Hardware-Accelerated Free-Form Deformation

Clint Chua and Ulrich Neumann

Computer Science Department
Integrated Media Systems Center
University of Southern California

Abstract

Hardware-acceleration for geometric deformation is developed in the framework of an extension to the OpenGL specification. The method requires an addition to the front-end of the OpenGL rendering pipeline and an appropriate OpenGL primitive. Our approach is to implement general geometric deformations so the system supports additional layers of abstraction, including physically based simulations. This approach would support a wide range of users with an accelerated implementation of a well-understood deformation method, reducing the need for software deformation engines and the execution time penalty associated with them.

CR Categories and Subject Descriptors: I.3.1 [Computer Graphics]: Hardware Architecture --- Graphics Processors

Additional Keywords: Free-Form Deformation, OpenGL

1 INTRODUCTION

Many animations are limited to rigid body transformations. While these approximate many ordinary object motions, rigid body animations preclude the class of elastic deformations found in soft or organic objects such cloth or human limbs. In order to achieve these elastic deformations, programmers have had to construct software deformation engines, often coupled to physical simulations such as mass-spring models [11].

Despite the large body of research literature on deformations, interactive systems tend to avoid deformation techniques due to their software implementation cost. Instead, application developers often use approximations based on precomputed key-frames or rigid body animations that hopefully mask any artifacts that arise. Often, in spite of these efforts, seams in the model and inter-penetrations are visible, especially in complex human or animal models. With the continuing increase of rendering quality provided by new graphics cards and gaming consoles [10], it becomes important to consider the addition of high-resolution model deformations that allow single-skin models to deform seamlessly and naturally. Although single-skin deformations are available on a next-generation gaming console, deformation does not have widespread support among most graphics cards or APIs. Deformable single-skin models allow arbitrary interactively controlled deformations not easily approximated by key frames and rigid transformations. The hardware and software API proposed in this paper provides these deformations in the framework of an incremental modification to OpenGL.

Our approach utilizes the well-known Extended Free-Form Deformation (EFFD) method [2]. We show how to integrate the EFFD system into the OpenGL API by extending the current "eval" functions already within OpenGL. This approach yields several benefits. The EFFD method is a widely used and intuitive approach that programmers can easily use for creating and controlling deformable objects. The EFFD deformation operations are simple and regular, therefore allowing efficient implementation as hardware (or optimized firmware) in the OpenGL rendering pipeline.

The remainder of the paper is arranged as follows. The next section provides background on EFFD. Section 3 details the proposed hardware additions. Section 4 discusses several possible hardware organizations. Section 5 describes the OpenGL API functions. The last section summarizes the utility of the system.

2 BACKGROUND

Free Form Deformation (FFD) is a popular technique for creating geometric deformations. There are several types of FFDs, ranging from the standard FFD [8], the EFFD [2], and FFDs with arbitrary topology [4]. We chose to work with the EFFD in our approach since it preserves the mathematical simplicity of the standard FFD yet it covers a wider variety of control lattices.

The intuition behind all types of FFDs is embedding an object into a piece of "Jell-O", and as the Jell-O deforms, the embedded object deforms with it. The different FFDs mentioned above only vary the initial shape of the Jell-O in which the object is embedded. A standard FFD works solely with rectangular parallelepipeds. Arbitrary topology FFDs allow any control lattice. The EFFD, while not as versatile as the arbitrary topology FFD, can work with cylinders, spheres and other regular shapes more complex than a rectangular parallelepiped, while still using the efficient deformation equations of the standard FFD.

All FFD techniques share some common properties. Here is a brief summary of these properties. FFDs can deform nearly any type of geometric model ranging from simple triangle patches to parametric surfaces. It can be applied hierarchically to perform both local and global space deformations. In addition, FFDs can control surface continuity as well as volume preservation. The reader is directed to [2,4,8] for more details on these properties.

* {chua | uneumann}@graphics.usc.edu

EFFD operation is divided into 3 steps. The first step embeds the object in the initial control lattice (Plates 1,2,5,6,9) and computes the parameterized coordinates of the object. The next step moves the control lattice points to new locations, thus deforming the enclosed region of space. The last step calculates the deformed positions of every object point based on the new locations of the control points (Plates 3,4,7,8,10). The deformed object is then ready for rendering.

