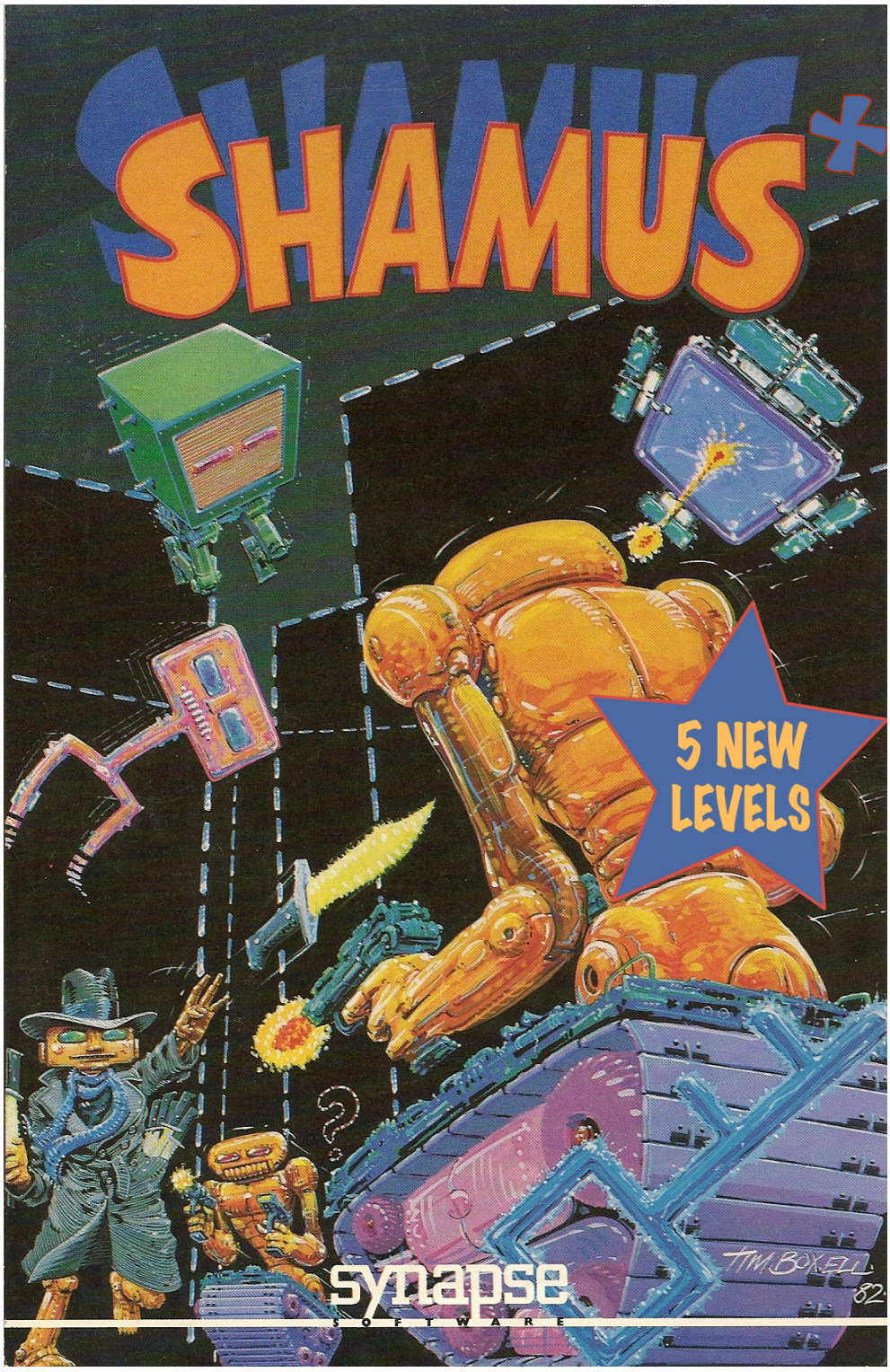
New Mazes for Shamus



slx, October 2017

This project is dedicated to Cathryn Mataga who created Shamus and Ihor Wolosenko who created Synapse Software, home of some of the finest period games and game art for Atari 8-bit computers.

Copyright Issues

Shamus was originally written by William Mataga and published by Synapse Software in 1982. Copyright is assumed to be with Cathryn Mataga at the time of this patch. The C64 levels are believed to have been developed by Jack L. Thornton who is named on the C64 version title page. I have been unable to contact either of them and hope that they approve of this patch.

Instructions

Use [OPTION] to select the starting Maze. "Tournament" will play through all mazes in sequence.[1]

When you die in tournament mode the next game will start at the beginning of the maze you died in. The maze selection will still read "Tournament". In order to start a tournament from scratch, use [OPTION] to rotate through all the mazes until you are in "Tournament" again.[2]

Use [SPACE] to pause and the fire button to end the pause.

[1] except the "Original C64" which is almost identical to the original Atari maze.

[2] while this is owed to the maze selection logic, consider it a "continue" feature ;-)

History

How it all began

When I got an Atari 800 for Christmas in 1982 I didn't get any software for it but BASIC and DOS 2.0S. I can't blame my parents as they had spent the equivalent of a nice used car on the 800/810 combo. Gaming had not been central to my lobbying for a computer despite *Star Raiders* being one of the core reasons not to have anything but an Atari. (My father always proposed other computers and let me argue for the Atari, maybe he considered that a kind of rhetoric training. The availability of Star Raiders was not something I could use in favor of the Atari.)

I sequestered the kitchen TV and started working through the BASIC book on Christmas morning[3]. Together with a friend who had received a 400/410 with *Pac-Man* instead of *Star Raiders* I had to wait for a full two days until we could make the pilgrimage to a small computer shop that carried Atari stuff. Apart from the core Atari lineup, there wasn't a lot of software available, so *Star Raiders* – dearly paid for with more than three months' worth of allowances – remained my only game for the time. I continued to fill my four floppies[4] with BASIC programs and typed up listings from Compute! which – probably considered useful English literature – was soon arriving by subscription paid by my parents.

Shamus!

On an Easter holiday trip to Germany the following year I managed to divert my parents to the small town of Holzkirchen southeast of Munich, home of one of the larger Atari dealers advertising in the German magazines as well as Hofacker, German twin of Elcomp publishing. I arrived with some saved money and great expectations as software was more readily available and a bit cheaper in Germany. The shop was much smaller than the Atari paradise I had envisioned when reading their ads and most of the programs on sale were not what I was looking for. I had read a favorable review of Shamus in Compute! # 33 and knew Synapse as a quality company from playing *Nautilus* with my friend, so I hurriedly bought a

[3] presents are given on Christmas Eve here

[4] A 10-pack of floppies cost as much as Star Raiders and was therefore out of budget. I had only been able to afford three floppies when I bought Star Raiders, the fourth was the blank that came with my 810.

copy of *Shamus* and *The Sands of Egypt* while my family was waiting in the car.

With three games in my library I played each of them a lot. I became quite adept at Shamus, drew a map of the first two levels and put a lot of effort into winning the game (I remember restarting when I had lost a life before reaching the green level). At the height of my prowess I reached 127K points clearing the complete maze twice in a row.[5] With pirated games eventually finding their way to Austria, there were more games to play later that year but to this day I consider Shamus my favorite Atari action game.

