3D Rotation

By Oswald/Resource and Bitbreaker/Oxyron/Nuance

Derived from the rotation in one plane, these are the basic equations to rotate a point defined with its x, y, z coordinates around all 3 axises:

Rotation about the x axis:

x' = x

y' = cos(xangle) * y - sin(xangle) * z

z' = sin(xangle) * y + cos(xangle) * z

Rotation about the y axis:

x' = cos(yangle) * x + sin(yangle) * z

y' = y

z' = -sin(yangle) * x + cos(yangle) * z

Rotation about the z axis:

x' = cos(zangle) * x - (sin(zangle) * y

y' = sin(zangle) * x + (cos(zangle) * y

z' = z

As you work through the equations you must use the 'new' rotated versions of the coordinates after having rotated around one axis. thus after calculating the rotation first (in this example) around the X axis, then you must use x', y' and z' instead of x, y, z for the rotation around the Y axis, and so on.

As for rotating around all three axises it is however advisable to set up a rotation matrix once and then get a bunch of vertices rotated by multiplying the vertice's x, y and z-position with this matrix. When dealing with a few vertices like with a cube, there is easier means to rotate (most of all because a cube is a nice special case). But as a generic approach this is the fastest way to rotate around an arbitary axis.

For more information on that approach it is wise to first read at least the article “A Different Perspective: Three-Dimensional Graphics on the C64” from C= Hacking Issue 8 and C= Hacking Issue 9 before reading the upcoming code, as most of it is based on the ideas of those articles.

Prerequisites

The maths stuff needs quite some tables, most of all a x^2 table for fast multiplications, as well as some copies of that table with various offsets added, to make things even faster (that is where the optimization begins). Also we will need tables for sine and cosine and a factor-table for calculating the

perspective. Some of the coding tricks applied you will find further described here.

For generating all that we make use of a tiny c-program:

```
#include <math.h>
#include <stdio.h>
#include <inttypes.h>

#define PI atan2 (0.0, -1.0)

void main() {
    int8_t result[0x0b00] = { 0 };
    FILE* fw;

    float d, z0, r;
    int z, q, i, c, a, x, q2;
    int degrees = 180;

    //sinus and cosinus table
    r = 0x20;
    for (a = 0; a < degrees; a++) {
        result[a + 0x000] = round(sin(2.0 * PI * a / (float)degrees) * r);
        result[a + 0x100] = round(cos(2.0 * PI * a / (float)degrees) * r);
    }

    //fact_z table for doing the perspective
    d = 280.0; z0 = 5.0;
    for (i = 0; i < 0x100; i++) {
        z = i;
        if(z > 127) z = z - 256;
        q = round(d / (z0 - z / 64.0));
        if(i < 128)
            result[0x200 + i + 128] = q + 0x80;
        else
            result[0x200 + i - 128] = q + 0x80;
    }

    //x^2 tables with different offsets and index shift
    for(i = 0; i < 0x200; i++) {
        x = i & 255;
        if(x > 127) x = 256 - x;
        q = round(x * x / 256.0);
        //tmath1
        result[0x300 + i] = (q & 0xff);
        //tmath1_40
        result[0x500 + i] = (q & 0xff) + 0x40;
        //tmath1_80
        result[0x700 + i] = (q & 0xff) + 0x80;
        //tmath2
        result[0x900 + i - 1] = (q & 0xff);
    }
```

```
    fw = fopen("sinus.bin", "wb");
    fwrite(&result[0], 1, 0xb00, fw);
    fclose(fw);
}
```

