Real-Time Manipulation of Texture-Mapped Surfaces

Masaaki Oka, Kyoya Tsutsui, Akio Ohba
Yoshitaka Kurauchi, Takashi Tago

Information Systems Research Center
Sony Corporation
Asahi-cho, Atsugi-shi 243
Japan

ABSTRACT

A system for real-time texture mapping was constructed. Here, "real-time" means that the system reacts to changes in parameter values which define the shape of surfaces and the viewing point that are given by its operator 30 times per second. This real-time processing enables interactive manipulation of texture-mapped free-form surfaces and various application software has been developed taking advantage of this ability. The system owes its performance to a new algorithm for texture mapping which is based on a newly proposed approximation scheme of mapping functions. In this scheme, a mapping function from the texture plane into the output screen is approximated by a linear function on each of the small regions which form the texture plane altogether. The algorithm is very simple and applicable to any smooth surface. It is especially efficient when implemented by a special-purpose hardware.

1. Introduction

Texture mapping is one of the most popular techniques in computer graphics and several algorithms have been proposed for it. Catmull [2] first introduced texture mapping and developed a subdivision algorithm. Blinn and Newell [1] proposed an algorithm which calculates the inverse image of each pixel on the output screen and assigns the average intensity of the inverse image to the pixel. Catmull and Smith [3] proposed a 2-pass algorithm which works in scanline order and can be implemented by video-rate hardware if the inverse mapping function for the 2nd pass is available. Unfortunately, in order to obtain the inverse mapping function, it is necessary to solve various equations and this difficulty restricts the kinds of surfaces available in these algorithms.

The goal of our research is to provide a means for easy manipulation of texture-mapped surfaces and to make texture mapping useful in such applications as free-form surface design and facial animation production. This requires a fast texture-mapping algorithm applicable to surfaces of arbitrary shapes. In this paper, we introduce a new algorithm which suits such purposes. It is very simple and efficient and is applicable to a wide variety of surfaces. The algorithm approximates a mapping function by a linear function on each of small regions which form the texture plane altogether (locally linear approximation). Its only constraint is that the mapping function be smooth enough. (The meaning of "smooth" is discussed later.) A practical method for anti-aliasing which works well with this algorithm is also proposed.

A real-time texture-mapping system was constructed using the above-mentioned algorithm. The system runs in real time in the sense that it not only transforms 30 images per second but also reacts to changes in transformation parameter values which are given by its operator at the same rate. The system is flexible in the sense that it can deal with free-form surfaces. Due to its real-time processing, the system allows its users to manipulate the shape of texture-mapped surfaces in an interactive manner and various application software has been developed taking advantage of this ability.

In the following section, the texture-mapping algorithm based on a locally linear approximation of mapping functions and the method for anti-aliasing are presented. In Section 3, how they have been integrated into a flexible system is explained. In Section 4, two applications of this system are introduced. In Section 5, some possible improvements are discussed.

2. Locally Linear Approximation

Texture mapping transforms a texture plane, $\{(u,v)\}$, onto a 3D surface, $\{(x',y',z')\}$, and then projects it into the output screen, $\{(x,y)\}$. Let f be the transformation from $\{(u,v)\}$ to $\{(x',y',z')\}$ and p the projection from $\{(x',y',z')\}$ to $\{(x,y)\}$. Then, $g(u,v)=p(f(u,v))$ is a mapping function from the texture plane into the output screen (see Fig.2.1).

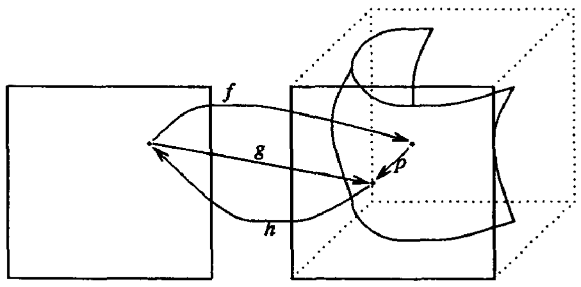


Fig.2.1
Functions in Texture Mapping

Note that the inverse of g, namely h, is not defined for all (x,y) on the output screen. Moreover, even for (x,y) such that $(x,y)=g(u,v)$ for some (u,v), $h(x,y)$ is not uniquely defined. Given (x,y), it is hidden-surface removal that assigns (u,v)-value such that $f(u,v)$ has the maximum z-value (the closest to the viewing point) among the candidates. Once h has been explicitly given, texture mapping is almost trivial, but it is difficult to compute h in general. So here, instead of calculating h, we approximate g using a locally linear function, namely $\hat{g}$, and calculate the inverse of $\hat{g}$, namely $\hat{h}$. Suppose g is in C^3.

$$\begin{bmatrix} x \\ y \end{bmatrix} = g(u,v) = \begin{bmatrix} g^1(u,v) \\ g^2(u,v) \end{bmatrix}. \tag{2.1}$$