A good friend of mine had gone the Commodore route from VIC-20 to C64 and showed me a pirated copy of Shamus. While I was rather disappointed with the colors, the lack of "glow" from the walls and the much more cartoonish looks of the Shamus, I noticed that Commodore players got five mazes to choose from.

30 years later

I did not become a software engineer but a pilot and with Internet and eBay some of the tempting stuff seen in Compute! and Antic ads in the 1980s is now within reach (if not reason), allowing me to amass a small (but far from complete) collection of Atari stuff. Lots of new gadgets promise easier use of Ataris than ever. ANTIC and other Atari and retro-computing podcasts make good listening on the way to the airport - although the rate of publication of ANTIC would actually require a move away from work to keep up. An Atari 130XE adorns the desk of one of my teenage sons and is used as an alarm clock. Both of us have submitted a 10-liner, a new challenge is required. While I do have ideas for whole games, I know that actually programming one would likely require more time than I can spare. While my reflexes are insufficient to play Shamus like I used to, even with the "made for Shamus" TAC-II hooked up to a Stelladaptor, I still like it and somehow the idea of porting the "extra" mazes on the C64 to the Atari takes hold.

I don't know anything about the inner workings of SID, VIC & Co., so I decide to start on the Atari side. As there is no way for 128 screens of maze to fit into an early Atari cart, the maze must be encoded. I plan to look for that "map code" and then see if I can find the same code on the C64. Once I find it I only need a little program to copy it where the Atari can use it. Should be simple.

[5] Then unknown to me *Softside* lists high scores almost twice as high.

Cheating with Altirra

Altirra's Cheater function is of great help as it takes only a few tries to find out that the current level is stored in $208 and $235[6] and setting a breakpoint on reads to that address reveals tables relevant to the map.

So I fire up the x64 emulator, get into the debugger and find ... nothing. That means I will have to dig deeper and find out how the mazes are coded in both versions and how to translate them.

Digging C64

While countless Atari columns, articles and books have left me with what I believe to be a fairly good understanding of how the Atari works (even if that doesn't translate into assembly programming mastery) a glance at C64 system descriptions convinces me that I feel too old to learn another 1980's architecture. Having had a classic 800 even Atari bank switching feels foreign to me, the more complicated C64 scheme with "write-through" ROM, etc. is clearly intimidating.

I therefore decide not to work from the display backwards but follow any traces the map selection mechanism in the C64 menu might leave. C64Debugger turns out to be just the right tool for the task. A wet dream of every 80's hacker showing a live view of running code and color coded blinking RAM displays indicating reads and writes, it looks as psychedelic as the C64 memory layouts are confusing. It soon reveals that changing the level changes just one byte but on starting the game a 384 byte long block is copied to a fixed address. That must be the C64 map!

Flipping bits

The task ahead is a bit tedious. I start to change the map bytes on both systems bit by bit and observe what that does to the maze layout. Changes only take effect when walking into a room, requiring a lot of jogging back and forth between rooms. Switching off collision detection saves the waiting Shamus from certain death. Different monitor/debugger command sets on x64 and Altirra as well as frequent "bit jogging" hopefully postpone certain death for my brain cells as soon I don't need tables or jotting paper to convert from binary to hex and back.

Walls, walls, walls

The basic method of coding "ordinary" maze rooms or "chambers" is quite similar on both systems. The grid of possible interior walls looks like a #, with each vertical and horizontal line or wall consisting of three parts. Single bits of the first map byte switch "on" and "off" horizontal interior "walls" and

[6] going to room 2 I look for addresses containing 2, then move to room 3, check which of them now read 3, and after one more room it's down to the 2 addresses I am looking for.

the second byte controls vertical walls. While the principle is the same on both systems, the order and position of bits within the map bytes is different. I find that thinking about an elegant way to invert a bit sequence in 6502 Assembler (maybe with `ASL`s and `ADC`s?) is very helpful in falling asleep almost immediately.

Other bits are not that straightforward. Turning on two bits will make any C64 room a "pod room" with vertically moving barriers with just a small slit through which you have to hit the target in the middle of the room to advance. No such bits are found on the Atari and sleuthing method number two doesn't work either. A suspicion that combinations of bits over several map bytes might be significant raise the frightening prospect of having to go through 65535 combinations in order to understand the maze map, so I set this aside for the time.

A maze of passages – but not alike

Apart from "ordinary" and "pod" rooms there are "corridors", tighter passages between the chambers I have successfully decoded. On the Atari setting the high bit of map byte one turns that room into a corridor room and the next three bits select the type of corridor. The C64 coding is a bit less obvious and a bit more flexible. After designing some "crazy" corridors with partly open walls I find out that corridor walls are defined in the same way as ordinary walls and a cleared low bit on the second map byte causes the area outside the walls to be "filled in" to form a corridor.

I make a note that this allows for corridor shapes not coded on the Atari, such as a vertical passage. I also note that this would make no sense due to the way that rooms are connected.

Moving through the maze

Leaving a room to the left or right simply decreases or increases the room number. That means room 25 will always be to the left of room 26 and to the right of room 24. Whether those rooms are actually next to each other on the map doesn't matter as the electrocuting walls will prevent any progress outside the maze (unless collision detection is turned off, that is).

Vertical movement is simpler than I expected. On the Atari all corridor rooms have the number of the vertically adjoining room in map byte two. The C64 stores the same information in byte three. Whether you move up or down is irrelevant and the corridor shapes on the Atari are all limited to either an up- or a downward exit. Both exits of a vertical corridor would lead to the same room.

Colored potions

Bytes three and four of the Atari map are used to encode

"objects", i.e. the extra lives, mystery (?) bonuses, keys and keyholes. With only four different objects and only 16 colors used[7], this seems like a rather lavish use of memory. The C64 code manages to squeeze the object information in the two unused lower bits of byte one and the high bit of byte three. As objects are only found in chambers which don't have vertical connections, byte three is used to store key/keyhole color for these rooms. Fixed, non-mapped colors are used for the potions and mystery bonuses.

While I feel that I know have a good knowledge of the map encoding on both machines, there are aspects that I did not manage to track down. I can't seem to find any encoding for the number of enemies per room although that number seems to vary within a certain bandwidth for every room. I decide to accept a forum opinion that this number is procedurally generated. I also don't know how the change from black to blue to green to red levels is effected and for the time assume that it is hard-coded somewhere.

I save the C64 map data which is conveniently located in a contiguous block from $610D to $688C to a "raw" file to feed my converter. While I do not intend to use the "original" level I decide to include it as comparison with the Atari map might be helpful in weeding out errors. A little more detective work leaves me with the addresses of the title and menu screens and the scroll routine.

Action! The next task is to write a "converter" that will turn the 384 bytes of each C64 map into 640 bytes of Atari map. To keep the retro spirit I decide to implement this in Action! on an Atari (emulator) rather than using Python (which I am not fluent in either but which would allow more comfortable code editing and easier debugging).

Transferring the C64 map file to the Atari is much easier than it would have been in the 80s and just a drag and a drop away in the Emulator disk manager. As this is going to be a quick and dirty little program only, I generously declare two arrays which will hold the complete input and output files.

