GAMASUTRA
The Art & Business of Making Games

PRINT

Postmortem: Pixels Past's SCSIcide

By Joe Grand

> SCSIcide*, as its name suggests, is nerdiness to the core. You are a disk drive read head, and your mission is to read the color-coded bits of data as they scream past you on 10 separate data tracks. As each bit is read, your read head changes color to indicate which random bit you must read next. Read all of the bits in the required order and you advance to the next platter (level). If you take too long, your latency buffer times out, your disk crashes and the game ends. Your score is displayed at the top of the screen in, of course, hexadecimal notation.*
> -Ben Valdes, staff writer for *Classic Gamer Magazine*

SCSIcide (pronounced 'skuzzy-side') was my first design attempt for the Atari 2600 console. I wanted to create a game that was simple yet addicting, and would incorporate enough features to make the development of the game a challenge and learning process. From concept to completion, the project was essentially a one-person show. The original Atari 2600 game developers were tasked with all aspects of the process, including concept, graphics, sound, design, and implementation; twenty years later, the creation of SCSIcide was no different.

How does one decide to create a game for a 20+ year-old console? I became actively involved with the Atari 2600 development community in 1997. My hope was to use *SCSIcide*, my first programming attempt for the Atari 2600 and my first publicly released game, as a stepping-stone into design for other systems, notably the Game Boy Advance -- which also offers interesting design constraints. By day, I have my own product development and engineering firm and spend most of my time thinking of ideas, creating prototypes, and licensing them to interested parties. By night, I do the same thing (after a few rounds of *SCSIcide*).

Essentially, *SCSIcide* is a psychotic, instinctive reflex game and was the first paddle-based homebrew game for the Atari 2600. Like the original *Pong*, *SCSIcide* was designed to be easy to understand yet difficult to master. Something that, if this were an arcade game, kids would be lining up and pumping quarters into the cabinet as quickly as they could.

The "homebrew" Atari 2600 game *SCSIcide*, released two decades after the manufacturer retired the console.

The original idea was based on a rotating circular disk, mimicking a real hard drive. However, I knew programming the 2600 was rather difficult due to the limited memory environment and tricky timing requirements (see sidebar), so I settled on an easier horizontally based, single-screen game. The game has slight similarities to *Kaboom!*, a simple but devilishly fast-paced game that made me nervous every time I played it. I wanted to create a game that would replicate this nervousness.

Not only was there the challenge of creating a game that people would play, I also needed to design game packaging and build physical cartridges. From laying out a new circuit board for the old Atari 2600 cartridge cases, to drawing cartridge labels, to printing manuals, to building the cartridges, to testing, it was a successful journey. But, it wasn't without its problems.

What Went Right

1. Expecting changes in the game plan.

As with any engineering or development project, it's important to have an idea of your final goal before you start. With *SCSIcide*, the game concept essentially stayed the same from its inception (control a hard drive head and read bits of data). After deciding on a single-screen horizontal view, I set to work on developing the core game kernel, drawing of objects to the screen, and game play intelligence. I was prepared for design changes to the game plan as implementation progressed. Because the game plan wasn't completely set in stone, I had an opportunity to implement features I had not originally intended, many of which came up as the result of programming problems and bugs.

My design document was a single paragraph:

"The player will control the hard drive head (i.e. can move it left or right between tracks using the paddle). The hard drive platter will spin clockwise (starting slowly). One sector in each cluster (column) will be illuminated. The player needs to move the head back and forth and hit the button when the lighted sector is under the head (essentially "reading" the sector). After the player "reads" 8 sectors, the platter will increase in rotation speed (i.e. the next level) and another combination of sectors will be illuminated. Speed will increase for every 8 sectors the player can read. Each sector needs to be read in sequential order (i.e. one column after the next). The player will lose (or the game will be over) if he misses an illuminated sector. Scoring will be based on the number of sectors the player successfully read."

I originally envisioned the game to have a memorization component like Milton Bradley's popular "Simon" game, possibly reading the bits in the proper color order and then having to re-write them back to the disk (a la defragmenting). As I started to program the core game, I decided to axe that part in order to keep game play simple. This decision left me with extra RAM and ROM to use for other features of my game.

