On Transformations For Animation

Sharat Chandran
Indian Institute of Technology
Bombay, India
sharat@cse.iitb.ernet.in

Yatin Kulkarni
MarchFirst
Chicago, USA
yatin.kulkarni@marchFirst.com

Partho Nath
Banaras Hindu University
Varanasi, India
partho_nath@hotmail.com

Abstract

This paper focuses on the transformation steps involved in implementing an arbitrary *generic* two-dimensional animation. For instance, we are interested in depicting the motion of the earth rotating, and revolving around the sun, which could itself be translating.

Unlike key-frame animation, the transformations discussed in this work are described algorithmically using certain primitives provided by an *animation algebra*. Such a process is most useful in building authoring systems and in virtual reality applications where the programmer is 'unaware' of the final use of his work.

The usual method in producing an animation sequence is to divide the effect into parts and render every part quickly, taking care to erase previous parts. In key-frame animation, a trial and error procedure is often adopted to generate intermediate positions when a linearity assumption cannot be made. An alternate method to generate a part is the use of a *forward matrix* transformation of the initial model, where the *programmer* needs to be aware of the parameters for producing each part. In this paper, we provide two methods to implement the algebra where the process of obtaining intermediate positions is automated, and analyse the complexity.

Keywords: Matrices, Transformations, Algorithmic animation, Algebra.

1 Introduction

In this paper, we are concerned with animation in the context of the primitive operations rotation, scaling, and translation. We consider a few scenario to motivate our work.

1.1 Rotation with Translation

Consider the motion of a wheel on a flat surface. As the wheel rotates, it also moves forward. A point A on the rim (representing a nail on a tire) thus undergoes simultaneous rotation and translation and reaches A'. It is possible to derive a closed from expression for the curve (called as trachos) representing this motion. In general, as we see below, a closed form expression is impossible.

1.2 Multiple rotations

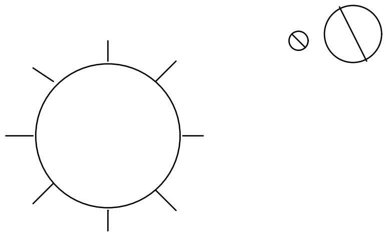
Consider, in a small way, how the solar system is at any given instant, and the question: "Is the sun in the line of sight from New Delhi?" Note that New Delhi might be "away" from the sun (night time), or the moon might be "in-between" (an eclipse).

To answer this question, we might model, in order, the earth's rotation, its revolution around the sun, the motion of other planetary bodies, and then a snapshot of the system at the given time. A computer generated picture at this point in time would then permit an easy answer to the posed question. To depict the earth, for instance, we model the earth as a sphere, and postulate an initial position for its center. The principal task is to calculate future orientation and positions of the earth. For simplicity, a "flat" approach is taken and we model the problem simply in two-dimensions: the earth then becomes a circle, with different cities appearing within the circle.as shown in figure 1(a).

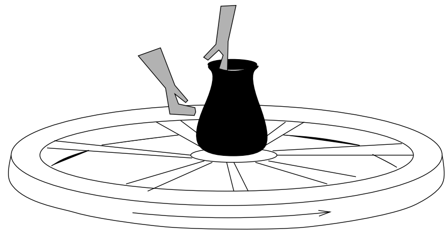
If the earth were to simply rotate around its center, it will always appear as a circle, but the absolute position of cities will change. It is common to calculate the new positions as a matrix transformation of the initial position. In mathematical terms, if P is a matrix that represents the initial position, and P' represents the new position, then $P' = RP$, where R is a well known matrix. However, since the center of the earth also rotates about the sun, the position of cities will, of course, change considerably with time. The matrix needed for the transformation is a bit more complicated. We don't have a closed form solution for the general case.

1.3 Rotation with Scaling

The traditional potter starts working with a mound of clay placed on a platform which constantly rotates. As the base rotates, he is able to obtain desired shapes by moulding. One might envisage this as a set of points that get scaled while rotating.