While probably screaming for pointers, it would be a first for me to use them, so I decide to use a little more clumsy array indices in order not to out-code myself and have code that is easier to understand to someone reared with BASIC who never

[7] for some reason which I come to understand only later, colors are stored in the lower nibble and shifted up four bits and ORd with a brightness of $06 before use.

progressed much beyond it. (I actually tried some pointers but failed to find the caret using a German keyboard with Atari800MacX.)

Just a quick hack!

Two loops run through 128 maze rooms five times in a row and a little `IF ... THEN` code will check the C64 map bytes and write out their Atari equivalents. With detailed notes in hand it should be a matter of an evening or two to finish this and move on to some assembly coding for the menu.

A lack of Action! proficiency, a little sloppy (sleepy?) late-evening coding and the discovery of a few unknowns extend that task to more than a month of on-and-off evening (and sometimes even daytime) coding. The Action! compiler will happily accept Syntax that is not what you meant to do but clean enough to address the same variable in a different way. Not only does this result in source code wrecked by rampant writes to nonexistent array elements but it also disarms traps I set to catch missing conversion routines.

I discard the idea of an elegant bit inverting routine using inline assembly code and write an ordinary Action! subroutine that does the job and is easier to understand when looking at the code. Speed and memory are of no concern as the converter is destined for single-use only.

For the corridors I write a function that checks for those combinations of map byte 1 and 2 on the C64 that will result in any of the 7 corridor types used on the Atari and returns the correct Atari code for byte 1. In order to catch if non of the `IF`s is true, it returns $FF when no matching corridor is found. At least that's my intention.

Where are those pod rooms?

Now I remember that I still don't know the Atari coding for the pod rooms. Am I really back to trying out 65535 combinations? While a true Shamus will follow any trail, he won't take the longest one possible unless he runs out of clues elsewhere. Indeed searching for the sequence of numbers of known pod rooms reveals a list of pod rooms and a breakpoint set on access to that list a snippet of code that checks if a room is a pod room whenever a room is entered! I generously decide to dedicate another 128 bytes per level to a list of pod rooms. While no C64 maze has more than 8, it will keep the map data evenly spaced and map addresses will be easier to remember. If someone should ever want to design a maze with all pod rooms, I am prepared.

Comparing the list of pod rooms between the Atari and "original" C64 maps, I find a few differences between them and

decide to include this C64 map as well. (You need to be quite good at Shamus to find the differences without using cheat mode or a debugger, as they're way into the maze.)

ARRAY troubles

A lot of time is lost due to some array indices pointing where they shouldn't through some errors in logical thinking I commit during my "(actually not so) quick and (actually quite) dirty" coding. Sometimes I would love to have used Python and now be able to quickly print out an array to see what went wrong instead of having to use `PrintBE(Outputfile($125))` to check a single cell. To see what's going on/wrong, my converter code is riddled with `Print` commands. While this allows tracking progress, it slows the converter down considerably. But even with Altirra limited to normal Atari speed, compiling is still lightning fast.

Having covered all the known maze encoding after about two weeks of evening coding during my vacation, it's now time to actually use the mazes on the Atari. My converter has delivered them as a 3.200 byte binary file consisting of five mazes of 640 bytes each. This time I use WUDSN rather than anything period and don't regret the choice. Nice Syntax highlighting, easy import of the original Atari game binary code as well as the binary code for the C64 maps and lightning fast assembly are even better than MAC/65 for someone who has not done any serious assembly coding in decades.

Patching fun

The "original" Atari code starts with a routine at $7000 that moves a lot of game code down $4000 bytes, I assume this puts file loaded code where the original boot disk code was located. With WUDSN I simply note the `JMP` address at the end of that routine, set a new `ORG` to the address of that `JMP` and start coding. While I could use this technique for every patch required, I would have to patch the code before relocation, thus patching to different addresses than where the code will eventually run. As I expect this to be confusing during debugging, I decide to patch the code after relocation using `LDA/STA` sequences. That needs a little more room but I can use the same technique for patches required before and during gameplay.

Just like the C64 program, my new maze select routine copies the selected maze data to where the Atari expects it. To be able to revert to the original Atari maze I copy the Atari maze data into a new "slot 0" during game initialization.

DLIST diversion

Altirra's `.dumpdlist` command comes in handy when adding a new line displaying the selected maze on the menu screen. As I

can't extend the original display list without disturbing graphics data immediately following it, I simply copy the display list data into my assembly code, including the jump instruction at the end. This lack of attention is going to haunt me for at least two evenings as I just can't understand why my new menu refuses to show a new line and why the old display list remains in use despite any writes to `DLISTLO/HI`.It's a head-banging moment when after hours of trapping writes to `DLISTLO` and single-stepping through code I realize that I set it up that way by copying the old jump instruction!

new [OPTION]s

Adding a check for `[OPTION]` being pressed is a matter of diverting the original console key checking mechanism with a `JSR` and adding the replaced instruction just before the `RTS`, or at least so I think. While the maze select and display code works quickly, I simply can't figure out how the delay code that requires to release `[SELECT]` again in order to select the next higher level works. I have to write my own delay routine which explains the slightly different behavior of `[SELECT]` and `[OPTION]`.

It's time to select "Original C64" on the Atari and start testing! It works as expected and as I am now confident that I will end up with a working game, I write to Cathryn Mataga to request her blessing for a release. As I know that she continued to use the Shamus franchise on modern platforms and released the last game only a few years ago, I'd really feel better not to publish this against her wishes.

Head to Head

My playtesting setup is as follows: the Action! converter utility is running on Atari800MacX. I read the disk image with Omnivore (which is also good for a quick check of map data) and use Omnivore to save the Atari map file to my WUDSN Eclipse workspace. From there it is compiled into an Atari executable that I load with Altirra (running under WINE). An x64 window next to Altirra allows immediate comparison of original and cloned levels. It is even possible to move Shamus on both emulators with the same joystick, but as the C64 Shamus runs much slower and does not speed up on higher levels they will not run in sync.

For testing I simply move both Shamus(es?) through the maze from left to right and check for differences in rooms and objects.

Where's the door?

I soon literally run into another obstacle. I have not given any thought to how the placement of the "door" in keyhole rooms is decided. Depending on the maze layout either the left or

right exit of a chamber with a keyhole remains closed until the correct key is presented. A little comparison between known door rooms shows this to be encoded in bit 4 of the color byte on the Atari and in bits 6/7 of the color byte on the C64. A simple `AND $10` to the Atari color byte should shift the door if required. Only it doesn't work. Door rooms inexplicably show completely different color data. After some hair pulling and head scratching I realize the color variable my converter uses for the `AND` is an index to a 16-byte array that contains an Atari color value for every C64 color. Adding $10 to the index has it fetch meaningless data from somewhere behind that array.

While troubleshooting I somehow manage to delete an `ELSEIF` without breaking compilation, resulting in another bout of debugging. Somehow I envy the guys who code complete games in an afternoon or at least during a demo party.

256 vs. 16 Further travels through the maze show that object color needs a bit more tweaking. With four out of 16 C64 colors (black, white, light grey, dark grey) represented by the same hue (chroma) on the Atari and only luma different, using a fixed luma of $06 will not work. As it would be quite shameful not to recreate the C64 colors on a machine that prides itself on having 16 times as many, I need to patch the color code (which I luckily located while working out maze coding) to allow encoding of both luma and chroma. Again a `JSR` is patched into the original code.

