Final Report

Space Invaders 8080

CS467/Spring/2024

Craig Harris, Nicholas Herman, Damian Russ, Julie Weber

Table of Contents

Harris, Herman, Russ, Weber

Introduction

Space Invaders holds a special place in the hearts of many as the quintessential arcade classic. Its simple yet addictive gameplay has captivated players around the world for decades, becoming an iconic symbol of the golden age of arcade gaming. Many on our team remember playing it as kids. Even if we didn't have access to the original arcade cabinets, the widespread availability of emulators allowed us to relive the thrill of battling waves of alien invaders.

Because of the ubiquity of space invaders and the simplicity of gameplay, it is an excellent project for starting into the world of emulation. Just three controls - left, right, and shoot - and a limited number of game sprites come together to create a very fun game. Moreover, given the iconic status of Space Invaders, it is easy to tell at a glance if everything is working as it should be.

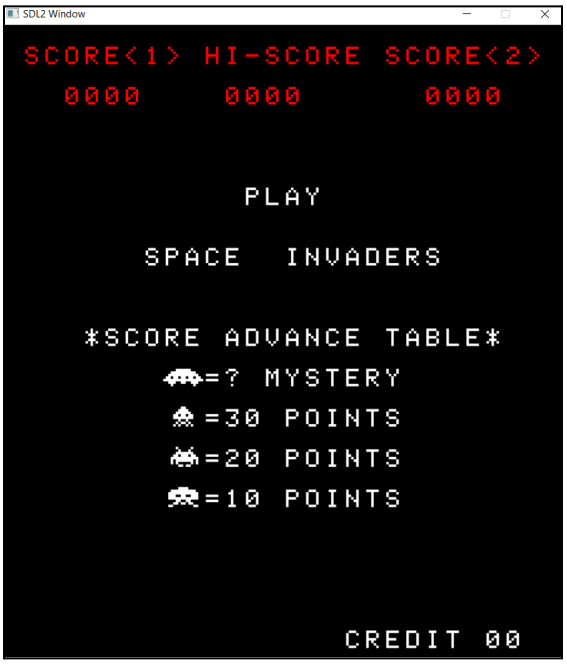
Additionally, emulating an 8080 CPU forces a deep understanding of the interface between hardware and software in a CPU. The 8080 is not a very complex chip by today's standards, but in order to emulate its instructions, an intimate knowledge of how the software is written and what the chip is doing at the hardware level is needed. The skills developed over the course of this project will greatly improve the abilities of our team as developers and carry forward into many other areas of programming at the professional level.

Software Function from User Perspective

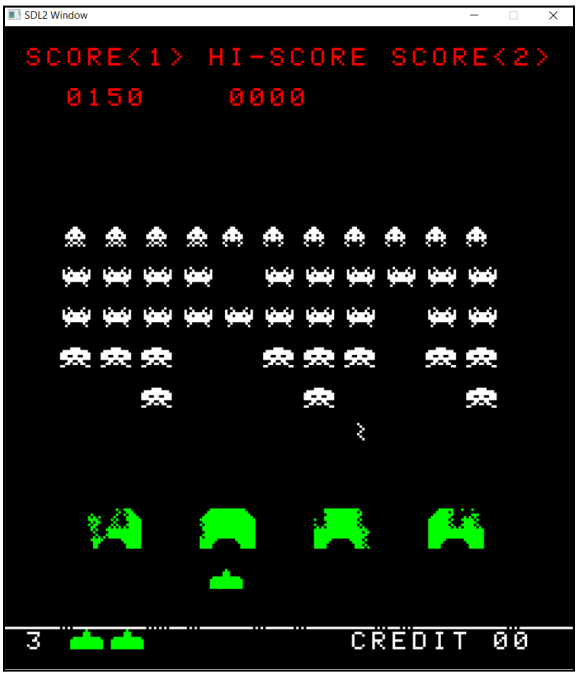
From a user's perspective, the emulator itself should be invisible. The executable will launch directly to the Space Invaders menu screen, and the game, once launched, should play as if the user is playing the 1978 arcade game.

The player presses c to insert a coin, then presses '1' to start the game. The player's current score and the high score are shown at the top of the screen.

The player can use the left and right arrow keys to move a cannon horizontally across the bottom of the screen, using the spacebar to shoot at the matrix of aliens above. The player must destroy all alien ships before they reach the bottom of the screen, and have three "lives" with which to do so. At times, there will be a bonus ship that flies across the top of the screen which will award more points. In between the cannon and the aliens are shields that deteriorate when shot by either the cannon or the aliens.

