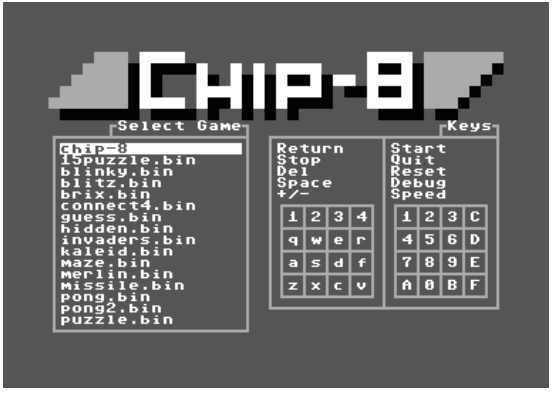

Chip-8
Select Game
Chip-8
15puzzle.bin
blinky.bin
blitz.bin
brix.bin
connect4.bin
guess.bin
hidden.bin
invaders.bin
kaleid.bin
maze.bin
merlin.bin
missile.bin
pong.bin
pong2.bin
puzzle.bin
Return
Stop
Del
Space
+/-
1 2 3 4
q w e r
a s d f
z x c v
Keys
Start
Quit
Reset
Debug
Speed
1 2 3 C
4 5 6 D
7 8 9 E
A 0 B F

What's CHIP-8?

- 1970's 8-bit virtual machine/programming language targeted at games programming. Originally implemented on kit-based 8-bit micros such as the mighty Telmac 1800 (2000 units sold mostly in Sweden and Finland)

- Revived in the late 80's due to an implementation on an HP48 graphing calculator

The plan

- Quick overview of the CHIP-8 architecture
- Dive into the Clojure implementation
- But first, a quick demo (so you know where we are heading....)

Why did I do this?

- Mostly to learn a bit more Clojure (so I don't have to look up the syntax of reduce every time ...)
- Really good project for a number of reasons:
  - Pretty small (which gives me a fighting chance of finishing..)
  - Once complete (or semi-complete) you gets lots of feedback as there already exists lots of games (Pong, Space Invaders, Pac-man, etc).
  - And it also has the nice side effect of teaching me a little more about 8-bits chips and games emulation.

CHIP-8 architecture (1)

- 4k of memory (interpreter in the lower 512 bytes)
- 16 8-bit data registers name V0 to VF
- 16-bit address register (I)
- Stack for subroutine return addresses (16 deep)
- 35 (2 byte) instructions

CHIP-8 architecture (2)

- Monochrome 64 x 32 pixel display
- Sound timer. 60 Hz. Counts down and beeps when non-zero
- Delay timer. 60 Hz. Counts down when non-zero
- Hex input keyboard (0x0 - 0xF)
- Sprites for 0x0 - 0xF pre-baked into interpreter memory address space

Emulator development algorithnm

10: Write the decoder + core fetch/decode/execute loop

20: Generate an empty implementation for each instruction (print the opcode and exit)

30: Play a game of your choosing until it crashes out on an unimplemented instruction

40: Implement the offending instruction (+ associated unit test)

50: Goto 30

Fetch/decode/execute

- Single machine state map represents the entire state of the machine (memory, registers, stack, etc.)
- Core fetch/decode/execute loop takes a machine state, and returns an updated machine state.
- Files: *machine_state.clj*, *core.clj*, *instructions.clj*

Threads and shared/mutable state

- 4 threads: *core*, *graphics*, *sound timer* and *delay timer*
    - Core -- atom[] --> Graphics (Screen updates to apply)
    - Core -- atom#{} --> Graphics (Keys currently pressed)
    - Core -- atom 0 --> Sound timer (Current value)
    - Core -- atom 0 --> Delay timer (Current value)
- Files: *main.clj*, *state.clj*

Graphics

- Using the Quil animation library
- All drawing done via single draw sprite instruction
- Files: *graphics.clj*
- *Demo*

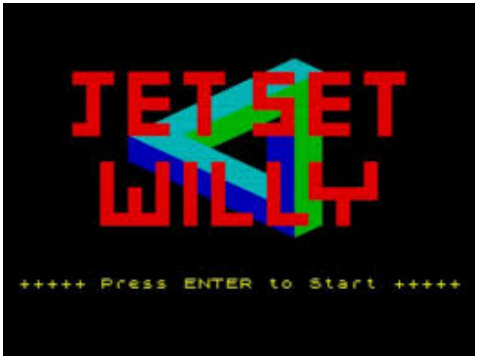
Sound

- Found it remarkably hard to make my Linux laptop make a sound!
- Tried overtone, which looks great, but still no sound.
- Ended up playing wav files using a command line utility (paplay on Linux, afplay on OSX)
- Files: *timer.clj*
- Demo*

Testing

- Why did I bother (seeing as this was a personal project)? Mostly to learn a little more about unit testing in Clojure. (More specifically, using core.test)
- Unit tests for each instruction (testing through the core/decoder). (Nice because the state of the chip can be passed in via the memory state, and you can simply check that it has been updated in the expected way)
- All other components tested manually by playing games (and looking at the instruction trace output).

What's next?



Links

- https://github.com/kristenjacobs/chip8-clj
- https://github.com/kristenjacobs/chip8-clj-slides
- CHIP-8 details + roms (games and demos)
    - https://en.wikipedia.org/wiki/CHIP-8
    - http://devernay.free.fr/hacks/chip8/C8TECH10.HTM
    - http://www.chip8.com/?page=84
    - http://www.zophar.net/pdroms/chip8/chip-8-games-pack.html