers, respectively, and some common models of color spaces.

Additive and subtractive color The term *color* stands for the cerebral sensation produced when the human eye is hit by electromagnetic radiation in the *visible spectrum*, i.e. in the range of wavelengths between 400 and 700 *nm* (nanometers), with $1\,nm = 10^{-9}$ meters.

When discussing the *perception* of color it is necessary to take into account both physical-optical and psycho-physiologic factors. In particular, some experimental apparatus is needed to define the concept of "visible color". In fact, the observer might get the same color perception even in the presence of sources emitting light radiations with different distributions of frequency.

Hermann von Helmholtz, a nineteenth-century German physiologist and physicist, proposed the well-accepted theory where he postulated that the human eye contains three physiologic structures which are able to perceive only the so-called fundamental colors *red*, *green* and *blue*.

Every different color is hence individually perceived as a proper summation of appropriate quantities of the three fundamental colors. The *additive theory* of perception of colored lights is built on such a very basic assumption. Conversely, to explain the color perceived by eyes/brain when receiving light reflected by materials which do not emit any radiation, such as e.g. press inks or colored clothes, the so-called *subtractive color* theory is used. In this case it is postulated that fabric material would subtract (by absorbing them) some part of fundamental colors from the incident daylight.

Frequency distribution spectrum The *average daylight* is sometimes called *white* or natural *light*. This light, when observed through a prism or spectroscope, appears to be composed of some different colored lights, that we collectively call the *spectrum of visible light*, and is shown in Figure 10.29. To the extremes of the visible spectrum there are the minimum and maximum wavelengths perceivable by a human, corresponding to violet (400 nm) and red (700 nm).

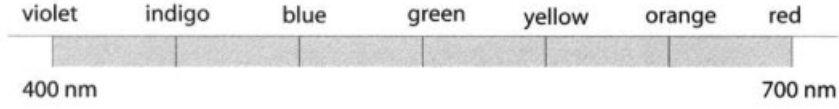


Figure 10.29 Schematic representation of the visible spectrum

In the first half of the twentieth century several empirical experiments were done to define the concept of *visible color*, in particular by using an experimental machinery where a light of unknown color is compared with suitable triples of lights of fundamental colors. See Figure 10.30.

In particular, let us imagine red, green, blue light sources and a further source of light of an unknown color, where some potentiometers allow control of the intensity of fundamental lights. Such fundamental colors are summed and the observer may change their individual intensities until they "match" the unknown light. If such a match is possible, then it makes sense to state that

$$I(C) = I(R) + I(G) + I(B).$$

It is on the basis of several successful experiments that the additivity of fundamental color lights was postulated.

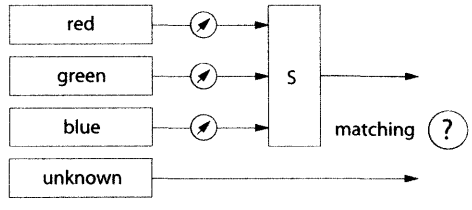


Figure 10.30 Experimental machinery for unknown color "matching" with fundamental colors

Actually, it is not always possible to sum three monochromatic lights and to match the unknown colored light. To understand why it is sufficient to consider that

monochromatic lights, like any other light, have some distribution on the frequency spectrum, and hence they contain also some portion of the two other monochromatic lights. Another problem comes from the fact that the same subjective color perception might be obtained from totally different spectra. A frequency spectrum was hence considered not useful for color characterization.

Normalized representation of color The *Comité Internationale de l'Éclairage* (CIE) in 1931 adopted the *additive* color theory, and the so-called *normalized representation* of color. The color system is assumed to be linear and purely additive. Three primary spectrum distributions, called X, Y and Z, respectively, were substituted to red, green and blue in the "matching" process. Each other color intensity can be expressed as a sum of primary intensities $I(X)$, $I(Y)$ and $I(Z)$. Three normalized ratios, called chromaticity values, are defined as

$$\begin{aligned} x &= \frac{I(X)}{I(X)+I(Y)+I(Z)} \\ y &= \frac{I(Y)}{I(X)+I(Y)+I(Z)} \\ z &= \frac{I(Z)}{I(X)+I(Y)+I(Z)} \end{aligned}$$

By definition, the sum of chromaticity values is hence unitary:

$$x + y + z = 1,$$

and each value is non-negative. Thus, two of such parameters are independent, and their values may be represented in a two-dimensional plane. In particular, each triplet (x, y, z) can be represented as a point in the unit triangle with vertex $(0,0)$, $(1,0)$ and $(0,1)$, as shown in Figure 10.31. The chromaticity values (x, y, z) can be considered to be the convex coordinates of points in such a triangle.

Chromatic diagram The average experimental results of the matching process can be represented as points in the unit triangle of chromaticity values. Such points give a representation of how an average observer perceives the visible colors. Even the pure colors in the daylight spectrum become points in such a diagram, and are distributed along a curve which resembles the shape of a boot sole. We may orderly recognize *red*, i.e. the radiation at 700 nm, followed by *yellow*, *green*, *cyan*, *blue*, *violet*, i.e. the radiation at 400 nm. Since it is assumed that the phenomenon is linearly additive, it is worth considering each visible color as represented by a point on the segment which connects two component colors. Analogously, if any triplet of colors is fixed, their triangle gives all the visible colors which are obtainable by their combination. In particular, the *centroid* of the triangle of primary colors is the point where the intensities of primary colored lights are equal, and corresponds to the *white* color.

This diagram is called chromaticity diagram or *standard chromaticity diagram*. It is the result of a quite complex standardization work. In particular, a *standard*

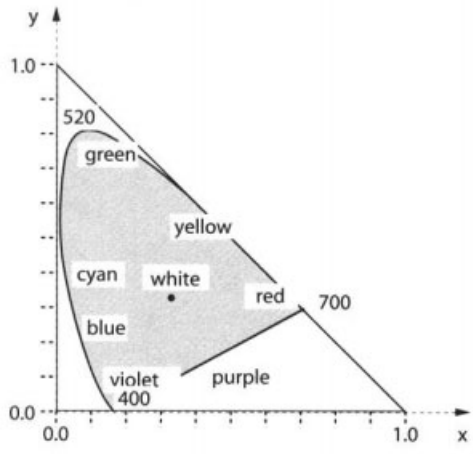


Figure 10.31 Standard chromaticity diagram by CIE

distribution was defined for three *primary lights*. The colors of the spectrum of daylight, called *pure* or *saturated* colors, are distributed on the boundary of the diagram. Each other *visible color* on this diagram, as an effect of the linearity assumption, can be considered to be obtained by mixing a saturated color with white. The corresponding coefficient is called the *degree of saturation* of the color. In particular, a color is saturated when it approximates to some pure color.

For each color c let us trace the segment passing for it and the white w, until to encounter the pure color p on the boundary of the chromaticity diagram. In this case we can write:

$$c = (1 - \alpha)\, w + \alpha\, p, \qquad 0 \leq \alpha \leq 1.$$

where the *saturation* of color c is defined as the α coefficient of the convex combination of p and w, or, in other words, as the ratio of distances of c from p and w. Usually, the color saturation is expressed as a percentage, so that a color saturated at 100% is pure, whereas one saturated at 0% coincides with white color.

Complementary colors Take any color c. The color b such that its summation with c gives the white w:

$$b + c = w$$

is called the *complementary color* of c. The complementary color of c is easily computed by considering the straight line for c and w, and getting the color point at same saturation on the halfline opposite to c with respect to w. Notice that the complement of a pure color is a pure color. The names of complementary colors of additive primaries are given in Table 10.2.

The standard chromatic diagram is closed by a segment joining the two extreme pure colors of daylight, thus enclosing the area shown as gray in Figure 10.31, where we have the pure colors. Such a "closure line", called the *purple line*, is only obtainable by convex combinations of the two extremes of the line, i.e. of *red* and *violet*. The internal points of such a diagram represent the set of *visible colors*, also called the *perceivable colors*.

Table 10.2 Mapping between primary additive colors and complementary colors

Primary color	Complementary color
Red	Cyan
Green	Magenta
Blue	Yellow

Gamout of a monitor The screen of a monitor device cannot generate the whole set of visible colors. Actually each screen is able to generate only a triangular subset of colors. This set of colors realizable from a monitor screen is called the monitor *gamout*. For each monitor a triplet of points in the chromaticity diagram is assumed to be representative of *red*, *green* and *blue* lights. See Figure 10.32.

The design of a monitor screen is thus characterized by a triplet of pairs of chromaticity values in the CIE chromaticity diagram. This triangle is also called the RGB (*red*, *green*, *blue*) triangle of the screen. The area of this triangle is a good index of the screen quality. The wider is the area, the better is the quality.

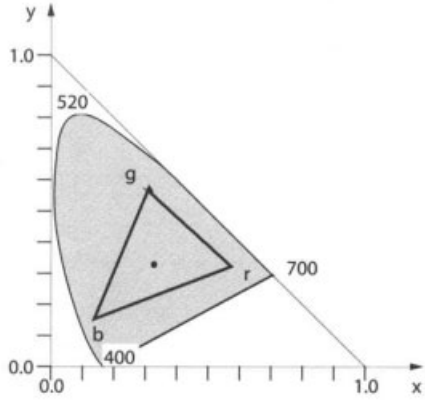


Figure 10.32 Typical monitor gamout, representative of the set of realizable colors

RGB color model In the chromaticity diagram the light intensity is not taken into account. For this purpose a three-dimensional diagram is introduced and used as a model of realizable colors, where the intensity is explicitly considered. In particular, the intensities of primary colors, normalized between 0 and 1, are associated with the axes of a 3D reference frame. Each realizable colored light is thus associated with the points of the standard unit cube. This model is called the *RGB cube* or *RGB color model.*

In this model each color point is represented by a triplet in $[0,1]^3$, by varying the intensities of primary colors. So, the point $R = (1,0,0)$ represents the *red* color at maximum intensity and analogously the point $G = (0,1,0)$ identifies the *green* color; the point $B = (0,0,1)$ gives the *blue* color. The origin $O = (0,0,0)$ with zero intensity for each primary light is associated with the *black* color. Analogously the

point $W = (1, 1, 1)$ corresponds to the *white* color. Each other point on the line segment between *black* and *white* is a triplet of equal numbers. Such a segment is called *line of grays*.

