Tutorial Walkthrough- How To Correct Rastaconverter Line Artefacts

Background Rastaconverter is an image rendering application that runs on a PC but is targeted at the Atari 8-bit computer line. It takes an input .png image and outputs an executable file (with a .xex extension) that displays an approximation of the original input image when executed on an 8-bit Atari. The algorithm that renders the image for display on an Atari makes use of an ANTIC graphics Mode E display list of 240 scanlines to produce a basic screen resolution of 160x240 pixels in 4 colours selected from a palette of 120 (PAL systems) or 128 (NTSC systems). Each pixel has an aspect ratio of 2:1, meaning the image has an overall aspect ratio of 4:3.

Various advanced techniques are applied in the algorithm to increase the number of colours displayed in the image. Overlaying player (sprite) graphics increases the basic number of colours displayable on a scanline (horizontal row of pixels) to 8, although at a basic level each of the 4 player colours can only appear on the image within a vertical band 32 pixels wide. However, Rastaconverter also uses a CPU kernel synchronised to the (actual or emulated) TV display to update any of the 8 available colours 'on-the-fly' to any other colour from the available palette, so that, subject to timing constraints (i.e. how fast the CPU can update colour registers), more than 8 colours can appear on any given scanline and the full available palette of 120 or 128 colours can be displayed within the full image.

Furthermore, the 32-pixel-wide vertical bands within which each of the 4 player colours may appear can also be shifted back & forth 'on-the-fly' so that they are not in fact constrained to the same horizontal position throughout the vertical height of the image. The first player's 32-pixel-wide band of colour overlay may therefore, for example, appear at the left margin of the image on some scan lines but on the right or centrally in others. Within timing constraints, players can even be 'reused' within a scanline such that, for example, the first player's colour overlay can appear showing one colour at the left margin and the same or a different colour at the right margin of the image all within the same horizontal line of pixels.

To achieve these tricks the CPU must update the register defining the horizontal position of a player 'on-the-fly', and this gives rise to one of the few annoyances of Rastaconverter. Due to timing delays in the Atari video circuitry, the electron beam 'drawing' the image on the TV screen has moved on 5 pixels (or, in video hardware terms, colour clocks) horizontally to the right before updates to the player horizontal position register are ready to trigger the display of a player. If the intended updated horizontal position (left margin) of the player falls within those 5 pixels, display of the player at the updated position will not be triggered on that scanline. The Rastaconverter algorithm is not 'aware' of this delay, with the result that not infrequently the kernel it generates inadvertently attempts to reposition players within the critical 5-pixel window, leading to a 'misfire' of a player's colour overlay on that particular scanline.

The outcome of this 'misfire' in the image displayed by the Atari executable is a short (often broken) horizontal line of an obviously wrong colour -one that doesn't appear in the corresponding .png image that Rastaconverter outputs to show how the algorithm believes the Atari image should appear. The background colour pixels the algorithm intended to be 'overlaid' by said player

colour on the relevant scanline are displayed instead- the 'incorrect' colour seen in the artefact represents these background pixels 'showing through'.

The solution is usually to reprogram the kernel to avoid the player reposition occurring (i.e. a CPU write to the relevant player horizontal position register HPOSPx completing) within the critical 5 colour clock window before the player is due to be displayed. This can be very easy or very tricky depending on what other critically-timed instructions fall in the vicinity. In the particular case of an attempted 'reuse' of a player within a scanline, the window of opportunity for completion of the write to HPOSPx lies between the triggering of the player at its previous position and the 5 colour-clock window during which potential re-triggering is suspended. Note that is possible to retrigger a player while it is still in the process of being displayed at its previous position, such that the repositioned & 'old' players overlap, or more accurately, display of the 'old' player is prematurely cut short in oder to start displaying the repositioned one.

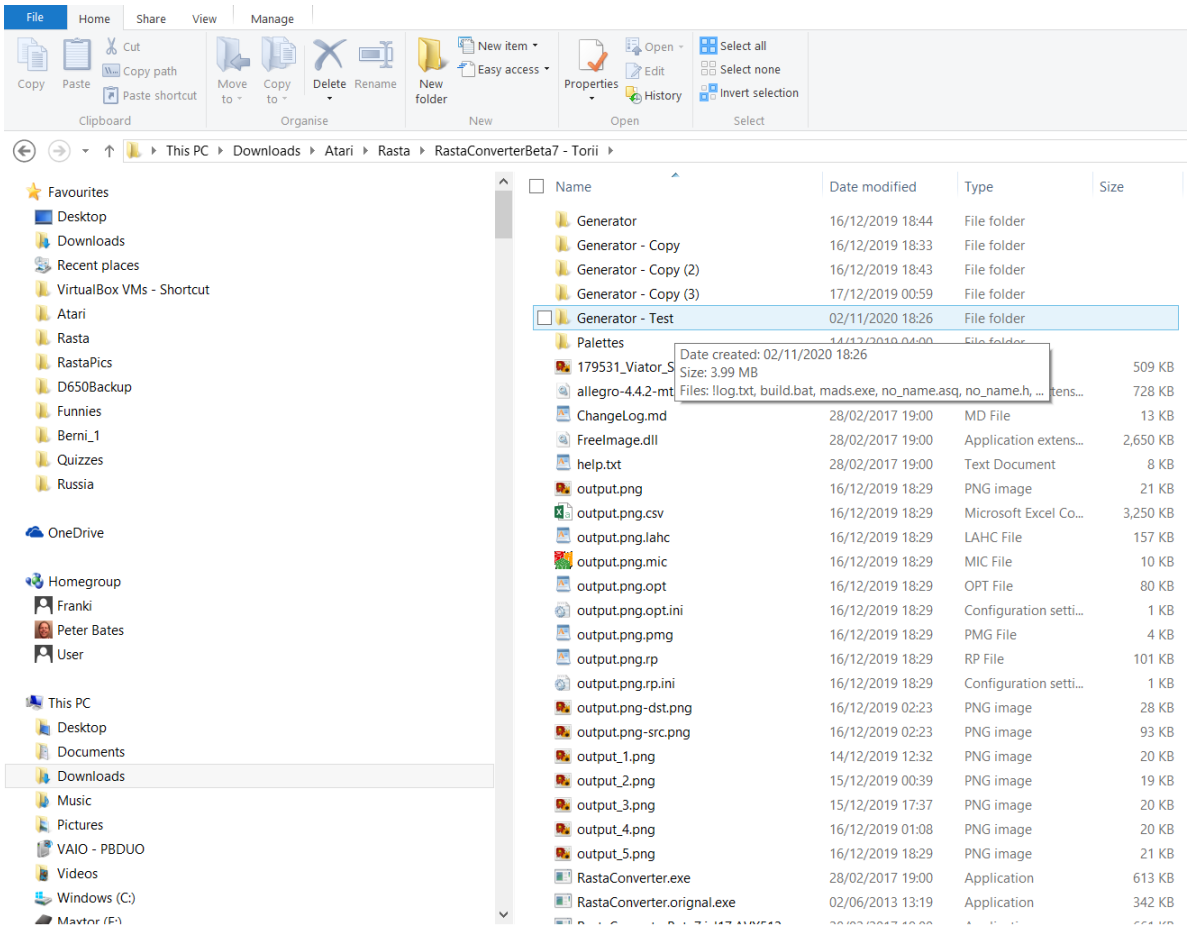
Occasionally the only simple way out is to edit the bitmap and/or the player missile data to cover up the offending pixels with either a playfield colour or another player colour. Obviously, how good a solution that turns out to be depends on how critical it is to the image that the pixels in the artefact are the colour Rastaconverter originally intended. Sometimes an alternate player used to 'cover up' the artefact can be switched if required to be the 'correct' intended colour then back again, or the background colour itself can be switched to and from the 'correct' colour during the relevant horizontal segment of the scanline.

