Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

Grado en Ingeniería Informática

Trabajo Fin de Grado

Conversión de "3D Monster Maze" a una Calculadora Gráfica, Mejorado con Trazado de Rayos

Autor: Alfredo Rodríguez Sanfrutos
Tutor: Juan Zamorano Flores

Madrid, mayo de 2023

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: Conversión de "3D Monster Maze" a una Calculadora Gráfica, Mejorado con Trazado de Rayos.
Mayo de 2023

Autor: Alfredo Rodríguez Sanfrutos

Tutor:
Juan Zamorano Flores
Departamento de Arquitectura y Tecnología de Sistemas Informáticos (DATSI)
ETSI Informáticos
Universidad Politécnica de Madrid

Agradecimientos

Quiero expresar mi más sincero agradecimiento a todas las personas que han contribuido de manera significativa en la realización de este Trabajo de Fin de Grado. Sus aportes y apoyo han sido fundamentales para culminar este proyecto con éxito.

En primer lugar, deseo agradecer a mi tutor, Juan Zamorano, por su paciencia y apoyo a lo largo de todo el proceso, además de permitirme llevar a cabo esta propuesta de desarrollo.

Además, quiero reconocer el apoyo brindado por mi familia, por su constante apoyo, comprensión y motivación para poder sacar este trabajo adelante.

No puedo dejar de mencionar a mis compañeros de clase y amigos, cuyos ánimos y palabras de aliento han sido un gran estímulo para seguir adelante en momentos de dificultad.

Por último, pero no menos importante, quiero agradecer a todas las fuentes de información y bibliografía consultadas para la realización de este trabajo, así como los miembros del foro cemetech.net por su soporte y por compartir sus conocimientos conmigo. La contribución de los autores y expertos en el campo ha sido fundamental para la fundamentación teórica y el desarrollo de las ideas expuestas en este documento.

Resumen

Este Trabajo de Fin de Grado tiene como objetivo principal el desarrollo de una conversión del reconocido videojuego "*3D Monster Maze*" de los años 80, para que pueda ser ejecutado en una calculadora gráfica actual (Texas Instruments TI-84 Plus). Es importante destacar que el juego original, programado para el ordenador Sinclair ZX81, hacía uso del procesador Zilog Z80. Este mismo procesador es el que se emplea en la calculadora TI-84 Plus, lo que constituye una ventaja considerable en la tarea de portar el programa a un sistema moderno.

Asimismo, se plantea como objetivo adicional incorporar la técnica del trazado o proyección de rayos (*ray casting*) a la parte gráfica del videojuego. Esta técnica, que permite generar imágenes en 3D en tiempo real, mejora significativamente la experiencia visual del usuario. El desarrollo se llevará a cabo utilizando el lenguaje ensamblador de Z80, lo cual permite un control preciso sobre el hardware.

El propósito final del proyecto es demostrar cómo un videojuego clásico puede ser adaptado a una calculadora gráfica, y mejorado mediante el empleo de técnicas actuales. Además, se espera que este trabajo sirva como ejemplo para futuros desarrollos similares, al demostrar las posibilidades de adaptar juegos clásicos a sistemas más modernos y la incorporación de nuevas técnicas que mejoran la calidad de la experiencia del usuario. En definitiva, se trata de una labor que aúna conocimientos de programación, ingeniería y creatividad, con la finalidad de demostrar las posibilidades y el potencial que poseen los videojuegos clásicos adaptados a la actualidad.

Palabras clave: *3D Monster Maze*, calculadora gráfica, TI-84 Plus, ray casting, Zilog Z80

Abstract

This Final Degree Project aims to develop a conversion of the renowned video game "3D Monster Maze" from the 1980s so that it can be executed on a current graphic calculator (Texas Instruments TI-84 Plus). It is important to note that the original game was programmed for the Sinclair ZX81 computer, which uses the Zilog Z80 processor. The TI-84 Plus calculator also employs the same processor, providing a significant advantage in the task of porting the program to a modern system.

Additionally, the objective is to incorporate the ray casting technique into the graphical part of the video game. This technique allows for the generation of 3D images in real-time, significantly enhancing the user's visual experience. The development will be conducted using the Z80 low-level assembly language, which enables precise control over the hardware.

The ultimate purpose of the project is to demonstrate how a classic video game can be adapted to a graphing calculator and improved with current techniques. Furthermore, it is expected that this work will serve as an example for future similar developments, by showcasing the possibilities offered by adapting a classic game to modern systems and incorporating new techniques that enhance the quality of the user experience. In summary, this is a task that combines knowledge of programming, engineering, and creativity, with the aim of demonstrating the possibilities and potential of classic video games adapted to the present day.

Key words: 3D Monster Maze, graphing calculator, TI-84 Plus, ray casting, Zilog Z80

Glosario

- **Zilog Z80**: Procesador de 8 bits desarrollado por la compañía electrónica Zilog, fabricado desde 1976 hasta el presente (2023). Basado en el procesador Intel 8080 [1].
- **Sinclair ZX81**: Ordenador de 8 bits de bajo coste británico, lanzado en 1981. Utilizaba el procesador Zilog Z80 y tenía 1KB de memoria RAM, ampliable hasta 16KB con un cartucho externo [2].
- **BASIC**: Lenguaje de programación de alto nivel, pensado para hacer más asequible la programación de los primeros ordenadores, como el ZX81. Sus siglas provienen de "*Beginner's All-purpose Symbolic Instruction Code*".
- **TI**: Siglas de la compañía estadounidense Texas Instruments.
- **TI-84 Plus o TI-84+**: Calculadora gráfica de Texas Instruments, sucesora de la TI-83 Plus. Lanzada al mercado en 2004.
- **8xp**: Formato del archivo de un programa en ensamblador ejecutable por las calculadoras TI-83 Plus y TI-84 Plus.
- ***Flash APP***: En las calculadoras TI-83 Plus y TI-84 Plus, una aplicación Flash es un programa, normalmente escrito en ensamblador, ejecutable desde el menú de *APPS* de la calculadora.
- **8xk**: Formato del archivo de una aplicación Flash ejecutable por las calculadoras TI-83 Plus y TI-84 Plus.
- **Función firmware** o **BCALL**: Función integrada en el firmware de la calculadora, que se puede llamar desde un programa ensamblador.
- **Ensamblador**: lenguaje de bajo nivel, en el que se indica al procesador qué instrucciones realizar sobre la memoria y los registros.
- ***Ray casting***: trazado o proyección de rayos desde una posición, normalmente la del jugador, hasta que este impacta con un obstáculo. Sirve para calcular la distancia a los objetos cercanos en un mapa bidimensional.
- ***Sprite***: imagen que representa un objeto o personaje de un programa gráfico, generalmente en videojuegos.
- ***Bitmap***: Mapa de bits. Cada bit identifica un píxel de la imagen, que en modo monocromático indica si el píxel debe ser blanco (0) o negro (1).
- ***Buffer***: Sección de memoria utilizada para el almacenamiento temporal de variables.
- ***LCD***: pantalla que utiliza cristales líquidos para producir imágenes en dispositivos electrónicos.
- ***FOV***: Campo de visión o *Field Of View*. Es el rango ángulos que una persona puede ver a través de los ojos.

Tabla de contenidos

Glosario .... **iv**
1 Introducción **1**
1.1 Motivación y necesidad del proyecto 1
1.2 Objetivos 2
1.3 Planificación 3
1.4 Estructura de la memoria 3
2 Marco teórico **4**
2.1 Historia y contexto de 3D Monster Maze 4
2.2 Antecedentes y estado del arte del desarrollo de programas en calculadoras Texas Instruments 5
2.3 Revisión bibliográfica y especificaciones de la plataforma de destino, Texas Instruments TI-84 Plus 6
2.4 Revisión bibliográfica del videojuego original 7
3 Diseño **8**
3.1 Código fuente del juego original para ZX81 8
3.1.1 Estructura 8
3.1.2 Análisis 9
3.1.2.1 Generación de laberintos 10
3.1.2.2 Renderización de la vista 3D 10
3.1.2.3 Movimiento de Rex 12
3.1.3 Posibles mejoras 14
3.2 Código fuente del juego portado a TI-84 Plus 14
3.2.1 Lógica modificada 14
3.2.2 Periféricos 14
3.2.2.1 Pantalla 15
3.2.2.2 Teclado 15
3.2.3 Formato de aplicación 16
3.2.4 Estructura 17
3.2.4.1 Memoria 18
4 Metodología para el desarrollo **20**
4.1 Herramientas para el desarrollo de software en TI-84 Plus - tecnologías empleadas 20
4.1.1 Imágenes y *sprites* 23
4.2 Entorno de trabajo 24
4.3 Programa 25
5 Desarrollo **26**
5.1 Aspectos básicos de los programas en ensamblador en la TI-84 Plus 26
5.2 Implementación del juego original en la nueva plataforma 27

5.2.1 Diseño gráfico y mostrado de *sprites* 27
5.2.2 Introducción del juego 28
5.2.3 Movimiento y detección de teclas pulsadas 29
5.2.4 Visión 3D 30
5.2.5 Generación de laberintos 31
5.2.6 Implementación de la lógica completa del videojuego 32
5.3 Implementación de ray casting 32
5.3.1 Funcionamiento del ray casting 32
5.3.2 Movimiento del jugador dentro de la celda 40
6 Evaluación 42
7 Resultados y conclusiones 44
7.1 Líneas de trabajo futuro 45
8 Análisis de impacto 46
Bibliografía 48
Anexo 1: Código del transcriptor de imagen PNG a bits (Java) 51
Anexo 2: Guía rápida de las instrucciones BASIC del Sinclair ZX81 52

Tabla de ilustraciones

Ilustración 1. Diagrama de Gantt con las tareas definidas 3

Ilustración 2. Captura de 3D Monster Maze desde un emulador de ZX81 [6]... 4

Ilustración 3. Carátula del casete del videojuego original [8] 5

Ilustración 4. Calculadoras TI-84 Plus y TI-84 Plus Silver Edition [16] [17]..... 6

Ilustración 5. Esquema hardware de la TI-84 Plus [19] 7

Ilustración 6. Parte del output del comando LIST del casete que contiene el videojuego original [24] 8

Ilustración 7. Caracteres de bloques usados para mostrar pseudográficos. Elaboración propia 11

Ilustración 8. Distribución de secciones en la vista 3D. Elaboración propia .. 11

Ilustración 9. Secuencia de representaciones de Rex.................................... 12

Ilustración 10. Última imagen mostrada de Rex previa a la finalización del juego 13