Other interesting points of the RGB model are the vertices where two coordinates are unitary and one is null. They correspond to complementary colors *cyan* $C = (1, 1, 0)$, *magenta* $M = (0, 1, 1)$ and *yellow* $Y = (1, 1, 0)$. The RGB color model is quite important because it closely resembles the way color is stored in memory. Triples of integers corresponding to intensities of primary colors are usually stored, with a number of bits depending on the number of available colors. As we will see in Section 10.7.2, the RGB model is used in VRML to specify the values of type *SFColor*.

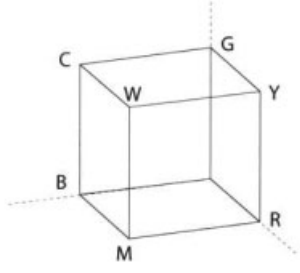


Figure 10.33 RGB cube with both primary (*red*, *green*, *blue*) and complementary (*cyan*, *magenta*, *yellow*) colors in the cube vertices. The remaining vertices are associated with *black* and *white*

Example 10.5.1 (RGB color cube generation)
An interesting example of color representation by embedding, written exploiting the dimension-independent geometry representation of the PLaSM language is given here. For this purpose it is sufficient to repeat each triplet of coordinates of vertices of the unit standard cube $[0, 1]^3$ as coordinates in RGB color space.

Script 10.5.1 (RGB color cube)

```
DEF RGBcube = MKPOL:<<
   <0,0,0, 0,0,0>, <1,0,0, 1,0,0>, <0,1,0, 0,1,0>, <1,1,0, 1,1,0>,
   <0,0,1, 0,0,1>, <1,0,1, 1,0,1>, <0,1,1, 0,1,1>, <1,1,1, 1,1,1>
>,<1..8>,<<1>>>;

VRML:RGBcube:'out.wrl';
VRML:(@1:RGBcube):'out.wrl';
```

It may be interesting to note that the intrinsic dimension of RGBcube object generated by Script 10.5.1 is 3, whereas the dimension of its embedding space (i.e. the number of its coordinates) is 6. As a matter of fact, we have:

```
RGBcube ≡ A-Polyhedral-Complex{3,6}
```

Since the output object has a number of coordinates comprised between 3 and 6, the generated VRML file contains a *color per vertex* representation of the polyhedral

parameter. According to the number of coordinates, the VRML file may contain either a representation with colors per vertex, or with normals per vertex, or both.

The results of exporting the geometric values generated by the expressions `RGBcube` and `@1:RGBcube` are shown in Figures 10.34a and 10.34b, respectively. Notice that the skeleton extractor function handles correctly a wire-frame model with `color-per-vertex` encoding.

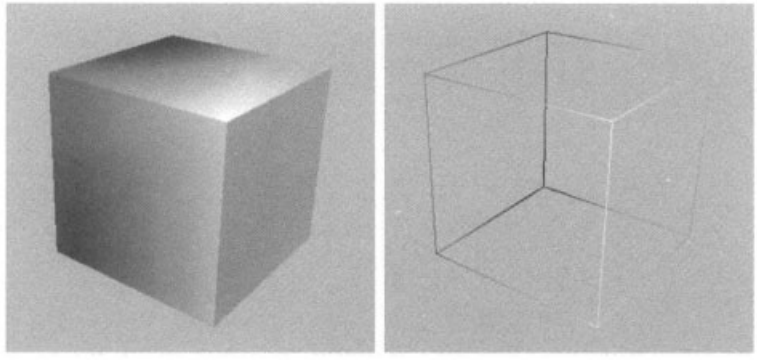


Figure 10.34 (a) VRML rendering of the 3D object `RGBcube` embedded in 6D space according to Script 10.5.1 (b) 1D skeleton of `RGBcube`

Prismatic HSV color model For user interaction the so-called *hue, saturation, value* (HSV) color model is probably more interesting to the standard user. This color model is based on the saturation of a pure color (hue) with the white, and on the variation of intensity.

In particular, the intensity value (V) is associated with an axis labeled with numbers in $[0, 1]$. On the plane $V = 1$ is given a hexagon with a unit radius of the circumscribed circle. The hexagon vertices are orderly labeled with pure colors *red, yellow, green, cyan, blue* and *magenta*.

The edges of such a hexagon correspond to the colors which are generated by a convex combination of the two colors associated with the edge vertices. The angular parameter, called hue (H) (also called *tint* or intrinsic color), which clearly varies in the interval $[0, 2\pi]$, gives a description of the pure colors. By convention, the red color is associated with $H = 0$.

The radial parameter S ranging in $[0, 1]$ is called *saturation*. The color point associated with $S = 0$ and $V = 1$ is the *white*; the one associated with $S = 0$ and $V = 0$ is the *black*. The segment between such points is called the *line of gray*, as in the RGB model.

Example 10.5.2 (HSV color prism generation)
An embedding technique similar to that of Example 10.5.1 may be used to create a VRML model of the HSV color prism through the `MKPOL` primitive, as provided in Script 10.5.2. The generated VRML model is shown in Figure 10.36. Notice that in this case a polyhedral cell made of six tetrahedral cells is defined. A direct construction as a single convex cell made by either seven or eight $\mathbb{E}^6$ points is not possible, because the convex hull of such points would have intrinsic dimension 4.

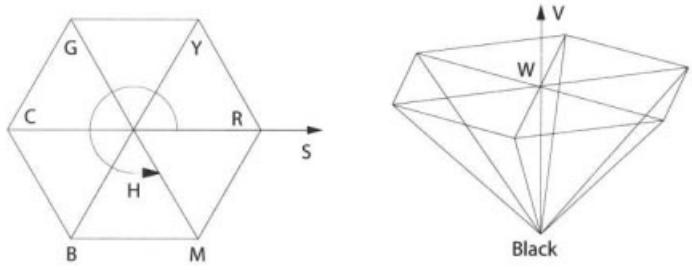


Figure 10.35 HSV prism color model with labeled verticeso

Script 10.5.2 (HSV color prism)

```
DEF HSVprism = MKPOL:<<
   < 0,0,0,                              0,0,0 >, % black   %
   < 0,0,1,                              1,1,1 >, % white   %
   < 1,0,1,                              1,0,0 >, % red     %
   < COS:(PI/3),SIN:(PI/3),1,            1,1,0 >, % yellow  %
   < COS:(2*PI/3),SIN:(2*PI/3),1,        0,1,0 >, % green   %
   < -1,0,1,                             0,1,1 >, % cyan    %
   < COS:(4*PI/3),SIN:(4*PI/3),1,        0,0,1 >, % blue    %
   < COS:(5*PI/3),SIN:(5*PI/3),1,        1,0,1 >  % magenta %
>,<
   <1,2,3,4>, <1,2,4,5>, <1,2,5,6>,
   <1,2,6,7>, <1,2,7,8>, <1,2,8,3>
>,<1..6>>;

VRML:HSVprism:'out.wrl';
```

The HSV prism is often transformed into a cone, and sometimes into a cylinder. To generate such models may be an interesting exercise for the reader. *Hint:* use the MAP primitive to apply a suitable coordinate transformation to some properly decomposed 3D domain embedded in 6D.

CMY color model The *cyan*, *magenta*, *yellow* (CMY) model of *primary subtractive* colors is particularly suited to color management of printing devices, which do not use light sources, such as screen monitors, or print inks. Such materials reflect the incident daylight after having absorbed the frequencies of the complementary additive primary color. Hence, e.g., the *cyan* ink, which is the complementary color of *red* absorbs from the daylight, which is a mixture of *red*, *gree* and *blue*, its red frequencies, so reflecting a light which is a mixture of *green* and *blue*. Analogously, the *magenta* and *yellow* inks respectively reflect (a) a mixture of *red* and *blue* and (b) a mixture of *red* and *green*.

When two primary subtractive inks are blended together, their mixture will subtract the frequencies of two additive primaries from the white light, thus reflecting the remaining additive primary color. Therefore, the mix of *cyan* and *magenta* will subtract *red* and *green* from the daylight and reflect the only frequencies of *blue*. Analogously for the other combinations. Consequently, the mixing of all three subtractive primary inks will subtract all the frequencies from daylight, thus giving *black*.

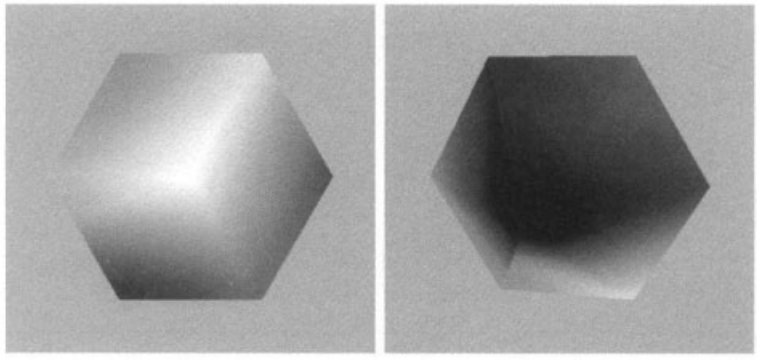


Figure 10.36 VRML color rendering: (a) object `HSVprism` generated in 6D space by `PLaSM` (b) view from below

The corresponding CMY color model is a cube analogous to the RGB model. But in this case we have $C = (1,0,0)$, $M = (0,1,0)$ and $Y = (0,0,1)$. Black and white clearly are exchanged: $W = (0,0,0)$ e $Black = (1,1,1)$. The transformation between the two color cubes is affine, and corresponds to a reflection with respect to the three coordinate planes, followed by a translation which moves the point $(-1,-1,-1)$ to the origin. Thus, in homogenous coordinates we have:

$$\begin{pmatrix} C \\ M \\ Y \\ 1 \end{pmatrix} = \boldsymbol{T}(1,1,1)\ \boldsymbol{S}(-1,-1,-1) \begin{pmatrix} R \\ G \\ B \\ 1 \end{pmatrix}. \tag{10.12}$$

And, in non-homogeneous coordinates:

$$\begin{pmatrix} C \\ M \\ Y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} R \\ G \\ B \end{pmatrix}.$$

The inverse transformation from CMY to RGB is consequently derived:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} C \\ M \\ Y \end{pmatrix}.$$

Such transformations between color models are clearly very easy to write in `PLaSM`. For this purpose we just have to write two lines of code, as given in Script 10.5.3. Notice that, according to textscvrml documentation, some virtual reality browsers, such as e.g. Cosmo Player, do not like the scaling transformations with negative coefficients, so that the result of the visualization is partly unpredictable, or better depends on the browser.

