# A FUNCTIONAL APPROACH TO GEOMETRIC APPLICATIONS IN COMPUTER GRAPHICS

Pee-Hong Chen
L. David Sabbagh

Computer Science Department
Lindley Hall 101
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 146

A FUNCTIONAL APPROACH TO GEOMETRIC APPLICATIONS
IN COMPUTER GRAPHICS

by

Pee-Hong Chen
L. David Sabbagh

August, 1983

# A FUNCTIONAL APPROACH TO GEOMETRIC APPLICATIONS IN COMPUTER GRAPHICS

Pee-Hong Chen

L. David Sabbagh

Computer Science Department
Lindley Hall 101
Indiana University
Bloomington, IN 47405

## Abstract

In this paper, we discuss the suitability of applying functional programming techniques to computer graphics. We show, using the programming language Scheme, how to implement a linear algebra package which is used in an interactive 3D projection system. The main idea we wish to present is that functional programming, together with Scheme, is a proper method for attacking graphics problems where operations of linear algebra play a major role. In addition, we want to show the power of Scheme in such applications and its utility for other types of numerical problems.

Key words and phrases: computer graphics, linear algebra, Scheme, vector, matrix, perspective projection, parallel projection, and clipping.

## §1 Introduction

Graphics is an area in computer science which deals with the simulation of our visual world on a two-dimensional screen. The basis for computer graphics is in linear algebra and geometry. Most researchers' work in this field has been concentrated on more effective simulation or transformation algorithms and better video devices. There has not been much attention to the underlying programming language in which the algorithms are implemented. People may argue that programs written in Ada can always be written in assembly code, so why bother to worry about any underlying language?

From the point of view of classical computation foundations, that statement is valid. But judging from our modern technology, it makes sense to use a better tool if the system efficiency can be improved. Moreover, with the advent of VLSI, we are able to build high performance hardware that realizes the compatibility between the tool and the application in a reasonable cost. The traditional tools for computer graphics are languages such as Basic, Fortran, and Pascal. These languages, however, do not reflect the geometric nature of graphics applications. We claim that applicative languages beat imperative languages in reflecting such a nature because linear algebra computations are in essence functional.

We will investigate the suitability of functional programming techniques for geometric applications in computer graphics. To support our claim, the programming language Scheme is used to implement a linear algebra package and an interactive 3D projection system. Through these examples, we demonstrate the implicit relation between the basic graphics applications and an applicative language. We start with a description of the basic concepts of functional programming. A brief review of Scheme then follows. We continue by introducing the way primitive linear algebra operators are implemented. In the section following we develop a system for 3D projections based upon these primitives. Finally, some anticipated results and research problems are mentioned in the conclusions.

## §2 Functional Programming

A functional language makes use of the properties of mathematical functions. A mathematical function is a relation that maps a set (the *domain*) to a set (the *range*) in a way that for each member in the domain, there is a unique image in the range. Thus a function can be defined by specifying the following: (1) the domain, (2) the range, and (3) the rules of mappings between the two. To *invoke* a function, we apply specific elements in the domain to yield a value in the range. For instance, the lambda expression ([Ch41])

$$\lambda x.\lambda y.xy$$

defines a curried two-argument function of multiplication such that the application (i.e. invocation)

$$((\lambda x.\lambda y.xy)2)3)$$

yields the result 6.

Let $\mathcal{F}$ be the set of all functions. A *functional form* is a function defined from $\mathcal{F}^n$ to $\mathcal{F}$. An important functional form is the function *composition* which takes two functions $F_1$ and $F_2$ as its arguments and yields another function $F$ such that $F$ produces the same result as first applying $F_1$ then applying $F_2$.

So a functional language can be defined axiomatically by specifying:

— Basic data types (the domain and the range).

— Primitive functions.

— Functional forms.

— Application rules.

Due to the fact that languages are implemented in computers which have certain physical constraints, not all properties of mathematical functions can be reflected in functional languages. For example, parameters are names for memory cells where their values are stored in programming languages instead of just the representations of values like in mathematical functions. However, functional languages have comparatively more freedom from side effects and thus more potential for parallel processing than most imperative style languages because the former is based on expressions which yield values whereas the latter is based on sequences of assignments of values to variables. More analyses on these topics can be found in [GhJa] and [He80].

Geometry is the core of computer graphics. Using a functional approach to applications in that category is a natural reflection of their properties. The best examples of pure functional languages are perhaps Lisp defined by McCarthy ([Mc60]) and FP proposed by Backus ([Ba78]). But the purity may somehow confine their power as programming languages. The next section will review a dialect of Lisp which we consider as a better tool.

## §3 A Small Subset of Scheme

The programming language Scheme was designed and first implemented at MIT in 1975 by Gerald J. Sussman and Guy L. Steele, Jr. as part of an effort to understand the actor model of computation. It is based on the lambda calculus described by Alonzo Church ([Ch41]) and serves as "a simple concrete experimental domain for certain issues of programming semantics and style". The revised report on Scheme was published by Steele and Sussman in 1978 ([St78]). In 1980 they had their first Scheme VLSI chip implemented and at the same time a full report on their work was also released ([St80]).

Scheme may be characterized as an applicative order, block structured, lexically scoped, tail recursive dialect of Lisp. Though it has many powerful features, only a small subset common to most Lisp family languages is used in our work. In particular, the language's functional nature turns out to be the dominant factor that influences the way our examples are implemented.

The syntax of the Scheme kernel is as follows :

```
 <expression>  ::=  <constant>
                  | <identifier>
                  | (if <expression> <expression> <expression>)
                  | (lambda (<identifiers>) <expression>)
                  | (change! <identifier> <expression>)
                  | <application>
                  | <syntactic-extension>
<identifiers>  ::=  <empty> | <identifier> <identifiers>
<application>  ::=  (<expressions>)
<expressions>  ::=  <empty> | <expression> <expressions>
<syntactic-extension>  ::=  (<keyword> <objects>)
```

To access a list, the most fundamental data structure in Lisp family languages, we have **car** and **cdr**. These primitives work as follows: **(car** *l***)** returns the first element of *l*; **(cdr** *l***)** returns a list containing everything in *l* excluding the first element. Thus **(cadr** *l***)**, which is equivalent to **(car (cdr** *l***))**, returns the second element of *l*. Similarly, **caddr** gives the third element of a list. To enhance the program readability, we use the aliases **&1**, **&2**, **&3**, **&4**, and so forth to represent, respectively, **car**, **cadr**, **caddr**, **cadddr**, *etc.* Also, we use **&rest** to denote **cdr**, and use **&last** to refer to the last element in a list.

The function **mapcar** takes a function (i.e. a lambda expression) **f** and a list *l*, invokes **f** on each element in *l*, and returns a list containing the results. A number of traditional syntactic extensions (macros) such as **list**, **cond**, **case**, **block**, **let**, and **letrec** ([La65]) are provided to extend the core system. Users may also define new syntactic extensions by using a macro facility.

The Scheme used in our work is ⟦Scheme 311⟧([Fe83]). It is built on top of Franz Lisp ([Fo80]) which is written in language C. Due to this multi-layer structure, the execution efficiency of our Scheme is reduced significantly especially when nested function invocations are performed (which is indeed our case). But that should not be regarded as a key issue because our purpose is to demonstrate functional programming techniques for graphics applications and Scheme is well suited for that goal. In fact, the Scheme efficiency may be promoted if it is implemented in some lower level languages or even as a machine itself.

If functionality is the only consideration, why not use the faster Franz Lisp instead of Scheme? We choose Scheme for several reasons including data abstraction and parallel processing (which can be added to our work quite smoothly when necessary). First, data abstraction is an important modern programming technique which hides unnecessary information of data from the system. Scheme — unlike most Lisp dialects — treats functions and continuations as first class objects. Thus data types can be abstracted as functions in Scheme. Second, it is clear that there is a close tie between function composition and program concurrency. ⟦Scheme 311⟧supports a parallel processing feature which may eventually be the key element in speeding up the bottleneck of our system (e.g. matrix multiplication).

