# Camera Trajectory Evaluation in Computer Graphics Based on Logarithmic Interpolation

Mikael Fridenfalk
Department of Game Design
Uppsala University Campus Gotland
Visby, Sweden
mikael.fridenfalk@speldesign.uu.se

***Abstract*—A new technique is presented within the field of multimedia software applications, based on a logarithmic shape-preserving piecewise cubic Hermite interpolant for evaluation of camera trajectories in mathematically generated large-scale geometries, such as 3D fractals, with the ability to eliminate the oscillations that currently are associated with interpolation of exponential zooms.**

***Keywords-fractal space; logarithmic; LPCHIP; PCHIP; spline***

## I. Introduction

Piecewise cubic Hermite splines are presently used for high-end interpolation of the trajectories of cameras and 3D objects in computer graphics [2,6,7], such as computer games, but also for computer-controlled cameras in film production.

On an implementation level, the standard method to control a camera in computer graphics is by an object called the target camera [8], defined by camera position, a look-at position and the orientation of the camera around the vector pointing from the position of the camera to the look-at position (called roll). To avoid causing the viewer disorientation or nausea, roll is often set to a constant value.

Presently, the spline interpolation techniques that are used in computer graphics are as a rule not based on shape-preserving ones, here defined as interpolants that are both harmonic and monotonic, such as the Piecewise Cubic Hermite Interpolating Polynomial (PCHIP) in MATLAB [1,4,5], which could be a better choice for camera trajectory control, since PCHIP eliminates the overshooting effects that are associated with the regular variant, see Figure 1 (left), thereby increasing the level of control in camera trajectory design without any practical downside, see Figure 1 (right). A reason for this could be that MATLAB, which is the application that introduced PCHIP to a wider audience, is presently not widely used in systems for generation of motion picture, but rather applications such as image processing.

In Figures 1-2 (left), the trajectories of two sets of breakpoints are evaluated by a regular piecewise cubic Hermite interpolant. While the implementation of the harmonic mean in Figure 1 (middle), eliminates the overshooting effects of the regular interpolant, Figure 2 shows that the harmonic mean does not always work properly, unless the tangents (or slopes) $m_1$ and $m_2$ are limited by locally monotonic constraints, see Figures 1-2 (right).

The main difference between a regular and a shape-preserving piecewise cubic Hermite interpolant is that here, the tangents $m_1$ and $m_2$ in the regular interpolant are functions of the mean values of the differences of adjacent *breakpoints* (or *keyframes*), while the shape-preserving version is based on locally monotonic functions of the harmonic mean of the same, see LPCHIP (for Logarithmic Piecewise Cubic Hermite Interpolation Polynomial) in Figure 12 for the example that was used for the generation of the graphs in this paper. LPCHIP in a non-logarithmic mode (*i.e.*, with the argument *lg* set to *false*), henceforth called the PCHIP equivalent, is not identical to the MATLAB function PCHIP, but a simplified version. The principal difference is that the PCHIP equivalent is designed specifically for a constant step size between the breakpoints. However, by the addition of separate interpolation along the horizontal axis, as shown in Figures 1-2, the step size between the breakpoints becomes automatically variable.

In Figure 3 (left), the effect of camera trajectory evaluation is demonstrated using the regular mean value for the evaluation of $m_1$ and $m_2$ in LPCHIP (with *mode* set to REGULAR) and in Figure 3 (middle), with the adjustments of $m_1$ and $m_2$ by multiplication with a factor of 0.25 instead of 0.5. As shown, while in the latter figure the overshooting effect of the camera position trajectory is reduced compared with the former, at the same time the look-at position trajectory has become rougher. This issue may be addressed by adaptive control, but is by default solved by the application of the PCHIP equivalent, see Figure 3 (right).

