```
                   ########
             ##################
         ######            ######
      #####
    #####  ####  ####      ##      #####    ####  ####  ####  ####  ####   #####
  #####    ##    ##      ####    ##   ##     ##  ###     ##    ####  ##   ##   ##
 #####    ########     ##  ##    ##        #####        ##    ## ## ##    ##
#####    ##    ##    ########  ##   ##   ##  ###      ##    ##  ####   ##   ##
#####  ####  ####  ####  ####  #####    ####  ####  ####  ####  ####   ######
#####                                                                     ##
 ######            ######           Issue #16
   ##################            April 26, 1998
       ########
```

....

Knowledge is power. -- Nam et ipsa scientia potestas est.

Francis Bacon, Meditationes Sacrae

....

BSOUT

The number 16 is an important number for Commodore folks. Commodore's most famous computer is so named for having 2^16 bytes. I'm sure many of us were greatly influenced by Commodore as 16-year olds. And it was just a little over sixteen years ago that the Commodore 64 first became available.

I want to convey to you what a remarkable fact this is. After sixteen years the Commodore 64 is not only still being used, but the entire community is still moving forwards. People are still using the machine seriously, new and innovative hardware and software is still being developed, and new and innovative uses for these old machines are still being found. I do not believe any other computer community can make this claim. How cool is that?

Thus does issue #16 boldly stride forwards into the deep realms of the Commodore unknown, secure in the knowledge that the Commodore community will blaze brightly well into the future, and in eager anticipation of the undiscovered algorithms and schematics which lie just around the corner.

And now a few words on the future of C=Hacking. As everyone knows, C=Hacking has been pining for the fjords for a while now. Now that it is once again displaying its beautiful plumage, I hope to keep new issues appearing reasonably regularly. My current thinking is that C=Hacking will appear on a "critical mass" basis: once enough articles have arrived, a new issue will be released. I will of course be meanwhile digging around for material and authors, and we'll see if four issues per year is too optimistic (one caveat: I will be trying to graduate soon, so you might have to wait a little bit longer than normal for the next issue or two).

I also expect to slim the issues down somewhat. The focus will now be technical -- as Jim mentioned in issue #15, the nontechnical stuff will move to another mag. Instead of a behemoth magazine with tons of articles, I am going to try using a critical mass of 2-3 main articles plus some smaller articles. (This might also make it possible to release issues more often).

The magazine now has four sections: Jiffies, The C=Hallenge, Side Hacking, and the main articles. Jiffies are just short, quickie and perhaps quirky programs, in the flavor of the old RUN "Magic" column, or "Bits & Pieces" from The Transactor. The C=Hallenge presents a challenge problem for readers to submit solutions to, to be published in future issues. Side Hacking is for articles which are too small to be main articles but nifty in their own right. Thus there is now room for articles of all sizes, from monstrous to a mere screenful. With the first two sections I am hoping to stimulate reader involvement, and we'll see if it works or not. In this issue I have included at least one of each type of article, to give a flavor of the different sections.

Otherwise, things ought to be more or less the same. I'd like to thank Jim Brain for keeping C=Hacking going over the last few years, and also for the use of the jbrain.com web site. I'd also like to say that although this issue seems to be the "Steve and Pasi issue", there's no particular reason to expect that to be the case for future issues.

And now... onwards!

....

....

```
..
.                                      C=H

:::::::::::::::::::::::::::::::::: Contents ::::::::::::::::::::::::::::::::::
```

```
................................. Stuff ....................................

Legalities
----------

        Rather than write yet another flatulent paragraph that nobody will
ever read, I will now make my own little contribution to dismantling
the 30 years' worth of encrustation that has led to the current Byzantine
US legal system:

        C=Hacking is a freely available magazine, involving concepts which
may blow up your computer if you're not careful.  The individual authors
hold the copyrights on their articles.

Rather than procedure and language, I therefore depend on common sense and
common courtesy.  If you have neither, then you probably shouldn't use
a 64, and you definitely shouldn't be reading C=Hacking.  Please email
any questions or concerns to chacking-ed@mail.jbrain.com.

General Info
------------

-For information on subscribing to the C=Hacking mailing list, send email to

        chacking-info@mail.jbrain.com

-For general information on anything else, send email to

        chacking-info@mail.jbrain.com

-For information on chacking-info@mail.jbrain.com, send email to

        chacking-info@mail.jbrain.com

To submit an article:

        - Invest $5 in a copy of Strunk & White, "The Elements of Style".
```

        - Read it.
        - Send some email to chacking-ed@mail.jbrain.com

Note that I have a list of possible article topics. If you have a topic you'd like to see an article on please email chacking@mail.jbrain.com and I'll see what I can do!

```
.......
....
..
.                                    C=H
```

.... Jiffies ....

Well folks, this issue's Jiffy is pretty lonely. I've given it a few friends, but I hope you will send in some nifty creations and tricks of your own.

$01 From John Ianetta, 76703.4244@compuserve.com:

This C-64 BASIC type-in program is presented here without further comment.

```
10 printchr$(147)
20 poke55,138:poke56,228:clr
30 a$="ibm":print"a$ = ";a$
40 b$="macintosh"
50 print:print"b$ = ";b$
60 print:print"a$ + b$ = ";a$+b$
70 poke55,.:poke56,160:clr:list
```

$02 I am Commodore, here me ROR!

First trick: performing ROR on a possibly signed number. Don't blink!

```
        CMP #$80
        ROR
```

Second trick: performing a cyclical left shift on an 8-bit number.

```
        CMP #$80
        ROL
```

Oooohh! Aaaaahhh! Another method:

```
        ASL
        ADC #00
```

```
........
....
..
.                                    C=H
```

.... The C=Hallenge ....

The chacking challenge this time around is very simple: write a program which clears the screen.

This could be a text screen or a graphics screen. Naturally ? CHR$(147) is awfully boring, and the point is to come up with an interesting algorithm -- either visually or code-wise -- for clearing the screen.

The purpose here is to get some reader involvement going, so submit your solutions to

        chacking-ed@mail.jbrain.com

and chances are awfully good that you'll see them in the next issue! (Source code is a huge bonus). And, if you have a good C=Hacking C=Hallenge problem, by all means please send it to the above address.

```
........
....
..
.                                    C=H
```

.... Side Hacking ....

PAL VIC20 Goes NTSC

by Timo Raita <vic@iki.fi> http://www.iki.fi/vic/
Pasi 'Albert' Ojala <albert@cs.tut.fi> http://www.cs.tut.fi/~albert/

Introduction

Recently Marko Mäkelä organized an order from Jameco's C= chip closeout sale, one of the items being a heap of 6560R2-101 chips. When we had the chips, we of course wanted to get some of our PAL machines running with these NTSC VIC-I chips. A couple of weeks earlier Richard Atkinson wrote on the cbm-hackers mailing list that he had got a 6560 chip running on a PAL board. He used an oscillator module, because he couldn't get the 14.31818 MHz crystal to oscillate in the PAL VIC's clock generator circuit.

We checked the PAL and NTSC VIC20 schematic diagrams but couldn't notice a significant difference in the clock generator circuits. There seemed to be no reason why a PAL machine could not be converted into NTSC machine fairly easily by changing the crystal and making a couple of small changes. Adding a clock oscillator felt a somewhat desperate quick'n'dirty solution.

Note that some old television sets require you to adjust their vertical hold knob for them to show 60 Hz (NTSC) frame rate. Some recent pseudo-intelligent televisions only display one frame rate (50 Hz in PAL-land). Multistandard television sets and most video monitors do display 60 Hz picture correctly. There is a very small chance that your display does not like the 60 Hz frame rate. Still, be careful if you haven't tried 60 Hz before.

You should also note that PAL and NTSC machines use different KERNAL ROM versions, 901486-06 is for NTSC and 901486-07 is for PAL. However, the differences are small. In an NTSC machine with a PAL ROM the screen is not centered, the 60 Hz timer is not accurate and the RS-232 timing values are wrong.

The Story

Timo:

At first I took a VIC20CR board (FAB NO. 251040-01) and just replaced the videochip and the crystal, and as you might suspect, it didn't work. I noticed that the two resistors in the clock generator circuit (R5 and R6) had a value of 470 ohms, while the schematics (both NTSC and PAL!) stated that they should be 330 ohms. I replaced those resistors, and also noticed the single 56 pF capacitor on the bottom side of the board. This capacitor was connected to the ends of the R5 resistor and was not shown in the schematics. As you might guess, the capacitor prevents fast voltage changes, and thus makes it impossible to increase the frequency.

Is this capacitor present also on NTSC-board? Someone with such a board could check this out. I removed the capacitor, and now it works. I didn't test the board between these two modifications, but Pasi confirmed that you only need to remove the capacitor.

Pasi:

I first tried to convert my VIC20CR machine, because the clock circuit in it seemed identical to the one a newer NTSC machine uses, except of course the crystal frequency. Some of the pull-up resistors were different, but I didn't think it made any difference. Pull-up resistors vary a lot without any reason anyway. I replaced the video chip and the crystal, but I could not get it to oscillate. I first thought that the 7402 chip in the clock circuit just couldn't keep up and replaced it with a 74LS02 chip. There was no difference. The PAL crystal with a PAL VIC-I (6561) still worked.

I turned my eyes to my third VIC20, which is an older model with all that heat-generating regulator stuff inside. It has almost the same clock circuit as the old NTSC schematic shows. There are three differences:

1. The crystal is 14.31818 MHz for NTSC, 8.867236 MHz for PAL.
2. Two NOR gates are used as NOT gates to drop one 74S04 from the design.
3. In PAL the crystal frequency is divided by two by a 7474 D-flipflop.

I could either put in a 28.63636 MHz crystal or I could use the 14.31818 MHz crystal and bypass the 7474 clock divider. I didn't have a 28 MHz crystal, so I soldered in the 14.31818 MHz crystal and bent the pin 39 (phi1 in) up from the 6560 video chip so that it would not be connected to the divided clock. I then soldered a wire connecting this pin (6560 pin 39) and the 14.31818 MHz clock coming from the 7402 (UB9 pin 10). The machine started working.

I just hadn't any colors. My monitor (Philips CM8833) does not show NTSC colors anyway, but a multistandard TV (all new Philips models at least) shows colors as long as the VIC-I clock is close enough to the NTSC color clock. The oscillator frequency can be fine-adjusted with the trimmer capacitor C35. Just remember that using a metallic unshielded screwdriver is a bad idea because it changes the capacitance of the clock circuit (unless the trimmer is an insulated model). Warming has also its effect on the circuit capacitances so let the machine be on for a while before being satisfied with the adjustment. With a small adjustment I had colors and was done with that machine.

Then I heard from Timo that the CR model has a hidden capacitor on the solder side of the motherboard, probably to filter out upper harmonic frequencies (multiples of 4.433618 MHz). I decided to give the VIC20CR modification another try. I removed the 56 pF capacitor, which was connected in parallel with R5, and the machine still worked fine.

I then replaced the crystal with the 14.31818 MHz crystal and inserted the 6560 video chip. The machine didn't work. I finally found out that it was because I had replaced the original 7402 with 74LS02. When I replaced it with a 74S02, the machine started working. I just could not get the frequency right and thus no colors until I added a 22 pF capacitor in parallel with the capacitor C50 and the trimmer capacitor C48 to increase the adjustment range from the original 5..25 pF to 27..47 pF. The trimmer orientation didn't have any visible effect anymore. I had colors and was satisfied.

To check all possibilities, I also replaced the 74S02 with 7402 that was originally used in the circuit (not the same physical chip because I had butchered it while soldering it out). I didn't need the parallel capacitor anymore, although the trimmer adjustment now needed to be correct or I lost colors.

As I really don't need two NTSC machines, I then converted this machine back to PAL. I made the modifications backwards. I replaced the crystal and video chip and then was stumped because the machine didn't work. I scratched my head for a while but then remembered the extra capacitor I had removed. And surely, the machine started working when I put it back. Obviously, the machine had only worked without the capacitor because it had 74LS02 at the time. 74S02 and 7402 won't work without it. So, if you are doing an NTSC to PAL conversion, you need to add this capacitor.

Summary

PAL VIC20 To NTSC

This is the older, palm-heating VIC20 model with the two-prong power connector and the almost-cube power supply.

1. Replace the 8.867236 MHz crystal with a 14.31818 MHz crystal
2. If UB9 is 74LS02, replace it with a 7402 (or 74S02 if you only use NTSC)
3. Bend pin 39 from the 6560 video chip so that it does not go to the socket.
4. Add a jumper wire from UB9 pin 10 to the video chip pin 39.
5. Adjust the trimmer capacitor C35 so that your multistandard television shows colors.

PAL VIC20CR To NTSC

1. Replace the 4.433618 MHz crystal with a 14.31818 MHz crystal
2. If your machine has a capacitor in parallel with R5, remove it (the parallel capacitor). The values in our machines were 56 pF.
3. If UB9 is 74LS02, replace it with a 7402 (or 74S02 if you only use NTSC)
4. If necessary, increase the clock adjustment range by adding a capacitor of 11..22 pF in parallel to C50 (15 pF in the schematics, 5 pF in my machine) and C48 (the trimmer capacitor 0..20 pF). With 7402 you can do with a smaller capacitor or with no additional capacitor at all.
5. Adjust C48 so that your multistandard television shows colors.

Trouble-shooting

```
     * There is no picture
          + You have not removed (or added for NTSC) the 56 pF capacitor
            - Remove/Add the capacitor
          + Your clock crystal is broken
            - Find a working one
          + Your machine has a 74LS02 instead of 7402 (or 74S02)
            - Replace the 74LS02 with a 7402
          + You forgot to replace the PAL video chip with the NTSC chip
            - Remove 6561 and insert 6560
          + For older model: You forgot the clock wire and/or to turn up
            6560 pin 39
            - Connect 6560 pin 39 to UB9 pin 10
     * There is picture, but it is scrolling vertically
          + Your monitor or television does not properly sync to 60 Hz
            frame rate
            - Adjust the monitor/TV's vertical hold setting
     * There is picture but no colors
          + Your monitor or TV does not understand the color encoding
            - Give up the color or get a better television
          + The color clock frequency is slightly off
            - Adjust the trimmer capacitor
          + The color clock frequency adjustment range is not enough
            - Add a 11-22 pF capacitor in parallel with the trimmer
```

::

Starfields, by S. Judd

If you've ever played Elite or Star Raiders you've experienced a starfield. The idea is to use stars to give a sense of motion while navigating through the universe. In this article we'll derive a simple algorithm for making a starfield -- the algorithm I used in Cool World.

As usual, a good way to start is with a little thinking -- in this case, we first need to figure out what the starfield problem even is! Consider the motion of the stars. As we move forwards, the stars should somehow move past us -- over us, to the side of us, etc. (A star coming straight at us wouldn't appear to move at all). And something that moves needs a starting place and an ending place. Certainly they end at the screen boundaries, and since anything really far away looks like it's right in front of us they must start at (or near) the center of the screen.

So far so good: stars start near the center of the screen and move outwards to the edges. What kind of path do they follow? Well it can't be a curve, right? If it were a curve, then all stars would have to curve in exactly the same way, since there is a rotation symmetry (just tilting your head won't alter the shape of the path of the stars). So it has to be a straight line. Since stars start at the center of the screen and move outwards, we know that, wherever a star might be on the screen right now, it started in the center. So, to get the path of the star's motion, we simply draw a line from the center of the screen through the star's current location, and move the star along that line.

Drawing lines is easy enough. If the center of the screen is located at (xc, yc) then the equation of a line from the center to some point (x0, y0) is just

```
        (x,y) = (xc, yc) + t*(x0-xc, y0-yc)
```

You should read a vector equation like the above as

```
        for t=blah to wherever step 0.0001
          x = xc + t*(x0-xc)
          y = yc + t*(y0-yc)
          plot(x,y)
        next
```

and you can see that when t=0

```
        (x,y) = (xc, yc) i.e. the center of the screen
```

and when t=1

```
        (x,y) = (x0, y0) i.e. the current location.
```

Thus, when t goes from 0 to 1 we hit all points in-between, and when t is a very small number (like 0.0001) we basically move from (xc,yc) to the point on the line right next to (xc,yc).

Great, now we know the path the stars take. But _how_ do they move along that path? We know from experience that things which are far away appear to move slowly, but when we're up close they move really fast (think of a jet plane, or the moon). And we know that in this problem that stars which are far away are near the center of the screen, and stars which are nearby are out near the edges. So, stars near the center of the screen ought to move slowly, and as they move outwards they ought to increase in speed.

Well, that's easy enough to do. We already have an easy measure of how far away we are from the center:

```
        dx = x0-xc
        dy = y0-yc
```

But this is the quantity we need to compute to draw the line. All we have to do is move some fraction of (dx,dy) from our current point (x0,y0):

```
        x = x0 + dx/8    ;Note that we started at x0, not xc
        y = y0 + dy/8
```

where I just divided by 8 for the heck of it. Dividing by a larger number will result in slower motion, and dividing by a smaller number will result in faster motion along the line. But the important thing to notive in the above equations is that when a star is far away from the center of the screen, dx and dy are large and the stars move faster.

Well alrighty then. Now we have an algorithm:

```
        draw some stars on the screen
        for each star, located at (x,y):
          erase old star
          compute dx = x-xc and dy = y-yc
          x = x + dx*velocity
          y = y + dy*velocity
          plot (x,y)
        keep on going!
```

where velocity is some number like 1/8 or 1/2 or 0.14159 or whatever. This is enough to move us forwards (and backwards, if a negative value of velocity is used). What about moving up and down, or sideways? What about rotating?

First and foremost, remember that, as we move forwards, stars always move in the same way: draw a straight line from the origin through the star, and move along that path. And that's the case no matter where the star is located. This means that to rotate or move sideways, we simply move the position of the stars. Then using the above algorithm they will just move outwards from the center through their new position. In other words, rotations and translations are pretty easy.

Sideways translations are easy: just change the x-coordinates (for side to side translations) or y-coordinates (for up and down translations). And rotations are done in the usual way:

```
        x = x*cos(t) - y*sin(t)
        y = x*sin(t) + y*cos(t)
```

where t is some rotation angle. Note that you can think of sideways motion as moving the center of the screen (like from 160,100 to 180,94).

Finally, what happens when stars move off the screen? Why, just plop a new star down at a random location, and propagate it along just like all the others. If we're moving forwards, stars move off the screen when they hit the edges. If we're moving backwards, they are gone once they get near enough to the center of the screen.

Now it's time for a simple program which implements these ideas. It is in BASIC, and uses BLARG (available in the fridge) to do the graphics. Yep, a SuperCPU comes in really handy for this one. Rotations are also left out, and I put little effort into fixing up some of the limitations of the simple algorithm (it helps to see them!). For a complete ML implementation see the Cool World source code.

```
10 rem starfield
15 mode16:gron16
20 dim x(100),y(100):a=rnd(ti)
25 xc=160:yc=100:n=12:v=1/8
30 for i=1 to n:gosub200:next
40 :
50 for i=1 to n
55 color0:plot x(i),y(i):color 1
60 dx=x(i)-xc:dy=y(i)-yc
```

```
65 x1=x(i)+v*dx+x0:y1=y(i)+v*dy+y0
67 x2=abs(x1-x(i)):y2=abs(y1-y(i)):if (x2+y2<3*abs(v)) then gosub 200:goto 60
70 if (x1<0) or (y1<0) or (x1>319) or (y1>199) then gosub 200:dx=0:dy=0:goto65
75 plot x1,y1:x(i)=x1:y(i)=y1
80 next
90 get a$
100 if a$=";" then x0=x0-3:goto 40
110 if a$=":" then x0=x0+3:goto 40
120 if a$="@" then y0=y0-3:goto 40
130 if a$="/" then y0=y0+3:goto 40
133 if a$="a" then v=v+1/32
135 if a$="z" then v=v-1/32
140 if a$<>"q" then 40
150 groff:stop
200 x(i)=280*rnd(1)+20:y(i)=170*rnd(1)+15:return
```

Line 200 just plops a new star down at a random position between x=20..300 and y=15..185. Lines 100-140 just let you "navigate" and change the velocity. The main loop begins at line 50:

```
55 erase old star
60 compute dx and dy
65 advance the star outward from the center of the screen
67 check if the star is too close to the center of the screen (in case
   moving backwards)
70 if star has moved off the screen, then make a new star
```

And that's the basic idea. Easy! It's also easy to make further modifications -- perhaps some stars could move faster than others, some stars could be larger that others, or different colors, and so on. Starfields are fast, easy to implement, and make a nifty addition to many types of programs.

::

Milestones of C64 Data Compression

Pontus Berg, bacchus@fairlight.org

One of the featured articles in this issue is on data compression. A very natural question to ask is: what about data compression on the C-64? The purpose of this article is therefore to gain some insight into the history of data compression on the 64. This article doesn't cover programs like ARC, which are used for storage/archiving, but instead focuses on programs which are compressed in executable format, i.e. decompress at runtime.

The earliest instance of compression comes with all computers: the BASIC ROMs. As everyone knows, BASIC replaces keywords with one-byte tokens. This not only takes less memory, but makes the program run faster as well. Clearly, though, the BASIC interpreter is a very special situation.

The general problem of 64 data compression is to *reversibly* replace one chunk of data with a different chunk of data which is shorter *on the average*. It won't always be shorter, simply because there are many more big chunks of data than there are small ones. The idea is to take advantage of any special structure in the data. The data in this case is 64 machine language programs, including graphics data, tables, etc.

The early years didn't feature any compression, but then "packers" (RLE compression) were introduced. The first packer I used myself was flash packer, but I can't tell if it was an early one. The idea of RLE -- run length encoding -- is simply to replace repeated bytes with a single byte and the number of times to repeat it. For example, AAAAAA could be replaced by xA6, where x is a control character to tell the decompressor that the next two bytes represent a run.

Then came the TimeCruncher, in 1986 or perhaps 1987. Basically, one can divide the world into compression before and after the TimeCruncher by Macham/Network. With TimeCruncher, Matcham introduced sequence crunching -- this is THE milestone in the evolution. The idea of sequencing is the same as LZ77: replace repeated byte sequences with a *reference* to earlier sequences. As you would imagine, short sequences, especially of two or three bytes, are very common, so methods of handling those cases tend to pay large dividends. See Pasi's article for a detailed example of LZ77 compression. It is worth noting that several 64 compression authors were not aware of LZ77 when designing their programs!

Naturally the 64 places certain limitations on a compression algorithm. With typical 64 crunchers you define a scanning range, which is the range in which the sequence scanner looks for references. The sequence cruncher replaces parts of the codes with references to equal sequences in the program. References are relative: number of bytes away and length. Building this reference in a smart and efficient way is the key to success. References that are far away require more bits, so a "higher speed" (bigger search area) finds more sequences, but the references are longer.

The next step of value was CruelCrunch, where Galleon (and to some extent Syncro) took the concept to where it could be taken. Following that, the next step was introducing crunchers which use the REU, where Antitrack took DarkSqeezer2 and modified it into a REU version. Actually this was already possible in the original Darksqeezer 3, but that was not available to ATT. Alex was pissing mad when it leaked out (rather soon) ;-).

The AB Cruncher, followed by ByteBoiler, was really the first cruncher to always beat the old CruelCrunch. With the REU feature and public availablility, this took the real power of crunchers to all users. It should be noted that the OneWay line of crunchers (ByteBoiler and AB Crunch, which doesn't even require an REU) actually first scan the files and then optimize their algorithms to perform their best, rather than letting the user select a speed and see the results after each.

Regarding compression times, a char/RLE packer typically takes the time it takes to read a file twice, as they usually feature two passes -- one for determining the bytes to use as controlbytes and one to create the packed file. Sequence crunchers like TimeCruncher typically took a few hours, and CruelCrunch as much as ten hours (I always let it work over night so I can't tell for sure - it's not something you clock while watching ;-). After the introduction of REU-based sequence crunchers (which construct tables of the memory contents and do a table lookup, rather than repeatedly scanning the data), and their subsequent optimization, the crunching times went down first to some 30 minutes and then to a few minutes. ByteBoiler only takes some two minutes for a full 200 block program, as I recall.

The RLE packers and sequence crunchers are often combined. One reason was the historical time saving argument - a file packed first would be smaller upon entering the crunching phase which could hence be completed much faster. A very sophistocated charpacker is however a big waste as the result they produce - a block or so shorther at best - is almost always eaten up by worse crunching. Some argue that you could as well crucnh right away without first using the packer, but then again you have the other argument - a charpacker can handle a file of virtually any length (0029 to ffff is available) whereas normally a cruncher is slightly more limited.

Almost any game or demofile (mixed contents of graphics, code, musicdata and player, etc.) normally packs into some 50-60% of its original size when using an RLE+sequence combination. The packer might compress the file by some 30%, depending on the number of controlbytes and such, and the cruncher can compress it an additional 10-20%.

Minimising the size was the key motivation for the development of these programs -- to make more fit on the expensive disks and to make them load faster from any CBM device (we all know the speed of them ;-). For the crackers one could mention the levelcrunchers as well. This is a way to pack data and have it depack transparently while loading the data, as opposed to adding a depacker to be run upon execution. The very same crunching algorithms are used, and the same programs often come in a level- and filecrunching edition.

```
.......
....
..
.                                       C=H
```

::

Featured Articles

::

Compression Basics

Pasi 'Albert' Ojala   albert@cs.tut.fi
http://www.cs.tut.fi/%7Ealbert/

Introduction

Because real-world files usually are quite redundant, compression can often reduce the file sizes considerably. This in turn reduces the needed storage size and transfer channel capacity. Especially in systems where memory is at premium compression can make the difference between impossible and implementable. Commodore 64 and its relatives are good examples of this kind of a system.

The most used 5.25-inch disk drive for Commodore 64 only holds 170 kB of data, which is only about 2.5 times the total random access memory of the machine. With compression, many more programs can fit on a disk. This is especially true for programs containing flashy graphics or sampled sound. Compression also reduces the loading times from the notoriously slow 1541 drive, whether you use the original slow serial bus routines or some kind of a disk turbo loader routine.

Dozens of compression programs are available for Commodore 64. I leave the work to chronicle the history of the C64 compression programs to others and concentrate on a general overview of the different compression algorithms. Later we'll take a closer look on how the compression algorithms actually work and in the next article I will introduce my own creation: pucrunch.

Pucrunch is a compression program written in ANSI-C which generates files that automatically decompress and execute themselves when run on a C64 (or C128 in C64-mode, VIC20, or C16/+4). It is a cross-compressor, if you will, allowing you to do the real work on any machine, like a cross-assembler.

Our target environment (Commodore 64 and VIC20) restricts us somewhat when designing the 'ideal' compression system. We would like it to be able to decompress as big a program as possible. Therefore the decompression code must be located in low memory, be as short as possible, and must use very small amounts of extra memory.

Another requirement is that the decompression should be relatively fast, which means that the arithmetic used should be mostly 8- or 9-bit which is much faster than e.g. 16-bit arithmetic. Processor- and memory-intensive algorithms are pretty much out of the question. A part of the decompressor efficiency depends on the format of the compressed data. Byte-aligned codes can be accessed very quickly; non-byte-aligned codes are much slower to handle, but provide better compression.

This is not meant to be the end-all document for data compression. My intention is to only scratch the surface and give you a crude overview. Also, I'm mainly talking about lossless compression here, although some lossy compression ideas are briefly mentioned. A lot of compression talk is available in the world wide web, although it may not be possible to understand everything on the first reading. To build the knowledge, you have to read many documents and understand _something_ from each one so that when you return to a document, you can understand more than the previous time. It's a lot like watching Babylon 5. :-)

Some words of warning: I try to give something interesting to read to both advanced and not so advanced readers. It is perfectly all right for you to skip all uninteresting details. I start with a Huffman and LZ77 example so you can get the basic idea before flooding you with equations, complications, and trivia.

Huffman and LZ77 Example

Let's say I had some simple language like "Chippish" containing only the letters _CDISV_. How would a string like

    SIDVICIIISIDIDVI

compress using a) Huffman encoding, and b) LZ77? How do compression concepts such as information entropy enter into this?

A direct binary code would map the different symbols to consequtive bit patterns, such as:

Symbol	Code
'C'	000
'D'	001
'I'	010
'S'	011
'V'	100

Because there are five symbols, we need 3 bits to represent all of the possibilities, but we also don't use all the possibilities. Only 5 values are used out of the maximum 8 that can be represented in 3 bits. With this code the original message takes 48 bits:

```
    SIDVICIIISIDIDVI ==
    011 010 001 100 010 000 010 010 010 011 010 001 010 001 100 010
```

For Huffman and for entropy calculation (entropy is explained in the next chapter) we first need to calculate the symbol frequencies from the message. The probability for each symbol is the frequency of appearance divided by the message length. When we reduce the number of bits needed to represent the probable symbols (their code lengths) we can also reduce the average code length and thus the number of bits we need to send.

'C'	1/16	0.0625
'D'	3/16	0.1875
'I'	8/16	0.5
'S'	2/16	0.125
'V'	2/16	0.125

The entropy gives the lower limit for a statistical compression method's average codelength. Using the equation from the next section, we can calculate it as 1.953. This means that however cleverly you select a code to represent the symbols, in average you need at least 1.953 bits per symbol. In this case you can't do better than 32 bits, since there are a total of 16 symbols.

Next we create the Huffman tree. We first rank the symbols in decreasing probability order and then combine two lowest-probability symbols into a single composite symbol (C1, C2, ..). The probability of this new symbol is therefore the sum of the two original probabilities. The process is then repeated until a single composite symbol remains:

```
    Step 1          Step 2         Step 3          Step 4
    'I' 0.5         'I' 0.5        'I' 0.5         C3  0.5\C4
    'D' 0.1875      C1  0.1875     C2  0.3125\C3   'I' 0.5/
    'S' 0.125       'D' 0.1875\C2  C1  0.1875/
    'V' 0.125 \C1   'S' 0.125 /
    'C' 0.0625/
```

Note that the composite symbols are inserted as high as possible, to get the shortest maximum code length (compare C1 and 'D' at Step 2).

