A Coordinate Free Geometry ADT

University of Waterloo, Research Report CS-97-15

Stephen Mann, Nathan Litke, and Tony DeRose*

University of Waterloo, Waterloo, Ontario N2L 3G1, CANADA

* Pixar Animation Studios, 1001 West Cutting Blvd., Richmond, CA 94804, USA

smann@cgl.uwaterloo.ca, njlitke@cgl.uwaterloo,ca, derose@pixar.com

Abstract

An algebra for geometric reasoning is developed that is amenable to software implementation. The features of the algebra are chosen to support geometric programming of the variety found in computer graphics and computer aided geometric design applications. The implementation of the algebra in C++ is described, and several examples illustrating the use of this software are given.

1 Introduction

Traditionally, computer graphics packages are implemented using homogeneous coordinates and a matrix package. Although this practice is widespread and successful, it does have its shortcomings. The basic problem with the traditional coordinate-based approach is due to differences between matrix computations and geometric reasoning. Although graphics programs require reasoning in affine and projective geometries, use of a matrix package places the programmer in a vector space setting; the geometric interpretation of these calculations is left to the imagination and discipline of the programmer (who often is unaware of these subtleties). Often, mysterious problems and hard to trace bugs arise, most of which stem from ambiguities in matrix operations that are misinterpreted by the programmer, with a few problems resulting from matrix operations that have no geometric meaning.

Consider, for example, the code fragment shown in Figure 1. This code fragment has at least three geometric interpretations: as a change of coordinates, as a transformation from the plane onto itself, and as a transformation from one plane onto another (see Figure 2). If T represents a change of basis, then it leaves the geometry of P unchanged, but instead changes the reference coordinate system (Figure 2(a)). If T represents a transformation of the plane onto itself, then P itself moves while the coordinate system remains fixed (Figure 2(b)). Finally, the interpretation as a transformation from one plane onto another involves two coordinate systems, one in the domain and one in the range (Figure 2(c)).

$$
\begin{aligned}
P &\leftarrow [p_1\ p_2\ 1]^t; \\
T &\leftarrow \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}; \\
P' &\leftarrow TP;
\end{aligned}
$$

Figure 1: A typical matrix computation.

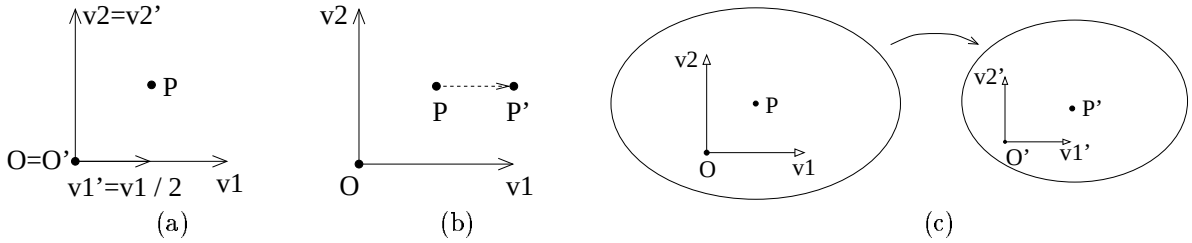

Figure 2: Three interpretations of the code fragment in Figure 1.

A common response to this ambiguity is that it does not matter which interpretation is used, since the coordinates in the matrix representation of P' are the same for all three interpretations. Unfortunately, this is incorrect since it is possible to distinguish between the interpretations. In particular, lengths and angles do not change in the first interpretation, but they can in the second interpretation.

For example, consider the length of the vector $P - P'$. If T represents a change of coordinates, then $|P - P'| = 0$ despite the fact that P and P' have different coordinate representations. If T represents a transformation of the plane onto itself, then $|P - P'| = \sqrt{(p_1 - 2p_1)^2 + (p_2 - p_2)^2} = |p_1|$. And if T represents a transformation from one plane onto another, then the question does not make sense (unless both planes are sub-planes of a common space, in which case we would need more information to compute $|P - P'|$).

There are also times that a valid matrix computation will have *no* valid geometric meaning. For example, suppose that $P = [1\ 0\ 1]^t$. The rules of homogeneous coordinates state that multiplying through by any non-zero factor does not alter the point being represented. Thus, P is also represented by $[-1\ 0\ -1]^t$. Now suppose we add these two matrix representations together:

$$[1\ 0\ 1]^t + [-1\ 0\ -1]^t = [0\ 0\ 0]^t,$$

which is a valid matrix computation. However, $[0\ 0\ 0]^t$ is not a valid homogeneous coordinate. In other words, there is no point with homogeneous coordinates $[0\ 0\ 0]^t$, so despite performing valid matrix computations, the operation is not geometrically valid.

While the previous example was a particularly bad, it is not common in practice. There are other types of errors that are more common, but subtle and difficult to detect. For example, many errors result because distinct coordinate systems are not explicitly represented. The applications programmer is expected to maintain a clear idea in which coordinate system (e.g., world coordinates, viewing coordinates, etc.) each point is represented. If extreme care is not taken, it is possible to perform geometrically meaningless operations such as combining two points represented relative to different coordinate systems.