10.6 Shading models

In most practical cases the light intensity in a point of a visible surface is not computed for each pixel using the appropriate illumination model, but is computed only for a

Script 10.5.3 (CMY cube)

```
DEF CMYcube = (T:<1,2,3>:<1,1,1> ~ S:<1,2,3>:<-1,-1,-1>):RGBcube;

VRML:CMYcube:'out.wrl';
```

suitable subset of points. Such points usually correspond either to the vertices of the surface (polygon) or to those of some triangulation of its interior. The computation of the intensities in other points is then done by convex interpolation of known values. This process is called either *color shading* or *normal shading* depending on the subject of the interpolation.

Intensity interpolation

For each RGB color component, the values on vertices of a triangle are interpolated in the discrete set of internal points, i.e. in the internal pixels. Such a method is called *Gouraud's shading.*

Let us consider as known the primary intensities I_a, I_b, I_c of a color component given on vertices $\boldsymbol{a}, \boldsymbol{b}$ and $\boldsymbol{c}$ of a triangle, as shown in Figure 10.38a. The intensity I_p of a point $\boldsymbol{p}$ on the horizontal line between two points $\boldsymbol{r}$ and $\boldsymbol{s}$ on the triangle boundary can be computed as:

$$I(\boldsymbol{p}) = (1-\gamma)I(\boldsymbol{r}) + \gamma I(\boldsymbol{s}), \qquad 0 \leq \gamma \leq 1, \tag{10.13}$$

where

$$I(\boldsymbol{r}) = (1-\alpha)I_a + \alpha I_b, \qquad 0 \leq \alpha \leq 1, \tag{10.14}$$

$$I(\boldsymbol{s}) = (1-\beta)I_a + \beta I_c, \qquad 0 \leq \beta \leq 1. \tag{10.15}$$

Example 10.6.1 (Color shading)
In Script 10.6.1 two colored triangles are generated and exported as VRML using color per vertex representation. Their images as rendered by *Cosmo Player*© are shown in Figure 10.37. Notice that the centroid of the first triangle does not give the *white*, and the centroid of the second one does not give the *black.* Is something wrong here? The answer is no. The matter here is not the additive or subtractive theory, but the color shading. The interpolated RGB values for the two centroids are $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ and $(\frac{2}{3}, \frac{2}{3}, \frac{2}{3})$, i.e. darker and lighter gray respectively, according to Figure 10.37.

Normal-vector interpolation

We have seen the interpolation of color intensity of vertices, computed on those points by using a suitable illumination model. If, conversely, the interpolated entities are the normal vectors of surfaces computed on the vertices, then it is possible to use the illumination models pointwise, by using the appropriate model and the interpolated values of the normals. Such interpolation of normals per vertex, followed by a local use of illumination models to compute the lighting cnd color intensities, is called *Phong's shading model.*

Script 10.6.1 (Color shading)

```
DEF color_triangle (c11,c12,c13, c21,c22,c23, c31,c32,c33::IsReal) =
MKPOL:<<
   < 0,0, c11,c12,c13 >,
   < 1,0, c21,c22,c23 >,
   < COS:(PI/3),SIN:(PI/3), c31,c32,c33 >
>,<1..3>,<<1>>>;

DEF RGB_triangle = color_triangle:< 1,0,0, 0,1,0, 0,0,1 >;
DEF CMY_triangle = color_triangle:< 0,1,1, 1,0,1, 1,1,0 >;

VRML:RGB_triangle:'out.wrl';
VRML:CMY_triangle:'out.wrl';
```

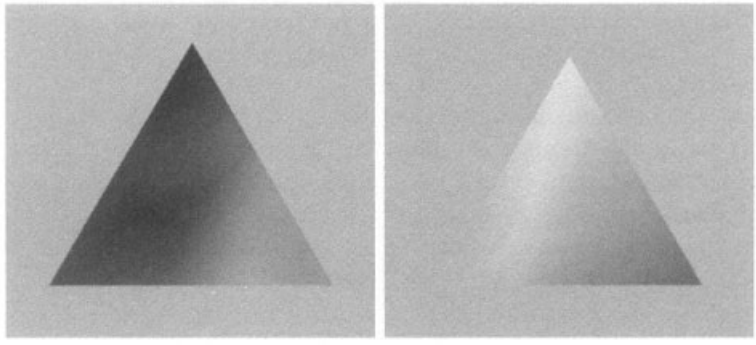


Figure 10.37 VRML rendering of two color shaded `RGB_triangle` and `CMY_triangle`

In this case some vector equations similar to equations (10.13–10.15) are used to compute the components of normal unit vector in the $\boldsymbol{p}$ point:

$$\boldsymbol{n}(\boldsymbol{p}) = (1-\gamma)\boldsymbol{n}(\boldsymbol{r}) + \gamma\boldsymbol{n}(\boldsymbol{s}), \qquad 0 \leq \gamma \leq 1. \tag{10.16}$$

ove

$$\boldsymbol{n}(\boldsymbol{r}) = (1-\alpha)\boldsymbol{n}(\mathbf{1}) + \alpha\boldsymbol{n}(\mathbf{2}), \qquad 0 \leq \alpha \leq 1. \tag{10.17}$$

$$\boldsymbol{n}(\boldsymbol{s}) = (1-\beta)\boldsymbol{n}(\mathbf{1}) + \beta\boldsymbol{n}(\mathbf{3}), \qquad 0 \leq \beta \leq 1. \tag{10.18}$$

Clearly, the illumination point must be applied pointwise to compute the light intensity in each point. As we can see, Phong's shading is much more realistic by its precise rendering of specular reflection effects, but it is also considerably more computationally intensive.

Example 10.6.2 (Normal per vertex)
Gouraud's shading is used in VRML rendering when the user specifies a normal vector for each model vertex. In Script 10.6.2 we generate a polyhedral approximation of the sphere by the function `Sphere_with_normals`. It may be interesting to notice that the dimension-independent of the language is exploited to accommodate the components of the normal as added coordinates of each vertex. Notice on this point the dimensions of the generated object, which is a 2-manifold embedded in 6D space:

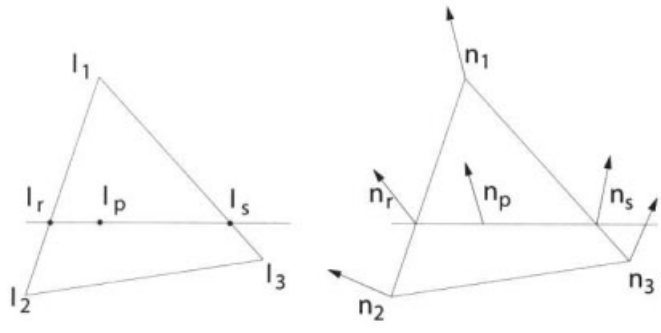


Figure 10.38 Bilinear interpolation: (a) Gouraud (b) Phong

```
Sphere_with_normals:1:<12,24> ≡ A-Polyhedral-Complex{2,6}
```

The VRML files exported by the two last expressions of Script 10.6.2 are shown in Figures 10.39a and 10.39b, respectively.

Script 10.6.2 (Color shading)

```
DEF Sphere_with_normals (radius::IsRealPos)(n,m::IsIntPos)
   = MAP:[fx,fy,fz, fx,fy,fz]:domain
WHERE
   fx = K:radius * - ~ SIN ~ S2 * COS ~ S1,
   fy = K:radius * COS ~ S1 * COS ~ S2,
   fz = K:radius * SIN ~ S1,
   domain = dom1D:<PI/-2,PI/2>:n * dom1D:<0,2*PI>:m
END;

VRML:(Sphere_with_normals:1:<12,24>):'out.wrl';
VRML:(MAP:[s1,s2,s3,K:-1,K:-1,K:-1,s1,s2,s3]:
   (Sphere_with_normals:1:<12,24>)):'out.wrl';
```

Example 10.6.3 (Crease angle)

Gouraud's shading of model surfaces with average normal per vertex can be easily generated by PLaSM programs, without actually generating the normal vectors, but using the VRML attribute relative to the *crease angle*, as shown by Script 10.6.3. In this case the `torus` function given in Script 5.2.13 with minor and major radiuses 1 and 3 is mapped on the domain $[0, 2\pi]^2$, thus giving the geometric value associated with `myTorus` symbol.

The last two expressions of Script 10.6.3 produce the objects displayed in Figures 10.40a and 10.40b, respectively. It is interesting to note that the polyhedral approximation of the true surface is done at the same resolution in both cases!

Example 10.6.4 (Color sphere)

In this example we export a unit 3D sphere embedded in 6D space. A 3D polyhedral approximation of such model is generated by the function `Sphere` given

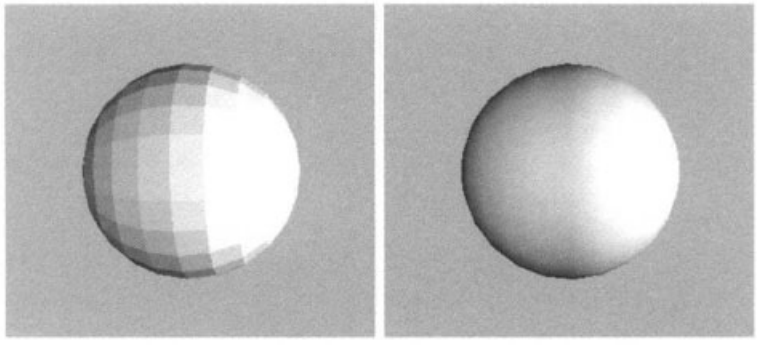


Figure 10.39 VRML rendering of spheres *at the same resolution*, without and with Gouraud's shading

Script 10.6.3 (Color shading)

```
DEF myTorus = MAP:(CONS:(torus:<1,3>)):(dom2D:<0,0,2*PI,2*PI>:<12,24>)

VRML:myTorus:'out.wrl';
VRML:(myTorus CREASE (PI/2)):'out.wrl';
```

in Script 10.6.4. The added coordinates are here used as RGB coordinates. This effect is obtained by suitably mapping the vector extractor function `[S4,S5,S6]` in Script 10.6.4.

The boundary polygons of the exported sphere are Gouraud shaded, using implicitly computed normal per vertex, because of the final invocation of the **CREASE** function. Figure 10.41 shows what happens when browsing the exsported VRML file with *Cosmo Player*©.