Ilustración 11. Niveles hardware y software de una calculadora gráfica de Texas Instruments. Elaboración propia.. 15

Ilustración 12. Estructura del programa en ensamblador. Elaboración propia 17

Ilustración 13. Flujo de generación y ejecución del programa en base al script utilizado. Elaboración propia 22

Ilustración 14. Captura de la introducción, desde el emulador Wabbitemu... 28

Ilustración 15. Representación gráfica de SEENMAZE 30

Ilustración 16. Captura de la visión tridimensional obtenida, vista desde el emulador Wabbitemu.. 31

Ilustración 17. Representación gráfica de la proyección de tres rayos por pasos 33

Ilustración 18. Mostrado de los tres rayos proyectados a modo de ejemplo ... 34

Ilustración 19. Capturas de la visión tridimensional obtenida con ray casting, vista desde el emulador Wabbitemu.. 34

Ilustración 20. Problema del trazado de rayos con paredes en diagonal 35

Ilustración 21. Captura de la interfaz del videojuego en la que se ve un pasillo sin salida, vista desde el emulador Wabbitemu 35

Ilustración 22. Representación gráfica de los rayos trazados desde el jugador (figura azul).. 36

Ilustración 23. Campo de visión del jugador con respecto a su dirección 39

Ilustración 24. Rex apareciendo en la vista dibujada con trazado de rayos, vista desde el emulador Wabbitemu 41

Ilustración 25. Comparación entre la versión normal y la versión ray cast. Ambas ejecutadas con el emulador Wabbitemu 43

1 Introducción

La evolución de la tecnología en los últimos años ha sido exponencial, y esto se ha traducido en el surgimiento de una amplia variedad de dispositivos con capacidades increíbles, desde smartphones con procesadores de última generación hasta robots que pueden llevar a cabo tareas complejas y autónomas. Sin embargo, también hay dispositivos más antiguos que, aunque no son tan avanzados como los últimos modelos, aún tienen un valor histórico y un gran potencial de uso.

En este contexto, la calculadora gráfica **TI-84 Plus** de Texas Instruments es un dispositivo que se ha mantenido relevante en la industria de la educación y la ingeniería. Esta calculadora utiliza el procesador **Zilog Z80**, lanzado al mercado en 1976 [3], y que es el mismo que se encontraba en el ordenador Sinclair ZX81 de los años 80. A pesar de que es un procesador veterano, es muy eficiente en términos de coste y consumo de energía, y sigue siendo utilizado en muchos sistemas embebidos.

En este Trabajo de Fin de Grado, se propone la conversión de un clásico videojuego llamado "*3D Monster Maze*" [4] para la calculadora gráfica TI-84 Plus. Este juego fue desarrollado por Malcolm Evans en 1982 para el ordenador ZX81 y es considerado uno de los primeros videojuegos en 3D de la historia. La tarea implica no sólo portar el juego a un sistema moderno, sino también agregar una técnica de trazado de rayos (*ray casting*) para mejorar la experiencia visual del jugador.

1.1 Motivación y necesidad del proyecto

En primer lugar, se busca difundir el conocimiento de la historia de los videojuegos y en especial de los videojuegos en 3D, explorando un título icónico "*3D Monster Maze*", que marcó un hito en la industria en la década de los 80. Al adaptar este juego a una calculadora gráfica actual, se pretende hacerlo más accesible y portable, permitiendo que un público más amplio pueda disfrutar de esta experiencia de juego clásica.

Además, el proyecto tiene como objetivo aprender **principios de desarrollo de videojuegos** y técnicas de motores gráficos en 3D. La incorporación de la técnica del trazado de rayos al videojuego permitirá generar una visualización tridimensional de una forma eficiente, fácilmente adaptable a sistemas actuales.

Otra motivación y necesidad del proyecto es la **reutilización de sistemas** que comienzan a estar obsoletos, como el procesador Zilog Z80 utilizado en el ZX81 original. Al adaptar el juego a una calculadora gráfica actual que también utiliza este procesador, se demuestra que estos sistemas pueden ser reutilizados de manera efectiva y aprovechados para desarrollar nuevas aplicaciones y juegos.

Por último, este proyecto también se enfoca en la evolución de la programación a lo largo de 40 años. Al utilizar el lenguaje de bajo nivel ensamblador de Z80, se pueden adquirir habilidades y conocimientos en programación a nivel de hardware que son relevantes para el desarrollo de software actual. Así, se muestra cómo la programación ha evolucionado y se ha adaptado a lo largo del tiempo para poder utilizar tecnologías más avanzadas y potentes.

1.2 Objetivos

El objetivo principal de este proyecto es **desarrollar una adaptación del videojuego "*3D Monster Maze*" para que pueda ser ejecutado en una calculadora gráfica** actual, concretamente la Texas Instruments TI-84 Plus, y mejorar la experiencia de juego incorporando la técnica del trazado de rayos, o *ray casting*.

Para lograr este objetivo principal, se plantean los siguientes subobjetivos.

1. Realizar un **análisis detallado del juego original** y su funcionamiento, para entender cómo se puede portar a la nueva plataforma.

2. **Adaptación del código del juego original a la arquitectura de la calculadora gráfica**, para que funcione correctamente en este dispositivo. Esto requiere una comprensión detallada de la estructura de la calculadora y la forma en que se interactúa con el hardware.

3. Investigar e implementar la técnica del **trazado de rayos** en lenguaje ensamblador de Z80, para poder incorporarla adecuadamente al juego, tras haber sido este implementado completamente en la TI-84 Plus.
4. **Implementar las mejoras en la experiencia de juego** que se desean, como la técnica del trazado de rayos, para mejorar la calidad visual del juego. Para lograr esto, se debe programar de manera eficiente y optimizada para aprovechar al máximo las capacidades del hardware disponible en la calculadora gráfica.
5. Realizar un proceso de **pruebas exhaustivas** para asegurarse de que el juego funciona correctamente y es estable en la calculadora gráfica. Se debe comprobar que no hay errores o problemas de rendimiento, y que la experiencia de juego es satisfactoria.

En conclusión, el objetivo principal de este proyecto es adaptar y mejorar el videojuego "*3D Monster Maze*" para que funcione en una calculadora gráfica actual, e incorporar la técnica del trazado de rayos para mejorar la calidad visual del juego. Para lograrlo, se deben cumplir los subobjetivos previamente mencionados, como el análisis del juego original y la técnica del trazado de rayos, la adaptación del código a la arquitectura de la calculadora, la implementación de mejoras en la experiencia de juego, y la realización de pruebas.

1.3 Planificación

Este proyecto se ha dividido en sendas subtareas, para agilizar el avance y mejorar la organización del trabajo. A continuación, se presenta la planificación del proyecto estructurado en un diagrama de Gantt:

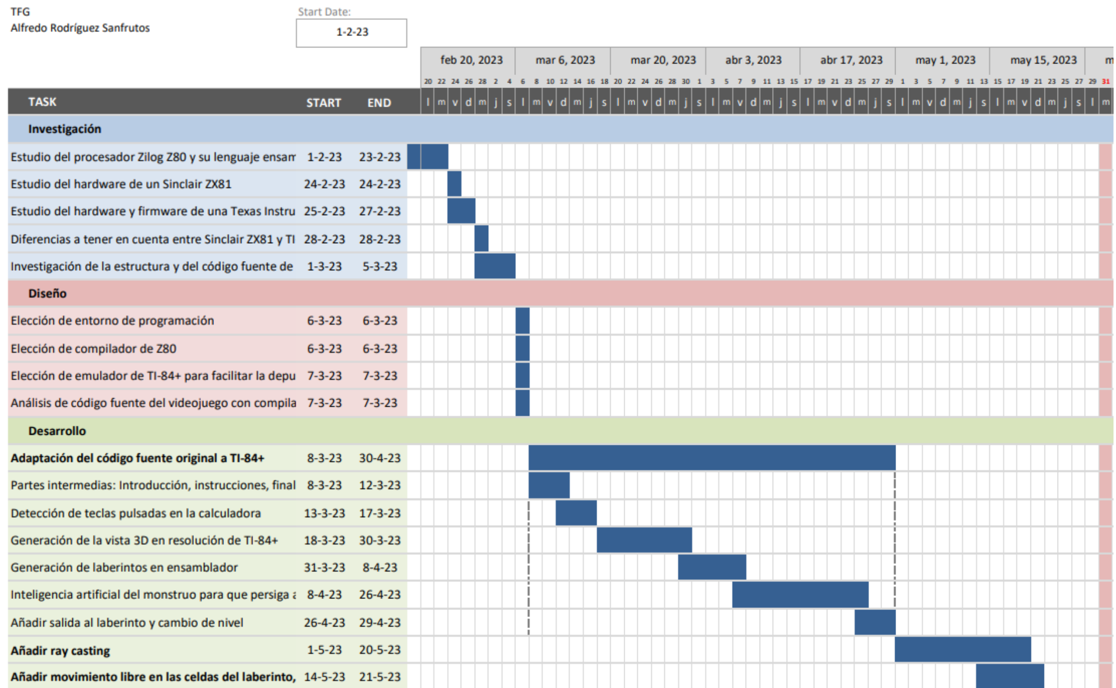


Ilustración 1. Diagrama de Gantt con las tareas definidas

1.4 Estructura de la memoria

La estructura de la memoria se divide en varias secciones:

1. Sección de Introducción, en la que se presenta la motivación y los objetivos del proyecto, junto con la planificación de este.
2. Marco teórico, donde se discute la historia y contexto del videojuego original, así como antecedentes y estado del arte del desarrollo de programas en calculadoras Texas Instruments.
3. Diseño, que recoge un análisis del código fuente del juego original y su portado a la TI-84 Plus.
4. Metodología para el desarrollo, donde se mencionan las herramientas usadas y el entorno de trabajo configurado.
5. Desarrollo. Aquí se describen los factores relevantes de la implementación y su proceso.
6. En los apartados de Evaluación y Resultados y conclusiones, se recogen los aspectos a destacar de este Trabajo de Fin de Grado, además de mencionar el impacto del proyecto en la sección de Análisis de impacto.
7. Bibliografía y los Anexos.

2 Marco teórico

Dentro de este apartado se revisa brevemente la historia del videojuego que inspira este trabajo, *3D Monster Maze*. Asimismo, se explora el estado actual del desarrollo de programas en ensamblador para sistemas basados en el procesador Z80, así como desarrollos actuales relacionados con *3D Monster Maze*.