This paper consists of the presentation of a new technique and a comparison with standard techniques presently used in computer graphics, represented by the term *regular interpolation*. In Section 2, the application of a logarithm is studied in context with camera trajectory interpolation in exponential zooms, to eliminate the oscillating effects that were discovered using standard interpolation. In Section 3, a detailed solution to the oscillation problem is offered, including the evaluation of interpolation points as a function of arbitrary points in time. This solution was further visually verified by implementation in a computer graphics application primarily designed for visualization of 3D fractals.

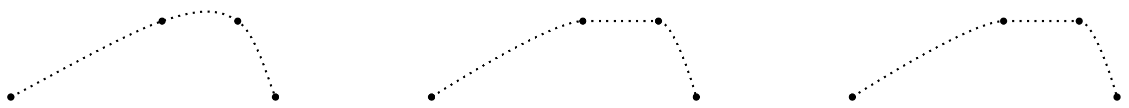
Figure 1: Regular (left), harmonic (middle), harmonic and monotonic (right). As shown in this example, the regular interpolant causes a slight overshoot between the second and the third breakpoints.

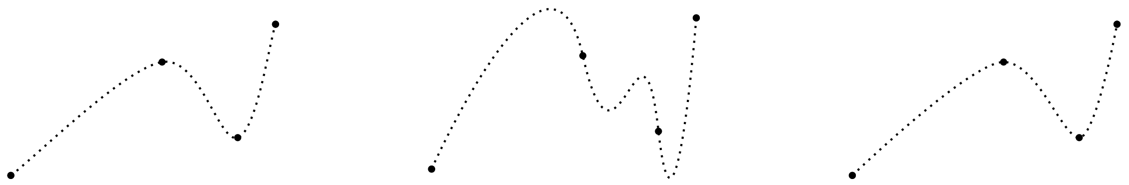
Figure 2: Regular (left), harmonic (middle), harmonic and monotonic (right). While harmonic interpolation solves the overshooting problem of the example in the previous figure, to work properly for all cases, it has to be monotonic.

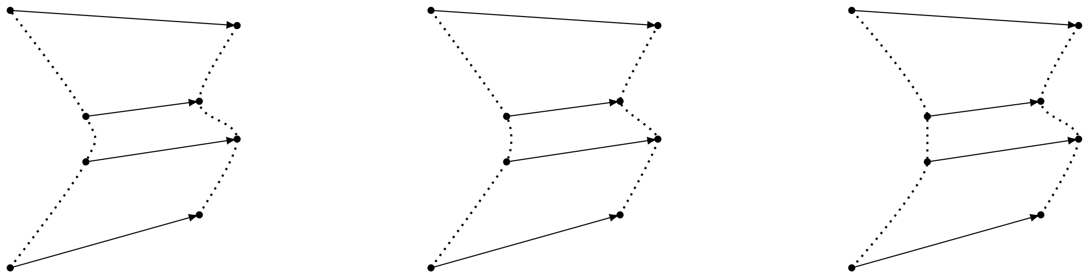
Figure 3: Regular (left), regular with adjusted weights (middle) and the PCHIP equivalent (right).

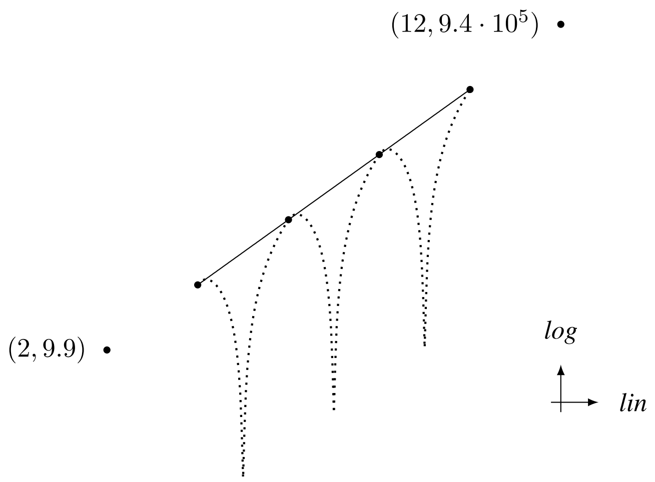

Figure 4: Regular interpolation (dotted) versus LPCHIP (solid) in an even exponential zoom.

