Performing In-place Affine Transformations in Constant Space

Ken Fishkin
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94034 USA

Abstract

Affine transformations of 2-D frame buffer images are a common computer graphics operation. Such transformations take a rectangular raster of image memory, perform some affine transformation (e.g. scale, shift, shear, rotate) upon it, and write the result into some other rectangular raster of image memory.

If the source and destination share the same memory, the operation is termed in-place. Previous in-place affine transformation algorithms on an m by n region required O(max(m,n)) space for internal buffers. The algorithm presented here requires O(1) (constant) space: this allows in-place affine transformations to be performed on large images on processors with small memory.

Keywords: affine transformations, Catmull-Smith, frame buffer algorithms.

1. Introduction

An affine transformation is a transformation of the form

$$x' = Ax + By + C,$$

$$y' = Dx + Ey + F,$$

for arbitrary real values A,B,C,D,E, and F.

Rotations, scales, shifts, and shears are all affine transformations. A computer graphics image is commonly considered as a rectangle whose contents are an array of pixels. An affine transformation can therefore be defined upon an image by performing the transformation upon the rectangle and then resampling [Catmull80].

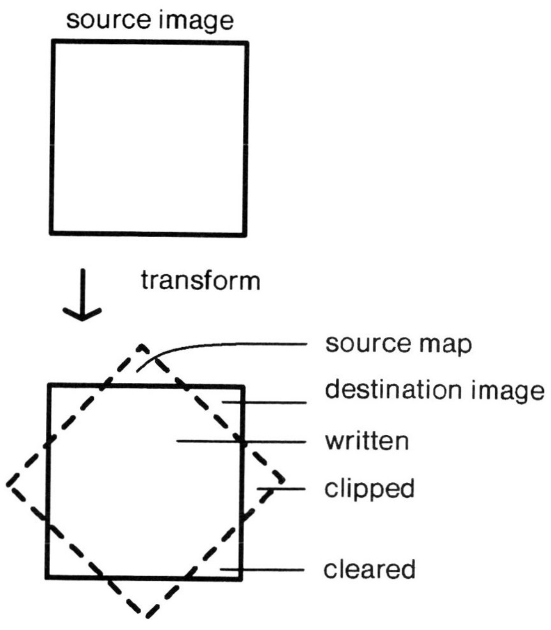

Figure 1: An affine transformation

2. Previous Work

2.1 The naive algorithm

A general affine transformation can be most easily implemented by creating a temporary image the same size as the source, and computing the value at each destination pixel in this temporary buffer. After all values have been computed, the temporary image over-writes the source image and is then freed. This approach requires O(m n) space when presented with an m by n source, and is only presented for comparison: it is not used in practice.

2.2 The Catmull-Smith algorithm

The popular Catmull-Smith algorithm [Catmull80] works by decomposing the affine transformation into two perpendicular shears. In the first shear the x' are computed, while the y coordinates are left untouched. In the second shear, the y' values are computed. To minimize artifacts the order of the two passes may be reversed, and transpositions or reflections may be required.

Each shear can be performed a scanline at a time, as y is unchanged by the x shear and x is unchanged by the y shear. Therefore, to transform an m by n image, a temporary scanline buffer of size O(max(m,n)) is required.

3. Motivation for the New Algorithm

The Catmull-Smith algorithm has become a fundamental and accepted part of computer graphics software environments, with only minor extensions [Fraser85, Smith87] in the twelve years since its original publication. Of what interest is a new algorithm based upon it? There are both theoretical and practical motivations for the new algorithm.

> *Theoretical*: Just as the Catmull-Smith algorithm was motivated by the inability to process an entire image in-core, this paper presents an algorithm which is motivated by the inability to process an entire scanline in-core. An extension to a common O(n) algorithm which uses O(1) space may be of some theoretical interest.
>
> *Practical*: As computer graphics matures, it has begun to tackle more sophisticated problem areas with less specialized processors. These new problem areas, such as pre-press and satellite imaging, often have images with thousands or tens of thousands of pixels per line, and with tens or hundreds of bits per pixel. Further, the applications which manipulate these images often must run on processors with limited address space, such as the IBM PC [Microsoft87], the Macintosh [Apple85], or the Pixar Image Computer [Levinthal84]. In such environments, an algorithm such as the one presented is a necessity, not a luxury.