Fortunately, these problems of ambiguity and validity can be solved. The solution we present consists of two subtasks. The first task is the identification of a *geometric algebra* (i.e., a collection of geometric objects and operations for manipulating them) appropriate for typical graphics applications. The second task is to implement this algebra as an abstract data type. The geometry and the abstract data type are closely related.

In this paper, we develop a geometric abstract data type (ADT) based on Euclidean geometry. Using this software package, each segment of code has a unique geometric meaning. Further, the software package prohibits matrix computations that have no geometric meaning.

Others have considered similar problems with using linear algebra for geometric computations (see, for example, the work of Goldman [4, 5]). Also, Barzel has described a similar software package [1]. The work we describe in this paper is an updated version of an earlier paper by DeRose [2]. While DeRose described a K&R C implementation of the software, in this paper, we describe a C++ implementation. Rather than repeat all of the material in the DeRose paper, we have limited the discussion of the mathematical background and removed most of the implementation details, since those details appear in the DeRose paper

and are unaffected by the choice of language. The background that we have included is what we consider the minimum needed to understand our software.

In the next section, we review the geometric concepts of affine and Euclidean geometry upon which our geometric ADT is based. Then, in Section 3 we present a C++ implementation of the geometric algebra, and show how the package can be used to disambiguate the example shown in Figure 1. Further examples of the software package are given in Section 4. As this paper is just an introduction to our software package, we do not fully describe its functionality. Thus, in Section 5, we discuss some of the features not illustrated in the previous sections, and in Appendix B, we discuss some of the implementation details.

2 A Geometric Algebra

In selecting a geometric algebra, our primary criteria is that the theory provide a model of the geometric spaces typically encountered in computer graphics. Three obvious candidates are vector spaces, Euclidean spaces, and projective spaces. We chose a Euclidean space abstraction because it best models the mathematics performed in computer graphics (see [2] for details on this choice). Our package also provides for projective transformations (since 3D graphics requires one for the perspective transformation). However, general projective spaces are not supported.

2.1 Affine Geometry

Although the geometric ADT will be based on Euclidean geometry, many of the geometric objects and operations used in computer graphics are founded in affine geometry. We have therefore chosen to develop the more general theory here, and then specialize to Euclidean geometry in Section 2.4.

There are many different approaches to affine geometry. The method we shall adopt is similar to that used by Dodson and Poston [3]. The development builds on vectors spaces, which we assume the reader is familiar with. We give only an abbreviated development here; the interested reader is referred to Dodson and Poston's book. For an introduction to vector spaces, see any standard text in linear algebra such as O'Nan's book [8].

The most basic object in our geometric algebra will be *affine spaces*, which in turn consist of *points* and *free vectors*. Intuitively, the only thing distinguishing one point from another is its position. Free vectors on the other hand have the attributes of magnitude and direction, but no fixed position; the modifier "free" refers to the ability of vectors to move about in the space. Free vectors will henceforth be referred to simply as vectors. We will conform to the convention that points will be written in upper case such as P and Q. Vectors will be written in lower case and ornamented with a diacritical arrow such as $\vec{v}$ and $\vec{w}$.

More formally, an affine space $\mathcal{A}$ is a pair $(\mathcal{P}, \mathcal{V})$ where $\mathcal{P}$ is the set of points and $\mathcal{V}$ is a set of vectors. We shall use $\mathcal{A}.\mathcal{P}$ and $\mathcal{A}.\mathcal{V}$ to denote points and vectors, respectively, of an affine space $\mathcal{A}$. The vectors of an affine space are assumed to form a vector space. If the vectors space is dimension n, then the affine space is called an *affine n-space*.

The points and vectors of an affine space $\mathcal{A}$ are related through the following axioms:

- *Subtraction:* There exists an operation of subtraction that satisfies:
  - For every pair of points P, Q, there is a unique vector $\vec{v}$ such that $\vec{v} = P - Q$.
  - For every point Q and every vector $\vec{v}$, there is a unique point P such that $P - Q = \vec{v}$.
- *The Head-to-Tail Axiom:* Every triple of points P, Q, and R satisfies

$$(P - Q) + (Q - R) = P - R.$$

Additionally, it is convenient to define the operation of addition between points and vectors: Define $Q + \vec{v}$ to be the unique point P such that $P - Q = \vec{v}$.

Thus far, the operations in the algebra can be summarized as follows:

$$\begin{array}{rcl} \text{vector+vector} & \mapsto & \text{vector} \\ \text{scalar*vector} & \mapsto & \text{vector} \\ \text{point}-\text{point} & \mapsto & \text{vector} \\ \text{point+vector} & \mapsto & \text{point} \end{array}$$

Notice the asymmetry in the way points and vectors are handled. In particular, notice that it is possible to add vectors, but that addition of points is not defined. Similarly, the process of multiplying a point by a scalar is undefined. The asymmetry should not be too surprising since points and vectors are being used in very different ways. In some respects the points are the primary objects of the geometry, whereas the role of the vectors is to allow movement from point to point by employing the operation of addition between points and vectors.

