Vol. 5, No. 3: 11–33

One-Dimensional Resampling with Inverse and Forward Mapping Functions

George Wolberg
City College of New York

H. M. Sueyllam, M. A. Ismail, K. M. Ahmed
Alexandria University

Abstract. Separable resampling algorithms significantly reduce the complexity of image warping. Fant presented a separable algorithm that is well suited for hardware implementation. That method, however, is inherently serial and applies only when the inverse mapping is given. Wolberg presented another algorithm that is less suited for hardware implementation and applies only when the forward mapping is given. This paper demonstrates the equivalence of the two algorithms in the sense that they produce identical output scanlines. We derive a variation of Fant's algorithm that applies when the forward mapping is given and a variation of Wolberg's algorithm that applies when the inverse mapping is given. Integrated hardware implementations that perform one-dimensional resampling under either forward or inverse mappings are presented for both algorithms based on their software descriptions. The Fant algorithm has the advantage of being simple when implemented in hardware, while the Wolberg algorithm has the advantage of being parallelizable and facilitates a faster software implementation. The Wolberg algorithm also has the advantage of decoupling the roundoff errors made among intervals since it does not accrue errors through the incremental calculations required by the Fant algorithm.

1. Introduction

Separable resampling algorithms are useful in casting image warping into a framework that is amenable to hardware implementation. Growing interest in this area has gained impetus from the widespread proliferation of advanced

© A K Peters, Ltd.

1086-7651/00 $0.50 per page

workstations and digital signal processors. Applications include geometric image manipulation for medical imaging, remote sensing, and computer vision, as well as real-time hardware for video effects. The central benefit of separable algorithms is the reduction in complexity of one-dimensional resampling algorithms, where efficient solutions exist for image reconstruction and antialiasing.

Resampling requires a mapping function to govern the geometric relationship between each point in the input and output images. The transformation is said to be a forward mapping if input pixels are mapped to the output. Similarly, the transformation is said to be an inverse mapping if output pixels are mapped to the input. Forward mapping is prominent in video processing where the input stream arrives in scanline order. Inverse mapping is common in computer graphics, where the output pixels are typically rendered in scanline order.

This paper demonstrates the equivalence of two separable resampling algorithms previously presented as distinct solutions for forward and inverse mappings. Fant presented a separable solution that is well suited for hardware implementation when the inverse mapping is given [Fant 86]. Wolberg presented another solution that is less suited for hardware implementation and applies only when the forward mapping is given [Wolberg 90]. The unification of these two approaches offers insight and flexibility in handling forward and inverse mappings under serial or parallel implementation.

2. Background

Image resampling is the process of transforming a sampled image from one coordinate system to another. The two coordinate systems are related to each other by a spatial transformation. The transformation is expressed as a mapping function that defines the geometric relationship between each point in the input and output images. The general mapping function can be given in two forms: either relating the output coordinate system to that of the input, or vice versa. Respectively, they can be expressed as

$$[x, y] = [X(u, v), Y(u, v)] \quad (1a)$$

$$[u, v] = [U(x, y), V(x, y)] \quad (1b)$$

where $[u, v]$ refers to the input image coordinates corresponding to output pixel $[x, y]$, and X, Y, U, and V are arbitrary mapping functions that uniquely specify the spatial transformation. Since X and Y map the input onto the output, they are referred to as the forward mapping. Similarly, the U and V functions are known as the inverse mapping since they map the output onto the input.

Geometric transformations have traditionally been formulated as either forward or inverse mappings operating entirely in two dimensions. Their advantages and drawbacks are given below to motivate the case for separable algorithms.

2.1. Forward Mapping

Forward mappings deposit input pixels into an output accumulator array. A distinction is made here based on the order in which pixels are fetched and stored. In forward mappings, the input arrives in scanline order (row by row) but the results are free to leave in any order, projecting into arbitrary areas in the output. In the general case, this means that no output pixel is guaranteed to be totally computed until the entire input has been scanned. Therefore, a full two-dimensional accumulator array must be retained throughout the duration of the mapping. Since the square input pixels project onto quadrilaterals at the output, costly intersection tests are needed to properly compute their overlap with the discrete output cells. Furthermore, an adaptive algorithm must be used to determine when supersampling is necessary in order to avoid blocky appearances on one-to-many mappings.

2.2. Inverse Mapping

Inverse mappings are more commonly used to perform spatial transformations. By operating in scanline order at the output, square output pixels are projected onto arbitrary quadrilaterals. In this case, the projected areas lie in the input and are not generated in scanline order. Each preimage must be sampled and convolved with a low-pass filter to compute an intensity at the output. This expensive antialiasing component is often approximated through the use of a mipmap, a pyramid of filtered and downsampled versions of the image [Williams 83]. Filtering over a quadrilateral preimage is approximated by sampling the pyramid at an appropriate level. Trilinear interpolation is necessary to sample between pyramid levels.

2.3. Separable Mapping

While either forward or inverse mappings can be used to realize arbitrary mapping functions, there are many transformations that are adequately approximated when using separable mappings.

There are several advantages to decomposing a mapping into a series of one-dimensional transforms. First, the resampling problem is made simpler since reconstruction, area sampling, and filtering can now be done entirely in one dimension. Second, this lends itself naturally to digital hardware im-

plementation. Note that no sophisticated digital filters are necessary to deal explicitly with the two-dimensional case. Third, the mapping can be done in scanline order both when scanning the input image and in producing the projected image. In this manner, an image may be processed in the same format in which it is stored in the framebuffer: rows and columns. This leads to efficient data access and large savings in I/O time. The approach is amenable to stream-processing techniques such as pipelining and facilitates the design of hardware that works at real-time video rates.

The first most general presentation of a separable two-pass algorithm for image warping is described in the seminal work of Catmull and Smith [Catmull, Smith 80]. Their technique decomposes a two-dimensional parallel warp, consisting of $x = X(u, v)$ and $y = Y(u, v)$, into a sequence of two one-dimensional serial warps consisting of the horizontal pass $x = F_v(u) = X(u, v)$, followed by the vertical pass $y = G_x(v) = Y(H_x(v), v)$, where $u = H_x(v)$ is the inverse of $X(u, v) - x = 0$ [Smith 87]. Here u and v constitute the input image (texture) space, and x and y constitute the output image (screen) space.

