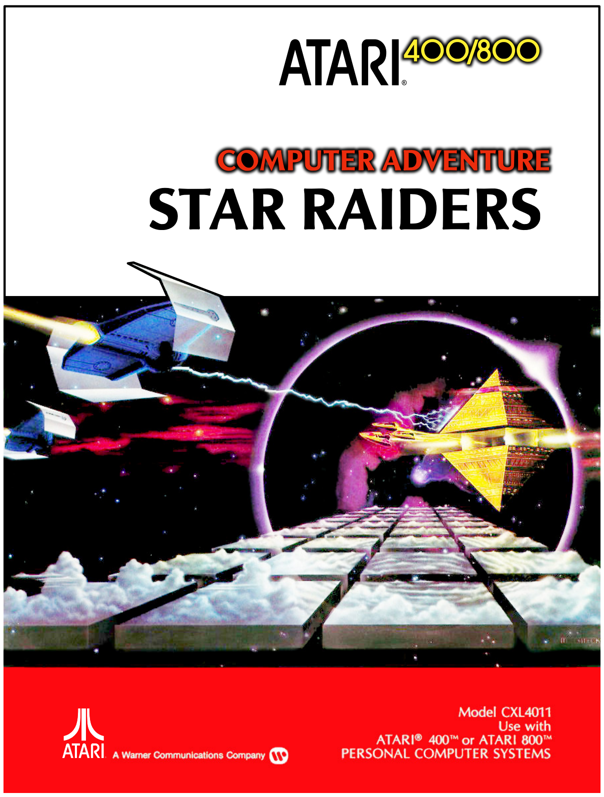
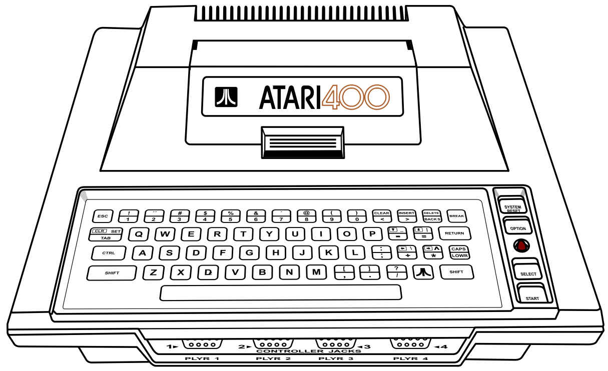
2 Reverse Engineering Star Raiders

by Lorenz Wiest

2.1 Introduction



STAR RAIDERS is a seminal computer game published by Atari Inc. in 1979 as one of the first titles for the original Atari 8-bit Home Computer System (Atari 400 and Atari 800). It was written by Atari engineer Doug Neubauer, who also created the system's POKEY sound chip. **STAR RAIDERS** is considered to be one of the ten most important computer games of all time.[2].



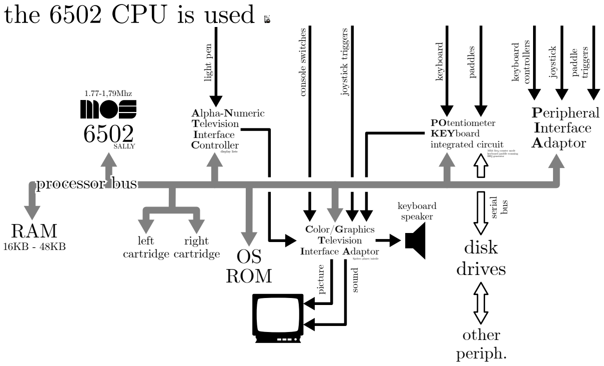
The game is a 3D space combat flight simulation where you fly your starship through space, shooting at attacking Zylon spaceships. The game's universe is made up of a 16 × 8 grid of sectors. Some of them contain enemy Zylon units, some a friendly starbase. The Zylon units converge toward the starbases and try to destroy them. The starbases serve as repair and refueling points for your starship. You move your starship between sectors with your hyperwarp drive. The game is over if you have destroyed all Zylon ships, have ran out of energy, or if the Zylons have destroyed all starbases.

At a time when home computer games were pretty static – think SPACE INVADERS (1978) and PAC MAN (1980) – **STAR RAIDERS** was a huge hit because the game play centered on the very dynamic 3D first-person view out of your starship's cockpit window.

The original Atari 8-bit Home Computer System

[2]"Is That Just Some Game? No, It's a Cultural Artifact." Heather Chaplin, The New York Times, March 12, 2007.

has up to 48 KB RAM and uses a Motorola 6502 CPU. The same CPU is also used in the Apple II, the Commodore C64 (a 6502 variant), and the T-800 Terminator.[3] Several proprietary Atari custom chips provide additional capabilities to the system. **STAR RAIDERS** shows off many of them: 5 Players (sprites), mixed text and pixel graphics modes, dynamic Display Lists, a custom character set, 4-channel sound, Vertical Blank Interrupt and Display List Interrupt code – even the BCD mode of the 6502 CPU is used.



I have been always wondering what made **STAR RAIDERS** tick. I was especially curious how that 3D first-person view star field worked, in particular the rotations of the stars when you fly a turn. So I decided to reverse engineer the game, aiming at a complete, fully documented assembly language source code of **STAR RAIDERS**.

In the following sections I'll show you how I approached the reverse engineering effort, introduce my favorite piece of code in **STAR RAIDERS**, talk about how the tight memory limits influenced the implementation, reveal some bugs, point at some mysterious code, and explain how I got a grip on documenting **STAR RAIDERS**. From time to time, to provide some context to you, I will reference memory locations of the game, which you can look up in the reverse engineered, complete, and fully documented assembly language source code of **STAR RAIDERS** available on GitHub.[4]

2.2 Getting Started

STAR RAIDERS is distributed as an 8 KB ROM cartridge, occupying memory locations `$A000` to `$BFFF`.

The obvious first step was to prod a ROM dump with a disassembler and to apply Atari's published hardware and OS symbols to the disassembly. To my surprise this soon revealed that code and data were clearly separated into three parts:

`$A000`	–	`$A149`	Data (Part 1 of 2)
`$A14A`	–	`$B8DE`	Code (6502 instructions)
`$B8DF`	–	`$BFFF`	Data (Part 2 of 2)

This clear separation helped me instantly to get an overview of the code part, as I could create a disassembly of the code in one go and not having to sift slowly through the bytes of the ROM, deciding which ones are instructions and which ones are data.

Closer inspection of the code part revealed that it was composed of neatly separated subroutines. Each subroutine handles a specific task. The largest subroutine is the main game loop `GAMELOOP` (`$A1F3`), shown in Figure 1. What I expected to be spaghetti code – given the development tools of 1979 and the substantial amount of game features crammed into the 8K ROM – turned out to be surprisingly structured code. Table 1 lists all subroutines of **STAR RAIDERS**, as their function emerged during the reverse engineering effort, giving a good overview how the **STAR RAIDERS** code is organized.

Figure 2 shows the "genome sequence" of the **STAR RAIDERS** 8 KB ROM: The 8192 bytes of the game are stacked vertically, with each byte represented by a tiny, solid horizontal line of 8 pixels. This stack is split into strips of 192 bytes, arranged side-by-side. Alternating light and dark blue areas represent bytes of distinct subroutines. Alternating light and dark green and purple areas represent bytes of distinct sections of data (lookup tables, graphical shapes, etc.). When data bytes represent graphical shapes, the solid line of a byte is replaced by its actual bit pattern (in purple color).

There are a couple of interesting things to see:

- The figure reflects the ROM's separation into a data part (green and purple), a code part (blue), and one more data part (green and purple).
- The first data part contains mostly the custom

[3] In the movie TERMINATOR (1984) there are scenes showing the Terminator's point of view in shades of red. In these scenes lines of source code are listed onscreen. Close inspection of still frames of the movie reveal this to be 6502 assembly language source code.