We first divide the (u,v)-plane into a 2D array of small square regions with edges of length d. We refer to each of these regions as a "block" in this paper. Let B be one of such blocks and (u_0,v_0) its center. Taking Taylor's expansion of g about (u_0,v_0), we have

$$g^1(u_0+d,v_0) = g^1(u_0,v_0) + d\cdot g^1_u(u_0,v_0) + d^2\cdot g^1_{uu}(u_0,v_0) + O(d^3) \tag{2.2}$$

$$g^1(u_0-d,v_0) = g^1(u_0,v_0) - d\cdot g^1_u(u_0,v_0) + d^2\cdot g^1_{uu}(u_0,v_0) + O(d^3). \tag{2.3}$$

We define

$$a_{11} \equiv \frac{g^1(u_0+d,v_0)-g^1(u_0-d,v_0)}{2d} = g^1_u(u_0,v_0)+O(d^2). \tag{2.4}$$

Similarly,

$$a_{12} \equiv \frac{g^1(u_0,v_0+d)-g^1(u_0,v_0-d)}{2d} = g^1_v(u_0,v_0)+O(d^2) \tag{2.5}$$

$$a_{21} \equiv \frac{g^2(u_0+d,v_0)-g^2(u_0-d,v_0)}{2d} = g^2_u(u_0,v_0)+O(d^2) \tag{2.6}$$

$$a_{22} \equiv \frac{g^2(u_0,v_0+d)-g^2(u_0,v_0-d)}{2d} = g^2_v(u_0,v_0)+O(d^2). \tag{2.7}$$

Then $\hat{g}$, a linear approximation of g on B, is given as follows:

$$\hat{g}(u,v) = \begin{bmatrix} \hat{g}^1(u,v) \\ \hat{g}^2(u,v) \end{bmatrix} = \begin{bmatrix} g^1(u_0,v_0) \\ g^2(u_0,v_0) \end{bmatrix} + \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}, \tag{2.8}$$

where

$$h_1=u-u_0 \quad , \quad h_2=v-v_0.$$

By this linear transformation, a block is transformed onto a parallelogram (see Fig.2.2). Note that the approximation was applied directly to g, a mapping into the output screen, but not to f, a mapping onto the surface.

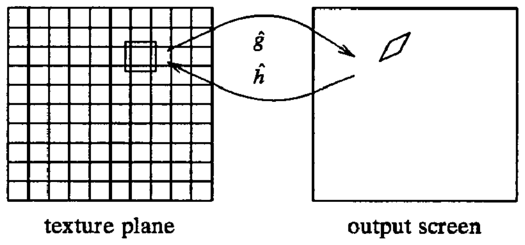


Fig.2.2
Locally Linear Approximation

Now, let's estimate the error caused by this approximation scheme. Since g can be expressed by

$$g(u,v) = \begin{bmatrix} g^1(u_0,v_0) \\ g^2(u_0,v_0) \end{bmatrix} + \begin{bmatrix} g^1_u(u_0,v_0) & g^1_v(u_0,v_0) \\ g^2_u(u_0,v_0) & g^2_v(u_0,v_0) \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} (h_1\frac{\partial}{\partial u}+h_2\frac{\partial}{\partial v})^2 g^1(u_0+\theta_1\cdot h_1, v_0+\theta_1\cdot h_2) \\ (h_1\frac{\partial}{\partial u}+h_2\frac{\partial}{\partial v})^2 g^2(u_0+\theta_2\cdot h_1, v_0+\theta_2\cdot h_2) \end{bmatrix} \tag{2.9}$$

for some $0\le\theta_1,\theta_2\le1$, taking (2.3) through (2.6) and $|h_1|,|h_2|\le d$ into consideration, we have

$$\hat{g}(u,v)-g(u,v) = \frac{1}{2} \begin{bmatrix} (h_1\frac{\partial}{\partial u}+h_2\frac{\partial}{\partial v})^2 g^1(u_0+\theta_1\cdot h_1, v_0+\theta_1\cdot h_2) \\ (h_1\frac{\partial}{\partial u}+h_2\frac{\partial}{\partial v})^2 g^2(u_0+\theta_2\cdot h_1, v_0+\theta_2\cdot h_2) \end{bmatrix} + \begin{bmatrix} O(d^3) \\ O(d^3) \end{bmatrix} = \begin{bmatrix} O(d^2) \\ O(d^2) \end{bmatrix}. \tag{2.10}$$

That is, if we measure the approximation error in terms of length, it is proportional to the square of the length of the edge of a block. This rapid convergence of the error is well illustrated in Fig.2.3, where the finely divided quadrilaterals look almost like parallelograms.

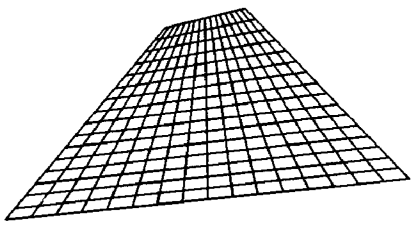


Fig.2.3
Effect of Fine Division

Given a mapping function, although we can reduce the approximation error by increasing the fineness of division, there will always remain some error. This may cause a crack between two adjacent parallelograms, resulting in a serious degradation of the image quality. Fortunately, these cracks can be avoided by setting some common area between two adjacent blocks (see Fig.2.4).

