wootwoot

DIY donut.c

รับจำนวนจริง r1, r2, t, p, xi, yi, zi, xd, yd, zd, sw, sh กับจำนวนเต็ม f เข้าแล้วแสดงผลเป็นโดนัทที่มีการหมุน

Input

r1 : [0.5, 5] - รัศมีของวงรอบนอกของโดนัท

r2 : [0.5, 5] - รัศมีของโดนัท

t : [0.05, 0.5] - theta (มุมที่แยกระหว่างจุดของวงกลมที่สร้างโดนัท)

p : [0.01, 0.1] - phi (มุมที่แยกระหว่างวงแต่ละวงของโดนัท)

xi, yi, zi : [-2π, 2π) - initial x, initial y, initial z (มุม euler angles เริ่มต้นของ donut)

xd, yd, zd : [-2π, 2π) - delta x, delta y, delta z (มุม euler angles ที่อธิบายการหมุนของโดนัทในแต่ละ frame)

sw : [20, 200] - screen width (ความกว้างของบริเวณแสดงผล)

sh : [20, 200] - screen height (ความสูงของบริเวณแสดงผล)

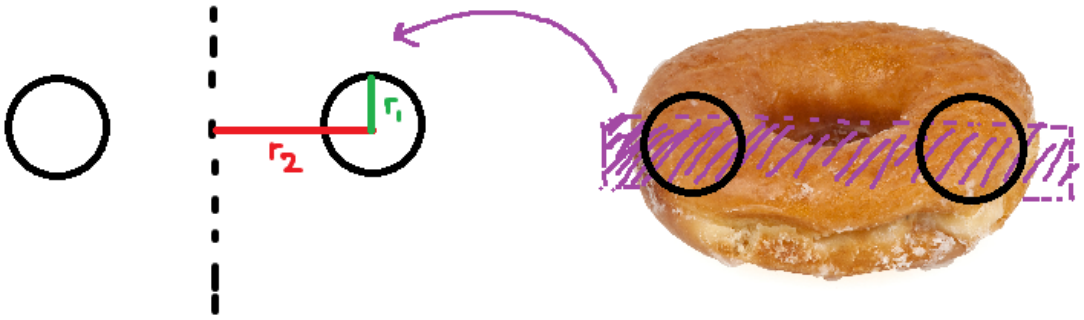
f : [1, 10] - frames (จำนวน frames ทั้งหมดที่ต้องแสดงผล)

ทั้งนี้

- โดนัทจะมีจุดศูนย์กลางและ pivot อยู่ที่ (0, 0, 0)
- กล้องมีพิกัดอยู่ที่ (0, 0, -1.5*r1+r2) หันอยู่แนวแกน +Z ด้วย horizontal fov ที่มุม π/3 และมี near clipping plane อยู่ที่ 0.1 และ far clipping plane อยู่ที่ 1000 เสมอ
- เป็น flat shading
- กำหนด √2 เป็นค่าคงที่ 1.414 และ π เป็นค่าคงที่ 3.14
- มี directional light source เดียวใน scene ที่มี direction vector เป็น (0, (√2)/2, -(√2)/2)
- การคำนวณทุกอย่าง ใช้ double และเป็นหน่วย radians หมด
- ชุดตัวอักษรแทนความสว่าง คือ ".,-~:;=!*#$@" มีทั้งหมด 12 ตัว รวมกับ " " อีก 1 ตัว สำหรับ pixel ที่มี luminosity น้อยกว่า 0
- ผลที่อยู่ใน framebuffer และ depthbuffer จะมีค่าอยู่ระหว่าง 0 และ 1

