The Mathematics of Graphical Transformations:
Vector Geometric and Coordinate-Based Approaches

DesignLab Technical Report DL-1997-03
January 1997
(Current Revision: 22 April 2008)

James R. Miller
Department of Electrical Engineering and Computer Science
University of Kansas
Lawrence, Kansas

The Mathematics of Graphical Transformations:

Vector Geometric and Coordinate-Based Approaches

DesignLab Technical Report DL-1997-03
January 1997
Printed with revisions April 8, 2003

James R. Miller
Department of Electrical Engineering and Computer Science
University of Kansas
Lawrence, Kansas

I. Introduction

If you had a lump of clay and wanted to make a bowl, you would most likely fashion the clay by translating your abstract notions of desired size and shape directly into the appropriate tugs and pulls. You would probably not be thinking too much about coordinate systems.

If you wanted to make a table on which to display your bowl, and then some chairs so people could sit around the table, you would again be thinking at a fairly high level of abstraction about appropriate overall sizes and desired spatial relationships. Even though you would be alternately thinking about how a leg fits on a chair and how each completed chair should be positioned around the table, these thoughts would not be cast in terms of “chair coordinate systems”, “table coordinate systems”, and how one relates to the other.

On the other hand, when we want to use computers to help us design and visualize the objects in our world, we need a formal way to communicate our notions of size and shape to the computer. Obviously this is where coordinate systems come in to play, and we all know how they work.

In fact, coordinate systems work so well in this context that most graphics systems use all sorts of them. Component objects like legs are modeled in so-called *local* coordinate systems. These coordinate systems are then instantiated multiple times inside of a chair local coordinate system; each chair system is placed in the local coordinate system of the table. This process of instantiating an object defined in one local coordinate system inside of another local coordinate system continues until we have defined the entire scene. This “special” final local coordinate system is commonly called the *world* coordinate system.

After having constructed the geometry in this fashion, we establish another coordinate system (often called an *eye coordinate system*) to describe how we want to look at our table and chairs. And we typically use yet other coordinate systems as well: projection coordinate systems, device coordinate systems, and so forth.

Clearly we must become proficient in the use of coordinate systems and comfortable with issues of how they relate to one another.

We will study a common set of such coordinate systems as well as transformations between them, but first a caveat appropriate to the primary theme of these notes: we must remember that coordinate systems *are* merely an artifact of the language we must use to communicate with a computer. That is, our base ideas of size, shape, and orientation are intrinsic properties of the objects we create and are independent of any coordinate system. We want to use coordinate systems where they are appropriate and necessary, but we don’t want our thinking or analytical processes to become unnecessarily dependent upon coordinate systems. This probably doesn’t make much sense just yet, but as you will see while studying the mathematics underlying computer graphics operations, the process of translating our original abstract notions of object manipulation, viewing specifications, and so forth into concrete computer-based actions can become overly complex if we force ourselves to think solely in terms of describing these actions relative to some specific coordinate system.

We describe as *vector geometric* that class of techniques which is based on representing and manipulating intrinsic relationships between objects which are independent of any coordinate system. Examples include computing the centroid of a group of points, or the vector normal to two others. By contrast, *coordinate-based* approaches generally operate by comparing and manipulating individual *x*, *y*, and *z* coordinates of points. For example, a particular algorithm may select one of two points based on whose *z* coordinate is larger. You will come to understand and appreciate these distinctions more completely later. What you will see is that vector geometric techniques are most suitable when operating on objects whose position and orientation with respect to the current coordinate system are completely general, and when no axis of the coordinate system has any special relationship to the current problem of interest. Coordinate-based methods are preferable when we either know *a priori* that the geometry is in a known, simple position and orientation in the coordinate system, or we pre-process it so that it is.

I am assuming here that the reader has experience with the use of one or more graphics packages such as OpenGL, PHIGS, or GKS, and that you are therefore somewhat familiar with the use of coordinate systems as mentioned above. I have taught various combinations of GKS, PHIGS, and OpenGL in introductory graphics courses. While my focus is currently on OpenGL, I prefer from an instructional and conceptual point of view the strict separation of modeling and viewing transformations as formalized in PHIGS to the combined “GL_MODELVIEW” approach of OpenGL. Others share this view. For example, Brodlie says the following about his course at the University of Leeds [Brodlie & Mumford 96]:

> Both PHIGS and OpenGL have their influence on the course: OpenGL for practical exercises, PHIGS for theoretical treatment of the viewing pipeline.

We review in the next section the basic concept of a graphics pipeline and the various coordinate systems we will encounter. The following two sections present required mathematical concepts and tools and discuss the differences between coordinate-based and vector-geometric approaches to analysis. The final sections discuss in turn the implementation of each of the transformations we encounter in the pipeline, treating both OpenGL and the more general PHIGS models.

A set of C++ classes has been developed by the author which include methods and definitions of overloaded operators implementing nearly all of the low-level operations and matrix construction algorithms presented in these notes. Appendices A-D give the primary class definitions. Appendix E shows some sample code using the definitions.

Final Introductory Remarks

While the original emphasis when this monograph was written was to cover the mathematics required to implement graphics pipelines in interactive graphics systems such as OpenGL and PHIGS, the majority of this material is much more widely applicable. The basic low-level mathematical tools related to point and vector operations in section III, for example, is required background material for any work in Geometric Modeling. The tools discussed there are also the primary ones that are needed in other types of graphics processing such as ray tracing. Similarly, the comments in section IV are directly applicable to any sort of geometric processing. Sections V and VI, while somewhat specific to graphics pipeline architectures, actually serve a dual purpose in that they can be viewed as examples of how the tools described in section III can be used.

II. The 3D Graphics Transformation Pipeline

As noted in the introduction, it is common to use many coordinate systems while describing the position, orientation, and size of geometric objects as well as how we want to view and place them on a computer screen. We need to tell the computer many things in order to generate a display, and different pieces of information are simply easier to specify in certain specific coordinate systems. For example, it is easiest to define the geometry of our table leg if we can assume it lies along the *y*-axis with its base at the origin; but it is easiest to define a desired field of view by using a coordinate system in which we are looking along, say, the *z*-axis. In general, these two coordinate systems are different.

Different graphics APIs ("application programming interface") employ slightly different names for these various coordinate systems. For reference and comparison, the table below summarizes the names used by PHIGS [ISO 89, Howard, et. al. 91] and OpenGL [Schreiner, et. al. 08].

	Local Coordinates	World Coordinates	Viewing Coordinates	Homogeneous Coordinates	Logical Device Coordinates	Physical Device Coordinates
OpenGL	Object Coordinates		Eye Coordinates	Clip Coordinates (HNDC)	Normalized Device Coordinates	Window Coordinates
PHIGS	Modeling Coordinates	World Coordinates	View Reference Coordinates	(HNPC)	Normalized Projection Coordinates	Device Coordinates

Table 2.1: *Equivalent Coordinate System Names*

Some remarks on the final three columns of the table are in order. As we will see in section III, we can describe geometry in *affine* (x,y,z) space or *projective* (x,y,z,w) space. Projective coordinates may be used for basic model definition in modeling coordinates, and/or they may be generated internally by the graphics system when applying a perspective transformation. The former represents a common way of defining *rational* curves and surfaces. In OpenGL, for example, one can define rational Bezier curves by passing projective coordinates for control points to *glMap1**. Here we focus primarily on the internal generation of projective coordinates since concepts surrounding the more general definition of models in projective coordinates are beyond the scope of these notes.

It is common to refer to projective coordinates generated internally in this fashion as *homogeneous coordinates*. The internal generation and use of homogeneous coordinates by graphics systems is transparent to application programmers. We will see why, when, and how it occurs when we discuss the graphics pipeline below. We show the abbreviations HNDC (Homogeneous Normalized Device Coordinates) and HNPC (Homogeneous Normalized Projection Coordinates) inside parentheses in Table 2.1 to emphasize the fact that they are not normally used by (or even visible to) application programmers.

Both PHIGS and OpenGL employ the notion of a "logical device". That is, coordinates are initially transformed onto a virtual graphics screen whose coordinate dimensions run from 0 to 1 (PHIGS) or -1 to +1 (OpenGL). In PHIGS, this coordinate system plays a fairly major role in that it appears as a part of the API and allows general screen layouts can be defined. All or part of such a layout can then be mapped to all or part of various physical devices. (See the *Aside* on Workstations in PHIGS below.) In OpenGL, this logical device coordinate system is nearly transparent to application programmers. The only time an OpenGL programmer needs to be aware of the role played by this logical device is if the programmer needs to create a custom projection matrix to be used as the GL_PROJECTION matrix.

The meaning of "Physical Device Coordinates" is obviously device-dependent. For example, when generating output for a non-interactive device (e.g., a plotter or a PostScript file), device coordinates refer to actual positions on the device. A more common scenario is to run interactively on a graphics device on which a window manager is used to allocate portions of the physical screen to independent applications. In this case, the window manager is in charge of allocating space on the physical screen for the window as well as tracking size and visibility changes. The graphics system "runs on top of" the window manager; that is, the only physical device coordinate system the graphics system knows is the region (i.e., the window) assigned to it by the window manager. Therefore, what the graphics system knows as device coordinate (0,0) is actually the corner of the window on the screen, not the corner of the screen itself. (See also the *Aside* on Workstations in PHIGS below.)

Even though we are generally concerned primarily with producing images on a flat (i.e., 2D) display medium, the device coordinate systems in PHIGS and OpenGL are 3D. Among other things, this allows these systems to perform hidden surface removal after all transformations and clipping are done.

Now we turn our attention to the so-called *graphics pipeline*, i.e., the ordered set of steps performed by a graphics system when actually drawing an object on the screen. Internally, graphics systems typically store the geometry exactly as specified by the application programmer: i.e., in modeling coordinates. To generate an image, we need to determine what pixels to illuminate for each geometric object. This process of going from modeling coordinates to device coordinates is described by the graphics pipeline. Figure 2.1 shows the basic steps in the PHIGS pipeline along with the coordinate systems and associated transformations it uses.

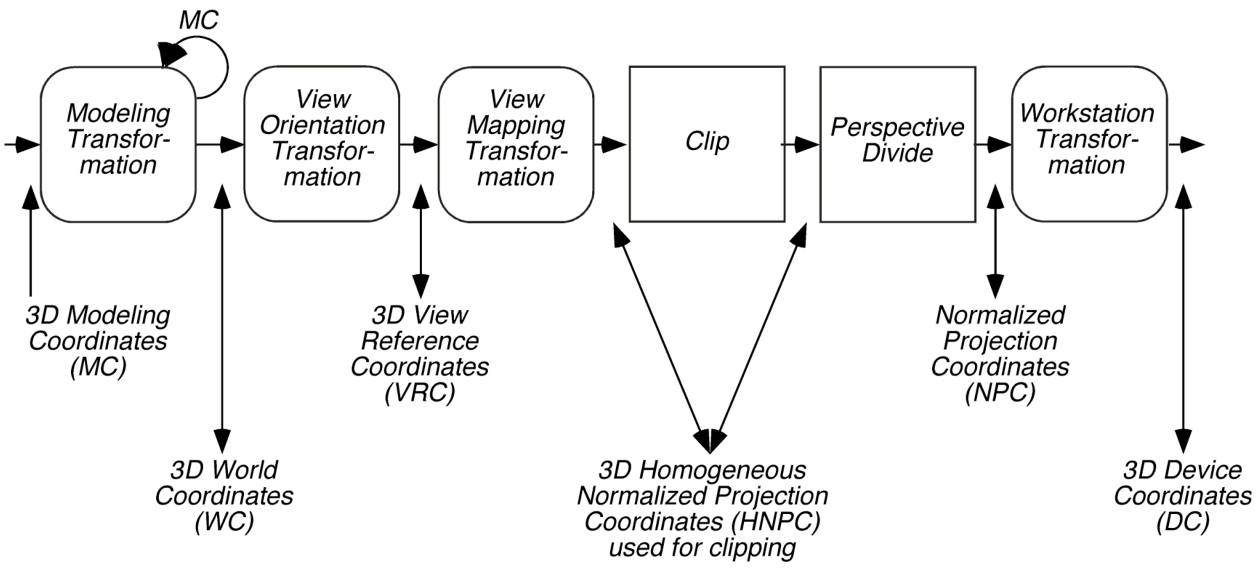

MC
Modeling Transfor-mation
View Orientation Transfor-mation
View Mapping Transfor-mation
Clip
Perspective Divide
Workstation Transfor-mation
3D Modeling Coordinates (MC)
3D World Coordinates (WC)
3D View Reference Coordinates (VRC)
3D Homogeneous Normalized Projection Coordinates (HNPC) used for clipping
Normalized Projection Coordinates (NPC)
3D Device Coordinates (DC)

Figure 2.1: *A PHIGS-like Transformation Pipeline*

Pipelines such as that of Figure 2.1 – as well as the methods used for implementing the various transformations – were designed to optimize our ability to manipulate and render 3D scenes at interactive rates. The geometry itself must be linear (points, lines, and polygons); more general curves and surfaces are approximated by piecewise linear elements before being sent down such a pipeline. In these notes, we use this graphics pipeline as a convenient motivation for the geometric tools we develop. This makes sense since many introductory graphics courses begin by introducing students to graphics packages such as PHIGS or OpenGL in which such a pipeline plays a central role. Other rendering strategies (e.g., ray tracing) are used in applications where interactive rates are not as important as are more sophisticated modeling, lighting, and viewing effects. In Section III, we develop a powerful set of vector geometric tools. While the subsequent discussion and applications of these tools in Sections IV, V, and VI focus on their use in the context of this pipeline, the tools are in fact quite general. For example, they are highly relevant to these other rendering strategies. Moreover, they are very commonly used in general geometric modeling applications – such as the design and manipulation of freeform curves and surfaces – for precisely the same reasons you will read about when learning the basic techniques in Section III.

Returning to the pipeline itself, notice that the focus is on transformation of model geometry. Sophisticated lighting and shading calculations are also often incorporated, many while maintaining interactive speeds. The line between what can be accomplished interactively and what is best done as a background process is constantly moving. To best appreciate the issues, one needs a firm understanding of lighting and shading models. Such a discussion is beyond the scope of these notes. Some excellent references include [Glassner 89, Foley, et. al. 90, Watt & Watt 92, Watt 93]. For our purposes, the following paragraph highlights many of the central issues and choices.

When maximum photorealism is required, programs designed to run as background processes are normally employed. In these programs, processing steps not shown in the pipeline above may be employed, some of the steps shown may not be performed at all, and the order of application may be different from that indicated. Reasonably sophisticated lighting models running at interactive speeds can be incorporated easily into a pipeline like that of Figure 2.1 after the first two transformations (i.e., in VRC coordinates). This is the OpenGL approach [Schreiner, et. al. 08]. Visible line and visible surface determination algorithms are typically performed much later. Algorithms in this category are generally either "Object Precision" Visible Line Determination (VLD) algorithms or "Image Precision" Visible Surface Determination (VSD) algorithms [Foley, et. al. 90]. Object Precision algorithms would be performed in NPC space, immediately following the perspective divide step. These algorithms are generally *not* interactive. Image Precision algorithms operate in pixel units, hence are performed following the workstation transformation. It is common for algorithms of this variety to operate at interactive speeds. Finally, ray tracing is a popular hybrid algorithm for VSD. Some implementations operate in VRC space; others operate in WC. See the comments in section VIII and [Goldman & Miller 97, Miller 97].

We can understand the operation of pipelines like that of Figure 2.1 by studying how a single point is processed. We start with the point defined in modeling coordinates. Recall that modeling (or "local") coordinate systems may be nested arbitrarily deeply. This is signified in Figure 2.1 by the arrow leaving the first block and immediately returning. Our point enters the pipeline at the left end, is transformed by a series of modeling transformations — each transforming it from one local coordinate system into the one which immediately contains it — until the coordinates of the point in the world (or “global”) coordinate system have been determined. These coordinates are then transformed according to the current viewing specifications, initially into View Reference (or “eye”) coordinates, and then into projection coordinates, taking into account whether a parallel or perspective projection is being generated. It is to support the possibility of perspective transformations that the system will internally generate a representation at this stage of the pipeline in homogeneous coordinates if they are not already stored that way.

The “Perspective Divide” step is the process by which a point in projective space is mapped back to normal affine space. We will come to understand what that means in Section III.

During the “Clip” stage of the pipeline, any portion of the geometry outside the current field of view is eliminated. Clipping can be performed solely in *HNPC* as implied here, or it can be performed following the “Perspective Divide” step. The mathematics are simpler in the latter case because parallel and perspective view volumes can be assumed to have been sheared and/or distorted into a common orthographic view volume for clipping. However an initial partial clipping operation must be performed in the case of perspective projections to handle the possibility of geometry behind the viewer. We do not discuss clipping algorithms here; hence we will implicitly assume that clipping is performed where indicated in the pipeline and not consider it further.

The final stage of the pipeline is the “Workstation Transformation”. This is where the system maps coordinates from the logical device space to the physical device. In PHIGS, the programmer can request that only part of the logical device be mapped to some part of the physical device, hence this process may require further clipping. In OpenGL, on the other hand, the entire logical device is mapped to the currently selected portion of the physical device, hence no additional clipping is required.

Aside: The Concept of “Workstations” in PHIGS

PHIGS employs the concept of a *workstation*. A workstation is the programmer’s interface to a physical device. If the device is an interactive one controlled by a window manager, the PHIGS program can actually have several logically different workstations opened at once on the device, each in its own window. A PHIGS program can simultaneously have several workstation interfaces to different physical devices active. For example, some select portion of a model may be displayed on a monitor while the program is directing a possibly different view of the entire model to a plotter. The main description of the PHIGS model (also called the Central Structure Store, or CSS) is maintained in model coordinates independent of any workstation. That is, all workstations operate from the same model. Viewing information, however, is considered to be workstation-dependent. Each workstation maintains viewing information independently. This viewing information includes the view orientation matrix, the view mapping matrix, and NPC clipping information. The workstations actually have a table, each entry of which holds a complete set of this viewing data. The model in the CSS specifies views indirectly by indicating the current table position index to be used at the workstation. The pipeline shown above therefore branches twice between the first two blocks in PHIGS. The first branch takes you to a specific workstation; the second takes you to a specific row in the viewing table. While these abilities provide considerable flexibility to the programmer, they do not affect the mathematics presented here, so we will not consider these matters further.

Note that the mapping transformation actually does two things. It first applies the current projection (parallel or perspective) by distorting the geometry into a common orthographic view volume. It then maps from user units in the VRC coordinate system to the normalized units of HNPC. That is, the basic units used by the application programmer (angstroms, meters, parsecs, or whatever) are in use throughout MC, WC, and VRC, but they are then supplanted by normalized units.

Aside: Low-Level Implementation Notes

All the operations shown inside ovals can be implemented by matrix operations when the model geometry is linear (points, lines, and polygons). Even though they are shown as distinct steps, most graphics implementations actually concatenate as many of the matrices as possible before transforming points. Hence a point would be mapped directly from MC to HNPC with a single matrix operation. (Those operations shown in the two rectangles – “Clip” and “Perspective Divide” – are not matrix operations, hence they “break the pipe”.) Some graphics systems do not introduce the extra NPC coordinate system; instead, their view mapping transforms directly into (homogeneous) device coordinates. Hence, following the perspective divide, the system would have clipped pixel coordinates ready to pass off to the hardware without further transformation. Whether an implementation does any of these things is usually transparent to application programmers.

This page was intentionally left blank.

III. Mathematical Preliminaries: Points, Vectors, and Matrices in Affine and Projective Spaces

In this section we review many very basic yet critical concepts related to points, vectors, and the spaces in which they live. Portions of this material may be review for some readers; they are here because of their fundamental importance with respect to understanding all that follows. I stress the development of a geometric intuition which will aid you in understanding the derivations which follow and help you to master the tools required to develop more advanced vector geometric analysis abilities.

We somewhat restrict our treatment of these topics — especially those related to projective spaces and projective maps — to what is required to support an interactive graphics pipeline such as that of Figure 2.1. There are several good references for other perspectives as well as for additional detail on this material. Examples include [Goldman 85a], [Goldman 85b], [DeRose 89], the Appendix of [Foley, et. al. 90], chapters 1 and 2 of [Farin 95], chapter 2 of [Farin 96], and Appendices B and C of [Angel 09].

In section III.1, we develop a formal understanding of points and vectors. In a sense, they are "performers in a play". Section III.2 then discusses spaces: vector spaces, affine spaces, projective spaces, etc. Extending our analogy, spaces represent the "sets" in which our "performers" operate. The remainder of section III discusses various transformations, maps, and other operations that are applicable to points and vectors in the various spaces. These operations characterize how our "performers" work with one another and move about within a given "set".

A thorough understanding of this material is important because no graphics or modeling system is likely to provide all the functionality you require in exactly the way you will need it. Inevitably the need will arise for you to derive and implement computations to obtain some required geometric quantity. If you cannot justify your derivations and computations in terms of the properties and allowed operations described in this section, your derivation will most likely prove invalid.

III.1 Points and Vectors

We all have an intuitive sense of what points and vectors are. A point is simply a position in space. In order to describe and manipulate a point numerically, we must "address" it by specifying signed distances from some reference point along linearly independent directions (i.e., we must assign it "coordinates"), but the point is the same regardless of the coordinate system we use to address it.

As a concrete example, the swing set in my back yard sits at a fixed location. From the door to my back yard, my kids go 20 feet west and 25 feet north to get to it; hence its coordinates with respect to the coordinate system of my back door are (-20,25). My neighbor's kids also like to play on our swing set, but from their back door, they go 30 feet south and 120 feet west. Hence the swing set's coordinates with respect to the coordinate system of *their* back door are (-120,-30). Certainly the swing set exists independent of either of our back doors; the doors merely provide two equally convenient reference frames to describe the location of the swing set. Similarly, points do not "belong" in any sense to any coordinate system; we merely describe their positions by reference to such systems.

By contrast, a vector *has no location*, rather it is defined solely as a direction and a length. Like the point, we describe its direction numerically with respect to some coordinate system, but —

also like the point — this numeric description is different in different coordinate systems.

> » The fact that a vector has no position means that we can draw the visual representation of a vector anywhere. We are free to drag such a visual representation anywhere we wish.

Many texts gloss over or totally ignore this distinction. However one of the advantages of carefully preserving the distinction between points and vectors is that we can derive very general expressions and algorithms for important operations which are independent of the position and orientation of the geometry with respect to any particular coordinate system.

There are concrete advantages realized in computer code written based on these observations. These advantages include improved computational speed, better numerical reliability, and simplicity of implementation. Remarks at the end of section V emphasize some of these advantages in the context of common modeling transformations.

When authors treat a point as if it were a vector, they are really employing a "position vector". That is, they are considering a vector from the origin *of some particular coordinate system* to the point in question. Thus the *general* concept of a point is being considered as a *specific* vector in a *specific* coordinate system. Symmetrically, when authors treat a vector as if it were a point, they are speaking of the point at which one arrives if you start at the origin of *some specific coordinate system* and traverse the length of the vector.

It is possible to get a quick sense for why points and vectors must be distinguished by considering the example below. We shall see that addition operations applied to the coordinates representing points is meaningless, whereas analogous arithmetic operations applied to the components of vectors *is* meaningful (not to mention useful!).

In Figure 3.1(a), we notice two points (P and Q) and two reference coordinate systems ($\boldsymbol{S}_1$ and $\boldsymbol{S}_2$). As measured in $\boldsymbol{S}_1$, P=(-1,2.5) and Q=(1,2.5). Similarly, in $\boldsymbol{S}_2$, P=(-1.5,6) and Q=(-1.5,4).

So what do we get if we add P and Q?

Well, first we need to agree what it *means* to add two points! This issue relates, after all, to a major concept we are trying to understand in this part of the notes. What do you get if you add two oranges? Two trees? Why should we *expect* that it makes sense to add two points?

Presumably because we can describe points with numeric coordinates in some coordinate system, we have come to believe that arbitrary arithmetic operations applied to these coordinates is sensible. But just because something can be represented numerically does not imply that arithmetic on the representation makes sense. Radio station *KXQY* is at 103.9 on the dial; *KRTU* is at 93.8. Of what significance is it that "*KXQY* + *KRTU* = 197.7"? None whatsoever. A recent survey revealed that 58,000 people in the city of Pranq listen to *KRTU*; 43,000 people in Pranq listen to *KRTU*. What is the significance of "101,000", where 101,000 is the sum of 58,000 and 43,000? Or of "15,000", where 15,000 is the difference between these two numbers?

> ***Exercise***: How are the two examples in the previous paragraph the same? How are they different? Try to think of as many ways as you can.

As it turns out, we can make more sense of the arithmetic differences in the two previous examples (e.g., "*KXQY* - *KRTU* = 10.1"). We will see that the situation is the same for points.

Well, enough talk. Let's return to the question "what do we get if we add P and Q?". Let's just assume that "add two points" means "add their corresponding coordinates".

Using their representation in $\boldsymbol{S}_1$, we get $R_1=P+Q=(0,5)$. Using their representation in $\boldsymbol{S}_2$, we get $R_2=P+Q=(-3,10)$. The numerical results are different, but we expected this. But wait! We needed two different *dots* for R_1 and R_2. That is, two different points have appeared! So what does this mean? Well, we started with two points in space, a concept independent of any coordinate system. Then we "added" the points, and obtained answers which depended upon which coordinate system we used to describe them. This is exactly the problem!

To be well-defined, an operation must produce results which are independent of any coordinate system when given inputs which are coordinate system independent. Clearly addition of points fails this test. You cannot "add" points any more than you can "add" trees or radio stations.

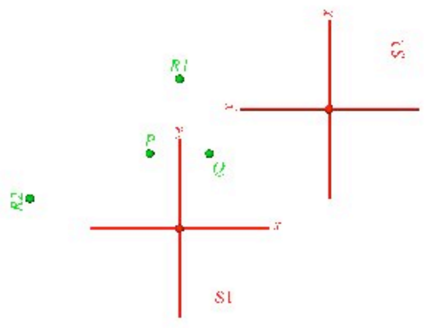


Figure 3.1
(a) Addition of Points is not well-defined

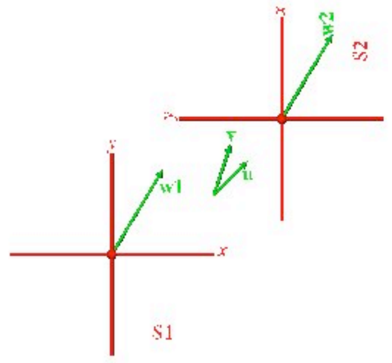


Figure 3.1
(b) Addition of Vectors is well-defined

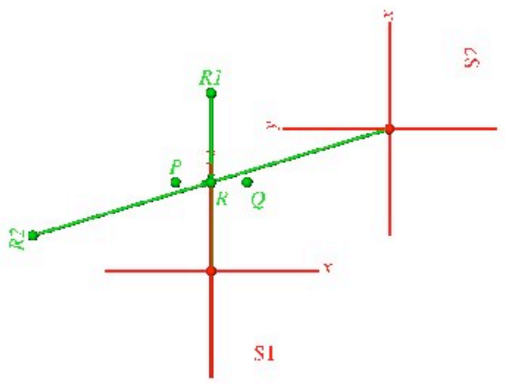


Figure 3.1
(c) Midpoint as an exception

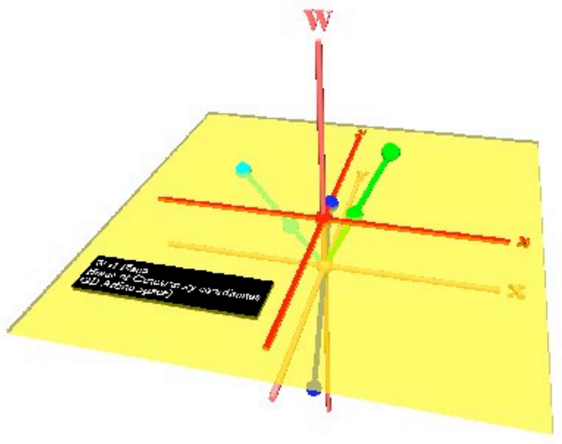


Figure 3.2
The 2D Affine Plane embedded in Projective Space

It is easy to construct similar examples illustrating why multiplication of points by arbitrary scalars (e.g., "2 * P") is meaningless. (Let us assume that multiplying a point by a scalar means multiplying each of its coordinates by the scalar.)

Exercise: construct such an example to convince yourself.

Now let's see how these problems do *not* exist with vectors. Figure 3.1(b) shows the same two coordinate systems with two vectors **u** and **v**. As measured in $\boldsymbol{S}_1$, the vector **u**=(1,1) and **v**=(0.5,1.5). Similarly, in $\boldsymbol{S}_2$, **u**=(1,-1) and **v**=(1.5,-0.5).

So what do we get if we add **u** and **v**?

Well again, we need to agree what it means to add two vectors. Like we did with points, let's just add their components and see what happens.

Using their representation in $\boldsymbol{S}_1$, we get $\mathbf{w}_1$=**u**+**v**=(1.5,2.5). Using their representation in $\boldsymbol{S}_2$, we get $\mathbf{w}_2$=**u**+**v**=(2.5,-1.5). As with the previous example, the *numbers* are different. However, *unlike* the previous example, the *vector* is the same in either case! To see this, drag the tail of the visual representation of $\mathbf{w}_1$ to the tail of that of $\mathbf{w}_2$[1]. The vectors are the same.

Was this just an accident, or are vectors fundamentally different from points? It turns out that they are fundamentally different, but we will wait until Section III.2 before trying to understand why. Addition of vectors – as well as multiplication of a vector by a scalar – yields the same result regardless of the coordinate system of measure.

Exercise: construct an example to convince yourself that scalar multiplication of vectors is also well-defined.

