# Polygon Scan Conversion Derivations*
# Caltech-CS-TR-91-12

Kurt Fleischer †

October 27, 1995

This report is a supplement to *Accurate Polygon Scan Conversion Using Half-Open Intervals* [Fleischer & Salesin]. It is assumed that the reader is familiar with that Graphics Gems III article, and the code included in its appendix. The material here is meant to be read in that context, and may be difficult to understand without it. Note: the Graphics Gems code is available via anonymous ftp [C Code].

This report includes:

- a more detailed derivation of the EdgeScan algorithm (for both integer and subpixel vertices),
- some explanatory diagrams,
- and short discussions concerning underflow and overflow in the fixpoint arithmetic for the subpixel version of the algorithm.

*For copies, contact the Computer Science Library, Caltech MS 256-80, Pasadena, CA, 91125.

†Dept of Computation and Neural Systems, Mail Stop 350-74, California Inst of Technology, Pasadena, CA, 91125.

# Contents

# 1 Forward

This Technical Report is informally written and meant to provide more details to the serious reader than were available in the short format of the Graphics Gem. Please contact me with corrections, or if you are really stumped with some part of the procedure `kurt@mailhost.gg.caltech.edu`.

The fixed point arithmetic package is included only to make the subpixel scan conversion possible; I do not encourage its use for other purposes, since it is not particularly efficient or complete. I was unable to find a public domain fixed point package that served my purposes, so I cooked this one up.

I'd like to thank David Salesin (co-author of the Graphics Gem [Fleischer & Salesin]) for his contributions towards a simple and clear form for the EdgeScan algorithm. Thanks also to Denis Zorin for thoughtful comments on the manuscript.

# 2 Introduction

The EdgeScan algorithm solves the following problem: given an edge $e$, compute the $x$-coordinate of the closest pixel whose center lies on or to the left of $e$, for each scanline that intersects $e$.[1] Note: we define pixel centers to lie on integer grid values.

The EdgeScan algorithm is used in our polygon scan conversion routines. It is closely related to Bresenham's line drawing algorithm [Bresenham]. The derivation closely follows Sproull's paper [Sproull]. Successive steps in developing the algorithm are presented in C, with the addition of a variable type *real* to describe unimplementable real numbers in the initial algorithm definitions.

In the Graphics Gem, we showed the algorithm as two routines: EdgeSetup and EdgeScan. For clarity, in this report we develop the algorithm as one routine, then separate it into the two routines at the last stage.

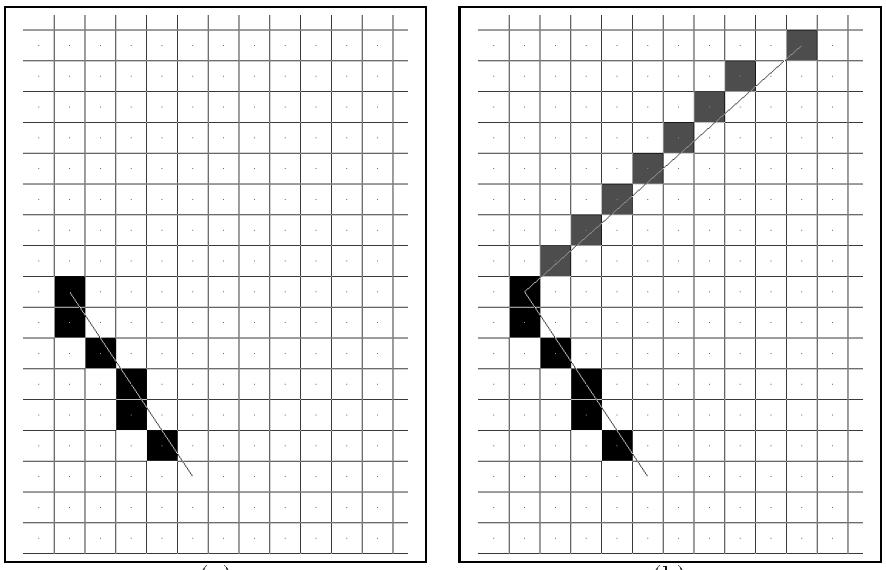
# 3 Example Figures

The figures for Example 1 on the following page present an example of the scan conversion for a particular triangle with integer vertices. Example 2, on the subsequent page, shows a triangle with subpixel vertices.

[1] Definitions: an *edge* is a line segment, and a *scanline* is a line $y$ = integer.