(a) A snapshot of the Solar System (not to scale)



(b) A potter's wheel

Figure 1: Possible scenarios

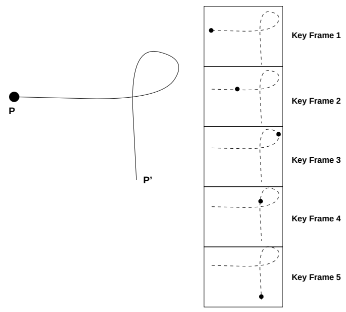
1.4 Animation Algebra

Given the three situations described above, it is natural to ask how a point would appear as it undergoes the effects of a multiple number of translations, rotations, and scaling. In fact, this leads us to an *algebra* consisting of the above three primitives, which might be particularly useful in editing animation. The physics of every case is bound to be different, but we ask the question: Given a transformation *sequence*, how do we generate intermediate positions. *In this paper, we systematically consider the various issues involved.*

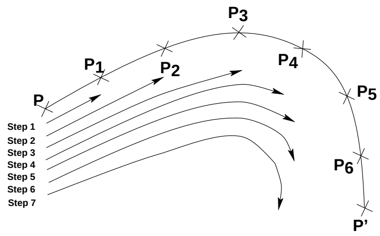
1.5 Algorithmic animation

The proposed problem (posed formally in Section 2) also raises the issue of traditional key frame animation versus, what we call, algorithmic animation. In key frame animation, we are given two event points P and P', and we are asked to define the scene between P and P'. Implicit in this method is the assumption that we know the path between P and P' precisely, or at least, we are allowed to assume this (typically, linear).

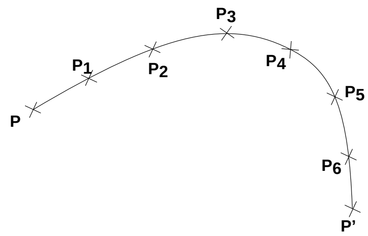
In algorithmic animation, we are essentially given P. The final position P' is *algorithmically* defined by virtue of operations such as rotation and scaling. We are asked to determine intermediate positions. As an example, for the nail on the tire (trachos), we are given



(a) Typical key frame animation



(b) Application of matrices for animation



(c) An alternative to calculate intermediate positions

the initial position P. We are told that the nail reaches a new position because of translation while rotation by one turn. We are asked to generate intermediate positions. In this specific case, we are fortunately able to generate the path. If the final position is described in an algorithmic way, the closed form expression for the path is impossible.

Our notion of algorithmic animation is similar to traditional *forward* animation (Figure 2(b)) in which a programmer uses matrices to specify intermediate positions. It differs, however, in the *crucial* aspect that, in our case, the parameters for computing intermediate positions are not explicitly specified.

In other words, algorithmic animation is like key-frame animation in that intermediate positions are not specified using any parameters generated. It is like forward animation in that we use a hierarchy of embedded matrices to generate the actual positions (once the parameters for intermediate positions are computed within the algorithm).

1.6 Summary of our work

In this paper, we introduce the problem of algorithmic animation algebra (examples of applicability are indicated in Section 1.7). We also provide two solutions for this problem: a matrix based method and a geometric method. These solutions may be viewed as "unraveling the matrix stack hierarchy," and are applicable in their own right as an optimization technique for matrix concatenation (as might appear in forward animation). Note that we do *not* automatically compute constraints involved in the physical world in this work.

1.7 Application domain

As alluded earlier, the methods suggested in this work would be most useful in an animation editing system. An animation editor would be a superset of a conventional picture editor, but would also allow an editing sequence such as "Show the animation when point A moves 100 units about another point B which in turn rotates 35 degrees about point C which in turn scales with respect to yet another point."