In summary, what we get when we add two points or multiply a point by an arbitrary scalar depends on the coordinate system in which we choose to work. This is bad. What we get when we add two vectors or multiply a vector by a scalar remains independent of the coordinate system in which we choose to work. This is good. Together these two observations demonstrate that points and vectors are indeed different concepts and should be treated as such. Many other examples will show the importance of differentiating between points and vectors.

We cannot quite leave this example yet, because it leaves us with a problem. Consider what happens if we multiply R_1 and R_2 in the example above by 1/2. We get $R_1/2=(0,2.5)$ in $\boldsymbol{S}_1$, and $R_2/2=(-1.5,5)$ in $\boldsymbol{S}_2$. These two points — labeled R in Figure 3.1(c) — are now the same! (As with the vector example, the *coordinates* are still different, but the points are indeed the same.) Of course, this should not be a surprise. Adding corresponding coordinates of two points and dividing by two gives us the *midpoint*, an operation which is certainly well-defined independent of any coordinate system.

Thus we seem to have two conflicting results: since addition and scalar multiplication are not well-defined in general, it would appear that our midpoint operation cannot be justified. But of course we know that it can be. So the question is: what is special about the midpoint calculation? And are there other "special" operations? How can we characterize them in general? And how does all this relate to the conditions under which it is meaningful to perform arithmetic operations on (the coordinates of) points and vectors?

[1] Recall that since vectors have no position (they describe a relative quantity), we are free to drag their visual representations wherever we wish.

In order to address these and other questions, we must formalize our understanding of the spaces in which points and vectors live. In the following section, we shall therefore introduce vector spaces, affine spaces, and others. Once you have an understanding of these spaces and the operations defined in them, you will be able to answer the questions just raised.

III.2 Spaces

So far we have focused on points and vectors — our performers — but not the spaces in which they live. If we wish to be able to make sense of and use in computations the numeric values associated with objects — whether the objects are radio stations or geometric points and vectors — we need to understand the significance of the numbers. That is, we need to understand the environment or the context to which the numbers relate. I realize it is probably obvious, but take a minute and think about the environment to which the numerical quantities in our radio station example relate.

In the context of points and vectors, we characterize this environment by studying the spaces in which these two different quantities live. In this section, we shall therefore study commonly accepted definitions of vector spaces, affine spaces, and others. As we turn our attention to these spaces, we will immediately see other differences between points and vectors. We will also see new ways to understand the relationships between them.

> ***Definition III.1***: An n-dimensional *vector space* consists of a set of vectors and two operations: addition and scalar multiplication. The vector space is closed under these two operations: addition of two vectors yields a vector in the vector space; multiplication of a vector by a scalar also produces a vector in the vector space. Finally, there is a distinguished member of the set called the *zero vector* $\mathbf{0}$ with the properties that $a \cdot \mathbf{0}=\mathbf{0}$ for all scalars a, and $\mathbf{0}+\mathbf{v}=\mathbf{v}+\mathbf{0}=\mathbf{v}$ for all vectors $\mathbf{v}$.

The concept of a vector space is a very general one with applications far removed from computer graphics. The set of all degree n polynomials, for example, forms an n-dimensional vector space. Our interest here is geometry, however, and when we stated in section III.1 that a vector was characterized by a direction and a length, we were speaking of a “geometric vector”. It turns out that such “geometric vectors” form a vector space in the general sense of Definition III.1. Hence it was no “accident” that the example of adding two vectors in the previous section produced a well-defined result.

In the context of geometric vectors, addition of two vectors can be interpreted visually by positioning the tail of one at the head of the other. The vector from the tail of the first to the head of the second represents the vector sum. Multiplying a vector by a scalar scales the length of the vector by the absolute value of the scalar. The direction does not change for positive scale factors; the direction flips for negative scale factors.

While we promise not to forget that a vector space in the sense of Definition III.1 is a general concept independent of geometry, let us agree that, unless explicitly stated otherwise, when we use the term “vector” in the remainder of this monograph, we mean “geometric vector”.

> ***Definition III.2***: An n-dimensional *affine space* is a set of points, an associated n-dimensional vector space, and two operations: subtraction of two points in the set and addition of a point in the set and a vector in the associated vector space. The former produces a vector in the

associated vector space, and the latter produces another point in the affine space. Unlike a vector space which has the distinguished vector **0**, there is *no* distinguished point in an affine space.

The geometric interpretation of the two allowed affine space operations should be clear. For example, given points P and Q in the affine space, the vector $(P - Q)$ has the following interpretation. Starting at the point Q, follow the vector $(P - Q)$, and you arrive at the point P. The algebraic version of this geometric interpretation is the simple identity: $Q + (P - Q) \equiv P$.

Notice that the definition of an affine space permits neither addition of points nor multiplication of points by arbitrary scalars. This should not be surprising since our examples from the previous section illustrated that such operations were not well-defined since they produced coordinate-system dependent results in the general case.

Let us return to the notion of the *length* of a (geometric) vector. Since Definition III.1 deals with the abstract concept of general vector spaces whose elements may not have a length (what’s the length of a polynomial?), there is no defined operation related to nor measure of length. (If elements of a particular vector space had a length – as do our geometric vectors – we could only say that the length of one vector was some multiple of the length of another.) Furthermore, there are no tools associated with affine spaces which allow us to talk about the strongly related notion of the absolute distance between two points. Euclidean spaces address these needs.

Definition III.3: A *Euclidean space* is an affine space with the additional concept of distance.

The primary additional operation applicable to vectors in a Euclidean space is the *dot product* (or inner product). The definition, properties, and various applications of the dot product are discussed along with other common vector operations in section III.7. Here we briefly introduce the properties of the dot product which make it useful as a distance measure.

The dot product of two vectors is a scalar. For two vectors **u** and **v**:

$$\mathbf{u}\cdot\mathbf{v} = \mathbf{v}\cdot\mathbf{u}$$

$$\mathbf{u}\cdot\mathbf{u} \geq 0 \text{(equality is only possible if } \mathbf{u} = \mathbf{0}\text{)}$$

$$(a\mathbf{u})\cdot\mathbf{v} = a(\mathbf{u}\cdot\mathbf{v})$$

We shall see in section III.7 that the length of a vector **u** can be defined in terms of the dot product as $|\mathbf{u}| = \sqrt{\mathbf{u}\cdot\mathbf{u}}$. (The vertical bars around **u** denote “length of **u**”.) Since we can characterize the distance between two points as the length of the vector between them, the dot product also gives us a way to measure distances between points in an affine space. Specifically, the distance between points P and Q is $|P - Q| = \sqrt{(P - Q)\cdot(P - Q)}$.

We have all grown up with affine (and Euclidean) spaces, and we are quite comfortable with them. Now let’s get uncomfortable. Imagine that the affine space we know is actually a plane inside of a space whose dimension is one greater than that of the affine space. For example, consider the conventional 2D *xy* affine plane and suppose that it is really the w=1 plane of a 3D *xyw* coordinate space. This “larger” space is called a *projective space*, and we say that our affine space is *embedded* in the projective space[2].

We formalize these notions with the following definitions.

> ***Definition III.4***: An *(n+1)-dimensional projective space* is the space in which the points of an *n*-dimensional affine space are embedded. We denote the extra coordinate dimension as *w* and say that the affine points lie in the *w*=1 plane of the projective space.

> ***Definition III.5***: All projective space points on the line from the projective space origin through an affine point on the *w*=1 plane are said to be *projectively equivalent to* the affine point.

Returning to two-dimensional affine space as an example, every point (x,y) in the affine space is actually a point $(x,y,1)$ in the projective space. (See Figure 3.2.) Furthermore, all projective space points (wx,wy,w) where $w \neq 0$ are *projectively equivalent to* the point (x,y). Stated in another way, corresponding to every **point** (x,y) in affine space is the **line** in projective space which passes through (0,0,0) and $(x,y,1)$.

Projective spaces (and the related notion of projective maps which are introduced in section III.3) are the subject of projective geometry. Our treatment of this subject is very cursory. The interested reader is referred to [Penna & Patterson 86] or [Coxeter 87] for further information on this rich topic.

The same relationship holds between 3D affine space and projective *xyzw* space. That is, our 3D world lives on the *w*=1 plane of projective *xyzw* space. Obviously this is harder to visualize. One of my favorite references when trying to understand geometry in dimensions greater than three is the nineteenth century book *Flatland* [Abbott 91]. More recent related references include [Burger 65] and [Dewdney 84].

Aside: Complex Projective Spaces

The various spaces we have discussed so far are sufficient for our purposes. For completeness, we present the following definition. Complex projective spaces are of theoretical and practical importance in various areas of Geometric Modeling, most notably in the study of curve and surface intersections.

> ***Definition III.6***: An *(n+1)-dimensional complex projective space* is a projective space in which the coordinates of points are allowed to be complex numbers.

Figure 3.3 illustrates the relationships between the various spaces we have mentioned.

[2] This relationship is superficially similar to the way that 2D affine space relates to 3D affine space. That is, the *xy* coordinate plane simply lives on the *z*=0 plane of 3D space. However the superficial similarity ends there as you will see.

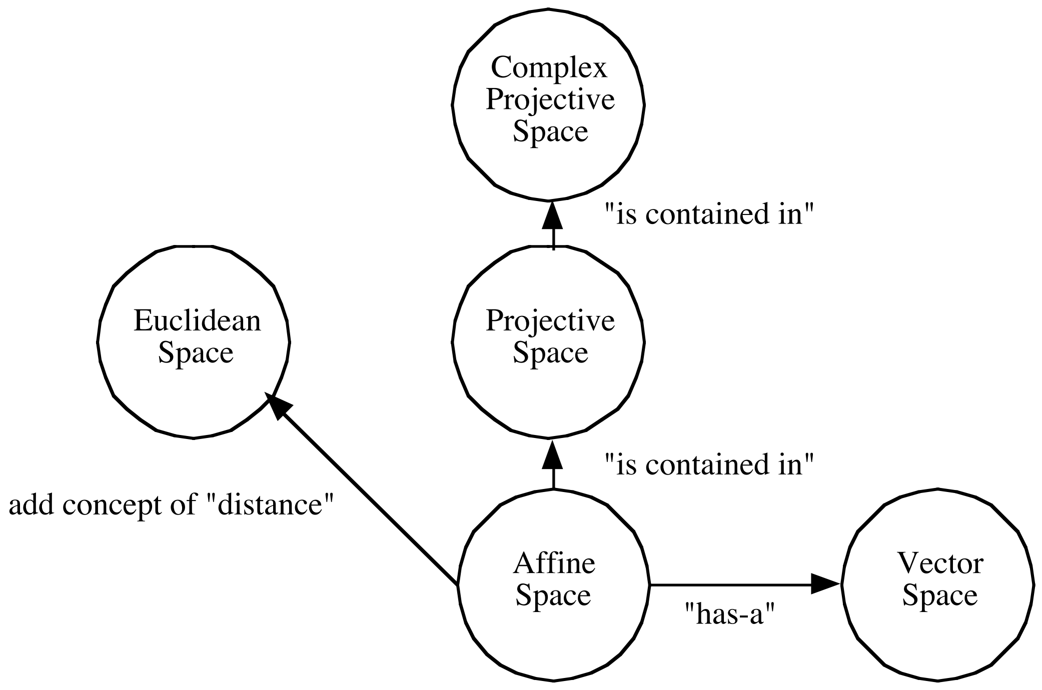


Figure 3.3

Relationships Between the Spaces

While we introduce and talk about projective spaces and projective maps, this machinery is not strictly necessary for the vast majority of common graphics operations. With the exception of perspective transformations for which a projective map will be needed, affine space and affine transformations are adequate for everything that we discuss here. We include a rudimentary discussion of projective spaces, in part for completeness, in part so that we can understand how they relate to affine transformations, and in part for the additional insight they give us into relationships between points and vectors as we discuss next.

Projective spaces give us new ways to think about the differences between points and vectors. Even though we know that they are different, it is annoying that points and vectors are both simply represented as tuples of numbers in affine space. In derivations and computer implementations, for example, it has simply been our responsibility to know whether we should interpret a particular tuple as a point in an affine space or as an element of the associated vector space.

Let's reconsider the process of creating a vector from the difference of two affine points. For purposes of this illustration, let's consider the example of points in a 3D affine space. If we embed the points in projective space and do the subtraction, we get:

$$P-Q=\left(P_x,P_y,P_z,1\right)-\left(Q_x,Q_y,Q_z,1\right)=\left(P_x-Q_x,P_y-Q_y,P_z-Q_z,0\right)$$

Notice that the w component is zero. If this 4-tuple were considered to be a point and projected back onto the w=1 plane, where would it go?

Projective space points which have w=0 are called *ideal points* or *points at infinity*. It is beyond the scope of these notes to explore the significance of points at infinity in the context of projective geometry — the interested reader is referred to [Penna & Patterson 86] or [Coxeter

87] — but clearly there is a one-to-one correspondence between points at infinity and elements of the vector space associated with an affine space.

While it is tempting to imagine that points at infinity and vectors are in fact one and the same, this is not true. Nevertheless we will return to the idea of appending a "1" or a "0" to the coordinates of points and components of vectors in the context of applying transformations in a unified fashion in section III.4.

Let's look at another illustration. Consider the affine point (a,b,c) and the vector **v** in the associated vector space from the origin to the point (a,b,c). The direction of the line passing through the affine origin and the point (a,b,c) has direction **v**. Consider the following sequence of points along this line:

$$(a,b,c,1),(2a,2b,2c,1),(3a,3b,3c,1)\cdots$$

These points are projectively equivalent to

$$(a,b,c,1),(a,b,c,1/2),(a,b,c,1/3)\cdots$$

In the limit, this latter sequence of points converges to $(a,b,c,0)$ in projective space: the ideal point that corresponds to the vector **v** with which we started.

We observed earlier that addition of points in affine space was not a well-defined operation. What happens if we express this operation in projective space?

> ***Exercise***: Show that the addition of n affine points, when expressed in projective space, yields a projective equivalent of the centroid of the n points.

An example of a projective space is HNPC as shown in the PHIGS pipeline of section II. Since HNPC is never visible at the API, most PHIGS discussions do not mention the existence of this coordinate system. We will allude to why HNPC is used in section III.3, and we will see in gory detail later exactly how it is used.

III.3 A Return to Points and Vectors: Characterizing Valid Arithmetic Expressions

Armed with the definitions from the previous section, let us now return to the questions we left dangling at the end of section III.1. At first glance, the results of the examples we studied there would lead us to believe that for arbitrary scalars, points, and vectors (a_i, P_i, and $\mathbf{v}_i$, respectively): $\sum_{i=0}^{n} a_i P_i = undefined$, but $\sum_{i=0}^{n} a_i \mathbf{v}_i = vector$. The problem is that we know of one specific example where such a linear combination of points is valid: midpoint computation.

We need to characterize the exact conditions under which such linear combinations can be considered "valid". Let us begin by rewriting our problematic linear combination of points:

$$\sum_{i=0}^{n} a_i P_i = \sum_{i=0}^{n} a_i P_0 + \sum_{i=0}^{n} a_i \left(P_i - P_0\right) = \alpha P_0 + \sum_{i=0}^{n} a_i \left(P_i - P_0\right) \tag{3.1}$$

where $\alpha = \sum_{i=0}^{n} a_i$ is a constant.

Since our definition of affine spaces tells us that subtracting two points yields a well-defined vector, the summation on the far right-hand side will always yield a well-defined vector since it is just a linear combination of vectors. Hence the original summation on the far left-hand side is well-defined if and only if we choose α such that αP_0 is well-defined. Recall we said earlier only that multiplication of points by *arbitrary* scalars was not well-defined. Perhaps we can find *particular* scalars which produce well-defined results when used to multiply points. Indeed we can.

Observe that if α=1, αP_0 is indeed well-defined in the sense that we have considered so far. That is, $1 \cdot P_0$ yields the same point regardless of the coordinate system in which we measure P_0. Since midpoint computation is simply an operation in which n=1 and a_0=a_1=1/2 (i.e., α=1), we now understand why that operation "works".

Expressions of the form $\sum_{i=0}^{n} a_i P_i$ where $\sum_{i=0}^{n} a_i = 1$ are very important and common. Such an expression is called a *barycentric combination* of points. Furthermore, if each of the P_i are linearly independent and n is the dimension of the space in which they are defined, then we say that the a_i are the *barycentric coordinates* of the point $Q=\sum_{i=0}^{n} a_i P_i$ with respect to the P_i.

Next observe that if α=0, αP_0 yields the n-tuple (0, … ,0). If we interpret this n-tuple as a point at the origin, then the result obviously depends on the coordinate system, hence it's meaningless. However, if we instead interpret this n-tuple as the *zero vector* in the associated vector space, the result <u>is</u> independent of the coordinate system. Therefore we say that multiplying a point by "0" yields the zero vector, independent of the coordinate system used. In the context of equation 3.1, then, the far right-hand side takes the form "*vector + vector*" when α=0.

There are no other possible choices for α. That is, for a scalar α and a point P, it can be proved that αP can be assigned a well-defined interpretation if and only if α is either 0 or 1. The proof is left as an exercise.

Collecting these facts, we can now state that:

$$\sum_{i=0}^{n} a_i P_i = \begin{cases} \textit{point} & \textit{if} \sum_{i=0}^{n} a_i = 1 \\ \textit{vector} & \textit{if} \sum_{i=0}^{n} a_i = 0 \\ \textit{undefined} & \textit{otherwise} \end{cases} \tag{3.2}$$

Not only does this characterization tell us what is allowed in terms of expressions involving points and vectors, but also it gives us a "check" of sorts for derivations that we perform. That is, we can check the "reasonableness" of some derivation by summing the coefficients on all the points in the expression. If they do not sum identically to 0 or 1, our formula cannot be correct. If they do, then we know the result is, respectively, a vector or a point.

Exercise: Based on the characterization of equation 3.2, state which of the following expressions yields (i) a point, (ii) a vector, or (iii) an undefined quantity. (Upper case italic letters denote points; lower case bold letters denote vectors; lower case italic letters denote scalars.)

$B + \mathbf{v} - Q$		$3Q - R - S - T$		$2\mathbf{v} + t\mathrm{R} - Q$	
$P - Q + R + S$		$wQ + (1-w)R$		$2P - R - S + T$	

A Concluding Observation

It is important to note that we have been switching fairly casually between two subtly different types of arithmetic operations. There is the type specifically provided by our definitions: vector spaces permit addition and scalar multiplication (i.e., $\mathbf{u} + \mathbf{v}$ and $a\mathbf{v}$); affine spaces define subtraction of points and addition of a point and a vector (i.e., $P - Q$ and $P + \mathbf{v}$). When we write one of the two valid forms of the right hand side of equation 3.1:

$$P_0 + \sum_{i=0}^{n} a_i \left(P_i - P_0\right) \tag{3.3}$$

or

$$\sum_{i=0}^{n} a_i \left(P_i - P_0\right) \tag{3.4}$$

we are using <u>only</u> these permitted operations. That is, every individual instance of an addition, subtraction, or multiplication operator in equations 3.3 and 3.4 is formally justifiable from the definitions. However, when we write the *algebraically equivalent* summation on the far left-hand side of equation 3.1 (or any specific example thereof like "$3P - Q - R + S - T$"), we are employing individual operations which, technically speaking, are *not* permitted. For example, the multiplication of P by 3 is not by itself a valid operation. Yet the expression as a whole *is* well-defined.

So what are the practical implications of this? As graphics and modeling programmers, we generally derive geometric expressions which we then implement in our programs. It is typically inconvenient to express desired formulas using only the "sanctioned" affine and vector space operations, and we would certainly not want our programming tools to force us to do so. However this implies that the onus is on us to ensure that expressions we implement in computer code are valid from the perspective of equation 3.2.

DeRose has formalized this concept in a geometric programming language which systematically performs various kinds of type checking utilizing this characterization [DeRose 89]. A broader and deeper theoretical treatment of this material is given in [Goldman 85a].

To this point, we have been considering general n-dimensional spaces, using 2D and 3D spaces as specific examples. For the remainder of this monograph, we shall focus on three-dimensional vector and affine spaces, the domain of traditional computer graphics systems.

III.4 Affine and Projective Maps

We have seen the importance of coordinate system transformations to the internal operation of graphics systems. We can view such a transformation as a function X that maps the coordinates of a point as measured in one coordinate system into the coordinates of that point as measured in another. Clearly not just any function X will do, however[3].

It suffices to restrict X to the family of *affine maps*. Since we are dealing with points in affine spaces, and recalling that we have already emphasized in section III.1 the fundamental importance of barycentric combinations of such points, it is prudent to base the definition of an affine map on barycentric combinations [Farin 96]. We therefore define an *affine map* as a mapping which preserves barycentric combinations. That is, if we define a point Q as the barycentric combination:

$$Q = \sum_{i=0}^{n} a_i P_i$$

and X is an affine map, then

$$X(Q) = \sum_{i=0}^{n} a_i X(P_i)$$

This equation simply states that if we first compute Q from the untransformed P_i, then transform Q, we get the same result as we do when we first transform the P_i, then compute the barycentric combination of the transformed P_i.

It is obvious but worth emphasizing that affine transformations map a point in affine space to another point in the same affine space. When we introduce projective maps below, we will see that projective maps do not, in general, map affine points to other points in the same affine space. That is, affine points will in general be transformed out of the w=1 plane.

Aside: Freeform Curves and Surfaces as Barycentric Combinations

The most commonly used freeform curves and surfaces are defined as functional barycentric combinations of points called *control points*. Instead of constants a_i, *blending functions* f_i are used in the barycentric combinations. For example, a curve might be defined as a mapping from $s \in \mathbf{R}^1$ to a point on the curve as

$$C(s) = \sum_{i=0}^{n} f_i(s) P_i$$

where we require that the blending functions f_i satisfy the now familiar constraint

$$\sum_{i=0}^{n} f_i(s) = 1$$

for all s in the domain.

We are guaranteed that the shape of such a curve (or surface) will never change when an affine transformation is applied to it. We are also ensured that it suffices to apply the affine transformation to the n+1 control points; that is, we need not individually transform every point on the curve (or surface).

It is easy to see that X will be an affine map if it takes the form

[3] For example, after some types of nonlinear transformations, a rectangular bear becomes a polar bear.

$$X(Q) = \mathbf{M}Q + \mathbf{t} \tag{3.5}$$

where **M** is a 3x3 matrix and **t** is a vector in the associated vector space [Farin 96]. We can use any combination of translation, rotation, and scaling in an affine transformation, and it is well-known that you can compose any affine map using some combination of these.

Affine maps preserve parallelism (e.g., two parallel line segments remain parallel following transformation by an affine map), and they preserve ratios of signed distances between collinear points [Farin 96]. As a consequence of these properties, it is meaningful to apply the 3x3 matrix **M** of an affine map to a vector **w** in the associated vector space. To see this, choose an arbitrary pair of points P and Q such that $\mathbf{w}=P\text{-}Q$. Now if we apply our affine transformation individually to P and Q, and then form the vector between the resulting transformed points, we obtain:

$$(\mathbf{M}P + \mathbf{t}) - (\mathbf{M}Q + \mathbf{t}) = \mathbf{M}P - \mathbf{M}Q = \mathbf{M}(P - Q) = \mathbf{M}\mathbf{w} \tag{3.6}$$

We are occasionally interested in a special affine transformation called the *identity transformation*. The identity transformation maps vectors and points to themselves: $X(\mathbf{v}) = \mathbf{v}$ and $X(Q) = Q$. The matrix **M** for an identity transformation is the *identity matrix*, **I**:

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The vector **t** for an identity transformation is of course the zero vector: $\mathbf{t} = \mathbf{0} = (0,0,0)$.

Nearly all of the transformations we require in computer graphics can be accomplished with affine maps. The exception is the perspective transformation required to model foreshortening effects; i.e., the idea that the farther an object is from the eye, the smaller it appears on the screen. Perspective transformations require a *rational linear* transformation; i.e., the transformation will employ a ratio involving the coordinates of the point being transformed.

Projective maps are a generalization of affine maps which can be used to model perspective foreshortening, or in general any operation in affine space which is rational linear in the coordinates. This is accomplished by mapping a given affine point to a point in projective space which is projectively equivalent to the desired affine point. One important difference between affine and projective maps is that projective maps do *not* preserve ratios of signed distances between collinear points. An important consequence of this is that projective maps are *not* defined on vectors. For example, suppose (Q, R) and (S, T) are two pairs of points in an affine space such that $\mathbf{w} = Q\text{-}R = S\text{-}T$. If X_P is a projective map, then in general:

$$X_P(Q) - X_P(R) \neq X_P(S) - X_P(T)$$

$$\Rightarrow \quad X_P(Q - R) \neq X_P(S - T)$$

$$\Rightarrow \quad X_P(\mathbf{w}) \neq X_P(\mathbf{w})$$

$$\Rightarrow \quad X_P \text{ is not well-defined on vectors}$$

We will look briefly at the “syntactical mechanics” of projective transformations in the next section. There we will see, for example, that projective transformations map a point in projective space to another point in projective space, but they do not map affine points to other affine points. That is, if a point happens to be in the affine w=1 plane, there is no guarantee that the transformation will leave it in the w=1 plane.

In section VII we shall study how a projective transformation can be constructed for a given perspective viewing specification. Otherwise, affine maps suffice for everything we wish to do in terms of coordinate transformations for computer graphics.

III.5 Matrices

In section III.3 we saw that general affine transformations could be represented by a 3x3 matrix **M** and a vector **t** from the associated vector space (equation 3.5). We also stated in section II that a given point typically undergoes a series of affine transformations as the system prepares to draw portions of geometry dependent upon the point. In order to minimize required computation, it is common to combine as many transformations as possible so that one merged transformation can be applied to points. The application of this merged transformation is equivalent to the successive application of each individual transformation.

This process is cumbersome with the representation of equation 3.5, and it does not allow the inclusion of projective transformations. Granted it is not common to employ projective transformations early in the pipeline; it *is* common to use one as the final step of the pipeline when implementing a perspective transformation. In order to facilitate the mechanics of concatenation of transformations and allow projective transformations as well, we embed the affine points in projective space and augment the representation of the affine transformation so that it describes a linear transformation in projective space. Specifically, we re-write equation 3.5 as:

$$X\left(Q^H\right) = X\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} = \begin{pmatrix} & \mathbf{M} & & \mathbf{t} \\ & & & \\ & & & \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix} \tag{3.7}$$

By setting the bottom row to (0,0,0,1), we are guaranteed that this transformation will not map points out of the affine plane. Therefore the 4x4 matrix in equation 3.7 describes the same affine transformation as does equation 3.5. The only difference is that this description is embedded in projective space.

Let's look a little more closely at the general form of this 4x4 matrix, focusing on what the various portions of the matrix do for us. First note that we will variously use two notations to describe 4x4 matrices. You should be equally comfortable with both:

$$\mathbf{M}_{4x4} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & t_x \\ m_{21} & m_{22} & m_{23} & t_y \\ m_{31} & m_{32} & m_{33} & t_z \\ b_1 & b_2 & b_3 & S \end{pmatrix} = \begin{pmatrix} \mathbf{M} & \mathbf{t} \\ & \\ \mathbf{B} & S \end{pmatrix} \tag{3.8}$$

where

$$\mathbf{M} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} \qquad \mathbf{t} = \begin{pmatrix} t_x & t_y & t_z \end{pmatrix} \qquad \mathbf{B} = \begin{pmatrix} b_1 & b_2 & b_3 \end{pmatrix} \qquad S \text{ is a scalar}$$

Obviously **M** and **t** describe an affine transformation as we have seen. When the bottom row is different from (0,0,0,1), however, the transformation represented by $\mathbf{M}_{4x4}$ is not affine. If **B**=(0,0,0) and $S \neq 1$, a scaling occurs.

We introduced the identity matrix in section III.4. When the context makes the intent clear, we will simply use **I** to denote either the 4x4 or 3x3 identity matrix. If there is any chance for confusion, we will write $\mathbf{I}_{3x3}$ or $\mathbf{I}_{4x4}$ for 3x3 or 4x4 versions of the identity matrix, respectively.

> ***Exercise***: In equation 3.8, suppose $\mathbf{M}=\mathbf{I}_{3x3}$, **t=B=0**, and $S>1$. Describe how $\mathbf{M}_{4x4}$ will uniformly scale any geometry to which it is applied. Does the geometry get larger or smaller in affine space?

> ***Exercise***: Given an arbitrary **M** and **t**, the transformation represented by equation 3.8 with **B=0** and $S \neq 1$ is an affine transformation. Describe how to compute $\mathbf{M}'$ and $\mathbf{t}'$ in terms of **M**, **t**, and *S* so that the effect of transforming a point *P* by the affine transformation determined by $(\mathbf{M}', \mathbf{t}')$ is the same as embedding *P* in projective space, applying the matrix of equation 3.8, and then projecting back to affine space.

Recall we stated that any affine transformation could be described as some combination of translation, rotation, scaling, and shear. In terms of $\mathbf{M}_{4x4}$, the information describing these transformations is strictly partitioned as follows: all rotation, scaling, and shearing operations are encoded in **M**; all translation is contained in **t**.

If we think about this partitioning for a minute, we can tie together our earlier discussion on the relationship between ideal points and vectors with the way equation 3.6 says affine transformations should be applied to vectors. Vectors have no position, hence it is meaningless to apply a translation to them. On the other hand, the translation component of affine transformations is of course critical for points, hence it must be included. Observe how an affine transformation encoded in a 4x4 matrix like $\mathbf{M}_{4x4}$ can be used to apply the transformation in a uniform fashion to points and (the ideal points associated with) vectors represented in projective space:

Applied to an affine point *Q*	Applied to the ideal point associated with **w**
$\begin{pmatrix} m_{11} & m_{12} & m_{13} & t_x \\ m_{21} & m_{22} & m_{23} & t_y \\ m_{31} & m_{32} & m_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} Q_x \\ Q_y \\ Q_z \\ 1 \end{pmatrix}$	$\begin{pmatrix} m_{11} & m_{12} & m_{13} & t_x \\ m_{21} & m_{22} & m_{23} & t_y \\ m_{31} & m_{32} & m_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} w_x \\ w_y \\ w_z \\ 0 \end{pmatrix}$

Table 3.2

It may be difficult to visualize at this point a situation in which you would want **B**≠(0,0,0). As we have remarked, such a transformation would not be affine since it would map points in the affine plane to locations in projective space not on the *w*=1 plane. We will see in section VII how such a mapping can be used to implement perspective projections.