2.1 Historia y contexto de 3D Monster Maze

3D Monster Maze se distingue como una obra pionera en el ámbito de los videojuegos en **tres dimensiones**, así como en el género del **terror**. Su desarrollador, **Malcolm Evans**, comenzó a trabajar en él después de que su esposa le regalara un Sinclair ZX81 por su cumpleaños, en abril de 1981 [5]. Antes de dedicarse al desarrollo de videojuegos, Malcolm Evans se especializaba en el diseño de hardware, ejerciendo de ingeniero en la industria aeroespacial. Sin embargo, después de recibir el ZX81, se interesó en el software y decidió explorar sus habilidades en este campo. Evans creó un generador de laberintos y, después de hablar con J.K. Greye, fundador de la empresa de software J.K. Greye Software, añadió un Tiranosaurio Rex y más mecánicas al juego. El resultado fue un videojuego desafiante y aterrador que sentó las bases para el desarrollo de futuros juegos en 3D.

Este videojuego rápidamente se convirtió en un juego icónico para el Sinclair ZX81, gracias a sus innovativos gráficos, su constante tensión, acrecentada con los periódicos mensajes sobre el estado del monstruo en el laberinto y los sobresaltos al encontrarse de frente con el temido dinosaurio.

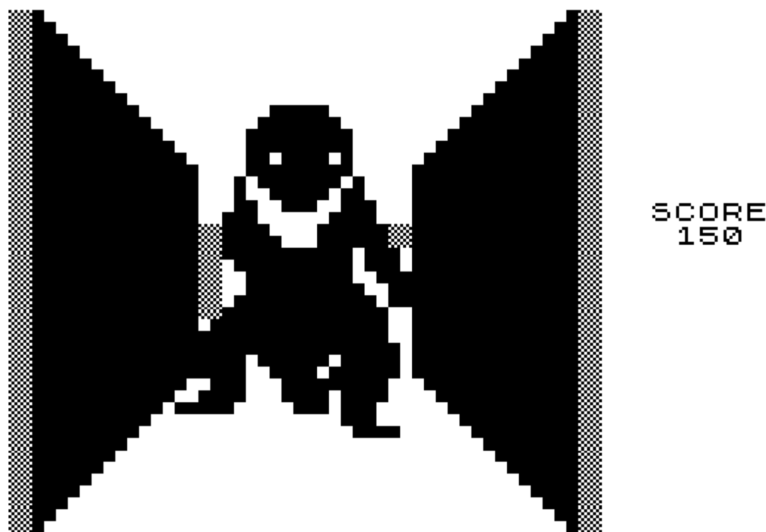


Ilustración 2. Captura de *3D Monster Maze* desde un emulador de ZX81 *[6]*

Después del éxito de *3D Monster Maze*, Malcolm Evans fundó *New Generation Software* y desarrolló más juegos, incluyendo *Trashman* [7]. *New Generation Software* se convirtió en una compañía importante en el campo de los videojuegos y continuó desarrollando juegos emocionantes y desafiantes, aprovechando la novedosa técnica tridimensional.

En España, *3D Monster Maze* fue lanzado por *Indescomp S.A.*, una distribuidora muy importante en el país. El juego fue muy bien recibido y se convirtió en un clásico en el género.

Ilustración 3. Carátula del casete del videojuego original *[8]*

La importancia de *3D Monster Maze* no puede ser subestimada. Fue uno de los primeros juegos en su campo y sentó las bases para muchos otros juegos en 3D que vinieron después. Su legado sigue vivo hoy en día y ha inspirado a muchos otros desarrolladores a crear juegos en 3D emocionantes y desafiantes.

2.2 Antecedentes y estado del arte del desarrollo de programas en calculadoras Texas Instruments

El desarrollo de videojuegos es un campo en constante evolución y, por tanto, existen numerosos trabajos en los que se ha investigado acerca de la adaptación de juegos a distintas plataformas. En este sentido, se han realizado múltiples proyectos en los que se ha portado videojuegos a dispositivos móviles, tabletas, sistemas embebidos y calculadoras gráficas, entre otros.

Se han encontrado numerosos trabajos en los que se ha utilizado la calculadora gráfica TI-84 Plus para el desarrollo de todo tipo de proyectos. Por ejemplo, en el foro *ticalc.org* [9] se encuentra un amplio catálogo de programas creados por la comunidad. De esta colección de videojuegos, se destacan los famosos títulos *Pong* [10], *Tetris* [11] e incluso una versión reducida de *DOOM* [12], además de *3dsnake* [13], que consiste en el mítico juego de la serpiente visto en primera persona, haciendo uso de *ray casting*.

Sin embargo, el videojuego *3D Monster Maze* sin modificaciones no ha sido portado a un gran número de plataformas, únicamente a formato web [14], iOS y Android [5], y Commodore PET (llamado CBM en Europa) [15].

2.3 Revisión bibliográfica y especificaciones de la plataforma de destino, Texas Instruments TI-84 Plus

La calculadora gráfica Texas Instruments TI-84 Plus es una herramienta de alta tecnología que se ha convertido en una de las plataformas más populares entre los estudiantes de secundaria y universidad. Esta plataforma, desarrollada por Texas Instruments, incorpora un procesador Zilog Z80 que permite ejecutar programas y videojuegos. También ha sido comercializada como la TI-84 Plus *Silver Edition*, la cual es idéntica a la original, salvo por la cantidad de memoria ROM, que en esta versión es mayor, y por la apariencia física. Ambas calculadoras, que son la plataforma elegida para este proyecto, se pueden ver en la Ilustración *4*.

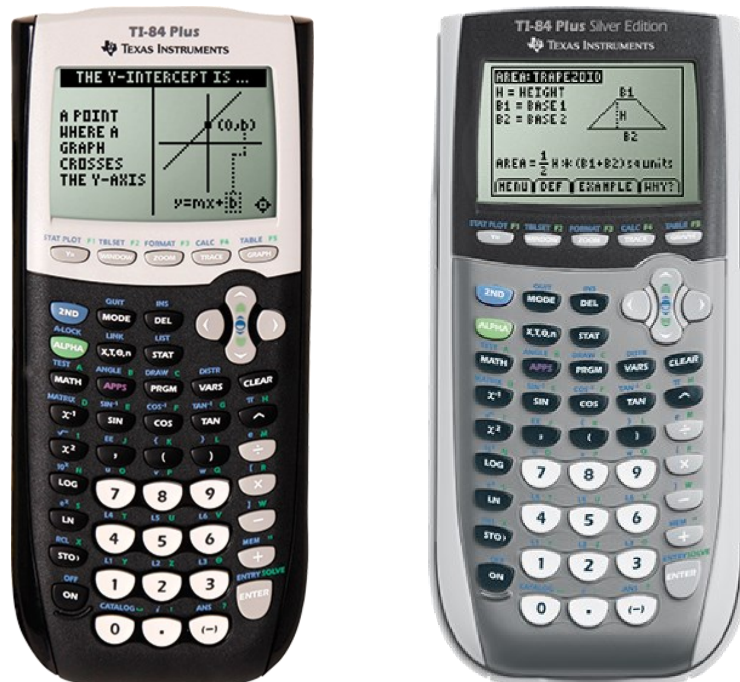


Ilustración 4. Calculadoras TI-84 Plus y TI-84 Plus Silver Edition *[16] [17]*

Es muy relevante destacar la gran popularidad que tiene esta calculadora en los Estados Unidos, donde su uso en la educación es obligatorio en varios estados [18].

En cuanto a las características hardware de esta calculadora, se destacan los siguientes apartados:

- **Procesador**: Zilog Z80 @ 15 MHz, pudiendo ser reducido a 6MHz para compatibilidad con otros programas.
- **Memoria RAM**: 24KB de memoria RAM accesible por el usuario, de un total de 128KB.
- **Memoria Flash/ROM**: 480KB de memoria ROM accesible por el usuario, de un total de 1MB.
- **Pantalla LCD monocromática**: en modo texto, 16x8 caracteres. En modo gráfico, 96x64 píxeles.
- **Periféricos**: mini-USB, conector 2.5mm I/O (DBUS), teclado de la calculadora.

En la Ilustración *5* se muestra la integración del sistema con sus componentes.

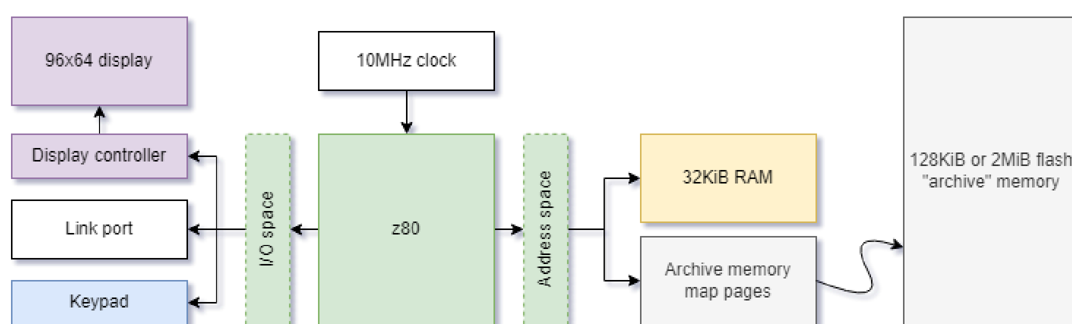


Ilustración 5. Esquema hardware de la TI-84 Plus *[19]*

2.4 Revisión bibliográfica del videojuego original

El videojuego *3D Monster Maze*, desarrollado por Malcolm Evans en 1982, es considerado uno de los primeros videojuegos en 3D. Este juego fue originalmente programado para el ordenador Sinclair ZX81, utilizando una técnica simple pero pionera para crear una experiencia visual en 3D.

En la literatura científica se han encontrado trabajos en los que se ha analizado el desarrollo del videojuego *3D Monster Maze* y su impacto en la industria de los videojuegos. Por ejemplo, en el trabajo de Filip Tołkaczewski [20] se presenta una revisión histórica de los videojuegos en 2D y 3D, en la que se destaca la importancia del juego *3D Monster Maze* como uno de los primeros juegos en utilizar y comenzar a popularizar la tecnología 3D.

Asimismo, es considerado por muchos como el primer videojuego del género de **terror-supervivencia** en 3D [21], un género que en la actualidad se ha hecho muy popular.

3 Diseño

En este apartado se describen las decisiones de diseño tomadas con el fin de facilitar los procesos en este proyecto, así como un breve análisis del código fuente original.