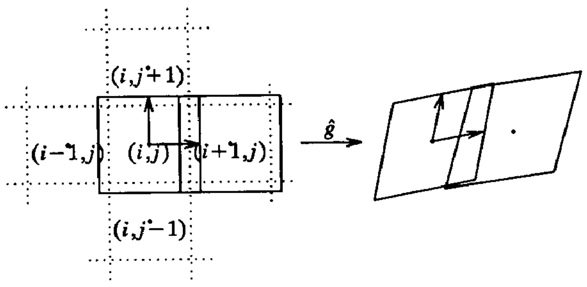


Fig.2.4
Transformation of Blocks with Common Area

Some pixels on the output screen will be accessed more than once, but if the blocks are sufficiently small, the approximation becomes sufficiently accurate and we do not care which access resulted in the final image. As is shown in (2.10), the approximation error is dominated by the second partial derivatives of g. Unfortunately, the values of these derivatives depend on many factors, such as the curvature of the surface and the location of the viewing point, so it is impossible to determine the appropriate block size in terms of surface parameters. It has been shown by experiment that division of a 512×512-pixel texture plane into 8×8-pixel blocks with 0.5-pixel-wide common areas works in most practical applications. As a general aid to understanding, in this paper we refer to mapping functions with reasonably small second partial derivatives as "smooth" mapping functions and surfaces which lead to smooth mapping functions as "smooth surfaces".

The employment of the locally linear approximation described above makes the algorithm for texture mapping very simple. First of all, the inverse of a linear function $\hat{g}$, namely $\hat{h}$, is also a linear function and is obtained by simply calculating the inverse matrix of A, namely $B(b_{ij})$, as follows:

$$\begin{aligned} D &= a_{11}\cdot a_{22} - a_{12}\cdot a_{21} \qquad r = \frac{1}{D} \\ b_{11} &= r\cdot a_{22} \qquad b_{12} = -r\cdot a_{12} \\ b_{21} &= -r\cdot a_{21} \qquad b_{22} = r\cdot a_{11}. \end{aligned} \tag{2.11}$$

This requires only 1 division, 6 multiplications, 1 subtraction and 1 change of sign. Moreover, since h is also a linear function, its own calculations are quite simple. Let $p_0=(x_0,y_0)$ be the center of an output pixel within the parallelogram and q_0 the image of p_0 by h. Let p_{mn} be the output pixel which lies m-pixels to the right of and n-pixels above p_0, and q_{mn} the image of p_{mn} by $\hat{h}$. Then,

$$q_{mn} = q_0 + \begin{bmatrix} m\cdot b_{11} + n\cdot b_{12} \\ m\cdot b_{21} + n\cdot b_{22} \end{bmatrix}. \tag{2.12}$$

(2.12) can be used to calculate $\hat{h}$ in an incremental manner. To calculate $\hat{h}(p_{2m2n})$, for example, we repeat (2.12) twice. Alternatively, (2.12) can be used to calculate $\hat{h}$ for various m and n directly. A combination of these two methods leads to the efficient algorithm shown in Fig.2.5.

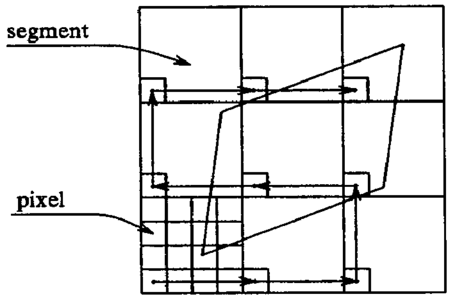


Fig.2.5
Calculation of Inverse Linear Transformation

The output screen is divided into rectangular segments. $\hat{h}$ is calculated by the former method for one pixel in each of the segments which intersect the parallelogram. To the rest of the pixels in each segment, the latter method is applied. Notice that the calculations within a segment can be processed in parallel. Also notice that the second term of the right hand side of (2.12) is a constant for fixed values of m and n. By synthesizing all the transformed blocks based on their z-coordinate, transformation of the whole texture image is completed.

As is well known, assigning to a given pixel on the output screen the intensity value of the texture point which corresponds to the center of the pixel causes serious aliasing problems. A popular method which produces results which are visually acceptable is to assign to the pixel the average intensity value over the texture area that corresponds to the output pixel square (see, for example, [5]). Another method, which was employed by Williams [6] and Crow [4] , is to assign to the pixel the average value of the texture in the minimal rectangle (or square) which covers the area that corresponds to the output pixel. Although this may cause an over-blurred image due to the extraneous areas considered for the average, it simplifies the calculation greatly. Since an output pixel corresponds to a parallelogram defined by the two vectors, (b_{11},b_{21}) and (b_{12},b_{22}) in our algorithm, the minimal rectangle is bounded by edges of length $|b_{11}|+|b_{12}|$ and $|b_{21}|+|b_{22}|$ (see Fig.2.6). Assuming that the center of the parallelogram coincides with the center of a pixel, pre-filtering of the texture image, whose coefficient values vary from one block to another, is possible. Pixels lying near boundary or silhouette edges of the transformed image may partially cover the background and the texture pattern in a remote location. Anti-aliasing along these edges is computationally much more expensive and is not considered in this paper.