Further, this system would be even more useful in a multimedia authoring system which allows one to develop lessons for students to understand various concepts in animation. A less gifted student might be interested in only two levels of rotation, for instance whereas an advanced student would be interested in a very complicated situation. Extended further, a likely domain for this work is in applications related to virtual reality and modern 3-D gaming, where, the programmer is unlikely to know in advance what operations a user is likely to be interested in.

This work would not be particular interesting if the objects to be modeled are so complex, that it is impossible to break up the animation as a sequence of the primitives supplied here. An example of this would be a handkerchief falling on a table – it is better to go directly to physics based modeling [1].

The rest of this paper is organized as follows. In the next section, we specify the problem formally and follow it up with generic solutions. Details of a straightforward method is given in Section 4. A new method is proposed in Section 5. Finally, we conclude in Section 6 with a summary of the paper and a tabular comparison of both methods.

2 Problem Definition

Given the assumption of a standard Cartesian coordinate system, we define an animation scenario as follows:

- Each translation is specified in terms of two variables, Δ_x and Δ_y, each representing the net translation of a point parallel to the two axes.

- Each rotation is defined in terms of three variables, c_x, c_y and Θ , where (c_x, c_y) is the center of rotation and Θ is the total angle of rotation that each point must undergo.

- Each scaling is defined in terms of four variables, s_x, s_y, Γ_x, and Γ_y, where (s_x, s_y) is the center of scaling and Γ_x and Γ_y are the scale factors in the X and Y directions, respectively.

- Any animation is specified in terms of a combination of zero or more translations, zero or more rotations and zero or more scalings.

The problem is now defined as follows: *Perform an animation by generating intermediate positions, given a specification in terms of some set of translations, rotations, and scaling.* The goal, of course, is to perform this as efficiently as possible.

2.1 The setting

An animation is performed by breaking up the entire motion into a given, finite number of steps, N. Then, at each step each point must be transformed by the appropriate amount (to be determined), erased from its previous position and drawn at the new location.

The number of steps required to perform an animation controls the speed of animation along with a delay factor that may be introduced between each step of animation.

An object may be defined as some finite set of points. To animate an object, the following procedure is adopted (from the computational efficiency point of view).

- Perform some calculations (termed *overheads* in this paper) common to all points that need not be repeated for each individual point.

- Perform transformations on each individual point.

3 Overview of the solutions

A matrix based solution (e.g., [2], [3]) would (i) keep a copy of the original points, (ii) get the specifications for each translation, rotation and scaling, (iii) at each step, calculate the transformation required for each translation, rotation and scaling and generate the corresponding matrix. (For example, if a translation of 100 pixels must be done in fifty steps, the transformation at the tenth step would be 20 pixels, or if a rotation of 180 degrees must be completed in hundred steps the transformation at the tenth step would be calculated for 18 degrees.), and (iv) the matrices are then multiplied to generate the final matrix and this matrix is "applied" to each point undergoing the animation. Figure 2(b) illustrates the paradigm.

One way to avoid matrix multiplications at run time is to calculate the incremental transformation that would be required at each step. This may be achieved by generating the incremental transformation matrix for each specification and then multiplying the matrices before beginning the animation. We see below (see Section 4.3) that this strategy is not always possible, ruling out the possibility of an easy way to embed the forward approach to algorithmic animation. This has been seen before in the use of quaternions [4, 5] for the simple case involving translation and rotation. Figure 2(c) describes this approach, which we term as the *differences* approach.

We also propose a geometric approach based on prior work [6]. The geometric method has essentially three stages. In the first stage some preprocessing is performed on an object. In the second stage the actual animation is displayed step by step and in the last stage some amount of "resetting" may be required. The geometric method is similar to the differences approach of the previous paragraph. However, the domain of situations that this method can handle is significantly larger.

4 The matrix based method

In Section 4.1 we first describe the method for translation, rotation, and scaling, and then explain in some detail how these are combined. The power of the method is apparent only in the latter cases (combination); the former is presented primarily for illustrative purposes.