Although we may not be able to add two points, there are other convenient operations that we can define. In particular, for points $Q_1, \ldots, Q_k$ and scalars $\alpha_1, \ldots, \alpha_k$ that sum to one, then we define

$$\alpha_1 Q_1 + \alpha_2 Q_2 + \alpha_3 Q_3 + \ldots + \alpha_k Q_k$$

to be the point

$$Q_1 + \alpha_2(Q_2 - Q_1) + \alpha_3(Q_3 - Q_1) + \ldots \alpha_k(Q_k - Q_1).$$

An expression such as this is called an *affine combination.*

2.2 Frames

To perform numerical computations and to facilitate the conversion between coordinates and geometric entities, we must understand how affine spaces are coordinatized. In this section, we introduce the idea of coordinate frames as a method of adding coordinates to an affine space.

Let $\mathcal{A} = (\mathcal{P}, \mathcal{V})$ be an affine n-space, let O be any point, and let $\vec{v}_1, \ldots, \vec{v}_n$ be any basis for $\mathcal{A}.\mathcal{V}$. We call the collection $(\vec{v}_1, \ldots, \vec{v}_n, O)$ a *frame* for $\mathcal{A}$. Frames play the same role in affine geometry that bases play in vector spaces, as indicated in the next claim:

Claim: If $\mathcal{F} = (\vec{v}_1, \ldots, \vec{v}_n, O)$ is a frame for some affine n-space, then every vector $\vec{u}$ can be uniquely written as

$$\vec{u} = u_1\vec{v}_1 + u_2\vec{v}_2 + \ldots + u_n\vec{v}_n \tag{1}$$

and every point P can be written uniquely as

$$P = p_1\vec{v}_1 + p_2\vec{v}_2 + \ldots + p_n\vec{v}_n + O. \tag{2}$$

The sets of scalars $(u_1, \ldots, u_n)$ and $(p_1, \ldots, p_n)$ are called the *affine coordinates* of $\vec{u}$ and P relative to $\mathcal{F}$.

Proof: The claim for $\vec{u}$ follows from $(\vec{v}_1, \ldots, \vec{v}_n)$ being a vector space for $\mathcal{A}.\mathcal{V}$. The claim for P is proven in [2].

2.3 Affine Transformations

The next geometric object to be added to our algebra is the affine transformation. Affine transformations are mappings between affine spaces that preserve the algebraic structure of the spaces. That is, affine transformations map points to points, vectors to vectors, frames to frames, and so forth.

To begin, let $\mathcal{A}$ and $\mathcal{B}$ be two affine spaces (it is sometimes the case that $\mathcal{A}$ and $\mathcal{B}$ are the same space). A map $F : \mathcal{A}.\mathcal{P} \mapsto \mathcal{B}.\mathcal{P}$ is said to be an *affine transformation* if the condition

$$F(\alpha_1 Q_1 + \alpha_2 Q_2 + \ldots + \alpha_k Q_k) = \alpha_1 F(Q_1) + \alpha_2 F(Q_2) + \ldots + \alpha_k F(Q_k)$$

holds for all points $Q_1, \ldots, Q_k$ and for all sets of α's that sum to one.

We can extend the domains of affine transformations to include the vectors as well. Let $F : \mathcal{A}.\mathcal{P} \mapsto \mathcal{B}.\mathcal{P}$ be an affine map, let $\vec{v}$ be any vector in $\mathcal{A}.\mathcal{V}$, and let P and Q be any two points in $\mathcal{A}.\mathcal{P}$ such that $\vec{v} = P - Q$. We define $F(\vec{v})$ to be the vector in $\mathcal{B}.\mathcal{V}$ given by $F(P) - F(Q)$. In equation form,

$$F(\vec{v}) \equiv F(P - Q) = F(P) - F(Q).$$

It is easily shown that $F(\vec{v})$ is well defined (i.e., any choice of P and Q such that $\vec{v} = P - Q$ yields the same vectors $F(\vec{v})$).

Using this definition of the action of affine maps on vectors, it is also easy to show that F is a linear transformation on the set of vectors. That is, F satisfies

$$F(u_1\vec{v}_1 + u_2\vec{v}_2 + \ldots + u_k\vec{v}_k) = u_1 F(\vec{v}_1) + u_2 F(\vec{v}_2) + \ldots + u_k F(\vec{v}_k)$$

Notice the similarity between this definition and the definition for affine transformations.

Note that an affine transformation is completely determined once we know how it maps all the elements of a frame. That is, given a frame $\mathcal{F} = (\vec{v}_1, \ldots, \vec{v}_k, O)$, if we know $F(\vec{v}_1)$, $F(\vec{v}_2)$, $\ldots$ $F(\vec{v}_k)$ and $F(O)$, then we can compute the transformation of $P = u_1\vec{v}_1 + \ldots + \vec{v}_k + O$ as

$$F(P) = u_1 F(\vec{v}_1) + \ldots + F(\vec{v}_k) + F(O).$$

2.4 Euclidean Geometry

In affine geometry, metric concepts such as absolute length, distance, and angles are not defined. However, in computer graphics, it is often necessary to represent metric information, for without this information it is not possible to define right angles or to distinguish spheres from ellipsoids.

When metric information is added to an affine space, the result is the familiar concept of a Euclidean space. In other words, a Euclidean space is a special case of an affine space in which it is possible to measure absolute distances, lengths, and angles. Consequently, all results that were obtained for affine spaces and affine maps also hold in a Euclidean space.