Example walkthrough There follows a walked-through example of a manual 'repair' to a faulty Rastaconverter kernel responsible for an obvious 'line artefact'.

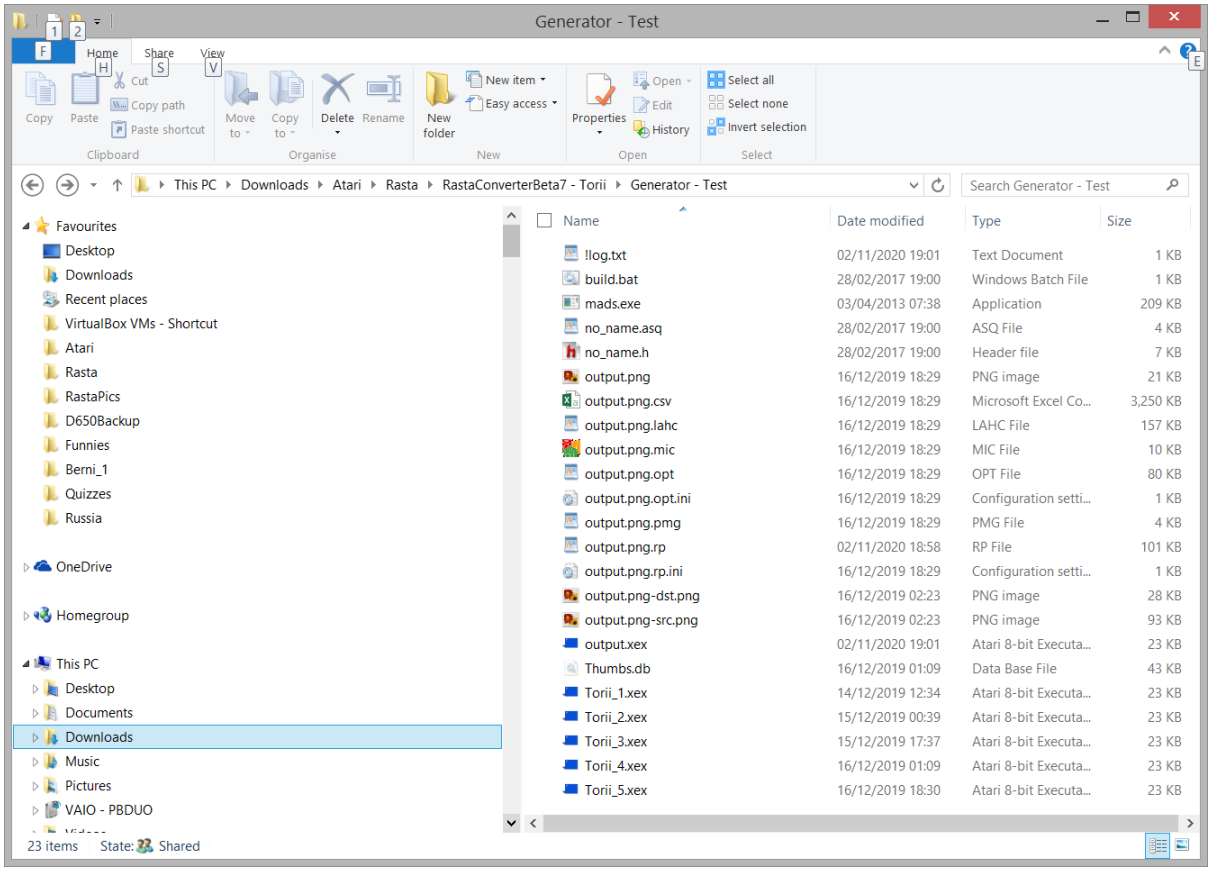
This image in fact has two very obvious artefacts (the two light blue horizontal lines in the clouds)- we are going to repair the lower of the two.

The first task is to use Windows Explorer to navigate to and open the 'Generator' folder within the Rastaconverter main folder. This is where Rastaconverter

places the bitmaps and 6502 assembly language programs used to generate the Atari executable .xex file.



The folder called ‘Generator’ is where the files used to generate the most recently-produced .xex file are located. Here I’ve made several working copies of that folder to use in editing the files, retaining the originals so that we can always revert back and start again if we get in an inextricable tangle with the edits. Here we’re going to be working from the folder called ‘Generator - Test'

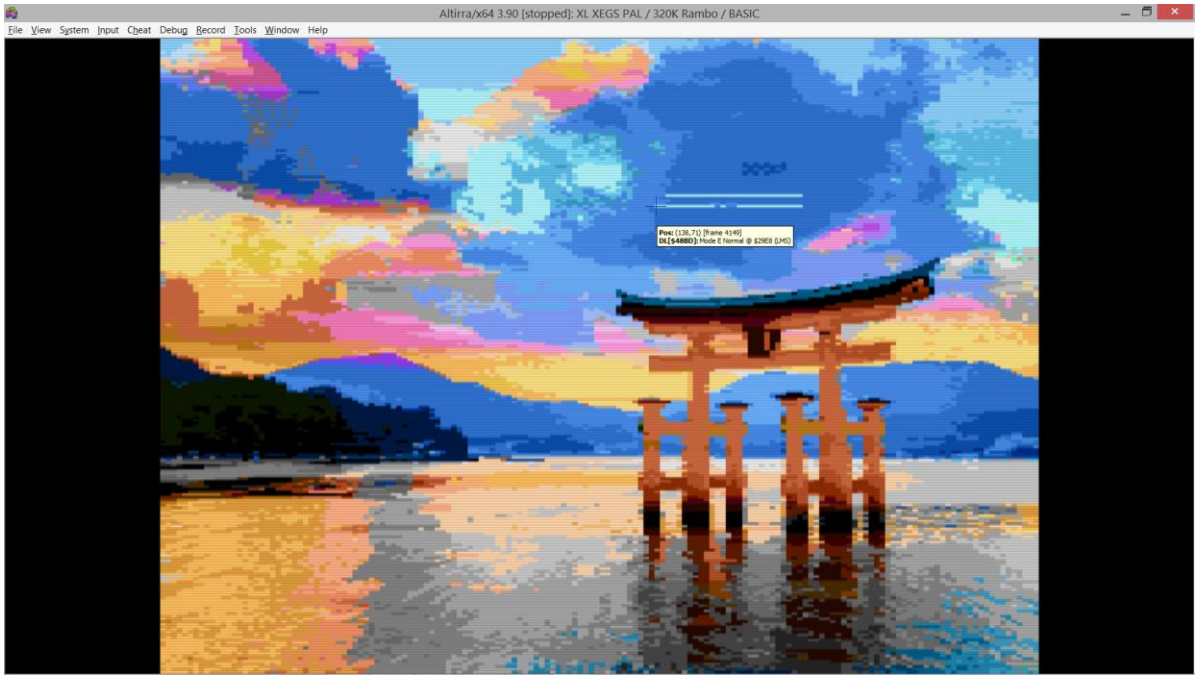


There is a detailed reference list and descriptions of the files you will find in the 'Generator' folder, in alphabetical order, at the end of this document in Appendix A.