> ***Exercise***: All of our comments here apply equally well to 2D affine

geometry embedded in projective *xyw* space. Suppose we have a 3x3 matrix which describes an affine transformation in 2D space. Describe and justify how a 4x4 matrix applying this same transformation to geometry in 3D affine space can be constructed. The justification should be based on the assumption that the *z* coordinates of points should not be changed by the transformation, nor should they influence the transformed *x* and *y* coordinates of points.

Finally, let's consider again the various coordinate systems introduced in the pipeline of section II (Figure 2.1). I specifically noted the existence of "HNPC", the Homogeneous Normalized Projection Coordinate system. In section III.2 I mentioned HNPC as an example of a projective coordinate system. Both PHIGS and OpenGL employ 4x4 matrices at the API for setting transformation matrices. Therefore in principle PHIGS and OpenGL coordinates can have $w \neq 1$ following the application of the first modeling transformation, and it could be argued that *all* coordinate systems should be labeled with an "H" (e.g., "HWC", etc.). Employing non-affine modeling transformations is extremely rare, and it introduces subtleties that we will not address here. Other than defining rational curves and surfaces as mentioned in section II, coordinates with $w \neq 1$ are most common when processing perspective transformations. Since our use of the "H" prefix was intended to emphasize where homogeneous coordinates are used to support such projections, we only explicitly show it in Figure 2.1 for HNPC.

III.6 Inverse Transformations

It is frequently desirable to apply the inverse of a transformation. For example, a user may click in the graphics window to indicate a position in space where some new feature is to be placed. The device click coordinates must be transformed back to an appropriate object coordinate system. As another example, given a location in world coordinate space on an instance of some component piece of geometry, we would like to be able to obtain the corresponding location in the local coordinate system in which the component was originally specified.

We can characterize the inverse of an affine transformation by solving equation 3.5 for Q:

$$Q = \mathbf{M}^{-1}\left(X(Q) - \mathbf{t}\right)$$

We thus see that the inverse of an affine transformation can be written as another affine transformation:

$$X'(P) = \mathbf{M}^{-1}P + \mathbf{t}'$$

where $\mathbf{t}' = -\mathbf{M}^{-1}\mathbf{t}$. If X maps points from system $\mathbf{S}_1$ to system $\mathbf{S}_2$, X' maps points from $\mathbf{S}_2$ back to $\mathbf{S}_1$.

Obviously $\mathbf{M}$ must be non-singular. This will be the case for all of the transformations we describe in these notes. Finding matrix inverses can be computationally expensive and numerically delicate. The next section describes an important class of affine transformations for which inverse computation is particularly easy and robust.

III.7 Orthogonal Matrices and Rigid Transformations

An *orthogonal matrix* is one which has the property that its inverse is the same as its transpose: $\mathbf{M}^{-1} = \mathbf{M}^{T}$. Equivalently stated, the rows of an orthogonal matrix are mutually perpendicular unit vectors; that is, they determine an *orthonormal coordinate system*.

A 3x3 orthogonal matrix whose rows determine a right-handed orthonormal coordinate system describes a *rigid transformation*. The rows determine a right-handed coordinate system if and only if:

$$(\mathbf{row1} \times \mathbf{row2}) \cdot \mathbf{row3} = +1.$$

(Section III.7.2 reviews the definition and use of the cross and dot products used in this expression.)

Rigid transformations are an important special class of affine transformations which preserve distances between points as well as lengths of and angles between vectors.

We have stated that all affine transformations can be constructed from translations, rotations, scales, and shears. The 3x3 matrices associated with translations and rotations are rigid; those associated with scales and shears are not.

III.8 Common Vector Operations

Earlier we saw that multiplying a vector by a scalar yielded another vector. This new vector had a different length and, if the scalar was negative, pointed in the opposite direction. We also saw that addition and subtraction of vectors was well-defined. The situation with points was more restrictive. While subtracting two points yielded a well-defined vector, general linear combinations of points were only valid under very special conditions.

In this section we review additional important operations applicable only to vectors in a Euclidean space. Most of this material should be review; but, as with previous sections, our approach here is to emphasize the geometric intuition behind these operations. A clear understanding of this geometric interpretation will be vital to an understanding of the material which follows.

We present three operations here: the dot product, the cross product, and the tensor product. For each, we present a mathematical definition from which the geometric intuition derives; a computational formula which is used internally to compute the product from two given vectors; and a list of properties which we implicitly exploit in later derivations.

The dot product and the tensor product are well defined on vectors of any dimension. By contrast, the cross product is unique to 3D Euclidean space.

In formulas and discussion below, we will use $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$, and $\hat{\mathbf{k}}$ to denote unit vectors – i.e., vectors whose length is 1 – along the *x*-, *y*-, and *z*-axes of some orthonormal coordinate system.

III.8.1 The Dot Product

Mathematical Definition For two arbitrary vectors **u** and **v**, their dot product is defined as the scalar:

$$\mathbf{u}\cdot\mathbf{v} = |\mathbf{u}||\mathbf{v}|\cos\theta$$

where θ ($0 \le \theta \le \pi$) is the angle between the two vectors when they are drawn "tail to tail". See Figure 3.4(a).

Properties Multi-linear: $\mathbf{u}\cdot(a\mathbf{v} + b\mathbf{w}) = a(\mathbf{u}\cdot\mathbf{v}) + b(\mathbf{u}\cdot\mathbf{w})$

Commutative: $\mathbf{u}\cdot\mathbf{v} = \mathbf{v}\cdot\mathbf{u}$

$\mathbf{u}\cdot\mathbf{0} = \mathbf{0}\cdot\mathbf{u} = 0$

if ($\mathbf{u}\neq\mathbf{0}$) and ($\mathbf{v}\neq\mathbf{0}$) then:

($\mathbf{u}\cdot\mathbf{v}=0$) if and only if **u** is perpendicular to (orthogonal to) **v**

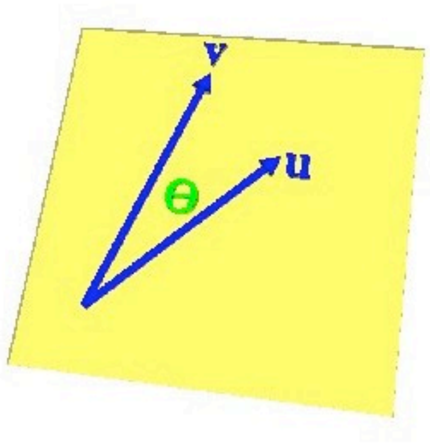


Figure 3.4
(a) The dot product of vectors u and v

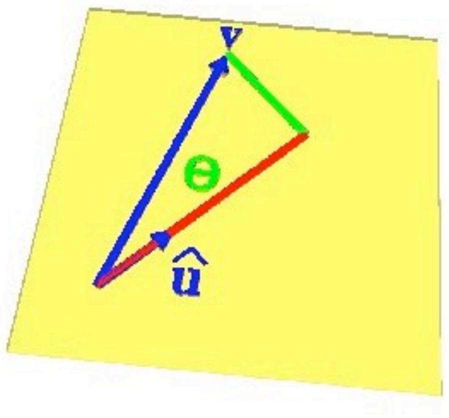


Figure 3.4
(b) The dot product as a length

Computational Formula Clearly $\hat{\mathbf{i}}\cdot\hat{\mathbf{i}} = \hat{\mathbf{j}}\cdot\hat{\mathbf{j}} = \hat{\mathbf{k}}\cdot\hat{\mathbf{k}} = 1$ and $\hat{\mathbf{i}}\cdot\hat{\mathbf{j}} = \hat{\mathbf{i}}\cdot\hat{\mathbf{k}} = \hat{\mathbf{j}}\cdot\hat{\mathbf{k}} = 0$. From these observations and the properties summarized above, it is straightforward to demonstrate that a numerical value for the dot product of two arbitrary vectors **u** and **v** can be computed as:

$$\mathbf{u}\cdot\mathbf{v} = u_x v_x + u_y v_y + u_z v_z$$

For example, if **u**=(1,-2,3) and **v**=(-2,1,1) then **u·v**=-1.

Notes

• The length of a vector is computed as $|\mathbf{u}| = \sqrt{\mathbf{u}\cdot\mathbf{u}}$.

• Since scalar multiplication and division is well-defined for vectors, we can *normalize* a vector (i.e., compute a vector of unit length parallel to a given vector) as $\hat{\mathbf{u}} = a\mathbf{u}$, where $a = 1/|\mathbf{u}|$. A vector of unit length is called a *unit vector*.

• Given a unit vector $\hat{\mathbf{u}}$, we can compute the length of an arbitrary vector **v** *in the direction of* $\hat{\mathbf{u}}$ as $\hat{\mathbf{u}}\cdot\mathbf{v}$. This is illustrated in Figure 3.4(b) where the red line is the result of projecting **v** in a direction perpendicular to $\hat{\mathbf{u}}$ onto the line containing $\hat{\mathbf{u}}$. The length of this red line is the <u>length of **v** in the direction of $\hat{\mathbf{u}}$</u>. This relationship and how it is described by the mathematical definition of the dot product given above (when **u** is a unit vector) can also be understood by considering the right triangle formed by the hypotenuse **v**, the red line, and the green line.

• Given a unit vector $\hat{\mathbf{u}}$, it is frequently useful to decompose an arbitrary vector **v** into the sum of two vectors, one parallel to $\hat{\mathbf{u}}$ and the other perpendicular to $\hat{\mathbf{u}}$. The formula makes use of the "length of **v** in the direction of $\hat{\mathbf{u}}$" formula just given. Specifically we write:

$$\begin{aligned} \mathbf{v}_{parallel,\hat{\mathbf{u}}} &= \mathbf{v}_{\parallel,\hat{\mathbf{u}}} = (\hat{\mathbf{u}} \cdot \mathbf{v})\hat{\mathbf{u}} \\ \mathbf{v}_{perpendicular,\hat{\mathbf{u}}} &= \mathbf{v}_{\perp,\hat{\mathbf{u}}} = \mathbf{v} - \mathbf{v}_{\parallel} = \mathbf{v} - (\hat{\mathbf{u}} \cdot \mathbf{v})\hat{\mathbf{u}} \end{aligned} \tag{3.9}$$

When there can be no ambiguity as to which unit vector is involved, we will usually use the shorthand notation: $\mathbf{v}_{\parallel}$ and $\mathbf{v}_{\perp}$.

Exercise: Describe the geometric meaning of these two formulas.

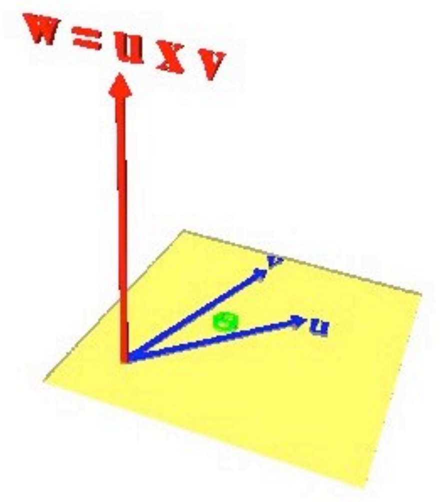


Figure 3.5
(a) The cross product of vectors u and v

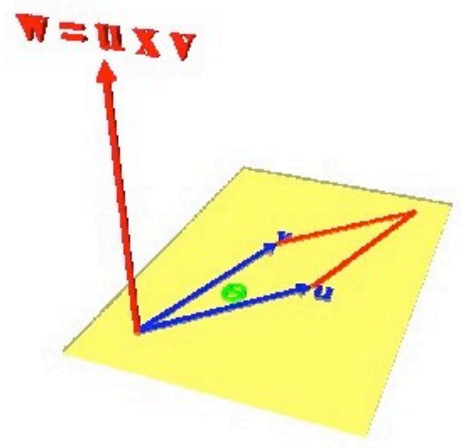


Figure 3.5
(b) The cross product as an area

III.8.2 The Cross Product

Mathematical Definition

For two arbitrary vectors **u** and **v**, their cross product is defined as a vector.

The ***length*** of the cross product vector is defined as:

$$|\mathbf{u} \times \mathbf{v}| = |\mathbf{u}||\mathbf{v}|\sin\theta$$

where θ (0≤θ≤π) is the angle between the two vectors when they are drawn "tail to tail".

The ***direction*** of the cross product vector is defined as perpendicular to both **u** and **v** with sense determined by the right hand rule. See Figure 3.5(a).

Properties

Multi-linear: $\mathbf{u} \times (a\mathbf{v} + b\mathbf{w}) = a(\mathbf{u} \times \mathbf{v}) + b(\mathbf{u} \times \mathbf{w})$

Anti-commutative: $\mathbf{u} \times \mathbf{v} = -\mathbf{v} \times \mathbf{u}$

$\mathbf{u} \times \mathbf{0} = \mathbf{0} \times \mathbf{u} = \mathbf{0}$

if (**u**≠**0**) and (**v**≠**0**) then:

$(\mathbf{u} \times \mathbf{v} = \mathbf{0})$ if and only if **u** is parallel to **v**

Computational Formula

Clearly $\hat{\mathbf{i}} \times \hat{\mathbf{i}} = \hat{\mathbf{j}} \times \hat{\mathbf{j}} = \hat{\mathbf{k}} \times \hat{\mathbf{k}} = \mathbf{0}$, $\hat{\mathbf{i}} \times \hat{\mathbf{j}} = \hat{\mathbf{k}}$, $\hat{\mathbf{j}} \times \hat{\mathbf{k}} = \hat{\mathbf{i}}$, and $\hat{\mathbf{k}} \times \hat{\mathbf{i}} = \hat{\mathbf{j}}$. From these observations and the properties summarized above, it is straightforward to demonstrate that a numerical value for the cross product of two arbitrary vectors **u** and **v** can be computed as:

$$\mathbf{u} \times \mathbf{v} = \begin{vmatrix} \hat{\mathbf{i}} & \hat{\mathbf{j}} & \hat{\mathbf{k}} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{vmatrix} = \left(\begin{vmatrix} u_y & u_z \\ v_y & v_z \end{vmatrix}, -\begin{vmatrix} u_x & u_z \\ v_x & v_z \end{vmatrix}, \begin{vmatrix} u_x & u_y \\ v_x & v_y \end{vmatrix} \right)$$

For example, if **u**=(1,-2,3) and **v**=(-2,1,1) then $\mathbf{u} \times \mathbf{v} = (-5,-7,-3)$.

Matrix Representation

Because the cross product of two vectors is itself a vector, and because the operation is linear, we can express the cross product as a matrix operation. That is, given an arbitrary vector **u**, we can create a matrix **U** such that, for any other arbitrary vector **v**, **Uv** = **u** x **v**. It is easy to verify that

$$\mathbf{U} = \begin{pmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{pmatrix} \tag{3.10}$$

Notes

• The cross product arises frequently in graphics derivations. We will see it, for example, when deriving general rotation matrices in a later section.

• The *length* of the cross product is the *area* of the parallelogram determined by the vectors **u** and **v**. See Figure 3.5(b) where the red lines complete the parallelogram determined by the vectors **u** and **v**.

Exercise: Suppose we define **w** = **u** x **v** and suppose we know only that this definition makes **w** perpendicular to both **u** and **v**. Using this fact and properties of the dot product, show that you can determine **w** up to a scale factor.

III.8.3 The Tensor Product

We will not make extensive use of the tensor product of two vectors in these notes. However, it does occasionally appear in the derivation of general modeling transformation matrices. Therefore we briefly introduce it here.

Mathematical Definition For two arbitrary vectors **u** and **v**, their tensor product — written $\mathbf{u} \otimes \mathbf{v}$ — is an affine transformation whose effect on an arbitrary vector **w** is defined as:

$$(\mathbf{u} \otimes \mathbf{v})\mathbf{w} = (\mathbf{v} \cdot \mathbf{w})\mathbf{u}$$

Properties Multi-linear: $\mathbf{u} \otimes (a\mathbf{v} + b\mathbf{w}) = a(\mathbf{u} \otimes \mathbf{v}) + b(\mathbf{u} \otimes \mathbf{w})$

$\mathbf{u} \otimes \mathbf{v} = (\mathbf{v} \otimes \mathbf{u})^{\mathrm{T}}$

Computational Formula It is easy to verify by direct expansion of the mathematical definition given above that the tensor product of two arbitrary vectors **u** and **v** can be computed as:

$$\mathbf{u} \otimes \mathbf{v} = \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix} \begin{pmatrix} v_x & v_y & v_z \end{pmatrix} = \begin{pmatrix} u_x v_x & u_x v_y & u_x v_z \\ u_y v_x & u_y v_y & u_y v_z \\ u_z v_x & u_z v_y & u_z v_z \end{pmatrix} \qquad (3.11)$$

Notes Equation 3.11 states that if we consider **u** and **v** to be column vectors, then $\mathbf{u} \otimes \mathbf{v} = \mathbf{u} * \mathbf{v}^T$.

Recall our formula for computing the component of one vector parallel to another (unit) vector. In particular, given a unit vector $\hat{\mathbf{u}}$, it is easy to show that:

$$\mathbf{v}_{\parallel} = (\hat{\mathbf{u}} \cdot \mathbf{v})\hat{\mathbf{u}} = (\hat{\mathbf{u}} \otimes \hat{\mathbf{u}})\mathbf{v} \qquad (3.12)$$

It is in this context that we shall see the tensor product appear when developing matrices for general mirrors and rotations. We shall see a slight generalization of this in the context of shears. Specifically, for arbitrary vectors **c**, **d**, and **e**, it is straightforward to demonstrate that:

$$(\mathbf{c} \cdot \mathbf{d})\mathbf{e} = (\mathbf{e} \otimes \mathbf{c})\mathbf{d} = (\mathbf{e} \otimes \mathbf{d})\mathbf{c}$$

III.8.4 Additional Notes on Vector Operations and Manipulations

The operations we have seen to this point comprise the valid set of operations one can perform between a pair of vectors. When performing vector geometric derivations, you must be careful that all steps in the derivation can be justified in terms of these operations. A common mistake, for example, is to conclude from:

$$\hat{\mathbf{n}} = \frac{\mathbf{n}}{|\mathbf{n}|}$$

that

$$|\mathbf{n}| = \frac{\mathbf{n}}{\hat{\mathbf{n}}}$$

and then to substitute $\mathbf{n} / \hat{\mathbf{n}}$ where $|\mathbf{n}|$ appears in some other expression. However the second of the two equations above is *invalid* since division of one vector by another is *not* a defined operation!

We oftentimes wish to compute an arbitrary vector perpendicular to a given one. We can use properties of the dot product to do so. For example, given a vector **w**, find its two components with maximum absolute value. Switch them, negating the component with larger absolute value. Set the third component to 0. The resulting vector, say **u**, will be perpendicular to **w**. (Why?) A third vector **v** can then be computed as **v** = **w** x **u**. Taken together, (**u**, **v**, **w**) can be normalized and used to define a right-handed orthonormal coordinate system. It is common to provide a utility implementing these ideas. Appendix B shows the declaration of a class method *aVector::coordinateSystemFromUW* which does so. Examples 4 and 8 of Appendix E illustrate its use.

IV. Coordinate versus Vector Geometric Approaches for Generation of Transformation Matrices

So far we have considered the roles that transformations play and how they can be represented as matrix operations. Starting in the next section we will study how matrices for specific transformations can be constructed. Here we briefly review two major approaches to the mathematical analysis required to derive expressions for required matrices: *coordinate-based* and *vector geometric*. While both approaches are sufficiently powerful to derive all matrices of interest, we shall see that the two approaches have complementary strengths. Each will emerge as ideal for different situations.

By "coordinate-based" methods, we mean mathematical analyses based directly on the relationship of geometry to some specific coordinate system. For example, I may ask how the z coordinates of two points compare. By contrast, "vector geometric" methods focus on relationships between geometric entities, not their individual positions and orientations with respect to some particular coordinate system. For example, I may compute the vector between two points and find its length in the direction of some other relevant unit vector.

You may argue at this point: "But how can you compute the vector and find its length in the specified direction without using coordinates of the points?" It is absolutely correct that to carry out these operations on a computer, you need to use coordinates. However, the level of abstraction used in the mathematical analysis remains focused on points and vectors in Euclidean spaces. We will see several examples of the benefit of this higher level of abstraction in sections V and VI when deriving the matrices for general modeling and viewing transformations. For example, coordinate-based approaches for constructing matrices for general rotations must employ numerically sensitive tests to determine whether intermediate floating point values are (close to) zero. By contrast, the vector-geometric approach that we will derive requires only one initial test to ensure that the programmer has not given us a zero vector as a rotation axis. Not only will we require just this one test, but it is also easier to make it numerically robust.

As a final rejoinder to the argument that coordinates must be used to actually carry out the computations, note that properly engineered modeling system libraries will hide those details from you as a programmer. That is, *point*, *vector*, *affine transformation*, and others will be abstract data types for which you will only see the fundamental Euclidean space operations of addition and subtraction of points and vectors, scaling of vectors, barycentric combinations of points, dot and cross products of vectors, etc.[4]

With respect to the pipeline of figure 2.1, we will find that vector geometric approaches are best suited when deriving transformations for the early stages of the pipeline (i.e., the various modeling and view orientation transformation matrices). But then the situation will change. We

[4] Well, graphics APIs like OpenGL and OpenInventor *do* ask you to provide coordinates, arrays of coordinates, arrays for matrices, and such. I am not suggesting that these and similar APIs are not "properly engineered". To the contrary, they were designed as general low-level APIs for 2D or 3D geometry in either affine or projective space. They make no specific assumptions about the application which uses them. My comments here should be interpreted as applying to higher level libraries used by your application. The translation to coordinates would then only be required to, for example, draw a picture using OpenGL of something you created in your program. In fact, that application-dependent translation could easily be hidden in your program in the implementation of some *draw* method for your application's graphical objects.

construct the view orientation transformation so that the resulting VRC system exhibits certain important relationships with respect to the graphics display device. Not only do these relationships simplify (both for the end-user as well as for the graphics programmer) the subsequent specification of projection and workstation mapping data, but also they can be exploited to simplify the analysis required to derive the projection and workstation transformation matrices. These matrices (as well as the clipping and projection algorithms) are most expediently derived using coordinate-based schemes that exploit the special properties that we "built into" VRC and NPC.

IV.1 Coordinate-Based Approaches

Coordinate-based approaches in effect force us to use a particular coordinate system as an intermediary when asking questions about the relationship between two objects. For example, instead of directly specifying a query between two entities A and B, I must first look at property p of A with respect to the coordinate system and compare the result to property p of B with respect to the same coordinate system. Based on the comparison, I must try to determine an answer to the original question. If I am lucky — i.e., if A and B happen to be described in a convenient coordinate system — everything is OK. Otherwise it gets hairy.

When this approach works, it generally works well. As an example, suppose we have transformed two points into an eye coordinate system, and now we need to know which is closer to the eye so that we will know which is visible. The eye coordinate system is constructed so that "distance from the eye" relates directly to "z coordinate". Hence we use "z coordinate" as "property p", and we can answer our original question simply by comparing the two z coordinates. Nothing could be simpler.

On the other hand, suppose we have two arbitrary vectors $\mathbf{u}$ and $\mathbf{v}$, and I want the matrix which rotates $\mathbf{u}$ onto $\mathbf{v}$. Since we cannot assume any special relationships among $\mathbf{u}$, $\mathbf{v}$, and the coordinate system, a coordinate-based approach would have to determine the series of transformations which would map each vector onto, say, the z axis. The desired transformation would then be the result of concatenating the transformations for $\mathbf{u}$ followed by *the inverse of* the transformations for $\mathbf{v}$. As we will see below, this is even more complicated than it sounds to implement correctly.

IV.2 Vector Geometric Approaches

Vector geometric approaches free us from the need to use coordinate systems as intermediaries when performing geometric operations or querying the relationship between objects. Instead they allow us to derive relevant expressions which are independent of any particular coordinate system.

Two primary considerations with respect to typical geometric analysis make the ability to define and manipulate expressions in this manner desirable:

- There are in general no special relationships between the entities with respect to a given coordinate system which can be exploited to make the derivation easier.
- While there is *in general* no special relationship that we can exploit, there *may coincidentally* be some special relationship that a coordinate-based algorithm would have to *detect* and *handle*.

Let us examine this second point more closely by again considering the example of rotating **u** onto **v** that we mentioned above. Suppose we just consider the first part of that process: namely computing the matrix which rotates **u** onto the *z*-axis. Following the development in [Foley, et. al. 90], the required matrix would be the product:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{\sqrt{u_x^2+u_z^2}}{|\mathbf{u}|} & \frac{-u_y}{|\mathbf{u}|} \\ 0 & \frac{u_y}{|\mathbf{u}|} & \frac{\sqrt{u_x^2+u_z^2}}{|\mathbf{u}|} \end{pmatrix} \begin{pmatrix} \frac{u_x}{\sqrt{u_x^2+u_z^2}} & 0 & \frac{u_z}{\sqrt{u_x^2+u_z^2}} \\ 0 & 1 & 0 \\ \frac{-u_z}{\sqrt{u_x^2+u_z^2}} & 0 & \frac{u_x}{\sqrt{u_x^2+u_z^2}} \end{pmatrix}$$

The difficulty arises because the denominators in the second matrix may be zero or nearly zero. Geometrically, this means that **u** is parallel to (or nearly parallel to) the positive or negative *y*-axis direction. Therefore, an implementation of this coordinate-based algorithm would have to first check for this "special case" rather than simply computing the product of the two matrices. This is unsatisfactory because whether **u** is parallel to some coordinate axis is totally irrelevant to the operation that is ultimately desired: namely rotating **u** onto **v**. When computing the analogous two matrices for **v**, similar problems of course arise. Finally, the matrix for rotating **u** onto **v** is computed as the product of four matrices: the two matrices shown above and the inverse of the two **v** matrices.

By contrast, a vector geometric approach to this operation would simply compute directly the matrix which rotates about the vector **w** (where **w** = **u** x **v**) by the acute angle between **u** and **v**. (Refer again to figure 3.4.) We can easily determine the sine and cosine of this angle from the formulas presented in the section III.8. We will see later in section V a formula for the desired matrix which requires only **w**, the sine, and the cosine.

Having derived appropriate formulas, we must of course use *some* coordinate system so that we can generate internal numerical representations to drive our computer programs. Again the point is that the choice of coordinate system is irrelevant: we will get the same answer whichever coordinate system we use.

In summary, mathematical derivations based on vector geometric analyses are oftentimes simpler, they are free of annoying coordinate-system dependent special cases, and the results are never worse and are almost always better in terms of efficiency of implementation. As a result, we will see very real and quantifiable differences in that the resulting code

- is more compact.
- is more computationally efficient.
- is more robust.
- has fewer, if any, special cases.

Most special cases in vector geometric approaches are trivially detected before the algorithm begins. By contrast, coordinate-based methods commonly require special case detection at intermediate steps of the algorithm based on quantities derived after a series of numerical computations. General rotations (or the example of rotating an arbitrary vector **u** onto another arbitrary vector **v**) are excellent examples of this.

However there are situations where it is useful to transform geometry into special coordinate systems where the orientation of the system can be exploited to dramatically simplify and accelerate certain types of imaging operations. Examples include clipping algorithms, visible line and visible surface determination, and intensity depth cueing.

V. Modeling Transformations

In this section and the next, we will describe how to derive various matrices which apply modeling transformations and viewing transformations to points and vectors in Euclidean spaces. We already saw in section II how these operations are used in typical graphics pipelines. Given what we now know about affine transformations and how they are represented with matrices, a logical question to ask is: "What is the difference between modeling and viewing transformations?" In a word, the answer is: "Nothing."

Given a matrix describing an affine transformation, it is not possible to characterize it as a "modeling transformation" or a "viewing transformation"; that is, it is not possible to state what the person who generated the matrix was thinking. Specifically, a given matrix can be interpreted either as transforming geometry in a fixed coordinate system (a modeling transformation), or as leaving the geometry fixed while applying the inverse of the transformation to the coordinate system (a viewing transformation). Both interpretations are equally defensible.

While there is no difference mathematically, there *is* a difference in the way in which application programmers (or end users) think about and want to specify these transformations. For example, in one case a user might want to prescribe a rotation axis and an angle about which some geometry is to be rotated (a specification of a modeling transformation); later this same user will find it more meaningful to specify a viewing transformation by telling us that an observer of the geometry is standing at some point P, and looking at some other point Q. Not surprisingly, it also turns out that different analytical approaches for deriving appropriate matrices are useful for these different specifications, hence the treatment in two different sections.

As argued in section IV, it is most often the case that vector geometric analyses are best for general modeling and viewing transformations. We will therefore not consider coordinate-based approaches for derivations of modeling and viewing transformations in these notes. Such descriptions are included in other more comprehensive texts such as [Foley, et. al. 90; Angel 09].

V.1 A General Strategy

Recall we saw in section III.4 that only $\mathbf{M}$ (the upper 3x3 portion of a transformation matrix) is required to specify how an affine transformation is applied to vectors. Points, on the other hand, require both $\mathbf{M}$ and $\mathbf{t}$. Therefore it is hardly surprising that it is easier to derive affine transformations for vectors than it is for points.

The alert reader (no — make that the *aggressive* reader) might also wonder if we could somehow complete the derivation of the transformation for points (i.e., go on to determine what $\mathbf{t}$ should be) by reducing the problem to a vector one. The aggressive reader would be correct. Here is a sneak preview.