## II. Logarithmic Extension

We developed a camera trajectory control system using Apple Xcode [9], based on a PCHIP equivalent during the NASA International Space Apps Challenge 2013, for the production of a video within the Ad Infinitum project on the challenge *Why We Explore*. Although the control system worked perfectly well within local room dimensions, yet the exponential zoom from microcosm to macrocosm showed to work less than satisfactory due to an uneven change of the experienced zooming speed.

This effect is demonstrated in Figures 4-8 by the dotted curves. In Figure 4, the effect is best shown using a regular cubic Hermite interpolant with six breakpoints defined by the function $3.146^x$, which was the largest base with three decimals that could be used before the interpolant caused a singularity. In this context, $x$ represents the linear horizontal axis in Figures 4-10. As shown in Figures 5-8 (dotted curves),

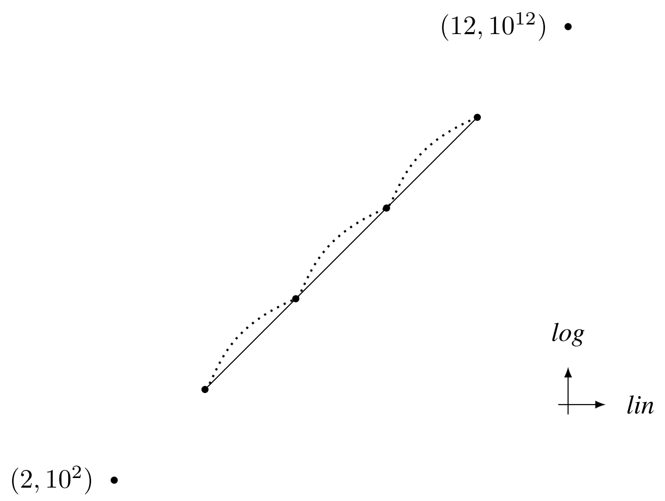

Figure 5: The PCHIP equivalent (dotted) versus LPCHIP (solid) in an even exponential zoom.

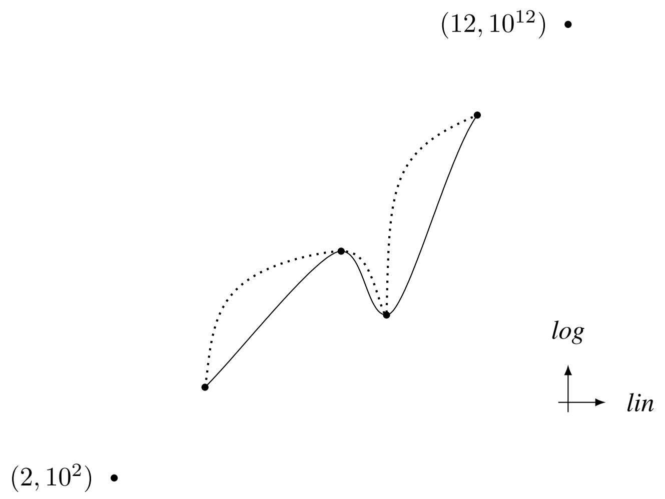

Figure 6: The PCHIP equivalent (dotted) versus LPCHIP (solid) in a dynamic exponential zoom.

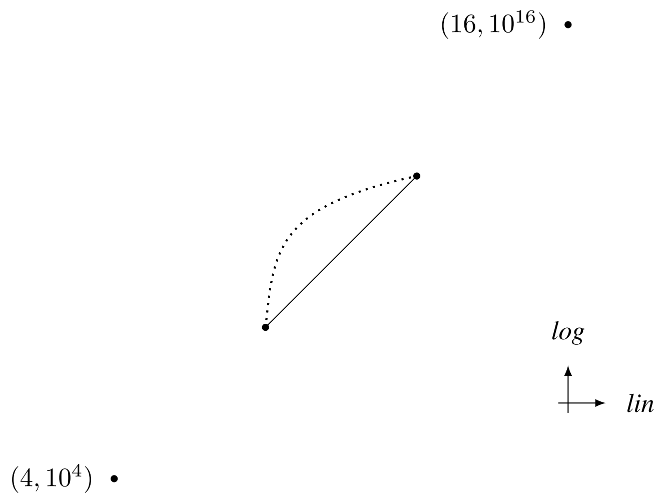

Figure 7: The PCHIP equivalent (dotted) versus LPCHIP (solid) in a moderately scaled exponential zoom.