[4] `git clone https://github.com/lwiest/StarRaiders` or `unzip pocorgtfo13.pdf StarRaiders.zip`

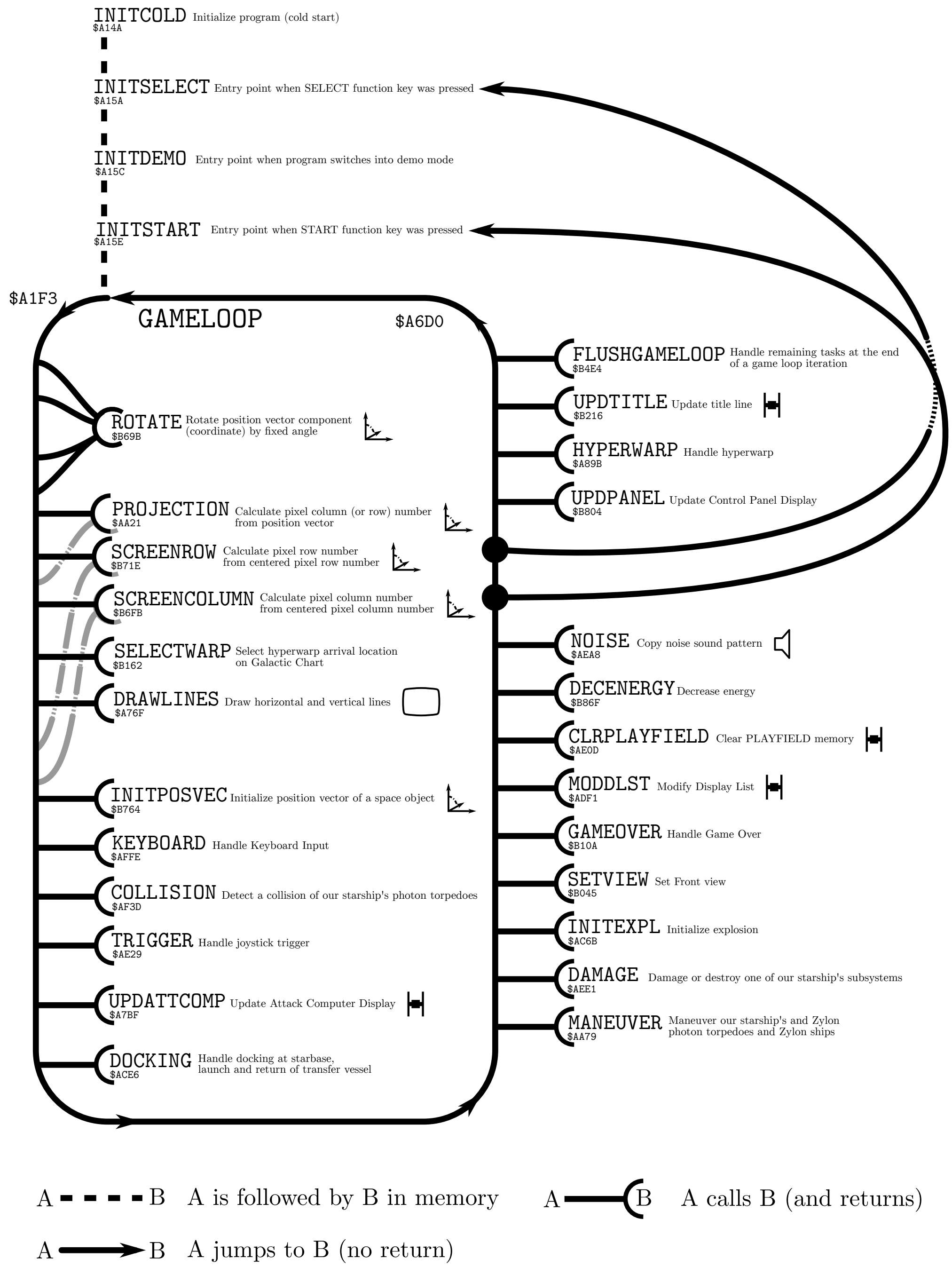


Figure 1. Simplified Call Graph of Start Up and Game Loop

$A14A	INITCOLD	Initialize program (Cold start)
$A15A	INITSELECT	Entry point when SELECT function key was pressed
$A15C	INITDEMO	Entry point when program switches into demo mode
$A15E	INITSTART	Entry point when START function key was pressed
$A1F3	GAMELOOP	Game loop
$A6D1	VBIHNDLR	Vertical Blank Interrupt Handler
$A718	DLSTHNDLR	Display List Interrupt Handler
$A751	IRQHNDLR	Interrupt Request (IRQ) Handler
$A76F	DRAWLINES	Draw horizontal and vertical lines
$A782	DRAWLINE	Draw a single horizontal or vertical line
$A784	DRAWLINE2	Draw blip in Attack Computer
$A7BF	UPDATTCOMP	Update Attack Computer Display
$A89B	HYPERWARP	Handle hyperwarp
$A980	ABORTWARP	Abort hyperwarp
$A987	ENDWARP	End hyperwarp
$A98D	CLEANUPWARP	Clean up hyperwarp variables
$A9B4	INITTRAIL	Initialize star trail during STAR TRAIL PHASE of hyperwarp
$AA21	PROJECTION	Calculate pixel column (or row) number from position vector
$AA79	MANEUVER	Maneuver our starship's and Zylon photon torpedoes and Zylon ships
$AC6B	INITEXPL	Initialize explosion
$ACAF	COPYPOSVEC	Copy a position vector
$ACC1	COPYPOSXY	Copy x and y components (coordinates) of position vector
$ACE6	DOCKING	Handle docking at starbase, launch and return of transfer vessel
$ADF1	MODDLST	Modify Display List
$AE0D	CLRPLAYFIELD	Clear PLAYFIELD memory
$AE0F	CLRMEM	Clear memory
$AE29	TRIGGER	Handle joystick trigger
$AEA8	NOISE	Copy noise sound pattern
$AECA	HOMINGVEL	Calculate homing velocity of our starship's photon torpedo 0 or 1
$AEE1	DAMAGE	Damage or destroy one of our starship's subsystems
$AF3D	COLLISION	Detect a collision of our starship's photon torpedoes
$AFFE	KEYBOARD	Handle Keyboard Input
$B045	SETVIEW	Set Front view
$B07B	UPDSCREEN	Clear PLAYFIELD, draw Attack
$B10A	GAMEOVER	Handle game over
$B121	GAMEOVER2	Game over (Mission successful)
$B162	SELECTWARP	Select hyperwarp arrival location on Galactic Chart
$B1A7	CALCWARP	Calculate and display hyperwarp energy
$B216	UPDTITLE	Update title line
$B223	SETTITLE	Set title phrase in title line
$B2AB	SOUND	Handle sound effects
$B3A6	BEEP	Copy beeper sound pattern
$B3BA	INITIALIZE	More game initialization
$B4B9	DRAWGC	Draw Galactic Chart
$B4E4	FLUSHGAMELOOP	Handle remaining tasks at the end of a game loop iteration
$B69B	ROTATE	Rotate position vector component (coordinate) by fixed angle
$B6FB	SCREENCOLUMN	Calculate pixel column number from centered pixel column number
$B71E	SCREENROW	Calculate pixel row number from centered pixel row number
$B764	INITPOSVEC	Initialize position vector of a space object
$B7BE	RNDINVXY	Randomly invert the x and y components of a position vector
$B7F1	ISSURROUNDED	Check **if** a sector is surrounded by Zylon units
$B804	UPDPANEL	Control Panel Display
$B86F	DECENERGY	Decrease energy
$B8A7	SHOWCOORD	Display a position vector component (coordinate) in Control Panel Display
$B8CD	SHOWDIGITS	Display a value by a readout of the Control Panel Display

