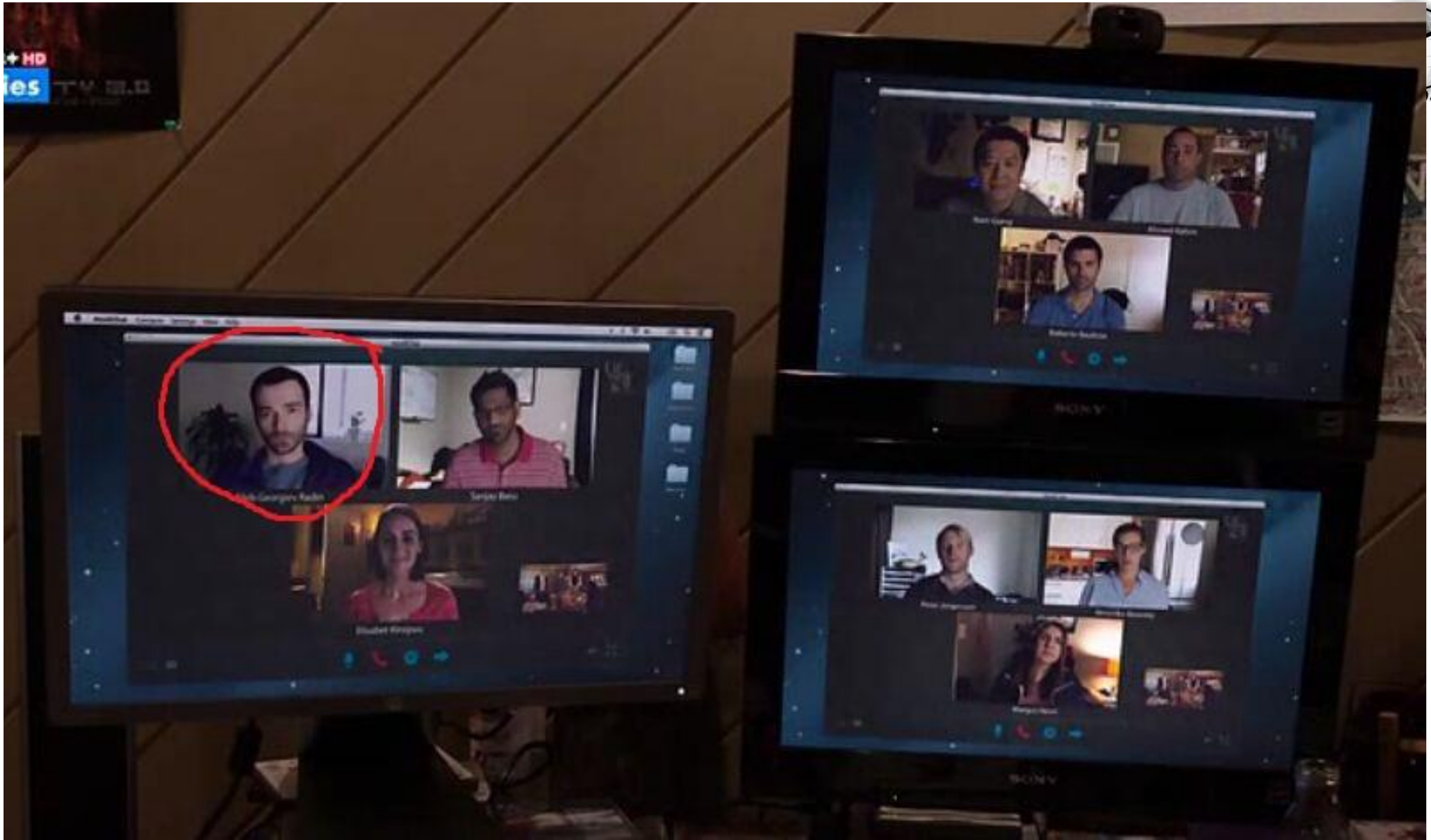
Introducción a la programación de emuladores de videojuegos

