Camera Keyframing Using Linear Interpolation of Matrices

Amy Hawkins and Cindy M. Grimm*
Washington University in St. Louis

Abstract

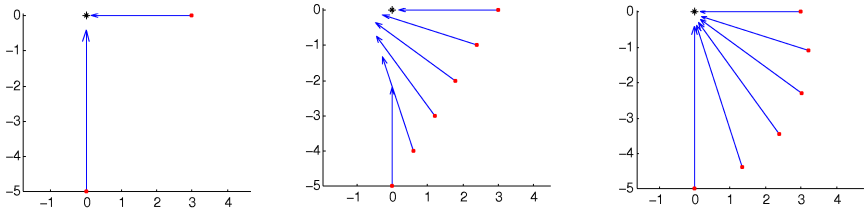
Alexa's method for linearly interpolating matrices is well-suited for application to camera matrices. This paper discusses implementation issues that arise when applying this method to camera interpolation. We show how to include the perspective matrix, even though Alexa's operators cannot be applied directly to it. We discuss cases where Alexa's operators fail to converge and show how to work around this problem. Additionally, we present implementation details for three interpolation methods: linear, spline-based smooth approximation, and smooth interpolation using subdivision.

We also discuss general issues with implementing Alexa's method. We note changes to his provided pseudocode and discuss suitable values for ε to maximize efficiency. Finally, we show examples of the technique and describe a quality metric that can be used to compare our technique to camera parameter interpolation.

1 Introduction

Camera keyframing is widely used in animation. A user places the camera in a sequence of "key" positions, and the computer produces a set of intermediate cameras. The camera has 11 degrees of freedom (position, orientation, etc.) [MC80]. One keyframing method is to interpolate each of these parameters separately to construct the intermediate matrices. Unfortunately, a simple example demonstrates how camera parameter interpolation fails. Figure 1(a) shows an object in the scene (the black star). Two cameras face this object. (The red dot represents the camera's location; the arrow represents the direction it faces.) Figure 1(b) shows the result of camera parameter interpolation. In many of the intermediate frames, the camera no longer faces the object.

*e-mail: {aeh1,cmg}@cse.wustl.edu. Funded in part by NSF grant CCF 0238062.



(a) Two keyframes (b) Parameter interpolation (c) Our interpolation

Figure 1: An example where camera parameter interpolation fails. The object in the scene is represented by a black star. Camera positions are shown as red dots. The orientation of each camera is shown by an arrow. b) Simply interpolating the parameters usually causes the object to move out of the field of view. c) Using linear matrix combinations does a better job of keeping the object centered in the view (although it is not guaranteed to do so).

To get around the rotation problem, most systems instead let the user specify a `from` and `at` point for each key frame and derive the rotation parameters, interpolating the remaining parameters separately. This is an effective method, but it does require some experience to decide where to place the `at` point to get the desired result. The `from` and `at` model also does not help with interpolating center of projection (COP) parameters. These parameters produce interesting perspective distortions, but are little used because, until recently, they were difficult to specify [GS05] and simply interpolating the parameters nearly always fails.

In this paper we show that the linear matrix interpolation introduced by Alexa [Ale02] can be adapted to direct interpolation of key frames. By performing interpolation on the camera matrix itself, rather than on each of the parameters separately, we generate reasonable in-between frames even for sequences containing both rotation and center of projection changes.

In Section 5.1, we show a way to quantify the difference between these two results. The most important property for a good interpolation is that the objects on the screen move in a way that is intuitive to the animator. In the example above, the black star is centered on the screen in each of the keyframes, but during the intermediate frames the object moves to the right side of the screen. Meanwhile, in our interpolation, the object stays close to the center of the screen during the entire animation. Since extra motion is unintuitive to the user, we provide a metric to quantify the overall amount of motion on the screen. The metric shows that in general, our method does a better job of minimizing this motion.