3.1 Código fuente del juego original para ZX81

El código del programa original se ha obtenido del desensamblado realizado por Paul Farrow y publicado en su web [22]. Este desarrollador ha extraído el código original y comentado todas las instrucciones y subrutinas para facilitar su comprensión. En los siguientes subapartados se realiza un análisis de este.

3.1.1 Estructura

Con motivo de la poca capacidad de memoria y procesamiento del Sinclair ZX81 (con expansión a 16K de RAM), la arquitectura del código del juego combina tanto el lenguaje de programación BASIC como código máquina. Esto implica que elementos como el texto inicial, con el maestro de ceremonias, que se desplaza por la pantalla, la generación del laberinto, los bucles principales del juego y las pantallas finales se administran con código BASIC y no están escritos en ensamblador.

Se puede inferir que Malcolm Evans mostraba una preferencia por el lenguaje de programación BASIC, reservando el uso del código máquina únicamente para situaciones en las que se necesitaba una mayor velocidad de procesamiento del juego. Esta elección por parte de Evans resulta comprensible, sobre todo si consideramos que el desarrollo de este videojuego marcó su debut en el mercado comercial del entretenimiento digital [23].

La parte en BASIC del programa cuenta con un total de 124 líneas de código. Es importante destacar que la primera línea contiene una declaración *REM* que engloba las secciones del juego escritas en código máquina. Debido a esto, al utilizar la función *LIST* para visualizar el programa, esta primera línea puede parecer ininteligible [23]. La salida de este comando se puede ver en la Ilustración 6.

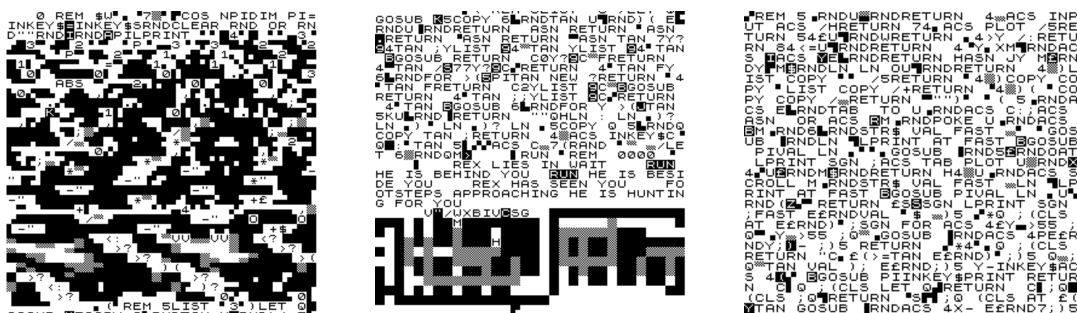


Ilustración 6. Parte del output del comando LIST del casete que contiene el videojuego original *[24]*

En lo que respecta a la comunicación entre las secciones de BASIC y código máquina, se lleva a cabo mediante la utilización de las declaraciones *PEEK* y *POKE* para pasar parámetros entre ambos programas a través de la memoria.

3.1.2 Análisis

Se ha revisado el código desensamblado de Paul Farrow [22], y se han extraído varias conclusiones. Por un lado, la utilización de etiquetas no descriptivas, tales como "*L3ED2*", hacen muy difícil su comprensión e interpretación. Sumado a esto, los flujos de ejecución del programa no están muy bien definidos, aun partiendo de la base de que es código ensamblador.

Otro aspecto importante es la sección de memoria utilizada para almacenar parámetros de la lógica del videojuego, como varios bytes utilizados para *flags* y una sección de 16x18 bytes que almacena cada celda del laberinto en un byte, sobre la que se hacen los cambios pertinentes entre _MW (pared, carácter 'W'), _ME (salida, carácter 'E'), _MP (pasillo, carácter espacio), etc.

Las "*flags*" o "banderas" son indicadores binarios que representan situaciones relevantes en un programa o juego. En este programa, cada bit del byte de *flags* tiene asignado uno o varios significados según la situación en la que se encuentre el programa, ya sea en la generación de laberintos o en el transcurso del juego en sí. En la Tabla *1* se denotan la descripción de cada *flag*.

Número de bit (menor peso conlleva menor índice)	Descripción
7	1 = El jugador ha sido atrapado
6	1 = El jugador se ha movido hacia adelante
5	1 = El jugador no se ha movido y, por lo tanto, no es necesario volver a dibujar la vista del laberinto
4	1 = La salida es visible
3	1 = Rex se ha movido
2	1 = Rex se ha movido a una nueva ubicación
1	1 = La pata izquierda de Rex está hacia adelante, 0= La pata derecha de Rex está hacia adelante
0	Controla la velocidad de movimiento de Rex. Se combina con los bits 1 y 2 para formar un contador de 3 bits. El bit 0 se fuerza a 1 cuando el jugador se mueve, lo que obliga a Rex a dar pasos más rápidos

Tabla 1. Significado de los *flags* del juego

De todo el código, se destacan varias subrutinas importantes en los siguientes puntos.

3.1.2.1 Generación de laberintos

La generación del laberinto se realiza mediante la ejecución del código en BASIC después de activar el modo rápido del ZX81 mediante el comando *FAST*. Por lo general, este proceso tarda aproximadamente 30 segundos en completarse, creando un laberinto de dimensiones 16x18 celdas.

El algoritmo de generación del laberinto es teóricamente simple, como se explica en el artículo de *Soft Tango UK* [23]. Inicialmente, se llenan todas las celdas con paredes. A continuación, se trazan pasillos de forma aleatoria, escogiendo una dirección (norte, sur, este u oeste) y una longitud (de 1 a 6 celdas), ambos componentes aleatorios para cada pasillo. El primer pasillo siempre se inicia en la posición de inicio del jugador, ubicada en la esquina sureste del mapa. Después, cada nuevo pasillo se inicia al final del anterior. El código también asegura que no haya pasillos con un ancho mayor de una celda, y que no se tracen pasillos en las paredes limítrofes del norte, sur, este u oeste.

El código continúa creando pasillos hasta que se han intentado convertir 800 celdas en pasillos. Aunque esto supone crear pasillos en más celdas de las que hay en el laberinto (240 celdas), se asume que se recorrerán algunos pasillos varias veces o se abandonarán algunos trazos debido a las restricciones anteriores. Una vez generado el laberinto, el código elige aleatoriamente una posición para la salida, buscando una celda vacía en las siete filas más al norte y, si es necesario, rellenando las celdas adyacentes para asegurarse de que la salida se encuentra al final de un callejón sin salida. No obstante, debido a la lógica utilizada para colocar la salida, es posible que la salida se encuentre en un túnel cerrado, lo que impide ganar el juego. No se ha implementado código para prevenir esta situación, aunque es poco común.

Por último, el código posiciona a Rex de forma aleatoria en las cinco filas más al norte del laberinto. Cabe destacar que, al final del juego, si Rex ha comido al jugador, el monstruo se posiciona aleatoriamente en las trece filas más al norte del laberinto, lo que puede acercarlo más al jugador si éste comienza una nueva partida.

3.1.2.2 Renderización de la vista 3D

La generación de la visualización en 3D del juego gira en torno al hecho de que el ZX81 no puede mostrar gráficos en pantalla, sino que se limita al modo texto. Por consiguiente, todo lo que se muestra en pantalla se compone de caracteres

alfabéticos integrados en el sistema, incluyendo aquellos que representan bloques.

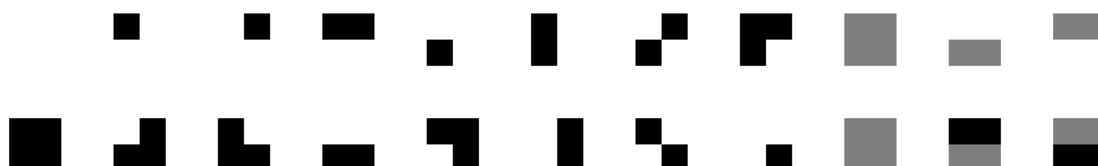


Ilustración 7. Caracteres de bloques usados para mostrar *pseudográficos.* Elaboración propia

El laberinto se representa en la pantalla mediante una subrutina de código máquina. Dibuja hasta 5 celdas hacia adelante dependiendo de los elementos que se encuentren frente al jugador en el laberinto. La pantalla está dividida en seis segmentos tal y como se puede apreciar en la Ilustración *8*.

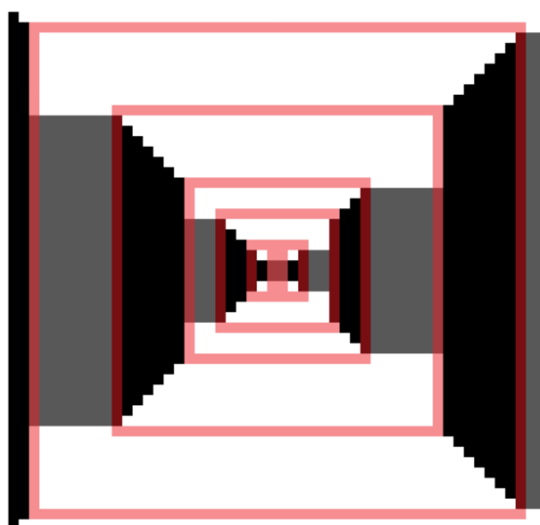


Ilustración 8. Distribución de secciones en la vista 3D. Elaboración propia

El segmento exterior muestra lo que está a los lados del jugador en su posición actual, representando una pared o un pasillo mediante un color negro o gris, respectivamente. Cada segmento interior subsiguiente representa entre 1 y 5 espacios hacia adelante, y si existe una pared a menos de 5 pasos de distancia, ocupará todo el espacio del segmento correspondiente.

A pesar de la simplicidad de los gráficos del ZX81, la representación visual en el juego es rápida y muy efectiva.

Si la salida se encuentra dentro del campo de visión del jugador, se representará como un patrón giratorio de caracteres aleatorios. Estos caracteres son seleccionados de forma aleatoria en la sección BASIC del bucle de juego y se introducen en la memoria utilizando el comando *POKE* para que la subrutina de código máquina se encargue de su representación.

3.1.2.3 Movimiento de Rex

Esta subrutina trata de mover a Rex hacia el jugador. En primer lugar, se calculan los *deltas*, o diferencias de distancia entre la posición del jugador y la posición de Rex tanto en el eje norte-sur como en el eje este-oeste. Se intenta mover a Rex a lo largo del eje con la diferencia más grande, pero si no es posible, se intenta mover a lo largo del otro eje.