We will use a dot product to introduce metric information. A dot product is an operator that assigns a real number to every pair of vectors, $\vec{u}$ and $\vec{v}$, denoted by $\vec{u} \cdot \vec{v}$. The dot product is a bi-variate operator on vectors, which we use to define length, distance, and angles in the standard fashion:

- The *length* of a vector:
$$|\vec{v}| \equiv \sqrt{\vec{v} \cdot \vec{v}}$$
- The *distance* between two points:
$$\text{Dist}(P, Q) = |P - Q|$$
- The angle between two vectors:
$$\text{Angle}(\vec{v}, \vec{w}) = \cos^{-1}\left(\frac{\vec{v} \cdot \vec{w}}{|\vec{v}|\ |\vec{w}|}\right)$$

Once we have a notion of distance, we can define unit vectors by associating with every vector $\vec{v}$ a unique vector $\hat{v}$ having unit length that points in the same direction as $\vec{v}$.

The definition of angles allows us to define the notion of *perpendicularity* or *orthogonality*: Two vectors $\vec{v}$ and $\vec{w}$ are said to be perpendicular if $\vec{v} \cdot \vec{w} = 0$. We can also define the vectors to be parallel if $\hat{v} \cdot \hat{w} = 1$.

The notions of unit vectors and orthogonality allow us to identify an important kind of coordinate frame for Euclidean spaces. A coordinate frame $(\vec{e}_1, \vec{e}_2, \ldots, \vec{e}_n, O)$ is said to be a *Cartesian frame* if the basis vectors are *orthonormal*; that is, if the basis vectors satisfy

$$\vec{e}_i \cdot \vec{e}_j = \begin{cases} 1 & \text{if } i = j; \\ 0 & \text{if } i \neq j. \end{cases}$$

In the special case of Euclidean 3-spaces, it is convenient to define another operation on vectors, namely the *cross product*. Given a pair of vectors $\vec{v}$ and $\vec{w}$ from a Euclidean 3-space, we define $\times$ by the equation

$$\vec{v} \times \vec{w} = |\vec{v}|\ |\vec{w}| \sin\theta \hat{n},$$

where θ is the angle between the vectors and $\hat{n}$ is the unique unit vector that is perpendicular to $\vec{v}$ and $\vec{w}$ such that $\vec{v}$, $\vec{w}$, and $\hat{n}$ satisfy the "right hand rule."

3 A Geometric Abstract Data Type

An abstract data type is a collection of data types together with a collection of operations or procedures that manipulate instances of the types. The abstract data type presented here for geometric programming is based on the geometric algebra described in previous sections. In particular, the objects in the algebra are the types of the abstract data type, and the operations in the algebra are the procedures of the abstract data type.

In this section, we describe a C++ implementation of the geometric algebra. C++ allowed us to overload the arithmetic operations, resulting in a software package that is more natural to use (e.g., to compute $P-Q$, you would make the call `P-Q`). We will not give a complete specification of our library; for complete details, see the manual pages, which are available from one of the ftp or web sites listed in Appendix A.

The supported data types include: `Scalar`, `ESpace`, `Point`, `Vector`, `Frame`, and `AffineMap`. Except for the `ESpace` datatype, the connection between these types and the geometric objects in the algebra should be apparent. The `ESpace` datatype corresponds to a Euclidean space (as opposed to the more general affine space) of the algebra. Using operator overloading, the operations on points and vectors listed at the end of Section 2.1 are all supported using '+', '*', and '−'. Vector dot and cross products are implemented with procedures `Dot` and `Cross`, and the function `Normalize` takes a vector as an argument and returns the unit vector parallel to this vector. The remainder of this section discusses the creation of points and vectors and the creation and manipulation of spaces, frames, and transformations, using short code sequences to illustrate these ideas.

The ADT provides for the creation of Euclidean spaces. The following code segment shows how to declare and create a Euclidean space:

```
ESpace s("3 space",3);
```

The first argument is the name of the space (intended for debugging purposes). The second argument is the dimension of the space, which must be a non-negative integer. On creation, the Euclidean space comes pre-equipped with a Cartesian frame, accessed via the command `s.StdFrame()` (in the case of a three dimensional space, the frame is defined to be right-handed). This frame is used to boot-strap the creation of points, vectors, and new frames:

```
ESpace World("World", 3);
Frame WorldFrame = World.StdFrame();
Point Eye;
Vector View, ViewX, ViewY, ViewZ;
Frame Camera;

Eye = Point(WorldFrame, 1, 2.5, 2);
View = Vector(WorldFrame, -1, -2.5, -2);
ViewZ = Normalize(View);
ViewX = Normalize(Cross(ViewZ, WorldFrame.BasisVector(2)));
ViewY = Cross(ViewX, ViewZ);
Camera = Frame("Camera", Eye, ViewX, ViewY, ViewZ);
```

Notice that the point `Eye` and the vector `View` are defined by giving their coordinates relative to the standard frame in the `World` space. Even though they were created by using coordinates, once created they take on a meaning that is independent of which coordinate systems they are represented in. Moreover, since all geometric entities are tagged with the space in which they reside, the system can do a substantial amount of type checking. For instance, before the frame creation command `Create` returns the new frame, it checks to see that the origin and the vectors reside in a common space and that the vectors form a basis for this space.