## §4 Linear Algebra Primitives

The basic data types in linear algebra are *vectors* and *matrices*. In Scheme, we use the keywords **vector** and **matrix** to create data of these two types. They are both aliases for the system macro **list**. So

```
(define V (vector 1 2 3 4))
```

binds the vector `[1 2 3 4]` to the identifier $V$. Note that the number of arguments that **vector** takes is arbitrary. Similarly,

```
(define M (matrix (vector 1 2) (vector 3 4)))
```

declares $M$ as the $2\times 2$ matrix $\begin{pmatrix}1&2\\3&4\end{pmatrix}$. Notice that we declare a matrix by specifying its row vectors. The fact that all row vectors must be of the same dimension is an implicit protocol. Our system does not complain about inconsistent row vector dimensions in a matrix declaration.

To access an individual component of a vector or a row vector of a matrix, we have the following primitive:

```
(define &comp
   (lambda (obj index)
      (letrec ([work (lambda (obj index)
                        (if (= index 1)
                            (&1 obj)
                            (work (&rest obj) (1- index))))])
               (if (and (> index 0) (<= index (length obj)))
                   (work obj index)))))
```

which takes an object (a vector or a matrix) and an index, and returns the designated element if the index is between 1 and the dimension of the object. Continuing our previous examples, (&comp $V$ 3) returns 3 and (&comp $M$ 2) returns the vector [3 4]. Furthermore, (&comp (&comp $M$ 2) 2) returns the component 4. The last example introduces the convenience of programming linear algebra computations in a functional way.

The function &ext takes an object $B$ and a non-negative integer $n$ and returns the first $n$ elements of $B$ as an ordered list:

```
(define &ext
   (lambda (OBJ n)
      (letrec ([work (lambda (obj i)
                        (if (= i 1)
                            (list (&1 obj))
                            (cons (&1 obj) (work (&rest obj) (1- i)))))])
                (if (> n (length OBJ))
                    OBJ
                    (work OBJ n)))))
```

In our examples, (&ext $V$ 2) returns the vector [1 2] and (&ext $M$ 1) returns the $1{\times}2$ matrix (1 2).

We now examine the primitives of linear algebra. Things not related to our 3D projection system such as matrix inverse, eigenvalue, Gaussian elimination, *etc.* are not included. As a convention, operators dealing with vectors are prefixed with an asterisk (*) and operators for matrices are prefixed with an exclamation mark (!). Incidentally, the component accessing functions are prefixed with an and sign (&). Also we use square brackets ([]) to denote points, vertices, or vectors, and use round brackets (()) to denote edges, polygons, objects, or matrices.

## 4.1 Primitive Vector Operators

I. Constant Vectors

Some constant vectors used throughout the rest of our examples are:

```
(define VCONST
   (block
      (define *ID2 (vector 0 0))
      (define *ID3 (vector 0 0 0))
      (define *ID4 (vector 0 0 0 0))
      (define *UX3 (vector 1 0 0))
      (define *UX4 (vector 1 0 0 0))
      (define -UX3 (vector -1 0 0))
      (define -UX4 (vector -1 0 0 0))
      (define +UY3 (vector 0 1 0))
      (define +UY4 (vector 0 1 0 0))
      (define -UY3 (vector 0 -1 0))
      (define -UY4 (vector 0 1 0 0))
      (define +UZ3 (vector 0 0 1))
      (define +UZ4 (vector 0 0 1 0))
      (define -UZ3 (vector 0 0 -1))
      (define -UZ4 (vector 0 0 -1 0))
      (define +UH4 (vector 0 0 0 1))))
```

where the IDs are vector additive identities, the UXs are unit vectors in either the positive or negative $x$-axis, similarly for the $y$ and $z$ axes, and finally **+UH4** is the unit vector in homogeneous coordinates from 3D.

## II. Identity and Unit Vectors

In addition to the constant vectors listed above, we often need to have identities of an arbitrary dimension or a unit vector in an arbitrary $n$-dimensional space. The following code satisfies our need.

```
(define *id+
   (lambda (D)
      (letrec ([work (lambda (D)
                        (if (= D 1)
                            (vector 0)
                            (cons 0 (work (1- D)))))])
              (if (> D 0) (work D)))))

(define *unth
   (lambda (D n)
      (letrec ([work (lambda (D)
                        (if (> D 0)
                            (cons (if (= D n) 1 0) (work (1- D)))))])
              (if (> D 0) (reverse (work D))))))
```

Given a dimension, ***id+** returns a zero vector of that dimension. Given a dimension and a specified coordinate, ***unth** returns a vector which has a 1 in the designated position and zeros in all others.

## III. Unary Operators

A unary operator takes a vector as its operand and returns either a scalar or a vector. The function ***dim** returns the dimension of a vector. It is just an alias of the Scheme primitive **length**. However, ***length** returns the length of a vector. Suppose $V$ is a vector, then its length is the square root of the inner product of $V$ and itself, i.e. $|V| = \sqrt{V \cdot V}$. The function ***tag** takes a vector and returns a function that when given an element (the tag) appends the vector with a singleton vector containing the tag. The scalar product of a vector is realized by a curried two argument routine ***scp**. Again, let $V$ be `[1 2 3 4]`, then `((*scp` $V$`) 2)` returns the vector `[2 4 6 8]`. The opposite vector is given by ***opp**. And given a vector, the unit vector in that direction can be obtained by calling ***unit**. The last three functions we present are all good examples of applying **mapcar** mentioned in the previous section. The following code summarizes what we just discussed.

```
(define *dim length)
```

```
(define *length (lambda (v) (if v (sqrt (*dot v v)))))

(define *tag (lambda (v) (lambda (TAG) (append v (vector TAG)))))

(define *scp (lambda (v) (lambda (s) (mapcar (lambda (e) (* e s)) v))))

(define *opp (lambda (v) (mapcar -- v)))

(define *unit
   (lambda (v)
      (let ([l (*length v)])
           (mapcar (lambda (x) (/ x l)) v))))
```

IV. Binary Operators

A binary operator takes two vectors as its arguments and returns either a scalar or a vector. The vector addition operator is defined as follows:

```
(define *add
   (lambda (v1 v2)
      (letrec ([work (lambda (v1 v2)
                        (let ([v1* (&rest v1)]
                              [v2* (&rest v2)])
                             (if v1*
                                 (cons (+ (&1 v1) (&1 v2))
                                       (work v1* v2*))
                                 (vector (+ (&1 v1) (&1 v2))))))])
           (if (and v1 v2)
               (let ([d1 (*dim v1)]
                     [d2 (*dim v2)])
                    (if (= d1 d2)
                        (let ([I  (*id+ d1)])
                             (if (equal? v1 I)
                                 v2
                                 (if (equal? v2 I)
                                     v1
                                     (work v1 v2)))))))))).
```

***add** takes two vectors **v1** and **v2**, first checks if they are of the same dimension. If so, then checks if either one is the zero vector and simply returns the other one, if true. Otherwise it adds each component and returns the sum vector. The component addition is performed by the local recursive routine **work**.

For vector difference, we have ***diff**:

```
(define *diff (lambda (v1 v2) (*add v1 (*opp v2))))
```

which is a simple application of ***add** described above. Suppose $V_1$ and $V_2$ are vectors, then (*diff $V_1$ $V_2$) returns $V_1 - V_2$ instead of the opposite. As it will be clear later, a 3D object can be represented as a collection of edges in which each edge is a list of two vertices. So given an edge, the edge vector between the two vertices can obtained by calling ***diff** with the end point as the first argument and the start point as the second. The following simple routine illustrates this.

```
(define *edgev (lambda (E) (*diff (&2 E) (&1 E))))
```

Now to vector products. We have a routine which computes the inner (dot) product of two vectors as shown above in `*length`.

```
(define *dot
   (lambda (v1 v2)
      (letrec ([work (lambda (v1 v2)
                        (let ([v1* (&rest v1)]
                              [v2* (&rest v2)])
                             (if v1*
                                 (+ (* (&1 v1) (&1 v2))
                                    (work v1* v2*))
                                 (* (&1 v1) (&1 v2)))))])
              (if (and v1 v2)
                  (let ([d1 (*dim v1)]  [d2 (*dim v2)])
                       (if (= d1 d2)
                           (let ([I  (*id+ d1)])
                                (if (or (equal? v1 I) (equal? v2 I))
                                    0
                                    (work v1 v2))))))))).
```

Note the similarity between `*dot` and `*add`. Their major difference is that `*dot` returns a scalar rather than a vector. Notice also that if either of the two vectors is the zero vector, zero is returned immediately.