The advantage of a two-pass spatial transformation algorithm over its one-pass counterpart is that it offers a more effective and efficient match with current implementation technology, despite the assumptions and errors that fall into this model of computation. The central benefit of separable algorithms is the reduction in complexity of one-dimensional resampling algorithms. When the input is restricted to be one-dimensional, efficient solutions are made possible for the image reconstruction and antialiasing components of resampling.

Separable resampling algorithms cannot accommodate all transformations. Although they have been shown useful for affine and perspective transformations, they are susceptible to bottleneck and foldover problems. After completing the first pass, it is sometimes possible for the intermediate image to collapse into a narrow area. If this area is much less than that of the final image, then there is insufficient data left to accurately generate the final image in the second pass. This phenomenon, referred to as the *bottleneck problem* in [Catmull, Smith 80], is the result of a many-to-one mapping in the first pass followed by a one-to-many mapping in the second pass. One example of this problem is a 90° rotation. Since the top row will map to the rightmost column, all of the points in the scanline will collapse onto the rightmost point. Similar operations on all the other rows will yield a diagonal line as the intermediate image. No possible separable solution exists for this case when implemented in this order. The solution to this problem lies in considering all the possible orders in which a separable algorithm can be implemented. If we first rotate the image by the closest multiple of 90°, then we can apply the two-pass technique on the difference between that multiple and the desired rotation angle. Despite the bottleneck problem, separable algorithms are widely used in affine, perspective, and rubbersheet transformations. The foldover problem is addressed in [Wolberg, Boult 89].

3. Fant's Algorithm

This section describes all aspects of one-dimensional resampling with Fant's algorithm (FALG). The original algorithm was formulated for use with inverse mapping functions [Fant 86]. We extend it to work with forward mapping as well.

3.1. FALG: Inverse Mapping

The input to Fant's algorithm consists of the following four variables:

- *inlen*: the length of the input scanline (in pixels);
- *outlen*: the length of the output scanline (in pixels);
- *in*: an array of input scanline pixels (intensity/color);
- *f*: an array of real-valued coordinates denoting the inverse mapping of each output pixel.

The algorithm fills an array *out* with the intensity values associated with the output scanline pixels. This process is driven by the inverse mapping *f*. Note that in the original paper, Fant used an array of inverse size factors instead of the inverse mapping as an input to the algorithm. In our presentation, we assume that the inverse mapping f is provided as an input, and we derive the inverse size factor required by the algorithm through simple subtraction operations.

The basic algorithm is very simple. It maintains two fractional-valued pointers, or position markers, in the stream of contiguous input pixels: *inseg* and *outseg*. The input pixel *inseg* indicates how much of the current input pixel remains to contribute to the next output pixel; this ranges from 0 (totally consumed) to 1 (entirely available). Analogously, *outseg* indicates how much of the current input pixel is required to complete the next output pixel.

The process begins by comparing the values of *inseg* and *outseg* to determine which is smaller. If *outseg* is smaller, then an *output cycle* occurs: an output pixel will be completed. The current input pixel intensity value is multiplied by *outseg* and added to the accumulator, *inseg* is decremented by the value of *outseg* to indicate that the *outseg* portion of the input pixel has been used. Then, *outseg* is reinitialized to *isf* $[x+1]$, which is the difference between two successive f values, $f[x+2]-f[x+1]$, where x is the index of the current output pixel. Indices $x+1$ and $x+2$ refer to the next and subsequent output pixels, respectively. The value of the current output pixel is determined by dividing the contents of the accumulator by *isf* $[x]$. Then, the accumulator is cleared and the process returns to compare *inseg* and *outseg*.

If *inseg* is smaller, then an *input cycle* occurs: an input pixel will be consumed. The current input pixel value is multiplied by *inseg* and added to the accumulator, *outseg* is decremented by the value of *inseg* to indicate that the *inseg* portion of the output pixel has been satisfied. Then, *inseg* is reinitialized to 1.0, and the next input pixel is fetched. The process then returns to compare the new values of *inseg* and *outseg*.

If *inseg* and *outseg* are equal, the input pixel will be consumed and an output pixel will be completed. In this case, either event can be chosen to occur first. For instance, we may consider an output cycle followed by a zero input cycle, in which case *inseg* is equal to zero, which causes a 0 to be added to the accumulator and uses up the current input pixel.

To complete the description of the algorithm, we need to address the following four issues: initialization, reconstruction, antialiasing, and termination.

3.1.1. Initialization

The variable, *inseg*, is always initialized to 1.0 at the end of every input cycle, indicating that the next input pixel is fully available to contribute to the output. However, before starting the process, *inseg* is initialized to $1.0 - Fraction(f[0])$ when $f[0] \geq 0$, indicating that only a partial output pixel remains to be filled; otherwise *inseg* is set to 1.0.

At the end of every output cycle, the variable *outseg* is always initialized to the inverse size factor associated with the output pixel that will be processed in the next cycle. However, before starting the process, *outseg* is initialized to $isf\,[0]$ when $f[0] \geq 0$; otherwise *outseg* is set to $isf\,[x] + f[x]$, where x is the rightmost pixel with the property $f[x] < 0$. Finally, the accumulator is cleared when we start.

3.1.2. Reconstruction

Image reconstruction refers to the interpolation of sampled image data. It permits us to evaluate the discrete signal at any desired position, not just the integer lattice upon which the sampled signal is given. This is necessary to avoid a blocky appearance upon magnification, i.e., preventing a sequence of output pixels from having the same intensity value when an input pixel contributes to more than one output pixel.