If both the image and the calculation buffer are stored in core, than an O(1) algorithm is only of interest in the rare case where memory is just barely big enough to fit one image. An O(1) algorithm is of more use when the image is kept in secondary storage (disk, frame buffer, etc.) and only the calculation buffer is stored in-core. In that case, the difference between an O(n) and O(1) buffer can be significant.

3.1 Software Architecture

The new algorithm analyzes the transformation and decomposes it into a series of calls to the Catmull-Smith algorithm. This allows the new algorithm to be incorporated as a extension to an existing library: no change in base software is necessary. Furthermore, any efficiencies or optimizations encoded into implementations of the Catmull-Smith algorithm can still be called upon: the wheel need not be re-invented. This does require, however, that the new algorithm issue "requests" for affine transformations which always transform rectangular sources into rectangular destinations, as this is the common format expected by software libraries.

The algorithm presented in sections 6-11, therefore, satisfies two constraints besides the O(1) constraint. First, that the Catmull-Smith code is sacrosanct, and may not be changed or modified. This may be the case if that algorithm is provided in firmware, is written in micro-code, or is directly supported in hardware. Secondly, that the overhead involved in invoking that code is sufficiently significant that the number of calls to it should be minimized. If neither of those two constraints apply, then the simple and general algorithm presented in section 11 is sufficient.

4. Notation

The *source image* is the rectangular array of pixels comprising the picture which is to be transformed. The *destination image* is the rectangular array of pixels which is to receive the transformation. An *in-place* transformation is one in which the source image equals the destination image. The *source map* is the parallelogram which is defined by the map of the affine transformation over the source image. The source map and destination image will often be quite different. Pixels may be in the image but not in the map, in which case they are to be cleared to some background value. Pixels may be in the map but not in the image, in which case they are clipped. See figure 1.

Without loss of generality, we focus on the work to be performed by the x shear, where the shear acts on the x coordinates of the image without displacing it vertically. Let *iw* and *ih* be the width and height, respectively, of the source image. Let *dw* and *dh* be the width and height, respectively, of the destination image.

In the equation $x' = Ax + By + C$, we will rename A as *scale*, B as *tilt*, and C as *offset*, yielding $x' = scale * x + tilt * y + offset$.

Define *dst*(x,y) as the x component in destination space of the image under the affine transformation of the given x,y coordinate in source space. Similarly, let $src(x',y')$ be defined as the x component in source space of the pre-image of the given x',y' coordinate in destination space. Note that $src(dst(x,y),y) = x$. Both the *dst* and *src* functions return a single x value, with no y value; they both map from R^2 to R^1.

For example, a transformation which scales the input image up by 10% and shifts it left by y pixels on the y'th scanline would have

$$x' = dst(x,y) = 1.1x - y, \text{ and}$$

$$x = src(x',y) = (x' + y) / 1.1$$

We also assume the function *bbox*(), which computes the bounding box in source space of the pre-image of the current destination rectangle, adding the appropriate amount at each end for filtering. Further, let *M* be the maximum size of a scanline in core, and let *f* be the amount which must be added to each end of a source scanline in order for a destination scanline to be calculated. The value of *f* is a function of the filter width, the particular filtering algorithm used, and the scale. When resampling, each source pixel influences source space for *f* pixels to the left and the right, yielding a total "penumbra" of $2f + 1$ pixels in reconstructed source space. When scaling down ($scale < 1$), this penumbra is spread over a great many destination pixels. Therefore, to ensure that at least 1 pixel can always be written, we require that

$$M >= (2f + 1) / \mathrm{MIN}(scale, 1)$$

iw	source image width
ih	source image height
dw	destination width
dh	destination height
off	x' = off + x *scale + y * tilt
bbox()	bounding box
dst()	map from source to destination space
src()	map from destination to source space
M	pixel bandwidth limit
f	source padding due to filtering

Table 1: Notation

5. Why is this hard? The feedback problem

What makes an O(1) solution difficult? If no more than *M* pixels can be read or written at a time, one might imagine calling the Catmull-Smith algorithm on each *M*-sized chunk of the input scanline, writing the results out a piece at a time. The problem with this approach is that when an output piece is written, it may well overwrite a future input piece. This problem, on a grander scale, was exactly why the Catmull-Smith algorithm took pains to ensure that y is unchanging in the x pass, and why the naive algorithm allocated a huge buffer: *dst*(x,y) may be less than x, equal to x, or greater than x, and this relation may vary even within a scanline. The problem is exacerbated by the fact that anti-aliasing requires that an entire neighborhood of source pixels be readable when computing a single destination pixel.