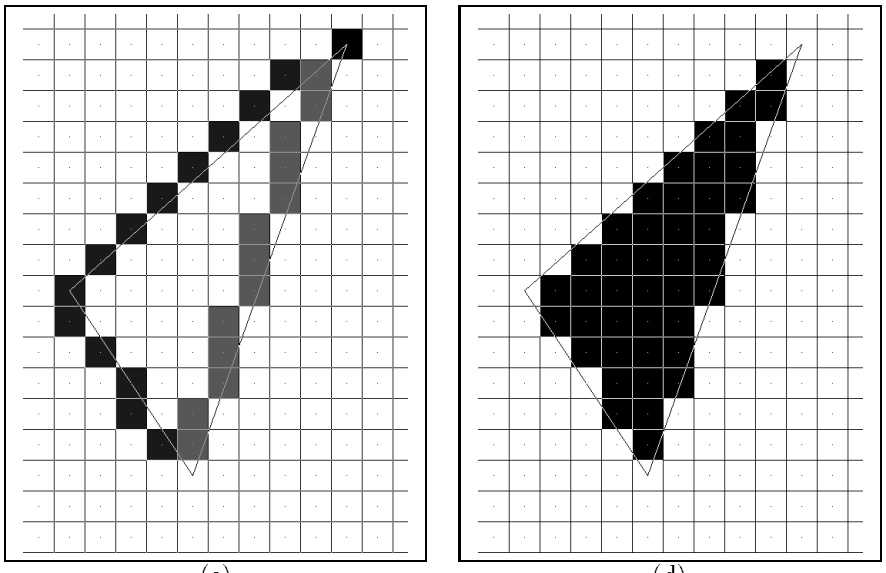
## Example 1: Triangle with integer pixel vertices: (5, 2) (1, 8) (10, 16)

Note: The origin (0, 0) is the lower left corner.
Note: Integer vertices are at pixel centers.



(a) (b)

Figure (a). Edge ((5, 2), (1, 8)). The EdgeScan algorithm marks one pixel on each scanline. It finds the closest pixel on or to the left of the edge for every scanline in the half-open interval $(y_{min}, y_{max}]$. Thus the initial pixel (at $y_{min} = 2$) is not marked, but the final pixel (at $y_{max} = 8$) is marked.
Figure (b). The two edges ((5, 2), (1, 8)) and ((1, 8), (10,16)) form the left side of the triangle.

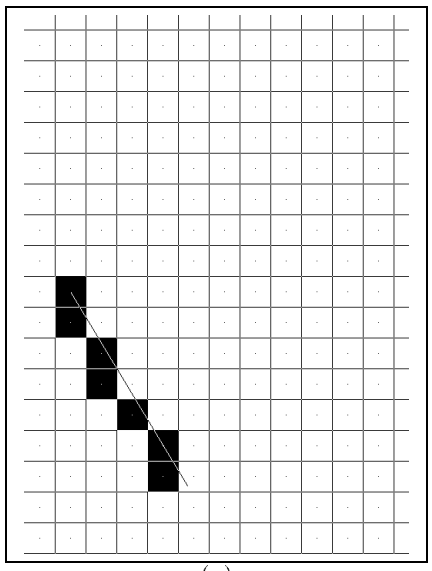


(c) (d)

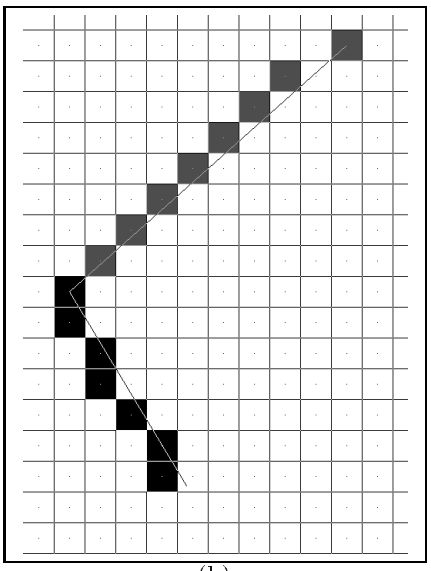
Figure(c). The two left edges are shaded darker, the right edge is shaded lighter, and the single overlapping pixel (contained in both left and right edges) is shaded black. Spans (horizontal runs of pixels) will be drawn on the half-open intervals $(x_{left}, x_{right}]$, from the left edge up to and including the right edge. Thus the pixels contained in the left edge will not be drawn, since they are the start of the half-open interval in $x$. Overlapping pixels, for example pixel $(10, 16)$, will *not* be drawn in the final triangle, since the half-open interval $(a, a]$ is empty.
Figure (d). The final triangle ((5, 2), (1, 8), (10, 16)).

## Example 2: Triangle with subpixel vertices: (4.8, 1.7) (1, 8) (10, 16)

The lower vertex has been modified, and is no longer at integer values. The figures are analogous to those of Example 1.



