donut.c without a math library

Jan 13, 2021

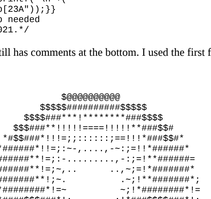
My little donut.c has been making the rounds again, after being featured in a couple YouTube videos (e.g., Lex Fridman and Joma Tech). If I had known how much attention this code would get over the years, I would have spent more time on it.

One thing that's always been sort of unfortunate is the heavy use of `sin` and `cos` – both because it necessitates linking the math library (`-lm`), but also because it makes it much more CPU-intensive than it really needs to be. This is especially apparent if you try to port it to an older CPU or an embedded device.

So, here's a revised version with no use of `sin`, `cos`, and no need for linking the math library (though this version still does use `float` types).

```
            i,j,k,x,y,o,N;
        main(){float z[1760],a
      #define R(t,x,y) f=x;x-=t*y\
   ;y+=t*f;f=(3-x*x-y*y)/2;x*=f;y*=f;
    =0,e=1,c=1,d=0,f,g,h,G,H,A,t,D;char
 b[1760];for(;;){memset(b,32,1760);g=0,
h=1;memset(z,0,7040);for(j=0;j<90;j++){
G=0,H=1;for(i=0;i<314;i++){A=h+2,D=1/(G*
A*a+g*e+5);t=G*A       *e-g*a;x=40+30*D
*(H*A*d-t*c);y=        12+15*D*(H*A*c+
t*d);o=x+80*y;N        =8*((g*a-G*h*e)
*d-G*h*a-g*e-H*h      *c);if(22>y&&y>
 0&&x>0&&80>x&&D>z[o]){z[o]=D;b[o]=(N>0
  ?N:0)["",.,-~:;=!*#$@"];}R(.02,H,G);}R(
  .07,h,g,);for(k=0;1761>k;k++)putchar
   (k%80?b[k]:10);R(.04,e,a);R(.02,d,
     c);usleep(15000);printf("\n'+(
        " donut.c! \x1b[23A"));}}
         /*no math lib needed
           .@a1k0n 2021.*/
```

It's a little misshapen and still has comments at the bottom. I used the first frame of its output as a template and there's *slightly* less code than filled pixels – oh well. Output is pretty much the same as before:



Defining a rotation

So, how do we get sines and cosines without using `sin` and `cos`? Well, the code doesn't really *need* sine and cosine *per se*; what it actually does is rotate a point around the origin in two nested loops, and also rotate two angles just for the animation. If you'll recall from the other article, the inner loop is just plotting dots in a circle, which goes around another, larger circle. In each loop, the sine/cosine terms are just moving by a small, fixed angle.

So we don't need to track the *angle* at all, we only need to start at cos=1, sin=0 and rotate a circle around the origin to generate all the sines and cosines we need. We just have to repeatedly apply a fixed rotation matrix:

$$\begin{bmatrix} c' \\ s' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} c \\ s \end{bmatrix}$$

[c's']=[cosθ−sinθsinθcosθ][cs]

So for example, if we were to use an angle of .02 radians in our inner loop, it would look something like:

```
float c=1, s=0;  // c for cos, s for sin
for (int i = 0; i < 314; i++) {  // 314 * .02 ~= 2π
  // (use c, s in code)
  float newc = 0.9998*c - 0.019998666*s;
  s = 0.019998666*c + 0.9998*s;
  c = newc;
}
```

Renormalizing

That works, but there's a problem: no matter how precisely we define our constants, after repeated iteration of this procedure, the magnitude of our (c, s)(c,s) vector will exponentially grow or shrink over time. If we only need to make one pass around the loop, maybe we can get away with that, but if we have to make several (for the rotating animation, we do), we need to fix that.


sine and cosine magnitude creep

an exaggerated illustration of what happens when repeatedly doing low-precision rotations

The simplest way to do that would be to multiply `c`c and `s`s by $1/\sqrt{c^2+s^2}$ 1/c2+s2, but then we're back to using the math library again. Instead, we can take advantage of the fact that our magnitude starts out very close to 1, and we're going to be iterating this procedure: we can do a Newton step after each rotation, and that will be enough to keep the magnitude "close enough" to 1 over time.

Our goal is to find the reciprocal square root (sound familiar?) of $a = c^2 + s^2$ a=c2+s2, our (c, s)(c,s) vector magnitude. Say we define a function $f(x) = \frac{1}{x^2} - a$ f(x)=1x2−a. The function is 0 when $x = \frac{1}{\sqrt{a}}$ x=1a. We can start with an initial guess of 1 for *x*, perform a Newton iteration to obtain *x'*, which will be "closer to" $\frac{1}{\sqrt{a}}$ 1a, the correct value to scale *c* and *s* by so that their magnitude $c^2 + s^2$ c2+s2 is "close to" 1 again.