At each step two lowest-probability nodes are combined until we have only one symbol left. Without knowing it we have already created a Huffman tree. Start at the final symbol (C4 in this case), break up the composite symbol assigning 0 to the first symbol and 1 to the second one. The following tree just discards the probabilities as we don't need them anymore.

```
              C4
           0 / \ 1
            /  'I'
           C3
        0 / \ 1
         /   \
        C2   C1
     0 / \1 0/ \ 1
     'D''S' 'V''C'
```

Symbol	Code	Code Length
'C'	011	3
'D'	000	3
'I'	1	1
'S'	001	3
'V'	010	3

When we follow the tree from to top to the symbol we want to encode and remember each decision (which branch to follow), we get the code: {'C', 'D', 'I', 'S', 'V'} = {011, 000, 1, 001, 010}. For example when we see the symbol 'C' in the input, we output 011. If we see 'I' in the input, we output a single 1. The code for 'I' is very short because it occurs very often in the input.

Now we have the code lengths and can calculate the average code length: 0.0625*3+0.1875*3+0.5*1+0.125*3+0.125*3 = 2. We did not quite reach the lower limit that entropy gave us. Well, actually it is not so surprising because we know that Huffman code is optimal only if all the probabilities are negative powers of two.

Encoded, the message becomes:

```
    SIDVICIIISIDIDVI ==
    001 1 000 010 1 011 1 1 1 001 1 000 1 000 010 1
```

The spaces are only to make the reading easier. So, the compressed output takes 32 bits and we need at least 10 bits to transfer the Huffman tree by sending the code lengths (more on this later). The message originally took 48 bits, now it takes at least 42 bits.

Huffman coding is an example of a "variable length code" with a "defined word" input. Inputs of fixed size -- a single, three-bit letter above -- are replaced by a variable number of bits. At the other end of the scale are routines which break the _input_ up into variably sized chunks, and replace those chunks with an often fixed-length _output_. The most popular schemes of this type are Lempel-Ziv, or LZ, codes.

Of these, LZ77 is probably the most straightforward. It tries to replace recurring patterns in the data with a short code. The code tells the decompressor how many symbols to copy and from where in the output to copy them. To compress the data, LZ77 maintains a history buffer, which contains the data that has been processed, and tries to match the next part of the message to it. If there is no match, the next symbol is output as-is. Otherwise an (offset,length) -pair is output.

```
    Output           History Lookahead
                             SIDVICIIISIDIDVI
    S                      S IDVICIIISIDIDVI
    I                     SI DVICIIISIDIDVI
    D                    SID VICIIISIDIDVI
    V                   SIDV ICIIISIDIDVI
    I                  SIDVI CIIISIDIDVI
    C                 SIDVIC IIISIDIDVI
    I                SIDVICI IISIDIDVI
    I               SIDVICII ISIDIDVI
    I              SIDVICIII SIDIDVI
                   ---       ---    match length: 3
                   |----9---|       match offset: 9
    (9, 3)      SIDVICIIISID IDVI
                          -- --     match length: 2
                          |2|       match offset: 2
    (2, 2)    SIDVICIIISIDID VI
                 --          --     match length: 2
                 |----11----|       match offset: 11
    (11, 2) SIDVICIIISIDIDVI
```

At each stage the string in the lookahead buffer is searched from the history buffer. The longest match is used and the distance between the match and the current position is output, with the match length. The processed data is then moved to the history buffer. Note that the history buffer contains data that has already been output. In the decompression side it corresponds to the data that has already been decompressed. The message becomes:

```
    S I D V I C I I I (9,3) (2,2) (11,2)
```

The following describes what the decompressor does with this data.

```
    History                 Input
                            S
    S                       I
    SI                      D
    SID                     V
    SIDV                    I
    SIDVI                   C
    SIDVIC                  I
    SIDVICI                 I
    SIDVICII                I
    SIDVICIII               (9,3)   -> SID
    |----9---|
    SIDVICIIISID            (2,2)   -> ID
              |2|
    SIDVICIIISIDID          (11,2)  -> VI
       |----11----|
    SIDVICIIISIDIDVI
```

In the decompressor the history buffer contains the data that has already been decompressed. If we get a literal symbol code, it is added as-is. If we get an (offset,length) pair, the offset tells us from where to copy and the length tells us how many symbols to copy to the current output position. For example (9,3) tells us to go back 9 locations and copy 3 symbols to the current output position. The great thing is that we don't need to transfer or maintain any other data structure than the data itself.

Compare this to the BASIC interpreter, where all tokens have the high bit set and all normal characters don't (PETSCII codes 0-127). So when the LIST routine sees a normal character it just prints it as-is, but when it hits a special character (PETSCII >= 128) it looks up the corresponding keyword in a table. LZ77 is similar, but an LZ77 LIST would look up the keyword in the data already LISTed to the screen! LZ78 uses a separate table which is expanded as the data is processed.

The number of bits needed to encode the message (>52 bits) is somewhat bigger than the Huffman code used (42 bits). This is mainly because the message is too short for LZ77. It takes quite a long time to build up a good enough dictionary (the history buffer).

Introduction to Information Theory

Symbol Sources

Information theory traditionally deals with symbol sources that have certain properties. One important property is that they give out symbols that belong to a finite, predefined alphabet A. An alphabet can consist of for example all upper-case characters (A = {'A','B','C',..'Z',..}), all byte values (A = {0,1,..255}) or both binary digits (A = {0,1}).

As we are dealing with file compression, the symbol source is a file and the symbols (characters) are byte values from 0 to 255. A string or a phrase is a concatenation of symbols, for example 011101, "AAACB". Quite intuitive, right?

When reading symbols from a symbol source, there is some probability for each of the symbols to appear. For totally random sources each symbol is equally likely, but random sources are also incompressible, and we are not interested in them here. Equal probabilities or not, probabilities give us a means of defining the concept of symbol self-information, i.e. the amount of information a symbol carries.

Simply, the more probable an event is, the less bits of information it contains. If we denote the probability of a symbol A[i] occurring as p(A[i]), the expression -log2(p(A[i])) (base-2 logarithm) gives the amount of information in bits that the source symbol A[i] carries. You can calculate base-2 logarithms using base-10 or natural logarithms if you remember that log2(n) = log(n)/log(2).

A real-world example is a comparison between the statements:

1. it is raining
2. the moon of earth has exploded.

The first case happens every once in a while (assuming we are not living in a desert area). Its probability may change around the world, but may be something like 0.3 during bleak autumn days. You would not be very surprised to hear that it is raining outside. It is not so for the second case. The second case would be big news, as it has never before happened, as far as we know. Although it seems very unlikely we could decide a very small probability for it, like 1E-30. The equation gives the self-information for the first case as 1.74 bits, and 99.7 bits for the second case.

Message Entropy

So, the more probable a symbol is, the less information it carries. What about the whole message, i.e. the symbols read from the input stream?

What is the information contents a specific message carries? This brings us to another concept: the entropy of a source. The measure of entropy gives us the amount of information in a message and is calculated like this: H = sum{ -p(A[i])*log2(p(A[i])) }. For completeness we note that 0*log2(0) gives the result 0 although log2(0) is not defined in itself. In essence, we multiply the information a symbol carries by the probability of the symbol and then sum all multiplication results for all symbols together.

The entropy of a message is a convenient measure of information, because it sets the lower limit for the average codeword length for a block-variable code, for example Huffman code. You can not get better compression with a statistical compression method which only considers single-symbol probabilities. The average codeword length is calculated in an analogous way to the entropy. Average code length is L = sum{-l(i)*log2(p(A[i])) }, where l(i) is the codeword length for the ith symbol in the alphabet. The difference between L and H gives an indication about the efficiency of a code. Smaller difference means more efficient code.

It is no coincidence that the entropy and average code length are calculated using very similar equations. If the symbol probabilities are not equal, we can get a shorter overall message, i.e. shorter _average_ codeword length (i.e. compression), if we assign shorter codes for symbols that are more likely to occur. Note that entropy is only the lower limit for statistical compression systems. Other methods may perform better, although not for all files.

Codes

A code is any mapping from an input alphabet to an output alphabet. A code can be e.g. {a, b, c} = {0, 1, 00}, but this code is obviously not uniquely decodable. If the decoder gets a code message of two zeros, there is no way it can know whether the original message had two a's or a c.

A code is _instantaneous_ if each codeword (a code symbol as opposed to source symbol) in a message can be decoded as soon as it is received. The binary code {a, b} = {0, 01} is uniquely decodable, but it isn't instantaneous. You need to peek into the future to see if the next bit is 1. If it is, b is decoded, if not, a is decoded. The binary code {a, b, c} = {0, 10, 11} on the other hand is an instantaneous code.

A code is a _prefix code_ if and only if no codeword is a prefix of another codeword. A code is instantaneous if and only if it is a prefix code, so a prefix code is always a uniquely decodable instantaneous code. We only deal with prefix codes from now on. It can be proven that all uniquely decodable codes can be changed into prefix codes of equal code lengths.

'Classic' Code Classification

Compression algorithms can be crudely divided into four groups:

1. Block-to-block codes
2. Block-to-variable codes
3. Variable-to-block codes
4. Variable-to-variable codes

Block-to-block codes

These codes take a specific number of bits at a time from the input and emit a specific number of bits as a result. If all of the symbols in the input alphabet (in the case of bytes, all values from 0 to 255) are used, the output alphabet must be the same size as the input alphabet, i.e. uses the same number of bits. Otherwise it could not represent all arbitrary messages.

Obviously, this kind of code does not give any compression, but it allows a transformation to be performed on the data, which may make the data more easily compressible, or which separates the 'essential' information for lossy compression. For example the discrete cosine transform (DCT) belongs to this group. It doesn't really compress anything, as it takes in a matrix of values and produces a matrix of equal size as output, but the resulting values hold the information in a more compact form.

In lossless audio compression the transform could be something along the lines of delta encoding, i.e. the difference between successive samples (there is usually high correlation between successive samples in audio data), or something more advanced like Nth order prediction. Only the prediction error is transmitted. In lossy compression the prediction error may be transmitted in reduced precision. The reproduction in the decompression won't then be exact, but the number of bits needed to transmit the prediction error may be much smaller.

One block-to-block code relevant to Commodore 64, VIC 20 and their relatives is nybble packing that is performed by some C64 compression programs. As nybbles by definition only occupy 4 bits of a byte, we can fit two nybbles into each byte without throwing any data away, thus getting 50% compression from the original which used a whole byte for every nybble. Although this compression ratio may seem very good, in reality very little is gained globally. First, only very small parts of actual files contain nybble-width data. Secondly, better methods exist that also take advantage of the patterns in the data.

Block-to-variable codes

Block-to-variable codes use a variable number of output bits for each input symbol. All statistical data compression systems, such as symbol ranking, Huffman coding, Shannon-Fano coding, and arithmetic coding belong to this group (these are explained in more detail later). The idea is to assign shorter codes for symbols that occur often, and longer codes for symbols that occur rarely. This provides a reduction in the average code length, and thus compression.

There are three types of statistical codes: fixed, static, and adaptive. Static codes need two passes over the input message. During the first pass they gather statistics of the message so that they know the probabilities of the source symbols. During the second pass they perform the actual encoding. Adaptive codes do not need the first pass. They update the statistics while encoding the data. The same updating of statistics is done in the decoder so that they keep in sync, making the code uniquely decodable. Fixed codes are 'static' static codes. They use a preset statistical model, and the statistics of the actual message has no effect on the encoding. You just have to hope (or make certain) that the message statistics are close to the one the code assumes.

However, 0-order statistical compression (and entropy) don't take advantage of inter-symbol relations. They assume symbols are disconnected variables, but in reality there is considerable relation between successive symbols. If I would drop every third character from this text, you would probably be able to decipher it quite well. First order statistical compression uses the previous character to predict the next one. Second order compression uses two previous characters, and so on. The more characters are used to predict the next character the better estimate of the probability distribution for the next character. But more is not only better, there are also prices to pay.

The first drawback is the amount of memory needed to store the probability tables. The frequencies for each character encountered must be accounted for. And you need one table for each 'previous character' value. If we are using an adaptive code, the second drawback is the time needed to update the tables and then update the encoding accordingly. In the case of Huffman encoding the Huffman tree needs to be recreated. And the encoding and decoding itself certainly takes time also.

We can keep the memory usage and processing demands tolerable by using a 0-order static Huffman code. Still, the Huffman tree takes up precious memory and decoding Huffman code on a 1-MHz 8-bit processor is slow and does not offer very good compression either. Still, statistical compression can still offer savings as a part of a hybrid compression system.

```
    For example:
    'A'     1/2     0
    'B'     1/4     10
    'C'     1/8     110
    'D'     1/8     111

    "BACADBAABAADABCA"                             total: 32 bits
    10 0 110 0 111 10 0 0 10 0 0 111 0 10 110 0     total: 28 bits
```

This is an example of a simple statistical compression. The original symbols each take two bits to represent (4 possibilities), thus the whole string takes 32 bits. The variable-length code assigns the shortest code to the most probable symbol (A) and it takes 28 bits to represent the same string. The spaces between symbols are only there for clarity. The decoder still knows where each symbol ends because the code is a prefix code.

On the other hand, I am simplifying things a bit here, because I'm omitting one vital piece of information: the length of the message. The file system normally stores the information about the end of file by storing the length of the file. The decoder also needs this information. We have two basic methods: reserve one symbol to represent the end of file condition or send the length of the original file. Both have their virtues.

The best compressors available today take into account intersymbol probabilities. Dynamic Markov Coding (DMC) starts with a zero-order Markov model and gradually extends this initial model as compression progresses. Prediction by Partial Matching (PPM), although it really is a variable-to-block code, looks for a match of the text to be compressed in an order-n context and if there is no match drops back to an order n-1 context until it reaches order 0.

Variable-to-block codes

The previous compression methods handled a specific number of bits at a time. A group of bits were read from the input stream and some bits were written to the output. Variable-to-block codes behave just the opposite. They use a fixed-length output code to represent a variable-length part of the input. Variable-to-block codes are also called free-parse methods, because there is no pre-defined way to divide the input message into encodable parts (i.e. strings that will be replaced by shorter codes). Substitutional compressors belong to this group.

Substitutional compressors work by trying to replace strings in the input data with shorter codes. Lempel-Ziv methods (named after the inventors) contain two main groups: LZ77 and LZ78.

Lempel-Ziv 1977

In 1977 Ziv and Lempel proposed a lossless compression method which replaces phrases in the data stream by a reference to a previous occurrance of the phrase. As long as it takes fewer bits to represent the reference and the phrase length than the phrase itself, we get compression. Kind-of like the way BASIC substitutes tokens for keywords.

LZ77-type compressors use a history buffer, which contains a fixed amount of symbols output/seen so far. The compressor reads symbols from the input to a lookahead buffer and tries to find as long as possible match from the history buffer. The length of the string match and the location in the buffer (offset from the current position) is written to the output. If there is no suitable match, the next input symbol is sent as a literal symbol.

Of course there must be a way to identify literal bytes and compressed data in the output. There are lot of different ways to accomplish this, but a single bit to select between a literal and compressed data is the easiest.

The basic scheme is a variable-to-block code. A variable-length piece of the message is represented by a constant amount of bits: the match length and the match offset. Because the data in the history buffer is known to both the compressor and decompressor, it can be used in the compression. The decompressor simply copies part of the already decompressed data or a literal byte to the current output position.

Variants of LZ77 apply additional compression to the output of the compressor, which include a simple variable-length code (LZB), dynamic Huffman coding (LZH), and Shannon-Fano coding (ZIP 1.x)), all of which result in a certain degree of improvement over the basic scheme. This is because the output values from the first stage are not evenly distributed, i.e. their probabilities are not equal and statistical compression can do its part.

Lempel-Ziv 1978
+++++++++++++++

One large problem with the LZ77 method is that it does not use the coding space efficiently, i.e. there are length and offset values that never get used. If the history buffer contains multiple copies of a string, only the latest occurrance is needed, but they all take space in the offset value space. Each duplicate string wastes one offset value.

To get higher efficiency, we have to create a real dictionary. Strings are added to the codebook only once. There are no duplicates that waste bits just because they exist. Also, each entry in the codebook will have a specific length, thus only an index to the codebook is needed to specify a string (phrase). In LZ77 the length and offset values were handled more or less as disconnected variables although there is correlation. Because they are now handled as one entity, we can expect to do a little better in that regard also.

LZ78-type compressors use this kind of a dictionary. The next part of the message (the lookahead buffer contents) is searched from the dictionary and the maximum-length match is returned. The output code is an index to the dictionary. If there is no suitable entry in the dictionary, the next input symbol is sent as a literal symbol. The dictionary is updated after each symbol is encoded, so that it is possible to build an identical dictionary in the decompression code without sending additional data.

Essentially, strings that we have seen in the data are added to the dictionary. To be able to constantly adapt to the message statistics, the dictionary must be trimmed down by discarding the oldest entries. This also prevents the dictionary from becoming full, which would decrease the compression ratio. This is handled automatically in LZ77 by its use of a history buffer (a sliding window). For LZ78 it must be implemented separately. Because the decompression code updates its dictionary in sychronization with the compressor the code remains uniquely decodable.

Run-Length Encoding
+++++++++++++++++++

Run length encoding also belongs to this group. If there are consecutive equal valued symbols in the input, the compressor outputs how many of them there are, and their value. Again, we must be able to identify literal bytes and compressed data. One of the RLE compressors I have seen outputs two equal symbols to indentify a run of symbols. The next byte(s) then tell how many more of these to output. If the value is 0, there are only two consecutive equal symbols in the original stream. Depending on how many bits are used to represent the value, this is the only case when the output is expanded.

Run-length encoding has been used since day one in C64 compression programs because it is very fast and very simple. Part of this is because it deals with byte-aligned data and is essentially just copying bytes from one place to another. The drawback is that RLE can only compress identical bytes into a shorter representation. On the C64, only graphics and music data contain large runs of identical bytes. Program code rarely contains more than a couple of successive identical bytes. We need something better.

That "something better" seems to be LZ77, which has been used in C64 compression programs for a long time. LZ77 can take advantage of repeating code/graphic/music data fragments and thus achieves better compression. The drawback is that practical LZ77 implementations tend to became variable-to-variable codes (more on that later) and need to handle data bit by bit, which is quite a lot slower than handling bytes.

LZ78 is not practical for C64, because the decompressor needs to create and update the dictionary. A big enough dictionary would take too much memory and updating the dictionary would need its share of processor cycles.

Variable-to-variable codes

The compression algorithms in this category are mostly hybrids or concatenations of the previously described compressors. For example a variable-to-block code such as LZ77 followed by a statistical compressor like Huffman encoding falls into this category and is used in Zip, LHa, Gzip and many more. They use fixed, static, and adaptive statistical compression, depending on the program and the compression level selected.

Randomly concatenating algorithms rarely produces good results, so you have to know what you are doing and what kind of files you are compressing. Whenever a novice asks the usual question: 'What compression program should I use?', they get the appropriate response: 'What kind of data you are compressing?'

Borrowed from Tom Lane's article in comp.compression:
It's hardly ever worthwhile to take the compressed output of one compression method and shove it through another compression method. Especially not if the second method is a general-purpose compressor that doesn't have specific knowledge of the first compression step. Compression is effective in direct proportion to the extent that it eliminates obvious patterns in the data. So if the first compression step is any good, it will leave little traction for the second step. Combining multiple compression methods is only helpful when the methods are specifically chosen to be complementary.

A small sidetrack I want to take:
This also brings us conveniently to another truth in lossless compression. There isn't a single compressor which would be able to losslessly compress all possible files (you can see the comp.compression FAQ for information about the counting proof). It is our luck that we are not interested in compressing all files. We are only interested in compressing a very small subset of all files. The more accurately we can describe the files we would encounter, the better. This is called modelling, and it is what all compression programs do and must do to be successful.

Audio and graphics compression algorithm may assume a continuous signal, and a text compressor may assume that there are repeated strings in the data. If the data does not match the assumptions (the model), the algorithm usually expands the data instead of compressing it.

Representing Integers

Many compression algorithms use integer values for something or another. Pucrunch is no exception as it needs to represent RLE repeat counts and LZ77 string match lengths and offsets. Any algorithm that needs to represent integer values can benefit very much if we manage to reduce the number of bits needed to do that. This is why efficient coding of these integers is very important. What encoding method to select depends on the distribution and the range of the values.

Fixed, Linear

If the values are evenly distributed throughout the whole range, a direct binary representation is the optimal choice. The number of bits needed of course depends on the range. If the range is not a power of two, some tweaking can be done to the code to get nearer the theoretical optimum log2(_range_) bits per value.

Value	Binary	Adjusted 1&2		
0	000	00	000	H = 2.585
1	001	01	001	L = 2.666
2	010	100	010	(for flat distribution)
3	011	101	011	
4	100	110	10	
5	101	111	11	

The previous table shows two different versions of how the adjustment could be done for a code that has to represent 6 different values with the minimum average number of bits. As can be seen, they are still both prefix codes, i.e. it's possible to (easily) decode them.

If there is no definite upper limit to the integer value, direct binary code can't be used and one of the following codes must be selected.

Elias Gamma Code

The Elias gamma code assumes that smaller integer values are more probable. In fact it assumes (or benefits from) a proportionally decreasing distribution. Values that use n bits should be twice as probable as values that use n+1 bits.

In this code the number of zero-bits before the first one-bit (a unary code) defines how many more bits to get. The code may be considered a special fixed Huffman tree. You can generate a Huffman tree from the assumed value distribution and you'll get a very similar code. The code is also directly decodable without any tables or difficult operations, because once the first one-bit is found, the length of the code word is instantly known. The bits following the zero bits (if any) are directly the encoded value.

Gamma Code	Integer	Bits
1	1	1
01x	2-3	3
001xx	4-7	5
0001xxx	8-15	7
00001xxxx	16-31	9
000001xxxxx	32-63	11
0000001xxxxxx	64-127	13
...		

Elias Delta Code

The Elias Delta Code is an extension of the gamma code. This code assumes a little more 'traditional' value distribution. The first part of the code is a gamma code, which tells how many more bits to get (one less than the gamma code value).

Delta Code	Integer	Bits
1	1	1
010x	2-3	4
011xx	4-7	5
00100xxx	8-15	8
00101xxxx	16-31	9
00110xxxxx	32-63	10
00111xxxxxx	64-127	11
...		

The delta code is better than gamma code for big values, as it is asymptotically optimal (the expected codeword length approaches constant times entropy when entropy approaches infinity), which the gamma code is not. What this means is that the extra bits needed to indicate where the code ends become smaller and smaller proportion of the total bits as we encode bigger and bigger numbers. The gamma code is better for greatly skewed value distributions (a lot of small values).

Fibonacci Code

The fibonacci code is another variable length code where smaller integers get shorter codes. The code ends with two one-bits, and the value is the sum of the corresponding Fibonacci values for the bits that are set (except the last one-bit, which ends the code).

1	2	3	5	8	13	21	34	55	89	
1	(1)									= 1
0	1	(1)								= 2
0	0	1	(1)							= 3
1	0	1	(1)							= 4
0	0	0	1	(1)						= 5
1	0	0	1	(1)						= 6
0	1	0	1	(1)						= 7
0	0	0	0	1	(1)					= 8
:	:	:	:	:	:					:
1	0	1	0	1	(1)					= 12

```
    0  0  0  0  0  1 (1)          = 13
    :  :  :  :  :  :  :              :
    0  1  0  1  0  1 (1)          = 20
    0  0  0  0  0  0  1 (1)       = 21
    :  :  :  :  :  :  :  :           :
    1  0  0  1  0  0  1 (1)       = 27
```

Note that because the code does not have two successive one-bits until the end mark, the code density may seem quite poor compared to the other codes, and it is, if most of the values are small (1-3). On the other hand, it also makes the code very robust by localizing and containing possible errors. Although, if the Fibonacci code is used as a part of a larger system, this robustness may not help much, because we lose the synchronization in the upper level anyway. Most adaptive methods can't recover from any errors, whether they are detected or not. Even in LZ77 the errors can be inherited infinitely far into the future.

Comparison between delta, gamma and Fibonacci code lengths

	Gamma	Delta	Fibonacci
1	1	1	2.0
2-3	3	4	3.5
4-7	5	5	4.8
8-15	7	8	6.4
16-31	9	9	7.9
32-63	11	10	9.2
64-127	13	11	10.6

The comparison shows that if even half of the values are in the range 1..7 (and other values relatively near this range), the Elias gamma code wins by a handsome margin.

Golomb and Rice Codes

Golomb (and Rice) codes are prefix codes that are suboptimal (compared to Huffman), but very easy to implement. Golomb codes are distinguished from each other by a single parameter m. This makes it very easy to adjust the code dynamically to adapt to changes in the values to encode.

Golomb	m=1	m=2	m=3	m=4	m=5	m=6
Rice	k=0	k=1		k=2		
n = 0	0	00	00	000	000	000
1	10	01	010	001	001	001
2	110	100	011	010	010	0100
3	1110	101	100	011	0110	0101
4	11110	1100	1010	1000	0111	0110
5	111110	1101	1011	1001	1000	0111
6	1111110	11100	1100	1010	1001	1000
7	:	11101	11010	1011	1010	1001
8	:	111100	11011	11000	10110	10100

To encode an integer n (starting from 0 this time, not from 1 as for Elias codes and Fibonacci code) using the Golomb code with parameter m, we first compute floor(n/m) and output this using a unary code. Then we compute the remainder n mod m and output that value using a binary code which is adjusted so that we sometimes use floor(log2(m)) bits and sometimes ceil(log2(m)) bits.

Rice coding is the same as Golomb coding except that only a subset of parameters can be used, namely the powers of 2. In other words, a Rice code with the parameter k is equal to Golomb code with parameter m = 2^k. Because of this the Rice codes are much more efficient to implement on a computer. Division becomes a shift operation and modulo becomes a bit mask operation.

Hybrid/Mixed Codes

Sometimes it may be advantageous to use a code that combines two or more of these codes. In a way the Elias codes are already hybrid codes. The gamma code has a fixed huffman tree (a unary code) and a binary code part, the delta code has a gamma code and a binary code part. The same applies to Golomb and Rice codes because they consist of a unary code part and a linear code (adjusted) part.

So now we have several alternatives to choose from. We simply have to do a little real-life research to determine how the values we want to encode are distributed so that we can select the optimum code to represent them.

Of course we still have to keep in mind that we intend to decode the thing with a 1-MHz 8-bit processor. As always, compromises loom on the horizon. Pucrunch uses Elias Gamma Code, because it is the best alternative for that task and is very close to static Huffman code. The best part is that the Gamma Code is much simpler to decode and doesn't need additional memory.

Closer Look

Because the decompression routines are usually much easier to understand than the corresponding compression routines, I will primarily describe only them here. This also ensures that there really _is_ a decompressor for a compression algorithm. Many are those people who have developed a great new compression algorithm that outperforms all existing versions, only to later discover that their algorithm doesn't save enough information to be able to recover the original file from the compressed data.

Also, the added bonus is that once we have a decompressor, we can improve the compressor without changing the file format. At least until we have some statistics to develop a better system. Many lossy video and audio compression systems only document and standardize the decompressor and the file or stream format, making it possible to improve the encoding part of the process when faster and better hardware (or algorithms) become available.

RLE

```
     void DecompressRLE() {
         int oldChar = -1;
         int newChar;

         while(1) {
             newChar = GetByte();
             if(newChar == EOF)
                 return;
             PutByte(newChar);
             if(newChar == oldChar) {
                 int len = GetLength();

                 while(len > 0) {
                     PutByte(newChar);
                     len = len - 1;
                 }
             }
             oldChar = newChar;
         }
     }
```

This RLE algorithm uses two successive equal characters to mark a run of bytes. I have in purpose left open the question of how the length is encoded (1 or 2 bytes or variable-length code). The decompressor also allows chaining/extension of RLE runs, for example 'a', 'a', 255, 'a', 255 would output 513 'a'-characters.
In this case the compression algorithm is almost as simple.

```
     void CompressRLE() {
         int oldChar = -1;
         int newChar;

         while(1) {
             newChar = GetByte();
             if(newChar==oldChar) {
                 int length = 0;

                 if(newChar == EOF)
                     return;
                 PutByte(newChar); /* RLE indicator */

                 /* Get all equal characters */
                 while((newChar = GetByte()) == oldChar) {
                     length++;
```

```
                }
                PutLength(length);
            }
            if(newChar == EOF)
                return;
            PutByte(newChar);
            oldChar = newChar;
        }
    }
```

If there are two equal bytes, the compression algorithm reads more bytes until it gets a different byte. If there was only two equal bytes, the length value will be zero and the compression algorithm expands the data. A C64-related example would be the compression of the BASIC ROM with this RLE algorithm. Or actually expansion, as the new file size is 8200 bytes instead of the original 8192 bytes. Those equal byte runs that the algorithm needs just aren't there. For comparison, pucrunch manages to compress the BASIC ROM into 7288 bytes, the decompression code included. Even Huffman coding manages to compress it into 7684 bytes.

```
    "BAAAAAADBBABBBBBAAADABCD"              total: 24*8=192 bits
    "BAA",4,"DBB",0,"ABB",3,"AA",1,"DABCD"  total: 16*8+4*8=160 bits
```

This is an example of how the presented RLE encoder would work on a string. The total length calculations assume that we are handling 8-bit data, although only values from 'A' to 'D' are present in the string. After seeing two equal characters the decoder gets a repeat count and then adds that many more of them. Notice that the repeat count is zero if there are only two equal characters.

Huffman Code

```
    int GetHuffman() {
        int index = 0;

        while(1) {
            if(GetBit() == 1) {
                index = LeftNode(index);
            } else {
                index = RightNode(index);
            }
            if(LeafNode(index)) {
                return LeafValue(index);
            }
        }
    }
```

My pseudo code of the Huffman decode function is a very simplified one, so I should probably describe how the Huffman code and the corresponding binary tree is constructed first.

First we need the statistics for all the symbols occurring in the message, i.e. the file we are compressing. Then we rank them in decreasing probability order. Then we combine the smallest two probabilities and assign 0 and 1 to the binary tree branches, i.e. the original symbols. We do this until there is only one composite symbol left.