Consider a generic point P to which we wish to apply some affine transformation X. We will assume that we know how to apply X to vectors. Now suppose we pick some other point F and write the identity: $P = F + (P - F)$. This identity has a simple vector geometric interpretation: start at the point F, travel along the vector from F to P, and you arrive at P. Recall we want to apply the affine transformation to P, hence we can write $X(P) = X(\ F + (P - F)\ )$. Since affine transformations distribute over addition, we can rewrite this as $X(P) = X(F) + X(P - F)$. Of course, $(P - F)$ is a vector, hence by assumption we can directly compute $X(P\text{-}F)$. Now here's the trick: suppose the F we picked is a *fixed point* of the transformation. That is, suppose $F\text{=}X(F)$.

This means we can directly apply X to points once we know how to apply X to vectors and have a fixed point.

Fixed points of modeling transformations are generally always known. For example, if I want a matrix which applies a rotation about some axis, every point on the rotation axis is a fixed point. I am therefore free to use any point on that axis as F.

We will therefore adopt the following general strategy when performing the analysis for modeling transformations.

General Strategy #1
1. Determine a 3x3 matrix **M** which applies the affine transformation to 3D vectors.
2. Use a fixed point of the transformation to determine the translation component **t** of X.
3. Use **M** and **t** to build $\mathbf{M}_{4x4}$ as in equation 3.8, setting **B**=(0,0,0) and S=1.

Clearly step 3 is just a mechanical process of plugging values into matrices. As implied in the "sneak preview" above, it turns out that step 2 is also completely mechanical. Let's assume we have determined **M** for our affine transformation, and we know a fixed point F. We determine **t** as follows:

$$\begin{aligned} X(F) &= F = \mathbf{M}F + \mathbf{t} \\ &\Rightarrow \mathbf{t} = F - \mathbf{M}F \end{aligned} \tag{5.1}$$

V.2 The Mirror Transformation

To get an idea for how the general strategy of the previous section works, let's consider mirror transformations. These are common modeling transformations which flip the "handedness" of geometry. For example, if I have a model of the left rear door of a car, I can apply a mirror transformation to obtain the geometry of the right rear door. Refer to Figure 5.1(a).

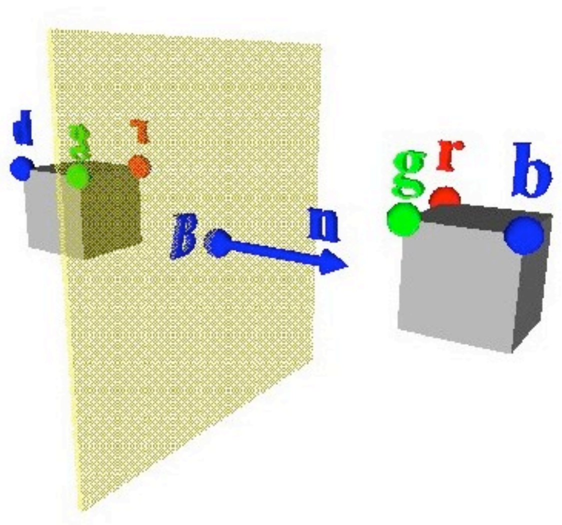


Figure 5.1
(a) A mirror transformation

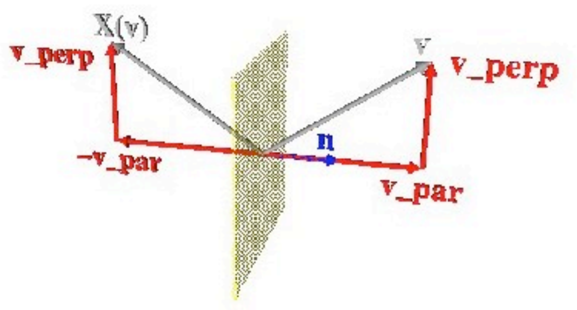


Figure 5.1
(b) The effect of the transformation on vectors

We stated earlier that the four basic affine transformations are: rotate, translate, scale, and shear. A general mirror transformation can be constructed from a combination of rotation, translation, and scale transformations. However since it is such a common operation and since specifying the mirror plane directly is so much simpler than determining and assembling the required component rotation, translation, and scaling transformations, we treat it directly here. Matrices which describe a common form of general scale will be treated as an exercise below after we have practiced our technique with mirror transformations.

The mirror transformation is defined by specifying a plane in space to use as the mirror. It is common to define this mirror plane by specifying a point *B* on the plane and a vector **n** perpendicular to the plane as illustrated in Figure 5.1(a). Recall that we cannot assume the plane has any particular relationship with respect to the coordinate axes. In our car example, the designer would give us the plane which cuts the car in half, splitting the driver's and passenger's sides. That is, *B* would be some point in the middle of the car; perhaps where the hood ornament is attached to the hood. The vector **n** would be a vector pointing from the left side of the car to the right side. (We could equally well use the vector pointing from the right side to the left side. See the *Exercise* below.) The mirror transformation would then map a point on one side of the plane to a point on the other side which is an equal distance away as shown in Figure 5.1(a).

A 4x4 matrix defined only in terms of *B* and **n** which represents this transformation is stated without proof in [Goldman 90]. The remainder of this example illustrates how the vector geometric tools we have been studying are used to derive this matrix. The example here concludes with sample C++ code implementing the formula.

Our general strategy says that we need to focus first on vectors. Think for a minute about how we could describe the way vectors are affected by a mirror transformation. In particular, think back to vector decomposition as discussed in section III.8.1 (and summarized in equation 3.9).

Consider an arbitrary vector **v** and its components parallel to and perpendicular to the plane normal **n**. How would each be affected by the mirror operation? The component perpendicular to **n** will not change, right? The component parallel to **n** is simply flipped (i.e., negated). Pretty simple. Figure 5.1(b) illustrates the basic idea.

To use our equations, we need to compute $\hat{\mathbf{n}}$ from the given mirror plane normal vector **n**. As we will see in the algorithm at the end of this section, it is common to perform this vector normalization at the start of the algorithm. This allows us to use simplified equations like equation 3.9, and it also allows us to check right away for input errors. In the case of mirror transformations, the only possible error is providing a zero vector for **n**.

Having done this, we determine how **v** is transformed as follows. (As has been our convention so far, we will use *X* to denote the current transformation; i.e., the mirror transformation.)

$$\begin{aligned} X(\mathbf{v}) &= X(\mathbf{v}_{\parallel} + \mathbf{v}_{\perp}) \\ &= X(\mathbf{v}_{\parallel}) + X(\mathbf{v}_{\perp}) \\ &= -\mathbf{v}_{\parallel} + \mathbf{v}_{\perp} \end{aligned}$$

Now using our formulas for vector decomposition from equation 3.9, we can write:

$$\begin{aligned} X(\mathbf{v}) &= -(\mathbf{v}\cdot\hat{\mathbf{n}})\hat{\mathbf{n}} + \left(\mathbf{v} - (\mathbf{v}\cdot\hat{\mathbf{n}})\hat{\mathbf{n}}\right) \\ &= \mathbf{v} - 2(\mathbf{v}\cdot\hat{\mathbf{n}})\hat{\mathbf{n}} \end{aligned}$$

We need to write this as a matrix operation[5], but there is this dot product stuff going on. Remember the role we mentioned for the tensor product? In particular, recall equation 3.12 which allows us to write the expression for $X(\mathbf{v})$ as the following matrix operation:

$$X(\mathbf{v}) = \mathbf{M}\mathbf{v}$$

where

$$\mathbf{M} = \mathbf{I} - 2(\hat{\mathbf{n}} \otimes \hat{\mathbf{n}}) \tag{5.2}$$

Exercise: Using equation 3.12, show that equation 5.2 is correct.

Step 1 of our general strategy is now complete. The matrix **M** of equation 5.2 is what we need for vectors.

For step 2 we need a fixed point. Clearly any point on the mirror plane is a fixed point. In particular, we just use the point *B* mentioned in the problem description and compute **t** using equation 5.1 with *F*=*B*. Step 3 then simply has us assemble the final 4x4 matrix as:

$$\mathbf{M}_{4x4} = \begin{pmatrix} & & & \\ & \mathbf{M} & & \mathbf{t} \\ & & & \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.3}$$

Exercise: Show that the matrix of equation 5.3 is unaffected if $\hat{\mathbf{n}}$ is replaced by -$\hat{\mathbf{n}}$. (Note that this requires that *both* **M** and **t** be unaffected.)

Exercise: Suppose *B*=(-1,3,2) and **n**=(2,1,-1). Show the 4x4 matrix which results from these values. Don't forget that **n** must be normalized before equation 5.2 (and hence equation 5.1) can be used!

Exercise: Show how the matrix of equation 5.3 specializes when the mirror plane is (i) the *xy*-plane, (ii) the *xz*-plane, (iii) the *yz*-plane. What do you use for *B* and **n** in each of the three cases? Do the resulting matrices make sense? Are they what you would expect?

Exercise: A generalized scaling operation can be defined with a point *C* and three scale factors *sx*, *sy*, and *sz*. Geometry is scaled about *C* by the given scale factors in the three coordinate directions. Show how to use General Strategy #1 to derive a 4x4 matrix which will apply this scale transformation to points.

Exercise: Characterize the conditions under which the general scale transformation of the previous exercise will be identical to the general mirror transformation derived above.

[5] If one only wishes to mirror individual vectors, then the formula just derived should be used. One needs the matrix we derive next only to allow concatenation of modeling and viewing transformations in the graphics pipeline.

Before moving on to the more complex example of general rotations, let's think briefly about how one would write code to implement the results of this analysis. In particular, code is only needed to implement the final matrix of equation 5.3, not the derivations leading up to it. To summarize, a pseudo-code implementation for general mirror matrix generation might look like:

```
algorithm BuildMirrorMatrix(B,n) returns 4x4Matrix
1. if |n| == 0 then flag error; return IdentityMatrix
2. nHat = normalize(n)
3. Compute M using equation 5.2
4. Compute t using equation 5.1 with F=B
5. return the 4x4 matrix of equation 5.3
end algorithm
```

Using the C++ class definitions for points, vectors, and matrices documented in the Appendices, we can give a concrete implementation of a version of this function which creates the 3x3 matrix **M** and the translation vector **t** for this affine transformation:

```
bool BuildMirrorTransformation(aPoint B, aVector n, Matrix3x3& M, aVector& t)
{   aVector nHat;
    double length = n.normalizeToCopy(nHat);
    if (length < tolerance)
        // a zero vector was provided for n. We cannot proceed.
        return false;
    Matrix3x3 T = Matrix3x3::tensorProductMatrix(nHat,nHat);
    M = Matrix3x3::IdentityMatrix - 2.0*T;
    // The point B is a fixed point of the transformation, hence:
    t = B - M*B;
    return true;
}
```

V.3 Rotation About an Arbitrary Axis in Space[6]

A general rotation in 3D Euclidean space is defined by specifying a rotation axis and a rotation angle. Figure 5.2 illustrates our cube from the previous section being rotated about such an axis. The rotation axis is simply a straight line which can be specified by providing a point on the line and a vector giving the orientation of the line. We will therefore consider the specification $(B, \hat{\mathbf{w}}, \theta)$ as defining a rotation by θ about the axis $(B, \hat{\mathbf{w}})$. The positive direction of rotation is determined by the right hand rule, with your right thumb pointing in the $\hat{\mathbf{w}}$ direction.

> ***Exercise***: Refer again to Figure 5.2. Assuming $0 < \theta < 180^o$, which cube is the "original", and which is the result of rotating the original about the axis?

[6] Ron Goldman first showed me this derivation when we were working together in the early 1980s. Having labored previously only with the "classical" approaches of standard computer graphics texts, this was my first insight into the power of vector geometric approaches. I have since seen similar expressions for the final matrix (e.g., page 596 of [Schreiner, et. al. 08]).

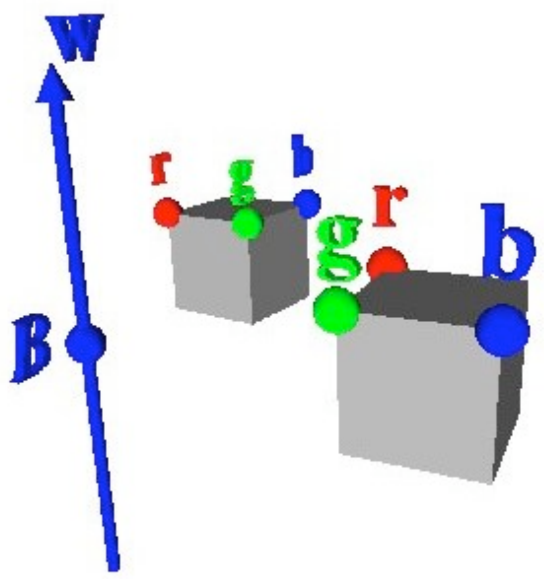


Figure 5.2
A rotation transformation

In [Goldman 90], the formula we are about to derive (equation 5.4 below) is stated without proof. Let us see how this formula can be derived.

As has been our practice, we begin by considering how an arbitrary vector $\mathbf{v}$ is rotated about the vector $\hat{\mathbf{w}}$. We will proceed as we did with mirror transformations. That is, we will consider how the components of $\mathbf{v}$ parallel and perpendicular to $\hat{\mathbf{w}}$ are transformed. It should be clear that any vector parallel to the rotation axis $\hat{\mathbf{w}}$ will be unaffected by the rotation. We therefore need only think about what happens to the perpendicular component. In effect, this allows us to reduce the 3D problem to a 2D one.

Consider the geometry in Figure 5.3. We see the rotation axis $\hat{\mathbf{w}}$, the generic vector $\mathbf{v}$, and the components of $\mathbf{v}$ parallel and perpendicular to the rotation axis $\hat{\mathbf{w}}$. Also indicated is a plane perpendicular to the rotation axis. It is in this plane that we want to work. Notice the two vectors in this plane labeled $\mathbf{v}_\perp$ and $\hat{\mathbf{w}} \times \mathbf{v}_\perp$. Clearly they are perpendicular to each other and lie in the plane. (Why?)

When we rotate $\mathbf{v}$ about $\hat{\mathbf{w}}$, we get $X(\mathbf{v})$ as indicated in the figure. As argued, $X(\mathbf{v}_\parallel)$ is simply the original $\mathbf{v}_\parallel$. We wish to characterize how $X(\mathbf{v}_\perp)$ relates to $\mathbf{v}$ and $\hat{\mathbf{w}}$. The figure indicates that:

$$X\left(\mathbf{v}_\perp\right) = \cos\theta\mathbf{v}_\perp + \sin\theta\left(\hat{\mathbf{w}} \times \mathbf{v}_\perp\right)$$

Exercise: Demonstrate that this formula is correct. (Hint: Think of $X(\mathbf{v}_\perp)$ as the hypotenuse of a right triangle and use the definitions and properties from section III.8.)

We now can write a vector expression describing how to compute $X(\mathbf{v})$:

$$\begin{aligned} X(\mathbf{v}) &= X(\mathbf{v}_{\parallel}) + X(\mathbf{v}_{\perp}) \\ &= \mathbf{v}_{\parallel} + \cos\theta\mathbf{v}_{\perp} + \sin\theta(\hat{\mathbf{w}} \times \mathbf{v}_{\perp}) \end{aligned}$$

Exercise: Show that this equation can be simplified to the following. (Refer again to the definitions and properties from section III.8.)

$$X(\mathbf{v}) = \cos\theta\mathbf{v} + (1-\cos\theta)(\mathbf{v}\cdot\hat{\mathbf{w}})\hat{\mathbf{w}} + \sin\theta(\hat{\mathbf{w}}\times\mathbf{v})$$

(Notice in particular that the final term uses $\hat{\mathbf{w}} \times \mathbf{v}$, *not* $\hat{\mathbf{w}} \times \mathbf{v}_{\perp}$.)

Exercise: Finally, show that the 3x3 matrix **M** (where $X(\mathbf{v}) = \mathbf{Mv}$) can be computed as:

$$\mathbf{M} = \cos\theta\mathbf{I} + (1-\cos\theta)\hat{\mathbf{w}}\otimes\hat{\mathbf{w}} + \sin\theta\mathbf{W} \tag{5.4}$$

where **W** is the 3x3 cross product matrix; i.e., $\mathbf{Wv} = \hat{\mathbf{w}} \times \mathbf{v}$. (Review equation 3.10.)

Exercise: We observed in section V.2 that replacing $\hat{\mathbf{n}}$ with -$\hat{\mathbf{n}}$ did not affect the mirror matrix. By contrast, here we know that replacing $\hat{\mathbf{w}}$ with -$\hat{\mathbf{w}}$ specifies a rotation in the opposite direction. What specific term(s) in equation 5.4 change when $\hat{\mathbf{w}}$ is replaced by -$\hat{\mathbf{w}}$? How does it (do they) change?

Of course the 4x4 matrix applying this rotation to points can now be computed as we have seen before by using equation 5.1 with *F*=*B*.

The OpenGL routines `glRotate[d|f]`(*angle*,*x*,*y*,*z*) provide just a slightly less general capability than what we have derived. The rotation axis for this OpenGL routine is defined as the line passing through the origin and the point (*x*,*y*,*z*) [Schreiner, et. al. 08]. In our notation, we can choose the origin as a fixed point of such a rotation (i.e., *F* in equation 5.1), hence **t** = (0,0,0). Therefore, using **t** = **0** and $\hat{\mathbf{w}}$ =(*x*,*y*,*z*), we see that the matrix shown on pages 776-777 of [Schreiner, et. al. 08] (what the authors state the `glRotate` call produces) is a special case of what we have derived here.

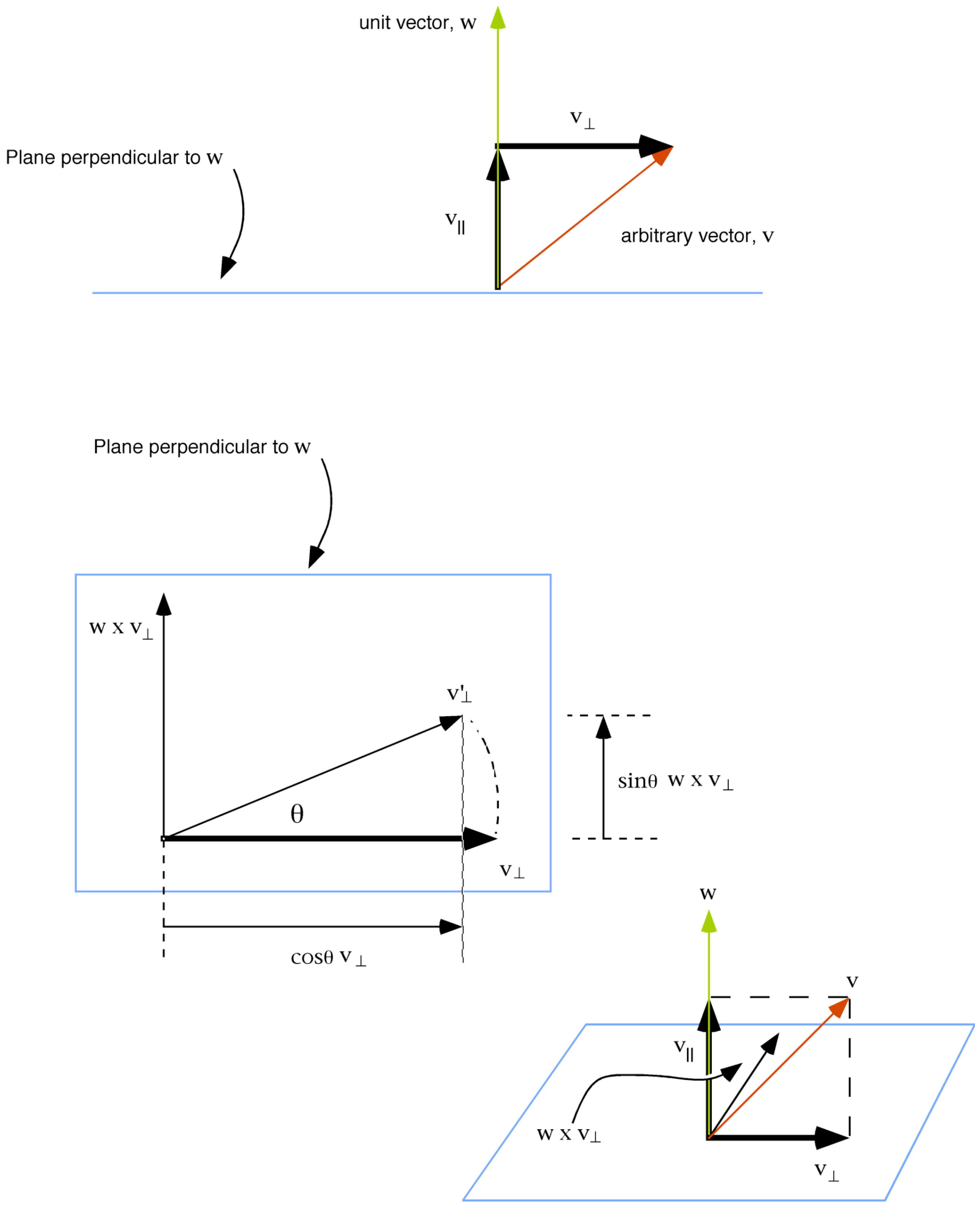


Figure 5.3: Rotating a vector **v** *about a unit axis vector* **w** *by θ*

Speaking of special cases, the table below lists the three basic primitive 3D rotation matrices (rotating about each of the coordinate axes by an angle θ). Coordinate-based derivations for these matrices are straightforward [Foley, et. al. 90].

Rotation About the *x*-axis by θ	Rotation About the *y*-axis by θ	Rotation About the *z*-axis by θ
$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$R_y = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	$R_z = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Table 5.3

Exercise: Show how the matrix for general rotations (equation 5.3 with **M** defined as shown in equation 5.4 and **t** as defined in equation 5.1) specializes to the matrices of Table 5.1. What do you use for *B* and $\hat{\mathbf{w}}$ in these three cases?

Exercise: Show the 4x4 rotation matrix which results from *B* = (0,-2,1), **w** = (-1,0,2), and θ = 30 degrees.

V.3.1 Representing an Arbitrary Rigid Transformation as a General Rotation (Optional)

Consider an affine transformation (**R**, **δ**) where **R** is a 3x3 matrix describing some rigid transformation, and **δ** is a vector. (Recall the definition of rigid transformations as presented in Section III.7.)

Here is a fact you probably did not know. For any such matrix, **R**, there exists a unit axis vector $\hat{\mathbf{w}}$ and an angle θ such that the 3x3 matrix **M** generated by equation 5.4 exactly matches **R**. Moreover, there is a point *B* and a (possibly zero) translation vector **t** such that the composite affine transformation which results from the rotation (*B*, $\hat{\mathbf{w}}$, θ) *followed by* a translation by **t** is equivalent to the original affine transformation (**R**, **δ**).

In other words, regardless of how **R** was computed (it need only be orthogonal and right-handed), the transformation it describes on vectors is equivalent to a rotation about the unit axis vector $\hat{\mathbf{w}}$ by θ. Furthermore, the transformation that (**R**, **δ**) describes on affine points is equivalent to first rotating them about (*B*, $\hat{\mathbf{w}}$) by θ, then translating them by the (possibly zero) vector **t**.

Let's make sure we understand what this means. Consider as an example the sequence of *m* transformations depicted in Figure 5.4(a), and suppose we construct our affine transformation (**R**, **δ**) by concatenating these *m* transformations. Let us assume that each of the X_i is a rigid transformation (i.e., either a rotation or a translation). Suppose the 4x4 matrices encoding each of these *m* transformations are stored in $\mathbf{M_1}, \ldots, \mathbf{M}_m$, respectively, and suppose we compute $\mathbf{R_{4x4}}$ as:

$$\mathbf{R_{4x4}} = \mathbf{M_m} * \cdots * \mathbf{M_1}$$

Exercise: Explain the order of multiplication in this equation.

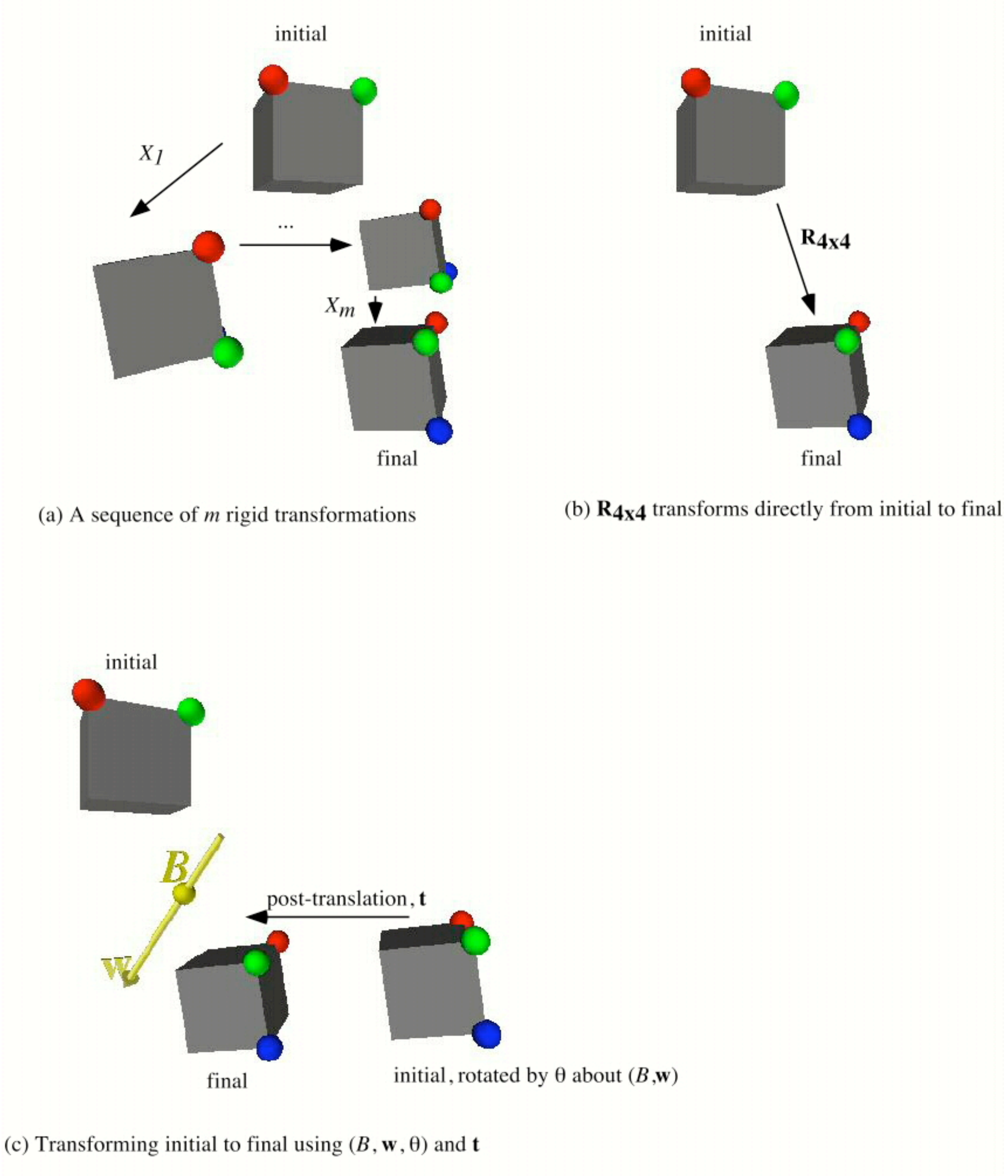

initial
X_1
...
X_m
final
(a) A sequence of m rigid transformations
initial
$\mathbf{R}_{4x4}$
final
(b) $\mathbf{R}_{4x4}$ transforms directly from initial to final
initial
B
$\mathbf{w}$
post-translation, $\mathbf{t}$
final
initial, rotated by θ about $(B, \mathbf{w})$
(c) Transforming initial to final using $(B, \mathbf{w}, \theta)$ and $\mathbf{t}$

Figure 5.4: Extracting Rotation Axis and Angle from Rigid Transformations

Since we chose to compute the transformation $(\mathbf{R}, \boldsymbol{\delta})$ by concatenating these m rigid transformations, we also know that

$$\mathbf{R}_{4x4} = \begin{pmatrix} & \mathbf{R} & & \delta \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Therefore, as implied in Figure 5.4(b), the matrix $\mathbf{R}_{4x4}$ will transform the cube from its original position to its final position directly. We knew this already. What is more interesting (i.e., the "fact you probably did not know") is that, as implied in Figure 5.4(c), there is a point B, a unit vector $\hat{\mathbf{w}}$, an angle θ, and a possibly zero vector **t** such that the rotation determined by (B, $\hat{\mathbf{w}}$, θ) followed by a translation by **t** will also transform the cube from its original position to its final position.

Let us see how we can determine B, $\hat{\mathbf{w}}$, θ, and **t** given only (**R**, **δ**).

Eigenvector analysis tells us how to find $\hat{\mathbf{w}}$. Given an nxn matrix **N**, if there is a scalar λ and an n-dimensional vector **e** such that **Ne** = λ**e**, then we say that λ is an eigenvalue of **N** and **e** is the corresponding eigenvector. We find the eigenvalues by rewriting the equation as (λ**I** - **N**)**e** = **0**. If we consider this as an equation in the unknown variable **e**, then there can be nontrivial solutions (i.e., solutions where **e**≠**0**) only if the determinant of (λ**I** - **N**) is zero. If this determinant is zero, then at most *n-1* of the *n* rows of (λ**I** - **N**) are linearly independent.

For the problem at hand, n=3, and the 3x3 matrix **N** whose eigenvectors we seek is the transformation matrix **R**. Any eigenvector of **R** corresponding to an eigenvalue λ=1 will give the direction of the rotation axis (i.e., the significance of an **e** which satisfies the equation is that it will be the desired 3D axis vector **w**).

Exercise: Explain why such an **e** would be the desired 3D axis vector.