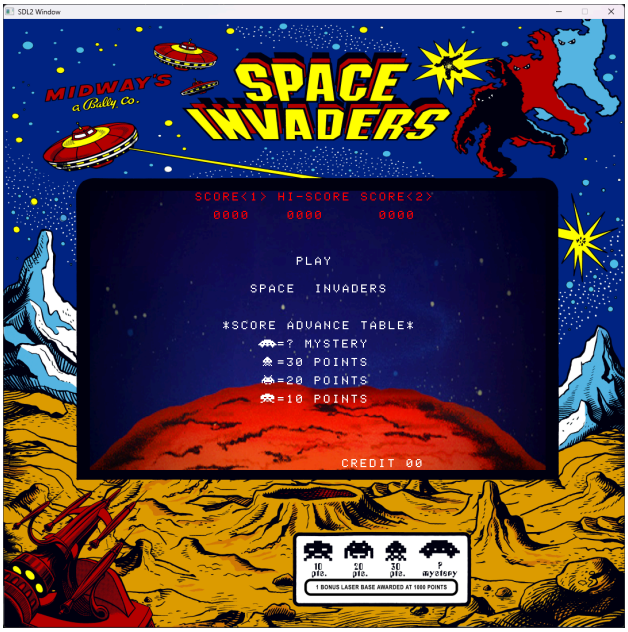


Screenshot of the game's menu screen.

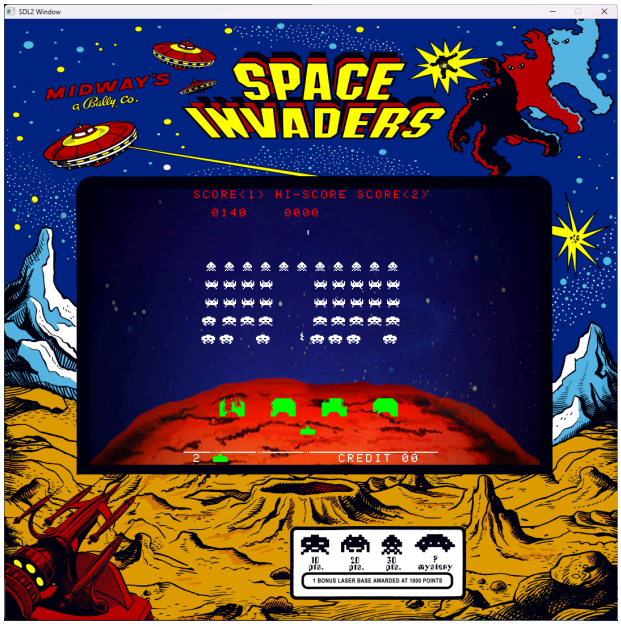


Screenshot of a game in progress.

Harris, Herman, Russ, Weber



Screenshot of the game's menu screen with arcade cabinet overlay. .



Screenshot of the game's menu screen with arcade cabinet overlay. .

Development Efforts vs. Project Plan

Everyone in our group worked on building the disassembler and then emulating each opcode. This development stage consumed most of our time and effort on this project.

First, we created a State struct to track the current state of the emulated CPU's registers, memory, flags, stack pointer, program counter, and I/O ports. The CPU has 7 registers, including the accumulator "A" and registers "B," "C," "D," "E," "H," and "L." We created a 65,536-byte array to represent the "memory," which stores both the game ROM and working RAM. The flags (sign, zero, aux_carry, parity, and carry), with three padding bits, are stored in a separate struct, which allows us to easily compress each "1" or "0" of the flags and padding into one byte when necessary. Our stack pointer and program counter are both 2 bytes to allow us to fully index the memory. We used an 8-byte array for the ports, which control input and output (display, sound, and controls).

Before emulating the opcodes, we created a disassembler. Our disassembler took an opcode as an input and printed out a human-readable description of the opcode being run, as well as any additional bytes used by the opcode.

After we printed our game file using the disassembler and confirmed that the correct opcodes were being read, we emulated the opcodes themselves. We used a large switch statement containing a value for each valid opcode (as well as many not used by Space Invaders 8080). Many of the opcodes performed an identical function on different registers, so we were able to write a few helper functions to speed up the process.