1.1 Contributions

We first discuss issues which arise when applying this method to camera interpolation. We show how to include important components of the perspective matrix, even though Alexa's operators cannot be applied directly to it. We discuss cases where Alexa's operators fail to converge and show how to re-phrase the problem in a more stable manner. Additionally, we present implementation details for three interpolation methods: linear, spline-based smooth approximation, and smooth interpolation using subdivision.

Next, we provide general information on Alexa's technique. Alexa gives iterative methods for implementing the two operators required by his technique. We note corrections to his pseudocode, and we discuss suitable values for ε to maximize efficiency.

Finally, we describe a metric that can be used to compare methods of camera interpolation, and show that our method is a significant improvement over camera parameter interpolation.

2 Background: The $\odot$ and $\oplus$ operators

Our method uses the $\odot$ and $\oplus$ operators developed by Alexa [Ale02] to generate linear combinations of camera matrices.

The $\odot$ operator implements scalar multiplication of a transformation matrix. Example: Given a scalar s and transformation matrices A and B, let $B = 2 \odot A$. Applying B gives the same result as applying A twice.

The $\oplus$ operator is similar to matrix multiplication, with the exception that $\oplus$ is commutative. The operator is defined so that if $AB = BA$, then $A \oplus B = AB = BA$. In the case where $AB \neq BA$, then $\oplus$ can be understood as applying A and B simultaneously.

The operators are implemented using the matrix logarithm and exponential:

$$s \odot A = e^{\,s \log A}, \tag{1}$$

$$A \oplus B = e^{\,\log A + \log B}. \tag{2}$$

(For more information about why the matrix logarithm and exponential are the right choice to achieve the properties explained above, see [Ale02].)

Once these operators are defined, we use them to write the usual linear interpolation equation:

$$[(1-t) \odot A] \oplus [t \odot B] \quad t \in [0,1]. \tag{3}$$

In the next section, we use this equation to interpolate between two camera matrices, and we examine some issues which arise. Then, we show how to use Alexa's operators

to perform two additional types of interpolation: spline-based smooth approximation, and smooth interpolation using subdivision.

3 Application to Camera Matrices

3.1 Decomposing the Perspective Matrix

Four matrices are multiplied to define the camera matrix: perspective, scale, rotation and translation. Michener and Carlbom [MC80] describe how to build these matrices (commonly denoted as P, S, R, and T).

Alexa's method was designed to work only for matrices which have no negative real eigenvalues. Unfortunately, the perspective matrix does not fit this requirement. While we could simply discard P when interpolating, this would prevent several useful parameters from being interpolated. The usual perspective matrix is:

$$P = \begin{bmatrix} \alpha & \gamma & u_0 & 0 \\ 0 & 1 & v_0 & 0 \\ 0 & 0 & \frac{-1}{1+k} & \frac{k}{1+k} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{4}$$

where α is the aspect ratio, γ is the skew, (u_0, v_0) is the center of projection, and k is the distance to the near clipping plane divided by the distance to the far clipping plane.

To overcome the restrictions for using Alexa's method, we decompose P into two parts. P_b contains the important parameters and also fits Alexa's requirements. P_a is generally the same for all keyframes and need not be interpolated.

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{-1}{1+k} & \frac{k}{1+k} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} \alpha & \gamma & u_0 & 0 \\ 0 & 1 & v_0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = P_a P_b \tag{5}$$

Assume we are given a function `Lerp(A,B,t)` which implements (3). Assume further that we have computed matrices P_{b1} and P_{b2} — one from each input keyframe, built from (5). Finally, assume that we have the remaining S, R, and T matrices for each input camera. We write the following code to perform the interpolation:

$$\texttt{Lerp(}\ P_{b1}(SRT)_1\ ,\ P_{b2}(SRT)_2\ ,\ t\ \texttt{);} \tag{6}$$

3.2 Issues with Large Rotations

Alexa claims that "a rotation by π together with a non-uniform scale [...] is the only type of transformation that cannot be handled". Unfortunately, we found this claim to be untrue. The matrix square root fails to converge for many transformations with rotational component $\geq \frac{\pi}{2}$ combined with a non-uniform scale. In a test batch of 1000 such matrices, we found that the computation failed to converge 130 times. We saw no obvious difference between those that converged correctly and those that failed. Therefore, for the purposes of this paper, we consider any matrix with the above properties potentially unsafe.

