# Donut math: comment fonctionne donut.c

*Juil 20, 2011*

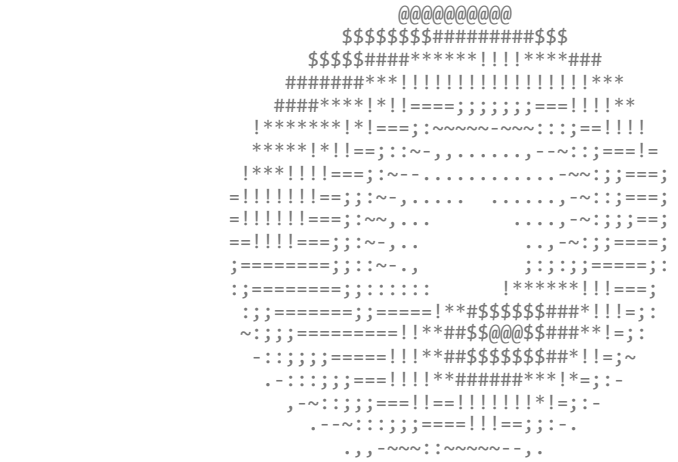
[**Mise à jour 1/13/2021**: J'ai écrit un suivi avec quelques optimisations. ]

Il y a eu un regain soudain d'intérêt pour mon code « beignet » de 2006, et j'ai J'ai eu quelques demandes pour expliquer celui-ci. Cela fait cinq ans maintenant, donc c'est pas vraiment frais dans ma mémoire, alors je vais le reconstruire à partir de zéro, en grand détailler et, espérons-le, obtenir à peu près le même résultat.

Voici le code :

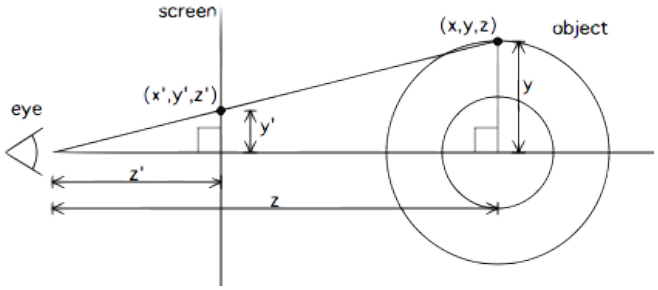
```
            k;double sin()
         ,cos();main(){float A=
       0,B=0,i,j,z[1760];char b[
     1760];printf("\x1b[2J");for(;;
  ){memset(b,32,1760);memset(z,0,7040)
  ;for(j=0;6.28>j;j+=0.07)for(i=0;6.28
 >i;i+=0.02){float c=sin(i),d=cos(j),e=
 sin(A),f=sin(j),g=cos(A),h=d+2,D=1/(c*
 h*e+f*g+5),l=cos      (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;}}/*****####*******!!=;:~
       ~::==!!!**********!!!==::-
         .,~~;;;========;;;:~-.
             ..,--------,*/
```

... et la sortie, animée en Javascript : basculer l'animation



À la base, c'est un framebuffer et un Z-buffer dans lequel je rends des pixels. Comme il ne s'agit que de rendre des illustrations ASCII relativement basse résolution, j'ai massivement Triche. Tout ce qu'il fait est de tracer des pixels le long de la surface du tore à incréments à angle fixe, et le fait assez densément pour que le résultat final semble solide. Les « pixels » qu'il trace sont des caractères ASCII correspondant au Valeur d'éclairement de la surface en chaque point: du plus faible au plus faible le plus brillant. Aucun raytracing requis.`.,-~:;=!*#$@`

Alors, comment pouvons-nous faire cela? Eh bien, commençons par les mathématiques de base derrière la 3D Rendu de perspective. Le diagramme suivant est une vue latérale d'une personne assis devant un écran, regardant un objet 3D derrière lui.