Inefficiency is a further limitation of a per-line approach. The new algorithm will work by means of calls to the Catmull-Smith algorithm, which works on rectangular regions. It is a waste of inter-line coherence to call it on scanlines (or worse yet, parts of scanlines) only: we wish to call it as few times as possible, on regions as large as possible.

The problem is therefore to divide the original region, which is too large to transform all at once, into a set of regions, such that no pixel in any source region is overwritten before it is no longer needed, and the number of regions is as few as possible.

6. An outline of the algorithm

It is advantageous to deal with affine transformations in what we define as *standard form*:

$$scale >= tilt, \text{ and } scale > 0$$

This form can be obtained by transposition (to satisfy the first clause) and reflection (to satisfy the second). Standard form allows a few more invariant assumptions: that increasing x increases *x'*, and that vertical inter-scanline coherence is greater than horizontal inter-pixel coherence.

The algorithm consists of a series of transformation algorithms, with a case analysis to decide which transformation algorithm to perform. The transformation algorithms employed are more and more general and take more and more time: the case analysis finds the least general (and hence quickest) transformation algorithm appropriate for the particular transformation. The case analysis begins by dividing the source image into between 1 and 3 sub-images, depending on the characteristics of the transform. This process is described in section 8. At most one of these sub-images will require further case analysis, described in section 9. Some of that sub-images sub-images may require even further case analysis, described in sections 10 and 11. This completes all cases.

This case analysis is performed purely for optimization reasons: increasingly more difficult cases are processed by increasingly more general (but slow) algorithms. One could avoid all case analysis by using only the algorithm of section 11, but that could be far slower.

The case analysis routines all assume a subroutine, termed Helper, which performs an affine transformation on an arbitrarily large source and destination image, given the maximum internal scanline width M, and an evaluation order: left-to-right vs. right-to-left, and bottom-to-top vs. top-to-bottom. Before describing the case analysis in detail, we first describe 'Helper', the foundation of the system.

7. The Helper subroutine

The Helper subroutine is used to perform a piece-wise affine transformation from a given source image into a given destination image. It assumes that it need not worry about feedback. The Helper subroutine is solely concerned with slicing up the transformation into manageable chunks.

The obvious way to perform the Helper subroutine is to march along the source, transforming rectangles in the source into parallelograms which would then be written into the destination. This method was not chosen because, as Catmulll [Catmull80] says, "[n]ot only is this inconvenient, it is also difficult to prevent aliasing errors".

Instead, the Helper subroutines marches along in *destination* space: each destination rectangle is inverse-transformed to obtain a parallelogram in source space. That parallelogram is then rounded out into a rectangular bounding box (padding by f for filtering), and the Catmull-Smith algorithm is called. Since the source bounding box may be significantly larger than the source parallelogram, the shear may try to write a set of pixels which lie outside the current destination rectangle. However, since the destination rectangle lies along exact pixel boundaries, simple clipping will reject these extra pixels. See Figure 2.

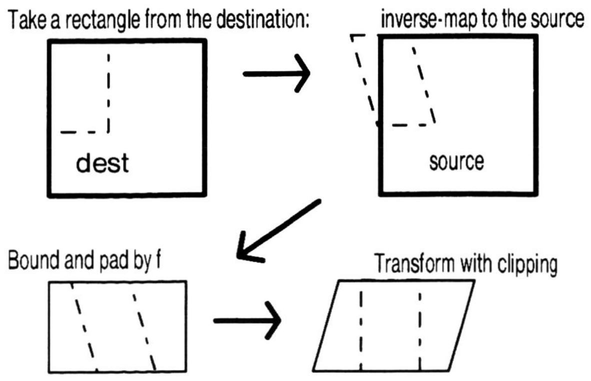
Take a rectangle from the destination:
inverse-map to the source
dest
source
Bound and pad by f
Transform with clipping

Figure 2: the Helper subroutine

What should be the dimensions of the destination rectangle? We wish to make it as large as possible without exceeding the M-pixel wide bottleneck.