We address this problem by making the following observation: We can transform both input matrices so that one of them lies at the origin, thus bringing the rotational components of both matrices down to a safe range. One or both of the input matrices may have a large rotational component *before* this transformation is performed. However, as long as the rotation *between* the two matrices is $< \frac{\pi}{2}$, then the transformation will ensure that both matrices are safe when Alexa's operators are applied. This also helps with stability and predictability because the interpolation is always performed at the same point in transformation space. The snippet of code in (6) should be changed accordingly:

$$\Big(\texttt{Lerp}(\ P_{b1}(SRT)_1(SRT)_1^{-1}\ ,\ P_{b2}(SRT)_2(SRT)_1^{-1}\ ,\ t\ ) \Big)\ \ (SRT)_1\ ; \tag{7}$$

If this transformation fails to bring the rotational components down to a safe range, it indicates that the user has chosen keyframes where the rotation from one keyframe to the next is $\geq \frac{\pi}{2}$. This is both rare and easily detected, and we can prompt the user to add an additional keyframe to solve the problem.

Finally, we note that the animation resulting from (7) may be slightly different from the animation resulting from (6), and that the numerical differences increase (from magnitude 10^{-4} to magnitude 10) as the camera (and scene) are moved away from the origin (by a distance of 10^3). In practice, we found that, whether or not the scene was centered at the origin, there was little, or no, noticeable *visible* difference between the two animations.

3.3 Smooth Interpolation

As an alternative to using the linear interpolation in (3), we present two methods for calculating smooth interpolations. For each method, we revisit the large rotation issue discussed above.

3.3.1 Approximate

For some applications, we may wish for the camera to take a smooth path rather than one that is discontinuous at each keyframe. If we are willing to settle for a path which approximates the keyframes but may not pass precisely through each one, then this can be easily accomplished. We can combine an arbitrary number of matrices by using Alexa's operators in the following way:

$$\bigoplus_i w_i \odot M_i \quad = \quad e^{\sum_i w_i \cdot \log M_i} \tag{8}$$

where M_i are the matrices and w_i are non-uniform B-spline weights [BBB87] for a spline curve of degree C^k. In our implementation, $k = 2$.

This requires a small change in the way that we deal with large rotations. Above, we considered each pair of keyframes, and transformed both so that one lay at the origin. Here, we may be summing up to $k+2$ matrices at once. Therefore, we transform all the input matrices so that the matrix with the largest weight lies at the origin.

Since we transform more than 2 matrices at a time, there is a slightly higher likelihood that the transformation will fail to bring the rotational components to a safe range. If we wish to add additional keyframes to correct this failure, we must bear in mind that no transformed matrix may have rotational component $\geq \frac{\pi}{2}$, and add intermediate keyframes accordingly. We reiterate that this case happens rarely — the user must specify fairly extreme changes between keyframes in order for our fix to fail.