Table 1. Star Raiders Subroutines

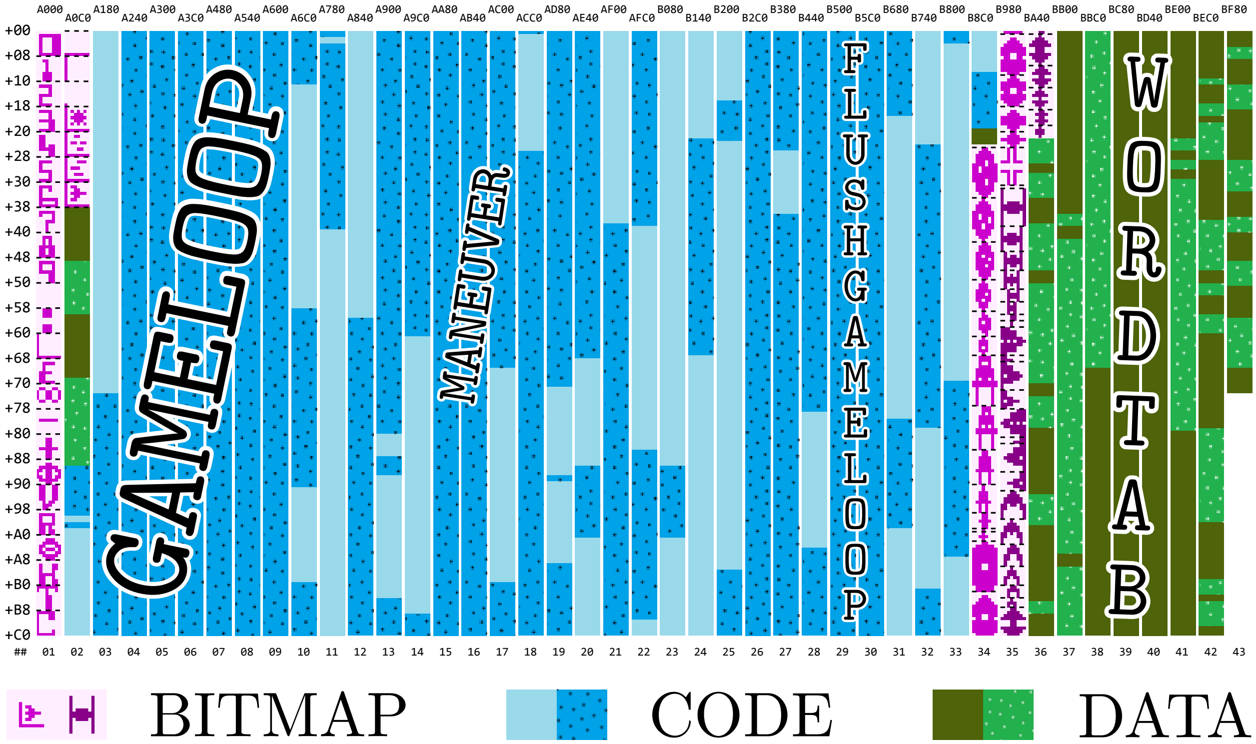


Figure 2. Genome Sequence of the STAR RAIDERS ROM

font (in strips 1-2).

- The largest contiguous (dark) blue chunk represents the 1246 bytes of the main game loop `GAMELOOP` (`$A1F3`) (in strips 3-10).
- At the beginning of the second data part are the shapes for the Players (sprites) (in strips 34-36).
- The largest contiguous (light) green chunk represents the 503 bytes of the game's word table `WORDTAB` (`$BC2B`) (in strips 38-41).

A good reverse engineering strategy was to start working from code locations that used Atari's published symbols, the equivalent of piecing together the border of a jigsaw puzzle first before starting to tackle the puzzle's center. Then, however, came the inevitable and very long stretch of reconstructing the game's logic and variables with a combination of educated guesses, trial-and-error, and lots of patience. At this stage, the tools I used mostly were nothing but a text editor (Notepad) and a word processor (Microsoft Word) to fill the gaps in the documentation of the code and the data. I also created a memory map text file to list the used memory locations and their purpose. These entries were continually updated – and more than often discarded after it turned out that I had taken a wrong turn.

2.3 A Programming Gem: Rotating 3D Vectors

What is the most interesting, fascinating, and unexpected piece of code in **STAR RAIDERS**? My pick would be the very code that started me to reverse engineer **STAR RAIDERS** in the first place: subroutine `ROTATE` (`$B69B`), which rotates objects in the game's 3D coordinate space (shown in Figure 3). And here is why: Rotation calculations usually involve trigonometry, matrices, and so on – at least some multiplications. But the 6502 CPU has only 8-bit addition and subtraction operations. It does not provide either a multiplication or a division operation – and certainly no trig operation! So how do the rotation calculations work, then?

Let's start with the basics: The game uses a 3D coordinate system with the position of our starship at the center of the coordinate system. The locations of all space objects (Zylon ships, meteors, photon torpedoes, starbase, transfer vessel, Hyperwarp Target Marker, stars, and explosion fragments) are described by a position vector relative to our starship.

A position vector is composed of an x, y, and z component, whose values I call the x, y, and z coordinates with the arbitrary unit <KM>. The range of a coordinate is -65536 to $+65535$ <KM>.

Each coordinate is a signed 17-bit integer number, which fits into three bytes. Bit 16 contains the sign bit, which is 1 for positive and 0 for negative sign. Bits 15 to 0 are the mantissa as a two's-complement integer.

```
Sign      Mantissa
     B16  B15...B8  B7....B0
       |  |      |  |      |
0000000*  ********  ********
```

Some example bit patterns for coordinates:

```
00000001 11111111 11111111 = +65535 <KM>
00000001 00000001 00000000 =   +256 <KM>
00000001 00000000 11111111 =   +255 <KM>
00000001 00000000 00000001 =     +1 <KM>
00000001 00000000 00000000 =     +0 <KM>
00000000 11111111 11111111 =     -1 <KM>
00000000 11111111 11111110 =     -2 <KM>
00000000 11111111 00000001 =   -255 <KM>
00000000 11111111 00000000 =   -256 <KM>
00000000 00000000 00000000 = -65536 <KM>
```

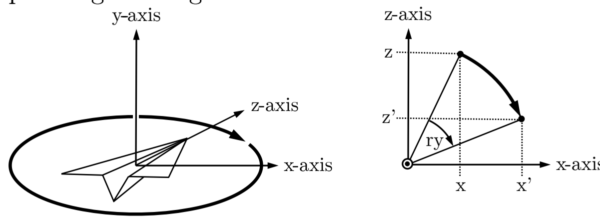
The position vector for each space object is stored in nine tables (3 coordinates × 3 bytes for each coordinate). There are up to 49 space objects used in the game simultaneously, so each table is 49 bytes long:

`XPOSSIGN` (`$09DE..$0A0E`)	`XPOSHI` (`$0A71..$0AA1`)	`XPOSLO` (`$0B04..$0B34`)
`YPOSSIGN` (`$0A0F..$0A3F`)	`YPOSHI` (`$0AA2..$0AD2`)	`YPOSLO` (`$0B35..$0B65`)
`ZPOSSIGN` (`$09AD..$09DD`)	`ZPOSHI` (`$0A40..$0A70`)	`ZPOSLO` (`$0AD3..$0B03`)

With that explained, let's have a look at subroutine `ROTATE` (`$B69B`). This subroutine rotates a position vector component (coordinate) of a space object by a fixed angle around the center of the 3D coordinate system, the location of our starship. This operation is used in 3 out of 4 of the game's view modes (Front view, Aft view, Long-Range Scan view) to rotate space objects in and out of the view.

2.3.1 Rotation Mathematics

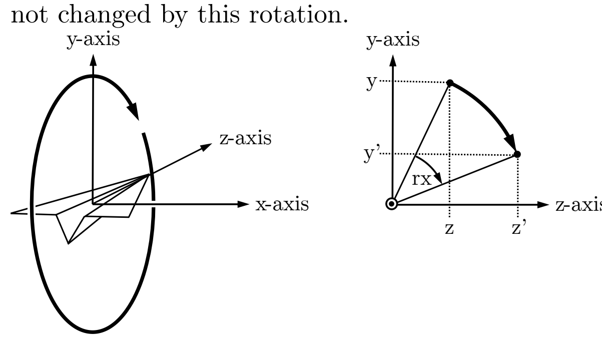
The game uses a left-handed 3D coordinate system with the positive x-axis pointing to the right, the positive y-axis pointing up, and the positive z-axis pointing into flight direction.



A rotation in this coordinate system around the y-axis (horizontal rotation) can be expressed as

$$x' = \cos(r_y)x + \sin(r_y)z \qquad (1)$$
$$z' = -\sin(r_y)x + \cos(r_y)z$$

where r_y is the clockwise rotation angle around the y-axis, x and z are the coordinates before this rotation, and the primed coordinates x' and z' the coordinates after this rotation. The y-coordinate is not changed by this rotation.