Suppose we decide on a destination rectangle of size dx by dy. Then the constructed source rectangle will have width

$$sx = ((dx + (dy - 1)\,|tilt|)/scale) + 2f$$

Either dx or sx will be bounded by M. When $dx > M$, let $dx = M$ and $dy = dh$. When $sx > M$, there are more pixels to read than to write,

$$dx < sx,$$
$$dx < ((dx + (dy - 1)\,|tilt|)/scale) + 2f$$

We wish to minimize the number of destination rectangles, and therefore maximize the area of each. The problem is now to maximize $dx * dy$, where

$$1 <= dx <= M$$
$$scale < (dx + (dy - 1)|tilt|) / (dx - 2f)$$
$$((dx + (dy - 1)|tilt|)/scale) <= M$$

This is a quadratic programming problem. We approximate it by maximizing dy (setting it to dh), and then solving for a putative dx. If $dx < dy$ and $dx < dw$, then the putative rectangle is tall and skinny. To process an area closer to a square (this is desirable since Catmull-Smith has a per-scanline overhead), we recursively subdivide the rectangle by splitting it in two in y.

8. The first level of analysis

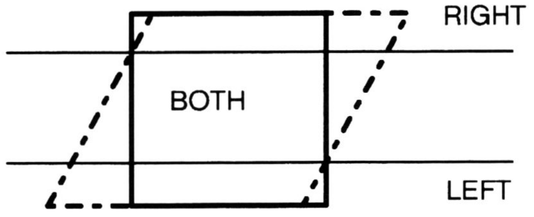

Figure 3: Different types of shear

The source image is subdivided into between one and three source sub-images, such that the map of the shear within each sub-image either:

1) overlaps it on the *left* on every line (type LEFT).
2) overlaps it on the *right* on every line (type RIGHT).
3) overlaps it on both the left *and* the right on every line (type BOTH).
4) overlaps it on *neither* the left *or* the right on every line (type NEITHER).

The dashed lines in figure 3 show the subdivision lines for that shear: the top region is of type RIGHT, the middle region is of type BOTH, and the bottom region is of type

LEFT.

8.1 Type LEFT

This case is easily processed. The Helper subroutine can be called on this region with an evaluation order of left-to-right.

8.2 Type RIGHT

The Helper subroutine can be called on this region with an evaluation order of right-to-left.

8.3 Type NEITHER

In this case, the transformation is scaling the source image down, and there is not a very pronounced tilt. However, evaluation order is still not clear: if *offset* (the translation component) is sufficiently large, then the first source pixels lie in the middle of the scanline: either right-to-left or left-to-right evaluations can fail. Therefore, this case is processed by decomposition into two shears:

1) The Helper subroutine is called to perform the scale component of the shear. Without any translation, *dst*(x,y) < x for all x, and so the algorithm should proceed left-to-right.
2) The Helper subroutine is called again to perform the non-scale component. If *offset* <= 0, then *dst*(x,y) < x, and so the algorithm should proceed left-to-right. Otherwise, it should proceed right-to-left.

8.4 Type BOTH

This case is complex and treated in detail in the next section.

9. The second level of analysis: type BOTH

Using the case analysis of section 8, 3 of the 4 possible shear configurations could be processed. This section focuses on the fourth, most difficult case, when the source map overflows the destination image on both sides. This case is more difficult because every pixel in the destination must be written, and hence every pixel in the source is at risk of being over-written before it is read.

Checks are first made to see if the shear falls into one of two special cases:

9.1 small-source

Suppose that the shear is a large scale-up, with little or no translation/tilt component. In this case, only a small part of the source is necessary. The algorithm can "cut out" that portion, and use the other portions as it wishes. The area of the source needed is considered "small" if and only if

$$bbox() <= dw$$

This means that the area of the source needed to compute the destination (*bbox*()) fits entirely within the destination (width *dw*). See Figure 4 for an example.