Depending on where we insert the composite symbols we get different Huffman trees. The average code length is equal in both cases (and so is the compression ratio), but the length of the longest code changes. The implementation of the decoder is usually more efficient if we keep the longest code as short as possible. This is achieved by inserting the composite symbols (new nodes) before all symbols/nodes that have equal probability.

```
    "BAAAAAADBBABBBBBAAADABCD"
    A (11)  B (9)  D (3)  C (1)

    Step 1              Step 2              Step 3
    'A' 0.458           'A' 0.458           C2  0.542 0\ C3
    'B' 0.375           'B' 0.375 0\ C2     'A' 0.458 1/
    'D' 0.125 0\ C1     C1  0.167 1/
    'C' 0.042 1/

             C3
          0 /  \ 1
           /   'A'
```

```
         C2
     0 /  \ 1
     'B'   \
            C1
        0 /  \ 1
        'D'  'C'
```

So, in each step we combine two lowest-probability nodes or leaves into a new node. When we are done, we have a Huffman tree containing all the original symbols. The Huffman codes for the symbols can now be gotten by starting at the root of the tree and collecting the 0/1-bits on the way to the desired leaf (symbol). We get:

```
    'A' = 1    'B' = 00    'C' = 011    'D' = 010
```

These codes (or the binary tree) are used when encoding the file, but the decoder also needs this information. Sending the binary tree or the codes would take a lot of bytes, thus taking away all or most of the compression. The amount of data needed to transfer the tree can be greatly reduced by sending just the symbols and their code lengths. If the tree is traversed in a canonical (predefined) order, this is all that is needed to recreate the tree and the Huffman codes. By doing a 0-branch-first traverse we get:

Symbol	Code	Code Length
'B'	00	2
'D'	010	3
'C'	011	3
'A'	1	1

So we can just send 'B', 2, 'D', 3, 'C', 3, 'A', 1 and the decoder has enough information (when it also knows how we went through the tree) to recreate the Huffman codes and the tree. Actually you can even drop the symbol values if you handle things a bit differently (see the Deflate specification in RFC1951), but my arrangement makes the algorithm much simpler and doesn't need to transfer data for symbols that are not present in the message.

Basically we start with a code value of all zeros and the appropriate length for the first symbol. For other symbols we first add the code value with 1 and then shift the value left or right to get it to be the right size. In the example we first assign 00 to 'B', then add one to get 01, shift left to get a 3-bit codeword for 'D' making it 010 like it should. For 'C' add 1, you get 011, no shift because the codewords is the right size already. And for 'A' add one and get 100, shift 2 places to right and get 1.

The Deflate algorithm in essence attaches a counting sort algorithm to this algorithm, feeding in the symbols in increasing code length order. Oh, don't worry if you don't understand what the counting sort has to do with this. I just wanted to give you some idea about it if you some day read the deflate specification or the gzip source code.

Actually, the decoder doesn't necessarily need to know the Huffman codes at all, as long as it has created the proper internal representation of the Huffman tree. I developed a special table format which I used in the C64 Huffman decode function and may present it in a separate article someday. The decoding works by just going through the tree by following the instructions given by the input bits as shown in the example Huffman decode code. Each bit in the input makes us go to either the 0-branch or the 1-branch. If the branch is a leaf node, we have decoded a symbol and just output it, return to the root node and repeat the procedure.

A technique related to Huffman coding is Shannon-Fano coding. It works by first dividing the symbols into two equal-probability groups (or as close to as possible). These groups are then further divided until there is only one symbol in each group left. The algorithm used to create the Huffman codes is bottom-up, while the Shannon-Fano codes are created top-down. Huffman encoding always generates optimal codes (in the entropy sense), Shannon-Fano sometimes uses a few more bits.

There are also ways of modifying the statistical compression methods so that we get nearer to the entropy. In the case of 'A' having the probability 0.75 and 'B' 0.25 we can decide to group several symbols together, producing a variable-to-variable code.

"AA"	0.5625	0

"B"	0.25	10
"AB"	0.1875	11

If we separately transmit the length of the file, we get the above probabilities. If a file has only one 'A', it can be encoded as length=1 and either "AA" or "AB". The entropy of the source is H = 0.8113, and the average code length (per source symbol) is approximately L = 0.8518, which is much better than L = 1.0, which we would get if we used a code {'A','B'} = {0,1}. Unfortunately this method also expands the number of symbols we have to handle, because each possible source symbol combination is handled as a separate symbol.

Arithmetic Coding

Huffman and Shannon-Fano codes are only optimal if the probabilities of the symbols are negative powers of two. This is because all prefix codes work in the bit level. Decisions between tree branches always take one bit, whether the probabilities for the branches are 0.5/0.5 or 0.9/0.1. In the latter case it would theoretically take only 0.15 bits (-log2(0.9)) to select the first branch and 3.32 bits (-log2(0.1)) to select the second branch, making the average code length 0.467 bits (0.9*0.15 + 0.1*3.32). The Huffman code still needs one bit for each decision.

Arithmetic coding does not have this restriction. It works by representing the file by an interval of real numbers between 0 and 1. When the file size increases, the interval needed to represent it becomes smaller, and the number of bits needed to specify that interval increases. Successive symbols in the message reduce this interval in accordance with the probability of that symbol. The more likely symbols reduce the range by less, and thus add fewer bits to the message.

```
     1                                                  Codewords
    +-----------+-----------+-----------+
    |           |8/9 YY     |  Detail   |<- 31/32    .11111
    |           +-----------+-----------+<- 15/16    .1111
    |    Y      |           | too small |<- 14/16    .1110
    |2/3        |    YX     | for text  |<- 6/8      .110
    +-----------+-----------+-----------+
    |           |           |16/27 XYY  |<- 10/16    .1010
    |           |           +-----------+
    |           |    XY     |           |
    |           |           |   XYX     |<- 4/8      .100
    |           |4/9        |           |
    |           +-----------+-----------+
    |           |           |           |
    |    X      |           |   XXY     |<- 3/8      .011
    |           |           |8/27       |
    |           |           +-----------+
    |           |    XX     |           |
    |           |           |           |<- 1/4      .01
    |           |           |   XXX     |
    |           |           |           |
    |0          |           |           |
    +-----------+-----------+-----------+
```

As an example of arithmetic coding, lets consider the example of two symbols X and Y, of probabilities 2/3 and 1/3. To encode a message, we examine the first symbol: If it is a X, we choose the lower partition; if it is a Y, we choose the upper partition. Continuing in this manner for three symbols, we get the codewords shown to the right of the diagram above. They can be found by simply taking an appropriate location in the interval for that particular set of symbols and turning it into a binary fraction. In practice, it is also necessary to add a special end-of-data symbol, which is not represented in this simple example.

This explanation may not be enough to help you understand arithmetic coding. There are a lot of good articles about arithmetic compression in the net, for example by Mark Nelson.

Arithmetic coding is not practical for C64 for many reasons. The biggest reason being speed, especially for adaptive arithmetic coding. The close second reason is of course memory.

Symbol Ranking

Symbol ranking is comparable to Huffman coding with a fixed Huffman tree. The compression ratio is not very impressive (reaches Huffman code only is some cases), but the decoding algorithm is very simple, does not need as much memory as Huffman and is also faster.

```
     int GetByte() {
         int index = GetUnaryCode();

         return mappingTable[index];
     }
```

The main idea is to have a table containing the symbols in descending probability order (rank order). The message is then represented by the table indices. The index values are in turn represented by a variable-length integer representation (these are studied in the next article). Because more probable symbols (smaller indices) take less bits than less probable symbols, in average we save bits. Note that we have to send the rank order, i.e. the symbol table too.

```
    "BAAAAAADBBABBBBBAAADABCD"              total: 24*8=192 bits
    Rank Order: A (11)  B (9)  D (3)  C (1)         4*8=32 bits
    Unary Code: 0       10     110    1110
    "100000001101010010101010100001100101110110"   42 bits
                                              total: 74 bits
```

The statistics rank the symbols in the order ABDC (most probable first), which takes approximately 32 bits to transmit (we assume that any 8-bit value is possible). The indices are represented as a code {0, 1, 2, 3} = {0, 10, 110, 1110}. This is a simple unary code where the number of 1-bits before the first 0-bit directly give the integer value. The first 0-bit also ends a symbol. When this code and the rank order table are combined in the decoder, we get the reverse code {0, 10, 110, 1110} = {A, B, D, C}. Note that in this case the code is very similar to the Huffman code we created in a previous example.

LZ78

LZ78-based schemes work by entering phrases into a dictionary and then, when a repeat occurrence of that particular phrase is found, outputting the dictionary index instead of the phrase. For example, LZW (Lempel-Ziv-Welch) uses a dictionary with 4096 entries. In the beginning the entries 0-255 refer to individual bytes, and the rest 256-4095 refer to longer strings. Each time a new code is generated it means a new string has been selected from the input stream. New strings that are added to the dictionary are created by appending the current character K to the end of an existing string w. The algorithm for LZW compression is as follows:

```
    set w = NIL
    loop
        read a character K
        if wK exists in the dictionary
            w = wK
        else
            output the code for w
            add wK to the string table
            w = K
    endloop
```

```
    Input string: /WED/WE/WEE/WEB

    Input   Output  New code and string
    /W      /       256 = /W
    E       W       257 = WE
    D       E       258 = ED
    /       D       259 = D/
    WE      256     260 = /WE
    /       E       261 = E/
    WEE     260     262 = /WEE
    /W      261     263 = E/W
    EB      257     264 = WEB
            B
```

A sample run of LZW over a (highly redundant) input string can be seen in the diagram above. The strings are built up character-by-character starting with a code value of 256. LZW decompression takes the stream of codes and uses it to exactly recreate the original input data. Just like the compression algorithm, the decompressor adds a new string to the dictionary each time it reads in a new code. All it needs to do in addition is to translate each incoming code into a string and send it to the output. A sample run of the LZW decompressor is shown in below.

```
    Input code: /WEDEB

    Input   Output  New code and string
    /       /
    W       W       256 = /W
    E       E       257 = WE
    D       D       258 = ED
    256     /W      259 = D/
    E       E       260 = /WE
    260     /WE     261 = E/
    261     E/      262 = /WEE
    257     WE      263 = E/W
    B       B       264 = WEB
```

The most remarkable feature of this type of compression is that the entire dictionary has been transmitted to the decoder without actually explicitly transmitting the dictionary. The decoder builds the dictionary as part of the decoding process.

See also the article "LZW Compression" by Bill Lucier in C=Hacking issue 6 and "LZW Data Compression" by Mark Nelson mentioned in the references section.

Conclusions

That's more than enough for one article. What did we get out of it ? Statistical compression works with uneven symbol probabilities to reduce the average code length. Substitutional compressors replace strings with shorter representations. All popular compression algorithms use LZ77 or LZ78 followed by some sort of statistical compression. And you can't just mix and match different algorithms and expect good results.

There are no shortcuts in understanding data compression. Some things you only understand when trying out them yourself. However, I hope that this article has given you at least a vague grasp of how different compression methods really work.

I would like to send special thanks to Stephen Judd for his comments. Without him this article would've been much more unreadable than it is now. On the other hand, that's what the magazine editor is for :-)

The second part of the story is a detailed talk about pucrunch. I also go through the corresponding C64 decompression code in detail. If you are impatient and can't wait for the next issue, you can take a peek into http://www.cs.tut.fi/%7Ealbert/Dev/pucrunch/ for a preview.

References

* The comp.compression FAQ
  http://www.cis.ohio-state.edu/hypertext/faq/usenet/compression-faq/top.html
* A Data Compression Review
  http://www.ics.uci.edu/%7Edan/pubs/DataCompression.html
* Data Compression Class
  http://www.cs.unt.edu/home/srt/5330/
* Mark Nelson's Homepage
  http://web2.airmail.net/markn/
    + LZW Data Compression
      http://web2.airmail.net/markn/articles/lzw/lzw.htm
    + Arithmetic Coding Article
      http://web2.airmail.net/markn/articles/arith/part1.htm
    + Data Compression with the Burrows-Wheeler Transform
      http://web2.airmail.net/markn/articles/bwt/bwt.htm
* Charles Bloom's Page
  http://wwwvms.utexas.edu/%7Ecbloom/
* The Redundancy of the Ziv-Lempel Algorithm for Memoryless Sources
  http://ei0.ei.ele.tue.nl/%7Etjalling/zivlem/zivlem.html
* Ross Williams' Compression Pages
  http://www.ross.net/home/

* DEFLATE Compressed Data Format Specification version 1.3
  http://www.funet.fi/pub/doc/rfc/rfc1951.txt
* Markus F.X.J. Oberhumer's Compression Links
  http://wildsau.idv.uni-linz.ac.at/mfx/compress.html
* The Lossless Compression (Squeeze) Page
  http://www.cs.sfu.ca/CC/365/li/squeeze/

```
........
....
..
.                                       C=H

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

3D Graphics for the Masses: lib3d and Cool World

by
Stephen L. Judd
sjudd@nwu.edu

Well folks, it's time once again for some 3D graphics. This will, I think, be more or less the final word on the subject, at least from me! I'm pooped, and I think these routines pretty much push these algorithms as far as they're going to go. And there are so many other interesting things to move on to!

These routines, not to mention this article, are the codification of all those years of algorithms and derivations and everything else. Those past efforts created working prototypes; this one is the production model. Nate Dannenberg suggested that this be done, and who am I to disobey? Besides, after a break of a few years it's going to be fun to rederive all the equations and algorithms from scratch, and do a "for-real" implementation, improving the good ideas and fixing the bad.

Right? Right.

So that's what I did, and that's what this article does. The first section of the article summarizes the basics of 3D graphics: projections and rotations. Since in my experience many people don't remember their high school math, I've also covered the basic mathematical tools needed, like trigonometry and linear algebra, and other related issues. The next section covers constructing a 3D world: representing objects, navigating through the world, things like that. The third section covers the main library routines and their implementation, including lots of code disassembly. The fourth section covers Cool World, a program which is a demonstration of the 3D library. The final section just summarizes the 3d library routines, calling conventions, parameters used, things like that. Since this article is enormous, I have left out the complete source code; the truly motivated can visit

http://stratus.esam.nwu.edu/~judd/fridge/

to check out the full source for both Cool World and the 3d library. The binaries are also present there.

By this time you may have asked, "What exactly _is_ this 3d library?" Glad you asked. It is a set of routines that are important for doing 3D graphics: rotations, projections, drawing polygons, that sort of thing. The routines are very flexible, compact, and extremely fast. They can draw to any VIC bank, and still leave enough room for eight sprite definitions. What the routines are not is a "3D world construction kit". A program has to be written which utilizes the routines.

Which leads to Cool World. CW is, simply, a program which demonstrates using the 3D library. It is very large. It has some seventeen objects in it -- the initial tetrahedron is the 'center'; straight ahead of it is squaresville; to the left is a bunch of stars; straight down is the Cobra Mk III; straight back is a line of tetrahedrons. It also has some other fun stuff like the starfield and a tune. It doesn't have Kim Basinger, unfortunately.

Oh yeah: the _whole point_ of the 3d library is for you to use it in your own programs! You can sell those programs if you like, too. Doesn't bother me; in fact I would be quite flattered to see it used in some cool and popular program. Why re-invent the wheel when the routines are already written? Use the 3d library. Live the 3d library. Do something interesting with it. If nobody uses it, it might as well have never been written, right? Right. You know you want to use it. So use it!

But before you use it, you better understand it. So let's start at the very beginning (a very good place to start).

Section 1: 3D Basics and review

This is just going to be a quick summary. For more detail you ought to go back to some of the earlier articles.

First we need to decide on a coordinate system. A convenient system for dealing with the computer is to have the x-axis point to the right and the y-axis to point down the screen -- the usual coordinate system with x=0 y=0 being in the upper-left corner of the screen. The z-axis can then point into the monitor (i.e. z=20 is behind the monitor screen somewhere, and z=-20 is somewhere behind where you're sitting); this keeps the coordinate system right-handed.

The next thing is to figure out an equation for projecting from three dimensions into two dimensions. In other words: how to paint a picture.
One of the great breakthroughs for art at the beginning of the Renaissance was the understanding of perspecitve. The first thing to notice is that far-away objects look smaller. The next thing to observe is that straight lines seem to meet in the middle -- like looking down a long road or sidewalk or railroad tracks. That's perspective: far-off objects get smaller _towards the center of vision_.
Well that's an easy equation: just take the coordinate, and divide by the distance away from us. If we're looking down the z-axis,

```
        x' = x/z        y' = y/z
```

gives us a pair of projected points. Objects which are far away have a large value for z, which means x' and y' will be proportionally smaller. For really large z, they go to zero -- they go to the center of our vision.
Now, how can we make objects appear larger? One way is to stand closer to them -- that changes the perspective. Another way is to magnify them, with a lens. This is how a telescope works, and also how the old Mark-I eyeball works as well. Whereas perspective makes far-off points behave differently than near ones, magnification just expands all points equally. And that's exactly how to put it in the equation:

```
        x' = d * x/z    y' = d * y/z
```

where d is some constant magnification factor (a number, like 12). And the above are the projection equations.
A very important skill is the ability to 'read' an equation. The above equation takes a point, divides by the distance away from us, and then multiplies by the magnification constant. It says that far-off objects will be small, and that all objects are magnified equally. It also implies that the z-coordinates better all be positive or all negative. If an object has some positive z-coordinates and some negative, that means that some of its points are in front of us and some are behind us. In other words, that we are standing inside the object!
A proper derivation is as follows: consider a pinhole camera. A light ray bounces off some object at a point (x,y,z) and passes through the pinhole, located at the origin (0,0,0). We then place a photographic plate parallel to the x-y plane, at z'=d, and figure out where the light ray hits the plate. The equation of the light ray is the line

```
        (x', y', z') = t*(x,y,z)
```

and it hits the plate when z'=d, which happens when

```
        tz = d
```

so when t = d/z. Thus the coordinates on the plate (the film) are

```
        x' = d/z * x    y' = d/z * y
```

which gives the earlier equations. Moving the film up and down will magnify the image, corresponding to changing the value of d. If d is negative, the image will be inverted. Positive d corresponds to having the film between the pinhole lens and the object, but mathematically it gives the same answer without inverting the object (since x' and y' won't have a minus sign in front of them).

Representing 3D objects

The important thing to recognize from the above is that straight lines in 3D will still be straight lines in 2D -- the projection doesn't turn straight lines into curves or anything like that.

So, to project a straight line from 3D into 2D all that is needed are the _endpoints_ of the line. Project those two endpoints from 3D into 2D, and then all that is needed is to draw a line between the two _projected_ points.

Most of the objects we will be dealing with will be made up of a series of flat plates, because these are easier to do calculations with (ever noticed the main difference between the stealth fighter and the stealth bomber?). In other words: polygons.

These polygons are just flat objects with several straight sides. All we need are the vertices of each polygon (the endpoints of each of the lines); from those we can reconstruct all the in-between points.

A very simple example is a cube. It has six faces. The eight corners of a cube might be located at (+/-1, +/-1, +/-1). Project those eight points and connect the projected points in the right way, and Viola! Violin! 3D graphics.

The next thing we will want to do is rotate an object. For this we need a little trig, and it's very helpful to know a little about linear algebra and vectors. What's that? You don't remember this stuff? For shame! But we can fix that up in no time.

Trig review in two paragraphs.

Take a look at a triangle sometime. No matter how large or small you draw a particular triangle, it still looks the same because all of the sides and angles are in proportion to one another. Now fix one of the angles at 90 degrees, to form a right triangle. The minute you choose one of the other angles, the opposite angle is determined (since the angles have to add up to 180 degrees). And when you know all the angles you know the _ratios_ of the sides to each other -- you don't know the actual length of the sides, but you know how the triangle looks and hence its proportions.

And that is pretty much all of trig. Draw a right triangle, and pick an angle (call that angle theta). The hypoteneuse is the longest side of a triangle, and hence the side opposite the 90 degree angle. The 'adjacent' side is the side which touches the angle theta; the 'opposite' side is the side opposite theta (imagine that!). We _define_ sine, cosine, and tangent as

```
        cos(theta) = adjacent/hypoteneuse
        sin(theta) = opposite/hypoteneuse
        tan(theta) = sin(theta)/cos(theta) = opposite/adjacent
```

and that's all of trig, right there. Everything else comes from those three definitions. If you can't remember them, just remember the famous Indian Chief SOHCAHTOA, who fought so bravely for the cause of his people and mathematicians everywhere (SOH: sin-opposite-hyp; CAH: cos-adjacent-hyp; TOA: tan-opposite-adjacent).

Polar coordinates

Now we've got some tools to do some useful calculations with, once we remember the Pythagorean theorem,

```
        a^2 + b^2 = c^2,
```

where c = the hypoteneuse and a,b are the sides of a right triangle. Before doing rotations it is helpful to understand polar coordinates.

Draw a set of normal coordinate axis on a piece of paper, and draw a point somewhere (at 3,2 say). Label this point x,y and draw a line from the origin (0,0) to that point, and call its length "r". Label the angle between that line and the positive x-axis as "theta", and draw a line straight down from x,y to the x-axis. Whaddaya know? It's a right triangle. The length of one side is x, the length of the other side is y, and the length of the hypoteneuse is r. And the trig definitions tell us that

```
        cos(theta) = x/r
        sin(theta) = y/r
        tan(theta) = y/x.
```

So now we can define sin, cos, and tan for all angles theta just by drawing a little picture.

Two coordinates, x and y, are used to locate the point. Another two coordinates in the picture are "r" and "theta", and they are just as good for locating where a point is. In fact, the above equations give the x and y coordinates of a point (r,theta):

```
        x = r * cos(theta)      y = r * sin(theta)
```

They also give us the r and theta coordinates of a point (x,y):

```
        tan(theta) = y/x
        r^2 = x^2 + y^2
```

You may also have noticed that the trig formula

```
        cos(theta)^2 + sin(theta)^2 = 1
```

follows from the above trig definitions, since x^2 + y^2 = r^2.
In the Cartesian (or rectangular) system there are two coordinates, x and y. The equation x=constant is a vertical line and the equation y=constant is a horizontal line, so this is a system of straight lines which cross each other at right angles. In polar coordinates the equation theta=constant is a radial line, and r=constant gives a circle (constant radius, you know). So polar coordinates consists of circles which intersect radial lines (note that they also intersect at right angles). And the equations above tell how to convert between the two systems.
There are many other coordinate systems, btw. There are elliptical coordinates and parabolic coordinates and other strange systems. Why bother? For one thing, if you're solving an equation on a round plate it's awfully convenient to use polar coordinates. For another, we can now do some rotations.

(And for another, you can draw cool graphs like r=cos(3*theta)).

Rotations

Start with a point (r,theta) and rotate it to a new point (r,theta+s) i.e. rotate by an amount s. The original coordinate was

```
        x = r*cos(theta)        y = r*sin(theta)
```

and the new coordinate is

```
        x' = r*cos(theta+s)     y = r*sin(theta+s)
```

so by using the trig identities for cos(a+b) and sin(a+b) we arrive at the simple expressions

```
        x' = x*cos(s) - y*sin(s)
        y' = x*sin(s) + y*cos(s)
```

for the rotated coordinates x' and y' in terms of the original coordinates x and y. The above is a two-dimensional rotation. Three-dimensional rotation is done with a series of two-dimensional rotations. The above rotates x and y, but keeps z fixed (I don't see a z anywhere in there, at least). In other words, it rotates about the z-axis. A rotation about the x-axis looks like

```
        x' = x*cos(s) - z*sin(s)
        z' = x*sin(s) + z*cos(s)
```

and a rotation about the y-axis looks like

```
        x' = x*cos(s)  + z*sin(s)
        z' = -x*sin(s) + z*cos(s)
```

(the opposite minus sign just comes from the orientation of the y-axis).

Rotations in three dimensions do NOT commute. Just by trying with a book it is very easy to see that a rotation of 45 degrees about the x-axis and then 45-degrees about the y-axis gives a very different result from first rotating about the y-axis and then the x-axis.

To really make rotations powerful we need a little linear algebra. But first...

Radians ("God's units")

Just for completeness... what is an angle, really? It's not a length, it's not a direction... what is it? And why are there 360 "angles" in a circle?

The answer to the second is "because the ancient Babylonians used base 60 for all their calculations." In other words, degrees are totally arbitrary. There could just as easily be 1000 degrees in a circle. So, what is an angle?

Think about a circle. It has two lengths associated with it: the length "across" (the diameter D) and the length "around" (the circumference C). And, like triangles, no matter how large or small the circle is drawn, those lengths stay in proportion. That is,

        C/D = constant

That constant is, of course, pi = 3.1415926... This then gives the equation for the circumference of a circle as C = pi*D = 2*pi*r where r=radius. But what is an angle?

Draw two radii in the circle. We _define_ the angle between those two radii as the length of the subtended circumference divided by the radius:

        angle = length / radius.

So again, no matter what the _size_ of the circle is the _angle_ stays the same. The length is just a fraction of the circumference, which is 2*pi*radius; this means that the angle is just a fraction of 2*pi.

These are radians. There are 2*pi of them in a circle. A quarter of a circle (90 degrees) is just 1/4 (2*pi) = pi/2 radians. An eighth is pi/4 radians. And so on. These are of course very natural units for an angle. The above definition has angle equal to a length divided by a length; radians have no dimension (whereas degrees are a dimension, just like feet or pounds or seconds are).

Degrees are useful for talking and writing, but radians are what is needed for calculations.

Vectors and Linear Algebra

A vector (in three dimensions) is simply a line drawn from the origin (0,0,0) to some point (x,y,z). Since they always emanate from the origin, it is correct to refer to "the vector (x,y,z)". The important thing is that it has both length _and_ direction. A physical example is the difference between velocity and speed. Speed is just a quantity: for example, "120 Miles per hour". Velocity, on the other hand, has _direction_ -- you might be going straight up, or straight down, or turning around a curve, etc. and the _length_ of the velocity vector gives the speed: 120 MPH.

But all we need to worry about here is the geometric meaning, and the difference between the _point_ (Px,Py,Pz) and the _vector_ (Px, Py, Pz). The point P is just a point, but geometrically the vector P is a line extending *from* the origin *to* the point P -- it has a length, and a direction: it points in the direction of (Px, Py, Pz). We will be using vectors for rotations, and to figure out what direction something points, and all sorts of other stuff.

The dimension of a vector is the number of elements in that vector. Let P be a vector. A 2D vector might be P=(Px,Py). A three dimensional vector example is P=(Px,Py,Pz). P=(p1,p2,p3,p4,p5,p6) would be a six-dimensional vector. So an n-dimensional vector is just a list of n independent quantities. We'll just be dealing with two and three dimensional vectors here, though, so when you see a sentence like "The vector v1" just think of (v1x, v1y, v1z).

The length of a vector is again given by Pythagorus:

        r^2 = Px^2 + Py^2 + Pz^2 + ...

i.e. the sum of the squares of all the elements. This is a good calculation to avoid in algorithms, since it is expensive, but it is useful to know.

The simplest thing one can do with a vector is change its length, by multiplying by a constant:

        c*(Px,Py,Pz) = (c*Px, c*Py, c*Pz).

Multiplying by a constant multiplies all elements in the vector by that constant. Just like with triangles all lengths increase proportionally, so it is a vector which points in the same direction but has a different length.

Two vector operations that are very useful are the "dot product"

and the "cross product". I'll write the dot product of two vectors
R and P as either R.P or <R,P>, and it is defined as

```
        R . P = |R| |P| cos(theta)
```

that is, the length of R times the length of P times the cosine of the
angle between the two vectors. From this it is easy to show that
the dot product may also be written as

```
        R . P = Rx*Px + Ry*Py + ...
```

that is, multiply the individual components together and add them up.
Note that the dot product gives a _number_ (a scalar), NOT a vector.
Note also that the dot product between two vectors separated by an
angle greater than pi/2 will be negative, from the first equation,
and that the dot product of two perpendicular vectors is zero.
The dot product is sometimes referred to as the inner (or scalar)
product.

The cross-product (sometimes called the vector product or
skew product) is denoted by RxP and is given by

```
        R x P = (Ry*Pz-Rz*Py, Rz*Px-Rx*Pz, Rx*Py-Ry*Px)
```

As you can see, the result is a _vector_. In fact, this vector
is perpendicular to both R and P -- it is perpendicular to the plane
that R and P lie in. Its length is given by

```
        length = |R| |P| sin(theta)
```

Note that P x R = - R x P; the direction of the resulting vector is
usually determined by the "right hand rule". All that is important
here is to remember that the cross-product generates perpendicular
vectors. We won't be using any cross products in this article.

There are lots of other things we can do to vectors. One
of the most important is to multiply by a _matrix_. A matrix is
like a bunch of vectors grouped together, so it has rows and columns.
An example of a 2x2 matrix is

```
        [a b]
        [c d]
```

an example of a 3x3 matrix is

```
        [a b c]
        [d e f]
        [g h i]
```

and so on. The number of rows doesn't have to equal the number of
columns. In fact, an n-dimensional vector is just an n x 1 (read
"n by 1") matrix: n rows, but one column.

We add matrices together by adding the individual elements:

```
        [a1 a2 a3]   [b1 b2 b3]   [a1+b1 a2+b2 a3+b3]
        [a4 a5 a6] + [b4 b5 b6] = [a4+b4 a5+b5 a6+b6]
        [a7 a8 a9]   [b7 b8 b9]   [a7+b7 a8+b8 a9+b9]
```

We can multiply by a constant, which just multiplies all elements
by that constant (just like the vector).

Matrices can also be multiplied. The usual rule is "row times
column". That is, given two matrices A and B, you take rows of
A and dot them with columns of B:

```
        [A1 A2] [B1 B2] = [A1*B1+A2*B3  A1*B2+A2*B4]
        [A3 A4] [B3 B4]   [A3*B1+A4*B3  A3*B2+A4*B4]
```

In the above, the (1,1) element is the first row of A (A1 A2) times
the first column of B (B1 B3) to get A1*B1+A2*B3. And so on. (With
a little practice this becomes very easy). We will be multiplying
rotation matrices together, and multiplying matrices times vectors.

Although "row times column" is the usual way that this is
taught, it can also be looked at as "columns times elements". The
easiest example is to multiply a matrix A times a vector x. The first
method gives:

```
                [a1 a2 a3]            [ <row1,x> ]   [a1*x1 + a2*x2 + a3*x3]
        let A = [a4 a5 a6] then Ax =  [ <row2,x> ] = [a4*x1 + a5*x2 + a6*x3]
                [a7 a8 a9]            [ <row3,x> ]   [a7*x1 + a8*x2 + a9*x3]
```

The right-hand part may be written as

$$x1*\begin{bmatrix}a1\\a4\\a7\end{bmatrix} + x2*\begin{bmatrix}a2\\a5\\a8\end{bmatrix} + x3*\begin{bmatrix}a3\\a6\\a9\end{bmatrix}$$

or, in other words,

$$Ax = x1*column1 + x2*column2 + x3*column3$$

that is, the components of x times the columns of A, added together.
This is a very useful thing to be aware of, as we shall see.
Note that normal multiplication commutes: 3*2 = 2*3. In matrix multiplication, this is NOT true. Multiplication in general does NOT commute, and AB is usually different from BA.

We can also divide by matrices, but it isn't called division.
It's called inversion. Let's say you have an equation like

$$5*x = b.$$

To solve for x you would just multiply both sides by 1/5 i.e. by the "inverse" of 5. To solve a matrix equation like

$$Ax = b$$

we just multiply both sides by the inverse of A -- call it A'.
And in just the same way that 1/a * a is one, a matrix times its inverse is the _identity matrix_ which is the matrix with "1" down the diagonal:

$$I = \begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}$$

It is called the identity because IA = A; multiplying by the identity matrix is just like multiplying by one.
Inverting a matrix is in general a very expensive operation, and we don't need to go into it here. We will be doing some special inversions later on though, so keep in mind that an inversion un-does a matrix multiplication.

Transformations -- more than meets the eye!

Now we have an _extremely_ powerful tool at our disposal.
What happens when you multiply a matrix times a vector? You get a new vector, of the same dimension as the old one. That is, it takes the old vector and _transforms_ it to a new one. Take the two-dimensional case:

$$\begin{bmatrix}a & b\\c & d\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix} = \begin{bmatrix}a*x + b*y\\c*x + d*y\end{bmatrix}$$

Look familiar? Well if a=cos(s), b=-sin(s), c=cos(s), and d=sin(s) we get the earlier rotation equations, which we can now rewrite as

$$P' = RP$$

where R is the rotation matrix

$$R = \begin{bmatrix}\cos(s) & -\sin(s)\\\sin(s) & \cos(s)\end{bmatrix}.$$

The way to think about this is that R _operates_ on a vector P.
When we apply R to P, it rotates P to a new vector P'.
So far we haven't gained much. But in three dimensions, the rotation matrices look like

$$Rz = \begin{bmatrix}\cos(s) & -\sin(s) & 0\\\sin(s) & \cos(s) & 0\\0 & 0 & 1\end{bmatrix} \quad Ry = \begin{bmatrix}\cos(s) & 0 & \sin(s)\\0 & 1 & 0\\-\sin(s) & 0 & \cos(s)\end{bmatrix} \text{ etc.}$$

Try multiplying Rz times a vector (x,y,z) to see that it rotates x and y while leaving z unchanged -- it rotates about the z-axis.
Now let's say that we're navigating through some 3D world and have turned, and looked up, and turned a few more times, etc.
That is, we apply a bunch of rotations, all out of order:

$$Rx\ Ry\ Ry\ Rz\ Rx\ Rz\ Ry\ Ry\ Rx\ P = P'$$

All of the rotation matrices on the left hand side can be multiplied together into a single matrix R

```
        R = Rx Ry Ry Rz Rx Rz Ry Ry Rx
```

which is a _new_ transformation, which is the transformation you would get by applying all the little rotations in that specific order. This new matrix can now be applied to any number of vectors. And this is extremely useful and important.

Another incredibly useful thing to realize is that the inverse of a rotation matrix is just its transpose (reflect through the diagonal, i.e. element i,j swaps with element j,i). It's very easy to see for the individual rotation matrices -- the inverse of rotating by an amount s is to rotate by an amount -s, which flips the minus sign on the sin(s) terms above. And if you take the transpose of the accumulated matrix R above, remembering that (AB)^T = B^T A^T, you'll see that R^T just applies all of the inverse rotations in the opposite order -- it undoes each small rotation one at a time (multiply R^T times R to see that you end up with the identity matrix).

The important point, though, is that to invert any series of rotations, no matter how complicated, all we have to do is take a transpose.

Hidden Faces (orientation)

Finally there is the issue of hidden faces, and the related issue of polygon clipping. In previous articles a number of different methods have been tried. For hidden faces, the issue is to determine whether a polygon faces towards us (in which case it is visible) or away from us (in which case it isn't).

One way is to compute a normal vector (for example, to rotate a normal vector along with the rest of the face). A light ray which bounces off of any point on this face will be visible -- will go through the origin -- only if the face faces towards us. The normal vector tells us which direction face is pointing in. If we draw a line from our eye (the origin) to a point on the face, the normal vector will make an angle of 90 degrees with the face when the face is edge-on. So by computing the angle between the normal vector and a vector from the origin to a point on the face we can test for visibility by checking if it is greater than or less than 90 degrees.

We have a way of computing the angle between two vectors: the dot-product. Any point on the face will give a vector from the origin to the face, so choose some vertex V on the polygon, then dot it with the normal vector N:

```
        <V,N> = Vx*Nx + Vy*Ny + Vz*Nz = |V| |N| cos(theta)
```

Since cos(theta) is positive for theta<90 and negative for theta>90 all that needs to be done is to compute Vx*Nx + ... and check whether it is positive or negative. If you know additional information about either V or N (as earlier articles did) then this calculation can be simplified by quite a lot (as earlier articles did!). And if you want to be really cheap, you can just use the z-coordinate of the normal vector and skip the dot-product altogether (this only works for objects which are reasonably far away, though).

Another method involves the cross-product -- take the cross-product of two vectors in the _projected_ polygon and see whether it points into or out of the monitor. Again, only the direction is of interest, so that usually means that only the z-component of the vector needs to be computed. On the downside, I seem to recall that in practice this method was cumbersome and tended to fail for certain polygons, making them never visible (because of doing integer calculations and losing remainders).

The final method is a simple ordering argument: list the points of the polygon in a particular order, say clockwise. If, after rotating, that same point list goes around the polygon in a counter-clockwise order then the polygon is turned around the other way. This is the method that the 3d library uses. It is more or less a freebie -- it falls out automatically from setting up the polygon draw, so it takes no extra computation time for visible polygons. It is best-suited to solid polygon routines, though.

Polygon clipping refers to determining when one polygon overlaps another. For convex objects (all internal angles are less than 180 degrees) this isn't an issue, but for concave objects it's a big deal. There are two convex objects in Cool World: the pointy stars and the Cobra Mk III. The pointy stars are clipped correctly, so that tines on the stars are drawn behind each other properly. The Cobra doesn't have any clipping, so you can see glitches when it draws one polyon on top of another when it should be behind it.

A general clipping algorithm is pretty tough and time-consuming. It is almost always best to split a concave object up into a group of smaller convex objects -- a little work on your part can save huge amounts of time on the computer's part.

And THAT, I think, covers the fundamentals of 3d graphics. Now it's on to constructing a 3D world, and implementing all that we've deduced and computed as an actual computer program.

Section 2: Constructing a 3D world

Now that we have the tools for making 3D objects, we need to put them all together to make a 3D world. This world might have many different objects in it, all independent and movable. They might see each other, run into each other, and so on. And they of course will have to be displayed on the computer.

As we shall see, we can do all this in a very elegant and simple manner.

Representing objects

The object problem boils down to a few object attributes: each object has a _position_ in the world, and each object has an _orientation_. Each object might also have a _velocity_, but velocity is very easy to understand and deal with once the position and orientation problem has been solved.

The position tells where an object is -- at least, where the center of the object is. The orientation vector tells in what direction the object is pointing. Velocity tells what direction the object is moving. Different programs will handle velocity in different ways. Although we are supposed to be tolerant of different positions and orientations, in these programs they are all handled the same way and are what will be focused on.

If you can't visualize this, just think of some kind of spaceship or fighter plane. It is located somewhere, and it points in a certain direction. The orientation vector is a little line with an arrow at the end -- it points in a certain direction and is anchored at some position. As the object turns, so does the orientation vector. And as it moves, so does the position.

Once positions and orientations are known we can calculate where everything is relative to everything else. "Relative" is an important word here. If we want to know how far we are from some object, our actual locations don't matter; just the relative locations. And if we are walking around and suddenly turn, then everything better turn around us and not some other point. Rotations are always about the origin, which means we are the origin -- the center of the universe. Recall also that the way projections work is by assuming we are looking straight down the z-axis, so our orientation better be straight.

So to find out what the world looks like from some object we need to do two things. First, all other objects need to be translated to put the specific object at the origin -- shift the coordinate system to put the object at (0,0,0). Second, the orientation vector of the object needs to be on the z-axis for projections to work right -- rotate the coordinate system and all of the objects around us to make the orientation vector be in the right direction.

Every time an object moves, its position changes. And every time it turns or tilts it changes its orientation. Let's say that after a while we are located at position P and have an orientation vector V, and want to look at an object located at position Q. Translating to to the origin is easy enough: just subtract P from all coordinates. So the original object will be located at P-P = 0, and the other object will be located at Q-P. What about rotating?

The orientation vector comes about by a series of rotations applied to the initial orientation. As was explained earlier, all of those rotations can be summed up as a single rotation matrix R, so if the intial vector was, say, z=(0,0,1) then

$$V = Rz$$

Given a position vector, what we need to do is un-rotate that position vector back onto the z-axis. Which means we just multiply by the _inverse_ of R, R'. For example, if we turn to the left, then the world turns to the right. If we turn left and then look up, the way to un-do that is to look back down and turn right. That's an inverse rotation, and we know from the earlier sections that rotation matrices are very easy to invert.

So, after translating and rotating, we are located at the origin and looking straight down the z-axis, whereas the other object Q is located at the rotated translated point

```
        Q2 = R' (Q-P)
```

i.e. Q-P translates the object, and R' rotates it relative to us.

Readers who are on the ball may have noticed that so far we have only deduced what happens to the _center_ of the object. Quite right. We need to figure out what happens to the points of an object.

First it should be clear that we want to define the points of an object relative to the center of the object. It is first necessary so that the object can rotate on its own. And it has all kinds of computational advantages, as we shall see in a later section; among them are that the numbers stay small (and the rotations fast), they are always rotated as a whole (so they don't lose accuracy or get out of proportion), and the points of objects which aren't visible don't have to be rotated at all.

So we need to amend the previous statement: we need to figure out what happens to the points of an object _only when that object is visible_.

Visibility

When is an object visible? When it is within your field of vision of course. It has to be in front of you, and not too far out to the sides.

So, after translating and rotating an object center,

```
        Q2 = R' (Q-P),
```

we need to check if Q2 is in front of us (if the z-coordinate is positive, and that it isn't too far away), and that it isn't too far out to either side. The field of vision might be a cone; the easiest one is a little pyramid made up of planes at 45 degrees to one another. That is, that you can see 45 degrees to either side or up and down. This is the easiest because the equation of a line (a plane actually) at 45 degrees is either

```
        x=z or x=-z,  y=z or y=-z
```

So it is very easy to check the x- and y-coordinates of the new center to see if -z < x < z and -z < y < z.

If the object is visible, then the rest of the object needs to be computed and displayed.

Computing displayed objects

Now let's say this object, which was located at Q, is visible. It has a bunch of points that define the object, relative to the center -- call those points X1 X2 X3 etc. where X1=(x1,y1,z1) etc. -- and it has an orientation W. Again, the orientation vector is computed by performing some series of rotations M on a vector which lies along the z-axis like (0,0,1).

First the points need to be rotated along with the orientation vector. This isn't an inverse rotation like before -- the points rotate in the same way as the orientation vector. So apply M to all the points; consider a single point X1:

```
        rotated point = M X1
```

Once the points have been rotated we have to find their actual location. Since they are defined relative to the center of the object, this means just adding them to the center of the object:

```
        rotated point + Q = M X1 + Q
```

This is the _actual_ location of the points in the world. Now as before we need to translate and rotate them to get their relative location:

```
        X1' = R' (M X1 + Q - P)
```

As before, subtract P and apply the inverse rotation R'. The above equation is the _entire_ equation of the 3D world!

We can rewrite this equation as

```
        X1' = R' M X1 + R'(Q-P)
```

and recognize that R'(Q-P) was calculated earlier, to see if the object was visible.

The above equation can be read as "Rotate the rotated object points backwards to the way we are facing, and add to the rotated and translated center." That is, if we turn one way, the center rotates in the opposite direction, and the entire object rotates in the opposite direction. Physically, if you turn your head sideways you still see the monitor facing you. If instead of turning your head you were to move the monitor, you would also have to rotate the monitor to keep it facing you (if you didn't rotate the monitor, and only changed its position, you would be able to see the sides, etc.). That rotation is in the opposite direction that your head would rotate (head turns to the right, monitor turns to the left).

Displaying objects

Now that everyone is rotated and translated and otherwise happy, they need to be projected and displayed. The projection is easy -- as before, divide by the z-coordinate and multiply by a magnification factor. Then connect the points in the right way to get the polygons and faces that make up the objects, using the appropriate method to remove faces that aren't visible, and draw to the screen.

One thing remains, though: depth-sorting the objects, to make sure that close-up objects overlap far-away objects. So, _before projecting_, all of the z-coordinates need to be looked at and the objects ordered so that they are drawn in the right way.

This is really a form of polygon clipping. You can do the same thing with various complex objects to get an easy way to clip.

Summary

All objects have a position and an orientation, and might have things like velocity. The position of the object is where the object is located. The points that make up any object are defined relative to this center; the center is thus the center of rotation of the object in addition to being its location. The orientation determines what direction the object is pointing in, and the velocity determines what direction it is moving in.

Navigating through the world amounts to changing the position and orientation (and velocity) of the object. To figure out how the world looks from any single object with position P, P=(Px,Py,Pz), and orientation matrix R (where R=cumulative rotation matrix), a very simple and elegant equation is used. First the equation

```
        R'(Q-P)
```

where R' = inverse of R, determines where the center of an object Q is located (by translating and then undoing the orientation of P), and tells whether the object at Q is visible or not. If it is, then the equation

```
        R' (M X + Q-P)
```

gives the new location of any point X on the object, where M is the orientation matrix for Q, and X is some point defined relative to Q. After depth-sorting the individual objects, these points are projected and drawn to the screen.

Doing things in this way gives a method that is very fast and efficient, and always retains accuracy -- everything is calculated relative to everything else. This is extremely powerful. It makes the programming of objects easy, it makes no roundoff errors from e.g. rotating rotated points, and no cycles are wasted calculating rotations of non-visible objects. Some PC algorithms you might come across do all sorts of God-awful things to overcome these problems, like rotate a single point and then use cross-products to preserve all the angles, and all sorts of other horrid mathematical butchery. That way leads to the Dark Side.

By the way, there are some subtleties (for example, computing orientation vectors) in this calculation -- see the discussion of Cool World in Section 4 for more detail on implementation issues.

Section 3: Implementing the 3D Library

Now that we've dined on a tasty and nutritous meal of some theory we need to figure out how to implement everything on the 64, and to do so efficiently! The idea of the 3D library is to collect all of the important and difficult routines into a single place. Before doing so it is probably a good idea to figure out just what the important routines are.

Obviously rotations and projections are important. In fact, there just isn't much else to do -- a few additions here and there, maybe a few manipulations, but everything else is really straightforward. Of course, once that data is rotated and projected it would probably be much more enjoyable for the viewer to display it on the screen. Drawing lines is pretty easy, but a good solid-polygon routine is pretty tough so it makes a good addition to the library.

So three things, and the first is rotations. There are two kinds of rotations. When we turn left or right we rotate the world, which means rotating the positions of objects. We also need to rotate the object itself, which means rotating the individual points that define the object.

Calculating rotation matrices

In both cases, we need a rotation matrix that we can use to transform coordinates. Previous programs just calculated a rotation matrix like

```
        R = Ry Rz Rx    so that for some point P:  RP = Ry Rz Rx P
```

that is, given three angles sx, sy, and sz, calculate the rotation matrix you get by first rotating around the x-axis by an amount sx (Rx times P), then the z-axis by an amount sz (Rz times Rx P), and finally the y-axis by the amount sy (Ry times (Rz times Rx P)).

Well, that kind of routine just doesn't cut it anymore. Why? Well, what happens when you turn left, turn left, look up, and roll? You get a matrix that looks like

```
        R = Rz Rx Ry Ry     (think Ry=turn left, Rx=look up, Rz=roll)
```

What three angles sx sy and sz would give us that exact rotation matrix? -We don't know-! As has been said many times, **matrix multiplications do not commute**, so rotations do not commute, so we can't just add up the rotation amounts or something. We have to keep a running rotation matrix, that just accumulates rotations each time we turn left or roll or whatever. In terms of a routine, that means we need to be able to compute

```
        Rx R  and  Ry R  and  Rz R
```

where R is the accumulated rotation matrix, and Rx etc. are the little rotation matrices about the x/y/z axis that correspond to us turning left and right such. Although this seems like a lot of work, as we shall see it is actually quite a bit faster than calculating an entire rotation matrix like the old routine.

Still, it is sometimes handy to be able to calculate a rotation matrix using the old method, like when rotating by large amounts, or if we just need an object to spin around in some arbitrary way, or if all we need is a direction and an elevation (think of a gun turret or a telescope). So an improved version of the old routine is also included.

Rotating points

That ought to take care of calculating the rotation matrix. Now we need to apply that matrix to some points. The first type of points are the object centers (i.e. positions in the world). Polygonamy used a single byte to represent positions, and that was really too restrictive. A full 16-bit number is needed, and it should be signed. So we need to be able to rotate 16-bit signed numbers.

The second type of points are the object points, which are defined relative to the object centers. Since they are relative to the centers these points can be 8-bits, but they should definitely be signed integers. There are many more object points than there are objects, so these rotations need to be really fast.

In both cases, it clearly makes much more sense to pass in a whole list of points to be rotated, instead of having to call

the routine for each individual point.
        Remember that the object points are only rotated if the object is actually visible. And, in point of fact, once they've been rotated they will need to be projected. So it makes sense to tack the projection routine onto the object-rotation routine (as opposed to the world-rotation routine, which rotates the object centers). To summarize, then, in addition to the matrix calculation routines we need two rotation routines: one for rotating 16-bit signed object centers, and one for rotating (and possibly projecting) the actual object points.

Drawing Polygons

        Once everything has been rotated and projected, it will need to be drawn. So a good solid polygon routine makes a great addition to the library. The algorithm will be built upon the polygonamy idea: start at the bottom of the polygon, and draw the left and right sides of the polygon _simultaneously_, filling in-between. That is, move up in the y-direction, calculate the left-endpoint, calculate the right-endpoint, and fill.
        To calculate the endpoints we just calculate the slopes of the lines, and draw. So all that is needed is to pass in a list of points, and perhaps tell the routine where to draw those points. Recall also that this gives a very easy way of doing hidden faces: if the points are given in counter-clockwise order for the normal polygon, then they will be in clockwise order if the polygon is facing away from us. So basically the polygon isn't visible if we can't draw it, and the hidden-face calculation is more or less a freebie. (That is, the hidden-face calculation takes no extra time if the face is visible; it will use up some time though if the face is not visible.)
        The polygonamy routine had two large disadvantages: it was huge, because of the way the fill-routines and tables were set up, and it was inflexible in the sense that it could only draw to a specific bitmap. It also couldn't handle things like negative coordinates, or drawing polygons which were partially off-screen. The code was also a bit convoluted, and in need of re-thinking (imagine a big prototype machine with exposed wires sticking out and large cables snaking around, all held together with duct tape and bailing wire, and you'll get an idea of what the polygonamy code looks like). Of course, since we already know the ideas and equations, it's not too tough to just rework everything from scratch.

Basic Routines

        Clearly, before writing all of the library routines we are going to have to figure out how to do some very general operations. We are going to need some 8- and 16-bit signed multiply routines. The projection routine is going to involve a divide of a 16-bit z-coordinate, and the polygon routine is going to need a division routine for calculating line slopes. Then we have to figure out how to represent numbers and on and on. Hokey smokes! This is all new stuff, too.

        But, first and foremost, we need a multiply. We will of course be using the fast multiply, which uses the fact that

```
        a*b = f(a+b) - f(a-b),   f(x) = x^2/4.
```

Understanding this routine will be crucial to understanding what will follow. The general routine to multiply .Y and .A looks like

```
        STA ZP1          ;ZP1 points to f(x) low byte
        STA ZP2          ;high byte
        EOR #$FF
        ADC #01
        STA ZP3          ;table of f(x-256) -- see below
        STA ZP4
        LDA (ZP1),Y      ;f(Y+A), low byte
        SEC
        SBC (ZP3),Y      ;f(Y-A), low byte
        TAX
        LDA (ZP2),Y      ;f(Y+A), high byte
        SBC (ZP4),Y      ;f(Y-A), high byte
                         ;result is now in .X .A = low, high
```

By using the indexed addressing mode, we let the CPU add A and Y together. If Y and A can both range from 0..255, then Y+A can range from 0..510. So we need a 510-byte table of f(x). What about Y-A? The complementing of A means that what we are really calculating on the computer is Y + 256-A, or 256+Y-A. So we need another table of f(x-256) to correctly get f(Y-A).

Consider Y=1, A=2. -A = $FE in 2's complement, so added together we get $FF, or -1. Next consider Y=2, A=1. In this case, Y-A gives 2+$FF = $101, or 257. Computationally, if Y-A is 0..255 it is a negative number, and subtracting 256 gives the correct negative number. If Y-A is 256..510, then it really represents 0..255, so again subtracting 256 gives the correct number. If you think about it, you'll see that the upper 256 bytes of this table is the same as the lower 256 bytes of the first table: f(x-256) for x=256..511 is the same as f(x) for x=0..255. This means that the first table can be piggy-backed on top of the second table in memory. For example, if the table were at $C000

```
        $C000   f(x-256)
        $C100   f(x)  (512 bytes)
```

would do the trick. WITH ONE EXCEPTION. And a rather silly one at that. That exception is A=0. Multiplying by zero should give a zero. But the complementing of A gives -256, which on the computer is... zero! That is, think of A=0 and Y=0. A-Y=0, and when we look it up in the table of f(x-256) we will get f(-256) instead of f(0), and get the wrong answer. So we have to check for A=0.

And oh yes: don't forget to round the tables correctly!

Extremely Cool Stuff

As usual, a little bit of thinking and a little bit of planning ahead will lead to enormous benefits.

After staring at the fast multiplication code for a bit, we can make a very useful observation:

Once the zero page locations are set up, they stay set up.

This makes all the difference in the world! Let's say we've just calculated a*b. If we now want to calculate c*b, _we don't have to set up b again_.

If we have a succession of calculations, x*a1, x*a2, x*a3, etc., we only have to set up the ZP location once (with x and -x) and keep reusing it. This makes the fast multiply become very fast! With a lot of multiplications it reduces to just 24 cycles or so for a full 16-bit result, as all of that zero-page overhead disappears.

Now think about rotations. We have to multiply a matrix times a vector (i.e. the three coordinates). It was pointed out in the discussion of linear algebra that we can write the multiplication as

```
          [x1]
        R [x2] = x1*column1 of R + x2*column2 of R + x3*column3 of R
          [x3]
```

And there is the payoff: we get three multiplies for every one set up of zero page. That is, we set x1 in the zero page variables, and then multiply by the three elements of the first column. Then do the same for x2 and x3. So we still do nine multiplications, but we only have to set up the zero-page locations three times. Just changing the way in which we do these multiplications has instantly given us a great big time savings!

Now we need to figure out how to do signed multiplies. The fast multiply runs into problems adjusting to signed numbers. Consider the simple example of x=-100 and y=1. We get that

```
        f(x+y) = f(256-100 + 1) = f(157).
```

But this is just what we'd get if we multiplied x=100 and y=57 together. You don't know whether 157 is -99 or if it really is 157. So we can't just modify the tables... unless... well, what about if we are able to remove this ambiguity from the tables? We can do just that if we restrict the range of allowed numbers. For example, if we restrict to

```
        x=-96..95 = 160..255,0..95  (-96 is 160 in 2's complement)
        y=-64..64 = 192..255,0..64  (-64 is 192, -63 is 193, etc.)
```

then

```
        if x+y = 0..159   the actual number is  0..159
```

```
                 160..255                        -96..-1
                 256..351                        0..95
                 352..510                        -160..-2
```

That is, the _only_ way you can get the number 159 from x+y is when
x=95 and y=64. The four ranges above are derived by considering
the four cases x>0 and y>0, x<0 and y>0, x<0 and y<0, x>0 and y<0.
All values of x+y and x-y, from 0 to 510, correspond to _unique_ values
of x and y, if we restrict x to the range -96..95 and restrict y to the
range -64..64. With those restrictions, we can construct a *single*
512-byte fast-multiply table to do a signed fast-multiply.
        How is this useful?

        One word: object rotations. If the rotation matrix values
range from -64..64, and the object points range from -96..95, then
we can do an _extremely_ fast multiply to perform the object rotations,
as we shall see in the discussion of the rotation routines. Since
there are quite a lot of object points to rotate, this makes an
enormous contribution to the overall speed of the routine.

        Although the above helps out greatly for the point rotations,
we still need a general signed multiply routine for the centers and
the projections and such. So, let's figure it out!
        Say we multiply two numbers x and y together, and x is negative.
If we plug it in to a multiplication routine (_any_ multiplication
routine), we will really be calculating

```
        (2^N + x)*y = 2^N*y + x*y
```

assuming that x is represented using 2's complement (N would be 8 or 16
or whatever). There are two observations:

```
        - If the result is _less_ than 2^N, we are done -- 2^N*y is all
          in the higher bytes which we don't care about.

