Modelado e Implementación de Pac-Man Simulado en MARIE

1st Felipe Rodríguez
Ing. en Ciencias de la Computación
USFQ
Quito, Ecuador
00330528

2nd Josué Ponce
Ing. en Ciencias de la Computación
USFQ
Quito, Ecuador
00330341

3rd Santiago Arellano
Ing. en Ciencias de la Computación
USFQ
Quito, Ecuador
00328370

***Resumen*—El presente presenta el procedimiento y resultados de la definición, implementación y prueba de una simulación del videojuego de Pac-Man a través del lenguaje de ensamblador MARIE. A través de la implementación y las pruebas realizadas se demostró la usabilidad de este lenguaje de ensamblador para el manejo de una simulación compleja y continua de un videojuego completo, manteniendo flexibilidad en la implementación, al mismo tiempo que manteniéndose fiel al modelo original del videojuego.**

***Index Terms*—simluación, MARIE, ensamblador**

I. Introduction

MARIE, es un lenguaje de ensamblador, basado en el modelo establecido por Linda Null en su libro *The Essentials of Computer Organization and Architecture* [1]. Este lenguaje de ensamblador, a diferencia de los diversos *flavors*, que existen y son ampliamente aplicados no solo en un ambiente educativo (x86, ARM, etc), forma parte de un entorno completo, diseñado para ser educativo, intuitivo y fácil de aprender.

En este contexto, MARIE se ha colocado como una herramienta rápida para el prototipado de aplicaciones de bajo nivel en ensamblador, así como una herramienta didáctica para evaluar el trabajo interno de un computador, específicamente de una unidad de procesamiento central (CPU).

A través de la simpleza de MARIE, y su portabilidad dado su entorno de desarollo integrado (IDE) web, MARIE se convierte en un lenguaje aplicable no solo al estudio del funcionamiento interno de bajo nivel de algoritmos, rutinas de programación como condicionales, punteros o iteradores, sino también en un lenguaje útil para la simulación de videojuegos que tradicionalmente se han implementado en este tipo de lenguajes.

Pac-Man, un videojuego de arcade tradicional, diseñado por Ken Uston en el año 1980 [2], es un videojuego con una simpleza gráfica poco replicada en la era actual. Con sus gráficas basadas en una grilla, sea horizontal o vertical, de obstáculos y monedas, entidades y *powerups*, presenta una lógica simple, interfaz gráfica distintiva, y dado la era en la que fue implementada, y las limitaciones de la misma, un *set* de instrucciones y responsabilidades entre sus actores reducida.

Basados en el modelo de Pac-Man, en su simpleza de implementación, y la facilidad de implementación de MARIE, se propuso realizar una implementación del videojuego, usando la tecnología disponible y la pantalla integrada en el IDE web de MARIE, proponiendo una replicación del procedimiento del juego, eliminando el input del usuario para simular el movimiento de todas las entidades y el avance del juego.

Para realizar esta implementación, se trabajó con varios conceptos importados de la programación orientada a objectos (POO), como el *Single Responsibility Principle, Open and Closed Principle, Separation of Concern Principle*, además, se aplicaron varias técnicas de programación funcional como la *Composition of Functions*, y adaptadores para el manejo de eventos.

II. Metodología

Para la implementación y desarrollo del programa, se implementaron diferentes secciones basadas en una técnica de *Test Driven Development*, en donde los autores realizabas pruebas de integración continuas para mantener la consistencia y concordancia entre diversas secciones del código. Asimismo, se trabajaron en varios *Sprints*, especializados en secciones del juego que requerían más o menos atención al detalle.

II-A. Levantamiento de Requerimientos y Análisis de Uso

Previo a implementar el programa, se realizó un análisis de requerimientos tanto de las figuras, colores, patrones y movimientos requeridos dentro de la aplicación, así como un análisis de uso que se le daría al videojuego y de posibles patrones de uso, determinando consideraciones críticas de la implementación a realizar, como son el manejo de entidades, mapa, etc.

Para este *Sprint*, al que se le asignó un máximo plazo de tres días, se trabajó con una organización presencial entre los autores, determinando responsabilidades del sistema, subsistemas requeridos y las responsabilidades de implementación de los autores.

II-B. Desarrollo de una GUI Desktop para la Creación de Mapas

Dado la complejidad del trabajo de guardado y manipulación de variables en el lenguaje de ensamblador de MARIE, y la extensión de estas al tratarse de variables de carga de un mapa, entidades, monedas, y otras constantes del juego, se decidió implementar una GUI desktop basada en las tecnologías

existentes en el lenguaje de programación de Python para tener una base concreta que permitiera cambiar rápidamente el mapa del juego y facilite las pruebas del código.

La implementación de esta base se dio mediante la utilización de librerías específicas de Python como son *Numpy, Pandas y PyQT5*, siendo estas usadas para el manejo de los mapas y la creación de la GUI respectivamente. El código de esta vista fue incluido en el repositorio generado para este proyecto.

II-C. Implementación del Algoritmo de Movimiento

El primer paso para la implementación de Pac-Man fue desarrollar el algoritmo de movimiento de las entidades, ya que consideramos que esta era la parte más importante del juego. Si esta funcionalidad no se lograba implementar correctamente, no sería posible garantizar el funcionamiento adecuado de los demás componentes, los cuales se describen más adelante.

En este contexto, la implementación se divide en tres fases fundamentales. La primera fase consiste en revisar el movimiento que la entidad desea realizar, evaluando la dirección indicada en el arreglo de movimientos. Dependiendo del movimiento asignado, se ajustan las coordenadas y la posición de la entidad en el display.

En la segunda fase, se verifica si la entidad ha sobrepasado los límites del display —ya sea hacia la derecha, izquierda, arriba o abajo— y, de ser así, se ajusta su posición en el display junto con sus coordenadas, de forma que se genere el efecto de "reaparición" correspondiente.

Por último, en la tercera fase, se comprueba si, después de moverse, la entidad chocará con un borde azul. En caso de detectarse una colisión, la posición y coordenadas de la entidad se restauran a sus valores anteriores, y se continúa con el siguiente movimiento del arreglo.

Esta arquitectura está basada en subrutinas *JnS*, lo cual nos permitió implementar un diseño robusto que garantiza que las cinco entidades del juego realicen sus movimientos de manera correcta, considerando tanto los obstáculos como los límites del display.

En el siguiente pseudocódigo, se puede observar el algoritmo descrito.

Algorithm 1: Algoritmo de Movimiento

Data: $IndexPointerToMovementArray$, $PosicionHexadecimalEntidadActual$, $CoordenadaEntityColumnaActual$, $CoordenadaEntityFilaActual$

Result: Movimiento de las entidades dependiendo de su lugar en el display, y de sus alrededores

```
1  internalMovementRevisionLogic() if IndexPointerToMovementArray es Derecha then
2      Sumamos UNO a la PosicionHexadecimalEntidadActual;
3      Sumamos UNO a la CoordenadaEntityColumnaActual;
4      internallyReviewCollisionsWithBoundaries;
5  else
6      internallyReviewIfTopIsFree;
7  internallyReviewIfTopIsFree() if IndexPointerToMovementArray es Arriba then
8      Restamos 16 a la PosicionHexadecimalEntidadActual;
9      Restamos UNO a CoordenadaEntityFilaActual;
10     internallyReviewCollisionsWithBoundaries;
11 else
12     internalllyReviewIfLeftIsFree;
```

Algorithm 1: Algoritmo de Movimiento Continuación

```
internalllyReviewIfLeftIsFree() if IndexPointerToMovementArray es Izquierda then
    Restamos UNO a la PosicionHexadecimalEntidadActual;
    Restamos UNO a la CoordenadaEntityColumnaActual;
    internallyReviewCollisionsWithBoundaries;
else
    internallyReviewIfBottomIsFree;
internallyReviewIfBottomIsFree() if IndexPointerToMovementArray es Abajo then
    Sumamos 16 a la PosicionHexadecimalEntidadActual;
    Sumamos UNO a la CoordenadaEntityFilaActual;
    internallyReviewCollisionsWithBoundaries;
else
    whileMovementsAreNotOneHundred;
internallyReviewCollisionsWithBoundaries() if CoordenadaEntityColumnaActual es MAYOR a 15 then
    Restamos 16 a la PosicionHexadecimalEntidadActual;
    CoordenadaEntityColumnaActual=0;
    internalBlueBorderColisionDetection;
else
    RevisarCoordenadaColumnaIzquierda;
RevisarCoordenadaColumnaIzquierda() if CoordenadaEntityColumnaActual es MENOR a 0 then
    Sumamos 16 a la PosicionHexadecimalEntidadActual;
    CoordenadaEntityColumnaActual=15;
    internalBlueBorderColisionDetection;
else
    RevisarCoordenadaFilaAbajo;
RevisarCoordenadaFilaAbajo() if CoordenadaEntityFilaActual es MAYOR a 15 then
    Restamos 256 a la PosicionHexadecimalEntidadActual;
    CoordenadaEntityFilaActual=0;
    internalBlueBorderColisionDetection;
else
    RevisarCoordenadaFilaArriba;
```

Algorithm 1: Algoritmo de Movimiento Final

```
RevisarCoordenadaFilaArriba() if CoordenadaEntityFilaActual es MENOR a 0 then
    Sumamos 256 a la PosicionHexadecimalEntidadActual;
    CoordenadaEntityFilaActual=15;
    internalBlueBorderColisionDetection;
else
    internalBlueBorderColisionDetection;
internalBlueBorderColisionDetection() if valor de PosicionHexadecimalEntidadActual es AZUL then
    revertimos a la posición anterior y a las coordenadas anteriores;
    Sumamos UNO al IndexPointerToMovementArray;
    Sumamos UNO a RepiteMovimiento;
    whileMovementsAreNotOneHundred;
else
    internallyReviewCollisionsWithBoundaries;
```

El algoritmo anterior muestra la estructura general de trabajo para el movimiento de las entidades. Como se describió anteriormente, esto ocurre en tres secciones. En la primera parte, se evalúa el movimiento que debe realizar la entidad obtenida desde el arreglo de movimientos, se actualiza su posición dependiendo del movimiento, y se la almacena en la posición hexadecimal actual.

Luego, realizamos una evaluación de la posición de la entidad teniendo en cuenta los límites del display. Si al moverse, la entidad se sale del display, se hacen los ajustes necesarios en la posición hexadecimal actual para que tenga el efecto de reaparición", es decir si se sale del display por el lado derecho, la entidad aparece en la misma fila, pero por el lado izquierdo.

Finalmente, en la tercera sección, se revisa si la posición a donde la entidad se va a mover es un obstáculo de color azul, y si es el caso, la entidad no se mueve, y se pasa al siguiente movimiento del arreglo de movimientos para intentar mover la entidad a otro lugar.

II-D. Implementación del Algoritmo de Detección de Cruce entre Entidades

Además del modelo de movimiento desarrollado en el segundo *Sprint* del proyecto, un peso importante se atribuyo al proceso de detección de un cruce entre entidades, tanto al proceso de cruce entre fantasmas, fantasmas con monedas, Pac-Man con fantasmas y fantasmas con Pac-Man. Cabe recalcar que se realiza una distinción técnica entre un cruce de un fantasma con Pac-Man, comparado con el cruce en el orden opuesto, esto dado a que en el modelado de la aplicación se requirió separar ambos conceptos para manejar correctamente el retorno de la entidad correspondiente hacia su posición inicial.

En este sentido,la implementación tiene dos fases importantes, en primera instancia, la fase de revisión de movimiento y carga de valores de movimientos correctos, y en segunda instancia una revisión agregada a esta que determina si el movimiento puede o no ser realizado basado en la posicion en el contenido de la posicion a donde el fantasma, o Pac-man tiene que ir y las consecuencias del mismo. Nuestro modelo, basdado en cinco posiciones posibles a las que cualquier entidad puede regresar asegura que aunque sea en una de las cinco la entidad se posicione y el juego continue. La primera sección de esta implementación se discutió en la sección anterior, la segunda, tiene una clara implicación en el manejo de eventos, y es donde la arquitectura usada para establecer las bases del proyecto demostró ser útil, extensible y sólida.

Por necesidades del proyecto y de los requerimientos de la aplicación, se requería analizar la dirección del movimiento, e internamente revisar si se estaba realizando un movimiento sobre un fantasma, en el caso de Pac-Man, o sobre Pac-Man en el caso de un fantasma. Para realizar esto, se utilizó el concepto de composición de funciones para lograr una arquitectura modular en donde los cambios podían ser aislados a las secciones de código correspondientes, más no a la arquitectura general. Para esto se implementaron varios métodos internos de tipo *JnS* para tener una revisión rápida dentro del algoritmo principal de movimiento y antes de la implementación de la transferencia de posición de color en la pantalla integrada y el registro de datos.

El siguiente pseudocódigo representa el proceso del método descrito en esta sección.

Algorithm 2: Algoritmo de Detección de Cruces Entre Entidades

Data: $isEntityPacman$, $isEntityTurquesa$, $isEntityRojo$, $isEntityVerde$, $isEntityRosado$, $CanComerFantasmas$, $PosicionHexadecimalEntityActAux$, $PosicionHexadecimalEntidadActual$, $PosicionHexadecimalEntidadAnterior$, $PosicionHexadecimalGhostMovidoAux$

Result: Entidades colocadas en posición correcta

```
internallyMoveAnEntity() if isEntityPacman then
    Guardar posición actual y limpiar anterior;
    if powerup activo then
        if posición actual es fantasma then
            updateGameTotalWhenFound
              AndSetUpEatOtherCondition;
    reviewIfFuturePositionIsAGhost;
    if CanComerFantasmas then
        Decrementar temporizador de powerup;
    if posición actual es moneda then
        Sumar moneda y reconstruir arreglo;
    else
        Sumar moneda al total;
    internallyMoveAnEntity;
else
    reviewAndPaintIf
    EntityisCyanGhost;
reviewAndPaintIfEntityisCyanGhost() if isEntityTurquesa then
    if posición destino es Pacman then
        if CanComerFantasmas then
            eatCyanGhostAndReturn
            ToCorrectPosition;
        else
            beEatenByGhostAnd
            ReturnToStart;
    Actualizar color y posición;
else
    reviewAndPaintIf
    EntitiyisRedGhost;
reviewAndPaintIfEntitiyisRedGhost() if isEntityRojo then
    if posición destino es Pacman then
        if CanComerFantasmas then
            eatRedGhostAndReturn
            ToCorrectPosition;
        else
            beEatenByGhostAnd
            ReturnToStart;
    Actualizar color y posición;
else
    reviewAndPaintIf
    EntityisGreenGhost;
```

Algorithm 2: Algoritmo de Detección de Cruces Entre Entidades Continuación

```
reviewAndPaintIfEntityisGreenGhost() if isEntityVerde then
    if posición destino es Pacman then
        if CanComerFantasmas then
            eatGreenGhostAndReturn
            ToCorrectPosition;
        else
            beEatenByGhostAnd
            ReturnToStart;
    Actualizar color y posición;
else
    reviewAndPaintIf
    EtityisPinkGhost;
reviewAndPaintIfEtityisPinkGhost() if isEntityRosado then
    if posición destino es Pacman then
        if CanComerFantasmas then
            eatPinkGhostAndReturn
            ToCorrectPosition;
        else
            beEatenByGhostAnd
            ReturnToStart;
    Actualizar color y posición;
reviewIfFuturePositionIsAGhost() if posición destino es fantasma turquesa then
    if CanComerFantasmas then
        eatCyanGhostAndReturn
        ToCorrectPosition;
    else
        beEatenByGhostAnd
        ReturnToStart;
else if posición destino es fantasma rojo then
    if CanComerFantasmas then
        eatRedGhostAnd
        ReturnToCorrectPosition;
    else
        beEatenByGhostAnd
        ReturnToStart;
else if posición destino es fantasma verde then
    if CanComerFantasmas then
        eatGreenGhostAnd
        ReturnToCorrectPosition;
    else
        beEatenByGhost
        AndReturnToStart;
else if posición destino es fantasma rosado then
    if CanComerFantasmas then
        eatPinkGhostAnd
        ReturnToCorrectPosition;
    else
        beEatenByGhost
        AndReturnToStart;
```

Algorithm 2: Algoritmo de Detección de Cruces Entre Entidades Final

```
beEatenByGhostAndReturnToStart() Restar vida;
Buscar posición inicial disponible para Pacman;
Actualizar posición y color;
```

El algoritmo anterior muestra la estructura general de trabajo para la revisión de entidades. En específico, el trabajo se realiza en tres secciones importantes, en primera instancia revisamos que tipo de entidad la entidad actual tendrá en su camino cuando se realize el movimiento, esto nos permite detener movimientos incorrectos antes de que sucedan. Para revisar el tipo de entidad que se tiene que trabajar, se utilizón una revisión interna del contenido de la celda de memoria a la que la figura debía ser impresa en el *display*. En base a este valor, trabajabamos para revisar si la posición a la que debe moverse era posible, teniendo en cuenta que tiene cinco posiciones posibles para moverse de regreso a la posición original de la entidad, y una vez revisado el valor, el movimiento se realizaba.

En segunda instancia, durante la realización del movimiento, es importante el trabajo extra introducido para manipular activamente arreglos e indexación de arreglos basados en la posición del fantasma actual, esto nos facilitó el registro en memoria de los cambios enviados hacia el display, sin tener que afectar la arquitectura principal del movimiento de las entidades en el caso de no tener estos casos especiales de cruce entre entidades.

Es importante notar que trabajamos con la posición actual, dado que nuestro programa, como se mostró en la sección anterior, determina que en la posición hexadecimal actual se guarde la posición a la que la figura se moverá. Luego de realizar esta revisión, el programa continua a la segunda, sección interna, de estos métodos. El programa esta diseñado para tener funciones adaptadoras y *super methods*, que permiten revisar la posición a la que se debe mover la entidad (sea Pac-Man o un fantasma) y determina la posición correcta del *reset*, que se realiza.

II-E. Implementación del Algoritmo de Carga y Descarga de Datos

En el desarrollo del proyecto, una de las tareas fundamentales es la carga del mapa y las entidades del juego en el display de 16x16, que se representa mediante un arreglo de 256 posiciones de color. Para ello, se utiliza el subprograma cargaDeMapaYEntidadesAlDisplay, el cual realiza la lectura del color actual desde el arreglo de colores y lo almacena en la posición actual del display. Posteriormente, se incrementan tanto el puntero del arreglo como el índice del display y se actualiza un contador para controlar la cantidad de movimientos realizados. El bucle principal intMainLoopLoadGame invoca a este subprograma repetidamente hasta alcanzar un límite definido, tras lo cual continúa con la inicialización de

otros componentes como la validación de paredes, revisión de entidades, y verificación del estado de vidas del jugador. Este mecanismo garantiza que el mapa inicial se renderice correctamente antes de comenzar la lógica dinámica del juego, a continuación los pseudocódigos correspondientes.

Algorithm 3: Carga de Mapa y Entidades al Display

```
1 cargaDeMapaYEntidadesAlDisplay()
    colorActual = Memoria[punteroColores];
2 Memoria[posicionDisplayActual] = colorActual;
3 posicionDisplayActual = posicionDisplayActual + 1;
4 punteroColores = punteroColores + 1;
5 contador = contador + 1;
6 if contador es mayor al LIMITE_MOVIMIENTOS then
7     cargaDeMapaYEntidadesAlDisplay;
8 else
9     intMainLoopLoadGame;
```

Algorithm 4: Bucle Principal de Carga del Juego

```
1 intMainLoopLoadGame()
    cargaDeMapaYEntidadesAlDisplay;
2 contador = 0;
3 RevisionContadorMovimientosPoder;
4 ValidacionParedes;
5 indexEntityArrayCopia = indexEntityArrayInicio;
6 whileLivesAreHigherThanZero;
7 Halt;
```

Los algoritmos presentados tienen como finalidad cargar el mapa inicial y las entidades del juego (Pac-Man, fantasmas, paredes, etc.) en la memoria de video del simulador MARIE, representada como un display de 16x16 celdas. La subrutina cargaDeMapaYEntidadesAlDisplay se encarga de recorrer un arreglo que contiene los colores correspondientes a cada celda del mapa, y copiar cada valor secuencialmente en la posición de memoria correspondiente del display. Esto se logra utilizando punteros que indican tanto la posición actual en el arreglo de colores como en el display. A medida que se copian los valores, se incrementan ambos punteros y un contador que controla cuántas veces se ha ejecutado esta operación. Una vez que se realizan todos los movimientos para llenar el display(256), el flujo continúa en el procedimiento intMainLoopLoadGame, el cual reinicia el contador, invoca procesos adicionales como la revisión de poderes, validación de colisiones con paredes, y la gestión de entidades, asegurando que el juego esté correctamente inicializado antes de comenzar la lógica interactiva.

II-F. Actualización de entidades y preservación del entorno gráfico

Antes de realizar cualquier movimiento en el juego, es fundamental mantener actualizada tanto la posición de cada entidad (como los fantasmas o Pac-Man) como el color original del espacio del que se movieron. Esta información es vital para asegurar que el entorno visual del juego (el display) se mantenga coherente y correctamente representado. Los siguientes algoritmos describen cómo se realiza este proceso: desde almacenar la nueva ubicación de una entidad hasta preservar el color anterior de cada celda, y finalmente reconstruir de forma precisa el estado gráfico del mapa.

Algorithm 5: GuardarPosicionFantasma

```
1 GuardarPosicionFantasma() Guardamos
    PosicionHexadecimalEntidadActual en el
    arreglo de entidades usando
    IndexPointerEntityArray;
2 Incrementamos el IndexPointerEntityArray en
    1 para apuntar al siguiente espacio del arreglo;
3 Saltamos a la rutina correspondiente para continuar
    con el flujo del programa;
```

Algorithm 6: GuardarColorAnterior

```
1 GuardarColorAnterior() if la entidad actual
    NO es Pac-Man then
2     Incrementamos
        IndexPointerToMovementArray;
3     if ColorTemp es distinto del color de moneda
        then
4         Saltamos a la rutina RevisionNaranja (no
            se detalla aquí);
5     else
6         Guardamos el valor de ColorTemp en
            IndexColorAnterior usando
            GuardarColor;
```

Algorithm 7: ReconstruirEntidadesEnDisplay

`ReconstruirEntidadesEnDisplay()` Usamos `IndexEntityArrayCopia` como índice de lectura para acceder a las posiciones guardadas de las entidades;
`ForCada` (*entidad (Pacman, Turquesa, Rojo, Verde, Rosado)*) Leemos la posición almacenada usando `IndexEntityArrayCopia`;
Escribimos su color correspondiente en esa posición del display;
Incrementamos `IndexEntityArrayCopia` para pasar a la siguiente entidad;
Restauramos el índice `IndexEntityArrayCopia` a su valor inicial para futuras iteraciones;
if *el jugador aún tiene vidas* **then**
    Continuamos con el movimiento de la siguiente entidad;
else
    Saltamos a la lógica de fin del juego;

Gracias a esta serie de procedimientos, el sistema logra preservar la integridad del entorno gráfico del juego, permitiendo que cada entidad se mueva sin borrar o corromper los elementos visuales del mapa. Por ejemplo, cuando un fantasma se desplaza, el color de la casilla que deja atrás es almacenado en el coloresArray y restaurado posteriormente, lo que asegura que las monedas, tanto normales como especiales, vuelvan a mostrarse correctamente. A su vez, las nuevas posiciones de cada entidad son actualizadas continuamente en el entityArray, lo que permite redibujar el estado actual del juego en cada ciclo de ejecución.

III. Resultados

Una vez implementada la aplicación, se realizaron varias pruebas unitarias que nos permitieron demostrar la compatibilidad de los modulos realizados y de la lógica implementada en estos. Para realizar estas pruebas, diversas reuniones y colaboraciones *online* fueron realizadas para probar y complementar el diseño del otro con ideas y cambios en la implementación general. En base a este trabajo conjunto, la aplicación estuvo funcional en un corto periodo de tiempo.

Con estos datos en mente, a continuación se muestran varias muestras de las pruebas realizadas a la aplicación durante su estado final, demostrando el funcionamiento del juego, especificamente del mapa, el movimiento y el manejo del puntaje.

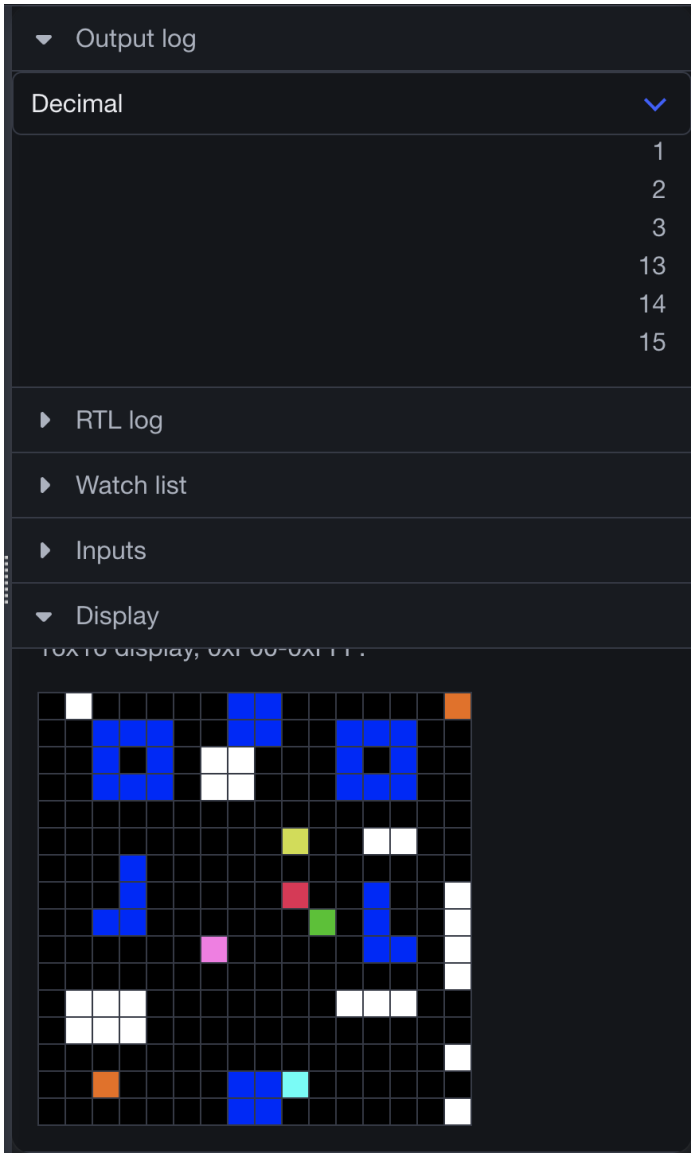

Figura 1. Ejemplo de ejecución 1

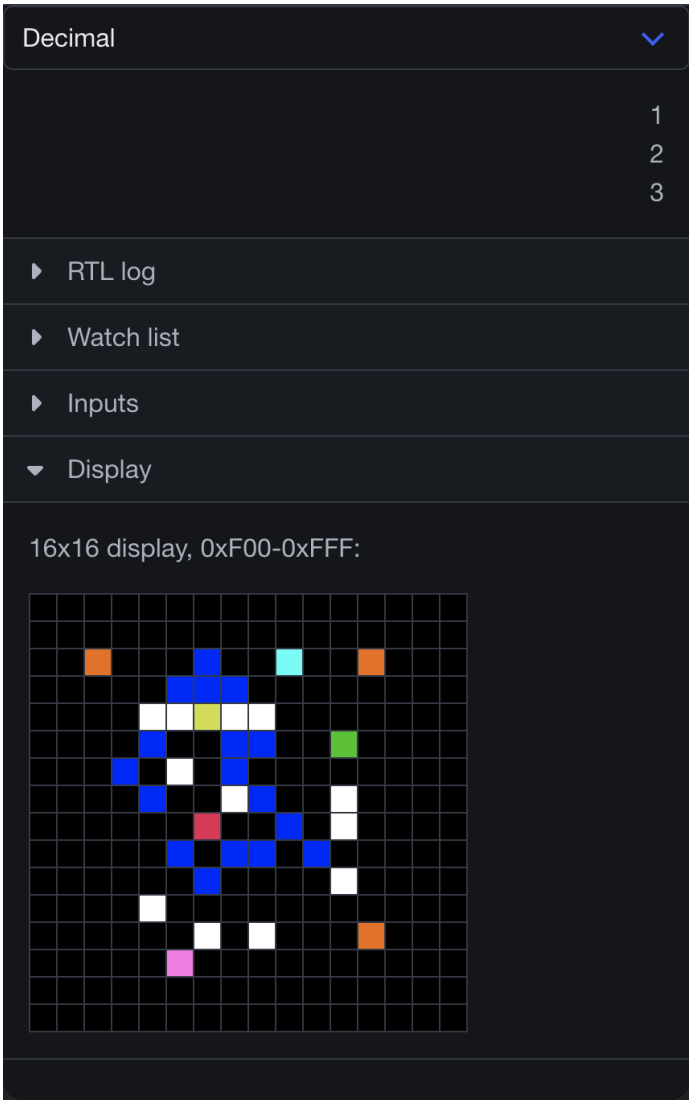

Figura 2. Ejemplo de ejecución 2

IV. Conclusiones

Una vez concluido el desarrollo de la aplicación y su implementación, el programa demuestra la factibilidad práctica de la utilización de MARIE como un lenguaje de ensamblador fácil de usar y aplicable a la simulación y el aprendizaje del lenguaje ensamblador como herramienta de bajo nivel.

Además, pudimos notar que la programación en ensamblador se beneficia grandemente de la aplicación de conceptos claves de los lenguajes de alto nivel y los patrones de diseño tradicionales que se utilizan en estos. Un claro ejemplo de esto es la utilización de adaptadores para manejar los eventos y cambios a los datos principales de la aplicación, así como el manejo de eventos generados por el sistema al mover una entidad, etc.

En general, el programa demostró un alto nivel de complejidad teórica, y la forma activa de reducirla basado en la aplicación, y extensión de conceptos de programación de alto nivel, traducidos a los componentes de un lenguaje de ensamblador como es MARIE.

Referencias

[1] L. Null “The Essentials of Computer Organization and Architecture“ 6th ed. 2023.

[2] Newman, J. Mazes, monsters and multicursality. Mastering Pac-Man 1980–2016. *Cogent Arts and Humanities*. **3**, 1-17 (2016,6)