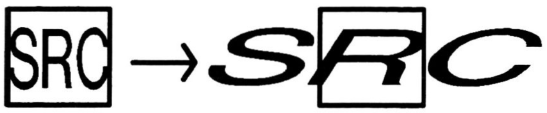

Figure 4: small-source: only the 'R' part of the source is needed

9.2 small-dest

Conversely, suppose that the shear is a large scale-down. Regardless of the translation/tilt component, in this case only a small number of destination pixels are truly "derived": the others will be set to a certain background color. In this case the algorithm can compute that portion and then shift/shear it into place. The area of the destination needed is considered "small" if and only if

$$iw * scale <= dw$$

This means that the width of the scaled source image (*iw * scale*) fits entirely within the destination. See Figure 5 for an example.

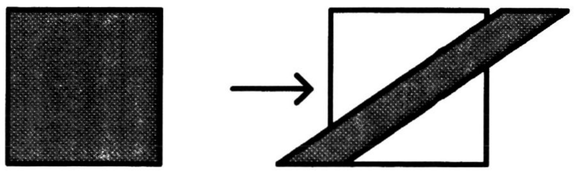

Figure 5: small-dest: only a small part of the destination is derived

9.3 if either succeeds

If either condition "small-source" or "small-dest" holds, then the affine can be implemented as follows (see Figure 6):

1) Copy the needed portion of the source, left-aligned, into the destination.
2) Perform the scale component of the transformation using the Helper subroutine. Perform this right-to-left if *scale* > 1, and left-to-right otherwise.
3) Recursively perform an affine transformation, using only the other components of the original affine transformation.

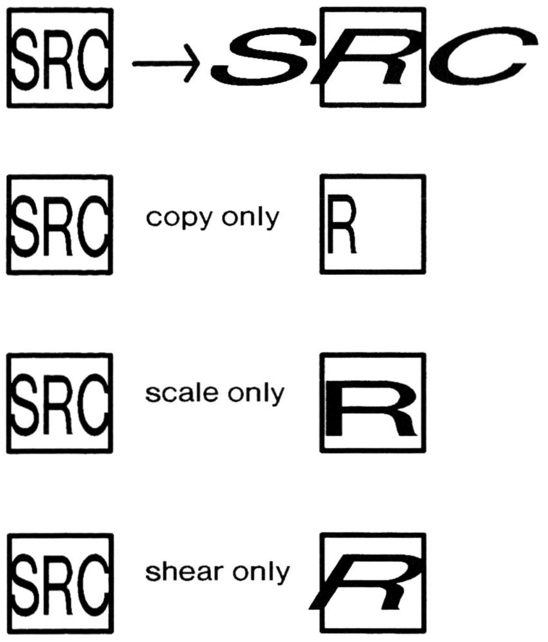

Figure 6: processing small-source

10. The third level of case analysis

Neither of the above special cases may hold. In that case, further analysis is required.

Consider a given (x,y) pixel in the destination. There are three cases for that pixel: it may be far to the left of its source pixel pre-image, it may be far to the right of it, or it may be neither. It is "far to the left" if

$$x' <= src(x',y) - f$$

and "far to the right" if

$$x' >= src(x',y) + f$$

Define the boolean predicate *left*(x',y) to be true if and only if a destination pixel is far to the left, and the boolean predicate *right*(x',y) to be true if and only if a destination pixel is far to the right. Since the transformation is in standard form, *left*(x',y) implies *left*(x'-1,y) and *right*(x',y) implies *right*(x'+1,y).

Due to the tilt in the shear, the rightmost *left*() pixel and the leftmost *right()* pixel may be at a different place on each line. Therefore, we define the function *LEFT*() as the greatest x' such that *left*(w',y) for all w' <= x', for all 0 <= y <= *dw*. *RIGHT*() is similarly defined as the least x' such that all pixels to the right of it have the *right*() property. Figure 8 shows a sample set of *LEFT*() and *RIGHT*() regions.