While working on the opcodes, we integrated CTest and began testing each individual opcode. We also compared our running game state against existing emulators' output. We diverged from our project plan during the testing stage, as we needed more time to test our implementation. This took away from our time for stretch goals, so we were not able to complete any. We had also originally planned to use the CUnit test library, but later realized that it would not integrate well with CMake and GitHub.

Once all the opcodes had been implemented and debugged, it was time to get the display up and running. Computerarcheology.com explains that the screen data is written to RAM starting from address 0x2400, and has 28 rows of 32 bytes where each bit represents a pixel on the screen. From this view, the screen is sideways and needs to be rotated counterclockwise by 90 degrees to be upright from the user's perspective.

The special hardware allowing for a multi-bit shifting also needed to be implemented before the sprites would get moved around the screen correctly.

Before we had SDL (Simple Mediadirect Layer) setup for our windowing system, Craig printed the video memory section to a PNG to confirm it was indeed populating as expected. With that completed, the program was then set to draw the screen (using SDL surfaces) directly from

video memory. This output a sideways black and white image, but was not updating as expected. Once it took over 20m to get the demo screen. Craig used debug statements that printed a description of which part of the game ROM the program was currently in (description provided by computerarcheology.com), and found that an internal delay counter was not updating correctly. The game uses full and midscreen interrupts, which regularly get sent 1/120th of a second based on the 60 Hz refresh rate of the screen, to decrement the delay counter. Since we had not implemented video interrupts, the delay counter was not functioning properly.

To send the interrupts at the right time, we needed to go back and account for the clock cycles of each instruction. We used a global counter variable to check when the correct number of cycles had elapsed before a video interrupt would have been sent. This is based on the 2MHz of the processor and the 60 Hz refresh rate. Once implemented, the game ran well, but too fast on power PCs.

After Craig implemented a timer to ensure that 1/120th of a second had fully elapsed since the last interrupt was sent, the game ran smoothly.

Once the game was running correctly, Damian rotated the screen counterclockwise by 90 degrees and applied color filters.

After the disassembler and emulator were finished, Julie implemented the controls. This task was planned to take two weeks, and it was accomplished within that time frame.

Julie began by studying SDL tutorials, including resources from lazyfoo.net and a website created by Justin Credible. Through these resources, she gained a solid understanding of SDL's event-driven model, which is essential for handling real-time input in games.

The controls are mapped to either port 1 or port 2, with each event having an associated bit. When a button event occurs, the state points to either port 1 or port 2, followed by a bit shift to execute the appropriate action. This method ensures that each key press or release accurately updates specific bits in the emulator's input state.

From the user's perspective, the gameplay closely mimics the arcade version, although the original arcade version used physical buttons and joysticks instead of a keyboard. This involved ensuring that the input state was updated in real-time without any lag, which was critical for maintaining an arcade-like experience. Julie achieved this by iteratively testing and refining the input handling code to make sure it responded accurately and promptly to player actions.

The final implementation of the controls allows players to interact with the emulator seamlessly. The game responds accurately to inputs, providing a smooth and engaging gameplay experience that honors the original arcade version. While Julie's work on the controls took longer than initially planned, the project remained on track overall, and the core functionality was delivered as expected.

In summary, implementing the controls ensured that the Space Invaders emulator offered an authentic and responsive gaming experience. The additional time spent on testing the emulator ensured its robustness and reliability, even though it meant sacrificing some of the planned stretch goals. This focus on quality and accuracy ultimately benefited the overall project, resulting in a well-functioning emulator that captures the essence of the original arcade game.

After the main portion of development, Nick began working on sound. The primary goal of the sound system was ensuring accurate sound effects; making sure that sounds start and stop when they are supposed to, sound as close as possible to those produced by the real hardware, and that sound effects are mixed together properly when more than one is played at a time. The original space invaders game used fairly complex dedicated sound chips that are difficult to emulate properly. We decided that the best way to handle this was to use recordings of the original sounds instead, and play these back as needed. This reduces the complexity considerably.

