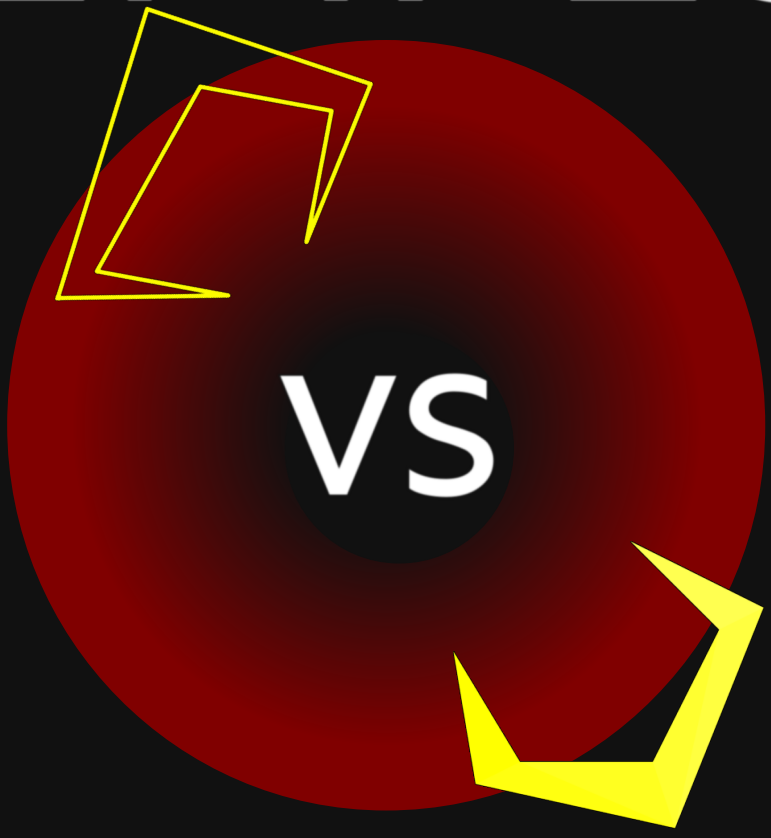
TEMPEST



TEMPEST

THE MAKING AND REMAKING OF ATARI'S ICONIC VIDEOGAME

TEMPEST
VS
TEMPEST

For Edna.

Edition Date: Tuesday 18th March, 2025

Contents

1	**mainline**	**13**
2	**mainloop**	**19**
3	**cry if i want to**	**31**
4	**what is blitting**	**37**
5	**character assassination**	**45**
6	**my first shader**	**53**
7	**unused stars**	**59**
8	**sounds**	**67**
9	**flipper**	**75**
10	**wells**	**81**
11	**webs**	**85**

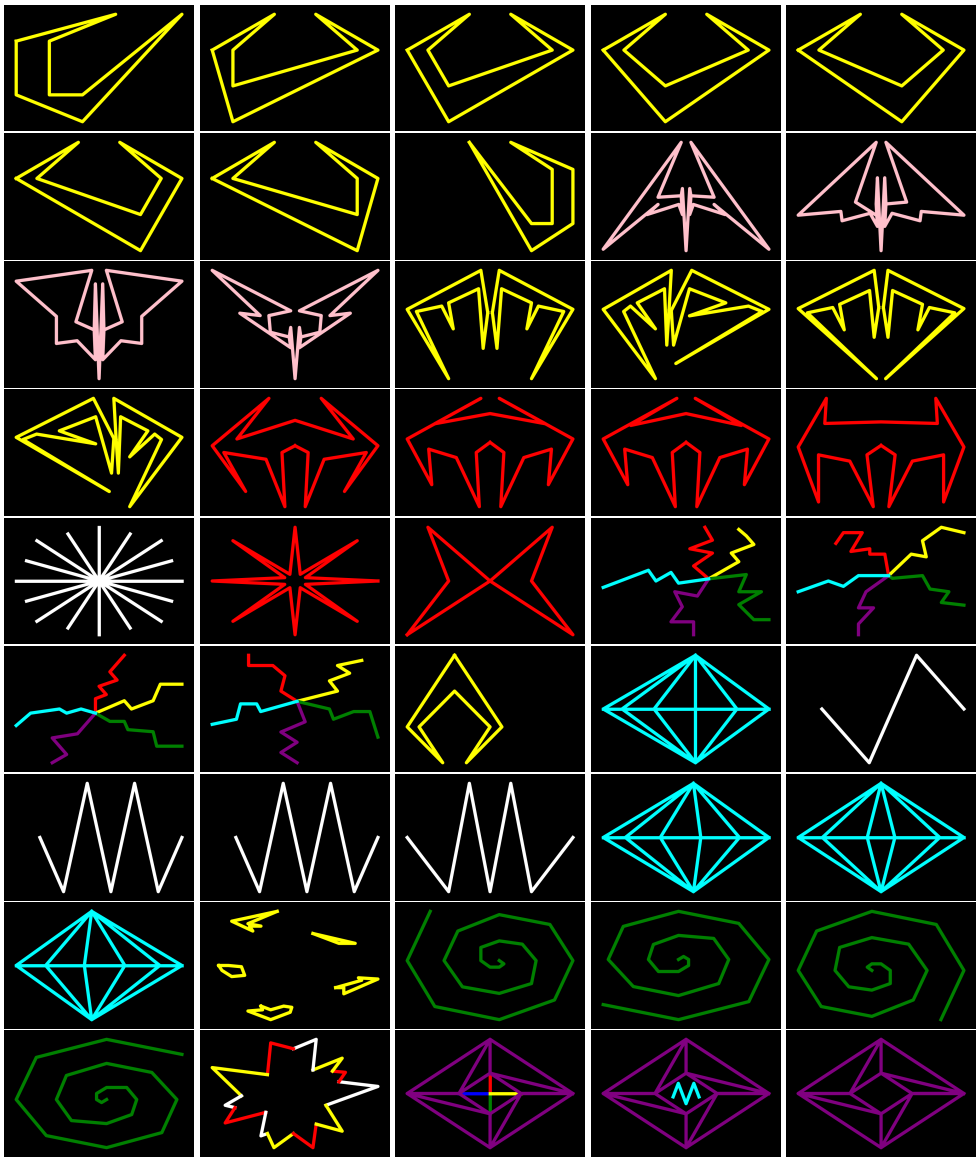
12 object list **91**

13 cursors **95**

14 claws **101**

15 more claws **105**

16 meltovision **111**



Artefacts from Tempest (1984).

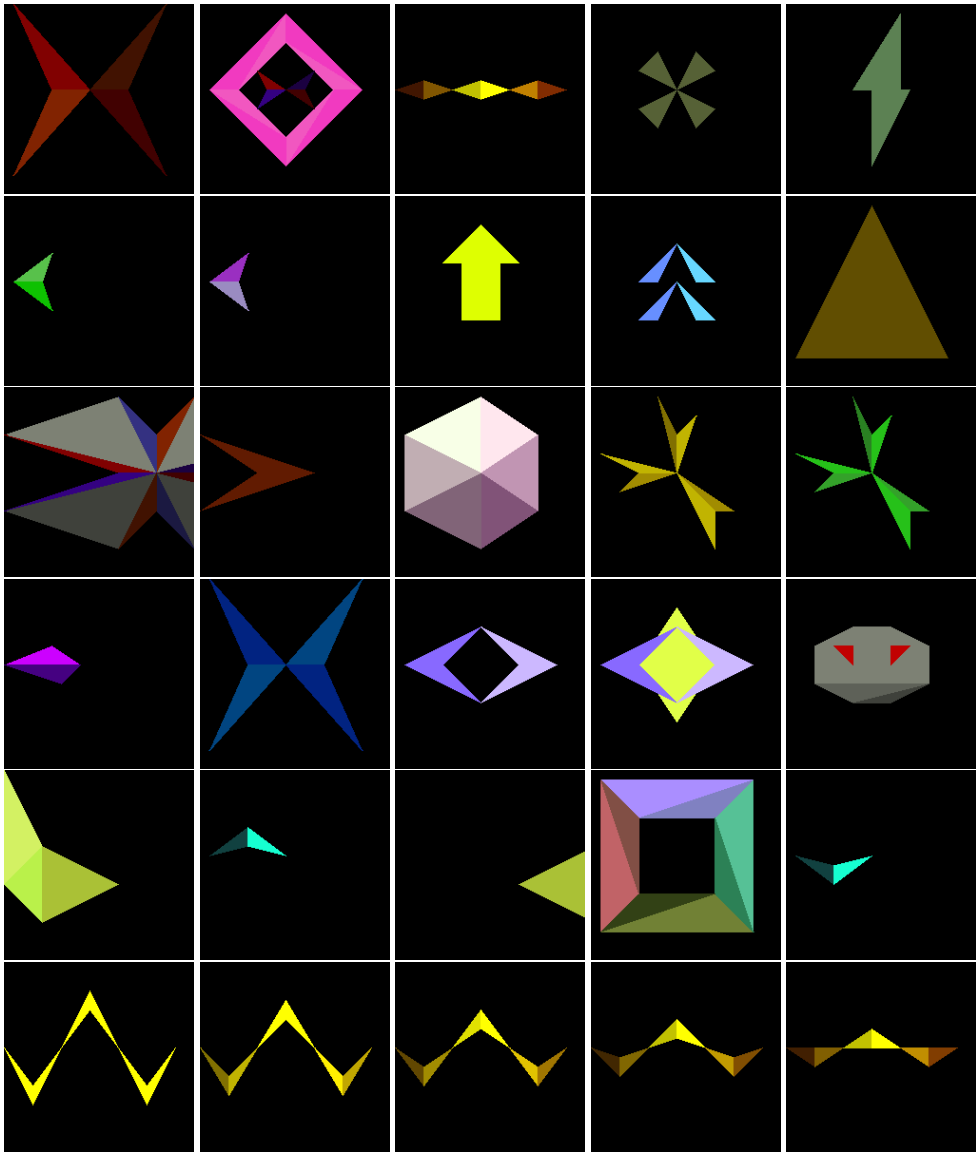
about this book

Tempest 2000 was a classic game. To its misfortune it was made for a console that crashed and burned with no survivors. The Atari Jaguar was supposed to be gaming's first great leap into the 64 bit era. Instead it promptly tripped over its shoelaces and left a number of great games, Tempest 2000 among them, behind as orphans.

This is a book about computer code. Each chapter aims to be an easily digested sketch of how a specific feature or effect was achieved using a combination of Motorola 68K assembly code and the Jaguar's powerful hardware platform.

There are lots of fascinating bits and pieces to pore over. These are the 25 easiest I could find.

Rob Hogan (@mwenge)
Dublin 2025



Artefacts from Tempest 2000 (1994).

TEMPEST
VS
TEMPEST

mainline

Tempest's 'mainline' is a short routine that runs the entirety of the game from a handful of lines of assembly code. Beneath lies an iceberg of strange machinery but at its core the game consists of running these 15 or so lines many hundreds of times per second:

```
;INPUT:          POWER ON RESET PREPARATION
;OUTPUT:         NONE

MAINLN: JSR INISOU              ;INITIALIZE SOUNDS
        LDA I,CNEWGA
        STA QSTATE
        BEGIN                   ;MAINLOOP
        BEGIN                   ;LOOP UNTIL CURRENT FRAME HAS BEEN UP X MS.
        LDA FRTIMR
        CMP I,9
        CSEND
        LDA I,0                 ;RESTART FRAME TIMER
        STA FRTIMR
        JSR EXSTAT              ;EXECUTE APPROPRIATE GAME STATE
        JSR NONSTA              ;EXECUTE NON-STATE DEPENDENT CODE
        JSR DISPLA              ;EXECUTE CODE TO DISPLAY NEW SCREEN
        CLC
        CSEND                   ;LOOP ALWAYS
```

The game's framerate is controlled by this tight little loop. As soon as `FRTIMR` reaches 9 a frame of the game is executed:

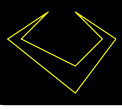
```
        LDA FRTIMR
        CMP I,9
        CSEND
```

Then it is reset to zero:

```
        LDA I,0                 ;RESTART FRAME TIMER
        STA FRTIMR
```

Increasing the value from `9` to `0F` (15) will slow the game down appreciably, while increasing it to `1F` (31) will bring it to an unplayable crawl. Lowering it to `0` or `1` does make the game slightly faster, but not noticeably and it is still playable.

claw says



But what increments `FRTIMR` you ask? The answer is a piece of code in what is known as an 'interrupt handler'. This runs separately from the main loop and is invoked by the CPU up to 60 times a second.

Once we've decided to execute a frame we are down to just three lines orchestrating the entirety of the game:

```
        JSR EXSTAT              ;EXECUTE APPROPRIATE GAME STATE
        JSR NONSTA              ;EXECUTE NON-STATE DEPENDENT CODE
        JSR DISPLA              ;EXECUTE CODE TO DISPLAY NEW SCREEN
```

The first routine we call is `EXSTAT`. It always updates the starfield (`PRSTAR`) but what it does next is dependent on the value contained in `QSTATE`. Take a brief look at the routine to see if you can identify how it uses `QSTATE`:

```
;INPUT:         QSTATE: CODE FOR STATE ROUTINE TO EXECUTE
;OUTPUT:        CONTROL PASSED TO ROUTINE

EXSTAT:
        LDA INOPO
        AND I,83
        CMP I,82
        IFNE                    ;FREEZE & FREE PLAY?
        JSR PRSTAR              ;PROCESS STAR FIELD
        LDX QSTATE
        LDA SWFINA              ;SET MUST PROCESS FLAG
        ORA I,MFAKE
        STA SWFINA
        LDA AX,ROUTAD+1
        PHA
        LDA AX,ROUTAD
        PHA
        ENDIF
NOOPR:  RTS
```

The answer is in these lines:

```
        LDX QSTATE
        ..
        LDA AX,ROUTAD+1
        PHA
        LDA AX,ROUTAD
        PHA
```

The value in QSTATE is loaded to the X register. This is then used an index to retrieve the two-byte address stored in the Xth - 1 index of the ROUTAD table:

```
;
;STATE ROUTINE ADDRESS
;
ROUTAD: .WORD NEWGAM-1          ;NEW GAME
        .WORD NEWLIF-1          ;NEW LIFE (AFTER LOSING A BASE)
        .WORD PLAY-1            ;PLAY
        .WORD ENDLIF-1          ;LIFE LOST
        .WORD ENDGAM-1          ;END OF GAME
        .WORD PAUSE-1           ;PAUSE
        .WORD 0                 ;NEW WAVE (AFTER SHOOTING ALL INVADERS)
        .WORD ENDWAV-1          ;END OF WAVE
        .WORD HISCHK-1          ;CHECK FOR HI SCORES
        .WORD GETINI-1          ;GET HI SCORE INITIALS
        .WORD DLADR-1           ;DISPLAY HI SCORE TABLE
        .WORD PRORAT-1          ;REQUEST PLAYER RATE
        .WORD NEWAV2-1          ;NEW WAVE PART 2
        .WORD LOGINI-1          ;LOGO INIT
        .WORD INIRAT-1          ;MONSTER DELAY/DISPLAY
        .WORD NEWLF2-1          ;NEW LIFE PART 2
        .WORD PLDROP-1          ;DROP MODE
        .WORD SYSTEM-1          ;END WAVE CLEAN UP AFTER BONUS
        .WORD PRBOOM-1          ;BOOM
```

Pushing each of the two bytes to the stack (using PHA) means that whatever is at the address of those two bytes will be executed next. For example, if the value in QSTATE is 0 it will retrieve the address of NEWGAM-1: which is 016B, made up of 6B which it retrieves first, and 01 which it retrieves second. When both of these bytes are pushed to the stack they are treated as an address for execution: the result is that the EXTSTAT routine will execute NEWGAM once it exits, beginning a new game.

So depending on the current state of the game, Tempest will populate QSTATE with the index for the function it needs to call next. To give a flavour of what NEWGAM looks like we've listed it below. As expected it mainly consists of accounting operations such as clearing the score, assigning the number of lives and so on.

```
NEWGAM: JSR INICHK              ;INITIALIZE LANGUAGE PTRS, OPTIONS
        JSR INIDSP              ;INITIALIZE DISPLAY
        LDA QSTATUS
        IFMI                    ;ATTRACT?
```

```
        JSR CLRSCO              ;NO. CLEAR SCORES
        ENDIF
        LDA I,0
        STA LIVES2              ;ONE PLAYER GAME (DEFAULT: PLAYER 2 DEAD)
        LDX NUMPLA              ;GIVE EACH PLAYER "NEW GAME" EQUIP
        STX PLAYUP
        BEGIN                   ;LOOP FOR EACH PLAYER IN GAME (1 OR 2)
        LDX PLAYUP
        LDA LVSGAM              ;GET # LIVES
        STA AX,LIVES1           ;INITIAL # OF LIVES (GUNS)
        LDA I,-1
        STA AX,WAVEN1           ;FORCE REQUEST RATE STATE
        DEC PLAYUP
        MIEND                   ;ENDLOOP AFTER ALL PLAYERS PROCESSED
        LDA I,0
        STA NEWPLA              ;START GAME WITH 1ST PLAYER UP.
        STA PLAGRO              ;DEACTIVATE STAR FIELD
        LDA NUMPLA              ;INDUCE "PLAY PLAYER 1" MESSAGE
        STA PLAYUP              ;IF 2 PLAYER GAME.
        JMP INIRA0              ;INITIALIZE FOR PLAYER RATE REQUEST
```

You will notice at the end that we jump to a routine called `INIRA0`. This continues the accounting set-up for a new game but it also does something important, which is update `QSTATE` to direct the flow the next time it passes through `MAINLN` to a different function. In this case, it loads `16` to `QSTATE`, which will direct us to the routine `PRORAT` at the next iteration:

```
        LDA I,CREQRAT           ;GO TO REQUEST
        STA QSTATE              ;RATE STATE
```

Another way of describing what we have here is a 'state machine'. As the state of the system changes, `EXSTAT` will execute a different routine, appropriate to the current state of the game. The one executed most often is the third one listed in our `ROUTEAD` table. This is called `PLAY` and its contents, as well as its name, are almost self-explanatory:

```
PLAY:
        JSR MOVCUR              ;MOVE CURSOR AROUND
        JSR FIREPC              ;FIRE PLAYER CHARGE
        JSR PROSUZ              ;PROCESS SUPER ZAP
        JSR MOVNYM              ;MOVE NYMPHS
        JSR MOVINV              ;MOVE INVADERS
        JSR MOVCHA              ;MOVE CHARGES
        JSR FIREIC              ;FIRE INVADER CHARGE
        JSR COLLIS              ;COLLISION DETECT
        JSR PROEXP              ;EXPLOSIONS
        JMP ANALYZ              ;ANALYZE PLAYER STATUS
```

Here every element of the game is updated and manipulated in some way. It is not

surprising that somewhere in the heart of a game's runtime the logic should ultimately settle on a list of items that need to be checked, one after the other. One important thing to note here is that we're not updating how any of the game elements are displayed. We are just updating their state, which we store separately from the details of how they are displayed. We will use this updated state to determine their display in a moment, but for now we are just updating our record of their position and condition. The second thing to note is that the order in which these operations are performed is not arbitrary. First we update the player and any bullets the player has fired. Then we move the enemies and any bullets they have fired. Now we can check if the player has collided with any of the enemies. If there has been a collision between any two objects we process an explosion and finally update the player's state to determine, for example, if they are dead. The purpose of this order is to ensure that information about the player and its enemies are updated before we detect to attempt any collisions.

A lot has happened underneath the single line..

```
        JSR EXSTAT              ;EXECUTE APPROPRIATE GAME STATE
```

.. however now that we have all the properties of the game in its current frame we can move on to updating how they are displayed:

```
        JSR DISPLA              ;EXECUTE CODE TO DISPLAY NEW SCREEN
```

Again this will in most cases boil down to updating all of the elements one by one. Unlike when we were concerned with updating state, the order in which we do things is not particularly important here. The `DISPLA` routine wants to build a list of lines to paint on the screen so it will compute the lines for each type of object currently displayed in a more or less abitrary sequence determined by the programmer. Notice that in the sequence below the display of each type of object follows a fixed pattern:

```
                                ;DISPLAY CURSOR (PLAYER)
        LDA I,BCCURS
        JSR SBCLOG
        JSR DSPCUR
        LDA I,BCCURS
        JSR SBCSWI
                                ;DISPLAY CHARGES
        LDA I,BCSHOT
        JSR SBCLOG
        JSR DSPCHG
        LDA I,BCSHOT
        JSR SBCSWI
                                ;DISPLAY INVADERS
        LDA I,BCINVA
        JSR SBCLOG
        JSR DSPINV
        LDA I,BCINVA
```

```
        JSR SBCSWI
                                ;DISPLAY EXPLOSIONS
        LDA I,BCEXPL
        JSR SBCLOG
        JSR DSPEXP
        LDA I,BCEXPL
        JSR SBCSWI
                                ;DISPLAY NYMPHS
        LDA I,BCNYMP
        JSR SBCLOG
        JSR DSPNYM
        LDA I,BCNYMP
        JSR SBCSWI
                                ;DISPLAY INFORMATION (SCORES, MSGS, ETC.)
        LDA I,BCINFO
        JSR SBCLOG
        JSR INFO
        LDA I,BCINFO
        JSR SBCSWI
        ..
        JSR DSPWEL              ;DISPLAY WELL

        LDA I,BCENEL            ;DISPLAY ENEMY LINES
        JSR SBCLOG
        JSR DSPENL
        LDA I,BCENEL
        JSR SBCSWI

        LDA I,BCSTAR            ;DISPLAY STAR FIELD
        JSR SBCLOG
        JSR DSTARF
        LDA I,BCSTAR
        JSR SBCSWI
```

In each case we prime the accumulator (`A`) with a value specific to that object, call the routine `SBCLOG`, call a routine specific to the object (e.g. `DSPCUR`), the close by calling the routine `SBCWI`. These top and tail routines are concerned with setting up and preparing the header and footer material for the list of lines that displays each object. The object-specific routine in the middle is concerned with collecting the lines themselves. We will get the opportunity to explore the detailed mechanics of how some of these objects are calculated and displayed in later sections.

mainloop

Unlike Tempest, the engine that runs Tempest 2000 is not contained within a mainline routine. Instead the donkey work is done during a vertical sync interrupt. Vertical sync is a point in time: specifically when the Atari Jaguar has finished writing pixels to the screen and has a short amount of time available before it starts writing pixels again from the top. An interrupt is a routine the system will call at a moment of the programmer's choosing. A vertical sync interrupt is when you combine the two, and in this case they are combined in a routine called `Frame`. It is this routine that manages the state of the game and prepares all the objects for display and the sound for output.