Using **R** for **N** and substituting λ=1, our 3D eigenvector equation becomes: (**I** - **R**)**w** = **0**. A vector **w** satisfying this equation is not unique; however a *unit* vector satisfying (**I** - **R**)**w** = **0** *is* unique up to a sign. We can compute such a vector by observing that specifying "(**I** - **R**)**w** = **0**" is equivalent to stating that each of the three rows of (**I** - **R**) is either the zero vector, **0**, or a non-zero vector perpendicular to **w**.

Exercise: Explain why this observation is true.

Therefore we can compute **w** by forming the cross product of any two linearly independent rows of (**I** - **R**). Normalizing this vector **w** yields a suitable unit axis vector $\hat{\mathbf{w}}$.

Exercise: How would you determine which two rows of (**I** - **R**) to use in order to maximize the numerical accuracy of the computed **w**? Justify your answer.

Given $\hat{\mathbf{w}}$, we can compute cosθ using equation 5.4. Notice that diagonal terms are of the form: "cosθ + (1-cosθ)w_i^2". The sum of the three diagonal elements (called the *trace* of the matrix) is therefore:

$$trace(\mathbf{M}) = M_{1,1} + M_{2,2} + M_{3,3} = 2\cos\theta + 1$$

Finally, any off-diagonal element $\mathbf{R}_{i,j}$ corresponding to a non-zero element $\mathbf{W}_{i,j}$ can be used to solve for sinθ since we know that $\mathbf{R}_{i,j} = (1-\cos\theta)w_iw_j + \sin\theta\mathbf{W}_{i,j}$ for any $i \neq j$.

> ***Exercise***: How would you determine which off-diagonal element is best to use in order to maximize the numerical accuracy of the computed sinθ? Justify your answer.

If a base point B exists which satisfies $(\mathbf{I} - \mathbf{M})B = \boldsymbol{\delta}$, then the vector $\mathbf{t}$ defining the post-translation is the zero vector.

> ***Exercise***: Use equation 5.1 to justify this statement. (*Warning*: Because of the context of that equation as compared to the context of this analysis, the vector "**t**" in equation 5.1 is the vector "**δ**" here.)

Since $(\mathbf{I} - \mathbf{M})$ is singular, however, "$(\mathbf{I} - \mathbf{M})B = \boldsymbol{\delta}$" represents at most two linearly independent equations in the three unknowns B_x, B_y, and B_z. Hence we have an over-constrained system for which a solution (i.e., an exact base point B) may not exist. Our approach is to construct a base point which will satisfy at least two of the three equations. We then compute the post-translation vector $\mathbf{t} = \boldsymbol{\delta} - (\mathbf{I} - \mathbf{M})B$. In the event that B <u>is</u> an exact base point, then clearly $\mathbf{t}$ will be $\mathbf{0}$.

The approach to computing B is very much like that used to find $\hat{\mathbf{w}}$. We begin by observing that we can arbitrarily prescribe one of the coordinates of B.

> ***Exercise***: Explain why we are able to arbitrarily prescribe a coordinate of B. For example, if x is the coordinate we choose to prescribe, we are saying that we can set B_x to anything we want, and then compute the remaining two coordinates B_y and B_z, given this choice for B_x. Include in your argument a characterization of *which* coordinates are candidates, and a rationale for selecting the *best* candidate so as to maximize the numerical accuracy of the point we ultimately compute for B.

> ***Exercise***: Suppose we set this coordinate to zero. Show how a cross product can be used to compute the other two coordinates.

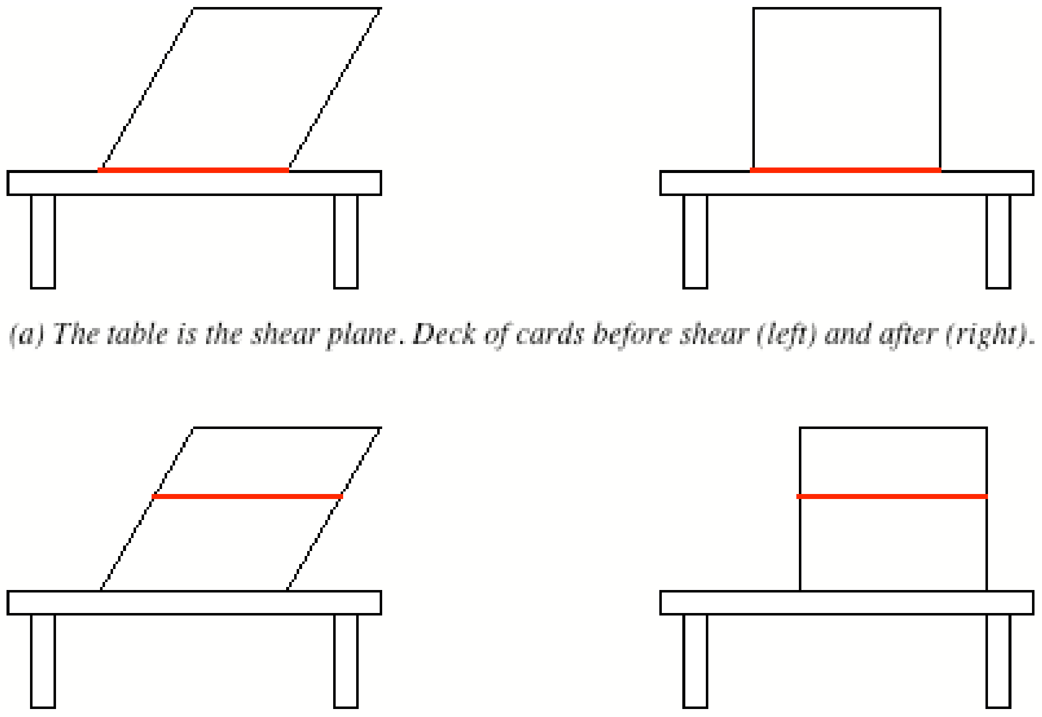
The analysis described here is very general. The example of Figure 5.4 showed taking an $(\mathbf{R}, \boldsymbol{\delta})$ computed by concatenating an arbitrary number of rigid transformations. We could equally well have computed $\mathbf{R}$ by generating three random mutually perpendicular unit vectors and then computed $\boldsymbol{\delta}$ by generating three more random numbers. The analysis described above would still have produced a $(B, \hat{\mathbf{w}}, \theta)$ and post-translation $\mathbf{t}$ describing a transformation equivalent to the one randomly generated.

V.4 Shear Transformations (Optional)

Shear transformations "slide" points in a given direction parallel to a shearing plane by an amount that is directly proportional to their signed distance from the plane. Points in the plane do not move at all (they are fixed points); points on opposite sides of the plane move in opposite directions.

You use a shear transformation every time you "straighten up" a deck of playing cards sitting on a table. If you use the table as the shearing plane as in Figure 5.5(a), all cards except the bottom one slide to the left. If instead we use a card in the middle of the deck for the shear plane as illustrated in Figure 5.5(b), cards in the top half of the deck slide to the left; those in the bottom half slide to the right, and the card in the middle does not move.

It may be difficult to imagine how you could create a 4x4 matrix that would have such an effect. As we will see, our tools from section III will make this straightforward.


(a) The table is the shear plane. Deck of cards before shear (left) and after (right).

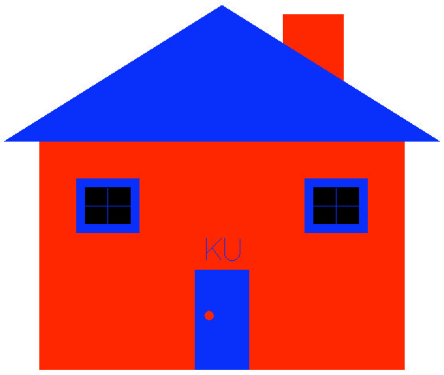
(b) The shear plane passes through a card in the middle of the deck; before shear (left) and after(right).

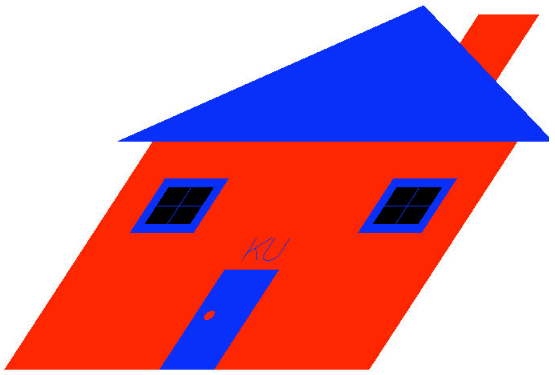
Figure 5.5: Using a shear transformation to straighten a deck of playing cards

Shear transformations are most commonly used in the context of projections (i.e., as a viewing rather than as a modeling transformation). For example, we will study coordinate-based techniques in section VII for constructing a matrix to shear a general oblique view volume into an orthogonal view volume, much like our deck of cards analogy above. In that context, z coordinates will remain unchanged; x and y coordinates will be sheared by an amount proportional to their z coordinate. Specifically, the farther they are away from the projection plane, the more they will be sheared.

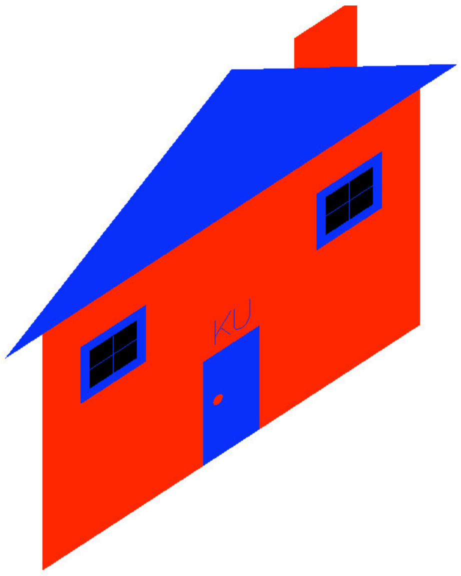
Using shear transformations to construct geometric models (i.e., using a shear transformation as a general modeling transformation) is certainly less common. Nevertheless, studying shears gives us an opportunity to exercise the use of the analytical tools we studied in section III. Moreover – taken together with our coordinate-based schemes in section VII – the derivation emphasizes the fact that the most appropriate analytical approach to a given problem (such as the construction of a matrix) depends upon the specific context. Recall the remarks in section I.

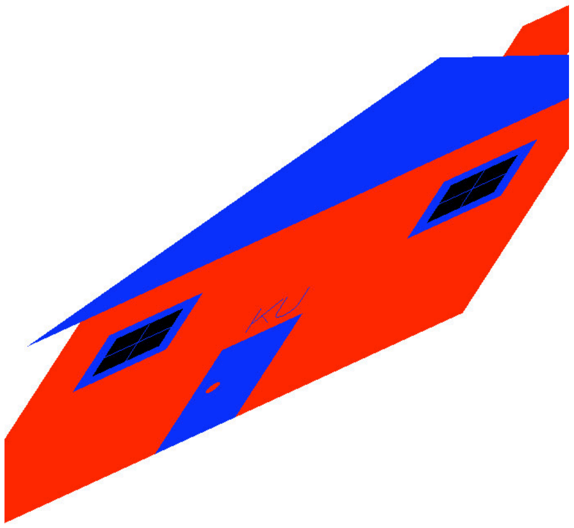
Figure 5.6 illustrates the effect of shear modeling transformations on a house. In Figure 5.6(a), we see the house as defined without any shearing. In Figures 5.6(b) and 5.6(c), we see the house after different shear transformations. In both cases, the shear plane passes through the upper left corner of the door. In 5.6(b), the plane is perpendicular to the ground, and the shear direction is straight up; in 5.6(c), the plane is parallel to the ground, and the shear direction is to the left. Just for fun, Figure 5.6(d) illustrates the house after being sheared first with the transformation used for 5.6(c), then with that used for 5.6(b).





(a) Original house before any shearing *(b) Original house sheared in x direction*





(c) Original house sheared in y direction *(d) House sheared first in y, then in x direction*

We can describe a desired general shear modeling transformation in any of a number of ways. For starters, let us assume we define the shear plane by specifying a point *B* on the plane, and a vector **n** perpendicular to the plane. A vector **u** in the plane will be provided as the shear direction[7]. Refer to Figure 5.7(a). We also must specify the amount of the shear. From what we have described so far, it should be clear that an arbitrary vector **v** will be sheared by an amount proportional to the length of **v** in the direction of **n**. Hence the specification of the shear must include specification of a scalar factor *f* as the constant of proportionality between "length of $\mathbf{v}_{\parallel}$" and "amount of shear".

As an example, we see in Figure 5.7(b) two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ such that $\mathbf{v}_{1,\perp,\hat{\mathbf{n}}} = \mathbf{v}_{2,\perp,\hat{\mathbf{n}}}$, but $\mathbf{v}_{2,\parallel,\hat{\mathbf{n}}} > \mathbf{v}_{1,\parallel,\hat{\mathbf{n}}}$. Notice in Figure 5.7(c) that $\mathbf{v}_2$ gets sheared more in the direction of $\hat{\mathbf{u}}$ than does $\mathbf{v}_1$.

To summarize, our shear transformation will be specified by $(B, \mathbf{n}, \mathbf{u}, f)$. That is, if someone gives us these four parameters, we will give them back the 4x4 matrix performing the shear.

As we have been doing all along, we first characterize how the shear affects an arbitrary vector **v**. To derive the required expression, we normalize **n** and compute the unit vector $\hat{\mathbf{u}}$ in the direction of the component of **u** perpendicular to $\hat{\mathbf{n}}$. Performing these two operations should be very familiar to you by now. Finally, we stipulate that a multiple of $\hat{\mathbf{u}}$ will be added to our arbitrary vector **v**; this multiple will be the constant of proportionality, *f*, times the length of the component of **v** parallel to $\hat{\mathbf{n}}$ (i.e., $\left|\mathbf{v}_{\parallel,\hat{\mathbf{n}}}\right|$).

> ***Exercise***: Explain why the following equation expresses the intent of the final sentence of the previous paragraph.
>
> $$X(\mathbf{v}) = \mathbf{v} + f(\mathbf{v} \cdot \hat{\mathbf{n}})\hat{\mathbf{u}} \tag{5.5}$$
>
> ***Exercise***: Explain why the use of equation 5.5 will shear **v** in the $\hat{\mathbf{u}}$ direction if **v** points to the same side of the plane as does $\hat{\mathbf{n}}$, but will shear **v** in the $-\hat{\mathbf{u}}$ direction otherwise.
>
> ***Exercise***: What is $X(\mathbf{v})$ if **v** lies in the shear plane? Why?
>
> ***Exercise***: Show that equation 5.5 can be written as $X(\mathbf{v})=\mathbf{M}\mathbf{v}$ where
>
> $$\mathbf{M} = \mathbf{I} + f\hat{\mathbf{u}} \otimes \hat{\mathbf{n}}\,. \tag{5.6}$$
>
> ***Exercise***: Show that we obtain the result of equation 5.5 if, as we have been doing, we characterize individually how the components of **v** parallel to and perpendicular to $\hat{\mathbf{n}}$ are affected.

An alternative way to specify the amount of the shearing is to specify the angle α formed between $\hat{\mathbf{n}}$ and $X(\hat{\mathbf{n}})$[8]. That is, we may be told how much $\hat{\mathbf{n}}$ is sheared. See Figure 5.8. Therefore instead of the shear being specified by $(B, \mathbf{n}, \mathbf{u}, f)$, it would be specified by $(B, \mathbf{n}, \mathbf{u}, \alpha)$.

> ***Exercise***: How are *f* and α related? That is, given an *f*, how could I compute an equivalent α, and vice versa?
>
> ***Exercise***: Assume $B=(3,-1,0)$, $\mathbf{n}=(-1,0,1)$, $\mathbf{u}=(1,0,1)$, and $f=2$. Show the 4x4 matrix which shears points about this shearing plane.
>
> ***Exercise***: Suppose we want to use the *yz*-plane as a shear plane and employ the *z* direction as the shear direction with $f=2$. What values do you specify for $(B, \hat{\mathbf{n}}, \hat{\mathbf{u}}, f)$? What assumptions did you make? Show

[7] Since a programmer will be giving us this vector, we cannot assume that it is actually in the plane (i.e., perpendicular to **n**). We therefore will need to actually consider only the component of **u** perpendicular to **n** as we will see shortly.

[8] Yet another angle-based alternative is given on page 152 of [Angel 97]. A shear in the *x* direction is presented there using an angle θ which (in our terms) is the angle between $X(\hat{\mathbf{n}})$ and $\hat{\mathbf{u}}$. See Figure 5.8.

the 4x4 matrix. Show how this matrix transforms the three affine points: (-1,0,0), (0,0,0), and (1,0,0). Are the results what you would expect?

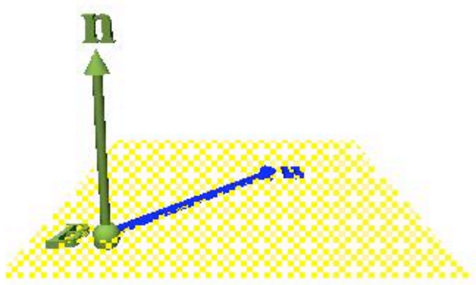


Figure 5.7(a): Shear Definition Vectors

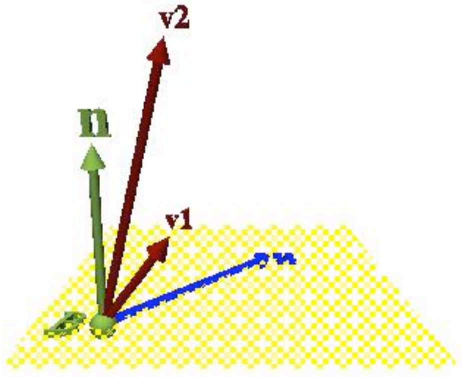


Figure 5.7(b): Two vectors to be sheared

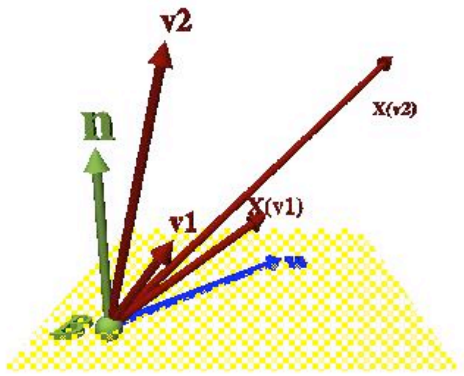


Figure 5.7(c): The effects of the shear on the vectors

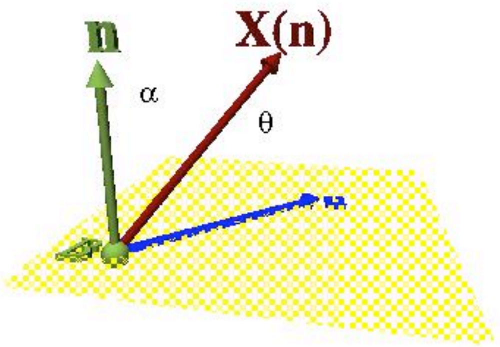


Figure 5.8: Two alternate angle-based shear specifications

It is important to remember that shears are not rigid transformations. The length of a vector will, in general, change when a shear transformation is applied to it. Equivalently, the distance between two points will, in general, change after they have both been transformed by the shearing operation.

> ***Exercise***: Since points in the shear plane are fixed points, the distance between any two points in the shear plane will obviously *not* change when the shear transformation is applied to them. Consider two points *not* in the shear plane. Under what circumstances will the distance between such points remain unchanged after the shear transformation is applied?

Shear transformations can be constructed from a combination of rotation, translation, and scale transformations.

> ***Exercise***: For the various shear transformation specifications mentioned above, show how to create the shear transformation by concatenating combinations of basic affine transformations (i.e., rotation, translation, and scale).

V.5 Summary and Retrospective

Our approach to the development of modeling transformations in this section was based solely on vector geometric analysis. Derivations for these matrices were occasionally tricky, but a major benefit is that the final computer code that one writes is much simpler, more compact, and free of nasty special case handling. It typically involves fewer arithmetic operations as well.

I will not obsess here with exhaustive operation counts, but I will make some general observations to justify the claim that fewer arithmetic operations are usually required. Consider equation 5.4 for the upper 3x3 part of the general rotation matrix. Given the sine and cosine of the rotation angle (required by any approach), it takes 36 multiplies and 19 adds for a completely dumb implementation of equation 5.4. By "dumb" I mean that we do not try to exploit any special properties of the terms in 5.4 such as the symmetry of $\hat{\mathbf{w}} \otimes \hat{\mathbf{w}}$, the zeros in the cross product matrix **W**, or the fact that $\cos\theta\mathbf{I}$ can actually be generated with no multiplications. If instead we symbolically expand equation 5.4 and implement the result, we can compute **M** with just 15 multiplies and 10 adds. Computing the translation vector **t** with equation 5.1 then requires an additional 9 multiplies and 9 adds. Therefore the total required to compute (**M**, **t**) (and hence the full 4x4 rotation matrix) is 45 multiplies and 28 adds for the dumb approach, or 24 multiplies and 19 adds for the optimized approach.

The traditional coordinate-based algorithm described in the standard texts embodies a "reduce to the previously solved problem" approach. It works by combining primitive affine transformations to achieve more general ones. A completely dumb implementation of the combination of two affine transformations entails multiplying the two 4x4 matrices describing them. This step requires 64 multiplies and 48 adds. Recognizing that concatenating two affine transformations really only requires multiplying their respective (**M**, **t**), we observe that we can optimize concatenation of two affine transformations so that it requires only 36 multiplies and 27 adds.

As noted, this traditional "reduce to the previously solved problem" approach of combining primitive affine transformations to achieve more general ones requires *several* concatenations of affine transformations. To do a rotation about a general axis in space, for example, one needs to concatenate *seven* affine transformations (i.e., perform *six* matrix multiplications). Therefore, assuming the optimized affine transformation implementation, it requires 216 multiplies and 162 adds to create a general rotation matrix. Clearly this is far more expensive than our optimized vector geometric approach which by comparison required only 24 multiplies and 19 adds.

Don't forget that the traditional approach is even more complex in that the construction of the various matrices has to be "guarded" by checking for certain special cases like distances computed for denominators being zero or nearly zero. By contrast, once we determine that we have not been given the zero vector for a rotation axis (analogous to the initial test in the *BuildMirrorTransformation* function at the end of section V.2), no special case detection of any sort is required to implement our vector geometric equation for **M**. Therefore, while it is true that the generation of a "primitive" rotation like those in Table 5.1 is best implemented by directly writing code to construct the appropriate matrix, anything more general is best done with an implementation of equations 5.4 and 5.1.

VI. View Orientation Transformations

I argued in section IV that vector geometric approaches for modeling transformations were useful because there were no special relationships between source and target coordinate systems that we could exploit, yet there were potential coincidental relationships that coordinate-based techniques would have to detect and handle. Moreover, there was little to be gained by attempting to exploit properties of either coordinate system since no operations other than subsequent transformations to other coordinate systems were carried out in them.

However, the situation changes at the stage of the pipeline where viewing transformations begin. We have a completely assembled geometric model of which we want to generate a view. Realistic images of 3D models typically require one or more of the following effects: hidden line or hidden surface removal; intensity depth cueing; depth clipping; perspective foreshortening; continuous tone shading; stereo projections; and others. While it is not our intent to discuss these and other techniques here (see, for example, standard graphics texts such as [Foley, et. al. 90; Watt & Watt 92; Angel 09]), it is prudent at this point to consider common mathematical requirements and what is done in viewing pipelines to facilitate them.

Let's consider an example. A fundamental operation in hidden line elimination is to determine which of two points P and Q is closer to the eye. Let us assume that E is the position of the eye and $\hat{\mathbf{e}}$ is the unit vector along the line of sight, both E and $\hat{\mathbf{e}}$ being represented in world coordinates. Now if P and Q are also represented in world coordinates, we would have to compute $d_P = (P\text{-}E)\cdot\hat{\mathbf{e}}$ and $d_Q = (Q\text{-}E)\cdot\hat{\mathbf{e}}$. The smaller of d_P and d_Q would then identify the point closer to the eye.

Each such computation of a "*d*" requires 3 multiplies and 5 adds. If P and Q had been transformed into a coordinate system in which, say, the eye was at the origin and the line of sight was along the negative z-axis, then *no* "*d*" computations would be required at all, and we would need only compare P_z to Q_z. Such comparisons are typically performed a large number of times in hidden line elimination algorithms (e.g., $O(n^2)$ where n is the number of surface elements in the scene). It therefore pays to transform geometry into such a coordinate system to save this large number of floating point computations.

This is not an isolated example. It turns out that a large number of the operations required to achieve realistic images in interactive graphics systems are view-dependent and can be most efficiently implemented in an "eye coordinate system" such as that mentioned in the previous paragraph. This fact is the major motivation for the view orientation transformation described in this section.

To avoid confusion over the labeling of the axes in the world and eye coordinate systems, let us associate the labels u, v, and n with the eye coordinate system axes. Then our task is to determine how to transform (x,y,z) world coordinates into (u,v,n) eye coordinates. We shall assume that the eye is at the origin of the eye coordinate system, the $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$ axes are parallel to the horizontal and vertical directions of the display surface, and the $\hat{\mathbf{n}}$ axis is perpendicular to the display. The $\hat{\mathbf{u}}\hat{\mathbf{v}}\hat{\mathbf{n}}$ axes form a right-handed orthonormal coordinate system in which $-\hat{\mathbf{n}}$ is the unit vector along the line of sight.

The eye coordinate system conventions described in the previous paragraph are typical of those in many graphics systems such as OpenGL. As we will see section VI.3, PHIGS uses a so-called view reference coordinate system which represents a slight generalization of these conventions.

To better understand the eye coordinate system and to begin to get a feel for how we can derive the matrix which represents the affine map from world coordinates to eye coordinates, you can imagine the $\hat{\mathbf{u}}\hat{\mathbf{v}}$-plane as a display screen which you drag around inside the 3D world coordinate system. Any given position and orientation of this display screen determines an eye system.

Numerically, we will describe the horizontal and vertical directions of this screen by computing the world coordinate descriptions of the vectors $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$. (Since the vectors define a right-handed orthonormal coordinate system, the third vector can be computed from a cross product once any two are known.)

Before studying specifically how we compute these numerical descriptions, let's first see how we will use them once we have them. That is, suppose we know the world coordinate descriptions of unit vectors $\hat{\mathbf{u}}$, $\hat{\mathbf{v}}$, and $\hat{\mathbf{n}}$. How can we use them to map an arbitrary vector **m** represented in world coordinates into its representation in eye coordinates?

To be able to answer this question, we need to understand the geometric significance of the components of a vector. That is, what is the significance of, say, m_x? It is simply the length of the vector **m** in the direction of the world coordinate *x* axis. That language should sound very familiar.

In section III.8.1 we read that the dot product of an arbitrary vector **m** with a unit vector was the length of **m** in the direction of the unit vector. Hence, $m_x = \mathbf{m}\cdot\hat{\mathbf{i}}$. It follows, then, that we can compute the components of **m** with respect to our eye coordinate system axes simply by computing dot products with the unit vectors $\hat{\mathbf{u}}$, $\hat{\mathbf{v}}$, and $\hat{\mathbf{n}}$. Specifically, $\mathbf{m}_{eye} = (\mathbf{m}\cdot\hat{\mathbf{u}}, \mathbf{m}\cdot\hat{\mathbf{v}}, \mathbf{m}\cdot\hat{\mathbf{n}})$. This means the three rows of the matrix **M** representing the view orientation affine transformation are simply these three unit vectors:

$$\mathbf{M} = \begin{pmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ n_x & n_y & n_z \end{pmatrix} \tag{6.1}$$

Hence determining the three unit vectors $\hat{\mathbf{u}}$, $\hat{\mathbf{v}}$, and $\hat{\mathbf{n}}$ represents completion of step 1 of "General Strategy #1" from section V. Given these vectors, we must use a fixed point to determine the translation vector of the affine transformation. What can we use as a fixed point?

Uh-oh.

In general there is no fixed point in a view orientation transformation. Imagine, for example, placing our display screen so that it is parallel to the *xy* plane, but translated away from the origin. There would be no point whose world coordinates would be the same as its eye coordinates.

VI.1 Generalizing Our General Strategy

For situations like we just encountered where there is no fixed point, a minor variation oftentimes works. Go back and review the discussion that led up to equation 5.1. Notice in particular that what was important about our fixed point *F* was that we knew what *X*(*F*) was. That is, we knew the representation of *F* in the target coordinate system. In fact, its representation was the same in the source and target systems; that is, *X*(*F*)=*F*.

Suppose we know a point Q in the source coordinate system, and we know the point Q' to which it gets mapped. Then a slight variation of equation 5.1 yields:

$$\begin{aligned} X(Q) &= Q' = \mathbf{M}Q + \mathbf{t} \\ \Rightarrow \mathbf{t} &= Q' - \mathbf{M}Q \end{aligned} \tag{6.2}$$

This derivation is applicable to any affine transformation for which we know such a Q and a Q'. In the context of the eye coordinate system specification, it is common for graphics APIs to require that programmers specify the world coordinates for the position of the viewer (the point E introduced at the start of section VI).

To what point does E get mapped by the view orientation transformation? According to the definition of the eye coordinate system, E must get mapped to (0,0,0) in the eye system.

Let us therefore consider the following generalization of General Strategy #1:

General Strategy #2
1. Determine a 3x3 matrix **M** which applies the affine transformation to 3D vectors.
2. Use a point whose coordinates are known before and after application of X (i.e., use a point whose coordinates are known in the source and target coordinate systems) to compute the translation component **t** of the affine transformation as in equation 6.2.
3. Use **M** and **t** to build $\mathbf{M}_{4x4}$ as in equation 3.8, setting **B**=(0,0,0) and S=1.