4.1 Elementary Operations

By elementary operations, we mean the operations that involve only translation, or only rotation, or only scaling. The gross deformation (i.e, the net result of the specification) in each of the three cases are given by [7] the matrices M_t, M_r and M_s where

$$M_t = \begin{pmatrix} 1 & 0 & \Delta_x \\ 0 & 1 & \Delta_y \\ 0 & 0 & 1 \end{pmatrix}$$

$$M_r = \begin{pmatrix} \cos\Theta & -\sin\Theta & c_x(1-\cos\Theta)+c_y\sin\Theta \\ \sin\Theta & \cos\Theta & c_y(1-\cos\Theta)-c_x\sin\Theta \\ 0 & 0 & 1 \end{pmatrix}$$

and

$$M_s = \begin{pmatrix} \Gamma_x & 0 & (1-\Gamma_x)s_x \\ 0 & \Gamma_y & (1-\Gamma_y)s_y \\ 0 & 0 & 1 \end{pmatrix}$$

The coordinates of the points are in a column vector format. For instance, the net result $P\prime = M_r P$ in the case of rotation.

4.2 The differences approach

Once the net result is known, we need to obtain intermediate positions. In each one of the three admittedly trivial cases, we have the situation where we are given a position vector P and a matrix M such that the new vector $P\prime$ is obtained as $P\prime = M \times P$.

In the differences approach, the goal is to compute a difference matrix A such that the N intermediate positions $P_1 = A \times P$, $P_2 = A \times P_1$, and so on, till $P_N = A \times P_{N-1}$ are obtained.

In general, then A is the N^{th} root of the transformation matrix which achieves the same effect in one step.

Fortunately, in the three cases under consideration, we are able to obtain A. In particular, the differences matrix A_t, A_r and A_s for pure translation, rotation and scaling are

$$A_t = \begin{pmatrix} 1 & 0 & \delta_x \\ 0 & 1 & \delta_y \\ 0 & 0 & 1 \end{pmatrix} \quad (1)$$

$$A_r = \begin{pmatrix} \cos\theta & -\sin\theta & c_x(1-\cos\theta)+c_y\sin\theta \\ \sin\theta & \cos\theta & c_y(1-\cos\theta)-c_x\sin\theta \\ 0 & 0 & 1 \end{pmatrix} \quad (2)$$

and

$$A_s = \begin{pmatrix} \gamma_x & 0 & (1-\gamma_x)s_x \\ 0 & \gamma_y & (1-\gamma_y)s_y \\ 0 & 0 & 1 \end{pmatrix} \quad (3)$$

Here, $\delta_x = \Delta_x/N$, $\delta_y = \Delta_y/N$, $\gamma_x = \log^{-1}((\log\Gamma_x)/N)$, $\gamma_y = \log^{-1}((\log\Gamma_y)/N)$ and $\theta = \Theta/N$

It can be shown [8] that the matrices A_t, A_r and A_s are indeed the N^{th} roots of the matrices M_t, M_r and M_s defined earlier. To summarize, to handle any elementary operation, we compute one of the relevant matrices above, and at run time, apply, for each point, the desired matrix. Once an intermediate position is obtained, the algorithm applies the same matrix to obtain a new intermediate position. The principal advantage of the differences approach is that the computational complexity at run time is significantly low; however, as seen below, this method cannot be applied in general.

4.3 A non-trivial case

Consider Fig 2 where a vertical slot indicated by the shaded circle translates, and while it translates, it also rotates.

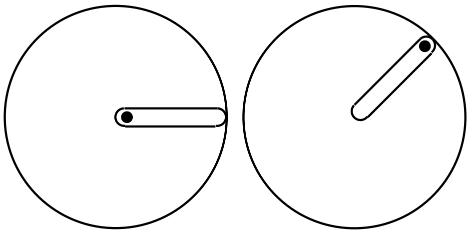


Figure 2: The top view of a vertical slot is shown. The slot on the left translates and rotates to reach the position on the right wheel.