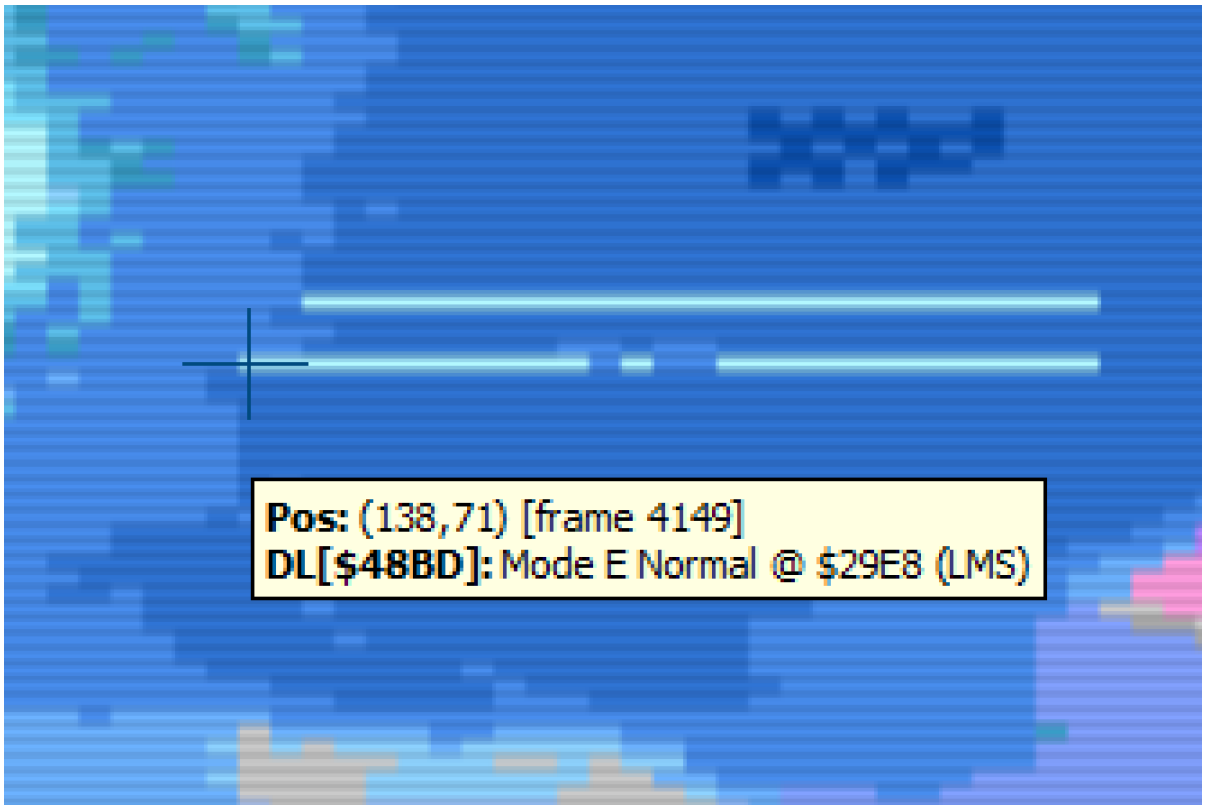
For our immediate purposes, we need to only consider the following:

build.bat	(=the Windows script that generates 'output.xex')
output.png.rp	(=the main kernel assembly language program)
output.xex	(=the Atari executable last generated by Rastaconverter)

To begin, open output.xex in Altirra. This should display the 'faulty' image with artefacts present. Press the [F8] key to stop the emulator. Press the [Alt] key then click on the image- a crosshair cursor and 'tooltip' text box should appear, giving information about the pixel under the cursor:



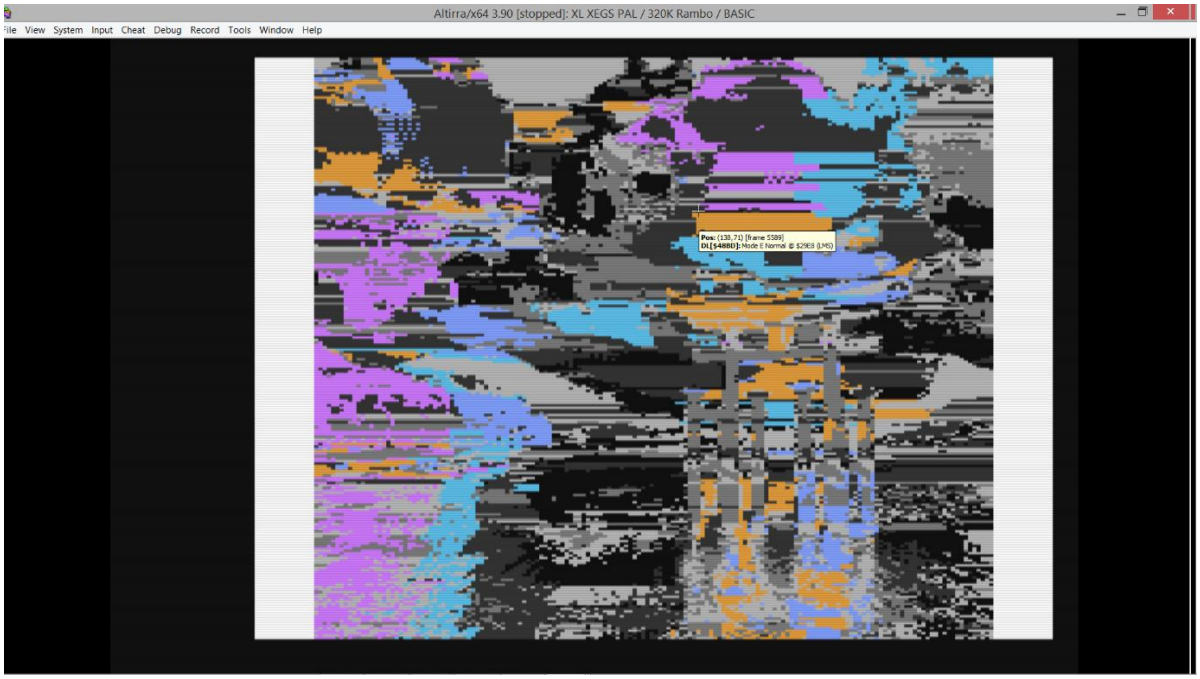
Move the crosshair to lie over the pixel at the extreme left edge of the artefact and make a note of the (horizontal) position and (vertical) scanline, in this case 138 and 71 respectively. It helps for aligning the crosshairs accurately to have scanline emulation switched on in Altirra: ([System] > [Configure System] > [Video] then tick the scanlines box. The intensity of the emulated scanlines can be adjusted with a slider under [View] > [Adjust Screen Effects]):



The implication of these numbers is that the player repositioning causing the problem takes place on scanline 71 and within a short period of time before the video hardware displays the pixel at horizontal position 138. Although the artefact first becomes visible at position 138, it may be that the intended left edge of the repositioned player is a short distance to the left of this, with the interval consisting of blank player pixels, or player pixels overlaid by non-background playfield pixels. We cannot therefore say for sure that the left border of the repositioned player falls exactly within 5 colour clocks (pixels) to the left of position 138. Note that the horizontal position count is a different thing to the horizontal pixel count. In ANTIC graphics Mode E both are counted in units of 1 colour clock, but in an unscrolled standard width Mode E, as we have here, the first (leftmost) pixel 0 on a line is displayed at horizontal position 48 ($30), the centre pixels 79 & 80 at horizontal positions 127 & 128 ($7F & $80) and the last (rightmost) pixel 159 at horizontal position 207 ($CF). It's necessary to add or subtract 48 ($30) to convert between them. The colour clocks for horizontal position are counted from a point well beyond the left border of a normal TV display, long before any on-screen pixels are displayed for that scanline. Similarly, the first full scanline to be displayed is scanline 8. The first 8 scanlines, 0-7, are 'above' the top of a standard TV display. The first line to be displayed from an ANTIC display list is therefore appears on the 9th scanline- scanline 8. It is therefore necessary to subtract 8 from the scanline indicated by Altirra to find the display line, i.e. the zero-indexed scanline count from the top of ANTIC's display area. This is important because we want to edit the kernel assembly language program and, inconveniently, this is labelled in terms of display lines, not scanlines. In this instance, the scanline of interest is 71, and 71-8 is 63, so we can say that the instructions repositioning a player to produce this artefact occur in the segment labelled 'line63' in 'output.png.rp'.