VI.2 Derivation of the OpenGL Viewing Transformation

The input parameters to the OpenGL `gluLookAt` routine are E=(*eyeX*, *eyeY*, *eyeZ*), C=(*centerX*, *centerY*, *centerZ*), and **up**=(*upX*, *upY*, *upZ*). The point C is an arbitrary point along the line of sight. It should therefore be clear that we can compute **n** = E - C (recall that -**n** is along the line of sight), and we normalize **n** to get $\hat{\mathbf{n}}$. The unit vector $\hat{\mathbf{v}}$ is defined as the unit vector in the direction of the component of **up** perpendicular to $\hat{\mathbf{n}}$ (recall equation 3.9). Finally, $\hat{\mathbf{u}}$ is simply the cross product of $\hat{\mathbf{v}}$ and $\hat{\mathbf{n}}$. Equation 6.1 specifies how to construct the 3x3 matrix **M** from these three unit vectors. Using equation 6.2 with Q=E, and Q'=(0,0,0) completes the specification of the affine transformation.

> ***Exercise***: Determine the 4x4 matrix which results from each of the following sets of parameters:
>
> - E=(0,0,0), C=(0,0,10), **up**=(0,1,0).
> - E=(3,1,10), C=(3,1,0), **up**=(0,1,0).
> - E=(3,1,10), C=(3,1,0), **up**=(1,0,0).
> - E=(7,3,-1), C=(0,3,9), **up**=(1,0,1).

VI.3 The PHIGS View Reference Coordinate System and View Orientation Transformation

PHIGS does not use an eye coordinate system per se, rather the programmer defines parameters for a *view reference coordinate system* (VRC). The VRC coordinate system is used as a reference system (hence the name) in which subsequent viewing information (including the actual position of the eye) is defined. As with OpenGL, the $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$ axes of VRC are parallel to the horizontal and vertical directions of the display surface, and the $\hat{\mathbf{n}}$ axis is perpendicular to the display. The difference is that -$\hat{\mathbf{n}}$ – while directly related to – is not necessarily parallel to any line of sight.

The input parameters to the PHIGS `evaluate_view_orientation_matrix` are the *view reference point V*, the *view plane normal* **vpn**, and the *view up vector* **up**. Like the point *E* in section VI.2, the point *V* is defined as the origin of VRC, hence it is mapped to (0,0,0) in VRC. (Again the difference is that *V* will not necessarily be the actual position of the eye.) The **vpn** vector is used to specify the $\hat{\mathbf{n}}$ axis direction. Specifically, the unit vector $\hat{\mathbf{n}}$ is obtained by normalizing **vpn**. PHIGS' **up** vector is interpreted exactly as OpenGL's **up** vector in `gluLookAt`.

> ***Exercise***: Determine the 4x4 matrix which results from each of the following sets of parameters:
>
> - *V*=(0,0,0), **vpn**=(0,0,1), **up**=(0,1,0).
> - *V*=(4,2,3), **vpn**=(0,0,1), **up**=(0,1,0).
> - *V*=(4,2,3), **vpn**=(0,0,1), **up**=(1,0,0).
> - *V*=(6,3,-17), **vpn**=(0,1,-1), **up**=(1,0,1).

VII. Projections and Device Mappings

To this point, we have been studying a succession of coordinate systems, their uses, and how to compute the transformations from one to the next: modeling coordinate systems, the world coordinate system, and the eye coordinate system. In all of these, the units along the axes have been those used by the application (e.g., centimeters, miles, angstroms, hours). In terms of coordinate systems, the final steps of the graphics pipeline involve projecting the 3D geometry onto a plane and mapping the units from those used by the application to those used by the output device. These steps may be done in either order, and the mapping to device units is often done in two steps, mapping first to a "logical device" whose dimensions along each axis are [0..1] (as in PHIGS) or [-1..+1] (as in OpenGL). If such a logical device is used, clipping is normally performed in that space, then a simple scaling and offset is performed to map the clipped model to the desired portion of the display surface.

The process of mapping a source rectilinear region of space (defined, for example, in application units in the eye coordinate system) to a destination rectilinear region of space on either a logical or physical device is specified using a so-called window-viewport transformation. Regardless of whether we use a logical device space or a physical device space (e.g., pixels), application units are left behind following application of the window-viewport transformation.

We begin this section by studying the window-viewport transformation. These transformations are fairly easy to understand, and they are often embedded in the matrices implementing the projection operation. Following coverage of the window-viewport transformation, the remainder of this section will be devoted to projections. We shall review the major types of projections, study the basic steps required to implement a projection, and finally derive projection transformation matrices, relating them to the graphics APIs we have been following: OpenGL and PHIGS.

VII.1 Window-Viewport Transformations

Window-viewport transformations are a common means of specifying a transformation between two coordinate systems defined by specifying how an nD rectangle in one is to be mapped onto an nD rectangle in the other. In 2D graphics packages, for example, the programmer might specify that the rectangle $-10 \le x \le 15$ and $30 \le y \le 55$ is to be mapped onto a portion of the screen whose pixel limits are $200 \le x \le 500$ and $200 \le y \le 500$.

An issue which we shall largely ignore here relates to aspect ratios. For any rectangular region (a window, a viewport, a pixel, etc.), the aspect ratio is defined as the ratio of the height of the region to its width. The window-viewport transformations as constructed below will blindly map the corners of the window to the corners of the viewport. If the aspect ratios of these two regions differ (taking into account not only their dimensions, but also the aspect ratio of the pixels), then the image will be scaled differently along the x and y directions, producing a distorted image on the device. Some graphics APIs leave this totally up to the programmer to control, others automatically prevent such distortion. PHIGS does a little of both: in its mapping to logical device space, any distortion introduced by the programmer is permitted; later in the workstation window-viewport transformation (the mapping from the logical device to a physical device), PHIGS will utilize only the largest portion of the physical device viewport whose aspect ratio matches that of the window defined in the logical device.

Since the aspect ratios of the window and the viewport in the example in the first paragraph of this section are both "1" (assuming square pixels), no distortion will be introduced by that mapping. If instead the y pixel limits were $200 \le y \le 350$, then the image would be stretched twice as much in x as in y.

Let us now focus on 3D window-viewport mappings as might be found in OpenGL or PHIGS. To re-establish our context in terms of where we left off in section VI, we shall assume we define our window in the *uvn* eye coordinate system. The 6-tuple $\mathbf{W} = (umin, umax, vmin, vmax, nmin, nmax)$ defines the 3D window, and a corresponding 6-tuple $\mathbf{V} = (xmin, xmax, ymin, ymax, zmin, zmax)$ defines the viewport. For our purposes here, **V** will be a region in either a logical or a physical device, but recall that window-viewport transformations can be used to specify a transformation between any two coordinate systems[9].

Depending on the other projection parameters, **W** may be a parallelepiped (orthogonal or sheared), or a frustum (again, orthogonal or sheared). However, the window-viewport transformation is defined as a linear transformation which only scales and translates. In 3D systems where sheared windows and/or windows which are frustums appear, we shall see that appropriate transformations will be applied first which will shear and/or stretch nonlinearly the window **W** so that it becomes an orthogonal parallelepiped. Only then will the window-viewport transformation as described below be created and applied.

For discussion here, we will use c to denote a generic window coordinate (c being one of u, v, or n) and C to denote a generic viewport coordinate (C being one of x, y, or z).

In PHIGS, $0 \le C \le 1$. Programmers can specify multiple viewports in a device independent fashion on this logical device, and the workstation transformation (discussed in section VIII) later maps this logical device layout to a physical device. To support this, programmers are given tools which allow them to specify arbitrary subsets of this 0..1 logical device when generating view mapping transformations. OpenGL assumes $-1 \le C \le +1$, but programmers never "see" this coordinate space. That is, the OpenGL projection matrices map the viewing window into this viewport, but then the "true" viewport which maps this entire space onto a physical device viewport is applied. In OpenGL, programmers are never allowed to specify subsets of the OpenGL logical device.

The window-viewport mapping operates independently in the three coordinate directions. That is, given window limits (*cmin*, *cmax*) and corresponding viewport limits (*Cmin*, *Cmax*) in corresponding directions c and C, the viewport coordinate corresponding to a window coordinate is computed as $C = a_c c + b_c$ where a_c and b_c are uniquely determined from these corresponding limits. For example, the following two equations can be solved for a_c and b_c in each coordinate direction to determine the constants defining the window-viewport map:

$$Cmin = a_c cmin + b_c$$

$$Cmax = a_c cmax + b_c$$

With one exception, we will therefore simply use a_c and b_c in the derivations below without

[9] Recall, for example, we stated above that PHIGS uses two window-viewport transformations. The first employs a **W** defined in view reference coordinates and a **V** defined in its [0..1] logical device space. In the second (the workstation transformation) **W** is defined in the [0..1] logical device space, and **V** is a region on the physical device. A similar mechanism is used in OpenGL, but only the former is really reflected in any matrices placed on OpenGL matrix stacks.

further comment. The exception is that we will see a slight variation in the *n* direction when dealing with perspective projections.

It should be clear that the derivations of this section are independent of whether **V** is measured in pixels, fixed at (-1,1 , -1,1 , -1,1) in OpenGL's logical device space, set to some subset of PHIGS' 0..1 space, or anything else. However we will use our understanding of choices such as these later when examining how the matrices shown in other references (e.g., [Schreiner, et. al. 08; Foley, et. al. 90]) relate to those developed here.

> ***Exercise***: Given window coordinate limits **W** = (-1,4 , 0,8 , -5,-1) and corresponding logical device viewport limits **V** = (0.0,0.2 , 0.6,0.8 , 0.0,0.3), derive the six constants of the window-viewport transformation a_u, b_u, a_v, b_v, a_n, b_n. (Assume **W** defines an orthogonal parallelepiped.) Does this window-viewport map preserve aspect ratios? Why or why not?
>
> ***Exercise***: Repeat the previous exercise using the 2D window and viewport limits stated in the first paragraph of this section.
>
> ***Exercise***: Repeat the previous exercise, but use the modified *y* limits as suggested in the third paragraph of this section: $200 \le y \le 350$.

VII.2 Types of Projections

Planar Geometric Projections refer to mappings of 3D geometry onto an infinite plane using a family of straight lines called *projectors*. The reader is strongly encouraged to read the excellent survey paper by Carlbom and Paciorek which describes the history, mathematics, and common uses of planar geometric projections [Carlbom & Paciorek 78]. In terms of the taxonomy of projections developed in that paper, the two most important and most commonly supported in graphics systems are *parallel* and *perspective* projections. *Orthogonal* and *oblique* projections are special cases of parallel projections.

As the name implies, the family of straight lines used to implement parallel projections are all parallel to one another. (For obvious reasons, they *cannot* be parallel to the projection plane.) By contrast, the family of lines used to implement perspective projections all share a common point. In the context of graphics system view specifications, this common point is the position of the viewer.

The role played by these projectors in the generation of a view is to simulate a line of sight. For each visible point on an object, the projection line which contains that point is followed to where it meets the projection plane. The collection of all such projected visible object points constitutes the image as seen by the viewer.

In interactive graphics systems based on linear objects (points, lines, and polygons), only vertices are actually projected this way. This is the context we will implicitly assume for the remainder of this section. In other rendering strategies (e.g., ray tracing), the focus is *not* on vertices, rather points in a dense sampling pattern across the interior of visible surface elements are projected in this fashion. While conceptually the same idea, the implementation details are much different, due to the overall operation typical of ray tracing algorithms.

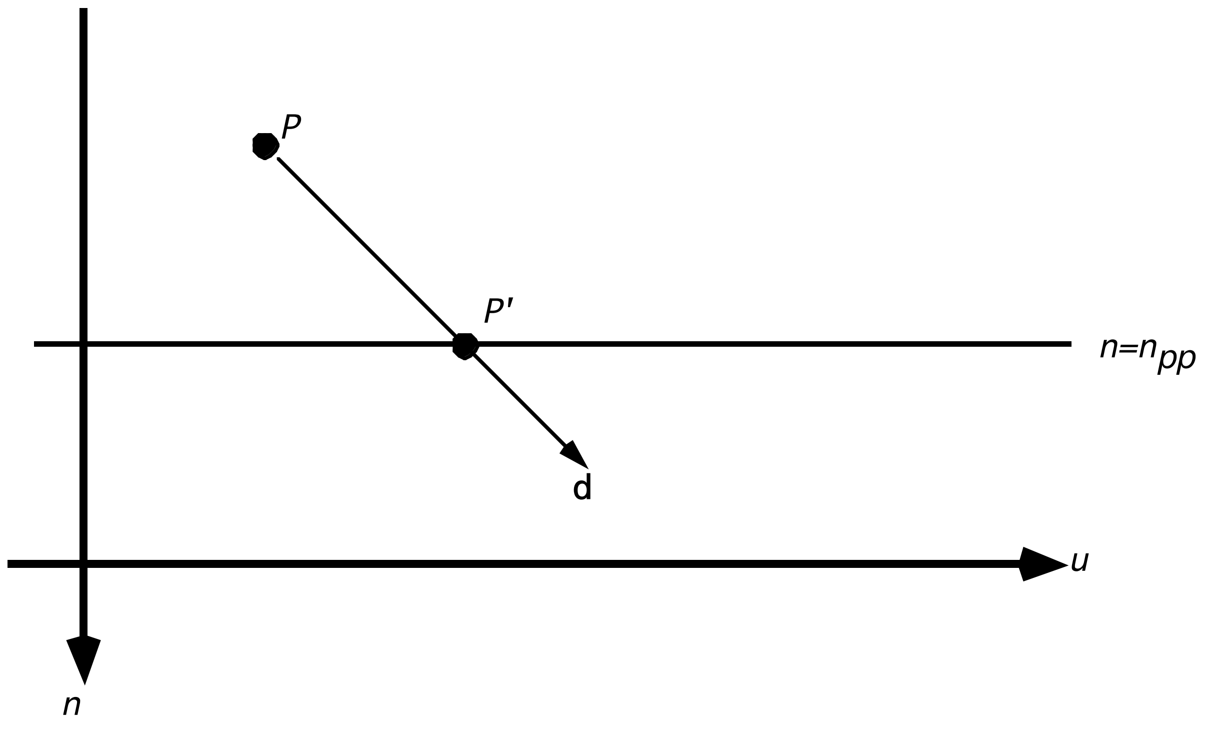


Figure 7.1: Projecting a point onto a projection plane

One of the purposes of converting to an eye coordinate system was that it simplified subsequent view-dependent operations like projections in interactive systems., In particular, the projection plane in this coordinate system is a plane of constant n. We will refer to this value as $n = n_{pp}$. Figure 7.1 is a view of the uv-plane of an eye or view reference coordinate system. Since we are looking in a direction perpendicular to the n-axis, the projection plane $n{=}n_{pp}$ appears as a line. This figure illustrates a generic vertex, P, a projection vector, $\mathbf{d}$, and the point P' to which P projects. (At this point, it doesn't matter whether this is a parallel or perspective projection. We will return to this idea shortly, but for now, we require only that $d_n \neq 0$.) We wish to compute P' from P, $\mathbf{d}$, and n_{pp}. Stated another way, we wish to compute the scalar, r, such that:

$$P' = P + r\mathbf{d} \tag{7.1}$$

Recalling that this represents three equations (one each in u, v, and n) and recalling that the n coordinate of P' must be n_{pp}, we get:

$$P'_n = n_{pp} = P_n + rd_n \Rightarrow r = (n_{pp} - P_n) / d_n \tag{7.2}$$

Since n_{pp} and d_n are constants, we can compute P' from P as required. As mentioned above, this analysis does not depend on whether parallel or perspective projections are involved. In this context, these projections differ only in how $\mathbf{d}$ (and hence d_n) is computed.

PHIGS provides the programmer with tools for creating matrices applying both general parallel as well as general perspective projections. By contrast, OpenGL provides matrix generation tools in its API only for orthogonal and general perspective projections. However both the OpenGL and PHIGS APIs allow arbitrary matrices to be specified for the projection operation, hence the programmer who understands the derivations presented in this section can compute matrices for use in an OpenGL program which accomplish any arbitrary projection.

Following an examination of desired properties of projections and their coordinate systems, we will proceed to develop the matrices used to implement projections in interactive graphics systems such as OpenGL or PHIGS. We will see that parallel projections which preserve depth information can be implemented as affine maps. Perspective projections, on the other hand, cannot be implemented with affine transformations. We shall see that we need to employ a more general projective transformation to accomplish this task.

VII.3 Desired Properties of the Projection Coordinate System

In section VI, we discussed how geometry was transformed into an eye (or view reference) coordinate system. We argued there that transforming into this special system allowed us to simplify a whole host of image rendering effects. An important example was visible line and visible surface determination. Major simplifications and algorithm speedups are enabled by decomposing the general 3D logic required into independent sets of 1D and 2D questions which are both simpler to answer and simpler to organize.

In visible surface determination, the critical 2D question is to determine whether two objects A and B overlap on the screen. Clearly this is a necessary condition for one to obscure the other. We would like to be able to make this determination based strictly on comparing their x and y coordinates after the projection transformation has been applied. Since the window view volume may have an arbitrary shape, however, we will have to make sure that the projection transformation distorts the window so that it maps into the orthogonal viewport volume **V** in such a fashion that screen position (and hence overlap) is determined solely by x and y coordinates. Furthermore, even though the analysis is based on projecting geometry onto the projection plane, we do not want our object to be flattened by the projection transformation. Instead we want the transformation to preserve exact depth relationships if possible, and at least preserve relative depth relationships otherwise. We shall see that this requires a shear transformation in the case of parallel projections and a special nonlinear transformation in the case of perspective.

As implied at the start of section VII, we will derive the various projection transformation matrices to accomplish both the projection operation *and* the window to viewport map. Following application of this matrix, coordinates will therefore be represented in either a logical or a physical device space, depending on the units used in the viewport specification.

To summarize, after the projection transformation has been applied, the graphics system need only look at the final x and y coordinates to draw geometry on the screen. Depth information will be preserved in the z coordinates so that we can perform not only the depth comparisons required for visible line and surface determination, but also calculations required for other effects such as intensity depth cueing.

VII.4 Parallel Projections, Part 1

We shall first study the simplest of projections: the orthogonal projection. We shall then learn how to employ a simple shear transformation to implement more general oblique projections.

VII.4.1 Orthogonal Projections

In orthogonal projections, all projectors are perpendicular to the projection plane, hence $\mathbf{d} = (0, 0, d_n)$. Using this fact in equations 7.1 and 7.2, it is clear that $P' = (P_u, P_v, n_{pp})$ for orthogonal projections. But as stated in the previous section, this would result in all our geometry being smashed flat onto the projection plane. Instead we wish to preserve the original depth, so we really want $P'=(P_u,P_v,P_n)$. That is, we want the transformation to map P to itself. Hence the matrix accomplishing our depth-preserving orthogonal projection is simply the identity matrix!

Recall, however, that we wish to apply the window-viewport transformation along with the projection. That only requires use of a_u, b_u, a_v, b_v, a_n, b_n as described in section VII.1. The desired matrix is therefore:

$$\underbrace{\begin{pmatrix} a_u & 0 & 0 & b_u \\ 0 & a_v & 0 & b_v \\ 0 & 0 & a_n & b_n \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{window}-\text{viewport map}} \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{orthogonal projection}} \Rightarrow \underbrace{\begin{pmatrix} a_u & 0 & 0 & b_u \\ 0 & a_v & 0 & b_v \\ 0 & 0 & a_n & b_n \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{complete orthogonal mapping matrix}}$$

VII.4.2 General Oblique Projections

General parallel projections can be specified in a number of ways. Let us assume the PHIGS specification. A programmer defines a parallel projections in PHIGS by providing (i) the VRC window **W**, (ii) the NPC viewport **V**, (iii) a *projection reference point* R, and (iv) the n coordinate of the projection plane n_{pp}. Most of these parameters are illustrated in Figure 7.2. The projection reference point is used to determine the common direction of projection, **d**, that we introduced in section VII.2:

$$\mathbf{d} = R - M$$

where M is the midpoint of the window on the projection plane. That is,

$$M = (\ (\mathbf{W}.umin + \mathbf{W}.umax)/2\ ,\ (\mathbf{W}.vmin + \mathbf{W}.vmax)/2\ ,\ n_{pp})$$

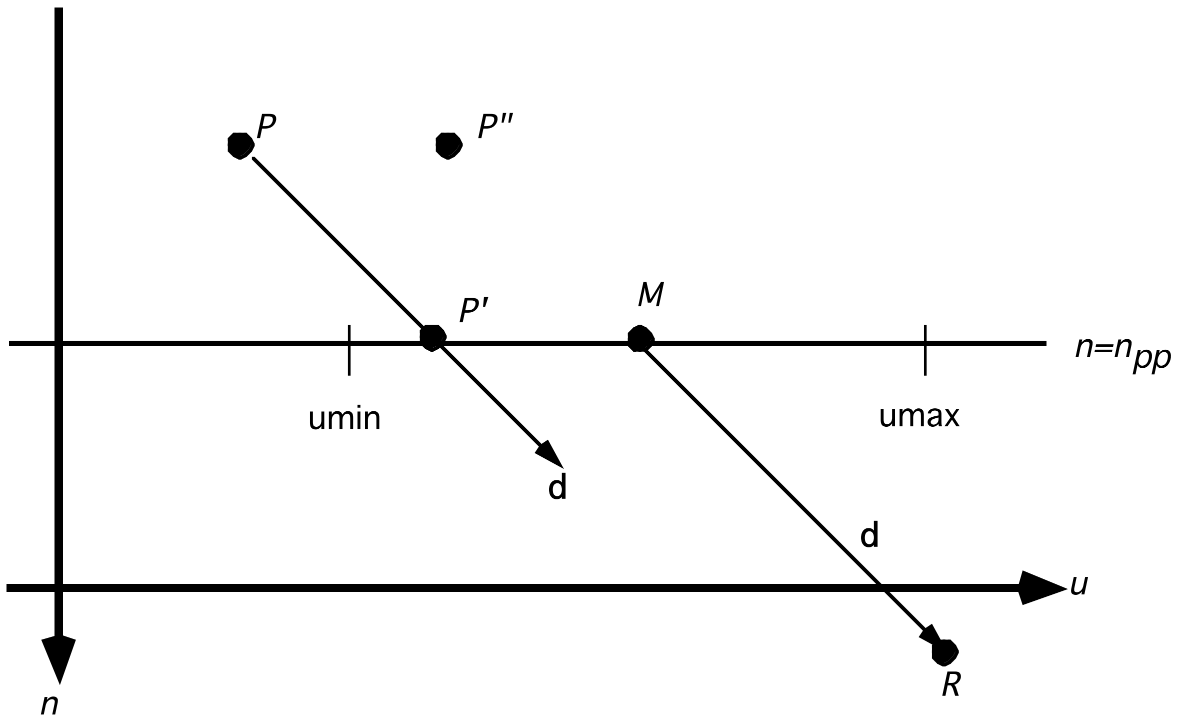


Figure 7.2: Parallel Projection Geometry

We saw in section VII.2 how to compute P' from P and $\mathbf{d}$. (Recall equations 7.1 and 7.2.) In order to preserve the original depth, however, we want to shear the view volume so that it maps P to P'' as indicated in Figure 7.2. Note that:

$$P'' = (P'_u, P'_v, P_n) \tag{7.3}$$

We therefore want a shearing matrix $\mathbf{S}$ such that $P'' = \mathbf{S}P$.

> ***Exercise***: From equations 7.1, 7.2, and 7.3, show that the desired shear matrix $\mathbf{S}$ can be written as:

$$\mathbf{S} = \begin{pmatrix} 1 & 0 & -\frac{d_u}{d_n} & \frac{n_{pp}d_u}{d_n} \\ 0 & 1 & -\frac{d_v}{d_n} & \frac{n_{pp}d_v}{d_n} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.4}$$

> ***Exercise***: Recall our derivation of general shear matrices in section V.4. How would you assign values to B, $\hat{\mathbf{n}}$, $\hat{\mathbf{u}}$, and f as functions of the PHIGS parameters described above to obtain this same matrix $\mathbf{S}$? Demonstrate that your assignment works.

The shear transformation $\mathbf{S}$ establishes an orthogonal view volume, but the units are still window (application) units. As we did in section VII.4.1, we compute a matrix to apply the window-viewport mapping. It should be clear that the complete projection matrix can be written as

$$\underbrace{\begin{pmatrix} a_u & 0 & 0 & b_u \\ 0 & a_v & 0 & b_v \\ 0 & 0 & a_n & b_n \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{window- viewport map}} \underbrace{\begin{pmatrix} 1 & 0 & -\frac{d_u}{d_n} & \frac{n_{pp}d_u}{d_n} \\ 0 & 1 & -\frac{d_v}{d_n} & \frac{n_{pp}d_v}{d_n} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{shear oblique view volume to orthogonal}} \Rightarrow \underbrace{\begin{pmatrix} a_u & 0 & -\frac{a_u d_u}{d_n} & \frac{a_u n_{pp} d_u}{d_n} + b_u \\ 0 & a_v & -\frac{a_v d_v}{d_n} & \frac{a_v n_{pp} d_v}{d_n} + b_v \\ 0 & 0 & a_n & b_n \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{complete oblique mapping matrix}} \tag{7.5}$$

> ***Exercise***: Suppose $\mathbf{W}$=(-1,4 , 3,8 , 0,5), R=(0,0,10), n_{pp}=4, and $\mathbf{V}$ = (0,1,0,1, 0,1). Show the resulting matrix.

Recall that the only direct interface provided by OpenGL for parallel projections (`glOrtho`) supports an orthogonal transformation as derived in section VII.4.1. Since general matrices can be loaded or concatenated onto either the GL_MODELVIEW or GL_PROJECTION matrix, however, an OpenGL programmer could create the matrix indicated above to achieve an oblique projection in an OpenGL program. Care must be taken to account for the specifics of the internal logical device coordinate space used in OpenGL, however. This comment leads directly to the topic of the next section.

VII.5 Interlude on Handedness of Viewing Coordinate Systems

Before we can show how our matrix expressions relate to those in common texts, we must look carefully at the notion of handedness of viewing coordinate systems. While general in nature, these concepts are most relevant to us here when understanding projection matrices, both parallel and perspective.

It was common for early graphics texts and early graphics APIs to employ right-handed coordinate systems for all modeling operations, and then have the viewing transformations switch to a left-handed eye coordinate system. Computing "distance from eye" was required for hidden line and hidden surface removal, hence early algorithms focused on keeping these distances and on using matrices which transformed coordinates in the z direction so that these distances, not z coordinates, appeared in the third position of transformed points. In addition, there seemed to be the belief that, intuitively, "increasing z coordinate" implied "increasing distance from the eye". Since it was desired to map the x and y directions to the horizontal and vertical screen directions, it was only possible to achieve the desired z direction by having the positive z axis go into the screen[10], hence a left-handed system emerged.

Whether this history and motivation is completely accurate is unimportant. What matters is that there is a legacy of the early practice of using left-handed coordinate systems that continues to survive in various ways in texts and APIs of today. In these notes, we derive the general formulas used for the PHIGS model. We then see how they specialize to the slightly simpler OpenGL model [Schreiner, et. al. 08].

To start, note that the primary projection interfaces in OpenGL are `glOrtho` (for orthogonal) and `glFrustum` (for perspective). The parameter sequence for both of these routines takes the window 6-tuple (*left*, *right*, *bottom*, *top*, *near*, *far*). The (*left*, *right*, *bottom*, *top*) parameters describe the u and v eye coordinate system window limits in the obvious way. The parameters "*near*" and "*far*", however, are **distances**, not coordinates. They may be signed in the case of `glOrtho`, but in any case, (i) the coordinate planes involved are n=-*near* and n=-*far*, (ii) the value supplied for "*far*" must be greater than that supplied for "*near*", and (iii) both "*near*" and "*far*" must be positive for `glFrustum`. Note that using right-handed system conventions, (n_{min}, n_{max}) = (-*far*, -*near*).

Once we have determined the coordinates of the near and far clipping planes, the OpenGL matrices map them to the viewport so that *near*⇒-1 and *far*⇒+1; i.e., a left-handed convention.

VII.6 Parallel Projections, Part 2

The derivation of parallel projections in [Foley, et. al. 90] and [Foley, et. al. 94] assumes that the projection plane is the uv plane. (They call it "xy".) In our notation, this means that n_{pp}=0. Making this substitution in equation (7.4), we obtain equation 6.30 of [Foley, et. al. 90] (equivalently, equation 6.20 of [Foley, et. al. 94]). Then, they transform into the canonical viewport volume (-1,1 , -1,1 , -1,0). They do the window-viewport map by translating the center of the window on the near clipping plane to the origin and then scaling about the origin. Although the final matrix they show (equation 6.36 of [Foley, et. al. 90]; equation 6.26 of [Foley,

[10] Of course the positive y direction on some screens is assumed to extend from the top of the screen towards the bottom. This was another case of hardware considerations (i.e., the refresh path) influencing external software interfaces. Using that orientation for y would allow a right-handed coordinate system with z going into the screen. This seems now to be a separate issue, however.

et. al. 94]) includes two matrices implementing the view orientation, it should be straightforward to show that the composite parallel projection matrix derived there is equivalent to our equation (7.5) above when using the indicated viewport view volume to derive the *a* and *b* constants of our window-viewport matrix.

Now let's consider the OpenGL matrices as presented in [Schreiner, et. al. 08]. First recall that OpenGL only provides a direct programming interface for orthogonal projections, hence we are working from the matrix developed in section VII.4.1. The OpenGL orthogonal projection is defined as mapping the supplied window to the canonical viewport (-1,1,-1,1,-1,1). As we noted above, this is straightforward in the *u* and *v* directions. In the *n* direction, points with *n*=-*near* get mapped to viewport points with *z*=-1; points with *n*=-*far* get mapped to viewport points with *z*=+1.[11] Using the abbreviations (*l*, *r*, *b*, *t*, *n*, *f*) for the formal parameters (*left*, *right*, *bottom*, *top*, *near*, *far*), the matrix generated is shown as follows on page 778 of [Schreiner, et. al. 08]:

$$\begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Exercise: Demonstrate that the matrix of section VII.4.1 is equivalent to this matrix when (i) the window-viewport constants are computed as described in section VII.1, (ii) the indicated viewport is used, and (iii) the indicated mapping in the *z* direction is applied.