        - Otherwise, subtract 2^N*y from the result, i.e. subtract
          y from the high bytes.
```

Now let's say that both x and y are negative. Then on the computer
the number we will get is

```
        (2^N + x)*(2^N + y) = 2^(2N) + 2^N*x + 2^N*y - x*y
```

Now it is too large by a factor of 2^2N, 2^N*x and 2^N*y. BUT
the basic observations haven't changed a bit! We still need to
subtract x and y from the high bytes. And the 2^2N is totally
irrelevant -- we can't get numbers that large by multiplying numbers
together which are no larger than 2^N.
        This leads to the following algorithm for doing signed
multiplications:

```
        multiply x and y as normal with some routine
        if x<0 then subtract y from the high bytes of the result
        if y<0 then subtract x from the high bytes
```

And that's all there is to it! Note that x and y are "backwards",
i.e. subtract y, and not x, when x<0. Some examples:

```
        x=-1, y=16  Computer: x=$FF y=$10  (N=8)
                    x*y = $0FF0
                    Result is less than 256, so ignore high byte
                        Answer = $F0 = -16
                    OR: subtract y from high byte,
                        Answer = $FFF0 = -16

        x=2112 y=-365 Computer: x=$0840 y=$FE93   (N=16)
                    x*y = $08343CC0
                    y<0 so subtract x from high bytes (x*2^16),
                        Answer = $F43CC0 = -770880

        x=-31 y=-41 Computer: x=$E1 y=$D7
                    x*y = $BCF7
                    x<0 so subtract $D700 -> $E5F7
                    y<0 so subtract $E100 -> $04F7 = 1271 = correct!
```

So, in summary, signed multiplies can be done with the same fast
multiply routine along with some _very simple_ post-processing.
And if we know something about the result ahead of time (like if
it's less than 256 or whatever) then it takes _no_ additional
processing!
        How cool is that?

Projections

After rotating, and adding object points to the rotated centers, the points need to be projected. Polygonamy had a really crappy projection algorithm, which just didn't give very good results. I don't remember how it worked, but I do remember that it sacrificed accuracy for speed, and as a result polygons on the screen looked like they were made of rubber as their points shifted around. So we need to re-solve this problem.

Recall that projection amounts to dividing by the z-coordinate and multiplying by a magnification factor. Since the object points will all be 16-bits (after adding to the 16-bit centers), this means dividing a 16-bit signed number by another 16-bit number, and then doing a multiplication. Bleah.

The first thing to do is to get rid of the divide. The projected coordinates are given by

```
        x' = d*x/z = x * d/z,   y' = d*y/z = y * d/z
```

where d is the magnification factor and z is the z-coordinate of the object. The obvious way to avoid the divide is to convert it into a multiply by somehow calculating d/z. The problem is that z is a 16-bit number; if it were 8 bits we could just do a table lookup. If only it were 8 bits... Hmmmmm...

Physically the z-coordinate represents how far away the object is. For projections, we need

- accuracy for small values of z (when objects are close to us and large)

- speed for large values of z

If z is large then accuracy isn't such a big deal, since the objects are far away and indistinct. So if z is larger than 8 bits, we can just shift it right until it is eight bits and ignore the tiny loss of accuracy. Moreover, note that the equations compute x/z and y/z; that is, a 16-bit number on top of another. Therefore if we shift _both_ x and z right we don't change the value at all! So now we have a nifty algorithm:

```
        if z>255 then shift z, x, and y right until z is eight bits
        compute d/z
        multiply by x, multiply by y
```

There are still two more steps in this algorithm, the first of which is to compute d/z. This number can be less than one and as large as d, and will almost always have a remainder. So we are going to need not only the integer part but the fractional part as well. The second step is to multiply this number times x.

Let d/z = N + R/256, so N=integer part and R=fractional part. For example, 3/2 = 1 + 128/256 = 1.5. Also let x = 256*xh + xl. Then the multiplication is

```
        x * d/z = (256*xh + xl) * (N + R/256)
                = 256*xh*N + xh*R + xl*N + xl*R/256
```

There are four terms in this expression. We know ahead of time that the result is going to be less than 16-bits (remember that the screen is only 320x200; if we are projecting stuff and getting coordinates like 40,000 then there is a serious problem! And nothing in the library is set up to handle 24-bit coordinates anyways).

The first term, xh*N, must therefore be an eight-bit number, since if it were 16-bits multiplying by 256 would give a 24-bit number, which we've just said won't happen. So we only really care about the lower eight bits of xh*N. The next two terms, xh*R and xl*N, will be 16-bit numbers. The last term, xl*R. will also be 16-bits, but since we divide by 256 we only care about the high 8-bits of the result (we don't need any fractional parts for anything).

And that, friends, takes care of projection: accurate when it needs to be, and still very fast.

Cumulative Rotations

There is still this problem with the accumulating rotation matrix. The problem to be solved is: we want to accumulate some general rotation operator

```
            [A B C]
```

$$M = \begin{bmatrix} D & E & F \\ G & H & I \end{bmatrix}$$

by applying a rotation matrix like

$$Rx = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(t) & \sin(t) \\ 0 & -\sin(t) & \cos(t) \end{bmatrix}$$

to it. So, just do it:

$$Rx\ M = \begin{bmatrix} A & B & C \\ D*\cos(t)+G*\sin(t) & E*\cos(t)+H*\sin(t) & F*\cos(t)+I*\sin(t) \\ -D*\sin(t)+G*\cos(t) & -E*\sin(t)+H*\cos(t) & -F*\sin(t)+I*\cos(t) \end{bmatrix}$$

You might want to do the matrix multiplication yourself, to double-check. Notice that it only affects rows 2 and 3. Also notice that only one column is affected at a time: that is, the first column of Rx M contains only D and G (the first column of M); the second column only E and H; etc. Similar expressions result when multiplying by Ry or Rz -- they just affect different rows. The point is that we don't need a whole bunch of routines -- we just need one routine, and to tell it which rows to operate on.

In general, the full multiplication is a pretty hairy problem. But if the angle t is some fixed amount then it is quite simple. For example, if t is some small angle like 3 degrees then we can turn left and right in 3 degree increments, and the quantities cos(t) and sin(t) are just constants. Notice also that rotating in the opposite direction corresponds to letting t go to -t (i.e. rotating by -angle). Since sine is an odd function, this just flips the sign of the sines (i.e. sin(-t) = -sin(t)).

If sin(t) and cos(t) are constants, then all that is needed is a table of g(x) = x*sin(t) and x*cos(t); the above calculation then reduces to just a few table lookups and additions/subtractions. There's just one caveat: if t is a small number (like three degrees) then cos(t) is very close to 1 and sin(t) is very close to 0. That is to say, the fractional parts of x*sin(t) and x*cos(t) are very important!

So we will need two tables, one containing the integer part of x*cos(t) and the other containing the fractional. This in turn means that we need to keep track of the fractions in the accumulation matrix. The accumulation matrix will therefore have eighteen elements; nine integer parts and nine fractional parts. (The fractional part is just a decimal number times 256).

The routines to rotate and project only use the integer parts; the only routines that really need the fractional parts are the accumulation routines.

How important is the fractional part? Very important. There is a class of transformations called area-preserving transformations, of which rotations are a member. That is, if you rotate an object, the area of that object does not change. The condition for a matrix to be area-preserving is that its determinant is equal to one; if the cumulative rotation matrix starts to get inaccurate then its determinant will gradually drift away from one, and it will no longer preserve areas. This in turn means that it will start to distort an object when it is applied to the object. So, accuracy is very important where rotation matrices are concerned!

There is one other issue that needs to be taken care of. Rotation matrices have this little feature that the largest value that any element of the matrix can ever take is: one! And usually the elements are all less than one. To retain accuracy we need to multiply the entire matrix by a constant; a natural value is 64, so that instead of ranging from -1..1 the rotation matrix elements will range from -64..64. And as was pointed out earlier, we can do some very fast signed arithmetic if one of the numbers is between -64 and 64.

The downside is that we have to remember to divide out that factor of 64 once the calculation is done -- it's just a temporary place-holder. For object point rotations we can incorporate this into the multiplication table, but for the 16-bit center rotations we will have to divide it out manually.

Polygon Routine Calculations

Finally, there's the polygon routine. Starting at the lowest point on the polygon, we need to draw the left and right sides simultaneously. Specifically we want to draw the lines to the left and right, and we only care about the x-coordinates of the endpoints at each value of y:

```
        start at the bottom
        Decrement y (move upwards one step)
        Compute x-coord of left line
        Compute y-coord of right line
        (Fill in-between)
```

Drawing a line is easy. The equation of a line is

```
        dy = m*dx,    m=slope, dy=change in y, dx=change in x
```

The question posed by the above algorithm is "If I take a single step in y, how far do I step in x?" Mathematically: if dy=1, what is dx? Obviously,

```
        dx = 1/m
```

i.e. the inverse slope. If the endpoints of the line are at (X1,Y1) and (X2,Y2), then the inverse slope is just

```
        1/m = DX/DY,  where DX=X2-X1 and DY=Y2-Y1.
```

Once this is known for the left and right sides, updating the x-coordinates is a simple matter of just calculating x+dx (dx=DX/DY). Note that if DY=0, the line segment is a straight horizontal line. Therefore we can just skip to the next point in the polygon, and let the fill routine -- which is very good at drawing straight lines -- take care of it.

Now we can start drawing the left and right sides. As we all know, lines on a computer are really a series of horizontal line segments:

```
                   *****      **
              *****             *           (lines with positive and
          ****                   **          negative slope)
     *****                         *
```

Note that the polygon routine doesn't calculate all points on the line, the way a normal line routine does. It only calculates the left or right endpoints of each of those horizontal line segments. For example, if dx=5 then the routine takes big steps of 5 units each; the fill routine takes care of the rest of the line segment.

There are two issues that need to be dealt with in the polygon routine. The first trick in drawing lines is to split one of those line segments in half, for the first and last points. That is, consider a line with DY=1 and DX=10. From the above equations, dx = 10, and if this line was just drawn naively it would look like

```
                 *
        **********
```

i.e. a single horizontal line segment of length 10, followed by a dot at the last point. What we really want is to divide that one line segment in half, to get

```
             *****
        *****
```

for a nice looking line. In the polygon routine, this means that the first step should be of size dx/2. All subsequent steps will be of size dx, except for the last point.

The other issue to be dealt with again deals with the endpoints. How to deal with them depends on two things: the slope of the line, and whether it is the left or right side of the polygon. Consider the below line, with endpoints marked with F and L (for first and last)

```
              *L**
          ****
        F*
```

and dx=4. You can see that we have overshot the last point. In point of fact, we may also have overshot the first point. Let's say the above was the left side of a polygon. Then we would want to start filling at the LEFT edge of each line segment -- we want that * to the left of L to be drawn, but want to start filling at F. On the other hand, if this were the right side of the polygon then we would want to fill to the right edge of each line segment. That means filling to point L, again grabbing the * to the left of L, and filling to the * to the right of F.

The correct way to handle this is actually very simple: place the x-coordinate update (x=x+dx) in just the right spot, either before or after the plot/fill routine. In the above example, the left line should do the update after the plot/fill:

```
        fill in-between and plot endpoints
        x=x+dx
```

The line will then always use the "previous" endpoint: it will start with F, and then with the last point of the previous line segment. The right line should do the updating before the plotting: it will then use the rightmost point of each segment, and when the end of the line is reached the next line will be computed, resetting the starting point.

Polygonamy put some wacky restrictions on DX and DY that just won't cut it here. The lib3d routine can handle negative coordinates, and polygons which are partially off-screen. The only restriction it puts on the numbers is that DX is 9-bits and DY is 8-bits; the coordinates X2 and Y2 etc. can be a full 16-bits, but the sides of the polygons can't exceed DX=9-bits and DY=8-bits.

In principle this allows you to draw polygons which are as large as the entire screen. In practice it was perhaps not a great decision: if these polygons are being rotated, then a polygon with DX=320 will have DY=320 after a 90-degree rotation. So in a sense polygons _sides_ are actually limited to be no larger than 256 units.

The important word here though is "side". A polygon can be much larger than 256x256 if it has more than one side! For example, a stopsign has eight sides, but each side is smaller than the distance from one end of the polygon to the other. Also, sides can always be split in half. This restriction is just one of those design decisions...

Anyways, DX=9-bits and DY=8-bits. To calculate this we need a divide routine. Polygonamy used a routine involving logs and some fudged tables and other complexities. Well most of that was pretty silly, because the divide is such a tiny portion of the whole polygon routine -- saving a few tens of cycles is awfully small potatoes in a routine that's using many thousands of cycles.

So the new strategy is to do the general routine: calculate log(DX) and log(DY), subtract, and take EXP() of the result to get an initial guess. Then use a fast multiply to see how accurate the guess was, and if it is too large or too small then a few subtractions or additions can fix things up. The remainder is also computed as a decimal number (remainder/DY times 256); the remainder is crucial, as accuracy is important -- if the lines are not drawn accurately, the polygon sides won't join together, and the polygon will look very weird.

Why not do a divide like in the projection routine? Well, first of all, I wrote the polygon renderer before deriving that routine. :) Second, that routine combines a multiply _and_ a divide into a single multiply; there is no multiply here, and if you count up the cycles I think you'll find that the logarithmic routine is faster, and with the remainder calculation it is simply better-suited for drawing polygons.

Although the remaining parts of the algorithm require a lot of blood and sweat to get the cycle count down, they are relatively straightforward. I leave their explanation, along with a few other subtleties, for the detailed code disassemblies, below.

Detailed Code Disassembly

After all of those preliminary calculations and ideas and concepts we are finally in a position to write some decent code. The full lib3d source code is included in this issue, but it is worth examining the routines in more detail. Five routines will be discussed below:

```
        CALCMAT - Calculate a rotation matrix
        ACCROTX - Accumulate the rotation matrix by a fixed amount
        GLOBROT - Global rotate (rotate centers)
        ROTPROJ - Local rotation (object points), and project if necessary
        POLYFILL- Polygon routine
```

Only the major portions of the routines will be highlighted. And the first routine is...

CALCMAT

```
*
* Calculate the local matrix
```

```
*
* Pass in: A,X,Y = angles around z,x,y axis
*
* Strategy: M = Ry Rx Rz where Rz=roll, Rx=pitch, Ry=yaw
*
```

CALCMAT calculates a rotation matrix using the old method: pass in the rotation amounts about the x y and z axis and calculate a matrix. There is a fairly complete explanation of this routine in the source code, but it has a very important feature: the order of the rotations.

As the above comment indicates, it first takes care of roll (tilting your head sideways), then pitch (looking up and down), and finally yaw (looking left or right). This routine can't be used for a general 3D world, but by doing rotations in this order it _can_ be used for certain kinds of motion, like motion in a plane (driving a car around), or an angle/elevation motion (like a gun turret).

Note also the major difference from the accumulation routines: they rotate by a fixed amount. Since this routine just uses the angles as parameters, the program that uses this routine can of course update the angles by any amount it likes. That is: the accumulation routines can only turn left or right in 3 degree increments; this routine can instantly point in any direction.

Next up are the accumulation routines. These routines apply a fixed rotation to the rotation matrix; i.e. they rotate the world by a fixed amount about an axis. The carry flag is used to indicate whether rotations are positive or negative (clockwise or counter-clockwise if you like).

As was pointed out earlier, only one routine is needed to do the actual rotation calculation. All it needs to know is which rows to operate on, so that is the job of ACCROTX ACCROTY and ACCROTZ.

ACCROTX

```
*
* The next three procedures rotate the accumulation
* matrix (multiply by Rx, Ry, or Rz).
*
* Carry clear means rotate by positive amount, clear
* means rotate by negative amount.
*
ACCROTX  ENT
```

The x-rotation just operates on rows 2 and 3; these routines just loop through the row elements, passing them to the main routine ROTXY. The new, rotated elements are then stored in the rotation matrix, and on exit the rotation matrix has accumulated a rotation about the x-axis.

```
         ROR ROTFLAG
         LDX #2
:LOOP    STX COUNT
         LDA D21REM,X     ;rows 2 and 3
         STA AUXP
         LDA G31REM,X
         STA AUXP+1

         LDY G31,X        ;.Y = row 3
         LDA D21,X        ;.X = row 2
         TAX
         JSR ROTXY
         LDX COUNT
         LDA TEMPX
         STA D21REM,X
         LDA TEMPX+1
         STA D21,X
         LDA TEMPY
         STA G31REM,X
         LDA TEMPY+1
         STA G31,X
         DEX
         BPL :LOOP
         RTS

*
* Rotate .X,AUXP .Y,AUXP+1 -> TEMPX,+1 TEMPY,+1
```

```
*
* If flag is set for negative rotations, swap X and Y
* and swap destinations (TEMPX TEMPY)
*
ROTXY
```

This is the main accumulation routine. It simply calculates
x*cos(delta) + y*sin(delta), and y*cos(delta) - x*sin(delta).
Since delta is so small, cos(delta) is very nearly 1 and sin(delta)
is very nearly zero: an integer and a remainder part of x and y
are necessary to properly calculate x*cos(delta) etc.

Let x = xint + xrem (integer and remainder). Then

```
        x*cos(delta) = xint*cos(delta) + xrem*cos(delta).
```

Since cos(delta) is nearly equal to 1, xint*cos(delta) is equal to
xint plus a small correction; that is, xint*cos(delta) gives an
integer part and a remainder part. What about xrem*cos(delta)? It
also gives a small correction to xrem. But xrem is already small!
Which means the correction to xrem is really really small, i.e outside
the range of our numbers. So we can just discard the correction.
This can generate a little bit of numerical error, but it is miniscule,
and tends to get lost in the noise.

The point of all this is that x*cos(delta) + y*sin(delta) is calculated
as

```
        xrem + xint*cos(delta) + yint*sin(delta)
```

where xrem*cos(delta) is taken to be xrem, and yrem*sin(delta) is taken
to be zero. This introduces a very small error into the computation,
but the errors tend to get lost in the wash.

```
:CONT    LDA CDELREM,X
         CLC
         ADC SDELREM,Y
         STA TEMPX
         LDA CDEL,X       ;x*cos(delta)
         ADC SDEL,Y       ;+y*sin(delta)
         STA TEMPX+1
         LDA TEMPX
         CLC
         ADC AUXP         ;+xrem
         STA TEMPX
         BCC :NEXT
         INC TEMPX+1

:NEXT    a similar piece of code to calculate y*cos - x*sin
```

And that is the entire routine -- quite zippy, and at 14 bits per
number it is remarkably accurate.

GLOBROT

Next up is GLOBROT. The centers are 16-bit signed numbers, and
the rotation matrix of course consists of 8-bit signed numbers.

```
*
* Perform a global rotation, i.e. rotate the centers
* (16-bit signed value) by the rotation matrix.
*
* The multiplication multiplies to get a 24-bit result
* and then divides the result by 64 (mult by 4).  To
* perform the signed multiplication:
*   - multiply C*y as normal
*   - if y<0 then subtract 256*C
*   - if C<0 then subtract 2^16*y
*
* Parameters: .Y = number of points to rotate
*
```

Recall that the object centers are 16-bit signed coordinates; therefore
a full 16-bit signed multiply is needed. Two macros are used for
the multiplies:

```
MULTAY   MAC              ;Multiply A*Y, store in var1, A
```

which does the usual fast multiply, and

```
QMULTAY  MAC              ;Assumes pointers already set up
```

```
         LDA (MULTLO1),Y
         SEC
         SBC (MULTLO2),Y
         STA ]1
         LDA (MULTHI1),Y
         SBC (MULTHI2),Y
         <<<
```

which does a "quick multiply", that is, it is the fast multiply
without any of the zero-page setup muckety-muck. As was pointed out
in the theory section, once the zero page variables are set up,
they stay set up! ROTPROJ can take better advantage of this,
but GLOBROT can also benefit. Remember also that if A=0 this
routine will fail; the routines below must check for A=0 ahead of
time.

Since the matrix elements are all "6-bit offset", that is, since
they are the actual value times 64, the final result after the
rotations needs to be divided by 64. The easiest way to divide the
24-bit result by 64 is to shift *left* twice, and keep the upper bits.
In the macro below, "C" is the "C"enter, and Mij is some element in
the rotation matrix.

```
* Fix sign and divide by 64

DIVFIX   MAC                ;If Mij<0 subtract 256*C
         LDY MULTLO1        ;If C<0 subtract 2^16*Mij
         BPL POSM
         STA TEMP3
         LDA TEMP2
         SEC
         SBC ]1             ;Center
         STA TEMP2
         LDA TEMP3
         SBC ]1+1           ;high byte
POSM     LDY ]1+1
         BPL DIV
         SEC
         SBC MULTLO1        ;Subtract Mij

DIV      ASL TEMP1          ;Divide by 64
         ROL TEMP2
         ROL
         ASL TEMP1
         ROL TEMP2
         ROL
         LDY TEMP2
         <<<                ;.A, .Y = final result


* Main routine

GLOBROT  ENT
```

Recall that there are two ways to look at multiplying a matrix times
a vector. The usual way is "row times column"... and that is exactly
how this routine works. Why not do the "column times vector element"
method, which we said offers this great computational advantage?

Doing things that way means calculating

```
        [A11]      [B12]      [C13]
        [D21]*CX + [E22]*CY + [F23]*CZ
        [G31]      [H32]      [I33]
```

where the A11 etc. are the columns of the rotation matrix. The
argument was that CX could be set up in zero page once, and then
multiplications could be done three at a time. Why not just do that?

Well, the most basic answer is that I had already written this
routine before I noticed the multiplication trick. So I could have
rewritten it using the above method, or modify what was already in
place. Since CX CY and CZ are all 16-bit signed numbers, I chose
to do the multiplication savings using A11 etc. as the zero
page variable (A11*CX = A11*CXLO + 256*A11*CXHI), so the multiplications
are done two at a time. I think I also decided that the total cycle
savings in rewriting the routine would have been miniscule compared
with the total routine time, and that the whole global rotation time
was small compared with the local rotations and especially the
polygon plotting.

In sum, doing a total rewrite didn't seem to be worth the effort.

Nevertheless, this routine can certainly be made faster.
Another item for improvement is to do the division by 64 only
at the very end of the calculation (the routines below multiply,
then divide by 64, then add; it would be better to multiply, add,
then after all additions divide by 64). Who knows what I was
thinking at the time?!

Oh well, that's why God invented version numbers :).

Here is the main routine loop:

```
GLOOP    DEY
         STY COUNT

         [copy object centers to zero page for fast access]
```

This part multiplies row 1 of the rotation matrix times the position
vector, and stores the new result in CX, the x-coordinate of the object
center.

```
         LDA A11          ;Row1
         LDX B12
         LDY C13
         JSR MULTROW      ;Returns result in .X .A = lo hi
         LDY COUNT
         STA (CXHI),Y
         TXA
         STA (CXLO),Y

         [Do the same thing for row 2 and the y-coordinate]

         [Do the same thing for row 3 and the z-coordinate]
```

The routine then loops around back to GLOOP, until all points have
been rotated. The important work of this routine is done in MULTROW:

```
*
* Multiply a row by CX CY CZ
* (A Procedure to save at least a LITTLE memory...)
*
M1       EQU CXSGN        ;CXSGN etc. no longer used.
M2       EQU CYSGN
M3       EQU CZSGN

MULTROW
         STA M1
         STX M2
         STY M3
```

This next part checks to make sure A is not zero; as was pointed out
in the discussion of the fast multiply, A=0 will fail (ask me how I
know).

```
         TAY
         BEQ :SKIP1
```

It is all pretty straightforward from here: compute M1*CX, adjust for
signed values, and divide by 64 (this is the divide by 64 that it
would have been better to leave for later). Since zero-page is
already set up, the second multiplication is done with a quick multiply.

```
         LDY CX+1
         >>> MULTAY,TEMP2
         STA TEMP3
         LDY CX
         >>> QMULTAY,TEMP1
         CLC
         ADC TEMP2
         STA TEMP2
         LDA TEMP3
         ADC #00
         >>> DIVFIX,CX    ;Adjust result and /64
:SKIP1   STY TM1
         STA TM1+1
```

The next two parts just calculate M2*CY and M3*CZ, adding to TM1 and
TM1+1 as they go, and the routine exits.

```
ROTPROJ
```

Now we move on to perhaps THE core routine. In general, *a whole lot* of object points need to be rotated and projected, so this routine needs to blaze. Remember that there are two rotations in the "world" equation: local rotations to get an object pointing in the right direction, and another rotation to figure out what it looks like relative to us. This routine can therefore just rotate (for the first kind of rotation), or rotate and project (for the second kind). The idea is to pass in a list of points, tell it how many points to rotate (and project), and tell it where to store them all!

```
*
* ROTPROJ -- Perform local rotation and project points.
*
* Setup needs:
*   Rotation matrix
*   Pointer to math table (MATMULT = $AF-$B0)
*   Pointers to point list (P0X-P0Z = $69-$6E)
*   Pointers to final destinations (PLISTYLO ... = $BD...)
*     (Same as used by POLYFILL)
*   Pointers to lists of centers (CXLO CXHI... = $A3-$AE)
*   .Y = Number of points to rotate (0..N-1)
*   .X = Object center index (index to center of object)
*
* New addition:
*   C set means rotate and project, but C clear means just
*   rotate.  If C=0 then need pointers to rotation
*   destinations ROTPX,ROTPY,ROTPZ=$59-$5E.
*
```

Recall that the rotation matrix is offset by 64 -- instead of ranging from -1..1, all elements range from -64..64. As in the case of GLOBROT above, the final result needs to be divided by 64. Also recall the result from the discussion of signed multiplication: if one set of numbers is between -64..64, and the other between -96..95, then all combinations a+b (and a-b) generate a unique 8-bit number. In other words, only one table is needed.

Next, think about rotations for a moment: rotations do not change lengths (if you rotate, say, a pencil, it doesn't magically get longer). This has two important implications. The first is that we know a priori that the result of this rotation will be an *8-bit* result, if the starting length was an 8-bit number. Thus we need only one table, and just two table lookups -- one for f(a+b) and the other for f(a-b). Since the upper 8-bits are irrelevant, the divide by 64 can be incorporated directly into the table of f(x).

The second implication is that the _length_ of the point-vectors cannot be more than 128. This is because if we ignore the upper 8-bits, we don't capture the sign of the result. Consider: each x, y, and z coordinate of a point can be between -96..95. A point like (80,80,80) has length sqrt(3*80^2) = 138.56. So if some rotation were to rotate that vector onto the x-axis it would have a new coordinate of (138,0,0). Although this is an 8-bit number, it is not an eight-bit signed number -- the rotation could just as easily put the coordinate at (-138,0,0), which requires extra bits. So some care must be taken to make sure that object points are not more than 128 units away from the object center.

As was pointed out earlier, we can save a lot of time with the matrix multiply by doing the multiplication as column-times-element, instead of the usual row-times-column: column 1 of the matrix times Px plus column 2 times Py plus column 3 times Pz, where (Px,Py,Pz) is the vector being rotated. Here is the entire signed multiplication routine (with divide by 64):

```
SMULT    MAC              ;Signed multiplication
         STA MATMULT
         EOR #$FF
         CLC
         ADC #01
         STA AUXP
         LDA (MATMULT),Y
         SEC
         SBC (AUXP),Y
         <<<
```

And this is the Quick Multiply, used after the zero-page variables have been set up:

```
QMULT    MAC               ;Multiplication that assumes
         LDA (MATMULT),Y  ;pointers are already initialized
         SEC
         SBC (AUXP),Y
         <<<
```

Pretty cool, eh? A 12-cycle signed multiply and divide. :)

On to the routine:

ROTLOOP is the main loop. It first rotates the points. If the
carry was set then it adds the points to the object center and
projects them.

```
ROTLOOP  LDY COUNT
         BEQ UPRTS
         DEY
         STY COUNT

         LDA (P0X),Y
         BNE :C1
         STA TEMPX
         STA TEMPY
         STA TEMPZ
         BEQ :C2
:C1      LDY A11           ;Column 1
         >>> SMULT
         STA TEMPX
         LDY D21
         >>> QMULT
         STA TEMPY
         LDY G31
         >>> QMULT
         STA TEMPZ
```

The above multiplies the x-coordinate (P0X) times the first column of
the rotation matrix. Note the check for P0X=0, which would otherwise
cause the multiplication to fail. Note that there is one SMULT,
which sets up the zero-page pointers, followed by two QMULTS.
The result in TEMPX, TEMPY, TEMPZ will be added to subsequent
column multiplies. And that's the whole routine!

After the third column has been multiplied, we have the rotated
point. If projections are taking place, then the point needs to
be added to the center and projected. The centers are 16-bit
signed coordinates, but so far the points are just 8-bit signed
coordinates. To make them into 16-bit signed coordinates we
need to add a high byte of either 00, for positive numbers, or
$FF, for negative numbers (e.g. $FFFE = -2). In the code below,
.Y is used to hold the high byte.

```
ADDC     LDY #00
         CLC
         ADC TEMPZ
         BPL :ROTCHK
         DEY               ;Sign

:ROTCHK  LDX ROTFLAG
         BMI :POS1
         LDY COUNT         ;If just rotating, then just
         STA (ROTP0Z),Y    ;store!
         LDA TEMPY
         STA (ROTP0Y),Y
         LDA TEMPX
         STA (ROTP0X),Y
         JMP ROTLOOP

:POS1    CLC               ;Add in centers
         ADC CZ
         STA TEMPZ
         TYA
         ADC CZ+1
         STA TEMPZ+1       ;Assume this is positive!
```

Remember that only objects which are in front of us -- which have
positive z-coordinates -- are visible. The projection won't
work right if any z-coordinates are negative. Thus the above
piece of code doesn't care about the sign of the z-coordinate.

Two similar pieces of code add the x/y coordinate to the x/y center

coordinate. The signs of the result are stored in CXSGN and CYSGN,
for use below.

Recall how projection works: accuracy for small values of Z, but
speed for high values of Z. If the z-coordinate is larger than
8-bits, we just shift all the coordinates right until z is 8-bits.
Since the X and Y coordinates might be negative, we need the
sign bits CXSGN and CYSGN; they contain either 00 or $FF, so
that the rotations will come out correctly (negative numbers will
stay negative!).

```
         LDA TEMPZ+1
         BEQ PROJ
:BLAH    LSR             ;Shift everything until
         ROR TEMPZ       ;Z=8-bits
         LSR CXSGN
         ROR TEMPX+1     ;Projection thus loses accuracy
         ROR TEMPX       ;for far-away objects.
         LSR CYSGN
         ROR TEMPY+1     ;(Big whoop)
         ROR TEMPY
         TAX
         BNE :BLAH
```

Finally comes the projection itself. It just follows the equations
set out earlier, in Section 3. PROJTAB is the table of d/z.
It turns out that there are a lot of zeros and ones in the
table, so those two possibilities are treated as special cases.
The PROJTAB value is used in the zero-page multiplication pointers,
to again make multiplication faster. By calculating both the
x- and y-coordinates simultaneously, the pointers may be reused
even further.

After multiplying, the coordinates are added to the screen offsets
(XOFFSET and YOFFSET), so that 0,0 is at the center of the screen
(i.e. XOFFSET=160 and YOFFSET=100). These values are changeable,
so the screen center may be relocated almost anywhere. The final
coordinates are 16-bit signed values.

Finally, there is the other core routine: POLYFILL, the polygon
renderer.

POLYFILL

POLYFILL works much like the old Polygonamy routine. Each object
consists of a list of points. Each face of the object is some
subset of those points. POLYFILL polygons are defined by a set
of indices into the object point list -- better to copy an 8-bit
index than a 16-bit point. The index list must go counter-clockwise
around the polygon, so the hidden-face routine will work correctly.
To draw the polygon it simply starts at the bottom, calculating
the left and right edges and filling in-between.

POLYFILL also greatly improves upon the old routine. It can draw
to bitmaps in any bank. It is very compact. It deals with
16-bit signed coordinates, and can draw polygons which are partially
(or even wholly) off-screen. It also correctly draws polygons which
overlap (in a very sneaky way).

Keep in mind that this routine needs to be just balls-busting fast.
A typical object might have anywhere from 3-10 visible polygons, and
a typical polygon might be 100-200 lines high. Just saving a few
tens of cycles in the main loop can translate to tens of thousands
of cycles saved overall. By the same token, there's no need to
knock ourselves out on routines which aren't part of the main loop;
saving 100 cycles overall is chump change. Finally, since this is
a library, there are some memory constraints, and some of the routines
are subsequently less efficient than they would otherwise be. Again,
though, those few added cycles are all lost in the noise when compared
with the whole routine.

There is a whole a lot of code, so it's time to just dive in. First,

```
*
* Some tables
*

XCOLUMN                  ;xcolumn(x)=x/8
]BLAH    = 0
         LUP 32
         DFB ]BLAH,]BLAH,]BLAH,]BLAH
         DFB ]BLAH,]BLAH,]BLAH,]BLAH
]BLAH    = ]BLAH+1
         --^

* Below are EOR #$FF for merging into the bitmap, instead
* of just ORAing into the bitmap.