There will be many instructions, together representing exactly 57 machine cycles of the 6502 processor, in 'line63'. There may be several that reposition players. To be sure of the one causing the problem, we need to know quite accurately where in this sequence of instructions the problem instruction must lie. We know that it must occur in a fairly limited time window before the pixel at horizontal position 138 is displayed. We also know that it attempts to position the left edge of a player to appear within a 5 colour-clock window after the 6502's write to the player horizontal position register (HPOSPx) is completed, causing a 'misfire' or failure to trigger for that player. Also, looking at how the Rastaconverter image is composed from playfield and player colours can often give a clear idea of which of the 4 players is involved.

To do this check, press the [F8] key to restart the emulator, them press Ctrl-[F8] to bring up a colour-map of how the image is composed, then press [F8] again to stop the emulator. You can now use Alt-Click again to bring up the crosshair cursor and position it on the same pixel (138,71) we noted previously:



In this image, pixels controlled by playfield colours are represented by shades of grey. Background colour (the COLBAK register) is shown as black, playfield colour 1 (COLOR0) as dark grey, playfield colour 2 (COLOR1) as medium grey, and playfield colour 3 (COLOR2) as light grey. Playfield colour 4 (COLOR3) is not used in ANTIC Mode E, but is used by Rastaconverter for the missile colour and appears as white. The missiles are always positioned down right and left borders of the image to mask changes in background colour made by the kernel as the screen is displayed from top to bottom, which would otherwise show as offputting horizontal bands of colour in the borders. The missiles also mask any player colours straying outside the image's horizontal borders. These could otherwise arise from players whose horizontal borders are positioned beyond the edges of the image. Pixels controlled by the colour of Player 0 (COLPM0), which will be overlain above 'background' pixels in the underlying bitmap, appear orange. Those of Player 1 (COLPM1) are pink, those of Player 2 (COLPM2) are mauve and those of Player 3 (COLPM3) are light blue. In summary:

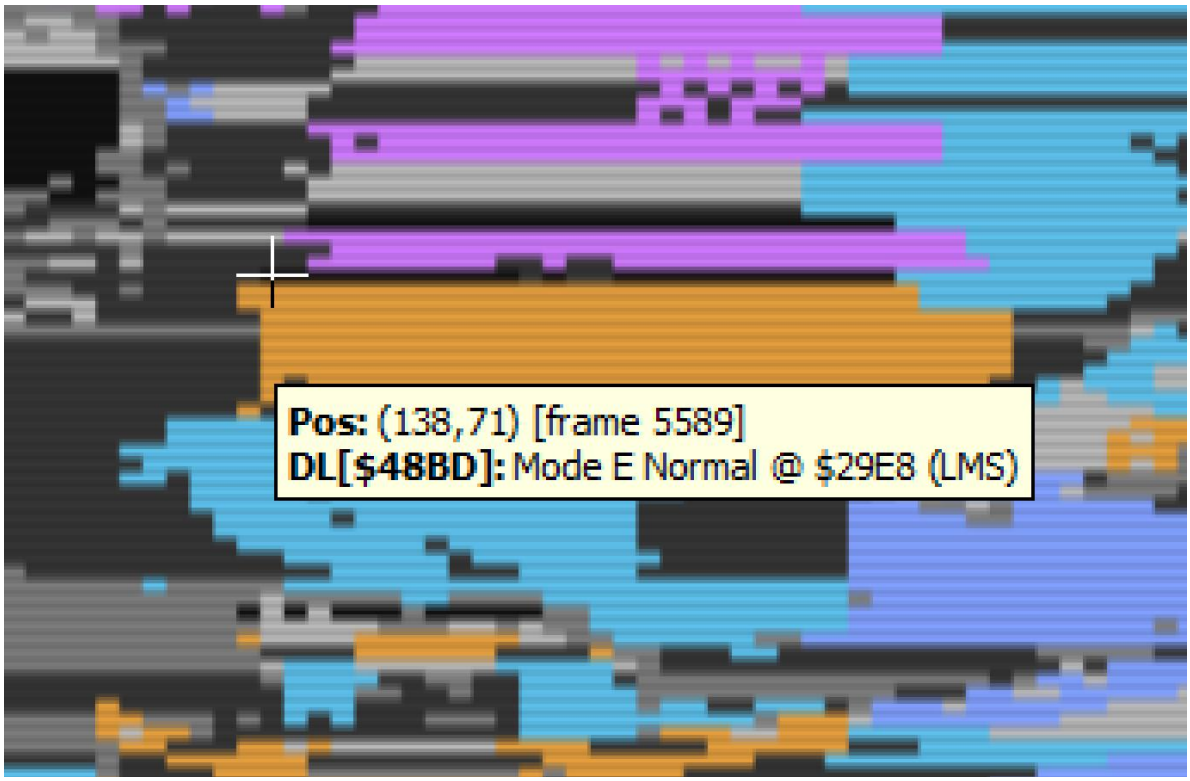
COLBAK	black	
COLOR0	dark grey	
COLOR1	medium grey	
COLOR2	light grey	
COLOR3	white	(missiles)
COLPM0	orange	
COLPM1	pink	
COLPM2	mauve	
COLPM3	light blue	

The colour-codes for the player colour registers can be recalled by the mnemonic **Orange Pi**gs **M**o**ve Lightb**ulbs

Examining this image, you can see how the kernel is shifting the horizontal positions of the players left and right as the image is composed from top to

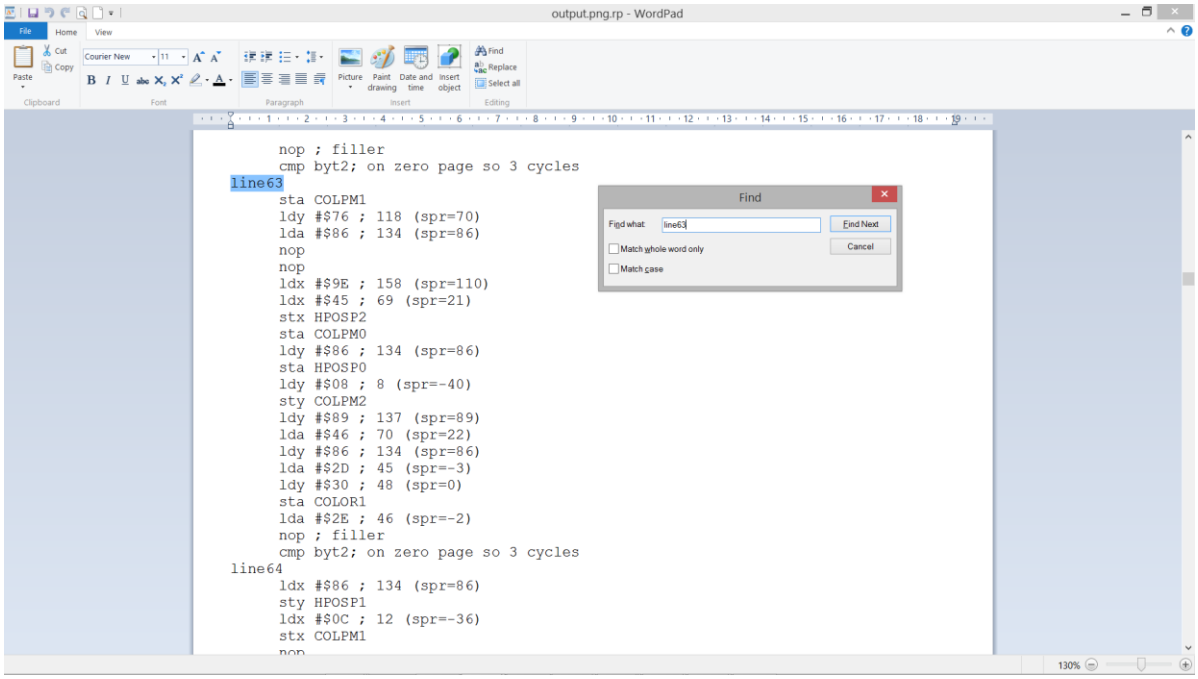
bottom to make up the final image. Player 3 (light blue) for example starts top right and ends up bottom left.