Sin embargo, debido a la estructura del laberinto y el algoritmo utilizado, puede ocurrir que Rex quede atrapado en un callejón sin salida o en una esquina, lo que impide su desplazamiento hacia el jugador. En ese caso, el dinosaurio se detiene y no se mueve en absoluto. También se muestra un mensaje que dice "*REX LIES IN WAIT*", informando de que Rex está parado, acechando.

Cabe destacar que Rex siempre intenta moverse hacia el jugador, lo que significa que solo es necesario representarlo de frente en el juego. No es posible acercarse al dinosaurio desde un lado, ni verlo desde ningún otro ángulo, ya que no hay gráficos disponibles para ello.

Asimismo, Rex sólo es visible a una distancia de hasta 5 celdas. Cuando se acerca al jugador, se muestran 10 cuadros de animación en los que el dinosaurio da pasos de medio cuadro hacia el jugador, los cuales se pueden ver en la Ilustración *9*.

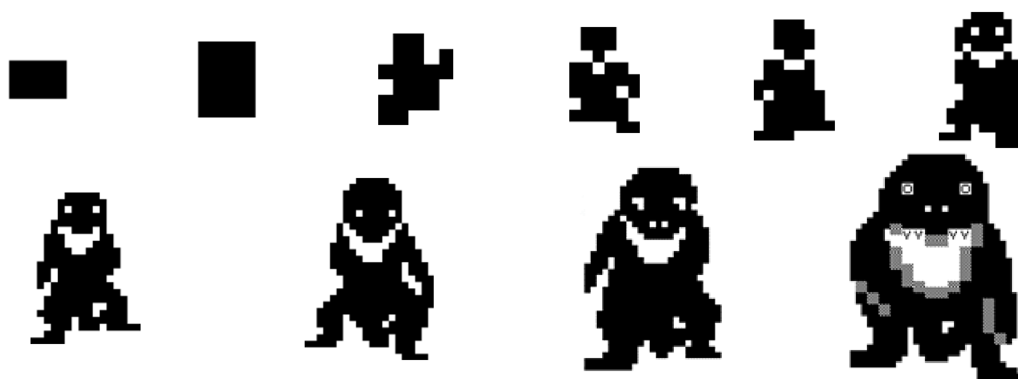


Ilustración 9. Secuencia de representaciones de Rex

Como Rex se mueve a la mitad de la velocidad del jugador, se alternan imágenes de la pata izquierda y derecha para lograr el efecto de movimiento.

Para controlar el movimiento de Rex, se utiliza un número de 3 bits en los *flags* del juego, que se incrementa en cada ciclo. Dependiendo de la combinación de bits, el dinosaurio avanza primero la pata izquierda o la derecha, y se mueve a dos velocidades distintas: 4 ciclos por ubicación cuando el jugador no se ha movido, y 2 ciclos por ubicación cuando el jugador sí se ha movido.

La Tabla *2* muestra las diferentes combinaciones que puede tomar este número:

Bit	2	1	0
Pata derecha delante	0	0	0
Pata derecha delante	0	0	1
Pata derecha delante	0	1	0
Pata izquierda delante	0	1	1
Pata derecha delante Rex se mueve a la nueva posición. Los tres bits se reinician a 000.	1	1	1

Tabla 2. Bits que gobiernan el movimiento de Rex

Cuando Rex sale de un túnel lateral y aparece frente al jugador, no hay animación lateral y simplemente aparece de repente. Después de mostrar el último cuadro de animación, el jugador ya no tiene oportunidad de escapar y la partida finaliza.

Ilustración 10. Última imagen mostrada de Rex previa a la finalización del juego

Los *sprites,* o imágenes de Rex están almacenados en la memoria utilizando un esquema rudimentario de compresión de datos, de manera que se omiten los espacios en blanco. Esta técnica de compactación de datos utiliza el método de codificación "*Run Length Encoding*" [25]. En este caso, en el primer byte se almacena el número de repeticiones del carácter con el código del siguiente byte, por ejemplo, el carácter de un espacio en blanco. A estos dos bytes, les sigue la lista de caracteres a mostrar, como se puede ver en el siguiente fragmento de código:

```
.DB $20, $01, _TOPBLACK
```

Esto significa que cuando se "descomprime" el *sprite* en la pantalla, se mantienen las posiciones de caracteres que estarían vacías. El laberinto se representa primero y luego Rex se superpone sobre la vista.

3.1.3 Posibles mejoras

En relación con las posibles mejoras a implementar, cabe destacar que algunos jugadores han expresado su insatisfacción con la capacidad de respuesta del juego a las pulsaciones de las teclas de movimiento.

Como se ha mencionado antes, el arreglo de flujos de ejecución es algo necesario si se quiere ampliar la funcionalidad del juego.

A su vez, lo más eficiente debería ser transcribir el código **BASIC** a ensamblador, para poder compilarlo y ejecutarlo de manera más simple y rápida.

3.2 Código fuente del juego portado a TI-84 Plus

Al convertir el código del programa original a la plataforma Texas Instruments TI-84 Plus, se ha procurado mantener la mayor parte de la lógica intacta, utilizando el código original y renombrando las etiquetas con nombres más significativos. Aun así, se han realizado cambios para facilitar la posterior integración de visualización 3D por medio de *ray casting*. Estos cambios se revisan en los siguientes puntos.

3.2.1 Lógica modificada

Tras analizar la manera de almacenar la posición del jugador, se concluyó que, a pesar de ser eficiente, no era fácilmente comprensible por el programador, y esto acarrearía un gran problema cuando se introdujera la proyección de rayos. Consecuentemente, se pasó a utilizar un modelo de coordenadas, *PX* y *PY*, para denominar la posición del jugador. Con ello, se implementaron las funciones *GET_MAZE* y *SET_MAZE* para acceder o modificar celdas del laberinto con las coordenadas.

3.2.2 Periféricos

En esta sección se destacan las cuestiones más importantes a la hora de convertir un programa gráfico e interactivo de una plataforma a otra.

Para comprender mejor la arquitectura del sistema y sus diferentes capas *hardware* y *software*, se ha elaborado el siguiente esquema, en la Ilustración *11*.

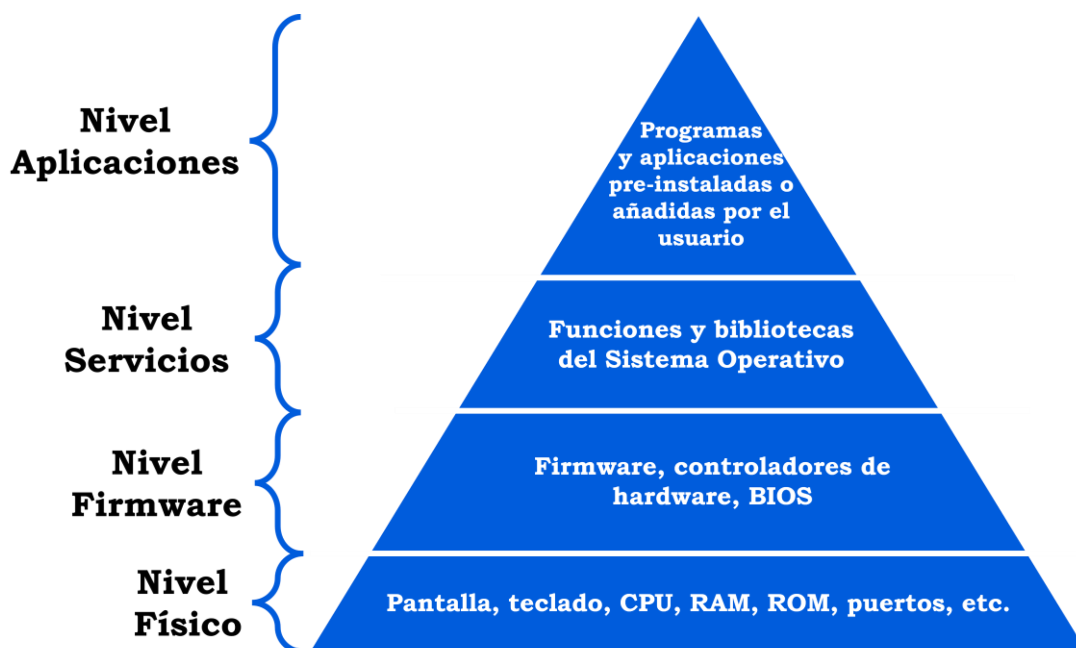


Ilustración 11. Niveles hardware y software de una calculadora gráfica de Texas Instruments. Elaboración propia

3.2.2.1 Pantalla

A diferencia de la pantalla del ZX81, que solo permite operar en modo de texto y limita la impresión de caracteres de 8x8 píxeles, la pantalla de la TI-84 Plus puede utilizarse en modo gráfico y permite la impresión de imágenes en formato bitmap monocromático, así como la capacidad de imprimir píxeles individuales. Sin embargo, un inconveniente importante es que la resolución de la pantalla del ZX81 es relativamente mayor que la de la calculadora. Mientras que el ZX81 puede alcanzar una resolución de 256x192 píxeles o 32x24 caracteres, la calculadora solo tiene una resolución de 96x64 píxeles.

Para solucionar esta discrepancia entre el modo de texto y el modo gráfico, se convirtió cada carácter de texto en dos píxeles. En combinación con el uso del conjunto de caracteres *pseudográficos* del ZX81 (como se muestra en la Ilustración *7*), Se soluciona el problema de escalado de gráficos en la sección que representa la vista en primera persona del jugador. Sin embargo, al mismo tiempo, el texto a mostrar se imprimirá con una mayor resolución en la nueva plataforma, ocupando así más espacio en pantalla. Por lo tanto, se procedió a redistribuir el texto en las áreas que no se ven afectadas por la conversión de un carácter a dos píxeles.

3.2.2.2 Teclado

La nueva plataforma, como es evidente, no posee las mismas teclas que la original. Por esto, se reasignaron las teclas utilizadas para el menú y para el movimiento del jugador, tal y como se puede ver en la Tabla *3*.

Tecla original	Tecla en la calculadora TI-84 Plus
5	Flecha izquierda ←
7	Flecha arriba ↑
8	Flecha derecha →
LIST	MODE
CONT	CLEAR
STOP	DEL

Tabla 3. Asignación de teclas en la calculadora

La relación de teclas se ha realizado teniendo en cuenta la tendencia habitual de asignar las teclas útiles de un programa a aquellas que se encuentran en la parte superior de la calculadora. Esto se puede deber a que también son las más próximas a las teclas de las flechas, que suelen ser las más utilizadas.