Since this case is a combination of translation and rotation, we are in fact justified in writing the final position of the slot as $P' = M_r M_t P$. In such situations, we might (correctly) write the net transformation matrix as

$$B = \begin{pmatrix} \cos\Theta & -\sin\Theta & \Delta_x \cos\Theta - \Delta_y \sin\Theta + c_x(1-\cos\Theta) + c_y \sin\Theta \\ \sin\Theta & \cos\Theta & \Delta_x \sin\Theta + \Delta_y \cos\Theta + c_y(1-\cos\Theta) - c_x \sin\Theta \\ 0 & 0 & 1 \end{pmatrix}$$

We are tempted to write the differential transformation (using the notation of Section 4.2) as

$$A = \begin{pmatrix} \cos\theta & -\sin\theta & \delta_x \cos\theta - \delta_y \sin\theta + c_x(1-\cos\theta) + c_y \sin\theta \\ \sin\theta & \cos\theta & \delta_x \sin\theta + \delta_y \cos\theta + c_y(1-\cos\theta) - c_x \sin\theta \\ 0 & 0 & 1 \end{pmatrix}$$

It can, however, be shown that in all but pathological situations, A does *not* serve the purpose.

4.4 The Full Matrix Approach

The previous section indicates that it is not always easy to find the difference matrix. We are essentially forced to calculate the N^{th} root of a matrix. In fact, if we had no way of knowing how a point reached its final position from the start position, we would have no recourse but to compute this quantity if we are to determine intermediate positions. In this work, this is not the unhappy situation. The final position is obtained from the initial position using only three primitives, namely, translation, rotation, and scaling.

Note that determining the N^{th} root of a matrix is not the same as finding the root of the elements of a matrix; for details see [9]. A further difficulty with the differences approach is that one may not be able to find a real difference matrix since the N^{th} root in general could contain complex numbers.

We therefore abandon determining a single difference matrix A. Instead, we compute N matrices A_i, $1 \leq i \leq N$ such that the N intermediate positions $P_1 = A_1 \times P$, $P_2 = A_2 \times P$, and so on, till $P_N = A_N \times P$ are obtained. This is the full matrix approach.

The first step in the full matrix approach is to determine the sequence of primitives (determined by the alphabet `T`, `R` and `S`) used in determining the final position. In the example of the previous section, the sequence is `RT` (indicating "rotation" and "translation") although the string could be "long" in general. Let the size of the string be m. Then each of the m primitives will determine an incremental matrix a_{i_k}, $1 \leq k \leq m$ such that A_i is the product of these individual matrices. Further, each a_{i_k} can be computed fairly simply.

We first demonstrate the approach for sample cases in the sequel.

4.4.1 Translation with Rotation

For the translating slot of figure 2, the string is simply `RT`. Here $P_1 = A_1 \times P$ where $A_1 = a_{1_1} \times a_{1_2}$ where $a_{1_1} = A_r$ and $a_{1_2} = A_t$ (See Equations 1 and 2). Next, $P_2 = A_2 \times P$ where $A_2 = a_{2_1} \times a_{2_2}$ and where a_{2_1} is the square of the matrix a_{1_1} and a_{2_2} is the square of the matrix a_{1_2}. In general, $P_i = A_i \times P$ where $A_i = a_{i_1} \times a_{i_2}$ and where a_{1_1} is the i^{th} root of the matrix a_{i_1} and where a_{1_2} is the i^{th} root of the matrix a_{i_2}. Note that the final position is correctly determined by the boundary conditions and the fact that we could determine the N^{th} root of the elementary cases. That is, $P_N = RTP$, where R and T represents the net rotation by definition, and $R = A_r^N$ from Section 4.2.