Examining the area to the right of position 137, you can also see the interrupted line of background-coloured pixels (here shown as black) which should be overlain by a player colour but are 'showing through' to produce the artefact:



We know that a player has failed to trigger on scanline 71, at or just to the left of position 138. There are actually a couple of players that appear to be being used in this area- Player 0 (orange) and Player 1 (pink). At this initial stage, of inspection, either could be involved. However, unless the failed repositioning is an attempted 'reuse' of a player already displayed in the left half of scanline 71, the player in question should not appear on scanline 71 at all. Inspecting scanline 71 closely, using the cursor to help with alignment, we can see, interestingly, that in this unusual instance both Player 0 and Player 1 are used earlier on scanline 71, so this does seem likely to be one of the rare instances where the artefact is due to a failed attempt at player reuse within a scanline. However, without directly inspecting the kernel assembly program, we remain unsure at this stage which player is involved except that it's almost certainly Player 0 or Player 1.

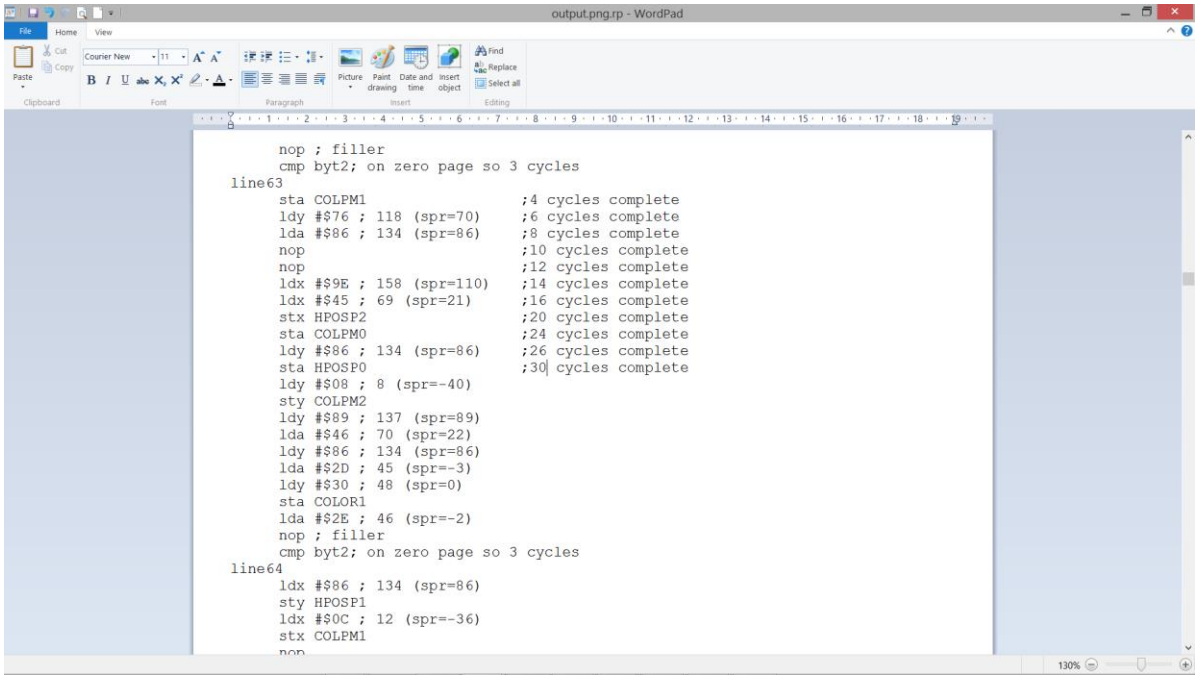
So now we open up output.png.rp in a text editor and find the sequence of 6502 assembly instructions following the label 'line63':



From a quick inspection of this there are two ‘sta HPOSPx’ instructions, one for player 2 and one for player 0. Now we’re already fairly certain that Player 0 is the culprit. However, if we want to make absolutely sure before doing any editing of the kernel, we can indulge in a little cycle-counting.

The 6502 instructions used in the kernel by Rastaconverter represent 2,3 or 4 machine cycles of the 6502 processor. The only one taking 3 cycles is the ‘cmp byt2’ found at the end of each 57-cycle line segment (byt2 is defined as a zero-page address, so ‘cmp’ takes 3 cycles instead of 4 to execute). Rastaconverter only uses the ‘cmp byt2’ instruction at the end of each scanline to bring the cycle count up to the odd (i.e. not even) number of 57. All the other instructions take either 2 cycles (nop; lda #$xx; ldx #$xx; ldy #$xx;) or 4 cycles (sta, stx or sty some-graphics-register) Note that although Rastaconverter never does so, there’s nothing to stop you from moving the ‘cmp byt2’ instruction to earlier in a scanline or inserting 2, 4 or 6 etc. ‘cmp byt2’ instructions in order to get higher temporal resolution (instructions completing on odd as well as even cycle count boundaries within the scanline) as long as they replace instructions totalling an equal number of cycles, such that the total cycle count per scanline is kept strictly to 57.

Anyway, if we start by labelling up the cycle counts on which the HPOSPx instructions finish:



Now, from inspecting the code we can see that the accumulator is loaded with $86 (or 134) (lda #$86 in line 3) before this value is eventually stored in HPOSP0 in line 11 (sta HPOSP0). It's looking increasingly certain that this 'sta HPOSP0' instruction is the culprit, since 134 ($86) is quite close to the left of horizontal position 138 ($88) where the artefact appears. But to absolutely clinch it without experimentation, we need to prove that this write to HPOSP0 completes within that critical 5 colour-clock window before position 134 ($86), causing a 'misfire' or failure to trigger.

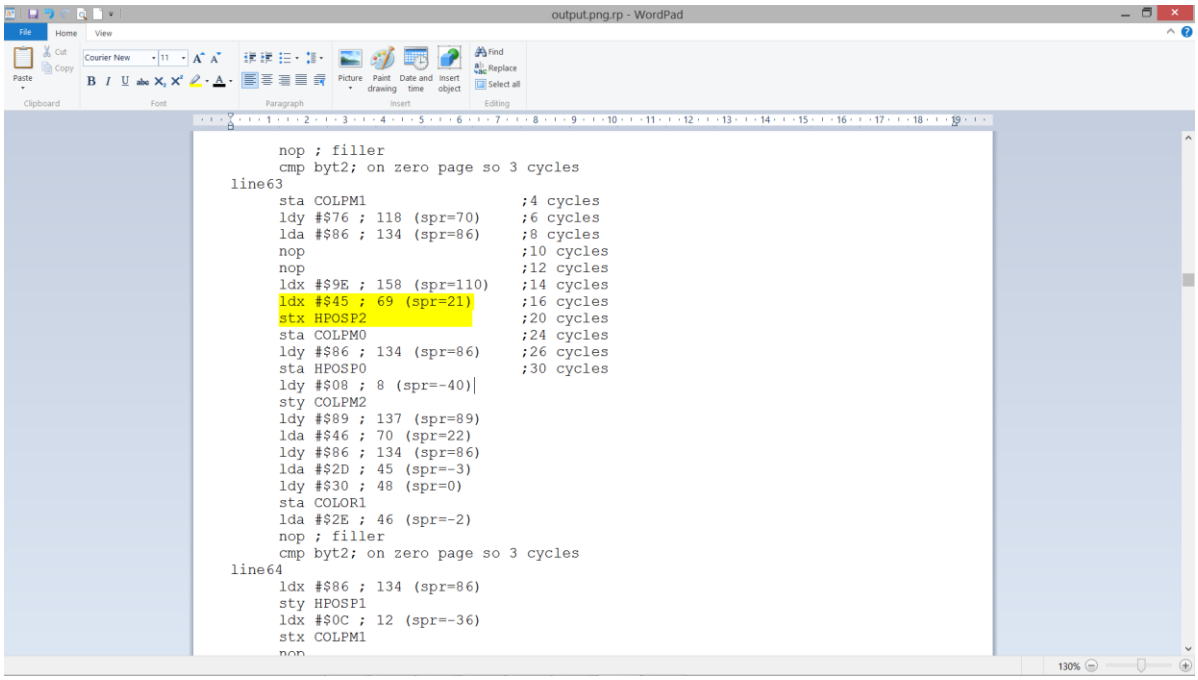
Fortunately there is a fixed correspondence between the machine cycle count in the kernel and the colour clock count within a scanline. On a basic level, one machine cycle takes exactly 2 colour clocks, but the situation is complicated by the CPU being repeatedly 'frozen' for a cycle at irregular albeit entirely predictable points in the scanline while ANTIC accesses memory for its own purposes. This direct memory access (DMA) by ANTIC is sometimes called 'cycle stealing' and explains why the CPU on an Atari appears to run up to 36% faster when the screen display is turned off by blocking ANTIC's DMA. Further cycles (9 in the case of ANTIC graphics Mode E) are lost each scanline as ANTIC freezes the CPU to perform memory refresh of the Atari's dynamic RAM. The upshot of all this complexity is that it's best to work from a pre-prepared table correlating kernel machine cycles to screen horizontal positions rather than calculate it from scratch:

Kernel machine cycle count	Screen horizontal position (colour clocks)	Screen horizontal position (hex)	Display pixel count (0-159)
1	219	$DB	-
2	221	$DD	-
3	223	$E1	-
4	225	$E3	-
5	15	$0F	-
6	17	$11	-
7	19	$13	-
8	21	$15	-
9	23	$17	-
10	25	$19	-
11	27	$1B	-
12	29	$1D	-
13	31	$1F	-
14	33	$21	-
15	35	$23	-
16	37	$25	-
17	41	$29	-
18	45	$2D	-
19	53	$35	5
20	61	$3D	13
21	69	$45	21
22	77	$4D	29
23	85	$55	37
24	93	$5D	45
25	101	$65	53
26	109	$6D	61
27	117	$75	69
28	121	$79	73
29	125	$7D	77
30	129	$81	81
31	133	$85	85
32	137	$89	89
33	141	$8D	93
34	145	$91	97
35	149	$95	101
36	153	$99	105
37	157	$9D	109
38	161	$A1	113
39	165	$A5	117
40	169	$A9	121
41	173	$AD	125
42	177	$B1	129
43	181	$B5	133
44	185	$B9	137
45	189	$BD	141
46	193	$C1	145
47	197	$C5	149
48	199	$C7	151
49	201	$C9	153
50	203	$CB	155
51	205	$CD	157
52	207	$CF	159
53	209	$D1	-
54	211	$D3	-
55	213	$D5	-
56	215	$D7	-
57	217	$D9	-

Note that this table defines the completion of (1-indexed) kernel machine cycles correlated to the simultaneous completion of (zero-indexed) horizontal position counts in colour clocks or (zero-indexed) display of pixels, such that for example rasta kernel machine cycle 50 completes simultaneously with the end of horizontal position (colour clock count) 203 and the display of pixel number 155. The next horizontal position and pixel to be displayed would be 204 and 156 respectively.

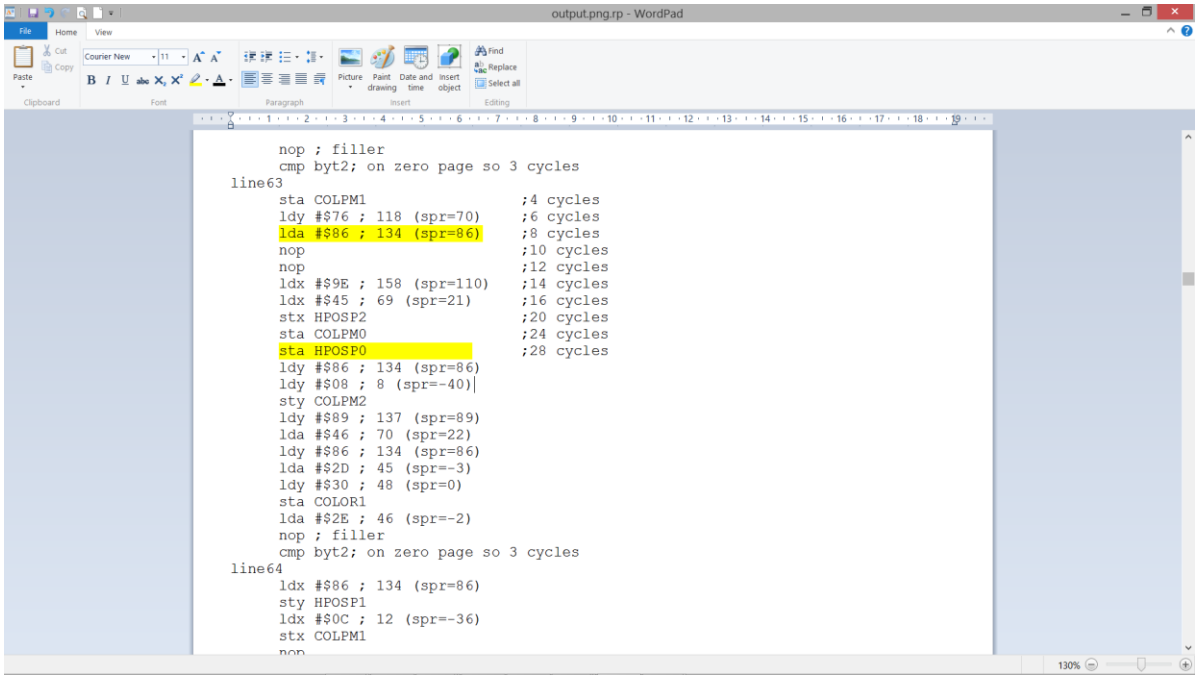
Returning to our kernel program, we can see that our suspect 'sta HPOSP0' instruction completes at the end of cycle 30. This corresponds to the end of colour clock 129 ($81). The critical 5-colour-clock window following that, during which triggering of Player 0 is suspended, would be colour clocks 130-134 ($82-$86). 134 ($86)was the intended new horizontal position of Player 0, which falls, by just 1 colour clock, within the critical window. Therefore this attempted repositioning with reuse of Player 0 fails, redisplay of Player 0 is not triggered at position 134 ($86) on this scanline and the artefact appears.

As a sanity check, applying the same logic to the 'stx HPOSP2' instruction earlier in line 63 of the kernel:



This completes at the end of cycle 20, which correlates to the end of colour clock 61 ($3D) and a critical window of colour clocks 62-66 ($3E-$42). The intended new position of Player 2 is 69 ($45), which falls outside the critical window, so the repositioning succeeds.