Instead of using the standard decimal scoring system that most games use (I had originally planned on just a 2-digit decimal score), I decided to implement a 6-digit hexadecimal scoring routine to add to the whole "hard drive" theme of *SCSIcide*. The first "byte" of the score became the level number, and the last 2 "bytes" of the score was the actual score. Towards the end of the game implementation, I had some unused memory space, and decided to create a title screen for a more "complete" gaming experience.

2. Solid development environment.

This might sound like a no-brainer, but having a development environment that has been fully tested and all components are known to work can save you lots of hair-pulling when a bug mysteriously appears in your Atari 2600 game. Each piece of my development system, such as my Windows PC, the Atari 2600 Jr., the television, and the Supercharger (to load a game through the PC sound card directly to the real 2600 hardware) was individually tested. So were all of the software components: the text editor, Atari 2600 emulator, and 6502 cross assembler. After each component was known to be functional, I combined them to form my development suite.

The *SCSIcide* development environment: PC, Atari 2600 Jr., Supercharger (plugged into Atari), and color TV.

Additionally, as the components were brought together to form the environment, they, too, were tested. For example, making sure the Supercharger works with this particular 2600 Jr. by trying to load known-good games and making sure the 2600 Jr. properly displayed graphics on this particular TV. By the time the entire environment was set up, I could be sure that if a bug or problem arose, the components themselves worked fine so I could remove some unknowns from the debugging process. Doing this helped me "Keep It Simple, Stupid" and let me focus on potential areas in the code that could have caused the problem.

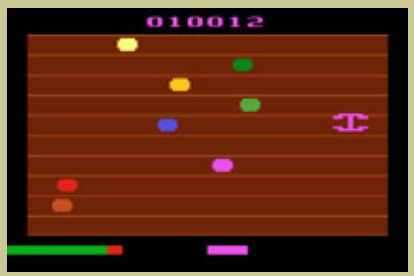
3. Play testing.

So I had a concept for a game and was on the way to making it a reality, but in order to make it good, I needed people to check it out and throw some opinions and criticism at me. Most game studios can do this in-house, by showing off code or game snippets to fellow developers or maybe beta testing a version of the game during lunch. Since SCSIcide was a one-person project, I needed a way to get some other gamer's perspectives.

The entire development process of *SCSIcide*, from inception to completion, was provided on mywebsite, including source code and compiled binaries of various incarnations of the game. By sharing my beta versions with the world (or at least those people who actually cared), I could receive much-needed comments about the game design, allowing me to tailor components as I went. I was also active on the Stella Development mailing list, which consists of a number of stellar Atari 2600 homebrew game designers and gamers eager to provide their expertise (see sidebar).

Some brilliant suggestions came from my play-testing colleagues. For one, some of the numbers and letters were hard to read with the font I drew by hand. I thought the font was fine, because I had been staring at it for a few days, but when I shared this version of *SCSIcide*, I received enough feedback about it to change the font to something a little simpler. Another suggestion was to have a "Latency Buffer" that the player needed to keep full by reading bits of data. The Latency Buffer decreases when the desired data bit is missed or if an incorrect data bit is read. When the buffer is empty, the hard drive crashes and your game is over. The buffer adds a sense of urgency to the game, especially at the higher levels when the data bits are moving fast. Little tweaks such as these led to a more exciting game to play than was originally intended.

Everybody has his or her own Great Idea™ about how any game should play. When receiving suggestions, I made sure to avoid feature creep by staying true to the original concept. I did not try to implement every suggestion offered, especially those that would affect my simplistic game model. Some great suggestions, which were unimplemented in this version of SCSIcide, were to include bad sectors (which would cause a hard drive crash if the read head simply touches the bad sector), vary the lengths of the data bits (requiring the player to keep the button down for a varying length of time to read the entire bit), and vary the data bit graphics.



The *SCSIcide* play screen.

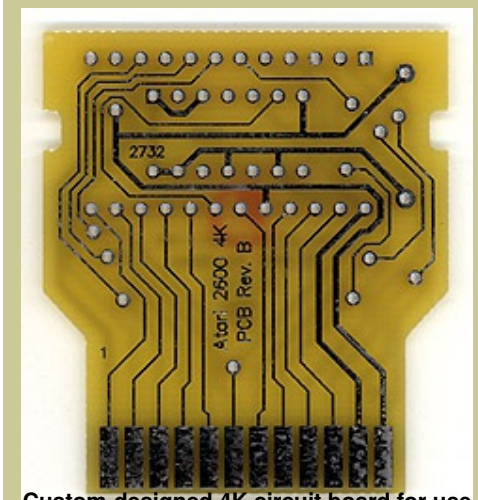
Any way you look at it, play testing and sharing your game during the development process is an absolute necessity. I would say there is no way to have a successful and popular game without getting the opinions of insiders, such as co-workers, and outsiders, such as the general game playing public.