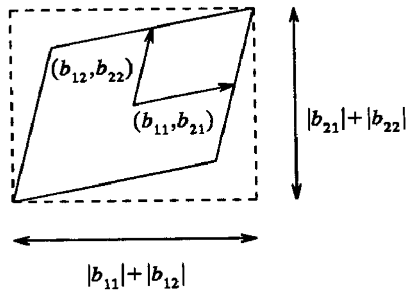


Fig.2.6
Texture Area Corresponding to One Output Pixel

3. Real-Time Texture-Mapping System

The methods described in the previous section were integrated to construct a texture-mapping system. In order to allow easy manipulation of free-form surfaces, the system was designed so that:

[1] Transformation into any smooth surface is possible.

[2] Transformation of texture image is done in real time.

[3] Response to changes in transformation data is made in real time.

Although our algorithm runs fast, it is impossible for any current single processor to satisfy these requirements. Fortunately, processing of texture mapping can be done as shown in Fig.3.1:

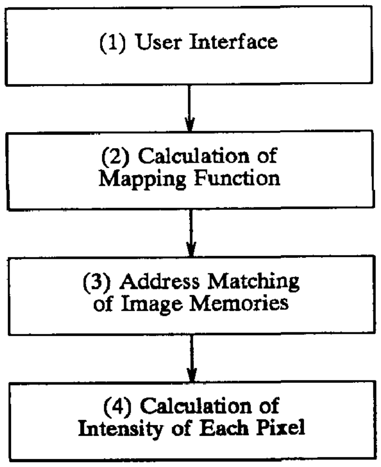

(1) User Interface
(2) Calculation of Mapping Function
(3) Address Matching of Image Memories
(4) Calculation of Intensity of Each Pixel

Fig.3.1
Processing of Texture Mapping

A notable characteristics of this process is that calculations done at an early step are unaffected by subsequent calculations. Making use of this characteristics, we constructed a hierarchical system which consists of software-driven processors and special hardware and implements pipeline processing.

There are several ways to define the shape of a surface, for example by equations and by interpolation of two given surfaces. In any case, the system has to cope with changes in the shape of surfaces. Among the processing steps in Fig.3.1, (1) and (2) require this flexibility and hence, should be controlled by software. For (2), our algorithm calculates the transformation of the central point of each block. Although the number of blocks is much smaller than that of pixels, it is still several thousands, so the processing for (2) has to be much faster than that for (1), which only has to be done once for a frame. In the constructed system, (1) is implemented by a versatile micro-processor and (2) is implemented by parallel micro-coded processors. On the other hand, the processings for (3) and (4) are only linear transformations which are substantially the same for any curved surface. These calculations are simple but as they must be done for each pixel, the system implements them by hardware.

Figure 3.2 shows the actual composition of the system. It has two major data channels, one for transformation data, the other for image data. Host Computer (HC) conducts user interface. It accepts transformation data from input devices and transfers programs and data to Micro-Coded Processors (MP) every frame according to the kind of curved surface. MP first calculates the 3D location of the central point of each block, applies 3D affine transformations to it and projects it onto the screen. Then it generates locally linear approximation data based on (2.4) through (2.7) and (2.11) in the previous section. It also controls hidden-surface removal by sorting the blocks according to the z-coordinate of their central points. This sorting is done by a bucket-sort and a linear list of these blocks is constructed. The data calculated by MP are stored in Block Data Memory (BDM) and transferred to Address Controller (AC) by double-buffering. AC calculates the inverse linear transformation using the algorithm described in the previous section and gives the read address to Input Image Memory (IIM) and write address to Output Image Memory (OIM). AC processes the blocks in the order of the linear list, from the furthest to the nearest. Thus the hidden-surface removal is automatically carried out. Pixel Processor (PP) interpolates the texture data of the nearest 4 pixels in IIM and writes it to OIM. Texture image is filtered for anti-aliasing before it is stored in IIM. Double buffering is also applied to IIM and OIM and one frame of transformed image is output every 33msec.

4. Applications

Various application software which makes use of the real-time processing of this system has been developed. In this section, two of them are introduced.