3.3.2 Exact

With a bit more care in implementation, it is also possible to generate a smooth path which passes through every keyframe. The four-point subdivision scheme in [DGL87] describes how, given a list of control points $(p_0, \ldots, p_{n-1})$, we can calculate a new point which lies between points p_i and p_{i+1}:

$$p_{new} \quad = \quad \frac{9}{16}(p_i + p_{i+1}) - \frac{1}{16}(p_{i-1} + p_{i+2}) \tag{9}$$

Given a list of keyframes $(M_0, \ldots, M_{n-1})$, we can rewrite this equation using Alexa's operators:

$$M_{new} \quad = \quad [\frac{9}{16} \odot (M_i \oplus M_{i+1})] \oplus [-\frac{1}{16} \odot (M_{i-1} \oplus M_{i+2})] \tag{10}$$

More succinctly, we can use (8) to perform the above calculation, where:

$$w \quad = \quad \{-\frac{1}{16}, \frac{9}{16}, \frac{9}{16}, -\frac{1}{16}\} \tag{11}$$

$$M \quad = \quad \{M_{i-1}, M_i, M_{i+1}, M_{i+2}\} \tag{12}$$

To do this, we must ensure that the boundary case is handled properly. When we wish to compute M_{new} which lies between M_0 and M_1 (or M_{new} between M_{n-2} and M_{n-1}), we must compute a "dummy" matrix for M_{-1} (or M_n). We apply what we know from the boundary case for points:

$$M_{-1} = \texttt{Lerp}\,(\,M_1,\,M_0,\,-1\,)\,; \tag{13}$$

$$M_n = \texttt{Lerp}\,(\,M_{n-2},\,M_{n-1},\,-1\,)\,; \tag{14}$$

If we are given n keyframes as input, and we perform m iterations of subdivision, the result is an animation that is $(n-1)2^m+1$ frames long. In order to accommodate an arbitrary, user-defined number of output frames, we subdivide until $(n-1)2^m+1$ is greater than the number of frames desired. Then, we use `Lerp()` to resample the subdivided frames appropriately.

For clarity, we have described the resampling as a postprocessing step. In reality, this is unnecessary — we can subdivide and resample in a single step. To accomplish this, we do not compute any camera matrices during the subdivision step. Instead, we compute an array where the i^{th} entry is a list of keyframes and weights that will be combined to build the i^{th} camera matrix. In this manner, we "unroll" the recursive nature of the subdivision algorithm. The end result is that every one of our final camera matrices is computed only as a combination of keyframes, not as a combination of previous levels of subdivision. Once we have computed this array, we can compute a frame of the output by using (8) just one time.

For example, say we are building frame i of the output. Then, say that the desired resampling dictates that frame i should be built using the j^{th} and $(j+1)^{th}$ entries of the subdivision array, with interpolation percentage t. Recall that the j^{th} entry in the subdivision array specifies a list of weights, w_j, and a list of matrices, M_j. We can build the resampled camera matrix for frame i by using (8) just a single time, where:

$$w = \{(1-t)w_{j_1}\ ,\ \ldots\ ,\ (1-t)w_{j_p}\ ,\ tw_{(j+1)_1}\ ,\ \ldots\ ,\ tw_{(j+1)_q}\} \tag{15}$$

$$M = \{M_{j_1}\ ,\ \ldots\ ,\ M_{j_p}\ ,\ M_{(j+1)_1}\ ,\ \ldots\ ,\ M_{(j+1)_q}\} \tag{16}$$

We deal with large rotations exactly as in the smooth approximation case. Before using (8), we transform all the input matrices so that the matrix with the largest weight lies at the origin.

4 Operator Implementation

Alexa's operators are implemented as shown in (1), using the matrix exponential and logarithm. Alexa shows how to implement these using iterative methods. In this section, we discuss aspects of this implementation. First, we review the numerical methods

used in the pseudocode, and we note two corrections. Second, we discuss termination conditions for these iterative methods.

4.1 Changes to Alexa's Pseudocode

Our changes to Alexa's pseudocode are noted by the boxed lines in Figure 3(a). Additionally, he uses ε three times to designate three distinct loop termination conditions. We annotate these appearances with subscripts ε_1, ε_2 and ε_3, to facilitate our discussion in the following section.