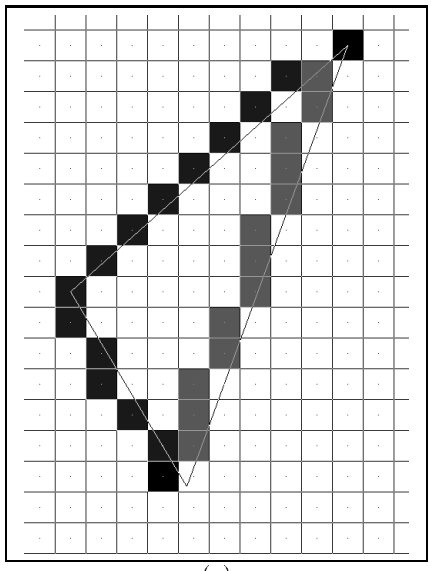
(a)



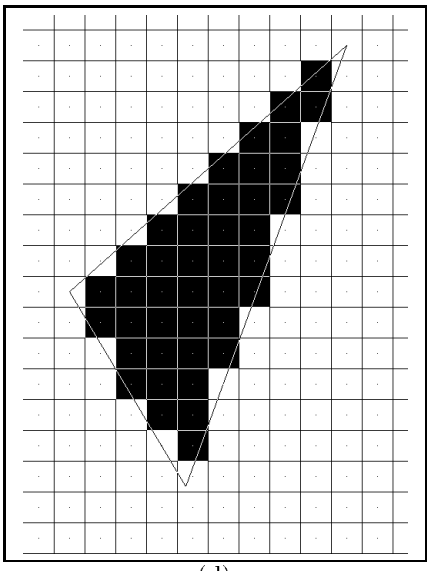
(b)

Figure (a).

Figure (b).



(c)



(d)

Figure(c). Now there are two pixels that are contained in both left and right edges: (4, 2) and (10, 16). As before, these do not appear in the final triangle, since the half-open interval between left and right edges along each of those scanlines is empty.

Figure (d).

# 4 Derivation of the EdgeScan Algorithm

I strongly encourage the reader to take a look at Sproull's paper [Sproull], which shows how to use program transformations to derive scan conversion algorithms. We use these techniques below.

The EdgeScan algorithm solves the following problem: given an edge $e$, compute the $x$-coordinate of the closest pixel whose center[2] lies *on or to the left of* $e$, for each scanline that intersects $e$. An edge is defined to be a line segment `((x0,y0), (x1,y1))`.

## 4.1 Algorithm 0: Infinite Precision

We start with the following code that uses infinite precision real numbers to compute the intersection of an edge with each scanline at infinite precision. This intersection is denoted `(x, yi)` for each scanline `yi`. The closest pixel on or to the left of the scanline intersection is `(floor(x), yi)`.

`y0int` is the y-intersection of the edge with the first scanline in the half-open interval `(y0, y1]`. In these notes, `int` stands for "intersection".

The edge is computed for all scanlines in the half-open interval `(y0, y1]`. We assume here that the vertices are sorted so that `y0 < y1`. If `y0 = y1`, then half-open interval `(y0, y1]` is empty, and the routine need not be called. These conditions are checked as assertions in the final version of the code.

```
EdgeSetupAndScan(x0,y0,x1,y1)
real x0, y0, x1, y1;
{
  real dx = x1-x0;
  real dy = y1-y0;
  int yi, y0int, y1int;

  /* y0int is the y-intercept with next scan line */
  y0int = floor(y0) + 1;
  y1int = floor(y1) + 1;

  for (yi = y0int; yi < y1int; yi++) {
         x = x0 + (dx/dy) * (yi - y0);  /* compute x-intercept */
         draw_point(floor(x), yi);
  }
}
```

[2] Note: we define pixel centers to lie on integer grid values.

## 4.2 Algorithm 1: Incremental Computation

Here we continue to use "real" variables, but we compute `x` incrementally in the inner loop. `x0int` is the x-intersection of the edge with the first scanline in the half-open interval `(y0, y1]`. Thus `(x0int, y0int)` is the point of intersection of the edge with the first scanline in that interval.

```
/*
 *      Incrementally compute x.
 */
EdgeSetupAndScan(x0,y0,x1,y1)
real x0, y0, x1, y1;
{
  real x0int;
  real dx = x1-x0;
  real dy = y1-y0;
  int yi, y0int, y1int;

  y0int = floor(y0) + 1;
  y1int = floor(y1) + 1;
  x0int = x0 + (dx/dy) * (y0int - y0);

  x = x0int;
  for (yi = y0int; yi < y1int; yi++) {
         draw_point(floor(x), yi);
         x += (dx/dy);
  }
}
```