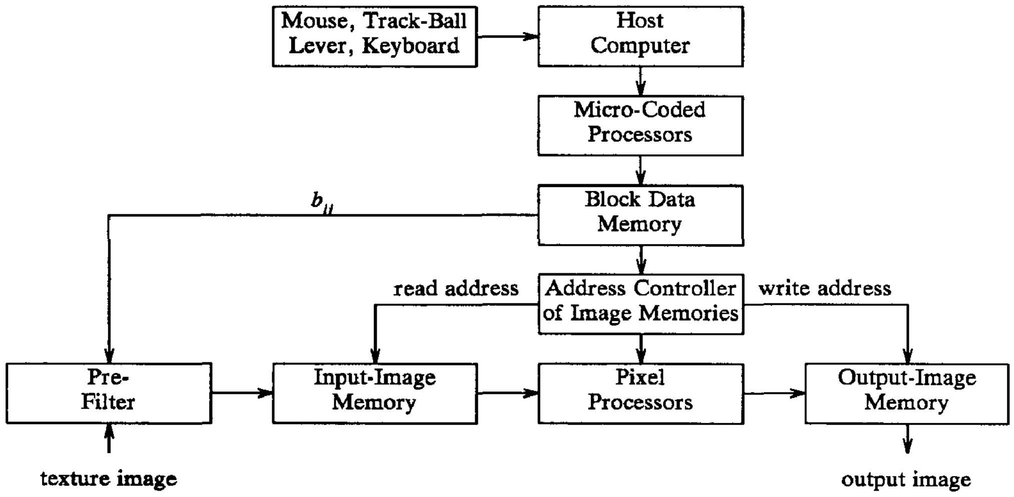


Fig.3.2
System Composition

4.1. Free-Form Surface Design

This application is to design and manipulate free form surfaces such as human faces in an interactive way by locally deforming a displayed curved surface by trial and error and step by step.

A non-deformed plane is taken as the initial shape of the surface and deformations are repeatedly added to it. Let p be a point on the texture plane. At the i-th deformation, p is transformed to a point in a 3D space, namely P_i. P_i is calculated as

$$P_i = P_{i-1} + V_i \cdot F(p, c_i, a_i), \tag{4.1}$$

where V_i is a deformation 3D vector at the i-th step and F is a scalar valued function which regulates the deformation area. F is controlled by two kinds of parameters, c_i and a_i. c_i is the central point of the deformation and a_i is a 2D vector which gives its extent. To restrict the deformation within the neighborhood of c_i, F should have an essentially finite support. For example, the following Gaussian Distribution Function, whose value is nearly 0 outside a certain limit, can be used as F:

$$F(p, c_i, a_i) = \exp\left(-\left(\left(\frac{p_{iu} - c_{iu}}{a_{iu}}\right)^2 + \left(\frac{p_{iv} - c_{iv}}{a_{iv}}\right)^2\right)\right) \tag{4.2}$$

Each point on the surface is moved by the action of $V_i \cdot F(p, c_i, a_i)$, and the surface is locally deformed near c_i. Deformations are repeatedly added to the surface until a satisfactory shape is obtained.

Figure 4.1 shows the surface obtained by adding a deformation to the initial plane. The "+" cursor indicates the central point of deformation and the oval shows its extent. Figure 4.2 is a schematic block diagram of the apparatus of this free-form surface design system. The trackball is used to control the viewing point and the lever and the mouse are used to control the above V_i, c_i and a_i. Figure 4.3 is a flowchart of this free-form surface design. In the Loop-1, a deformation which corresponds to (4.1) is implemented. Thanks to the real-time processing, the user can control the deformation interactively and check it from various viewing points quite easily. In the Loop-2, deformations are repeatedly added and the surface data are renewed. At the same time, deformation parameters such as V_i, c_i and a_i are added to a data list. Since the shape of a surface is independent from the order of the deformations and each deformation is reversible, it is easy to edit this list to modify the shape of the surface. Figure 4.4 is a relief generated from a picture of a face using the above-mentioned method. Fifty deformations were added to generate this surface. Figure 4.5 shows other examples which were generated from the same picture as Fig.4.4, where S_0 was created first and then some modifications were added to it to generate S_1 through S_9.

4.2. Multiple 3D Inbetweening and Real-Time Animation

3D inbetweening, a popular technique in 3D computer animation, interpolates two surfaces to define new surfaces. It is not necessary to restrict the number of surfaces to be interpolated to two if some means of controlling the interpolation is provided. Increasing the number of the surfaces gives more potential variety to the shape of the interpolated surface. So, we extend 3D inbetweening to a multiple 3D inbetweening and apply it to a real-time animation.

In a multiple 3D inbetweening, a weighted sum of several surfaces is calculated as

$$T = \sum_{i=0}^{n} I_i \cdot S_i, \tag{4.3}$$

where S_i's are the given surfaces, I_i's the coefficients and T the interpolated surface. The sum of the coefficient values, I_i's, is 1, but each of them is not necessarily restricted to between 0 and 1. Coefficient values which do not lie between 0 and 1 extrapolate the given surfaces instead of interpolating them.