Intuitively, the *left*() and *right*() predicates detect those destination pixels which are "thrown clear" of their source pixels. The *LEFT*() and *RIGHT*() regions are the largest rectangles contained within the parallelogram-shaped *left*() and *right*() regions.

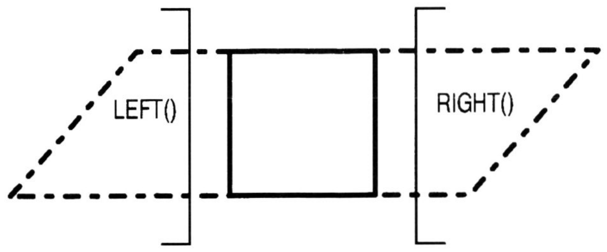

Figure 8: The LEFT() and RIGHT() regions

If *LEFT*() or *RIGHT*() regions exist, then the Helper algorithm is performed on the destination/source regions defined by them, and the problem is trimmed accordingly. However, not all source pixels are freed by this operation: without further help, this operation would quickly grind to a halt.

Therefore, the remaining source image is now split in two along y, and those affine transformations are now recursively analyzed. Splitting the image in this manner reduces the effect of the tilt in the shear, and may create *LEFT*() and *RIGHT*() regions in the sub-images which could not be formed in the originals. In Figure 9, for example, there are no *LEFT*() or *RIGHT*() regions originally, but subdivision creates 4 such regions.

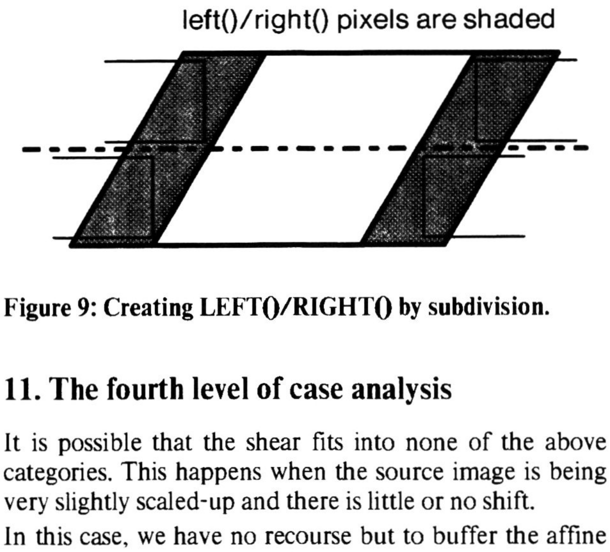

Figure 9: Creating LEFT()/RIGHT() by subdivision.

11. The fourth level of case analysis

It is possible that the shear fits into none of the above categories. This happens when the source image is being very slightly scaled-up and there is little or no shift.

In this case, we have no recourse but to buffer the affine transformation with saves/restores of selected areas of image memory, "stitching" the borders between the panels. Specifically, we now need two buffers B1 and B2, of *f* pixels each. On each scanline:

1) Find the fixed point F, the pixel such that *src*(F,y) = F.

2) Read the pixels from [F-f..F] into B1. These pixels will be written when the left half of the scanline is processed,

but their original values are still needed to process the right half.

3) call the Helper algorithm to evaluate the scanline from [0..F], left-to-right.

4) Read the pixels from [F-f..F] into B2, and then write them with B1.

5) call the Helper algorithm to evaluate the scanline from F onwards, right-to-left.

6) Write the pixels from[F-f..F] from B2.

This algorithm can be performed in *all* cases, but it is very slow: it calls the Helper algorithm twice per scanline, and also must perform 4 buffer I/O operations. If, however, the algorithm is to replace the Catmull-Smith code, then these objections no longer hold, and the algorithm above yields a slower but more general replacement.

12. Summary

An algorithm has been presented to perform in-place affine transformations in constant space. The algorithm subdivides the transformation into a series of smaller transformations. Each smaller transformation is then performed using the Catmull-Smith algorithm. In this way, the new algorithm provides an additional level of capability to a graphics software library, which is particularly appropriate in environments where image sizes are huge and/or processor memory size is limited.