On assembler side we define some macros to make the code appear cleaner and we set up some labels within the generated sinus.bin to have easy access to the tables:

```
!align 255,0
tmath1
!bin "sinus.bin",$200,$300

!align 255,0
tmath1_40
!bin "sinus.bin",$200,$500

!align 255,0
tmath1_80
!bin "sinus.bin",$200,$700

!align 255,0
tmath2
!bin "sinus.bin",$200,$900

!align 255,0
sinus
!bin "sinus.bin",45,$000

cosinus
!bin "sinus.bin",135,$000+45
!bin "sinus.bin",45,$000

!align 255,0
fact_z
!bin "sinus.bin",$100,$200

;arithmetic shift right
!macro asr {
!ifdef USE_ILLEGALS {
        anc #$fe
        ror
} else {
        ;copy bit 7 to carry
        cmp #$80
        ;rotate right -> if negative, bit 7 is set again, else it is
cleared -> arithmetic shift right
        ror
        clc
}
}
```

```
;adds two angles
!macro adda .a, .b {
        adc .a
        ;>255 -> in any case do a subtract
        bcs ++
        cmp #180
        ;>180 and < 256 -> so subtract
        bcc +
++
        sbc #180
        clc
+
        sta .b
}

;subtracts two angles
!macro suba .a, .b {
        sbc .a
        bcs +
        adc #180
+
        sta .b
}

;negates value
!macro neg {
        eor #$ff
        clc
        adc #$01
}

;sets up value and -value for fast multiply
!macro store .a, .nega {
        sta .a+1
        +neg
        sta .nega+1
}
```