Note also that we could have created `Eye` using the constructor:

```
Point Eye(WorldFrame, 1, 2.5, 2);
```

To demonstrate that points and vectors have meaning that transcend coordinate systems, we give the following code sequence, in which two points are created with respect to the two coordinate frames created in the previous code sequence. We then compute the distance between these two points:

```
Q1 = Point(WorldFrame, 10, 1, 0);
Q2 = Point(Camera, 0, 0, 1);
d = Dist(Q1, Q2);
```

which gives the geometrically correct distance of 9.43373.

The use of `Point` above shows that points can be created by giving their coordinates relative to arbitrary frames. Conversely, coordinates of points and vectors can be extracted relative to arbitrary frames using the `Coords` command. For instance, the coordinates of `Q1` can be extracted relative to the `Camera` frame as follows:

```
Q1.Coords(Camera, &Qx, &Qy, &Qz);
```

Since a programmer will commonly need the coordinates of points and vectors relative to the standard Cartesian coordinate frame, we have overloaded the square brackets to provide direct access to these coordinates as a programming convenience. Thus, to test if a three dimensional point lies in the x-y plane, rather than having to use something like

```
Q1.Coords(WorldFrame, &Qx, &Qy, &Qz);
if ( Qz == 0 ) {
```

you may write the following

```
if ( Q1[2] == 0 ) {
```

Note that this is a notational convenience only. Further, the square bracket notation always extracts coordinates relative to the standard Cartesian frame of the space. Thus, if you wanted to test if a point lies in the x-y plane relative to another coordinate frame, you would have to extract the coordinates using the `Coords` member function. Further, if `F` is not the standard frame, then the following code fragment

```
Q1 = Point(F, 1,2,3);
cout << Q1[1] << endl;
```

will not in general print '2'. Finally, to prevent sloppy use of coordinates, you can not assign values to the coordinates of point and vectors using the square bracket notation. E.g., the following code fragment will produce a compile time error:

```
Q1[0] = 2.;     // Illegal statement
```

Affine maps also have coordinate-independent meanings. For instance, suppose we want to create a two dimensional transformation that represents rotation about an arbitrary point P through an angle *theta*. This is easily accomplished using the constructor for `AffineMaps`. First, we create a new frame called `RotateFrame` having its origin at P, and `x` and `y` vectors inherited from the standard frame in the containing space. To use the `AffineMap` constructor, we must determine the image of the elements of the `RotateFrame` under the rotation. The following procedure accomplishes this task.

```
AffineMap Rotate2D(Point P, Scalar theta)
{
    Frame RotateFrame;
    Vector stdx, stdy, xprime, yprime;
    stdx = P.Space().StdFrame().BasisVector(0);
    stdy = P.Space().StdFrame().BasisVector(1);
    RotateFrame = Frame("Rotate", P, stdx, stdy);
    xprime = Vector(RotateFrame, cos(theta), sin(theta));
    yprime = Vector(RotateFrame, -sin(theta), cos(theta));
    return AffineMap(RotateFrame, P, xprime, yprime);
}
```

The first argument of the constructor for affine maps is a frame in the domain space. The remaining arguments are the images of the elements of this domain frame. So in our example, `RotateFrame.Origin()` will map to `P`, `RotateFrame.BasisVector(0)` will map to `xprime`, and `RotateFrame.BasisVector(1)` will map to `yprime`. `AffineMap` requires the number of arguments after the domain frame (`RotateFrame` in the above example) to match the number of elements in this frame. Further, these additional arguments must all be members of the same space (which may be different than the space for the domain frame).

To apply an affine map to a point or vector, we have overloaded the parenthesis operator, so that you make what appears to be another function call. For example, the code

```
T = Rotate2D(P, 30);
Q = T(R);
```

will rotate the point `R` by 30 degrees around the point `P`, putting the result in Q.

To compose two affine maps, we just call the procedure `Compose(A,B)`, which returns the composition of affine maps `A` and `B`.

3.1 Ambiguity Revisited

In the introduction, it was claimed that the ADT solves the ambiguity problem in that a given code fragment can have one and only one geometric interpretation. As a demonstration of how this is accomplished, we refer again to the code fragment of Figure 1, the geometric interpretations of which are shown in Figure 2.

Using the ADT, each geometric interpretation is unambiguously reflected in the code. For instance, if the programmer intended a change of coordinates as indicated by Figure 2(a), the appropriate code fragment would be something like:

```
Frame f1, f2;
Point P;
Scalar px, py;
    ...
    f2 = Frame("f2", f1.Origin(), 0.5*f1.BasisVector(0), f1.BasisVector(1));
    P = Point(f1, p1, p2);
    P.Coords(f2, &px, &py);
```

where `f1` and `f2` are two frames having the geometric relationship indicated in the figure. If the programmer was instead intending to effect a transformation on the space as indicated by Figure 2(b), the appropriate code would be something like:

```
AffineMap T;
ESpace S;
Frame F;
Point P;
    ...
    T = AffineMap(F, F.Origin(), 2*F.BasisVector(0), F.BasisVector(1));
    ...
    P = T(P);
```

Finally, if a transformation between separate spaces is to be applied as indicated by Figure 2(c), the code would be something like

```
AffineMap T;
ESpace S1, S2;
Point P, Oprime;
Vector xprime, yprime;
    ...
    Compute Oprime, xprime, and yprime in S2
    ...
    T = AffineMap(S1.StdFrame(), Oprime, xprime, yprime);
    ...
    P = T(P);
```

To reiterate, each of the code fragments above has an unambiguous geometric interpretation that is undeniably apparent from the code. The fact that identical matrix computations are being performed at a lower level is invisible (and irrelevant).