That said, Tempest 2000 does have a mainline routine but it is there to solve a problem encountered in development rather than as part of a grand design. As we shall see there are a number of ways of generating graphical data on the Atari Jaguar, and two of them have their own dedicated processors. These are the Graphics Processor (GPU), which specializes in the fast trigonometric operations required for 3D displays and the Blitter, which is suited for large copy and fill operations. It is worth being clear that neither of these processors or any of their operations actually write graphics to the screen. Instead this function is performed by a third dedicated processing unit, the Object Processor. It is the Object Processor that turns data into light. This unit takes a list of operations set up by the programmer for each frame and uses them to write pixels to the screen. These operations will usually include data prepared by the GPU and the Blitter in a long list of tasks for the Object Processor known as an Object List. It's the programmer's job to have a new Object List ready every time the Object Processor is about to paint the screen.

It was the initial intention that the `Frame` routine would be solely responsible for this task in Tempest 2000. But there was a snag: it kept crashing. We know this because

Jeff Minter left us one of his rare comments at the head of the `mainloop routine`:

```
; This loop runs the GPU/Blitter code.  I found that if you
; started up the GPU/Blitter pair from inside the FRAME
; Interrupt, the system would fall over if they got really heavily
; loaded.  MAINLOOP just waits for a sync from the FRAME routine,
; launches the GPU, then loops waiting for another sync.

mainloop:
        move #1,sync                ; Reset the sync
        move #1,screen_ready        ; Reset the screen ready.
        move pauen,_pauen           ; Reset the pause indicator.

main:   tst sync                    ; loop waiting for another sync
        bne main                    ; from the interrupt in 'Frame'

        move #1,sync                ; reset the sync so that we wait for a
    new frame next time around.
        move.l dscreen,gpu_screen

        move.l mainloop_routine,a0 ; do the actual mainloop work,
    mainloop_routine
        jsr (a0)                    ; is usually a reference to the '
    draw_objects' routine.
```

So instead of doing all the necessary GPU and Blitter operations during a vertical sync interrupt, we do them here in this `mainloop`. Notice the busyloop in the second paragraph above where we wait for the value in `sync` to change to a zero. This forces the CPU to wait until the `Frame` routine resets it to zero. Once it has been reset we can go ahead and execute our `mainloop_routine`. This is actually a reference to the `draw_objects` routine - and it is this routine that does all the GPU and Blitter work required to calculate the pixels for display.

Here we see that when the game is initialized, `draw_objects` is selected as the destination for `mainloop_routine`, and then we enter the `mainloop`.

```
go_in:  move.l #rotate_web,routine
        move.l #0,warp_count
        move.l #draw_objects,mainloop_routine  ; Make draw_objects the
    mainloop_routine
        move pauen,_pauen
        clr db_on
        bra mainloop
```

We will investigate the mechanics of using the GPU and the Blitter in later sections, but for now let's get a sense of the kind of things the `draw_objects` routine uses the GPU and Blitter for. A quick glance through the top of the routine shows us drawing the starfield (`dostarf`).

```
draw_objects:
        tst h2h                 ; Are we in head to head mode?
        bne nodraw              ; Yes, go to 'nodraw'.
        clr h2hor
stayhalt:
        tst drawhalt            ;
        bsr clearscreen
        move.b sysflags,d0
        and.l #$ff,d0
        move.l d0,_sysflags     ;pass sys flags to GPU
        tst sf_on               ; Is the starfield active?
        bne dostarf             ; If yes, go to 'dostarf'.
        bra gwb                 ; Otherwise do the web.

dostarf:                        ; Prepare the starfield!
        move.l #3,gpu_mode      ; mode 3 is starfield1
        move.l vp_x,in_buf+4    ; Put x pos in the in_buf buffer.
        move.l vp_y,in_buf+8    ; Put y pos in the buffer.
        move.l vp_z,d0          ; Get the current z pos.
        add.l vp_sf,d0          ; Increment it.
        move.l d0,in_buf+12     ; Add it to the buffer.
        move.l #field1,d0       ; Get the starfield data structure.
        move.l d0,in_buf+16     ; And put it in the buffer.
        move.l warp_count,in_buf+20 ; Add the warp count.
        move.l warp_add,in_buf+24   ; Add the warp increment.
        lea fastvector,a0       ; Get the GPU routine to use.
        jsr gpurun              ; do gpu routine
        jsr gpuwait             ; Wait until its finished.
```

As you can see 'doing the starfield' involves setting up a number of registers and variables before finally invoking the GPU (`jsr gpurun`) to do its magic and waiting for it to finish (`jsr gpuwait`). We will take a closer look at the mechanics of starfield generation in a later chapter, but the above begins to give us a flavour of the formula required to get a GPU routine up and running.

The next element we find is a routine for drawing the playfield of Tempest 2000: the web. As you can see there is a lot going on here and the truth is, this isn't even all of it. And not just that, there are a number of different GPU routines used for drawing webs scattered throughout the game depending on the mode we are playing or whether there is more than one player. Just looking briefly through the code below (and you should confine yourself to that for now) gives you a sense of the overhead involved in setting up a reasonably complex piece of GPU code. Note that at the end of the listing below we load a variable called `equine2` to the `A0` register. This is an address to the actual GPU code that the Graphics Processor will run, using all the data set up in the previous lines. So there is much more complexity and detail underneath, even here. We will cover this in more detail in the chapter devoted to webs.

```
gwb:
```

```
        move.l vp_x,d3
        move.l vp_y,d4
        move.l vp_z,d5

dscud:
solweb:
        move #120,d0
        add palfix2,d0
        and.l #$ff,d0
        move.l d0,ycent
        tst l_solidweb
        beq vweb
        cmp #1,webcol           ;our 'transparent' webs...
        beq vweb
        lea _web,a6             ;draw a solid poly Web
        tst 34(a6)
        beq n_wb
        lea in_buf,a0
        move.l 46(a6),d0
        move.l d0,(a0)
        move.l 4(a6),d0
        sub.l d3,d0
        move.l d0,4(a0)
        move.l 8(a6),d0
        sub.l d4,d0
        move.l d0,8(a0)
        move.l 12(a6),d0
        sub.l d5,d0
        bmi n_wb
        move.l d0,12(a0)
        move l_solidweb,d0    ; The web data structure.
        and.l #$ff,d0
        move.l d0,16(a0)
        move 28(a6),d0
        and.l #$ff,d0
        move.l d0,24(a0)
        move frames,d0
        and.l #$ff,d0
        move.l d0,28(a0)
        move.l #w16col,32(a0)
        move.l #0,gpu_mode
        lea equine2,a0         ; The GPU shader routine for webs.
        jsr gpurun
        jsr gpuwait
        jsr WaitBlit
```

You may note at the end that we call a routine called `WaitBlit`. This is because in addition to computing the polygons that make up the 3D web, the GPU also 'blits' or writes its results to a buffer that will ultimately be used by the Object Processor to write the pixels to the screen. It is this dual operation that caused Minter a headache when attempting it from within the vertical sync interrupt. (My own suspicion is that there just wasn't enough time in the interrupt to do everything he wanted on the GPU and Blitter -

so moving it here to the mainloop ensured that it would only be done opportunistically and at the risk of missing a frame every now and then.)

But there is more to Tempest 2000 than starfields and webs so there must be plenty of other things being drawn in here too. At first it is not easy to see where though. The next section of the `draw_objects` routine is cryptic at best but on close inspection does contain some clues. As elsewhere, I've added comments to assist understanding:

```
n_wb:   move.l activeobjects,a6 ; activeobjects is a list of things to draw!
        bsr d_obj
        bra odend

        ; A loop for processing everything in 'activeobjects'.
d_obj:  cmpa.l #-1,a6       ; Have we reached the end of activeobjects?
        beq oooend          ; If yes, skip to end.
        move 50(a6),d0      ; Is the object marked for deletion?
        beq no_unlink       ; If not, skip to no_unlink and draw it.

        ; This verbose section up until no_unlink is concerned entirely
        ; with deleting the dead object from the activeobjects list.
        move.l 56(a6),d1
        bmi tlink
        move.l d1,a5
        move.l 60(a6),60(a5) ; Make it invisible to the vsync interrupt.

tlink:  move #-1,50(a6)     ; mark it bad
        move.l 60(a6),-(a7)
        move d0,-(a7)
        move.l a6,a0
        move 32(a6),-(a7)   ;save player ownership tag
        move (a7)+,d1
        move (a7)+,d0
        lea uls,a1
        asl #2,d0
        move.l -4(a1,d0.w),a1
        jmp (a1)

uls: dc.l afinc,ashinc,pshinc

pshinc: tst d1              ;player ownership of an unlinked bullet
        beq ulsh1
        add #1,shots+2
ulo:    move #1,locked
        bsr unlinkobject
        clr locked
        bra nxt_o
ulsh1:  add #1,shots
        bra ulo
ashinc: add #1,ashots
afinc:  add #1,afree
        bra ulo
```

```
        ; Actually draw the object.
        ; No need to remove the object, just draw it.
no_unlink:
        lea draw_vex,a0       ; Get our table of draw routines.
        move 34(a6),d0        ; Is this object smaller than a pixel?
        bpl notpxl            ; If not, go to notpxl.
        move.l #draw_pel,a0   ; Use draw_pex for pixel-size objects.
        bra apal              ; Jump to the draw call.
notpxl: asl #2,d0             ; Multiply the val in d0 by 2.
        move.l 0(a0,d0.w),a0  ; Use it as an index into draw_vex.
apal:   move.l 60(a6),-(a7)   ; Store the index of next object in a7.
        jsr (a0)              ; But first call the routine in draw_vex.
        jsr gpuwait           ; Wait for the GPU to finish.
nxt_o:  clr locked            ; Clear 'locked' just in case.
nxt_ob: move.l (a7)+,a6       ; Put the index of next object back in a6.
        bra d_obj             ; Go to the next object.
oooend: rts
```

This part of the `draw_objects` routine is concerned with processing a linked list called `activeobjects` using a loop that runs from `d_obj` to `bra d_obj` in the second last line from the end. This `activeobjects` list contains the detail for all objects that are active on the screen and require drawing by the GPU/Blitter. Some of the original (but sparse) code comments added by Minter suggest that this was initially conceived as a list for managing just the player and enemy bullets but it expanded over time.

Each object in the `activeobjects` list has the following structure:

Bytes			
0-4	Vector Object or Solid Object		
4-8	X		
8-12	Y		
12-16	Z		
16-20	Position on web.		
20-24	Velocity		
24-28	Acceleration & XY orientation		
28-30	XZ Orientation & XZ orientation		
30-32	Y Rotation		
32-34	Z Rotation		
34-36	Index into draw routine in `draw_vex`.	36-38	Start address of pixel data. & Delta Z
38-40	Y		Colour change value
40-42	Colour		
42-44	Scale factor		
44-46	Mode to climb, descend or cross rail		
46-48	Size of Pixel Data & Duration.		
48-50	Fire Timer		
50-52	Marked for deletion		
52-54	Whether an enemy or not.		
54-56	Object Type		
56-60	Address of Previous Object		
60-64	Address of Next Object		

The structure of objects in `activeobjects`.

So in the 64 bytes of each object we cover quite a bit of ground. There is colour, co-

ordinate, motion, and orientation information. There's also detail that defines the behaviour of the object and most immediately relevant to what we are looking at here, an index into the draw routine to use for the object at bytes 34-36.

This index is a value that references the routine given at the appropriate position in the `draw_vex` array:

```
draw_vex:
          dc.l rrts,draw,draw_z,draw_vxc,draw_spike,draw_pixex,draw_mpixex,
     draw_oneup,draw_pel,changex
          dc.l draw_pring,draw_prex,dxshot,drawsphere,draw_fw,dmpix,dsclaw,
     dsclaw2
```

The draw routine can vary depending on the type of object. There's a distinct treatment of vector object and objects that are solid polygons. Vector objects require the least work and if the header in bytes 0-4 indicates that it requires vector drawing only (for example the player's claw) then a vector draw in the GPU will suffice. The `draw` routine in the second item in `draw_vex` is the base routine for deciding whether a vector draw is sufficient or not, and most of the other routines make use of it in addition to the object-specific detail they manage:

```
draw:
         move.l a6,oopss    ; Stash the header.
         move.l (a6),d0     ; Is the header value greater than zero?
         bpl vector         ; If yes, then a vector draw will suffice.

         ; Otherwise we need to do add this object to the 'apriority'
         ; list so that it can be drawn as a solid polygon.

         ; The 'apriority' list stores objects in the descending order
         ; of their Z co-ordinate. This ensures that nearer objects are
         ; painted in front of objects that are further away or 'behind' them
     .
         move.l fpriority,a0  ;get a free priority object
         move.l a6,(a0)
         move.l 12(a6),d0  ;get 'z'
         move.l d0,12(a0)  ;put z in prior object
         move.l apriority,a1
         move.l a1,a2
chklp:
         cmp.l #-1,a1        ;no objects active?
         bne prio1
         bra insertprior     ;we are at top of list then, if we are first a1=
     a2=-1

prio1:
         cmp.l 12(a1),d0     ;check against stored 'z'
         bge insertprior     ;behind, insert on to list
         move.l a1,a2
         move.l 8(a1),a1     ;get next object
```

```
        bra chklp    ;loop until list end or next object in front of us
        rts      ;return with object at right place in the list
```

Reading through the above we can see that if a `vector` draw won't do the job then the object gets added to a list called `apriority` and nothing else is done with it for now. So what we are doing in our `d_obj` loop is a first pass through the `activeobjects` list, with any items on the list that require attention to the order in which they are painted, passed off to the `apriority` list. Notice that we don't mutate or update the objects in any way, we simply pass them over so the structure of the objects we described above remains unchanged.

Once we have finished this first run through the `activeobjects` list our next order of business is to process the `apriority` list we've populated. This is done in `drawpolyos` which we call right after we've finished with our first pass of `activeobjects`.

```
        ; We've finished our first pass of activeobjects.
odend:
        bsr showscore      ; Show the score.
        ; In Tempest Classic mode we don't need solid polygons.
        tst blanka         ; Are we doing solid polygons?
        beq odvec          ; If not, skip.
        bsr drawpolyos     ; if we are, draw them.
```

Drawing our polygons happens below. We iterate through the 'apriority' list, deleting items in it as we go, and calling the object-specific draw routine for each to render them to the screen.

```
solids:
        dc.l rrts,cdraw_sflipper,draw_sfliptank,s_shot,draw_sfuseball
        dc.l draw_spulsar,draw_sfusetank,ringbull,draw_spulstank
        dc.l draw_pixex,draw_pup1,draw_gate,draw_h2hclaw,draw_mirr
        dc.l draw_h2hshot,draw_h2hgen,dxshot,draw_pprex,draw_h2hball
        dc.l draw_blueflip,ringbull,supf1,supf2,draw_beast,dr_beast3
        dc.l dr_beast2,draw_adroid

; Routine for drawing all solid polygons.
; Process each object in the 'apriority' list. We remove each item after
; processing.The draw routine for each item is given by its index into
; the 'solids' array.
drawpolyos:
        move.l #192,xcent    ; Set 192 as X centre.
        move.l #120,d6       ; Set 120 as Y centre.
        add palfix2,d6       ; Adjust for PAL if necessary.
        move.l d6,ycent      ; Store it as Y centre.
        move.l apriority,a0 ; Get our 'apriority' list.
dpoloop:
        cmp.l #-1,a0
        beq rrts             ; End of list was reached
        move.l (a0),a6       ; Get the index to 'solids'
```

```
        move.l (a6),d0      ; Store it in d0.
        move.l a0,-(a7)     ; Stash our current position in the list.
        bsr podraw          ; Go do object type draw
        jsr gpuwait         ; wait for gpu
        move.l (a7)+,a0     ; Get our current position in the list
        move.l 8(a0),-(a7)  ; Get the next position in the list
        bsr unlinkprior     ; Delete the current object.
        move.l (a7)+,a0     ; Move to the next position in the list.
        bra dpoloop         ; Loop until all objects drawn and unlinked

podraw:
        move.l #9,d4        ; Set X centre as 9.
        move.l #9,d5        ; Set Y centre as 9.
soldraw:
        neg d0
        lea solids,a4       ; Get the 'solids' list.
        lsl #2,d0           ; Multiply our index by 2.
        move.l 0(a4,d0.w),a0  ; Get the draw routine address from 'solids'.
        move.l 4(a6),d2     ; Get the X position from our object.
        sub.l vp_x,d2       ; Subtract our X viewpoint.
        move.l 8(a6),d3     ; Get the Y position from our object.
        sub.l vp_y,d3       ; Subtract our Y viewpoint.
        move.l 12(a6),d1    ; Get the Z position from our object.
        sub.l vp_z,d1       ; Subtract our X viewpoint.
        bmi rrts            ; Skip if not visible.
        move 28(a6),d0      ; Get orientation of object.
        and.l #$ff,d0       ; Use only the least significant bytes.
        jmp (a0)            ; Call the objects draw routine.
                            ; The draw routine returns to 'dpoloop'.
```

The `Frame` Routine

We set up our frame interrupt handler near the very start of initialisation:

```
;*****int setup
        jsr scint               ;set intmask according to controller prefs
        move.l #Frame,$100
        move.w n_vde,d0
        or #1,d0
        move d0,VI
        move pit0,PIT0
        clr d0
        move.b intmask,d0
        move.w  d0,INT1         ;enable frame int
        move.w  sr,d0
        and.w   #$f8ff,d0
        move.w  d0,sr           ;interrupts on
```

```
Frame:
```

```
;       add #1,frames

;       move.l #$FFFFFFFF,BORD1
        movem.l d0-d5/a0-a2,-(a7)       ;simple thang to make

; vertical blank code goes here



fr:     move INT1,d0
        move d0,-(a7)
        btst #0,d0
        beq CheckTimer          ;go do Music thang


        movem.l d6-d7/a3-a6,-(a7)
        move.l blist,a0
        move.l dlist,a1
        move.l #$30,d0
xlst:   move.l (a0)+,(a1)+      ;copy built list to displayed list
        dbra d0,xlst

        bsr RunBeasties         ;build the next one


setdb:
;
; this code writes the proper screen address to double buffered objects in
    the display list

;       lea 32(a0),a0           ;skip background object

        tst screen_ready        ;is GPU ready with a new screen
        beq no_new_screen       ;no
;       tst pawsed
;       bne no_new_screen
        tst sync
        beq no_new_screen
        move.l cscreen,d1
        move.l dscreen,cscreen
        move.l d1,dscreen       ;swap screens
        clr screen_ready
        clr sync
no_new_screen:

        move.l dlist,a0
        move db_on,d7           ;check for double buffer on (d7 holds # of
    contiguous DB'ed screens)
        bmi no_db



stdb:   move.l cscreen,d6       ;get address of current displayed screen
        and.l #$fffffff8,d6     ;lose three LSB's
```

```
        lsl.l #8,d6             ;move to correct bit position
        move.l (a0),d1          ;get first word of the BMO
        and.l #$7ff,d1          ;clear data pointer
        or.l d6,d1              ;mask in new pointer
        move.l d1,(a0)          ;replace in OL
        lea 32(a0),a0           ;skip to nxt object
;       dbra d7,stdb            ;loop for all DB backgrounds


no_db:
        jsr dowf

        tst pal
        bne dtoon
        add #1,tuntime          ;do NTSC interrupts only 5/6 times
        cmp #4,tuntime
        ble dtoon
        cmp #6,tuntime
        bne ntoon
        clr tuntime
dtoon:  tst modstop
        bne ntoon
        jsr NT_VBL
ntoon:


;       bsr readpad                     ;get joy values
        tst pawsed
        bne zial
        add #1,frames
        move.l fx,a0
        jsr (a0)                ;for plaette changes etc
zial:
        tst locked
        beq doframe
;       move #1,drawhalt
        bra loseframe
doframe:
        move.l routine,a0
        jsr (a0)                ;call do thangs routine
        clr drawhalt

loseframe:
        bsr checkpause       ;do pause and overriding reset command
        bsr domod               ;do music handling
        btst.b #0,sysflags
        bne chit                ;check for no h.interlace
        move frames,d0
        and #$01,d0
        add #SIDE,d0
        sub palside,d0
        move d0,beasties        ;H. Interlace mode
chit:   movem.l (a7)+,d6-d7/a3-a6
```

```
CheckTimer: move (a7)+,d0       ;get back int status
        move d0,-(a7)
        btst #3,d0
        beq exxit

        tst roconon             ;Rotary Controller enabled?
        bne roco
        jsr dopad
        bra exxit               ;Yeah, interrupts at 8x normal speed, go do
    special stuff

roco:   move pitcount,d1
        and #$07,d1
        bne rotonly
        jsr dopad
rotonly: add #1,pitcount
        bsr readrotary



exxit:  move (a7)+,d0
        lsl #8,d0
        move.b intmask,d0
        move d0,INT1
        move d0,INT2
        movem.l (a7)+,d0-d5/a0-a2
;       move.l #0,BORD1
;       move.w #$101,INT1       ;do interrupt stuff
;       move.w  #0,INT2
        rte
```

cry if i want to

Before the dawn of the three horsemen *png*, *gif* and *jpg*, computer people were free to invent their own schemes for storing images. In 1991 technology was still rudimentary enough that console and computer developers were largely pre-occupied with waging something resembling a *Color War*. So when Atari sat down to brew its own method of storing and representing image data one of its overriding concerns was *how many colours can we fit into this thing*.

The **CRY** file format is the result. 'Format' in this case is a slightly grandiose term for: here is a list of bytes in a file, each pair of bytes in the list represents a color. Since a single byte has 256 possible values, two bytes has 65,536 possible values. So welcome to our system that has a state-of-the-art 65,536 possible colours. Well, not quite. Let's see why.

claw says

Good luck figuring out what CRY is an acryonym of. Maybe it refers to the three axes of the 'colour space': cyan, red, and yellow. Maybe it refers to the three components contained in each byte pair: color, radiance, and luminosity?

In your innocence dear reader, you may expect that this color scheme is as simple as assigning a color to each of the possible values and considering ourselves done for the day. But that is nowhere near complex enough to generate true job sastifaction. Instead, we must declare something called a 'colour space' and declare co-ordinates along an X and Y axis. Even better, since this is not the 1980s anymore, we ought to thinking in three dimensions and throw in a Z axis while we're at it.

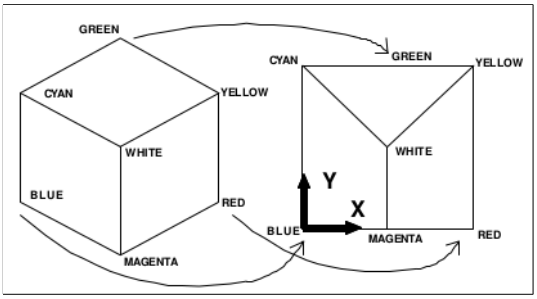


Figure 3.1: This kind of thing. This is what we want.