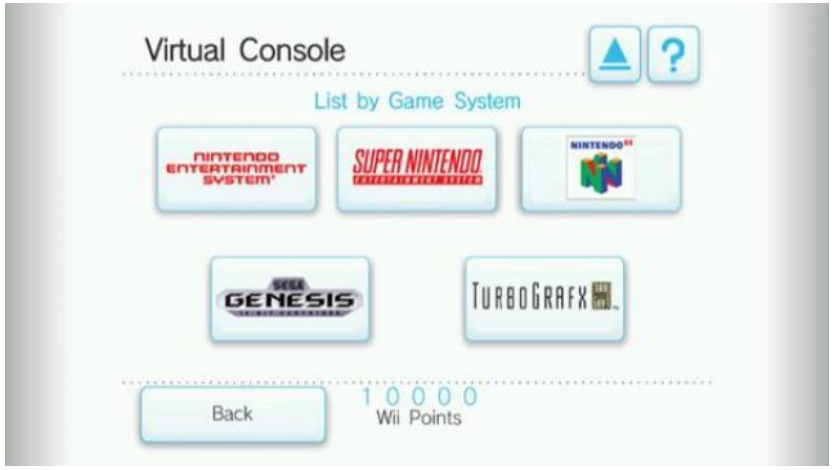
Y cómo sobrevivir en el intento

Uxío
Fuentefría
@Uxio0


LandRober72




KODI
RETROPIE


Virtual Console
List by Game System
NINTENDO ENTERTAINMENT SYSTEM
SUPER NINTENDO
GENESIS
TURBOGRAFX
Back
1 0 0 0 0
Wii Points


MVS
METAL SLUG
SUPER VEHICLE-001
METAL SLUG
INSERT COIN

Estado del arte

Libretro

- API para la creación de juegos y emuladores.
- Programas lo que llaman el “core”, con la implementación de vídeo/audio/entrada del emulador.
- Te despreocupas de programar diversos drivers de vídeo (OpenGL, Direct3D) o para diferentes APIs de sonido, entrada…
- Pasas a tener todo accesible desde un mismo frontend, te olvidas de programar una GUI.

¿Por qué me metí yo en esto?

- Salió al mercado en 1996 y aún no está bien emulada.
- Se orientó el desarrollo a velocidad frente a precisión.
- Se consiguió que la mayor parte de los juegos conocidos funcionaran y se paró el desarrollo.
- El principal emulador durante muchos años, Project64, no fue Open Source hasta las últimas versiones.
- Estos últimos años están apareciendo movimientos hacia una emulación “pixel perfect” (Cemu64 y paraLLEl)

High Level Emulation vs Low Level Emulation

- Low Level Emulatión (LLE)
  - Intenta imitar tal cual los recursos de la máquina, y responder al código tal como lo haría la máquina original.
  - Si se implementa correctamente, existirá una emulación perfecta, pero requiere muchos más recursos.
- High Level Emulation (HLE)
  - Permite tomar “atajos” para conseguir una velocidad mucho mejor a la hora de emular.
  - Pueden aparecer muchos “glitches” y problemas a la hora de emular, a parte de tener muchas veces que ir parcheando el emulador para cada programa que corre en él.

¿Por qué CHIP8?

- No necesitamos realmente emular un hardware, ya que CHIP8 es un intérprete (como la JVM, por ejemplo).
- Tiene muy pocas instrucciones y poco tediosas de implementar.
- El audio se reduce a zumbidos.
- El vídeo consiste en un conjunto de píxeles monocromos.
- Muy bien documentada.
- Otra plataforma muy interesante aunque más compleja para empezar podría haber sido la GameBoy.

¿Por dónde empezamos?