The outer (cross) product is cleaner but is restricted to 3D only. We have:

```
(define *cross
   (lambda (v1 v2)
      (let ([det (lambda (k)
                    (- (* (&comp v1 (1+ (mod      k 3)))
                          (&comp v2 (1+ (mod (1+ k) 3))))
                       (* (&comp v1 (1+ (mod (1+ k) 3)))
                          (&comp v2 (1+ (mod      k 3))))))])
           (if (and v1 v2)
               (let ([d1 (*dim v1)]  [d2 (*dim v2)])
                    (if (= d1 d2)
                        (if (= d1 3)
                            (vector (det 1) (det 2) (det 3)))))))))
```

in which a local routine `det` for computing determinants is defined. Suppose $V_1$ is `[1 2 3]` and $V_2$ is `[4 5 6]`, `(det 1)` returns the determinant of $\begin{pmatrix}2&3\\5&6\end{pmatrix}$, `(det 2)` returns that of $\begin{pmatrix}3&1\\6&4\end{pmatrix}$, and `(det 3)` returns that of $\begin{pmatrix}1&2\\4&5\end{pmatrix}$. Note that `(*cross` $V_1$ $V_2$`)` returns $V_1 \times V_2$ instead of the reverse. The *ad hoc* rule that the right thumb points to the direction of the cross product of two vectors is implied by a right-handed system. In a left-handed system, however, the same routine holds, but the tool that determines the direction must be replaced by the left hand correspondingly.

### 4.2 Primitive Matrix Operators

I. Constant Matrices

Two constant matrices are of special importance to our application, namely, the $4\times4$ matrix additive identity and the $4\times4$ matrix multiplicative identity.

```
(define MCONST
   (block
      (define MI4+ (matrix +ID4 +ID4 +ID4 +ID4))
      (define MI4* (matrix +UX4 +UY4 +UZ4 +UH4))))
```

II. Identity Matrices

Given any dimension, we can compute the matrix additive and multiplicative identities. Here the dimension is a list of two integers rather than a single one. The routine `!id+` returns an $i \times j$ zero matrix if the dimension $(i\ j)$ is specified. On the other hand, `!id*` returns the $i \times j$ matrix multiplicative identity whose n^th^ row has an 1 in its n^th^ position and zeros elsewhere. The following routines shows this.

```
(define !id+
   (lambda (D)
      (let ([i (&1 D)]
            [j (&2 D)])
           (let ([I (*id+ j)])
                (letrec ([work (lambda (n)
                                   (if (= n 1)
                                       (matrix I)
                                       (cons I (work (1- n)))))])
                       (work i))))))

(define !id*
   (lambda (D)
      (let ([i (&1 D)]
            [j (&2 D)])
           (letrec ([work (lambda (n)
                              (if (= n 1)
                                  (matrix (*unth j n))
                                  (cons (*unth j n) (work (1- n)))))])
                  (reverse (work i))))))
```

where `*id+` and `*unth` are the operators mentioned earlier which return a zero vector and a unit vector, respectively.

III. Unary Operators

We now define a few basic unary operators on matrices. The dimension of a matrix is given by invoking `!dim`. The additive inverse of a matrix can be obtained by calling `!opp`. A most useful operator is the matrix transpose operator which returns a matrix whose column vectors are identical to the row vectors of the original one. Finally, `!det2` gives the determinant of a $2 \times 2$ matrix.

```
(define !dim (lambda (m) (list (length m) (*dim (&1 m)))))

(define !opp (lambda (m) (mapcar *opp m)))

(define !xpose
   (lambda (m)
     (if (&1 m)
         (cons (mapcar &1 m) (!xpose (mapcar &rest m))))))

(define !det2
   (lambda (m)
      (let ([v1 (&1 m)]
            [v2 (&2 m)])
           (- (* (&1 v1) (&2 v2))
              (* (&1 v2) (&2 v1)))))).
```

So (!dim *M*) returns the list (*i j*) if *M* is an $i \times j$ matrix. **!opp** takes the opposite of each of the row vectors in the original matrix and returns a new one. The routine **!xpose** gives the transpose of its operand. It takes full advantages of **mapcar** in a way that the first column vector can be extracted and the routine itself can recurse on the rest of the column vectors. Finally, **!det2** does a standard computation to get the determinant of a 2 by 2 matrix.

## IV. Binary Operators

Based on binary vector operators, we can further define binary operators on matrices. First is matrix addition.

```
(define !add
   (lambda (m1 m2)
      (letrec ([work (lambda (m1 m2)
                        (let ([v1 (&1 m1)]
                              [v2 (&1 m2)])
                             (if v1
                                 (cons (*add v1 v2)
                                       (work (&rest m1) (&rest m2))))))])
              (if (and m1 m2)
                  (let ([d1 (!dim m1)]
                        [d2 (!dim m2)])
                       (if (equal? d1 d2) (work m1 m2))))))).
```

In adding vectors, the real add operation will not occur unless the operands are of the same dimension and neither of them is the identity. Unlike its counterpart in vectors, **!add** checks only if its two operands are of the same dimension. The purpose of identity verification is to avoid the time consuming addition process. Since a similar testing is performed by ***add** at a lower level, the procedure is eliminated here.

Now to matrix multiplication. The operation is realized in a brute force fashion; a row vector is multiplied by a column vector to yield an entry. We have:

```
(define !mult2
   (lambda (m1 m2)
      (letrec ([work (lambda (m1 m2)
                        (let ([v (&1 m1)])
                             (if v
                                 (cons (mapcar (lambda (x) (*dot v x)) m2)
                                       (work (cdr m1) m2)))))])
               (if (and m1 m2)
                   (let ([d1 (!dim m1)]
                         [d2 (!dim m2)])
                        (if (equal? (&2 d1) (&1 d2))
                            (work m1 (!xpose m2))))))))
```

which takes the dot product of each row vector in **m1** and each row vector in **m2** transpose and forms the product matrix. Recall that since our emphasis is to demonstrate graphics programming in a functional style, matrix multiplication in our system is implemented in a naive way although some algorithms have been proposed to speed up such an operation. Clearly matrix multiplication is the bottleneck of all systems built upon our linear algebra primitives. We believe, however, that this efficiency problem can be solved if the device architecture reflects the underlying functionality.

V. N-ary Operators

N-ary operators can be implemented using the macro facility which expands the source into a series of binary operations. The most useful n-ary operator in matrix manipulation is the generalized multiplication operator which takes an arbitrary number of matrices as operands and returns their composite, if possible. We define the following macro **!mult**:

```
(macro !mult
   (lambda (l)
      (let ([mat-list (&rest l)])
           (case (length mat-list)
              [0   nil]
              [1   (&1 mat-list)]
              [2   (cons '!mult2 mat-list)]
              [otherwise (list '!mult2
                                (&1 mat-list)
                                (cons '!mult (&rest mat-list)))]))))).
```

## §5 3D Projections

Projections transform points in an $n$-dimensional coordinate system into points in a coordinate system of dimension less than $n$. Here we concern only with the class of *planar projections* in which points are projected onto a plane rather than a curved surface using straight projectors. Specifically, in this case, a projection of a line segment is still a line, hence only its endpoints have to be projected. In particular, the projections in our computer context are the ones that transform objects from 3D (the visual world) into 2D (the computer graphics display). These projections can be categorized into two subclasses: *perspective* projections and *parallel* projections.

The basics of 3D or 2D geometric manipulations are the linear algebra primitives presented previously. Other major issues include representations of entities, coordinate systems, transformations, hidden face elimination, *etc.* The art of functional programming and the concepts in linear algebra will be the center of the code we present.

### 5.1 Representations of Entities

For convenience, the geometric entities of interest are restricted to straight lines which only include, as a hierarchy, *points* (*vertices*), *edges, polygons* (*faces*), and *objects*. The first order entity is denoted as a vector, the rest, matrices. Figure 1 shows a tent (let it be called $\mathcal{T}$ ) in a right-handed 3D coordinate system. Note that the highest order geometric entity given in $\mathcal{T}$ is the tent itself which is a collection of faces; each face is a collection of edges; each edge is a pair of vertices. Once again, we let **point**, **edge**, **polygon**, and **object** all be aliases of **list**.

From the user's point of view, the simplest way to represent 3D straight line objects is to denote the second order entities (the polygons) as a list of vertices directly.