As I don't want to redo the code that checks "door bit" #4, I decide to keep the "inverted" color setup, stuff luma in the high nibble and reverse them at runtime. (As luma does not use the lowest bit, this leaves the door position encoding undisturbed.) For some reason original Atari maze objects look quite dark now and I realize that I need to change the original Atari map data accordingly while it is copied during game initialization, otherwise all objects have a luma value of zero.

Next level? On to further room-by-room checking. Level boundaries do change from map to map on the C64, so they need to be stored somewhere. I make notes of the rooms that are level boundaries and a memory hunt for those three numbers comes up with a table containing them for every maze. Hunting for the same bytes on the Atari is without result, so I decrease them by one and Bingo! I decide that 124 pod rooms should be sufficient and put the three "level boundaries" - decreased by one from the C64 to the Atari - into the last three bytes of map segment five. (I could have patched it right into the maze se-

lect routine as well but decide to store it in map data to allow for eventual loading for further maps.)

Action still keeps crashing on me and zapping code. I suspect that my long arrays eat up too much memory and re-write the code to load and convert one maze at a time.

Broken rooms?

Further testing reveals a seemingly "broken" room which is impassable. As it is blocked off in another copy of C64 shamus as well, I check a map of the level and find that it's just a clever maze design my converter handled correctly. Different colors for ? and potion bottles on the blue, green and red levels are handled with a little more code.

Re-writing the Action! converter takes longer than expected as I inadvertently save a "zapped" source code that is barely longer than the visible screen and have to re-do changes from a previous version, hopefully resulting in cleaner code. It also unearthes a bug caused by a wrong variable declaration in the corridor conversion function. That bug had prevented the function from returning $FF for unknown corridor configurations and I suddenly find that some C64 mazes use corridors

New Corridors!

which do not translate into Atari codes. I briefly consider patching the Atari maze engine but quickly realize that I have no clue how it works. Players will have to make do with "functional equivalents", rooms walled off to simulate right and left "dead ends". Vertical T junctions used in one maze have to be replaced by angled corridors at the expense of making the maze layout a bit less mysterious (the C64 layout loops back onto itself thanks to up- and downward exits leading to the same room). Fortunately there is no vertical corridor which no kind of trickery could simulate.

A black swan?

Before I return to testing the remaining mazes room by room, I start writing what is to become this document. Returning to testing that turns out to have been a bit premature, as I come across a room that should not exist. I had incorrectly assumed that all pod rooms are empty, as they are on the Atari. What I had considered an automatism that removed internal walls from pod rooms on the C64 had just been a wrong conclusion of sloppy testing. Fortunately the Atari can display pod rooms with internal walls, so just a little change to the converter does the trick. (The spacing of the pod room barriers on the C64 is a bit different though, as they obscure the vertical walls on the Atari.)

Testing also reveals that I need to either completely rewrite or disable the bonus object shuffle routine as it moves bonus

objects into corridors in some C64 mazes. Fixing this would require mapping all C64 mazes in order to shuffle objects between chambers only. As I might want to play those mazes without knowing them by heart, and as the C64 game does not shuffle objects at all, I decide to simply disable this for the C64 mazes.

Fixing an old bug

By now my code works! There seems to be a bug in the original game, however. The Atari version of Shamus increases the speed of the game with higher difficulty settings as well as higher levels. (This is done by decreasing $0206 which controls how often code starting at $2F98 runs a simple loop of the Y register from $FF to $00. The value in $0206 is decreased by 2 for every difficulty level and by 1 for every level in the maze. NOVICE will have $07 at $0206, when making it to blue level it will drop to $06, etc.)

The problem with this is that any decrease below 0 will slow down the game so much as to make it unplayable (this causes the inner Y loop to run 255 times for a total of 65535 loops).

In order to notice that you have to either start at "ADVANCED" and make it to the red level or start at "EXPERT" and make it to the blue level. (I don't think I ever played the game beyond "NOVICE" level.)

While the code that needs fixing is not time-critical as it is executed during level changes only, actually fixing it turns out to be complicated. My first idea is to simply prevent the value in $0206 from dropping below 1 (and maybe allowing for a gentle increase in game speed by reducing the $FF in the inner loop.)

Speed up? - speed down!

As it is possible to return to lower levels within the game, that value can increase as well and so I would need a major re-write to change the whole "speed setting" code to keep it from increasing that value (but maybe change back the inner loop to FF) when returning to a lower level.[8]

A tournament anyone?

As fixing the speed bug led me to the "end of maze" code (which also changes speed) the idea of a "tournament mode" comes along. A "thumbs up" on AtariAge later I start coding. While tournament turns out to be an easy patch, it makes display of the current maze (almost) a necessity rather than a nice to have feature. I remember the ANTIC jump instruction

[8] just "cutting off" values below zero would slow the game down on returning to a lower level even if the speed-up when entering the higher level was nixed by the cut-off routine.

which allows for easier patching of display lists and replace the jump at the end of the original list with a jump to my display list patch which after displaying the newly inserted lines jumps back to the original list.

After much trial and error I find a simple solution for the speed bug. I allow the speed value to go up and down freely and even drop below zero but patch the delay loop code itself to ignore speed values below zero (a patch made super easy by a superfluous `CMP` instruction that can be conveniently replaced by a branch out of the loop for negative values).

Not that it was likely someone would ever play Shamus long enough to run across those bugs, but releasing my favorite game with known bugs...?

Playing five mazes in a row without a pause might be a bit tough, and I'm quite sure that Shamus' lack of a pause function is not intended to increase difficulty but rather owed to its early release date.

Time for a pause

Pausing requires some way to detect keyboard input, either by watching keys during vertical blank or by using keyboard interrupt code. First experiments are not very encouraging until I find out that the vertical blank vector addresses are used to store game variables. So I need to "redirect" all game use of them to a safe spot. While this works, it causes a strange "discoloration" of the game until I realize that I should not return to `SYSVBV` but rather `XITVBV` after my VBI to prevent `SYSVBV` from copying all kinds of random data from repurposed shadow registers.

Pausing is rather easy, with a new "Pause" display list shutting down the display list interrupt-based game mechanics as well. Unpausing turns out to be a lot trickier. I can't seem to get rid of the original keystroke whatever I try. I finally settle on checking for "fire" rather than another keystroke to end the pause.

As I am a bit concerned about VBI code stealing at least 71 cycles[9] during every VBI, I try to start the pause routine with a keyboard interrupt. This fails because of an inexplicable perseverance of the interrupt that re-pauses the game whenever I unpause it, regardless of my attempts to clear the interrupt by writing to `IRQEN` and `SKRES`.

As there is no obvious way to correct for the 71 extra cycles,

[9] Most of that is used by the system VBI code.

Shamus+ will be a teeny weeny little bit slower than the original.[10]

Maps While a Shamus: Case II-like map (showing completed parts of the maze only, of course) would be a nice addition and has indeed been requested on AtariAge, it is not that easy to implement. As the maze encoding allows for coding of "twisted" and "stretched" mazes with geometries that could not exist in the real world, automatic mapping will not work for all possible mazes. While I could "hand-map" existing levels this would not work with a possible future (preferably "third-party") map editor and besides, I still might want to actually play the C64 levels.