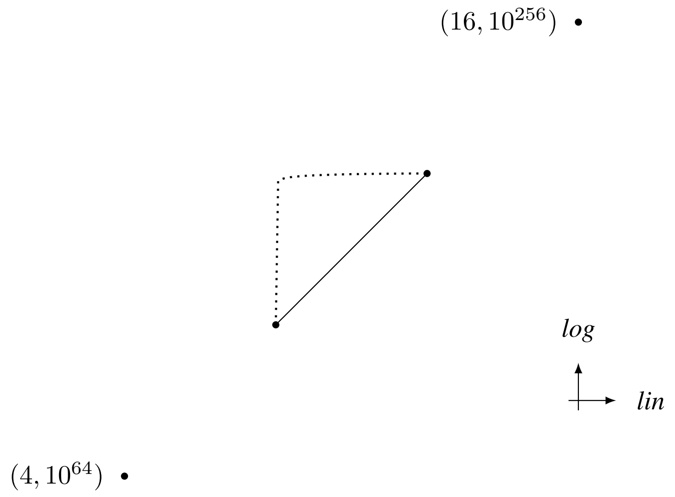

Figure 8: The PCHIP equivalent (dotted) versus LPCHIP (solid) in a large-scale exponential zoom.

where Figure 8 displays a zoom using the function $10^{16x}$, the PCHIP equivalent did not cause any singularities. In this context, it is possible to increase the robustness of the regular interpolant by weight adjustments, although as previously demonstrated at a tangible cost. The conclusion from Figures 7-8 is that it is possible to minimize the oscillation effects of PCHIP (and its equivalent) in exponential camera zooms, if the breakpoints are placed closely enough. The problem is however that the whole idea using interpolation is to eliminate the manual generation of the finer details of a trajectory. Thus, PCHIP (and its equivalent) fails to operate properly if the distance between the breakpoints are exponentially increased.

## III. Results

The solution to this problem showed to be that the interpolant that was implemented in the Ad Infinitum project, had to be redefined to be able to map the distance between the look-at and the camera position into logarithmic space (and back after the interpolation was performed), which is done by setting the LPCHIP argument *lg* to *true* in Figure 12. Thus, the solid lines in Figures 4-8 are obtained, which are identical to the desired trajectory we initially wanted the regular and the PCHIP equivalent interpolants to follow.

This new interpolant is called Logarithmic PCHIP (LPCHIP), a name inspired by the MATLAB PCHIP function. However, in order to work properly, the new interpolant has to be implemented with some caution, since any position value equal or less than zero exceeds the range of the function.

The solution is therefore not to apply LPCHIP (in logarithmic mode) directly on camera trajectory breakpoints, one for each of the six dimensions (three degrees of freedom for the camera position and three for the look-at position), but rather only to interpolate the distance between the camera and the look-at position, since by definition, this distance can never be equal or less than zero. Thus, the arguments $x_0$ to $x_3$ of LPCHIP in Figure 12 do not have to be limited by any safeguards.

Cam_LPCHIP in Figure 13 shows how the new technique is implemented in practice. Briefly expressed, the interpolation is performed the conventional way by separation of the mentioned six degrees of freedom. However, the difference here is that using LPCHIP, the distance between the look-at point and the camera is modified so that it follows a logarithmic trajectory instead of a Euclidean.

