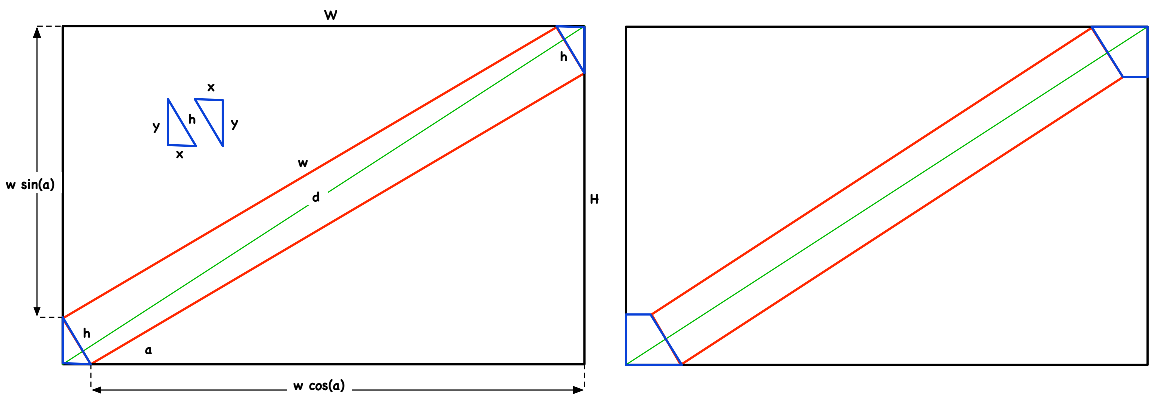
# Drawing a string along the diagonal of a given rectangle: ‘non-centered’ method

Wagner L. Truppel

August 27, 2006

Say that your view rectangle is $W$ pixels wide and $H$ pixels tall and you want to draw a slanted string in the general direction of the view’s diagonal. You need to find the rectangle inside which to draw the string. In the figure below, that rectangle is drawn in red, with width $w$ and height $h$, and the view’s diagonal is drawn in green (with length $d$). On the left picture, note the two identical triangles at the corners (drawn in blue, and reproduced separately for clarity), with the given dimensions. Note also that $a$ is the angle by which the string rectangle is inclined, but it’s generally *not* the angle by which the diagonal is inclined.



This is different from the ‘centered’ solution, shown on the right, in that this time the string rectangle is not always centered along the view rectangle’s diagonal. The advantage of the non-centered method is that it allows for longer strings than the centered method, for a given font height.

So, what is known? As before, we only know $W$ and $H$.

What’s needed? Just as before, we need to determine $w$, $h$, $x$, and $a$, and an appropriate font size to draw the string with. $w$ and $h$ determine the string rectangle, and $x$ determines the anchor point around which the string rectangle must be rotated (counterclockwise, by the angle $a$).

The general strategy to solve the problem remains the same: choose a font size and use it to determine $w$, from which we'll obtain all other variables. We then compare the computed value of $h$ with the height of the tallest glyph of the string, for the chosen font size: if $h$ is smaller, the tallest glyph is too tall, and we must decrease the font size. If, instead, $h$ is larger than the height of the tallest glyph, we can attempt to increase the font size. Either way, $w$ is changed (along with the values of all the other variables) and we must repeat the procedure just described until we converge on the optimal font size: the largest font size for which the string rectangle's width $w$ is larger than, but as close as possible to, the actual string width and the string rectangle's height $h$ is larger than, but as close as possible to, the height of the tallest glyph. Alternatively, we can choose to work with $h$ instead, then compute $w$, compare it with the actual string length, and so on. Either way, this increasing and decreasing of the font size can be done most efficiently through a simple binary search.

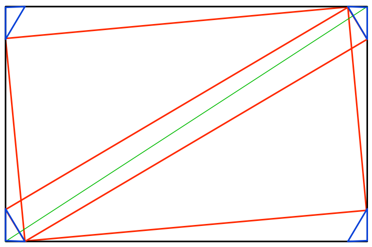
Once again, we'll start with some equalities derived from considering various similar triangles:

$$\begin{aligned}\sin(a) &= \frac{H-y}{w} &= \frac{x}{h}\,,\\ \cos(a) &= \frac{W-x}{w} &= \frac{y}{h}\,,\\ \tan(a) &= \frac{H-y}{W-x} &= \frac{x}{y}\,.\end{aligned}$$

From the last equality, we can determine $y$ in terms of $x$ and known quantities:

$$\frac{H-y}{W-x} = \frac{x}{y} \quad \Rightarrow \quad x\,(W-x) = y\,(H-y) \quad \Rightarrow \quad y_{1,2} = \frac{H}{2} \pm \sqrt{(\frac{H}{2})^2 - x\,(W-x)}\,.$$

Note that there are generally two possible solutions, corresponding to the two rectangles shown below, for a given $x$. Note also that the two solutions are related by $y_1 + y_2 = H$, but have different values of $h$ and $w$.