The embedding process starts with a lattice of regularly spaced control points. For now, we can assume that the control points are arranged in a rectangular parallelepiped. An object is embedded within the lattice by computing a transformation of coordinate systems from the object coordinate system to the local lattice coordinate system. For a rectangular parallelepiped control lattice, the embedding or "freezing" of the object is accomplished by a simple affine coordinate transformation [8].

Since more complex control lattice shapes are possible with the EFFD, the parameterization also becomes more complex, requiring a solution to a system of non-linear equations [2]. This system of non-linear equations is defined by the equations of deformation so we will first introduce these equations and then show how the embedding is done.

Assuming that each object point $X=(x,y,z)$ has a parameterized local coordinate (s,t,u), then with the set of control points P, we calculate the deformed position q with

$$q_{i,j,k}(s,t,u) = \sum_{l,m,n=0}^{3} P_{i+l,j+m,k+n} B_l(s) B_m(t) B_n(u) \qquad (1)$$

where $P_{i,j,k}$ is the i^{th}, j^{th}, k^{th} control point and the Bs are the uniform cubic B-spline blending functions shown below.

$$B_0(u) = \tfrac{1}{6}(1 - 3u + 3u^2 - u^3)$$
$$B_1(u) = \tfrac{1}{6}(4 - 6u^2 + 3u^3)$$
$$B_2(u) = \tfrac{1}{6}(1 + 3u + 3u^2 - 3u^3)$$
$$B_3(u) = \tfrac{1}{6}u^3$$

Thus, given a set of parameterized local coordinates, we evaluate the above equation to get the set of deformed points based on the new positions of the control points.

For non-rectangular lattices, we need to derive the system of non-linear equations to determine the parameterized local coordinates. From equation (1), we can derive the system of non-linear equations for the (s,t,u) parameterization of an object point X. The intuition behind the embedding procedure for the EFFD is that it is the inverse of deformation of the object in the initial control lattice to a rectangular parallelepiped control lattice. Hence, the initial EFFD control lattice shape is constrained by the existence of a mapping or morph between it and a standard rectangular parallelepiped. The morph must not incur any space folding, that is, it must be invertible. From this we can see why the EFFD is not appropriate for lattices of arbitrary topology since the morph from the arbitrary lattice to the rectangular parallelepiped may not exist. To derive the system of non-linear equations, we set the object point X equal to q, the deformed point in Equation (1). The goal then is then to find a parameterization (s,t,u) that satisfies this equation. In order to find (s,t,u), we rearrange the Equation (1) to obtain the following equation.

$$\sum_{l,m,n=0}^{3} P_{i+l,j+m,k+n} B_l(s) B_m(t) B_n(u) - X_{i,j,k} = 0 \qquad (2)$$

where $X_{i,j,k}$ is the object point within the deformable region of $P_{i,j,k}$ to $P_{i+3,j+3,k+3}$. Since this is a 3-D vector equation, we have one equation from each dimension, and 3 unknowns, namely (s,t,u). We use a Newton-Raphson root-finding method [7] with initial guess 0.5 to find (s,t,u).

While the calculation of parameterized coordinates could be done multiple times, this process is typically done only once. In other words, once the initial control lattice has been selected and the parameterization done for it, the control lattice is not changed. In this case, the embedding procedure is done as a pre-processing step for each model and its initial control lattice. At run time, the control lattice points P are moved (between key frames or based on user input) and the deformed model coordinates are evaluated by using Equation (1). It is this evaluation that requires efficiency and hardware acceleration.

3 HARDWARE ARCHITECTURE

3.1 OpenGL System

The hardware architecture illustrates how we accelerate the evaluation of Equation (1). We take advantage of the similarity between the form of Equation (1) and the evaluation of spline-based curves and surfaces already handled by OpenGL. Let us first take a look at OpenGL's curve and surface rendering system.