4.1.1 Matrix exponential

The matrix exponential (Figure 2) is computed iteratively using a Padé approximation with scaling. For the parameter q (the desired number of iterations), we use the value found in MATLAB's implementation, $q = 6$. [ML03]

4.1.2 Matrix logarithm

The matrix logarithm, shown in Figure 3(a), is computed in two steps. First, take the square root of the matrix repeatedly, bringing it within ε_1 of the identity matrix. This is done so that the second step, a truncated Taylor series, will converge properly. The Taylor series continues until the added terms become smaller than ε_2.

The matrix logarithm makes repeated calls to the matrix square root, shown in Figure 3(b). Here, we can directly calculate the quality of our result — simply square the output matrix and compare it to the input matrix. When the result of this comparison is $< \varepsilon_3$, the calculation terminates.

We discuss appropriate values for ε_1, ε_2 and ε_3 in the next section.

```
Compute X = e^A
    j = max (0, 1 + log2(||A||))
    A = 2^-j A
    D = I;  N = I;  X = I;  c = 1
    for k = 1 to q
            c = c(q-k+1) / (k(2q-k+1))
            X = AX
            N = N + cX
            D = D + (-1)^k cX
    end for
    X = D^-1 N
    X = X^2^j
```

Figure 2: Pseudocode for computing the matrix exponential.

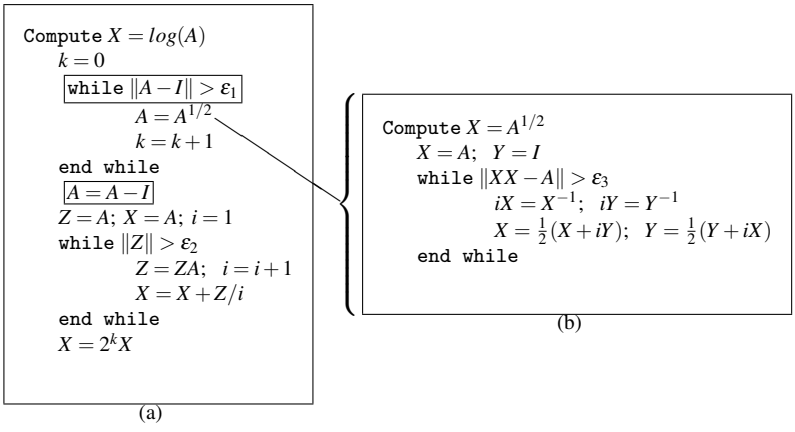


Figure 3: a) Pseudocode for computing the matrix logarithm. Boxed lines show changes from the pseudocode given by Alexa. b) Pseudocode for computing the matrix square root.

4.2 Suitable ε Values

Alexa reminds us that "all while loops in the pseudo codes should terminate after a fixed number of iterations since numerical problems might lead to poor convergence". With that in mind, we study both the expected and maximum number of iterations taken for each of these calculations to converge. Based on our studies, we present suggested values for ε_1, ε_2 and ε_3. We also give a suggested maximum number of iterations to run each computation. (That is, we give a number of iterations, n, such that if the computation has not converged to the specified ε after n iterations, it is unlikely to ever converge.)

We work "from the inside out", considering the pseudocode in the reverse order that it was presented above.

4.2.1 Square root: ε_3

To test the square root function, we generated 100 random transformation matrices (rotations: composite of x,y,z rotation, with rotation angle in $[0, \pi]$, scales: $[10^{-2}, 10^2]$, translations: $[10^{-1}, 10^2]$). We computed the square root of each matrix, and calculated $\|XX - A\|$ (see pseudocode) at each iteration. We would like to choose ε_3 to be as small as possible without causing the expected number of iterations to be too high. The reader should bear in mind that the square root function is called multiple times

by the logarithm function, and thus must be very fast. However, it also needs to be accurate enough that numerical errors do not propagate upwards.

We found that running the computation for at least seven iterations seems to be critical. By the sixth iteration, many of the test matrices had only converged to $\varepsilon_3 = 10^{-2}$. However, by the seventh iteration, every single test matrix had converged to $\varepsilon_3 = 10^{-6}$ or better.