4.4.2 Translation with Scaling

Was it unfortunate that we were unable to determine the incremental matrix in the above case but had to fall back to full matrix multiplication at run time? We investigate further for the case of figure 3 where an elastic compressible ball falls down (translates) due to gravity while simultaneously undergoing reduction in size. Note that the use quaternions is precluded when it comes to combining operations with scaling.

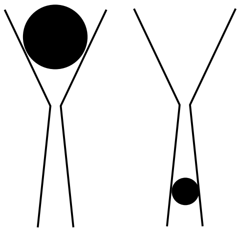


Figure 3: An elastic, compressible object traveling through a funnel.

In this case, once again the gross deformation is obtained from

$$M = \begin{pmatrix} \Gamma_x & 0 & \Gamma_x\Delta_x + (1-\Gamma_x)s_x \\ 0 & \Gamma_y & \Gamma_y\Delta_y + (1-\Gamma_y)s_y \\ 0 & 0 & 1 \end{pmatrix} \quad (4)$$

It is shown [8] by means of proof by contradiction that presuming the difference matrix to be

$$A = \begin{pmatrix} \gamma_x & 0 & \gamma_x\delta_x + (1-\gamma_x)s_x \\ 0 & \gamma_y & \gamma_y\delta_y + (1-\gamma_y)s_y \\ 0 & 0 & 1 \end{pmatrix} \quad (5)$$

is incorrect.

Therefore, we return to the full matrix solution. Here the editing sequence is `ST`.

For this case, $P_1 = A_1 \times P$ where $A_1 = a_{1_1} \times a_{1_2}$ where $a_{1_1} = A_s$ and $a_{1_2} = A_t$ (See Equations 1 and 3). Next, $P_2 = A_2 \times P$ where $A_2 = a_{2_1} \times a_{2_2}$ and where a_{2_1} is the square of the matrix a_{1_1} and a_{2_2} is the square of the matrix a_{1_2}. In general, $P_i = A_i \times P$ where $A_i = a_{i_1} \times a_{i_2}$ and where a_{1_1} is the i^{th} root of the matrix a_{i_1} and where a_{1_2} is the i^{th} root of the matrix a_{i_2}.

4.4.3 The general case

We are now ready to describe the general case. The animation is specified as a string $f_{j_1} f_{j_2} \dots f_{j_m}$ from the alphabet `T` `R` and `S`.

The algorithm generates N matrices A_i, $1 \leq i \leq N$.

As the base case, A_1 is the product of m matrices a_{1_k}, $1 \leq k \leq m$ where, $a_{1_k} = A_t$ if f_{j_k} is `T`; $a_{1_k} = A_s$ if f_{j_k} is `S`; and $a_{1_k} = A_r$ if f_{j_k} is `R`

To generate the i^{th} point P_i, we use A_i as the product of m matrices a_{i_k}, $1 \leq k \leq m$ where, a_{i_k} is the i^{th} power of the matrix a_{1_k}.

The method to perform animation is then as follows. For the first intermediate position, we compute the matrix A_1 and then apply this matrix on all points. We next erase these intermediate position, compute the matrix A_2 and then apply this matrix on all points to generate the second stage of the animation. The algorithm proceeds in a similar manner for subsequent stages.

4.5 Computational Complexity

In the following, by overheads we refer to pre-processing (and post-processing) computational costs that are not incurred in the main animation loop. We illustrate only one case here (for a comprehensive treatment, see [8]) to show the optimizations that can be performed.

