Summed-Area Tables for Texture Mapping

Franklin C. Crow
Computer Sciences Laboratory
Xerox Palo Alto Research Center

Abstract

Texture-map computations can be made tractable through use of precalculated tables which allow computational costs independent of the texture density. The first example of this technique, the "mip" map, uses a set of tables containing successively lower-resolution representations filtered down from the discrete texture function. An alternative method using a single table of values representing the integral over the texture function rather than the function itself may yield superior results at similar cost. The necessary algorithms to support the new technique are explained. Finally, the cost and performance of the new technique is compared to previous techniques.

CR Categories and Subject Headings: I.3.3 [Computer Graphics]: Picture/Image Generation – display algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism – color, shading, shadowing and texture.
General Terms: Algorithms, Performance
Additional Keywords and Phrases: antialiasing, texture mapping, shading algorithms, table lookup algorithms

1.0 Introduction

A frequent criticism of early attempts at realism in computer-synthesized images was that the surfaces lacked interest. At first all surfaces had a dull matte finish. Later surfaces acquired shininess and transparency. However, much of the attraction of real surfaces lies in the incredibly complex local surface variations known as texture. These variations are much too complicated to be modeled by conventional means which require enough vertices or control points to accurately reproduce the surface.

In 1974, Catmull [3] conceived and implemented the first system to use images of texture applied to surfaces to give the affect of actual texture. Blinn and Newell [1] generalized Catmull's work and extended it to include environmental reflections. Blinn [2] then further extended the notion (rather spectacularly!) to achieve the appearance of undulations on the surface (the earlier efforts achieved only flat texture, such as the fake wood texturing found on many plastic desk tops). Carrying things a bit farther, researchers at Ohio State [7] experimented with various expansions of polygonal surfaces to achieve "real" texture. Although some very interesting images resulted, the technique would be too cumbersome for anything very complex.

When texture is mapped onto a surface it must be stretched here and compressed there in order to fit the shape of the surface. 3-D perspective views further distort texture mapped onto a curving surface. As a digital image is synthesized, a pair of texture coordinates must be calculated for each pixel representing a textured surface. The most straightforward implementation of texture mapping simply chooses the pixel from the texture image which lies closest to the computed texture coordinates (the "nearest pixel" algorithm). This works well for a certain class of textures and surfaces.

A frequent example of texture mapping uses a rectangular texture image mapped onto a sphere. Here the compression that each part of the texture image will undergo when mapped is known in advance. The texture can be designed in such a way that it is "pre-stretched" along the top and bottom where it will be mapped near the poles of the sphere. However, unless the texture image is very smooth, with no sharp detail, aliasing becomes an immediate problem. Sharp details will become jagged and the texture will break up where it is highly compressed. Where the mapping is not known in advance, aliasing cannot be controlled just by judiciously designing the texture.

Blinn [2] and later Feibush et al [5] discuss this problem in detail and implemented good, but very expensive solutions. If the pixel being computed is considered a small area, texture coordinates may be computed for the corners of each such area. The pixel intensity is then the average of all texture elements bounded by the corners, weighted by a filter function. In places where the texture is highly compressed (e.g., at the poles of a sphere), this operation may require a weighted sum of hundreds of texture values.

Catmull and Smith [4] show a way of simplifying the calculation of the texture intensity by separating the convolution into two passes. The method was initially applied just to represent transformed images on a raster. A horizontal pass over the texture is followed by a vertical pass, producing texture values as they should appear in the image. The simplicity of the process makes it amenable to hardware implementation; a similar technique is currently very much in vogue for special-effects in television. However, where the texture is highly compressed, many texture pixels must still be processed to yield a single image pixel.