In the sample code in C++ that is presented in Figures 12-15 (which in this specific case was assessed to be as clear and succinct as pseudocode for this level of detail, but more straightforward to implement), mCamCoords is a matrix of the type *double* of size mCamCoordsN $\times$ 7, where each row consists of a breakpoint and the first column consists of the time associated with each breakpoint followed by the camera position (columns 2-4) and the look-at position (columns 5-7).

A question in this context is how LPCHIP affects interpolation where the distance between the breakpoints are relatively constant (or more specifically non-exponential). Figure 9 shows that the deviation between the PCHIP equivalent and LPCHIP is in this specific example too small to be visually detectable in this graph. In Figure 10, the difference between the PCHIP equivalent and LPCHIP (in Figure 9) has been magnified, which for this example gives a peak and mean deviation equal to 0.0043 versus 0.0011. This is relatively insignificant and hardly even noticeable for camera trajectory control applications, since the deviation is a smooth curve without any discontinuities.

This example is however only a near best case and in real applications the deviation should be usually quite visible. As an example, in Figure 11, the corresponding average deviation was estimated to 0.14 (or 2.4%), which is a more realistic number. A large number, such as this, is however not necessarily a disadvantage for LPCHIP compared with the PCHIP equivalent but could rather be a measure of the discrepancy of the latter compared with a well-designed interpolant specifically developed for camera trajectory evaluation.

Regarding the evaluation of $t$ in Cam_Auto in Figure 14, to be accurate, a reverse interpolant has to be used. Although such interpolant can be derived symbolically, the solution showed to be relatively complex. This is why a more pragmatic approach was adopted by the application of Newton's method [3], where $h'_k$ for any $k$ denotes the derivative of $h_k$ in:

$$t_{i+1} = t_i - \frac{h_1 x_1 + h_2 x_2 + h_3 m_1 + h_4 m_2 - y}{h'_1 x_1 + h'_2 x_2 + h'_3 m_1 + h'_4 m_2} \qquad (1)$$

The InvPCHIP method in Cam_Auto in Figure 14, takes $y$ as an argument and returns $t$. This method is obtained by the addition of (1) inside a loop after the evaluation of $h_k$ and $h'_k$ in the PCHIP equivalent (with an iteration start value of $t_0 = 0.5$). Figure 11 shows an example of the application of LPCHIP as a function of time, using inverse time interpolation to obtain a smooth trajectory based on six breakpoints using identical start and end-points with totally 500 interpolation line segments. In the example in Figure 11, it took in average 4.08 versus 4.72 iterations to find a solution within an error interval in Newton's method of $10^{-6}$ versus $10^{-9}$. In this case, when the time is measured in seconds, this is equal to accuracy levels in the order of microseconds versus nanoseconds.

Note that for correct performance, the current implementation of this camera trajectory evaluation technique requires a continuously increasing time value along the first column of mCamCoords.

## IV. Conclusion

The new camera control system suggested in this paper showed to exceed current systems used in computer graphics. This new system is categorized by (1) utilization of a local monotonic function of the harmonic mean for the evaluation of the tangents of the piecewise cubic Hermite interpolator (in similarity with PCHIP), in combination with (2) operation in logarithmic space instead of Euclidean regarding the evaluation of the distance between the camera and the look-at point, thereby eliminating trajectory oscillations associated with interpolation of exponential zooms using present techniques.

## Acknowledgments

Many thanks to the NASA International Space Apps Challenge 2013, where the mathematical problem solved in this paper was first identified. In addition, special thanks to Ellinor Dahl at Almi Gotland and Mikael von Dorrien at Almi Väst for the encouragement that led to this work.

## References

[1] C. de Boor, K. Höllig, and M. Sabin, "High accuracy geometric Hermite interpolation", Computer Aided Geometric Design, vol. 4 (1987), no. 4, pp. 269-278.
[2] M. Christie, P. Olivier, and J. Normand, "Camera Control in Computer Graphics", Computer Graphics, vol. 27 (2008), no. 8, pp. 2197-2218.
[3] P. Deuflhard, Newton Methods for Nonlinear Problems: Affine Invariance and Adaptive Algorithms, Springer, 2011.
[4] F. N. Fritch and R. E. Carlson, "Monotone Piecewise Cubic Interpolation", SIAM Journal on Numerical Analysis, vol. 17 (1980), no. 2, pp. 238-246.