3.2.3 Formato de aplicación

La conversión a la nueva plataforma se basa en la estructura básica de una aplicación *Flash* de TI. Para definir que el programa sea una *Flash APP*, se debe introducir una cabecera con el nombre de la aplicación número de páginas que ocupa, etc. A continuación, se muestra la cabecera con todos sus campos.

```
.org   4000h
; Campo 'Master'
.DB   $80, $0F, 0, 0, 0, 0
; Nombre del programa
.DB   $80, $48, "3DMM    "
; Deshabilitar TI 'splash screen' (pantalla inicio)
.DB   $80, $90
; Páginas
.DB   80h, 81h, 1
; Firmar con clave/ID
.DB   80h, 12h, 1, 4
; Fecha
.DB   03h, 22h, 09h, 00h
; Firma de fecha
.DB   02h, 00
; Campo final
.DB   80h, 70h
```

Gracias al compilador utilizado, ***SPASM-ng***, se pueden utilizar directivas que facilitan enormemente la división del programa en páginas, además de crear la cabecera personalizada automáticamente. Estas directivas se mencionan y explican más en profundidad en el apartado de Aspectos básicos de los programas en ensamblador en la TI-84 Plus.

3.2.4 Estructura

Desde un principio se empezó a desarrollar teniendo en cuenta la legibilidad del código y la buena estructuración del programa. En la Ilustración *12* se muestran las diferentes secciones del programa.

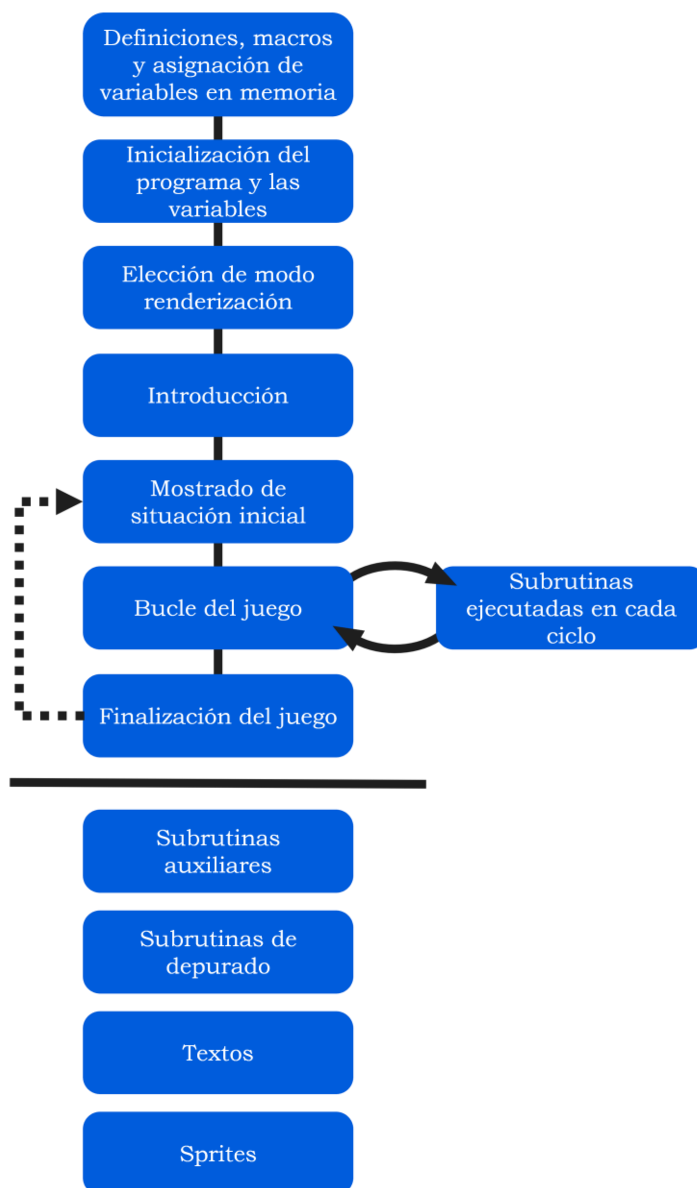


Ilustración 12. Estructura del programa en ensamblador. Elaboración propia

Al principio del archivo del programa se registran las definiciones de palabras para facilitar la comprensión del código, además de las macros y la asignación de variables en la memoria RAM. A continuación, se inicializan variables necesarias del juego, tales como la posición del jugador, de Rex, se inicializan las *flags* a 0, etc. Una vez hechas estas tareas iniciales, se pregunta al usuario para que elija el modo de renderización que prefiera, entre el modo original y el modo con trazado de rayos.

Lo siguiente consiste en mostrar el *sprite* del maestro de ceremonias y se muestra el texto de introducción desplazándose verticalmente. Cuando se acaba de mostrar este texto, el usuario puede consultar los controles del mismo modo, cerrar el juego o empezarlo. Tras consultar los controles o empezar el juego, se genera el laberinto y se muestra el punto de vista inicial. En cuanto el usuario pulsa una tecla, se entra en un bucle en el que se mueve a Rex, se mueve al jugador si se presiona alguna tecla y se ejecutan más subrutinas auxiliares para poder ejecutar el juego.

Cuando el jugador se sitúa sobre la celda de la salida, se vuelve a generar otro laberinto y se vuelve a entrar al bucle. En el caso de que Rex alcance al jugador, este tiene dos opciones: continuar y volver al mismo laberinto o “apelar”, lo que significa que tendrá un 50% de probabilidad de salir del juego, o si no, volver al laberinto.

3.2.4.1 Memoria

Las ubicaciones de la RAM disponibles para el uso del programador se pueden consultar en el manual de desarrollo de TI-83 Plus [26]. Estas son partes de RAM que no son utilizadas por las rutinas del sistema, excepto en circunstancias especiales. Por lo tanto, están disponibles como RAM temporal para la aplicación. Estas áreas son comúnmente conocidas por los programadores como "*safe RAM*", o RAM segura.

Cabe destacar que sólo son seguras de utilizar hasta que la aplicación se cierra. Una vez ocurra esto, los datos podrán ser destruidos por otras aplicaciones. En la Tabla *4* se describen estas secciones y el uso que se las da en el programa.

Identificador [dirección de memoria en hexadecimal]	**Tamaño** (en Bytes)	**Descripción**
`plotsScreen` [9340h]	768	Se utiliza como un búfer que almacena los píxeles de la pantalla antes de mostrarla, como, por ejemplo, a la hora de pintar la vista 3D. Con una llamada a la función de firmware "*BufCpy*", se puede copiar todo su contenido a la pantalla.
`saveSScreen` [86ECh]	768	Se utiliza como un búfer para almacenar los *sprites* antes de mostrarlos. Esto debe ser así para poder utilizar la función de firmware "*DisplayImage*", que toma un puntero para mostrar la imagen por pantalla. Si el puntero se encuentra dentro de la propia página de la *APP*, al ser esta retirada de la memoria RAM para ejecutar la función de firmware, apuntará a datos inesperados, por lo que primero se mueve a esta sección de RAM segura, que permanece intacta en llamadas a funciones firmware.
`appBackUpScreen` [9872h]	768	En esta sección se almacenan todas las variables del programa. Entre otras, se guarda el laberinto, que son 16x18 Bytes (288 Bytes), las posiciones del jugador y del monstruo, los *flags* del programa, etc.

Tabla 4. Secciones de RAM seguras utilizadas en el programa

4 Metodología para el desarrollo

Para lograr el objetivo de este trabajo, el proceso de desarrollo se dividió en varias fases. En primer lugar, se trabajó en la programación de las partes básicas del juego, como la introducción y la visión en 3D. Esto permitió sentar las bases para la implementación de la lógica del juego y, en última instancia, la implementación del *ray casting*. De esta forma, también sirvió para la familiarización con el lenguaje ensamblador de Zilog Z80 dentro de una TI-84 Plus.

Además, se partió de cero para garantizar la organización y la eficiencia del código a medida que se avanza en el proyecto. Se utilizaron metodologías de desarrollo ágiles para garantizar que el proyecto se desarrollase de manera rápida y eficiente, con el objetivo de lograr una versión funcional del juego en el menor tiempo posible.

También se utilizaron herramientas de depuración y pruebas para garantizar que el juego fuera estable y libre de errores. Además, se realizaron pruebas de rendimiento para garantizar que el juego se ejecuta sin problemas en la calculadora.

4.1 Herramientas para el desarrollo de software en TI-84 Plus - tecnologías empleadas

Por un lado, se valoró la elección de **entorno de programación**. Aunque existen *IDEs* creados específicamente para el desarrollo, compilación y depuración de programas de Z80, ya están obsoletos y la compatibilidad con sistemas de 64bits como *Windows 10* es mínima. Por esto, se decidió aprovechar la extensa utilidad de **Visual Studio Code** con *plugins* como "*Z80 Assembly*" [27] para aportar *syntax highlighting*[1] e información sobre el conjunto de instrucciones del procesador.

A su vez, se buscó un **compilador** compatible con las calculadoras TI-84 Plus. Se concluyó que lo mejor era utilizar uno orientado específicamente a esta plataforma, que incluyera características adicionales para detección de errores y funciones especiales de esta familia de calculadoras, como las llamadas a funciones del firmware de esta. Finalmente se escogió el compilador ***SPASM-ng***, una versión renovada del compilador *SPASM*. Algunas de las razones por las que se optó por este compilador son las siguientes:

- **Errores personalizados**, relacionados con el mapa de memoria, por ejemplo.

[1] *Syntax highlighting* se refiere a la utilización de colores en el texto del programa para distinguir la correcta sintaxis de cada instrucción, mostrando colores distintos para nombres de instrucciones, datos inmediatos, etc.

- Opción de **compilación** directamente al archivo de ensamblador ejecutable directamente desde la misma calculadora (**8xp**), así como al tipo de archivo **8xk**, que define una aplicación *Flash*.

- **Facilidad de uso**. Con un sólo comando se puede compilar y generar el archivo correspondiente.

- **Código abierto**. Se puede comprobar el código fuente abiertamente y es un programa de uso gratuito.

- **Modernidad**. Aunque la primera versión fuera publicada en 2012, se sigue manteniendo por la comunidad.

- **Documentación** disponible.

Por otro lado, el desarrollo no podría ser posible sin un **emulador** de la calculadora. Pese a que se podría transferir el programa a una TI-84 Plus real, este proceso lleva varios minutos y se corre el riesgo, aunque mínimo, de corromper el firmware del dispositivo por accesos a memoria indebidos o acciones similares. Es por esto por lo que se buscó un emulador de calculadoras Texas Instruments para solventar este inconveniente y hacer la labor de depuración más conveniente.

Tras intentar utilizar otros emuladores como *jsTIfied* [28] o *TilEm 2.x* [29], se seleccionó ***Wabbitemu*** [30] por su facilidad de uso, estabilidad y buena calidad. También, permite abrir una instancia de la calculadora emulada especificando por línea de comandos la ruta del programa que se quiere enviar a la calculadora. Gracias a esto, se creó un punto de guardado con el comando de ejecución del programa compilado para poder ejecutarlo simplemente pulsando *Enter* en la propia calculadora. Otro punto importante de este emulador es su depurador incorporado, muy completo y útil.

Cabe destacar que, debido a que el sistema operativo de las calculadoras Texas Instruments es propiedad de la empresa homónima, es necesario extraerlo desde una calculadora original, algo que *Wabbitemu* da la posibilidad de hacer fácilmente, conectando la calculadora por *mini-USB*.

Como muestra el programador *Chibiakumas* [31], se puede utilizar un *script* (*batch file* en Windows) para compilar y emular, el cual ha servido como referencia para elaborar un *script* propio ajustado a las necesidades de este trabajo, que se muestra a continuación:

```
@echo off
:: Comprobación de que existe el archivo
if not exist "SRC\%1.asm" (
    echo File %1.asm does not exist in SRC folder.
    goto :eof
)
```

```
:: Compilar programa
echo Compiling...
cd SPASM
spasm64.exe ..\SRC\%1.asm program.8xk -I ..\INCLUDE\
cd ..
move SPASM\program.8xk BIN >nul
echo Compiled!

:: Ejecutar programa
cd EMU\Wabbitemu

:: Matar proceso Wabbitemu si estuviera ejecutándose ya
taskkill /IM wabbitemu.exe /F >nul

echo Executing...
:: Arrancar Wabbitemu en segundo plano con START /B
START /B Wabbitemu.exe ..\LaunchWabbitemu_FLASH.sav  ..\..\BIN\program.8xk
cd ..\..
```

En resumen, se compila el código fuente elegido, se mueve el archivo "*8xp*" generado a la carpeta de binarios (*BIN*) y se ejecuta *Wabbitemu*, enviando el archivo compilado a la memoria de la calculadora emulada.

Gracias a este *script* se puede compilar y ejecutar con un único comando, pasando como argumento el nombre del archivo que contiene el código fuente del proyecto, que deberá estar almacenado en la carpeta *src*.

El flujo de compilación, generación y emulación o ejecución del código se puede ver en la Ilustración *13*.

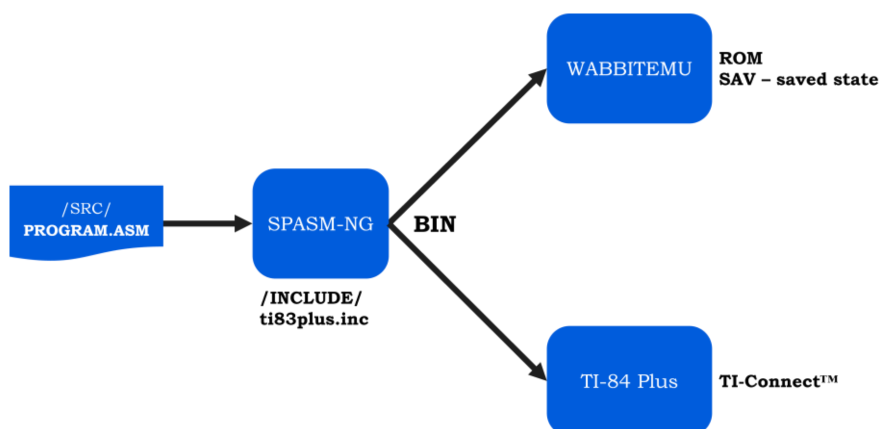


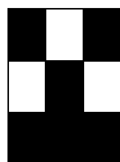
Ilustración 13. Flujo de generación y ejecución del programa en base al script utilizado. Elaboración propia

Como se observa en el esquema, una vez compilado el programa en el archivo *8xk*, se puede ejecutar en el emulador *Wabbitemu* o enviarlo a una TI-84 Plus física por medio de un cable *mini USB*, utilizando el programa *TI-Connect*™ de Texas Instruments.

4.1.1 Imágenes y *sprites*

En cuanto a la parte artística, se han creado los *bitmaps* necesarios para el programa, por ejemplo, del monstruo Rex, del *Ring Master* (maestro de ceremonias), etc. Para facilitar la declaración de los mapas de bits en ensamblador, ya que es necesario hacerlo bit a bit, se creó una función en *Java* (Anexo 1: Código del transcriptor de imagen PNG a bits (Java)), que convierte una imagen en formato *png* a las instrucciones necesarias para declarar bit a bit la imagen que se le pase como argumento. Con este simple programa se puede dibujar una imagen píxel a píxel en un editor (*Paint 3D* [32] en este caso) y pasarlo rápidamente al programa. A continuación, se muestra un ejemplo de su funcionamiento:

Ilustración de entrada:

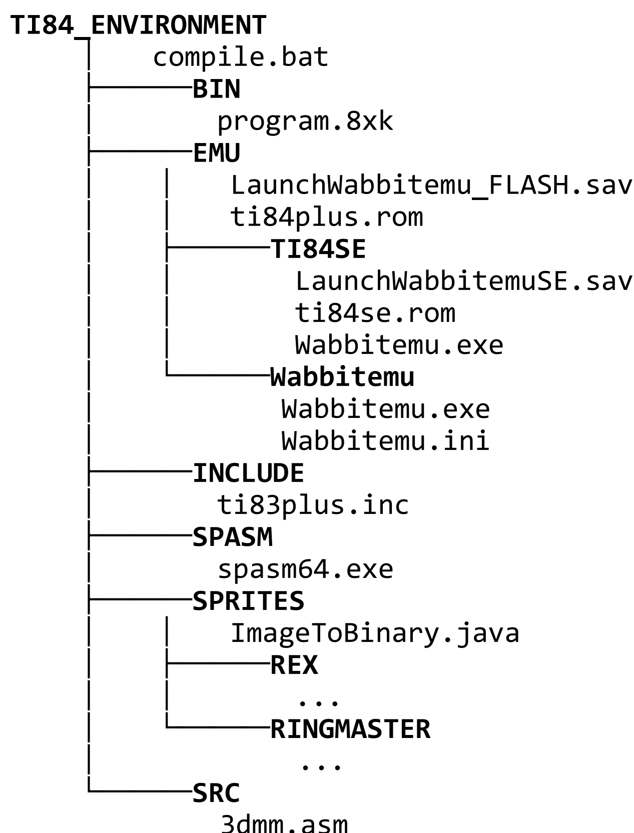


Salida:

```
.DB 3,8 ; altura y anchura de la imagen
.DB %10100000
.DB %01000000
.DB %11100000
```

4.2 Entorno de trabajo

El entorno de trabajo consiste en una carpeta que contiene las siguientes subcarpetas y archivos:



Simplemente con esta carpeta y un editor de texto, ya se puede editar, compilar y ejecutar el programa. A continuación, se explica brevemente la utilidad de cada carpeta y archivo:

- **BIN**: En esta subcarpeta se almacenan los ficheros compilados, principalmente en formato *8xk*.

- **EMU**: Contiene todos los ficheros necesarios para el uso de *Wabbitemu*, el emulador de calculadoras gráficas Texas Instruments, junto con un fichero de guardado que parte del menú de *APPS* de la calculadora.

- **INCLUDE**: Aquí se almacena el fichero que contiene macros y definiciones útiles para el desarrollo de programas en TI-83/84+, proporcionado por Texas Instruments [26].

- **SPASM**: En esta carpeta se guarda el ejecutable del compilador SPASM-ng.

- **SPRITES**: Aquí se guardan los archivos en formato *png* de los *sprites* utilizados en el programa, así como el programa Java que los convierte en sentencias de asignación de memoria de Z80.

- **SRC**: Esta subcarpeta guarda el fichero de código fuente del programa, en este caso *3dmm.asm*, que es el que contiene el videojuego convertido.

- **Compile.bat**: Este fichero es un *script* de Windows (*batch file*) que ejecuta los comandos que contiene, línea por línea. En este caso, hace uso de las carpetas previamente mencionadas para compilar y ejecutar el emulador de la calculadora con el programa listo para ejecutar.

Al tener todo lo necesario en una misma carpeta, la integración del desarrollo en la plataforma **GitHub** se convierte en una tarea muy simple, que ayuda a tener controlados los cambios, un historial de versiones y una copia de seguridad en la nube. Esta carpeta se almacena en un repositorio nuevo de GitHub y se va actualizando con los cambios realizados al programa.

4.3 Programa

En un principio se evaluó la posibilidad de realizar el programa completamente como un fichero ejecutable por la calculadora en ensamblador, con el comando *Asm(prgm)*. Sin embargo, más tarde se descubrió que este tipo de programas tiene un límite de tamaño de aproximadamente 8KB. Por lo tanto, no sería posible implementar el videojuego completamente con imágenes y textos embebidos en el código.

Después de un análisis exhaustivo de varias alternativas para permitir programas de tamaño mayor a 8KB, como utilizar un *shell* como *ION* [33] para ejecutar estos programas, se decidió migrar este programa a una aplicación ***Flash*** [34].

Estas aplicaciones se dividen en páginas de 16KB establecidas por el programador, las cuales se intercambian entre ROM y RAM según se requiera. Se pueden ejecutar desde el menú de aplicaciones, pulsando en la tecla de *APPS* de la calculadora y eligiendo el programa a ejecutar.

En el apartado de Aspectos básicos de los programas en ensamblador en la TI-84, se explican más a fondo las diferencias entre ambos tipos de programas, con relación a las directivas necesarias, y los aspectos a considerar por el programador al utilizar cada uno de ellos.

5 Desarrollo

Tras haber preparado el entorno con las mejores herramientas (apartado 4.1) y analizar el código fuente original de *3D Monster Maze*, se comenzó el desarrollo del programa partiendo de cero.

Se optó por comenzar a programar subrutinas específicas para el entorno de una calculadora gráfica en contraste con el ZX81. De esta manera, se adquieren conocimientos sobre las características fundamentales de la programación Z80 en la TI-84 Plus, al tiempo que se familiariza con las instrucciones de ensamblador de Z80.