With just two bytes (or 16 bits) to work with we have to come up with a way that is suitably complicated for representing all our colours along so many co-ordinate axes. So let's imagine an 8x8x8 colour cube with something resembling white at its top corner and all the colours of the spectrum radiating out of it.

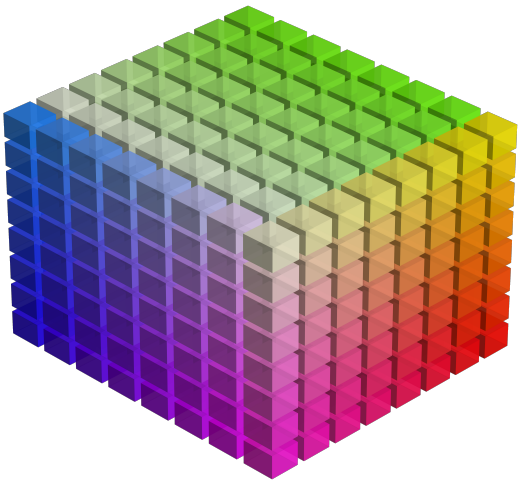


Figure 3.2: This is not quite right, but you get the idea.

Now if we project this three dimensional object onto a flat surface, with our white top corner at the centre, we will have a 16x16 square with 256 different colours:

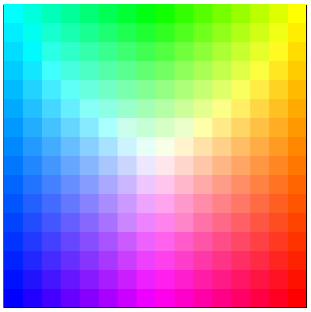


Figure 3.3: 3d, but in 2d: it's the 1990s.

Since 256 is exactly the number of permutations in a single byte we have lit upon a use-case for the first of our two bytes.

Figure 3.4: The first 8 bits now have a purpose. The first four are our X axis on the square, the second our Y axis.

This is all very nice, we have 256 colors going on and a whole other 8 bits to do something with. After thinking about it a bit, we don't have too many options. In fact we have one option: use our extra 8 bits to vary each of our 256 colors in 256 different ways. We can make this twiddle factor sound intelligent by calling it 'luminosity'. So depending on how much light we think is being cast on a surface with the color in question we can use our final 8 bits to dial its brightness up or down.

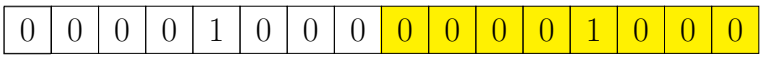


Figure 3.5: The 8 bits representing the luminosity/intensity of the color chosen by the first 8 bits.

To get an idea of the effect this has in practice we can visualize our little color square repeating across a much bigger square with ascending values of luminosity starting at 0 and ending at 255.

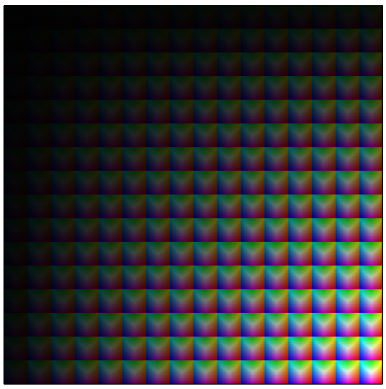


Figure 3.6: A big square with all our little squares in ascending luminosity.

That's a lot of dark. We better not wear shaded spectactles while playing games on this thing. Is there really 65,536 colors in there, it seems like we have an awful lot of black going on. Let's try something and see what we get if we ignore all color values that appear more than once.

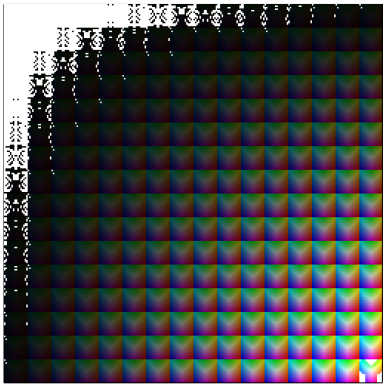


Figure 3.7: Of course the 65,536 colors is a lie. I've counted them so you don't have to: there are 63,762 unique colours.

There are some strange little fractal shapes in the black stuff there. In the bottom right hand corner we can see that we eventually run out of luminosity, our last little sqaure can only muster up 120 or so new colours.

the tempest files

Now that we know what the bytes in a CRY image mean let's take a peek at the 6 image files in the Tempest 2000 sources to see what some of these bytes look like in practice. (Don't worry we'll be looking at all of these files in detail under one pretext or another during the course of the rest of this book.) The seven image files are given slightly random, but very Minter-ish names: `beasty3.cry`, `beasty4.cry`, `beasty5.cry`, `beasty6.cry`, `beasty7.cry` and `beasty8.cry`.

claw says

Oh and let's not forget `beasty8-xtra.cry`. This is a slightly larger version of `beasty8.cry` included with the sources but for some reason is not used. There is also a story to tell about `beasty3.cry` but we will come to that later.

Remember there is nothing else in these files but the series of byte-pairs. There's no information about how big the image is or what its dimensions are. It really is just a list. To cope with this fact the game hard-codes its own expectation that the images are 320 pixels wide (i.e. the pixel width of the Jaguar's screen) and fixes the address that each image at will be loaded at. It does this early on in the declaration of constants at the top of the main file containing all the game's code (`yak.s`):

```
pic             EQU $820000             ; beasty3.cry
pic2            EQU pic+$1f400          ; beasty4.cry
pic3            EQU pic2+$1f400         ; beasty5.cry
pic4            EQU pic3+$25800         ; beasty6.cry
pic5            EQU pic4+(640*128)      ; beasty7.cry
pic6            EQU pic5+(640*200)      ; beasty8.cry
```

Each image has been loaded to a specific position in memory and the game is instructed exactly where each one is. While `beasty3.cry` is loaded to `$820000` (and assigned the variable name `pic`), `beasty4.cry` is loaded to the address 128,000 (`$1f400`) bytes after that and assigned the name `pix2` because 128,000 bytes is how large `beasty3.cry` is. You get the idea hopefully.

As you can see in the above listing extract some of the images are larger than others so need more memory, but in all cases the assumption of treating every image as 320 pixels wide (640 bytes, since we have 2 bytes per pixel) holds.

Before passing on let's take a look at the contents of `beasty7.cry`.

Figure 3.8: `beasty7.cry`, packed with goodness.

Looks great, you have to agree. We're in for some fun.

what is blitting

Blitting is the process of writing pixels to the screen. A bltter is a *bit block processor*. When we *blit* an image to the screen we are copying a specific block of image data from storage to the screen where the user can see it.

This short extract from the routine that looks after the 'High Score' table in Tempest 200 gives you an idea of how easy using the blitter can be. All we have to do is point at the position in RAM containing our pixel data (in this case the pixels from `beasty7.cry` that we loaded to `pic5`), tell the blitter the x and y position we want to copy from, the width and height we want to copy, and the x and y position on the screen we want to copy to.

```
; Paint the 'Top Guns' graphic
move.l #pic5,a0     ; file images/beasty7.cry, our source data.
move.l #screen3,a1  ; the destination we're copying to.
move #1,d0          ; x position in pic5
move #1,d1          ; y position in pic5
move #223,d2        ; width of block to copy
move #79,d3         ; height of block to copy
move #70,d4         ; x pos of destination
move #10+8,d5       ; y pos of destination
jsr CopyBlock
```

So in the above we specify a block in the image data from *beasty7.cry* opposite from the top-left corner (x:1, y:1) with a width of 223 pixels and a height of 79 pixels. We then tell it that we want to write this to x position 70 and y position 18 on the screen, in other words somewhere just offset from the top left of the screen. This is the piece of the image we are selecting for copying:

Figure 4.1: The bit of `beasty7.cry` we've selected for copying.

If we look more closely at the snippet I extracted above we can wee that We seem to be doing this 'specifying' by moving our values into things called `a0`, `a1`, `d0`, `d1`, and so on. These are 'registers'. An old computer such as the Motorola 68K CPU in the Atari Jaguar has only so many fingers and thumbs it can use for counting, and these are them. `a0`, `a1` and so on are 'address registers'. So if we have the address of a piece of code or data we want the CPU to use an 'address register' is where we store them. If we have some 'values' on the other hand, i.e. some numbers, then the 'data registers' `d0`,`d1` are where we put them.

As you read on you'll find it's a common pattern in Tempest 2000's source code to load up a bunch of registers with bits and pieces and then call a function (or 'routine' in assembler parlance) that expects those registers to contain the values it should work with. If this sounds familiar, and that we are essentially using these registers as parameters to a function, then you are correct: that's exactly what we're doing.

The simple operation we're performing here is to copy a portion of the `beasty7.cry` image file to the screen. The `CopyBlock` routine opposite just needs to know the position and dimensions of the portion we're copying and the position on the screen we want to place it. To achieve this the snippet of game code on the previous page sets up the registers as follows:

Register	Type	Value	Description
`a0`	Address	`pic5`	Source Screen
`a1`	Address	`screen3`	Destination Screen
`d0`	Data	`1`	X Position in Source Screen
`d1`	Data	`1`	Y Position in Source Screen
`d2`	Data	`223`	Width to Copy
`d3`	Data	`79`	Height to Copy
`d4`	Data	`70`	X Position in Destination Screen to Copy to
`d5`	Data	`18`	Y Position in Destination Screen to Copy to

Setting up the registers for `CopyBlock`.

```
CopyBlock:
;
; Copy from screen at a0 to screen at a1
; d0/d1=origin of sourceblock
; d2/d3=width and height of block to copy
; copy from blitter a1 to a2.
; d4/d5=destination XY
;
; This simple routine will assume both screens are the same width
; Using this blitter is a piece of piss.

  move.l #PITCH1|PIXEL16|WID320|XADDINC,d7
  move.l d7,A1_FLAGS  ;a1 (Source) Gubbins

  move.l #PITCH1|PIXEL16|WID384|XADDPIX|YADD1,d7
  move.l d7,A2_FLAGS  ;a2 (Dest) Gubbins

  move d3,d7
  swap d7
  move d2,d7
  move.l d7,B_COUNT   ;set inner and outer loop counts

  move d1,d7
  swap d7
  move d0,d7
  move.l d7,A1_PIXEL  ;origin of source

  move d5,d7
  swap d7
  move d4,d7
  move.l d7,A2_PIXEL  ;origin of destination

  move.l #0,A1_FPIXEL


  move.l #$0001,A1_INC
  move.l #$0,A1_FINC

  move #1,d7
  swap d7
  move d2,d7
  neg d7
  move.l d7,A1_STEP
  move.l d7,A2_STEP    ;set loop steps

  move.l a0,d7
  move.l d7,A1_BASE
  move.l a1,d7
  move.l d7,A2_BASE    ;set screen window bases

  move.l #SRCEN|UPDA1|UPDA2|DSTA2|LFU_A|LFU_AN,d7
  move.l d7,B_CMD
  bra WaitBlit
```

claw says

The source and destination are described as 'screens' in the Tempest 2000 code. This is a useful way of thinking of our pixel data, which although it is just a list of bytes, always represents a screen of a certain height and width. While in practice we are copying a section from an image to the screen here, we can think of it as copying from one screen to another - with the destination screen the one being shown to the player.

The mechanics of how the `CopyBlock` routine uses this information to actually copy a segment of our image to the screen are not obvious to the unintiated. This is because it involves a lot of precise specification of many different instructions to the Blitter (Bit Block Processor) hardware. The amount of set up involved makes our preparation of the `a0-a1` and `d0-d5` registers look modest in comparison. There is a quite involved amount of bit-twiddling required to get the Blitter set up the way we want it. Using this routine may be a 'piece of piss', but writing it was fairly painstaking. For example:

```
    move.l #PITCH1|PIXEL16|WID320|XADDINC,d7
    move.l d7,A1_FLAGS  ;a1 (Source) Gubbins
```

Here we are formulating some of the parameters we want the Blitter to use when it copies the pixels from our source. We're combining the parameters into a single 64-bit value, storing it in the `d7` data register and then writing that to the Blitter's `A1_FLAGS` register. This is the register it will read to figure out some of the things we want to do with the copy operation.

Parameter	Bits				Hex	Description
PITCH1	00000000	00000000	00000000	00000000	00000000	Pixel data has no gaps
PIXEL16	00000000	00000000	00000000	00110000	00000020	Pixel Size 16 bits
WID320	00000000	00000000	01000010	00000000	00004200	Screen Width of 320
XADDINC	00000000	00000011	00000000	00000000	00030000	Add 1 to X at each pass
A1_FLAGS	00000000	00000011	01000010	00110000	00034220	Blitter's A1 Flags Register

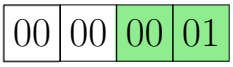
Combining our parameters into the Blitter's A1 Flags Register.

There is a lot more of this kind of thing in there and it would definitely be too tedious to itemize each operation here in detail. But there is at least one other pattern worth pointing out in the routine, exemplified by the following snippet that copies the X (`d0`) and Y (`d1`) positions in *beasty7.cry* that we are telling the blitter to use as its origin for copying:

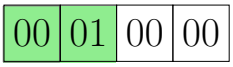
```
    move d1,d7
    swap d7
```

```
    move d0,d7
    move.l d7,A1_PIXEL   ;origin of source
```

Here the `A1_PIXEL` register we're writing to (like all our address and data registers) is 32-bits (or 4 bytes) long. This means it can contain both our X and Y values which will be at most 2 bytes long each. So to get our `d0` and `d1` values in there we first write the `d1` value to `d7` giving:



Next, we swap its position:



Now we write `d0` to `d7`:

The result is our X and Y values are both successfully stowed in `d7`, which we can now write in its totality to `A1_PIXELS`. This little exercise illustrates the granularity at which our code has to operate.

After setting up all our registers on the Blitter we finally get to tell it to set to work:

```
    move.l #SRCEN|UPDA1|UPDA2|DSTA2|LFU_A|LFU_AN,d7
    move.l d7,BLIT_CMD
```

As with everything else Blitter-related this is achieved above by writing a value to the Blitter command register. The register affords up to 30 different commands and here we just use a few of them by populating the register with six different values.

Parameter	Bits				Hex	Description
SRCEN	00000000	00000000	00000000	00000001	00000001	Enable Source Read in Loop
UPDA1	00000000	00000000	00000001	00000000	00000100	Add the A1 value in the outer loop
UPDA2	00000000	00000000	00000010	00000000	00002000	Add the A2 value in the outer loop
DSTA2	00000000	00000000	00000100	00000000	00004000	Reverse role of A1 and A2 registers
LFU_A	00000001	00000000	00000000	00000000	01000000	AND the bits
LFU_AN	00000000	10000000	00000000	00000000	00800000	AND NOT the bits
BLIT_CMD	00000000	00000011	01000010	00110000	01804210	Blitter's Command Register

Combining our parameters into the Blitter's Command Register.

In a nutshell this command is instructing the blitter to read through our source data using an inner loop for the values to write along the X co-ordinates and an outer loop for the Y co-ordinates. The implementation details are obviously complex and we won't get into them here. Instead we can treat this as a recipe for the blitter when we have a chunk from a source image that we want to position on the screen as we do here.

One last feature is worth pointing out explicitly. The final step in the `CopyBlock` routine is to call the `WaitBlit` sub-routine. As the name suggests this piece of code busy-loops until the `BLIT_CMD` register indicates that the blitter has finished:

```
WaitBlit:
  move.l BLIT_CMD,d7 ;get Blitter status regs
  btst #0,d7         ;check if d7 is still zero
  beq WaitBlit       ;loop if d7 is zero
  rts
```

Before rounding up this piece let's take a look at another instance of `CopyBlock` in use. This time we're in the `versionscreen` routine which is responsible for painting the title screen that greets us when we launch Tempest 2000.

Figure 4.2: The title screen of Tempest 2000.

Here we find a slightly different order of business in setting up the blitting copy:

```
WaitBlit:
  move #4,d0          ; x position in pic5
  move #84,d1         ; y position in pic5
  move #197,d2        ; width to copy
  move #65,d3         ; height to copy
  move #92,d4         ; x pos of write to
  move #120-15,d5     ; y pos to write to
```

```
    tst pal             ; are we PAL or NTSC?
    beq mypal
    add #10,d5          ; we're NTSC so offset the y pos by 10 pixels
mypal:
    move.l #pic5,a0a    ; file 'images/beasty7.cry' again
    move.l #screen3,a1  ; destination we're copying to
    jsr CopyBlock
```

Can you guess the piece of `beasty7.cry` we're copying here? It's this:

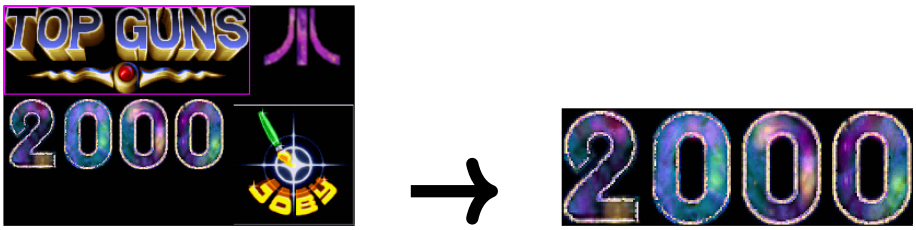


Figure 4.3: Extracting the '2000' chunk from `beasty7.cry`.

There is a little extra twiddling in our code to figure out whether the display we're on is PAL or NTSC. These were slightly conflicting standards for analogue television display through the 1960s to the 1990s. NTSC was the standard in the US and PAL the standard in Europe. Most video games of the time needed to account for the slightly different dimensions each standard offered. In the case of our title screen that meant adding an extra offset of 10 pixels to the Y position of the '2000' when painting it to the screen on an NTSC set in order to ensure that it appeared correctly positioned.

Now that we've got the general idea of how to get at least some pictures on to the screen, let's take a look at how we get some text up there.

character assassination

Writing some text to the screen, such as the copyright information on the title screen, is surprisingly convoluted in its detail, it will even introduce us to the machinations of the Jaguar's GPU. But although there is a surprising amount of code required from the developer to get a string of characters onto the display the general principle is one familiar from other platforms: you define a font, you specify the text you want to display and where you want to display it, and finally you call a routine that will take care of all the detail.

In our case the routine that takes care of all the detail will be our first taste of the implementation and use of what are known as 'shaders', free-standing programs that are written in a customized version of Motorola 68000 assembly lanage for the Atari's Graphics Procesing Unit(GPU), capriciously dubbed 'Tom' by the hardware designers of Atari at the time.

Before we get to that we can coast through the relatively easy part of defining our font and specifying our string of text, as well as plugging in the values to the GPU's buffer that will be required by the shader when it takes care of actually putting pixels on screen.

When it comes to defining a font for the copyright information on Tempest 2000's title screen you may have already guessed where information is contained: in one of our `beasty` cry files, in this case `beasty4.cry`:

Figure 5.1: `beasty4.cry` with the copyright font at the bottom.

And as you may remember the data in `beasty4.cry` is pointed by the variable `pic2`:

```
pic             EQU $820000             ; beasty3.cry
pic2            EQU pic+$1f400          ; beasty4.cry
pic3            EQU pic2+$1f400         ; beasty5.cry
pic4            EQU pic3+$25800         ; beasty6.cry
pic5            EQU pic4+(640*128)      ; beasty7.cry
pic6            EQU pic5+(640*200)      ; beasty8.cry
```

So all we have to do to is devise a scheme that maps from a given letter or number to its location (and dimensions) in `beasty.cry`. For example, if we want to display the letter 'A' we need to come up with a way of extracting this:

A.

It helps us that all of our font characters in the `cry` file are the same dimensions so we can make the height and width a consistent assumption for every glyph. Armed with the location of the pixel data and this dimension information all that remains is to create a look-up table that tells us the X and Y co-ordinates in `beasty4.cry` for each character.

The easiest form of look-up table we can make is an 'indexed' one. In other words, a procedure that converts each letter and digit in our character set into a number that we then use as an index into a list of the X and Y locations. For example, if 'A' is converted to the number 41 then we fetch its coordinates in `beasty4.cry` from the

42nd member of the list. If 'B' is coverted to number 42 we fetch its co-ordinates from the 43rd member of the list, and so on. Fortunately we don't have to invent a method of our own for relating letters and digits to a number. The ASCII committee has done this for us already. So as long as we store the co-ordinates in our list in the same order as given by the ASCII committe we will simply be able to use the corresponding ASCII value for each letter and digit as our index.

With the main planks of our cunning plan settled in place we can iron out a few details. The best way of storing our co-ordinate information in this list is to cram the X and Y values into a single place. More specifically, we can assume our X and Y values will always be between 0 and 65,536 so each will fit in one half of a single 4 byte (64 bit value). Another way of describing a value of this length is as a 'long' and in 68K assembler the notation to denote such a value is `dc.l`.

So let's assume we have 'A' at Y co-ordinate 165 and X co-ordinate 2 in our `beasty4.cry` 'sprite sheet'. Since we're dealing with zero-based indices we subtract 1 from each of these values to give us 164 and 1 respectively. In hex 164 is `$A4` and 1 is `$01`. We can fit this into our 4 byte entry as follows:

00	$A4$	00	01

Likewise with our fixed set of dimensions for each of the characters or glyphs. We get out our ruler, apply it to the screen, and determine they each have a height of 19 and a width of 16. We subtract 1 from each to give us 18 and 15. Convert to hex: `$12` and `$0F`. Then plug them into our 4 byte value:

00	12	00	$0F$

With this information we can now isolate our 'A' glyph and extract it.

Figure 5.2: The 'A' glyph located in a red box by our dimension and co-ordinate information.

The mechanics of our look-up table are now fully worked out so we are ready to create it. Since it has over a hundred entries we store it in a separate file called `afont.s` and give it the very descriptive name `afont`. I think this is intended as an abbreviation of 'Atari Font', since it is the font used to write the Atari copyright information. An incomplete excerpt of the file follows below with ellipses (...) to indicate lines that have been omitted.

```
*
*
* Page 2 font, by Joby

afont:
  dc.l pic2       ; pixel data
  dc.l $0012000f  ; height ($0012) and width ($000f)

  dc.l $b6011e    ;Space
  dc.l $b600d2    ;!
  dc.l $a40001    ;"
  dc.l $a40001    ;#
  dc.l $a40001    ;$
  dc.l $a40001    ;%
  dc.l $a40001    ;&
  dc.l $a40001    ;'
  dc.l $9100a5    ;(
  dc.l $9100ba    ;)
  dc.l $a40001    ;*
  dc.l $a40001    ;+
  dc.l $b600f8    ;,
  dc.l $b6010b    ;-
  dc.l $b600c2    ;.
  dc.l $a40001    ;/
  ...
  dc.l $a40001   ;A <-- This is the one! Y: $00a4, X: $0001
  dc.l $a40014   ;B
  dc.l $a40027   ;C
  ...
```

Listing 5.1: The opening block of the `afont` data structure in source file `afont.s`.