LBITP                      ;Left bit patterns
         LUP 32
* DFB $FF,$7F,$3F,$1F,$0F,$07,$03,$01
         DFB 00,$80,$C0,$E0,$F0,$F8,$FC,$FE
         --^
RBITP                      ;right bit patterns
         LUP 32
* DFB $80,$C0,$E0,$F0,$F8,$FC,$FE,$FF
         DFB $7F,$3F,$1F,$0F,$07,$03,$01,$00
         --^
```

I will put off discussion of the above tables until they are
actually used. It is just helpful to keep them in mind, for later.

Next up is the fill routine. The polygonamy fill routine looked like

```
        STA BITMAP,Y
        STA BITMAP+8,Y
        STA BITMAP+16,Y
        ...
```

There were a total of 25 of these -- one for each row. Then there
were two bitmaps, so 50 total. Each bitmap-blaster was page-aligned.

Fills take place between two columns on the screen. By entering
the above routine at the appropriate STA xxxx,Y, the routine
could begin filling from any column. By plopping an RTS onto
the appropriate STA the routine could exit on any column. After
filling between the left and right columns, the RTS was restored
to an STA xxxx,Y.

Although the filling is very fast, the routine is very large, locked
to a specific bitmap, and a bit cumbersome in terms of overhead. So
a new routine is needed, and here it is:

```
*
* The fill routine.  It just blasts into the bitmap,
* with self-modifying code determining the entry and
* exit points.
*
FILLMAIN
]BLAH    = 0               ;This assembles to
         LUP 32            ;LDY #00  STA (BPOINT),Y
         LDY #]BLAH        ;LDY #08  STA (BPOINT),Y
         STA (BPOINT),Y    ;...
]BLAH    = ]BLAH+8         ;LDY #248 STA (BPOINT),Y
         --^

         INC BPOINT+1      ;Advance pointer to column 32
COL32    LDY #00
         STA (BPOINT),Y
         LDY #08           ;33
         STA (BPOINT),Y
         LDY #16
         STA (BPOINT),Y
         LDY #24
         STA (BPOINT),Y
         LDY #32
         STA (BPOINT),Y
         LDY #40           ;37
         STA (BPOINT),Y
         LDY #48
         STA (BPOINT),Y
         LDY #56           ;Column 39
         STA (BPOINT),Y
FILLEND  RTS               ;64+256=320
FILL
         JMP FILLMAIN      ;166 bytes
```

As before, self-modifying code is used to determine the entry and
exit points. As you can see, compared with the polygonamy routine
this method takes 3 extra cycles per column filled. But now there
is just one routine, and it can draw to any bitmap. As an added

bonus, if we do the RTS just right, then (BPOINT),Y will point to
the ending-column (also note the INC BPOINT+1 in the fill code, for
when column 32 is crossed).

There are two parts to doing the filling. The above fill routine
fills 8 bits at a time. But the left and right endpoints need to
be handled specially. The old routine had to recalculate those
endpoints; this method gets it automatically, and so saves some
extra cycles in overhead.

In the very worst case, the old routine takes 200 cycles to fill
40 columns; the new routine takes 325 cycles. The main routine
below takes around 224 cycles per loop iteration. Ignoring the
savings in overhead (a few tens of cycles), this means that in
the absolute worst case the old method will be around 20% faster.
For typical polygons, they become comparable (5% or so). With all
the advantages the new routine offers, this is quite acceptable!

A few extra tables follow:

```
LOBROW   DFB 00,64,128,192,00,64,128,192
         (and so on)...

HIBROW
         DFB 00,01,02,03,05,06,07,08
         etc.

COLTAB                    ;coltab(x)=x*8
         DFB 0,8,16,24,32,40,48,56,64,72,80
         etc.
```

LOBROW and HIBROW are used to calculate the address of a row in
the bitmap, by adding to the bitmap base address. COLTAB then gives
the offest for a given column. Thus the address of any row,column
is quickly calculated with these tables.

Finally, the next two tables are used to calculate the entry and exit
points into the fill routine.

```
*
* Table of entry point offsets into fill routine.
* The idea is to enter on an LDY
*
FILLENT
         DFB 0,4,8,12,16,20,24,28,32,36,40 ;0-10
         DFB 44,48,52,56,60,64,68,72,76,80 ;11-20
         DFB 84,88,92,96,100,104,108,112   ;21-28
         DFB 116,120,124,128                ;29-32
         DFB 134          ;Skip over INC BPOINT+1
         DFB 138,142,146,150,154,158 ;34-39
         DFB 162          ;Remember that we use FILLENT+1
*
* Table of RTS points in fill routine.
* The idea is to rts after the NEXT LDY #xx so as to
* get the correct pointer to the rightmost column.
*
* Thus, these entries are just the above +2
*
FILLRTS
         DFB 2,6,10,14,18,22,26,30,34,38,42 ;0-10
         DFB 46,50,54,58,62,66,70,74,78,82  ;11-20
         DFB 86,90,94,98,102,106,110,114,118,122,126
         DFB 132          ;Skip over INC
         DFB 136,140,144,148,152,156,160    ;33-39
```

All that's left now is the main code routines. Before drawing the
polygon there is some setup stuff that needs to be done. The
lowest point needs to be found, and the hidden face check needs to
be done. The hidden face check is simply: if we can't draw this
polygon, then it is hidden!

As was said before, the list of point indices moves around the
polygon counter-clockwise when the polygon is visible. When the
polygon gets turned around, these points will go around the
polygon clockwise. What does this mean computationally? It
means that the right side of the polygon will be to the LEFT of
the left side, on the screen. A simple example:

```
         ____
         \  /
       L  \/  R
```

When the above polygon faces the other way (into the monitor), R will be to the left of L. The polygon routine always computes R as the "right side" and L as the "left side" of the polygon; if, after taking a few steps, the right side is to the left of the left side, then we know that the polygon is turned around. The hidden face calculation just computes the slopes of the left and right lines -- which we need to do to draw the polygon anyways -- and takes a single step along each line. All that is needed is to compare the left and right points (or the slopes), and either draw the polygon or punt.

There are two sides to be dealt with: the left and right. Each of those sides can have either a positive or negative slope, therefore a total of four routines are needed. A second set of four routines is used for parts of the polygon that have y-coordinates larger than 200 or less than zero, i.e. off of the visible screen; these routines calculate the left and right sides as normal, but skip the plotting calculations.

All four routines are similar. The differences in slope change some operations from addition to subtraction, and as was explained earlier affects whether the x=x+dx update comes before or after the plotting. It also affects the very last point plotted, similar to the way the x=x+dx update is affected. The truly motivated can figure out these differences, so I'll just go through one routine in detail:

```
* Left positive, right negative
*
* End: left and right advance normally

DYL3      >>> DYLINEL,UL3
LEFTPM    >>> LINELP,LEFTMM
          JMP LCOLPM

LOOP3     DEC YLEFT
          BEQ DYL3
UL3       >>> PUPDATE,LEFTXLO;LEFTXHI;LEFTREM;LDY;LDXINT;LDXREM
LCOLPM    >>> LCOLUMN

          DEC YRIGHT
          BNE UR3
          >>> DYLINER,UR3
RIGHTPM   >>> LINERM,RIGHTPP
          JMP RCOLPM
UR3       >>> MUPDATE,RIGHTXLO;RIGHTXHI;RIGHTREM;RDY;RDXINT;RDXREM
RCOLPM    >>> RCOLUMN
          >>> UPYRS,LOOP3
```

That's the whole routine, for a polygon which ends in a little peak, like /\ i.e. left line positive slope and right line negative. Let's read the main loop:

```
LOOP3   Decrease the number of remaining points to draw in the left line;
          if the line is finished, then calculate the next line.
        PUPDATE: The "P" is for "Plus": compute x=x+dx for the left side.
        LCOLUMN: Calculate the left endpoint on the screen, and set
                 up the screen pointer and the fill entry point.
        Decrease the number of remaining points in the right line;
          if finished, calculate the next line segment (note BNE).
        MUPDATE: "M"inus update, since slope is negative: calculate x=x-dx
        RCOLUMN: Calculate the right endpoint and fill column,
                 do the fill, and plot the left/right endpoints.
        UPYRS: Update the bitmap pointer, and go back to LOOP3
```

So it's just what's been said all along: calculate the left and right endpoints, and fill. When a line is all done, the next side of the polygon is computed. DYL3 performs the computations for the left line segment. DYLINEL computes the value of DY. Note that DY always has the same sign, since the polygon is always drawn from the bottom-up. LINELP computes DX and DX/DY, the inverse slope. If DX is negative it will jump to LEFTMM. Since we are in the "plus-minus" routine, i.e. left side = plus slope, right side = minus slope, then it needs to jump to the "minus-minus" routine if DX has the wrong sign. LEFTMM stands for "left side, minus-minus", in just the same way that LEFTPM in the above code means "left side, plus-minus". The right line segment is similar.

Note what happens next: the routine JMPs to LCOLPM. It skips over the PUPDATE at UL3. Similarly, the right line calculation JMPs over UR3 into RCOLPM. What this does is delay the calculation of x=x+dx or x=x-dx until *after* the plotting is done, to fill between

the correct left and right endpoints. See the discussion of polygon
fills in Section 3 for a more detailed explanation of what is going
on here.

Now let's go through the actual macros which make up
the above routine. First, the routines to calculate the slope of
the left line segment:

```
*
* Part 1: Compute dy
*
* Since the very last point must be plotted in a special
* way (else not be plotted at all), ]1=address of routine
* to handle last point.
*

DYLINEL  MAC               ;Line with left points
BEGIN    LDX LINDEX
L1       DEC REMPTS        ;Remaining lines to plot
         BPL L2            ;Zero is still plotted
         INC YLEFT
         LDA REMPTS
         CMP #$FF          ;Last point
         BCS ]1
EXIT     RTS
```

The above code first checks to see if any points are remaining.
If this is the last point, we still need to fill in the very last
points. If the right-hand-side routine has already dealt with
the last endpoint, REMPTS will equal $FF and the routine will really
exit; in this case, the last part has already been plotted.

```
L3       LDX NUMPTS
L2       LDY PQ,X
         STY TEMP1
         LDA (PLISTYLO),Y ;Remember, y -decreases-
         DEX
         BMI L3            ;Loop around if needed
         LDY PQ,X
         STY TEMP2
         SEC
         SBC (PLISTYLO),Y
         BEQ L1            ;If DY=0 then skip to next point
```

Next, it reads points out of the point queue (PQ) -- the list of point
indices going around the polygon. X contains LINDEX, the index
of the current point on the left-hand side. We then decrease this
index to get the next coordinate on the left-hand side; this index
is increased for right-hand side coordinates. That is, left lines
go clockwise through the point list, and right lines go counter-
clockwise. If DY=0, then we can just skip to the next point in
the point list -- the fill routine is very good at drawing horizontal
lines.

Sharp-eyed readers may have noticed that only the low-bytes
of the y-coordinates are used. We know ahead of time that DY is
limited to an eight-bit number, and since we are always moving up
the polygon we know that DY always has the same sign. In principle,
then, we don't need the high bytes at all. In practice, though,
the points can get screwed up (like if the polygon is very very
tiny, and one of the y-coordinates gets rounded down). The JSR FLOWCHK
below checks the high byte.

```
         STA LDY
         STA YLEFT
         STA DIVY
         STX LINDEX

         JSR FLOWCHK       ;Just in case, check for dy < 0
         BMI EXIT          ;(sometimes points get screwed up)
         <<<
```

The code for FLOWCHK is simply

```
FLOWCHK
         LDY TEMP1
         LDA (PLISTYHI),Y
         LDY TEMP2
         SBC (PLISTYHI),Y
         RTS
```

Why put such a tiny thing in a subroutine? Because I was out of
memory. It was very annoying. Remember the rule though: a few

occasional wasted cycles are a drop in the bucket. And, of course,
this is why God invented version numbers. (It can be made more
efficient, anyways -- as you might have guessed, this was kludged
in at the last moment, long after the routine had been written).

Okee dokee, the next part of computing the line slope is to compute DX.
For this routine, the left line has positive slope.

```
*
* Part 2: Compute dx.  If dx has negative the expected
* sign then jump to the complementary routine.
*
* dx>0 means forwards point is to the right of the
* current point, and vice-versa.
*

LINELP   MAC               ;Left line, dx>0
         LDY TEMP2
         LDA (PLISTXLO),Y ;Next point
         LDY TEMP1
         SEC               ;Carry can be clear if jumped to
         SBC (PLISTXLO),Y ;Current point
         STA DIVXLO
         LDY TEMP2
         LDA (PLISTXHI),Y
         LDY TEMP1
         SBC (PLISTXHI),Y
         BPL CONT1         ;If dx<0 then jump out
         JMP ]1            ;Entry address

CONT1    STA DIVXHI
         LDA (PLISTXLO),Y ;Current point
         STA LEFTXLO
         LDA (PLISTXHI),Y
         STA LEFTXHI

         JSR DIVXY         ;Returns int,rem = .X,.A
         JSR LINELP2
DONE2    <<<               ;Now .X=low byte, .A=high byte
                           ;of current point
```

Pretty short. It first computes DX, and jumps to the complementary
routine if DX has the wrong sign. Recall that DX can be 9-bits,
so both the low and high bytes of PLISTX are used. The current
line coordinate is stored in LEFTXLO/LEFTXHI, and DIVXY thencomputes
DX/DY, both the integer part and remainder part in fixed 16-bit format.
That is, the number returned is xxxxxxxx.xxxxxxxx i.e. 8 bit integer,
8 bit remainder (256*rem/DY).
        A few brief words should be said about DIVXY. It uses a
table of logs to compute an estimate for DX/DY. It then multiplies
that estimate by DY, and fixes up the estimate if it is too large
or too small. At this point, it has the integer part N and the
remainder R, i.e. DX/DY = N + R/DY. To compute R/DY it just uses
the log tables again but with a different exponential table,
since LOG(R) - LOG(DY) will always be *negative*. It then uses
this estimate as the actual remainder -- it doesn't try to correct
with a multiplication. By my calculations this can indeed cause a
tiny little error, but only for very very special cases (extremely
large lines/polygons), and it is something that is difficult to even
notice. I am always happy to trade tiny errors for extra cycles.
        After DIVXY is all done the subroutine LINELP2 is used --
again, it was moved out to conserve memory. Recall that the first
step needs to be of size dx/2. LINELP2 and siblings perform this
computation:

```
LINELP2
         STX LDXINT

         CMP #$FF          ;if dy=1
         BNE :NOT1         ;then A=dx/2
         LDA #00
         ROR               ;Get carry bit from dx/2
         STA LDXREM
         LDA #$80
         STA LEFTREM
         LDX LEFTXLO
         LDA LEFTXHI
         BCC :RTS          ;And start from current point
```

DIVXY sets .A to #$FF if DY=1 -- although you'd expect that $FF is
a perfectly valid remainder, it turns out that it doesn't happen

because of the way DIVXY computes remainders. If DY=1, then DX/DY is exactly DX. More importantly, this is the only case in which DX/DY might be 9-bits; any other DY will return an 8-bit value of DX/DY. For this reason it is handled specially.

```
:NOT1    STA LDXREM
         LDA #$80
         STA LEFTREM       ;Initialize remainder to 1/2
         TXA               ;x=x-dxdy/2
         LSR
         STA TEMP2
         LDA LEFTXLO
         TAX               ;.X = current point
         SEC
         SBC TEMP2
         STA LEFTXLO
         LDA LEFTXHI
         BCS :DONE
         DEC LEFTXHI
         TAY               ;Set/clear Z flag
:DONE    CLC
:RTS     RTS
```

The above probably looks a little strange. The slope is positive, so we want to calculate x=x+dx/2, but the above calculates x=x-dx/2. The reason is that the main loop will add dx to it, so that the next iteration will take it out to +dx/2. Why do it on the next iteration? For the same reason the x=x+dx update is delayed until after the plotting: for left lines with positive slope, we always need to start at the left endpoint of each horizontal line segment; in this case, that left endpoint is exactly the starting point. Notice that before computing x-dx/2 the above routine puts LEFTXLO/LEFTXI -- the starting point -- in .X/.A. The plot/fill calculations are done using the point in .X/.A and so the plot will be done from the starting point.

The above are the routines to calculate the slope of the line. They then jump over the next part: the part of the main loop which calculates x=x+dx.

```
*
* Next, the parts which update the X coordinates
* for positive and negative dx (in real space, of
* course -- the stored value of dx is always positive)
*
* Parameters passed in are:
*    ]1 = lo byte x coord
*    ]2 = high byte x coord
*    ]3 = remainder
*    ]4 = dy
*    ]5 = dx/dy, integer
*    ]6 = dx/dy, remainder
*

PUPDATE  MAC               ;dx>0
         LDA ]3
         CLC
         ADC ]6
         STA ]3
         LDA ]1            ;x=x+dx/dy
         ADC ]5
         TAX
         STA ]1
         BCC CONT
         INC ]2            ;High byte, x
         CLC
CONT     LDA ]2
         <<<               ;Carry is CLEAR on exit
                           ;.X = lo byte xcoord
                           ;.A = hi byte
```

As you can see, it is a very simple 16-bit addition. As it says, it leaves carry clear and .X/.A = lo/hi for the column calculation:

```
*
* Compute the columns and plot the endpoints
*
* .X contains the low byte of the x-coord
* .A was JUST loaded with the high byte
*
* Carry is assumed to be CLEAR
```

```
*

LCOLUMN  MAC
* LDA LEFTXHI
         BEQ CONT
         CMP #2
         BCC CONT1
         ASL
         BCS NEG

POS      LDA #$FF          ;Column flag
         STA LCOL
         BNE DONE

NEG      LDA #00           ;If negative, column=0
         STA LCOL
         STA LBITS         ;(will be EOR #$FF'ed)
         LDA #4            ;Start filling at column 1
         STA FILL+1
         BNE DONE
```

The routine first checks if the coordinate lies in one of the 40 columns on the screen, by checking the high byte. If the high byte is larger than one then the coordinate is way off the right side of the screen. If the left coordinate is past the right edge of the screen then there is nothing on-screen to fill, so LCOL is set to $FF as a flag. If the coordinate is negative then LCOL, the left column number, is set to 0, which will flag the plotting routine later on. LBITS will be explained later, and FILL is the entry point for the fill routine, consisting of a JMP xxxx instruction. Setting FILL+1 to 4 starts the fill routine at column one. Why not start at column zero, and let the fill routine take care of it? Because that requires a flag, and hence a check of that flag in the plotting routine. That flag check means a few extra cycles on each loop iteration, and actually screws up the plotting logic. For the relatively rare event of negative columns, this is totally not worth it.

There is still another branch above -- if the high byte is equal to one. If it is, then we need to check if the x-coordinate is less than 320. If it is, then the bitmap pointer and column entry point need to be set up correctly.

```
CONT1    CPX #64           ;Check X<320
         BCS POS
         LDA #32           ;Add 32 columns!
         INC BPOINT+1      ;and advance pointer
CONT     ADC XCOLUMN,X
         STA LCOL
         TAY

         LDA LBITP,X
         STA LBITS         ;For later use in RCOLUMN

         LDA FILLENT+1,Y  ;Get fill entry address
         STA FILL+1
DONE     <<<
```

By using the ADC XCOLUMN,X instead of the simple LDA XCOLUMN,X we can just add the extra 32 columns in very easily. Remember that the routine is entered with C clear, and A=0 if X<256. This gives the column number, 0-39 (the XCOLUMN table is way up above). A table lookup is cheaper than dividing by eight. LBITP is a table which gives the bit patterns for use in plotting the endpoints -- the part of the line not drawn by the fill routine. Plotting this endpoint is put off until later, because, among other things, the left and right endpoints might share a column (at the tips of the polygon, or for a very skinny polygon); plotting now would hose other things on the screen. Finally, FILLENT is used to set the correct entry point into the fill routine, FILL.

The right side of the polygon uses a similar set of routines to calculate slopes and do the line updating. The column calculation is similar at first, but does the actual plotting and filling:

```
*
* Note that RCOLUMN does the actual filling
*
* On entry, .X = lo byte x-coord, and .A was JUST loaded
*   with the high byte.
*
* Carry must be CLEAR.
*
```

```
RCOLUMN  MAC
* LDA RIGHTXHI
         BEQ CONT
         CMP #2
         BCC CONT1
         ASL
         BCS JDONE        ;Skip plot+fill if x<0

POS      LDX LCOL
         BMI JDONE        ;Skip if lcol>40!
         LDY YCOUNT       ;Plot the left edge
         LDA (FILLPAT),Y
         STA TEMP1
         LDY COLTAB,X     ;Get column index
         EOR (BPOINT),Y   ;Merge into the bitmap
         AND LBITS        ;bits EOR #$FF
         EOR TEMP1        ;voila!
         STA (BPOINT),Y
         LDA TEMP1
         JSR FILL         ;Just fill to last column
JDONE    JMP DONE
```

If the right column is negative, then the polygon is totally off the left side of the screen. If the right coordinate is off the right side of the screen, and the left column is still on the screen, then we just fill all the way to the edge. The left endpoints are then merged into the bitmap -- merging will be discussed, below.

```
UNDER    LDY LCOL         ;It is possible that, due to
         BMI DONE         ;rounding, etc. the left column
         LDX #00          ;got ahead of the right column
                          ;x=0 will force load of $FF
SAME     LDA RBITP,X      ;If zero, they share column
         EOR LBITS
         STA LBITS
         LDY YCOUNT
         LDA (FILLPAT),Y
         STA TEMP1
         LDX LCOL
         LDY COLTAB,X
         EOR (BPOINT),Y   ;Merge
         AND LBITS
         EOR TEMP1
         STA (BPOINT),Y   ;and plot
         JMP DONE

CONT1    CPX #64          ;Check for X>319
         BCS POS
         LDA #32          ;Add 32 columns!
CONT     ADC XCOLUMN,X
         CMP LCOL         ;Make sure we want to fill
         BCC UNDER
         BEQ SAME
```

After the right column is computed it is compared to the left column. If they share a column (or if things got screwed up and the right column is to the left of the left column), then the left and right sides need to be combined together and merged into the bitmap. Again, merging is described below. Otherwise, it progresses normally:

```
         LDY RBITP,X
         STY TEMP1
         LDX LCOL
         STA LCOL         ;Right column

         LDY YCOUNT       ;Plot the left edge
         LDA (FILLPAT),Y
         STA TEMP2
         LDY COLTAB,X     ;Get column index
         EOR (BPOINT),Y   ;Merge into bitmap
         AND LBITS
         EOR TEMP2
         STA (BPOINT),Y
```

it uses LCOL as temporary storage for the right column, sets the fill pattern, and then plots the left edge. Recall that LCOLUMN set up the bitmap pointer, BPOINT, if the x-coord was larger than 255. COLTAB gives the offset of a column within the bitmap, i.e. column*8. The pattern is loaded and the left endpoint is then merged into the bitmap.

What, exactly, is a 'merge'? Consider a line with LEFTX=5, i.e. that starts in the middle of a column. The left line calculation assumes that the left edge will be 00000111, and the fill takes care of everything to the right of that column. We can't just STA into the bitmap, because that would be bad news for overlapping or adjacent polygons. We also don't want to ORA -- remember that this is a pattern fill, and if that pattern has holes in it then this polygon will combine with whatever is behind it, and can again lead to poor results. What we really want to do is to have the pattern stored to the screen, without hosing nearby parts of the screen. A very crude and cumbersome way of doing this would be to do something like

```
        LDA BITP         ;00000111
        EOR #$FF
        AND (BITMAP)     ;mask off the screen
        STA blah
        LDA BITP
        AND pattern
        ORA blah
        STA (BITMAP)
```

That might be OK for some lame-o PC coder, but we're a little smarter than that I think. Surely this can be made more efficient? Of course it can, and stop calling me Shirley. It just requires a little bit of thinking, and the result is pretty nifty.

First we need to specify what the problem is. There are three elements: the BITP endpoint bits, the FILLPAT fill pattern bits, and the BPOINT bitmap screen bits. What we need is a function which does

```
        bitmask   pattern   bitmap   output
        -------   -------   ------   ------
           0         y        x        x
           1         y        x        y
```

In the above, 'y' represents a bit in the fill pattern and 'x' represents a bit on the screen. They might have values 0 or 1. If the bitmask (i.e. 00000111) has a 0 in it, then the bitmap bit should pass through. If the bitmask has a 1, then the *pattern* bit should pass through. A function which does this is

```
        (pattern EOR bitmap) AND (NOT bitmask) EOR pattern
```

This first tells us that the bitmask should be 11111000 instead of 00000111. This method only involves one bitmap access, and doesn't involve any temporary variables. In short, it merges one bit pattern into another, using a mask to tell which bits to change. So let's look at that bitmap merge code again:

```
         LDY YCOUNT
         LDA (FILLPAT),Y
         STA TEMP2
         LDY COLTAB,X     ;Get column index
         EOR (BPOINT),Y   ;Merge into bitmap
         AND LBITS
         EOR TEMP2
         STA (BPOINT),Y
```

And there it is. Pattern EOR bitmap AND not bitmask EOR pattern. Note that because the pattern is loaded first, .Y can be reused as an index into the bitmap, and X can stay right where it is (to be again used shortly). With these considerations, you can see that trying to mask off the bitmap would be very cumbersome indeed!

Note that if the left and right endpoints share a column, their two bitmasks need to be combined before performing the merge. This just means EORing the masks together -- remember that they are inverted, so AND won't work, and EOR is used instead of ORA in case bits overlap (try a few examples to get a feel for this).

Okay, let's remember where we are: the columns have all been computed, the fill entry points are computed, and the left endpoint has been plotted. We still have to do the fill and plot the right endpoint. LCOLUMN set the fill entry point, but RCOLUMN needs to set the exit point, by sticking an RTS at the right spot.

```
         LDY LCOL         ;right column
         LDX FILLRTS,Y    ;fill terminator
         LDA #$60         ;RTS
         STA FILLMAIN,X
         LDA TEMP2        ;Fill pattern
         JSR FILL
         EOR (BPOINT),Y
         AND TEMP1         ;Right bits
         EOR TEMP2         ;Merge
         STA (BPOINT),Y
         LDA #$91          ;STA (),Y
         STA FILLMAIN,X    ;Fill leaves .X unchanged
                           ;And leaves .Y with correct offset
DONE     <<<
```

The fill routine doesn't change X, leaves (BPOINT),Y pointing to the
last column, and leaves the fill pattern in A. The right side of the
line can then be immediately plotted after filling, and the STA (),Y
instruction can be immediately restored in the fill routine.
And that's the whole routine!
        After plotting, the loop just takes a step up the screen and
loops around. Since BPOINT, the bitmap pointer, may have been changed
(by LCOLUMN, or by the fill routine), the high byte needs to be restored.