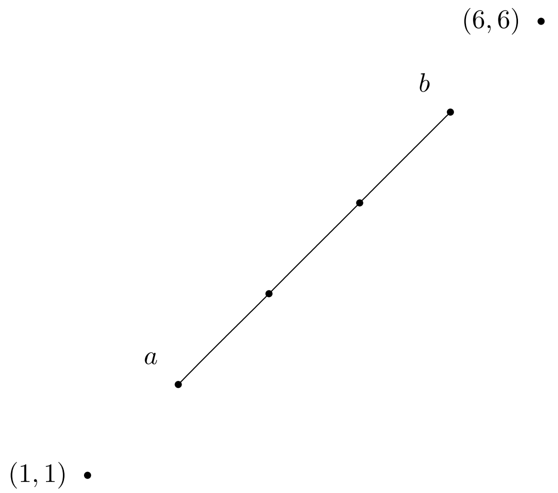

Figure 9: In a non-exponential zoom, the PCHIP equivalent and LPCHIP virtually coincide in this example.

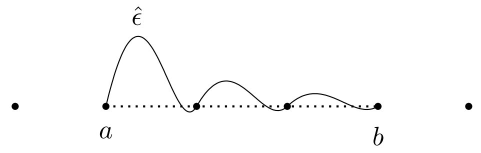

Figure 10: The difference between the PCHIP equivalent and LPCHIP in previous figure, gives a peak value of $\hat{\epsilon} = 0.0043$.

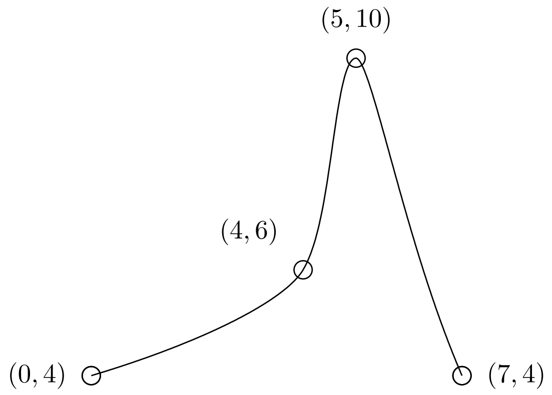

Figure 11: An example of LPCHIP interpolation as a function of time during 7 seconds. To obtain the correct parameter value $t$ in Cam_Auto, an inverse version of the PCHIP equivalent is used.

[5] C. Moler, Numerical Computing with MATLAB, Society for Industrial and Applied Mathematics, 2010.
[6] T. Mullen, Mastering Blender, John Wiley & Sons, 2010.
[7] T. Palamar and E. Keller, Mastering Autodesk Maya 2012, Sybex, Hoboken, NJ, USA, 2011.
[8] H. Smith, Foundation 3ds Max 8: Architectural Visualization, Dreamtech Press, 2007.
[9] Xcode, Apple Inc. <https://developer.apple.com/xcode/> [retrieved: August 7, 2013].

```
double GFX::LPCHIP(double x0, double x1, double x2, double x3,
                   double t, int mode, bool lg){

    if (lg){x0 = log(x0); x1 = log(x1); x2 = log(x2); x3 = log(x3);}

    double epsilon = 1e-20;
    double den, m1, m2, d0, d1, d2, t2, t3, h1, h2, h3, h4, y;
    d0 = x1 - x0; d1 = x2 - x1; d2 = x3 - x2;
    bool a0 = mode == SHAPE_PRES && (d0 * d1 < 0.);
    bool a1 = mode == SHAPE_PRES && (d1 * d2 < 0.);
    bool b = fabs(d1) < epsilon;

    if (mode >= HARMONIC){
        if (a0 || fabs(d0) < epsilon || b ||
            fabs(den = 1./d0 + 1./d1) < epsilon) m1 = 0.;
        else m1 = 2./den;
        if (a1 || b || fabs(d2) < epsilon ||
            fabs(den = 1./d1 + 1./d2) < epsilon) m2 = 0.;
        else m2 = 2./den;
    }
    else {m1 = .5 * (d0 + d1); m2 = .5 * (d1 + d2);}

    t2 = t * t; t3 = t2 * t;
    h1 =  2. * t3 - 3. * t2 + 1.;
    h2 = -2. * t3 + 3. * t2;
    h3 =       t3 - 2. * t2 + t;
    h4 =       t3 -      t2;
    y  = h1 * x1 + h2 * x2 + h3 * m1 + h4 * m2;

    if (lg) return exp(y); return y;
}
```