### 4.3 Algorithm 2: Separate integer and fractional parts

Representing each real number as two parts, integer and fractional, prepares us for further optimizations later. The fractional part is still shown as an unimplementable 'real' number.

```
/*
 Split x and slope (real numbers) into integer + real using
    x = (xi + xf)
    slope = dx/dy = si + sf

 using definitions (w/similar representation for slope as si, sf)
    xi = floor(x)
    xf = x - xi

 Note: xi is integer part, (positive or negative),
       0 <= xf < 1  represents fractional part, non-negative
 (This choice of representation for x is convenient for a
  conditional test introduced in algorithm 5. See Alg 5 for details.)
*/
EdgeSetupAndScan(x0,y0,x1,y1)
real x0, y0, x1, y1;
{
  real x0int;
  real dx = x1-x0;
  real dy = y1-y0;
  int yi, y0int, y1int;
  real xf, sf;
  int xi, si;

  y0int = floor(y0) + 1;
  y1int = floor(y1) + 1;
  x0int = x0 + (dx/dy) * (y0int - y0);

  /* xf and sf are always nonnegative and <=1 */
  xi = floor(x0int);
  xf = x0int - xi;
  si = floor(dx/dy);
  sf = dx/dy - si;

  for (yi = y0int; yi < y1int; yi++) {
     draw_point(xi, yi);
     xf += sf;
     xi += si;

     /* if xf overflows (>=1), transfer integer part to xi */
     /* we know here 1 <= xf < 2  */
     if (xf >= 1) {
        xi += 1;
        xf -= 1;
     }
  }
}
```

### 4.4 Algorithm 3: Shuffle adds

Rather than adding two things into `xi` and `xf` when the condition `(xf >= 1)` is true, we consolidate the add operations in one branch of the conditional. (This can remove about 2 adds per iteration if `si+1` and `sf-1` are stored in local variables.)

```
/* Consolidate adds inside conditional (removes 2 adds per iteration). */
EdgeSetupAndScan(x0,y0,x1,y1)
real x0, y0, x1, y1;
{
  real x0int;
  real dx = x1-x0;
  real dy = y1-y0;
  int yi, y0int, y1int;
  real xf, sf;
  int xi, si;

  y0int = floor(y0) + 1;
  y1int = floor(y1) + 1;
  x0int = x0 + (dx/dy) * (y0int - y0);

  xi = floor(x0int);
  xf = x0int - xi;
  si = floor(dx/dy);
  sf = dx/dy - si;

  for (yi = y0int; yi < y1int; yi++) {
         draw_point(xi, yi);
         if (xf + sf >= 1) {
                xi += si + 1;
                xf += sf - 1;
         }
         else {
                xi += si;
                xf += sf;
         }
  }
}
```

## 4.5 Algorithm 4: Avoid division

Change the variables `x0int`, `xf`, and `sf` by multiplying through by `dy`. Thus `x0int_dy = x0int * dy`. We modify the meanings of `xf` and `sf` without changing their names, since we like the short names. Note also that the condition must compare to `dy` instead of 1, and `xf` is incremented by `dy` also.

NOTE: we can compute `floor(x/y)` quickly for `x,y` $\in$ integer, as will be shown in the Final Integer Algorithm on page 13. This is used to compute `xi` and `si` below.

```
/*  Multiply x0int, xf and sf by dy to avoid doing divides */
EdgeSetupAndScan(x0,y0,x1,y1)
real x0, y0, x1, y1;
{
  real x0int_dy;
  real dx = x1-x0;
  real dy = y1-y0;
  int yi, y0int, y1int;
  real xf, sf;
  int xi, si;

  y0int = floor(y0) + 1;
  y1int = floor(y1) + 1;
  x0int_dy = x0*dy + dx * (y0int - y0);

  xi = floor(x0int_dy/dy);
  xf = x0int_dy - xi*dy;
  si = floor(dx/dy);
  sf = dx - si * dy;

  for (yi = y0int; yi < y1int; yi++) {
         draw_point(xi, yi);
         if (xf + sf >= dy) {
                xi += si + 1;
                xf += sf - dy;
         }
         else {
                xi += si;
                xf += sf;
         }
  }
}
```

## 4.6 Algorithm 5: Another variable definition

Define a new `real` variable `r = xf + sf - dy`, which enables us to remove `xf`, and to make the condition compare to zero (faster on many machines). As mentioned in Algorithm 2, the chioce of representation for `x` as signed `xi` and non-negative `xf` enables this to be a test against 0.

The resulting algorithm contains no operations that will generate fractional results if the input vertices are integers. Thus all 'real' variables can be implemented as integers, as shown on the next page.