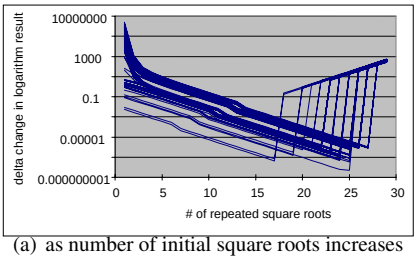
The Verdict: We suggest setting $\varepsilon_3 = 10^{-6}$. We expect the computation to take ≈ 7 iterations, so we run the computation for a maximum of 10 iterations. Note that it is this part of the computation which occasionally fails to converge with large rotations. Therefore, if the computation has run for more than 10 iterations, we return "no convergence" and prompt the user to add an additional keyframe.

4.2.2 Logarithm: ε_2

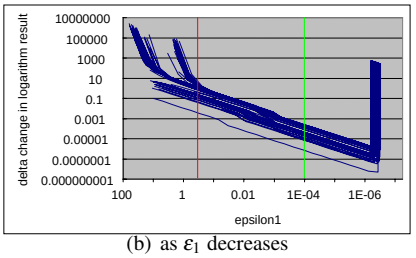
To test the Taylor series portion of the logarithm function, we again generate 100 random transformation matrices and compute their logarithms. We would like to discover how long it takes for the Taylor series computation to converge in practice. We would also like to discover how dependent this computation is upon the preliminary portion of the logarithm algorithm (the repeated square roots).

We find that, if the Taylor series does converge, then $\|Z\|$ always becomes small quickly, regardless of how many square roots we take in the preliminary step. In our test cases, $\varepsilon_2 < 10^{-9}$ after no more than four iterations. The primary reason for lack of convergence is not enough preliminary square roots (ε_3 not sufficiently small).

The Verdict: We suggest setting $\varepsilon_2 = 10^{-9}$. We expect the computation to take ≈ 4 iterations, so we run the computation for a maximum of 7 iterations, returning "no convergence" if we hit 7 iterations. If we ensure $\varepsilon_3 < 10^{-3}$ then we can show that this is more than sufficient. The Taylor series of the log is $\log(Z) = \log(I+B) = |B| - (1/2)|B|^2 + (1/3)|B|^3 \ldots$ If we stop the computation after n iterations than the remaining error can be bounded by $1/(1-\varepsilon_3)\varepsilon_3^n$ which, after 4 iterations, is less than 10^{-9} as desired.



(a) as number of initial square roots increases



(b) as ε_1 decreases

Figure 5: Change in logarithm result with all 100 test matrices overlaid. The vertical lines on (b) represent $\varepsilon = 0.5$ and $\varepsilon = 10^{-4}$

4.2.3 Logarithm: ε_1

To test the preliminary step of the logarithm function, we generate 100 random transformation matrices as before. We know that $\|A - I\|$ must be small, or else the subsequent Taylor series calculation will not converge. We test how $\|A - I\|$ affects the final output of the logarithm function.

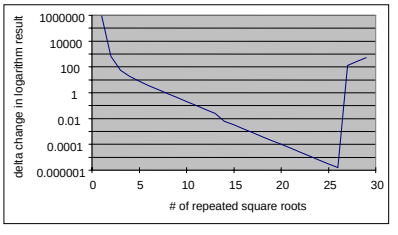


Figure 4: Change in logarithm result as the number of initial square roots increases. (1 test matrix).

For each of our test matrices, we compute its logarithm 30 times, each time increasing the number of preliminary square roots taken. We compare each result to the previous one, and plot the change. Figure 4 shows the graph of one of our test matrices. The x-axis represents the number of repeated square roots taken, and the y-axis shows the change in result as more square roots are added.