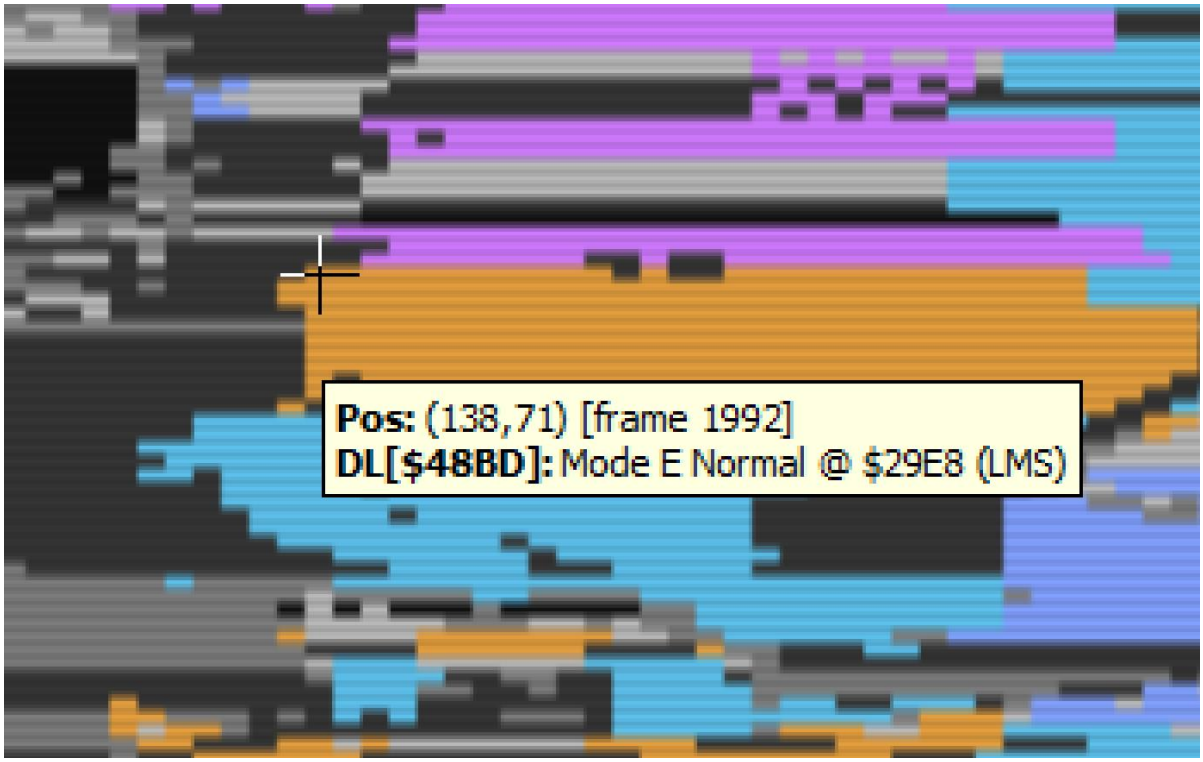
Now we can turn our mind to fixing the problem. Inspection of the kernel rapidly shows that the 'ldy #$86' immediately preceding our 'sta HPOSP0' instruction can be swapped with it, moving our 'sta HPOSP0' back 2 machine cycles to complete at the end of colour clock 121 ($79), with a 'critical window' of colour-clocks 122-126 ($7A-$7E), well before the intended new player position of 134 ($86).



Although moving the new position of the player right by one colour clock to 135 ($87) by editing the 'lda #$86' instruction in line 3 would also move its left border beyond the 'critical window' and avoid the 'misfire', altering player positions like this usually leads to a cascade of knock-on errors in the image that must then be corrected and is therefore best avoided if at all possible.

We can now save our new kernel program and test it by double-clicking on 'build.bat' to generate a new 'output.xex' file. If all goes well, 'output.png' will be displayed and the new 'output.xex' will be launched in your emulator, and the artefact will have vanished:

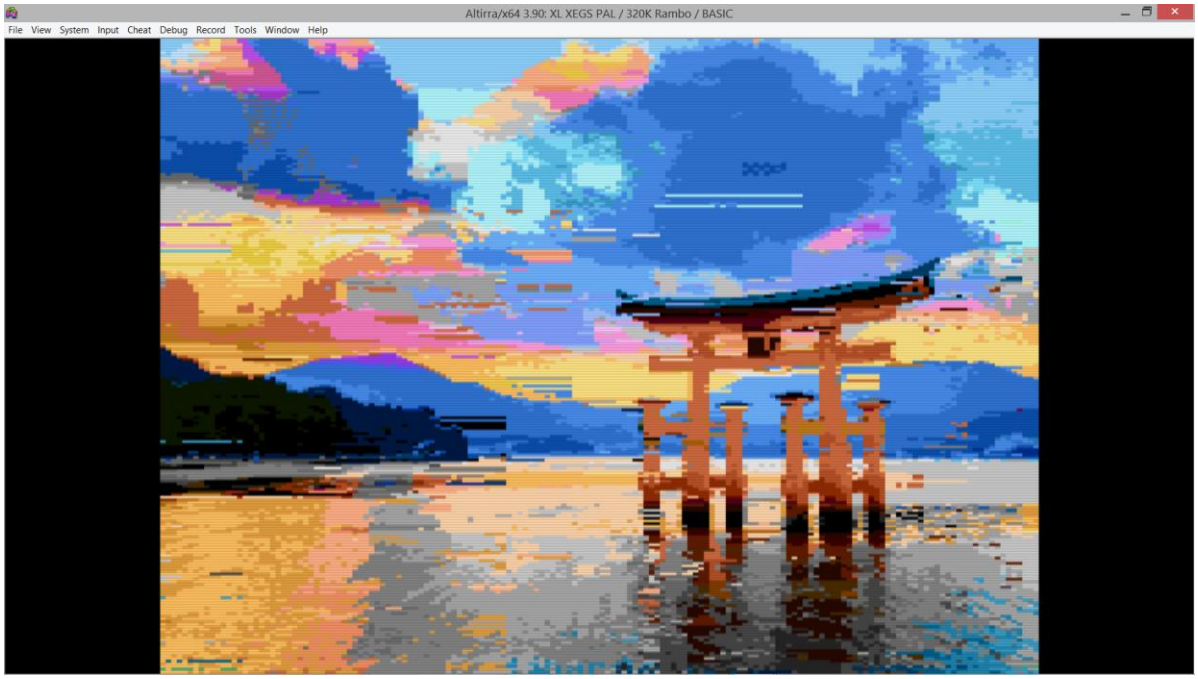
As a sanity check, inspecting the corrected image with Altirra's colour-mapping function shows the previously artefactual background-coloured pixels (black) now overlain by Player 0 (orange):



Troubleshooting

If something has gone wrong, an error message will appear saying 'unable to locate output.xex' or words to that effect. This means the assembler has been unable to successfully interpret your edited kernel program. This is usually for one of four simple reasons: putting in a comment without a leading semi-colon; misspelling one of the colour registers or HPOSPx registers; forgetting to put a $ in a 'lda/ldx/ldy #$xx' instruction; misspelling an assembly instruction. Examining the '!log.txt' text file should point you in the right direction.

If output.xex is successfully produced but the colours below the original artefact appear 'scrambled' you have introduced a timing error by having a line in your kernel that does not total 57 machine cycles:



You'll need to go back and carefully cycle-count the corresponding line in the kernel from where the scrambling starts.