For the time I'll follow a forum opinion that mapping (or memorizing) the mazes is part of the game. A map mode will have to wait until an eventual V 2.0.

Even more maps? I briefly consider adding mazes from other versions of the game. The VIC-20 maps are quite different from the Atari and C64 maps, however.[11] Chambers can have top and bottom exits, requiring a re-write of the game engine for use on the Atari. The CoCo and TI-99 versions look like the original. While googling for maps I read that the PC (DOS) version of Shamus has yet another maze layout. Does that mean I'll have to learn 8088 assembly? Another "to do" for a future update.

Enjoy! Thanks for bearing with me. The following chapter explains how the maps are coded on the C64 and Atari and explains the expanded Atari map format I used to encode the C64 mazes on the Atari. It is required reading if you plan to write a map editor (nudge, nudge) or want to code a new maze yourself. If you don't plan to do either, just go ahead and enjoy the best 2nd person shooter you'll find on the Atari.

[10] the code will execute for about 2 1/2 thousands of a second on an NTSC system, which equals approximately 1/4 of a percent of slowdown.

[11] as observed on YouTube

Shamus Maze Coding

This description is based on the "Homesoft" file version[12] of Shamus (chosen because of the fixed Synapse title music bug) and several cracked versions of C64 Shamus which seem to be identical except for the "Trainer" screens. It is provided without any guarantee whatsoever except that it probably contains errors based on wrong deductions.

Terminology

Chambers

"Chambers" are wide rooms with a left and right exit which may have some interior walls. The interior walls are laid out along a grid resembling a # with each line having having three parts.

```
  |  |
--+--+--
  |  |
--+--+--
  |  |
```

Figure 1:
Atari maze room 0

[12] The ROM version uses different memory locations, including those for game variables.

Corridors

"Corridors" are smaller passages with more "fill" around the walkable part. The walls conform to the same grid as described above:

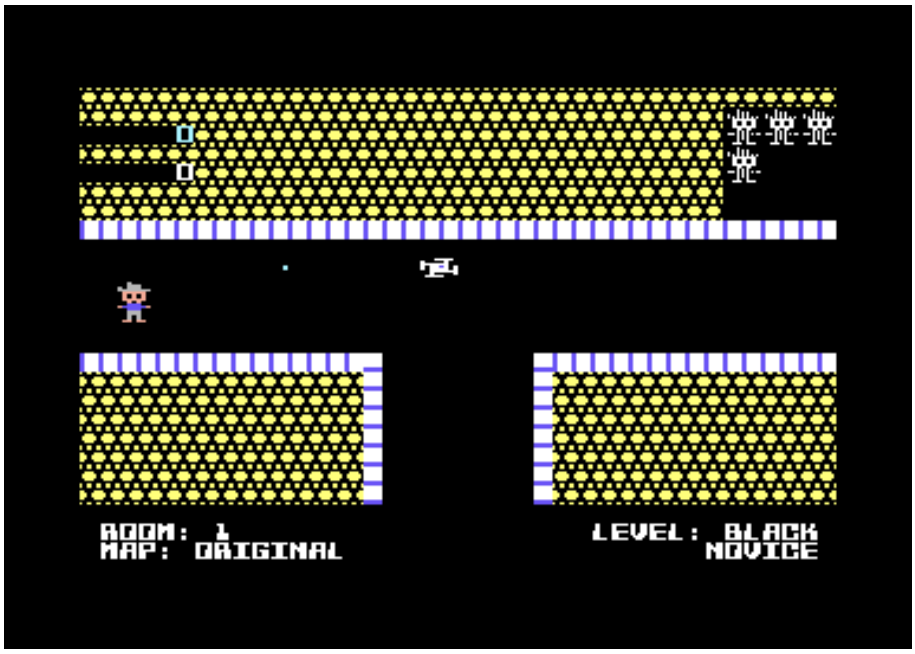


Figure 2: C64 "Original" maze room 1

Pod Rooms

"Pod rooms" are chambers ~~without interior walls~~ containing vertically moving barriers with a small horizontal slit that the player must shoot through and hit the bonus object in the center of the pod room to open the barriers and continue.

Figure 3:
Atari pod room 18

Level *"Level"* comprises all rooms within a maze that have the same color. On the Atari the screen background is colored according to the level while on the C64 the maze walls indicate the color. In addition the current level is spelled out on the lower right of the screen for both versions.

Maze A *"maze"* comprises 128 rooms numbered from 0 to 127 ($00 to $7F). The Atari has only one maze while the C64 has five.

Segment *"Segment"* refers to a 128 byte long table, with byte 0 referring to room 0, etc.[13]

Object *"Objects"* refer to the items found in the maze and comprise potions (extra lives), mystery bonuses (?), keys, keyholes and the Shadow.

Bits Bit numbering is from 0 for the least significant bit to 7 for the most significant bit, with bits 0-3 referred to as the "lower nibble" and bits 4-7 as the higher nibble.

C64 encoding

The main part of the levels is coded in three segments of 128/$80 contiguous bytes each and located as follows:

Original	Holmes	Cluseau	Marlowe	Bond
$610D	$628D	$640D	$658D	$670D

When choosing a maze using `[F3]` $15/16 are changed to point to the selected maze and when starting a game with `[F7]` the maze data is copied to $7000-$717F.

Tables with five bytes[14] each indicate the first rooms of the blue, green and red levels for every maze:

table for starts at	blue level $60FE		green level $6103		red level $6108	
Original	$26	38	$43	67	$5D	93
Holmes	$2B	43	$47	71	$66	102
Cluseau	$26	38	$4C	76	$73	115
Marlowe	$1C	28	$3E	62	$68	104
Bond	$1D	29	$38	56	$6A	106

[13] with the exception of segment 5

[14] $60FE contains the first blue room for the "Original" maze, $60FF for Holmes, etc.

Segment 1

Bits 2-7 encode the horizontal walls with a set bit generating a wall and a zero bit leaving out the respective part of the wall. Room 0 shown above therefore would therefore have a code of %01010000 or $50 and room 1 has a code of %11110100 or $F4.

```
  |  |
-7+-6+-5
  |  |
-4+-3+-2
  |  |
```

Bits 0 and 1 encode objects which upon entering a room will be randomly placed at any of the 9 positions seperated by the grid shown above:

$00	no object
$01	potion (extra life)
$02	mystery bonus (?)
$03	key (If bit 7 of segment 3 is set as well this denotes a lock)