Norton, Rockwood and Skomolski [9] report a method for limiting texture detail to the appropriate level by expressing

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1984 ACM 0-89791-138-5/84/007/0207 $00.75

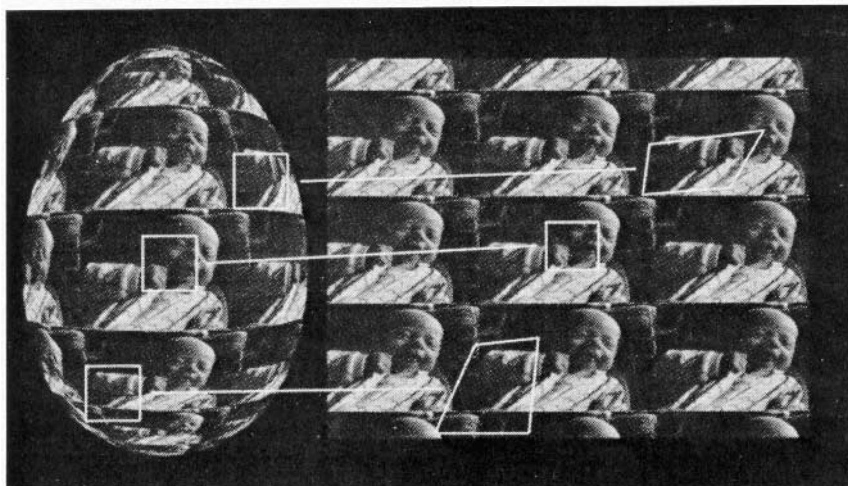

Figure 1: Texture distortion under mapping.

texture as a sum of band-limited terms of increasing frequencies. Where the frequency of a term (i.e., its level of detail) exceeds the pixel frequency, that term is "clamped" (forced to the local average value for the term). The method has been applied, in a very restricted way, but with excellent effect, in a real-time visual system for flight simulators. In order to use this method the texture must be divided into terms using Fourier analysis or similar techniques. Alternatively, the texture may be synthesized from Fourier terms.

In a remotely similar vein, Haruyama and Barsky [8] describe implementation of a fractal texture synthesis technique as suggested by Fournier, Fussell, and Carpenter [6]. Fractal synthesis has the advantage that the level of detail is controlled very naturally. This makes antialiasing easier as the texture is compressed on the surface. However, we are still left with no way to handle sharp detail in a texture.

Williams [10], some time ago, conceived and directed the implementation of a very clever algorithm which extends practical texture mapping to a much, much larger class of textures. Instead of using a single texture image, many images at varying resolution are derived from the original by averaging down to lower resolutions. Thus, in a lower resolution version of the texture, each pixel represents the average of some number of pixels in the higher resolution version. Since only a limited number of tables may be stored, values from two adjacent tables must be blended to avoid obvious differences between areas of texture represented at different resolutions. Now where highly compressed texture must be dealt with, computation need only determine which tables to address. Texture computation can be more or less constant over all pixels.

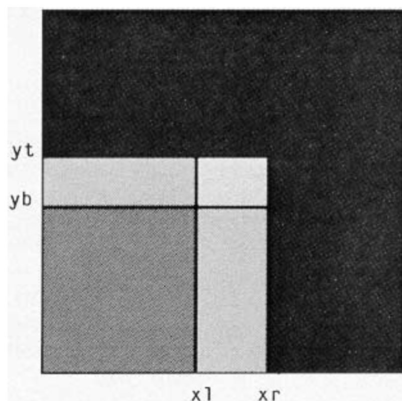

Figure 2: Calculation of summed area from table.

Williams calls his technique "mip" mapping (for "multum in parvo", Latin for "many things in a small place"). Mip mapping achieves speed at the expense of some accuracy by assuming that texture intensity at any pixel can be adequately represented by the average over a square region of texture. Square regions assume that texture compression is symmetric. However, where a surface curves away from the viewer, texture may be compressed along only one dimension (figure 1). Since table addressing must be based on the axis of maximum compression, mip mapped texture may appear fuzzier than would otherwise be necessary.

A generalization of Williams' technique can provide a better approximation to the proper texture intensity by allowing rectangular regions of texture to be used. A single table of much larger numbers is used, from which a virtually continuous range of texture densities may be drawn.

2.0 Using A Table Of Summed Areas For Texture Mapping

2.1 The Basic Technique

Mip mapping can be done using a single table in which each texture intensity is replaced by a value representing the sum of the intensities of all pixels contained in the rectangle defined by the pixel of interest and the lower left corner of the texture image. The sum of all intensities in any rectangular area of the texture may easily be recovered from such a table. Dividing the sum by the area of the rectangle gives the average intensity over the rectangle.