We used SDL2 as the primary library for interfacing with the audio hardware and loading in sound samples. This was fairly straightforward, but required that all of the sound samples have the same bitrate, frequency and number of channels, since this must be declared when initializing the connection to the audio device. A few of the samples had to be converted to ensure that this was working correctly. The original plan was to use SDL2 as the only library, handling audio, video, and controls. Unfortunately, there were a few issues. The first is that by default with SDL2, sound playback will be done on the main thread. This will block the main thread. To handle this, a threading API is provided, but it is difficult to interface with and greatly increases the complexity of handling audio. Additionally, SDL2 does not have a good way to mix more than one sound to a single output device, and does not recommend using this API for more than two samples at a time. Since Space Invaders will regularly use 3 or 4 effects at once, this will not work. To fix these issues, Nick integrated SDL2_mixer. SDL2_mixer is a supplementary library for SDL2 that fixes both of these issues. It provides an easy interface for defining as many mixing channels as are needed. It also does not block during playback, abstracting much of the complexity of handling threading.

Integrating the sound playback into the game is handled by intercepting the port outputs from the ROM. Space invaders will attempt to write out to ports 3 and 5 to call the sound effects. This is intercepted and passed to the play sound function to trigger the appropriate sound effect. Most of the sound effects are handled the same way, just play the sound once when the output goes high, but there are two sound effects that need to be handled differently. When the bonus UFO is on screen, a sound is played that must loop the whole time. SDL2_mixer makes this simple with the use of a built-in loop parameter. By passing -1, the sound will loop until stopped. The second is the shoot sound. This is supposed to only trigger once when the player fires, but Space Invaders holds the port output high the entire time the projectile is on screen, causing it to play multiple times. To fix this, Nick added a toggle to require this output to go low again before the sound can play again.

Development Tools/Libraries/Systems

We chose C as our programming language; we were the most familiar with C, and the extra features in C++ were not necessary to write a Space Invaders emulator.

We chose CLion as our IDE because it provided everything we needed to write our emulator. It is usable in Windows, Linux, and Mac environments, and we had members of our team using all three environments. It also integrates with GitHub, making our pushes and pulls much easier.

CMake was used to build our project since it made it easy to target multiple platforms, it came default in CLion and included the CTest functionality.

We used CTest as our testing library because we had already decided to use CMake, and our research indicated that CTest would pair well with both CMake and GitHub. None of us had worked with unit testing in C before, so we struggled with the initial setup. Craig and Damian integrated CTest with the project and got the initial tests working on GitHub, and then Nick and Craig later refined our test formatting once we had more experience with the library.

We chose SDL2 after moving on to hardware emulation. We had split to work on our separate assignments (display, controls, and sound) and all independently realized that SDL2 would support our needs. After this fact came up in one of our group meetings, we decided to formally integrate SDL2 into our project.

We had some initial issues getting SDL2 set up consistently across all our development environments. We were eventually able to resolve this issue once we realized that some of the library files had accidentally been excluded from the GitHub upload.

We have also been having issues building the latest version of our project on GitHub after integrating SDL2_mixer. Some additional dependencies introduced by SDL2_mixer to audio codec libraries are not properly discovered during the automated build process, causing it to fail. We have so far been unsuccessful in determining why this is the case.

Conclusion

Coding a Space Invaders emulator for Intel 8080 seemed daunting, but doable. We started this project believing we could have it running in a few weeks and be able to move onto stretch goals. However, the project proved to be more difficult than we imagined, so even though we created a playable game, we were not able to get it to the standard we had set for ourselves at the beginning of the term.

The future of the project may include additional features like alternate control inputs, alternate sound effects, or alternate color schemes. We also hope to resolve the dependencies issues and simplify the release and install process.

Resources

[1] "Space Invaders Emulator," *Justin Unterreiner*, Apr. 01, 2020. https://www.justin-credible.net/2020/03/31/space-invaders-emulator (accessed Jun. 05, 2024).

[2] "Lazy Foo' Productions - Beginning Game Programming v2.0," *lazyfoo.net*. https://lazyfoo.net/tutorials/SDL/index.php

[3] "Emulator 101 - Welcome," *www.emulator101.com*. http://www.emulator101.com/ (accessed Jun. 05, 2024).

[4] C. Cantrell, *Computer Archeology*. https://computerarcheology.com/ (accessed Jun. 04, 2024).

[5] *Intel 8080 Assembly Language Programming Manual,* Intel Corp., Santa Clara, California, USA, 1975.

[6] *Intel 8080 Microcomputer Systems User's Manual,* Intel Corp., Santa Clara, California, USA, 1975.

[7] K. Smith, Microcosm Associates, *8080/8085 CPU Diagnostic Version 1.0*, Simi Valley, CA, 1980

[8] C. Double, *Bluish Coder*. https://bluishcoder.co.nz/js8080/ (accessed Jun. 04, 2024).