A subpixel algorithm can also be derived from this, as described on page 17.

```
/* Define r = xf + sf - dy to make condition on 0, and remove xf. */
EdgeSetupAndScan(x0,y0,x1,y1)
real x0, y0, x1, y1;
{
  real x0int_dy;
  real dx = x1-x0;
  real dy = y1-y0;
  int yi, y0int, y1int;
  real r, sf;
  int xi, si;

  y0int = floor(y0) + 1;
  y1int = floor(y1) + 1;
  x0int_dy = x0*dy + dx * (y0int - y0);

  xi = floor(x0int_dy/dy);
  si = floor(dx/dy);
  sf = dx - si * dy;

  r = x0int_dy - xi*dy + sf - dy;

  for (yi = y0int; yi < y1int; yi++) {
         draw_point(xi, yi);
         if (r >= 0) {
                xi += si + 1;
                r += sf - dy;
         }
         else {
                xi += si;
                r += sf;
         }
  }
}
```

# 5 Algorithm for Integer Vertices

## 5.1 Final Algorithm for Integer Vertices

If the inputs are integers, we can modify Algorithm 5 appropriately. We no longer need the `real` variables, so we end up with an implementable algorithm.

```
/* Integer Algorithm.

Since x0, y0, x1, y1 are integers, y0int, y1int and x0int are
easy to compute:

    y0int = floor(y0) + 1 = y0 + 1
    y1int = floor(y1) + 1 = y1 + 1
    x0int_dy = x0*dy + dx * (y0int - y0)
             = x0*dy + dx
substitute these values below.

Also, we can simplify a bit by replacing x0int_dy with
its definition, and noting that
 xi = floor(x0int_dy/dy) = x0 + floor(dx/dy)
                         = x0 + si

We introduce here an auxiliary function floor_div(a,b)
that computes floor(a/b) quickly for integers a and b
(using standard truncation).

*/

/* floor(x/y). Assumes y>0. */
int floor_div(int x, int y)
{
  assert(y>0);
  if (x >= 0) return(x/y);
  else return((x/y) + (((x % y) == 0) ? 0 : -1));
}

EdgeSetupAndScan(x0,y0,x1,y1)
int x0, y0, x1, y1;
{
  int dx = x1-x0;
  int dy = y1-y0;
  int yi, y0int, y1int;
  int r, sf;
  int xi, si;

  if (y0 == y1) return;   /* no scanlines in half-open interval */
  assert (y0 < y1);       /* vertices should be sorted */

  y0int = y0 + 1;
  y1int = y1 + 1;

  si = floor_div(dx,dy);  /* this is floor(dx/dy) */
  xi = x0 + si;
  sf = dx - si * dy;
```

```
  r = 2*sf - dy;      /*  r = (x0*dy + dx - xi*dy) + sf - dy; */

  for (yi = y0int; yi < y1int; yi++) {
        draw_point(xi, yi);
        if (r >= 0) {
              xi += si + 1;
              r += sf - dy;
        }
        else {
              xi += si;
              r += sf;
        }
  }
}
```

## 5.2 Integer Vertex EdgeSetup and EdgeScan

Here we introduce a C structure to store the values, so we can split the code up into two routines, `EdgeSetup()` and `EdgeScan()`. To compute the an edge, we call the setup routine once, and then repeatedly call the `EdgeScan()` to iterate along the edge.

This code is identical to that in the Graphics Gem [Fleischer & Salesin, C Code]. A usage example is given in Section 5.3, and also in the Graphics Gems code.

For clarity in the final code, we substitute the names `ymin`, `ymax` for `y0int-1`, `y1int-1`. This change is reflected in the loop start and end conditions when using these routines, as in Section 5.3.

For horizontal edges, `y0 = y1` and no pixels will be marked since the half-open interval in `y` is empty. `EdgeScan()` will not be called in this case, and `EdgeSetup()` can therefore omit computing several values, and avoid dividing by zero.

```
struct edge {
  int ymin, ymax, xi, si;
  int r, inc, dec;
};

/*
*  Initializes the Bresenham-like scan conversion for a single edge,
*  setting values in the structure containing the increment variables.
*/
struct edge *EdgeSetup(struct edge *e, int x0, int y0, int x1, int y1)
{
  int sf, dx = x1-x0, dy = y1-y0;
  assert(y0 <= y1);
  e->ymin = y0;
  e->ymax = y1;
  if (y0 != y1) {   /* avoid divide by zero */
    e->si = floor_div(dx, dy);
    e->xi = x0 + e->si;
    sf = dx - e->si * dy;
    e->r = 2*sf - dy;
    e->inc = sf;
    e->dec = sf - dy;
  }
  return(e);
}

/*  Returns the intersection of edge e with the next scanline.
*/
int EdgeScan(struct edge *e)
{
  int x = e->xi;

  if (e->r >= 0) {
    e->xi += e->si + 1;
    e->r += e->dec;
  }
  else {
    e->xi += e->si;
    e->r += e->inc;
  }
  return x;
}
```