```
*
* Finally, we need to decrease the Y coordinate and
* update the pointer if necessary.
*
* Also, since BPOINT may have been altered, we need
* to restore the high byte.
*
* ]1 = loop address
*
UPYRS    MAC
         LDA ROWHIGH
         STA BPOINT+1
         DEC BPOINT
         DEC YCOUNT
         BMI FIXIT
         JMP ]1

FIXIT    LDA #7
         STA YCOUNT
         LDY YROW
         BEQ EXIT          ;If y<0 then exit
         DEY
         STY YROW
         ORA LOBROW,Y
         STA BPOINT
         LDA HIBROW,Y
         CLC
         ADC BITMAP
         STA BPOINT+1
         STA ROWHIGH
         JMP ]1
EXIT     RTS
         <<<
```

Of course, the 64 bitmap is made up of 'rows of eight', i.e. after
every eight rows we have to reset the bitmap pointer. If we move
past the top of the screen (i.e. Y<0) then there is no more polygon
left to draw. Otherwise the new bitmap location is computed, by
adding the offset to BITMAP, the variable which lets the routine
draw to any bitmap (BITMAP contains the location in memory of the
bitmap). Note also that the high byte is stored to ROWHIGH --
ROWHIGH is what the routine immediately above uses to restore the
high byte of BPOINT!
        And that, friends, is the essence of the polygon routine,
and concludes the disassembly of the 3d library routines.

Section 4: Cool World

        Now that we have a nice set of routines for doing 3D graphics,
it's time to write a program which actually uses those routines to
construct a 3D world. That program is Cool World. Not only does it
demonstrate the 3d library, but it makes sure that all of the library
routines actually work!
        Although the algorithms for constructing a world were discussed
way back in Section 2, it's probably a good idea to very quickly
summarize those algorithms, just as a reminder. After that, this
section will go through the major portions of the Cool World code,
and explain how (and why) things work.

        In the 3d world, each object has certain properties. It has
a position in the world. It also has an orientation; it points in

some direction. It might also have some velocity, and other stuff,
but all we'll worry about in Cool World is position and orientation.
        Each object is defined by a series of points which are defined
relative to the object center (the object's position). The orientation
tells how to rotate those points about the center -- to get the object
rotating in the right direction, we just apply a rotation matrix to
the object.
        In Cool World, we will be the only thing moving around in
the world -- all the other objects are fixed in position, although
some of them are rotating. To figure out how objects look from our
perspective, we first figure out if the object is visible by using
the equation R'(Q-P). P is our position, and is subtracted from
the object position Q to get its relative position. R' then rotates
the world around us, to get it pointing in the direction of our
orientation vector. If the object is visible, then we compute the
full object by using the equation

        R' (M X + Q-P).

X represents the points of an object. M is the rotation matrix to
get the object pointing in the right direction. The rotated points
are then added to the relative center (Q-P) and rotated according to
our orientation. The points are then projected. The job of
Cool World and the 3D library is to implement that tiny little
equation up there.
        Once all the visible objects are rotated and projected,
they need to be depth-sorted to figure out what order to draw them
in. Each object is then drawn, in order, and around and around it
goes. With this in mind:

```
*
* Cool World (Polygonamy II: Monogonamy)
*
* Just yer basic realtime 3d virtual world program for
* the C64.  This program is intended to demonstrate the
* 3d library.
*
* SLJ 8/15/97
*
```

The code starts with a bunch of boring setup stuff, but soon gets
to the

```
* Main loop

MAINLOOP
```

which resets a few things, swaps the buffers, and clears the new draw
buffer. The screen clearing is done in a very brain-dead way; as
you may recall, polygonamy only cleared the parts of the screen that
needed clearing. I decided this time around that this was more trouble
than it was worth, especially with the starfields hanging around.
Still, a smart screen clear might keep track of the highest and
lowest points drawn, and only clear that part of the screen.
        Once the preliminaries are over with, the main part gets
going:

```
:C1      JSR READKEY
         JSR UPD8POS      ;Update position if necessary

         JSR UPD8ROT      ;Update second rotation matrix

         JSR PLOTSTAR

         JSR SORT         ;Construct visible object lst
```

First the keyboard is scanned. If a movement key is pressed, then the
stars are updated; if we turn, then our orientation vector is
updated via calls to ACCROTX, ACCROTY, and ACCROTZ.
        The orientation vector is very easy to keep track of.
We enter the world pointing straight down the z-axis, so initially
the orientation vector is V=(0,0,1). Now it gets a little trickier.
Let's say we turn to the left. We can think of this two ways:
our orientation vector rotates left, or the world rotates around
us to the right. The problem here is that these aren't the
same thing!
        "What?!" you may say. The problem here is not the single
rotation, but the rotations that come after it. Imagine a
little x-y-z coordinate axis coming out of your shirt, with
the z-axis pointing straight ahead of you. When you turn in
some direction, that coordinate system needs to turn with you,

because rotations are always about those axis. In other words,
the world needs to rotate around those axis; the *inverse* of
those rotations gives the orientation vector.

What would happen if we instead rotated the orientation vector?
Imagine a pencil coming out of your chest -- that's the orientation
vector. Now turn it to the left, with you still looking at the
monitor. Once it's out at 45 degrees or so, let's say we started
spinning around the z-axis. Well that z-axis is still facing
towards the monitor, and the pen rotates about THAT axis, drawing
a circle around the monitor as it goes around (or, if you like,
making a cone-shape with the tip of the cone at your chest).
If, on the other hand, YOU turn left, and then start spinning,
you can see that something very different happens -- the monitor
now circles around the pencil.

The point is that we need to rotate the world around us.
Specifically, by using ACCROTX and friends we maintain a rotation
matrix which will rotate the world around us. But we still need
an orientation vector, to tell us what direction to move in when
we move forwards or backwards. That rotation matrix tells us
how to rotate the whole world so that it "points in our direction";
therefore, if we un-do all of those rotations, we can figure out
what direction we were originally pointing in. In other words,
the inverse rotation matrix, when applied to the original orientation
vector (0,0,1), gives the current orientation vector.

Recall that inverting a rotation matrix is really easy:
just take the transpose. And what happens when apply this new
rotation matrix to V=(0,0,1)? With a simple calculation, you can
see that M V just gives the third row of M, for any matrix M.
So, let's say that we've calculated the accumulated rotation
matrix R'. To invert it just take the transpose, giving R.
The orientation vector is thus RV: the third row of R. But
the third row of R is just the third *column* of R', since R'
is the transpose of R. Therefore, the orientation vector -- the
direction we are currently pointing in -- is simply the third
column of the accumulated rotation matrix. We don't have to
do any extra work at all to compute that orientation vector; it
just falls right out of the calculation we've already done!

So, to summarize: Keys are read in. Any rotations
will cause an update of the rotation matrix. Our orientation
vector is simply the third column of that matrix. (Note that
the same logic holds for other objects, if they happen to move).
If we move forwards or backwards, then the stars are updated and
a flag is set for UPD8POS.

JSR UPD8POS

Truth in advertising: UPD8POS updates positions. Specifically,
our position, since we are the only ones moving. The world equation
is R' (MX + Q-P), which we can write as R'MX + R'(Q-P). Q is the
position of an object; changing our position P just changes Q-P. In
Cool World there's no need to keep track of our position in the world;
we might as well just keep track of the relative positions of objects.
Instead of updating a position P when we move, Cool World just
changes the locations Q of all the objects.

Since objects never move, the quantity R'(Q-P) won't change as
long as WE don't move. UPD8POS calculates that quantity, and does so
only if we move or rotate -- if an object were to move, this quantity
would have to be recalculated for that object. Note that the calculation
is done using GLOBROT, the lib3d routine to rotate the 16-bit centers.
Later on, those rotated 16-bit centers will be added to R'MX, the rotated
object points, if an object is visible.

Cool World lets you travel at different speeds by pressing
the keys 0-4. Different speeds are easy to implement. Recall that
the rotation matrix is multiplied by 64. Grabbing the third column
of that matrix gives a vector of length 64 (rotations don't change
lengths, so multiplying a rotation matrix times (0,0,1) must
always return a vector of the same length). By changing the length
of that orientation vector -- for example, by multiplying or dividing
by two -- we can change the amount which we subtract from the object
positions (or add to our position, if you like), and hence the speed
at which we move through the world.

JSR UPD8ROT

Since some of the objects need to rotate, a second set of
rotation matrices is maintained. As far as the lib3d routines are
concerned there is only one matrix, in zero page. Therefore the
global rotation matrix needs to be stored in memory before the
new matrices are calculated.

The new rotation matrices are calculated using CALCMAT,
the routine which only needs the x, y, and z angles. In fact,

only one matrix is calculated using this routine. The other
matrices are just permutations of that one rotation matrix; for
example, the transpose (the inverse, remember) is a new rotation
matrix. We can also get new matrices by, say, a cyclic permutation
of the rows -- this is the same as permuting the coordinates
(i.e. x->y y->z z->x), which is the same as rotating the whole
coordinate system. As long as the determinant is one, the new
matrix will preserve areas.

Once the new matrix is calculated, it is stored in memory
for later use by the local rotation routines (ROTPROJ). To get
a different matrix, all we have to do is copy the saved matrix
to zero page in a different way. That is, we don't have to store
ALL of the new rotation matrices in memory; we can do any row swapping
and such on the fly, when copying from storage into zero page.

Note that by using CALCMAT we can use large angle increments,
to make the objects appear to spin faster. ACCROTX and friends can
only accumulate by a fixed angle amount.

JSR PLOTSTAR

PLOTSTAR is another imaginatively named subroutine which,
get this, plots stars. The starfield routine is discussed elsewhere
in this issue.

JSR SORT

Finally, this routine -- you won't believe this -- sorts the
objects according to depth. The sort is just a simple insertion
sort -- the object z-coordinates are traversed, and each object
is inserted into an object display list. The list is searched
from the top down, and elements are bumped up the list until the
correct spot for the new object is found. The object list is thus
always sorted. This kind of sort is super-easy to implement in
assembly, and "easy" is a magic word in my book.

SORT also does the checking for visibility. As always, the
easiest and stupidest approach is taken: if the object lies in
a 45-degree wedge in front of us, then it is visible. This wedge
is easiest because the boundaries are so simple: the lines z=x, z=-x,
z=y, and z=-y. If the object center lies within those boundaries --
if -x < z < x and -y < z < y -- then the object is considered to
lie in our field of vision. Since we don't want to view objects
which are a million miles away, a distance restriction is put on
the z-coordinate as well: if z>8192 then the object is too far away.
We also don't want to look at objects which are too close -- in
particular, we don't want some object points in front of us and
some behind us -- so we need a minimum distance restriction as well.
Since the largest objects have points 95 units away from the center,
checking for z>100 is pretty reasonable.

By the way -- that "distance" check only checks the z-coordinate.
The actual distance^2 is x^2 + y^2 + z^2. This leads to the "peripheral
vision" problem you may have noticed -- that an object may become
visible towards the edge of the screen, but when you turn towards it
it disappears.

So, once the centers have been rotated, the extra rotation
matrix calculated, and the object list constructed, all that is
left to do is to plot any visible objects:

```
:LDRAW   LDY NUMVIS       ;Draw objects from object list
         BEQ :DONE
         LDA OBLIST-1,Y   ;Get object number
         ASL
         TAX
         LDA VISTAB,X
         STA :JSR+1
         LDA VISTAB+1,X
         STA :JSR+2
:JSR     JSR DRAWOB1
         DEC NUMVIS
         BNE :LDRAW
:DONE
         JMP MAINLOOP

VISTAB   DA DRAWOB1
         DA DRAWOB2
         ...
```

The object number from the object list is used as an index into
VISTAB, the table of objects, and each routine is called in order.
It turns out that in Cool World there are two kinds of objects:
those that rotate, and those that don't. What's the difference?

Recall the half of the world equation that we haven't yet calculated: R'MX. Objects that don't rotate have no rotation matrix M. And for objects that do rotate, I just skipped applying the global rotation R' (these objects are just spinning more or less randomly, so why bother). This becomes very apparent when viewing the Cobra Mk III -- no matter what angle you look at it from, it looks the same! A full implementation would of course have to apply both R' and M. Anyways, with this in mind, let's take a closer look at two representative objects: the center tetrahedron, which doesn't rotate, and one of the stars, which does.

```
*
* Object 1: A simple tetrahedron.
*
* (1,1,1) (1,-1,-1) (-1,1,-1) (-1,-1,1)

TETX     DFB 65,65,-65,-65
TETY     DFB 65,-65,65,-65
TETZ     DFB 65,-65,-65,65
```

These are the points that define the object, relative to its center. As you can see, they have length 65*sqrt(3) = 112. This is within the limitation discussed earlier, that all vectors must have length less than 128. Other objects -- for example, the cubes -- have coordinates like (55,55,55), giving length 55*sqrt(3) = 95.

```
* Point list

FACE1    DFB 0,1,2,0
FACE2    DFB 3,2,1,3
FACE3    DFB 3,0,2,3
FACE4    DFB 3,1,0,3
```

A tetrahedron has four faces, and the above will tell POLYFILL how to connect the dots. They go around the faces in counter-clockwise order. Note that the first and last point are the same; this is necessary because of the way POLYFILL works. When drawing from the point list, POLYFILL only considers points to the left or to the right. When it gets to one end of the list, it jumps to the other end of the list, and so always has a point to the left or to the right of the current spot in the point list. (Bottom line: make sure the first point in the list is also the last point).

```
TETCX    DA 0             ;Center at center of screen
TETCY    DA 00
TETCZ    DA INITOFF       ;and back a little ways.


OB1POS   DFB 00           ;Position in the point list
OB1CEN   DFB 00           ;Position of center
```

TETCX etc. are where the object starts out in the world. OB1POS is a leftover that isn't used. OB1CEN is the important variable here. The whole 3d library is structured around lists. All of the object centers are stored in a single list, and are first put there by an initialization routine. This list of object centers is then used for everything -- for rotations, for visibility checks, etc. OB1CEN tells where this object's center is located in the object list.

        This next part is the initialization routine, called when the program is first run:

```
*
* ]1, ]2, ]3 = object cx,cy,cz
*
INITOB   MAC
         LDA ]1
         STA C0X,X
         LDA ]1+1
         STA C0X+256,X
         LDA ]2
         STA C0Y,X
         LDA ]2+1
         STA C0Y+256,X
         LDA ]3
         STA C0Z,X
         LDA ]3+1
         STA C0Z+256,X
         <<<
```

```
INITOB1                    ;Install center into center list
         LDX NUMOBJS
         STX OB1CEN
         >>> INITOB,TETCX;TETCY;TETCZ
         INC NUMOBJS
         RTS
```

As you can see, all it does is copy the initial starting point, TETCX, TETCY, TETCZ, into the object list, and transfer NUMOBJS, the number of objects in the center list, into OB1CEN. It then increments the number of objects. In this way objects can be inserted into the list in any order, and a more or less arbitrary number of objects may be inserted. (Of course, REMOVING an object from the list can be a little trickier).

Now we move on to the main routine:

```
*
* Rotate and draw the first object.
*
* PLISTXLO etc. already point to correct offset.

* ]1,]2,]3 = object X,Y,Z point lists
SETPOINT MAC
         LDA #<]1          ;Set point pointers
         STA P0X
         LDA #>]1
         STA P0X+1
         LDA #<]2
         STA P0Y
         LDA #>]2
         STA P0Y+1
         LDA #<]3
         STA P0Z
         LDA #>]3
         STA P0Z+1
         <<<
```

SETPOINT simply sets up the point list pointers P0X P0Y and P0Z for the lib3d routines. It is a macro because all objects need to tell the routines where their actual x, y, and z point coordinates are located, so this little chunk of code is used quite a lot. The routine called by the main loop now begins:

```
DRAWOB1
         LDX OB1CEN        ;Center index
ROTTET                     ;Alternative entry point

         >>> SETPOINT,TETX;TETY;TETZ

         LDY #4            ;Four points to rotate
         SEC               ;rotate and project
         JSR ROTPROJ
```

The second entry point, ROTTET, is used by other objects which are tetrahedrons but not object #1 (in Cool World there is a line of tetrahedrons, behind our starting position). This of course means there won't be a lot of duplicated code.

Next the points need to be rotated and projected. SETPOINT sets up the list pointers for ROTPROJ, .Y is set to the number of object points in those lists, and C is set to tell ROTPROJ to rotate and project. If we were just rotating (C=clear, i.e. to calculate MX), we would have to set up some extra pointers to tell ROTPROJ where to store the rotated points; those rotated points can then be rotated and projected, as above. Note that the rotation matrix is assumed to already be set up in zero page. Once the points have been rotated and projected, all that remains is to...

```
* Draw the object

SETFILL  MAC
                           ;]1 = number of points
                           ;]2 = Face point list
                           ;]3 = pattern
         LDY #]1
L1       LDA ]2,Y
         STA PQ,Y
         DEY
         BPL L1
         LDA #<]3
         STA FILLPAT
         LDA #>]3
         STA FILLPAT+1
         LDX #]1
         <<<
```

Again we have another chunk of code which is used over and over and over. SETFILL just sets stuff up for POLYFILL, the polygon drawing routine. It first copies points into the point queue; specifically, it copies from lists like FACE1, above, into PQ, the point queue used by POLYFILL. It then sets FILLPAT, the fill-pattern pointer, to the 8 bytes of data describing the fill pattern. Finally it loads .X with the number of points in the point queue, for use by POLYFILL. All that remains is to call POLYFILL -- all the other pointers and such are set up, and POLYFILL does the hidden face check for us.

```
DRAWTET                 ;Yet another entry point

         >>> SETFILL,3;FACE1;DITHER1
         JSR POLYFILL

         >>> SETFILL,3;FACE2;ZIGZAG
         JSR POLYFILL

         >>> SETFILL,3;FACE3;CROSSSM
         JSR POLYFILL

         >>> SETFILL,3;FACE4;SOLID
         JMP POLYFILL
```

And that's the whole routine! DITHER1, ZIGZAG etc. are just little tables of data that I deleted out, containing fill patterns. For example, SOLID looks like

```
SOLID    HEX FFFFFFFFFFFFFFFF
```

i.e. eight bytes of solid color.

The next object, a star, is more involved. Not only is it rotating, but it is also a concave object, and requires clipping. That is, parts of this object can cover up other parts of the object. We will therefore have to draw the object in just the right way, so that parts which are behind other parts will be drawn first. Compare with the way in which we depth-sorted the entire object list, to make sure far-off objects are drawn before nearby objects. When applied to polygons, it is called polygon clipping.

Incidentally, I *didn't* do any clipping on the Cobra Mk III. Sometimes you will see what appear to be glitches as the ship rotates. This is actually the lack of clipping -- polygons which should be behind other polygons are being drawn in front of those polygons.

There are two types of stars in Cool World, but each is constructed in a similar way. The larger stars are done by starting with a cube, and then adding an extra point above the center of each face. That is, imagine that you grab the center of each face and pull outwards -- those little extrusions are form the tines of the star. The smaller stars are similarly constructed, starting from a tetrahedron. The large stars therefore have six tines, and the smaller stars have four.

Clipping these objects is really pretty easy, because each of those tines is a convex object. All we have to do is depth-sort the *tips* of each of the tines, and then draw the tines in the appropiate order.

Recall that in the discussion of objects, in Section 2, it was pointed out that any concave object may be split up into a group of convex objects. That is basically what we are doing here.

```
*
* All-Stars: A bunch of the cool star things.
*

STOFF    EQU 530           ;Offset unit

STCX     EQU -8000         ;Center
STCY     EQU 1200
STCZ     EQU -400+INITOFF

STAR1CX  DA STCX
STAR1CY  DA STCY
STAR1CZ  DA STCZ
```

```
OB7CEN   DFB 00

STAR1X   DFB 25,-25,25,-25,50,50,-50,-50
STAR1Y   DFB -25,-25,25,25,-50,50,50,-50
STAR1Z   DFB 25,-25,-25,25,-50,50,-50,50

S1T1F1   DFB 4,2,0,4       ;Star 1, Tine 1, face 1
S1T1F2   DFB 4,1,2,4
S1T1F3   DFB 4,0,1,4

S1T2F1   DFB 5,0,2,5
S1T2F2   DFB 5,3,0,5
S1T2F3   DFB 5,2,3,5

S1T3F1   DFB 6,2,1,6
S1T3F2   DFB 6,3,2,6
S1T3F3   DFB 6,1,3,6

S1T4F1   DFB 7,1,0,7
S1T4F2   DFB 7,0,3,7
S1T4F3   DFB 7,3,1,7

INITOB7
         LDX NUMOBJS
         STX OB7CEN
         >>> INITOB,STAR1CX;STAR1CY;STAR1CZ
         INC NUMOBJS
         RTS
```

As before, the above stuff defines the position, points, and
faces of the object, and INITOB7 inserts it into the center list.
The main routine then proceeds similarly:

```
DRAWOB7
         LDX OB7CEN       ;Center index

ROTSTAR1 JSR RFETCH       ;Use secondary rotation matrix

SETSTAR1 >>> SETPOINT,STAR1X;STAR1Y;STAR1Z

         LDY #8           ;Eight points to rotate
         SEC              ;rotate and project
         JSR ROTPROJ
```

Note the JSR RFETCH above. RFETCH retrieves one of the rotation
matrices from memory, and copies it into zero page for use by ROTPROJ.
Again, different rotation matrices may be constructed by performing
this copy in slightly different ways. The tines must then be
sorted, (and the accumulation matrix is restored to zero page),
and once sorted the tines are drawn in order. The code below
just goes through the sorted tine list, and calls :TINE1 through
:TINE4 based on the value in the list. The sorting routine will
be discussed shortly.

```
         JSR ST1SORT      ;Sort the tines
         JSR IFETCH       ;Restore accumulation matrix

* Draw the object.  In order to handle overlaps correctly,
* the tines must first be depth-sorted, above.
DRAWST1
         LDX #03
ST1LOOP  STX TEMP
         LDY T1LIST,X     ;Sorted tine list
         BEQ :TINE1
         DEY
         BNE :C1
         JMP :TINE2
:C1      DEY
         BNE :TINE4
         JMP :TINE3
:TINE4
         >>> SETFILL,3;S1T4F1;SOLID
         JSR POLYFILL
         >>> SETFILL,3;S1T4F2;DITHER1
         JSR POLYFILL
         >>> SETFILL,3;S1T4F3;DITHER2
         JSR POLYFILL
:NEXT    LDX TEMP
         DEX
         BPL ST1LOOP
         RTS
```

```
:TINE1  ...draw tine 1 in a similar way

:TINE2   >>> SETFILL,3;S1T2F1;ZIGS
        ...
:TINE3   >>> SETFILL,3;S1T3F1;BRICK
        ...


T1LIST   DS 6              ;Tine sort list
T1Z      DS 6
```

Now we get to the sorting routine. It depth-sorts the z-coordinates, so it actually needs the z-coordinates. Too bad we rotated and projected, and no longer have the rotated z-coordinates! So we have to actually calculate those guys.

Now, if you remember how rotations work, you know that the z-coordinate is given by the third row of the rotation matrix times the point being rotated -- in this case, those points are the tips of the individual tines. Well, we know those points are at (1,-1,-1), (1,1,1), (-1,1,-1), and (-1,-1,1). Actually, they are in the same *direction* as those points/vectors, but have a different length (i.e. STAR1X etc. defines a point like 50,-50,-50). The lengths don't matter though -- whether a star is big or small, the tines are still ordered in the same way. And order is all we care about here -- which tines are behind which.

So, calculating the third row times points like (1,-1,-1) is really easy. If that row has elements (M6 M7 M8), then the row times the vector just gives M6 - M7 - M8. So all it takes is an LDA and two SBC's to calculate the effective z-coordinate. Just for convenience I added 128 to the result, to make everything positive.

After calculating these z-coordinates, they are inserted into a little list. That list is then sorted. Yeah, an insertion sort would have been best here, I think. Instead, I used an ugly, unrolled bubble-like sort.

By the way, the cubic-stars are even easier to sort, since their tines have coordinates like (1,0,0), (0,1,0) etc. Calculating those z-coordinates requires an LDA, and nothing else!

```
*
* Sort the tines.  All that matters is the z-coord,
* thus we simply dot the third row of the matrix
* with the tine endpoint, add 128 for convenience,
* and figure out where stuff goes in the list.
*
ST1SORT                    ;Sort the tines
         LDX #00
         LDA MATRIX+6      ;Tine 1: 1,-1,-1
         SEC
         SBC MATRIX+7
         SEC
         SBC MATRIX+8
         EOR #$80          ;Add 128
         STA T1Z           ;z-coord
         STX T1LIST

         LDA MATRIX+6      ;Tine 2: 1,1,1
         CLC
         ADC MATRIX+7
         CLC
         ADC MATRIX+8
         EOR #$80
         STA T1Z+1
         INX
         STX T1LIST+1

         LDA MATRIX+7      ;Tine 3: -1,1,-1
         SEC
         SBC MATRIX+6
         SEC
         SBC MATRIX+8
         EOR #$80
         STA T1Z+2
         INX
         STX T1LIST+2

         LDA MATRIX+8      ;Tine 4: -1,-1,1
         SEC
         SBC MATRIX+7
         SEC
         SBC MATRIX+6
         EOR #$80
         STA T1Z+3
         INX
         STX T1LIST+3

         CMP T1Z+2        ;Now bubble-sort the list
         BCS :C1          ;largest values on top
         DEX              ;So find the largest value!

        ...urolled code removed for sensitive viewers.
```

And that's all it takes!

And that, friends, covers the main details of Cool World.

Can it possibly be true that this article is finally coming to an end?

Section 5: 3d library routines and memory map

There are seven routines:

```
$8A00   CALCMAT          - Calculate a rotation matrix
$8A03   ACCROTX          - Add a rotation to rotation matrix around x-axis
$8A06   ACCROTY          - ... y-axis
$8A09   ACCROTZ          - ... z-axis
$8A0C   GLOBROT          - 16-bit rotation for centers
$8A0F   ROTPROJ          - Rotate/Rotate and project objects
$8A12   POLYFILL         - Draw a pattern-filled polygon
```

Following a discussion of the routines there is a memory map and discussion of the various locations.

Library Routines

```
$8A00 CALCMAT   Calculate a rotation matrix

        Stuff to set up: Nothing.

        On entry: .X .Y .A = theta_x theta_y theta_z
        On exit: Rotation matrix is contained in $B1-$B9
        Cycle count: 390 cycles, worst case.


$8A03 ACCROTX   This will "accumulate" a rotation matrix by one rotation
                around the x-axis by the angle delta=2*pi/128.  Because
                this is such a small angle the fractional parts must also
                be remembered, in $4A-$52.  These routines are necessary
                to do full 3d rotations.

        On entry: carry clear/set to indicate positive/negative rotations.
        On exit: Updated matrix in $B1-$B9, $4A-$52
        Cycle count: Somewhere around 150, I think.

$8A06 ACCROTY   Similar around y-axis

$8A09 ACCROTZ   Similar around z-axis


$8A0C GLOBROT   Perform a global rotation of 16-bit signed coordinates
                (Rotate centers)

        Stuff to set up:
          MULTLO1 MULTLO2 MULTHI1 MULTHI2 = $F7-$FE
            Multiplication tables
          C0XLO, C0XHI, C0YLO, C0YHI, C0ZLO, C0ZHI = $63-$6E
            Pointers to points to be rotated.  Note also that $63-$6E
            has a habit of getting hosed by the other routines.
          CXLO, CXHI, CYLO, CYHI, CZLO, CZHI = $A3-$AE
            Pointers to where rotated points will be stored.

        On entry: .Y = number of points to rotate (0..Y-1).
        On exit: Rotated points are stored in CXLO CXHI etc.


$8A0F ROTPROJ   Rotate and project object points (vertices).  Points must
                be in range -96..95, and must be with 128 units of the
                object center.  Upon rotation, they are added to
                the object center, projected if C is set, and stored
                in the point list.

                The _length_ of a vertex vector (sqrt(x^2 + y^2 + z^2)) must
                be less than 128.

        Stuff to set up:
          Rotation matrix = $B1-$B9 (Call CALCMAT)
          ROTMATH = $B0
          P0X P0Y P0Z = $69-$6E
            Pointers to points to be rotated.  As with GLOBROT, these
            pointers will get clobbered by other routines.  Points are
            8-bit signed numbers in range -96..95, and must have
            length less than 128.
          PLISTXLO PLISTXHI ... = $BD-$C4
            Where to store the rotated+projected points.
          CXLO CXHI ... = $A3-$AE
            List of object centers.  Center will be added to rotated point
            before projection.
          XOFFSET, YOFFSET = $53, $54
            Location of origin on the screen.  Added to projected points
            before storing in PLIST.  160,100 gives center of screen.

        On entry: .X = Object center index (center is at CXLO,X CXHI,X etc.)
                  .Y = Number of points to rotate (0..Y-1).
                  .C = set for rotate and project, clear for just rotate
        On exit: Rotated, possibly projected points in PLIST.


$8A12 POLYFILL  Draw pattern-filled polygon into bitmap.

        Stuff to set up:
          MULTLO1, MULTLO2, MULTHI1, MULTHI2 = $F7-$FE
          BITMAP = $FF
          FILLPAT = $BB-$BC
          PLISTXLO, PLISTXHI, PLISTYLO, PLISTYHI = $BD-$C4
          Point queue = $0200.  Entries are _indices_ into the PLISTs,
            must be ordered _counter clockwise_, and must _close_ on
            itself (example: 4 1 2 4).

        On entry: .X = Number of points in point queue (LDX #3 in above
          example)
        On exit: Gorgeous looking polygon in bitmap at BITMAP.


Memory map
----------

Zero page:

$4A-$52 ACCREM          Fractional part of accumulating matrix

$53     XOFFSET         Location of origin on screen (e.g. 160,100)
$54     YOFFSET

$55-$72 Intermediate variables.  These locations are regularly hosed
        by the routines.  Feel free to use them, but keep in mind that
        you will lose them!

$A3-$AE C0XLO, C0XHI, C0YLO, ...
        Pointers to rotated object centers, i.e. where the objects are
        relative to you.  Centers are 16-bit signed (2's complement).

$AF-$B0 ROTMATH
        Pointer to the multiplication table at $C200-$C3FF.

$B1-$B9 Rotation matrix.

$BB-$BC FILLPAT
        Pointer to fill pattern (eight bytes, 8x8 character)

$BD-$C0 PLISTXLO, PLISTXHI
$C1-$C4 PLISTYLO, PLISTYHI
        Pointers to rotated object points, e.g. used by polygon renderer.
        Numbers are 16-bit 2's complement.

$F7-$FE MULTLO1, MULTLO2, MULTHI1, MULTHI2
        Pointers to math tables at $C400-$C9FF
        MULTLO1 -> $C500
        MULTLO2 -> $C400
        MULTHI1 -> $C800
        MULTHI2 -> $C700
        (Only the high bytes of the pointers are actually relevant).