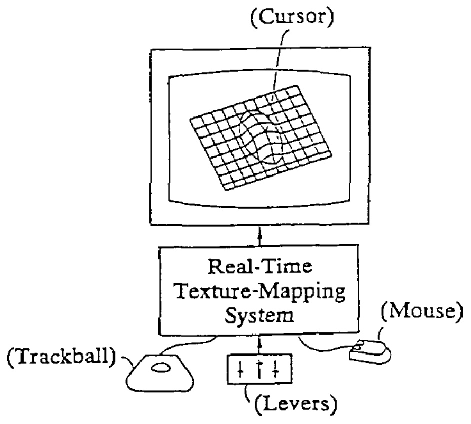
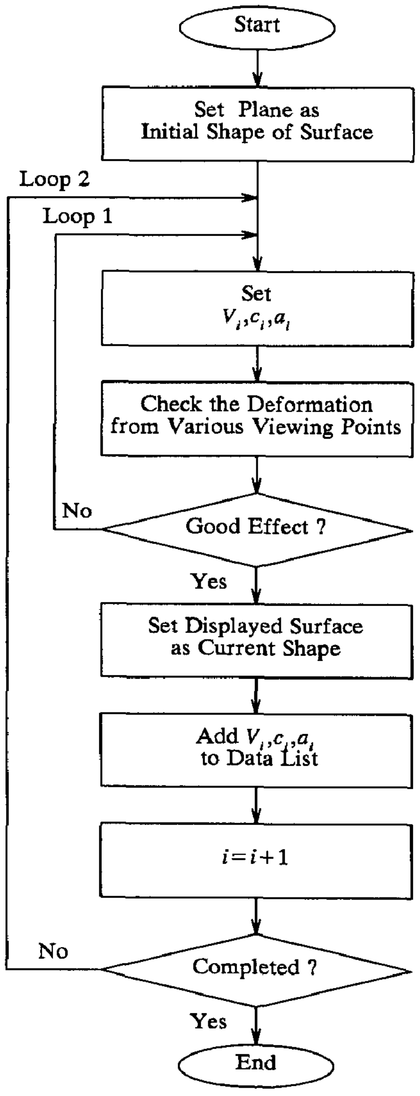


Fig.4.2
Apparatus for Free Form Surface Design


Start
Set Plane as Initial Shape of Surface
Loop 2
Loop 1
Set V_i, c_i, a_i
Check the Deformation from Various Viewing Points
No
Good Effect ?
Yes
Set Displayed Surface as Current Shape
Add V_i, c_i, a_i to Data List
$i = i + 1$
No
Completed ?
Yes
End
Fig.4.3
Flowchart of Free Form Surface Design

It is easy to generate smooth movement of a free-form surface using a multiple 3D inbetweening. A key-frame animation is produced with this system in the following manner. First, the animator selects several surfaces and stores them in the internal memories of MPs. Then, key frames are designated and the coefficients of the multiple 3D inbetweening are given at each of the key frames in an interactive way. Finally, the system interpolates the parameters along the time axis and an animated texture mapping is realized.

Animation of facial expressions proved to be an interesting application of this method and will be quite useful when combined with speech recognition and synthesis. It is possible to synthesize a variety of facial expressions with fewer surfaces if they have been chosen in an efficient way. For this purpose, deformations of different parts of a face should be put into separate surfaces. Reduction of the number of surfaces helps the control of animation as well as saving storage space. The facial expressions in Fig.4.6 were synthesized from the surfaces in Fig.4.5 using the coefficient values shown in the table. Notice that T_0 in Fig.4.6 was generated by extrapolating the original facial relief, S_0, and the one with a closed mouth, S_5.

5. Some Possible Improvements

Although in the interest of simplicity we did not integrate them into our first system, improvements on hidden-surface removal and shading are possible at a rather small expense. As stated in Section 3, hidden-surface removal was realized by sorting the blocks according to the z-coordinate of their central points and processing them from the furthest to the nearest. So, if two blocks intersect with each other or two non-adjacent blocks are positioned too close, this hidden-surface removal does not work correctly. This is avoided by calculating the z-coordinate of each pixel and implementing a z-buffer algorithm. Locally linear approximation in z-direction will be useful for calculating the z-coordinate of each pixel. Gouraud and Phong-type shadings are also possible by linearly approximating the intensity or the normal direction in each block.

6. Conclusions

A real-time texture mapping system was constructed. The system works in real time in the sense that it reacts to changes in parameter values of the surfaces and the viewing point which are interactively given by its operator 30 times per second. The system makes use of a newly proposed algorithm based on a locally linear approximation of mapping functions. The algorithm is simple, runs fast and can be applied to any smooth surface. A practical anti-aliasing method was also proposed and integrated into the system. Real-time texture mapping proved to be very useful for human-machine interface and several applications were introduced.

Acknowledgements

The authors would like to express their appreciation to Mr. Y. Kodera, Mr. H. Yoshida and Mr. M. Morizono of Sony Corporation, for giving them the opportunity to study this subject and the permission to publish this paper. They also wish to thank Mr. H. Kanemaki, Mr. N. Asamizuya and many other colleagues in Sony Corporation who have contributed to the work described in this paper.

References

1. Blinn, James F., Newell, Martin E. Texture and Reflection in Computer Generated Images. Communications of the ACM, 19,10 (October 1976), 542-547.

2. Catmull, Edwin. Computer Display of Curved Surfaces. Proceedings of IEEE Conference on Computer Graphics, Pattern Recognition, and Data Structures (Los Angeles, California, May 1975), 11-17.

3. Catmull, Edwin., Smith, Alvy R. 3-D Transformation of Images in Scanline Order. Proceedings of SIGGRAPH'80. In Computer Graphics 14,3 (July 1980), 279-285.