Script 10.6.4 (Color sphere)

```
DEF vect = [S1,S2,S3];
DEF out = MAP:(CAT ~ [vect,vect]):(Sphere:1:<12,24>)
VRML:(out CREASE (PI/2)):'out.wrl';
```

10.7 VRML rendering

As we have already seen in the previous examples, the rendering mechanism provided by VRML may be quite refined, since the language allows modeling of both point-shaped and directional lights with varying color and intensity, as well as the material properties of surfaces to be modeled quite carefully. Furthermore, the VRML language allows texture-mapping of 2D images on the 3D surfaces of the scene, thus greatly enhancing the realism of visual rendering. The present section is hence devoted to describing how lights, colors and textures may be specified in VRML, within the theoretical framework we discussed in the previous sections.

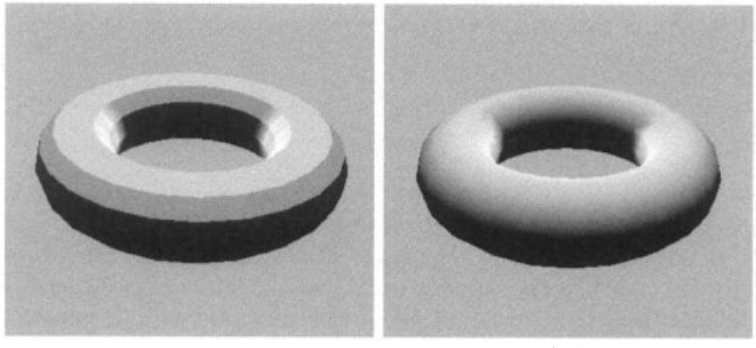


Figure 10.40 VRML rendering of toruses *at the same resolution*, without and with Gouraud's shading

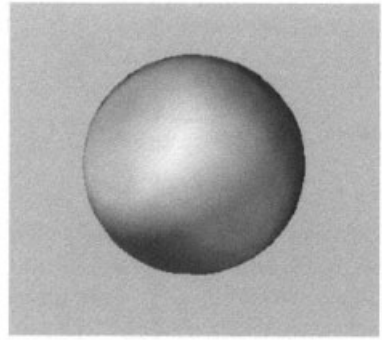


Figure 10.41 VRML shading of the unit spere, using the normals as color values

10.7.1 Illumination

First of all, let us remember that VRML describes the scene as a hierarchical graph, where nodes may describe geometry, grouping, transformations, lights, textures, sounds, videos and so on. As a general rule, let us remember that each node applies its effect on the subgraph rooted in it.

An illumination node describes how the subscene rooted in it should be illuminated. Such a kind of node, in particular, specifies the position and orientation of the light source, the light color and other lighting characteristic, such as the contribution of the source to the diffuse lighting of the ambient. Conversely, a VRML illumination node does not specify a geometric shape of light source. If necessary, a Shape node can be assigned to the source with a suitable geometry and a high value of the field emissiveColor in the Material node of the appearance field.

VRML lights do not produce *shadows* in the scene, but only specify the characteristics of the radiation which would incise on the scene surfaces, as if they were isolated. If shadow rendering is needed, in order to give an appropriate level of realism to the scene, shadows can be simulated by using colors and/or textures. The simulation of shadows can be very refined, if performed according to suitable HSR computations, polygon fragmentation and possibly according to global radiosity algorithms, which may generate the detailed geometric data.

Lighting nodes In particular, three types of *lighting nodes* are available:

1. node DirectionalLight;
2. node PointLight;
3. node SpotLight.

The *scope* of a light of type PointLight or SpotLight is spherical and specified by a field radius of SFFloat type. All the geometry outside the scope of a light is not illuminated by it. Conversely, a node of type DirectionalLight has a hierarchical scope: it will work only on nodes in the same group (i.e. with the same father node) and on their descendants. In other words, a DirectionalLight node will illuminate only the hierarchical subgraph rooted on its father.

Common fields The three illumination nodes have some common fields, whose types and default values are given in the following.

```
on                  TRUE    # SFBool
intensity           1       # SFFloat
ambientIntensity    0       # SFFloat
color               1 1 1   # SFcolor
```

The Boolean field on modifies the status of the light source. This value may be modified at run time, by sending appropriate events to the node. The field color of RGB value defines the light color emitted from the source. It interacts with Material nodes to determine the color aspect of sources hit by light. The fields ambientIntensity and intensity have a real value between 0 and 1. The scalar field intensity is a scaling factor which multiplies the three color components to define the three intensity components of the light source. The product of color components times both the intensity fields is summated to color components of ambient light.

Attenuation An attenuation field is used by PointLight and SpotLight nodes. It is a single valued real field between 0 and 1 used to multiply the intensity field as well as the distance of the considered point from the light source. The field attenuationField is conversely of SFVec3f type, i.e. is a single field with a 3D vector of reals. The first term is used as a multiplier for constant attenuation; it is functionally equivalent to the K addendum at the denominator in the illumination model of equation (10.9). Second and third components are used for linear and quadratic attenuation, respectively.

DirectionalLight The DirectionalLight node is used to define point-shaped sources which project parallel light rays, coming from the point at infinity of the direction vector. There is no attenuation field for directional lights, since it is not possible to compute the distance of a point from the light source. The default definition is the following:

```
DirectionalLight {
    on                  TRUE    # SFBool
    intensity           1       # SFFloat
    ambientIntensity    0       # SFFloat
```

```
    color               1 1 1   # SFcolor
    direction           0 0 -1  # SFVec3f
}
```

The scope of a directional source is hierarchical. The source illuminates all the "brother" nodes and the subgraphs rooted in them. It can be used, e.g. to open a light in a room, where the adjacent rooms must remain in the dark. Not every browser supports a hierarchical scope for lights. In order to maximize the portability of models it is better to use directional lights at the root level of a VRML world.

PointLight The PointLight node is used to define non-directional and point-shaped light sources. As for the other lighting nodes, it is defined in local coordinates by the location field, which is affected by the action of the current transformation matrix at the traversal of hierarchical scene graph.

```
PointLight {
    on                  TRUE    # SFBool
    intensity           1       # SFFloat
    ambientIntensity    0       # SFFloat
    color               1 1 1   # SFcolor
    location            0 0 0   # SFVec3f
    radius              100     # SFFloat
    attenuation         1 0 0   # SFVec3f
}
```

SpotLight The SpotLight node is used to define point-shaped light sources with a preferred direction of light and an action cone. This is defined around the axis of the direction vector by two angles beamWidth and cutOffAngle given in radians. The source is assumed to emit at maximum intensity within the angle beamWidth, and is also assumed not to emit ouside the angle cutOffAngle. The types and default values of the node fields are the following:

```
SpotLight {
    on                  TRUE    # SFBool
    intensity           1       # SFFloat
    ambientIntensity    0       # SFFloat
    color               1 1 1   # SFcolor
    location            0 0 0   # SFVec3f
    radius              100     # SFFloat
    attenuation         1 0 0   # SFVec3f

    direction           0 0 -1  # SFVec3f
    beamWidth           1.5707  # SFFloat
    cutOffAngle         0.7853  # SFFloat
}
```

10.7.2 Color

The VRML language uses the the RGB color model to describe colors. Thus, when some node field has a value of type SFColor (Single Field Color), it must hold three real components in $[0, 1]$, to be interpreted as a point in the normalized RGB color cube. In other words, fields of SFColor type must be specified as normalized RGB triples.

Color information can be used both to assign a *global color* to some primitive or subgraph and to give a *local color* to some portion of a geometric primitive, respectively. The use of Color nodes is hence allowed within:

1. Material node;
2. DirectionalLight, PointLight and SpotLight nodes;
3. IndexedFaceSet, IndexedLineSet, PointSet and ElevationGrid nodes.

Material The Material node defines the material properties of surfaces in geometrical nodes associated with it. In particular, the diffuseColor field defines the diffusion constant K_d for each of the three primary color components, the field specularColor defines the three constants of specular reflection K_s. The field emissiveColor is used to specify the color of light emitted from a luminous body. It can be useful to simulate the results of a radiosity computation, where some surfaces — e.g. the panes of a window — are considered as emitting light.

The shininess field, normalized between 0 and 1, has a meaning similar to the n coefficient of cos of the angle between reflection and viewing directions in equation (10.8). Not all browsers actually support some partial transparency, as specified in the VRML document.

The various possible fields of `Material` node, their types and the default values are as follows:

```
Material {
    diffuseColor     0.8 0.8 0.8 #SFcolor
    ambientIntensity 0.2         #SFFloat
    specularColor    0   0   0   #SFcolor
    emissiveColor    0   0   0   #SFcolor
    shininess        0.2         #SFFloat
    transparency     0           #SFFloat
}
```

Color per face In geometric primitives of type IndexedFaceSet and ElevationGrid it is possible to specify a field color of type SFnode bound to a Color node value, used to specify a set of colors as RGB triples:

```
Color {
    color   [
        0.8 0.8 0.8 ,   # 1
        0   0   1   ,   # 2
       .. .. .. .. .. .. ..
    ]
```

}

If in the geometric primitive the field colorPerVertex is FALSE, then such colors are used as *face colors*. Actually two different methods are available to make the association between faces and colors:

1. By specifying so many colors as many faces. In this case an explicit association is not necessary. The first face will be associated with the first color, the second face with the second color, and so on.
2. By specifying any number of colors, usually less than the number of faces. In this case the face colors must be explicitly given by using the field colorIndex, which contains a sequence of integer references to the values in the Color node, indexed by faces.

Color per vertex The colors to use with geometric primitives IndexedFaceSet and ElevationGrid can be specified in greater detail. Such a detailed specification is used when the field colorPerVertex maintains the default value TRUE. In this case the set of polygons is rasterized by using the Gouraud's shading method. Three types of association of colors with vertices are possible here:

1. By giving as many colors as there are vertices specified in the field point of node Coordinate of field coord. In this case the explicit association between vertices and colors is not needed.
2. By giving colors of vertices as indices in field colorIndex, with reference to the triples contained in the Color node. Such references orderly correspond to vertex numbers.
3. As above, but with color references in colorIndex organized by faces. In this case there are as many lists in field colorIndex as there are faces. Each list will contain the reference to the color counterclockwise associated with the vertices of the associated face, and is terminated by an element with -1 value.