4. Prepared in advance for cartridge manufacturing.

About halfway through programming *SCSIcide*, I knew that I needed to have a plan for building the cartridges. My goal was to release 50 cartridges, complete with color game manual and packaging, at the 2001 Classic Gaming Expo in Las Vegas, NV. I had made a public announcement on the expo site and on various classic gaming mailing lists and websites. I wanted to generate some buzz. There was no going back.

The date was March 2001. I had exactly five months to complete the game, design a circuit board, receive the prototype, test it, receive a production order, burn EPROMs, solder chips onto the boards, put the assembled boards into cartridges, and clean and label the cartridges.

The first thing I did was to create a custom circuit board for use in the standard Atari 2600 cases. I didn't want to have to modify the existing circuitry in the old cartridges, which takes a lot of time and careful soldering. The 2600 boards I made for *SCSIcide* support 2K and 4K EPROMs. All necessary components are easily obtainable at most electronics stores. This made the manufacturing process much easier and trouble-free.



Custom-designed 4K circuit board for use in standard Atari 2600 cartridge shells (Purchasing information available at http://www.pixelspast.com/homebrew.)

Once I had the circuit boards designed and manufactured, it was simply a matter of soldering the components onto the custom boards, preparing the cartridges (stripping the labels off old common Atari 2600 games, such as *Combat*, *Pac Man*, *Missile Command*, *Asteroids*, all cheaply available on eBay), removing the old guts of the cartridge, putting the new populated circuit boards in, testing, putting the cartridge together, testing again, and finally putting on the new label. Piece of cake! The 50 cartridges were successfully built by CGE and were extremely well received.


1. Soldering components.


2. Boards with ROMs.


3. Label removal.


4. Label preparation.

5. Testing.


6. Making cartridges.


7. Labeling.


8. The final product.

The various stages of *SCSIcide* cartridge manufacturing.

5. Cost-effective production.

I allocated a $500 budget to develop and build *SCSIcide*. With such a meager sum, I needed to have an efficient and cost-effective production process. The design and production of the cartridge PCBs and purchase of the EPROMs and logic ate up most of my available funds. By

planning in advance, I knew I could save a lot of money by avoiding rush fees and expensive overnight shipping charges.

The packaging and presentation of *SCSIcide* was kept to a minimum: low-cost and simple, but effective. I wanted the game to stand out in a way that no other homebrew game in the past had. I wanted the packaging and appearance to be special but not so flashy as to take away from the simple experience of the game. I didn't consider using a cardboard box because of the manufacturing costs associated with high-quality boxes and offset printing. I have seen other homebrew and low-budget game vendors use a cheap inkjet printer and flimsy cardboard packaging. This was not an option for me. I decided to package the game and manual in an anti-static bag, like a hard drive commonly is. This was a cheap way to provide a "box" for the game. It was funny and catchy, since it helped keep the "hard drive" theme of the game, and the price was right - just under $0.15 a piece!

What Went Wrong

1. The game concept was initially difficult for people to understand.

To a non-technical crowd, the concept behind *SCSIcide* is not exactly the most exciting and appealing one. One comment I received from a fellow gamer when I explained *SCSIcide* to him was "You need to inspire people to play the game. By itself, being a hard drive head is not very inspiring. Being a hard drive head is not as exciting as being a cowboy or a weird gobbling creature. The world that the drive head inhabits is too sterile and lifeless." Not wanting to reinvent the wheel and fall into an overused game mechanic, I kept the *SCSIcide* concept as-is. Being original can have its merits, though sometimes it can be risky.

One advantage of keeping the game simple is that it forces players to use their imagination. Primitive graphics allows this to be done, as opposed to pseudo-realistic, graphic-intense games that put a fixed scenario into the player's mind. For *SCSIcide*, this simplicity ended up being a good thing. Those that didn't want to appreciate or understand the hard drive theme could easily think up another scenario to immerse themselves in. For example, one gamer's three-year-old daughter thought of the concept as "Move your basket to gather the colored Easter Eggs before time runs out." His eight-year-old son thought along the lines of "Fly a helicopter and lift the disabled cars from the lanes of a fast-moving freeway before traffic stalls."