Notice that the order in which we have defined the location of each character corresponds to the order (or value) of each character according to the ASCII specification. This means that when we define the text we want to display all we have to do is convert each character in the text to its corresponding ASCII value 'n' and get that nth value from our `afont` list.

So let's define the text we want to display:

```
ataricop1: dc.b "COPYRIGHT 1981,",0
ataricop2: dc.b "1994 ATARI CORP.",0
```

This is straightforward enough. Note that we have a zero at the end of each string. This makes them 'null-terminated' - a feature that will come in handy later when we need to identify that the end of the string when writing it to the screen.

Now that we have our text defined we can start the process of writing it to the screen. These two concise little paragraphs do this job for us. Given the amount of info encoded in our `afont` data structure we only need to set up one additional piece of info for the `centext` routine that will do our heavy lifting. This is the Y position that we want to write the text to on the screen (adjusted for whether we're on a PAL or NTSC screen).

```
    lea afont,a1        ; Load the font data structure
    lea ataricop1,a0    ; Load the first line of the copyright info
    move #190-8,d0      ; Set the Y position for the text
    add palfix2,d0      ; Adjust for PAL screens
    jsr centext         ; Write the text

    lea afont,a1        ; Load the font data structure
    lea ataricop2,a0    ; Load the second line of the copyright info
    move #207-5,d0      ; Set the Y position for the text
    add palfix2,d0      ; Adjust for PAL screens
    jsr centext         ; Write the text
```

The `centext` routine (short for 'center the text') we call into at the end of each paragraph above is where things start to get detailed. It is still not the tangled core of this particular beast but it is certainly a busier proposition that anything we've looked at so far. But we can settle our nerves for the moment because it is in fact doing something quite straightforward: it is setting up a buffer of data for the GPU to use. It stores this buffer at an address in the GPU RAM given by the `inbuf` variable. This buffer is a 36 byte array containing the 4-byte addresses of all the different bits of data the GPU will need to render the text. Here are all the bits and piece in the order that they appear in `inbuf` once `centext` is finished:

GPU RAM Address	Name	Description
$F0003F60	ataricop1	Address in RAM of the text to write
$F0003F64	afont	Address in RAM of the font data structure
$F0003F68	$00000000	Drop Shadow Vector X axis
$F0003F6C	$00000000	Drop Shadow Vector Y axis
$F0003F70	$00010000	Text Scale X axis
$F0003F74	$00010000	Text Scale Y axis
$F0003F78	$00000000	Text Shear X axis
$F0003F7C	$00000000	Text Shear Y axis
$F0003F84	$00B600A7	Y & X Co-ordinates to Write Text To
$F0003F80	$00000000	Drop Shadow Mode Enabled

The 10 4-byte addresses stored beginning in `inbuf` by `centext`.

In addition to the items of data we've collected so far we can see that we plug in a few extras in the `centext` routine. These are graphical effects such as the scaling to apply to each character, the dimensions of a drop shadow to apply on each, and finally the 'shear' to apply. 'Shear' distorts an object along the depth or 'Z' axis to make it appear like we are looking at it at an angle in 3 dimensions. The only one of these parameters we using is the scaling, for the rest we just plug in zero. But here's what the copyright text would look like with some shear applied:

Figure 5.3: What the characters look like when we apply a shear value of 800 on both the X and Y axes.

And here's what they looks like if we bump up the scale on the X and Y axis:

Figure 5.4: What the characters look like when we apply a scale value of 2000 instead of 1000. The pixellated, exploded appearance gives some hint that 'scale' here is not simply concerned with size but a specific 'explosion type' effect.

We'll encounter effective use of these features elsewhere in Tempest 2000 so won't go any further into them for now.

```
centext:
  move.l a0,d2           ; Store 'text' in d2 so we don't overwrite it.
  lea in_buf,a0          ; Use 'in_buf' as the buffer to write to.
  move.l d2,(a0)         ; Store 'text' in the first position.
  move.l a1,4(a0)        ; Store 'afont' in the 2nd position.
  move.l #0,8(a0)        ; Store 0 in the 3rd position.
  move.l #0,12(a0)       ; Store 0 in the 4th possition (Dropshadow vector).
  move.l #$10000,16(a0)  ; Store 10000 in the 5th position (X Text Scale).
  move.l #$10000,20(a0)  ; Store 10000 in the 6th position (Y Text Scale).
  move.l #0,24(a0)       ; Store 0 in the 7th position (X Text Shear).
  move.l #0,28(a0)       ; Store 0 in the 8th position (Y Text Shear).
  move.l #0,36(a0)       ; Store 0 in the 10th position.

  ; This paragraph is about figuring out the X pos we want to place
  ; the texta nd then storing that along with the Y pos in the 10th
  ; position in the buffer
  bsr g_textlength ; Get the length of the text in d2 and store it in d7.
```

```
    ; This uses the text length to center it along the X axis and
    ; store the result in d0;
    lsr #1,d7
    neg d7
    add #192,d7
    ; d0 contains the Y pos, so move it to the left hand side
    ; of the 4 byte word.
    swap d0
    move d7,d0       ; Move the X pos we've calculated into the right.
    move.l d0,32(a0) ; Move the X/Y pos in the 9th position.

    ; We're now ready to run the GPU.
    lea texter,a0
    jsr gpurun
    jmp gpuwait
```

Listing 5.2: The `centext` routine responsible for setting up the values in our GPU buffer.

`centext`, given above, stuffs all these items into a buffer of GPU RAM called `in_buf` and when that is done loads and runs a separate little GPU module called a 'shader'. In this case the name of the shader program is `texter`, giving us a little clue to its domain of expertise. This shader is a free-standing piece of code written in a slightly different flavour of 68000 assembly language. Despite it's name `texter` is contained in a source file called `stoat.gas`. (Nearly all of the shaders, like the main program source file, are named after beasts of one sort or another. There is no logic or convention that can be applied to the names. They're just a bit of fun.)

```
texter:
 .incbin  "bin/stoat.o"
```

`texter` is slightly more than just a 'writing strings to the screen' routine. In addition to deforming the text as we saw in the 'shear' example above, it can also 'explodes' the text using the 'scale' parameters. A comment near the top of `stoat.gas` (the source file for the `texter` routine) tells the possible inspiration for this effect:

```
;*****
;*
;* REX: Robotron explosion generator. Takes an image from the source screen
    and expands
;* it in X and Y, then uses a1_n to draw the resultant matrix of single
    pixels.
;*
;*  Provide: dest screen in gpu_screen, in_buf: 0=source image address
;*  4=source image start pixel address, 8=x and y size of source, 12=X scale
     (16:16),
;*  16=Y scale (16:16), 20=X shear (16:16), 24=Y shear,
;*  28=Mode (0=Top edge, 1=Centered), 32=Dest X and Y
;*
;*****
```

The 'robotron explosion' is the effect visible below. As you gradually dial up the X and Y scale parameters and call this shader at each iteration it can appear as though the text is 'exploding' outwards. There is no real mystery to how the shader writes our characters to the screen. It uses the same blitting operation we covered previously but with the difference that the preparation of the blitting command and all the special calculations we might want to do if adding shear, scaling and drop shadows in advance of calling the blitter are done from within the special purpose processor of the GPU rather than in the CPU. The reason for doing it this way is because the GPU is designed to be fast at a small number of arithmetic operations such as adding and multiplying so if we're going to do a lot of them and don't want to slow the rest of the game down, the GPU is the man for our job. We'll next take a look at the internals of the `texter` shader, including the relatively trivial task of blitting our characters to the screen, in the next chapter.

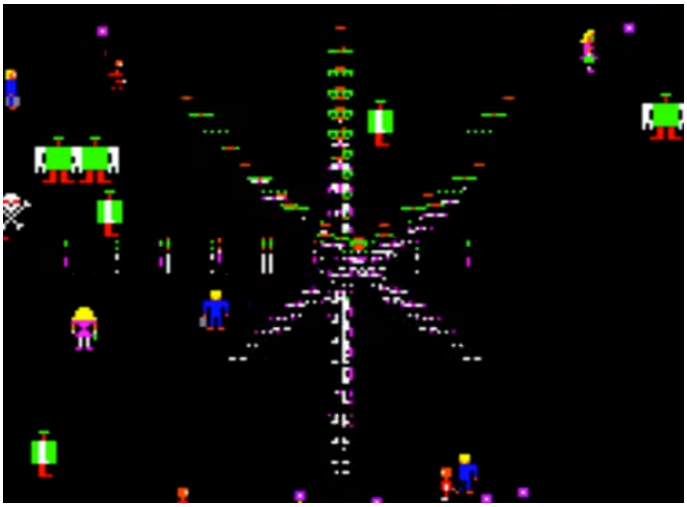


Figure 5.5: The explosion effect in 'Robotron 2084'.

my first shader

The busy heart of the `texter` shader is its `textloop`. This is a simple, but not especially short, piece of code that will write an individual character to the screen, dilating, distorting and deforming it as required. The `textloop` will keep doing this until the string in `textptr` runs out of characters, i.e. hits a `$00` byte such as at the end of `ataricop1`:

```
ataricop1: dc.b "COPYRIGHT 1981,",0
```

This 'have we run out of characters to paint' logic is at the very start of the loop:

```
textloop:
  loadb (textptr),r0 ; Put our text in r0.
  movei #ntxt,r1     ; Put our cleanup-and-exit routine in r1.
  addq #1,textptr    ; Move to the next character in textptr
  cmpq #0,r0         ; Check if it's $00
  jump eq,(r1)       ; If it is, jump to our cleanup-and-exit routine.
```

In this instance `textptr` is pointing to `ataricop1`. As soon as we hit the `0` at the very end we will `jump` to `ntxt`, which still stop the shader and bail out - unless 'drop shadow' mode was selected: in which case it will paint the string a second time (slightly offset) to achieve the dropshadow effect:

```
ntxt:
  cmpq #1,mode       ; In drop-shadow mode?
  jr ne,StopGPU      ; No, bail.
  nop
  moveq #0,mode      ; Turn off drop-shadow flag so we bail next time.
  movei #textloop,r0 ; Put our textloop routine in r0.
  movefa r6,textptr  ; Rest textptr to the start of the string.
```

```
  movefa r7,dstxy     ; Reset our destination position to its original.
  jump (r0)           ; Jump to r0 (textloop)
```

Listing 6.1: In fact there is no offset for drop-shadow mode implemented so the feature is unused. I suspect this is because the code is copy-pasted from elsewhere as we shall encounter a cousin of this routine later on.

Actually stopping the GPU involves clearing its command set to zero and loading that to the command register:

```
StopGPU:
  movei #G_CTRL,r1    ; Get our GPU commands.
  load (r1),r0        ; Load them to r0.
  bclr #0,r0          ; Clear the GPU flags in r0 to make it stop.
  store r0,(r1)       ; Actually stop the GPU by loading the cleared flags.
stoploop:
  jr stoploop         ; Spin until the GPU actually stops.
```

Now that we see how our loop through the text starts and exits, we will obviously be curious to know how the 'writing to the screen' part is effected. This is nearly identical to the method we stepped through for the 'Blitter'. The moment that pixels are written to the screen is in these statements at the end of `textloop`:

```
  movei #(SRCEN|CLIP_A1|UPDA1F|UPDA1|UPDA2|LFU_A|LFU_AN|DCOMPEN),r0
  ...
  store r0,(blit)     ;draw the sprite
```

Here `blit` is the equivalent of the `B_CMD` register we encountered in blitting. This is defined earlier on in the routine:

```
  movei #B_CMD,blit    ; Use blit as an alias for B_CMD
```

By writing the contents of the `r0` register to `blit` we are initiating a paint to the screen using all of the data we have setup earlier in the loop. This leads naturally to the question: what data? Well, not unlike 'Blitting' we set up a bunch of parameters and data points defining our 'source' (`A1`) and our 'destination'(`A2`). For both, these parameters and data points are concerned with things like positon, dimensions, screen width, and so on, not to mention where we are going to get the pixels for drawing from (in our case we already know that this is going to be from `pic`, the variable point to the contents of `beasty4.cry`).

We'll see how this data is calculated and derived shortly but first we'll take a look at the parameters and data points that are fed to the GPU and how. Below we have the GPU blitter registers that control the way our pixel data is written to the screen, in the order in which our `texter` routine writes them:

Common Name	T2K Name	Address in GPU RAM	Sample Value	Description
A1_PIXEL	a1_n+_pixel	$F0220C	00000001	Current X Position in the Pixel Data
A1_FPIXEL	a1_n+_nfpixel	$F02218	00000001	Current Y Position in the Pixel Data
A1_STEP	a1_n+_step	$F02210	00000001	Current Y Position in the Pixel Data
A1_FSTEP	a1_n+_fstep	$F02214	00000001	Current Y Position in the Pixel Data
A1_INC	a1_n+_inc	$F0221C	00000001	Current Y Position in the Pixel Data
A1_FINC	a1_n+_finc	$F02220	00000001	Current Y Position in the Pixel Data
A1_FLAGS	a1_n+_flags	$F02204	00000001	Current Y Position in the Pixel Data
A1_BASE	a1_n	$F02200	00000001	Current Y Position in the Pixel Data
A1_CLIP	a1_n+_clip	$F02208	00000001	Current Y Position in the Pixel Data

The GPU blitter registers in the order in which `texter` populates them in the listing below.

And here is the relevant code in the `texter` populating the registers:

```
   blit     REGEQU r13
   a1_n     REGEQU r14
   a2_n     REGEQU r15
   ...
rex:
   ...
   movei #A1_BASE,a1_n                          ; Make a1_n A1_BASE
   ...
   store xx,(a1_n+_pixel)                       ; Write to A1_PIXEL
   store yy,(a1_n+_fpixel)                      ; Write to A1_FPIXEL
   store r0,(a1_n+_step)                        ; Write to A1_STEP
   store r1,(a1_n+_fstep)                       ; Write to A1_FSTEP
   store xinc,(a1_n+_inc)                       ; Write to A1_INC
   store yinc,(a1_n+_finc)                      ; Write to A1_FINC

   movei #gpu_screen,r0          ; Prep gpu_screen as our destination
   movei #(PITCH1|PIXEL16|WID384|XADDINC),r1  ; Prep r1 for A1_FLAGS
   load (r0),r31                                ; Store gpu_screen in r31

   store r1,(a1_n+_flags)                       ; Write flags to A1_FLAGS
   movei #$1180180,r1                           ; Prep clip value.
   store r31,(a1_n)                             ; Write gpu_screen to A1_BASE
   store r1,(a1_n+_clip)                        ; Write clip value to A1_CLIP
```

This takes care of how we want the pixels to be written. On the other side of the equation we have to define for the GPU how want the source data in `beasty7.cry` to be read. Here is the order in which `texter` populates this other side of the scale:

Common Name	T2K Name	Address in GPU RAM	Value	Description
A2_BASE	a2_n	$F02224	_bass	Address of the source data
A2_PIXEL	a2_n+_pixel	$F02230	spixel	Address of the pixel data in `pic5/beasty7.cry`
A2_STEP	a2_n+_step	$F02234	Derived from ssize	The X and Y values for stepping
A2_FLAGS	a2_n+_flags	$F02228	Calculated	Current Y Position in the Pixel Data

The GPU blitter registers in the order in which `texter` populates them in the listing below.

And here we are doing the populating:

```
   store _bass,(a2_n)                    ; Write source address to A2_BASE
   store spixel,(a2_n+_pixel)            ; Write pixel data address to A2_PIXEL
   move ssize,r0                         ; Calculate the step size
   and lomask,r0
   neg r0
   and lomask,r0
   bset #16,r0                           ; Set the width of the character
   movei #(PITCH1|PIXEL16|WID320|XADDPIX),r1 ; Prep flags
   store r0,(a2_n+_step)                 ; Write step size to A1_STEP
   store r1,(a2_n+_flags)                ; Write flags to A1_FLAGS
```

So far we've worked backwards from writing to the screen and got to the point where we completed the necessary admin of stuffing our source (`A2`) and destination (`A1`) registers with the necessary odds and ends the GPU needs to write a character of our string to the screen. We've even received some clue as to what some of this data may actually mean.

The value in `_bass` that we write to `A2_BASE` is the value at the very start of our `afont` data structure, in other words `pic2`, the contents of `beasty7.cry` where our font pixels live. This is the blob of data the GPU will index into for its pixels to actually paint:

```
afont:
  dc.l pic2      ; pixel data
```

We loaded this from the `in_buf` passed to `texter` by first storing it in `fontbase` and then passing it into `_bass`:

```
   movei #in_buf,r0    ;load the parameters into registers
   load (r0),textptr    ;point at start of textstring
   addq #4,r0
   load (r0),fontbase    ;point to start of font datastructure
   ...
   load (fontbase),_bass
```

The value `spixel` we write to `A2_PIXEL` is the X and Y position of the character we want to paint in `beasty7.cry/pic2`. We get this value by indexing the ASCII value of our current character into our `afont` data structure.

```
textloop:
  loadb (textptr),r0 ; Put our text in r0.
  movei #ntxt,r1     ; Put our cleanup-and-exit routine in r1.
  addq #1,textptr    ; Move to the next character in textptr
  cmpq #0,r0         ; Check if it's $00
  jump eq,(r1)       ; If it is, jump to our cleanup-and-exit routine.
  nop                ; It's no zero so we can get the x/y value for it from
    afont.
```

```
    subq #32,r0          ;ASCII 0-31 do NOT print
    shlq #2,r0
    add fontbase,r0      ; Add the fontbase address to our offset so we get the
                         ; actual address afont
    load (r0),spixel     ; Write the actual address to spixel.
```

In the above, if our character is `A`, for example, this will result in us pulling out the following entry in `afont`

```
afont:
   ...
 dc.l $a40001 ;A
```

As we saw earlier this contains the Y (`a4`) and X (`01`) position of our `A` character in the `beasty7.cry` image:

Figure 6.1: The 'A' glyph located in a red box by our dimension and co-ordinate information.

unused stars

Tempest 2000 is full of starfields, from the title screen to the game itself - there's always a field of stars up to something in the background.

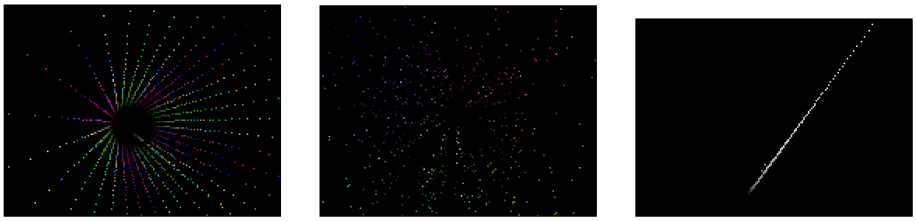


Figure 7.1: Example of starfield effects. The middle one is not used by the game.

We'll start in totally the wrong place by looking at piece of code that actually goes unused in the game. A routine called `initstarfield` that populates a random space of stars. This was probably a first iteration at drawing a star-spangled background and dropped later when fancier alternatives were developed. Before we look at the fancy, and necessarily more complex alternatives, let's take a look at what populating a random-ish starfield looks like.

What we have to do first is populate a data structure for our starfield. This is a list (or array) of elements with each element containing an X, Y, and Z co-ordinate plus a color value for the star. There are 127 such elements in total, and we store the number of elements as a 'header' or first entry in the array. We'll store this array in an address called `field1` and do the work of populating it in a routine called `initstarfield`.

The values we come up with for X and Y are totally random. All we do is come up with a number between 0 and 127 for the X and Y co-ordinates and a number between 0 and 512 for our Z co-ordinate. We then use the X and Y co-ordinates to seed a color value.

Here is the routine. As we said before, it ended up unused but has the merit of being relatively simple to read and understand:

```
initstarfield:
;
; initialise a starfield data structure for the GPU to display

  lea field1,a0      ; field1 is where the data structure is stored
  move #127,d7       ; The number of times to loop through 'isf' below.
  move.l #128,(a0)+ ; Store the number of stars at the start of field1.
  ; Create 128 stars and store them as an array in field1.
  ; Each star is: X,Y,Z,cry_index
isf:
  bsr rannum       ; Get a random number between 0 and 255 and store in d0
  move d0,d2       ; Stow d0 in d2 for use later
  sub #$80,d0      ; Get rid of the high bit so the num is between 0 and 128
  swap d0          ; Turn e.g. 00000032 into 00320000
  move.l d0,(a0)+ ; Store our rand num as our X co-ordinate.

  bsr rannum       ; Get a random number between 0 and 255  and store in d0
  move d0,d3       ; Stow d0 in d3 for use later
  sub #$80,d0      ; Get rid of the high bit so the num is between 0 and 128
  swap d0          ; Turn e.g. 00000032 into 00320000
  move.l d0,(a0)+ ; Store our rand num as our Y co-ordinate.

  bsr rannum       ; Get a random number between 0 and 255  and store in d0
  asl #1,d0        ; Multiply the number by 2 so that its between 0 and 512
  swap d0          ; Turn e.g. 00000032 into 00320000
  move.l d0,(a0)+ ; Store our rand num as our Z co-ordinate.

  ; Use our X and Y co-ordinates to come up with a random color for the star
    .
  ; So if X(d2) is 00000032 and Y(d3) is 00000088:
  and #$f0,d2   ; 00000032 -> 00000030
  lsr #4,d3     ; 00000088 -> 80000008
  and #$0f,d3   ; 80000008 -> 00000008
  or d2,d3      ; 00000030 or 00000008 -> 00000038
  lsl #8,d3     ; 00000038 -> 00003800
  move d3,(a0) ; Store our color value.

  lea 20(a0),a0   ; Move a0 20 bytes ahead ready for the next element.
  dbra d7,isf     ; Loop until d7 is 0.
  rts
```

Listing 7.1: Populating an unused starfield data structure. This is a fuzzier version of the ring starfield used in the credits screen.

This is our data structure (or at least the first 2 elements in it) in a table:

Value	Decimal	Description
0000000E	127	Number of Elements
00064000	127	Element 1: X co-ordinate
00064000	127	Element 1: Y co-ordinate
00064000	127	Element 1: Z co-ordinate
00064000	127	Element 1: Color Value
00064000	127	Element 2: X co-ordinate
00064000	127	Element 2: Y co-ordinate
00064000	127	Element 2: Z co-ordinate
00064000	127	Element 2: Color Value

First 2 elements of the data structure created by `initstarfield`.

And this is what our data structure looks like when its animated by our shader `fastvector` (more of which later).