Further reading:

For more information on the operation of the Atari 8-bit computers' video hardware, including ANTIC, GTIA, graphics modes, display lists, colour clocks and cycle counting see:

The Atari Home Computer System Technical Reference Notes © Atari Inc. 1982

The Altirra Hardware Manual © Avery Lee 2019

De Re Atari – A Guide To Effective Programming © Atari Inc. 1982

Appendix A- Files in the Generator Folder

!log.txt a log file produced by the assembler of any errors occurring during generation of the .xex file

build.bat the Windows script that, when run, evokes the assembler 'mads.exe' to build the Atari executable file output.xex file using the program and data files in this folder. The script then opens 'output.png' in the default .png viewer and opens 'output.xex' in the default Atari 8-bit emulator associated with the .xex extension

mads.exe the cross-assember that when evoked by build.bat with relevant parameters reads the assembly language and data files in this folder and outputs to this folder an Atari executable .xex file called 'output.xex'

no_name.asq the assembly language file for the main screen kernel

no_name.h the header file for 'no-name.asq' containing named register equates for the Atari, such as HPOSPx for the player horizontal position registers, COLORx and COLBAK for playfield colour registers, COLPMx for the player colour registers

output.png the (typically) 160x240 resolution image in (typically) 320x240 pixel .png format that Rastaconverter expects to be displayed in (typically) 160x240 pixels by running the generated output.xex executable file on an Atari 8-bit computer

output.png.csv a comma-separated-value database of information relating to the elapsed time and scoring of solutions being considered by Rastaconverter when last stopped- for internal use by the Rastaconverter algorithm in its evaluations

output.png.lahc text file with the first 2 lines containing (1) the number of simultaneous solutions specified for consideration by Rastaconverter when last running and (2) the total number of solutions run through when Rastaconverter was last stopped. The remaining rows contain internal data for use by the Rastaconverter algorithm in its evaluations

output.png.mic data file containing the bitmap generated by Rastaconverter representing the (typically) 160x240 playfield pixels in the final .xex file generated by build.bat. This bitmap data can if needed be loaded, edited and resaved by a suitable graphics application, such as Grph2fnt

output.png.opt text file containing a version of the assembly language program used to compile the machine-code main kernel program. A functionally equivalent program is to be found in 'output.png.rp' but here it is optimised for easy reading by removing from the raw Rastaconverter output all redundant CPU instructions that don't ultimately affect a graphics register, e.g. an LDX #$00 that is superseded by an LDX #$04 without any intervening STX instruction. This file is for convenience only- it is not used in the building of the .xex, which uses the functionally equivalent but less intelligible 'output.png.rp'

output.png.opt.ini text file containing a version of the initiation header of the assembly language program used to compile the machine-code kernel program. A functionally equivalent program is to be found in 'output.png.rp.ini' but like

'output.png.opt' it is optimised for easy reading. This file is for convenience only- it is not used in the building of the .xex, which uses the functionally equivalent 'output.png.rp.ini'. In practice, since Rastaconverter never generates redundant instructions in the header the two files are identical.

output.png.pmg a text file containing the player-missile data for the .xex, stored in assemby language format. Missile data is not actually used (the kernel 'locks up' the GTIA missile graphics register to continually display $FF) so the file starts by reserving 256 of unused memory to represent the missile data area. There follow 4 x 256 bytes of player data organised in tables of 16x16 hexadecimal values. The first and last 8 bytes of data for any player are never used by the Atari display hardware, so these are always 00. The eighth byte of data is for display on the eighth scanline of the frame, which corresponds to line 0 of the display list (display lists always start on the 9th scanline- scanline 8- of the frame). Therefore, to find which byte in the tables corresponds to a given line label in the rasta kernel assembly program ('output.png.rp')- which are defined by display list lines- it's necessary to add 8. e.g. the player data being displayed while 'line54' of the rasta kernel is running can be found in the 62nd values of each of the 4 player data tables.§ If you need to edit player data, you can manually do so in this file and resave it.

§ With hindsight, life would have been simpler if Rastaconverter had labelled the kernel assembly program with scanline numbers instead of display line numbers!

output.png.rp text file containing the assembly language program used to compile the main kernel machine-code program of the .xex file. After a few dummy instructions to synchronise the CPU with the video display hardware, the kernel consists of 6502 instructions organised into labelled 'lines' of exactly 57 machine cycles each, this representing the number of cycles the 6502 completes during each and every horizontal scanline of the display list defined in 'no_name.asq'.* To determine which part of the kernel is running when a given scanline is being displayed, it is necessay to subtract 8 from the scanline number and then search 'output.png.rp' for the corresponding line label. The line labels reference display list lines not scanlines (display lists always start on the 9th scanline- scanline 8). e.g. the 57 machine cycles running during scanline 54 are defined by assembly instructions following the label 'line46' in 'output.png.rp' §

* This display list consists of 240 lines of ANTIC mode E, each with an LMS bit set that reloads ANTIC's playfield memory fetch counter, finally ending with a JVB. Triggering a JVB beyond the 248th scanline (240th display list line) risks the display flickering- with alternate frames being blank- but the kernel avoids this by directly reloading the display list instruction counter to point to the start of the display list during vertical blank.

output.png.ini text file containing the initialisation header of the assembly language program used to compile the machine-code kernel program. This initialisation code runs each frame at the end of vertical blank to set up the colour registers and player horizontal positions ready for the 1st display line, then zeros the CPU data registers and waits for the 1st display line (display list line 0, which is the 9th scanline- scanline 8) before control passes to the main kernel program in 'output.png.rp'. The initialisation header also contains an equate indicating the vertical resolution in pixels of the image to be displayed,

which is typically 240 but may be fewer. This information is used when assembling the .xex file to organise the storage of bitmap data in memory. ‘output.png.rp’ will also consist of this number of labelled 57-machine-cycle lines. e.g. for a 192 pixel-high image, the last 57-cycle segment of ‘output.png.rp’ will be labelled ‘line191’

output.png-dst.png the (typically) (typically) 160x240 resolution image in (typically) 320x240 pixel .png format that Rastaconverter is trying to reproduce as closely as possible. It represents ‘output.png-src.png’ with any dither and/or other image-processing selected in Rastaconverter applied, then matched according to the chosen parameters to the selected 120/128-colour Atari palette

output.png-src.png the (typically) 160x240 resolution image in (typically) 320x240 pixel .png format achieved by resizing the selected source image according to the parameters chosen in Rastaconverter

output.xex the Atari executable file generated when ‘build.bat’ invokes the assembler, using the program and data files in this directory.

Appendix B – Named Graphics Registers Used in the Kernel and Corresponding Colours in Altirra's Colour-mapping Function

COLBAK	black	playfield background colour	(bit pattern 00)
COLOR0	dark grey	playfield colour 1	(bit pattern 01)
COLOR1	medium grey	playfield colour 2	(bit pattern 10)
COLOR2	light grey	playfield colour 3	(bit pattern 11)
COLOR3	white	playfield colour 4	(missiles)
COLPM0	orange	player 0 colour	
COLPM1	pink	player 1 colour	
COLPM2	mauve	player 2 colour	
COLPM3	light blue	player 3 colour	
HPOSP0	-	horizontal position player 0	
HPOSP1	-	horizontal position player 1	
HPOSP2	-	horizontal position player 2	
HPOSP3	-	horizontal position player 3	

Appendix C – Correspondence of Kernel Machine Cycles with Horizontal Position (colour clock count) and Pixel Count

Kernel machine cycle count	Screen horizontal position (colour clocks)	Screen horizontal position (hex)	Display pixel count (0-159)
1	219	$DB	-
2	221	$DD	-
3	223	$E1	-
4	225	$E3	-
5	15	$0F	-
6	17	$11	-
7	19	$13	-
8	21	$15	-
9	23	$17	-
10	25	$19	-
11	27	$1B	-
12	29	$1D	-
13	31	$1F	-
14	33	$21	-
15	35	$23	-
16	37	$25	-
17	41	$29	-
18	45	$2D	-
19	53	$35	5
20	61	$3D	13
21	69	$45	21
22	77	$4D	29
23	85	$55	37
24	93	$5D	45
25	101	$65	53
26	109	$6D	61
27	117	$75	69
28	121	$79	73
29	125	$7D	77
30	129	$81	81
31	133	$85	85
32	137	$89	89
33	141	$8D	93
34	145	$91	97
35	149	$95	101
36	153	$99	105
37	157	$9D	109
38	161	$A1	113
39	165	$A5	117
40	169	$A9	121
41	173	$AD	125
42	177	$B1	129
43	181	$B5	133
44	185	$B9	137
45	189	$BD	141
46	193	$C1	145
47	197	$C5	149
48	199	$C7	151
49	201	$C9	153
50	203	$CB	155
51	205	$CD	157
52	207	$CF	159
53	209	$D1	-
54	211	$D3	-
55	213	$D5	-
56	215	$D7	-
57	217	$D9	-