Using the result above, we obtain

$$\frac{h}{w} = \frac{x}{H-y} \qquad \Rightarrow \qquad x = \frac{h}{w}\,(H-y) = \frac{W-(w/h)\,H}{1-(w/h)^2}\,,$$

after a little algebra. Then, solving for $y$, we see that there's a unique solution for a given string rectangle's aspect ratio $w/h$:

$$x = \frac{W-(w/h)\,H}{1-(w/h)^2} \quad \text{and} \quad y = \frac{H-(w/h)\,W}{1-(w/h)^2} \qquad \Rightarrow \qquad \tan(a) = \frac{x}{y} = \frac{W-(w/h)\,H}{H-(w/h)\,W}\,.$$

Note that these results are valid regardless of whether $W \geq H$ or $W \leq H$. This is true because of the symmetric nature of the inscribed rectangle.

It would seem now that we have everything we need since, for a given font size, we can compute both $w$ and $h$ and, thus, get a unique solution. There's a problem, however. It may be the case that, for a given font and font size, the string rectangle's aspect ratio $w/h$ is such that, for the view rectangle in question, $x$ or $y$, or even both, is outside the view rectangle. Worse still, they can have infinite values (if $w = h$).

In other words, for a given view rectangle, font size, and string, there may *not* exist a solution after all. We already knew that, of course, and the idea is to try different font sizes until we zero-in on a solution.

It's easy to find out what conditions the string rectangle's aspect ratio $w/h$ must satisfy in order for a solution to exist:

- if $w/h > 1$:

$$\begin{array}{rcl c rcl}
x & \geq & 0 & \text{iff} & w/h & \geq & W/H\,, \\
x & \leq & W & \text{iff} & w/h & \geq & H/W\,, \\
\\
y & \geq & 0 & \text{iff} & w/h & \geq & H/W\,, \\
y & \leq & H & \text{iff} & w/h & \geq & W/H\,,
\end{array}$$

- if $w/h < 1$:

$$\begin{array}{rcl c rcl}
x & \geq & 0 & \text{iff} & w/h & \leq & W/H\,, \\
x & \leq & W & \text{iff} & w/h & \leq & H/W\,, \\
\\
y & \geq & 0 & \text{iff} & w/h & \leq & H/W\,, \\
y & \leq & H & \text{iff} & w/h & \leq & W/H\,.
\end{array}$$

Thus, we can guarantee that both $x$ and $y$ are within the view rectangle by requiring that:

- if $w/h > 1$:
$$w/h \quad \geq \quad \frac{\max(W,H)}{\min(W,H)}\,,$$
- if $w/h < 1$:
$$w/h \quad \leq \quad \frac{\min(W,H)}{\max(W,H)}\,.$$

Nonetheless, the issue remains of how to effectively impose these conditions. Both $w$ and $h$ are dependent on the actual string to be drawn, and on the font and font size with which to draw it. It's not possible to **separately** control $w$ and $h$, and thus neither their ratio. We do have control over the choice of font size but, although both $w$ and $h$ are monotonic functions of the font size (they both increase as the font size increases), **their ratio is not**. The problem is even worse for proportional fonts, since changing the font size does not change the string's aspect ratio, which means that $x$ and $y$ are **fixed** for a given string and wiew rectangle, for such fonts.

The bottom line is that we can't use the string's aspect ratio; we need to be able to control either $w$ or $h$ so we can perform a binary search on the font size — we need to obtain $x$ (or $y$) as a function of $w$ alone or of $h$ alone.

In order to do that, let's start by imposing the condition that $x$ and $y$ must both lie within the view rectangle, that is, let's impose that

$$x = r \times W \qquad \text{and} \qquad y = s \times H\,,$$

with both $r$ and $s$ constrained to be in the open interval $(0, 1)$.