For the sequence `TS` we have

$$A_k = M_t^k \times M_s^k = \begin{pmatrix} \gamma_x^k & 0 & \begin{matrix} s_x(1-\gamma_x^k) \\ +k\delta_x \end{matrix} \\ 0 & \gamma_y^k & \begin{matrix} s_y(1-\gamma_y^k) \\ +k\delta_y \end{matrix} \\ 0 & 0 & 1 \end{pmatrix}$$

where γ_x, γ_y, δ_x and δ_y are as defined previously. The use of the above matrix would yield the point say P_k from P. To obtain the next intermediate position say P_{k+1} from P we have the matrix

$$A_{k+1} = \begin{pmatrix} \gamma_x^{k+1} & 0 & (1-\gamma_x^{k+1})s_x + (k+1)\delta_x \\ 0 & \gamma_y^{k+1} & (1-\gamma_y^{k+1})s_y + (k+1)\delta_y \\ 0 & 0 & 1 \end{pmatrix}$$

We observe that the elements of matrix A_{k+1} can be obtained by re-substituting the elements of A_k. Infact the elements of A_{k+1} can be computed from the variables that constitute the elements of matrix A_k as is obvious from the definition of matrix A_{k+1}. The additional computations required for this is less than full matrix multiplications.

4.5.1 Overhead Analysis

- 4 Non-Arithmetic computations(log and antilog),
- 4 Divisions(2 to compute γ_x and γ_y and 2 to compute δ_x and δ_y),
- 2 Multiplication, 2 Additions and 2 Subtractions to setup the initial matrix.

4.5.2 Per Step Run-Time Analysis

- $2N_p$ Multiplications, $2N_p$ Additions
- 4 Multiplications, 4 Additions and 2 Subtractions in order to obtain matrix A_{k+1} from matrix A_k

5 Geometric Method

The disadvantage of the full matrix method is that it, by and large, does not generate a new intermediate point based on points already computed. We need to do matrix multiplication at run time. We have developed a different method, based on geometry that avoids run time multiplication. The disadvantage of this method is that it requires a case by case analysis of the string employed and is, as a result, harder to implement.

The geometric method has essentially three stages. In the first stage some preprocessing is performed on an object. In the second stage the actual animation is displayed step by step and in the last stage some amount of "resetting" may be required.

The intuition behind this is simple to understand when we consider the case of elementary scaling, and elementary rotation, discussed in Section 5.1. More complex cases are considered in [8]. A comparison with traditional matrix based methods is also made later and this might well enhance the understanding of this method.

5.1 Rotation about an arbitrary point

To rotate a point (x, y) about the origin we use the equations for shear rotation[10], which sets in order

$$\begin{aligned} x &= x + (a * y) \\ y &= y + (b * x) \\ x &= x + (a * y) \end{aligned} \tag{6}$$

where $a = -\tan(\theta/2)$ and $b = \sin(\theta)$.

The above equations have the same effect of the matrix M_r of Section 4.1 if we assume that the center of rotation is at the origin. In the general case, the animation consists of the stages

1. Before animation translate the object such that the center of rotation is at the origin. That is, for each point (x, y) in the object do $x = x - c_x$ and $y = y - c_y$.
2. At each step
   - Rotate each point using Equation 6
   - Draw the object using the co-ordinates $(x + c_x, y + c_y)$ without actually incrementing the point (x, y).
3. After the animation is completed the final locations are available by setting $x = x + c_x$ and $y = y + c_y$.

We remark that the primary motivation for adopting Equation 6 is to reduce run time computations. Equation 2 could well have been employed here. We also notice that a third stage is necessary in this case.

6 Concluding Remarks.

In this paper we have introduced the notion of algorithmic animation, and placed it in the context of key-frame animation (intermediate positions are obtained by trial and error), and forward animation (intermediate positions are specified by a programmer before compilation). To our knowledge, this topic has not been explored elsewhere, although it is conceivable that some proprietary commerical systems may have implemented animation algebra in partial form. We defined algorithmic animation in terms of primitive transformations, and considered the cases where these transformations form a possibly long, animation algebra string. We noted that a naive way of embedding the forward animation in the context of algorithmic animation is incorrect, and provided two different solutions to the problem. We have implemented some of the transformations on the X-windows platform.