Reconstruction is performed by convolving the discrete input signal with a continuous interpolating function. In Fant's algorithm, reconstruction is achieved by using pixel values from a two-pixel-wide averaging window traversing the raw input pixels. A triangle filter is used as the filter kernel. This produces results identical to linear interpolation. The placement of the kernel upon the input is based on the value of *inseg*. Thus, using the triangle filter, we have the following 2-point convolution:

```
intensity = inseg*in[u] + (1.0-inseg)*in[u+1];
```

where $in[u]$ and $in[u{+}1]$ are the intensity values of two successive input pixels. For the last input pixel, we let $u+1$ revert to u to realize border replication. Note, that when *inseg* is equal to 1.0, such as at the left boundary of an input pixel, no interpolation takes place. More sophisticated reconstruction filters can be used instead of the 2-point triangle filter. For example, a 4-point cubic convolution[Wolberg 90] filter can be used. Its interpolation kernel can be coded in a function $h(x)$ as follows:

```
float h(float x) {
   if(x<0) x = -x;                      /* the kernel is symmetric; use positive x  */
   if(x<1) return (1+x*x*(-2+x));       /* cubic polynomial for 0 <= x < 1          */
   if(x<2) return (4+x*(-8+x*(5-x)));  /* cubic polynomial for 1 <= x < 2          */
   return 0;                            /* h is a 4-point kernel: h(x)=0 for |x|>=2 */
}
```

The 2-point convolution given earlier may be replaced by the following 4-point convolution:

```
intensity = h(2.0-inseg)*in[u-1] + h(1.0-inseg)*in[u] +
                  h(inseg)*in[u+1] + h(1.0+inseg)*in[u+2];
```

Of course, $in[-1]$, $in[inlen]$, and $in[inlen+1]$ have to be reserved and set to the value of the nearest border pixel, i.e., border replication. This could be accomplished by reserving an array $in1$ of dimension $inlen+3$, and defining a pointer in which is initialized to $in1+1$.

3.1.3. Antialiasing

Antialiasing refers to the filtering used to counter the aliasing artifacts symptomatic of undersampling. Artifacts such as moire patterns may become readily apparent when important image detail is lost through downsampling. Thus, antialiasing filtering is particularly necessary when decimating an image.

Antialiasing, as originally implemented by Fant, uses box filtering: intensities of all input pixels contributing to the current output pixel are weighted by their area coverage (the smaller of *inseg* and *outseg*) and summed together. Implicit in this operation is an underlying piecewise constant model of the input intensity function. A piecewise linear model would be superior. Antialiasing may now be computed by evaluating the area under the (linear) curve that contributes to the output. The integral is readily computed by the average of the curve endpoints. Therefore, in an input cycle, instead of simply computing

```
out[x] = intensity * inseg;
```

we may compute

```
intensity2 = in[u+1];
out[x]     = (intensity+intensity2)*inseg / 2.0;
```

Similarly, in an output cycle, instead of simply computing

```
out[x] = intensity * outseg;
```

we may compute

```
intensity2 = (inseg-outseg)*in[u] + (1.0-(inseg-outseg))*in[u+1];
out[x]     = (intensity+intensity2)*outseg / 2.0;
```

3.1.4. Termination

The algorithm terminates when all input pixels are consumed or when all output pixels are completed, whichever occurs first.

3.1.5. Software Implementation

A software implementation of the algorithm in C follows. In all subsequent algorithms, we use the triangle filter for reconstruction and the box filter for antialiasing.

```
fantinv(double f[], int in[], int out[], int inlen, int outlen)
{
   int     u,               /* index into input  image             */
           x,               /* index into output image             */
           xl,              /* index of the  leftmost output pixel */
           xr;              /* index of the rightmost output pixel */
   double  isf,             /* inverse scale factor                */
           inseg,           /* fraction of input pixel available   */
           outseg,          /* fraction needed to complete output  */
           intensity;       /* interpolated input intensity value  */

   /* clear output/accumulator array */
   for (x=0; x<outlen; x++) out[x] = 0;

   /* check if no output scanline pixel projects into [0,inlen] range */
   if(f[outlen] < 0 || f[0] > inlen) return;      /*  100% clipping  */

   /* init xl, the left-extent of the output scanline   */
   for(x=0; f[x]<0; x++);                   /* advance x */
   xl = (x>0) ? x-1 : x;                    /* init   xl */

   /* init xr, the right-extent of the output scanline  */
   for(x=outlen; f[x]>inlen; x--);          /* advance x */
   xr = (x == outlen) ? outlen-1 : x;       /* init   xr */

   /* initialization */
   isf = f[xl+1] - f[xl];
```

```
   if(f[xl] < 0) {
           inseg  = 1.;
           outseg = isf + f[xl];
   } else {    /* xl == 0 */
           inseg  = 1. - (f[0] - (int) f[0])
           outseg = isf;
   }

   /* main loop */
   u = (f[xl] < 0) ? 0 : f[xl];
   for(x=xl; x<=xr; ) {
     /* linearly interpolated intensity */
     intensity = (inseg*in[u]) + ((1.-inseg)*in[u+1]);

     /* inseg<outseg: input pixel is entirely consumed before output pixel*/
     if(inseg < outseg) {             /* input cycle                      *
        out[x] += (intensity*inseg);  /* accumulate (partial) input pixel */
        outseg -= inseg;              /* input needed to complete output  */
        u++;                          /* advance input index              */
        if(u == inlen) {              /* are all input pixels consumed?   */
           out[x] /= isf;             /* normalize accumulated value      */
           break;                     /* completed scanline: exit loop    */
        }
        inseg = 1.;                   /* restore inseg for next cycle     */
     }

     /* inseg>=outseg: input pixel is not fully consumed before output pxl*/
     else {                           /* output cycle                     */
        out[x] += (intensity*outseg); /* accumulate (partial) input pixel */
        out[x] /= isf;                /* normalize accumulated value      */
        inseg -= outseg;              /* input available for next output  */
        x++;                          /* advance output index             */

        /* not needed; kept for compatibility with the forward mapping alg*/
        if(x == outlen) break;        /* are all output pixels completed? */

        /* restore isf and outseg for next cycle */
        isf    = f[x+1] - f[x];
        outseg = isf;
     }
   }
}
```

Our observation is that the algorithm is inherently serial. It proceeds in cycles in which the values of *inseg* and *outseg* are incrementally updated. Therefore, we cannot process several output pixels in parallel; the starting values for *inseg* and *outseg* for a particular output pixel is determined only after all preceding output pixels have been processed. This also has the potential problem of accruing roundoff errors in the spatial domain, due to the continued mutual subtraction of the two nearly equal values of *inseg* and *outseg*.