Colored lines and points Both specification and rendering of colors with primitives IndexedLineSet and PointSet are absolutely similar to what as already been discussed for the IndexedFaceSet and ElevationGrid primitives. Gouraud's shading is used when colorPerVertex is TRUE (default). If colorPerVertex is FALSE, then a single color is used for each line, with color values assigned by using any one of the two specification methods seen for faces.

10.7.3 Shading

Without any light specification, the color of faces is assumed to be exactly equal to the one specified in their Color node. Also, without any Material specification, no lighting computations are executed at all. Conversely, if the geometric node is subject to the action of some lighting node, then the normal vectors of the faces are automatically generated and used to compute the face intensities, by using for this purpose the reflectance information stored in the Material node.

The default shading method, used when only colors per face are specified, is the *flat shading*. When specifying both normals and colors per face, a flat shading is again

used. By giving normals per face and colors per vertices the *Gouraud's shading* is applied. Gouraud's shading is also used when normal per vertices are given, because Phong's shading was considered to be too computationally expensive for use with VRML browsers.

Normals per face In nodes IndexedFaceSet and ElevationGrid the user may give a normal field of type SFnode, with value:

```
Normal {
    vector   [
       0.267 0.535 0.801 ,   # 1
       0     0     1     ,   # 2
         .. .. .. .. .. .. ..
    ]
}
```

Notice from previous example that normals should always be defined as unit vectors, i.e. as vectors with unit length. The node IndexesFaceSet has some specialized fields for normals, i.e.: normal, normalIndex and normalPerVertex. The normalIndex field is not acceptable within the ElevationGrid node.

Vectors in a Normal node are paired with faces if the normalPerVertex field is set to FALSE. Such normals are used for lighting computations if there exists a Material node acting on the geometric primitive. The methods that we have already seen to associate colors and faces may be used to pairwise associate normals and faces. In particular:

1. the simplest but also more verbose method consists in giving as many normals as there are faces. Remember that faces may be specified using the field coordIndex.
2. the more efficient method consists in giving the normals, using the field normalIndex, as references to the values in the Normal node.

Shading with normals per vertex As we already said, normal vectors can be associated with vertex points. In this case the intensities of primary colors are computed on each vertex by using a complete reflectance model and the material properties. Such intensities are then interpolated on each polygon by using Gouraud's shading.

The same vertex may even have several normals associated with it. In particular, there can be as many normals on a vertex as there are faces incident on it. This association requires that the normalPerVertex field is set to the default value TRUE. As for the colors per vertex, there are three methods to specify the normals per vertex, by respectively giving:

1. the *same number* of normals and vertices;
2. the normals in the field normalIndex as references, indexed *on vertices*, to values in Normal node;
3. the normals in normalIndex as references, indexed *on faces*, to values in Normal node. In this case each list of normals in normalIndex must be terminated by -1, and must contain a reference to a normal vector for each vertex of the counter-clockwise oriented face.

10.7.4 Textures

The realism of 3D rendering may be greatly enhanced by using the texture mapping, a graphics technique available in the past only in top-level graphics workstations. Conversely, in the last few years graphics accelerators are common on the PC market with hardware support for advanced operations, including texture mapping, at the cost of few hundred dollars.

A Texture Map is a 2D raster image to be mapped on a 3D surface, and therefore subject to be processed in the 3D pipeline consistently with the supporting geometric data. In VRML 2.0 a texturing node allows specification of:

1. the texture, i.e. the raster picture (image) to map on the 3D surface;
2. the texture transformation, with a TextureTransform node;
3. some mapping rules and, in a certain sense, the type of mapping algorithm.

In VRML it is possible to choose between three types of textures, respectively associated with different types of nodes:

1. The ImageTexture node contains as attribute the url of either a JPEG or a PNG image file, so that it allows a photograph to be mapped on a surface. Such a *texture* may be generated in any way, even by using a scanner on the surface of a solid material. It becomes thus possible to visually simulate with extreme realism the surface aspect of solid models, and also to emulate geometric details that would be too expensive to model exactly.
2. The MovieTexture node contains the url of a MPEG-1 file, thus allowing mapping a movie on some surface of the scene, possibly with synchronized sound. In such a case a Sound node should be simultaneously used.
3. The PixelTexture node contains an explicit texture coding with hexadecimal values.

Mapping algorithm Every geometric primitive, i.e. the Cube, Cylinder, Cone, IndexedFaceSet and ElevationGrid nodes, requires a specific *default algorithm* to map the texture on the surface of the primitive.

A mapping different from the default one may be specified by giving two corresponding sets of points on the 2D texture and on the 3D target surface. For this purpose, both the texture and the surface are triangulated using the corresponding points, and a piecewise affine map (i.e. a *simplicial map* — see Section 2.2.2) is accordingly built, that maps each texture triangle into the corresponding surface triangle.

Texture components A texture is specified as an image defined in an s, t bidimensional space, usually coincident with $[0, 1]^2$. There are four types of VRML textures:

1. With one component: the texture contains only intensity values. The only PNG file format is allowed.
2. With two components: there are both intensity and transparency values. The only PNG format is allowed.

3. With three components: the texture contains RGB values. Either JPEG or PNG or GIF file formats are allowed.
4. With four components: both RGB and transparency values are stored in this case. Either PNG or GIF formats are allowed.

10.7.5 PLaSM *lighting*

PLaSM makes direct reference to the VRML lighting model. In particular, it provides *point*, *directional* and *spot* light sources. The LIGHT binary operator must be used for this purpose, and applied to a *polyhedral complex* and to a PLaSM *object* of GenericLight type.

An object of this type is generated by applying the GenericLight function to a triplet < type, appearance, geometry >, where type $\in$ { 0,1,2 } stands for *pointSource*, *directionalSource* and *spotSource*, respectively. The appearance and geometry values are generated by GenericLightAppearance and GenericLightGeometry functions, both built-in the 'colors' library.

A generic light function In Script 10.7.1 we implement a generic light function TheLight, that is able to generate a coloured light source of every type, depending on the actual values of its parameters.

Notice that the parameters of GenericLightAppearance are color, intensity, ambientIntensity and isOn, according to the VRML lighting model. The fields of GenericLightGeometry are location, direction, attenuation, radius, beamWidth and cutOffAngle, also in accordance with the VRML model.

Script 10.7.1 (Generic lights)

```
DEF TheLight(type::isInt)(theColor::TT) =
   GenericLight:< type, appearance, geometry >
WHERE
   appearance = GenericLightAppearance:
      <color,intensity,ambientIntensity,isOn>,
   color = theColor,
   intensity = 1,
   ambientIntensity = 0.4,
   isOn = TRUE,

   geometry = GenericLightGeometry:
      <location,direction,attenuation,radius,beamWidth,cutOffAngle>,
   location = <0,0,0>,
   direction = <1,0,0>,
   attenuation = <1,0,0>,
   radius = 10,
   beamWidth = (PI/4),
   cutOffAngle = (PI/6)
END;
```

Example 10.7.1 (Point, directional and spot lights)
In Script 10.7.2 we produce a visual comparison, shown in Figure 10.42, of the types of light source, by adding either a point, directional or spot `MAGENTA` light to a suitable decomposition of a unit cube. A decomposition of the cube, produced by `grid3D:<10,10,10>:0.1`, is used so that the VRML viewer may produce a better rendering of the illuminated surfaces.

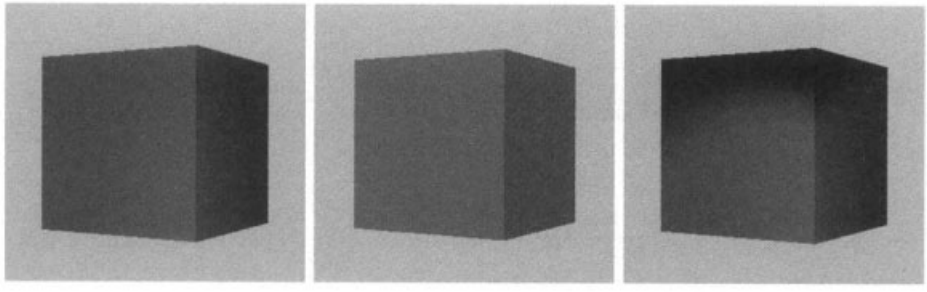


Figure 10.42 Light sources: (a) *point* (b) *directional* (c) spot

Script 10.7.2 (Point, directional and spot lights)

```
DEF cube = (T:1:1 ~ R:<1,2>:(PI/-6) ~ grid3D:<10,10,10> ):0.1;
DEF grid3D (m,n,p::IsIntPos)(a::IsRealPos) =
   ((Q ~ #:m) * (Q ~ #:n) * (Q ~ #:p)):a ;

DEF test1 = CenteredCameras:(cube LIGHT TheLight:0:MAGENTA);
DEF test2 = CenteredCameras:(cube LIGHT TheLight:1:MAGENTA);
DEF test3 = CenteredCameras:(cube LIGHT TheLight:2:MAGENTA);

VRML:test1:'/path/light1.wrl';
VRML:test2:'/path/light2.wrl';
VRML:test3:'/path/light3.wrl';
```

Notice that the three lit cubes `test1`, `test2` and `test3` are generated and exported with the associated *camera* nodes produced by the `CenteredCameras` operator discussed in Section 9.4.1. This operator allowed us to produce the three images of Figure 10.42 from exactly the same viewpoint. Notice also the color attenuation due to distance with point source in Figure 10.42a, whereas no distance attenuation is present with directional source in Figure 10.42b. Finally, notice that we set to *off* the *headlight* automatically set on the viewpoint by the VRML viewer.

Example 10.7.2 (Colored spot lights)
`RED`, `GREEN` and `BLUE` spot lights are associated with a square without material properties in Script 10.7.3. The images produced by a vrml VIEWER, without and with a headlight, are shown in Figure 10.43. For this purpose a `BASESPOTLIGHT` operator is used. Analogous `BASEDIRLIGHT` and `BASEPOINTLIGHT` operators are also available in the `'colors'` library. The unused fields, passed as null values `<>`, are suitably filled by

the library itself.