To find the sum of intensities over an arbitrary rectangle, it is sufficient to take a sum and two differences of values from the table. As an example assume that we want the sum over an area bounded by xl on the left, xr on the right, yb on the bottom, and yt on the top (figure 2). The sum is given by:

T[xr,yt] - T[xr,yb] - T[xl,yt] + T[xl,yb]
where T[x,y] is the value in the table addressed by the coordinate pair (x,y).

In other words, starting with the table entry at the upper right corner of the area, subtract first the table entry at the lower right then the entry at the upper left. This removes all area below the rectangle of interest then all area to the left. Note that the area lying below *and* to the left of the rectangle has been subtracted twice. This area must be restored by adding back in the table entry at the lower left of the rectangle.

The best approximation to the proper texture intensity would be calculated by multiplying by a filter function superposed over the texture rectangle before summing the intensities. Unfortunately, since the sums must be precomputed, only a constant filter function is possible. However, very convincingly antialiased images are routinely made under this restriction.

As with the multiple table mip map described above, it is necessary to interpolate between table entries to smoothly represent sharp edges when the texture is not greatly compressed (or when the texture is expanded). Furthermore, if texture values are taken only at pixel locations, the resulting discretized mapping will sometimes cause jittering as the surface moves. More accurate mapping may be obtained by allowing the corners of a rectangular region to lie between texture pixels. Therefore, the summed area at each corner of the rectangle must be calculated by interpolating from four values in the table. Once the corner values are found, the computation proceeds as above.

2.2 Comparisons with the Multiple Table Mip Map

To get a rough idea of the relative expense of the summed area table mip map versus the multiple table mip map, let us count the necessary arithmetic operations and texture accesses. This will give us an approximation of the processing power and bandwidth to the texture memory required. Texture memory bandwidth is important when the texture is stored in a frame buffer, where access may be slow, or in virtual memory, where many accesses may substantially increase the short-term working set size, causing excess page swapping.

For both methods, most of the cost goes into linear interpolations. A linear interpolation costs two multiplicative operations and two additive operations if done as:
b * (1 - alpha) + c * alpha
or, better, two additive operations and one multiplicative operation if done as:
b + (c - b) * alpha,
where alpha is used to interpolate between b and c. A bilinear interpolation, interpolating to a value lying within a region defined by four adjacent pixels, requires three simple interpolations.

The multiple map method requires two bilinear interpolations to get a value from each of two adjacent tables, plus an additional interpolation between the two values. This requires a total of 8 texture accesses, 7 multiplicative operations, and 14 additive operations. The summed area table method requires four bilinear interpolations to get the four corners of a rectangle, then three additive operations to get the sum over the rectangle, then a multiply to get the area and a divide to find the average. That adds up to a total of 16 texture accesses, 14 multiplicative operations and 27 additive operations.

There is an optimization for the summed area method where the texture is highly compressed. In such cases the large size of the rectangular region from the summed area table makes the effect of interpolation at the corners negligible. Without interpolation, the cost of the summed area method reduces to 4 texture accesses, two multiplicative operations and 3 additive operations.

Of course, the above is concerned only with finding the value for the texture intensity once the texture coordinates and the size of the texture area are known. For the multiple table approach, the two tables must be selected and the value of "d" (used to interpolate between the tables) calculated [10]. For the summed area table method, increments giving the texture coordinates at the adjacent pixels may be used to define the corners of the rectangular area. In both cases, the necessary computation appears small next to the texture intensity calculation.

2.3 Calculating Texture Coordinates

To be more specific, the summed area table code which produced figures herein calculates the texture rectangle as follows: A given scan segment is generated by linear interpolation of the state at its endpoints. The endpoints of a set of scan segments representing a portion of surface are generated by linear interpolation between the endpoints of the top and bottom segments. Since the texture coordinates are included in the endpoint information, they are linearly interpolated along with everything else.

The texture coordinate at a pixel is calculated by an incremental bilinear interpolation; at each pixel, the texture coordinates are found by adding increments to the coordinates from the previous pixel. Those increments, which are constant over a scan segment, are used to partially determine the texture rectangle. Let's call them the horizontal increments.