3.2. FALG: Forward Mapping

The formulation of the forward mapping instance of FALG follows a similar process to that of the original inverse mapping formulation. We maintain two fractional-valued pointers in the stream of contiguous output pixels: *inseg* and *outseg*; *inseg* indicates how much of an output pixel is required to be completed by the current input pixel; *outseg* indicates how much of the current output pixel is available to be completed by the current input pixel.

Reconstruction is achieved by using pixel values from a two-pixel-wide averaging window traversing the raw input pixels based on the ratio of *inseg/sf*. Thus, using the triangle filter, we have

```
intensity = (inseg/sf)*in[u] + ((sf-inseg)/sf))*in[u+1]
```

The rationale behind this interpolation is that the remaining fraction of the current input pixel available to complete output pixels is given by *inseg/sf*, which is the remaining fraction of *inseg* in the stream of output pixels. This, of course, makes the reasonable assumption that pixels have the same fixed size in the input and output. A software implementation in C follows.

```
fantfwd(double f[], int in[], int out[], int inlen, int outlen)
{
   int     u,               /* index into input  image            */
           x,               /* index into output image            */
           ul,              /* index of the  leftmost input pixel */

           ur;              /* index of the rightmost input pixel */
   double  sf,              /* input-to-output scale factor       */
           inseg,           /* fraction of input pixel available  */
           outseg,          /* fraction needed to complete output */
           intensity;       /* interpolated input intensity value */

   /* clear output/accumulator array */
   for (x=0; x<outlen; x++) out[x] = 0;

   /* check if no input scanline pixel projects to [0,outlen] range */
   if(f[inlen] < 0 || f[0] > outlen) return;       /* 100% clipping */

   /* init ul, the left-extent of the input scanline    */
   for(u=0; f[u]<0; u++);                 /* advance u */
   ul = (u>0) ? u-1 : u;                  /* init   ul */

   /* init, ur the right-extent of the input scanline   */
   for(u=inlen; f[u]>outlen; u--);        /* advance u */
   ur = (u == inlen) ? inlen-1 : u;       /* init   ur */

   /* initialization */
   sf = f[ul+1] - f[ul];
   if(f[ul] < 0) {
```

```
         inseg  = sf + f[ul];
         outseg = 1.;
   } else {    /* ul == 0 */
         inseg  = sf;
         outseg = 1. - (f[0] - (int) f[0]);
   }
   /* main loop */
   x = (f[ul] < 0) ? 0 : f[ul];
   for(u=ul; u<=ur; ) {
        /* linearly interpolated intensity */
        intensity = (inseg/sf * in[u]) + ((sf-inseg)/sf * in[u+1]);

        /* inseg<outseg: input pixel is entirely consumed before output pixel*/
        if(inseg < outseg) {            /* input cycle                       */
           out[x] += (intensity*inseg); /* accumulate (partial) input pixel */
           outseg -= inseg;             /* input needed to complete output  */
           u++;                         /* advance input index              */
           if(u == inlen) {             /* are all input pixels consumed?   */
              out[x] /= 1.0;            /* compatibility with inverse alg   */
              break;                    /* completed scanline:  exit loop   */
           }

           /* restore sf and inseg for next cycle */
           sf    = f[u+1] - f[u];
           inseg = sf;
        }

        /* inseg>=outseg: input pixel is not fully consumed before output pxl*/
        else {                          /* output cycle                      */
           out[x] += (intensity*outseg); /* accumulate (partial) input pixel */
           out[x] /= 1.0;               /* compatibility with inverse alg   */
           inseg -= outseg;             /* input available for next output  */
           x++;                         /* advance output index             */
           if(x == outlen) break;       /* are all output pixels completed? */
           outseg = 1.0;                /* restore outseg for next cycle    */
        }
   }
}
```

4. Resampling with WALG

This section describes all aspects of one-dimensional resampling with Wolberg's algorithm (WALG). The original algorithm was formulated for use with forward mapping functions [Wolberg 90]. We extend it to work with inverse mapping as well.

4.1. WALG: Forward Mapping

In the forward mapping formulation of WALG, an input pixel is treated as a unit-length area sample that can be transformed into an output interval that may lie fully embedded in an output pixel or may straddle several output pixels. In the first case, the input intensity is weighted by its partial contribution to the output pixel, and then deposited into an accumulator. The accumulator will ultimately be stored in the output array only when the input interval passes across the rightmost boundary of an output pixel, assuming that the algorithm proceeds from left to right. In the second case, the input pixel actually crosses, or straddles, at least one output pixel boundary. A single input pixel may give rise to a "left straddle" if it occupies only a partial output pixel before it crosses its first output boundary from the left side. As long as the input pixel continues to fully cover output pixels, it is said to be in the "central interval." Finally, the last partial contribution to an output pixel on the right side is called a "right straddle." A software implementation in C is given below.