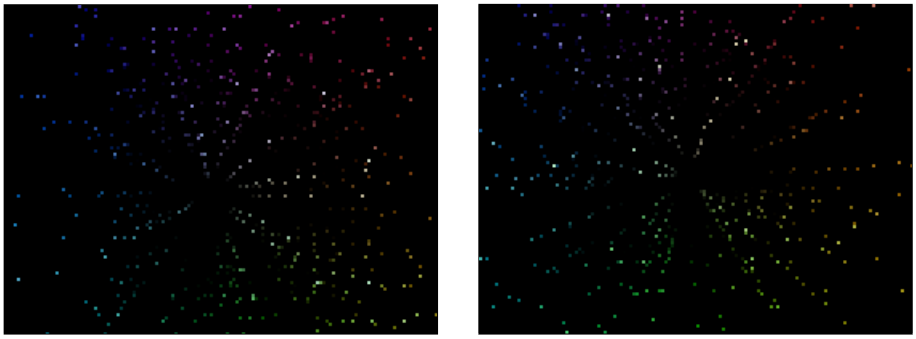


Figure 7.2: Frames of the unused starfield during animation.

The actual starfield used in the credits screen makes use of a sine table to accurately calculate the co-ordinates of pixels in an expanding ring structure.

```
sines:
dc.w $0003,$0609,$0C0F,$1215,$181B,$1E21,$2427,$2A2D
dc.w $3033,$3639,$3B3E,$4144,$4649,$4B4E,$5053,$5557
dc.w $595C,$5E60,$6264,$6667,$696B,$6D6E,$7071,$7274
dc.w $7576,$7778,$797A,$7B7B,$7C7D,$7D7E,$7E7E,$7E7E
dc.w $7E7E,$7E7E,$7E7E,$7D7D,$7C7B,$7B7A,$7978,$7776
dc.w $7573,$7271,$6F6E,$6C6B,$6967,$6563,$615F,$5D5B
dc.w $5957,$5552,$504D,$4B48,$4643,$403E,$3B38,$3533
dc.w $302D,$2A27,$2421,$1E1B,$1815,$120F,$0C08,$0502
```

```
dc.w $00FD,$FAF7,$F4F0,$EDEA,$E7E4,$E1DE,$DBD8,$D5D2
dc.w $CFCD,$CAC7,$C4C1,$BFBC,$B9B7,$B4B2,$B0AD,$ABA9
dc.w $A6A4,$A2A0,$9E9C,$9A98,$9795,$9392,$908F,$8D8C
dc.w $8B8A,$8988,$8786,$8584,$8483,$8382,$8282,$8282
dc.w $8282,$8282,$8283,$8383,$8485,$8586,$8788,$898A
dc.w $8B8D,$8E8F,$9192,$9496,$9799,$9B9D,$9FA1,$A3A5
dc.w $A7AA,$ACAE,$B1B3,$B6B8,$BBBD,$C0C3,$C5C8,$CBCE
dc.w $D1D4,$D7D9,$DCDF,$E2E6,$E9EC,$EFF2,$F5F8,$FBFE
dc.w $0000,$0192,$0323,$04B5,$0645,$07D5,$0963,$0AF0
dc.w $0C7C,$0E05,$0F8C,$1111,$1293,$1413,$158F,$1708
dc.w $187D,$19EF,$1B5C,$1CC5,$1E2A,$1F8B,$20E6,$223C
dc.w $238D,$24D9,$261F,$275F,$2899,$29CC,$2AFA,$2C20
dc.w $2D40,$2E59,$2F6B,$3075,$3178,$3273,$3366,$3452
dc.w $3535,$3611,$36E4,$37AE,$3870,$3929,$39DA,$3A81
dc.w $3B1F,$3BB5,$3C41,$3CC4,$3D3D,$3DAD,$3E14,$3E70
dc.w $3EC4,$3F0D,$3F4D,$3F83,$3FB0,$3FD2,$3FEB,$3FFA
dc.w $3FFF,$3FFA,$3FEB,$3FD2,$3FB0,$3F83,$3F4D,$3F0D
dc.w $3EC4,$3E70,$3E14,$3DAD,$3D3D,$3CC4,$3C41,$3BB5
dc.w $3B1F,$3A81,$39DA,$3929,$3870,$37AE,$36E4,$3611
dc.w $3535,$3452,$3366,$3273,$3178,$3075,$2F6B,$2E59
dc.w $2D40,$2C20,$2AFA,$29CC,$2899,$275F,$261F,$24D9
dc.w $238D,$223C,$20E6,$1F8B,$1E2A,$1CC5,$1B5C,$19EF
dc.w $187D,$1708,$158F,$1413,$1293,$1111,$0F8C,$0E05
dc.w $0C7C,$0AF0,$0963,$07D5,$0645,$04B5,$0323,$0192 ; <-- sine ($0C7C)
dc.w $0000,$FF6E,$FDDD,$FC4B,$FABB,$F92B,$F79D,$F610
dc.w $F484,$F2FB,$F174,$EFEF,$EE6D,$ECED,$EB71,$E9F8
dc.w $E883,$E711,$E5A4,$E43B,$E2D6,$E175,$E01A,$DEC4
dc.w $DD73,$DC27,$DAE1,$D9A1,$D867,$D734,$D606,$D4E0
dc.w $D3C0,$D2A7,$D195,$D08B,$CF88,$CE8D,$CD9A,$CCAE
dc.w $CBCB,$CAEF,$CA1C,$C952,$C890,$C7D7,$C726,$C67F
dc.w $C5E1,$C54B,$C4BF,$C43C,$C3C3,$C353,$C2EC,$C290
dc.w $C23C,$C1F3,$C1B3,$C17D,$C150,$C12E,$C115,$C106 ; <-- cosine ($C23C)
dc.w $C101,$C106,$C115,$C12E,$C150,$C17D,$C1B3,$C1F3
dc.w $C23C,$C290,$C2EC,$C353,$C3C3,$C43C,$C4BF,$C54B
dc.w $C5E1,$C67F,$C726,$C7D7,$C890,$C952,$CA1C,$CAEF
dc.w $CBCB,$CCAE,$CD9A,$CE8D,$CF88,$D08B,$D195,$D2A7
dc.w $D3C0,$D4E0,$D606,$D734,$D867,$D9A1,$DAE1,$DC27
dc.w $DD73,$DEC4,$E01A,$E175,$E2D6,$E43B,$E5A4,$E711
dc.w $E883,$E9F8,$EB71,$ECED,$EE6D,$EFEF,$F174,$F2FB
dc.w $F484,$F610,$F79D,$F92B,$FABB,$FC4B,$FDDD,$FF6E
```

A table like this allows us to 'hard code' the points of an arbitrarily sized circle by providing the sine and cosine values for each point. To get the X and Y co-ordinates of a specific point in the circle you pull out the corresponding sine and cosine values from the table, multiply each by your desired radius, and that gives you your X and Y value to draw your point on the screen. You do that for each point (say 32 in total) and you have a ring of dots that you can connect to form a circle.

$$x = \cos(\theta) * radius \tag{7.1}$$

$$y = \sin(\theta) * radius \tag{7.2}$$

The `sines` table seems inscrutable at first glance but the specific use of it made by the `ringstars` routine is straightforward. The routine plots a circle of 32 points and for each point it uses the current point number (e.g. 31) to pull out a sine and cosine value from the first entry in the corresponding row on the table. If we imagine we're plotting the 31st point this means for the sine value, we will take the first value from row 31, and for the cosine value it will take the first value 8 rows after that. As you can see in the above table this corresponds to `$0C7C` for the sine and `$C23C` for the cosine.

With this in hand, and a radius of 200 pixels, we can calculate our X and Y co-ordinates. The steps are the same for both the X and Y values, except for the use of `sine` and `cosine`:

Step No.	Description	Cosine	Sine
1	Value taken from `sine` table	`$0C7C`	`$C23C`
2	Use last byte only	`$007C`	`$003C`
3	Multiply by radius (200)	`$60E0`	`$2EE0`
4	Shift Left by 7 bits	`$00307000`	`$00177000`
5	Final value treated as 16:16 fraction	X:`48.28672`	Y:`23.28672`

Steps to calculate the X and Y value for Point 31.

The whole process is admittedly convoluted but the last step may seem especially mysterious. In order to treat the X and Y values as fractions rather than whole numbers, the blitter will split our final value in half and treat the left-hand side as a whole number and the right-hand side as a fraction. So `$00307000`, for example, becomes `$0030` and `$7000`, which are `48` and `28672` in decimal respectively, giving us a decimal fractional value of `48.28672`.

There is another complication we have glossed over in Step 2 above. Since sine and cosine values can be positive or negative we have to know how +/- values are indicated here. The answer is that any value of `$80` or above is treated as negative: so `$FE` is -1, `$FD` is -2, all the way down to `$81` which is -127 and `$8000` which is -128. In our example both `$7C` and `$3C` are less than `$80` so both are treated as positive.

Here then is the first half of the `ringstars` routine where the process we've outlined is implemented:

```
ringstars:
;
; 'initialise a starfield of 8 rings of 64 stars each': it says this,
; but its actually 8 rings of 32 stars each.
```

```
   move #200,d5        ; the radius we will use for the ring
rst:
   lea field1,a0       ; field1 is where we'll store it
   lea sines,a1        ; our sine table (see later) in a1
   lea p_sines,a2      ; our positive-only sine table in a2
   move #7,d7          ; d7 will track our 8 rings
   move.l #256,(a0)+   ; header of our structure: no. of stars (256)

   move #$0000,d4      ; d4 will contain the star color
ring1:
   move #32,d6         ; d6 will track no. of stars per ring (32)
ring2:
   move d6,d0          ; Put no. of current star in d0
   asl #3,d0           ; Multiply by 8 to get the row in our sine table
   move.b 0(a1,d0.w),d1  ; Get the sine from position d0 in our sine table
   add.b #$40,d0       ; Add offset to index the cos in our sine table
   move.b 0(a1,d0.w),d2  ; Get the cos from position d0 in our sine table.
   ext d1              ; Chop off everything but the last byte for sine.
   ext d2              ; Chop off everything but the last byte for cos.
   muls d5,d1          ; Get Y by multiplying sin * radius
   muls d5,d2          ; Get X by multiplying cos * radius
   asl.l #7,d1         ; Shift left 7 bits to create an X value
   asl.l #7,d2         ; Shift left 7 bits to create a Y value
   move.l d1,(a0)+     ; Move X into our star data structure.
   move.l d2,(a0)+     ; Move Y into our star data structure.
```

Listing 7.2: Populating the data structure for the starfield used in the Tempest 2000 credits screen.

And here is what plotting these points ourselves looks like. Iterating above for a full 32 points, pulling values from our `sines` table as we go, gives us a simple circle as expected:

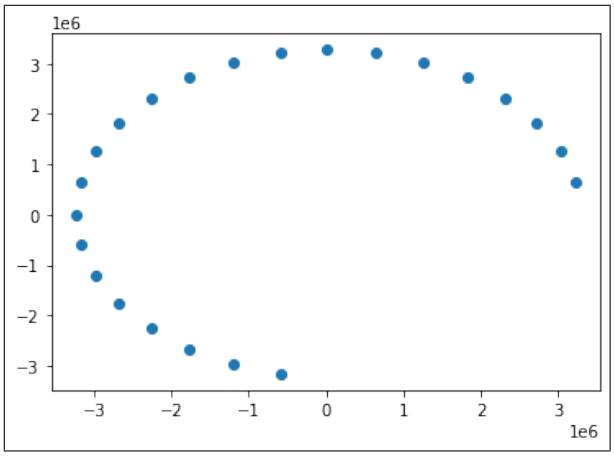


Figure 7.3: Each of the 32 points as plotted using the X/Y values calculated using the procedure above.

The remainder of the routine calculates a Z value for our point in 3D space and this one is even more intricate.

```
   ; Calculate the Z value.
   lsl #2,d0
   and #$ff,d0
   move.b 0(a1,d0.w),d1
   move.b 0(a2,d0.w),d2
   and #$f0,d2
   lsl #4,d2
   ext d1
   bpl sposss
   neg d1
sposss:
   swap d1
   clr d1
   asr.l #1,d1
   clr.l d0
   move d7,d0
   swap d0
   lsl.l #6,d0    ;Z position according to ring no.
   add.l d1,d0
   move.l d0,(a0)+ ; Move Z into our star data structure

   move d4,d0
   add d2,d0
   move d0,(a0)    ;colour
   lea 20(a0),a0
   dbra d6,ring2
   add #$2000,d4
   dbra d7,ring1
   rts
```

Listing 7.3: Calculating the Z and color value for the starfield data structure used in the Tempest 2000 credits screen.

sounds

The sound effects in Tempest 2000 are stpred in the game as Pulse Code Modulation (PCM) data. This is the most common format for storing raw audio and flavors of it are used in everything from CDs to telephony. PCM data is as simple as a stream of numbers. Each of the numbers is between a lower and upper limit:, for example between -128 and 128. If you plot enough of these numbers along a graph you get what looks like a wave. This is pretty much the sound wave, the analog version of our digital values. If you put these values through the right piece of hardware, such as a Digital Analog Converter (DAC) you can get a sound out the other side.

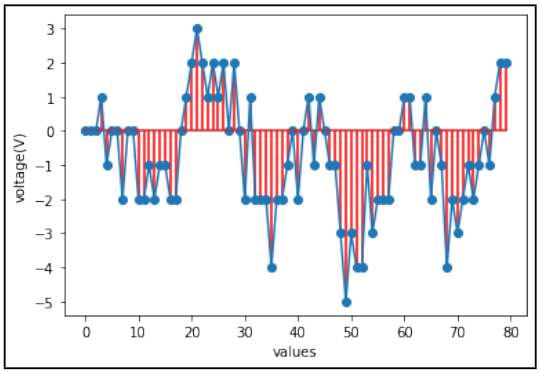


Figure 8.1: A sample of 80 bytes plotted as a wave graph.

The PCM data in Tempest 2000's sound samples is nothing more than this series of bytes, each one representing a value between -128 and 128, which can be converted

into sound. A single series of bytes will give us one 'channel' of sound, i.e. a mono sound sample. What we actually have are stereo sound samples so each sample contains two channels. The simplest way of incorporating two separate channels in the sample (one for the left speaker and one for the right) is to interleave them, in alternating bytes. We can see that this is the way the Tempest samples are encoded if we split out an arbitrary sample into separate streams of alternate bytes and graph them.

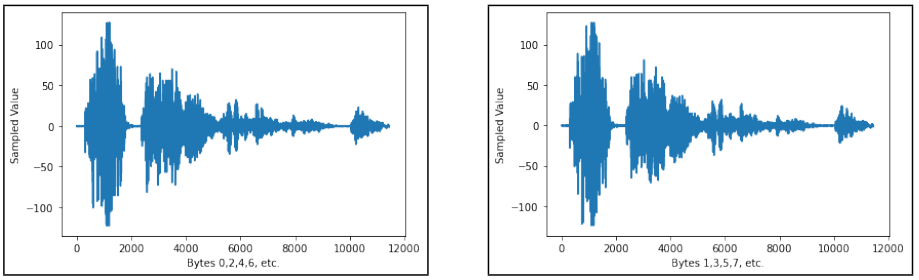


Figure 8.2: Plotted sound waves for the left and right channels respectively.

Since the sounds we want to play are just blobs of data, we need a way of knowing where these blobs are, and perhaps some information on what we have to do with them. This is provided by `samtab`, a block of data at address `$9ac800` that tells us everything we need to know:

```
samtab        EQU $9ac800
```

Just byt itself, the `samtab` table reveals some interesting potential for ways that we can use the raw PCM bytes.

```
;                      Prio-                                    Repeat     Repeat
;Name                  rity   Period Start     Length        -- Start      Length
;--------------------- -----  ------ --------- ------- --  --- --------- ------- ---
'Engine Noise 1      ', $0001, $01ac, $009acd00, $0011a0, $00, $009acd02, $00119e, $00
'Player Shot Normal 2', $0002, $01ac, $009adea4, $0008e8, $00, $009adea4, $000000, $00
'Engine Noise        ', $0003, $01ac, $009ae290, $003378, $00, $009aee9e, $002594, $00
'Player Death        ', $0004, $00d6, $00000000, $00549a, $00, $00000000, $000000, $00
'Player Death 2      ', $0005, $01ac, $00000000, $002458, $00, $00000000, $000000, $00
'Player Shot Normal  ', $0006, $01ac, $009b160c, $0007a4, $00, $009b160c, $000000, $00
'Player Jump         ', $0007, $01ac, $009b1db4, $0018de, $00, $009b1db4, $000000, $00
'Crackle             ', $0008, $00d6, $009b3696, $004594, $00, $009b3696, $000000, $00
'Cleared Level       ', $0009, $01ac, $009b7c2e, $0037a2, $00, $009b7c2e, $000000, $00
'Warp                ', $000a, $0238, $009bb3d4, $006ec8, $00, $009bb3d4, $000000, $00
'Large Explosion     ', $000b, $01ac, $009c22a0, $0050c2, $00, $009c22a0, $000000, $00
'Powered Up Shot     ', $000c, $01ac, $009c7366, $001976, $00, $009c7366, $000000, $00
'Get Power Up        ', $000d, $01ac, $009c8ce0, $001aea, $00, $009c8ce0, $000000, $00
'Tink For Spike      ', $000e, $00fe, $009ca7ce, $00040e, $00, $009ca7ce, $000000, $00
'NME At Top Of Web   ', $000f, $01ac, $009cabe0, $00001e, $00, $009cabe0, $000000, $00
'Pulse For Pulsar    ', $0010, $0358, $009cac02, $0019fe, $00, $009cac02, $000000, $00
'Normal Explosion    ', $0011, $00d6, $009cc604, $002ab6, $00, $009cc604, $000000, $00
'Extra Explosion     ', $0012, $0358, $009cf0be, $0018ca, $00, $009cf0be, $000000, $00
```

```
'Static or Pulsar    ', $0013, $011c, $009d098c, $003fe4, $00, $009d098c, $000000, $00
'Pulsar Pulse        ', $0014, $0358, $009d4974, $000f0c, $00, $009d4974, $000000, $00
'Off Shielded NME    ', $0015, $00aa, $009d5884, $0027ca, $00, $009d5884, $000000, $00
'Excellent           ', $0016, $0200, $009d8052, $005976, $00, $009d8052, $000000, $00
'Superzapper Recharge', $0016, $0200, $009dd9cc, $00a958, $00, $009dd9cc, $000000, $00
'yes                 ', $0018, $0200, $009e8328, $005a6c, $00, $009e832a, $005a6a, $00
'oneup               ', $0019, $0200, $009edd98, $0043ae, $00, $009edd98, $000000, $00
'screeeam            ', $001a, $0200, $009f214a, $004568, $00, $009f214a, $000000, $00
'sexy yes 1          ', $001b, $0200, $009f66b6, $002c54, $00, $009f66b6, $000000, $00
'sexy yes 2          ', $001c, $0200, $009f9362, $003236, $00, $009f9362, $000000, $00
'tink                ', $001e, $0200, $009fc59c, $0005ce, $00, $009fc59c, $000000, $00
'zero                ', $001f, $0200, $009fcb6e, $000008, $00, $009fcb6e, $000000, $00
'dummy               ', $0020, $0200, $009fcb7a, $00a1d8, $00, $009fcb7a, $000000, $00
```

Listing 8.1: The contents of `samtab` at `$9AC800`.

Of course the best way of seeing how this works is to follow one through in practice. Here is the little routine that shouts 'Excellent!' when you navigate the options menu. We can see that we do a little set up such as selecting the sample from the table above ('Excellent' is at position 21 (`$16` in hex)), setting its pitch and then calling another routine called `fox` that inches nearer the actual work of playing the sample:

```
; ******************************************************************
; Say 'Excellent', e.g. when the user naviates to an option.
; ******************************************************************
sayex:
  move #21,sfx            ; Select effect $16 (21) from samtab, 'Excellent'.
  move #101,sfx_pri       ; Set the priority.
  ;move #$ff,sfx_vol      ; Setting the volume is commented out.
  move.l #$160,sfx_pitch  ; Set the pitch.
  jsr fox                 ; Play the effect (see below).
  move #101,sfx_pri       ; Do it again.
  ;move #$ff,sfx_vol
  move.l #$162,sfx_pitch  ; This time with a different pitch.
  jmp fox                 ; Play it.
```

`fox` (why waste an opportunity to pun on something in the animal kingdom) sets up our data for playing the sample. It's main preoccupation is turning our index into the `samtab` table to a pointer to the data there.

```
; ******************************************************************
; Play a selected sound sample.
; ******************************************************************
fox:
  movem.l d0-d3/a0,-(a7)  ; Save d0,d1,d2,d3,a0 to the a7 register.
                          ; Because we're going to clobber them here.
  move sfx,d0             ; Store sfx in d0, this is '$16' for 'Excellent'.

  ; Use '21' to point to the address of sample data, i.e. directly after
  ; the 'Excellent    ' in this entry in samtab:
  ; 'Excellent           ', $0016, $0200, $009d8052, $005976, $00,

  ; Since each entry in samtab is 40 bytes long, multiplying our index by
```

```
; 40 will bring us to the start of the 'Excellent' entry. The steps below
; are a fast way of doing this multiplication by 40.
lsl #3,d0    ; Turn $16(21) into $a8 by left-shifting.
move d0,d1   ; Store in d1.
lsl #2,d0    ; Turn $a8 into $2a0 by left-shifting.
add d1,d0    ; d0 is now $348 (840), i.e. 21 * 40.

; Now that we have an offset for our sample information in samtab, we'll
; add 20 to it (so that it's after the sample name) and store it in a0.
lea samtab,a0           ; Store samtab address in a0.
lea 20(a0,d0.w),a0      ; Add 840 + 20 to it.

; Now we can set up the last few items our sound synth needs.
move sfx_pri,d1
move sfx_vol,d2
move.l sfx_pitch,d3

; Invoke the sound synth to play our sample.
jsr PLAYFX2

; Clean up.
move d0,handl
clr sfx_pri
clr sfx_vol
clr.l sfx_pitch

movem.l (a7)+,d0-d3/a0 ; Restore the old unclobbered values we saved off
                       ; at the start.
rts
```

To recap, at this point we have set up our registers as follows:

Register	Value	Description
a0	$9acb5c	Address of the 'Excellent' metadata in `samtab`.
d1	101	`sfx_pri`.
d2	$ff	`sfx_vol`: the volume to play at.
d3	$162	`sfx_pitch`: the pitch to play at.