Segment 2

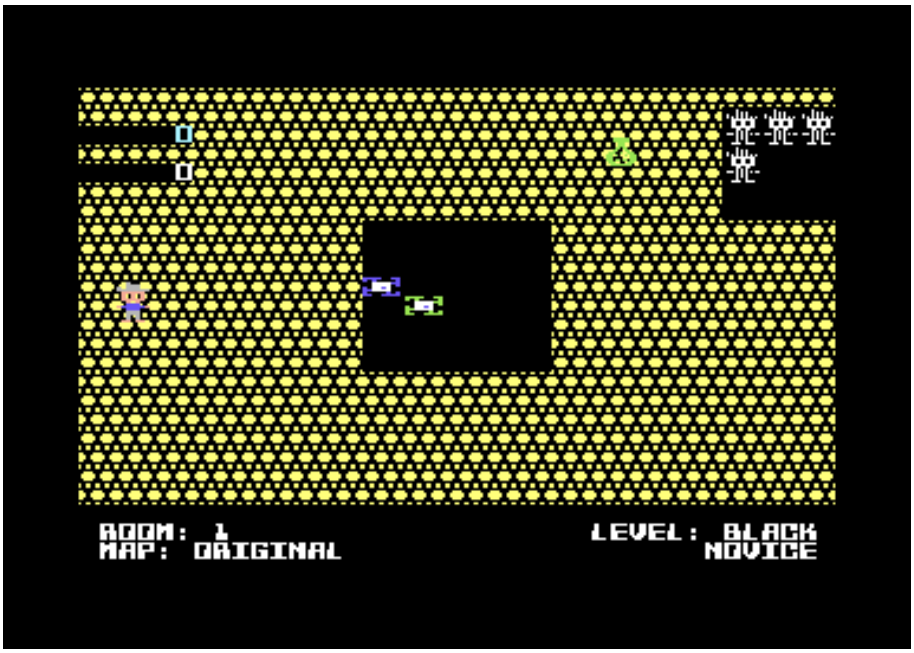
Bits 2-7 encode the vertical walls with a set bit generating a wall and a zero bit leaving out the respective part of the wall. Room 0 shown above therefore would therefore have a code of %01001000 or $48 and room 1 has a code of %00100100 or $24.

```
  7  4
--+--+--
  6  3
--+--+--
  5  2
```

If bits 0 and 1 are set the room is a pod room. Coding a pod room without an object will likely result in the room being impassable because the moving barrier only seems to disappear when hitting the object.

If bit 0 is *not* set, the room will be a corridor. The shape of the corridor will be determined by the enclosing walls as encoded in bits 2-7 of segment 1 and 2.

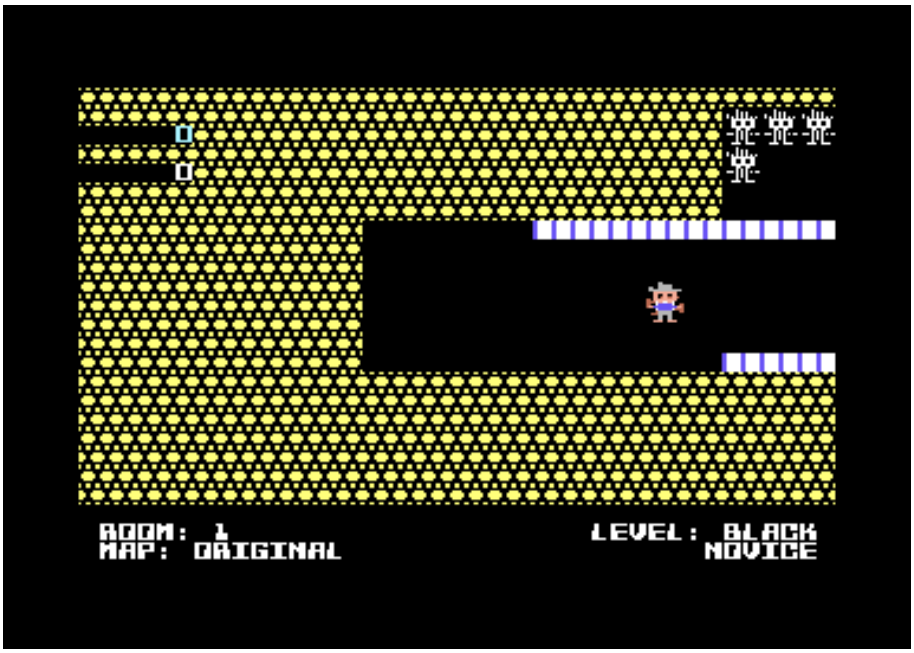
Figure 4:
Segment 1: $00
Segment 2: $00



The center of the corridor is always present, even if all relevant bits are zeroed. Note that object coding still works but random placement may result in objects placed within the solid part of the maze.

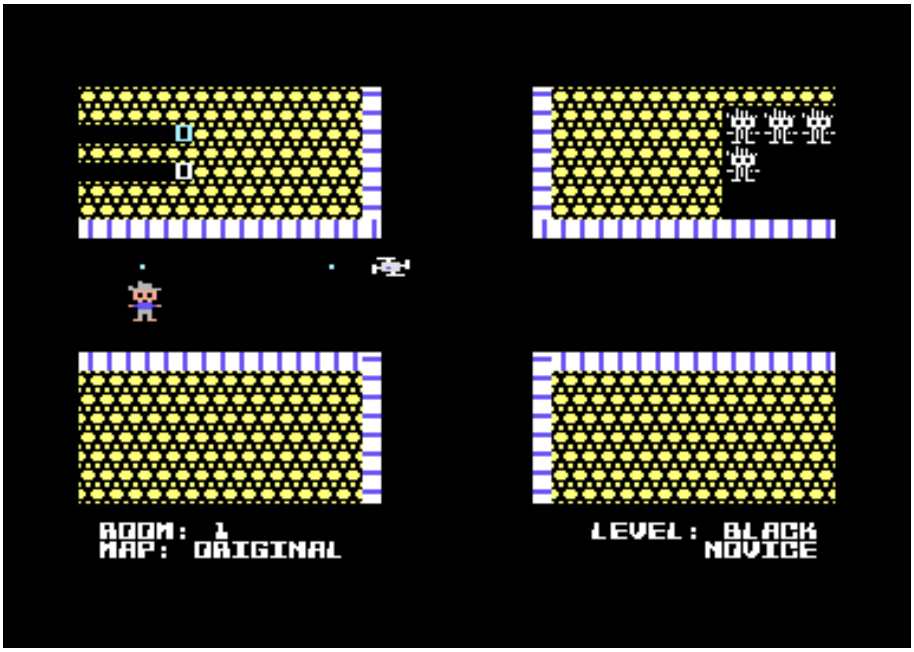
Any walls that do not enclose this center area will cause another part of the corridor to be "cleared" and open up a passageway in that direction. Note that this can result in "open" corridors which are not fully enclosed within electrocuting maze walls. Contact with the solid part of the maze will nevertheless kill the Shamus (apparently a player-playfield collision).

Figure 5:
Segment 1: $20
Segment 2: $00

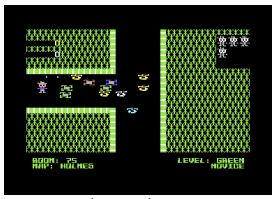


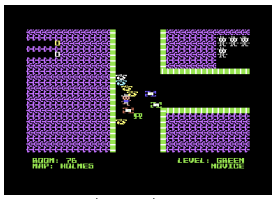
This allows for flexible coding of corridors but also allows to code corridors that cannot be coded on the Atari.