- Lo primero será leer la referencia técnica de la consola. En el caso de CHIP8:
  - https://es.wikipedia.org/wiki/CHIP-8
  - http://devernay.free.fr/hacks/chip8/C8TECH10.HTM
- Decidimos un lenguaje de programación para el emulador. Para emuladores de máquinas con más requisitos, es importante elegir un lenguaje “rápido” (C, C++, Go, ensamblador…). En este caso no es necesario.
- Descargamos alguna ROM para ir probando, en el caso de CHIP8 hay varias de dominio público.
  - https://www.zophar.net/pdroms/chip8.html

Lectura de instrucciones de la ROM

- Una buena forma de empezar es ser capaces de leer instrucciones de la ROM.
- Según la especificación, todas las instrucciones tienen 2 bytes de longitud.
- Definimos un *array* de *bytes*, abrimos la ROM como fichero binario, y vamos almacenando *byte* por *byte* en el *array*.
- Una “optimización” que intenté cuando empezaba a programar el emulador es usar un array de enteros de 16 bits.

Relacionar las instrucciones con opcodes

- Tenemos que ir iterando el array que hemos leído, cogiendo cada elemento N y N+1, para formar la instrucción completa.
- A partir de ahí, empezaremos a casar las instrucciones (*opcodes*) de la ROM con las de la especificación.
  - Por ejemplo 00E0 -> Limpiar pantalla, o 00EE -> Volver de una subrutina.
- Para ello usaremos máscaras de bits.
- También podemos empezar ya a extraer los argumentos de algunas instrucciones
  - Por ejemplo 1NNN -> Salta a la dirección NNN. Podemos extraer el NNN.
- Una vez hecho esto tendremos buena parte del trabajo realizado.

Programar la CPU

- Usando la referencia, definimos los elementos que necesita la CPU para funcionar:
  - Memoria: 4KB, desde 0x0 a 0xFFF (4095), con 1 byte en cada posición.
    - Los programas empiezan en 0x200(512)
  - Registros:
    - 16 registros de 8 bits cada uno, desde VA a VF.
    - Registro I, de 16 bits. Se usa generalmente para almacenar posiciones de memoria.
    - 2 registros especiales para temporizadores de sonido y retardo, de 8 bits cada uno.
    - Contador del programa (PC) de 16 bits.
    - Pila: Array de 16 elementos de 16 bits. Puntero a la pila de 8 bits.
  - Pantalla: 64x32 monocroma
  - Teclado: Tiene un teclado hexadecimal (0-9 y A-F)

Programar la CPU

```
type Chip8Engine struct {
    memory      [0xFFF]byte
    register    [16]byte
    iRegister   uint16
    stack       [16]uint16
    delayTimer  byte
    soundTimer  byte
    pc          uint16
    screen      [64][32]bool
    stackPointer byte
    sdlWindow   SDLWindow
}
```

Programar salida de vídeo

- En un principio de forma abstracta usar un array de booleanos no parece mala idea.
- Una vez estaba todo funcionando decidí usar OpenGL.
- Simplemente se pinta cada pixel del array en la pantalla de OpenGL.
- No es lo más óptimo, pero funciona bien y es “pixel perfect”

Programar salida de vídeo (fuentes)

- Un tema pendiente serían las fuentes.
- CHIP8 soporta 16 caracteres, A-F y 0-9.
- Se definen como máscaras de bits, y se almacenan en la memoria.
- Con la instrucción **FX29** (siendo X el valor en hexadecimal del caracter) se coloca el **registro I** apuntando al comienzo del *sprite* preparándolo para ser dibujado con **Dxyn**.

Programar el teclado

- OpenGL da soporte para comprobar las pulsaciones de teclado en la pantalla.
- En mi caso lo uso de forma bastante rudimentaria.

¿Por dónde seguir?

- Emular GameBoy es un buen paso.
  - http://marc.rawer.de/Gameboy/Docs/GBCPUman.pdf
- Seguir la comunidad de emulation y emudev en Reddit.
- Animarse a colaborar en un proyecto grande, como Cemu64.