Fundamentally, OpenGL's curve and surface capability is based on the evaluator system [5]. This system advocates (but does not require) the use of hardware accelerated polynomial evaluators [9]. The evaluations of the currently supported curves and surfaces all take the form of Equation (1). By changing the blending functions, we obtain different families of curves ranging from Bezier and B-spline to Hermite. To simplify the system, OpenGL chose to fix the blending functions to the Bernstein functions. By doing so, hardware implementers are able to optimize their designs for a single polynomial evaluator using the Bernstein blending functions shown below.

$$B_{a,b}(c) = \binom{a}{b}(1-c)^{a-b} c^a$$

The choice of the Bernstein blending functions does not limit OpenGL's capability for rendering other families of curves. In fact, the GLU library fully implements a NURBS renderer on top of OpenGL's evaluator system. This is achieved by finding a transformation between families of curves. In order to represent one type of curve with another, there must exist a mapping between the curve's original type of control points and the resulting family's control points such that the curve represented by both control points are the same. This well-known transformation is described in [3]. Thus, OpenGL can render a wide variety of curves and surfaces by using this transformation.

3.2 Proposed System

OpenGL does not constrain implementations, so the hardware designer can choose how many hardware polynomial evaluators to include. Clearly, a single evaluator is sufficient to increase the

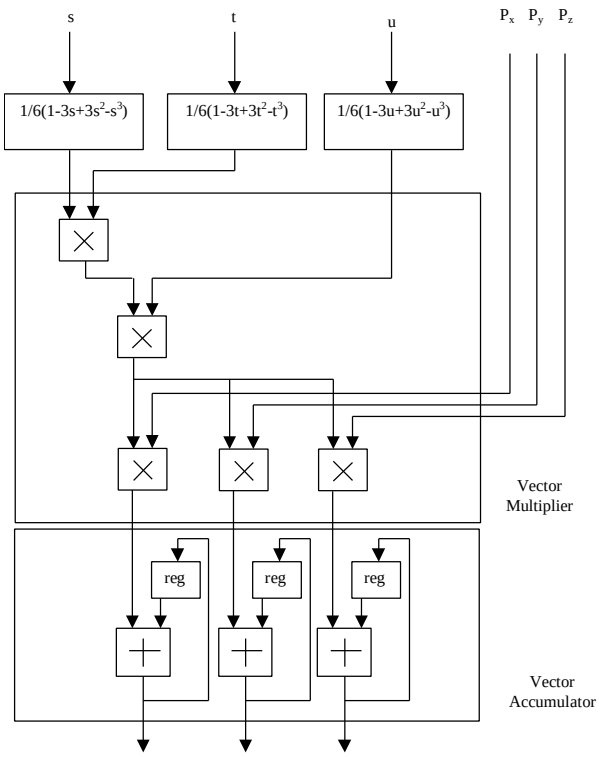

Figure 2. Data Flow of the Pipeline with optional components. This shows how the data flows through the pipeline with the general polynomial evaluators set to the uniform cubic B-spline blending

speed for evaluating Equation (1) but a second evaluator allows parallel evaluations in bivariate hyperpatches (surfaces). This design freedom leads to our proposed architecture, shown in Figure 1

Our conceptual design (Fig. 1) shows a three stage pipelined trivariate EFFD evaluator. The first stage uses three polynomial evaluators to evaluate each blending function in parallel. The next stage is a four-input floating-point vector multiplier that multiplies the results of the blending functions with the control point vector. The last stage is the vector-accumulator that computes the summations for each dimension in Equation (1). A 6-bit counter controls the asynchronous reset of the accumulators after every 64 additions. The system also shows two optional components: the register file for polynomial coefficients and the control point cache. Their functions are discussed in later sections.

The base system without optional components works as follows. First the parameterized coordinates *(s,t,u)* for a model vertex and a control point P are fed into the polynomial evaluator. Once the blending functions are evaluated, all the results are multiplied together and this product is then multiplied with the control point vector. The final vertex coordinate is accumulated for each dimension after every 64 additions. With a pipleline architecture, vertex coordinates are fed into the pipeline continuously and removed at the same rate after a fixed processing latency. These deformed vertices are then passed to the transformation engine for rendering. A summary of the data flow is shown in Figure 2.