## 5.3 Using EdgeSetup and EdgeScan

Here is an example using `EdgeSetup()` and `EdgeScan()`. Recall that it draws the closest pixel whose center lies on or to the left of the edge `((x0, y0), (x1, y1))`, for every scanline that intersects the edge in the interval `(y0, y1]`.

```
void draw_edge(int x0, int y0, int x1, int y1)
{
  struct edge e;
  int yi, x;

  /* sort the vertices so y1 >= y0 */
  int tmp;
  if (y0>y1) { SWAP(y0,y1,tmp); SWAP(x0,x1,tmp); }
  EdgeSetup(&e, x0, y0, x1, y1);

  /* Scan edge. */
  for (yi = e.ymin + 1; yi <= e.ymax; yi++) {
    x = EdgeScan(&e);
    draw_point(x, yi);
  }
}
```

# 6 Algorithm for Subpixel Vertices

## 6.1 Final Algorithm for Subpixel Vertices

This version is also derived from Algorithm 5, and computes to subpixel accuracy, using fixed point numbers (I like to call them "fixpoint" numbers).[3]

As with the final algorithm for integer vertices (page 13), we start with some knowledge about the inputs and then propagate this through the code to implement all variables as `int`s or `fixpoint`s instead of `real`s.

The statements with prefix "`fp`" are calls to a fixpoint arithmetic package (functions are listed on page 20). The function `fp_floor_div(a,b)` computes $\lfloor \frac{a}{b} \rfloor$ (it cannot overflow as used here; see discussion on page 21).

It requires a bit of manipulation to get the following algorithm from Algorithm 5; I haven't noted all intermediate steps here. New temporary variables `alpha`, `beta`, `xi_` and `si_` are introduced to avoid recomputing things in the setup phase.

The computation of the quantity `alpha` is done in fixed point representation with twice as may bits as the other computations (`dblfixpoint`). The extra computation is unfortunate, but is only necessary once at the beginning of the edge.

See pages 21 and 22 for discussions about overflow and underflow.

**[Algorithm on next page]**

[3] A *fixpoint* number has a fixed number of bits representing its integer part and its fractional part. There is a little more detail about this on page 21

```
EdgeSetupAndScan(x0,y0,x1,y1)
fixpoint x0,y0,x1,y1;
{
  fixpoint y0int, y1int, r, inc, dec;
  fixpoint dx = x1-x0;
  fixpoint dy = y1-y0;
  fixpoint xi_, si_;    /* temporary variables (fixpoint xi and si) */
  fixpoint alpha, beta; /* temporary variables */
  fixpoint sf;
  int xi, si, yi;

  y0int = fp_floor(y0) + fp_fix(1.0);
  y1int = fp_floor(y1) + fp_fix(1.0);

  if (y0int != y1int) {
     /* alpha is computed at double precision, then truncated (this
      * computes the subpixel containing the x-intersection of the
      * edge with the scanline.
      * Note: x0 is split into integer and fractional parts here.
      */
    alpha = fp_trunc(fp_dbladd(fp_dblmultiply(fp_fraction(x0), dy),
                               fp_dblmultiply(dx, y0int - y0)));
    beta = fp_floor_div(alpha, dy);
    xi_ = fp_floor(x0) + beta;
    /* If dy < 1, then these variables are not used since
       edge crosses at most one scanline. We also avoid
       causing underflow that could occur with dy<1. (Sec. 9) */
    if (dy >= fp_fix(1.0)) {
      si_ = fp_floor_div(dx, dy);
      sf = dx - fp_multiply(si_, dy);

      /* (alpha - beta*dy) = fractional part of the x-intercept. */
      r = alpha - fp_multiply(beta, dy) + sf - dy;
      si = fp_integer(si_);
      inc = sf;
      dec = sf - dy;
    }
    xi = fp_integer(xi_);
  }

  for (yi = fp_integer(y0int); yi < fp_integer(y1int); yi++) {
         draw_point(xi, yi);

         if (r >= 0) {
                xi += si + 1;
                r += dec;
         }
         else {
                xi += si;
                r += inc;
         }
  }
}
```

## 6.2 Subpixel vertex version of `EdgeSetup` and `EdgeScan`

As we did for the integer vertex version (page 15), we introduce a structure to store the values, so we can split the code up into two routines, `EdgeScan()` and `EdgeSetup()`. We also substitute `ymin`, `ymax` for `y0int-1`, `y1int-1`, as in the integer version.

It turns out that `EdgeScan()` for the subpixel vertex version is *identical* to the integer vertex version, and is omitted here. **Thus computing edges with subpixel endpoints is as efficient as with integer endpoints! The only additional computation is in the setup phase.**

The `edge` structure is also unchanged from the integer vertex version. I am using an implementation-dependent fact here, ie that the `int` variables `r`, `inc` and `dec` can hold fixpoint values, and that addition and subtraction with fixpoint values is identical to that for integers.