Pour rendre un objet 3D sur un écran 2D, nous projetons chaque point (*x*,*y*,*z*) dans 3D-space sur un plan situé *z'* unités loin du spectateur, de sorte que le la position 2D correspondante est (*x'*,*y'*). Puisque nous regardons de côté, Nous ne pouvons voir que les axes *y* et *z*, mais les mathématiques fonctionnent de la même manière pour l'axe *des x* (faites semblant qu'il s'agit plutôt d'une vue de dessus). Cette projection est vraiment facile Pour obtenir : notez que l'origine, l'axe *des* y et le point (*x*, *y*, *z*) forment un triangle rectangle, et un triangle rectangle similaire est formé avec (*x'*,*y'*,*z'*). Ainsi, les proportions relatives sont maintenues:

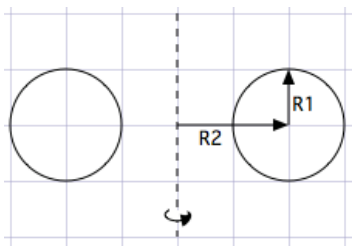
$$\frac{y'}{z'} = \frac{y}{z}$$
$$y' = \frac{yz'}{z}.$$

Donc, pour projeter une coordonnée 3D en 2D, nous mettons à l'échelle une coordonnée par l'écran distance *z'*. Puisque *z'* est une constante fixe, et non fonctionnellement une coordonnées, renommons-la *en* $K_1$, donc notre équation de projection Devient $(x', y') = (\frac{K_1 x}{z}, \frac{K_1 y}{z})$. Nous pouvons choisir $K_1$ arbitrairement en fonction du champ de vision que nous voulons montrer dans notre Fenêtre 2D. Par exemple, si nous avons une fenêtre de pixels de 100x100, alors la vue est centré à (50,50); et si nous voulons voir un objet qui fait 10 unités de large dans notre espace 3D, reculez de 5 unités par rapport au spectateur, puis $K_1$ devrait être choisi de telle sorte que la projection du point *x*=10, *z*=5 soit toujours sur le écran avec *x'* < 50: 10 K 1/5 < 50, ou$K_1$ < 25.

Lorsque nous traçons un tas de points, nous pourrions finir par tracer différents pointe au même endroit (*x'*,*y'*) mais à des profondeurs différentes, donc nous maintenons un z-buffer qui stocke La coordonnée *z* de tout ce que nous dessinons. Si nous avons besoin de tracer un emplacement, nous Vérifiez d'abord si nous complotons devant ce qui est déjà là. Il Aide également à calculer $z^{-1} = \frac{1}{z}$ et utilisez-le lorsque la profondeur Mise en mémoire tampon pour les raisons suivantes :

- $z^{-1}$ = 0 correspond à une profondeur infinie, nous pouvons donc pré-initialiser Notre Z-Buffer à 0 et avoir l'arrière-plan être infiniment loin
- Nous pouvons réutiliser $z^{-1}$ Lors du calcul *de x'* et *y'*: En divisant une fois et en multipliant par $z^{-1}$ deux fois moins cher que en divisant deux fois par *Z*.

Maintenant, comment dessiner un beignet, alias tore? Eh bien, un tore est un solide de révolution, donc une façon de le faire est de dessiner un cercle 2D autour d'un point dans Espace 3D, puis faites-le pivoter autour de l'axe central du tore. Voici un Coupe transversale passant par le centre d'un tore:




Nous avons donc un cercle de rayon *R1* centré au point (*R* 2,0,0), dessiné sur le plan *xy*. Nous pouvons dessiner cela en balayant Un angle — appelons-le $\theta$ — de 0 à $2\pi$:

$$(x, y, z) = (R_2, 0,0\ ) + (R_1 \cos\theta, R_1\ \text{péché}\ \theta,0\ )$$

Maintenant, nous prenons ce cercle et le faisons pivoter autour de l'axe *y* par un autre angle — appelons-le φ. Pour faire pivoter un point 3D arbitraire autour de l'un des axes cardinaux, la technique standard consiste à multiplier par une matrice de rotation. Donc, si Nous prenons les points précédents et tournons autour de l'axe *des y* que nous obtenons:

$$(\ R_2 + R_1 \cos\theta, \quad R_1\ \text{péché}\ \theta),\cdot \begin{pmatrix} \cos\varphi & 0 & \text{péché}\ \varphi \\ 0 & 1 & 0 \\ -\text{Péché}\ \varphi & 0 & \cos\varphi \end{pmatrix}$$

$$= (\ (R_2 + R_1 \cos\theta)\cos\varphi, \quad R_1\ \text{péché}\ \theta, (R_2 + R_1 \cos\theta)\sin\varphi\ )$$