The other necessary information is the pair of increments needed to get the texture coordinates at the corresponding pixel on the next scanline. Increments are kept which allow incremental interpolation of the texture coordinates of the endpoints of one scan segment from those of the previous one. Using those increments at each end of a scan segment, a pair of vertical texture coordinate increments can be computed by incremental interpolation between the vertical increments at the scan segment ends.

Given both horizontal and vertical texture increments at a pixel the texture rectangle is determined by taking the maximum of the absolute values of both x-coordinates and similarly for the y-coordinates. This gives a sort of bounding box on the true texture area which works quite well. Since a rough approximation to the true texture area is all that can be achieved using the summed area table, a more efficient determination of the rectangle may be possible. However this computation is only a small part of the total; any improvements will only marginally effect total running time.

2.4 Handling Texture Image Boundaries

Since a texture map may be replicated many times over a surface, or may cover only part of a surface, it is important to properly handle the case where a pixel contains a texture image boundary. In the case where a surface is only partially covered by texture, it is sufficient to truncate texture image

coordinates while calculating the area from the unmodified coordinates.

Where a texture is replicated many times over a surface, values lying to the right or above a boundary must be increased by the value at the boundary. In extreme cases, a pixel may contain several boundaries, implying that several whole texture images are mapped into one pixel. This case would require several additions to arrive at the proper values for the right and upper corners of the rectangle.

3.0 Building And Storing A Summed Area Table

Computing the values to be stored in a summed area table is quite simple. A table can be generated at an arithmetic cost of two adds per entry. The most straightforward method would be to invert the method used for taking summed areas from the table. To get a table entry: Add the pixel intensity to the sum at the pixel below plus the sum at the pixel to the left. Doing this counts the sum at the pixel below and to the left twice. Therefore, that sum must then be subtracted. The arithmetic cost is three additive operations.

The table can be built with only two additive operations per entry by maintaining a sum of intensities along a scanline. Using this method, a table entry is calculated by adding the pixel intensity to the sum for the scanline then adding that to the sum at the pixel below. Generating the table is inexpensive enough (for reasonably-sized texture images) that it should not be an important consideration in deciding whether to use a mip map technique or the more accurate (and expensive) techniques of Blinn [2] and then Feibush et al. [5].

A potential disadvantage for the summed area table is that it requires many more bits per entry than there are bits per texture intensity. If the texture intensity is stored in 8 bits, then a 1024 by 1024 entry table could require entries as long as 28 bits. A table could be built with as little as 24 bits per entry by restricting texture images to 256 by 256 pixels. However, most machines handle 32-bit words more gracefully than 24-bit words, so why restrict ourselves?

Various bit-saving techniques may be concocted to reduce the number of bits per entry to 16 or less. For example, the texture image may be divided into regions of 16 pixels square to limit the sum within such a region to 16 bits (256 entries of 8 bits each). A 32-bit quantity would be stored for each region giving the sum at its lower left corner. To recover a value from the table would require adding the appropriate 32-bit quantity to the table entry. Trying to reduce the number of bits per entry to 12 yields diminishing returns. Storing a 32-bit quantity for each group of 16 entries involves an overhead of 2 bits per entry, for an effective 14 bits per entry.

It must be noted that the multiple table mip map method does considerably better in terms of required storage. The number of bits per texture pixel is increased by only one-third in preparing the table, as opposed to a factor of from two to four for the summed area table.

4.0 Conclusions

As can be seen from figures 3-7, the summed area table works well for antialiasing mapped texture. The egg-shaped surfaces are polygonal approximations, which causes the apparent creases in the texture patterns. The examples used here were deliberately chosen to try to show any inadequacies in the texturing techniques. Ideally the texture should roll off smoothly into a uniform grey at the ends of the striped eggs in figure 3. The more accurate renditions afforded by more expensive means [2, 5] may do better in such situations. However, nearly all images are more forgiving than the examples used here. Such differences are most often not visible.