Figure 12: The LPCHIP interpolant (a pedagogic version), called by Cam_LPCHIP.

```
void GFX::Cam_LPCHIP(int idx, double t, bool lg){
    double X[6];
    For (i,6) X[i] = LPCHIP(mCamCoords[idx-1][i+1],
                            mCamCoords[idx][i+1],
                            mCamCoords[idx+1][i+1],
                            mCamCoords[idx+2][i+1],
                            t,SHAPE_PRES,false);
    if (lg){
        double dx[4],dy[4],dz[4],d[4],dist,eye[3],u[3],factor;
        For (i,4){
            dx[i] = mCamCoords[idx+i-1][1] - mCamCoords[idx+i-1][4];
            dy[i] = mCamCoords[idx+i-1][2] - mCamCoords[idx+i-1][5];
            dz[i] = mCamCoords[idx+i-1][3] - mCamCoords[idx+i-1][6];
        }
        For (i,4) d[i] = sqrt(dx[i]*dx[i]+dy[i]*dy[i]+dz[i]*dz[i]);
        dist = LPCHIP(d[0],d[1],d[2],d[3],t,SHAPE_PRES,true);
        For (i,3){eye[i] = X[i]; mLookAt[i] = X[i+3];}
        For (i,3) u[i] = eye[i] - mLookAt[i];
        factor = dist/sqrt(u[0]*u[0]+u[1]*u[1]+u[2]*u[2]);
        For (i,3) u[i] *= factor;
        For (i,3) mEye[i] = mLookAt[i] + u[i];
    }
    else For (i,3){mEye[i] = X[i]; mLookAt[i] = X[i+3];}
}
```

Figure 13: The LPCHIP camera control method, called by Cam_Auto.

```
void GFX::Cam_Auto(){
    int idx = mCamera_CurrentInterpIdx;
    if (mCamCoords[idx+1][0] < mTime && idx < mCamCoordsN-3)
        mCamera_CurrentInterpIdx = ++idx;
    double t = InvPCHIP(mCamCoords[idx-1][0],
                        mCamCoords[idx][0],
                        mCamCoords[idx+1][0],
                        mCamCoords[idx+2][0],
                        mTime,SHAPE_PRES);
    Cam_LPCHIP(idx,t,true);
    glLoadIdentity();
    gluLookAt(mEye[0], mEye[1], mEye[2],
              mLookAt[0], mLookAt[1], mLookAt[2], 0.0, 1.0, 0.0);
}
```

Figure 14: The main evaluation method, called once for each rendered frame.

```
#define For(i,N)  for (int (i) = 0; (i) < (N); (i)++)
...
class GFX ... {
    enum {REGULAR, HARMONIC, SHAPE_PRES};
    ...
    double mTime;//Time in Seconds
    double mLookAt[3], mEye[3];//Camera Position
    static const int MAX_CAM_COORDS_N = 1024;
    double mCamCoords[MAX_CAM_COORDS_N][7];//Breakpoints (Including Timestamps)
    int    mCamCoordsN;//The Total Number of Breakpoints
    int    mCamera_CurrentInterpIdx;//Current Breakpoint (Start Value = 1)
};
```

Figure 15: A selection of declarations.