Rotation matrix

Now comes the first fun part, some real long code to setup the rotation matrix. Let the code speak for itself, as it is well documented:

```
        ;check rotation angles for overflow
        ldx deg_x
        cpx #180
        bcc +
        ldx #$00
```

```
          stx deg_x
+
          lda deg_z
          cmp #180
          bcc +
          lda #$00
          sta deg_z
+
          lda deg_y
          cmp #180
          bcc +
!ifdef USE_ILLEGALS {
          anc #$00
} else {
          lda #$00
          clc
}
          sta deg_y
+
          ;carry is clear

          ;---------------------------------
          ;angles to add:
          ;---------------------------------

          ;t2 = sy+sz
          ;t8 = sy+sz-sx = t2-sx
          tay
          +adda deg_z, t2
          sec
          +suba deg_x, t8

          ;t9 = sy-sx
          ;t1 = sy-sz
          tya
          +suba deg_x, t9
          tya
          +suba deg_z, t1

          ;t4 = sx-sz
          ;t6 = sx-sy+sz = sx-t1
          txa
          +suba deg_z, t4
          txa
          +suba t1, t6

          ;t3 = sx+sz
          ;t5 = sx+sy+sz = sx+t2
          ;t7 = sx+sy-sz = sx+t1
          ;t10= sy+sx
          clc
```

```
        txa
        +adda deg_z, t3
        txa
        +adda t2, t5
        txa
        +adda t1, t7
        txa
        +adda deg_y, t10

        ;---------------------------------
        ;now do the easy stuff
        ;C = sin(sy)
        ;A = (cos(t1) + cos(t2)) / 2
        ;B = (sin(t1) - sin(t2)) / 2
        ;F = (sin(t9) - sin(t10)) / 2

        ;and the ugly stuff ... so we better cut this into small pieces ...
        ;D = ( sin(t3) - sin(t4)) / 2 + (-cos(t5) + cos(t6) + cos(t8) -
cos(t7)) / 4
        ;H = ( sin(t3) + sin(t4)) / 2 + (-cos(t5) + cos(t6) - cos(t8) +
cos(t7)) / 4
        ;E = ( cos(t3) + cos(t4)) / 2 + ( sin(t5) - sin(t6) - sin(t8) -
sin(t7)) / 4
        ;G = (-cos(t3) + cos(t4)) / 2 + (-sin(t5) + sin(t6) - sin(t8) -
sin(t7)) / 4

        ;for that we calc for the left part:
        ;D_ = sin(t3) - sin(t4)
        ;H_ = sin(t3) + sin(t4)
        ;E_ = cos(t3) + cos(t4)
        ;G_ = cos(t4) - cos(t3) = -cos(t3) + cos(t4)

        ;for the right part:
        ;tmp4 = -cos(t5) + cos(t6)
        ;tmp1 =  sin(t5) - sin(t6)
        ;tmp3 = -sin(t7) - sin(t8)
        ;tmp2 = -cos(t7) + cos(t8)

        ;finally sum all up and divide by two
        ;D = D_ + (tmp4 + tmp2) / 4
        ;H = H_ + (tmp4 - tmp2) / 4
        ;E = E_ + (tmp1 + tmp3) / 4
        ;G = G_ + (tmp3 - tmp1) / 4
        ;---------------------------------

        ;NOTE: values for sin and cos are already / 2, so one asr is enough
for those terms divided by 4. Term
        ;C has to be multiplied by two. All terms divided by 2 are divided
are no more divided.

        ;---------------------------------
```

```
        ;calc the easy stuff
        ;---------------------------------

        ;sin(y) * 2
        lda sinus,y
        ;clc
        adc sinus,y
        +store C_1, C_2

        ldx t1
        ldy t2

        ;cos(t1) + cos(t2)
        lda cosinus,x
        clc
        adc cosinus,y
        +store A_1, A_2

        ;sin(t1) - sin(t2)
        lda sinus,x
        sec
        sbc sinus,y
        +store B_1, B_2

        ldx t9
        ldy t10

        ;sin(t9) - sin(t10)
        lda sinus,x
        sec
        sbc sinus,y
        +store F_1, F_2

        ;cos(t9) + cos(t10)
        lda cosinus,x
        clc
        adc cosinus,y
        +store I_1, I_2

        ;---------------------------------
        ;calc left part of terms
        ;---------------------------------

        ldx t3
        ldy t4

        ;sin(t3) - sin(t4)
        lda sinus,x
        sec
        sbc sinus,y
        sta D_+1
```

```
        ;sin(t3) + sin(t4)
        lda sinus,x
        clc
        adc sinus,y
        sta H_+1

        ;cos(t3) + cos(t4)
        lda cosinus,x
        clc
        adc cosinus,y
        sta E_+1

        ;-cos(t3) + cos(t4) = cos(t4) - cos(t3)
        lda cosinus,y
        sec
        sbc cosinus,x
        sta G_+1

        ;---------------------------------
        ;calc first and second halfs of right part
        ;---------------------------------

        ldx t7
        ldy t8

        ;tmp2 = - cos(t7) + cos(t8)
        lda cosinus,y
        sec
        sbc cosinus,x
        sta tmp2

        ;tmp3 = -sin(t7) - sin(t8)
        lda #$00
        sec
        sbc sinus,x
        sec
        sbc sinus,y
        sta tmp3

        ldx t5
        ldy t6

        ;tmp1 = sin(t5) - sin(t6)
        lda sinus,x
        sec
        sbc sinus,y
        sta tmp1

        ;tmp4 = -cos(t5) + cos(t6)
        lda cosinus,y
```

```
          sec
          sbc cosinus,x
          sta tmp4+1

          ;---------------------------------
          ;sum all up and divide by two
          ;---------------------------------

          ;H = (tmp4 + tmp2) / 2 + D_
          clc
          adc tmp2
          +asr
D_        adc #$00
          +store D_1, D_2

          ;H = (tmp4 - tmp2) / 2 + H_
tmp4      lda #$00
          sec
          sbc tmp2
          +asr
H_        adc #$00
          +store H_1, H_2

          ;E = (tmp3 + tmp1) / 2 + E_
          lda tmp1
          clc
          adc tmp3
          +asr
E_        adc #$00
          +store E_1, E_2

          ;G = (tmp3 - tmp1) / 2 + G_
          lda tmp3
          sec
          sbc tmp1
          +asr
G_        adc #$00
          +store G_1, G_2
```

Rotate and project

Note the bunch of values set up statically for all vertices to be transformed and projected (A-I). Thus we now save a lot of time by focusing on the variable part, our vertices. Also, if you imagine for e.g. a cube, you will see that the values for 4 vertices will always be the same for a single axis. In my case i sorted the vertices by Z (of course the indexes of your faces have to follow that sorting as well) to be able to cache the calculations for Z. Also i added another table to the mesh. This way i can set up the next point on when to update Z easily and with less cycles. By adding an offset of $80 to the first value of each multiplication of a matrix column we can forgo on any setting and clearing of the carry, as results will go from $40 to $c0 as a maximum this way, and we never under- nor overflow. Biggest