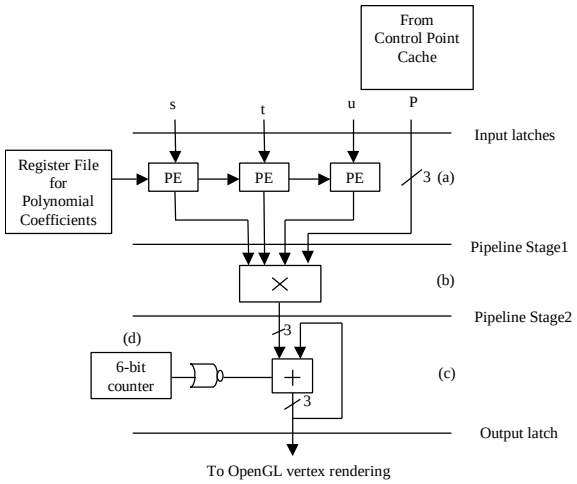

Figure 1. Block-level design of the OpenGL Evaluator Sub-block.
(a) PE refers to a Polynomial Evaluator.
(b) Stage 2 is a floating point vector multiplier
(c) The output stage is a vector accumulator
(d) 2-bit counter's output's go through a NOR gate and is attached to the asynchronous reset of the accumulator's register.

3.3 Other Features

To optimize the throughput of this system, we must push the parameterized coordinates *(s,t,u)* and the control points into the pipeline as fast as possible. The addition of three features will help address this problem.

3.3.1 Control Point Cache

This optional component insures that the control points are available to the pipeline as required. The control points are first loaded into the control point cache prior to sending the parameterized coordinates *(s,t,u)* for any vertices in a model. Once the control points are loaded vertices in the model can be deformed efficiently.

Since the model is embedded in the control lattice as a preprocess, a synchronized sequence of model vertices and control points can be computed and stored for all or any part of a model (like a vertex array) to optimize cache performance and minimize its required size. Although the control point positions change to obtain deformations, their number and neighborhood relationships within the control lattice and the model vertices do not change. It is clearly possible to maintain either separate model and control point arrays or a single unified model and control point array. In either case, cache performance can be statically determined by the degree of array synchronization.

3.3.2 Data Interleaving and Direct Memory Access

Another optimization to evaluating the FFD is DMA. The parameterized object points (and their synchronized control lattice points) can reside in a fixed block of memory. A DMA controller can copy the model and control data directly to the OpenGL pipeline. This can insure that the evaluator pipeline is fully utilized while the CPU is free to execute the application.

One method that takes advantage of DMA is interleaving parameterized local coordinates and their corresponding 64 control points. As data is streamed into the hardware, control points are sent directly to the polynomial evaluator while the control points are sent to the cache (Figure 3). This method is inefficient due to the redundancy of control points in the data stream. Currently, we are investigating caching strategies and optimized arrangements of the data in order to reduce redundancy.

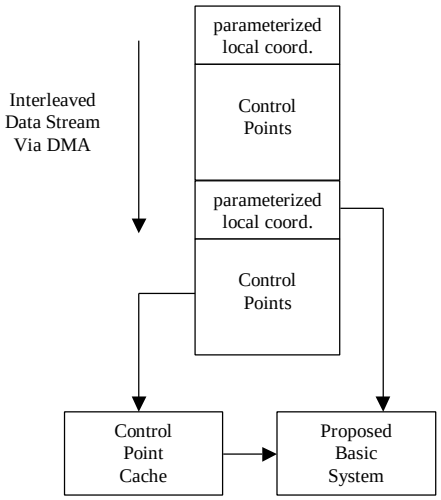

Figure 3: The Interleaved Data Stream. We see how the Control Points associated with each parameterized local coordinate is loaded into the cache while the local coordinate is sent to the proposed system for evaluation.

3.3.3 Polynomial Coefficient Register File

Although predetermining the polynomial evaluator has the advantage of simplifying the hardware design, it forces programmers to convert from one family of curves to another. This additional conversion process could be seen as another point of optimization. Instead of using CPU cycles in computing this transformation, we could add specialized hardware just for this transformation. This is however not optimal since the transformation involves matrix multiplication. Duplicating the existing multiplication hardware is wasteful and rerouting the existing OpenGL matrix multiplication hardware disrupts the pipeline.