4. Crow, Franklin C. Summed-Area Tables for Texture Mapping. Proceedings of SIGGRAPH'84 (Minneapolis, Minnesota, July 23-27). In Computer Graphics 18,3 (July 1984), 207-212

5. Glassner, Andrew. Adaptive Precision in Texture Mapping. Proceedings of SIGGRAPH'86 (Dallas, Texas, August 18-22, 1986). In Computer Graphics 20,4 (August 1986), 297-306.

6. Williams, Lance. Pyramidal Parametrics. Proceedings of SIGGRAPH'83 (Detroit, Michigan, July 25-29, 1983). In Computer Graphics 17,3 (July 1983), 1-11.

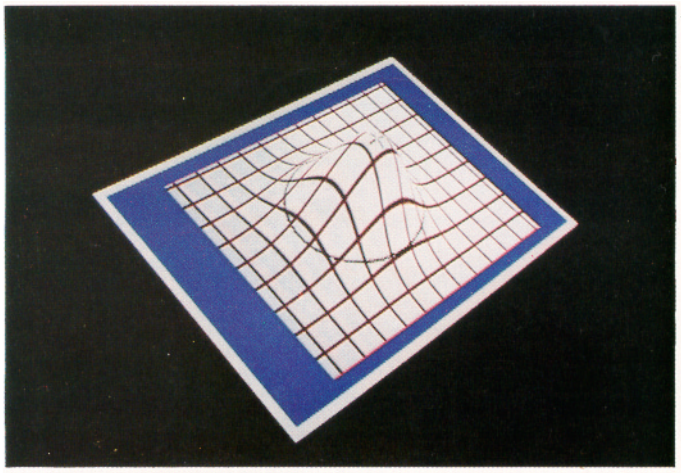


Fig.4.1
Surface with One Deformation

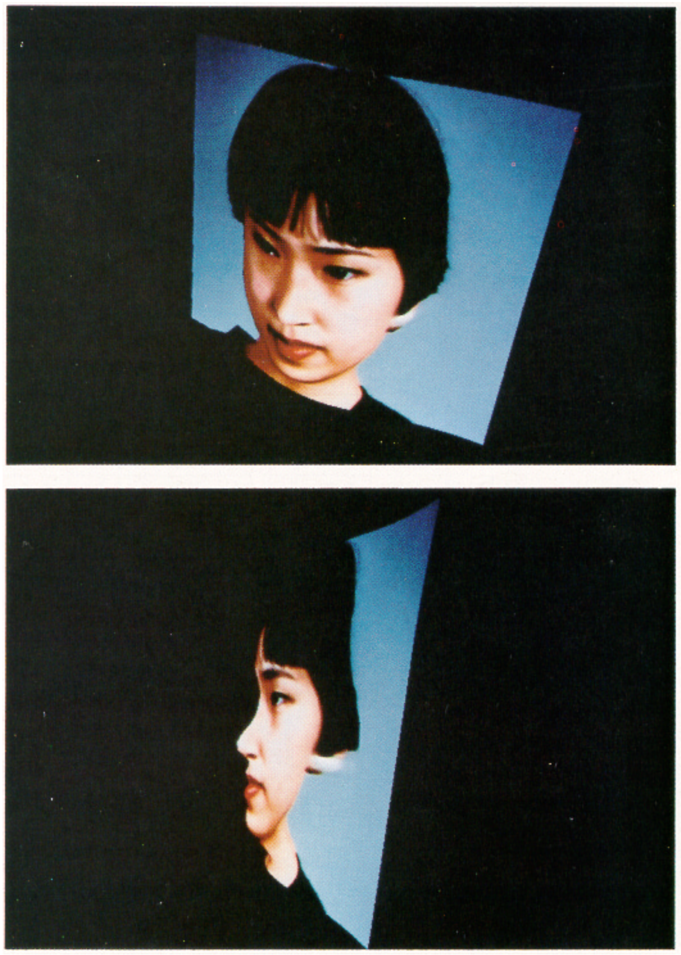


Fig.4.4