A rotation in this coordinate system around the x-axis (vertical rotation) can be expressed as

$$z' = \cos(r_x)z + \sin(r_x)y \qquad (2)$$
$$y' = -\sin(r_x)z + \cos(r_x)y$$

where r_x is the clockwise rotation angle around the x-axis, z and y are the coordinates before this rotation, and the primed coordinates z' and y' the coordinates after this rotation. The x-coordinate is not changed by this rotation.

2.3.2 Subroutine Implementation Overview

A single call of subroutine `ROTATE` (`$B69B`) is able to compute one of the four expressions in Equations 1 and 2. To compute all four expressions to get the new set of coordinates, this subroutine has to be called four times. This is done twice in pairs in `GAMELOOP` (`$A1F3`) at `$A391` and `$A398`, and at `$A3AE` and `$A3B5`, respectively.

The first pair of calls calculates the new x and z coordinates of a space object due to a horizontal (left/right) rotation of our starship around the y-axis following the expressions of Equation 1.

The second pair of calls calculates the new y and z coordinates of the same space object due to a vertical (up/down) rotation of our starship around the x-axis following the expressions of Equation 2.

If you look at the code of `ROTATE` (`$B69B`), you may be wondering how this calculation is actually executed, as there is neither a sine nor cosine function call. What you'll actually find implemented, however, are the following calculations:

Joystick Left

$$\begin{aligned} x &:= x + z/64 \\ z &:= -x/64 + z \end{aligned} \tag{3}$$

Joystick Right

$$\begin{aligned} x &:= x - z/64 \\ z &:= x/64 + z \end{aligned} \tag{4}$$

Joystick Down

$$\begin{aligned} y &:= y + z/64 \\ z &:= -y/64 + z \end{aligned} \tag{5}$$

Joystick Up

$$\begin{aligned} y &:= y - z/64 \\ z &:= y/64 + z \end{aligned} \tag{6}$$

2.3.3 CORDIC Algorithm

When you compare the expressions of Equations 1–2 with expressions of Equations 3–6, notice the similarity between the expressions if you substitute[5]

$$\begin{aligned} \sin(r_y) &\rightarrow 1/64 \\ \cos(r_y) &\rightarrow 1 \\ \sin(r_x) &\rightarrow 1/64 \\ \cos(r_x) &\rightarrow 1 \end{aligned}$$

From $\sin(r_y) = 1/64$ and $\sin(r_x) = 1/64$ you can derive that the rotation angles r_y and r_x by which the space object is rotated (per game loop iteration) have a constant value of 0.89°, as $\arcsin(1/64) = 0.89°$.

What about $\cos(r_y)$ and $\cos(r_x)$? The substitution does not match our derived angle exactly, because $\cos(0.89°) = 0.99988$ and is not exactly 1. However, this value is so close that substituting $\cos(0.89°)$ with 1 is a very good approximation, simplifying calculations significantly.

Another significant simplification results from the division by 64, as the actual division operation can be replaced with a much faster bit shift operation.

This calculation-friendly way of computing rotations is also known as the "CORDIC (COordinate Rotation DIgital Computer)" algorithm.

2.3.4 Minsky Rotation

There is one more interesting mathematical subtlety: Did you notice that expressions of Equations 1 and 2 use a new (primed) pair of variables to store the resulting coordinates, whereas in the implemented Equations 3–6, the value of the first coordinate of a coordinate pair is overwritten with its new value and this value is used in the subsequent calculation of the second coordinate? For example, when the joystick is pushed left, the first call of this subroutine calculates the new value of x according to first expression of Equation 3, overwriting the old value of x. During the second call to calculate z according to the second expression of Equation 3, the new value of x is used instead of the old one. Is this to save the memory needed to temporarily store the old value of x? Is this a bug? If so, why does the rotation calculation actually work?

Have a look at the expressions of Equation 3 (the other Equations 4–6 work in a similar fashion):

$$\begin{aligned} x &:= x + z/64 \\ z &:= -x/64 + z \end{aligned}$$

If we substitute 1/64 with e, we get

$$\begin{aligned} x &:= x + ez \\ z &:= -ex + z \end{aligned}$$

[5] *This substitution gave a friendly mathematician who happened to see it a nasty shock. She yelled at us that $cos^2x+sin^2x = 1$ for all real x and forever, and therefore this could not possibly be a rotation; it's a rotation with a stretch! We reminded her of the old joke that in wartime the value of the cosine has been known to reach 4. —PML*

Note that x is calculated first and then used in the second expression. When using primed coordinates for the resulting coordinates after calculating the two expressions we get

$$\begin{aligned} x' &:= x + ez \\ z' &:= -ex' + z \\ &= -e(x + ez) + z \\ &= -ex + (1 - e^2)z \end{aligned}$$

or in matrix form

$$\begin{pmatrix} x' \\ z' \end{pmatrix} = \begin{pmatrix} 1 & e \\ -e & 1 - e^2 \end{pmatrix} \begin{pmatrix} x \\ z \end{pmatrix}$$

Surprisingly, this turns out to be a rotation matrix, because its determinant is $(1 \times (1 - e^2) - (-e \times e)) = 1$. (Incidentally, the column vectors of this matrix do not form an orthogonal basis, as their scalar product is $1 \times e + (-e \times (1 - e^2)) = -e^2$. Orthogonality holds for $e = 0$ only.)

This kind of rotation calculation is described by Marvin Minsky in AIM 239 HAKMEM[6] and is called "Minsky Rotation."

2.3.5 Subroutine Implementation Details

To better understand how the implementation of this subroutine works, we must again look at Equations 3–6. If you rearrange the expressions a little, their structure is always of the form:

$$\texttt{TERM1} := \texttt{TERM1}\ \texttt{SIGN}\ \texttt{TERM2}/64$$

or shorter

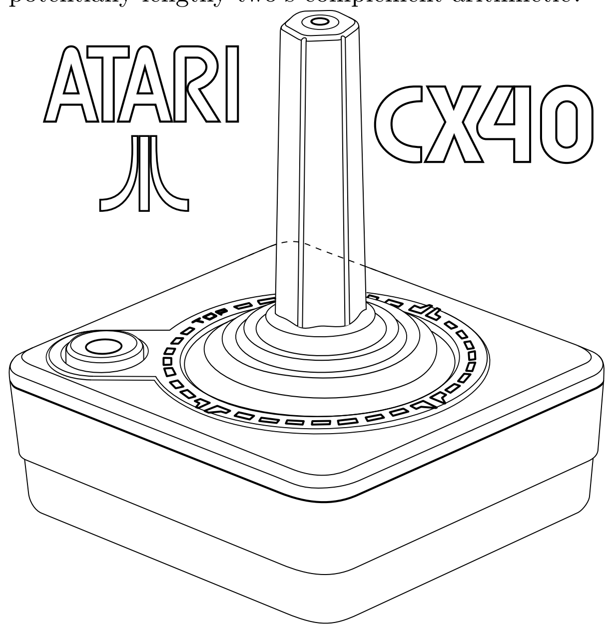
$$\texttt{TERM1} := \texttt{TERM1}\ \texttt{SIGN}\ \texttt{TERM3}$$

where `TERM3` := `TERM2`/64 and `SIGN` := + or − and where `TERM1` and `TERM2` are coordinates. In fact, this is all this subroutine actually does: It simply adds `TERM2` divided by 64 to `TERM1` or subtracts `TERM2` divided by 64 from `TERM1`.

When calling this subroutine the correct table indices for the appropriate coordinates `TERM1` and `TERM2` are passed in the CPU's `Y` and `X` registers, respectively.

What about `SIGN` between `TERM1` and `TERM3`? Again, have a look at Equations 3–6. To compute the two new coordinates after a rotation, the `SIGN` toggles from plus to minus and vice versa. The `SIGN` is initialized with the value of `JOYSTICKDELTA` (`$6D`) before calling subroutine `ROTATE` (`$B69B`, Figure 3) and is toggled in every call of this subroutine. The initial value of `SIGN` should be positive (+, byte value `$01`) if the rotation is clockwise (the joystick is pushed right or up) and negative (−, byte value `$FF`) if the rotation is counter-clockwise (the joystick is pushed left or down), respectively. Because `SIGN` is always toggled in `ROTATE` (`$B69B`) before the adding or subtraction operation of `TERM1` and `TERM3` takes place, you have to pass the already toggled value with the first call.