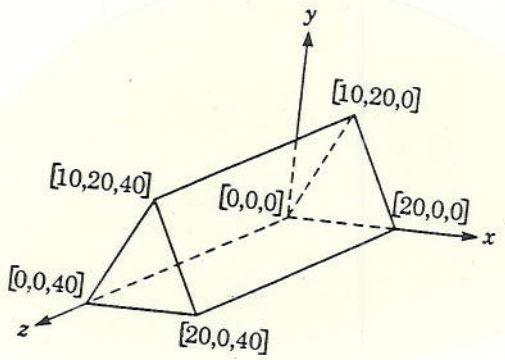

*Figure 1. A tent in a right-handed system*

As a result, the third order entities (edges) become implicit. Let $R$, $L$, $B$, and $F$ be the right, left, back, and front faces of $\mathcal{T}$, respectively (bottom face need not be considered). We can define:

```
(define R (polygon '[10 20 40] '[10 20 0] '[20 0 0] '[20 0 40]))
(define L (polygon '[10 20 0] '[10 20 40] '[ 0 0 40] '[ 0 0 0]))
(define B (polygon '[10 20 0] '[ 0 0 0] '[20 0 0]))
(define F (polygon '[10 20 40] '[20 0 40] '[ 0 0 40])).
```

In the examples the vertices in a face are sequenced clockwise. This ordering is essential to our simple hidden face algorithm (see Section 5.5). Thus $\mathcal{T}$ is just

```
(define T (object R L B F))
```

The vertex representation of a polygon can easily be transformed into its corresponding edge representation. For example, $F$ can be redefined as a collection of three edges $E_1$, $E_2$, and $E_3$, where $E_1 =$`(edge '[10 20 40] '[0 0 40])`, $E_2 =$`(edge '[0 0 40] '[20 0 40])`, and $E_3 =$`(edge '[20 0 40] '[10 20 40])`. The following routine does the transformation.

```
(define xrep
   (lambda ()
      (letrec
         ([xf (lambda (msg)
                 (case msg
                    [poly (lambda (P)
                             (letrec ([work (lambda (E)
                                                (if (&rest E)
                                                   (cons (edge (&1 E) (&2 E))

                                                         (work (&rest E)))
                                                   (polygon
                                                      (edge (&1 E) (&1 P)))))])

                                (if (> (!dim POLYGON) 2)
                                    (work P)
                                    P)))]
                    [obj  (lambda (OBJECT) (mapcar (xf 'poly) OBJECT))]))])
         xf))).
```

The above routine can transform an object represented in vertex-oriented faces into a representation of edge-oriented faces.

## 5.2 Homogeneous Coordinates

The concepts of homogeneous coordinates in geometry were applied to computer graphics in the 1970's. If a point or a vector in an $n$-dimensional space is $[x_1\ x_2\ \ldots\ x_n]$, it can be represented in a homogeneous coordinate system as $[hx_1\ hx_2\ \ldots\ hx_n\ h]$, where $h$ is any non-zero scalar. Thus if $[z_1\ z_2\ \ldots\ z_n\ h]$ is a point in its homogeneous system, its original coordinates are $[\frac{z_1}{h}\ \frac{z_2}{h}\ \ldots\ \frac{z_n}{h}]$. Without loss of generality, we let $h = 1$ so that to obtain the non-homogeneous coordinates requires only the elimination of the homogeneous coordinate instead of a series of divisions.

The significance of homogeneous coordinates is that transformations can be performed "homogeneously" via a matrix multiplication model. This idea will be clear when we introduce transformation matrices in Section 5.3. Let us now first investigate how to get the homogeneous coordinates for points in Scheme. We have the following routine:

```
(define hom
   (lambda ()
      (letrec
         ([hf (lambda (msg)
                 (case msg
                    [point (lambda (POINT)   ((*tag POINT) 1))]
                    [edge  (lambda (EDGE)    (mapcar (hf 'point) EDGE))]
                    [poly  (lambda (POLYGON) (mapcar (hf 'poly) POLYGON))]
                    [obj   (lambda (OBJECT)  (mapcar (hf 'obj) OBJECT))]))])
          hf)))
```

which returns the homogeneous coordinates for a geometric entity (point, edge, polygon, or object) according to the message specified. For instance, if $E$ is an edge denoted by the pair of vertices `([1 2 3] [4 5 6])`, then `(((hom) 'edge)` $E$`)` yields `([1 2 3 1] [4 5 6 1])`. Note that to get the homogeneous coordinates of a point, we simply append the original coordinates with the vector `[1]`. The homogenization of higher order entities is just a series of **mapcar** down to the lowest level (homogenization of a point).

To dehomogenize points is less trivial. Intuitively, since the homogeneous coordinate is always 1, we can just discard that coordinate and return the rest. However, since the points usually are involved in a series of transformations, there is no guarantee that the value can be preserved. Consequently, we must divide each non-homogeneous coordinate by the value of the homogeneous coordinate if it is neither 0 nor 1. The following code shows what we want.

```
(define deh
   (lambda (D)
      (letrec
         ([df (lambda (msg)
                 (case msg
                    [point (lambda (POINT)
                              (let ([h (float (&last POINT))]
                                    [l (&ext POINT
                                             (if (> (*dim POINT) D) D (1- D)))])
                              (if (or (= h 0) (= h 1))
                                  1
                                  (mapcar (lambda (p) (/ p h)) l))))]
                   [edge  (lambda (EDGE)    (mapcar (df 'point) EDGE))]
                   [poly  (lambda (POLYGON) (mapcar (df 'edge) POLYGON))]
                   [obj   (lambda (OBJECT)  (mapcar (df 'poly) OBJECT))]))])
       df))).
```

The structure of **deh** is symmetric to that of **hom**. Unlike **hom**, however, the dimension of the dehomogenized coordinate system must be specified. Let $P$ be the point `[2 2 2 2]`, then `(((deh 2) 'point)` $P$`)` yields the 2-dimensional point `[1.0 1.0]`. Note that the dehomogenization will make sense only if the resulting dimension is less than its operand's current dimension.

## 5.3 Transformation Matrices

Based upon the homogeneous system, all 3D transformations can be represented by matrices.

I. Translation

In a fixed coordinate system, translating a point to the origin is equivalent to translating the world to the new position in an opposite direction. Thus, this translation can be viewed as creating a new coordinate system with origin at that point. Consider a point $P=$ [$x_0$ $y_0$ $z_0$] in 3D. An arbitrary point $Q$ in the world relative to $P$ can be obtained by

$$[x\ \ y\ \ z\ \ 1]\cdot\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_0 & -y_0 & -z_0 & 1 \end{pmatrix}$$

where [$x$ $y$ $z$ 1] is the homogeneous point of $Q$ in the old world.

In Scheme, this translation matrix can be represented by:

```
(define xlate
  (lambda (P)
     (matrix *UX4  *UY4  *UZ4 (((hom) 'point) (*opp P)))))
```

where P is the new origin after translation.

II. Scaling

The transformation of a point $Q =$ [$x\ y\ z$] with a scale $S =$ [$s_1$ $s_2$ $s_3$], which gives an $x$-direction scale of $s_1$, a $y$-direction scale of $s_2$, and a $z$-direction scale of $s_3$, can be represented as:

$$[x\ y\ z\ 1]\cdot\begin{pmatrix} s_1 & 0 & 0 & 0 \\ 0 & s_2 & 0 & 0 \\ 0 & 0 & s_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

In Scheme, the scaling matrix is defined as:

```
(define scale
   (lambda (S)
      (matrix (vector (&1 S)    0       0   0)
              (vector   0     (&2 S)    0   0)
              (vector   0       0    (&3 S) 0)
              *UH4)))
```

where S is the scaling factor.

III. Translation and Scaling Combined

It is not unusual that a transformation is sometimes a combination of a translation and a scaling. Considering the $P$ and $S$ as before, in matrix representation we have:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_0 & -y_0 & -z_0 & 1 \end{pmatrix} \cdot \begin{pmatrix} s_1 & 0 & 0 & 0 \\ 0 & s_2 & 0 & 0 \\ 0 & 0 & s_3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} s_1 & 0 & 0 & 0 \\ 0 & s_2 & 0 & 0 \\ 0 & 0 & s_3 & 0 \\ -s_1x_0 & -s_2y_0 & -s_3z_0 & 1 \end{pmatrix}.$$

Note that matrix multiplication is not commutative, so translation and scaling combined differs from scaling and translation combined.