4.1.1. Software Implementation

```
wolbergfwd(double f[], int in[], int out[], int inlen, int outlen)
{
   int     u,               /* index into input  image           */
           x,               /* index into output image           */
           ul,              /* index of the  leftmost input pixel */
           ur,              /* index of the rightmost input pixel */
           ix0,             /* index of the left side of interval */
           ix1;             /* index of the rightside of interval */
   double  x0,              /* index of the left side of interval */
           x1,              /* index of the rightside of interval */
           dI;              /* intensity increment  over interval */

   /* clear output/accumulator array */
   for (x=0; x<outlen; x++) out[x] = 0;

   /* check if no input scanline pixel projects to [0,outlen] range */
   if(f[inlen] < 0 || f[0] > outlen) return;      /* 100% clipping */

   /* init ul, the left-extent of the input scanline    */
   for(u=0; f[u]<0; u++);                  /* advance u */
   ul = u;                                 /* init   ul */

   /* process first interval: clip left straddle */
   if(u > 0) {
          u   = ul - 1;
          x0  = f[u];
          x1  = f[u+1];
```

```
        ix0 = x0 - 1;
        ix1 = x1;

        /* central interval; will be clipped for x<0 */
        dI = (in[u+1] - in[u]) / (x1 - x0);
        for(x=0; x<ix1 && x<outlen; x++)
             out[x] = in[u] + dI*(x-x0);

        /* right straddle */
        if(ix1!=x1 && ix1<outlen)
             out[ix] = (in[u] + dI*(ix1-x0)) * (x1-ix1);
}
/* init, ur the right-extent of the input scanline   */
for(u=inlen; f[u]>outlen; u--);          /* advance u */
ur = (u == inlen) ? inlen-1 : u;         /* init   ur */

/* check if only one input pixel covers scanline */
if(u == ul) return;

/* process last interval: clip right straddle */
if(u < inlen) {
       ix0 = x0 = f[u];     /* int- and real-valued left  index */
       ix1 = x1 = f[u+1];   /* int- and real-valued right index */
       /* left straddle  */
       out[ix0] += in[u] * (ix0-x0+1);

       /* central interval: will be clipped for x>=outlen */
       dI = (in[u+1] - in[u]) / (x1 - x0);
       for(x=ix0+1; x<ix1 && x<outlen; x++)
            out[x] = in[u] + dI*(x-x0);
}
/* main loop */
for(u=ul; u<=ur; u++) {
       ix0 = x0 = f[u];     /* int- and real-valued left  index   */
       ix1 = x1 = f[u+1];   /* int- and real-valued right index   */

       /* check if interval is embedded in one output pixel       */
       if(ix0 == ix1) {
            out[ix1] += in[u] * (x1-x0);    /* accumulate pixel   */
            continue;                       /* next input pixel   */
       }

       /* left straddle */
       out[ix0] += in[u] * (ix0-x0+1);         /* add input fragment */

       /* central interval */
       dI = (in[u+1] - in[u]) / (x1 - x0);     /* for linear intrp   */
       for(x=ix0+1; x<ix1; x++)                /* visit  all  pixels */
            out[x] = in[u] + dI*(x-x0);        /* init output pixel  */

       /* right straddle */
```

```
            if(x1 != ix1)                                 /* output pxl fragment*/
                out[ix1] = (in[u] + dI*(ix1-x0)) * (x1-ix1);
    }
}
```

It is clear from the above description that the processing of a specific interval is independent of the processing of other intervals. The processing of an individual interval only requires the position of its left and right boundaries. Thus, the algorithm has potential for parallelism.

Note that unlike FALG, interpolation of the input intensity is not performed unless it is necessary, i.e., while processing the central interval and the right straddle. We can do the same in FALG by adding the test `inseg == sf` to the interpolation statement, and setting the intensity to $in[u]$ if the condition is true, or setting it to the interpolated value otherwise. Also note that the zero cycles are eliminated in this approach. Thus, WALG is faster than FALG when implemented in software. The serial software implementation of the algorithm can be further accelerated by adopting incremental computation for the central interval. Thus in the main loop, as well as in the preprocessing, the two lines for computing the central interval

```
        for(x=ix0+1; x<ix1; x++)
                out[x] = in[u] + dI*(x-x0);
```

can be rewritten as

```
        if(ix1>ix0+1) {
                out[ix0+1] = in[u] + dI*(ix0+1-x0);
                for(x=ix0+2; x<ix1; x++)
                        out[x] = out[x-1] + dI;
        }
```

This replaces the multiplication operations with addition. However, for a parallel implementation, we keep the cells decoupled as much as possible, and, therefore, the replacement does not take place.

An alternate approach to preprocessing intervals that are mapped outside the allowable range $[0, outlen]$ is to associate output assignments inside the main loop with guard conditions. This is slower when implemented in software, but it is more suitable for a hardware implementation. A software implementation in C follows:

```
wolbergfwd_guards(double f[], int in[], int out[], int inlen, int outlen)
{
   int     u,              /* index into input  image           */
           x,              /* index into output image           */
           ix0,            /* index of the left side of interval */
           ix1;            /* index of the rightside of interval */
   double  x0,             /* index of the left side of interval */
           x1,             /* index of the rightside of interval */
```

```
           dI;              /* intensity increment  over interval */

   /* clear output/accumulator array */
   for (x=0; x<outlen; x++) out[x] = 0;

   /* main loop */
   for(u=0; u<inlen; u++) {
          ix0 = x0 = f[u];        /* int- and real-valued left  index   */
          ix1 = x1 = f[u+1];      /* int- and real-valued right index   */

          /* check if interval is embedded in one output pixel          */
          if(ix0==ix1 && ix0>=0 && ix0<outlen) {
               out[ix1] += in[u] * (x1-x0);    /* accumulate pixel   */
               continue;                       /* next input pixel   */
          }

          /* left straddle */
          if(ix0>=0 && ix0<outlen)
               out[ix0] += in[u] * (ix0-x0+1); /* add input fragment */

          /* central interval */
          dI = (in[u+1] - in[u]) / (x1 - x0);     /* for linear intrp   */
          for(x=ix0+1; x>=0 && x<ix1 && x<outlen; x++)  /* visit pixels */
               out[x] = in[u] + dI*(x-x0);     /* init output pixel  */

          /* right straddle */
          if(ix1>=0 && ix1<outlen && x1 != ix1)   /* output pxl fragment*/
               out[ix1] = (in[u] + dI*(ix1-x0)) * (x1-ix1);
   }
}
```

4.2. WALG: Inverse Mapping

In the inverse mapping formulation of WALG, an output pixel is treated as a unit-length area sample that can be transformed into an input interval that may lie fully embedded in an input pixel, or may straddle several input pixels. In the first case, the input intensity is computed at the left boundary of the embedded interval. Since this position may not necessarily coincide with an input pixel boundary, interpolation may be necessary to yield the value of the current output pixel. In the second case, a single output pixel may map to an input interval that consists of a left straddle, a central interval, and/or a right straddle.

For the left straddle, the input intensity is interpolated at the left boundary of the left straddle and then scaled by the length of the left straddle. This intensity contribution is deposited in the accumulator associated with that output pixel. For the central interval, no interpolation or scaling is necessary,

since the central interval starts on input pixel boundaries and covers whole input pixels. Thus, the intensity of the input pixels in the central interval are simply added to the accumulator. For the right straddle, no interpolation is needed, but scaling by the length of the right straddle is required before adding the final intensity contribution to the accumulator.