By modifying the behavior of the polynomial evaluators in our system, we can bypass the transformation step. The only modification is to implement a more general polynomial evaluator. This means that the polynomial evaluator can evaluate a three-degree polynomial with arbitrary coefficients. Thus, our general polynomial evaluators are of the following form.

$$GPE = ax^3 + bx^2 + cx + d$$

With this general polynomial evaluator, we can change the blending functions by changing the coefficients and degree of the polynomial evaluator. This is where the polynomial coefficient register file is useful. In order to maintain state for the family of curve we are currently rendering, we set the current coefficients and polynomial degree in the register file.

3.4 Further Enhancements

Another possible enhancement is the addition of a combination generator. From Equation (1), we see that the summation goes over all the 64 possible combination of the blending function for each of the three inputs *s*, *t*, and *u*. This means that if we did a straight evaluation of all possible combinations of the blending functions, we would be repeating 189 evaluations. One idea is to generate each blending function with each input *s*, *t*, and *u* and then forward the results to a combination generator that will then feed into the stage 2 multiplier of Figure 2. This introduces additional memory and logic into the pipeline but it is worth considering, as shown in the next section.

Revisiting Equation (1) reveals that the multiply-accumulate operation is used often. It is natural to consider DSP floating-point multiply-accumulate (FMAC) structures. Although not shown here, we can replace the Vector Multiplier and the Vector Accumulator in Figure 3 with FMAC hardware.

3.5 Other Issues

For numerical representation, single-precision floating point suffices, but the efficiency of fixed-point is desirable. The concerns with fixed-point calculations are precision and range. Since the local coordinates *(s,t,u)* all range between 0 and 1, fixed point representations appear feasible. We plan to investigate a fixed-point optimization in the future.

4 DISCUSSION

The system described so far is the core hardware unit. There are several ways to attain better performance. In order to gauge performance, we need to calculate the upper bound time complexity of each hardware organization.

Let *m* be the time it takes to calculate a floating-point multiplication. We can also use *m* as an upper bound on addition and other operations. For the polynomial evaluator, we designed a simple iterative nested-form evaluator (Figure 4). This design comes from the transformation shown below.

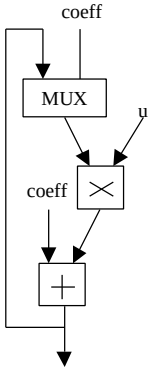

Figure 4: Polynomial Evaluator. Given the parameter *u* and the coefficients of a cubic polynomial, we are able to evaluate the result in 3 loops.

$$ax^3 + bx^2 + cx + d = d + (c + (b + au)u)u)$$

Coefficient *a* is first multiplexed into the loop and multiplied with the parameter *u*. Then the result is added to the next coefficient. The result of the addition is then looped back into the multiplier and the process is continued two more times. The whole process takes 3 loops and performs 2 operations in each loop for a total of *6m* time.

4.1 Basic System

The first approach is to use the core unit by itself. This is the simplest implementation and the slowest. From Figure 2, we see that Stage a takes *6m* time, Stage *b* takes *3m* time, and Stage *c* takes *1m* time. This loops 64 times which gives us a total of 64(6+3+1) = *640m* time.

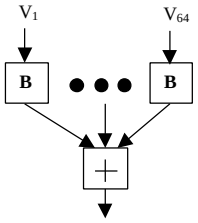

Figure 5: Fully Parallel System. The vectors are fed into a row of 64 parallel units and then their results are all added together by a tree adder. The *B* block is the basic system with out the Vector Accumulator

4.2 Fully Parallel System

On the other extreme, we can parallelize the whole operation by duplicating the basic system 64 times. Then we replace all the Vector Accumulators of each basic system with a single tree adder for each dimension (Figure 5). This calculates all 64 basis function combinations in *9m* time (*6m* for the polynomial evaluator and *3m* for the Vector Multiplier) while the tree adder takes $\log_2(64)$ = *6m* time to add all the 64 results. The fully parallel system takes a total of *15m* time.