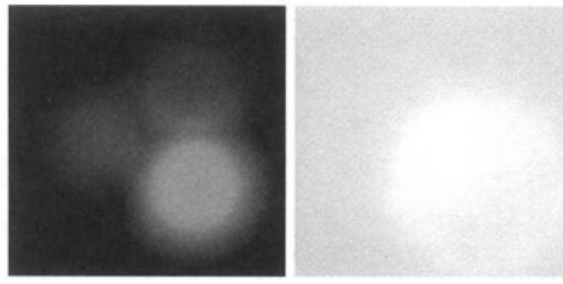


Figure 10.43 Three spot coloured lights: (a) without headlight on the viewer (b) with headlight

Notice that a useful Spot generator is defined in Script 10.7.3, depending only on color, location and orientation parameters. It is *instanced* three times with different colors and locations in our example. Notice also that the LIGHT binary operator is used *infix* between its operands, and remember that binary operators are *left-associative.*

Script 10.7.3 (Spot lights)

```
DEF grid2D (m,n::IsIntPos)(a::IsRealPos) = ((Q ~ #:m) * (Q ~ #:n)):a ;

DEF spot (color,location,orientation::TT) =
   BASESPOTLIGHT:< spotAppearance, spotGeometry >
WHERE
   spotAppearance = < color, <>, <>, <>>,
   spotGeometry = < location, orientation, <>, 100, PI/16, PI/4 >
END;

DEF object = grid2D:<30,30>:1
   LIGHT spot:< RED, <10,15,20>,<0,0,-1>>
   LIGHT spot:< GREEN, <20,10,20>,<0,0,-1>>
   LIGHT spot:< BLUE, <20,20,20>,<0,0,-1>>;

VRML:object:'out.wrl';
```

10.7.6 PLaSM *texturing*

The syntax and semantics of texturing operators used in PLaSM to easily export VRML files are described in this section. The main goal in designing the PLaSM exporting interface was to be as close as possible to the VRML semantics.

In particular we have:

1. CREASE:< *pol*, α > $\equiv$ *pol*
   where α is the lower limit of the angle between adjacent faces of *pol* complex. used as the threshold value for the automatic computation of normals per

vertices;

2. `TEXTURE:<` *pol*, *texture* `>` $\equiv$ *pol*, where

   texture $\equiv$ `< url,`
   `< repeatS, repeatT >,`
   `< translationS, translationT >,`
   `< rotation >,`
   `< scalingS, scalingT >,`
   `< centerS, centerT >>`

   with internal fields equivalent to the ones with the same name defined in VRML nodes `ImageTexture` and `TextureTransform`;

3. `SimpleTexture:`*'url'* $\equiv$ *texture*
   where *'url'* is a string representing a local or web filename with extensions `jpg` or `pnc`.

Example 10.7.3 (Textured Gioconda)
In this example we discuss the mapping of a jpeg image of Leonardo's Mona Lisa portrait (in the Louvre, Paris) over a cylinder, a sphere and a cube, respectively. The PLaSM code which generates the VRML files displayed in Figure 10.44 is given in Script 10.7.4.

First a mapping of "Gioconda" on the `CYLINDER` of radius 1 and height 2, approximated with 12 lateral facets is generated, by using a "crease angle" attribute with $\alpha = \frac{\pi}{2}$.

Notice that both the `CREASE` and the `TEXTURE` operations are binary operators, so that they can be used infix to their operands. Remember also that a multiple infix expression like `arg1 op1 arg2 op2 arg3` is evaluated in leftmost order.

Figure 10.44c is obtained by mapping the jpeg file on the 2-skeleton of the cube. A direct mapping on a 3D object would give a different result, with a more "solid" appearance.

Figure 10.44 The Gioconda's image mapped on cylinder, sphere and cube, respectively

Example 10.7.4 (Textured sun)
A 3D-textured model of the sun is produced in Script 10.7.5 and displayed in Figure 10.45. In this case a unit `Sphere` is used as the target surface of the image

Script 10.7.4

```
VRML:(CYLINDER:<1,2>:18
   CREASE (PI/2) TEXTURE SimpleTexture:'gioconda.jpg'):'out.wrl';

VRML:(Sphere:1:<12,24>
   CREASE (PI/2) TEXTURE SimpleTexture:'gioconda.jpg'):'out.wrl';

VRML:((@2 ~ CUBOID):<1,1,1>
   TEXTURE SimpleTexture:'gioconda.jpg'):'out.wrl';
```

texture contained in `sun.jpg` file. The `Sphere` generating function used here is that given in Script 2.2.7. Notice that the VRML standard mapping algoritm for `IndexedFaceSet` nodes with `creaseAngle` field (quite) correctly maps a circular texture on the two halves of the sphere.

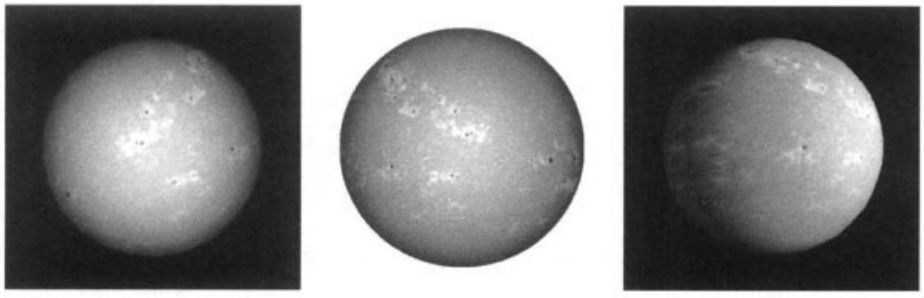


Figure 10.45 (a) and (c) 3D-textured model of the sun (b) 2D texture

Script 10.7.5

```
DEF mySphere = Sphere:1:<24,12>;

VRML:(mySphere CREASE (PI/2) TEXTURE SimpleTexture:'sun.jpg'):'out.wrl';
```

Texture repetition and transformation The rules used for repeating a texture on a surface are not difficult, but are sometimes puzzling. Therefore, we discuss this point with some detail.

First of all, notice that `texture` and `textureTransform` are two fields of the node `Appearance`, of type `ImageTexture` and `TextureTransform`, respectively. The contents of such nodes and their default values follow.

```
appearance Appearance {
  texture ImageTexture {
    url "filename.jpg"
    repeatS FALSE
    repeatT FALSE
  }
```

```
    textureTransform TextureTransform {
      translation 0.0 0.0
      rotation 0.0
      scale 1.0 1.0
      center 0.0 0.0
    }
}
```

Aspect ratio of target surface First, the content of `ImageTexture` file is mapped on the standard unit interval $[0,1]^2$, giving a *normalized texture* in (s,t) space. Then this is mapped onto the containment box of the *target surface.*

Script 10.7.6 (Aspect ratio of target surface)

```
DEF picture1 = CUBOID:<4,4> TEXTURE SimpleTexture:'gioconda.jpg';
DEF picture2 = CUBOID:<3,4> TEXTURE SimpleTexture:'gioconda.jpg';
DEF picture3 = CUBOID:<4,3> TEXTURE SimpleTexture:'gioconda.jpg';
```

Figure 10.46 Mapping of texture on surfaces with different aspect ratios

So, how does one generate a correctly sized model of Leonardo's masterpiece? The correct result is obtained by combining the normalized texture mapping on a square surface with a suitable modeling transformation, in this case a scaling in the y direction:

Script 10.7.7 (3D Mona Lisa)

```
DEF aspectRatio = 404/600;
DEF MonaLisa = S:2:(1/aspectRatio):(CUBOID:<4,4>) * QUOTE:<0.5>
   TEXTURE SimpleTexture:'gioconda.jpg';
```

The `aspectRatio` parameter is the ratio of the number of horizontal pixels of the texture to the number of vertical pixels. The geometric value associated with the

`MonaLisa` symbol is shown in Figure 10.47.

Figure 10.47 The fascinating gaze of Leonardo's Mona Lisa in 3D

Texture mapping on a polygon What happens when mapping a texture on a polygon? The mapping algorithm is substantially unchanged, and may be described as follows:

1. Texture is normalized.
2. Texture orientation and portion to be mapped are chosen with respect to the aspect ratio of the containment box of the target polygon.
3. The normalized texture is accordingly mapped and clipped.

The last point actually corresponds to the user view of the process. In practice, there is no texture clipping: at polygon rasterization time, for each rasterized pixel, a reverse mapping from device coordinates to the normalized texture space is performed, in order to compute the set of *texels* (i.e. texture elements) to map in that pixel, and to compute their averaged color.

The `PLaSM` mapping of Mona Lisa on a 2D `target` polygon is coded in Script 10.7.8. The geometric objects generated by the three last expressions are shown in Figures 10.48 a, b and c, respectively.

Script 10.7.8 (Texture mapping on a polygon)

```
DEF target = triangleStripe:
   <<0,0>,<1,4>,<1.5,2.5>,<3.5,4>,<3,2.5>,<4,0>,<3,1>,<1.5,1>,<1.5,2>>

target ;
target TEXTURE SimpleTexture:'gioconda.jpg' ;
target * QUOTE:<0.5> TEXTURE SimpleTexture:'gioconda.jpg' ;
```

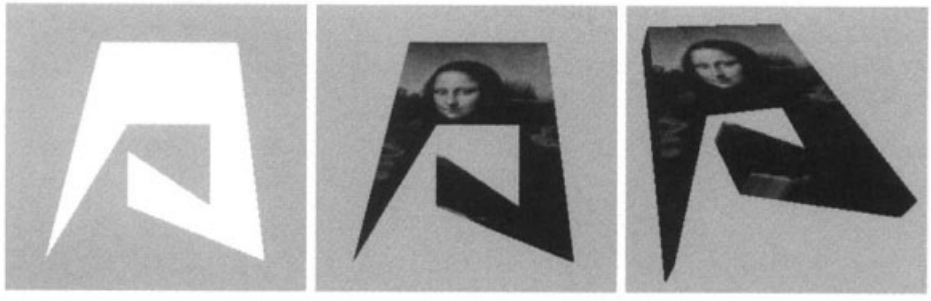


Figure 10.48 (a) Polygon generated by a `triangleStrip` (b) Texture mapping on polygon; (c) polygon extrusion followed by texture mapping

Texture transformations As we have already seen, a 2D transformation $\boldsymbol{X}$ may be applied to the *normalized texture space* s, t. This transformation is composite by scaling and rotation about a fixed center, followed by translation. In formal terms, we may write, for the texture applied to the geometric surface:

$$\boldsymbol{X}(t_s, t_t, \alpha, s_s, s_t, c_s, c_t) = \boldsymbol{T}(t_s, t_t)\ \boldsymbol{T}(c_s, c_t)\ \boldsymbol{R}(\alpha)\ \boldsymbol{S}(s_s, s_t)\ \boldsymbol{T}(-c_s, -c_t)$$

where (c_s, c_t) is the transformation center, i.e. the fixed point, (s_s, s_t) are the scaling parameters, α is the rotation angle, and (t_s, t_t) is the final translation.