What I have realized is that many classic gamers are focused on game play more than anything else, possibly given the lack of high-resolution graphics in games they played during their youth. So even though it was initially difficult to get some people to adopt the concept of *SCSIcide*, the game play was appealing enough to make it a successful game.

Simple *SCSIcide* packaging: Anti-static bag, two-sided color manual, and high-quality color cartridge label.

2. Dealing with the challenges of Atari 2600 programming.

The challenges of programming for the Atari 2600 are definitely one of the reasons it has become such a popular system for homebrew gamers (see sidebar). Its beauty is also its bane, however, as simple changes in routines or a typo of a single opcode can render the game unusable and set you up for hours of debugging.

The requirement to keep the code space below 4K and use only 128 bytes of RAM throws many of the typical structured programming techniques right out the window. In many cases, re-using code in-line instead of calling external functions will save precious cycles, since the time associated with saving state is avoided.

I had a number of problems throughout the development cycle that were fairly quickly fixed, such as timing problems with sprite generation (drawing, positioning, and moving the individual data bits) and hexadecimal score generation (a section of the score drawing function was over a page boundary which threw off the delicate timing of the routine).

My hardest challenge was correctly implementing paddle control. Since *SCSIcide* was the first homebrew game to use the paddle, there was no previous work to reference. One byte of RAM was used to store the current numerical value of the paddle (which corresponds to the vertical position of the drive head on the playfield). At the beginning of each frame (in the vertical blank), the capacitor inside the paddle controller is discharged and, a few cycles later, set to recharge. During every scanline draw, the value of the capacitor is read. Depending on how long the capacitor in the paddle takes to charge (based on a simple RC time constant) determines the vertical position of the drive head on the screen. For example, the less resistance in the potentiometer of the paddle will cause the capacitor to charge more quickly, and place the drive head towards the top of the screen. If the paddle was moved in the other direction, increasing the resistance of the potentiometer, the capacitor will take a longer time to charge, and the drive head will be placed lower down the screen. Programming efficient and non-fluttering paddle control took the longest amount of development time and required a great deal of experimentation with the Atari 2600 system.

3. Color compatibility issues.

Because *SCSIcide* is a fast-paced, intense game, quickly identifying the proper color bit to read is extremely important. Knowing right from the beginning of development that colors were an integral part of the game, I picked a visually simple color scheme and 10 unique and non-similar colors. I ended up with a brown background and data bits with bright colors throughout the spectrum.

Sometimes the data bit colors appear very similar to each other, making it difficult to know which bit to read next. This is crucial in the later stages of the game where one false move could cause a hard drive crash, ending your game. It turns out that there are major variations in how televisions display color, so two colors that I had picked to be easily distinguishable might be displayed as similar colors. This was a design flaw in SCSIcide that was not uncovered until the final testing stages. It is really only important that the colors of the 10 data bits are easily distinguishable, not what colors actually are.

Besides the television variations, *SCSIcide* was designed for NTSC (the North American television standard, also known as "Never The Same Color"), not for overseas standards such as PAL or SECAM. This is a potential problem because both PAL and SECAM display colors differently than NTSC. During *SCSIcide* development, there was some play-testing overseas and it was decided that the game will play fine on PAL sets, even though the colors were slightly off (but they were still distinguishable from one another). The proper solution would have been to create a suitable PAL color scheme and have the NTSC or PAL mode user-selectable. This was decided against due to lack of time but would definitely be implemented in any future game. SCSIcide was not tested on any SECAM systems. The color generation on SECAM Atari consoles is quite odd and the colors might all be converted to shades of grey. That would not make for a fun and exciting game of *SCSIcide*. Considering the huge range of overseas gamers, globalization is a necessity.

4) Not enough time to add more features.

As much as I tried to plan enough in advance to complete the game, build the cartridges, and package the final product, I did not have enough time to add all the features into SCSIcide that I would have liked.

The most neglected issue, though the game is still fully playable, is the screen flicker in between levels. This flicker is caused by the length of time required to calculate the colors and positions of the upcoming level's data bits using multiple calls to a pseudo-random number generator function. The routine is called immediately before the next level starts. Execution time of the routine takes too long and causes too many scanlines to be written to the TV screen by the Atari. This in turn causes a flicker until the Atari can resynchronize with the TV. The flicker initially happened every time a data bit was read, but I modified the code to have it happen only at the beginning of each level, making game play much nicer and suitable for release. Dealing with timing issues is one of the challenges of programming for the Atari 2600 (see sidebar).