4.3 Iterative Tree System

Instead of building all 64 units, we can build n units and loop several times to accumulate the calculation. The organization is the same as the Fully Parallel System (Figure 5) except we replace the tree adder with a tree accumulator. Each unit will take *9m* time to calculate the basis functions and then it will take $\log_2(n)$ for the tree accumulator to add the results. This organization will then loop ceil(64/n) times. The total time for this organization is shown below.

$$time = (9 + \log_2(n))\left\lceil \frac{64}{n} \right\rceil$$

4.4 Basic System with a Combination Generator

In Section 3.4, we pointed out that a lot of the calculations were repeated. In order to enhance the performance of the system, a new hardware unit called the Combination Generator (CG) could be used. This unit (Figure 6) is 3 banks of register files that contain 4 floating-point registers each. The CG stores 12 unique basis function calculations. Once stored, the CG generates all possible combinations and feeds them into the Vector Multiplier. The organization that uses the CG (Fig. 7) is not different from the basic system. We simply insert a CG in between the Polynomial Evaluator and the Vector Multiplier in Figure 1.

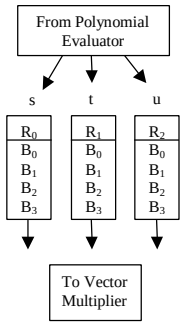

Figure 6: Combination Generator. It is composed of 3 register banks each with 4 locations. The Polynomial evaluator will calculate all 12 values and store the results in the appropriate location. The data is then selected from each bank and sent to the vector multiplier

To load the CG with 12 values, assuming we only have 1 polynomial evaluator, takes 12*6=*72m* time. Then assuming that the CG takes no more than *1m* time to generate a combination, it takes *5m* time (CG + Vector Multiplier + Vector Accumulator) to perform one loop. The total time for this organization is 72+64*5=*392m* time. Compared to the basic system, this amounts to a 39% savings.

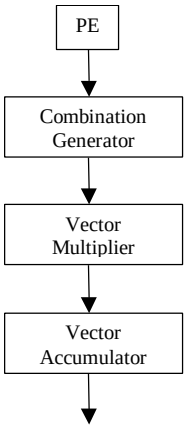

Figure 7: System with Combination Generator. We add in a Combination Generator between the polynomial evaluator and the vector multiplier of the basic system.

To gain an additional 7% savings, we can use three Polynomial Evaluators in parallel. This takes 4*6=*24m* time to fill the CG. The total time using three Polynomial Evaluators only contributes to the time it takes to fill the CG. We get a total of 24+64*5= *344m* time which is a 46% savings.

5 OPENGL COMMAND EXTENSIONS

The OpenGL extensions are described in two variations. The first set are the extensions needed if the general polynomial evaluator is not used while the other set is for use with the general polynomial evaluator.

5.1 Basic System

In the basic system, we need to provide the OpenGL API that allows an application to interface with the hardware without the general polynomial evaluator. Since the GLU libraries already have a well-known interface for rendering NURBS, we utilize the same approach to render FFDs. Below we list the necessary OpenGL API calls.

- gluNewFfdRenderer – generate a new FFD object to refer to when rendering.
- gluFfdProperty – change common properties like line width, etc.
- gluFfdCallback – Callback used to monitor the progress of the rendering. Also used for error monitoring.

- gluFfdVolume – Used to generate the surface based on the control points and the model. This is the main call that accepts the array of parameterized local coordinates and control points. The programmer does not need to explicitly interleave the data but they do need to provide a fourth dimension that indexes into the control point array. The OpenGL implementation can then repackage the data by interleaving parameterized local coordinates with the appropriate control points.

This extension of the GLU library is similar to the GLU NURBS interface. In fact, it is used in exactly the same way [5].

5.2 System with Optional Components

By adding the general polynomial evaluator, we no longer have to work in the GLU library interface. With the general polynomial evaluator hardware, we can access the evaluators directly. Here are the corresponding OpenGL API calls for this system [5].

- glMap3 – similar to glMap2, it is used to specify the array of control points.
- glEvalCoord3 – used to evaluate a parameterized object point. This and glMap3 is used to evaluate a single parameterized local coordinate.
- glBlendCoefficient – used to specify the blending polynomial degree and coefficients.
- glFFD – This call mimics the behavior of gluFFDVolume. It accepts an interleaved stream and sends it directly to the hardware unit.

Notice that similar calls to glMapGrid2 or glEvalMesh2 do not exist. This is because it does not make sense to evaluate a regularly spaced grid since most of the time the coordinates to be evaluated will come from a complex model. An implementation may include them but they would be of limited use.