Next, assuming $h$ to be known, $x$ and $y$ must satisfy

$$x^2 + y^2 = h^2 \qquad \Rightarrow \qquad r^2\,W^2 + s^2 H^2 = h^2\,.$$

We need another equation, of course, so we can eliminate $s$. The only independent equation we have which does not contain $w$ is

$$\frac{H-y}{W-x} = \frac{x}{y}\,,$$

which, as we saw, leads to

$$y = \frac{H}{2} \pm \sqrt{(\frac{H}{2})^2 - x\,(W-x)}\;.$$

In terms of $r$ and $s$, this result reads

$$s = \frac{1}{2} \pm \sqrt{\frac{1}{4} - r\,(1-r)\,(\frac{W}{H})^2}\;.$$

Inserting this into $r^2\,W^2 + s^2 H^2 = h^2$ allows us to obtain a polynomial equation for $r$ in terms of only $W$, $H$, and $h$:

$$p(r) \equiv 4W^4\,r^4 - 4W^4\,r^3 + W^2(d^2 - 4h^2)\,r^2 + 2W^2h^2\,r + h^2(h^2 - H^2) = 0\,,$$

where $d$ is the length of the view rectangle's diagonal, $d \equiv \sqrt{W^2 + H^2}$. We can simplify matters considerably by defining the ratios

$$a \equiv (\frac{H}{W})^2 \qquad \text{and} \qquad b \equiv (\frac{h}{W})^2\,,$$

in terms of which our polynomial can be written as

$$p(r) \equiv r^4 - r^3 + \frac{(1 + a - 4b)}{4}\,r^2 + \frac{b}{2}\,r + \frac{b\,(b-a)}{4} = 0\,.$$

Our task is now to find the (real) roots of this equation, in the open interval $(0, 1)$. Since the coefficients of the different powers of $r$ are all real, we know that the *total* number of real roots is either 0, 2, or 4. Unfortunately, there's no general method to solve a polynomial equation of degree 4, which means we'll have to resort to a numerical method.

The simplest procedure is to divide the interval $(0, 1)$ into a fixed number of sub-intervals and compute the value of $p(r)$ at their extremes. Whenever the sign of $p(r)$ changes from one extreme to the next, we know that there is a root in that sub-interval, in which case we can use a straight-forward bissection method to zero-in on the root. The problem, of course, is that if the number of sub-intervals is not large enough, we may miss roots that may in fact exist. Since we're interested in drawing strings on a computer screen, however, we can use a self-adjusting division scheme so that $x$ never changes by more than a few pixels from one sub-interval to the next. In most cases, this is sufficient to guarantee that we'll find a solution if there's one.

To summarize, then, we'll follow the general strategy outlined on page 2, using $h$ as our base quantity: choose a font size, compute $h$, search for a root of $p(r)$, compute $x$ and $y$ if a root is found, then compute $w$, then compare $w$ with the actual string width, and change the font size accordingly, repeating until we converge on the optimal font size. And, of course, whenever we do not find a root of $p(r)$, we also have to change the font size and repeat.

Let's now outline a strategy to decide whether $p(r)$ has a root and, if so, how to obtain it. First, recall that $x = r\,W$, so a change in $r$ by an amount $\Delta r$ implies a change in $x$ by the amount

$$\Delta x = W\,\Delta r\,.$$

If we want $\Delta x$ to be no more than $K$ pixels large, then we need $\Delta r$ to be equal to

$$\Delta r = \frac{\Delta x}{W} = \frac{K}{W}\,,$$

giving us a sub-division of the interval $[0, 1]$ which guarantees that every $K$-th pixel on the $x$ axis is tested as a potential solution:

$$r_n = n\,\Delta r = \frac{K}{W}\,n\,, \qquad n = 0, 1, 2, \ldots, \lceil \frac{W}{K} \rceil\,.$$

Now, suppose that while scanning the interval $[0, 1]$ we find a sub-interval $[r_n, r_{n+1}]$ such that $p(r_n)\,p(r_{n+1}) < 0$, meaning that $p(r)$ has a different sign on each extreme of that interval. We know then that $p(r)$ crosses the $r$-axis somewhere inside the interval, that is, we know that $p(r)$ has a root there. We can then do a binary search for that root: look at the sign of $p(r)$ for the mid-point of the interval and repeat, converging on the root. Here's some pseudo-code for a general function implementing this idea:

```
// a function that returns the smallest real root r, in the open interval
// (ra, rb), of a function f(r) continuous and bounded in the interval given
// but otherwise arbitrary. If no such root exists, the function returns the
// upper end of the interval, rb, as a marker for no root found. Function
// parameters other than ra and rb are: the step size dr, and the root
// tolerance tol (two otherwise distinct values are considered equal if
// the absolute value of their difference is less than or equal to tol).

float find_root(float tol, float dr, float ra, float rb)
{
    float dab = rb - ra; // make sure that dr is small enough
    if (dr >= dab)       // so we can step from ra to rb
    { dr = dab / 100; }  // we'll split (rb - ra) in 100 parts

    float r1 = ra;       // the lower end of a given sub-interval
    float r2 = ra + dr;  // the upper end of a given sub-interval

    float p1;            // the value of f(r) at r = r1
    float p2;            // the value of f(r) at r = r2
    while (r2 <= rb)
    {
        p1 = f(r1);
        if (|p1| < tol and r1 > ra and r1 < rb) // we found a root
        { return r1; }

        p2 = f(r2);
        if (p1 * p2 < 0.0) // we found a sub-interval bracketing a root
        {
            float oldr, r, p;
            oldr = r1;
               r = r2;
            while (|r - oldr| > tol)
            {
                oldr = r;

                r = (r1 + r2) / 2.0;
                p = f(r);

                if (p * p1 < 0.0)
                {
                    r2 = r;
                    p2 = p;
                }
                else if (p * p2 < 0.0)
                {
                    r1 = r;
                    p1 = p;
                }
                else // p = 0.0, so r is a root
                { break; }
            }
            return r; // the root we converged on
        }
        r1 = r2;   // move on to the
        r2 += dr;  // next sub-interval
    }
    return rb; // no root found
}
```