SCSIcide is also hard, if not impossible, for color-blind individuals to play, because the game requires the player to read colored bits in the correct order. If the player cannot differentiate colors, the game is essentially useless. I wanted to have the next required bit blink or be identifiable in a way other than color, but the idea came too late in the implementation cycle to make it work before the release date.

5. More demand than supply.

Fifty cartridges were manufactured by hand for the initial release of the game. I didn't know what type of response I'd receive at the 2001 Classic Gaming Expo. There was a slight buzz about the game being released, but the classic gaming community is very small (but growing) compared to the mainstream gaming community, so I decided to play it safe and do a small run. *SCSIcide* was to be released on Sunday, August 12, 2001 at 10:15am. An announcement was made at the show about 30 minutes prior to allow people to form a line to purchase a copy. I decided in advance to sell my game for only $20 complete. This price was much lower than the $30 or $40 that classic homebrew games are commonly being sold for.

Gamers await the *SCSIcide* release at Classic Gaming Expo 2001.

By 10:30am, all the SCSIcide cartridges were gone! I was amazed and slightly embarrassed, as there were at least 50 more people at the show who wanted to buy a copy of the game. However, people signed up on a waiting list to reserve a future copy of the game. Looking back, I should have taken a bigger risk up front and made more cartridges, considering that I had already spent the time and effort to design the game, though with my self-imposed $500 budget, this might not have been possible.

As The Hard Drive Turns

Many of the same challenges exist when programming for a classic game console as they do for newer game environments. In just nine months of part-time work, *SCSIcide* was brought to life. With over 200 copies sold since its release in August 2001, it is one of the highest selling homebrew videogames ever for the Atari 2600 system. *SCSIcide* was selected for the All Game Guide's "Best of 2001" and voted as runner-up for the MyAtari 2002 Awards "Best Game Release of the Year."

It was a wonderful experience to create a physical cartridge-based game from a simple, non-tangible idea and release it for the public to enjoy. Watching gamers play *SCSIcide* is a thrill and I can only hope that my next game will be just as well received. My high score is 174E10. Can you beat it?

Game Data

SCSIcide

Publisher: Pixels Past
Budget: $500
Number of Developers: 1
Length of Development: Approximately 9 months, part time.
Release Date: August 12, 2001
Platform: Atari 2600

Development software used: TextPad 4.50, DASM 2.0 (6502 cross assembler), StellaX 1.1.3a, z26, WPlayBin 1.0 (for use with the Supercharger), Photoshop, OrCAD Capture and Protel PCB (for cartridge circuit board development).

Development hardware used: Pentium II 233MHz w/ 256MB RAM, Atari 2600 Jr., Supercharger, 13" color TV, EPROM programmer, soldering iron.

Project Size: Approximately 2,100 lines of assembly language source code for game; 2,557 bytes of ROM (programmed into 4KB cartridge); 115 bytes of RAM (out of an available 128 bytes); 2 easter eggs.

Sidebar: Crash Course in Atari 2600 Programming

Programming for the Atari 2600 is unlike that of any current videogame console. Sure, it might be an "obsolete" system with a 1MHz system clock, only 4K of address space, 128 bytes of RAM, complex timing constraints, and primitive sound capabilities, but that's the beauty!

The Atari 2600 hardware (which had a project name of "Stella") was originally intended to play Pong-style games. By design, there is a static playfield and 5 movable objects (2 player sprites, 2 missile sprints, and a ball sprite). The elegant system design contains three integrated circuits of note: the 6507 microprocessor (a 6502 core with no external INT pin), the 6532 Peripheral Interface Adaptor (PIA), and the Atari-custom Television Interface Adapter (TIA), which creates the television picture and sound.

Arguably the trickiest part of Atari 2600 programming is the need to keep track of the television electron beam and write to the screen when intended. Although the TIA provides automatic horizontal timing, the programmer needs to write to the screen on a per-line basis within the available time (in the case of NTSC, 76 clock cycles per scanline). The game software must also control the vertical sync (to reposition the electron beam to the top of the screen to start a new frame). More details of the television protocol can be found in the *Stella Programmer's Guide* written by Steve Wright in December 1979 (see resources section, below).