Note that the accumulated sum needs to be divided by the *isf* entry of the current output pixel to yield the correct output intensity value. This division is not necessary when the output pixel is embedded in the input pixel, unless we scale the interpolated intensity by the length of the embedded interval, as is done in the hardware implementation.

A software implementation in C follows. We chose the implementation with preprocessing of values of mapping outside the allowable range $[0, inlen]$. An implementation with guard conditions could be coded as in the preceding section.

```
wolberginv(double f[], int in[], int out[], int inlen, int outlen)
{
   int     u,                /* index into input  image           */
           x,                /* index into output image           */
           xl,               /* index of the  leftmost output pixel*/
           xr,               /* index of the rightmost output pixel*/
           iu0,              /* index of the left side of interval */
           iu1;              /* index of the rightside of interval */
   double  u0,               /* index of the left side of interval */
           u1,               /* index of the rightside of interval */

   /* clear output/accumulator array */
   for (x=0; x<outlen; x++) out[x] = 0;

   /* check if no output scanline pixel projects to [0,inlen] range */
   if(f[outlen] < 0 || f[0] > inlen) return;       /* 100% clipping */

   /* init ul, the left-extent of the input scanline    */
   for(x=0; f[x]<0; x++);                 /* advance x */
   xl = x;                                /* init   xl */

   /* process first interval: clip left straddle */
   if(x > 0) {
       x   = xl - 1;
       iu0 = u0 = f[x];        /* int- and real-valued left  index   */
       iu1 = u1 = f[x+1];      /* int- and real-valued right index   */
       /* central interval; will be clipped for x<0 */
       for(u=0; u<iu1 && u<inlen; u++)
               out[x] += in[u];

       /* right straddle */
       if(iu1!=u1 && iu1<inlen)
               out[x] += (in[iu1] * (u1-iu1);
```

```
        /* normalize accumulator */
        out[x] /= (f[x+1] - f[x]);
    }

    /* init, ur the right-extent of the input scanline   */
    for(x=outlen; f[x]>inlen; x--);         /* advance x */
    xr = x;                                 /* init   xr */

    /* check if only one output pixel covers scanline */
    if(x == xl) return;

    /* process last interval: clip right straddle */
    if(x < outlen) {
           iu0 = u0 = f[x];        /* int- and real-valued left  index */
           iu1 = u1 = f[x+1];      /* int- and real-valued right index */

           /* left straddle  */
           out[x] = (in[iu0] + (u0-iu0)*(in[iu0+1]-in[iu0])) * (iu0-u0+1);

           /* central interval: will be clipped for x>=outlen */
           for(u=iu0+1; u<iu1 && u<inlen; u++)
                out[x] += in[u];

           /* normalize accumulator */
           out[x] /= (f[x+1]-f[x]);
    }
    /* main loop */
    for(x=xl; x<xr; x++) {
        iu0 = u0 = f[x];        /* int- and real-valued left  index   */
        iu1 = u1 = f[x+1];      /* int- and real-valued right index   */

        /* check if interval is embedded in one output pixel          */
        if(iu0 == iu1) {
             out[x] += in[iu0] + (u0-iu0)*(in[iu0+1]-in[iu0]);
             continue;                       /* next input pixel   */
        }

        /* left straddle */
        out[x] = (in[iu0] + (u0-iu0)*(in[iu0+1]-in[iu0])) * (iu0-u0+1);

        /* central interval */
        for(u=iu0+1; u<iu1; u++)                /* visit  all  pixels */
             out[x] += in[u];

        /* right straddle */
        if(u1 != iu1)                           /* output pxl fragment*/
             out[x] += in[iu1] * (u1-iu1);

        out[x] /= (f[x+1]-f[x]);
    }
}
```

5. Hardware Implementations

This section presents hardware implementations for FALG and WALG in order to clarify the similarities and differences between the forward and inverse implementations of each algorithm. The integrated hardware implementations permit FALG and WALG to perform resampling under either forward or inverse mappings.

5.1. FALG: Hardware for Inverse and Forward Mappings

The inverse and forward mapping formulations of FALG are shown integrated in Figure 1. The figure helps to show that only minor modifications are necessary to switch between the forward and inverse mappings. Note that there are two control signals, *input cycle* and *output cycle*, that govern the execution of the circuit. A control signal *Get new input pixel* is generated when an input cycle is terminated. The control signal *Mapping* is used to control the initialization of *inseg*, *outseg*, and the final value of the accumulator.

Once a new input pixel is fetched, it is passed onto the *interpolate* block, shown in Figure 2. The interpolated value is then normalized and added to the accumulator, where it contributes to the output pixel. The new output pixel value is available when an output cycle is terminated.

In the software implementation, the values of the mapping were clipped against the allowable ranges: $[0, inlen]$ for inverse mapping and $[0, outlen]$ for

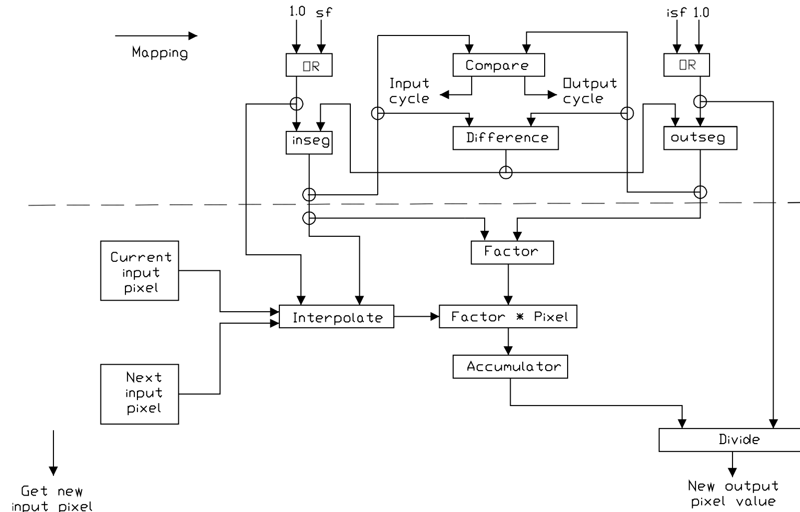

Figure 1. Integrated FALG resampling algorithm.

