a1k0n.net

About

Donut math: how donut.c works

Jul 20, 2011

There has been a sudden resurgence of interest in my "donut" code from 2006, and I've had a couple requests to explain this one. It's been five years now, so it's not exactly fresh in my memory, so I will reconstruct it from scratch, in great detail, and hopefully get approximately the same result.

This is the code:

```
            k;double sin()
          ,cos();main(){float A=
        0,B=0,i,j,z[1760];char b[
      1760];printf("\x1b[2J");for(;;
   ){memset(b,32,1760);memset(z,0,7040)
   ;for(j=0;6.28>j;j+=0.07)for(i=0;6.28
   >i;i+=0.02){float c=sin(i),d=cos(j),e=
   sin(A),f=sin(j),g=cos(A),h=d+2,D=1/(c*
   h*e+f*g+5),l=cos        (i),m=cos(B),n=s\
   in(B),t=c*h*g-f*        e;int x=40+30*D*
   (l*h*m-t*n),y=          12+15*D*(l*h*n
   +t*m),o=x+80*y,         N=8*((f*e-c*d*g
   )*m-c*d*e-f*g-l        *d*n);if(22>y&&
   y>0&&x>0&&80>x&&D>z[o]){z[o]=D;;;b[o]=
   ".,-~:;=!*#$@"[N>0?N:0];}}/*#****!!-*/
   printf("\x1b[H");for(k=0;1761>k;k++)
    putchar(k%80?b[k]:10);A+=0.04;B+=
      0.02;}}/*****####*******!!=;:~
        ~::==!!!**********!!!==::-
          .,~~;;;=========;;;:~-.
              ..,--------,*/
```

...and the output, animated in Javascript: toggle animation

```
                    ###$$$$@@@$
                 **!!!**##$$@@@@@$$
               **=;;:;=!*#$$$@@@@@$$#
              *#*!;::~,:=*##$$$$@@$$$$##
             ###*!=;~-..!*##$$$$$$$$$$#**
            #$$##!=;~,.;!##$$$$$$$$$$$##*!
           !$@@$$#*=:~.!*##$$$$$$$$$$####*!
           #$@@@$#*!=;!*####$$$$$$$$#####*!!=
          #$$$$$##*!**#####$$$$$$########***!=;
          ##$##*!!!***#################***!!!==:
         *###****##################****!*!!=;:
        =********##############*#****!*!!==;:~
        =!******************###**!!!===;;:~
        ;=!!!************!*!*!!!!!!==;;:~,
        ;;==!!!!!!!!!!!!!!!!!!=!===;;::~-
        ,:;======!=!=!!=!=====;;;::~~,
         ,~:;;;;;=;==;=;;;;;;:::~~-.
          -~~:::;:::::::::~~~--,.
```

At its core, it's a framebuffer and a Z-buffer into which I render pixels. Since it's just rendering relatively low-resolution ASCII art, I massively cheat. All it does is plot pixels along the surface of the torus at fixed-angle increments, and does it densely enough that the final result looks solid. The "pixels" it plots are ASCII characters corresponding to the illumination value of the surface at each point: `.,-~:;=!*#$@` from dimmest to brightest. No raytracing required.

So how do we do that? Well, let's start with the basic math behind 3D perspective rendering. The following diagram is a side view of a person sitting in front of a screen, viewing a 3D object behind it.
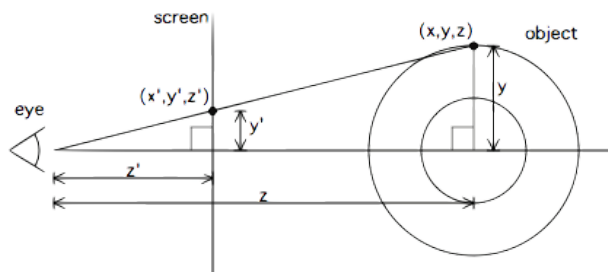
To render a 3D object onto a 2D screen, we project each point (x,y,z) in 3D-space onto a plane located z' units away from the viewer, so that the corresponding 2D position is (x',y'). Since we're looking from the side, we can only see the y and z axes, but the math works the same for the x axis (just pretend this is a top view instead). This projection is really easy to obtain: notice that the origin, the y-axis, and point (x,y,z) form a right triangle, and a similar right triangle is formed with (x',y',z'). Thus the relative proportions are maintained:

$$\frac{y'}{z'} = \frac{y}{z}$$
$$y' = \frac{yz'}{z}.$$

So to project a 3D coordinate to 2D, we scale a coordinate by the screen distance z'. Since z' is a fixed constant, and not functionally a coordinate, let's rename it to K_1, so our projection equation becomes $(x',y') = \left(\frac{K_1 x}{z}, \frac{K_1 y}{z}\right)$. We can choose K_1 arbitrarily based on the field of view we want to show in our 2D window. For example, if we have a 100x100 window of pixels, then the view is centered at (50,50); and if we want to see an object which is 10 units wide in our 3D space, set back 5 units from the viewer, then K_1 should be chosen so that the projection of the point x=10, z=5 is still on the screen with x' < 50: $10K_1/5 < 50$, or $K_1 < 25$.

When we're plotting a bunch of points, we might end up plotting different points at the same (x',y') location but at different depths, so we maintain a z-buffer which stores the z coordinate of everything we draw. If we need to plot a location, we first check to see whether we're plotting in front of what's there already. It also helps to compute $z^{-1} = \frac{1}{z}$ and use that when depth buffering because:

- z^{-1} = 0 corresponds to infinite depth, so we can pre-initialize our z-buffer to 0 and have the background be infinitely far away
- we can re-use z^{-1} when computing x' and y': Dividing once and multiplying by z^{-1} twice is cheaper than dividing by z twice.

Now, how do we draw a donut, AKA torus? Well, a torus is a solid of revolution, so one way to do it is to draw a 2D circle around some point in 3D space, and then rotate it around the central axis of the torus. Here is a cross-section through the center of a torus:
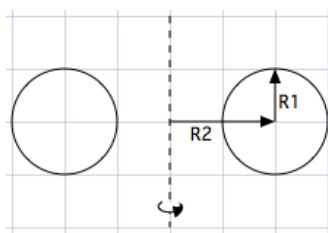
So we have a circle of radius R_1 centered at point $(R_2,0,0)$, drawn on the xy-plane. We can draw this by sweeping an angle — let's call it θ — from 0 to 2π:

$$(x,y,z) = (R_2,0,0) + (R_1\cos\theta, R_1\sin\theta, 0)$$

Now we take that circle and rotate it around the y-axis by another angle — let's call it φ. To rotate an arbitrary 3D point around one of the cardinal axes, the standard technique is to multiply by a rotation matrix. So if we take the previous points and rotate about the y-axis we get:

$$\left(R_2 + R_1\cos\theta, R_1\sin\theta, 0\right) \cdot \begin{pmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{pmatrix}$$
$$= \left((R_2 + R_1\cos\theta)\cos\phi, R_1\sin\theta, -(R_2 + R_1\cos\theta)\sin\phi\right)$$

But wait: we also want the whole donut to spin around on at least two more axes for the animation. They were called *A* and *B* in the original code: it was a rotation about the *x*-axis by *A* and a rotation about the *z*-axis by *B*. This is a bit hairier, so I'm not even going write the result yet, but it's a bunch of matrix multiplies.

$$\left(R_2 + R_1\cos\theta, R_1\sin\theta, 0\right) \cdot \begin{pmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos A & \sin A \\ 0 & -\sin A & \cos A \end{pmatrix} \cdot \begin{pmatrix} \cos B & \sin B & 0 \\ -\sin B & \cos B & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Churning through the above gets us an (*x,y,z*) point on the surface of our torus, rotated around two axes, centered at the origin. To actually get screen coordinates, we need to:

- Move the torus somewhere in front of the viewer (the viewer is at the origin) — so we just add some constant to *z* to move it backward.
- Project from 3D onto our 2D screen.