Unclear still are three instructions starting at address `$B6AD`. They seem to set the two least significant bits of `TERM3` in a random fashion. Could this be some quick hack to avoid messing with exact but potentially lengthy two's-complement arithmetic?



2.4 Dodging Memory Limitations

It is impressing how much functionality was squeezed into **STAR RAIDERS**. Not surprisingly, the bytes of the 8 KB ROM are used up almost completely. Only a single byte is left unused at the very end of the code. When counting four more bytes from three orphaned entries in the game's lookup tables, only five bytes in total out of 8,192 bytes are actually not used. ROM memory was extremely precious. Here are some techniques that demonstrate

[6] `unzip pocorgtfo13.pdf AIM-239.pdf` #Item 149, page 73.

```
                ; INPUT
                ;
                ; X = Position vector component index of TERM2. Used values are:
                ; $00..$30 -> z-component (z-coordinate) of position vector 0..48
                ; $31..$61 -> x-component (x-coordinate) of position vector 0..48
                ; $62..$92 -> y-component (y-coordinate) of position vector 0..48
                ;
                ; Y = Position vector component index of TERM1. Used values are:
                ; $00..$30 -> z-component (z-coordinate) of position vector 0..48
                ; $31..$61 -> x-component (x-coordinate) of position vector 0..48
                ; $62..$92 -> y-component (y-coordinate) of position vector 0..48
                ;
                ; JOYSTICKDELTA ($6D) = Initial value of SIGN. Used values are:
                ; $01 -> (= Positive) Rotate right or up
                ; $FF -> (= Negative) Rotate left or down

                                                ; TERM3 is a 24-bit value, represented by 3 bytes as
                                                ; $(sign)(high byte)(low byte)
     =006A      L.TERM3LO   = $6A               ; TERM3 (high byte), where TERM3 := TERM2 / 64
     =006B      L.TERM3HI   = $6B               ; TERM3 (low byte),  where TERM3 := TERM2 / 64
     =006C      L.TERM3SIGN = $6C               ; TERM3 (sign),      where TERM3 := TERM2 / 64

B69B BDAD09     ROTATE      LDA ZPOSSIGN,X      ;
B69E 4901                   EOR #$01            ;
B6A0 F002                   BEQ SKIP224         ; Skip if sign of TERM2 is positive
B6A2 A9FF                   LDA #$FF            ;

B6A4 856B       SKIP224     STA L.TERM3HI       ; If TERM2 pos. -> TERM3 := $0000xx (= TERM2 / 256)
B6A6 856C                   STA L.TERM3SIGN     ; If TERM2 neg. -> TERM3 := $FFFFxx (= TERM2 / 256)
B6A8 BD400A                 LDA ZPOSHI,X        ; where xx := TERM2 (high byte)
B6AB 856A                   STA L.TERM3LO       ;

B6AD AD0AD2                 LDA RANDOM          ; (?) Hack to avoid messing with two-complement's
B6B0 09BF                   ORA #$BF            ; (?) arithmetic? Provides two least significant
B6B2 5DD30A                 EOR ZPOSLO,X        ; (?) bits B1..0 in TERM3.

B6B5 0A                     ASL A               ; TERM3 := TERM3 * 4 (= TERM2 / 256 * 4 = TERM2 / 64)
B6B6 266A                   ROL L.TERM3LO       ;
B6B8 266B                   ROL L.TERM3HI       ;
B6BA 0A                     ASL A               ;
B6BB 266A                   ROL L.TERM3LO       ;
B6BD 266B                   ROL L.TERM3HI       ;

B6BF A56D                   LDA JOYSTICKDELTA   ; Toggle SIGN for next call of ROTATE
B6C1 49FF                   EOR #$FF            ;
B6C3 856D                   STA JOYSTICKDELTA   ;
B6C5 301A                   BMI SKIP225         ; If SIGN negative then subtract, else add TERM3

                ;*** Addition ***************************************************************
B6C7 18                     CLC                 ; TERM1 := TERM1 + TERM3
B6C8 B9D30A                 LDA ZPOSLO,Y        ; (24-bit addition)
B6CB 656A                   ADC L.TERM3LO       ;
B6CD 99D30A                 STA ZPOSLO,Y        ;

B6D0 B9400A                 LDA ZPOSHI,Y        ;
B6D3 656B                   ADC L.TERM3HI       ;
B6D5 99400A                 STA ZPOSHI,Y        ;

B6D8 B9AD09                 LDA ZPOSSIGN,Y      ;
B6DB 656C                   ADC L.TERM3SIGN     ;
B6DD 99AD09                 STA ZPOSSIGN,Y      ;
B6E0 60                     RTS                 ;

                ;*** Subtraction ************************************************************
B6E1 38         SKIP225     SEC                 ; TERM1 := TERM1 - TERM3
B6E2 B9D30A                 LDA ZPOSLO,Y        ; (24-bit subtraction)
B6E5 E56A                   SBC L.TERM3LO       ;
B6E7 99D30A                 STA ZPOSLO,Y        ;

B6EA B9400A                 LDA ZPOSHI,Y        ;
B6ED E56B                   SBC L.TERM3HI       ;
B6EF 99400A                 STA ZPOSHI,Y        ;

B6F2 B9AD09                 LDA ZPOSSIGN,Y      ;
B6F5 E56C                   SBC L.TERM3SIGN     ;
B6F7 99AD09                 STA ZPOSSIGN,Y      ;
B6FA 60                     RTS                 ;
```

Figure 3. ROTATE Subroutine at $B69B

the fierce fight for each spare ROM byte.

2.4.1 Loop Jamming

Loop jamming is the technique of combining two loops into one, reusing the loop index and optionally skipping operations of one loop when the loop index overshoots.

How much bytes are saved by loop jamming? As an example, Figure 4 shows an original 19-byte fragment of subroutine `INITIALIZE` (`$B3BA`) using loop jamming. The same fragment without loop jamming, shown in Figure 5, is 20 bytes long. So loop jamming saved one single byte.

Another example is the loop that is set up at `$A165` in `INITCOLD` (`$A14A`). A third example is the loop set up at `$B413` in `INITIALIZE` (`$B3BA`). This loop does not explicitly skip loop indices, thus saving four more bytes (the `CMP` and `BCS` instructions) on top of the one byte saved by regular loop jamming. Thus, seven bytes are saved in total by loop jamming.

2.4.2 Sharing Blank Characters

One more technique to save bytes is to let strings share their leading and trailing blank characters. In the game there is a header text line of twenty characters that displays one of the strings "`LONG RANGE SCAN`," "`AFT VIEW`," or "`GALACTIC CHART`." The display hardware directly points to their location in the ROM. They are enclosed in blank characters (bytes of value `$00`) so that they appear horizontally centered.

A naive implementation would use $3 \times 20 = 60$ bytes to store these strings in ROM. In the actual implementation, however, the trailing blanks of one header string are reused as leading blanks of the following header, as shown in Figure 6. By sharing blank characters the required memory is reduced from 60 bytes to 54 bytes, saving six bytes.

2.4.3 Reusing Interrupt Exit Code

Yet another, rather traditional technique is to reuse code, of course. Figure 7 shows the exit code of the Vertical Blank Interrupt handler `VBIHNDLR` (`$A6D1`) at `$A715`, which jumps into the exit code of the Display List Interrupt handler `DLSTHNDLR` (`$A718`) at `$A74B`, reusing the code that restores the registers that were put on the CPU stack before entering the Vertical Blank Interrupt handler.

This saves another six bytes (`PLA`, `TAY`, `PLA`, `TAX`, `PLA`, `RTI`), but spends three bytes (`JMP JUMP004`), in total saving three bytes.

2.5 Bugs

There are a few bugs, or let's call them glitches, in **STAR RAIDERS**. This is quite astonishing, given the complex game and the development tools of 1979, and is a testament to thorough play testing. The interesting thing is that the often intense game play distracts the players' attention from noticing these glitches, just like what a skilled parlor magician would do.

2.5.1 A Starbase Without Wings

When a starbase reaches the lower edge of the graphics screen and overlaps with the Control Panel Display below (Figure 8 (left), screenshot) and you nudge the starbase a little bit more downward, its wings suddenly vanish (Figure 8 (right), screenshot).