4 Additional Examples

As additional examples of using our software package, we show how to evaluate tensor product Bézier surfaces with our package, and show how to use the projective geometry functions in our library.

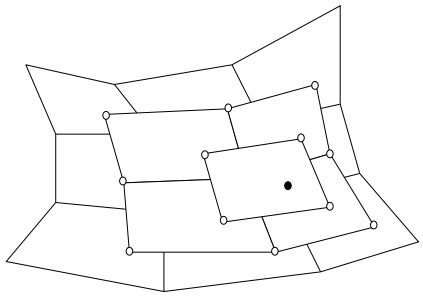

Figure 3: Repeated bi-linear interpolation.

4.1 Tensor Product Surfaces

The following code evaluates a bi-cubic tensor product Bézier patch using the variation of de Casteljau's algorithm illustrated in Figure 3. (See [7] for details on this and other similar tensor product evaluation algorithms.) This variation of de Casteljau's algorithm repeatedly computes bi-linear combinations such as

$$\begin{aligned} L_0 &= (1-u)P + uQ; \\ L_1 &= (1-u)R + uS; \\ T &= (1-v)L_0 + vL_1; \end{aligned}$$

Rather than perform three affine combinations, in our implementation below we expand these equations into

$$T = (1-u)(1-v)P + u(1-v)Q + (1-u)vR + uvS.$$

In the code below, the function `AffineP` takes a set of points P_i and weights w_i (that sum to one) and computes $\sum w_i P_i$.

```
Point TP::Eval(double u, double v) const
{
    int i,j;
    TP tp=*this;

    for (int d=3; d>=1; d--) {
        for (i=0; i<d; i++) {
            for (j=0; j<d; j++) {
                tp.CP[i][j] = AffineP(4, (1-u)*(1-v),tp.CP[i][j], u*(1-v),tp.CP[i][j+1],
                                         (1-u)*v,tp.CP[i+1][j], u*v,tp.CP[i+1][j+1]);
            }
        }
    }
    return tp.CP[0][0];
}
```

4.2 Projective Geometry

While our package is based on Euclidean geometry, most computer graphics software requires one projective transformation, the viewing transformation. Typically, only one projective transformation is used, and it is

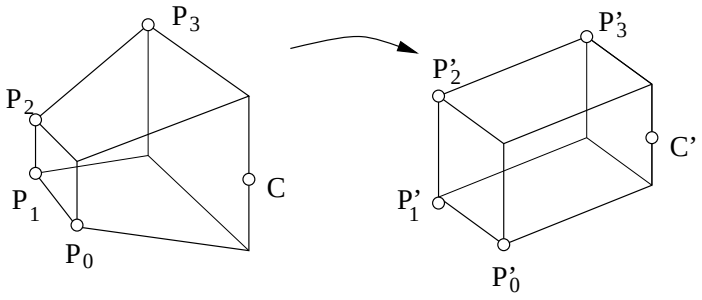

Figure 4: A perspective transformation.

used at the end of the transformation sequence; usually, the only additional processing that we perform is scan conversion. Our software package includes projective transformations to allow for this projection, but note that most of the geometric reasoning about a program is affine or Euclidean rather than projective.

As an example of constructing a projective map, we give the code to implement the equivalent of the OpenGL `glFrustum` routine. To specify the viewing frustum for the projective transformation, we must give five points on the frustum and their five images in the projected space as shown in Figure 4.

The following code will create this viewing transformation:

```
ProjectiveMap Frustum(  const Frame world, const Frame view,
                        double left, double right,
                        double bottom, double top,
                        double near, double far)
{
    // Define a 3-simplex in the world frame.
    Point P0(world, right, bottom, near);
    Point P1(world, left, bottom, near);
    Point P2(world, left, top, near);
    Point P3(world, (left / near) * far, (top / near) * far, far);
    Point C(world, (right / near) * far, (top + bottom) / 2.0, far);

    // Define a 3-simplex in the view frame.
    Point P0p(view, right, bottom, near);
    Point P1p(view, left, bottom, near);
    Point P2p(view, left, top, near);
    Point P3p(view, left, top, far);
    Point Cp(view, right, (top + bottom) / 2.0, far);

    // Construct the projective map.
    ProjectiveMap viewTransform(P0, P1, P2, P3, C,
                                P0p, P1p, P2p, P3p, Cp);
    return (viewTransform);
}
```

This viewing transformation is applied to points in the same manner as an affine transformation.

5 Additional Functionality

The previous sections describe most of our software package. However, in the interests of brevity, we have not listed all the operations provided for the data types we discussed (e.g., there are several ways to create affine transformations), and we have omitted one interesting and important piece, normals.

Normals are usually treated as free vectors in most graphics package (normals are commonly used to denote the tangent plane of a surface). While normals behave like vectors most of the time, there are times when they behave differently. For example, since affine transformations do not preserve angles, a vector representing a normal might not remain perpendicular to the tangent plane after an affine transformation. To retain this perpendicularity, we implemented normals as a separate data type. See the DeRose technical report [2] for a further discussion of normals.