A typical example in 3D projection process that belongs to this category is the view port to physical device coordinates transformation. It takes a transformed view reference point (not necessarily in the origin) and a view window (a list of four corner coordinates) and yields a system with origin at the bottom left corner of the video display and scaled from the window sizes to the screen sizes. The routine **xcale** gives such a matrix:

```
(define xcale
   (lambda (VRP W)
      (let ([VRPx (&1 VRP)]  [VRPy (&2 VRP)]
            [umin (&1 W)]    [umax (&2 W)]
            [vmin (&3 W)]    [vmax (&4 W)])
         (let ([Px (+ umin VRPx)]
               [Py (+ vmin VRPy)]
               [Sx (/ *xres (- umax umin))]
               [Sy (/ *yres (- vmax vmin))])
            (matrix (vector       Sx              0        0 0)
                    (vector       0               Sy       0 0)
                    (vector       0               0        0 0)
                    (vector (-- (* Sx Px)) (-- (* Sy Py)) 0 1))))))
```

where ***xres** and ***yres** are the resolutions of the video screen global to the whole system. Notice that in this particular example, the combined transformation works on 2D only, hence the third row vector as well as the third column are both zero vectors.

IV. Rotation

An *othogonal* matrix satisfies (1) row vectors are unit vectors, and (2) row vectors are mutually perpendicular. A rotation around any of the three principle axes is an othogonal transformation. The implication is that the composition of any number of rotations is still othogonal. Also, rotations preserve lengths and angles while translations and scalings don't.

Consider a typical case where we have three points in a right-handed system, say $P_0$, $P_1$, and $P_2$. Our goal is to get a rotation matrix that when applied to the three points yields the new $\overline{P_0P_1}$ parallel to the negative $z$-axis, the new $\overline{P_0P_2}$ parallel to the positive $y$-axis, and the normal to the plane of $P_0$, $P_1$, and $P_2$ parallel to the positive $x$-axis. Since the composite rotation matrix is othogonal, it can be realized by taking unit vectors in the various directions mentioned above. If $M$ is the upper left $3 \times 3$ matrix of our rotation submatrix, then $M$'s 3rd column is the unit vector in the negative $\overline{P_0P_1}$ direction, its 2nd column is the unit vector normal to the plane determined by those three points, and its 1st column is just the cross product of the other two unit vectors. A more detailed discussion on this topic is given in [FoVa].

To be more precise, the following Scheme code returns such a rotation matrix:

```
(define rotate
   (lambda (VPN VUP)
      (let ([rz (*opp (*unit VPN))]
            [rx (*unit (*cross VPN VUP))])
           (let ([ry (*cross rz rx)])
                (!xpose (matrix ((*tag rx) 0)
                                ((*tag ry) 0)
                                ((*tag rz) 0)
                                *UH4))))))
```

where VPN and VUP are analogous to the vectors $\overline{P_0P_1}$ and $\overline{P_0P_2}$, respectively. Their geometric meanings will be clear when projections are introduced in Section 5.4.

V. RHS to LHS

The right-handed system is familiar to most people. However, in 3D computer graphics, the left-handed coordinate system is more realistic because having the positive $z$-axis pointing into the screen rather than out of the screen reflects the correct relationship between an object's viewing size and its distance from the viewer. As a result, the transformation from a right-handed system to a left-handed one is useful. We have the matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

or equivalently:

```
(define r-1 (lambda () (matrix *UX4 *UY4 -UZ4 *UH4))).
```

## 5.4 Projection Matrices

There are basically two subclasses in planar geometric projections: *perspective* and *parallel*. If the distance from the center of projection to the projection plane is finite, the projection is perspective, otherwise it is parallel. So for perspective projections, we specify the *center of projection* (COP); for parallel projections, we specify the *direction of projection* (DOP).

The perspective projection creates a more realistic view. In perspective projections, an object's projected size is inversely proportional to its distance from the center of projection, hence a realistic view is reflected. This effect is known as the *perspective foreshortening*. In parallel projections, the foreshortening phenomenon is less obvious. However, parallelism among lines is preserved in parallel projections but is generally lost in perspective projections.

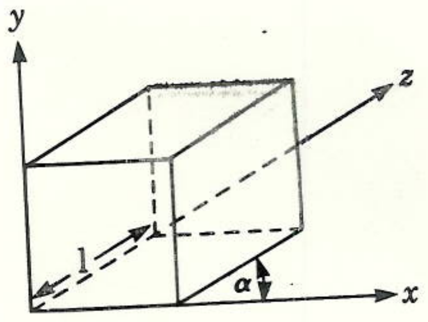

*Figure 2. A projected unit cube in the xy-plane*

I. Perspective Projections

The family of lines not parallel to the projection plane will converge to a point called the *vanishing point* in perspective projections. If the family is parallel to one of the three principle axes, the vanishing point is further described as *principle.* Perspective projections are categorized by the number of principle vanishing points in the system. In other words, they are characterized by the number of principle axes cut by the projection plane. A simple calculation (see, for example, [FoVa]) shows the one-point perspective projection can be given by:

```
(define per
   (lambda (d)
      (matrix *UX4
              *UY4
              (vector 0 0 1 (/ 1.0 d))
              *ID4)))
```

where **d** is the distance from the projection plane to the COP.

II. Parallel Projections

Parallel projections are characterized by the relation between their DOP and corresponding projection planes. A parallel projection can be categorized as *othographic* if its DOP is normal to the projection plane, otherwise it is *oblique.* Consider a unit cube in the left-handed system projected onto the $xy$-plane as illustrated in figure 2. The DOP of the parallel projection is, in general, given by $[l\cos\alpha \;\; l\sin\alpha \;\; 1]$. If $l = 0$, we have an othographic parallel projection, otherwise $(l > 0)$ it is an oblique parallel projection. In particular, if $l = 1$, the projection is named *cabinet,* and if $l = \frac{1}{2}$, it is called *cavalier.* Thus if $l$ and $\alpha$ are specified, we can use the following code to get the DOP:

```
(define mkDOP
   (lambda (l alfa)                              ; alfa in degrees
      (if (0? l)
          *UZ3
          (let ([alfa-prime (de-rad alfa)])
               (vector (* l (cos alfa-prime))
                       (* l (sin alfa-prime))
                       1))))).
```

Note that if $l = 0$, the DOP is just the positive $z$-axis.

The parallel projection matrix returned by **par** maps the world onto the projection plane ($xy$-plane):

```
(define par
   (lambda (DOP)
      (matrix *UX4
              *UY4
              (vector (&1 DOP) (&2 DOP) 0 0)
              *UH4)))
```

where DOP is the direction of projection produced by mkDOP.

Incidently, the othographic projection does not have to go through the process of mkDOP and its corresponding matrix can be generated more efficiently by:

```
(define oth (lambda () (matrix *UX4 *UY4 *ID4 *UH4)))
```

whose 3rd row is a zero vector and the other rows are the unit vectors identical to those given in par.

## 5.5 Mappings of 3D Entities

So far we have constructed a unified model of 3D transformations (including projections) in terms of homogeneous 4×4 matrices. Our motivation is to make the mappings of 3D geometric entities work as a continuous procedure. This procedure can easily be established based upon the functional nature of matrix multiplication in linear algebra and in our underlying language. Hence given an arbitrary geometric entity $G$ and a transformation matrix $M$, the result of $M$ applied to $G$ can be summarized by:

```
(define map
   (lambda (M)
      (letrec
         ([mf (lambda (msg)
                 (case msg
                    [point (lambda (POINT)   (!mult2 (matrix POINT) M))]
                    [edge  (lambda (EDGE)    (!mult2 EDGE M))]
                    [poly  (lambda (POLYGON) (mapcar (mf 'edge) POLYGON))]
                    [obj   (lambda (OBJECT)  (mapcar (mf 'poly) OBJECT))]))])
          mf)))
```

which is in fact a series of matrix multiplications. Recall that the first order entities (points) are represented as vectors and the higher order entities are represented as matrices. Therefore, if the entity is first order (a point), it must be converted into a matrix before the multiplication can be performed.