The temporary variables `alpha`, `beta`, `si_` and `xi_` are used only in setup, and aren't needed in the `edge` structure.

```
struct edge *
SubEdgeSetup(struct edge *e,
     fixpoint x0, fixpoint y0, fixpoint x1, fixpoint y1)
{
  fixpoint ymin, ymax, alpha, beta, sf, xi, si;
  fixpoint dx = x1-x0, dy = y1-y0;
  assert(y0 <= y1);
  ymin = fp_floor(y0);
  ymax = fp_floor(y1);

  if ((dy != 0) && (ymin != ymax)) {
    /* alpha is computed at double precision, then truncated (this
     * computes the subpixel containing the x-intersection of the
     * edge with the scanline. */
    /* fp_fix(1.0) can be replaced by the appropriate constant */
    alpha =
      fp_trunc(fp_dbladd(fp_dblmultiply(fp_fraction(x0), dy),
                         fp_dblmultiply(dx, ymin - y0 + fp_fix(1.0))));
    beta = fp_floor_div(alpha, dy);
    xi = fp_floor(x0) + beta;

    /* If dy < 1, then these variables are not used since
       edge crosses at most one scanline. We also avoid
       causing underflow that could occur with dy<1. (Sec. 9) */
    if (dy >= fp_fix(1.0)) {
      si = fp_floor_div(dx, dy);
      sf = dx - fp_multiply(si, dy);

      /* (alpha - beta*dy) = fractional part of the x-intercept. */
      e->r = alpha - fp_multiply(beta, dy) + sf - dy;
      e->si = fp_integer(si);
      e->inc = sf;
      e->dec = sf - dy;
    }
    e->xi = fp_integer(xi);
  }
  e->ymin = fp_integer(ymin);
  e->ymax = fp_integer(ymax);

  return(e);
}
```

# 7 Fixed Point Arithmetic Package (`fixpoint`)

The code for the fixpoint package is in the Graphics Gem [Fleischer & Salesin], and is available with the rest of the code [C Code]. However, it is not a particularly lovely implementation and is provided only to complete the presentation of the subpixel triangle algorithm.

For maximum efficiency in production code (or hardware), you should write an appropriate machine-specific implementation of fixed point arithmetic. Sadly, these are not generally distributed as part of standard computational libraries.

Here's a brief description of the data types and routines used:

**fixpoint** data type for fixed point numbers (specified number of bits for integer and fraction parts).

**int fp_integer(fixpoint x)** return the integer part of a fixpoint number (truncation)

**int fp_fraction(fixpoint x)** return the fractional part of a fixpoint number ($x - \lfloor x \rfloor$) as an integer.

**fixpoint fp_floor(fixpoint x)** $\equiv \lfloor x \rfloor$

**fixpoint fp_floor_div(fixpoint x, fixpoint y)** for $y >= 0$, return $\lfloor \frac{x}{y} \rfloor$

**fixpoint fp_multiply(fixpoint x, fixpoint y)** multiply two numbers, truncates low order bits.

**fixpoint fp_fix(double x)** convert a double into a fixpoint number

Some computations require greater precision, so we also need to define double length fixed point, with twice as many bits. The routines are analogous to the other fixpoint routines, with the addition of:

**dblfixpoint** data type for fixed point numbers with twice as many bits (specified number of bits for integer and fraction parts).

**fixpoint fp_trunc(dblfixpoint a)** create a fixpoint by truncating a dblfixpoint.

# 8 Avoiding Overflow in the Arithmetic Operations

This section considers the possibility of overflow in the subpixel algorithm.

Since the fixpoint code was written for demonstration purposes, we were not overly concerned with tight bounds on the number of bits used to encode the numbers. With careful analysis we probably could get away with one fewer bit for the same screen resolution. Here is a description of the implementation.

How many bits do we need for the fixpoint numbers? At least enough to represent the number of subpixels across the screen. When performing some computations, we may temporarily need more bits. For example, when adding two numbers about the size of the screen resolution, something like $dx + dx$, we need at least $n + 1$ bits. I chose to use fixpoint numbers with $n + 2$ bits to represent an array with $2^n$ subpixels (using twos-complement representation). This probably wastes a bit, but is sufficient for demonstrating that `EdgeScan()` works.

One multiply operation occurs in the computation of $\alpha$ (described below), and we must use twice as many bits to hold the results. This is implemented using the `dblfixpoint` numbers.

Another multiply operation occurs in the computation of `sf = dx - fp_multiply(si_, dy)`, where `fp_multiply(si_, dy)` $= (\lfloor \frac{dx}{dy} \rfloor) \cdot dy$. This term is clearly close to $dx$, so there is no danger of overflow, and we do not need to use `dblfixpoint`.