Figure 6:
Segment 1: $B4
Segment 2: $B4

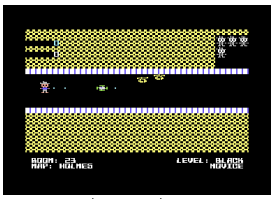


‘The following ‘non-Atari” corridors are used in the C64 game:


$90-$BC


$24-$F4


$D8-$08


$FC-$00

Segment 3

Vertical Connections

For corridor rooms[15] the lower 7 bits contain the room number of the vertically connecting room. As there is only one number, Shamus will arrive at the same room regardless of whether he leaves a room through the top or bottom exit.[16] This can point to any room in the game but will result in instant death when leading to a room without a corresponding passage, e.g. entering a chamber from above or below or entering a corridor without a passage at the side of entry[17]. As there are no restrictions to the selection of connecting rooms it is possible to construct “twisted mazes” as well as mazes were returning through the entrance will lead to another room than the one Shamus came from. (This might make it hard if not impossible to program a WYSIWYG maze editor.)

Keys and Locks

Bit 7, unused by the 7-bit room number, is a flag that will change a key object into a lock object when set. Chambers with locks have the right exit closed when bit 6 is zero and the left exit closed when bit 6 is one. The respective exit will open when the Shamus touches the lock and has a matching key in his inventory. While obviously intended for use in chambers, the encoding mechanism works for corridors as well.[18] As the random object placement may result in the lock being hidden in the solid part of maze, the player would have to exit and return to the room repeatedly in order to be able to unlock and open the door.

[15] The game engine actually does not care whether a room is a corridor room or a chamber but without disabling collision detection it is not possible to leave a chamber vertically.

[16] This seems to be a remnant of the Atari maze layout which does not contain any rooms with both top and bottom exits. Using rooms with top and bottom exits will result in mazes looping back onto themselves that are harder to map but playable.

[17] As Shamus will continue to respawn at the point of entry to the room, this will almost instantaneously wipe out all remaining lives.

[18] Bit 6 would still be used for the vertical connection info in this case.

C64	color	Atari	C64	color	Atari	C64	color	Atari	C64	color	Atari
$00	black	$00	$04	purple	$58	$08	orange	$18	$0C	grey	$0A
$01	white	$0E	$05	green	$BA	$09	brown	$E6	$0D	l. green	$BE
$02	red	$26	$06	blue	$74	$0A	l. red	$2A	$0E	l. blue	$7A
$03	cyan	$9C	$07	yellow	$E8	$0B	d. grey	$06	$0F	l. grey	$0C

16 colors The lower nibble of segment 3 contains color information for keys and locks. This is coded with the standard C64 color palette (equivalent Atari colors are shown).

Potions (extra lives) and mystery bonus (?) objects have a fixed color for every level:

Level	Mystery Bonus	Potion
black	$04 violet	$0D light green
blue	$04 violet	$0E light blue
green	$05 green	$09 brown
red	$08 orange	$09 brown

Atari

The Atari file format is based on the different method of storing the map in the Atari version of Shamus. It consists of 5 segments of 128/$80 bytes each, stored contiguously for a file size of 5x640=3200 bytes.

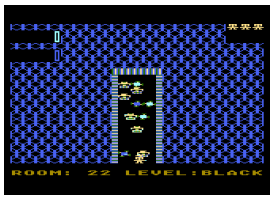
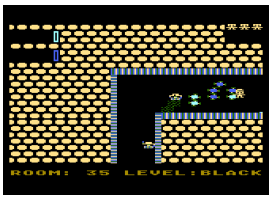
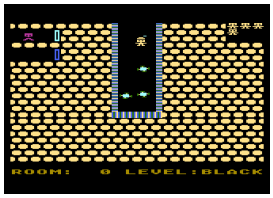
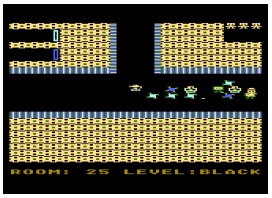
Segment 1

This is stored at $23F2 to $2471. Bits 0-5 encode the horizontal walls with a bit set generating a wall and a zero bit leaving out the respective part of the wall. Room 0 shown above therefore would therefore have a code of %00001010 or $0A.

```
  |  |
-0+-1+-2
  |  |
-3+-4+-5
  |  |
```

If bit 7 is set the room is a corridor room and bits 4 to 6 control the type of corridor according to the table below.[19] Equivalent C64 segment 1/2 coding is listed.

[19] Corridor type $C0 is not used in the original Atari maze.

			
$80 dead end open at bottom	$90 bottom to right angled corridor	$A0 bottom to left angled corridor	$B0 T with bottom exit
C64: $40/$6C	C64: $64/$64	C64: $D0/$2C	C64: $F4/$24
			
$C0 dead end open at top	$D0 top to right angled corridor	$E0 top to left angled corridor	$F0 T with top exit
C64: $08/$D8	C64: $2C/$D0	C64: $98/$98	C64: $BC/$90

Segment 2

This is stored at $2472 to $24F1. Bits 0-5 encode the vertical walls with a bit set generating a wall and a zero bit leaving out the respective part of the wall. Room 0 shown above therefore would therefore have a code of %00010010 or $12.

```
  0  3
--+--+--
  1  4
--+--+--
  2  5
```

For corridor rooms[20], this contains the number of the vertically connecting room.

Segment 3

This is stored to $1D43 to $1DC2 and is used to encode objects as follows:

[20] as on the C64, this works regardless of room type with normal rooms preventing exit at the top and bottom unless collision detection is disabled.

$00	no object
$01	keyhole (equivalent to C64 lock)
$02	key
$03	mystery bonus (?)
$04	potion (extra life)
$06	Shadow

The actual location of some objects on every level is randomly swapped by code at $18AF when starting a new game.

Segment 4

This is stored to $1DC3 to $1E42 and stores the color of objects. The normal Atari color code is stored in the lower nibble. In the original Atari maze all objects are assigned a luma (brightness) of 6 by shifting the value from this segment up 4 bits and `AND`ing $06.

As the Atari and the C64 use different color palettes and several C64 colors (black, white and three shades of grey) are represented by one Atari color with different brightness values, an extension of the original format is required to encode C64-like colors.[21]

The luma value is therefore stored in bits 5-7 and the Atari game code is patched to "rotate" the color byte at runtime and use it as a normal Atari color value.

While it might appear easier to use "real" Atari color values right away, this is not done because bit 4 is used to encode the position of the door in chambers with keyholes. If bit 4 is set, the left exit is closed, a cleared bit 4 closes the right exit. (As the lowest bit of Atari color values is not used, its presence does not change the color when the two nibbles of the color byte are exchanged.

The original Atari maze data is converted to this format during game initialization, with all colors coded with a brightness value of $06.

Segment 5

This segment is used to store information that is not encoded in the "main" map segments described above but rather stored within the game executable code and/or uses a completely

[21] Failure to at least attempt this would be quite shameful for a programmer taking pride in using a computer that has a color palette 16 times as varied as that of the C64.

different system of encoding on both machines. It does not correspond to "native" storage of the Atari version.

Byte 0 of segment 5 is a count of the number of pod rooms in the maze.[22] The following bytes contain the room numbers of the pod rooms. The Atari does not store "pod room" as a property of a room in a list indexed by room number but has a separate list of pod rooms, stored at $1EE3 to $1EEA which is checked by code at $1FEE whenever a new room is entered. This code is modified as follows: The number of pod rooms from byte 0 is patched to $1FEF and the address of the pod room list is patched to $1FF6//1FF7.