A trial implementation of the multiple table mip map method has yielded inconclusive results. Both the summed table and multiple table methods roughly doubled the time needed to compute an image. My decidedly non-optimized implementation of the multiple table method runs about ten percent slower than my implementation of the summed area method, which would appear to contradict the implications of section 2.2 above. Since neither implementation has been subjected to careful scrutiny for bottlenecks, however, speed comparisons must be considered inconclusive.

The images in figures 3-6 appear to show that the summed area method offers some superiority in image quality over the multiple table method. However, I would prefer independent confirmation of that result. Both methods offer ample opportunities for tuning. Furthermore, the multiple table method has not really reached its potential as yet. Both Williams' and my implementations use tables which are generated using unweighted averages. Tables generated using proper filtering techniques could well yield better results. On the other hand, the summed area approach may well have extensions allowing the use of better filters.

It should be pointed out that the general notion of recovering the integral over a rectangular region of a function of two variables undoubtedly has broader application than shown here. I know of no other applications as yet, but I believe that they must exist.

This work was made possible and pleasurable by the incomparable facilities of the Xerox Palo Alto Research Center and my colleagues there in imaging. All text, figures and code development were done on a Dorado personal workstation using the Cedar programming environment.

References

1. Blinn, J. and Newell, M., "Texture and Reflection on Computer Generated Images", *Communications of the ACM*, Vol. 19, #10, Oct. 1976.
2. Blinn, J., "Computer Display of Curved Surfaces", PhD. Dissertation, Department of Computer Science, University of Utah, December 1978.
3. Catmull, E., "A Subdivision Algorithm for Computer Display of Curved Surfaces", PhD. Dissertation, Department of Computer Science, University of Utah, Tech. Report UTEC-CSc-74-133, December 1974.
4. Catmull, E. and Smith A. R., "3-D Transformation of Images in Scanline Order", *Computer Graphics* (Proc. Siggraph '80), Vol. 14, July 1980.

5. Feibush, E. A., Levoy, M., and Cook, R. L., "Synthetic Texturing Using Digital Filters", *Computer Graphics* (Proc. Siggraph '80), Vol. 14, July 1980.
6. Fournier, A., Fussell, D., and Carpenter, L., "Computer Rendering of Stochastic Models", *Communications of the ACM*, Vol. 25, #6, June 1982.
7. Hackathorn, R. and Parent, R., Private Communication, 1980.
8. Haruyama, S. and Barsky, B. A., "Using Stochastic Modeling for Texture Generation", IEEE *Computer Graphics and Applications*, Vol. 4, # 3, March 1984.
9. Norton, A., Rockwood, A. P., and Skomolski, P. S., "Clamping: A Method of Antialiasing Textured Surfaces by Bandwidth Limiting in Object Space", *Computer Graphics* (Proc. Siggraph '82), Vol. 16, #3, July 1982.
10. Williams, L., "Pyramidal Parametrics", *Computer Graphics*, Vol. 17, #3, July 1983.