Animation String	Matrix Method	Geometric Method
T	$2N_p$	$2N_p$
S	$4N_p$	$2N_p$
R	$8N_p$	$8N_p$
T S	$4N_p + 10$	$2N_p$
S T	$4N_p + 12$	$2N_p$
T R	$8N_p + 18$	$8N_p + 2$
R T	$8N_p + 23$	$10N_p + 6$
R S	$8N_p + 31$	$16N_p$
S R	$8N_p + 27$	$10N_p$
T R S	$8N_p + 33$	$16N_p + 2$
R R R ... n	$8N_p + 14n - 20$	$8N_p + 8n - 4$

Table 1: Run-time complexity comparison for one animation step for an object with N_p points. T, R, and S stand for translation, rotation, and scaling respectively.

The comparison of the two methods been tabularized in Table 1 and Table 2. In several cases, the geometric method is superior.

The geometric method philosophy mixes ideas from the incremental differences method (Section 4.2) with the correct full matrix method (Section 4). When a speedup is obtained, it is from two sources: shifting a large part of the calculations to pre and post run-time overheads, and by storing a few intermediate results to do "incremental" calculations. This leads to an increase in the overheads, which unfortunately depends on the number of points. These overheads reduce the running time during the main animation loop.

A direct implementation of the matrix method would make it inferior to the geometric method in all

	Pre Processing		Post Processing
Animation String	Matrix Method	Geometric Method	Geometric Method
T	2	2	0
S	6	8	0
R	10	$2N_p + 4$	$2N_p$
TR	12	$2N_p + 4$	$2N_p$
RT	18	$2N_p + 2$	$2N_p$
ST	14	$6N_p + 8$	0
TS	14	$6N_p + 6$	0
RS	35	$6N_p + 6$	$2N_p$
SR	34	$16N_p + 15$	$2N_p$
TRS	35	$6N_p + 10$	$2N_p$
RRR...n	10n	$2N_p$ + 6n - 2	$2N_p$ + 2n - 1

Table 2: Comparison of overheads for the two methods. The Matrix method requires no post procssing. The numbers are for one animation step for an object with N_p points.

situations. By reusing prior computed values, we narrowed the gap between the two methods in some cases, and made it superior to the geometric method in some cases.

References

[1] S. Chandran and N. Damodharan, "Modeling the behaviour of cloth drape," in *ICVGIP 98* (S. Nayar and S. Chaudhuri, eds.), Viva Publishers, December 1998.

[2] "Matrix FAQs." `http://skal.planet-d.net/demo/matrixfaq.htm`, September 1997.

[3] J. Neider, T. Davis, and M. Woo, *The Official Guide to Learning OpenGL*. Addison-Wesley Publishing Company, 1994.

[4] K. Shoemake, "Quaternions and 4×4 matrices," in *Graphics Gems II* (J. Arvo, ed.), ch. VII, pp. 351–354, Academic Press, 1991.

[5] D. Hearn and M. Baker, *Computer Graphics*. Prentice-Hall, 1995. pp. 419–420.

[6] M. G. Nanda and S. Chandran, "On transformations for two-dimensional animation," in *Proceedings of the Second National Conference on CAD/CAM*, (Coimbatore–641 004, India), PSG College of Technology, August 1994.

[7] D. F. Rogers and J. Alan, *Mathematical Elements for Computer Graphics*. McGraw-Hill Publishing Company, Singapore, 1990.

[8] S. Chandran, Y. Kulkarni, and P. Nath, "On transformations for animation," tech. rep., Computer Science and Engg Department, IIT-Bombay, 2000. `http://www.cse.iitb.ernet.in/˜ sharat/twod.html`.

[9] G. H. Golub and C. F. Van Loan, *Matrix Computations*, ch. 11, pp. 397–400. North Oxford, Oxford, 1983.

[10] A. W. Paeth, "A fast algorithm for general raster rotation," in *Graphics Gems* (A. S. Glassner, ed.), pp. 179–195, Academic Press, 1990.