Mais attendez: nous voulons aussi que tout le beignet tourne sur au moins deux autres axes pour l'animation. Ils étaient appelés *A* et *B* dans le code original: c'était un rotation autour de l'axe *des x* par *A* et rotation autour de l'axe *z* par *B*. C'est un peu plus poilu, donc je ne vais même pas encore écrire le résultat, mais c'est un Le tas de matrice se multiplie.

$$(\ R_2 + R_1 \cos\theta, \quad R_1\ \text{péché}\ \theta),\cdot \begin{pmatrix} \cos\varphi & 0 & \text{péché}\ \varphi \\ 0 & 1 & 0 \\ -\text{Péché}\ \varphi & 0 & \cos\varphi \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos A & \text{péché}\ A \\ 0 & -\sin A & \cos A \end{pmatrix} \begin{pmatrix} \cos B & \text{péch} \\ -\text{péché}\ B\cos B & \\ 0 & 0 \end{pmatrix}$$

En parcourant ce qui précède, nous obtenons un point (*x*, *y*, *z*) à la surface de notre Tore, tourné autour de deux axes, centré à l'origine. Pour réellement obtenir l'écran coordonnées, nous devons :

- Déplacez le tore quelque part devant le spectateur (le spectateur est au origin) — donc nous ajoutons simplement une constante à *z* pour la faire reculer.
- Projetez de la 3D sur notre écran 2D.

Nous avons donc une autre constante à choisir, appelez-la $K_2$, pour la distance du beignet du spectateur, et notre projection ressemble maintenant à:

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

$K_1$ et $K_2$ peuvent être modifiés ensemble pour changer le champ de vue et aplatir ou exagérer la profondeur de l'objet.

Maintenant, nous pourrions implémenter une routine de multiplication matricielle 3x3 dans notre code et Mettez en œuvre ce qui précède de manière simple. Mais si notre objectif est de réduire le coder autant que possible, alors chaque 0 dans les matrices ci-dessus est une opportunité pour simplifier. Alors multiplions-le. Barattage à travers un tas de algèbre (merci Mathematica!), le résultat complet est:

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

Eh bien, cela a l'air assez hideux, mais nous pouvons précalculer certains sous-expressions (par exemple, tous les sinus et cosinus, et $R_2 + R_1 \cos\theta$) et réutilisez-les dans le code. En fait, j'ai trouvé un en tenant compte du code original, mais cela reste un exercice pour le lecteur. (Le code original échange également les sinus et les cosinus de A, tournant efficacement de 90 degrés, donc je suppose que ma dérivation initiale était un peu différente, mais c'est D'accord.)

Maintenant, nous savons où placer le pixel, mais nous n'avons même pas encore envisagé lequel ombre à l'intrigue. Pour calculer l'éclairement lumineux, nous devons connaître la normale de surface... la direction perpendiculaire à la surface en chaque point. Si c'est le cas, Ensuite, nous pouvons prendre le point produit de la surface normale avec la direction de la lumière, que nous pouvons choisir arbitrairement. Cela nous donne le cosinus de l'angle entre la direction de la lumière et la direction de la surface : si le produit de points est >0, la surface est orientée vers la lumière et si elle est <0, elle fait face à la lumière. Plus le valeur, plus la lumière tombe à la surface.