Facial Relief

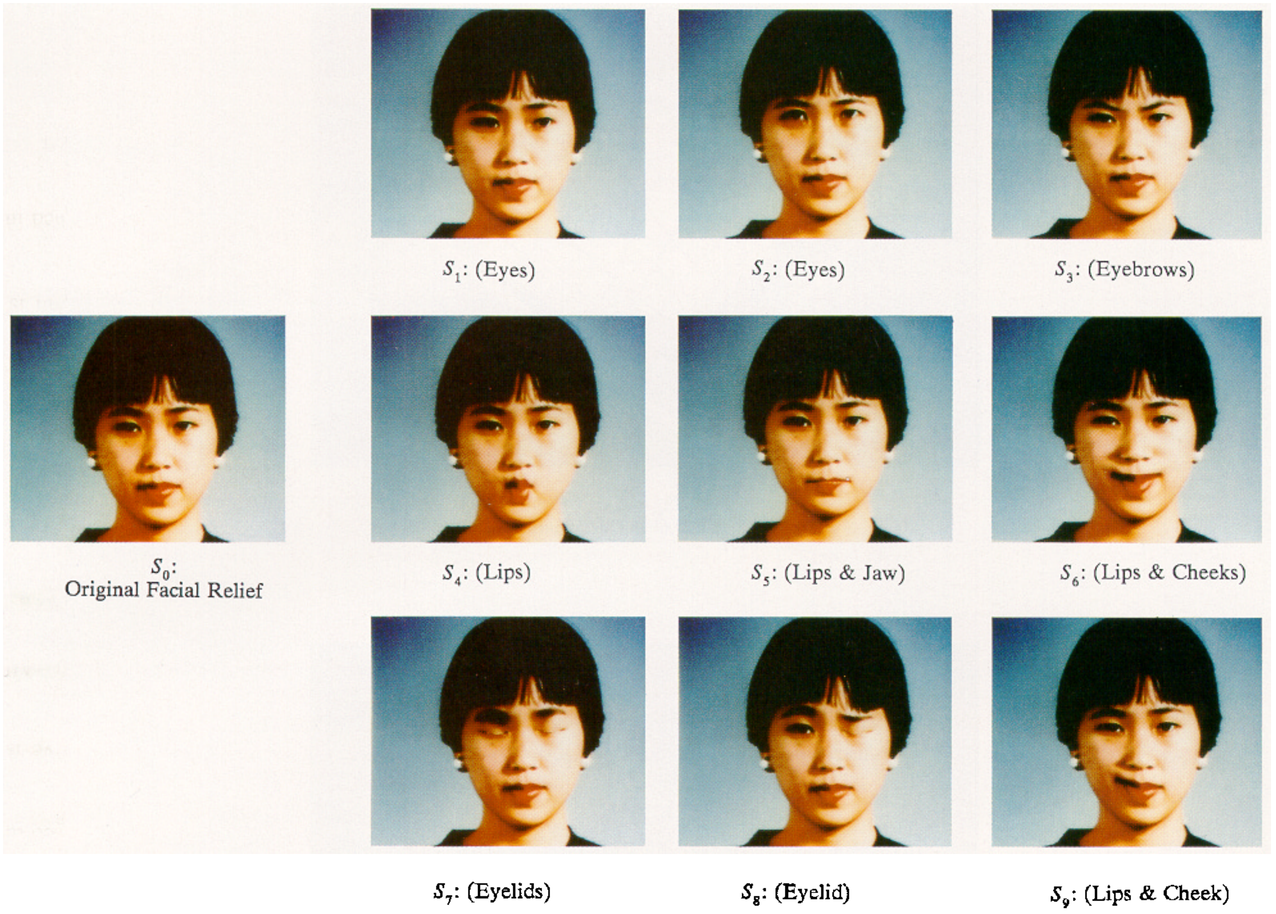
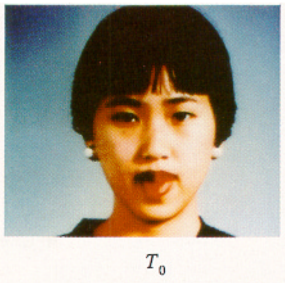


Fig.4.5
Various Facial Relieves



$$T=\sum_{i=0}^{9} I_i \cdot S_i$$

$$\sum_{i=0}^{9} I_i = 1.0$$

	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9
T_0	2.30	0.00	0.00	0.00	0.00	-1.30	0.00	0.00	0.00	0.00
T_1	-0.28	0.26	-0.32	1.30	-0.14	0.26	-0.34	0.26	0.00	0.00
T_2	0.61	-0.55	0.72	-0.68	-0.28	0.52	0.66	0.00	0.00	0.00
T_3	1.74	-0.52	-0.46	-0.68	0.78	0.56	-0.46	0.04	0.00	0.00
T_4	-1.13	0.36	0.00	0.68	-0.08	0.00	0.36	0.77	-0.68	0.72

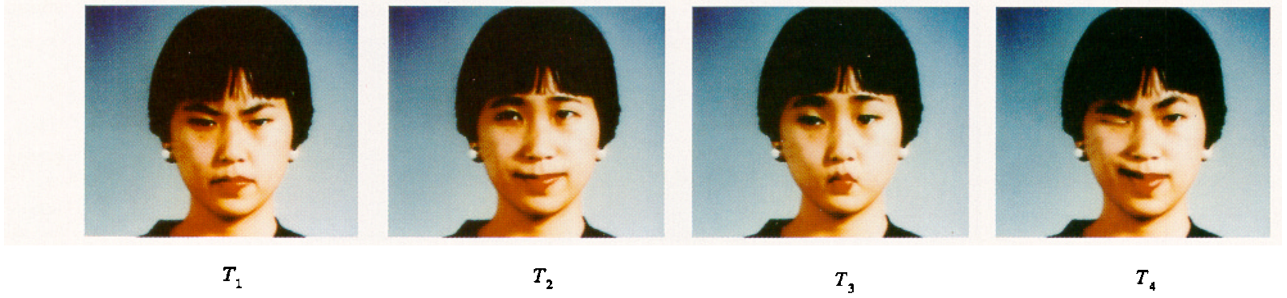


Fig.4.6
Facial Expressions by Multiple 3D Inbetweening