These are the garnish for a choice piece of mystery meat, `PLAYFX2`. `PLAYFX2` is a library routine provided to Atari by a crowd called 'Imagitec Designs'. It is not something we have the source code for, and nor did Minter when he was writing Tempest 2000. Instead, he was given a binary blob called `syn6.o` and a list of routine names to use when he wanted to play one of his samples. This list is given at the start of the main `yak.s` source file:

```
INIT_SOUND     EQU  $4040  ;jump table for the SFX/Tunes module
NT_VBL         EQU  $4046
PT_MOD_INIT    EQU  $404c
```

```
START_MOD      EQU  $4052
STOP_MOD       EQU  $4058
PLAYFX2        EQU  $405e
CHANGE_VOLUME  EQU  $4064
SET_VOLUME     EQU  $406a
NOFADE         EQU  $4070
FADEUP         EQU  $4076
FADEDOWN       EQU  $407c
ENABLE_FX      EQU  $4082
DISABLE_FX     EQU  $4088
CHANGEFX       EQU  $409a  ;new in syn6
HALT_DSP       EQU  $408e
RESUME_DSP     EQU  $4094
intmask        EQU  $40a0
```

These are all entry points in the game binary for each routine. So for `PLAYFX2` the entry point is `$405e`. In the source code this routine exists as raw machine code only. Along with all the other Imagitec sound routines it lives in a file called `moomoo.dat`:

```
.include "moomoo.dat"
```

This is a slightly modified version of the `syn6.o` sound synthesizer binary provided by Imagitec Designs. With modern tooling, it is possible for us to recreate `moomoo.dat` by partially disassembling it, then reassembling and relinking it. This will more or less re-enact the build steps Minter would have followed to generate it.

The initial bytes of `moomoo.dat` can be disassembled as follows into a file we'll call `moomoo_header.s`.

```
; Header for moomoo.s
;.org $4040
.include "jaguar.inc"

init_sound     EQU $000041ba
nt_vbl         EQU $0000425e
pt_mod_init    EQU $000041d6
start_mod      EQU $000045e2
stop_mod       EQU $00004606
playfx2        EQU $00004370
change_volume  EQU $000044ea
set_volume     EQU $00004546
nofade         EQU $00004598
fadeup         EQU $000045b2
fadedown       EQU $000045ca
enable_fx      EQU $0000464e
disable_fx     EQU $00004660
resume_dsp     EQU $00004470

INIT_SOUND:     jmp     (init_sound).l
NT_VBL:         jmp     (nt_vbl).l
```

```
PT_MOD_INIT:    jmp     (pt_mod_init).l
START_MOD:      jmp     (start_mod).l
STOP_MOD:       jmp     (stop_mod).l
PLAYFX2:        jmp     (playfx2).l
CHANGE_VOLUME:  jmp     (change_volume).l
SET_VOLUME:     jmp     (set_volume).l
NOFADE:         jmp     (nofade).l
FADEUP:         jmp     (fadeup).l
FADEDOWN:       jmp     (fadedown).l
ENABLE_FX:      jmp     (enable_fx).l
DISABLE_FX:     jmp     (disable_fx).l
CHANGEFX:       jmp     (disable_fx).l
HALT_DSP:       jmp     (disable_fx).l
RESUME_DSP:     jmp     (resume_dsp).l

intmask:        dc.b 0
; Address 0x40A2
return_early:
                rts

; Address 0x40A4
update_interrupt:
                move.w  d0,-(sp)
                bset    #3,(intmask).l
                clr.w   d0
                move.b  (intmask).l,d0
                move.w  d0,(INT1).l ; RW CPU Interrupt Control Register
                move.w  (sp)+,d0
                rts
```

Listing 8.2: The contents of `moomoo_header.s`.

It turns out that the address we're using to invoke `PLAYFX2` simply jumps to another adress in the binary, one that contains the actual `playfx2` routine:

```
PLAYFX2:        jmp     (playfx2).l
```

So to recreate the steps used to generate `moomoo.dat` we first assemble this file:

```
rmac -fa -i moomoo_header.s -o moomoo_header.o
```

Then we link `moomoo_header.o` with the `syn6.o` file provided by Imagitec:

```
rln -n -z -u -v -a 4040 xd xd -e moomoo_header.o syn6.o -o moomoo.dat
```

Voila, we have our `moomoo.dat`. If you'd like to follow these steps yourself, you can use this Python notebook.

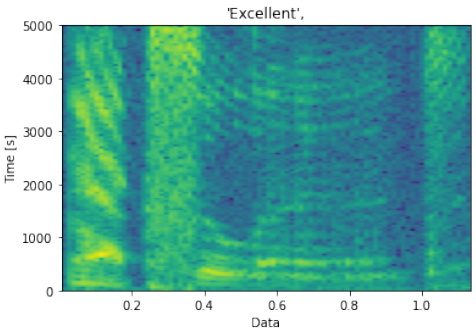
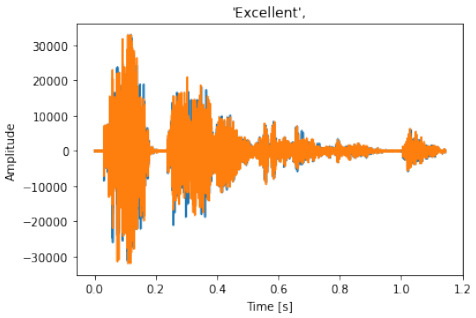
Spectrogram	Amplitude	Sound
		

Figure 8.3: Spectrogram and Sine Plot of the 'Excellent' sound sample. On a PDF viewer that supports, it you can click 'Play' to hear it.

flipper

The data structure as given in the source code is:

```
s_flipper:
  dc.l 4          ; 4 faces in this object, a shaded solid Flipper

 dc.w $f0         ; Face colour - Red.
 dc.w 3,$8000     ; vertex index, intensity.
 dc.w 1,$c000     ; vertex index, intensity.
 dc.w 0,$ff00     ; vertex index, intensity.
 dc.w 0

 dc.w $f4         ; Face colour - Orange.
 dc.w 4,$4000     ; vertex index, intensity.
 dc.w 2,$8000     ; vertex index, intensity.
 dc.w 0,$c000     ; vertex index, intensity.
 dc.w 0

 dc.w $f4         ; Face colour - Red.
 dc.w 5,$8000     ; vertex index, intensity.
 dc.w 1,$c000     ; vertex index, intensity.
 dc.w 0,$ff00     ; vertex index, intensity.
 dc.w 0

 dc.w $f0         ; Face colour - Orange.
 dc.w 6,$4000     ; vertex index, intensity.
 dc.w 2,$8000     ; vertex index, intensity.
 dc.w 0,$c000     ; vertex index, intensity.
 dc.w 0

fverts: dc.w 9,9,5,9,13,9,1,0,17,0,1,18,17,18
```

A little more explanation, taking the first entry in the structure:

```
dc.w $f0    ; Face colour - Red.
dc.w 3,$8000 ; Index into fverts, followed by intensity value.
dc.w 1,$c000 ; Index into fverts, followed by intensity value.
dc.w 0,$ff00 ; Index into fverts, followed by intensity value.
dc.w 0        ; Padding that rounds up the entry to 16 bytes
```

We can see that the middle three entries start with an index into the array `fverts` we listed above. The index is pair-wise, so for example an index of `3` must be multiplied by 2, and then counting into `fverts` starting from 0 we end up at the 7th element and the pair: `1,0`. We can make this more explicit by breaking out the presentation of `fverts` to match the way it's used:

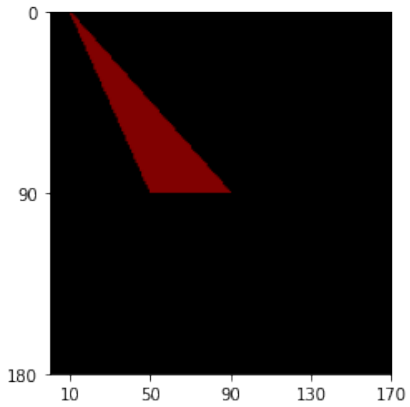
```
fverts: dc.w 9,9   ; Index 0
        dc.w 5,9   ; Index 1
        dc.w 13,9  ; Index 2
        dc.w 1,0   ; Index 3
        dc.w 17,0  ; Index 4
        dc.w 1,18  ; Index 5
        dc.w 17,18 ; Index 6
```

What are these pairs of numbers? They are (x,y) co-ordinates of course. It turns out `vertex` is just a fancy term for a point in two-dimensional space. So we can now begin to discern what this data structure is: it contains 4 sets of 3 vertices each. If you have 3 points in space, you have a triangle. So what we have here is a description of 4 triangles and each triangle is beign referred to as a 'face'.

Let's use the indices in the first entry to see what the first face gives us. We already saw how to get from an index of `3` to a vertex of `1,0` above. When we do the other two the three pairs of co-ordinates turn out as: to be as follows:

Index	X	Y	Description
3	1	0	First vertex.
1	5	9	Second Vertex.
0	9	9	Third Vertex.

With these, and our specified colour of red (`$f0`), we can make a triangle:




```
        dc.w $f0        ; Colour - Red.
        dc.w 3,$8000    ; index, intensity.
        dc.w 1,$c000    ; index, intensity.
        dc.w 0,$ff00    ; index, intensity.
        dc.w 0
```

```
        dc.w 9,9    ; Index 0
        dc.w 5,9    ; Index 1
        dc.w 1,0    ; Index 3
```

Figure 9.1: Our three vertices joined together in a triangle. Notice that we've scaled up our co-ordinates by 10. So `(9,9)`, for example, is given as `(90,90)`.

One triangle is all very nice, but we have a flipper to build. Should we try a second triangle? Let's see how that goes.

The instructions for our new triangle are in the second paragraph of the `s_flipper` data structure:

```
dc.w $f4        ; Face colour - Orange.
dc.w 4,$4000    ; vertex index, intensity.
dc.w 2,$8000    ; vertex index, intensity.
dc.w 0,$c000    ; vertex index, intensity.
dc.w 0
```

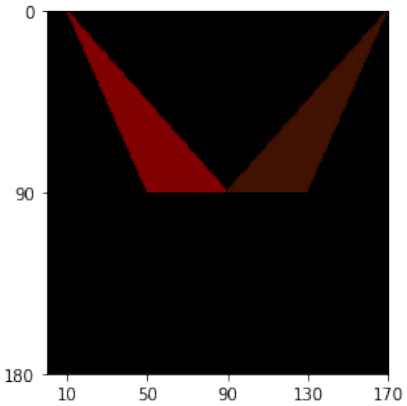
When we translate this to co-ordinates from `fverts` we get:

Index	X	Y	Description
4	17	0	First vertex.
2	13	9	Second Vertex.
0	9	9	Third Vertex.

Not wildly different than our previous effort. Putting our vertices for the two faces together in a single table we get:

Face	Vertex 1	Vertex 2	Vertex 3
1	1,0	5,9	9,9
2	17,0	13,9	9,9

Let's see what we get when we add it in:




```
dc.w $f4        ; Colour - Orange.
dc.w 4,$4000    ; index, intensity.
dc.w 2,$8000    ; index, intensity.
dc.w 0,$c000    ; index, intensity.
dc.w 0
```

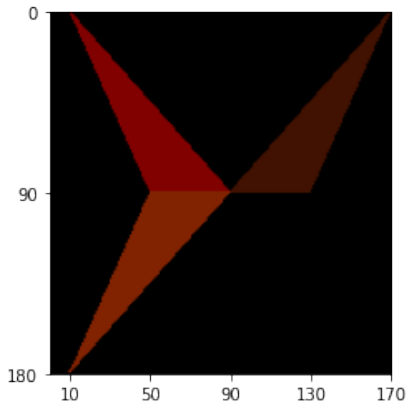
```
dc.w 9,9    ; Index 0
dc.w 13,9   ; Index 2
dc.w 17,0   ; Index 4
```

Figure 9.2: Adding in our second triangle

OK, things are starting to take shape. Here are the vertices for all four faces together:

Face	Vertex 1	Vertex 2	Vertex 3
1	1,0	5,9	9,9
2	17,0	13,9	9,9
3	1,18	5,9	9,9
4	17,18	13,9	9,9

Let's see what it looks like when we add in the third triangle:

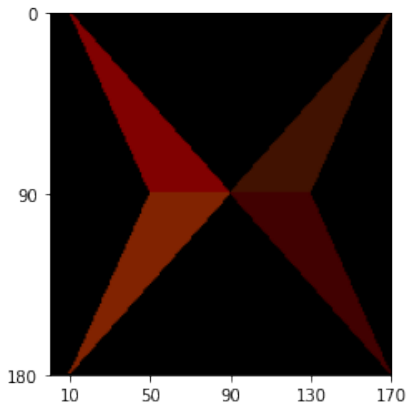



```
dc.w $f4        ; Colour - Red.
dc.w 5,$8000    ; index, intensity.
dc.w 1,$c000    ; index, intensity.
dc.w 0,$ff00    ; index, intensity.
dc.w 0
```

```
dc.w 9,9    ; Index 0
dc.w 5,9    ; Index 1
dc.w 1,18   ; Index 5
```

Figure 9.3: Adding in our third triangle

And the fourth and final triangle:




```
dc.w $f0        ; Colour - Orange.
dc.w 6,$4000    ; index, intensity.
dc.w 2,$8000    ; index, intensity.
dc.w 0,$c000    ; index, intensity.
dc.w 0
```

```
dc.w 9,9    ; Index 0
dc.w 13,9   ; Index 2
dc.w 17,18  ; Index 6
```

Figure 9.4: Adding in our final triangle

```
draw_sflipper:
  lea s_flipper,a1
  bra drawsolidxy
```

```
drawsolidxy:
  lea in_buf,a0
```

```
    move.l a1,(a0)+ ; pointer to our s_flipper data structure
    move.l d2,(a0)+ ; x position to draw flipper
    move.l d3,(a0)+ ; y position to draw flipper
    move.l d1,(a0)+ ; z position to draw flipper
    move.l d4,(a0)+ ; x position of flipper's origin
    move.l d5,(a0)+ ; y position of flipper's origin
    move.l d0,(a0)  ; angle

    move.l #0,gpu_mode
    lea equine,a0
    jsr gpurun      ;do clear screen
    jsr gpuwait
    rts
```

wells

Wells are fun and should be easy to draw. To start with, we can describe a 2d shape of our choice using a bunch of x/y co-ordinates as points.

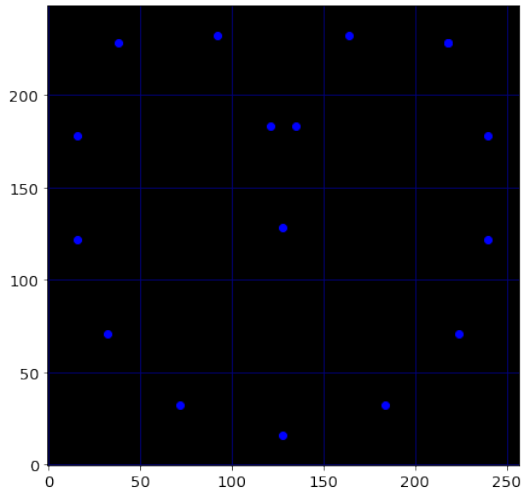


Figure 10.1: Make our dots.

Next we can join these together to give us a 2d shape.

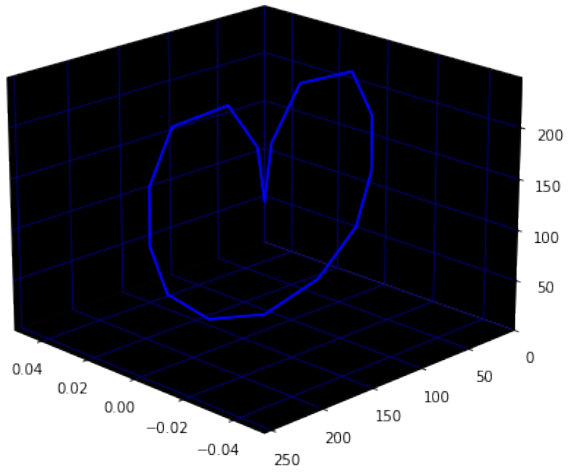


Figure 10.2: Start with a 2d shape..

To make this thing three dimensional all we have to do is project our shape onto some chosen distant position along the Z plane.

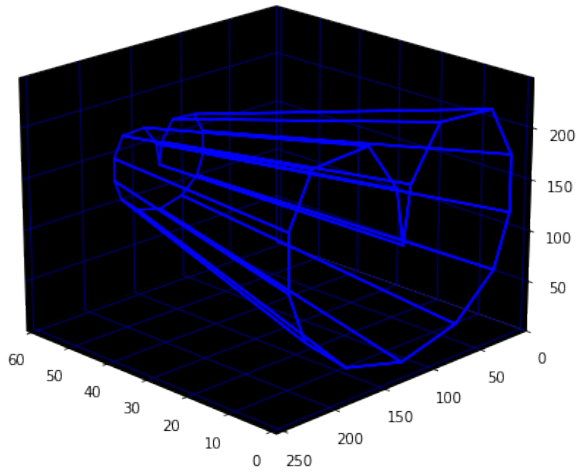


Figure 10.3: ..then make it three dimensional.

Seems simple enough. With this as our objective the data structure below gives us everything we need to create the web. In addition to listing the 16 vertices that make up the web's shape in two dimensions.

```
; Data structure for the 'HEART' well.
; X Co-Ordinates
    .BYTE 0DA,0A4, 87, 80, 79, 5C, 26, 10    ;HEART
    .BYTE  10, 20, 48, 80,0B8,0E0,0F0,0F0
; Y Co-Ordinates
    .BYTE 0E4,0E8,0B7, 80,0B7,0E8,0E4,0B2     ;HEART
    .BYTE  7A, 47, 20, 10, 20, 47, 7A,0B2
```

To help you relate this table to the co-ordinates in our graphs let's list it in decimal instead:

```
; Data structure for the 'HEART' well.
; X Co-Ordinates
    .BYTE 218,164,135,128,121, 92, 38, 16    ; HEART
    .BYTE  16, 32, 72,128,184,224,240,240
; Y Co-Ordinates
    .BYTE 228,232,183,128,183,232,228,178    ; HEART
    .BYTE 122, 71, 32, 16, 32, 71,122,178
```

```
        .SBTTL  UTILITY-DRAW WELL SHAPE
DSPHOL:
        JSR LVLWEL              ;SET UP WELL INDEX & ID
        STA SAVEY               ;WELL INDEX
        STX SAVEX               ;CYCLE
        LDA I,0
        STA VGBRIT
        LDA I,5                 ;MAKE WELL REALLY SMALL
        JSR VGSCA1
        LDA SAVEX               ;GET CYCLE (TIMES THRU ALL WELLS
        AND I,7
        TAX
        LDY X,SPWECO            ;GET SPECIAL WELL COLOR FOR CYCLE
        STY COLOR
        LDA I,MZCOLO
        JSR VGSTAT              ;SET WELL COLOR
        LDX WELLID
        LDA SAVEY
        LDY X,HOLRAP
        IFEQ                    ;PLANAR?
        SEC                     ;NO. START BEAM AT FIRST POINT
        SBC I,0F                ;IN TABLE (FOR CLOSED WELLS)
        ENDIF
        TAY
        LDA Y,NEWLIZ
        STA PYL
        EOR I,80                ;ADJUST Z SIGN
        TAX
```

```
        LDA Y,NEWLIX            ;SAVE COORDS OF 1ST PT
        STA PXL
        EOR I,80                ;ADJUST X SIGN
        JSR VGVTR1              ;POSITION BEAM AT 1ST PT ON WELL
        LDA I,0C0               ;TURN BEAM ON
        STA VGBRIT
        LDX I,NLINES-1
        STX INDEX2
        BEGIN                   ;LOOP FOR EACH PT ON EDGE
        LDY SAVEY
        LDA Y,NEWLIX            ;
        TAX
        SEC
        SBC PXL                 ;DELTA X
        PHA                     ;
        STX PXL                 ;CURRENT X OLD X
        LDA Y,NEWLIZ
        TAY
        SEC
        SBC PYL                 ;DELTA Z
        TAX
        STY PYL                 ;CURRENT Z>OLD Z
        PLA
        JSR VGVTR1              ;DRAW VECTOR TO NEXT PT.
        DEC SAVEY
        DEC INDEX2
        MIEND
        LDA I,1                 ;NORMAL SIZE AGAIN
        JMP VGSCA1
```

webs

Webs are fun and should be easy to draw. To start with, we can describe a 2d shape of our choice using a bunch of x/y co-ordinates as points.

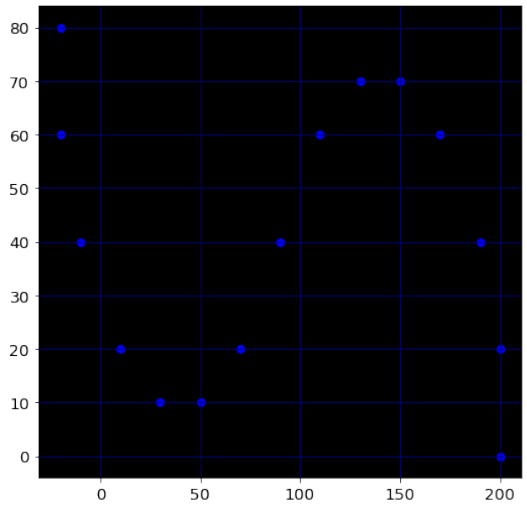


Figure 11.1: Make our dots.

Next we can join these together to give us a 2d shape.

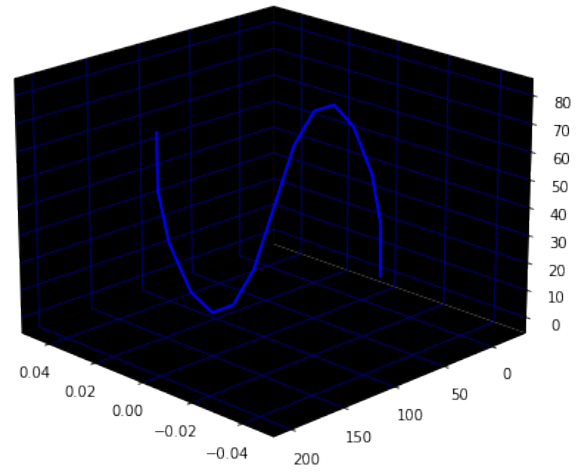


Figure 11.2: Start with a 2d shape..

To make this thing three dimensional all we have to do is project our shape onto some chosen distant position along the Z plane.