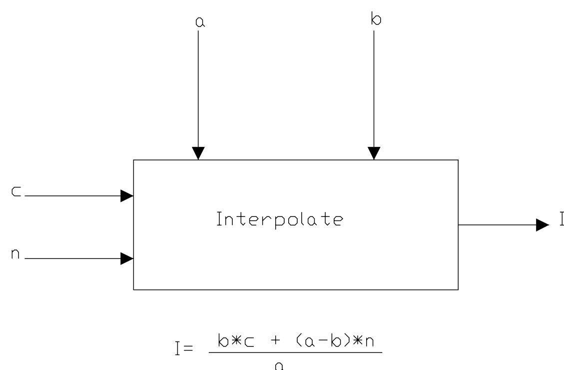

Figure 2. The interpolate block.

forward mapping, This clipping, as well as the initialization of *inseg* and *outseg* at the start of the process, are not shown in Figure 1 to keep the diagram simple.

5.2. WALG: Hardware for Inverse Mappings

This section presents a parallel hardware implementation for WALG when the inverse mapping is given. The motivation for this section is two-fold. First, resampling given the inverse mapping is very common. Second, integrating both the forward and inverse mapping in WALG will require some changes to the straightforward implementation presented in this section and is not as simple as in FALG. These changes are due to the fact that when given the forward mapping, a single output pixel accumulator can have values coming from more than one interval (cell). The hardware is illustrated in Figure 3.

Note that there are four control signals (modes) for the circuit: **E** (embedded interval), **L** (left straddle), **C** (central interval), **R** (right straddle). Each mode occurs depending on the values of $u0$, $iu0$, and $iu1$ as follows:

- **E**: $iu0 == iu1 \ \&\& \ u0 \geq iu0$;
- **L**: $iu0 < iu1 \ \&\& \ u0 \geq iu0$;
- **C**: $iu0 < iu1 \ \&\& \ u0 < iu0$;
- **R**: $iu0 == iu1 \ \&\& \ u0 < iu0$.

In the **E** mode, interpolation is performed with the values 1.0 and $(iu0+1)-u0$, and factor is set to $u1-u0$. Note, that unlike the software implementation,

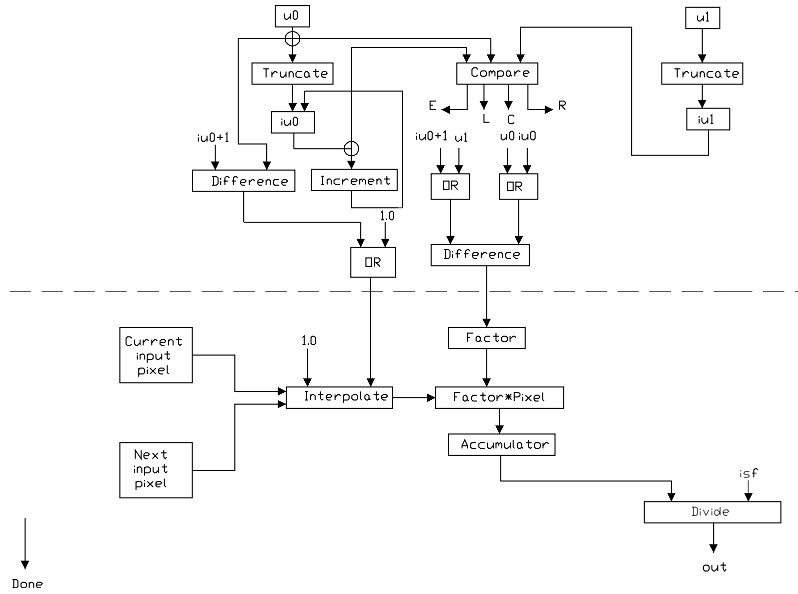

Figure 3. Inverse mapping WALG for a single cell (output pixel).

we multiply by $u1 - u0$, and then divide by $isf = u1 - u0$. Finally, a *Done* signal is generated.

In the **L** mode, interpolation is performed with the values 1.0 and $(iu0 + 1) - u0$, and factor is set to $(iu0 + 1) - u0$. Finally, $iu0$ is incremented.

In the **C** mode, interpolation is performed with the values 1.0 and 1.0, i.e., no actual interpolation takes place. Factor is set to $(iu0 + 1) - iu0$ and $iu0$ is incremented.

In the **R** mode, interpolation is performed with the values 1.0 and 1.0, and factor is set to $u1 - iu0$. Finally, a *Done* signal is generated. Note, that in all cases, the current input pixel is indexed by $iu0$.

The upper half of the diagram illustrates the extra overhead used to replace *inseg* and *outseg* by the interval boundaries $u0$ and $u1$, and thereby parallelize FALG. Note that the lower half of the diagram is identical to that originally presented by Fant. Maximum parallelism is achieved when there are as many cells as there are output pixels. For simplicity, the preprocessing and guard conditions are omitted from the diagram.

In the hardware implementation of WALG, a single circuit is used to process all types of intervals (**E**, **L**, **C**, and **R**) to reduce cost and size. As a result, some redundant calculations are performed while processing certain types of intervals. In the case of the software implementation, we treat each type of interval separately, and we avoid redundant computations.

5.3. *WALG: Hardware for Inverse and Forward Mappings*

The block diagram for the integrated forward-inverse formulation of WALG is similar to that of inverse mapping. The integrated hardware, however, has the accumulator value written through a data bus, and has an address bus contain either the address of the current cell (inverse mapping) or *ileft* (forward mapping). In the latter case, *ileft* corresponds to $iu0$ in 3. This method properly handles the forward mapping case where the the accumulator value may come from multiple cells (intervals).

Interpolation is done according to Table 1. In the table a and b are as defined in Figure 2. The values *left, right* and *iright* correspond to *u0, u1* and *iu1*, respectively, in Figure 3.

The factors multiplied by pixel are the same under both the inverse mapping and the forward mapping. Also, incrementing *ileft* and generating the *Done* signal are the same in both mappings.