The algorithm works by case analysis, chipping away at the problem by gradually imposing slower and more general algorithms on more difficult portions of the affine transformation: Figure 10 provides a summary.

```
   A summary of the algorithm:
/* split the source into between
 * one and three sub-regions.
 * calls to 'Helper' are of the form
 * Helper(source,dest,
 *        eval order, special notes);
 */

dir2 = (offset <= 0.0) ?LtoR:RtoL;
foreach subregion S do
  switch (type) {
  case LEFT:
    /* section 8.1 */
    Helper(S,S,LtoR);
    break;
  case RIGHT:
    /* section 8.2 */
    Helper(S,S,RtoL);
    break;
  case NEITHER:
    /* section 8.3 */
    Helper(s,s,LtoR,scale);
    Helper(s,s,dir2, non-scale);
    break;
  case BOTH:
    if small-source or small-dest {
      /* see figures 4,5, and 6,
         sections 9.1 - 9.3 */
      Copy(needed-part(S),
        left-part(S),left-aligned);
      dir1 = (scale <= 1.0)?LtoR:RtoL;
      Helper(left-part(S),S,
        dir1,scale);
      Helper(S,S,dir2,non-scale);
    } else {
      /* section 10, figure 8 */
      if LEFT region exists {
        Helper(S,LEFT,LtoR);
        S = pruned-part(S);
      }
      if RIGHT region exists {
        Helper(S,RIGHT,LtoR);
        S = pruned-part(S);
      }
      /* figure 9 */
      if (height(S) > 1) {
        Recurse(top-half(S));
        Recurse(bottom-half(S));
      } else {
        /* section 11 */
        stitch
      }
    }
    break;
}
```

Figure 10: Summary of the Algorithm

13. Extensions and modifications

The algorithm may be extended to handle cases when the source image is a proper subset of the destination image. That case has not been discussed here for presentation purposes. For a discussion of that extension and other implementation issues, the interested reader is referred to [Fishkin89].

14. Acknowledgements

The author wishes to thank Don Conway and David De Francesco (Pixar) for their help with the screen shots, and Andrew Glassner (Xerox PARC) and Tony DeRose (the University of Washington) for their many helpful comments on the organization of the paper.

15. References

[Apple85] Apple Computer, "Inside Macintosh", Addison-Wesley, volume 2, 1985.

[Catmull80] Catmull, Edwin, and Smith, Alvy Ray, "3-D Transformations of Images in Scanline Order", *Computer Graphics* **14**(3), July 1980, pp. 279-285.

[Fishkin89] Fishkin, Ken, "Performing piece-wise in-place affine transformations", Pixar Technical Memo #188, February 1989.

[Fraser85] Fraser, Donald, Schowengerdt, Robert A., and Briggs, Ian, "Rectification of Multichannel Images in Mass Storage Using Image Transposition", *Computer Vision, Graphics, and Image Processing,* **29**(1), pp. 23-26, January 1985.

[Levinthal84] Levinthal, Adam, and Porter, Thomas, "Chap - a SIMD Graphics Processor", *Computer Graphics* **18**(3), July 1984, pp. 77-82.

[Microsoft87] Microsoft Corporation, "Microsoft C Optimizing Compiler: User's Guide", 1987, Chapter 6.

[Porter84] Porter, Thomas, and Duff, Tom, "Compositing Digital Images", *Computer Graphics* **18**(3), July 1984, pp. 253-259.

[Smith87] Smith, Alvy Ray, "Planar 2-Pass Texture Mapping and Warping", *Computer Graphics* **21**(4), July 1987, pp. 263-272.

Appendix: An Example

These pictures show an in-place affine transformation of a 1024 by 768 image, when no more than 256 pixels may be read or written at any time. The image is rotated by 10 degrees and also scaled up by 10%. The dark lines delimit the LEFT, BOTH, and RIGHT regions. The light lines delimit the individual regions passed to Catmull-Smith.

