Vv214 Final Project

Donut.c

Pingbang Hu, Xiaoyu Chen, Jinyi Wu

University of Michigan-Shanghai Jiao Tong University Joint Institute

April 28, 2021

Overview

Motivation

Back in 2006, there's an interesting c project called Donut.c, which will print a rotating donut in the terminal.

Looking into the source code and a blog updated recently by the author, we find out that this program is highly depend on what we have learned in Vv214.

After some discussion, we decide to dive into the source code and analyze the whole mechanism behind it.

Introduction

To start with, we first need to design a way to show the donut. Seems the terminal has a dark background with white character.

We let the **size** of the character to mimic the brightness of a single pixel. We show the character array as below, from the darkest to the brightest:

$$\{. \quad , \quad - \quad ; \quad = \quad ! \quad * \quad \# \quad \$ \quad @\}$$

With these characters, we can generate some black & white *ascii art* in the terminal!

Draw a donut

Now we have the way to show some diagram in the terminal, then we start to consider how to actually generate a rotating donut in the terminal.

In a mathematical way to produce a donut, we start with the concept of *parametric equation*. Since a donut basically is a circle rotating respected to the axis parallel to the fixed diameter, with some simply geometry, we have:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} R_2 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} R_1 \cos\theta \\ R_1 \sin\theta \\ 0 \end{pmatrix}, \quad \theta \in [0, 2\pi)$$

which produce two circles in the $x - y$ plane.

Draw a donut

Since we now have two circles, we now only need to rotate these circles for π degree by applying a *rotation matrix w.r.t. y-axis*.

With the parametric equation we have derived, we have:

$$\underbrace{\begin{pmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{pmatrix}}_{\text{Y : rotation matrix for y-axis}} \cdot \underbrace{\begin{pmatrix} R_2 + R_1\cos\theta \\ R_1\sin\theta \\ 0 \end{pmatrix}}_{\text{two circles' parametric eqs.}}, \quad \begin{cases} \theta \in [0, 2\pi) \\ \phi \in [0, \pi) \end{cases}$$

This produce a full donut in the 3 dimensional space. And for simplicity, we will now omit the angles' ranges from now on.

Draw a donut

Rotate a donut

Now let say we want to rotate this donut respect to x and z axis, we just need to apply the rotation matrix as below:

$$\overbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\psi_1 & \sin\psi_1 \\ 0 & -\sin\psi_1 & \cos\psi_1 \end{pmatrix}}^{\text{X : rotation matrix for x-axis}} \cdot \overbrace{\begin{pmatrix} \cos\psi_2 & \sin\psi_2 & 0 \\ -\sin\psi_2 & \cos\psi_2 & 0 \\ 0 & 0 & 1 \end{pmatrix}}^{\text{Z : rotation matrix for z-axis}} \cdot \underbrace{\begin{pmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{pmatrix}}_{\text{Y}} \cdot \begin{pmatrix} R_2 + R_1\cos\theta \\ R_1\sin\theta \\ 0 \end{pmatrix}$$

Then now, we have a rotating donut in 3 dimensional space, which rotating speed is now fully controlled by the *rate of change* of angles ψ_1 and ψ_2!

Bright and Dark

Now, let us think about how we actually get the brightness for an image. What makes the differences of bright and dark? The **light direction** and the **normal vector** for the surface!

Let us choose a light direction vector, say

$$L := \begin{pmatrix} L_x \\ L_y \\ L_z \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}$$

Bright and Dark

And by simply geometry, the normal vector can be derived from the same way, we first consider a particular θ when creating the donut, we find out that for such a point, say

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} R_2 + R_1 \cos\theta \\ R_1 \sin\theta \\ 0 \end{pmatrix}$$

the normal vector is:

$$N := \begin{pmatrix} N_x \\ N_y \\ N_z \end{pmatrix} = \begin{pmatrix} \cos\theta \\ \sin\theta \\ 0 \end{pmatrix}$$

then we just need to apply the same rotation matrix X and Z, we can get every normal vector on the surface of a donut!

Bright and Dark

In order to show how much light is directly shot on the particular pixel on a surface, we use **inner product** between the *normal vector* and *light vector*, namely

$$B_{(x,y,z)} :=< N, L >= L^T N = (L_x, L_y, L_z) \cdot \begin{pmatrix} N_x \\ N_y \\ N_z \end{pmatrix}$$

where N_i and L_i is known on every point for $i = x, y, z$.

Projection into terminal

We have now derived the complete mathematical equation for the rotating donut! The only thing left is how to let the viewer to *see* the donut in the terminal.

There are two things we need to solve:

- Move the donut somewhere in front of the viewer(the viewer is at the origin).
- Project from 3 dimensional space onto our 2 dimensional terminal.

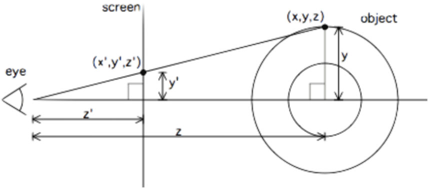
It's easy to obtain a simple relation for the projection, but there will have some problem we need to solve, namely *overlapping*.

From the figure, we set the screen at z', and we get

$$\frac{y'}{z'} = \frac{y}{z + K_1} \quad \Rightarrow \quad y' = \frac{yz'}{z + K_1}$$

where K_1 is the constant added to move the donut backward. Since z' is another fixed number, we let it be a constant, say K_2, then the projection equation becomes

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \frac{K_2 x}{K_1 + z} \\ \frac{K_2 y}{K_1 + z} \end{pmatrix}$$



z-buffer

As we mentioned before, when plotting a bunch of points, we might plot different points at the same 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 whether we're plotting in front of what's there already.

A trick to do this is by computing $z^{-1} = \frac{1}{z}$ and use that when depth buffering because:

- $z^{-1} = 0$ corresponds to infinite depth, so we can 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', since multiplication if much faster than division.

Discrete Dynamic System

Now, we are fully prepared to plot the result on the terminal. But one thing we need to be aware of is we can't really plot the 'animation', instead we plot frames one by one, which corresponding to the concept to *discrete dynamic system*. To be clearer, we have

$$R(t+T) = R(t) \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

where T is the time between two frames, $R(t)$ corresponds to all the linear transformation we apply to a point.

Discrete Dynamic System

If we want to calculate a frame after a long time, we can actually *diagonalize* $R(t)$ as

$$R(t) = SD(t)S^{-1}$$

where S is the change of basis matrix for $R(t)$, $D(t)$ is a diagonal matrix whose diagonal consists of *eigenvalue* of $R(t)$.

Then we can quickly get $R^t(t)$ for large n by

$$R^n(t) = SD^n(t)S^{-1}$$

But in this case we actually need to calculate every frame, so we do not need to change basis procedure.

Demonstration

Now let we have some interesting demonstration!

Demonstration

Demonstration

Demonstration

```
             k;double sin(),
         cos();int 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;}}/*****####*******!!=;:~
       ~::==!!!**********!!!==::-
         .,~~;;;========;;;:~-.
             ..,--------,*/
```

Reference

- https://www.a1k0n.net/2011/07/20/donut-math.html
- https://en.wikipedia.org/wiki/3D_computer_graphics
- https://www.javatpoint.com/computer-graphics-z-buffer-algorithm