The function above depends on the function `f(float r)` which computes the value of $f(r)$. For our polynomial, it would be implemented as follows (presumably, $A$, $B$, and $C$ are visible to $f$):

```
float f(float r)
{
    return (((r - 1.0) * r + A) * r + B) * r + C;
}
```

We now have all we need to draw a slanted string in the general direction of the view rectangle's diagonal. Here's a section of pseudo-code that determines the optimal font size, the string rectangle's width $w$ and height $h$, $x$ and $y$:

```
  FS_TOL = 0.1;             // some appropriately small positive number
ROOT_TOL = 0.0001;          // some appropriately small positive number
       K = 1.0;             // determines dr

MINhOVERd = 0.05;           // minimum allowed value of h/d
MAXhOVERd = 0.25;           // maximum allowed value of h/d

MINROOT = 0.05;             // minimum allowed value of x/W
MAXROOT = 0.95;             // maximum allowed value of x/W

float d = sqrt(W*W + H*H); // length of view rectangle's diagonal
float dr = K/W;             // the step in scanning p(r) for roots

float a = (H/W);
a *= a;                     // a = (H/W)^2

float oldfs, sw, w, h, r, x, y;

float minfs = MINFS;        // minimum font size
float maxfs = MAXFS;        // maximum font size
float fs = maxfs;           // initial font size

while (true)
{
    oldfs = fs;
    fs = (minfs + maxfs) / 2.0;
```

```
if (|fs - oldfs| < FS_TOL)
{ break; } // we’ve converged on the optimal font size, so we’re done

// h = height of tallest glyph for the current font and font size
h = appropriate Cocoa call here

float hod = h / d;
if (hod < MINhOVERd) // reject the current font size if h too small
{
    minfs = fs; // increase font size
    continue;
}
if (hod > MAXhOVERd) // reject the current font size if h too large
{
    maxfs = fs; // decrease font size
    continue;
}

float b = h/W;
b *= b;    // b = (h/W)^2

float A = (1.0 + a - 4.0 * b) / 4.0;
float B = b / 2.0;
float C = b * (b - a) / 4.0;

r = find_root(ROOT_TOL, dr, 0.0, MAXROOT);

if (r == MAXROOT) // no solution found
{
    maxfs = fs; // decrease font size
    continue;
}

// if need be, search for a larger root
float rprime = r;
while (rprime < MINROOT)
{ rprime = find_root(ROOT_TOL, dr, rprime, MAXROOT); }

// if we found a better (larger) root, use it; otherwise,
// stick with the sub-optimal one we found before.
```

```
    if (rprime < MAXROOT)
    { r = rprime; }

    x = r * W;
    y = sqrt(h*h - x*x);

    float Wmx = W - x;
    float Hmy = H - y;
    w = sqrt(Wmx * Wmx + Hmy * Hmy);

    // sw = string length in pixels for the current font and font size
    sw = appropriate Cocoa call here

    if (sw > w) // string too long
    { maxfs = fs; } // decrease font size
    else
    { minfs = fs; } // increase font size
}
```

At this point, we've determined the width ($w$) and height ($h$) of the rectangle inside of which the string will be drawn, as well as the optimal font size for the string in question. We can then use these values to center the string both vertically and horizontally inside this rectangle, using the appropriate Cocoa calls. We've also found $x$ and $y$, the location of the anchor point around which to rotate the string rectangle.

Finally, position the string rectangle such that its lower left vertex lies at position $(x, 0)$, then rotate it counterclockwise by an angle $\arctan(x/y)$. Here are some sample results, from an actual application implementing the algorithm above: