Iñatrix Overflow

Memoria del proyecto.

Aingeru García
Martín Jiménez
Joseba Gómez

Índice

1. Introducción

Tras el estudio en la asignatura de conceptos a tan bajo nivel como lo son el lenguaje máquina y/o ensamblador, hemos pasado a adentrarnos en el mundo de los sistemas de entrada y salida. Un mundo en el que tanto los controladores de periféricos como las diversas posibilidades de gestión de memoria cobran un papel muy importante.

Para trabajar el aspecto práctico, se nos ha planteado la opción de desarrollar un videojuego a, relativamente, bajo nivel. Utilizando el lenguaje de programación C y libnds; una librería que nos permita abstraernos de las complejidades y entresijos de una máquina como la Nintendo DS.

La asignatura de por sí es muy interesante, ya que nos ayuda a seguir adentrándonos en la rama de la arquitectura de computadores, pero si le añadimos un edulcorante tan fascinante como lo es el programar un videojuego, hace que se disfrute mucho de poner en práctica lo aprendido durante el curso, ya que en todo momento se nos ha hecho mucho hincapié enla importancia de aplicar debidamente lo trabajado durante en la asignatura.

En ésta memoria vamos a exponer en detalle todo el proceso de desarrollo, desde el diseño del prototipo inicial hasta el momento de liberación del código en el repositorio de **Github** [1].

Además, explicaremos el eje principal que ha guiado el proceso de desarrollo: el motor gráfico que hemos ido creando poco a poco; un soporte subyacente que nos ha permitido agilizar, generalizar y simplificar el diseño de un videojuego para la Nintendo DS.

1.1. Motivación

Tras un profundo estudio de la librería que íbamos a utilizar (**libnds**[2]) y hacer unas pruebas con la plantilla base que se nos proporcionó, decidimos animarnos a crear un juego un poco diferente y hacer homenaje a un profesor recién jubilado, al cual hemos cogido mucho cariño durante el breve espacio de tiempo en el que le hemos conocido: **Iñaki Morlan**. Ya que era el profesor de la asignatura de Principios de Sistemas Digitales y, de entre las muchas bromas y/o frases célebres que citaba, comentaba distintos pasajes de la maravillosa película Matrix, así que, juntando todos las ideas no vimos otra posibilidad más que titularlo: **Iñatrix Overflow**.

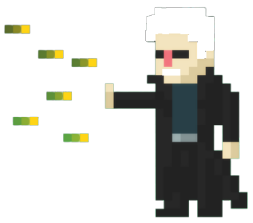

Figura 1: Iñatrix, "The One".

Se nos ocurrió que podríamos jugar con matrices y, basándonos en el estilo de juego retro-arcade, que el jugador tuviese una manera de ir ganando puntos, mientras algo va en su contra y se va complicando comforme va transcurriendo el tiempo.

La idea desde un primer momento era la de centrarnos primordialmente en el diseño de las mecánicas, estructuras y gestión de las matrices así como su asociación con los gráficos/sprites, en lugar de orientarlo a algo que puidera tener algo más de impacto visual.

Observamos que a través de esta idea inicial, podríamos expresar perfectamente los requisitos de la asignatura en relación a los diferentes métodos y peculiaridades de los sistemas de E/S; por encuesta, por interrupción, utilización del periférico de la pantalla táctil, configuración de controladores... y además, aprender mucho a lo largo de todo el proceso.

1.2. El mini-motor

Con lo que, con todo esto presente, nos percatamos de que algunos de los sistemas existentes, como por ejemplo el de la gestión de sprites, podrían cambiarse con objeto de poder automatizar tareas y generalizar, así como crear estructuras que realizaran una asociación entre sprites-gfx y que a su vez estuviesen accesibles desde diferentes partes del código. En el momento de empezar a solidificar las ideas y darles forma; sentimos la necesidad de poder crear eventos de manera secuencial, como si de un guión se tratara. De ahí nació el gestor de eventos.

Consecuentemente, cada vez que aparecía una nueva necesidad implementábamos un módulo que tratase sus acciones de una manera genérica. Habiendo trabajado un poco con diferentes motores gráficos (a mucho más alto nivel que el par `C - libnds`), pensamos en intentar emular muy tímidamente esta clase de sistemas. Con lo que, nos centramos en desarrollar una base muy modularizada, que permitiese más adelante programar muy rápida y eficazmente cosas adicionales que se nos ocurriesen, habiendo creado sin darnos cuenta un `mini-motor` o API para la NDS.

Diseñando así un motor que se encarga de los sistemas genéricos y que da soporte a las funcionalidades básicas de un videojuego; la creación del mismo se simplifica y se agiliza de una manera exponencial; mientras el código queda muy organizado al abstraer los diferentes conceptos; **cada cosa en su sitio**. Esto propicia que el cambiar partes del juego, arreglar bugs y demás fases del desarrollo, sean muchísimo más rápidas. El motor también es de gran ayuda a la hora de declarar nuevos elementos (sprites, gfx...etc), ya que en lugar de existir múltiples funciones, con una sola línea de codigo es sufienciente.

Claro está, si bien nos ha sido de gran ayuda a nosotros y a pesar de creer que pueda serlo también para otras personas en un futuro, es **limitado y requiere muchas mejoras**. Durante éste verano tenemos planeado separar el contenido de **Iñatrix Overflow** y publicar en otro repositorio como código libre u Open Source el 'esqueleto' de éste motor. Para que cualquier persona y/o futuros alumn@s puedan tanto utilizarlo, como colaborar en su desarrollo si lo desean. El repositorio de código libre está listo y es donde será publicado [3].

Ahora sí,

```
>_ {enterTheMatrix}
```

2. Iñatrix Overflow - El juego

Figura 2: Iñatrix Overflow

Iñatrix es el super héroe que colabora analizando el sistema que tiene la función de 'realidad' en éste mundo, gracias a un exhaustivo análisis es capaz de proporcionar estabilidad con su superpoder de eliminación de `overflows` en la estructura de **Matrix**. La realidad que rige éste universo, está en constante decadencia y la única manera de conseguir que no ocurra un `fatal system failure` es eliminando los bits incongruentes de su estructura intrínseca.

Para ello, Iñatrix será capaz de duplicarse e ir destruyendo desde lo más profundo de la raíz matricial bidimensional los bits que hacen de centro del `overflow`.

Pero Iñatrix todopoderoso no está solo; nosotros le ayudaremos a identificar los bits infectos. El tiempo juega en nuestra contra; tendremos que poner en marcha nuestros cerebros para calcular lo más rápidamente posible y poder así salvar la realidad.

Trabajo en equipo, pero,

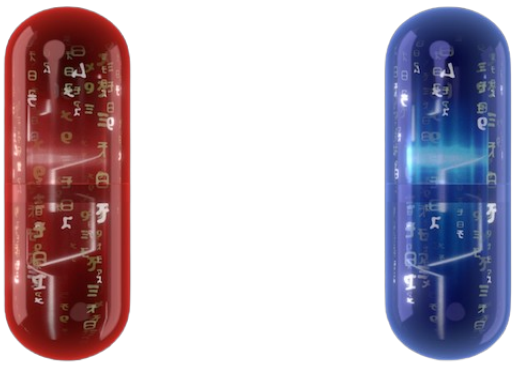

Figura 3: ¿Cual es tu decisión?

2.1. Lore

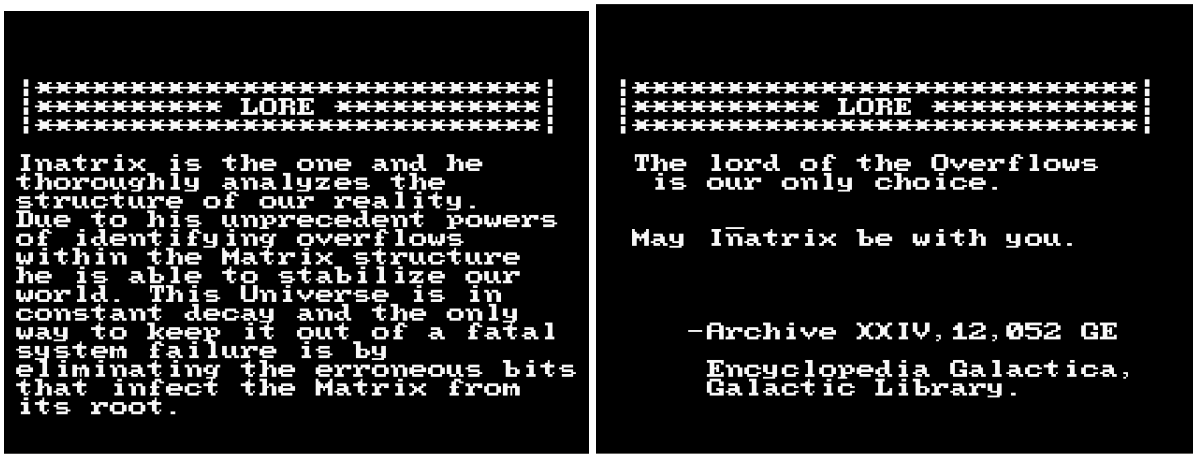

Figura 4: Lore

Ésta pantallas a las cuales se accede a través del menú principal, muestran brevemente al jugador un poco de Lore del mundo de **Iñatrix Overflow**.

2.2. Mecánicas/Gameplay

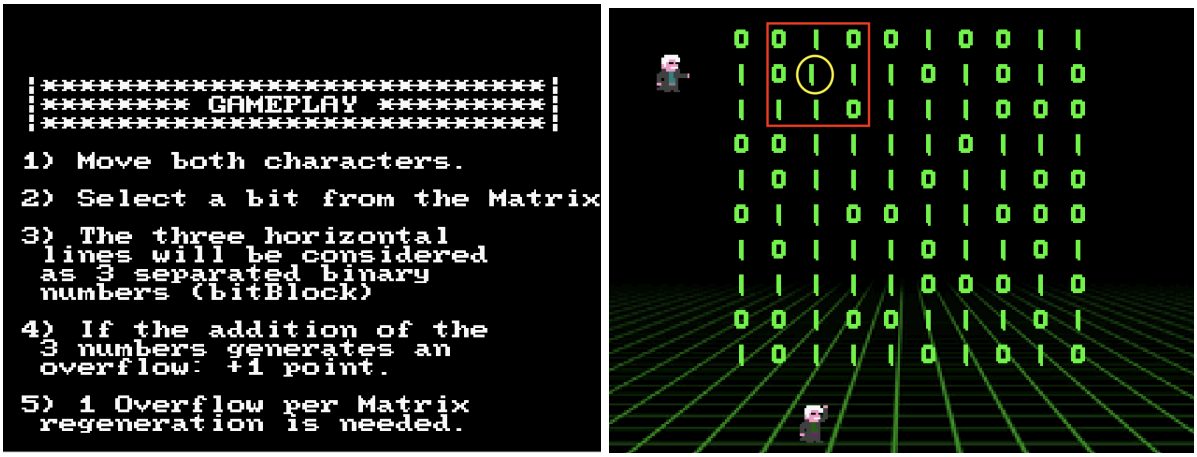

Figura 5: Gameplay

La idea base radica en la existencia una matriz y un Iñatrix duplicado que se deberá de ir desplazando a través de de los dos ejes X e Y. El objetivo principal consistirá en ir **seleccionando bits o elementos pivote**, de manera que se generen `overflows`. Es decir, moviendo ambos Iñatrix, se debe de ir buscando que el bit que sea intersección de entre las posiciones de los dos personajes, al obtener el bloque de `9 bits` de alrededor del mismo, produzca `overflow` cuando se sumen los 3 números binarios que se generan (1 por cada fila). Si se produce `overflow`, este bloque de bits de romperá y sumará un punto de `Overflow`.

Cuando una de las dos versiones de nuestro personaje sea desplazada, el elemento pivote que sea pasivamente seleccionado (bit que corresponda con el vector posición asociado con las posiciones de Iñatrix), tendrá un efecto de "shake". Este efecto permanecerá constante mientras los personajes sigan en esa posición; con objeto de que el jugador pueda apreciar cual será su selección si pulsa el botón correspondiente.

La cantidad que los 3 números binarios han de sumar para producir `overflows`, vendrá determinada por el modo o dificultad elegida por el jugador, sobre lo cual hablaremos más adelante.

Otra de las mecánicas base es el hecho de que la matriz se destruirá cada ciertos segundos e intentará regenerar posteriormente (tiempo también determinado por la dificultad de la partida). Si el jugador ha conseguido al menos un `overflow` desde la última destrucción/regeneración, entonces se considerará que ha estabilizado la estructura de la matriz y podrá seguir jugando. En caso contrario, si no ha conseguido ningún `overflows` y el timer de la destrucción llega a 0, existirá un `system failure` y el juego terminará, **Game Over**.

El jugador irá acumulando `Overflows`, y en caso de no tenga ninguno (un total de 0); o bien porque la partida acaba de empezar o bien porque, a pesar de haber conseguido varios, ha fallado un número igual de veces - la partida terminará. **Game Over**

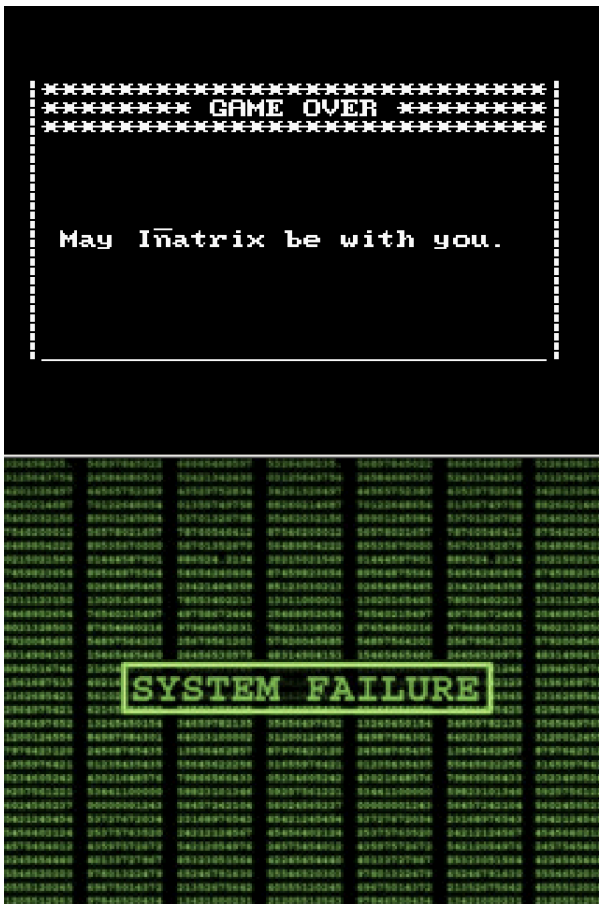

Figura 6: Game Over

2.3. Controles y dificultad.

Figura 7: Iñatrix Overflow

- **Menú principal**:

  Derecha: Muestra controles.

  Izquierda: Explica brevemente las mecánicas del juego.

  Arriba: Un poco de Lore o trasfondo del juego.

  Start: Comienza el juego.

- **Juego**:

  Arriba/Abajo: Desplazar Iñatrix Y (vertical) a través del eje Y.

  Izquierda/Derecha: Desplazar Iñatrix X (vertical) a través del eje X.

  A: Selección del bit pivot correspondiente al bloque de bits.

  SELECT / B: Surrender, interferencia en Matrix que produce que el juego termine.

Nota: si Iñatrix está en la primera/ultima posición, el movimiento se invertirá y creará un efecto rebote, **colisionando** con la parte final de la matriz.

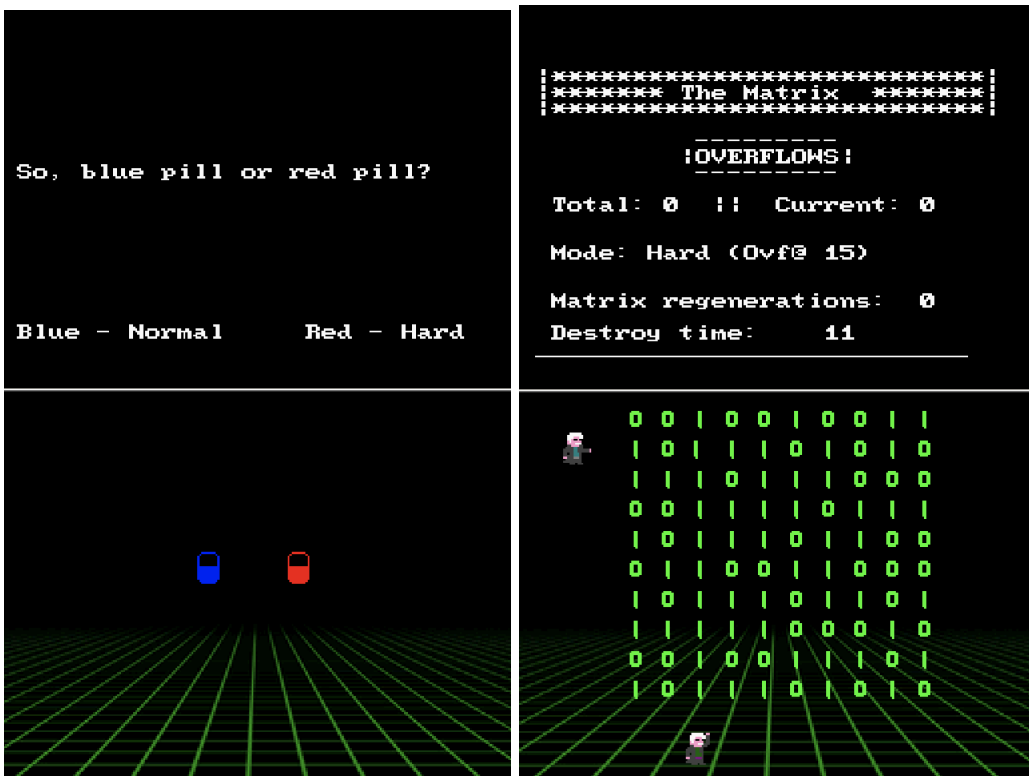

Figura 8: Game

Hard Mode/Normal mode

Se mostrará al usuario la posibilidad de elegir de entre dos cápsulas (como en la película), de manera que se establecerá la dificultad del juego:

- **Cápsula azul**: Normal mode - Entre los 3 números binarios se necesitará llegar al menos a 9 para que exista `overflow` y el timer para la Matrix regeneration será de 25 segundos.

- **Cápsula roja**: Hard mode - Entre los 3 números binarios se necesitará llegar al menos a 15 para que exista `overflow` y el timer para la Matrix regeneration será de 15 segundos.

2.4. UI Info

Existen diferentes puntuaciones y estadísticas que se irán logrando en el juego, se mostrarán en la pantalla de arriba, son:

- **Overflows**: Se incrementa cuando hay un `overflow` y decrementa cuando hay un fail.
- **Total**: Número total de `overflows` desde el inicio de la partida.
- **Current**: Número de `overflows` desde la última regeneración de la Matriz (o inicio de partida cuando aún no han habido regeneraciones aún).
- **Matrix regerations**: número de regeneraciones en ésta partida.
- **Destroy time**: tiempo en segundos hasta la próxima destrucción.

2.5. Guiños y referencias.

Inicialmente queríamos haber introducido una plétora de frases y guiños hacia Iñaki, pero por diferentes motivos hemos terminado incluyendo únicamente dos citas las cuales nos parecen bastante graciosas y apropiadas.

Cuando el jugador selecciona un bit y falla (es decir, **no** produce *Overflow*), se mostrará en la pantalla de arriba un mensaje aleatorio de entre una de las frases que Iñaki solía decir en clase en diferentes situaciones.

Por supuesto, no podíamos desarrollar un juego sin hacer algunas referencias, por leves que sean, a algunas de las obras maestras que nos apasionan. Falta el Señor de los Anillos, entre muchas otras. Pero como no podemos introducir todo lo que nos gustaría, nos quedamos con unas pocas.

No mencionamos cuales, ¡mejor que sean descubiertas!

- **Matrix** (sólo a la primera película, ¡las demás no se lo merecen!).
- Referencia a **la Fundación** de Isaac Asimov.
- Referencia a **Star Wars**.
- Referencia a la saga de juegos **Dark Souls** [5].

3. Desarrollo

Como se ha mencionado con anterioridad, no hemos utilizado ninguna librería a parte de la propia `libnds`. Todos los sistemas y módulos utilizado para facilitar el posterior desarrollo del contenido del juego en sí, han sido diseñados por nosotros mismos desde 0. Es posible visualizar el historial y progreso de cada uno de los cambios del proyecto en las diferentes branches que hemos creado, desde la plantilla base que se nos proporcionó, en el enlace del repositorio del juego que vinculamos en el apartado de la memoria correspondiente [1].

3.1. Estructura y modelo

En el directorio del proyecto, podemos observar la siguiente estructura:

```
basajaun@macbook ~/Workspace/Projects/Inatrix_Overflow  master tree -I 'Doxygen|gfx|base_template'
.
├── CmakeLists.txt
├── Inatrix_Overflow.arm9
├── Inatrix_Overflow.elf
├── Inatrix_Overflow.nds
├── LICENSE
├── Makefile
├── README.md
├── audio
│   └── audio.wav
├── build
├── doc
│   ├── latex
│   ├── memoria
│   │   └── Memoria.pdf
│   └── prototipo
│       └── Iñatrix Overflow.pdf
├── include
│   ├── audioMgr.h
│   ├── backgrounds.h
│   ├── consoleUI.h
│   ├── controllers.h
│   ├── defines.h
│   ├── engine.h
│   ├── eventMgr.h
│   ├── game.h
│   ├── gfxInfo.h
│   ├── input.h
│   ├── matrix.h
│   ├── movementMgr.h
│   ├── objectMgr.h
│   ├── sprites.h
│   ├── timer.h
│   └── utils.h
├── resources
│   ├── Game.png
│   ├── InatrixWhite.png
│   └── StartingMenu.png
└── source
    ├── audioMgr.c
    ├── backgrounds.c
    ├── consoleUI.c
    ├── controllers.c
    ├── engine.c
    ├── eventMgr.c
    ├── game.c
    ├── gfxInfo.c
    ├── input.c
    ├── main.c
    ├── matrix.c
    ├── movementMgr.c
    ├── objectMgr.c
    ├── sprites.c
    ├── timer.c
    └── utils.c
```

Figura 9: Estructura de directorio

3.2. Nomenclatura

Para la gestión de todos los recursos, tenemos diferentes estructuras que nos permiten el fácil acceso a los mismos. Hemos realizado una minuciosa gestión de la memoria dinámica (sin *memory leaks*) con cada uno de los diferentes módulos.

En cuanto a la manera de programar y estructurar la información, hemos intentado emular (muy) tímidamente el paradigma de la orientación a objetos, con lo que para ello nos hemos visto obligados a utilizar el siguiente formato para la nomenclatura de módulos/funciones:

`módulo.h`:

```
void modulo_funcionParaAlgo();
```

Dando así lugar a nombres de funciones más largos de lo que nos gustarían, y de lo que dictan las convenciones del lenguaje C y/o mejores prácticas. No estamos cómodos con ello, pero pese a su longitud creemos que, al menos en este proyecto es algo positivo; hace más bien que mal sobre todo de cara al programador ajeno al proyecto.

Como ejemplo de porqué creemos que, pese a su longitud, el nombre de funciones que hemos utilizado es relativamente apropiado:

> Entiéndase como **módulo** el sistema que viene dado por su cabecera '.h' y respectiva implementación de su código fuente en el fichero '.c'

1. Miramos el código de la función de inicialización de la matriz.

```
void matrix_initSystem(){
    gfxInfo_initMatrix(baseMatrix[0], MATRIX_SIZE);
    gfxInfo_initMatrix(baseBitBlockBuffer[0], BITBLOCK_SIZE);
    pivot = malloc(sizeof(MatrixPivot));
}
```

2. Observamos que para poder inicializarla llama a:

```
gfxInfo_initMatrix(baseMatrix[0], MATRIX_SIZE);
```

3. Sabemos que el módulo `gfxInfo` tendrá una función llamada así y realizará sus operaciones con los parámetros pasados.

Creemos que dada esta estructura, el proyecto puede resultar relativamente sencillo y claro de leer a ojos de programador ajeno (lo cual, a veces puede resultar en una ardua tarea). Nuestra intención ha sido que sea algo intuitivo para cualquiera que se anime a adentrarse en ello.

3.3. Documentación

Hemos trabajado mucho en éste proyecto con lo que, existe una gran cantidad de código y sistemas llegando a las varias miles de líneas de código. Si explicáramos todo aquí, consideramos que la memoria se centraría más otros aspectos y abstraería del contexto. Con lo que hemos documentado el código exhaustivamente para que se pueda comprender cada detalle implementado, de manera que **Iñatrix Overflow** ha quedado enteramente documentado.

De ésta manera podemos explicar en ésta memoria únicamente las relaciones entre los diferentes sistemas o módulos, para poder comprender la estructura o jerarquía de manera global. Si se desea profundizar en el funcionamiento de un elemento en particular, creemos que el código ofrece la posibilidad de guiar y describir el proyecto en su totalidad.

Para ello, hemos preparado lo siguiente:

- 1) Se han escrito comentarios descriptivos a lo largo de todo el proyecto, anotados con formato **Doxygen** [4]:

  1.1) Todo módulo/archivo contiene una descripción al inicio sobre su funcionalidad.

  1.2) Los structs han sido documentados en las cabeceras (de sus propios módulos o "defines.h"); las funciones en sus fuentes.

- 2) Documentación generada con Doxygen a partir de las anotaciones. Abrir: */doc/Doxygen/**index.html*** con cualquier browser, de manera que se puede navegar gracias a una interfaz web a través de todos los elementos del código (structs, funciones, archivos...etc).

3.4. Módulos

Vamos a detallar los módulos creados por nosotros mismos o re-diseñados enteramente, para poder exponer las diferentes relaciones entre los mismos y justificar el porqué del modelo. De ésta manera, en conjunción con la documentación del código, se podrá comprender el funcionamiento en detalle.

- **EventMgr**: Con la necesidad de poder crear eventos secuenciales (no relacionados a la programación orientada a eventos que se ha implementado a través de los sistemas de Entrada/Salida), decidimos diseñar una especie de lista donde se fuesen acumulando todos los eventos pendientes. Es un sistema centrado en la lógica y estructura del contenido del juego, que permite de una manera natural, organizada y muy sencilla, acumular eventos en el tiempo.

  Cada evento tendrá asignado un identificador y un tiempo de ejecución, que será el momento en el que deberá de procesarse. Cuando el sistema gestor de eventos detecte que ha llegado el momento (a través del timer, el cual es llamado por el handler / rutina de atención), lo ejecutará, borrará de la lista y reorganizará la misma. Para ello, se comprobará si el momento de ejecución de un determinado evento es igual o menor al momento actual.

  Este sistema nos ha permitido crear el contenido del juego de manera muy rápida, y ha hecho muy sencilla la gestión de diferentes estados como la intro, o incluso la programación de eventos provocados por una determinada acción en secuencias cíclicas, como cuando el jugador selecciona un bit, que ocurren múltiples cosas a la vez.

Veamos un ejemplo de lo sencillo e intuitivo que sería su utilización por parte de un programador usuario:

```
/*
**********************
**********************
******* INTRO *******
**********************
**********************
*/
case EVENT_INTRO_PRE_START:
    background_setBackground(BG_MATRIX);
    eventMgr_ScheduleEvent(EVENT_CLEAR_CONSOLE, NO_WAIT);
    eventMgr_ScheduleEvent(EVENT_INTRO_START, IN_4_SECONDS);
    break;
case EVENT_INTRO_START:
    iprintf("\x1b[09;10H _");
    iprintf("\x1b[10;00H Wake up, Inatrix...");
    background_setBackground(BG_MATRIX_INATRIX);
    eventMgr_ScheduleEvent(EVENT_CLEAR_CONSOLE, IN_3_SECONDS);
    eventMgr_ScheduleEvent(EVENT_INTRO_TEXT1, IN_4_SECONDS);
    break;
case EVENT_INTRO_TEXT1:
    iprintf("\x1b[10;00H The Matrix has you...");
    eventMgr_ScheduleEvent(EVENT_CLEAR_CONSOLE, IN_3_SECONDS);
    eventMgr_ScheduleEvent(EVENT_INTRO_TEXT2, IN_5_SECONDS);
    break;
case EVENT_INTRO_TEXT2:
    iprintf("\x1b[10;00H Follow the white rabbit.");
    background_setBackground(BG_RABBIT);
    eventMgr_ScheduleEvent(EVENT_INTRO_TEXT3, IN_5_SECONDS);
    eventMgr_ScheduleEvent(EVENT_INTRO_RABBIT2, IN_3_SECONDS);
    eventMgr_ScheduleEvent(EVENT_CLEAR_CONSOLE, IN_3_SECONDS);
    break;
case EVENT_INTRO_RABBIT2:
    background_setBackground(BG_RABBIT2);
    break;
```

- **GfxInfo**: Este módulo se encarga de organizar y de gestionar los GFX, asociándolos y sincronizándolos con los sprites.

  Básicamente crea, inicializa y asocia dichos GFX a los sprites en memoría dinámica, así como escribe en el banco de memoria main de la NDS todos los bitmaps asignados que inicialmente estarán ocultos.

  De manera adicional, inicializa la matriz de manera separada al resto de sprites. Como con los demás GFX, se encarga de reservar memoria en el banco de memoria de la NDS y, una vez obtenido el puntero al sprite, se vincula a su posición correspondiente en la matriz.

  `gfxInfo` trabaja en conjunción con los módulos de `sprites` y `matrix`, para llevar a cabo una correcta preparación de todo los GFX/Sprites en Memoria y dejar los structs que los representan

perfectamente accesibles en los arrays correspondientes.

El sistema de gfxInfo, en comunicación con el módulo de sprites, permite que con una linea de código se puedan crear sprites, abstrayendo de su funcionamiento intrínseco al programador. Veamos otro ejemplo de lo sencillo e intuitivo que sería su utilización por parte de un programador usuario, donde `gfxInfo_setGfx` se encarga de asociar a bitmaps, inicializar structs...etc. y el primer parámetro `GFX_*` no es más que un índice:

```
/**
 * @brief Función que permite la adición de nuevos gráficos.
 */
void gfxInfo_init(){

    /* CAPSULES */
    gfxInfo_setGfx(GFX_CAPSULE_BLUE, SpriteSize_16x16);
    gfxInfo_setGfx(GFX_CAPSULE_RED, SpriteSize_16x16);

    /* INATRIX */
    gfxInfo_setGfx(GFX_INATRIX_X, SpriteSize_16x16);
    gfxInfo_setGfx(GFX_INATRIX_Y, SpriteSize_16x16);

    /* INATRIX SPELLS */
    gfxInfo_setGfx(GFX_INATRIX_SPELL_X, SpriteSize_16x16);

}
```

- **Sprites**:

  Sistema encargado de gestionar la memoria de la NDS y, en conjunción con gfxInfo, crear una asociación de los gráficos y dejarlos bien estructurados y accesibles para el programador.

  El orden de escritura de los bitmaps en el banco de memoria de la NDS es el siguiente:

  ```
  GFX_SPRITES -> MATRIX_SPRITES -> BITBLOCK_SPRITES
  ```

  GFX_SPRITES: Se reservará espacio (y escribirán los bitmaps posteriormente) para el número de sprites deseados por el programador.

  MATRIX_SPRITES: Depende del tamaño de la matriz, en nuestro caso $(10 \cdot 10) = 100$ sprites.

  BITBLOCK_SPRITES: Depende del tamaño del bitblock, en este caso $(3 \cdot 3) = 9$ sprites.

Nosotros hemos utilizado $5 + (10 \cdot 10) + (3 \cdot 3) = 114$ sprites (teniendo en cuenta el máximo de **128** establecido por la NDS).

Una vez definidio el GFX como se ha mostrado en la descripción de **gfxInfo**, el usuario podrá acceder al sprite alojado en memoria dinámica con tan sólo:

```
sprites[GFX_INATRIX_X]->spriteEntry->x += 1;
```

Estamos evitando entrar en este tipo de detalles, pero a modo de ilustración, veamos qué constituye el tipo `Sprite` que hemos definido, ya que el array sprites es un array de punteros a struct de éste tipo:

```
/**
 * @struct Sprite
 * @brief Sirve para asociar un sprite a los GFX.
 * @var index: índice del sprite.
 * @var spriteEntry: puntero a SpriteEntry (libnds).
 *                   Sirve para obtener/modificar vector posición.
 * @var speed: velocidad de movimiento.
 * @var gfx: Puntero a @struct GfxData asociado.
 */
typedef struct {
    uint8 index;
    SpriteEntry* spriteEntry;
    int speed;
    GfxData* gfx;
} Sprite;
```

- **Matrix**: Gestión de la matriz y sus efectos. Al ser uno de los sistemas principales del proyecto, hemos considerado hacerlo de manera totalmente separada. Este sistema se encarga de multitud de operaciones, entre ellas la de permutar y la de regenerar tanto la matriz como el bitBlock, siendo éste último el bloque de bits 9 que rodean al pivote seleccionado por el jugador.

  Es uno de los módulos más importantes - pero no trabaja sólo, como se ha mencionado previamente, se comunica con los módulos **gfxInfo y sprites**, aunque el **eventMgr** también se encarga de invocar las diferentes funcionalidades de matrix.

  La matrix es un array bidimensional de punteros a **structs** del tipo `MatrixElement`, que contienen información acerca de los sprites, gfx y dígito binario que representan. De nuevo, para ilustrar la idea, expongamos brevemente:

```
/**
 * @struct MatrixElement
 * @brief La/el matriz/bitblock están compuestos por punteros a este tipo de
 *          struct.
 * @var sprite: dirección de memoria del struct @struct Sprite.
 * @var bit: valor del digito binario, del tipo @typedef Binary.
 */
typedef struct{
    Sprite* sprite;
    Binary bit;
} MatrixElement;
```

Veamos en qué consisten las **regeneraciones**:

Figura 10: Matrix/Bitblock regen

- Matrix Regen: Cuando se regenera la matriz, se procesa el efecto de destrucción/caída y después se oculta la misma, para posteriormente cambiarle de base (de 2D a 1D) y permutar las direcciones de memoria con el algoritmo Fisher-Yates (también conocido como 'Knuth shuffle'), utilizando su versión modernizada en 1964 por Richard Durstenfeld (explicado en detalle en el código). Después de este proceso, se vuelve a mostrar la 'nueva' matriz al jugador.

- Bitblock Regen: Cuando el usuario selecciona un bit/pivote, se separa de la matriz principal el bloque de 9 bits de alrededor de dicho pivote. Esos `MatrixElement`, de la matriz principal, a través de los punteros a los sprites (y estos a los punteros de `SpriteEntry`), mueven y 'separan' visualmente los bits. Para la regeneración de dicho Bitblock, se dispone de un búffer el cual se inicializa al comienzo de la partida y reemplazará al conjunto de bits 'destruidos'. Este conjunto de bits, se permutarán y almacenarán en el búffer. Se produce un swap o intercambio de direcciones de memoria y ajuste de posiciones en la matriz (y visualmente en la pantalla).

- **Game**: Es el módulo "director". Se encarga de gestionar los estados y las fases del juego. La estructura implementada nos permite separar lógica y, gracias a ello, se puede centrar únicamente en cambiar de estados y fases en base a diferentes eventos.

  La implementación de fases radica en que un estado general, como por ejemplo puede ser el estado `GAMES_STATE_INTRO`, pueda disponer de diferentes fases internas (`PHASE_WAITING_PLAYER_INPUT`, `PHASE_INTRO_SCENE_ACTIVE`...) Algunas fases son generales, es decir, son utilizadas por múltiples estados (como la de `PHASE_WAITING_PLAYER_INPUT`). Sin embargo otras son específicas a un estado en concreto y sirven para simplemente organizar y dividir dichos estados (e.g: `PHASE_SHOW_CONTROLS` `PHASE_REGENERATING_MATRIX`,).

  En función del estado en el que se encuentre y acción/evento que ocurra, el Game se encargará de dar la orden oportuna a los diferentes módulos y/o de procesar la información relacionada al mismo juego cuando sea necesario. Por ejemplo: leer el input del jugador, cambiar de estado, o realizar alguna función intrínseca al juego en sí (gestionar las estadísticas, procesar la finalización del juego...etc).

  Veamos los posibles estados del juego, definidos inicialmente en el autómata:

  `GAME_STATE_MAIN_MENU`: Estado en el que se estará esperando a una entrada por parte del usuario, para mostrar una de las opciones cuando sea seleccionada, o bien comenzar la partida.

  `GAME_STATE_INTRO`: Los eventos secuenciales relacionados a la introducción. Se introducirá al jugador mediante una escena textual (pantalla de arriba) y gráfica (pantalla de abajo). Se avanzará de estado una vez elegida una de las dos cápsulas.

  `GAME_STATE_GAME`: Siempre y cuando se esté 'jugando' en el ámbito de la Matriz y los dos Iñatrix, se estará en este evento cíclico. Únicamente avanzará cuando se 'muera' o el jugador pulse el botón de pausa.

  `GAME_STATE_PAUSE`: Cuando se presione el botón de pausa, todo el sistema se pausará hasta que se vuelva a presionar. Ni el gestor de eventos ni ningún otro sistema se actualizarán. Única y exclusivamente se estará a la espera de que el usuario presione el botón con objeto de reanudar todos los sistemas y como consecuencia, la partida.

  `GAME_STATE_GAME_OVER`: Cuando se produzca una de las condiciones necesarias para que el jugador 'muera', se avanzará a este estado, en el cual se mostrará el fondo, texto en la pantalla de arriba etc.

  `GAME_STATE_STATS`: Estado en el cual se volcarán las estadísticas de los datos recogidos durante la partida.

- **ConsoleUI**: Mostrar textos emulando diferentes menús en la consola de la pantalla de arriba a través de las funcionalidades que ofrece libnds. Mostramos estadísticas, textos para intentar generar una sensación de dinamicidad y sincronía con lo que ocurre en el juego (a través del gestor de eventos).

- **MovementMgr**: Gestionar el movimiento de los objetos del juego. Por ahora gestiona un único tipo de movimiento, que es el movimiento en bloque de Iñatrix y lo asocia a la matriz, para que el sistema tenga constancia en todo momento del bit/pivote seleccionado de manera pasiva (es decir, pivote autoseleccionado con el movimiento de Iñatrix. Como recordatorio, la selección se considera activa una vez el jugador presiona la tecla para seleccionar).

También existe un control para asegurarnos de que Iñatrix permanezca en ciertos límites. Comienza en una posición cuya fila/columna de la matriz es 2, para evitar que se puedan coger bits de los extremos (ya que son los 9 bits que rodean al pivote). A parte existen diferentes checks de seguridad para que esos bits no puedan ser seleccionados de ninguna manera.

- **ObjectMgr**: Éste módulo se encarga de configurar y ejecutar acciones en lo referente a ciertos 'objetos' presentes en el juego; E.g: como el spawn/despawn de Iñatrix/cápsulas, detectar si se ha hecho click en alguna de las cápsulas...etc.

- **Timer**: Módulo encargado de configurar y actualizar el timer (por interrupción, llamado a través de la **rutina de atención**), así como de gestionar los datos necesarios y asegurarse de invocar la actualización del gestor de eventos para éste procese lo que considere necesario.

3.5. Sistemas de Entrada/Salida

¿Cómo influyen los diferentes sistemas de entrada y salida en éste modelo?

Constituyen los fundamentos o pilares centrales, ya que al gestionar el timer por interrupción, es cómo se consigue dar vida al juego gracias a la rutina de atención que es la encargada de, entre otras cosas, que el gestor de eventos se mantenga funcional y actualizado o incluso de que un determinado estado no avance hasta que el usuario presione una tecla en concreto. Para todo ello, el módulo principal que se encargará de dirigir las inicializaciones es el de **controllers** y gracias al software `Interrupt Dispatcher` de la NDS, se conseguirá leer los registros de interrupción para que todo esté funcional.

Input es el encargado de leer las entradas por parte del jugador (pantalla táctil y teclado) y el **timer** es el encargado del temporizador. Para ello definimos en el módulo de **controllers** 2 rutinas de atención o *handlers* dedicadas las interrupciones:

```
//Controllers.c
void controllers_SetInterruptionVector()
{
    irqSet(IRQ_KEYS, controllers_KeyPadHandler);
    irqSet(IRQ_TIMER0, controllers_TimerHandler);
}
```

3.5.1. Pantalla táctil (módulo input.c)

Cápsulas: selección de la dificultad.

El usuario puede seleccionar a través de la pantalla táctil una de las cápsulas y el sistema lo gestionará de manera oportuna.

También se mostrarán diferentes textos en función de cual se elija. En éste caso, libnds nos permite de manera muy sencilla el detectar cuando y en qué vector posición la pantalla táctil ha sido tocada o clickada.

Una vez detectada la posición, no tenemos más que hacer los cálculos necesarios para identificar si el área de la pantalla que nosotros deseamos es la que ha sido pulsada.

Observemos el código para su funcionamiento:

```
/**
 * @brief Detecta si se ha utilizado la pantalla táctil (click izquierdo del ratón
 * en el emulador).
 * @return true si ha sido pulsada, false en caso contrario.
 */
bool input_touchScreenUsed() {
    touchRead(&screen);
    return (screen.px != 0 && screen.py != 0);
}

/**
 * @brief Consulta la posición x del punto donde ha sido presionada la pantalla táctil.
 * @return Entero que indica la posición en el eje X.
 */
int input_getTouchScreenX() {
    touchRead(&screen);
    return screen.px;
}

/**
 * @brief Consulta la posición y del punto donde ha sido presionada la pantalla táctil.
 * @return Entero que indica la posición en el eje y.
 */
int input_getTouchScreenY() {
    touchRead(&screen);
    return screen.py;
}
```

```
/**
 * @brief Detecta si un GFX determinado ha sido clickado a través de la
 * pantalla táctil.
 * @param x coordenada X de la posición donde se ha hecho click.
 * @param y coordenada Y de la posición donde se ha hecho click.
 * @return @enum GfxID del objeto clickado, -1 si ninguno.
 */
int objectMgr_objectAreaClicked(int x, int y){
    if(x >= 99 && x <= 108 && y >= 81 && y <= 95)
        return GFX_CAPSULE_BLUE;
    else if(x >= 140 && x <= 159 && y >= 81 && y <= 95)
        return GFX_CAPSULE_RED;

    return -1;
}
```

3.5.2. Teclado (módulo input.c)

Configuración de registros: Todas las teclas son leídas por encuesta, salvo `<SELECT> <B>` que son leídas por interrupción.

A parte de habilitar las interrupciones a través del bit 12 de `IE`, hemos configurado de la siguiente manera en el módulo de **controllers** los registros para que las teclas `<SELECT> <B>` produzcan interrupción:

```
//controllers.c
void controllers_ConfigureInput(){
    input_ConfigureInput(0x4000 | 0x0006);
}

void controllers_EnableKeyPadInt(){
    IME=0;
    IE |= IRQ_KEYS;
    IME=1;
}

void controllers_DisableKeyInt(){
    IME=0;
    IE &= ~IRQ_KEYS;
    IME=1;
}
```

```
//input.c
void input_ConfigureInput(int mask)
{
    TECLAS_CNT |= mask;
}
```

Teclas por Interrupción: `Surrender` (mientras el juego está activo): Provoca que el jugador se rinda y el juego termine. Interrupción producida bajo demanda del jugador - interferencia en Matrix. Mismo efecto que el Game Over pero con textos diferentes.

Para gestionar las teclas leídas por encuesta, dejamos la información con respecto a las mismas en un struct que hemos preparado, y consultamos constantemente su estado. Para ello, invocamos la siguiente función en cada tick del bucle principal en `game.c`:

```
/**
 * @brief Función encargada de actualizar el @struct KeyData con la información
 * extraída de los registros apropidos.
 * @return Devuelve 1 si detecta que se ha pulsado alguna tecla, -1 en caso contrario.
 */
void input_UpdateKeyData()
{
    keyData.isPressed = input_KeyDetected();

    if(keyData.isPressed)
        keyData.key = input_KeyPressed();
    else
        keyData.key = -1;
}
```

La implementación de las funciones `input_KeyDetected()` e `input_KeyPressed()` es la siguiente:

```
/**
 * @brief Control de pulsación de teclas.
 * @return Devuelve TRUE si detecta que se ha pulsado alguna tecla.
 */
int input_KeyDetected()
{
    return (~TECLAS_DAT & 0x03FF) != 0 ? 1 : 0;
}
/**
 * @brief Función para detectar qué tecla ha sido pulsada.
 * @note Los registros de las teclas X e Y, al igual que los de la pantalla táctil,
 * sólo son accesibles desde el procesador ARM7, por esa razón y puesto que en este
 * caso tenemos disponibles el resto de las teclas, estas no las vamos a utilizar.
 * @return Identificador de la tecla asociada correspondiente.
 */
int input_KeyPressed()
{
    switch(TECLAS_DAT){
        case 0x03FE:
            return INPUT_KEY_A;
        case 0x03FD:
            return INPUT_KEY_B;
        case 0x03FB:
            return INPUT_KEY_SELECT;
        case 0x03F7:
            return INPUT_KEY_START;
        case 0X03EF:
            return INPUT_KEY_RIGHT;
        /*Resto de casos eliminados para la memoria, por ser redundantes
        y ocupar demasiado espacio*/
        default:
            return -1;
    }
}
```

Cada vez que se presione una tecla que haya sido configurada para ser gestionada por interrupción, se invocará la rutina de atención `controllers_KeyPadHandler`. Y esta, comprobará si el estado del juego es `GAME_STATE_GAME` (estado principal, donde se está manejando a ambos Iñatrix para generar `Overflows` en la Matriz), en cuyo caso se le ordenará al juego que termine la partida inmediatamente.

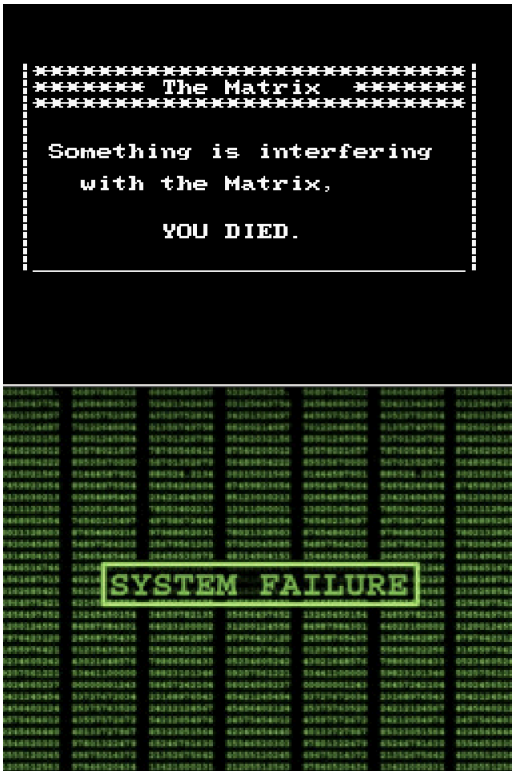

Figura 11: Surrender

3.5.3. Timer (módulo timer.c)

Para su configuración, habilitamos las interrupciones y luego especificamos mediante el bit 7, que se active el timer y mediante el 6 que si existe `overflow` produzca la interrupción correspondiente. Adicionalmente, para que el registro de control del timer interrumpa **512** veces por segundo, hemos establecido el `latch` a 0. Tras jugar un poco con los diferentes valores ofrecidos en la documentación que se nos ha proporcionado, nos hemos decantado por ésta.

```
/**
 * @brief Habilitar interrupciones para el timer.
 */
void timer_EnableInterruptions(){
    IME = 0;
    IE |= IRQ_TIMER0;
    IME = 1;
}
```

```
/**
 * @brief Configurara el timer.
 * Latch = 65.536 - (1/(ticks/seg)) * (33.554.432/x);
 * x = bits 1-0 del registro TIMER0_CNT (Ir probando hasta obtener);
 *
 * @param latch
 * @param mask
 */
void timer_ConfigureTimer(int latch, int mask)
{
    timer_EnableInterruptions();

    timer.ticks = 0;
    timer.latch = latch;
    timer.conf = mask;
    timer.time = 0;
    timer.totalTicks = 0;

    TIMER0_CNT |= 0x00C0 | timer.conf;
    TIMER0_DAT |= timer.latch;
    timer_StartTimer();
}
```

Gracias al timer, el sistema de eventos será invocado cada vez que una interrupción llame a la rutina de atención o Handler `controllers_TimerHandler`. También se traducirán los ticks a segundos y se tendrá una estructura con información relacionada al timer constantemente accesible.

Por lo tanto, las interrupciones repercutirán en dar vida al juego, afectando de manera directa a:

- Control de segundos.
- `UpdateScheduledEvents`.
- `UpdatePhases`
- `UpdateAnimations`

3.6. Autómata Inicial vs Autómata Final

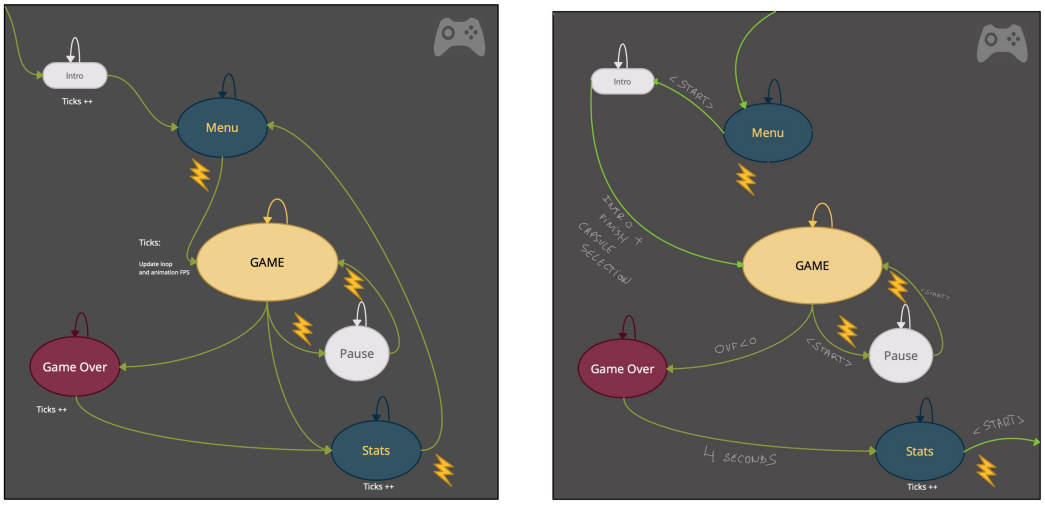

Figura 12: Autómata Inicial vs Final

Estamos bastante contentos con el resultado, ya que hemos sido bastante fieles a la idea o diseño original, tal y como se puede observar en la imagen. Hemos hecho algunos cambios pequeños entre la relación entre estados pero no hemos incluido ninguno nuevo.

La diferencias más notables, quizá podrían ser el hecho de que una vez arrancado el juego, se le lleva directamente al menú y, cuando éste presiona la tecla `<START>`, es entonces cuando da comienzo a la intro. Cuando dicha Intro termine y el jugador haya seleccionado la cápsula, es entonces cuando comienza la partida y se avanza de estado.

Por lo demás, hemos hecho que, cuando el usuario a través de la teclas configuradas por interrupción, decida rendirse, se le lleve al proceso de Game Over (originalmente iba a ser directamente mostrar Stats).

Para terminar, cuando el jugador esté en la pantalla de Stats y pulse la tecla `<START>`, el juego se cerrará.

¿Por qué hemos realizado estos cambios?

Consideramos que, pese a los mismos, hemos mantenido bastante fija la estructura inicial he hemos ido pivotando alrededor de ella. Los cambios no son muy drásticos y, los hemos ido realizando según nos ha ido pareciendo conveniente.

3.7. Ideas/features 'in the works'

Durante el desarrollo del proyecto, ha habido varias cosas que teníamos planeadas pero que, por falta de tiempo no hemos podido terminar implementando. Estas features están casi finalizadas y, aunque no la podemos presentar en la entrega oficial, lo finalizaremos y pondremos en el repositorio público.

Animación 'Bit Conjunction'
Como se puede apreciar en el módulo matrix, hay una función para un efecto lista para empezar a ser implementada. Se hicieron varias pruebas y tenía muy buen aspecto, pero decidimos dejarlo para más adelante ya que otras funcionalidades tenían más prioridad al ser esta algo meramente estético. La idea radica en que durante la regeneración de un bitBlock, los bits spawneen 'fuera' de la pantalla, cada uno en una posición aleatoria, y 'vuelen' a sus respectivas posiciones de la matriz, del bloque recién destruido.

Mejorar el texto de Overflow
Consideramos que no está claro del todo y necesita ser mejorado drásticamente, será mejorado en breves aunque no para la entrega del proyecto.

Spells de Iñatrix
Simplemente consiste en unas auras que rodean a Iñatrix cada vez que se selecciona un bit, Iñatrix hace un gesto y se presenta un spell diferente para cada uno de los Iñatrix.

Audio
Tenemos el audio funcional, pero tras unos problemas que nos ocasionó con las frecuencias decidimos deshabilitarlo y dejarlo para otro momento.

Intro
Animación intro más fluida, cuando Iñatrix persigue al conejo. Es bastante sencillo el hacerlo, simplemente con diferentes fondos etc, pero por tiempo nos ha sido imposible.

3.8. Problemas y desafíos

Inicialmente ideamos un sistema de sobreescritura del banco de memoria main de la NDS en lugar del sistema actual. Tenía como objetivo manipular los distintos elementos de la matriz, nuestra intención era la de crear un sistema que fuese eliminando/creando sprites de manera dinámica - y funcionaba (más o menos). Pero, aunque no estamos seguros, nos dió la sensación de que libnds tiene varios bugs en ese aspecto, asi que decidimos reescribir completamente la manera en la que se gestionan los sprites de la matriz.

De ahí surgió la idea del búffer para el bitblock.

3.9. Herramientas utilizadas y workflow.

Hemos utilizado multitud de herramientas diferentes, así como diferentes sistemas operativos (GNU/Linux Debian, Ubuntu, MacOS). Para el desarrollo, Vim y CLion - por mayor comodidad, hemos preferido programar en nuestros ordenadores en lugar de en la máquina virtual. Sin embargo, consideramos más oportuno mencionar cómo ha sido el *workflow*.

Para un desarrollo fluído y sin contratiempos, consideramos necesario la utilización de herramientas como los sistemas de control de versiones. Así que, el eje ha estado centrado en Github y, como hoy en día ha avanzado muchísimo con respecto a unos años, hemos utilizado varias de sus herramientas:

- **Github-Issues**: Al utilizar **Git** como sistema de control de versiones - Github no tiene rival y por lo tanto, no existe otra posibilidad mejor a utilizar como bug-tracker o sistema de reporte de incidencias que el mismo ya integrado de manera automática en cada repositorio.

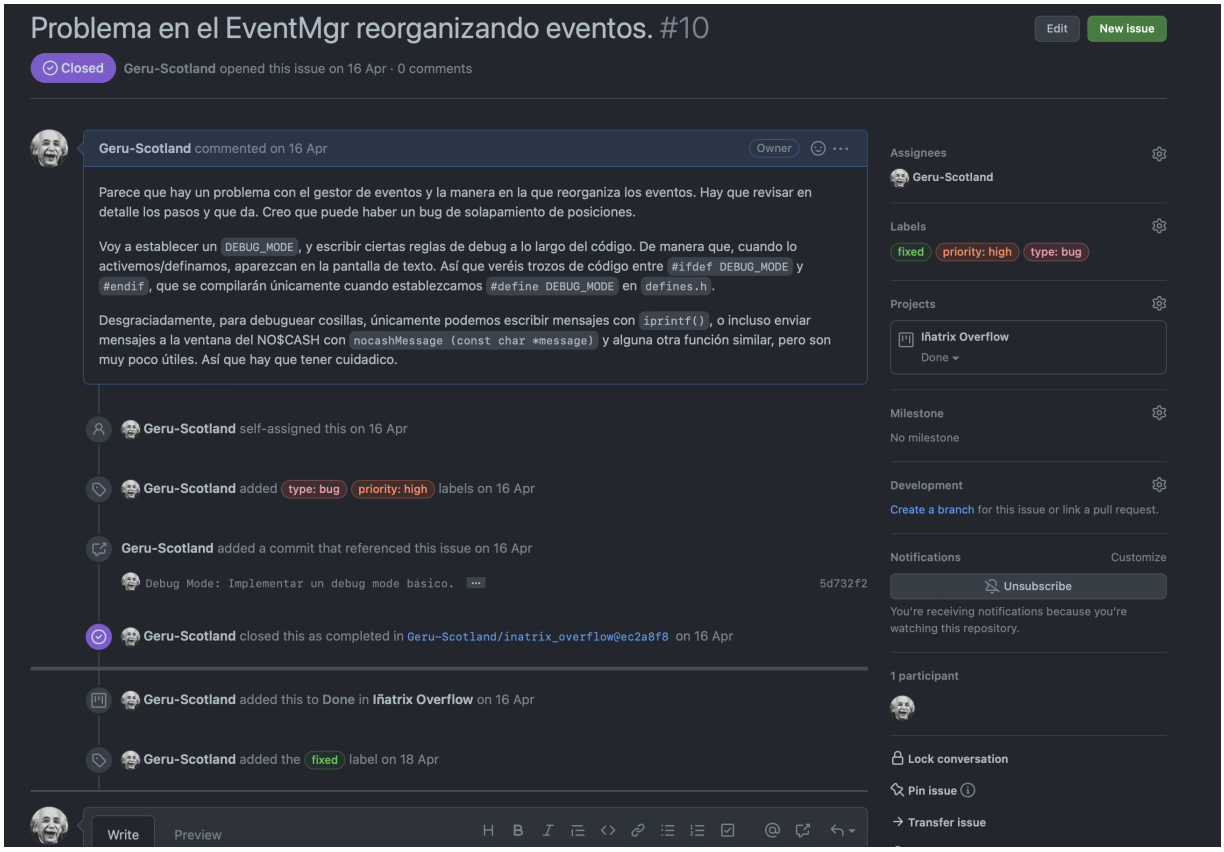

Figura 13: Issues

- **Github-Projects**: Aunque aún está en Beta, emula las funcionalidades de herramientas tan extendidas como lo eran **Trello**. De ésta manera podemos organizar tareas muy cómodamente.

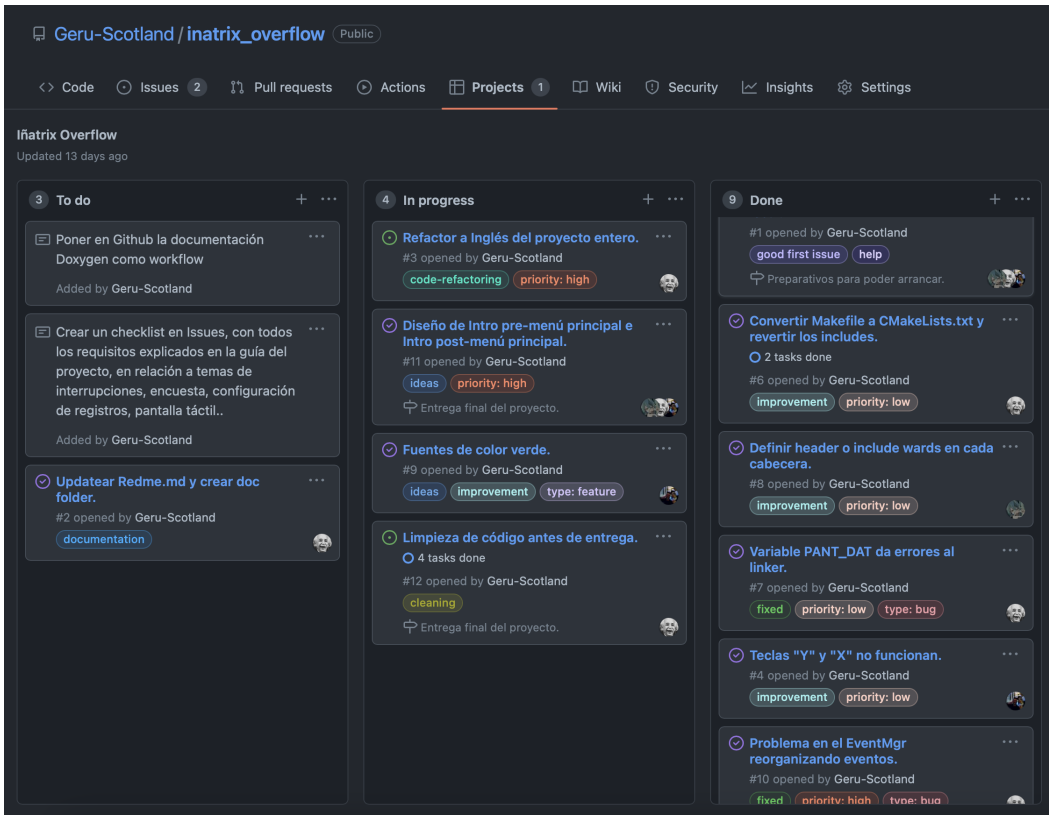

Figura 14: Github Projects

- **Codiga**: Plugin instalado en Github para revisión de código, con objeto de cumplir ciertos estándares y no cometer violaciones.

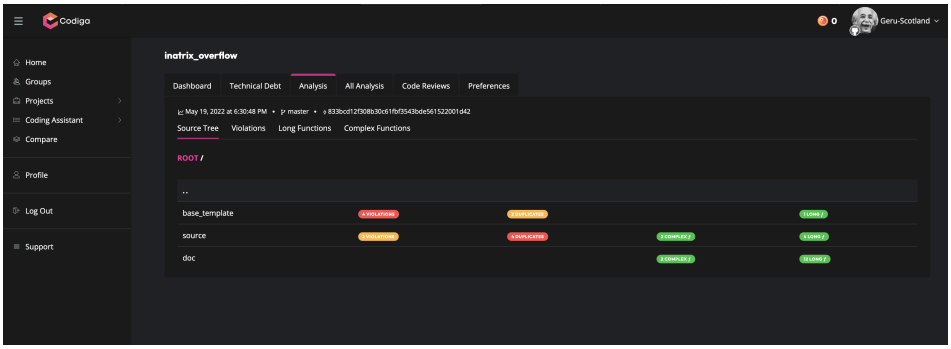

Figura 15: Codiga

4. Video demostración

Hemos grabado un vídeo [6] haciendo una breve demostración con varias de las peculiaridades del juego, para verlo, se pude seguir el siguiente enlace a Youtube:

Figura 16: Video demostración.

5. Conclusiones

Nos ha encantado trabajar en éste proyecto. Nos hemos divertido mucho y la verdad, el mundo de los videojuegos es apasionante. El poder trabajar a bajo nivel (con respecto a otros motores y posibilidades modernas) es algo muy enriquecedor que nos ha abierto los ojos en cuanto al funcionamiento interno de ciertos sistemas.

Aunque hayamos terminado construyendo lo que hemos denominado 'mini-motor', cuya intención es facilitar el trabajo y crear una capa de abstracción para el programador: el comienzo, cuando todo lo que hacíamos y trasteábamos era para poder configurar y gestionar los registros oportunos, ver cómo funcionaban los bancos de memoria de la NDS, integrar las interrupciones con los sistemas que teníamos en mente, o el empezar a dar forma al temporizador - ha sido realmente entretenido e interesante. Relacionar conceptos tan a nivel de hardware con los videojuegos, nos ha parecido una excelente idea por parte del profesado.

¡Eskerrik asko!

6. Bibliografía

Referencias

[1] https://github.com/Geru-Scotland/inatrix_overflow

[2] https://libnds.devkitpro.org

[3] https://github.com/Geru-Scotland/libnds-mini-engine

[4] https://www.doxygen.nl

[5] https://en.wikipedia.org/wiki/Souls_(series)

[6] https://www.youtube.com/watch?v=ibOBxk2QBKM

«Que Iñatrix esté contigo.»

17 de Mayo de 2022

```
>_ sudo rm -rf /
```