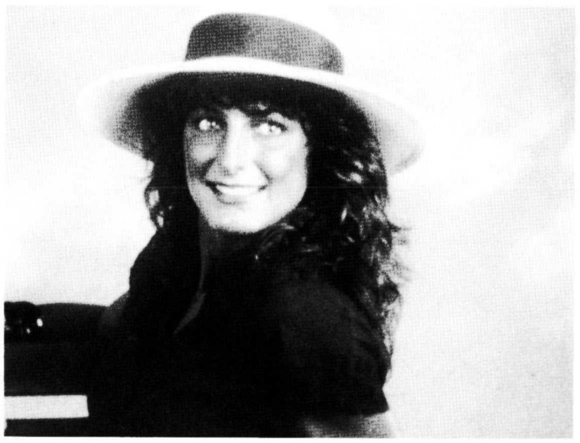
Figure A.1: The Original Image

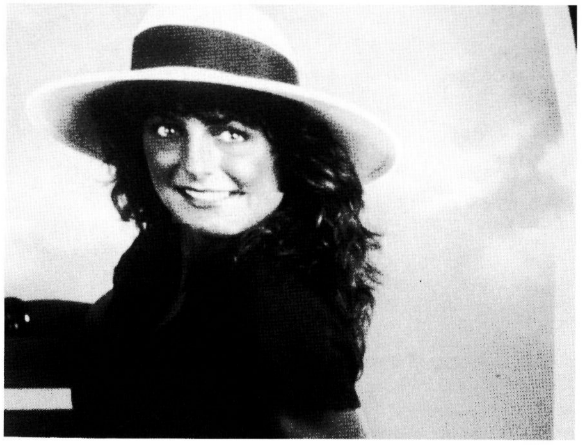
Figure A.2: In the middle of the first pass. The algorithm is transforming the area around the mouth.

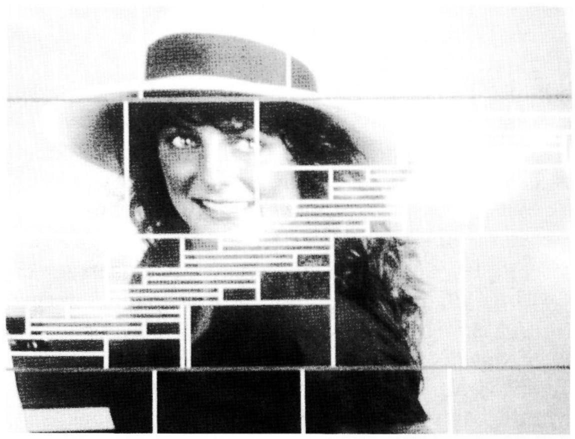
Figure A.3: The First Pass

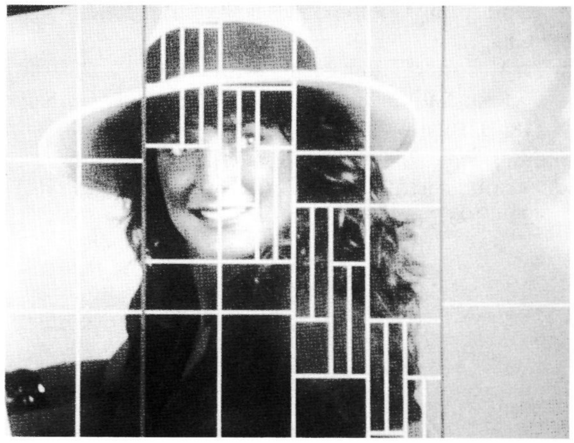
Figure A.4: The Second Pass

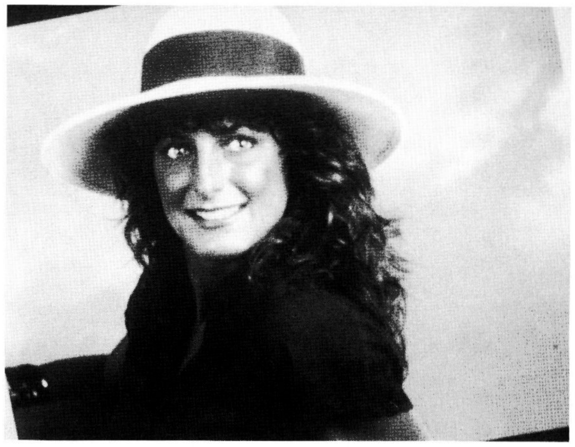
Figure A.5: The Final Image