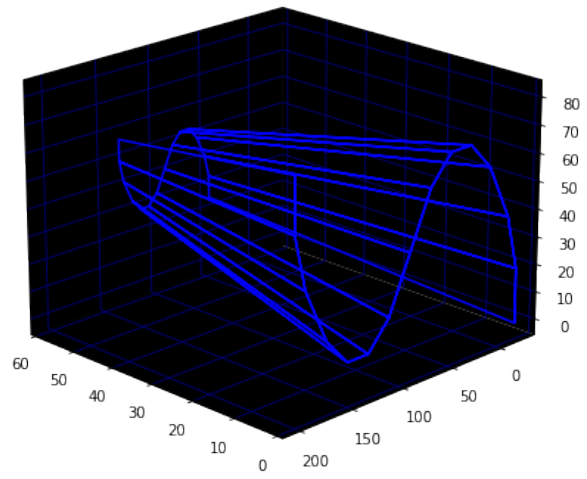


Figure 11.3: ..then make it three dimensional.

Seems simple enough. With this as our objective the data structure below gives us everything we need to create the web. In addition to listing the 16 vertices that make up the web's shape in two dimensions, we also include some information on whether the web should be open (like the one pictured above) or closed (like a circle or square). Since a web of 16 vertices will give us 14 lanes which the player and enemies can occupy we also add a 'Rotation Table' to tell us how players and enemies should be oriented when on that lane.

```
; Data structure for the 'Sine Wave' web.
web13:
  dc.w 14     ; Number of lanes in the web.
  dc.w  6     ; The lane the player starts on.

  ; The x/y pairs of all vertices in the web. So for example
  ; -2/14 indicates the X and Y co-ordinates of the first vertex.
  ; There are always 16 in total.
  dc.w -2,14,-2,12,-1,10, 1, 8
  dc.w  3, 7, 5, 7, 7, 8, 9,10
  dc.w 11,12,13,13,15,13,17,12
  dc.w 19,10,20, 8,20, 6,-2,14

  dc.w 0      ; 0 = Open Web, -1 = Closed Web

  ; Rotation Table (angle of an object within a particular lane)
  ; The length of this list is specified by the '14' above.
  dc.w -64,-48,-32,-16, 0,16,32,32,16, 0,-16,-32,-48,-64
```

The values given in the Rotation Table are angles of roll to apply to the claw (and to the claw's enemies) when placing them in the lane. Negative values rotate the object to the right, while positive values rotate it to the left.

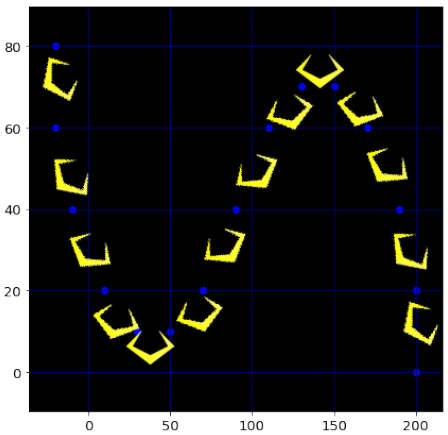


Figure 11.4: The rotation table applied to the claw in each lane of the web.

This is the routine that converts the web data structure into a list of vertices and lines between vertices that will make up the web itself:

```
extrude:
;
; extrude a web from a list of 16 pairs of XY coordinates addressed by (a1)
;
; a0 = vector ram space; a2.l = z depth to extrude to; d0-d7 as above

  move.l vadd,a0
  movem.l d0-d7/a0/a2,-(a7)    ;save so routine can return address
  clr connect
  move.l a2,-(a7)    ;save z depth
  bsr initvo    ;make header, do standard vector object init
  move.l a3,a4    ;save first vertex
  move.l (a7)+,d7    ;retrieve z-depth
  move d7,d0
  asr #1,d0
  move d0,web_z    ;Current Web z centering
  clr.l d0
  clr.l d1
  clr d5      ;to catch highest X point
  move (a1)+,d6            ; No of lines in the web.
  move d6,web_max          ; Keep it in web_max
  move (a1)+,web_firstseg  ; first position on web
  move.l a1,web_ptab       ;position table
  move.l a3,(a5)+          ;first vertex to lanes list
```

```
  ; Read in the x/y pairs
xweb:
  ; Get the current x and y pair
  move (a1)+,d0
  move (a1)+,d1    ;get X and Y
  ext.l d0
  ext.l d1

  ; Check if this is the large X value so far
  cmp d5,d1       ; Compare x with the largest so far, stored in d5.
  blt xweb2       ; If it's less, skip to xweb2 below.
  move d1,d5      ; It's bigger, so save it in d5.

xweb2:
  ; Store the x,y,z value for the near point in the web
  move.l d0,(a2)+ ; x value
  move.l d1,(a2)+ ; y value
  clr.l (a2)+     ; z value for near point (always 0)

  ; Store the x,y,z value for the far point in the web
  move.l d0,(a2)+ ; x value
  move.l d1,(a2)+ ; y value
  move.l d7,(a2)+ ; z value for far point (calculated by initvo).

  move d3,(a3)+    ;vertex ID to conn list
```

```
tst d6
beq lastpoint    ;special case for last point!

; Connect the vertices
move d3,d4       ;copy vertex #
addq #1,d4
move d4,(a3)+    ;connect to n+1
addq #1,d4
move d4,(a3)+    ;connect to n+2
move #0,(a3)+    ;end vertex
subq #1,d4       ;point to n+1
move d4,(a3)+
addq #2,d4       ;n+3
move d4,(a3)+    ;connect
move #0,(a3)+    ;delimit
move.l a3,(a5)+  ;to v.conn list
add #2,d3        ;move 2 vertices

; Get the next pair
dbra d6,xweb
```

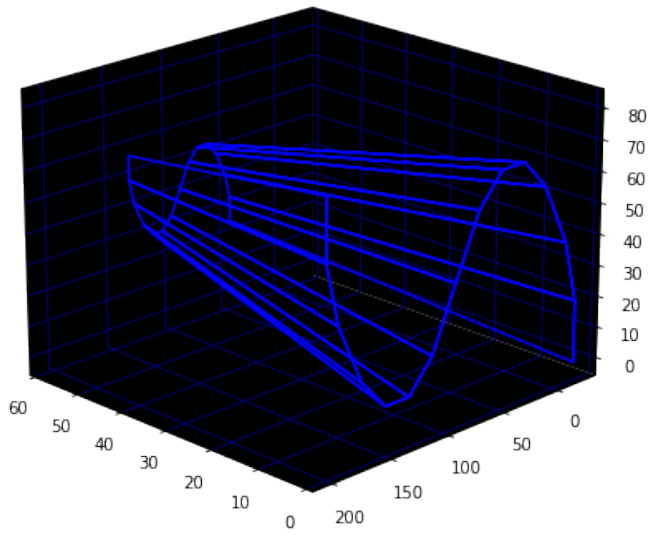


Figure 11.5: Adding in our second triangle

object list

About 60 times a second the Jaguar will want to write something to the screen. A module in the Jaguar system called the 'Object Processor' will take whatever the current list of things to draw happens to be and draw them. It's Tempest 2000's job to keep this beast well fed with new stuff. The way to do this is to keep shovelling fresh imagery into a structure called an 'Object List' for the Object Processor to chew on. As the name suggests this is a list of objects, but since everything is an object these days we might need to be more specific about what these objects are and what they contain.

There are a few different types of objects, but the only one of real interest is the one that contains something that can be displayed: an image. This image takes the form of a section of data similar to the cry pixel data we looked at in 'cry if i want to'. There can be a number of different flavours of this data that make the Object Processor's job easier in different circumstances but in Tempest 2000 we only ever detain ourselves with the rich cry data, the good stuff.

Below we set out the raw contents of an actual Object List used for a single frame in the 'Demo' sequence. The interesting objects in the list are the 'Bit Mapped Objects'. You can tell they're interesting because we have to specify so much about them that we need two 'phrases' (i.e. two sets of 8 bytes each) to encapsulate it all.

Data	Object Type
13,48,00,1D,6E,45,C1,60	Bit Mapped Object (First Phrase)
00,00,80,06,01,80,CF,F8	Bit Mapped Object (Second Phrase)
00,00,00,00,00,00,00,04	Stop Object
00,00,00,00,00,00,00,04	Stop Object
05,00,00,1D,72,0C,01,E0	Bit Mapped Object (First Phrase)
00,00,80,06,01,80,CF,F8	Bit Mapped Object (Second Phrase)

Figure 12.1: Object List

It is kind of amazing how much you can pack into 16 bytes if you try. Below we parse out the contents of the first 'Bit Mapped Object' in our Object List and show the screen of pixel data it references in the first 8-byte phrase.

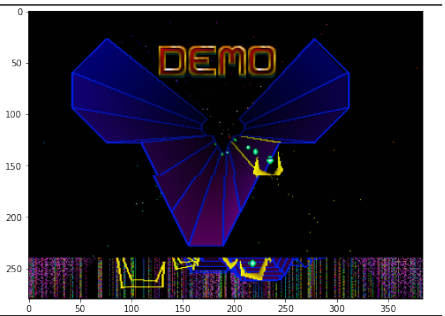
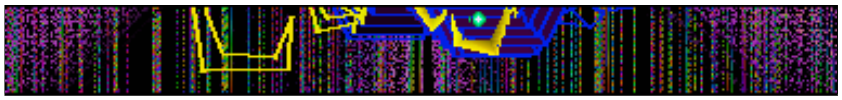
Raw Data	Parsed Data	Image at `DATA: 0x134800`
13,48,00,1D,6E,45,C1,60	TYPE: 0 LINK: 0xeb70 YPOS: 44 DATA: 0x134800 HEIGHT: 279	
00,00,80,06,01,80,CF,F8	XPOS: -7 REFLECT: 0 DEPTH: 4 RMW: 0 PITCH: 1 TRANS: 1 DWIDTH: 96 RELEASE: 0 IWIDTH: 96 FIRSTPIX: 0 INDEX: 0	

Figure 12.2: First Bit Mapped Object in the Object List

Here is what a selection of the parameters we supply above are a way of saying:

`DATA: 0x134800` ⟶ Paint the data found at address `0x134800` in RAM.
`DEPTH: 4` ⟶ Treat the data as using 16 bits for each pixel.
`YPOS: 44` ⟶ Start painting this image at line 44.
`XPOS: -7` ⟶ Start painting this image at X pos -7.
`DWIDTH: 96` ⟶ Treat the data as having a line width of 380 (96*4).
`IWIDTH: 96` ⟶ Treat the image as having a width of 380 (96*4).
`HEIGHT: 279` ⟶ Treat the data as having a height of 280 lines.

You'll notice we have what looks like a lovely bit of glitch at the bottom of our image data.



This handsome scramble is the left-over data from previously drawn frames in the RAM. It is not visible because as we noted above we are painting this image at Y position 49, so although we are painting the full 280 lines of data only the first 231 are visible, and our glorious glitch sits off screen.

These vacant 49 lines at the top of the screen are occupied by our second 'Bit Mapped Object'. This is the player's score and their remaining lives:

Raw Data	Parsed Data	Image in Data `0x134800`

`05,00,00,1D,72,0C,01,E0`	`TYPE: 0 LINK: 0xeb90`
	`YPOS: 60 DATA: 0x50000`
	`HEIGHT: 48`	

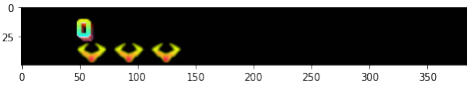
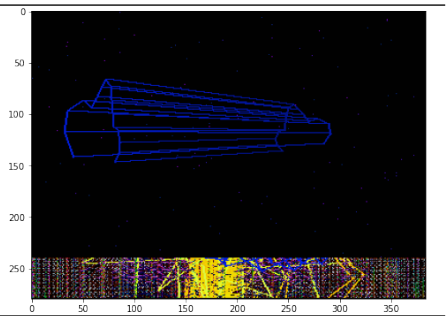
`00,00,80,06,01,80,CF,F8`	`XPOS: -7 REFLECT: 0`
	`DEPTH: 4 RMW: 0`
	`PITCH: 1 TRANS: 1`
	`DWIDTH: 96 RELEASE: 0`
	`IWIDTH: 96 FIRSTPIX: 0`
	`INDEX: 0`
		

Figure 12.3: Second Bit Mapped Object in the Object List

We can be flexible about how many Bit Mapped Objects we have and even what we put in them. Here are the two objects we use for the high score screen. The first is the rotating web in the background:

Raw Data	Parsed Data	Image at `DATA: 0x100000`

`10,00,00,1D,6E,45,C1,60`	`TYPE: 0 LINK: 0xeb70`
	`YPOS: 44 DATA: 0x100000`
	`HEIGHT: 279`
		

`00,00,80,06,01,80,CF,F8`	`XPOS: -7 REFLECT: 0`
	`DEPTH: 4 RMW: 0`
	`PITCH: 1 TRANS: 1`
	`DWIDTH: 96 RELEASE: 0`
	`IWIDTH: 96 FIRSTPIX: 0`
	`INDEX: 0`	

The second is the high-score screen itself:

Raw Data	Parsed Data	Image at DATA: 0x100000
05,00,00,1D,72,45,C1,60	TYPE: 0 LINK: 0xeb90 YPOS: 44 DATA: 0x5000 HEIGHT: 279	
00,00,80,06,01,80,CF,F8	XPOS: -7 REFLECT: 0 DEPTH: 4 RMW: 0 PITCH: 1 TRANS: 1 DWIDTH: 96 RELEASE: 0 IWIDTH: 96 FIRSTPIX: 0 INDEX: 0	

Notice that unlike our previous case we are superimposing these two images (or compositing them) so that the high score appears above the rotating web.

cursors

The Tempest source code refers to the player's ship as a `CURSOR`. It does this because it is the 1980s and that is what you call something on the screen that you can control.

The graphics in Tempest are generated using Atari's 'Quadrascan' vector technology. The images are all defined using a series of vectors. A vector in this case is a (X,Y) value pair that moves the beam in an X,Y direction to a new point on the screen. A series of vectors moving from point to point around the screen will eventually form a complete image.

When we take a look at the data structure defining a relatively upright version of the player's ship we find this. I'm going to speculate that `NCSR4S` is short for 'New Cursor 4 Start' and `NCSR4E` for 'New Cursor 4 End':

```
NCRS4S:
        VEC 3,-2
        VEC 5,2
        VEC -3,1
        VEC 2,-1
        VEC -4,-1
        VEC -2,1
        VEC 2,1
        VEC -3,-1
NCRS4E:
```

The number pairs in the above listing are the vectors we're talking about, and we're going to see how we can use them to build up an image of the ship assuming an origin point of zero. Our first step, therefore, is to draw a line from (0,0) to (3,-2) as follows:

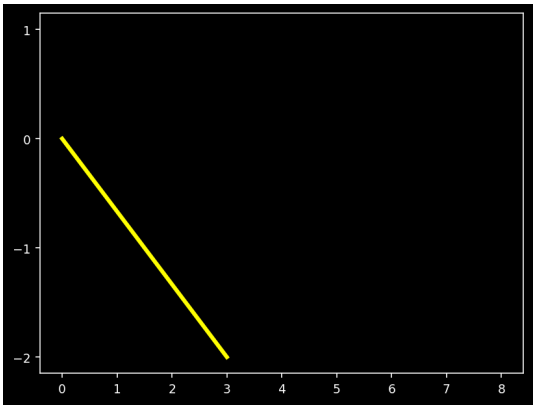


Figure 13.1: Draw a line from `(0,0)` to `(3,-2)`.

Our next step reveals a bit more about the actual nature of the operation we are performing. To draw a line from our new position using a vector of `5,2` we add it to our current position of `(3,-2)` to draw a line to `8,0`:

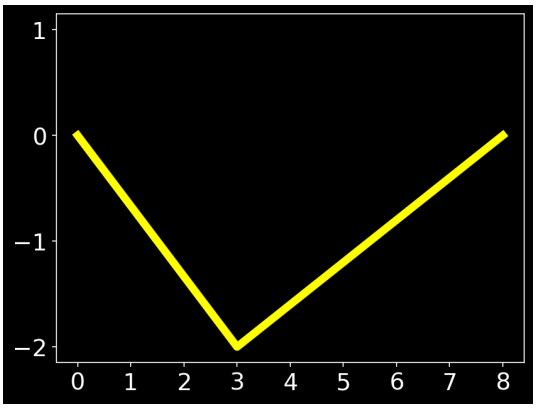
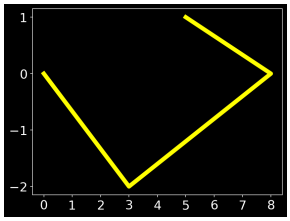
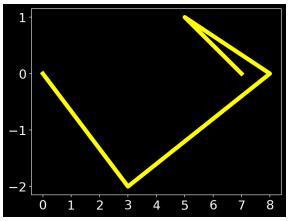


Figure 13.2: `(3,-2)` + `(5,2)`. ⟶ `(8,0)`

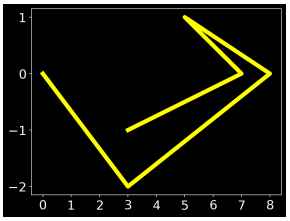
With this as our method, we can start building up our complete image, adding each vector in our array to the previous result to define a new line to draw.



(8,0) + (-3,1). ⟶ (5,1)



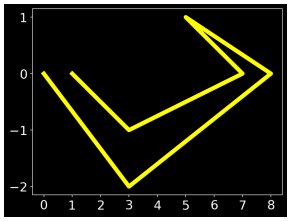
(5,1) + (2,-1). ⟶ (7,0)



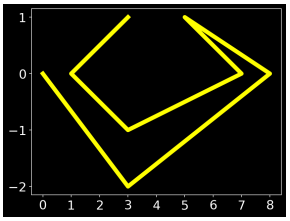
(7,0) + (-4,-1). ⟶ (3,-1)

Figure 13.3: Starting to take shape.

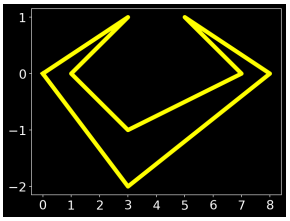
Adding our last three vectors completes the picture, and we have a claw.



(3,-1) + (-2,1). ⟶ (1,0)



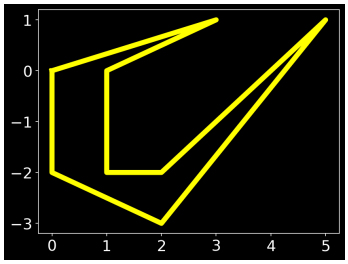
(1,0) + (2,1). ⟶ (3,1)



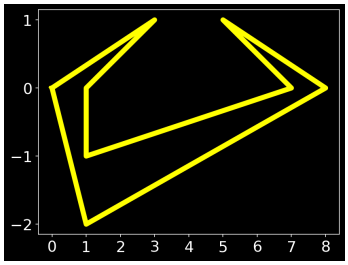
(3,1) + (-3,-1). ⟶ (0,0)

Figure 13.4: Putting the final pieces in place.

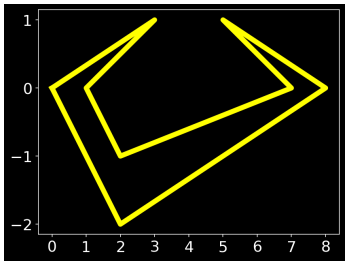
Finally, here are all of the `CURSOR` elements along with their data structures.



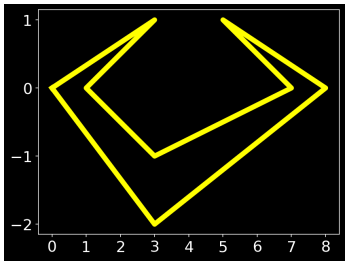

```
NCRS1S:
 VEC 0,-2
 VEC 2,-1
 VEC 3,4
 VEC -3,-3
 VEC -1,0
 VEC 0,2
 VEC 2,1
 VEC -3,-1
NCRS1E:
```



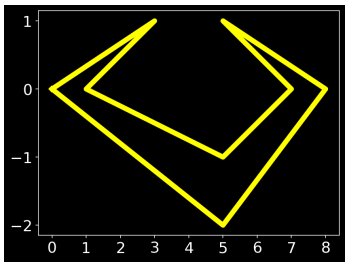

```
NCRS2S:
 VEC 1,-2
 VEC 7,2
 VEC -3,1
 VEC 2,-1
 VEC -6,-1
 VEC 0,1
 VEC 2,1
 VEC -3,-1
NCRS2E:
```



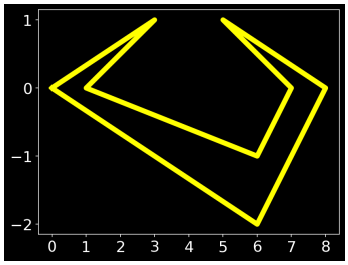

```
NCRS3S:
 VEC 2,-2
 VEC 6,2
 VEC -3,1
 VEC 2,-1
 VEC -5,-1
 VEC -1,1
 VEC 2,1
 VEC -3,-1
NCRS3E:
```



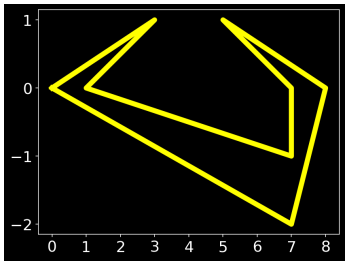

```
NCRS4S:
 VEC 3,-2
 VEC 5,2
 VEC -3,1
 VEC 2,-1
 VEC -4,-1
 VEC -2,1
 VEC 2,1
 VEC -3,-1
NCRS4E:
```



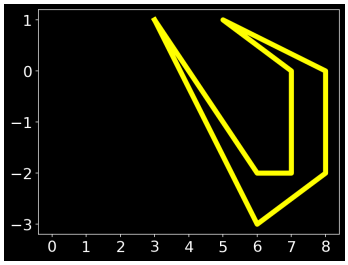

```
NCRS5S:
 VEC 5,-2
 VEC 3,2
 VEC -3,1
 VEC 2,-1
 VEC -2,-1
 VEC -4,1
 VEC 2,1
 VEC -3,-1
NCRS5E:
```




```
NCRS6S:
 VEC 6,-2
 VEC 2,2
 VEC -3,1
 VEC 2,-1
 VEC -1,-1
 VEC -5,1
 VEC 2,1
 VEC -3,-1
NCRS6E:
```




```
NCRS7S:
 VEC 7,-2
 VEC 1,2
 VEC -3,1
 VEC 2,-1
 VEC 0,-1
 VEC -6,1
 VEC 2,1
 VEC -3,-1
NCRS7E:
```




```
NCRS8S:
 VEC 3,1,0
 VEC 3,-4
 VEC 2,1
 VEC 0,2
 VEC -3,1
 VEC 2,-1
 VEC 0,-2
 VEC -1,0
 VEC -3,3
NCRS8E:
```

claws

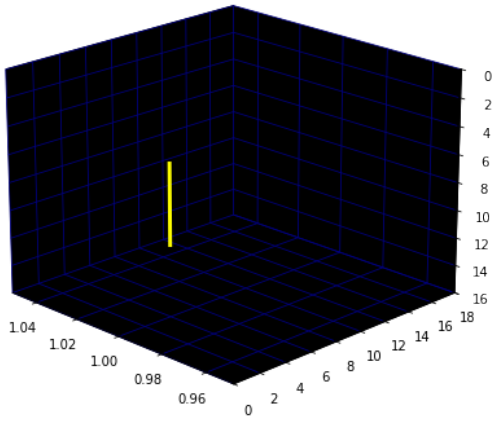
In Tempest 2000, for no good reason, the 'classic' tempest claws use trios of bytes to give co-ordinates in 3D space. I say 'for no good reason' because the claw is still flat, so the Z co-ordinate is always a `1`. We define our claws using a list of `X,Y,Z` co-ordinates, and each co-ordinate is paired with a list of points that it connects to. It is this list of connecting points that allows us to draw lines:

```
claw3:  dc.b 4,5,1              ; Co-ordinate 1
        dc.b 2,8,0              ; Draw line to co-ords 2 and 8.
        dc.b 4,11,1             ; Co-ordinate 2
        dc.b 3,0                ; Draw line to co-ord 3.
        dc.b 11,12,1            ; Co-ordinate 3
        dc.b 4,0                ; Draw line to co-ord 4.
        dc.b 15,8,1             ; Co-ordinate 4
        dc.b 5,0                ; Draw line to co-ord 5.
        dc.b 12,4,1             ; Co-ordinate 5
        dc.b 6,0                ; Draw line to co-ord 6.
        dc.b 17,8,1             ; Co-ordinate 6
        dc.b 7,0                ; Draw line to co-ord 7.
        dc.b 11,15,1            ; Co-ordinate 7
        dc.b 8,0                ; Draw line to co-ord 8.
        dc.b 2,12,1             ; Co-ordinate 8
        dc.b 0,0                ; Don't draw a line.
```

This is what it looks like when we consume the first co-ordinate and its paired points:

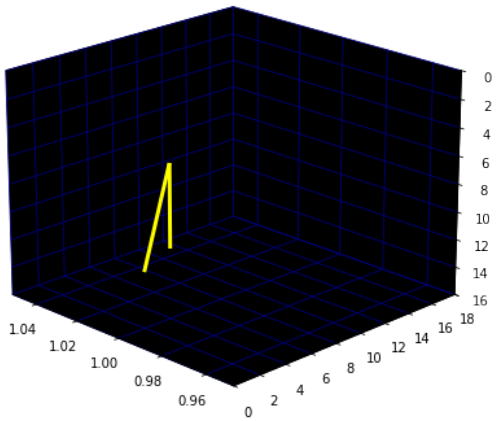
```
claw3:  dc.b 4,5,1              ; Co-ordinate 1
        dc.b 2,8,0              ; Draw line to co-ords 2 and 8.
```

This is telling us to connect our first trio `4,5,1` to co-ordinates '2' (`4,11,1`)..