## 5.6 3D Views

To get an arbitrary 3D view, a number of parameters must be specified. First, we need a *view plane* (or *projection plane*) which is determined by a point called the *view reference point* (VRP) on the plane, a vector called the *view plane normal* perpendicular to the plane, and another vector called the *view up vector* (VUP). Second, a *view window* relative to the VRP must be given. This is a list of the four corner vertices of a rectangular region in the view plane. The window also determines a 2D coordinate system with origin at the VRP and the $y$-axis parallel the projection of the VUP along the direction of VNP to the view plane. Collectively, these planar coordinates and the VPN (as the $z$-axis) form a left-handed system (so that the $x$-axis can also be uniquely determined). Finally, we need to specify a *view volume* in order to get a 3D view. This volume is defined by the window and the COP (in perspective projections) or the DOP (in parallel projections). In the case of perspective projections, the view volume will be a pyramid with apex at the COP and sides passing through the window and expanding to infinity. For parallel projections, it is an infinite rectangular pipe with direction parallel to the DOP and sides passing through the window. The importance of the view volume is that the world is confined in this region against which the projected objects can be clipped.

The view volume may become finite if front and back faces are specified. Given a closed view volume, six-face clipping on a projected object can be applied (see Section 5.7). In our system, a plane $ax + by + cz + d = 0$ can be represented as a homogeneous vector `[`$a\ b\ c\ d$`]`. A volume, as a direct result, may then be represented as a $6 \times 4$ matrix in which each row vector is a plane. Suppose `W` is the window, `FB` is the list of the distances of the front and back faces from the view plane, and VRP and DOP are as defined, then a view volume in either projection can be given by `vvol`:

```
(define vvol
   (lambda (W FB)
      (let ([umin (&1 W)]     [umax (&2 W)]
            [vmin (&3 W)]     [vmax (&4 W)]
            [F    (&1 FB)]    [B    (&2 FB)])
        (lambda (msg)
           (case msg
              [perspective
                 (lambda (VRP)
                    (let ([Vx (&1 VRP)]  [Vy (&2 VRP)]  [Vz (&3 VRP)])
                      (let ([xmin (+ Vx umin)]  [xmax (+ Vx umax)]
                            [ymin (+ Vy vmin)]  [ymax (+ Vy vmax)])
                        (let ([v1 (vector xmax ymin Vz)]
                              [v2 (vector xmax ymax Vz)]
                              [v3 (vector xmin ymax Vz)]
                              [v4 (vector xmin ymin Vz)])
                          (matrix
                             ((*tag (*opp (Vx3 v1 v2))) 0)         ; right
                             ((*tag (*opp (Vx3 v3 v4))) 0)         ; left
                             ((*tag (*opp (Vx3 v2 v3))) 0)         ; top
                             ((*tag (*opp (Vx3 v4 v1))) 0)         ; bottom
                             (vector 0 0 1 (-- (+ Vz B)))          ; back
                             (vector 0 0 1 (-- (+ Vz F)))))))]     ; front
              [parallel
                 (lambda (DOP)
                    (let ([N (*opp (*cross V -UX3))])
                        (matrix
                           (vector 1 0 0 (-- umax))                ; right
                           (vector 1 0 0 (-- umin))                ; left
                           ((*tag N) (-- (*dot N (vector umax vmax 0)))) ;top
                           ((*tag N) (-- (*dot n (vector umin vmin 0)))) ;bot
                           (vector 0 0 1 (-- B))                   ; back
                           (vector 0 0 1 (-- F))))))])))))         ; front
```

Each row vector in the returned matrix (volume) is an equation denoting one of the volume's side planes. The ordering of the side planes in the volume is arbitrary. However, for the purpose that will not clear until clipping is introduced, the order of (1) right, (2) left, (3) top, (4) bottom, (5) back, and (6) front is adopted.

## 5.7 3D Clipping

We now present a 3D version of the Cohen-Sutherland clipping algorithm. A detailed description of its 2D version can be found in [FoVa].

Let $L$: $ax + by + cz + d = 0$ be a plane. Then $L$ can be represented by the vector $[a\ b\ c\ d]$. We claim that a point $P = [x_0\ y_0\ z_0\ 1]$ is in the positive side of the space relative to $L$ if $ax_0 + by_0 + cz_0 + d > 0$. This is equivalent to saying that the inner product $P{\cdot}L > 0$. Conversely, $P$ is in the non-positive side of $L$ if $P \cdot L <= 0$. Thus we can represent the *outcode* of a point relative to a volume as a 6D vector of 1's and 0's where a 1 corresponds to the fact that the point is outside the volume relative to a specific side plane and a 0 corresponds to reverse case. Using the ordering mentioned earlier, the outcode referring to the point is, in order, (1) right of, (2) left of, (3) above, (4) beneath, (5) behind, or (6) in front of the view volume if its corresponding position has an 1. The routine **outcodes** gives such a vector.

```
(define outcodes
   (lambda (p VV)
      (letrec
         ([work (lambda (volume even?)
                   (if volume
                       (cons (if ((if even? > <) (*dot p (&1 volume)) 0) 1 0)
                             (work (&rest volume) (not even?)))))])
         (work VV t))))
```

where p is the point to be examined and VV is the target view volume. Note that the outcode has a bit 1 in its $i^{th}$ position only if the point is outside the volume relative to its $i^{th}$ side plane. The outside refers to, for even numbered side planes (right, top, and back), the positive half-space, or for odd numbered side planes (left, bottom, and front), the negative half-space. This explains why the adopted ordering is essential to specifying the view volume. Furthermore, the following constant vectors play special roles in our clipping procedure:

```
(define *allzero (vector 0 0 0 0 0 0))

(define *allone  (vector 1 1 1 1 1 1)).
```

As far as the six-face clipping against a view volume is concerned, an edge is trivially rejected (i.e. both endpoints are outside the volume) if the outcodes of its two endpoints have an inner product greater than zero. Similarly, an edge is trivially accepted (i.e. neither of the endpoints is outside the volume) if their outcodes are both zero vectors. That is:

```
(define reject (lambda (OC1 OC2) (> (*dot OC1 OC2) 0)))

(define accept
   (lambda (OC1 OC2)
       (and (equal? OC1 *allzero)
            (equal? OC2 *allzero)))).
```

The intersection of an edge and a plane can be computed in vector form as follows. Let $E = \overline{P_1 P_2}$ be an edge and $H$ be a plane. The intersection point $P$ of $E$ and $H$ is

$$P = P_1 + -\tfrac{P_1 \cdot H}{E \cdot H}\, E$$

which can also be given by:

```
(define intersect
   (lambda (E PL)
      (let ([p1 (&1 E)])
           (let ([ve (*edgev E)])
                (((homo) 'point)
                 (&ext (*add p1
                             ((*scp ve) (/ (float (-- (*dot p1 PL)))
                                           (*dot ve PL))))
                      3)))))).
```

The following routine clips a geometric entity of the 2nd order or higher against a view volume.

```
(define clip
 (lambda (VV)
  (letrec
   ([*mapcar (lambda (f l)
                (letrec ([work (lambda (l)
                                 (if l
                                     (let ([ans (f (&1 l))])
                                          (if ans
                                              (cons ans (work (&rest l)))
                                              (work (&rest l))))))])
                         (work l)))]
    [cf (lambda (msg)
         (case msg
           [edge (lambda (EDGE)
                   (letrec
                    ([work (lambda (E oc2 order)
                            (let ([oc1 (outcodes (&1 E) VV)])
                              (if (accept oc1 oc2)
                                  (if order E (reverse E))
                                  (if (> (*dot oc1 *allone) 0)
                                      (work (edge (intersect E
                                                             (which? oc1 VV))

                                                  (&2 E))
                                            oc2
                                            order)
                                      (work (edge (intersect (reverse E)
                                                             (which? oc2 VV))

                                                  (&1 E))
                                            oc1
                                            (not order)))))))])
                    (let ([oc1 (outcodes (&1 EDGE) VV)]
                         [oc2 (outcodes (&2 EDGE) VV)])
                        (if (reject oc1 oc2) nil (work EDGE oc2 t)))))]
          [poly (lambda (POLYGON) (*mapcar (cf 'edge) POLYGON))]
          [obj  (lambda (OBJECT)  (*mapcar (cf 'poly) OBJECT))]))])
  cf)))
```

where ***mapcar** is a special case of **mapcar** in the sense that only **non-nil** results are returned in a list. The recursive local routine **cf** is a function of one argument which clips an edge, a polygon, or an object depending on the message specified. As before, operations on higher order entities are built on top of lower ones via the **mapcar** technique. Here, ***mapcar** is used instead because the trivially rejected lower order entities need not to be involved in further transformations.