A Newton step is defined as $x' = x - \frac{f(x)}{f'(x)}$ x'=x−f(x)f'(x). I used SymPy to do the derivative and simplification and came up with $x' = \frac{x(3 - ax^2)}{2}$ x'=x(3−ax2)2. Since we're only doing one step, we can plug in our initial guess of 1 for `x`x and back-substitute $c^2 + s^2$ c2+s2 for `a`a to finally get our adjustment: $x' = (3 - c^2 - s^2)/2$ x'=(3−c2−s2)/2.

Further simplifying the rotation

But now that we don't have to worry so much about the magnitude of our result (within limits), we can take another shortcut (I got this idea studying old CORDIC algorithms). If we divide out the cosines from our original rotation matrix, we get

$$\begin{bmatrix} c' \\ s' \end{bmatrix} = \frac{1}{\cos\theta} \begin{bmatrix} 1 & -\tan\theta \\ \tan\theta & 1 \end{bmatrix} \begin{bmatrix} c \\ s \end{bmatrix}$$

[c's']=1cosθ[1−tanθtanθ1][cs]

using the trig identity $\tan\theta = \frac{\sin\theta}{\cos\theta}$ tanθ=sinθcosθ. Since we're only dealing with small angles, the leading $\frac{1}{\cos\theta}$ 1cosθ term is close enough to 1 that we can ignore it and have our Newton step take care of it.

And now we can finally understand how the rotation is done in the code. Towards the top of the donut code is this #define, which I've reindented:

```
#define R(t,x,y) \
  f = x; \
  x -= t*y; \
  y += t*f; \
  f = (3-x*x-y*y)/2; \
  x *= f; \
  y *= f;
```

This does an in-place rotation of a unit vector `x, y` where `t` is $\tan\theta$ tanθ. `f` is a temporary variable; the first three lines do the "matrix multiplication" on `x, y`. `f` is then re-used to get the magnitude adjustment, and then finally `x` and `y` are multiplied by `f` which moves them back onto the unit circle.

With that operation in hand, I just replaced all the angles with their sines and cosines and ran the rotation operator `R()` instead of calling `sin`/`cos`. The code is otherwise identical.

Getting rid of `float`, too

We can use exactly the same ideas with integer fixed-point arithmetic, and not use any `float` math whatsoever. I've redone all the math with 10-bit precision and produced the following C code which runs well on embedded devices which can do 32-bit multiplications and have ~4k of available RAM:

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define R(mul,shift,x,y) \
  _=x; \
  x -= mul*y>>shift; \
  y += mul*_>>shift; \
  _ = 3145728-x*x-y*y>>11; \
  x = x*_>>10; \
  y = y*_>>10;

int8_t b[1760], z[1760];

void main() {
  int sA=1024,cA=0,sB=1024,cB=0,_;
  for (;;) {
    memset(b, 32, 1760);  // text buffer
    memset(z, 127, 1760);   // z buffer
    int sj=0, cj=1024;
    for (int j = 0; j < 90; j++) {
      int si = 0, ci = 1024;  // sine and cosine of angle i
      for (int i = 0; i < 324; i++) {
        int R1 = 1, R2 = 2048, K2 = 5120*1024;

        int x0 = R1*cj + R2,
            x1 = ci*x0 >> 10,
            x2 = cA*sj >> 10,
            x3 = si*x0 >> 10,
            x4 = R1*x2 - (sA*x3 >> 10),
            x5 = sA*sj >> 10,
            x6 = K2 + R1*1024*x5 + cA*x3,
            x7 = cj*si >> 10,
            x = 40 + 30*(cB*x1 - sB*x4)/x6,
            y = 12 + 15*(cB*x4 + sB*x1)/x6,
            N = (-cA*x7 - cB*((-sA*x7>>10) + x2) - ci*(cj*sB >> 10) >> 10) - x5 >> 7;

        int o = x + 80 * y;
        int8_t zz = (x6-K2)>>15;
        if (22 > y && y > 0 && x > 0 && 80 > x && zz < z[o]) {
          z[o] = zz;
          b[o] = ".,-~:;=!*#$@"[N > 0 ? N : 0];
        }
        R(5, 8, ci, si)  // rotate i
      }
      R(9, 7, cj, sj)  // rotate j
    }
    for (int k = 0; 1761 > k; k++)
      putchar(k % 80 ? b[k] : 10);
    R(5, 7, cA, sA);
    R(5, 8, cB, sB);
    usleep(15000);
    printf("\x1b[23A");
  }
}
```

The output is pretty much the same.