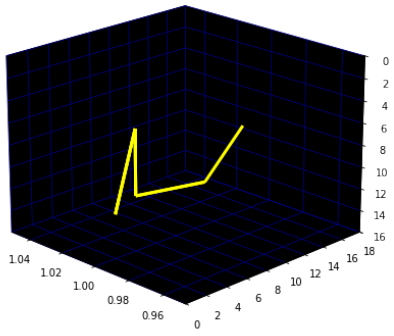
Draw a line from `(4,5,1)` to `(4,11,1)`.

.. and '8' `(2,12,1)`.

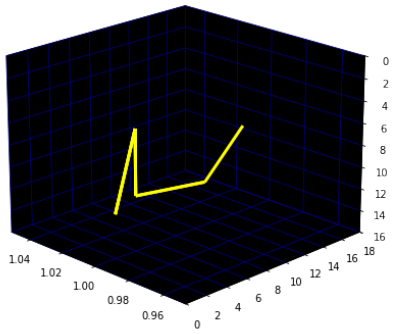


Draw a line from `(4,5,1)` to `(2,12,1)`.

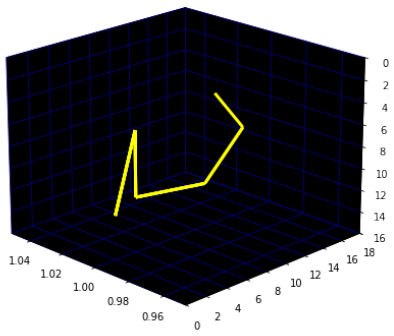
With this as our method, we can start building up our complete image, adding each point in our array to the previous result to define a new line to draw.



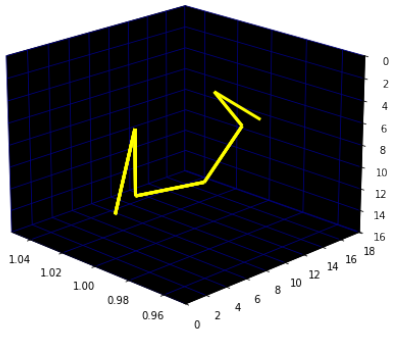

```
dc.b 4,11,1   ; Co-ordinate 2
dc.b 3,0      ; Draw line to co-ord 3.
dc.b 11,12,1  ; Co-ordinate 3
```



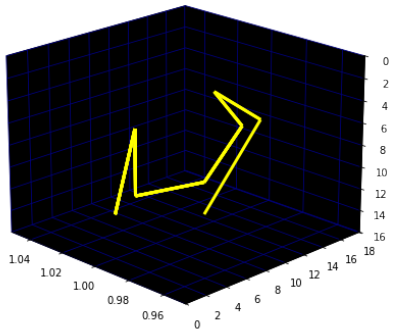

```
dc.b 11,12,1 ; Co-ordinate 3
dc.b 4,0     ; Draw line to co-ord 4.
dc.b 15,8,1  ; Co-ordinate 4
```



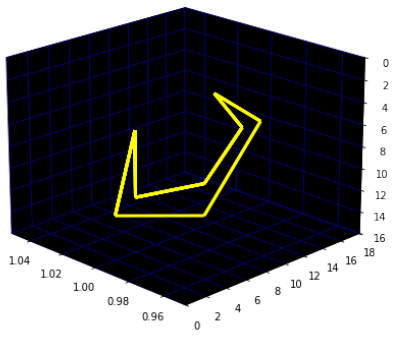

```
dc.b 15,8,1  ; Co-ordinate 4
dc.b 5,0     ; Draw line to co-ord 5.
dc.b 12,4,1  ; Co-ordinate 5
```




```
dc.b 12,4,1   ; Co-ordinate 5
dc.b 6,0      ; Draw line to co-ord 6.
dc.b 17,8,1   ; Co-ordinate 6
```

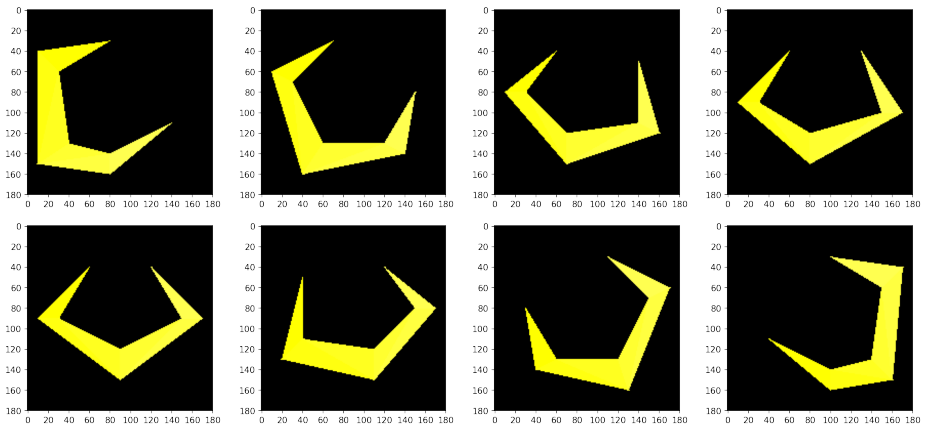



```
dc.b 17,8,1    ; Co-ordinate 6
dc.b 7,0       ; Draw line to co-ord 7.
dc.b 11,15,1   ; Co-ordinate 7
```




```
dc.b 11,15,1 ; Co-ordinate 7
dc.b 8,0     ; Draw line to co-ord 8.
dc.b 2,12,1  ; Co-ordinate 8
```

more claws



The claws for Tempest 2000.

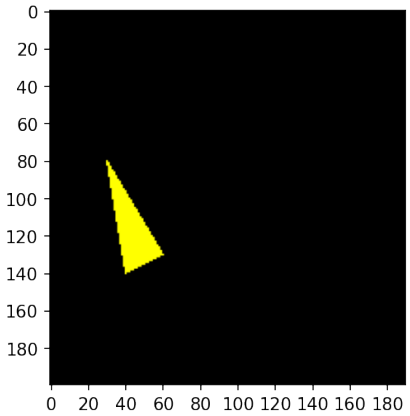
The Tempest 2000 claws are built using the same form of data structure we saw for 'flippers'. So although we don't need to rehearse the details in full again here. It is still worth seeing how a claw is built up from its component polygons.

The data structure for an upright claw in Tempest 2000 mode is as follows:

```
sclaw6:
  dc.l 6                ; 4 faces in this object, a shaded solid Flipper

 dc.w $ff              ; Face colour - Yellow.
  dc.w 0,$ff00          ; vertex index, intensity.
  dc.w 1,$ff00          ; vertex index, intensity.
  dc.w 2,$c000          ; vertex index, intensity.
  dc.w 0

  dc.w $fe              ; Face colour - Yellow.
  dc.w 1,$ff00          ; vertex index, intensity.
  dc.w 2,$c000          ; vertex index, intensity.
  dc.w 4,$ff00          ; vertex index, intensity.
  dc.w 0

  dc.w $fd              ; Face colour - Yellow.
  dc.w 2,$c000          ; vertex index, intensity.
  dc.w 3,$8000          ; vertex index, intensity.
  dc.w 4,$ff00          ; vertex index, intensity.
  dc.w 0

 dc.w $fd              ; Face colour - Yellow.
  dc.w 3,$8000          ; vertex index, intensity.
  dc.w 5,$c000          ; vertex index, intensity.
  dc.w 4,$ff00          ; vertex index, intensity.
  dc.w 0

  dc.w $fe              ; Face colour - Yellow.
  dc.w 4,$ff00          ; vertex index, intensity.
  dc.w 5,$c000          ; vertex index, intensity.
  dc.w 6,$ff00          ; vertex index, intensity.
  dc.w 0

  dc.w $ff              ; Face colour - Yellow.
  dc.w 5,$c000          ; vertex index, intensity.
  dc.w 6,$ff00          ; vertex index, intensity.
  dc.w 7,$ff00          ; vertex index, intensity.
  dc.w 0

verts:
 dc.w 3,8              ; Index 0
  dc.w 4,14             ; Index 1
  dc.w 6,13             ; Index 2
  dc.w 12,13            ; Index 3
  dc.w 13,16            ; Index 4
  dc.w 15,7             ; Index 5
  dc.w 17,6             ; Index 6
  dc.w 11,3             ; Index 7
```

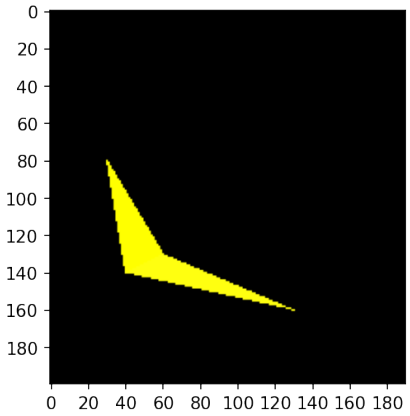
As before we are constructing our claw with a series of triangles. Each claw is made up 6 triangles, and for each triangle we specify 3 vertices.

With these, and our specified colour of yellow (`$ff`), we can make a triangle:



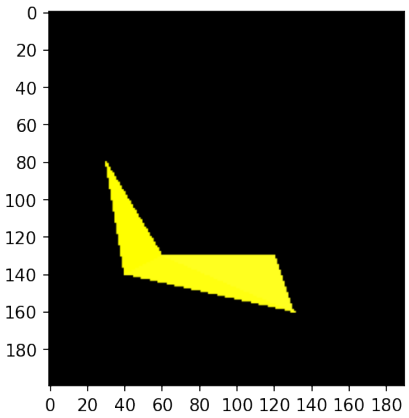

```
dc.w $ff      ; Face colour - Yellow.
dc.w 0,$ff00 ; vertex index, intensity.
dc.w 1,$ff00 ; vertex index, intensity.
dc.w 2,$c000 ; vertex index, intensity.
dc.w 0
```

```
dc.w 3,8     ; Index 0
dc.w 4,14    ; Index 1
dc.w 6,13    ; Index 2
```



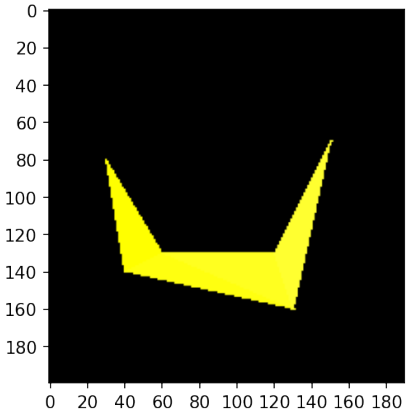

```
dc.w $fe       ; Face colour - Yellow.
dc.w 1,$ff00   ; vertex index, intensity.
dc.w 2,$c000   ; vertex index, intensity.
dc.w 4,$ff00   ; vertex index, intensity.
dc.w 0
```

```
dc.w 4,14    ; Index 1
dc.w 6,13    ; Index 2
dc.w 13,16   ; Index 4
```



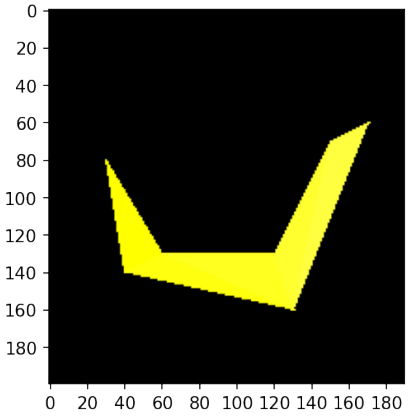

```
dc.w $fd       ; Face colour - Yellow.
dc.w 2,$c000   ; vertex index, intensity.
dc.w 3,$8000   ; vertex index, intensity.
dc.w 4,$ff00   ; vertex index, intensity.
dc.w 0
```

```
dc.w 6,13   ; Index 2
dc.w 12,13  ; Index 3
dc.w 13,16  ; Index 4
```



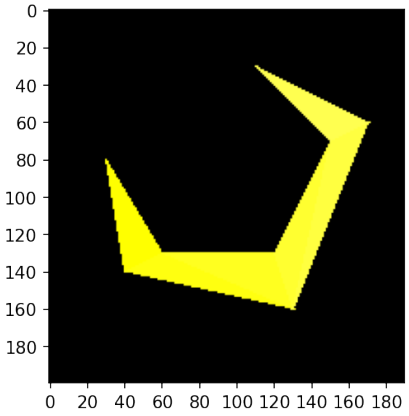

```
dc.w $fd       ; Face colour - Yellow.
dc.w 3,$8000   ; vertex index, intensity.
dc.w 5,$c000   ; vertex index, intensity.
dc.w 4,$ff00   ; vertex index, intensity.
dc.w 0
```

```
dc.w 12,13   ; Index 3
dc.w 15,7    ; Index 5
dc.w 13,16   ; Index 4
```




```
dc.w $fe       ; Face colour - Yellow.
dc.w 4,$ff00   ; vertex index, intensity.
dc.w 5,$c000   ; vertex index, intensity.
dc.w 6,$ff00   ; vertex index, intensity.
dc.w 0
```

```
dc.w 13,16 ; Index 4
dc.w 15,7  ; Index 5
dc.w 17,6  ; Index 6
```




```
dc.w $ff       ; Face colour - Yellow.
dc.w 5,$c000   ; vertex index, intensity.
dc.w 6,$ff00   ; vertex index, intensity.
dc.w 7,$ff00   ; vertex index, intensity.
dc.w 0
```

```
dc.w 15,7 ; Index 5
dc.w 17,6 ; Index 6
dc.w 11,3 ; Index 7
```

meltovision

```
fade:
;
; go into FADE after merging screen3 to current screen and turning off
    BEASTIES+64

        tst beasties+76
        bmi ofade
        move.l #screen3,a0
        move.l gpu_screen,a1
        moveq #0,d0
        moveq #0,d1
        move #384,d2
        move #240,d3
        add palfix1,d3
        moveq #0,d4
        moveq #0,d5
        tst mfudj
        beq pmf2
        sub #8,d3
pmfade: tst mfudj
        beq pmf2
        add #8,d5
        clr mfudj
pmf2:   jsr MergeBlock
        move #-1,beasties+76


ofade:
;       move pauen,-(a7)
        clr _pauen                      ;can't pause in fade
        move.l #ffade,demo_routine
        move.l #failcount,routine
```

```
        move #150,pongx
        jsr rannum
        and #$7,d0
        sub #$03,d0
        and #$ff,d0
        move d0,pongz
;       clr pongz
        move.l #$f80000,delta_i
        move z,-(a7)
        clr z
        bsr gogame
        move.l #screen3,a0
        jsr clrscreen
        move (a7)+,z
;       move #-1,db_on
;       move #1,screen_ready
        move #1,sync
;       move (a7)+,pauen
        rts
```

[escapechar=%]

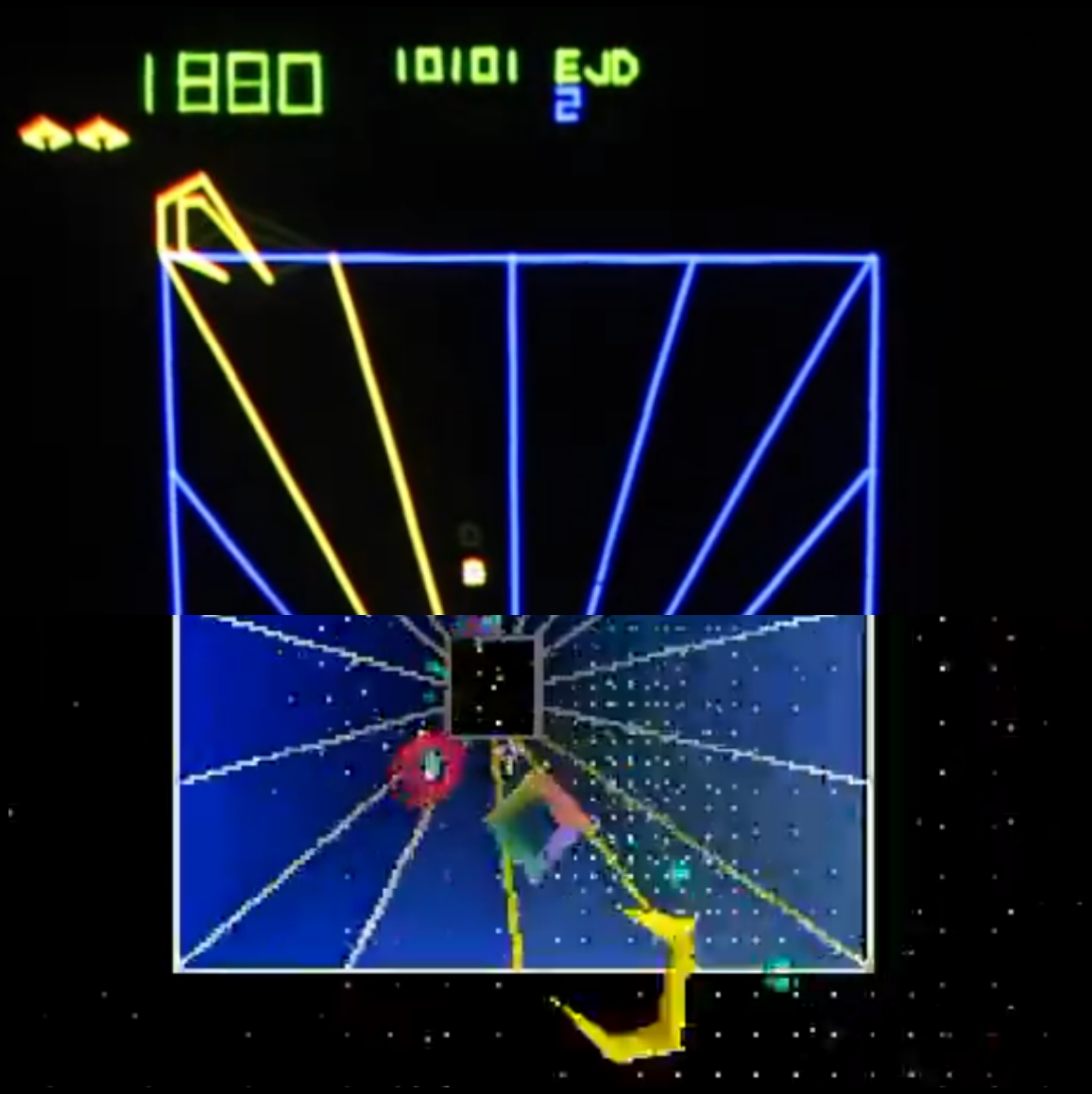
```
ffade:  add #1,pongy
        move pongy,d0
        and #$03,d0
        bne failfade
        tst.b pongz+1
        beq failfade
        bmi ffinc
        sub.b #2,pongz+1
ffinc:  add.b #1,pongz+1
failfade:
        move.l #(PITCH1|PIXEL16|WID384),d0
        move.l d0,source_flags
        move.l d0,dest_flags
        lea in_buf,a0
        move pongz,d0
        and.l #$ff,d0
        move.l cscreen,(a0)             ;source screen is already-displayed
    screen
        move.l #384,4(a0)
        move.l #240,d1
        add palfix1,d1
        move.l d1,8(a0)         ;X and Y dest rectangle size
        move.l #$1f4,12(a0)
        move.l #$1f4,16(a0)             ;X and Y scale as 8:8 fractions
        move.l d0,20(a0)                        ;initial angle in brads
        move.l #$c00000,24(a0)          ;source x centre in 16:16
        move.l #$780000,d0
        add.l palfix3,d0
        move.l d0,28(a0)                ;y centre the same
        move.l #$0,32(a0)               ;offset of dest rectangle
        move.l delta_i,36(a0)           ;change of i per increment
```

```
        move.l #2,gpu_mode              ;op 2 of this module is Scale and
    Rotate
        move.l #demons,a0
        jsr gpurun                      ;do it
        jmp gpuwait
```

[escapechar=%]

Index

claw says, 14, 31, 35, 40


1880
10101 EJD
2