อธิบายสมการของ donut และหลักการ perspective projection

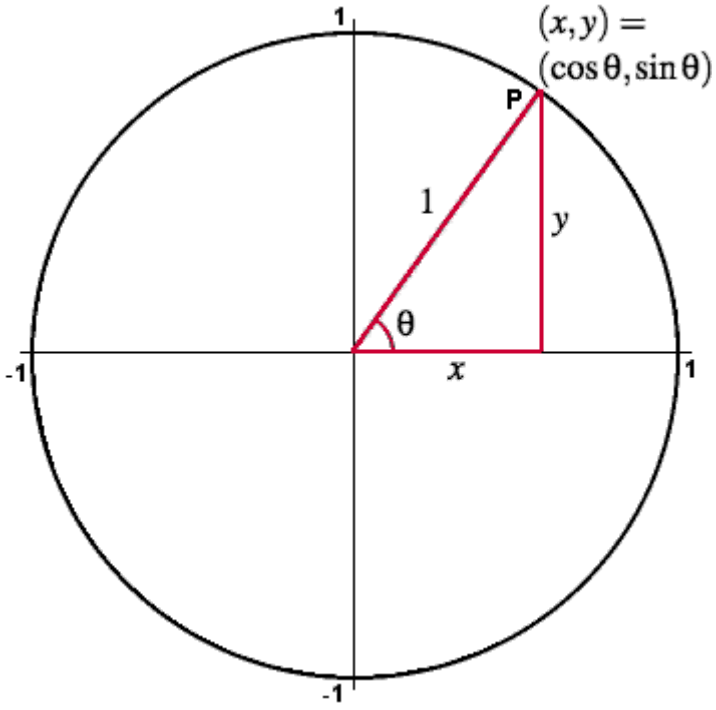
โดนัท เป็นขนมแป้งทอดหรืออบ ที่มีเนื้อคล้ายกับขนมเค้ก มีลักษณะกลมมีรูตรงกลางคล้ายกับห่วงยาง มีหลายรสชาติ ถ้าเป็นของไทยจะมีน้ำตาลอยู่ที่ผิวของขนม ซึ่งนอกจากจะอร่อยและให้พลังงานกับร่างกายเพื่อเข้าสู่กระบวนการหายใจระดับเซลล์แล้ว ยังเป็นรูปทรงที่สามารถอธิบายได้ด้วยสมการทางคณิตศาสตร์อีกด้วย ซึ่งหากสังเกตดีๆแล้ว จะเห็นได้ว่า ภาพหน้าตัด (cross-section) ของโดนัทจะประกอบไปด้วยวงกลม 2 รูป ซึ่งจริงๆแล้ว การจะวาดโดนัทแล้วใช้เพียงสมการของวงกลมก็วาดได้



อย่างที่รู้กัน สมการของวงกลมในรูปแบบที่นิยมรูปแบบหนึ่ง คือ $(x-h)^2+(y-b)^2=r^2$ แต่การที่เราจะต้องไล่ค่า x กับ y ทุกค่าเพื่อหาว่าค่าใดอยู่บนเส้นของวงกลมนั้น คงยากและเสียเวลามากเกิน เพราะฉะนั้นเราจะใช้สมการของวงกลมที่ว่า

$$y = \sin(\theta)$$

$$x = \cos(\theta)$$



และเพียงเราไล่ค่าของ θ จาก 0 ถึง 2π ทุกๆ increment ของ t เราก็จะสามารถวาดวงกลมออกมาได้ และหากเราต้องการนำค่า r1 และ r2 ของเรามาใส่ในสมการ ก็จะได้

$$(x, y) = (r1*\cos(\theta) + r2, r1*\sin(\theta))$$

เราได้วิธีการวาดวงกลมแล้ว หากเราต้องการจะวาดโดนัท เราก็เพียงนำจุดที่เกิดจากการไล่ θ มาคูณกับ y rotation matrix ซึ่งก็คือ matrix ที่มีไว้สำหรับการแปลง vector เพื่อให้หมุนตามแกน y ทำให้เกิดเป็นรูปทรงโดนัทขึ้น ในที่นี้เราจะหมุนตามมุม ϕ จาก 0 ถึง 2π ทุกๆ

wootwoot

increment ของ p นอกจากนั้นแล้วเรายังต้องคูณกับ 3-axis rotation matrix ต่อ เพื่อหมุนทั้งโดนัทให้เกิดเป็นภาพเคลื่อนไหว ซึ่ง rotation matrix ทั้งสองนั้นเราสามารถนำมา premultiply ก่อนได้เพื่อความสะดวกในการทำโจทย์ แล้วนำมาเขียนเป็น function ได้ดังนี้

```
vec3 getRotatedPoint(vec2 point, double phi, double a, double b, double c)
{
    vec3 rotated =
    {
        .x = point.x*(cos(c)*cos(b)*cos(phi)-sin(phi)*(sin(c)*sin(a)+cos(c)*sin(b)*cos(a))) + point.y*(cos(c)*sin(b)*sin(a)-sin(c)*cos(a)),
        .y = point.x*(sin(c)*cos(b)*cos(phi)-sin(phi)*(sin(c)*sin(b)*cos(a)-cos(c)*sin(a))) + point.y*(cos(c)*cos(a)+sin(c)*sin(b)*sin(a)),
        .z = point.x*(-sin(b)*cos(phi)-cos(b)*cos(a)*sin(phi)) + point.y*cos(b)*sin(a)
    };

    return rotated;
}
```

หากสังเกตดูดีๆ โจทย์ได้บอกไว้ว่า กล้องของเราต้องอยู่ที่พิกัด (0, 0, -1.5*r1*r2) โดยหันอยู่ในแนวแกน +Z หมายความว่า เราต้องผลักโดนัทของเราหลังจากการคูณกับ rotation matrix ไปในแนวแกน +Z ด้วยขนาด 1.5*r1*r2 หรือ เราต้องนำ vector ที่ได้ไปบวกกับ (0, 0, 1.5*r1*r2)