The current input pixel index is given by either the address of the current cell (forward mapping) or by *ileft* (inverse mapping). The accumulator is divided by *isf* = *right-left* for inverse mapping or by 1.0 (i.e., no division) for forward mapping. This division takes place when the *Done* signal of the corresponding cell is generated. However, in the case of forward mapping, the *out* value of a particular cell is available only after the *Done* signals of all cells are generated. This must be the case, since the *Done* signal refers to an input pixel, and not to an output pixel as in the case of inverse mapping.

	Forward mapping	Inverse mapping
E	$a = 1.0, b = 1.0$	$a = 1.0, b =$ *(ileft+1)-left*
L	$a = 1.0, b = 1.0$	$a = 1.0, b =$ *(ileft+1)-left*
C	$a =$ *right-left*, $b =$ *right-ileft*	$a = 1.0, b = 1.0$
R	$a =$ *right-left*, $b =$ *right-iright*	$a = 1.0, b = 1.0$

Table 1. Interpolation table.

6. Equivalence of the Two Algorithms

In this section, we demonstrate the equivalence of the two algorithms. We consider the case when the inverse mapping is given. The forward mapping case can be handled similarly.

First, we consider the case when an output pixel maps to an interval that is fully embedded in an input pixel. In this case, $outseg = isf$, and $inseg > outseg$. This condition generates an output cycle whereby the accumulator is set to the interpolated input intensity value multiplied by *outseg*, and *out* is set to the accumulator divided by *isf*. However, since *isf* = *outseg*, *out* is simply set to the interpolated intensity as is done in WALG. Note,

that in the software implementation of WALG, we avoid the multiplication and the division operations which cancel each other, thereby accelerating the computation. The interpolated value is the same in both algorithms, since *inseg* starts at 1, and gets reduced by *outseg* to keep track of the remaining portion of the input pixel, i.e., the left boundary of the embedded interval.

When an output pixel straddles several input pixels, we consider each interval separately. For the left straddle, $outseg = isf$ and $outseg > inseg$. This condition generates an input cycle whereby the accumulator is set to the interpolated input intensity value multiplied by *inseg*. The interpolation applied here is identical to the embedded case, thus the result is the same for both algorithms. Furthermore, the value of *inseg* is equal to the length of the left straddle, i.e., the portion of the input pixel that is available to contribute to the current output pixel. Hence, this input cycle of FALG yields the same value in the accumulator as the one produced by processing the left straddle in WALG.

For the central interval, $inseg = 1$, and it remains less than *outseg*. Hence, for each input pixel in the central interval an input cycle occurs, and the interpolated input intensity value is multiplied by *inseg* and added to the accumulator. The same computation is performed by WALG, except that it avoids the multiplication by *inseg* (1.0) and the unnecessary interpolation.

Finally, for the right straddle, $outseg < 1$ while $inseg = 1$. This condition generates an output cycle whereby the interpolated input intensity value multiplied by *outseg* is added to the accumulator. Once more, WALG avoids the unnecessary interpolation that will be performed in FALG. Since *outseg* represents the portion of the input pixel that is required to complete the next output pixel, it is equal to the length of the right straddle. Thus, the value added to the accumulator is identical in both algorithms.

The above argument is valid for an arbitrary output pixel. Therefore, the two algorithms produce identical output scanlines. The above analysis also shows that a software implementation of WALG is faster than FALG. This superior performance is enhanced further by the elimination of the zero cycles.

7. Conclusions

This paper has reviewed FALG and WALG, two algorithms for one-dimensional resampling, and has demonstrated that they are equivalent under both forward and inverse mappings. Software and hardware solutions were presented. In addition to being faster for a serial implementation, WALG has been shown to have potential for parallelism. Parallelism of one-dimensional resampling can be achieved by treating each interval spanned by the output (input) pixels separately. This method has the additional advantage of decoupling errors made among intervals, thereby avoiding the potential problem of accruing roundoff errors due to the continued subtraction of *inseg* and *outseg* in FALG.

Integrated hardware implementations that perform one-dimensional resampling under either forward or inverse mappings were presented for both algorithms based on their software descriptions. For the hardware implementations given, the parallel Wolberg formulation has a Fant-like basic block per cell. The cell in the parallel algorithm, however, has extra hardware compared to Fant's basic block. This extra hardware is needed to achieve parallelism.

References

[Catmull, Smith 80] Edwin Catmull and Alvy Ray Smith. "3-D Transformations of Images in Scanline Order." *Computer Graphics (Proc. SIGGRAPH '80)* 14(3): 279–285 (1980).

[Fant 86] Karl M. Fant. "A Nonaliasing, Real-Time Spatial Transform Technique." *IEEE Computer Graphics and Applications* 6(1): 71–80 (January 1986). See also "Letters to the Editor" in 6(3): 66-67 (March 1986) and 6(7): 3, 8 (July 1986).

[Smith 87] Alvy Ray Smith. "Planar 2-Pass Texture Mapping and Warping." *Computer Graphics (Proc. SIGGRAPH '87)* 21(4): 263–272 (1987).

[Williams 83] Lance Williams. "Pyramidal Parametrics." *Computer Graphics (Proc. SIGGRAPH '83)* 17(3): 1–11 (1983).

[Wolberg 90] George Wolberg. *Digital Image Warping.* Los Alamitos, CA: IEEE Computer Society Press, 1990.

[Wolberg, Boult 89] George Wolberg and Terrance E. Boult. "Image Warping with Spatial Lookup Tables." *Computer Graphics (Proc. SIGGRAPH '89)* 23(3): 369–378 (1989).

Web Information:

http://www.acm.org/jgt/papers/WolbergEtAl00

George Wolberg, Department of Computer Science, City College of New York, New York, NY 10031 (wolberg@cs.ccny.cuny.edu)

H. M. Sueyllam, Department of Computer Science, Alexandria University, Alexandria 21544, Egypt (sueyllam@yahoo.com)

M. A. Ismail, Department of Computer Science, Alexandria University, Alexandria 21544, Egypt (engdean@netscape.net)

K. M. Ahmed, Department of Computer Science, Alexandria University, Alexandria 21544, Egypt (drkhalil@usa.net)

Received April 28, 1998; accepted in revised form November 28, 2000.