The reason is shown in the insert on the right side of the figure: The starbase is a composite of three Players (sprites). Their bounding boxes are indicated by three white rectangles. If the vertical position of the top border of a Player is larger than a vertical position limit, indicated by the tip of the white arrow, the Player is not displayed. The relevant location of the comparison is at `$A534` in `GAMELOOP` (`$A1F3`). While the Player of the central part of the starbase does not exceed this vertical limit, the Players that form the starbase's wings do so, and are thus not rendered.

This glitch is rarely noticed because players do their best to keep the starbase centered on the screen, a prerequisite for a successful docking.

2.5.2 Shuffling Priorities

There are two glitches that are almost impossible to notice (and I admit some twisted kind of pleasure to expose them, ;-):

- During regular gameplay, the Zylon ships and the photon torpedoes appear *in front of* the cross hairs (Figure 9 (left)), as if the cross hairs were light years away.
- During docking, the starbase not only appears *behind* the stars (Figure 9 (right)) as if the starbase is light years away, but the transfer vessel moves *in front of* the cross hairs!

```
B3BA A259      INITIALIZE    LDX #89             ; Set 89(+1) GRAPHICS7 rows from DSPLST+5 on
B3BC A90D      LOOP060       LDA #$0D            ; Prep DL instruction $0D (one row of GRAPHICS7)
B3BE 9D8502                  STA DSPLST+5,X      ; DSPLST+5,X := one row of GRAPHICS7
B3C1 E00A                    CPX #10             ;
B3C3 B005                    BCS SKIP195         ;
B3C5 BDA9BF                  LDA PFCOLORTAB,X    ; Copy PLAYFIELD color table to zero-page table
B3C8 95F2                    STA PF0COLOR,X      ; (loop jamming)
B3CA CA        SKIP195       DEX                 ;
B3CB 10EF                    BPL LOOP060         ;
```

Figure 4. `INITIALIZE` Subroutine at `$B3BA` (Excerpt)

```
B3BA A259      INITIALIZE    LDX #89             ; Set 89(+1) GRAPHICS7 rows from DSPLST+5 on
B3BC A90D      LOOP060       LDA #$0D            ; Prep DL instruction $0D (one row of GRAPHICS7)
B3BE 9D8502                  STA DSPLST+5,X      ; DSPLST+5,X := one row of GRAPHICS7
B3C1 CA                      DEX                 ;
B3C2 10F8                    BPL LOOP060         ;
B3C4 A209                    LDX #9              ;
B3C6 BDAABF    LOOP060B      LDA PFCOLORTAB,X    ; Copy PLAYFIELD color table to zero-page table
B3C9 95F2                    STA PF0COLOR,X      ;
B3CB CA                      DEX                 ;
B3CC 10F8                    BPL LOOP060B        ;
```

Figure 5. `INITIALIZE` Subroutine Without Loop Jamming (Excerpt)

The reason is the drawing order or "graphics priority" of the bit-mapped graphics and the Players (sprites). It is controlled by the `PRIOR` (`$D01B`) hardware register.

During regular flight, see Figure 9 (left), `PRIOR` (`$D01B`) has a value of `$11`. This arranges the displayed elements in the following order, from front to back:

- Players 0-4 (photon torpedoes, Zylon ships, ...)
- Bit-mapped graphics (stars, cross hairs)
- Background.

This arrangement is fine for the stars as they are bit-mapped graphics and need to appear behind the photon torpedoes and the Zylon ships, but this arrangement applies also to the cross hairs – causing the glitch.

During docking, see Figure 9 (right), `PRIOR` (`$D01B`) has a value of `$14`. This arranges the displayed elements the following order, from front to back:

- Player 4 (transfer vessel)
- Bit-mapped graphics (stars, cross hairs)
- Players 0-3 (starbase, ...)
- Background.

This time the arrangement is fine for the cross hairs as they are bit-mapped graphics and need to appear in front of the starbase, but this arrangement also applies to the stars. In addition, the Player of the white transfer vessel correctly appears in front of the bit-mapped stars, but also in front of the bit-mapped cross hairs.

Fixing these glitches is hardly possible, as the display hardware does not allow for a finer control of graphics priorities for individual Players.

2.6 A Mysterious Finding

A simple instruction at location `$A175` contained the most mysterious finding in the game's code. The disassembler reported the following instruction, which is equivalent to `STA $0067,X`. (`ISVBISYNC` has a value of `$67`.)

```
A175 9D6700     STA ISVBISYNC,X
```

The object code assembled from this instruction is unusual as its address operand was assembled as a 16-bit address and not as an 8-bit zero-page address. Standard 6502 assemblers would always generate shorter object code, producing `9567` (`STA $67,X`) instead of `9D6700` and saving a byte.

In my reverse engineered source code, the only way to reproduce the original object code was the following:

```
                ;*** Header text of Long-Range Scan view (shares spaces with following header) *
A0F8 00006C6F LRSHEADER        .BYTE $00,$00,$6C,$6F,$6E,$67,$00,$72 ; `` LONG RANGE SCAN''
A0FC 6E670072
A100 616E6765                  .BYTE $61,$6E,$67,$65,$00,$73,$63,$61
A104 00736361
A108 6E                        .BYTE $6E

                ;*** Header text of Aft view (shares spaces with following header) *************
A109 00000000 AFTHEADER        .BYTE $00,$00,$00,$00,$00,$00,$61,$66 ; ``      AFT VIEW   ``
A10D 00006166
A111 74007669                  .BYTE $74,$00,$76,$69,$65,$77,$00,$00
A115 65770000
A119 00                        .BYTE $00

                ;*** Header text of Galactic Chart view ****************************************
A11A 00000067 GCHEADER         .BYTE $00,$00,$00,$67,$61,$6C,$61,$63 ; ``   GALACTIC CHART   ``
A11E 616C6163
A122 74696300                  .BYTE $74,$69,$63,$00,$63,$68,$61,$72
A126 63686172
A12A 74000000                  .BYTE $74,$00,$00,$00
```

Figure 6. Header Texts at `$A0F8`

```
A6D1 A9FF     VBIHNDLR         LDA #$FF                ; Start of Vertical Blank Interrupt handler
              ...
A715 4C4BA7   SKIP046          JMP JUMP004             ; End of Vertical Blank Interrupt handler
              ...
A718 48       DLSTHNDLR        PHA                     ; Start of Display List Interrupt handler
              ...
A74B 68       JUMP004          PLA                     ; Restore registers
A74C A8                        TAY                     ;
A74D 68                        PLA                     ;
A74E AA                        TAX                     ;
A74F 68                        PLA                     ;
A750 40                        RTI                     ; End of Display List Interrupt Handler
```

Figure 7. `VBIHNDLR` and `DLSTHNDLR` Handlers Share Exit Code

```
; HACK: Fake STA ISVBISYNC,X with 16b addr
A175 9D         .BYTE $9D
A176 6700       .WORD ISVBISYNC
```

I speculated for a long time whether this strange assembler output indicated that the object code of the original ROM cartridge was produced with a non-standard 6502 assembler. I have heard that Atari's in-house development systems ran on PDP-11 hardware. Luckily, the month after I finished my reverse engineering effort, the original **STAR RAIDERS** source code re-surfaced.[7] To my astonishment it uses exactly the same "hack" to reproduce the three-byte form of the `STA ISVBISYNC,X` instruction:

```
A175 9D       .BYTE $9D    ; STA ABS,X
A176 67 00    .WORD PAGE0  ; STA PAGE0,X (ABSOLUTE)
```

Unfortunately the comments do not give a clue why this pattern was chosen. After quite some time it made click: The instruction `STA ISVBISYNC,X` is used in a loop which iterates the CPU's X register from 0 to 255 to clear memory. By using this instruction with a 16-bit address ("indexed" mode operand) memory from `$0067` to `$0166` is cleared. Had the code been using the same operation with an 8-bit address ("indexed, zero-page" mode operand), memory from `$0067` to `$00FF` would have been cleared, then the indexed address would have wrapped back to `$0000` clearing memory `$0000` to `$0066`, effectively overwriting already initialized memory locations.

2.7 Documenting Star Raiders

Right from the start of reverse engineering **STAR RAIDERS** I not only wanted to understand how the game worked, but I also wanted to document the result of my effort. But what would be an appropriate form?