The Cohen-Sutherland algorithm is implemented as a function bound to the message designator **edge** in the **cf** body. The local routine **work** takes a line segment (**E**), its second endpoint's outcode (**oc2**), and a Boolean value (**order**) which keeps track of the endpoint switching, and tries to compute the intersections of **E** and the view volume, if possible. It will halt only if the edge is either trivially rejected, trivially accepted, or non-trivially accepted when the intersections are found. In our algorithm the edge's first endpoint is always the one that is closer to the view volume. This ordering is not necessarily consistent with what's in the object's original representation which must be preserved for hidden face elimination analysis. Finally, we define:

```
(define which?
  (lambda (OC VV)
    (cond [(= (&1 OC) 1) (&1 VV)]
          [(= (&2 OC) 1) (&2 VV)]
          [(= (&3 OC) 1) (&3 VV)]
          [(= (&4 OC) 1) (&4 VV)]
          [(= (&5 OC) 1) (&5 VV)]
          [(= (&6 OC) 1) (&6 VV)])))
```

that returns the first side plane for which a point is outside the view volume.

In summary, by Cohen-Sutherland's algorithm, a 3D object can be clipped against a 3D view volume before the projected picture is plotted. This is parallel to the case that a 2D object can be clipped against a 2D closed region. [FoVa] suggests an additional process called *shearing* that transforms the view volume into a *canonical* form (the unit pyramid in perspective projections or the unit cube in parallel projections). Clearly, its purpose is to simplify the intersection calculation in clipping. In our implementation, however, such a step is omitted and the object is clipped directly against the "raw" volume. This is due to the following facts: (1) to obtain the canonical volume is no faster than to obtain the raw volume, and (2) to compute the intersections between the object and either volume requires the same procedure, and finally, (3) shearing introduces yet another scaling transformation in addition to itself (see [FoVa]). It is obvious that, to our system, shearing is nothing but overhead.

## 5.8 Hidden Face Elimination

The visual effect of a projected object can be further improved if its hidden faces are eliminated. Earlier, we defined a face by its vertices in a clockwise order. This ordering is essential to the hidden face elimination of a 3D object. [Ha83] proposes the following simple analysis.

Let $F$ be a face with the vertex representation of ($P_0$ $P_1$ $P_2$ ...). The order of $P_0$, $P_1$, $P_2$, ..., *etc.* is clockwise. $F$ can also be represented by its corresponding edge form ($E_1$ $E_2$ ...), where $E_1 =$ ($P_0$ $P_1$), and $E_2 =$ ($P_1$ $P_2$). Let $N$ be normal to $F$ and $N = E_1 \times E_2$.

Consider first the case of perspective projections. Let $V$ be the vector from the COP to any point in $F$, typically, the vector from the COP to $P_0$. We claim that $F$ is visible if the angle from $N$ to $V$ is greater than 90 degrees. More specifically, if $N{\cdot}V < 0$, then $F$ is a light face, else it is a dark face. Since $V$ is the direction that light propagates from the viewer's eyeball and $N$ is the direction of light reflected by $F$, then if the two directions are opposite relative to the projection plane, the reflected light can be sensed by the viewer. Therefore the face is visible.

The case of parallel projections is even simpler. The same argument holds with the vector $V$ replaced by the DOP. Note, however, that $N$ is obtained by a cross product in a left-handed system. As mentioned previously, `*cross` works as well in a LHS, but the conventional "direction of thumb" is given by the left hand rather than the right.

For an arbitrary object whose composing faces are in edge representation, the following routine `hfe` returns a list of Boolean values in which a `t` corresponds to a light face and a `nil` corresponds to a dark face. This list will then be sent, together with the object, to the plotting routine which draws appropriate lines (solid vs. dashed, or solid vs. blank) based upon its individual components. We show `hfe` below.

```
(define hfe
 (lambda (OBJ)
  (lambda (msg)
   (case msg
     [yes (lambda (R)
            (letrec
              ([work (lambda (OBJ)
                       (let ([POLY (&1 OBJ)]
                             [OBJ* (&rest OBJ)])
                        (let ([e1 (((deh 3) 'edge) (&1 POLY))]
                              [e2 (((deh 3) 'edge) (&2 POLY))])
                          (let ([c (*dot (if R R (&1 e1))
                                         (*cross (*edgev e1) (*edgev e2)))])
                            (let ([visibility (< c 0)])
                              (if OBJ*
                                  (cons visibility (work OBJ*))
                                  (list visibility)))))))])
             (if OBJ (work OBJ))))]
     [no  (mapcar (lambda (e) t) OBJ)])))).
```

We implement the algorithm described above as a function of one argument bound to the message designator `yes`. The argument `R` is `nil` for perspective projections because the vector $V$ is just $P_0$ (because the COP is at the origin) while it is the DOP in parallel projections for the reason argued before. Note that if the designator `no` is specified when invoking `hfe`, a list of `t`'s will be returned implying no hidden face elimination is requested.

It must be emphasized that the algorithm presented here does not work for the projections of overlapped objects.

## 5.9 Plotting

The plotting of an object is the last step in a complete 3D projection process. If there is no hidden face elimination involved, the plotting is straightforward: grab an edge and draw a line between the endpoints. Our system supports a simple case of hidden face elimination and the plotting routine becomes less trivial.

An important property is that a boundary line between adjoining faces can be both visible and invisible due to the fact that one face is light while the other is dark. For this kind of dual attribute edge, our decision is to favor its visible attribute. The strategy is:

> **1.** *If an edge belongs to a light face, we simply draw a light line if it hasn't been drawn as a light line, or we redraw a light line if it has been drawn as a dark line, otherwise we draw nothing.*
>
> **2.** *If it belongs to a dark face, we draw nothing if it hasn't been drawn as a light line, or we draw a dark line if it hasn't been drawn as a dark line, otherwise we draw nothing.*

Don't be confused by the terms used: light lines here refer to solid lines whereas dark lines refer to dashed or blank lines.

To realize the above idea, two queues corresponding to the light and dark families of faces are needed. Each time a line is drawn, it is enqueued into one of the two queues depending on the drawing attribute. Furthermore, due to the restriction of clockwise ordering, the two vertices in the same edge may be represented in opposite orders in adjoining faces. Consequently, the reverse representation of an edge must also be enqueued whenever the edge is to be enqueued. This is illustrated by the following routine:

```
(define plot
 (lambda ()
  (let ([lq  (**queue)]
        [dq  (**queue)]
        [enq (lambda (q e) ((((q 'enqueue) e) 'enqueue) (reverse e)))])
   (letrec
    ([pf (lambda (msg)
           (case msg
             [poly (lambda (POLY FLAG)
                     (let ((E (&1 POLY)))
                       (if E
                           (if (member E lq)
                               ((pf 'poly) (&rest POLY) FLAG)
                               (if (member E dq)
                                   (block (if FLAG
                                              (line2 E FLAG))
                                              ((pf 'poly) (&rest POLY) FLAG))

                                   (block
                                     (if FLAG (enq lq E) (enq dq E))
                                     (line2 E FLAG)
                                     ((pf 'poly) (&rest POLY) FLAG)))))))]
             [obj  (lambda (OBJ HFL)
                     (let ([POLY (&1 OBJ)]
                                          [OBJ* (&rest OBJ)]
                                          [FLAG (&1 HFL)]]
                                          [HFL* (&rest HFL)])
                                         (if OBJ*
                                             (block ((pf 'poly) POLY FLAG)
                                                    ((pf 'obj)  OBJ* HFL*))
                                             ((pf 'poly) POLY FLAG))))]))])
   pf))))
```

where the invocation of **queue returns an abstract queue object. The routine creates two queues (lq and dq) when called. In fact, the queues can also be stacks, arrays, sets,or any list-type data structures because the only purpose here is to do membership testing. The routine bound to **poly** is the implementation of the general plotting strategy described above. It takes a polygon (i.e. a face) and a light/dark flag, scans edge by edge and draws lines if necessary. Note that the function **line2** is the actual routine which causes line drawing side effects. It takes an edge and the flag and draws a light or dark line depending on the flag. The father routine **obj** takes an object and the list of light/dark flags and recursively invokes **poly**.

## 5.10 Integration

Let us now integrate our system together from the pieces described in the previous sections. In Section 5.1, we defined a 3D tent $\mathcal{T}$ whose composing faces are represented as vertices. We develop the following procedures for perspective and parallel projections of $\mathcal{T}$.