6 EXAMPLES

All of the models were parameterized within a single cube of deformation consisting of a 64-point control lattice. The cat and the second chair (Plates 9 and 10) were embedded inside the deformable region while the office chair (Plates 5,6,7,8) had its stem and wheels embedded in the deformable region. Most deformations shown here were obtained by rotating the top 4 corner control points by 90 degrees about the y-axis. All other control points are linearly interpolated based on the position of the 8 corner control points. The shearing deformation on the office chair was obtained by lowering the bottom left corner control points while lifting the bottom right corner control points (See Plates 6 and 8).

Plates 1 and 2 show 2 views of a cat being embedded in a control lattice. Plates 3 and 4 show the cat being twisted after the control points are rotated 90 degrees around the Y- axis. Plates 5 and 6 show an office chair being embedded in a control lattice. Plates 7 and 8 show the legs of the office chair being sheared. Plate 9 shows another chair being embedded in a control lattice. Plate 10 shows the chair being twisted.


Plate 1


Plate 2

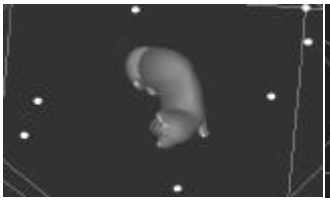
Plate 3

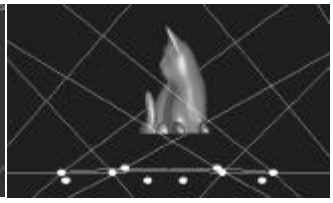
Plate 4

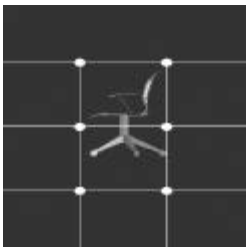
Plate 5


Plate 6

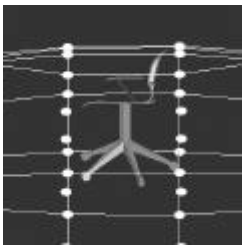
Plate 7

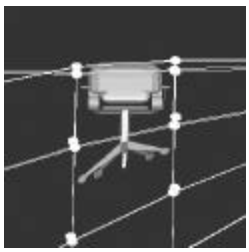
Plate 8


Plate 9


Plate 10

7 CONCLUSIONS

We show how to integrate free form deformation, a popular geometric deformation technique, into OpenGL by adding a few OpenGL API calls. In addition, we propose a block-level design of the OpenGL evaluator sub-system in order to support FFD evaluation in hardware. By implementing these changes, programmers would have access to a standard geometric deformation programming interface as well as a hardware accelerated deformation system.

References

[1] D. Bechmann. Space Deformation Models Survey. *Computers & Graphics* 1994, 18(4), pages 571-586.

[2] S. Coquillart. Extended Free-Form Deformation: A Sculpturing Tool for 3D Geometric Modeling. *SIGGRAPH 1990*, volume 24, pages 187-196.

[3] J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes. Computer Graphics Principles and Practice, 2nd edition in C. Addison-Wesley, 1996, pp. 510-511.

[4] R. MacCracken and K.I. Joy. Free-Form Deformations With Lattices of Arbitrary Topology. *SIGGRAPH 1996*, pages 181-188.

[5] J. Neider, T. Davis, M. Woo. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1. Addison-Wesley, 1993.

[6] OpenGL Architecture Review Board. OpenGL Reference Manual: The Official Reference Document for OpenGL, Release 1. Addison-Wesley, 1993.

[7] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. Numerical Recipes in C: The Art of Scientific Computing. 2nd Edition, Cambridge University Press, 1992, pages 379-383.

[8] T. W. Sederberg and S.R. Parry. Free-Form Deformation of Solid Geometric Models. *SIGGRAPH 1986*, volume 20, pages 151-160.

[9] M. Segal, K. Akeley. The OpenGL Graphics Pipeline. 1993. http://trant.sgi.com/opengl/docs/white_papers/oglGraphSys/opengl.html

[10] Sony Computer Entertainment of America. PlayStation 2 Technical Specifications and Technical Demos. http://www.playstation.com/news/ps2.asp.

[11] A. Witkin and D. Baraff. Physically-Based Modeling. *SIGGRAPH 1997 Course Notes*. Course 19, §§ Differential Equation Basics.