First, I combined the emerging memory map file with the fledgling assembly language source code in

[7] `https://archive.org/details/AtariStarRaidersSourceCode`
`unzip pocorgtfo13.pdf StarRaidersOrig.pdf`

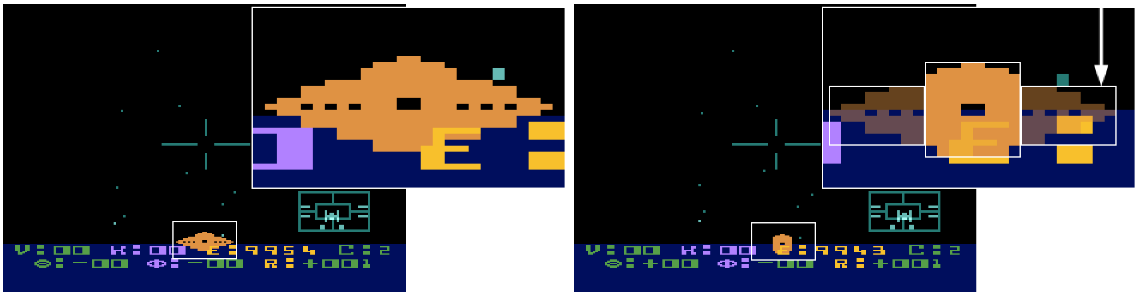


Figure 8. A Starbase's Wings Vanish

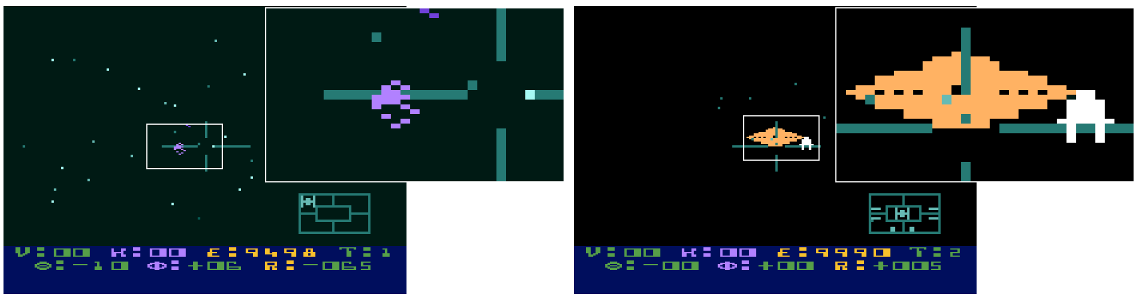


Figure 9. Photon torpedo in front of cross hairs and a starbase behind the stars!

order to work with just one file. Then, I switched the source code format to that of MAC/65, a well-known and powerful macro assembler for the Atari 8-bit Home Computer System. I also planned, at some then distant point in the future, to assemble the finished source code with this assembler on an 8-bit Atari.

Another major influence on the emerging documentation was the Atari BASIC Source Book, which I came across by accident[8]. It reproduced the complete, commented assembly language source code of the 8 KB Atari BASIC interpreter cartridge, a truly non-trivial piece of software. But what was more: The source code was accompanied by several chapters of text that explained in increasing detail its concepts and architecture, that is, how Atari BASIC actually worked. Deeply impressed, I decided on the spot that my reverse engineered **STAR RAIDERS** source code should be documented at the same level of detail.

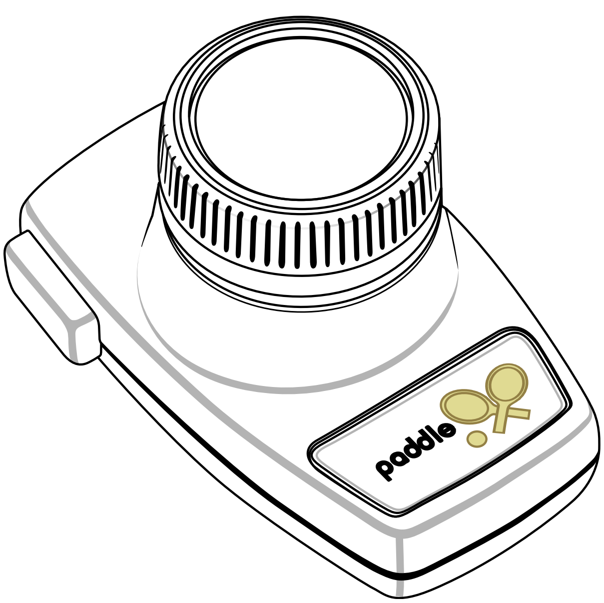
The overall documentation structure for the source code, which I ended up with was fourfold: On the lowest level, end-of-line comments documented the functionality of individual instructions. On the next level, line comments explained groups of instructions. One level higher still, comments composed of several paragraphs introduced each subroutine. These paragraphs provided a summary of the subroutine's implementation and a description of all input and output parameters, including the valid value ranges, if possible. On the highest level, I added the memory map to the source code as a handy reference. I also planned to add some chapters on the game's general concepts and overall architecture, just like the Atari BASIC Source Book had done. Unfortunately, I had to drop that idea due to lack of time. I also felt that the detailed subroutine documentation was quite sufficient. However, I did add sections on the 3D coordinate system and the position and velocity vectors to the source code as a tip of the hat to the Atari BASIC Source Book.

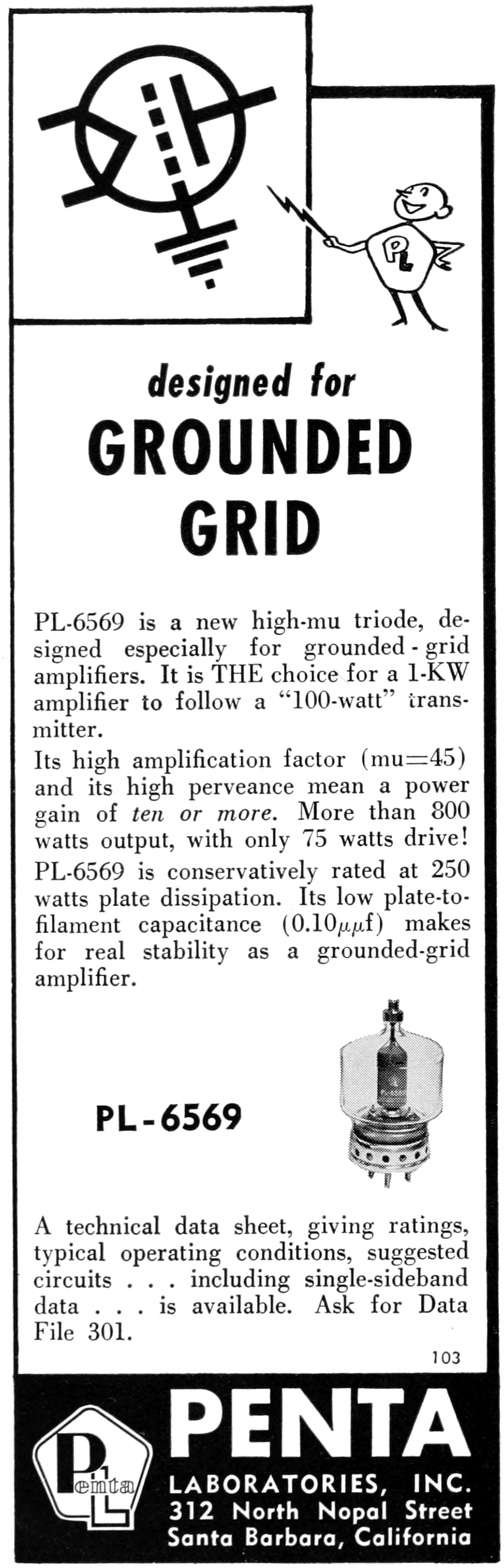
After I was well into reverse engineering **STAR RAIDERS**, slowly adding bits and pieces of information to the raw disassembly of the **STAR RAIDERS** ROM and fleshing out the ever growing documentation, I started to struggle with establishing a consistent and uniform terminology for the documentation (Is it "asteroid," "meteorite," or "meteor"? "Explosion bits," "explosion debris," or "explosion fragments"? "Gun sights" or "cross hairs"?) A look into the **STAR RAIDERS** instruction manual clarified only

[8]The Atari BASIC Source Book by Wilkinson, O'Brien, and Laughton. A COMPUTE! publication.

a painfully small amount of cases. Incidentally, it also contradicted itself as it called the enemies "Cylons" while the game called them "Zylons," such as in the message "`SHIP DESTROYED BY ZYLON FIRE`."