It is important to understand that the fields of the VRML `TextureTransform` node actually contain the parameters of the inverse transformation $\boldsymbol{X}^{-1}$. This is the true reason for the odd behavior of the `TextureTransform` VRML node, which makes its correct usage very difficult for the naïve user.

To give the correct values to the VRML `TextureTransform` node, it is actually very easy, by remembering that the inverse of a scaling tensor has reciprocal parameters, and the inverse of rotation and translation tensors have opposite parameters, according to the discussion in Section 6.2.8. Thus, within the `'colors'` library we have the following settings:

```
< centerS, centerT > = < -c_s, -c_t >
rotation = -α ,
< scalingS, scalingT > = < 1/s_s, 1/s_t >,
< translationS, translationT > = < -t_s, -t_t >,
```

and

```
repeatS, repeatT ∈ { FALSE, TRUE } .
```

At this point it is easy to understand how the transformed textures of Figure 10.49 were produced by the VRML viewer.

1. The first texture has a rotation of $\alpha = -\frac{\pi}{4}$ around the center $(c_s, c_t) = (0.5, 0.5)$ of normalized texture space, and no repetition.
2. The second one has same rotation and center, with a further scaling $(c_s, c_t) = (\frac{1}{2}, \frac{1}{2})$, and no repetition.
3. The third texture has the same transformation parameters of the previous one, but the repetition is activated in both coordinate directions.

The defining `PLaSM` code is given in Script 10.7.9.

Script 10.7.9 (Texture transformations)

```
DEF out1 = CUBOID:<1,1> TEXTURE FullTexture:<'gioconda.jpg', FALSE, FALSE,
   <0.5, 0.5>, PI/-4, <1,1>, <0, 0>>;

DEF out2 = CUBOID:<4,4> TEXTURE FullTexture:<'gioconda.jpg', FALSE, FALSE,
   <0.5, 0.5>, PI/-4, <1/2,1/2>, <0, 0>>;

DEF out3 = CUBOID:<4,4> TEXTURE FullTexture:<'gioconda.jpg', TRUE, TRUE,
   <0.5, 0.5>, PI/-4, <1/2,1/2>, <0, 0>>;
```

Figure 10.49 (a) Texture rotation with center of normalized space as fixed point (b) Rotation and scaling (c) Rotation and scaling with $repeatS = repeatT =$ TRUE

Texture mapping with repetition The texture may be repeated on the target surface by giving in VRML a TRUE value to Boolean fields `repeatS` and `repeatT`.

With the VRML approach to texture transformation previously described, the `scalingS` and `scalingT` fields of `TextureTransform` node have the role of *repetition parameters*, since their reciprocal values give the number of columns and rows, respectively, in the array of repeated texture instances within the normalized texture space.

In Script 10.7.10 we show how to obtain a repeated texture with 3×2 and 2×3 undeformed image instances, respectively. Both the results and the intermediate reasoning are displayed in Figure 10.50, which is produced by the STRUCT expression of the script. The `aspectRatio` parameter for the Mona Lisa's texture was defined in Script 10.7.7.

Script 10.7.10 (Texture mapping with repetition)

```
DEF repeatedTexture (scaleS,scaleT::IsReal) = CUBOID:<1,1> TEXTURE
   FullTexture:<'gioconda.jpg',TRUE,TRUE,<0,0>,0,<scaleS,scaleT>,<0,0>>;

DEF out = STRUCT:<
   repeatedTexture:<1/3,1/2>, T:1:1.2,
   repeatedTexture:<1/2,1/3>, T:1:1.2,
   S:2:(1/aspectRatio):(repeatedTexture:<1/2,1/3>), T:1:1.2,
   S:<1,2>:<aspectRatio, 1/aspectRatio>:(repeatedTexture:<1/2,1/3>)
>;
```

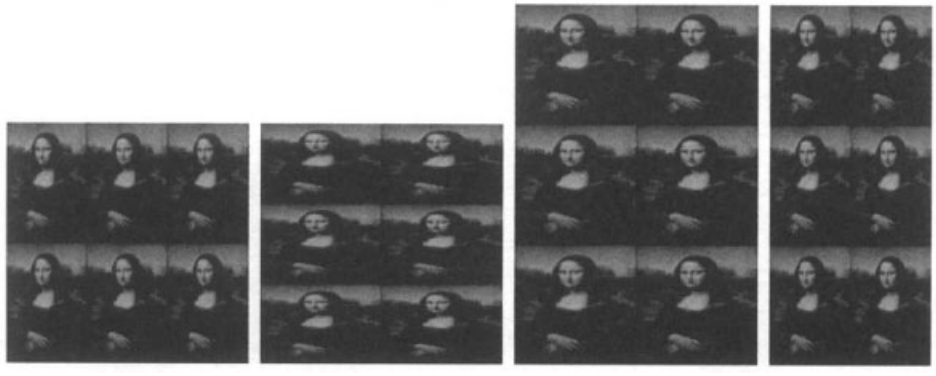


Figure 10.50 Texture mapping with repetition

10.8 Examples

10.8.1 Orthogonal projection on any viewplane

We want to compute the view model that produces an orthogonal projection on the plane of the Cartesian equation $ax + by + cz + d = 0$. The solution is straightforward:

$$\text{VPN} = \text{DOP} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}, \qquad \text{VRP} = \mathbf{0}.$$

Any vector value is feasible for VUV, provided that it is neither parallel to VPN nor equal to $\mathbf{0}$. In other words, we must just guarantee that

$$\text{VUV} \times \text{VPN} \neq \mathbf{0}.$$

10.8.2 Model of example house

Script 10.8.1 (Walls and ground model)

```
DEF front = MKPOL:< <<0,0>,<3,0>,<5,0>,<10,0>,<3,5>,<5,5>,
   <6,5>,<9,5>,<0,7>,<5.5,7>,<10,7>,<6,2>,<9,2>>,
   <<1,2,5,9>,<3,12,7,10,6>,<3,4,12,13>,<4,13,8,11>,
   <5,6,9,10>,<7,8,10,11>>, <1..6>>;

DEF side = MKPOL:< <<14,0>,<7,0>,<0,0>,<10,2>,<8,2>,<5,2>,<3,2>,
   <10,5>,<8,5>,<5,5>,<3,5>,<14,7>,<7,7>,<0,7>>,
   <<1,4,8,12>,<1,2,4,5>,<2,5,6,9,10,13>,<2,3,6,7>,
   <3,7,11,14>,<8,9,12,13>,<10,11,13,14>>, <1..7>>;

DEF pattern0 (n::IsInt) = (QUOTE ~ ##:n):<1,-1>;
DEF pattern1 (n::IsInt) = (QUOTE ~ ##:n):<-1,1>;

DEF ground = STRUCT:<
   STRUCT:<pattern0:7 * pattern0:5, pattern1:7 * pattern1:5> COLOR white,
   STRUCT:<pattern1:7 * pattern0:5, pattern0:7 * pattern1:5> COLOR blue >;
```

The PLaSM definition of the simplified house model used in this chapter to show the effects of different view models is given in Scripts 10.8.1 and 10.8.2. The `front` and `side` symbols return 2D polyhedral values corresponding to the house main walls. The `ground` symbol returns the 2D house floor with the checkerboard pattern.

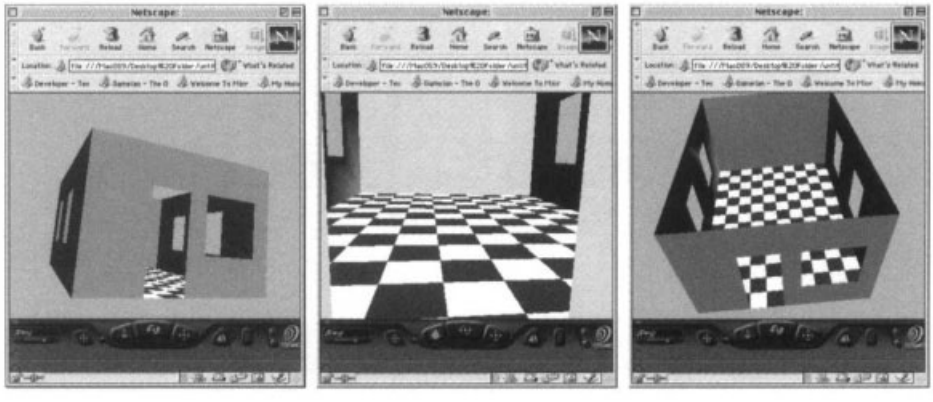


Figure 10.51 Browsing the VRML file produced when exporting the `model` value

Script 10.8.2 (Model of house scene)

```
DEF gold = RGBCOLOR:<0.73,0.6,0.1>;
DEF white = RGBCOLOR:<1,1,1>;
DEF blue = RGBCOLOR:<0,0,1>;

DEF house_model = STRUCT:<
   (T:1:14 ~ R:<1,2>:(PI/2) ~ R:<2,3>:(PI/2) ~ EMBED:1):front,
   (R:<1,2>:(PI/2) ~ R:<2,3>:(PI/2) ~ EMBED:1 ~ CUBOID):<10,7>,
   (T:2:10 ~ R:<2,3>:(PI/2) ~ EMBED:1):side,
   (R:<2,3>:(PI/2) ~ EMBED:1):side,
   EMBED:1:ground > COLOR gold;

house_model;
```

The 3D `house_model` is generated as an assembly of properly embedded and oriented `front`, `side` and `ground` instances. Notice in particular that the `gold` color is applied to the whole `house_model`, but does not modify the subassemblies (such as the `ground`) where specific colors were previously applied.

Some images taken from the screen during the browsing of the VRML model generated when exporting the polyhedral value of the `house_model` symbol are given in Figure 10.51.

It may be interesting to notice that exactly the same result is obtained by substituting the `ground` definition of Script 10.8.1, with the one given in Script 10.8.3. where `pattern.jpg` contains the tiled image shown in Figure 10.52.