We must also consider overflow for the computations involving `fp_floor_div(a,b)` $\equiv \lfloor \frac{a}{b} \rfloor$:

1. computation of $\lfloor \frac{\alpha}{dy} \rfloor$ that is used in `xi_` and `r`
2. computation of $\lfloor \frac{dx}{dy} \rfloor$ used in `si_`

## Computation of $\lfloor \frac{\alpha}{dy} \rfloor$ used in `xi_` and `r`

If there is no intersection between the edge and a scanline, then no line is drawn, so we avoid overflow by not setting up the incremental computations. Otherwise, we know that the first intersection with a scan line must occur in the edge, so ($\lfloor y0 \rfloor + 1 <= y1$). Thus, $dy = y1 - y0 >= \lfloor y0 \rfloor + 1 - y0$.

The rest follows easily. Let $\{x0\}$ be the fractional part of $x0$, which must be $< 1$ by definition. Then

$$\begin{aligned}
\lfloor \frac{\alpha}{dy} \rfloor &= \lfloor \frac{\{x0\} \cdot dy + dx(\lfloor y0 \rfloor + 1 - y0)}{dy} \rfloor \\
&= \lfloor \{x0\} + dx \cdot \frac{(\lfloor y0 \rfloor + 1 - y0)}{dy} \rfloor \\
&= \lfloor \{x0\} + dx \cdot (something_less_than_or_equal_to_one) \rfloor \\
&< dx + 1
\end{aligned}$$

So, no overflow. (When comparing this discussion to the code, recall that `y0int` $= \lfloor y0 \rfloor + 1$.)

## Computation of $\lfloor \frac{dx}{dy} \rfloor$ used in `si_`

Split into two cases:

1. If ($dy >= 1$) then there is no overflow since $dx/dy < dx$.
2. else ($dy < 1$), so at most one span will be drawn, and EdgeScan will never be called... so we can avoid computing this term. Hence the condition `(dy >= fp_fix(1.0))` in the code.

# 9 Avoiding Underflow in the Arithmetic Operations

This section considers the possibility of underflow in the subpixel algorithm. Underflow can occur when multiplying two fixpoint numbers, so we will consider in turn each multiply in the algorithm.

## 9.1 Underflow not possible if multiplicand is an integer

In both of the following statements,

```
sf = dx - fp_multiply(si_,dy)
r = alpha - fp_multiply(beta, dy) + sf - dy
```

one of the multiplicands is the result of a `floor` operation, and has no fractional part (no low order bits). Thus these multiplications can't produce underflow.

## 9.2 Why truncation of the `dblfixpoint` multiply is OK.

Another multiply occurs when $\alpha$ is computed. Since we use `dblfixpoint` routines to compute it (with twice as many bits), we have no underflow...yet. But then we truncate with `fp_trunc()`, throwing away the lowest bits of the `dblfixpoint`.

Why is it ok to truncate alpha to a single fixpoint? Note that $\alpha$ is used only in the initial computation of `xi` and `r`, which is essentially finding the initial subpixel of the edge. We consider both of these cases:

### Case I: Computation of `xi`

In computing `xi`, we use $\beta = \lfloor \frac{\alpha}{dy} \rfloor$. The truncation of $\alpha$ cannot affect $\beta$, since no values of the truncated bits of $\alpha$ could cause it to cross the next integral $dy$ boundary (remember that $dy$ is a single fixpoint).

### Case II: Computation of `r`

Recall from the code that `r = alpha - fp_multiply(beta, dy) + sf - dy;`

The only term that is affected by the truncation is $\alpha$ (`alpha`), so `r` might be off by the amount truncated.

Since `r` is always incremented by `si` and `sf`, which are `fixpoint` numbers, an initial offset in the low order bits of a `dblfixpoint` can never have an effect...it is a constant sub-subpixel offset.

This truncation of underflow corresponds to subpixel quantization in computing the x-intersection of the edge with the scanline, and can't be avoided. However, it only produces artifacts for degenerate databases, in which adjacent polygons don't necessarily have coincident vertices (see [Lathrop, Kirk, & Voorhies] for a discussion).

# 10 References

# References

[Bresenham] Bresenham, J. E., *Algorithm for Computer Control of a Digital Plotter,* IBM Systems Journal **4**(1), 25–30, 1965.

[Fleischer & Salesin] Fleischer, Kurt and David Salesin, *Accurate Polygon Scan Conversion Using Half-Open Intervals,* Graphics Gems III, Academic Press 1992.

[C Code] Fleischer, Kurt. Code for the Graphics Gem [Fleischer & Salesin] is available via anonymous ftp from ftp.princeton.edu in pub/Graphics/GraphicsGems/GemsIII/*.

[Lathrop, Kirk, & Voorhies] Lathrop, Olin, Kirk, David, and Voorhies, Doug *Accurate Rendering by Subpixel Addressing,* IEEE Computer Graphics and Applications, pp 45–53, **10**(5), 1990.

[Sproull] Sproull, Robert F., *Using Program Transformations to Derive Line-Drawing Algorithms,* ACM Transactions on Graphics (1)4:257–273, October 1982.