But I was not only after uniform documentation, I also wanted to unify the symbol names of the source code. For example, I had created a hodge-podge of color-related symbol names, which contained fragments such as "`COL`," "`CLR`," "`COLR`," and "`COLOR`." To make matters worse, color-related symbol names containing "`COL`" could be confused with symbol names related to (pixel) columns. The same occurred with symbol names related to Players (sprites), which contained fragments such as "`PL`," "`PLY`," "`PLYR`," "`PLAY`," and "`PLAYER`," or with symbol names of lookup tables, which ended in "`TB`," "`TBL`," "`TAB`," and "`TABLE`," and so on. In addition to inventing uniform symbol names I also did not want to exceed a self-imposed symbol name limit of 15 characters. So I refactored the source code with the search-and-replace functionality of the text editor over and over again.





I noticed that I spent more and more time on refactoring the documentation and the symbol names and less time on adding actual content. In addition, the actual formatting of the emerging documented source code had to be re-adjusted after every refactoring step. Handling the source code became very unwieldy. And worst of all: How could I be sure that the source code still represented the exact binary image of the ROM cartridge?

The solution I found to this problem eventually was to create an automated build pipeline, which dealt with the monotonous chores of formatting and assembling the source code, as well as comparing the produced ROM cartridge image with a reference image. This freed time for me to concentrate on the actual source code content. Yet another incarnation of "separation of form and content," the automated build pipeline was always a pleasure to watch working its magic. (Mental note: I should have created this pipeline much earlier in the reverse engineering effort.) These are the steps of the automated build pipeline:

1. The pipeline starts with a raw, documented assembly language source code file. It is already roughly formatted and uses a little proprietary markup, just enough to mark up sections of meta-comments that are to be removed in the output as well as subroutine documentation containing multiple paragraphs, numbered, and unnumbered lists. This source code file is fed to a pre-formatter program, which I implemented in Java. The pre-formatter removes the meta-comments. It also formats the entries of the memory map and the subroutine documentation by wrapping multi-line text at a preset right margin, out- and indenting list items, numbering lists, and vertically aligning parameter descriptions. It also corrects the number of trailing asterisks in line comments, and adjusts the number of asterisks of the box headers that introduce subroutine comments, centering their text content inside the asterisk boxes.

2. The output of the pre-formatter from step 1 is fed into an Atari 6502 assembler, which I also wrote in Java. It is available as open-source on GitHub.[9] Why write an Atari 6502 assembler? There are other 6502 assemblers readily available, but not all produce object code for the Atari 8-bit Home Computer System, not all use the MAC/65 source code format, and not all of them can be easily tweaked when necessary. The output of this step is both an assembler output listing and an object file.

3. The assembler output listing from step 2 is the finished, formatted, reverse engineered **STAR RAIDERS** source code, containing the documentation, the source code, and the object code listing.

4. The assembler output listing from step 2 is fed into a symbol checker program, which I again wrote in Java. It searches the documentation parts of the assembler output listing and checks if every symbol, such as "`GAMELOOP`," is followed by its correct hex value, "(`$A1F3`)." It reports any symbol with missing or incorrect hex values. This ensures further consistency of the documentation.

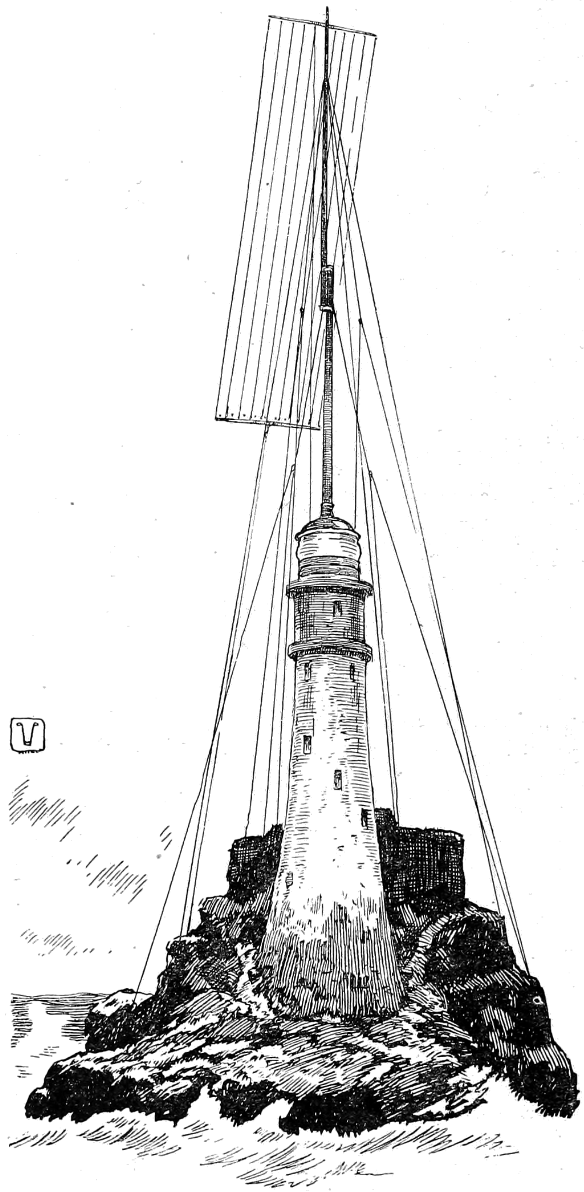
5. The object file of step 2 is converted by yet another program I wrote in Java from the Atari executable format into the final Atari ROM cartridge format.

6. The output from step 5 is compared with a reference binary image of the original **STAR RAIDERS** 8 KB ROM cartridge. If both images are the same, then the entire build was successful: The raw assembly language source code really represents the exact image of the **STAR RAIDERS** 8 KB ROM cartridge

[9] `git clone https://github.com/lwiest/Atari6502Assembler`
`unzip pocorgtfo13.pdf Atari6502Assembler.zip`

Typical build times on my not-so-recent Windows XP box (512 MB) were 15 seconds.

For some finishing touches, I ran a spell-checker over the documented assembly language source code file from time to time, which also helped to improve documentation quality.



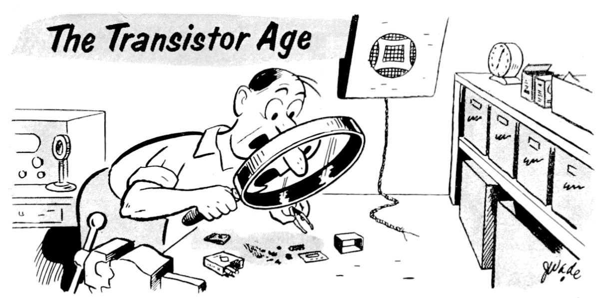
FASTNET LIGHT AS IT WOULD APPEAR IF CONVERTED INTO A "BLIND LIGHTHOUSE."

2.8 Conclusion

After quite some time, I achieved my goal to create a reverse engineered, complete, and fully documented assembly language source code of **STAR RAIDERS**. For final verification, I successfully assembled it with MAC/65 on an Atari 800 XL with 64 KB RAM (emulated with Atari800Win Plus). MAC/65 is able to assemble source code larger than the available RAM by reading the source code as several chained files. So I split the source code (560 KB) into chunks of 32 KB and simply had the emulator point to a hard disk folder containing these files. The resulting assembler output listing and the object file were written back to the same hard disk folder. The object file, after being transformed into the Atari cartridge format, exactly reproduced the original **STAR RAIDERS** 8 KB ROM cartridge.

2.9 Postscript

I finished my reverse engineering effort in September 2015. I was absolutely thrilled to learn that in October 2015 scans of the original **STAR RAIDERS** source code re-surfaced. To my delight, inspection of the original source code confirmed the findings of my reverse engineered version and caused only a few trivial corrections. Even more, the documentation of my reverse engineered version added a substantial amount of information – from overall theory of operation down to some tricky details – to the understanding of the often sparsely commented original (quite expected for source code never meant for publication).



"Now, where is that audio amplifier?"