6 Conclusions

We have shown that traditional coordinate-based approaches to geometric programming lead to geometrically ambiguous programs that are potentially geometrically invalid. To combat these deficiencies, we defined a geometric algebra and an associated coordinate-free abstract data type. Programs written using the abstract data type are geometrically unambiguous; moreover, they are guaranteed to be geometrically valid.

This geometry software has been successfully used to teach an introductory graphics course at the University of Washington. The students used this software to implement a Z-buffer renderer, where coordinates were used for input and for scan conversion. At all steps inbetween, the coordinates (and matrices) were hidden behind the geometry software abstraction.

The software has also been used in our Computer Aided Geometric Design (CAGD) research as detailed in the paper of Lounsbery et al. [6]. In this context, the software allowed us to concentrate on CAGD algorithms without worrying about the underlying matrix computations.

We believe that the use of the abstract data type also results in programs that are easier to develop, debug, and maintain. The development and debugging phases are improved due to the high degree of type-checking. Maintainability is improved because the abstract data type forces the applications programmer to clearly defined coordinate systems and geometric spaces. This extra level of specification allows programmers other than the original author to quickly understand the geometric behavior of the code.

Unfortunately, this geometric abstraction does not come without expense. In this instance, the expense is three-fold:

1. The coordinate-free approach to affine/Euclidean geometry must be learned by the applications programmer.

2. The ADT we describe in this paper is more difficult to implement than standard matrix packages. In fact, our package is most conveniently built on top of a matrix package.

3. A run-time performance penalty is incurred. Although most of the type checking can be done at compile time using a strongly typed language like C++, the package must still perform some run-time type checking. A second source of performance degradation is a result of coordinate-freedom. In the coordinate-free approach, the programmer does not have direct access to the coordinates of the geometric objects with respect to any frame except the standard from. For example, in a coordinate based package, after transforming a vector to the viewing coordinate system, to test if this vector is perpendicular to the viewing direction, a direct coordinate test can be performed by testing the z coordinate of the vector to see if it is zero. In the coordinate-free approach, this test requires a dot product (requiring three multiplications, two additions, and one test for zero).

The important trade-off then is between ease of development, debugging, and maintenance on one hand, and raw performance on the other hand. This is, of course, one of the classic trade-offs in software development.

It occurs, for instance, any time a high-level language is used in place of assembly language. Our experience has been that the ease of development facilitated by the geometry package far outweighs the performance penalty.

7 Acknowledgments

There are numerous people involved in discussions about this software who deserve our gratitude. However, we would especially like to thank Greg Nielson, Ronald Goldman, and Richard Bartels for their theoretical insights, and Kevin Sullivan and Michael Lounsbery for their early C++ implementations of the software.

Appendix A: Getting the Software

All versions of the software are available via anonymous ftp at

`ftp.cgl.uwaterloo.ca`

in

`pub/users/smann/Geometry/`

and at the web site

`http://cgl.uwaterloo.ca/software/Geometry/`

Appendix B: Implementation Details

The geometry package provides a Euclidean geometry abstraction to the application programmer. However, underneath these Euclidean operations lies a matrix package. The main function of the geometry package is to restrict the matrix operations the application programmer can perform, and do type checking to ensure that all objects reside in the proper spaces.

The initial prototype of the library was implemented in K&R C, which was later ported to ANSI C. These C versions are restricted to spaces of 2 and 3 dimensions. We have also implemented C++ and Java versions of the library (the C++ version is described in this paper). These versions remove the dimension restrictions, and were implemented with efficiency issues in mind.

For the C++ version, we wrote a matrix layer that sits between the geometry and the matrix operations, allowing the geometry package to use any matrix package simply by changing the matrix layer. The C++ geometry package comes with a simple matrix package we wrote that just supports the matrix operations needed for the geometry package. In addition, we have written (and provide with the distribution) an interface to the `lapack++` matrix package.

Further discussion of the implementation of the geometry ADT described in this technical report is provided for the benefit of those who wish to modify the software. The following sections describe some of the issues that were raised in the development of the library and the decisions that were made.

The Matrix Sub-layer

The geometry library relies on a separate matrix package for its matrix operations. This design allows the user to choose a matrix library that is optimized for a particular platform. To integrate a matrix package with the geometry library, the user must provide matrix functions that conform to an interface of our design. This integration involves creating a short list of functions which in turn call the matrix package for standard matrix operations. We have included one such interface for the LAPACK++ matrix library found at

`http://www.netlib.org/~lapack.`

We have also implemented a simple matrix library that provides the basic functionality required by the geometry library. A detailed description of the matrix interface is included with the geometry library.

Design Issues

Affine transformations are a subset of projective transformations; that is, since affine transformations preserve cross-ratios they are also projective. This relationship reflected in the geometry library class hierarchy by deriving the `AffineMap` class from `ProjectiveMap`. As a result, an `AffineMap` may be substituted whenever a `ProjectiveMap` is required in a function call. `AffineMap` extends `ProjectiveMap` to include transformations of vectors and normals.