If more general parallel projection matrices are to be given to OpenGL, it is the responsibility of the programmer to build them so that the left-handed convention as discussed above and demonstrated in the exercise is observed.

Exercise: Suppose we want to create a projection matrix for use in an OpenGL program which implements an oblique parallel projection with **W**=(-1,4 , 3,8 , 0,5) and **d**=(2,6,10). Assume the standard OpenGL convention for the location of the projection plane and the logical device viewport. Show the matrix you would have to pass to `glLoadMatrix` to accomplish this projection.

VII.7 Perspective Projections

In this section, we first derive the perspective projection matrix using the OpenGL convention that the viewer is at the origin of the eye coordinate system. Next we consider the more general PHIGS model in which the actual position of the viewer is specified by an arbitrary placement of the projection reference point, *R*, inside the view reference coordinate system.

VII.7.1 Eye at the Origin

Figure 7.3 illustrates the basic viewing geometry and uses the same conventions as figures 7.1

[11] Recall since the window is defined in eye coordinates, we use *u*, *v*, and *n* for its axes. We use *x*, *y*, and *z* for the axes of the viewport system.

and 7.2.

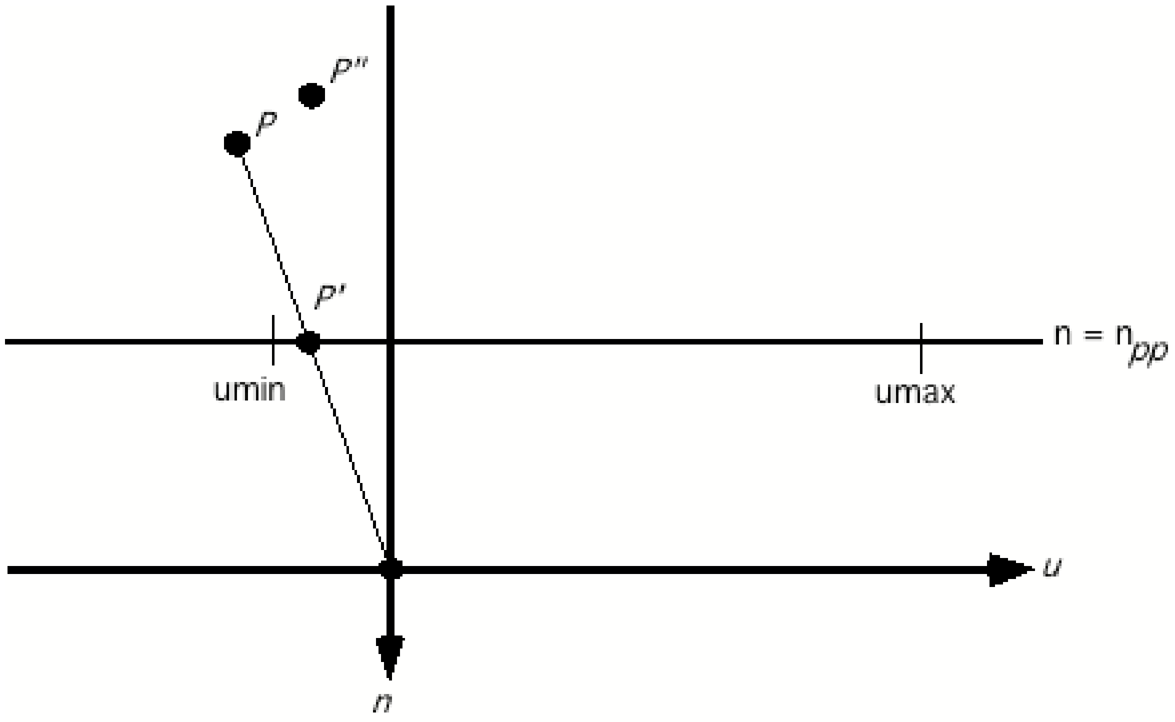


Figure 7.3: Perspective projection geometry with the eye at the origin

Unlike parallel projections, the direction of projection **d** is *not* common. Instead, the required projector is defined by passing a line from the point to be projected through the origin (the assumed position of the eye). Hence for an arbitrary point P:

$$\mathbf{d} = (0, 0, 0) - P = (-P_u, -P_v, -P_n) \tag{7.6}$$

Equation 7.1 still applies, and equation 7.2 becomes:

$$P'_n = n_{pp} = P_n + r(-P_n) \Rightarrow r = (n_{pp} - P_n) / -P_n \tag{7.7}$$

Notice that the n coordinate of our generic point P appears in the denominator! This result tells us that we cannot use an affine transformation to represent the effect of equation 7.1 using the r from equation 7.7.

But notice that, while nonlinear, equation 7.1 with this value for r is *rational linear*. That is, it includes terms expressed as the quotient of two terms which are linear in the coordinates of the point being projected. Rational linear transformations on affine points can be described as linear operations when the points are embedded in projective space. Recall that the affine point (x,y,z) is projectively equivalent to the projective space points (wx,wy,wz,w) for all $w\neq 0$. To see how we can use this projective equivalence, let's use our expressions for **d** and r from equations 7.6 and 7.7 to expand equation 7.1:

$$\begin{aligned} P' &= P + r\mathbf{d} \\ &= P + \frac{n_{pp} - P_n}{-P_n}(-P) \end{aligned} \tag{7.8a}$$

$$\begin{aligned} P'(-P_n) &= P(-P_n) + (n_{pp} - P_n)(-P) \\ &= (-n_{pp})P \end{aligned} \tag{7.8b}$$

Based on this expression and recalling that we actually want a P'' which preserves at least relative depth, we want to establish a transformation which computes a projective equivalent of P'', namely $(wP_u', wP_v', wP_n'', w)$ where $w = -P_n$. From these observations, it should already be clear that we will need a 4x4 matrix whose bottom row is different from (0,0,0,1). In fact, much more should be obvious as the following exercise indicates.

Exercise: Show that rows 1, 2, and 4 of a matrix **M** which accomplishes the mapping in the u and v directions are given by:

$$\mathbf{M} = \begin{pmatrix} -n_{pp} & 0 & 0 & 0 \\ 0 & -n_{pp} & 0 & 0 \\ ? & ? & ? & ? \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{7.9}$$

Now what about the n direction? That is, what should the third row of **M** be so that we can preserve depth information? Equation 7.8b implies that it should be $(0,0,-n_{pp},0)$, but this would leave *all* geometry smashed flat on the projection plane.

Let's step back to re-examine things. Clearly the u and v coordinates of P must not be allowed to affect P_n''. That is, two points which have the same n coordinate should get mapped to points which have the same viewport z coordinate, regardless of their u and v coordinates. Therefore the first two elements of the third row must be "0". Recalling also that equation 7.8b tells us the third row must produce $P_n''(-P_n)$, we need only determine the third and fourth elements of the third row of **M** (let's call them A_n and B_n) so that:

$$P_n''(-P_n) = A_n P_n + B_n \tag{7.10}$$

With parallel projections, we were able to establish the matrix so that the actual depth was preserved. That is, $P_n = P_n''$ in the parallel case. Can that work here? That is, can we find A_n and B_n so that the actual depth is preserved? Unfortunately not. From the form of the equation above, it can be shown that for any given constants A_n and B_n, it is only possible for at most two specific values of P_n to be preserved.

It turns out that the best we can do is to preserve *relative depth*. That is, the best we will be able to do will be to establish A_n and B_n so that, for any two points P and Q, if $P_n > Q_n$, then $P_n'' > Q_n''$.

To establish reasonable values for A_n and B_n, it suffices to apply the relevant window-viewport limits. For example, we can use the OpenGL conventions for the n direction as studied in section VII.5.

The hard part is done. As a final step, we need to apply the window-viewport transformation like we did with parallel projections. In the u and v directions, we will simply use a_u, b_u, a_v, b_v just like we did for parallel. The required window-viewport map in the n direction has of course already been done.

The complete mapping matrix for perspective projections can now be assembled as:

$$\underbrace{\begin{pmatrix} a_u & 0 & 0 & b_u \\ 0 & a_v & 0 & b_v \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{window- viewport map}} \underbrace{\begin{pmatrix} -n_{pp} & 0 & 0 & 0 \\ 0 & -n_{pp} & 0 & 0 \\ 0 & 0 & A_n & B_n \\ 0 & 0 & -1 & 0 \end{pmatrix}}_{\text{perspective projection, eye at origin}} \Rightarrow \underbrace{\begin{pmatrix} -a_u n_{pp} & 0 & -b_u & 0 \\ 0 & -a_v n_{pp} & -b_v & 0 \\ 0 & 0 & A_n & B_n \\ 0 & 0 & -1 & 0 \end{pmatrix}}_{\text{complete perspective mapping matrix}} \tag{7.11}$$

Let's consider a specialization of this result. If assume that the eye at the origin, a generic point (x,y,z) is projected onto the projection plane to a point (x_p,y_p) such that

$$\frac{x_p}{d} = \frac{x}{z} \quad \text{and} \quad \frac{y_p}{d} = \frac{y}{z} \tag{7.12}$$

where d is the z coordinate of the projection plane. (With the eye at the origin, it should be clear that the sign of x must agree with the sign of x_p (same for y and y_p), hence, the sign of d must agree with that of z. That is, d must be the actual projection plane z coordinate, *not* a distance.) By substituting $n_{pp}=d$, $P'=(x_p,y_p,d)$, and $P=(x,y,z)$ into equation (7.8a), we immediately obtain equation (7.12).

> ***Exercise***: Make these substitutions to verify that (7.12) follows from (7.8a).

Now let us look at the OpenGL matrix generated by a `glFrustum` call. The projection plane assumed by this routine is the near clipping plane, hence n_{pp}=*-near*. Recall that the projection includes a mapping from the supplied window to the canonical viewport (-1,1,-1,1,-1,1), and recall that in the n direction, n=*-near* is mapped to z=-1; n=*-far* is mapped to z=+1. Using the abbreviations (l, r, b, t, n, f) for the formal parameters (*left*, *right*, *bottom*, *top*, *near*, *far*), the matrix generated is shown as follows on page 777 of [Schreiner, et. al. 08]:

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{7.13}$$

> ***Exercise***: Using the remarks of the previous paragraph, show that the matrix of equation (7.11) produces the matrix of equation (7.13) when the indicated parameter assignments are employed.

VII.7.2 General Placement of the Eye in VRC

As we have stated before, the PHIGS viewing model interprets the coordinate system determined by the view orientation matrix – what we have been calling the *eye coordinate system* – as a *view reference (or VRC) system*. That is, this system is one in which subsequent viewing information, including the actual position of the eye, is defined. Here we must focus on the position of the eye. In section VII.4.2, we saw that PHIGS used the *projection reference point*, R, to characterize the common direction of projection for oblique projections. In the perspective case, this same point R is interpreted as the actual position of the eye. Since R can be positioned anywhere in this VRC system, the projector associated with an arbitrary point, P, is defined with the following slight generalization of equation 7.6:

$$\mathbf{d} = R - P \tag{7.14}$$

Figure 7.4 illustrates this geometry. Equation 7.7 for r then becomes:

$$r = (n_{pp} - P_n) / (R_n - P_n) \tag{7.15}$$

Continuing our generalization of the treatment from the previous section, equation 7.8 becomes:

$$\begin{aligned} P' &= P + r\mathbf{d} \\ &= P + \frac{n_{pp} - P_n}{R_n - P_n}(R - P) \end{aligned} \tag{7.16a}$$

$$\begin{aligned} P'(R_n - P_n) &= P(R_n - P_n) + (n_{pp} - P_n)(R - P) \\ &= (R_n - n_{pp})P - P_n R + n_{pp} R \end{aligned} \tag{7.16b}$$

With this generalized equation and our understanding from the previous section, we can start working on the projection matrix.

Exercise: Show that rows 1, 2, and 4 of a matrix **M** which accomplishes the mapping in the u and v directions are given by:

$$\mathbf{M} = \begin{pmatrix} R_n - n_{pp} & 0 & -R_u & R_u n_{pp} \\ 0 & R_n - n_{pp} & -R_v & R_v n_{pp} \\ ? & ? & ? & ? \\ 0 & 0 & -1 & R_n \end{pmatrix} \tag{7.17}$$

Just as in the previous section, the third row of **M** must be $(0, 0, A_n, B_n)$ where A_n and B_n can be determined by applying the window-viewport limits in the n direction in the following equation:

$$P_n''(R_n - P_n) = A_n P_n + B_n \tag{7.18}$$

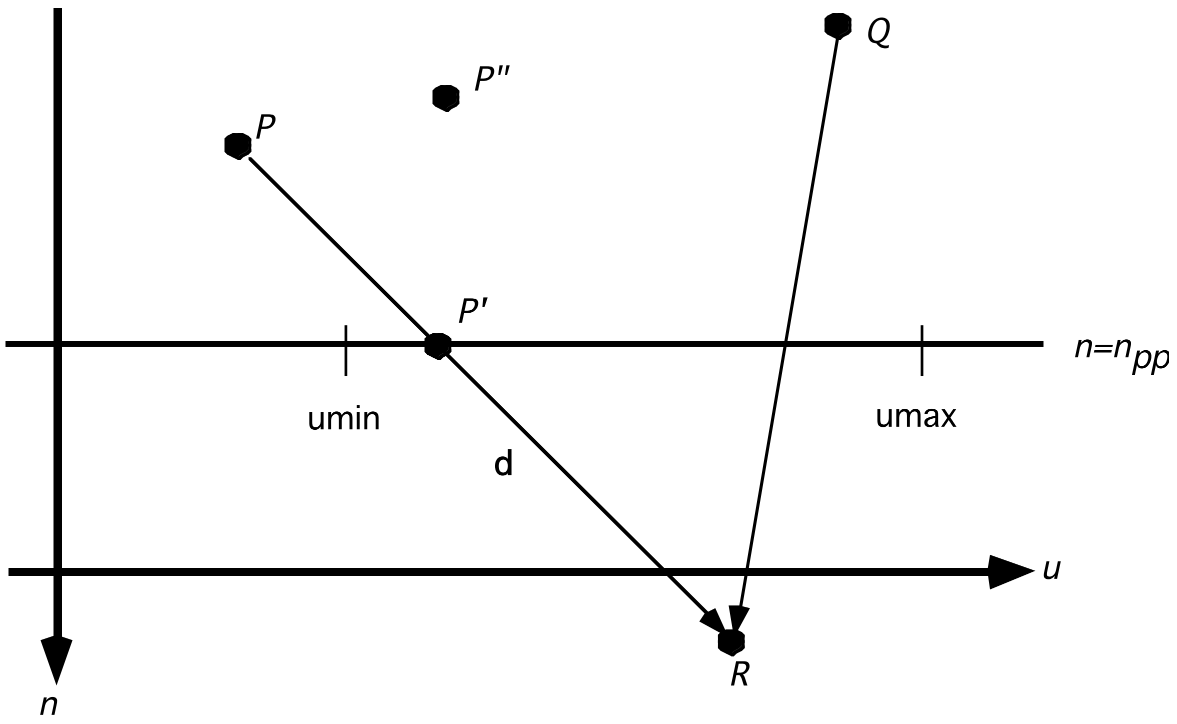


Figure 7.4: Perspective Projection Geometry with the eye at an arbitrary point R in VRC

The complete view mapping matrix for perspective projections with general placement of the eye in VRC can be obtained by applying the remainder of the window-viewport transformation as we did with parallel projections:

$$\begin{pmatrix} a_u & 0 & 0 & b_u \\ 0 & a_v & 0 & b_v \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} R_n - n_{pp} & 0 & -R_u & R_u n_{pp} \\ 0 & R_n - n_{pp} & -R_v & R_v n_{pp} \\ 0 & 0 & A_n & B_n \\ 0 & 0 & -1 & R_n \end{pmatrix}$$

$$= \begin{pmatrix} a_u(R_n - n_{pp}) & 0 & -a_u R_u - b_u & a_u R_u n_{pp} + b_u R_n \\ 0 & a_v(R_n - n_{pp}) & -a_v R_v - b_v & a_v R_v n_{pp} + b_v R_n \\ 0 & 0 & A_n & B_n \\ 0 & 0 & -1 & R_n \end{pmatrix} \tag{7.19}$$

Now let's continue our comparison of these results to those in the various texts and in OpenGL. First, since equation (7.19) simplifies to equation 7.11 when we make the substitution $R = (0, 0, 0)$, all the comparisons mentioned there apply to equation 7.19.

The Foley texts mention an alternative formulation of the perspective matrix which has the projection plane on the *xy* plane and the eye (i.e., *R*) placed at (0,0,*d*). One can show either directly from the geometry or by specialization of equation (7.16a) that:

$$\frac{x_p}{d} = \frac{x}{d-z} \quad \text{and} \quad \frac{y_p}{d} = \frac{y}{d-z} \tag{7.13}$$

VIII. Concluding Topics

We have completed our tour of the mathematical techniques required to implement the various pieces of the traditional graphics pipeline. Here we make a few additional observations before summarizing.

VIII.1 Remaining Pieces of the Graphics Pipeline

We have really only covered the first three of the six blocks in the graphics pipeline introduced in section II. We are therefore only half done!

Well, not really.

The fourth block — "Clipping" — is the process of partitioning the transformed geometry into two disjoint sets: one outside and the other inside the field of view. Typically the portion outside the field of view is discarded. Several coordinate-based as well as vector geometric algorithms have been developed for this operation. Descriptions of the major algorithms and critical comparisons among them are given in [Foley, et. al. 90], [Foley, et. al. 94], and [Angel 09].

The fifth block — "Perspective Divide" — is simply the process of mapping points in projective space back to affine space. This is required if the view mapping matrix (or any earlier transformation, for that matter) mapped affine points off the w=1 plane.

Finally, the sixth and last block — "Workstation Transformation" — is the process of mapping all or part of the logical device to a portion of the physical device. This is implemented as a window-viewport map with an associated clipping operation. No techniques other than those we have already described are required.

VIII.2 Applications of Vector Geometric Analysis

The power, utility, and applicability of vector geometric analyses extends far beyond what we have seen here. The tools developed in section III and whose use was illustrated in sections V and VI are employed throughout graphics and geometric modeling analyses.

Ray tracing is one of the most popular image generation algorithms in use today for situations that require realistic modeling of light interactions with surfaces [Glassner 89]. The method basically uses straight lines to model (i) the line of sight, and (ii) rays of light which bounce from object to object in a scene. The core operations are to (i) *generate a line* (for example, corresponding to a line of sight, a ray of light reflected or refracted from a surface, a "shadow test", etc.), and (ii) *compute points of intersection* with the objects in the scene. We describe vector geometric approaches for both of these core operations in [Goldman & Miller 1997] and [Miller 1997].

The design of curves and surfaces, especially freeform curves and surfaces such as those constructed using the Bezier and B-spline formulations, make extensive use of vector geometric analysis as well as projective coordinate systems. The projective coordinate systems are used to represent the rational forms of these polynomial curves and surfaces. See [Farin 95 or Farin 96] for an introduction.

VIII.3 Summary

We have reviewed the basic operations in common interactive graphics pipelines and developed two different styles of analysis: vector geometric and coordinate-based. The former were most applicable in situations where no assumptions could be made reliably about how geometry related to coordinate systems. We saw that vector geometric methods operate by expressing coordinate-system-independent relationships between points and vectors. Not only did these expressions not rely upon special configurations of the geometry with respect to each other or the coordinate system, but also they did not exhibit anomalous behavior in the event that special relationships *were* involved. This proved to be a powerful analytical technique that led to highly efficient and robust computer implementations.

However there are situations where it is useful to transform geometry into special coordinate systems where the orientation of the system can be exploited to dramatically simplify and accelerate certain types of imaging operations. As examples, we listed clipping algorithms, visible line and visible surface determination, and intensity depth cueing.

The vector geometric tools themselves were introduced and described in section III. The material in sections V and VI illustrated their use and emphasized their advantages over coordinate-based techniques for general geometric manipulations. Section VII discussed the generation of parallel and perspective projections, serving as one example of where coordinate-based schemes are useful. The eye coordinate system employed to derive these projection matrices was also described as facilitating a host of other view-dependent operations in graphics systems since “distance from eye” is directly represented in the z coordinates of points.

Acknowledgments

Ron Goldman first introduced me to the power and simplicity of vector geometric approaches while we were working together in the early 1980s. I thank him for that as well as for the time he spent giving me comments on various drafts of this material.

The development of the software implementing the various techniques described here was performed in DesignLab, a multidisciplinary research laboratory at the University of Kansas funded in part by NSF Grant CDA-94-01021.

IX. References

[Abbott 91]	Edwin Abbott Abbott, *Flatland: A Romance of Many Dimensions*, Princeton University Press, 1991. (Originally published circa 1880.)
[Angel 09]	E. Angel, *Interactive Computer Graphics: A Top-Down Approach Using OpenGL*, (fifth edition), Addison-Wesley, 2009.
[Brodlie & Mumford 96]	K. W. Brodlie and A. M. Mumford, United Kingdom: Support Group Provides National Focus for Computer Graphics Education, *Computer Graphics*, Vol. 30, No. 3, August 1996, pp. 28-29.
[Burger 65]	D. Burger, *Sphereland: A fantasy about curved spaces and an expanding universe*, Crowell, 1965.
[Carlbom & Paciorek 78]	I. Carlbom and J. Paciorek, Planar Geometric Projections and Viewing Transformations, *ACM Computing Surveys*, Vol. 10, No. 4, December 1978, pp. 465-502.
[Coxeter 87]	H. S. M. Coxeter, *Projective Geometry*, Springer-Verlag, 1987.
[DeRose 89]	T. DeRose, A Coordinate-Free Approach to Geometric Programming, in *Contemporary Approaches to Geometry for Computer Graphics and Computer-Aided Design*, SIGGRAPH ‘89 Short Course #14, August 1989. (a condensed version also appeared in W. Strasser and H. Seidel, editors, *Theory and Practice of Geometric Modeling*, Springer-Verlag, Berlin, 1989, pp. 291-306.)
[Dewdney 84]	A. K. Dewdney, *The planiverse: computer contact with a two-dimensional world*, Poseidon Press, 1984.
[Farin 95]	G. Farin, *NURB Curves and Surfaces: From Projective Geometry to Practical Use*, A. K. Peters, 1995.
[Farin 96]	G. Farin, *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*, Academic Press, 4th edition, 1996.
[Foley, et. al. 90]	J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, 1990.
[Foley, et. al. 94]	J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, and R. L. Phillips, *Introduction to Computer Graphics*, Addison-Wesley, 1994.
[Georgiades 92]	P. Georgiades, Signed Distance from Point to Plane, in *Graphics Gems III*, D. Kirk, editor, Academic Press, 1992, pp. 223-224.
[Glassner 89]	A. S. Glassner, editor, *An Introduction to Ray Tracing*, Academic Press, 1989.
[Glassner 90]	A. S. Glassner, Useful 3D Geometry, in *Graphics Gems*, A. S. Glassner, editor, Academic Press, 1990, pp. 297-300.
[Goldman 85a]	R. N. Goldman, Illicit Expressions in Vector Algebra, *ACM Transactions on Graphics*, Vol. 4, No. 3, July 1985, pp. 223-243.
[Goldman 85b]	R. N. Goldman, Vector Geometry: A Coordinate-Free Approach, *SIGGRAPH ‘85 Short Course #16*, July 1985.
[Goldman 90]	R. N. Goldman, Matrices and Transformations, in *Graphics Gems*, A. S. Glassner, editor, Academic Press, 1990, pp. 472-475.
[Goldman & Miller 97]	R. N. Goldman and J. R. Miller, *Coordinate Free Vector Equations for Natural Quadrics With Applications to Ray Tracing*, DesignLab Technical Report DL-1997-01, University of Kansas, 1997.

[Howard, et. al. 91]	T. L. J. Howard, W. T. Hewitt, R. J. Hubbold, and K. M. Wyrwas, *A Practical Introduction to PHIGS and PHIGS Plus*, Addison-Wesley, 1991.
[ISO 89]	International Organization for Standardization, *ISO 9592: Programmer's Hierarchical Interactive Graphics System (PHIGS)*, 1989.
[Miller 97]	J. R. Miller, *Some Notes On Ray Tracing Planes, Polygons, and Solids*, DesignLab Technical Report DL-1997-02, University of Kansas, 1997.
[Penna & Patterson 86]	M. Penna and R. Patterson, *Projective Geometry and its Applications to Computer Graphics*, Prentice Hall, 1986.
[Schreiner, et. al. 08]	D. Schreiner, M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2.1*, (sixth edition), Addison-Wesley, 2008.
[Watt 93]	A. Watt, *3D Computer Graphics*, Addison-Wesley, 1993.
[Watt & Watt 92]	A. Watt and M. Watt, *Advanced Animation and Rendering Techniques*, Addison-Wesley, 1992.

Appendix A

```
// aPoint.h -- 3D Affine points

// When spherical coordinates are used below, the meaning of the (theta,phi)
// angles is the following. Assume R is the vector from the origin to a
// given point in space. Assume Rxy is the projection of the vector R onto
the
// xy-plane. Then:
//     phi   = angle between R and the z-axis (0 <= phi <= PI)
//     theta = angle from the x-axis to Rxy. Positive angles are counter-
//             clockwise from the x-axis. Normal range: (-PI <= theta <=
+PI).
// Notice that this convention is different from the convention used in some
// contexts such as image synthesis where the roles of theta and phi are
often
// reversed, probably because theta has historically been used in Image
// Synthesis to characterize an angle of incidence. That is, to describe the
// angle between the outward pointing normal vector to a surface (a local "z"
// axis) and an incoming light direction.

#ifndef APOINT_H
#define APOINT_H

#include "Basic.h"
#include "ioSpec.h"

#ifndef A_VECTOR
class aVector;
#endif

// indices for extracting coordinates from points

const int W = -1;
const int X =  0;
const int Y =  1;
const int Z =  2;

class aPoint
{
public:

      // --------- basic constructors
      aPoint(); // 3D point at origin
      aPoint(const aPoint& p);
      aPoint(const Dxyzw p);
      aPoint(const Fxyzw p);

      // --------- other constructors
      aPoint(const aVector& v);
      aPoint(real xx, real yy, real zz=0.0);

      // --------- destructor
      virtual ~aPoint();

      // --------- operators as methods
      aPoint      operator=(const aPoint& rhs);
      aPoint      operator+=(const aVector& rhs);
```

```
    aPoint      operator+=(const aPoint& rhs);
    aPoint      operator-=(const aVector& rhs);
    aPoint      operator*=(real f);
    aPoint      operator/=(real f);
    real        operator[](int index) const; // read-only indexing
                      // see indexing constants above

    // --------- operators as friend functions (Points only)
    friend aPoint     operator+(const aPoint& p1, const aPoint& p2);

    friend aPoint     operator*(real f, const aPoint& p);
    friend aPoint     operator*(const aPoint& p, real f);
    friend aPoint     operator/(const aPoint& p, real f);

    friend ostream&   operator<<(ostream& os, const aPoint& p);
    friend istream&   operator>>(istream& is, aPoint& p);

    // --------- operators as friend functions (Points and Vectors)
    friend aVector    operator-(const aPoint& p1, const aPoint& p2);
    friend aPoint     operator+(const aPoint& p1, const aVector& v2);
    friend aPoint     operator-(const aPoint& p1, const aVector& v2);

    // ---------- General Methods
    //                  Extract affine coordinates into ...
    DxyzP       aCoords(Dxyz coords) const; // ... a double[]
    FxyzP       aCoords(Fxyz coords) const; // ... a float[]
    bool        coincidentWith(const aPoint& p) const;
    real        distanceFromOrigin() const;
    real        distanceSquaredFromOrigin() const;
    real        distanceSquaredTo(const aPoint& p) const;
    real        distanceTo(const aPoint& p) const;
    real        normalize(); // move point along line through
                            // origin until it lies on the unit sphere
    DxyzwP      pCoords(Dxyzw coords, real w=1.0) const;
                            // extract coords, represent in projective
                            // space using provided "w" -> double array
    FxyzwP      pCoords(Fxyzw coords, real w=1.0) const;
                            // extract coords, represent in projective
                            // space using provided "w" -> single array
    void        toCylindrical(real& r, real& theta, real& z) const;
    void        toSpherical(real& rho, real& theta, real& phi) const;

    // ---------- General Class Methods
    static void       addPoint(const aPoint& P, aPoint* list, int& nInList,
                                    int listLength);
    static aPoint     centroid(const aPoint p[], int nPoints);
    static aPoint     fromCylindrical(real r, real theta, real z);
    static aPoint     fromSpherical(real rho, real theta, real phi);
    static real       getCoincidenceTolerance();
    static char*      getInputFormat(int iAspect);
    static char*      getOutputFormat(int oAspect);
    static void       inputFormat(int iAspect, char iValue);
    static void       inputFormat(int iAspect, const char* iValue);
    static void       ioFormat(int ioAspect, char ioValue);
    static void       ioFormat(int ioAspect, const char* ioValue);

          // find the point in the given buffer which is farthest
          // away in the given direction. If that point is 'P', the
```

```
            // function return value is "(P-ref).dir". If there is
            // exactly one such farthest point, then the indices i1
            // and i2 will both contain the index of 'P' in 'buf'. If
            // several points tie, then i1 and i2 are the indices of the
            // first succession of adjacent points all at that offset.
            // If there are multiple sets of adjacent tieing points,
            // only the indices of the first are returned.
      static real       maxOffsetInDirection(
                              const aPoint& ref, const aVector& dir,
                              const aPoint buf[], int bufSize,
                              int& index1, int& index2);
      static void       outputFormat(int oAspect, char oValue);
      static void       outputFormat(int oAspect, const char* oValue);
      static real       ratio(const aPoint& a, const aPoint& b,
                              const aPoint& c);
      static void       setCoincidenceTolerance(real tol);

    // ---------- Global constants

      // 1. Special Points
      static const aPoint     origin;
      static const aPoint     xAxisPoint;
      static const aPoint     yAxisPoint;
      static const aPoint     zAxisPoint;

      // 2. Values for "ioAspect" in "ioFormat"
      static const int        openDelimiter;  // e.g.: '(', '[', '{', '<', '
'
      static const int        separator;      // e.g.: ' ', '\t', '\n', ','

protected:

      // --------- instance variables
      real  x;
      real  y;
      real  z;

private:

      static real       sCoincidenceTol;

      static ioSpec     iSpec; // for input
      static ioSpec     oSpec; // for output

      // --------- private methods
};

#endif
```

This page was intentionally left blank.