We see that as more iterations of square roots are done, the change in the final answer becomes very small, indicating convergence. However, at 26 iterations, this convergence is suddenly lost. This surprising discovery indicates that it is possible to take *too many* preliminary square roots!

To explain this phenomenon, we recall the purpose of the repeated square roots — we aim to bring the input matrix closer to the identity matrix so that the Taylor series will converge. Referring back to the pseudocode, we see that if the input matrix is extremely close to the identity, then Z, A, and especially ZA will become so close to zero that we may lose numerical precision.

Figure 5(a) shows the graphs from all 100 test matrices overlaid on each other. From

this, we see that the threshold for losing numerical precision is not determined by the number of iterations we take. In Figure 5(b), we regraph the information shown in Figure 5(a). This time, the x-axis represents decreasing ε_1 rather than increasing the number of iterations. Here it is clear that we lose numerical precision when ε_1 drops below 10^{-6}. We also note that setting $\varepsilon_1 = 0.5$, as Alexa suggests, yields results which are quite poor. We can see this by examining the red line in Figure 5(b).

The Verdict: The choice of ε_1 will depend somewhat on specific applications and the need for speed versus accuracy. At one extreme, we could set ε_1 just above the numerical precision threshold of 10^{-6}. However, if we cross-reference Figures 5(a) and 5(b), we can see that this will cost us ≈ 20 iterations in the average case, and nearly 30 iterations in the worst cases.

In practice we set $\varepsilon_1 = 10^{-4}$. We expect the computation to take ≈ 12 iterations, and we run the computation for a maximum of 20 iterations. We can convince ourselves that the final result is sufficiently accurate by examining the green line in Figure 5(b). Since the change in the result at this point is quite small for all of our test matrices, we conclude that the result is not being significantly affected by performing any further square roots.

5 Examples and Discussion

Our interpolation is well-behaved, even in situations where camera parameter interpolation fails. In this section we show two examples where our method performs better. We then perform analysis on randomly generated keyframe sequences to show how our method performs on average.

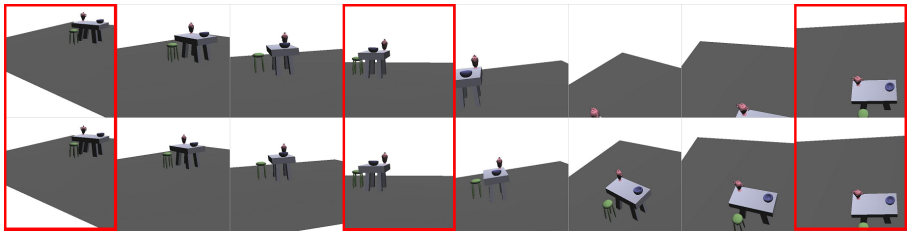


Figure 6: A comparison of camera parameter camera interpolation (top) with our method (bottom). Keyframes are marked in red.

Figure 6 compares our results with those of the camera parameter approach. The top row shows how the table rotates out of view using the camera parameter method. Using our method, (shown in the bottom row) the table stays in view.

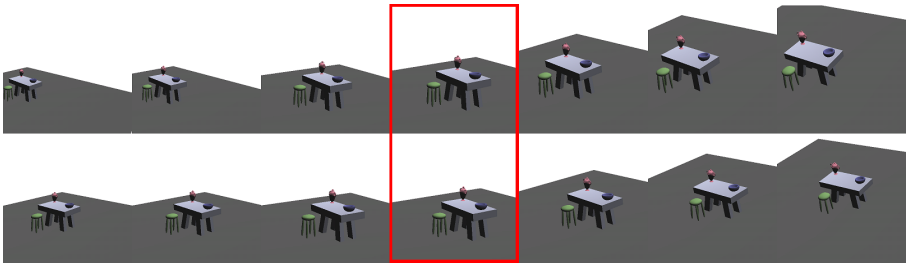


Figure 7: A comparison of camera parameter camera interpolation (top) with our method (bottom). Keyframes are marked in red.