Certain geometric operations may be written more concisely with function calls that use variable argument lists. For example, calculating the affine combination of a set of points {`P0, P1, P2, P3`} with coefficients {`a0, a1, a2, a3`} can be performed with

```
Point Q = AffineP(4, a0, P0, a1, P1, a2, P2, a3, P3);
```

Since references to class objects cannot be passed in a variable argument list, we are forced to pass the parameters either by value or with a pointer. We chose to pass objects by value for the sake of clarity and to allow the output of a geometric operation to be passed directly in the function call. This decision incurs a performance penalty in that more data must be placed on the call stack for each object in the list. However, the complexity of the operation performed in the function call is invariably much greater than the cost of performing the call itself, so the decrease in performance is minimal. Where this performance penalty is still a concern, one may use the alternative version of the operation in which the argument list is replaced by an array of objects.

For the Point, Vector, and Normal creation routines, we had several goals, including the use of either integers or Scalars as arguments, and the use of variable arguments to allow for the creation of arbitrary dimensional points. Thus, we wanted to be able to write

```
Point Q(Frame, 1);
Point R(Frame, 1, 2);
Point S(Frame, 1, 2, 3);
Point T(Frame, 1, 2, 3, 4);
```

Unfortunately, the C++ ... operator places operands on the stack by value. Since the creation routines interpret the coordinate arguments as Scalars, an attempt to pass an integer will produce unexpected results. To allow for integer arguments in the common cases (two and three dimensions), we have overloaded the creation routine with versions that explicitly have Scalar arguments (which results in integer arguments being automatically type converted to Scalar). However, if you wish to create higher dimensional points, you should pass only Scalar arguments, e.g.,

```
Point T(Frame, 1., 2., 3., 4.);
```

We recommend that if you expect to work with spaces of dimension greater than three, then you should write creations routines for the dimensions you wish to work with to avoid accidentally passing an integer argument where you needed to use a floating point argument.

Compatibility Issues

We have attempted to make the geometry library compatible with as many software platforms as possible. However, not all C++ compilers are created equal and these difference are reflected in our software.

The geometry library uses exceptions to signal critical run-time errors, such as an attempt to perform an illegal geometric operation. Since not all C++ compilers support exception handling, the software defines

a macro that is executed when an exception is raised. If the preprocessor identifier `EXCEPTIONS` is defined, true C++ exception handling is used. Otherwise, a detailed error message is written to the error output stream and a segmentation fault is generated.

Another difficulty arises due to the non-standard treatment of passing class objects in a variable argument list. As discussed above, we pass geometric objects by value in a variable argument list in certain operations. Since some C++ compilers do not allow for passing variable argument lists by value, the operations are only compiled if the preprocessor identifier `OBJECT_VALIST` is defined. For those C++ compilers that do not support variable argument lists of objects, you can only use the overloaded versions of these operations that we provided for two- and three-dimensional affine spaces. For higher dimensions, or more arguments, we have included simple instructions with the software for creating extra functions as required (although you can also use the array versions of the creation operators without extending the library). Note that only a finite number of overloaded functions is required since the number of parameters passed is predetermined in the function call. If a particular version is not present in the geometry library, it will be identified by the compiler, and the user of our package will have to extend the list of versions of overloaded operations.

Numerical Accuracy

The geometry library attempts to control numerical accuracy by using numerically stable methods in its real-valued calculations. Additionally, the geometry library and matrix sub-layer use an abstracted numerical model. This model is defined by a scalar value type used to represent real values and an epsilon value that defines exactness in the scalars. The definition of these terms is controlled centrally in the header file for the matrix sub-layer with the defined value `EPSILON`. This allows the user to choose a scalar type to suit particular requirements for numeric stability. By default, the geometry library uses double-precision numbers and it considers two values to be identical if they differ by less than 10^{-15}.

References

[1] Ronen Barzel. *Physically-Based Modeling for Computer Graphics.* Academic Press, 1992.

[2] Tony D. DeRose. Coordinate-free geometic programming. Technical Report 89-09-16, University of Washington, Department of Computer Science, Seattle, WA 98195 USA, September 1989.

[3] C.T.J. Dodson and T. Poston. *Tensor Geometry: The Geometric Viewpoint and Its Uses.* Graduate Texts in Mathematics. Springer-Verlag, 2nd edition, 1991.

[4] Ronald N. Goldman. Illicit expressions in vector algebra. *ACM TOG*, 4(3):223–243, July 1985.

[5] Ronald N. Goldman. Vector geometry: A coordinate-free approach. In *SIGGRAPH '87 Tutorial Course Notes, Course No. 19.* 1987.

[6] Michael Lounsbery, Charles Loop, Stephen Mann, David Meyers, Tony DeRose James Painter, and Kenneth Sloan. A testbed for the comparison of parametric surface methods. In Leonard A. Ferrari and Rui J. P. de Figueiredo, editors, *Curves and Surfaces in Computer Vision and Graphics*, volume 1251, pages 94–105. SPIE, SPIE, February 1990.

[7] Stephen Mann and Tony DeRose. Computing values and derivatives of bezier and b-spline tensor products. *Computer Aided Geometric Design*, 12(1), February 1995.

[8] Michael O'Nan. *Linear Algebra.* Harcourt Brace Jovanovich, second edition, 1976.