The last three bytes ($7D-$7F) store the level boundaries. While the C64 encodes this as the first room of the higher level the Atari stores the last room of the lower level, so these numbers are one lower than the corresponding C64 boundaries. These are patched to $2F48 to $2F4A.

[22] This is one higher than the number of pod rooms in the converted C64 maze, with the extra room being room 127. In C64 Shamus the Shadow always appears in the middle of room 127 while on the Atari that room needs to be a pod room to allow the player to finish the maze.

Various Notes:

Moving through the maze

Leaving a room to the left will put Shamus in the room with the next lower number while leaving it to the right will send him to the room with the next higher number. That means leaving room 25 to the left will always lead to room 24 and to the right to room 26. Whether those rooms are actually located side-by-side on the map doesn't matter as the electrocuting walls will prevent any progress outside the maze (unless collision detection is turned off, that is). Up/down movement is coded as described above.

Shamus strategies

Finish off as many enemies as possible from the entrance to the room and check the patterns of those you can't reach from there. Stand still, hold down the trigger and tap the joystick for rapid shooting. (This is easier with a joystick with short throw. My absolute favorite for this is the Suncom TAC-II.) Then advance and finish off the remaining enemies (if required). If shot at, running back to the previous room is often easier than dodging bullets in a small corridor and might be your only choice when several bullets approach. Be aware that Shamus will speed up on the Atari once you've killed all enemies. More than once that had me running into a wall.

Differences between Atari and C64[23] versions:

Disclaimer Some of this might be a matter of personal preference.

- Where the Atari has nice, glowing (pulsating) walls - especially on a real CRT - the C64 walls look a lot cleaner with their alternating of white and the level color. The "wallpaper" (solid part of the maze) around is a much brighter yellow, altogether giving the Atari a darker, more "space dungeony" feeling. (To me the C64 maze looks too bright and friendly.)[24]

[23] as played on the VICE x64 emulator.

[24] I am used to the PAL version. The NTSC version looks a bit brighter as well.

- The same goes for the robots which have gloomier colors on the Atari.
- Atari ION-SHIVs can kill enemies through walls (when the enemy touches the wall) which doesn't work on the C64.
- ION-SHIV graphics look a bit more elaborate on the Atari, ION-SHIVs are larger compared to the player.
- The C64 has a "splash" animation for ION-SHIVs hitting something though.
- Enemy shots have a ("ping") sound on the Atari but are silent on the C64.
- The Atari version plays some background sound during gameplay and a "warning" sound a short time before the Shamus actually approaches.
- While the Atari Shamus has a small gap between hat and head that is big enough for shots to pass through, the C64 Shamus is "solid", making it harder to dodge bullets.
- The C64 Shamus and Shadow seems to be a bit taller compared to the maze than on the Atari.
- On the Atari the keys, keyholes and bonuses disappear when the Shadow appears (probably because they use the same player) while they stay on the C64.
- At least on the emulated C64 the Shamus can be seen stationary for a couple of seconds before it moves towards the player while it starts moving right away on the Atari.
- Snap Jumpers and Robo Droids seem to be one color only on the C64.
- I am not entirely sure but some of the enemy patterns look different on the C64 (although I have not played that version long enough to be certain).
- The C64 enemies seem to shoot each other less than those on the Atari.
- The Atari Shamus walks faster than the C64 Shamus and speeds up on every new level and at higher difficulty settings. The C64 Shamus seems to keep a constant speed throughout the game.
- The Atari Shamus walks faster after a room has been cleared of enemies.

- On average there seem to be fewer enemies in the C64 game.
- When finishing off the Shamus in room 127 the Atari game increases the difficulty and loops back to room 1. The C64 games shows a long scrolling text at the end and then drops the player back to the game menu.
- When Shamus loses a life the whole screen flashes on the C64 while only Shamus flashes on the Atari.
- The Atari swaps some bonus objects around every time a new game is started is loaded[25], the C64 doesn't.
- The C64 version has extra maps ;-)

[25] this is performed by code at $18AF and affects rooms stored from $18F6 to $18FD. As this does not work on C64 mazes which have a corridor where this routine expects a chamber, this is disabled when playing C64 mazes on the Atari.

Speed

Shamus speed on the Atari is controlled by the value in $0206. It controls how often a delay loop at $2F98 counting from $FF to zero is executed. It is set to $07 when starting in NOVICE and 2 less per higher difficulty setting, i.e. $01 for EXPERT mode. It is decreased by 1 when advancing a level and increased 1 when returning to the previous level. When completing a maze it is set to $2(3 - number\ of\ finished\ mazes) + 2$ by code at $2B6B.

While the speed setting code does not allow the value of $0206 to drop *below* zero the delay loop will execute FFxFF times if it drops *to* zero. The game becomes basically unplayable then.

I have "fixed" this by ending the delay loop after one $FF-$00 iteration when $0206 is zero or less.[26] While the (useless) "don't drop below $00" logic at $1F0F prevents speed from dropping below zero, it will still increase the speed value (thus slowing Shamus) when returning to the previous level. So if Shamus enters a level at speed 0, speed will remain at zero for the new level but increase to one when returning to the previous level, making him slower than he was initially. As I consider this an anomaly I have "fixed" it by dropping the check for negative speed during speed adjustment.

[26] "less" meaning $80 to $FF, i.e. a number causing the N flag to be set.

Conversion

The C64 map data from $610D to $688E is saved to a binary raw data file (in this case called `SHAM_C64.MAP`).

This map data file is processed by an Action! program called `SHAMCONB.ACT` which saves an Atari map data file named `SHAM_A8.MAP` which is saved as `SHAM_A8.MAP.dat` and inserted into the patched binary for the Atari.

New Corridors

Corridor shapes not available on the Atari are replaced by an approximation. "Dead End" corridors and horizontal corridors are replaced with chambers that are walled off to create a dead end (which will have some enemies in walled off areas, making it impossible to "clear" the level and speed up the Shamus). "T" shaped corridors with up/down exits (Holmes rooms 75/76) are replaced by "L" shaped corridors which connect to each other vertically and allow to connect to rooms 74 and 77 horizontally (which will make the maze easier to map as possible confusion by rooms looping back on each other is removed).

Acknowledgements

Special thanks to the authors of all the utilities used in the creation of this patch, especially phaeron for Altirra, David Firth for Atari800MacX, Peter Dell for WUDSN, Rob McMullen for Omnivore, the VICE team members for x64, Clinton Parker & JAC! for Action!, samar productions for C64Debugger.

Thanks to Al for AtariAge, without which using an Atari in 2017 would be much harder and to all who answered my questions or provided moral support by following and even liking my progress.

Thanks to my family who endured my staring at a 40-column screen and occasional sighs during quite some time.

And finally, thanks to all the Atari podcasters who keep the Atari world alive in a new medium.

LaTeX and the Atari

While my profession does not result in a lot of written output, I have enjoyed dabbling with LaTeX since the 1990s, when I first used it on my Atari ST. I simply like the aesthetics of Computer Modern fonts and the inimitable style of documents typeset using the TeX system. I always wanted to use the refman/refart/refrep type of document but never before had to write anything requiring it. This is my first attempt at using it.

Version History

This is version 1.01 of this document, correcting several typos and a wrong filename in the Conversion section.