$FF     BITMAP
        High byte of bitmap base address.
```

Library:

$0200-$027F	Point queue.
$8600-$9FFF	Routines, tables, etc.
$8A00	Jump table into routines

Tables:

$8400-$85FF	ROTMATH, pointed to by $AF-$B0.
$C000-$C2FF	MULTLO, pointed to by $F7-$F8 and $F9-$FA
$C300-$C5FF	MULTHI, pointed to by $FB-$FC and $FD-$FE
$C600-$C6FF	CDEL, table of f(x)=x*cos(delta)
$C700-$C7FF	CDELREM remainder
$C800-$C8FF	SDEL x*sin(delta)
$C900-$C9FF	SDELREM
$CA00-$CAFF	PROJTAB, projection table, f(x)=d/x, integer part
$CB00-$CBFF	PROJREM, projection table, 256*remainder
$CC00-$CCFF	LOG, logarithm table
$CD00-$CDFF	EXP, exponential table
$CE00-$CEFF	NEGEXP, exponential table
$CF00-$CFFF	TRIG, table of 32*sin(2*pi*x/128) -- 160 bytes.

ROTMATH is the Special 8-bit signed multiplication table for ROTPROJ. MULTLO and MULTHI are the usual fast-multiply tables. CDEL and SDEL are used by ACCROTX, to accumulate matrices. PROJTAB is used by ROTPROJ, to project the 16-bit coordinates. LOG EXP and NEGEXP are used by the POLYFILL divide routine, to calculate line slopes. Finally, TRIG is used by CALCMAT to calculate rotation matrices.

The tables from $C600-$CFFF are fixed and must not be moved. The tables ROTMATH, MULTLO, and MULTHI on the other hand are only accessed indirectly, and thus may be relocated. Thus there is room for a color map and eight sprite definitions in any VIC bank.

Section 6: Conclusions

Wow. You really made it this far. I am totally impressed. What stamina.

Well I hope you have been able to gain something from this article. It is my sincere hope that this will stimulate the development of some nice 3d programs. After a long, multi-year break from 3d graphics I really enjoyed figuring out all this stuff again, and finally doing the job right. I also enjoyed re-re-figuring it out, to write this article, since over half a year has passed since I wrote the 3d library! I hope you have enjoyed it too, and can use this knowledge (or improve on it!) to do some really nifty stuff.

SLJ 4/3/98

Section 7: Binaries

There are three files below. The first is the 3d library. The second is the tables from $C000-$CFFF. The third is the ROTMATH table ($8400-$85FF above).

:::::::: lib3d.o ::::::::

```
begin 644 lib3d.o
M`(8```````````$!`0$!`0$!`@("`@("`@(#`P,#`P,#`P0$!`0$!`0$!04%
M!04%!04&!@8&!@8&!@<'!P<'!P<'"`@("`@("`@)"0D)"0D)"0H*"@H*"@H*
M"PL+"PL+"PL,#`P,#`P,#`T-#0T-#0T-#@X.#@X.#@X/#P\/#P\/#Q`0$!`0
M$!`0$1$1$1$1$1$2$A(2$A(2$A,3$Q,3$Q,3%!04%!04%!05%145%145%186
M%A86%A86%Q<7%Q<7%Q<8&!@8&!@8&!D9&1D9&1D9&AH:&AH:&AH;&QL;&QL;
```

```
M&QP<'!P<'!P<'1T='1T='1T>'AX>'AX>'A\?'Q\?'Q\?`(#`X/#X_/X`@,#@
M\/C\_@"`P.#P^/S^`(#`X/#X_/X`@,#@\/C\_@"`P.#P^/S^`(#`X/#X_/X`
M@,#@\/C\_@"`P.#P^/S^`(#`X/#X_/X`@,#@\/C\_@"`P.#P^/S^`(#`X/#X
M_/X`@,#@\/C\_@"`P.#P^/S^`(#`X/#X_/X`@,#@\/C\_@"`P.#P^/S^`(#`
MX/#X_/X`@,#@\/C\_@"`P.#P^/S^`(#`X/#X_/X`@,#@\/C\_@"`P.#P^/S^
M`(#`X/#X_/X`@,#@\/C\_@"`P.#P^/S^`(#`X/#X_/X`@,#@\/C\_@"`P.#P
M^/S^`(#`X/#X_/X`@,#@\/C\_G\_'P\'`P$`?S\?#P<#`0!_/Q\/!P,!``'\_
M'P\'`P$`?S\?#P<#`0!_/Q\/!P,!``'\_'P\'`P$`?S\?#P<#`0!_/Q\/!P,!
M`'\_'P\'`P$`?S\?#P<#`0!_/Q\/!P,!``'\_'P\'`P$`?S\?#P<#`0!_/Q\/
M!P,!``'\_'P\'`P$`?S\?#P<#`0!_/Q\/!P,!``'\_'P\'`P$`?S\?#P<#`0!_
M/Q\/!P,!``'\_'P\'`P$`?S\?#P<#`0!_/Q\/!P,!``'\_'P\'`P$`?S\?#P<#
M`0!_/Q\/!P,!``'\_'P\'`P$`?S\?#P<#`0!_/Q\/!P,!``'\_'P\'`P$`?S\?
M#P<#`0"@`)%MH`B1;:`0D6V@&)%MH""1;:`HD6V@,)%MH#B1;:!`D6V@2)%M
MH%"1;:!8D6V@8)%MH&B1;:!PD6V@>)%MH("1;:"(D6V@D)%MH)B1;:"@D6V@
MJ)%MH+"1;:"XD6V@P)%MH,B1;:#0D6V@V)%MH."1;:#HD6V@\)%MH/B1;>9N
MH`"1;:`(D6V@$)%MH!B1;:`@D6V@*)%MH#"1;:`XD6U@3`")`$"`P`!`@,``
M0(#``$"`P`!`@,``0(#````!`@,%!@<("@L,#0\0$1(4%187&1H;'!X`"!`8
M("@P.$!(4%A@:'!X@(B0F*"HL+C`R-#8X.CP^``($!@@*#`X3#*:3"J;3%:;
M3(*;3`V<3*N=3&:*``0(#!`4&!P@)"@L,#0X/$!$2$Q05%A<8&1H;'!T>'R`
MAHJ.DI::GJ("!@H.$A8:'B(F*BXR-CH^0D9*3E)66EYB9FIN<G9Z?H2(C)"4
MF)R@AEB&6:D`A56%5H5:O``"L<'%5;'#,`[E5I`*AEJQP855L<.%5LK0Y*9:
MT`%@AEN\``*QO85DA6>QOX5EA6BE5496:D96:D96:H5KJJ55*0>%:AVFB85M
MO;^)&&7_A6Z%;Z9:QED0"^9>I5G)_["]8*98O``"A%6QP<HP]+P``H16./'!
MM\-V%7(5>A7*&6B#WF##<I%6QO85DI%8X\;V%<*15L;^%9:16\;\0`TSKBX5Q
M(*Z9(`"9(("+I%:QO:15./&]A7"D5K&_I%7QOQ`#3(*,A7&QO85GL;^%:""N
MF2!9F856I60XY6>%<*5EY6B%<:5P..5@L`/&<3CE8H5PI7'I`#`%!7#P`6"D
M:\`9L`6E5DQ?CJ@83)V5\"+)`I`6"K`&J?^%;-`FJ0"%;(57J02-I(G0&>!`
ML.FI(.9N?0"&A6RHO0"'A5>Y%HJ-I(FF6\99$`FE6<G^L"=@H@#D6+#ZO``"
MA%6QP>@XO``"A%;QP?#=A5V%7X5RAEL@]Y@PVF!H:&"D5K&]I%4X\;V%<*16
ML;^D5?&_$`-,^8J%<;&]A62QOX5E(*Z9("F9(("+I%:QO:15./&]A7"D5K&_
MI%7QOQ`#3.J,A7&QO85GL;^%:""NF2!9F856I60XY6>%<*5EY6B%<:5P&&5@
MD`+F<3CE8H5PI7'I`#`0!7#0"Z5A..5CI6#E8I`!8*1KP!FP!:563$60I588
M3+*6I%6QO85GI%8X\;V%<*15L;^%:*16\;\0`TSYBH5Q(*Z9((&9A5:E9#CE
M9X5PI67E:(5QI7`XY6"P`L9Q&&5BA7"E<6D`,!`%<-`+I6,XY6&E8N5@D`%@
MI&O`&;`%I59,^Y.E5AA,SYBD5;&]A6>D5CCQO85PI%6QOX5HI%;QOQ`#3.J,
MA7$@KID@@9F%5J5D.5GA7"E9>5HA7&E<!AE8)`#YG$896*E<6D`,`%@I&O`
M&;`%I59,/F%5J5D..5GA7"E9>5HA7&E<!AE8)`#YG$896*E<6D`,`%@I&O`
M&;`%I59,`F4RTC6#&7O"BI68XY6&%9J5DY6"%9*JP`L9EI648\"+)`I`6
M"K`&J?^%;-`FJ0"%;(57J02-I(G0&>!`L.FI(.9N?0"&A6RHO0"'A5>Y%HJ-
MI(G&7]!=IEO&61`)I5G)_K"A8*(`Y%BP^KP``H15L<'H.+P``H16\<'PW85=
MA5^%<H9;(/>8,-JD5K&]I%4X\;V%<*16L;^D5?&_$`-,OY.%<<;&]A6>QOX5H
M(*Z9(%F93%^.I6D896.%::5G96*JA6>0`^9H&*5E\$W)`I!#%:5J5D.5GA7"E
M9>5HA7&E<!AE8)`#YG$896*E<6D`,`%@I&O`&;`%I59,;N%5J5D..5GA7"E
M9>5HA7&E<!AE8)`#YG$896*E<6D`,`%@I&O`&;`%I59,,CH_&7O"?I688
M M96&%9J5D96"JA620`^9E&*5E\"+)`I`6"K`&J?^%;-`FJ0"%;(57J02-I(G0
M&>!`L.FI(.9N?0"&A6RHO0"'A5>Y%HJ-I(G&7]!IIEO&61`)I5G)_K!18*(`
MY%BP^KP``H15L<'H.+P``H16\<'PW85=A5^%<H9;(/>8,-JD5K&]I%4X\;V%
M<*16L;^D5?&_$`-,WY&%<;&]A6>QOX5H(*Z9(%F93$600``"L;VJL;\83$60
MI6D896.%::5G96*JA6>0`^9H&*5E\$W)`I!#%:5J5D.5GA7"E9>5HA7&E<!AE8
M)5=%59%MI54@HXE,TY"D;#!DH@"]`(A%5X57I&JQNX55IFR\V(E1;2571561
M;4S3D.!`L+RI('T`AL5LD-#PU+P`B(15IFR%;*1JL;N%5KS8B5%M)5=%5I%M
MI&R^/HJI8)T`B:56(*.)46TE5456D6VID9T`B:5OA6[&;<9J,`-,=8^I!X5J
MI&OP%8B$:QFFB85MN;^)&&7_A6Z%;TQUCV"F6L99$`OF7J59R?^P4V"F6+P`
MM`H15L<'*,/2\``*$5CCQO85PI%:QOX5HI%;QOQ`#3"D5K&]I%4X\;V%<*16
MMOZ15\;\0`TP4DX5QL;V%9+&_I%;QOQ`#3*4X\;V%<*16L;^D5?&_$`-,9%"J
MMA620`^9E&*5E\"+)`I`6"K`&J?^%;-`FJ0"%;(57J02-I(G0&>!`L.FI(.9N
M?0"&A6RHO0"'A5>Y%HJ-I(G&7]!9IEO&61`)I5G)_K!-8*(`Y%BP^KP``H15
ML<'H.+P``H16\<'PW85=A5^%<H9;(/>8,-JD5;&]A6>D5CCQO85PI%6QOX5H
MI%;QOQ`#3/F/A7$@KID@@9E,&Y*E:3CE8X5II6?E8H5GJK`"QFBE:!CP3<D"
MD$,*L!JF;#`6I&JQNX55O-B)46TE5T55D6VE52"CB4RIDJ1L,&2B`+T`B$57
MA5>D:K&[A55IFR\V(E1;2571561;4S3*F2X$"PO*D@?0"&Q6R0T/#4O``"(A%6F
M;(5LI&JQNX56O-B)46TE5T56D6VD;+Q+X^BJE@G0")I58@HXE1;2551515:1;:1
M MG0")I6^%;L9L9MQFHP`TQ;D:D'A6JD:_`5B(1K&::)A6VYO7D89?^%;H5O3%N1
M8.9?O``"L;VJL;\83$60I6D896.%::5G96*JA6>0`^9H&*5E\$W)`I!#%:5J5D
MA%8X\<'PW85<A5Z%<H9;(/>8,-JD5;&]A6>D5CCQO85EI%6QOX5HI%;QOQ`#
M3#"1A7$@KID@`)E,5)/&7O"CI68XY6&%9J5DY6"%9*JP`L9EI648\"+)`I`6
M"K`&J?^%;-`FJ0"%;(57J02-I(G0&>!`L.FI(.9N?0"&A6RHO0"'A5>Y%HJ-
MI(G&7]!9IEO&61`)I5G)_K!-8*(`Y%BP^KP``H15L<'H.+P``H16\<'PW85=
MA5^%<H9;(/>8,-JD5;&]A6>D5CCQO85HI%6QOX5HI%;QOQ`#3!^.A7$@KID@
M@9E,^Y.E:3CE8X5II6?E8H5GJK`"QFBE:!CP3<D"D$,*L!JF;#`6I&JQNX55
MO-B)46TE5T55D6VE52"CB4R)E*1L,&2B`+T`B$57A5>D:K&[A55IFR\V(E1;2
M)5=%59%M3(F4X$"PO*D@?0"&Q6R0T/#4O``"(A%6F;(5LI&JQNX56O-B)46TE
M5T56D6VD;+Q+X^BJE@G0")I58@HXE1;2551515:1;:1F1G0")I6^%;L9L9MQFHP[
MDZD'A6JD:_`5B(1K&::)A6VYOXD89?^%;H5O3#N38,9>T%JF6L99$`OF7J59
MR?^P2V"F6+P``H15L<'*,/2\``*$5CCQO85PI%6QOX5HI%;QOQ`#3*15L;V%
M9*16./&]A7"D5;&_A6BD5O&_$`-,^Y6%<;&]A62QOX5E(*Z9(%F9CCE885FI63E
```

```
M8(5DJK`"QF6E91C&7]!=IEO&61`)I5G)_K#:8*(`Y%BP^KP``H15L<'H.+P`
M`H16\<'PW85=A5^%<H9;(/>8,-JD5K&]I%4X\;V%<*16L;^D5?&_$`-,DYB%
M<;&]A6>QOX5H(*Z9(%F93)V5I6D896.%::5G96*JA6>0`^9H&*5HQFHP`TRT
ME*D'A6K&:Z1K&::)A6VYOXD89?^%;H5OP!CP`TRTE$R;C<9>T%VF6L99$`OF
M7J59R?^P3V"F6+P``H15L<'*,/2\``*$5CCQP?#=A5R%7H5RAEH@]Y@PW*16
ML;VD53CQO85PI%:QOZ15\;\0`TSJE(5QL;V%9+&_A64@KID@*9E,.Y:E9AAE
M885FI61E8*J%9)`#YF48I67&7]!>IEO&61`)I5G)_K!18*(`Y%BP^KP``H15
ML<'H.+P``H16\<'PW85=A5^%<H9;(/>8,-JD5K&]I%4X\;V%<*16L;^D5?&_
M$`-,AI>%<;&]A6>QOX5H(*Z9(%F93+*68*5I&&5CA6FE9V5BJH5GD`/F:!BE
M:,9J,`-,Q96I!X5JQFND:QFFB85MN;^)&&7_A6Z%;\`8\`-,Q95,=8_&7M!=
MIEK&61`+YEZE6<G_L+!@IEB\``*$5;'!RC#TO``"A%8X\<'PW85<A5Z%<H9:
M(/>8,-RD5K&]I%4X\;V%<*16L;^D5?&_$`-,(9B%<;&]A62QOX5E(*Z9("F9
M3%"7I68896&%9J5D96"JA620`^9E&*5EQE_06:9;QED0":59R?ZP36"B`.18
ML/J\``*$5;'!Z#B\``*$5O'!\-V%785?A7*&6R#WF#"<I7J;JHF@I(*TP'#&
ML7XP)CQ0Z\XSO?P!Q#B]6>^#?1HRT+'_EEIJ6H0)86WC#BTW>7B[CW&45HI.
MI6$E?#XF4B:#:Z4(#&OT>,Z0Z\#GE4!E4AN8+U)4\;[XTM-3D2XP=R+/+K0
M8\JRT@Y2L;$K:8X-@:>68&$`I%@TF4TQ-8#)5T<%ZQ<]06_U2B$4J60T<U6
MI'6AH@+VZ7=9]:CG>B<2EG]VQ*LHLJ_]Q@;FX2P1\X)YI'*9L]QE=R'=']\3
M2:ULK:W)L4DDWKC?(@>!BMC0:/B[X"@A;):98GN;EC8S4=E=/4T"=:7,U5(U
MI<NWZM0&)'/>FT+ABTJ&4[,'*.S('O/Z/<D/^/[YH%M/.%SQ^XJ4+/HL9V,,
M+FC,Z?8*%(%C\K>%;6=BNB/7!2WXO,^O4UO,4,)SI<C2CF/VSAQ@[9ZG&V1\
M/#9^S)5+.'2CKJ80N^YFG]\W@@0B^1?,!/#K-&-:WQNNYZ5XC(!+%2ZE5XH^
M]R1T:#%Z6&&,5]:LAB/X@*QK*#@PWBP48!!B>=.EYP$XF3P5W:KCN<8)Z6:Y
ML[@$L>3@$8IE]94:00`F+#,LH/(\'3.&4NE:(=*+Q*J?H&X(L`'3Y`K
MAB^;#E)K'$)'"*N5WZ'I/T"D*L-X5EPB:?X0'3VNBFI:,GEEB_)N?
```

```
M8(5DJK`"QF6E91C&7]!=IEO&61`)I5G)_K#:8*(`Y%BP^KP``H15L<'H.+P`
M`H16\<'PW85=A5^%<H9;(/>8,-JD5K&]I%4X\;V%<*16L;^D5?&_$`-,DYB%
M<;&]A6>QOX5H(*Z9(%F93)V5I6D896.%::5G96*JA6>0`^9H&*5HQFHP`TRT
ME*D'A6K&:Z1K&::)A6VYOXD89?^%;H5OP!CP`TRTE$R;C<9>T%VF6L99$`OF
M7J59R?^P3V"F6+P``H15L<'*,/2\``*$5CCQP?#=A5R%7H5RAEH@]Y@PW*16
ML;VD53CQO85PI%:QOZ15\;\0`TSJE(5QL;V%9+&_A64@KID@*9E,.Y:E9AAE
M885FI61E8*J%9)`#YF48I67&7]!>IEO&61`)I5G)_K!18*(`Y%BP^KP``H15
ML<'H.+P``H16\<'PW85=A5^%<H9;(/>8,-JD5K&]I%4X\;V%<*16L;^D5?&_
M$`-,AI>%<;&]A6>QOX5H(*Z9(%F93+*68*5I&&5CA6FE9V5BJH5GD`/F:!BE
M:,9J,`-,Q96I!X5JQFND:QFFB85MN;^)&&7_A6Z%;\`8\`-,Q95,=8_&7M!=
MIEK&61`+YEZE6<G_L+!@IEB\``*$5;'!RC#TO``"A%8X\<'PW85<A5Z%<H9:
M(/>8,-RD5K&]I%4X\;V%<*16L;^D5?&_$`-,(9B%<;&]A62QOX5E(*Z9("F9
M3%"7I68896&%9J5D96"JA620`^9E&*5EQE_06:9;QED0":59R?ZP36"B`.18
ML/J\``*$5;'!Z#B\``*$5O'!\-V%785?A7*&6R#WF#"<I7J;JHF@I(*TP'#&
MJ3;$4N%P`)`ALD36:?R0)+E.Y'H1J$#8<0JD/MET$*Q)YH0BP6``H$'BA";)
M;!"T6?ZD2O&80.B1.N2..>20/.F61/*A4`"P81+$=BG<D$3YKF0:T8A`^+%J
M)-Z95!#,B48$PH%``,"!0@3&B4P0U)E>).JQ>$`(T9ID+OG$D%PI]L2283``
MT*%R1!;IO)!D.0[DNI%H0!CQRJ1^6300[,FFA&)!(`#@P:*$9DDL$/39OJ2*
M<5A`*!'ZY,ZYI)!\:59$,B$0`/#ATL2VJ9R0A'EN9%I12$`X,2HD'AD4$`P)
M!@0"`0!`/S\^/CT]/#P[.SHZ.3DX.#<W-C8U-34T-#,S,C(Q,3$P,"\O+BXM
M+2TL+"LK*RHJ*2DI*"@G)R<F)B4E)20D)",C(B(B(2$A("`?'Q\>'AX='1T<
M'!P;&QL:&AH9&1D9&!@8%Q<7%A86%145%104%!,3$Q,2$A(2$1$1$1`0$!`/
M#P\/#@X.#@T-#0T,#`P,#`L+"PL*"@H*"@D)"0D)"0@("`@(!P<'!P<'!@8&
M!@8&!04%!04%!00$!`0$!`0$`P,#`P,#`P,"`@("`@("`@("`0$!`0$!`0$!
M`0$!`0$`````````````````````````````````````````````````````
M```````````````````````````````!`0$!`0$!`0$!`0$!`0("`@("`@("
M`@(#`P,#`P,#`P0$!`0$!`0$!04%!04%!08&!@8&!@<'!P<'!P@("`@("0D)
M"0D)"@H*"@H+"PL+#`P,#`P-#0T-#@X.#@\/#P\0$!`0$1$1$1(2$A(3$Q,3
M%!04%145%186%A<7%Q@8&!D9&1D:&AH;&QL<'!P='1T>'AX?'Q\@("$A(2(B
M(B,C)"0D)24E)B8G)R<H*"DI*2HJ*RLK+"PM+2TN+B\O,#`Q,3$R,C,S-#0U
M-34V-C<W.#@Y.3HZ.SL\/#T]/CX_/T!`04%"0D-#1$1%149&1T=(2$E)2DI+
M3$Q-34Y.3T]045%24E-35%155E975UA965I:6UQ<75U>7U]@8&%B8F-D9&5E
M9F=G:&EI:FIK;&QM;FYO<'!Q<G)S='1U=G9W>'EY>GM[?'U]?G]_@(&"@H.$
MA(6&AX>(B8J*BXR-C8Z/D)"1DI.3E)66EI>8F9F:FYR=G9Z?H*"AHJ.DI*6F
MIZBIJ:JKK*VMKJ^PL;*RL[2UMK>WN+FZN[R]O;Z_P,'"P\3$Q<;'R,G*R\O,
MS<[/T-'2T]34U=;7V-G:V]S=WM_@X>'BX^3EYN?HZ>KK[.WN[_#Q\O/T]?;W
M^/GZ^_S]_O\```$"`P0%!@<("0H+#`T.#Q`1$A,4%187&!D:&QP='1X?("(C
M)"4F)R@I*BLL+2XO,#$R,S0U-C<X.3H[/#T^/T!!0D-$149'2$E*2TQ-3D]0
M45)35%565UA96EM<75Y?8&%B8V1E9F=H:6IK;&UN;W!Q<G-T=79W>'EZ>WQ]
M?H"!@H.$A8:'B(F*BXR-CH^0D9*3E)66EY*8F9J;G)V>GZ"AHJ.DI::GJ*FJ
MK*VNK[&RM+6WN+FZN[R]OK_`P<+#Q,7&Q\C)RLO,S<[/T-'2T]35UM?8V=K;
MV=K;W-W>W^#AXN/DY>;GZ.GJZ^SM[N_P\?+S]/7V]_CY^OO\_?[_`/____[^
M_OW]_?S\_/O[^_OZ^OKY^?GX^/CW]_?V]O;U]?7T]/3S\\_/R\O+Q\?'P\.#P
M\._O[^_N[N[M[>WL[.SKZ^OJZNKIZ>GHZ.CGY^?FYN;EY>7DY./CX^+BXN'A
MXN+AX>'@X.#?W]_>WM[=W=W<V]O;V=C7UM;5U=34T]33TM+2T='1T(R8F)24E
M(R,B(B(A(2$@("`@'Q\?'AX>'1T='!P<&QL;&AH:&1D9&!@8&!@7%Q<6%A85
M%144%!04$Q,3$Q(2$A(1$1$1`0$!`/#P\/#@X.#@T-#0T-#`P,#`P,"`@("`@(
M!P<'!@8&!04%!`0$!`,#`P("`@$!`0````````````````````````````````
M``$!`0$!`0$!`0$!`0$!`0$!`@("`@("`@("`@("`@("`@("`@("`@("`P,#
M`P,#`P,#`P,#`P,$!`0$!`0$!`0$!`0$!`0$!`0$!`4%!04%!04%!04%!04%
M!04%!04%!04%!04%!08&!@8&^?GY^?GY^OKZ^OKZ^OKZ^OKZ^OKZ^OKZ^OKZ
M^_O[^_O[^_O[^_O[^_O[^_O\_/S\_/S\_/S\_/S\_/S\_/W]_?W]
M_?W]_?W]_?W]_?W]_?[^_O[^_O[^_O[^_O[^_O[^____________
M______________\`#!DE,CY+5V1Q?8J6HZ^\R-7B[OL'%"`M.D937VQXA9&>
MJ[?$T-WI]@,/'"@U04Y:9W2`C9FFLK_+V.7Q_@H7(S`]259B;WN(E*&NNL?3
MX.SY!A(?*SA$45UJ=X.0G*FUPL_;Z/0!#1HF,T!,665R?HN7I+&]RM;C[_P)
M%2(N.[C$T=WJ]@,0'"DU0DY;:'2!C9JFL[_,V.7R_@L7)#`]2E9C;WR(E:*N
N\?4X.WY!A,?+#A%45YK=X20G:JVP\_<Z/4!#AHG-$!-669R?XN8I,&]RM;D
M\/P)%B(O.TA486YZAYN@K+G%TM_K^`01'2HW0U!<:7:"CIN@M,?/:,S`-(S`)I
M1C0J(QX:%Q43$1`/#@T,"PL*"@D)"0@("`@'!P<'!P<'!@8&!@8&!@8&!@8&
M!@8&!0#`P,#`P,#`P("`@("`@("`@("`@("`@("`@("`@("`@("`@("`@("
M`@("`@("`@("`@$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!
M`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!
M`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0``````````````````````
M`````````````````````````````````````(````"`````0%4`%X`G```@
M6JL-@`",(<!F%,>`/@#&D%TM`-6MAV)`'P#BQJN1>&!),QX*]N31P*^?CX!Q
M8U5(.R\B%PL`]>O@ULW#NK&IH)B0B(!X<6IC7%5/2$(\-J)1\:%`\*!0#[
M]O+MZ>3@W-C3S\O(Q,"\N;6RKJNGI*&>FI>4D8Z+B(:#@'U[>'5S<&YK:69D
M8E]=6UE65%)03DQ*2$9$0D`^/#HY-S4S,3`N+"LI)R8D(R$?'AP;&1@7%A02
M$1`.#0L*"0@'!@0#`@$`__OS[^OGX]_7T\_+Q\.\N[>SKZNGHY^;EY./BX>#?
MWMW<V]O:V=C7UM74U-,``!\R/TI265]E:6YR=GE\?X*%AXF,CI"2E)67F9J<
MGI^@HJ.DIJ>HJ:JLK:ZOL+&RL[2TM;:WN+FZNKN\O;V^O\#`P<+#P\3$Q<;&
MQ\?(R<G*RLO+S,S-S<[.S]#0T='2TM/3T]34U=76U];7U]C8V=G9VMK;V]O<
MW-W=W=[>WM_?W^#@X.'AX>+BXN/CX^3DY.7EY>;FYN?GY^CHZ.GIZ>KJZNOK
MZ^OK[.SL[.WM[>[N[N_O[_#P\/'Q\?+R\O/S\_3T]/7U]?;V]O?W]_CX^/GY
M]O;V]O?W]_?X^/CX^?GY^?KZ^OKZ^_O[^_O\_/S\_/W]_?W]_?[^_O[^_O[_
M`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0$"`@("`@("`@("`@("`@("`@("
M`@("`@("`P,#`P,#`P,#`P0$!`0$!`0$!`0$!`04&!@8&!@8&!P<'!P<'!@8&
M!P<'!P<'"`@("`@("0D)"0D*"@H*"@L+"PL,#`P,#`T-#0T.#@X.#P\/$!`0$1$2
M$A(3$Q,4%!04%146%A<7%Q@8&!D9&1H:&AL;&QP<'!T='1X>'Q\?("`@(2$B
M(B(C(R,D)"0E)24F)B<G)R@H*"DI*BHJ*RLL+"PM+2TN+B\O,#`P,3$R,C,S
M-#0U-38V-S<X.#DY.CH[.SP\/3T^/S]`0$%!0D)#0T1$149&1T=(24E*2TM,
M3$U.3D]045%24E-45%5655976%A96EM;7%U>7F!@86)B8V1D96=G:&EJ:FML
M;6YO<'%R<W1U=G=X>7I[?'U^?X"!@H.$A8:'B(F*BXR-CH^0D9*3E)66EY::
M?8'AX>&AX>'AX>'AX>+CX^/CX^3DY.7EY>;FYN?GY^CHZ.GIZ>KJZNOK
MZ^OK[.SL[.WM[>[N[N_O[_#P\/'Q\?+R\O/S\_3T]/7U]?;V]O?W]_CX^/GY
M^?KZ^OO[^_S\_/W]_?[^_O___P$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!`0$!
M`0$!`0$!`0$!`@("`@("`@("`@("`@("`@("`@("`@("`@("`@("`P,#`P,#
M`P,#`P,#`P,#`P0$!`0$!`0$!`0$!`0%!04%!04%!04%!04%!04&!@8&!@8&
M!@8&!@8&!@8'!P<'!P<'!P<'!P@("`@("`@("`D)"0D)"0D)"@H*"@H*"@L+
M"PL+"PL,#`P,#`P,#0T-#0T-#@X.#@X.#P\/#Q`0$!`1$1$1$A(2$A,3$Q,4
M%!04%145%186%A<7%Q@8&!D9&1H:&AL;&QP<'!T='1X>'Q\?("`@(2$B(B(C
M(R,D)"0E)24F)B<G)R@H*"DI*BHJ*RLL+"PM+2TN+B\O,#`Q,3$R,C,S-#0U
M-34V-C<W.#@Y.3HZ.SL\/#T]/CX_/T!`04%"0D-#1$1%149&1T=(2$E)2DI+
M3$Q-34Y.3T]045%24E-35%155E975UA965I:6UQ<75U>7U]@8&%B8F-D9&5E
M9F=G:&EI:FIK;&QM;FYO<'!Q<G)S='1U=G9W>'EY>GM[?'U]?G]_@(&"@H.$
M:E;@ZW%R'@X0'!J,KJ*#YJLE46J]V)+S]&2X=B2G&S:\P"T!NBR@IRM'FDI
M1&7[X6G!FQZS>T7]@B'0J4W;;!%\:1X%R^X(Q&>Q9B\!HV5>?A*ODH_%X=Q
M02U-5SG8O+DWEF=V.L^9;,.$`,A?I-&'4>28UN3!VWO7+*#?'%V>@?AGG1[9
M1(]6*ZO1;2-T5B]*T1'S>\Z'!$'/]D\\G\/D+X[B4$93.U>A8\[>;,JP^7ZSD
M0]MCR(5X<R@G@MZ9-XS1.=&7FL)/%``?Z,YS]/7Z+J?&;]=:.&>7-3%/*<?R
MH6:4.04%2+WBG,96>W,,5M&;1L)5/H?$&R!%=4%5F5I9!$(T_<*7%6B#:9&5
JD0]1G6,;@_AW=U>8$R^:60N1S#*4^_M5G$<QU6UZ?=F*N,G0:46H93,P
```

end

:::::::::: table.rotmath ::::::::::

```
begin 644 table.rotmath
M`(0````````````````!`0$!`0$!`0("`@("`@,#`P,$!`0$!04%!08&!@<'
M!P@("`D)"0H*"PL+#`P-#0X.#P\0$!$1$A(3$Q04%146%Q<8&!D:&AL<'!T>
M'A\@("$B(R,D)28F)R@I*2HK+"TN+B\P,3(S-#4U-C<X.3H[/#T^/T!!0D-$
M149'2$E*2TU.3U!14E-45E=865I;75Y?8&)C)",C(B$@(!\>'AT<'!L:&AD8
M&!<7%A45%!03$Q(2$1$0$`\/#@X-#0P,"PL+"@H)"0D("`@'!P<&!@8%!04%
M!`0$!`,#`P,"`@("`@(!`0$!`0$!`0````````````````````````````
M`0$!`0$!`0$"`@("`@(#`P,#!`0$!`4%!04&!@8'!P<("`@)"0D*"@L+"PP,
M#0T.#@\/$!`1$1(2$Q,4%!45%A<7&!@9&AH;'!P='AX?("`A(B,C9&-B8%]>
M75M:65A75E134E%03TY-2TI)2$=&141#0D%`/SX]/#LZ.3@W-C4U-#,R,3`O
M+BXM+"LJ*2DH)R8F)20C(R(A("`?'AX='!P;&AH9&!@7%Q85%104$Q,2$A$1
M$!`/#PX.#0T,#`L+"PH*"0D)"`@(!P<'!@8&!04%!00$!`0#`P,#`@("`@("
3`0$!`0$!`0$```````````````@(
`
end
```

```
........
....
..
.                                      -fin-
```

See you next issue!