So we have another constant to pick, call it K_2, for the distance of the donut from the viewer, and our projection now looks like:

$$\left(x', y'\right) = \left(\frac{K_1 x}{K_2 + z}, \frac{K_1 y}{K_2 + z} \right)$$

K_1 and K_2 can be tweaked together to change the field of view and flatten or exaggerate the depth of the object.

Now, we could implement a 3x3 matrix multiplication routine in our code and implement the above in a straightforward way. But if our goal is to shrink the code as much as possible, then every 0 in the matrices above is an opportunity for simplification. So let's multiply it out. Churning through a bunch of algebra (thanks Mathematica!), the full result is:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} (R_2 + R_1\cos\theta)(\cos B\cos\phi + \sin A\sin B\sin\phi) - R_1\cos A\sin B\sin\theta \\ (R_2 + R_1\cos\theta)(\cos\phi\sin B - \cos B\sin A\sin\phi) + R_1\cos A\cos B\sin\theta \\ \cos A(R_2 + R_1\cos\theta)\sin\phi + R_1\sin A\sin\theta \end{pmatrix}$$

Well, that looks pretty hideous, but we we can precompute some common subexpressions (e.g. all the sines and cosines, and $R_2 + R_1\cos\theta$ and reuse them in the code. In fact I came up with a completely different factoring in the original code but that's left as an exercise for the reader. (The original code also swaps the sines and cosines of A, effectively rotating by 90 degrees, so I guess my initial derivation was a bit different but that's OK.)

Now we know where to put the pixel, but we still haven't even considered which shade to plot. To calculate illumination, we need to know the surface normal — the direction perpendicular to the surface at each point. If we have that, then we can take the dot product of the surface normal with the light direction, which we can choose arbitrarily. That gives us the cosine of the angle between the light direction and the surface direction: If the dot product is >0, the surface is facing the light and if it's <0, it faces away from the light. The higher the value, the more light falls on the surface.

The derivation of the surface normal direction turns out to be pretty much the same as our derivation of the point in space. We start with a point on a circle, rotate it around the torus's central axis, and then make two more rotations. The surface normal of the point on the circle is fairly obvious: it's the same as the point on a unit (radius=1) circle centered at the origin.

So our surface normal (N_x, N_y, N_z) is derived the same as above, except the point we start with is just (cos θ, sin θ, 0). Then we apply the same rotations:

$$\left(N_x, \; N_y, \; N_z\right) = \left(\cos\theta, \sin\theta, 0\right) \cdot \begin{pmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos A & \sin A \\ 0 & -\sin A & \cos A \end{pmatrix} \cdot \begin{pmatrix} \cos B \\ -\sin \\ 0 \end{pmatrix}$$

So which lighting direction should we choose? How about we light up surfaces facing behind and above the viewer: $(0, 1, -1)$. Technically this should be a normalized unit vector, and this vector has a magnitude of √2. That's okay – we will compensate later. Therefore we compute the above (*x,y,z*), throw away the *x* and get our luminance *L* = *y−z*.

$$L = (N_x, \ N_y, \ N_z) \cdot (0, \ 1, \ -1)$$
$$= \cos\phi\cos\theta\sin B - \cos A\cos\theta\sin\phi - \sin A\sin\theta + \cos B(\cos A\sin\theta - \cos$$

Again, not too pretty, but not terrible once we've precomputed all the sines and cosines.

So now all that's left to do is to pick some values for R_1, R_2, K_1, and K_2. In the original donut code I chose R_1=1 and R_2=2, so it has the same geometry as my cross-section diagram above. K_1 controls the scale, which depends on our pixel resolution and is in fact different for x and y in the ASCII animation. K_2, the distance from the viewer to the donut, was chosen to be 5.

I've taken the above equations and written a quick and dirty canvas implementation here, just plotting the pixels and the lighting values from the equations above. The result is not exactly the same as the original as some of my rotations are in opposite directions or off by 90 degrees, but it is qualitatively doing the same thing.