Appendix B

```
// aVector.h -- 3D affine vectors

#ifndef A_VECTOR
#define A_VECTOR

#include "Basic.h"
#include "ioSpec.h"

#ifndef A_POINT
class aPoint;
#endif

// indices for extracting components from vectors

const int DW = -1;
const int DX =  0;
const int DY =  1;
const int DZ =  2;

class aVector
{
public:

      // --------- basic constructors
      aVector(); // zero vector
      aVector(const aVector& v);

      // --------- other constructors
      aVector(const aPoint& p);
      aVector(real Dx, real Dy, real Dz=0.0);
      aVector(const double xyz[]);
      aVector(const float xyz[]);

      // --------- destructor
      virtual ~aVector();

      // --------- operators as methods
      aVector     operator=(const aVector& rhs);
      aVector     operator+=(const aVector& rhs);
      aVector     operator-=(const aVector& rhs);
      aVector     operator*=(real f);
      aVector     operator/=(real f);
      bool  operator==(const aVector& rhs);
      bool  operator!=(const aVector& rhs);
      real        operator[](int index) const; // read-only indexing
                              // see indexing constants above

      // --------- operators as friend functions (Vectors only)
      friend aVector    operator+(const aVector& v1, const aVector& v2);
      friend aVector    operator-(const aVector& v1, const aVector& v2);
      friend aVector    operator-(const aVector& v);

      friend aVector    operator*(real f, const aVector& v);
      friend aVector    operator*(const aVector& v, real f);
      friend aVector    operator/(const aVector& v, real f);
      friend ostream&   operator<<(ostream& os, const aVector& v);
      friend istream&   operator>>(istream& is, aVector& v);

      // --------- operators as friend functions (Points and Vectors)
      friend aVector    operator-(const aPoint& p1, const aPoint& p2);
      friend aPoint     operator+(const aPoint& p1, const aVector& v2);
      friend aPoint     operator-(const aPoint& p1, const aVector& v2);

      // ---------- General Methods
      void              arbitraryNormal(aVector& normal) const;
      aVector           cross(const aVector& rhs) const;
      void              decompose(const aVector& arbitraryVector,
                              aVector& parallel, aVector& perpendicular)
                              const;
      real              dot(const aVector& rhs) const;
      real              length() const;
      real              lengthSquared() const;
      real              maxAbsComponent(int& componentIndex) const;
      real              minAbsComponent(int& componentIndex) const;
      real              normalize();
      real              normalizeToCopy(aVector& normalizedCopy) const;
      bool              parallelTo(const aVector& v) const;

      // ---------- Class Methods
                  // The following assumes U and W have values (even if
                  // all zero). W gets normalized; the component of U
                  // perpendicular to W becomes U. Finally V <- W x U.
                  // If W is zero vector, then (xu, yu, zu) are copied
                  // into U, V, and W. If the component of U perpendicular
                  // to W is zero, then an arbitrary vector perpendicular
                  // to W is created and used.)
      static void       coordinateSystemFromUW
                                    (aVector& U, aVector& V, aVector& W);
                  // The following is analogous to previous, but with the
                  // roles of V and U reversed.
      static void       coordinateSystemFromVW
                                    (aVector& U, aVector& V, aVector& W);
      static aVector    cross(const aVector& v1, const aVector& v2);
      static real       dot(const aVector& v1, const aVector& v2);
      static char*      getInputFormat(int iAspect);
      static char*      getOutputFormat(int oAspect);
      static void       inputFormat(int iAspect, char iValue);
      static void       inputFormat(int iAspect, const char* iValue);
      static void       ioFormat(int ioAspect, char ioValue);
      static void       ioFormat(int ioAspect, const char* ioValue);
      static void       outputFormat(int oAspect, char oValue);
      static void       outputFormat(int oAspect, const char* oValue);
```

```
      // ---------- Global constants

      // 1. Special Vectors
      static const aVector    xu;
      static const aVector    yu;
      static const aVector    zu;
      static const aVector    zeroVector;

      // 2. Values for "ioAspect" in "ioFormat"
      static const int  openDelimiter;          // '(', '[', '{', or ' '
      static const int  separator;              // ' ', '\t', '\n', or ','

protected:

      // --------- instance variables
      real  dx;
      real  dy;
      real  dz;

private:

      static ioSpec     iSpec; // for input
      static ioSpec     oSpec; // for output

      // --------- private methods
};

#endif
```

This page was intentionally left blank.

Appendix C

```
// Matrix3x3.h -- 3x3 Matrices

// This interface is designed for the specification and manipulation of
// 3x3 matrices as a part of Affine transformations specified on 3D
// vectors. As such, it provides facilities to create 3x3 matrices which
// rotate and mirror vectors, form tensor products of vectors, and provide
// other support required by Matrix4x4 in its role of representing
// transformations of 3D geometry. This interface is NOT intended to
// provide facilities for representing 2D transformations in homogeneous
// form.

#ifndef MATRIX3x3_H
#define MATRIX3x3_H

#include "aPoint.h"
#include "aVector.h"

class Matrix3x3
{
public:

      // --------- basic constructors
      Matrix3x3(); // Identity Matrix
      Matrix3x3(const Matrix3x3& M);
      Matrix3x3(
            real m11, real m12, real m13,
            real m21, real m22, real m23,
            real m31, real m32, real m33);

      // Mirror matrix construction
      Matrix3x3(const aVector& mirrorPlaneNormal); // General
      Matrix3x3(char axis); // mirror plane is perpendicular to indicated
axis

      // Rotation matrix construction (angle is in radians)
      Matrix3x3(const aVector& rotationAxis, real angle); // General
      Matrix3x3(char axis, real angle); // axis is the indicated axis

      // Scale matrix construction
      Matrix3x3(real sx, real sy, real sz);

      // construct as tensor product matrix
      Matrix3x3(const aVector& u, const aVector& v);

      // --------- destructor
      virtual ~Matrix3x3();

      // --------- operators as methods
      Matrix3x3         operator=(const Matrix3x3& rhs);
      bool              operator==(const Matrix3x3& rhs) const;

      Matrix3x3         operator*=(const Matrix3x3& rhs);
      Matrix3x3         operator*=(real f);
      Matrix3x3         operator+=(const Matrix3x3& rhs);

      aPoint            operator*(const aPoint& p) const;
      aVector           operator*(const aVector& v) const;

      // --------- operators as friend functions
      friend Matrix3x3  operator*(const Matrix3x3& m1, const Matrix3x3& m2);
      friend Matrix3x3  operator+(const Matrix3x3& m1, const Matrix3x3& m2);
      friend Matrix3x3  operator-(const Matrix3x3& m1, const Matrix3x3& m2);
      friend Matrix3x3  operator*(real f, const Matrix3x3& m);

      friend ostream&   operator<<(ostream& os, const Matrix3x3& m);
      friend istream&   operator>>(istream& is, Matrix3x3& m);

      // ---------- General Methods

      real        determinant() const;
      real        elementAt(int i, int j) const;
      int         extractAxisAngle(aVector& w, real& theta) const;
      void        extractRows(aVector& row1, aVector& row2,
                              aVector& row3) const;
      bool        isOrthogonal() const;
      bool        isRightHanded() const;
      real        largestDiagonalElement(int& pos) const;
                  // in the following, 'a' and 'b' are asssumed to be
                  // arrays[3]
      void        multiply(const real a[], real b[]) const;
      void        setElementAt(int i, int j, real newValue);
      real        trace() const;
      void        transpose();

      // ---------- General Class Methods

      // If any matrix cannot be built due to zero-length vectors, an
identity
      // matrix will be returned.
      static Matrix3x3  crossProductMatrix(const aVector& u);
      static char*      getInputFormat(int iAspect);
      static int        getOutputElementFieldWidth();
      static char*      getOutputFormat(int oAspect);
      static void       inputFormat(int iAspect, char iValue);
      static void       inputFormat(int iAspect, const char* iValue);
      static void       ioFormat(int ioAspect, char ioValue);
      static void       ioFormat(int ioAspect, const char* ioValue);
      static Matrix3x3  mirrorMatrix(const aVector& mirrorPlaneNormal);
      static void       outputFormat(int oAspect, char oValue);
      static void       outputFormat(int oAspect, const char* oValue);
      static void       outputFormat(int oAspect, int  oValue);

      // All angles in following are measured in radians....
      // If any matrix cannot be built due to zero-length vectors, an
identity
      // matrix will be returned.
      static Matrix3x3  rotationMatrix(
                           const aVector& rotationAxis, real angle);
      static Matrix3x3  rotationMatrix(
                           const aVector& uFrom, const aVector& uTo);
      static Matrix3x3  rotationMatrix(
                           const aVector& uFrom, const aVector& vFrom,
                           const aVector& uTo,   const aVector& vTo);
      static Matrix3x3  scaleMatrix(real sx, real sy, real sz);
      static Matrix3x3  shearMatrix(
                           const aVector& n, const aVector& u, real f);
      static Matrix3x3  tensorProductMatrix(
                           const aVector& u, const aVector& v);
      static Matrix3x3  xRotationMatrix(real angle);
      static Matrix3x3  yRotationMatrix(real angle);
      static Matrix3x3  zRotationMatrix(real angle);

      // ---------- Global constants

      // 1. Special Matrices
      static const Matrix3x3  IdentityMatrix;
      static const Matrix3x3  ZeroMatrix;

      // 2. Values for "ioAspect" in "ioFormat"
      static const int  elementFieldWidth;   // the integer width
      static const int  openMatrixDelimiter; // e.g., '(', '[', '{', or ' '
      static const int  openRowDelimiter;    // e.g., '(', '[', '{', or ' '
      static const int  elementSeparator;    // e.g., ' ', '\t', '\n', or ','
      static const int  rowSeparator;        // e.g., ' ', '\t', '\n', or ','

      // 3. Return codes for extraction of unit axis vector & rotation angle
      //    from 3x3 matrices which are 'supposed to be' orthogonal and
      //    right-handed.

      // 3.1: Abnormal internal errors which should never be returned:
      static const int  ImMDeterminantNotZero;
      static const int  CannotDetermineUnitAxisVector;
      static const int  CosTermsNotEqual;
      static const int  SinTermsNotEqual;

      // 3.2: "Normal" errors if user provides inappropriate matrix:
      static const int  NotOrthogonal;
      static const int  NotRightHanded;

      // 3.3: Successful extraction of unit axis vector and angle:
      static const int  Extracted_wTheta;

      friend class Matrix4x4;

protected:

      // --------- instance variables
      real  mElem[3][3];

private:

      static      ioSpec      matrix_iSpec;
      static      ioSpec      matrix_oSpec;

      static      ioSpec      row_iSpec;
      static      ioSpec      row_oSpec;

      // --------- private methods
      void  copy(const Matrix3x3& rhs);
      int   computeImMRows(Matrix3x3& ImM,
                              // the rows of I-M and the indices of the
                              // two most linearly independent rows.
                              aVector rows[], int& r1, int& r2) const;
      int   extractPrimitiveAxisAngle(int pos, aVector& w, real& theta)
const;
};

#endif
```

Appendix D

```
// Matrix4x4.h -- 4x4 Matrices

// See comments at start of Matrix3x3.h

#ifndef MATRIX4x4_H
#define MATRIX4x4_H

#include "Matrix3x3.h"

#ifndef PPOINT_H
class pPoint;
#endif

class Matrix4x4
{
public:

	// --------- basic constructors
	Matrix4x4(); // Identity Matrix
	Matrix4x4(const Matrix4x4& m);
	Matrix4x4(
		real m11, real m12, real m13, real m14,
		real m21, real m22, real m23, real m24,
		real m31, real m32, real m33, real m34,
		real m41, real m42, real m43, real m44);
	Matrix4x4(const float OGLMatrix[16]);
	Matrix4x4(const double OGLMatrix[16]);

	// From 3x3
	// ** This is (currently) the only interface in either Matrix3x3 or
	// ** Matrix4x4 that supports 2D transformations in any way.
	// This assumes "M" describes a 2D xform in projective space.
	// The implementation slides the third row and third column over
	// to the fourth, and inserts a third row and third column from
	// a 4x4 Identity matrix.
	Matrix4x4(const Matrix3x3& M);

	// From an Affine transformation spec: (M, t). 4th row gets (0,0,0,1)
	Matrix4x4(const Matrix3x3& M, const aVector& t);

	// From an affine matrix M and a fixed point:
	Matrix4x4(const Matrix3x3& M, const aPoint& FixedPoint);

	// From an affine matrix M and a point whose pre- and post-imagea
	// are known
	Matrix4x4(const Matrix3x3& M,
			const aPoint& PreImage, const aPoint& PostImage);

	// Translation matrix construction
	Matrix4x4(const aVector& v);

	// --------- destructor
	virtual ~Matrix4x4();

	// --------- operators as methods
	Matrix4x4   operator=(const Matrix4x4& rhs);
```

```
      bool        operator==(const Matrix4x4& rhs) const;

      Matrix4x4   operator*=(const Matrix4x4& rhs);
      Matrix4x4   operator*=(real f);
      Matrix4x4   operator+=(const Matrix4x4& rhs);

      aPoint      operator*(const aPoint& p) const;
      pPoint      operator*(const pPoint& p) const;
      aVector     operator*(const aVector& v) const;

      // --------- operators as friend functions
      friend Matrix4x4  operator*(const Matrix4x4& m1, const Matrix4x4& m2);
      friend Matrix4x4  operator+(const Matrix4x4& m1, const Matrix4x4& m2);
      friend Matrix4x4  operator-(const Matrix4x4& m1, const Matrix4x4& m2);
      friend Matrix4x4  operator*(real f, const Matrix4x4& m);

      friend ostream&   operator<<(ostream& os, const Matrix4x4& m);
      friend istream&   operator>>(istream& is, Matrix4x4& m);

      // ---------- General Methods

      real        determinant() const;
      real        elementAt(int i, int j) const;

                  // Two extraction routines useful in OpenGL, OpenInventor,
                  // etc.
      float*      extract(float m[16]) const;  // GLfloat
      double*     extract(double m[16]) const; // GLdouble

                  // The following method does not check to see if this IS an
                  // affine transformation. Clients should use the
                  // method "isAffineTransformation" if there is a
                  // possibility that it is not.
      void        extractAffineMt(Matrix3x3& M, aVector& t) const;

                  // The following routine uses Matrix3x3::extractAxisAngle
to
                  // find the unit axis vector and angle to reconstruct the
                  // upper 3x3 portion of the matrix. It then computes a base
                  // point B which, when combined with (w,theta) will at
least
                  // approximate the entire 4x4 matrix. At worst, a residual
                  // "postTranslation" will be required to reproduce the
                  // effect of the original 4x4 matrix. That translation
                  // (possibly 0) is returned in "postTranslation".
      int         extractAxisAngle(
                        aPoint& B, aVector& w, real& theta,
                        aVector& postTranslation) const;
      bool        inverse(Matrix4x4& mInv) const;
      bool        isAffineTransformation() const;
                  // in the following, 'a' and 'b' are asssumed to be
                  // arrays[4]
      void        multiply(const real a[], real b[]) const;
      void        setElementAt(int i, int j, real newValue);
      Matrix3x3   subMatrix(int skipRow, int skipCol) const;

      // ---------- General Class Methods
      static char*      getInputFormat(int iAspect);
      static int        getOutputElementFieldWidth();
      static char*      getOutputFormat(int oAspect);
      static void       inputFormat(int iAspect, char iValue);
      static void       inputFormat(int iAspect, const char* iValue);
      static void       ioFormat(int ioAspect, char ioValue);
      static void       ioFormat(int ioAspect, const char* ioValue);
      static void       outputFormat(int oAspect, char oValue);
      static void       outputFormat(int oAspect, const char* oValue);
      static void       outputFormat(int oAspect, int  oValue);
      static Matrix4x4  xRotationMatrix(real angle);
      static Matrix4x4  yRotationMatrix(real angle);
      static Matrix4x4  zRotationMatrix(real angle);

    // ---------- Global constants

      // 1. Special Matrices
      static const Matrix4x4  IdentityMatrix;
      static const Matrix4x4  ZeroMatrix;

      // 2. Values for "ioAspect" in "ioFormat"
      static const int  elementFieldWidth;   // the integer width
      static const int  openMatrixDelimiter; // e.g., '(', '[', '{', or ' '
      static const int  openRowDelimiter;    // e.g., '(', '[', '{', or ' '
      static const int  elementSeparator;    // e.g., ' ', '\t', '\n', or ','
      static const int  rowSeparator;        // e.g., ' ', '\t', '\n', or ','

      // 3. Return codes for extraction of base point, unit axis vector,
      //    and rotation angle from 4x4 matrices which are 'supposed to
      //    describe' affine transformations whose upper 3x3 matrix is
      //    'supposed to be' orthogonal and right-handed. (See the Matrix3x3
      //    method "extractAxisAngle".)

      // 3.1: Abnormal internal errors which should never be returned:
      static const int  InternalBasePointComputationError;

      // 3.2: "Normal" errors if user provides inappropriate matrix:
      static const int  NotAffine;
      // (Note that the Matrix3x3 error codes may be returned as well.)

      // 3.3: Successful extraction of unit axis vector and angle:
      //      If the 3x3 extraction of (unit axis vector, angle) is
      //      successful, then the 4x4 utility tries to determine a base
      //      point. If successful, then "Matrix4x4::Extracted_BwTheta" is
      //      returned, otherwise the 3x3 code
("Matrix3x3::Extracted_wTheta")
      //      is returned, signifying that the (unit axis vector, angle)
      //      information was successfully computed and returned, but only an
      //      approximation to a rotation axis base point was able to be
      //      determined. In this case, 'postTranslation' will contain a
      //      non-zero vector which, if added to a point after rotation, will
      //      yield the correct point.
      static const int  Extracted_BwTheta;

protected:

      // --------- instance variables
      real  mElem[4][4];
private:

      static      ioSpec      matrix_iSpec;
      static      ioSpec      matrix_oSpec;

      static      ioSpec      row_iSpec;
      static      ioSpec      row_oSpec;

      // --------- private methods
                  void  copy(const Matrix4x4& rhs);

      static      int determineBasePoint(aVector ImMRows[], int maxWCompLoc,
                        int r1, int r2, const aVector& trans, aPoint& B);

                  void  installMt(const Matrix3x3& M, const aVector& t);
};

#endif
```

Appendix E

Sample Code Using the C++ Point, Vector, and Matrix Classes

Many queries and operations common in graphics and geometric modeling are easily expressed using applications of the vector geometric techniques developed in this monograph. Computations involving distances and signed distances between various combinations of points, lines, planes, and other basic curves and surfaces are frequently required.

In our first two examples here we show C++ code using vector geometric operations for two common distance queries. Derivations for these and similar queries can be found in the literature (e.g., [Glassner 90, Georgiades 92, Miller 97a]). Others are left as an exercise for the reader.

The examples in this appendix assume various subsets the definitions given in the header files in Appendices A-D.

Example 1: Finding the Signed Distance of a Point *Q* from a Plane

A plane in space can be characterized by a point *B* on the plane and a vector **n** perpendicular to the plane (e.g., the mirror plane of Figure 5.1(a)). The signed distance of an arbitrary point *Q* from such a plane is defined as the actual distance if *Q* is on the side of the plane to which **n** points, and the negative of this distance if it is on the other side. A C++ routine using the point and vector utilities described in Appendices A and B and implementing this definition can be written as:

```
bool signedDistancePntPlane(
   aPoint Q,             // the point
   aPoint B, aVector n,  // the plane
   real& signedDist)     // the signed distance
{
   aVector     nHat;
   double      length = n.normalizeToCopy(nHat);
   if (length < BasicDistanceTol)
         // a zero vector was provided for n. We cannot proceed.
         return false;
   signedDist = aVector::dot( (Q-B) , nHat );
   return true;
}
```

Example 2: Testing Whether a Point *Q* lies on a Plane

A point is on a plane if and only if its signed distance is zero (within a tolerance).

```
bool PointOnPlane(
   aPoint Q,             // the point
   aPoint B, aVector n)  // the plane
{
   real signedDist = 0.0;
   if (signedDistancePntPlane(Q,B,n,signedDist))
         return ( abs(signedDist) < BasicDistanceTol );
   else
         return false;
}
```

Example 3: Finding the Distance of a Point *Q* from a Line

A line in space can be defined by a point *B* on the line and a vector **w** specifying the direction of the line (e.g., the rotation axis of Figure 2). We frequently need to compute the distance between an arbitrary point *Q* and the line. We could adopt an approach based on the Pythagorean Theorem by considering the right triangle formed by *B*, *Q*, and the perpendicular projection of *Q* onto the line. In the code below, however, we compute the component of the vector (*Q*-*B*) which is perpendicular to **w**, and then return the length of this vector.

```
bool distancePntLine(
   aPoint Q,            // the point
   aPoint B, aVector w,  // the line
   real& distance)       // the distance
{
   aVector     wHat;
   double      length = w.normalizeToCopy(wHat);
   if (length < tolerance)
         // a zero vector was provided for w. We cannot proceed.
         return false;
   aVector     vParallel, vPerpendicular;
   wHat.decompose( (Q-B), vParallel, vPerpendicular );
   distance = vPerpendicular.length();
   return true;
}
```

Example 4: Testing Whether a Point *Q* lies on a Line (Uses the "Point on Plane" utility)

```
bool PointOnLine(
   aPoint Q,            // the point
   aPoint B, aVector w)  // the line
{
   if (w.length() < BasicDistanceTol)
         // a zero vector was given
         return false;

   aVector     u, v;
   aVector::coordinateSystemFromUW(u,v,w);

   // "w" is parallel to the line direction vector; "u" and "v" are
   // perpendicular to "w". Therefore "Q" is on the line if and only
   // if it is on the two planes:
   return PointOnPlane(Q,B,u) && PointOnPlane(Q,B,v);
}
```

Example 5: Create a sphere to contain *N* points

```
// This routine will find a sphere containing the given points. This
// routine will NOT in general find a "good" such containing sphere.
// It is only meant to illustrate use of the PVM classes!

void ContainingSphere(aPoint pts[], int N,
                      aPoint& center, real& radius)
{
   center = aPoint::centroid(pts,N);
   radius = pts[0].distanceTo(center);
   for (int i=1 ; i<N ; i++)
   {
         real r = pts[i].distanceTo(center);
         if (r > radius)
               radius = r;
   }
}
```

Example 6: Rotate *N* points by 30 degrees about a supplied rotation axis

```
// Create a matrix to rotate about an axis by 30 degrees. Transform
// the points in a given array by the matrix.

void Transform(aPoint pts[], int N, aPoint AxisBase, aVector
AxisDir)
{
   Matrix3x3   M(AxisDir, degreesToRadians(30.0));
   Matrix4x4   Rot(M, AxisBase); // AxisBase is a Fixed Point
   for (int i=0 ; i<N ; i++)
         pts[i] = Rot * pts[i];
}
```

Example 7: Use OpenGL and the glut to draw a Bounded Cone

```
#include <math.h>
#include <GL/glut.h>
void DrawBoundedCone(aPoint Vertex, aPoint Base, real rAtBase,
                     int nSides)
{
   // Create transformations which will map the canonical bounded
   // cone created by glutSolidCone (base at origin; vertex at
   // (0,0,height)) to the desired position and orientation.
   aVector w = Vertex - Base;
   real  height = w.normalize();
   aVector     rotAxis = aVector::cross(aVector::zu,w);
   real  sine = rotAxis.normalize();
   real  cosine = aVector::dot(aVector::zu,w);
   real  angle = atan2(sine,cosine);
   // insulate callers from the effects of these xforms
   glPushMatrix();
   glTranslated(Base[X],Base[Y],Base[Z]);
   glRotated(radiansToDegrees(angle) ,
             rotAxis[DX],rotAxis[DY],rotAxis[DZ]);
   glutSolidCone(rAtBase,height,nSides,2);
   glPopMatrix();
}
```

Example 8: Use OpenGL to draw a Bounded Cylinder (glut has no bounded cylinder)

```
#include <math.h>
#include <GL/glut.h>
void DrawBoundedCylinder(aPoint P1, aPoint P2, real radius,
                         int nSides)
{
   aVector     u, v; // initialized to zero vectors
   aVector w = P2 - P1;
   aVector::coordinateSystemFromUW(u,v,w);
   if (nSides < 3)
         nSides = 3;
   real  dTheta = TWO_PI / real(nSides);
   real  theta  = 0.0;
   glBegin(GL_QUAD_STRIP);
   for (int i=0 ; i<=nSides ; i++)
   {
         aVector     offset = radius * (cos(theta)*u +
sin(theta)*v);
         aPoint c1Pt = P1 + offset;
         glVertex3d(c1Pt[X],c1Pt[Y],c1Pt[Z]);
         aPoint c2Pt = P2 + offset;
         glVertex3d(c2Pt[X],c2Pt[Y],c2Pt[Z]);
         theta += dTheta;
   }
   glEnd();
}
```

Example 9: Extract a rotation axis from a matrix and draw it. This example uses the utilities from examples 7 and 8.

```
#include <math.h>
#include <GL/glut.h>

#include "Matrix4x4.h"

void DrawVector(aPoint Tail, aPoint Common, aPoint Vertex,
                real rCyl, real rCon)
{
   DrawBoundedCylinder(Tail, Common, rCyl, 10);
   DrawBoundedCone(Vertex,Common,rCon,10);
}

void ExtractAndDrawAxis(const Matrix4x4& M,
                        real lengthCyl, real rCyl, real rCon)
{
   aPoint      B;
   aVector     w, postTranslation;
   real        theta;

   int  outcome = M.extractAxisAngle(B, w, theta, postTranslation);
   if (outcome == M4x4_Extracted_BwTheta)
   {
         // successfully extracted (B, w, theta) & postTranslation.
         // Draw them.

         aPoint      Common = B + lengthCyl*w;
         DrawVector(B, Common, Common + rCon*w, rCyl, rCon);
         if (postTranslation != aVector::zeroVector)
         {
               // The given matrix is equivalent to a rotation about
               // (B,w,theta) followed by a translation. Draw the
               // translation vector.

               Common = B + lengthCyl*postTranslation;
               DrawVector(B, Common, Common + rCon*postTranslation,
                          rCyl, rCon);
         }
   }
}
```

Example 10: Compute a vector perpendicular to a cone at a given point on the cone

It is frequently necessary to compute a unit vector perpendicular to a surface at a given point on the surface. Intersection computations and rendering algorithms are two examples. Suppose we wish to compute the outward pointing normal to a cylinder — defined by its axis (B, $\hat{\mathbf{w}}$) and radius r — at a point Q on the surface of the cylinder. The implementation would be identical to what we just saw in "distancePntLine", except rather than computing the length of "vPerpendicular" at the end, we would normalize it and return it as the unit outward pointing normal. Furthermore, if Q did not precisely lie on the cylinder, the normal so computed would be the normal for the point on the cylinder closest to Q. This extended notion of "normal at Q" only fails if Q lies on the axis of the cylinder, that is, if "vPerpendicular" is the zero vector.

A slightly more complicated but still relatively straightforward example is finding the outward pointing normal to a right circular cone at a point Q on the cone different from the vertex. Up to a sign, the desired normal is the component of the cone axis vector perpendicular to the ruling containing Q. In Figure 4, the red vector along the cone ruling is "wParallel"; the red vector pointing into the cone is "wPerpendicular".

```
bool normalToCone(
   aPoint Q,              // the point
   aPoint V, aVector w,   // the cone axis; V is the vertex
   aVector& normal)       // the outward pointing normal
{
   aVector     ruling = Q - V;
   double d = ruling.normalize();
   if (d < tolerance)
         // Q is at the vertex. We cannot proceed.
         return false;

   aVector     wParallel, wPerpendicular;
   ruling.decompose( w, wParallel, wPerpendicular );
   d = wPerpendicular.normalizeToCopy(normal);
   if (d < tolerance)
         // Q is on the cone axis -or- w is the zero vector. We
         // cannot proceed in either case.
         return false;
   // invert normal if necessary (The outward pointing normal must
   // point away from the cone axis.)
   if (aVector::dot(w,ruling) > 0.0)
         normal = -normal;
   return true;
}
```

Example 11: Find the intersection between a line and a plane

```
int intLinePlane(
      aPoint linePnt, aVector lineDir,     // the line
      aPoint plnPoint, aVector plnNormal,  // the plane
      aPoint& intPoint)                // the computed point of intersection

      // return value “-1” ⇒ line parallel to plane
      // return value “ 0” ⇒ line contained in plane
      // return value “+1” ⇒ single point of intersection returned in
      //                      “intPoint”
{
      aVector     nHat, uHat;
      plnNormal.normalizeToCopy(nHat);
      lineDir.normalizeToCopy(uHat);
      if (abs( aVector::dot(uHat,nHat) ) < BasicUnitTol)
            // line direction is parallel to plane
            if ( PointOnPlane(linePnt, plnPoint, plnNormal) )
                  return 0;
            else
                  return -1;

      double numerator = aVector::dot(nHat,(plnPoint - linePnt));
      double distance = numerator / aVector::dot(uHat,nHat);

      intPoint = linePnt + distance*uHat;
      return 1;
}
```