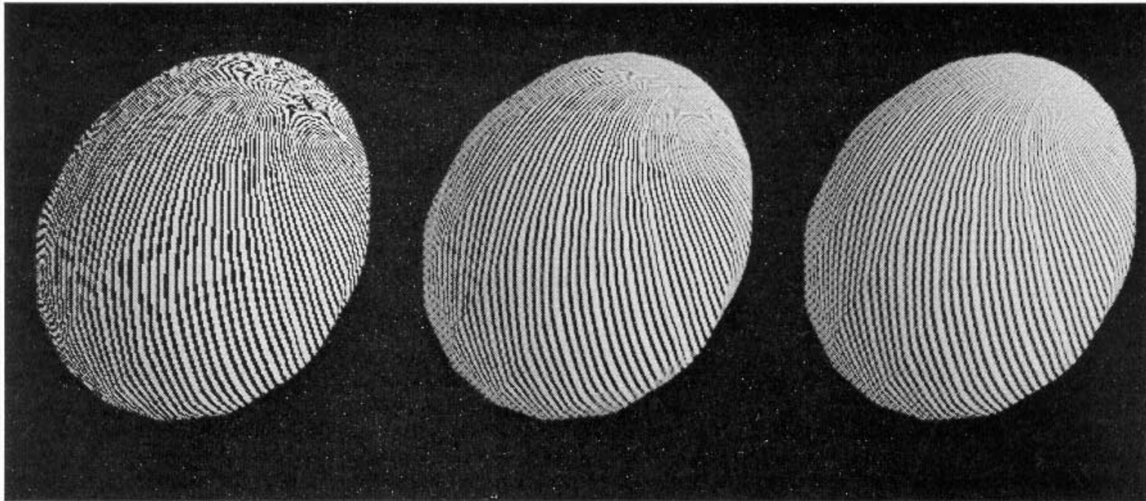

Figure 3: Left: nearest pixel (1 min. CPU time), middle: multiple table (2 1/4 min.), right: summed table (2 min.).

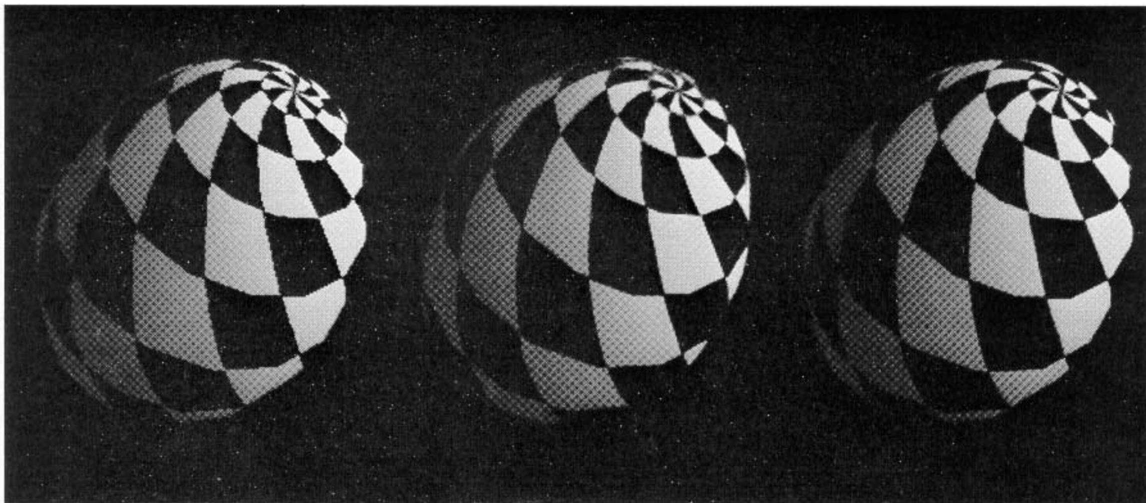

Figure 4: Left: nearest pixel, middle: multiple table, right: summed table.

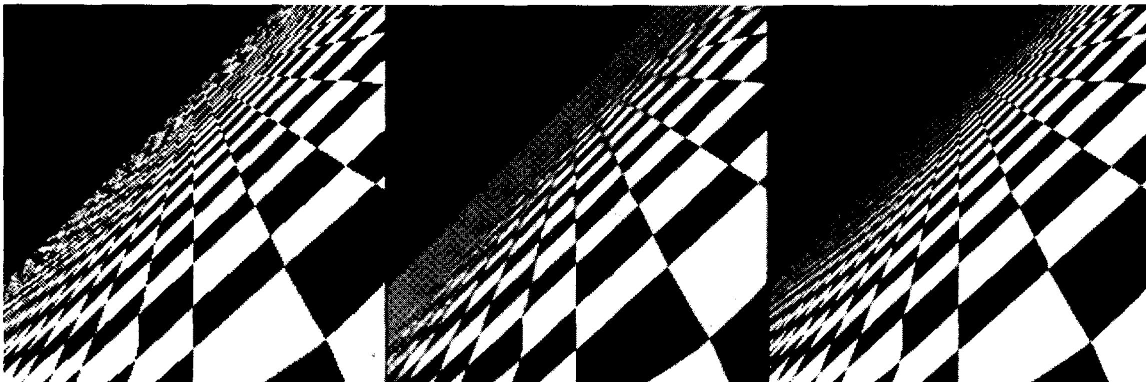

Figure 6: CheckerBoards showing laterally compressed texture.