La dérivation de la direction normale de surface s'avère être à peu près la identique à notre dérivation du point dans l'espace. Nous commençons par un point sur un cercle, faites-le pivoter autour de l'axe central du tore, puis faites-en deux autres Rotations. La normale de surface du point sur le cercle est assez évidente: C'est le même que le point sur un cercle unité (rayon=1) centré à l'origine.

Donc, notre normale de surface ($N_x$, $N_y$, $N_z$) est dérivé de la même manière que ci-dessus, sauf que le point par lequel nous commençons est juste (cos θ, sin $\theta$, 0). Ensuite, nous appliquons les mêmes rotations:

$$(N_x, \quad N_y, \quad N_z) = (\cos\theta, \quad \text{péché } \theta, \quad 0) \cdot \begin{pmatrix} \cos\varphi & 0 & \text{péché}\varphi \\ 0 & 1 & 0 \\ -\text{Péché } \varphi & 0 & \cos\varphi \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos A & \text{péché } A \\ 0 & -\sin A & \cos A \end{pmatrix} \begin{pmatrix} \cos B \\ -\text{péch} \\ 0 \end{pmatrix}$$

Alors, quelle direction d'éclairage devrions-nous choisir? Que diriez-vous d'éclairer les surfaces Face à l'arrière et au-dessus du spectateur : $(0,1, -1)$. Techniquement Il devrait s'agir d'un vecteur unitaire normalisé, et ce vecteur a une magnitude de √2. Ce n'est pas grave, nous compenserons plus tard. Par conséquent, nous calculons le au-dessus (x,y,z), jetez le $x$ et obtenez notre luminance $L = y-z$.

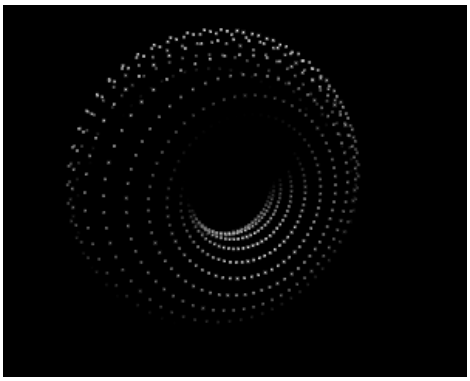
$$\begin{aligned} L &= (N_x, \quad N_y, \quad N_z) \cdot (0, \quad 1, \quad -1) \\ &= \cos\varphi \cos\theta \sin B - \cos A \cos\theta \sin\varphi - \sin A \sin\theta + \cos B(\cos A \sin\theta - \cos\theta \sin A \sin\varphi \end{aligned}$$

Encore une fois, pas trop joli, mais pas terrible une fois que nous avons précalculé tous les sinus et cosinus.

Il ne reste donc plus qu'à choisir des valeurs pour R 1, $R$ 2, $K_1$ et $K_2$. Dans le beignet original code J'ai choisi R 1 = $_1$ et $R$ 2 = $_2$, donc il a le même géométrie comme mon diagramme en coupe transversale ci-dessus. $K_1$ contrôle le échelle, qui dépend de notre résolution en pixels et est en fait différente pour $x$ et $y$ dans l'animation ASCII. $K_2$, la distance du spectateur au beignet, a été choisi pour être 5.

J'ai pris les équations ci-dessus et écrit une toile rapide et sale mise en œuvre ici, en traçant simplement les pixels et les valeurs d'éclairage du équations ci-dessus. Le résultat n'est pas exactement le même que l'original que certains Mes rotations sont dans des directions opposées ou décalées de 90 degrés, mais c'est qualitativement faire la même chose.

Le voilà: basculer l'animation



C'est un peu hallucinant parce que vous pouvez voir à travers le tore, mais le Les maths fonctionnent! Convertissez-le en un rendu ASCII avec mise en mémoire tampon $z$, et Vous avez vous-même un petit programme intelligent.

Maintenant, nous avons toutes les pièces, mais comment écrivons-nous le code? À peu près comme ça (certaines libertés de pseudo-code ont été prises avec les tableaux 2D):

```
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.
      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');
  }

}
```

La source Javascript pour le rendu ASCII et canvas est ici.