Figure 7 is an example in which the center of projection changes between keyframes. Once again, the camera parameter method (top row) causes the table to nearly move out of view. Using the new method (bottom row), the table stays more centered on the screen.

Although the speed of the camera movement is not readily apparent from the figure, a video of the animation sequence shows that the camera changes speeds rather wildly when camera parameter interpolation is used. Our method results in a much steadier pace that is pleasing to watch.

5.1 Quality Metric

It is important to be able to quantify the results of various interpolation methods. To this end, we have devised a quality metric based on the distance that mesh vertices appear to travel as the camera moves.

To illustrate how our metric is computed, Figure 8 shows two sets of locations, *in camera space*, of a single mesh vertex. The red points are the locations of this vertex during camera parameter interpolation, with t values $\{0.1, 0.2, ...0.9\}$. The blue points are the locations of the same vertex during our interpolation. The location of the vertex at each of the keyframes is also shown.

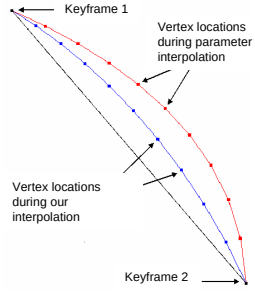


Figure 8: The path taken by a single vertex during our interpolation (blue), versus camera parameter interpolation (red).

To compare the paths taken by the vertex during two different interpolations, we compute the total length of the line segments connecting each set of points. Call these lengths l_{trad} and l_{exp}. Finally, we normalize against l_0 (the length of the shortest possible path between the two

keyframes) by computing:

$$L_{trad} = 100\left(\frac{l_0 - l_{trad}}{l_0}\right)$$
$$L_{exp} = 100\left(\frac{l_0 - l_{exp}}{l_0}\right)$$

These values can be interpreted to mean "during camera parameter interpolation, the vertex traveled a path that was L_{trad} percent longer than the straight line path". We average this value over all the vertices in the mesh. This metric is meaningful because it quantifies the amount of motion seen on the screen.

We randomly generated a test set of 100 pairs of keyframes, and found that our method performs substantially better using this metric. Using our method, the average path taken by a mesh vertex is 4% longer than the straight line path. With camera parameter interpolation, the average path is 23% longer.

6 Web Information

Animations, source code, and an executable demo program are available online at http://www.cse.wustl.edu/~aeh1/camera-interp-jgt/

References

[Ale02] Marc Alexa. Linear combination of transformations. In *SIGGRAPH*, pages 380–387. ACM, 2002.

[BBB87] Richard H. Bartels, John C. Beatty, and Brian A. Barsky. *An introduction to splines for use in computer graphics & geometric modeling*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[DGL87] N. Dyn, J. A. Gregory, and D. Levin. A 4-point interpolatory subdivision scheme for curve design. *Computer Aided Geometric Design*, 4:257–268, 1987.

[GS05] Cindy Grimm and Karan Singh. Implementing the ibar camera widget. *Journal of Graphics Tools*, 10(3):51–64, November 2005. This is the full implementation details for the UIST 2004 paper. There is source code available.

[Hig97] N. J. Higham. Stable iterations for the matrix square root. *Numerical Algorithms*, 15(2):227–242, 1997.

[MC80] J. C. Michener and I. B. Carlbom. Natural and efficient viewing parameters. In *SIGGRAPH*, pages 238–245. ACM, 1980.

[ML03] C. B. Moler and C. F. V. Loan. Nineteen dubious ways to compute the matrix exponential, twenty-five years later. *SIAM Review*, 45(1):3–49, 2003.

[Sho85] K. Shoemake. Animating rotations with quaternion curves. In *SIGGRAPH*, pages 245–254. ACM, 1985.

[ZS99] D. Zorin and P. Schroder. Subdivision for modeling and animation. In *ACM SIGGRAPH Course Notes*, 1999.