Script 10.8.3 (Textured ground)

```
DEF ground = S:<1,2>:<14,10>:(CUBOID:<1,1> TEXTURE
   FullTexture:<'pattern.jpg', TRUE, TRUE, <0, 0>, 0, <7,5>, <0, 0>>) ;
```

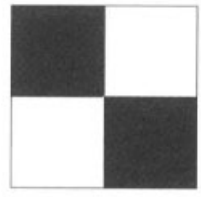


Figure 10.52 Texture mapped on the `ground` of house model

10.8.3 Cell extraction

It can be quite important in various modeling problems to extract the cells of a polyhedral complex, in order to make possible some *ad hoc* handling of individual cells.

Therefore we show in Script 10.8.4 a `SPLIT` function from polyhedral complexes to *sequences* of polyhedral complexes, which takes a complex as input and gives as output the sequence of its convex cells, returned as isolated complexes. This function may be further specialized to extract the cells of the 1-, 2- or 3-skeleton of the input complex, as shown in the script.

The `SplitCells` implementation is quite simple. The input polyhedral `scene` is first decomposed into the `dataset` triplet; its vertices are moved into the `points` object; its `cells` are reconstructed as proper subsets of `points`. Finally, each element in the `cells` sequence is transformed into an individual complex by the combined action of functions `[ID,[INTSTO ~ LEN],K:<<1>>]` and `MKPOL`.

Script 10.8.4 (Convex cells)

```
DEF SplitCells (scene::IsPol) =
   AA:(MKPOL ~ [ID,[INTSTO ~ LEN],K:<<1>>]):cells
WHERE
   cells = ((CONS ~ AA:(CONS ~ AA:SEL) ~ S2):dataset):points,
   points = S1:dataset,
   dataset = UKPOL:scene
END;

DEF extract_wires (scene::IsPol) = (SplitCells ~ @1):scene;
DEF extract_polygons (scene::IsPol) = (SplitCells ~ @2):scene;
DEF extract_bodies (scene::IsPol) = (SplitCells ~ @3):scene;

extract_bodies:(ColRow:4);
extract_polygons:(ColRow:4);
```

The last two rows of Script 10.8.4 produce the geometric assemblies which are shown exploded in Figure 10.53. The generating expression `ColRow:4` of the input complex was presented in Script 2.4.3 while discussing the `Temple` example.

The extraction of the components of a complex may usefully return the polyhedral

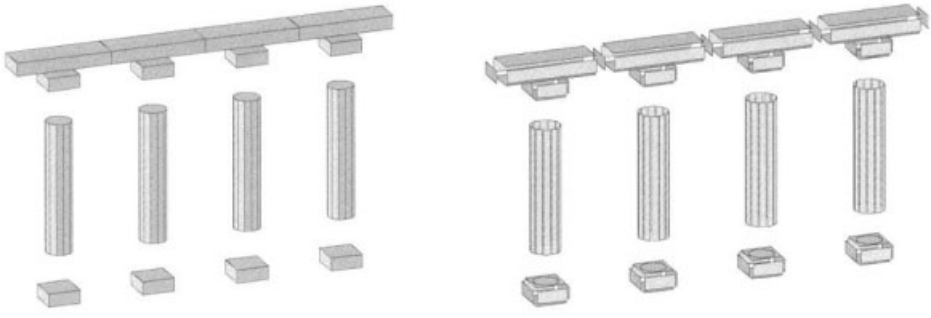


Figure 10.53 (a) z-exploded view of solid cells (b) xyz-exploded view of boundary polygons

cells. For this purpose it is sufficient to slightly modify, as shown in Script 10.8.5, the `SplitCells` operator previously given. Two pictures showing an exploded view of either convex or polyhedral extracted cells from the `hole` object given below is shown in Figure 10.54. The definitions of `triangleStrip` and `Q` operators are given in Scripts 7.2.17 and 6.4.4, respectively.

Script 10.8.5 (Polyhedral cells)

```
DEF SplitPols (scene::IsPol) =
   (AA:(MKPOL ~ [S1,S2,[INTSTO ~ LEN ~ S2]]) ~ DISTL):< points, pols >
WHERE
   points = S1:dataset,
   pols = ((CONS ~ AA:(CONS ~ AA:SEL) ~ S3):dataset):(S2:dataset),
   dataset = UKPOL:scene
END;

DEF hole = (triangleStrip * K:(Q:1)):
   <<0,3>,<1,2>,<3,3>,<2,2>,<3,0>,<2,1>,<0,0>,<1,1>,<0,3>,<1,2>>;

DEF extract_polygons (scene::IsPol) = (SplitPols ~ Q2):scene;
(STRUCT ~ explode:<1.2,1.2,1.5> ~ extract_polygons):hole;
```

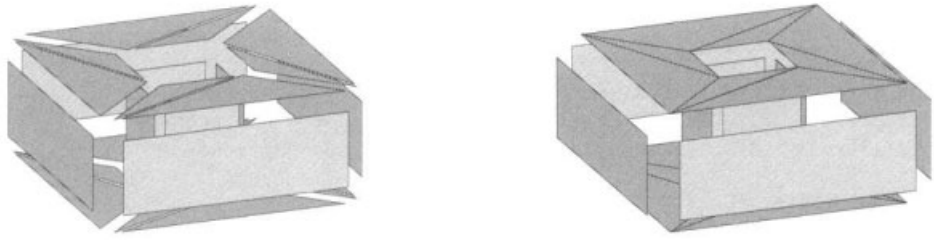


Figure 10.54 xyz-exploded view of boundary: (a) convex cells (b) polyhedral cells

10.8.4 Exploded views

In several applications of mechanical and architectural CAD it may be very useful to produce some *exploded view* of an assembly. The technique to use in this case is quite simple. First, choose a triplet of scaling coefficients $s_x, s_y, s_z \geq 1$ to apply to some assembly points; then, for each object i in the input sequence do the following operations:

1. choose an internal point $\boldsymbol{p}_i$, e.g. the centroid of the containment box;
2. apply a scaling tensor $\boldsymbol{S}_{xyz}(s_x, s_y, s_z)$ to $\boldsymbol{p}_i$, so generating its image $\boldsymbol{p}_i^*$;
3. compute the vector $\boldsymbol{t}_i = \boldsymbol{p}_i^* - \boldsymbol{p}_i = \begin{pmatrix} t_{i_x} & t_{i_y} & t_{i_z} \end{pmatrix}^T$;
4. finally apply the translation tensor $\boldsymbol{T}_{xyz}(t_{i_x}, t_{i_y}, t_{i_z})$ to the object.

In Script 10.8.6 we give, through the function `explode`, an implementation of the algorithm previously discussed. Such a function must be first applied to a triplet `sx,sy,sz` of scaling coefficients. The input `scene` in entered as a sequence of polyhedral complexes. The exploded views produced by the last two expressions of Script 10.8.6 are shown in Figure 10.53. Notice that a unit value of a scaling coefficient produces no mutual translation of parts along the corresponding direction.

The `MK` and `UK` functions, used to transform a point into a 0-complex, are given in Script 3.3.15; the function `vectDiff`, to compute the difference of two points or vectors, is given in Script 3.1.2.

Script 10.8.6 (Exploded view)

```
DEF explode (sx,sy,sz::IsReal) (scene::IsSeqOf:IsPol) =
   (AA:APPLY ~ TRANS):< translations, scene >
WHERE
   scalings = #:(LEN:centers):(S:<1,2,3>:<sx,sy,sz>),
   translVectors = (AA:vectDiff ~ TRANS):< scaledCenters,centers >,
   centers = AA:(MED:<1,2,3>):scene,
   scaledCenters = (AA:(UK ~ APPLY) ~ TRANS):< scalings, AA:MK:centers >,
   translations = AA:(T:<1,2,3>):translVectors
END;

(STRUCT ~ explode:<1,1,1.5> ~ extract_bodies ~ ColRow):4;
(STRUCT ~ explode:<1.2,1.2,1.5> ~ extract_polygons ~ ColRow):4;
```

10.8.5 Standard view models

In Figure 10.55 we show three examples of projections chosen from those discussed in Section 10.2. In particular we show a central (oblique) projection, an orthogonal (isometric) parallel projection and a oblique (cabinet) parallel projection, all generated by PLaSM and exported as Flash files. The PLaSM code is given in Script 10.8.7. A complete listing of the view models associated to standard projection types is given in Script 10.2.1. Clearly, in order to produce different projection it is sufficient to change the name of the view model in the generating line. Let us remember that the semantics of the `flash` exporting operator was discussed in Section 7.

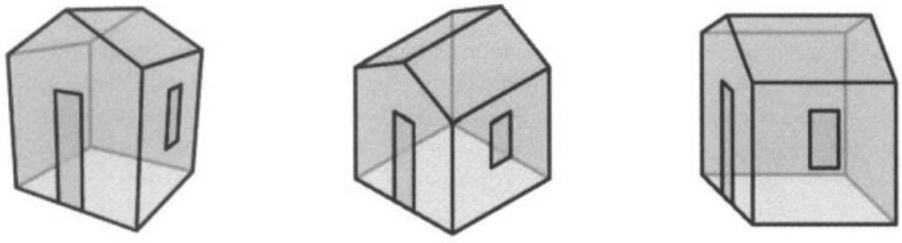


Figure 10.55 Projections exported as Flash files: (a) perspective oblique (b) parallel isometric (c) parallel cabinet

The house model is defined in Script 8.5.21; the FILLCOLOR, LINECOLOR, LINESIZE operators, to be used with FLASH exporting, are discussed in Section 15.3.2.

Script 10.8.7 (View models)

```
DEF house1 = projection: perspective: threepoints: house;
DEF house2 = projection: parallel: isometric: house;
DEF house3 = projection: parallel: cabinet: house;

DEF out (object::IsPol)(name::IsString) = FLASH:(object
   FILLCOLOR RGBAcolor:<0,1,1,0.5>
   LINECOLOR RGBAcolor:<0,0,0,1>
   LINESIZE 5):300:name;

out: house1: 'house1.swf';
out: house2: 'house2.swf';
out: house3: 'house3.swf';
```

10.9 Annotated references

Michael McKenna discussed in [McK87] a $O(n^2)$ worst-case optimal hidden-surface removal algorithm. This was an improvement over the previous best worst-case performance of $O(n^2 \log n)$. It was established that the hidden-line and hidden-surface problems have an $O(n^2)$ worst-case lower bound, so the algorithm is optimal.