Here it is: toggle animation
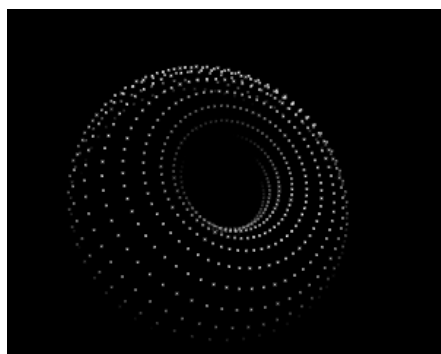
It's slightly mind-bending because you can see right through the torus, but the math does work! Convert that to an ASCII rendering with z-buffering, and you've got yourself a clever little program.

Now, we have all the pieces, but how do we write the code? Roughly like this (some pseudocode liberties have been taken with 2D arrays):

```c
const float theta_spacing = 0.07;
const float phi_spacing   = 0.02;

const float R1 = 1;
const float R2 = 2;
const float K2 = 5;
// Calculate K1 based on screen size: the maximum x-distance occurs
// roughly at the edge of the torus, which is at x=R1+R2, z=0.  we
// want that to be displaced 3/8ths of the width of the screen, which
// is 3/4th of the way from the center to the side of the screen.
// screen_width*3/8 = K1*(R1+R2)/(K2+0)
// screen_width*K2*3/(8*(R1+R2)) = K1
const float K1 = screen_width*K2*3/(8*(R1+R2));

render_frame(float A, float B) {
  // precompute sines and cosines of A and B
  float cosA = cos(A), sinA = sin(A);
  float cosB = cos(B), sinB = sin(B);

  char output[0..screen_width, 0..screen_height] = ' ';
  float zbuffer[0..screen_width, 0..screen_height] = 0;

  // theta goes around the cross-sectional circle of a torus
  for (float theta=0; theta < 2*pi; theta += theta_spacing) {
    // precompute sines and cosines of theta
    float costheta = cos(theta), sintheta = sin(theta);

    // phi goes around the center of revolution of a torus
    for(float phi=0; phi < 2*pi; phi += phi_spacing) {
      // precompute sines and cosines of phi
      float cosphi = cos(phi), sinphi = sin(phi);

      // the x,y coordinate of the circle, before revolving (factored
      // out of the above equations)
      float circlex = R2 + R1*costheta;
      float circley = R1*sintheta;

      // final 3D (x,y,z) coordinate after rotations, directly from
      // our math above
      float x = circlex*(cosB*cosphi + sinA*sinB*sinphi)
        - circley*cosA*sinB;
      float y = circlex*(sinB*cosphi - sinA*cosB*sinphi)
        + circley*cosA*cosB;
      float z = K2 + cosA*circlex*sinphi + circley*sinA;
      float ooz = 1/z;  // "one over z"

      // x and y projection.  note that y is negated here, because y
      // goes up in 3D space but down on 2D displays.
```

```c
    int xp = (int) (screen_width/2 + K1*ooz*x);
    int yp = (int) (screen_height/2 - K1*ooz*y);

    // calculate luminance.  ugly, but correct.
    float L = cosphi*costheta*sinB - cosA*costheta*sinphi -
      sinA*sintheta + cosB*(cosA*sintheta - costheta*sinA*sinphi);
    // L ranges from -sqrt(2) to +sqrt(2).  If it's < 0, the surface
    // is pointing away from us, so we won't bother trying to plot it.
    if (L > 0) {
      // test against the z-buffer.  larger 1/z means the pixel is
      // closer to the viewer than what's already plotted.
      if(ooz > zbuffer[xp,yp]) {
        zbuffer[xp, yp] = ooz;
        int luminance_index = L*8;
        // luminance_index is now in the range 0..11 (8*sqrt(2) = 11.3)
        // now we lookup the character corresponding to the
        // luminance and plot it in our output:
        output[xp, yp] = ".,-~:;=!*#$@"[luminance_index];
      }
    }
   }
  }
 }

 // now, dump output[] to the screen.
 // bring cursor to "home" location, in just about any currently-used
 // terminal emulation mode
 printf("\x1b[H");
 for (int j = 0; j < screen_height; j++) {
   for (int i = 0; i < screen_width; i++) {
     putchar(output[i,j]);
   }
   putchar('\n');
 }

}
```

The Javascript source for both the ASCII and canvas rendering is right here.

---

a1k0n.net