โอเค เราได้โดนัทของเราที่หมุนและเคลื่อนย้ายเรียบร้อยแล้ว แต่เราจะสามารถนำรูปโดนัทของเราที่เป็นรูปทรง 3 มิติแสดงแแกมาเป็น 2 มิติได้อย่างไรละ คำตอบคือ projection ในที่นี้ เราจะใช้ perspective projection ในการแปลงจุดในพิกัด 3 มิติ ออกมาเป็นจุดในพิกัด 2 มิติ โดยเราสามารถทำได้โดยการนำ vector 4 มิติ มาคูณกับ perspective projection matrix ได้ดังนี้

$$S = \frac{1}{tan(\frac{fov}{2})}$$

$$\begin{bmatrix} S/A & 0 & 0 & 0 \\ 0 & S & 0 & 0 \\ 0 & 0 & \frac{f}{(n-f)} & -1 \\ 0 & 0 & \frac{n*f}{(n-f)} & 0 \end{bmatrix} \begin{bmatrix} P_{dx} \\ P_{dy} \\ P_{dz} \\ 1 \end{bmatrix}$$

โดย fov คือ horizontal fov, n คือ near clipping plane, f คือ far clipping plane และ A คือ aspect ratio ซึ่งได้มาจากการนำความสูงของหน้าจอหารด้วยความกว้างของหน้าจอ (h/w)

อาจสงสัยว่า ทำไมจึงเป็น matrix4x4 สาเหตุก็เพราะในการคำนวณ projection นั้น สิ่งที่เราจะได้จากการนำ vector 4 มิติที่สามารถแปลงมาจาก vector 3 มิติและกำหนดค่าสุดท้าย (w) เป็น 1 นั้นก็เพราะค่า w ที่ได้หลังจากการคูณกับ projection matrix จะถูกนำไปใช้เพื่อทำ perspective divide หรือการหาร vector ลัพธ์ คือ นำ vec4(vec3, 1) * projection matrix แล้วนำไปหารด้วย w component เพื่อได้พิกัด 2 มิติใน x, y component ที่มีความเป็น perspective นั่นเอง