problem is, that we would best need two index-registers for the adc/sbc, but also need it for storing the results. That also forces us to propagate the resulting indexes from each summed up multiplication via zeropage instead of a simple register transfer (sta .persp+1 + lda $xxxx instead of for e.g. tay lda $xxxx,y)

```
!macro project_z_cached ~.vertx, ~.verty, ~.vertz, ~.vertc, ~.A1, ~.B1,
~.C1, ~.D1, ~.E1, ~.F1, ~.G1, ~.H1,
~.I1, ~.A2, ~.B2, ~.C2, ~.D2, ~.E2, ~.F2, ~.G2, ~.H2, ~.I2 {
.start
        sec
        ;update index value for next change in Z
.vertc  lda $1000,x
        sta .next+1

        ;update values for Z
.vertz  ldy $1000,x

        ;as long as vert_z is the same, the value for its multiplication is
the same, as the other multiplicand
        ;only changes per rotation
.G1     lda tmath1_80,y
.G2     sbc tmath1,y
        sta .z1+1

.A1     lda tmath1_80,y
.A2     sbc tmath1,y
        sta .z2+1

.D1     lda tmath1_80,y
.D2     sbc tmath1,y
        sta .z3+1
        ;carry is always set
        ;sec
-
        ;backup x
        stx tmp1
        ;load registers with x and y coordinates
.verty  ldy $1000,x
.vertx  lda $1000,x       ;meh, can't we use lax somehow?
        tax               ;13

        ;now we load the first value with $80 added beforehand, as we now
add and subtract 2 values that are
        ;all below $40 we will neither overflow nor underflow thus the
carry for the next addition/subtraction
        ;is predictable, and even if it is set when adding and cleared when
subtracting, this is no problem.
        ;as a + 1 - b - 1 = a - b

        ;z' = (G * vz + H * vx + I * vy) + $80
```

```
.z1     lda #$00
.H1     adc tmath1,x
.H2     sbc tmath1,x
.I1     adc tmath1,y
.I2     sbc tmath1,y

        ;now we would need to subtract $80 again by doing an eor #$80 to
avoid clobbering of the carry
        ;however we can calculate that offset into the fact_z-table
(perspective correction)
        ;also we have added the offset of $80 again to the values, that
saves us the eor #$80 before the
        ;multiplications that result in the final x/y-position

        ;set index in perspective-table without clobbering y or x

        sta .persp+1        ;22
.persp  lda fact_z
        sta z1
        eor #$ff
        sta z2              ;12

        ;x' = (A * vz + B * vx + C * vy) + $80
.z2     lda #$00
.B1     adc tmath1,x
.B2     sbc tmath1,x
.C1     adc tmath1,y
.C2     sbc tmath1,y
        ;set index without clobbering y or x
        sta tmp2            ;21

        ;y' = (D * vz + E * vx + F * vy) + $80
.z3     lda #$00
.E1     adc tmath1,x
.E2     sbc tmath1,x
.F1     adc tmath1,y
.F2     sbc tmath1,y
        tay                 ;20
        ;restore X
        ldx tmp1
        ;calc perspective y'
        ;yp = y' * fact_z[z' - $80] + $40 (-> unsigned)
        ;z1 points to tmath1_40, a square-table with an added offset of $40
        ;z2 points to tmath2, a square-table with an index-offset of 1 (as
clc adc #$01 is omitted on negation)
        lda (z1),y
        ;carry is still set and upcoming sbc does not underflow
        sbc (z2),y
        sta vertices_y,x    ;18

        ;restore x'
```

```
        ldy tmp2
        ;calc perspective x'
        ;xp = x' * fact_z[z' - $80] + $40 (-> unsigned)
        lda (z1),y
        ;carry still set
        sbc (z2),y
        sta vertices_x,x   ;18
        dex
.next   cpx #$00
        bpl -              ;7   branch as long as we don't underrun. In
that case also carry is always set for free \o/
;--------------------------;131 cycles per simple loop
        +lbcc .start       ;    if negative and carry clear then x got
below 0
}
```

Zooming

Zooming can be applied by multiplying the rotation-matrix with a factor. However you can also do a fake zoom when just changing the offset into the fact_z table for perspective. For that, the table needs to be extended to reach from zero to maximum zoom. Also the added offset (optimization) has to be removed and done in software. But you'll get a zoom for 2-3 cycles extra per vertice (eor #$ff + index crossing a page when reading from the fact_z table).