I. Perspective Projections

Assume that the initial view parameters VRP, VPN, VUP, COP, W, and FB are given and that the COP, W, and FB are all specified relative to VRP. We have the following procedure:

1. Get the composite transformation matrix $M_{trl}$ by: (1) translate the COP to the origin, then (2) rotate so that the VPN becomes parallel to the negative $z$-axis and the projection of the VUP onto the view plane is parallel to the positive $y$-axis, and finally (3) combine it with the matrix that transforms the world from RHS to LHS. That is:

   **(define** $M_{trl}$ **(!mult (xlate (*add VRP COP)) (rotate VPN VUP) (l-r)))**

2. Get the projection matrix $M_{per}$ by specifying the absolute value of COP's $z$-coordinate which is:

   **(define** $M_{per}$ **(per (abs (&3 COP))))**

3. Get the view port to physical device coordinates transformation matrix $M_w$ by:

   **(define** $M_w$ **(xcale VRP W))**

4. Get the perspective view volume $VV_{per}$ by specifying W, FB, and VRP. We have:

   **(define** $VV_{per}$ **(((vvol W FB) 'perspective) VRP))**

5. Transform $\mathcal{T}$ into its homogeneous edge-based representation $\mathcal{T}_1$, that is:

   **(define** $\mathcal{T}_1$ **(((xrep) 'obj) (((hom) 'obj)** $\mathcal{T}$ **)))**

6. Apply the transformation $M_{trl}$ to the new object $\mathcal{T}_1$ to return $\mathcal{T}_2$:

(define $\mathcal{T}_2$ (((map $M_{trl}$) 'obj) $\mathcal{T}_1$))

7. Clip $\mathcal{T}_2$ against $VV_{per}$ to yield $\mathcal{T}_3$:

   (define $\mathcal{T}_3$ (((clip $VV_{per}$) 'obj) $\mathcal{T}_2$))

8. Apply hidden face analysis on $\mathcal{T}_3$ and get the Boolean list $H$:

   (define $H$ (((hfe $\mathcal{T}_3$) 'yes) nil))

9. Apply the composite $M_{per} \cdot M_w$ to $\mathcal{T}_3$ and return $\mathcal{T}_4$:

   (define $\mathcal{T}_4$ (((map (!mult2 $M_{per}$ $M_w$)) 'obj) $\mathcal{T}_3$))

10. Dehomogenize $\mathcal{T}_4$ to the 2D object $\mathcal{T}_5$:

    (define $\mathcal{T}_5$ ((deh 2) $\mathcal{T}_4$))

11. Plot $\mathcal{T}_5$ with the help of light/dark face information given by $H$:

    (((plot) 'obj) $\mathcal{T}_5$ $H$)

## II. Parallel Projections

Assume that the basic parameters of the parallel projection, namely, the VRP, VPN, VUP, DOP, W, and FB are given and that the W and FB are specified relative to the VRP. The following summarizes the parallel projection process:

1. Get the composite matrix $M_{trl}$ by translating the VRP (instead of the COP) to the origin and by rotating such that VNP is parallel to the negative *z*-axis and the projection of VUP on the projection plane is parallel to the positive *y*-axis. The last thing to be done in this step is again the transformation from the right-handed system to the left. So:

   (define $M_{trl}$ (!mult (xlate VRP) (rotate VPN VUP) (r-1)))

2. Get the parallel projection matrix $M_{par}$ by specifying the DOP:

   (define $M_{par}$ (par DOP))

3. Get the view port to physical device coordinates transformation matrix $M_w$ as before.

   (define $M_w$ (xcale VRP W))

4. Get the parallel view volume $VV_{par}$ by specifying W, FB, and DOP:

   (define $VV_{par}$ (((vvol W FB) 'parallel) DOP))

5. Transform $\mathcal{T}$ into its homogeneous edge-based representation $\mathcal{T}_1$ as before:

   (define $\mathcal{T}_1$ (((xrep) 'obj) (((hom) 'obj) $\mathcal{T}$ )))

6. Apply the transformation $M_{trl}$ to the new object $\mathcal{T}_1$ to return $\mathcal{T}_2$, that is:

   (define $\mathcal{T}_2$ (((map $M_{trl}$) 'obj) $\mathcal{T}_1$))

7. Clip $\mathcal{T}_2$ against $VV_{par}$ to yield $\mathcal{T}_3$:

   (define $\mathcal{T}_3$ (((clip $VV_{par}$) 'obj) $\mathcal{T}_2$))

8. Apply hidden face analysis on $\mathcal{T}_3$ and get the Boolean list $H$:

   (define $H$ (((hfe $\mathcal{T}_3$) 'yes) nil))

9. Apply the composite $M_{par} \cdot M_w$ to $\mathcal{T}_3$ and return $\mathcal{T}_4$:

```
(define $\mathcal{T}_4$ (((map (!mult2 $M_{par}$ $M_w$)) 'obj) $\mathcal{T}_3$))
```

**10.** Dehomogenize $\mathcal{T}_4$ to the 2D object $\mathcal{T}_5$:

```
(define $\mathcal{T}_5$ (((deh 2) 'obj) $\mathcal{T}_4$))
```

**11.** Plot $\mathcal{T}_5$ with the help of light/dark face information given by $H$:

```
(((plot) 'obj) $\mathcal{T}_5$ $H$)
```

In fact, most steps listed above are common to both projection types and can therefore be combined as one.

## §6 Conclusions

We have demonstrated the close relation between functional style programming and geometric applications in graphics through the linear algebra primitives and a complete package of 3D projections. What's important here is the fact that this programming style reflects our thinking in terms of algebra and geometry directly. For instance, the matrix notation in Scheme is almost identical to what it traditionally looks like and applying transformations is as natural as writing down equations in mathematics.

Functional programming is grounded largely on recursions and applications (function invocations) which may bring down the execution efficiency in the conventional imperative machines. There has been calls for non-von Neumann computer architectures for years among which Lisp machines are of high potential. We foresee that graphics devices of the future will echo the compatibility between the geometric nature in computer graphics and its underlying programming language.

In addition, we feel that the time has come to do other types of numerical calculations in functional languages. As we have seen, in many cases it is easier to implement algorithms in these languages. Also the parallel processing mechanism of Scheme opens many doors to more efficient implementations of numerical algorithms. We are investigating these ideas presently.

As a final note, we mention that the above package is implemented at the Computer Science Department, Indiana University on a VAX 11/780 running under Berkeley Unix using a DEC GIGI terminal as the graphics device.

## §7 References

[Ba78] John Backus, "Can programming be librated from the von Neumann style? a functional style and its algebra of programs," *Comm. ACM*, Vol. 21, No. 8 (August 1978), pp. 613-641.

[Ch41] Alonzo Church, "The Calculi of Lambda Conversion," *Annals of Mathematics Studies*, Number 6, Princeton University Press, 1941.

[Fe83] Carol Fessenden, W. D. Clinger, D. P. Friedman, and C. T. Haynes, "⟦Scheme 311⟧version 4 reference manual," technical report #137, Department of Computer Science, Indiana University, February 1983.

[Fo80] John K. Foderaro, *The FRANZ LISP Manual*, University of California, Berkeley, 1980.

[FoVa] J. D. Foley and A. Van Dam, "*Fundamentals of Interactive Computer Graphics*", Addison-Wesley Publishing Company, Inc., 1982.

[GhJa] Carlo Ghezzi and Mehdi Jazayeri, "*Programming Language Concepts*", John Wiley and Sons, Inc., 1982.

[Ha83] Steven Harrington, *Computer Graphics: A Programming Approach*", McGraw Hill Book Company, Inc., 1983.

[He80] Peter Henderson, *Functional Programming,* Prentice-Hall International, 1980.

[La65] P. J. Landin, "A correspondence between Algol 60 and Church's lambda notation," *Comm. ACM,* Vol 8, No. 2-3 (February and March 1965), pp. 89-101 and 158-165.

[Mc60] John McCarthy, "Recursive functions of symbolic expressions and their computation by machine," *Comm. ACM,* Vol. 3, No. 4 (April 1960), pp. 185-195.

[St78] G. L. Steele and G. J. Sussman, "The revised report on SCHEME, a dialect of LISP," MIT Artificial Intelligence Laboratory Memo #452. Cambridge, MA (January 1978).

[St80] G. L. Steele and G. J. Sussman, "Design of a LISP-based microprocessor," *Comm. ACM,* Vol 23, No. 11 (Nov. 1980), pp. 628-645.