The first Atari games were limited to 2K or 4K of ROM, due to the cost of memory at the time. With specialized bank switching techniques using logic devices, the memory space of a cartridge gradually increased in the early 1980's to 8K and 16K. Only one released Atari game was 32K (Fatal Run in 1990), but many homebrew games are being developed today that will require such a size.

It didn't take long for Atari game to break out of the intended *Pong* game mechanic. Games like Activision's *Pitfall!* came onto the scene with unique concepts and astonishing game graphics that were never imagined nor intended by the original hardware designers. Still today, continual poking and prodding by classic game programmers are leading to new and amazing technical feats on the Atari 2600. Many tricks have been discovered to enable such things as asymmetric playfield drawing, displaying multiple sprites on a given scanline without screen flicker, and creating fully customizable music scores.

The development environment that the original game programmers had to work with was drastically different than what we have now. The design/debug cycle was time consuming, to say the least. Programmers had to use time-share mainframes on which to compile their code, then burn the resulting binary onto EPROMs and try the result on real hardware. Later on, throughout Atari, Activision, and Imagic, emulators and simulators running on desktop computer systems were used. Nowadays, development is much easier. Simply write some code, compile it, and immediately test it on an emulator or real hardware. Sounds easy!

Most of the development can be done right on a PC with a standard text editor, 6502 cross assembler, and an emulator. PC-based 2600 emulators have helped immensely to speed up the game development cycle. When it comes time to test your game on actual hardware, you can use a Supercharger to load the game through your sound card directly to the 2600. There's nothing quite like playing an Atari game on the real thing.

The process of building cartridges for the Atari 2600 is simple. Before *SCSIcide* was produced, the most common method for homebrew developers to make their own games was to obtain common Atari 2600 games, modify the existing cartridge circuitry (removing the ROM, cutting traces on the circuit board, soldering in a new EPROM), clean the cartridges, and put them back together. Cartridge manufacturing is slightly easier now given the availability of custom circuit boards.

Within the past three years, the classic development and gaming community has grown greatly. Not only are people playing these older games with a notion of nostalgia, they truly enjoy them. Current 2600 programmers are extremely helpful and willing to share their code and experiences. The resources now available open up game development to the masses.

Roger Williams, a participant on the Stella Development list, sums up programming for the Atari 2600 in a few concise statements:

"Stella isn't a nice girl. Stella is a bitch. She seduces you, but then you work your balls off to support her one megahertz 8-bit no-video-RAM habit. Sweat comes off your testicles as you try to get 48 horizontal pixels of unflickering video where you want them on the screen. In other environments, you deploy MUL instructions and rewrite code to increase indirection; on Stella you unroll countdown loops because you can't afford the overhead of the DEC and BNE instructions."

I couldn't have said it better myself. People come to the Atari 2600 for the challenge, the frustration, and the love.

Resources

1. 2600 101: A Tutorial By Kirk Israel, A detailed guide for learning 2600 programming, http://alienbill.com/vgames/guide

2. The Stella Mailing List, Active development list for Atari 2600 programming, many helpful, technical individuals and an archive dating back to October 1996, http://www.biglist.com/lists/stella

3. The Dig: Stella Archive Excavation, Extensive resource list, required documentation, and code examples archived from the Stella Mailing List, http://www.neonghost.com/the-dig/index.html

4. AtariAge, The ultimate resource for anything Atari, including new, homebrew production services, message boards, history, game lists, documentation, and more, http://www.atariage.com/

5. AtariAge: Titles In Development, List of on-going game development projects for all Atari systems, http://www.atariage.com/development_list.html

6. The Atari 2600 FAQ, A good starting point for beginners, http://www.atariage.com/2600/faq/?SystemID=2600

7. Stella Programmer's Guide, The essential technical guide written by Steve Wright in December 1979, http://www.neonghost.com/the-dig/dox/stella.pdf

8. DASM, 6502/6507 Cross Assembler, http://alienbill.com/vgames/guide/bin/dasm.zip

9. Paul Slocum's 2600 Music and Sound Programming Guide, Everything you need to know about sound generation, includes Paul's music driver, a player, an example song, and detailed instructions, http://qotile.net/sequencer.html

Emulators

1. z26, http://www.whimsey.com/z26/z26.html

2. StellaX, http://www.emuunlim.com/stellax

3. PCAE, http://pcae.vg-network.com/