เราเกือบได้โดนัทของเราแล้ว แต่เรายังขาด depth testing และ flat shading อยู่

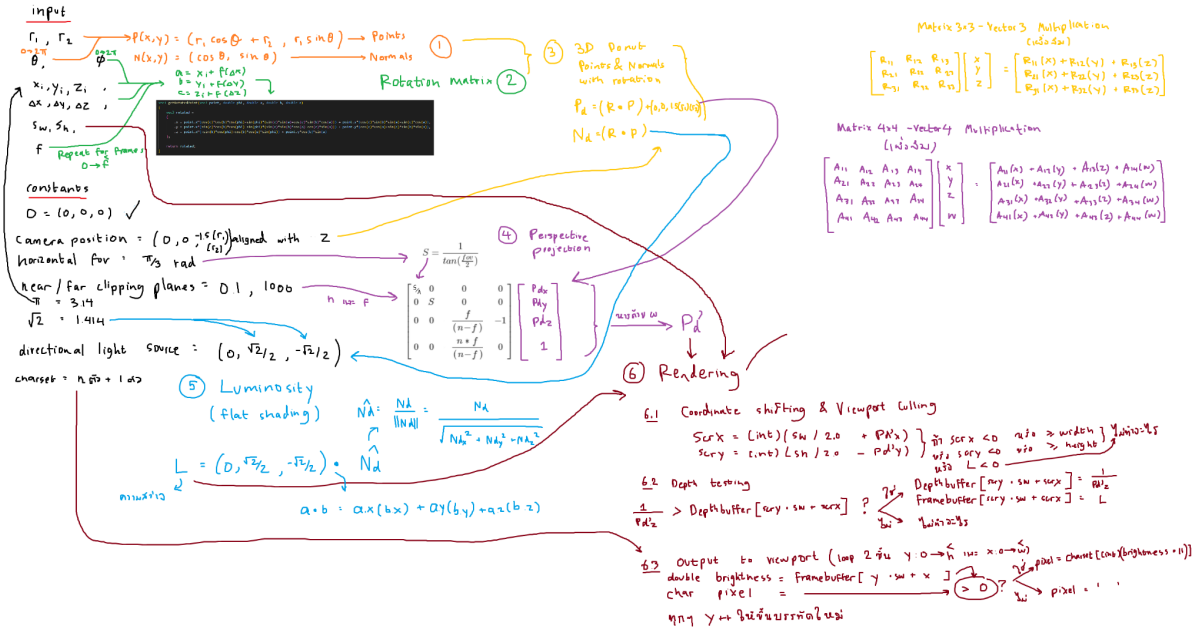
wootwoot

สำหรับ depth testing คือการทดสอบก่อนที่จะวาดแต่ละจุดว่า ณ ตำแหน่งนั้นบนหน้าจอ มีจุดที่วาดที่อยู่ด้านหน้าจุดใหม่แล้วหรือยัง เพราะถ้าหากมีแล้วก็ไม่ควรวาดจุดใหม่ทับ การคำนวณว่าจุดไหนอยู่หน้าอยู่หลังเราจะใช้ depth buffer ที่มี depth value เป็น 1/z จากพิกัดที่เราทำ perspective mapping เรียบร้อยแล้ว ถ้าหากค่า 1/z ของจุดใหม่มีค่ามากกว่าค่าเดิมใน depth buffer เราจึงจะกำหนดค่าใหม่ใน depth buffer ให้เป็น depth value ของจุดใหม่ มิฉะนั้นเราจะไม่วาดทับจุดฯนั้น

ท้ายสุด เราเหลือ flat shading หรือ การกำหนดค่าความสว่างให้กับจุดแต่จะจุด เราสามารถทำได้โดยการหา dot product ระหว่าง direction vector ของ directional light source กับ world space normal vector ของจุดบนโดนัท ซึ่งความจริงแล้ว object space normal vector ของจุดบนโดนัท มันก็คือสมการของวงกลมตรงๆ เลย เราก็เพียงต้องแค่นำ object space normal vector นั้นไปคูณกับ rotation matrix ตัวเดียวกับที่เราใช้กับการคำนวณพิกัดของโดนัทใน 3 มิติ เพื่อให้ได้ world space normals ออกมา แต่เราต้องไม่นำไปบวกกับ (0, 0, 1.5*r1*r2) เพราะเรากำลังทำการ transform normal vector อยู่ เพราะฉะนั้นการ translate หรือการ offset vector จะไม่มีผลต่อ normal vector ของเรา แล้วอย่าลืมนำ world space normals ที่ได้ไปหารด้วย magnitude ของตัวมันเองเพื่อทำการ normalize ด้วย

หลังจากนั้นเราก็แค่ remap ระบบพิกัดให้ origin อยู่ที่จุดกึ่งกลางของหน้าจอและกลับแกน y ให้เปลี่ยนจากล่างขึ้นบนเป็นบนไปล่าง แล้วทำการ culling ด้วย viewport/frustum culling กับ cull จุดที่มีค่าจาก dot product ก่อนหน้าที่น้อยกว่า 0 และแปลงออกเป็นตัวอักขระเพื่อนำเสนอบนหน้าจอตาม luminosity โดยทุกๆ frame และ scanline จะให้ขึ้นบรรทัดใหม่

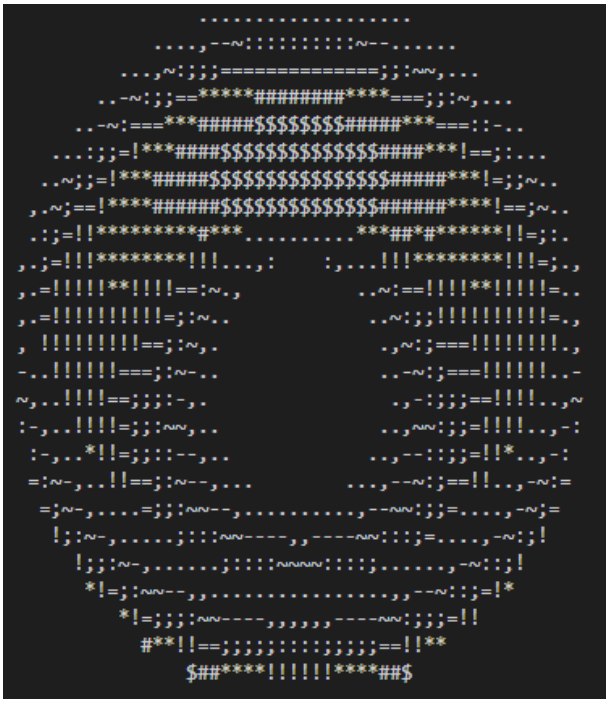
หมดแล้วกับหลักการวาดโดนัทของเรา เหลือแค่เพียงเขียนโค้ดเท่านั้น แต่ก่อนที่จะไปทำโจทย์ขอฝากแผนภาพสรุป basic rendering pipeline กับตัวอย่าง input output ไว้ก่อน (ซูมเอา)



wootwoot

ตัวอย่าง input output

Input: 0.5 1 0.07 0.02 1.57 0 0 0 0 0 50 25 1



Input: 0.5 1 0.07 0.02 0 0 1.57 0 0 0 50 25 1

wootwoot

Input: 1 2 0.07 0.02 1.57 0 0 1 0 -1 75 37.5 3