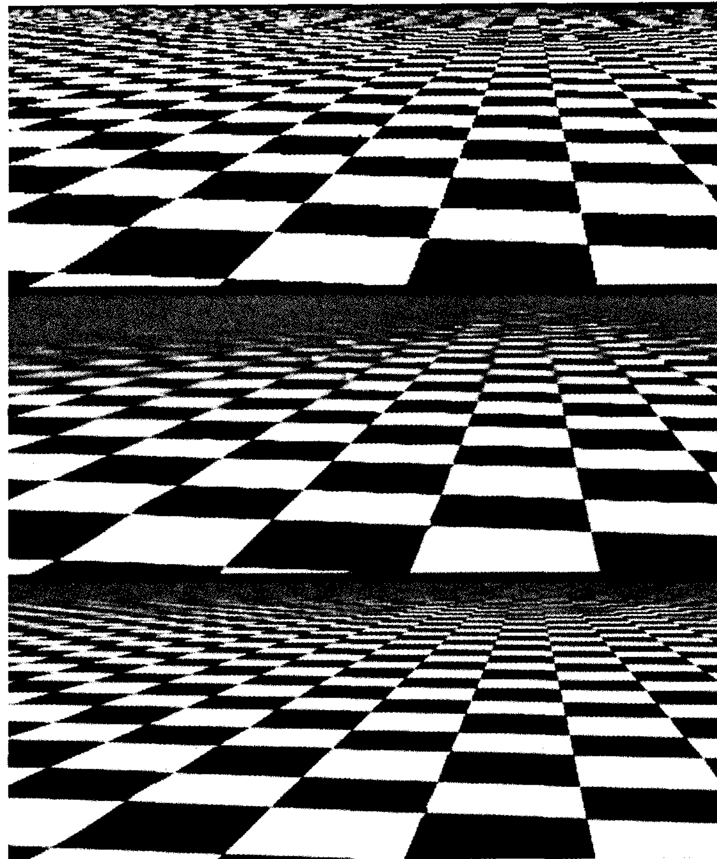

Figure 5: CheckerBoards mapped onto a square showing vertically compressed texture.

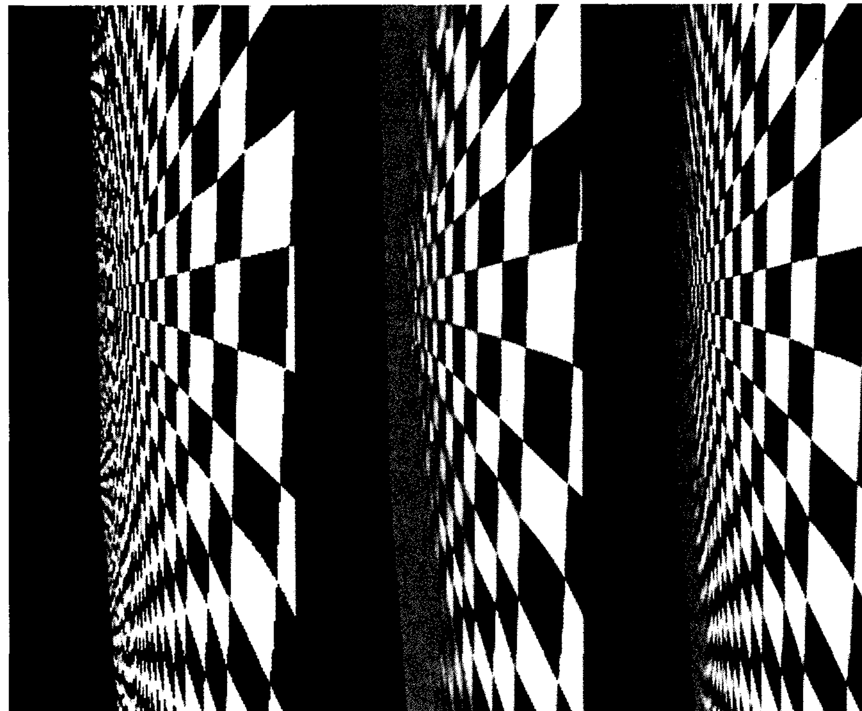

Figure 7: CheckerBoards showing horizontally compressed texture.