Además, esta estrategia permite estructurar el programa de manera que facilite la posterior incorporación del trazado de rayos (*ray casting*) al programa original.

5.1 Aspectos básicos de los programas en ensamblador en la TI-84 Plus

Como se ha mencionado en el punto 4.3, para desarrollar el programa se decidió crear una **aplicación *Flash*** (*Flash APP*), la cual se almacena en la memoria ROM de la calculadora, de mucha más capacidad que la memoria RAM. Se transfieren dinámicamente páginas del programa de 16KB cada una a la RAM, según su uso. Por esto, es necesario indicar manualmente la distribución de conjuntos de instrucciones en páginas en el código. Para este menester se utilizan directivas de *SPASM* para marcar el comienzo y final de cada página, *.defpage(nPag, "PROGRAMA")* y *.validate*, respectivamente. La directiva *defpage* también establece la cabecera correspondiente que indica que el programa es una aplicación *Flash*. Por último, para finalizar la ejecución de un programa, se debe utilizar la función de firmware *JForceCmdNoChar*, la cual termina la aplicación y devuelve el control al sistema operativo de la calculadora.

Una dificultad inherente a este tipo de aplicaciones es la imposibilidad de utilizar funciones de firmware que requieren un puntero como entrada. Esto se debe a que, al ejecutar dichas funciones, es necesario cambiar la página de memoria almacenada en la RAM, lo que implica que el puntero ya no apuntará a la dirección correcta. Para superar este inconveniente, se pueden almacenar los datos que se utilizarán en las llamadas a estas funciones firmware en áreas de RAM seguras, tal y como se ha mencionado en el apartado de Memoria. Estas zonas están siempre disponibles en la RAM, independientemente de la página del programa que se haya cargado.

Debido a la extensión del programa desarrollado, finalmente se decidió implementarlo en una *Flash APP*.

Por otro lado, las funciones de firmware son de gran importancia en la programación en ensamblador para TI84+, ya que permiten acceder a las características del hardware y realizar operaciones complejas. Para utilizar estas funciones, se puede hacer uso de las siguientes instrucciones:

```
RST $28
DW $44FB ; Código hexadecimal del ID de la función firmware
```

Con el fin de compactar el código y mejorar su legibilidad, Texas Instruments ha definido una macro en el archivo *ti83plus.inc* llamada *BCALL(funciónFirmware)*, siendo *funciónFirmware* el código hexadecimal de esta.

Por otro lado, al no disponer este procesador de instrucciones que realicen multiplicaciones, divisiones ni operaciones matemáticas que vayan más allá de sumar o restar, fue necesario implementarlas desde cero en subrutinas auxiliares, basando su estructura en implementaciones anteriores para ensamblador de Z80, debidamente referenciadas en el código.

Otros aspectos importantes son la configuración adecuada de la calculadora al comenzar la ejecución de un programa de este tipo. En este, se llaman a las siguientes funciones:

```
BCALL(_RunIndicOff) ; Deshabilitar indicador de ejecución/ocupación
BCALL(_clrLcdFull)  ; Limpiar la pantalla
BCALL(_ForceFullScreen) ; Forzar a ejecutar en pantalla completa
BCALL(_DisableApd)  ; Deshabilitar apagado automático
```

Con estas funciones se deshabilita el indicador de que la calculadora está ocupada ejecutando un programa, se limpia la pantalla, se ejecuta en pantalla completa, y se previene que la calculadora entre en estado de hibernación mientras se ejecuta el programa, para evitar que se borre la memoria RAM segura.

5.2 Implementación del juego original en la nueva plataforma

A continuación se detalla, dividido en secciones, el proceso de desarrollo del juego original.

5.2.1 Diseño gráfico y mostrado de *sprites*

Utilizando *Paint 3D* [32] se dibujaron, píxel a píxel, las imágenes del maestro de ceremonias (*Ring Master*) hablando y saludando con el sombrero, y las distintas apariciones de Rex, acercando las imágenes adaptadas lo máximo posible a las originales.

Posteriormente se introdujeron al programa haciendo uso del programa en Java desarrollado (apartado 4.1), que transforma imágenes a asignaciones de memoria en ensamblador de Z80.

Para imprimir por pantalla estás imágenes, se creó una subrutina que mueve el contenido de la imagen a mostrar a *safe RAM*, y después llama a la función de firmware *DisplayImage*, que recibe como parámetros las coordenadas X e Y para

la esquina superior izquierda de la imagen, y la dirección de la imagen en memoria, pasando ambos argumentos por registros.

5.2.2 Introducción del juego

Se eligió comenzar desarrollando esta parte debido a la aparente poca complejidad que conlleva, y no afecta a la lógica del juego. Aparte, utiliza varias funciones de firmware de la calculadora para mostrar imágenes o texto por la pantalla de la calculadora. Estas funciones fueron integradas en subrutinas propias: el mostrado de imágenes, que mueve la imagen a RAM segura antes de imprimirla, o la escritura de texto, que lo imprime carácter a carácter, recibiéndolos por registro.

En primer lugar, se situó la imagen del maestro de ceremonias en la esquina superior izquierda de la pantalla, tal y como aparece en el programa original. Después, tras comprobar que las líneas de texto originales contenían demasiados caracteres como para mostrar toda la frase al lado del *Ring Master*, se distribuyeron las palabras en más líneas, de hasta 18 caracteres de 5x7 píxeles cada uno, que es lo mínimo posible en la pantalla de la TI-84 Plus.

Acto seguido se creó la subrutina "*WRITESCREENTEXT*" con el propósito de simular el efecto de desplazamiento de texto conocido como "*scrolling text*". Dicha subrutina emplea un *buffer* de dimensiones 18 caracteres de ancho por 8 líneas de alto, el cual almacena las frases que se desean mostrar en pantalla. La subrutina introduce la siguiente línea de datos almacenados en la memoria de frases en la línea más baja del *buffer*, y simultáneamente desplaza hacia arriba el texto de cada línea, de tal manera que cada línea ocupe la posición que anteriormente ocupaba la línea superior. Este proceso se repite sucesivamente para simular el efecto de desplazamiento de texto en la pantalla. Un inconveniente de este enfoque es que la fuente de tamaño reducido del sistema operativo de Texas Instruments es que no es monoespaciada, por lo que una línea de *buffer* vacía no elimina una línea en la que previamente había habido texto. Esto se subsanó imprimiendo un rectángulo con todos los píxeles apagados, a modo de parche.

Ilustración 14. Captura de la introducción, desde el emulador *Wabbitemu*

En la introducción del juego, es esencial detectar las pulsaciones de las teclas, ya que el jugador debe decidir si desea mostrar los controles o comenzar el juego de inmediato. Para lograr esto, se utiliza la función de firmware llamada *_GetKey* para registrar las teclas pulsadas de forma bloqueante[2], y luego se comprueba el registro A para conocer el código de la tecla pulsada.

5.2.3 Movimiento y detección de teclas pulsadas

Para el movimiento del jugador, es necesario comprobar las teclas de las flechas arriba, izquierda y derecha y actualizar la posición del jugador de acuerdo con el movimiento a realizar.

Previo a la actualización de la posición del jugador, resulta necesario registrar cuál fue la última tecla que ha sido presionada. Para tal efecto, no se puede utilizar la función *_GetKey*, ya que es bloqueante y pausaría el programa indefinidamente hasta que se detectase una tecla, no pudiendo ejecutar el bucle del juego mientras tanto. Por tanto, se utiliza la función *GETKD*/*GETKDH*, desarrollada por Joe Pemberton [35]. Esta función, además de no ser bloqueante, hace posible la identificación de teclas modificadoras, incluyendo la tecla *2ND*. A diferencia de *_GetKey*, conviene mencionar que esta guarda la información de la tecla registrada en el registro B.

Una vez creado el bucle infinito del videojuego, se llama a *GETKD* para obtener la tecla más recientemente pulsada. En base a esto, se bifurca el programa dependiendo de la acción a realizar. Primero, se comprueba si se ha pulsado *2ND* + *MODE*, lo que en la calculadora es *QUIT*. En caso afirmativo, se finaliza la ejecución del programa. Si no, se comprueban las demás teclas.

Para el movimiento del jugador, se requiere verificar el estado de las teclas de dirección correspondientes a las flechas arriba, izquierda y derecha. En el caso de la flecha **arriba**, es necesario examinar la dirección actual del jugador, la cual se encuentra almacenada en la etiqueta PD ocupando un byte, y posteriormente actualizar la coordenada X o Y del jugador, ya sea incrementándola o decrementándola según corresponda.

En lo que respecta a la flecha **izquierda**, debido a la elección de utilizar los números en el rango [0,3] para denotar la dirección del jugador, girar a la izquierda implica incrementar el valor de PD y actualizarlo en la memoria. También es preciso verificar si al incrementar el valor se sale del rango (en cuyo caso se alcanzaría el valor 4) y, de ser así, asignarle el valor 0.

En cuanto a la flecha **derecha**, se sigue un proceso similar, pero en lugar de incrementar el valor de PD, se procede a decrementarlo. Asimismo, es necesario revisar si hay desbordamiento (por ejemplo, si se disminuye el valor de 0 a -1, que en 8 bits sería 255) y, en caso afirmativo, cambiar el valor a 3.

---

[2] Una detección de teclas bloqueante significa que se detiene la ejecución del programa hasta que se presiona una tecla.

5.2.4 Visión 3D

Se ha seguido el procedimiento original de renderizado, explicado en el código desensamblado [22]. Para facilitar la obtención de las celdas en frente del jugador, dependiendo de su orientación, se utiliza un *buffer*, denominado *SEENMAZE*, que almacena una matriz de 7x3, siendo cada elemento de esta una celda del laberinto de las posiciones de enfrente del jugador, como se puede ver en la Ilustración *15*.

Ilustración 15. Representación gráfica de SEENMAZE

De esta manera, es más sencillo recorrer *SEENMAZE* que el laberinto entero, ya que la dirección es indiferente, puesto que la subrutina *GETSEENMAZE* ya se encarga de copiar la parte frontal vista del laberinto al *buffer*, dependiendo de la dirección.

Una vez se actualiza *SEENMAZE*, se pintan las 7 secciones en el *buffer* de la pantalla, *SCREEN*.

Primero, para la sección 0, se dibujan dos columnas de píxeles en cada extremo, dependiendo de si las posiciones 0 y 14 del *buffer* contienen una pared o un pasillo. A continuación, para pintar las siguientes secciones, primero se comprueba si la celda de enfrente del jugador es una pared. En caso afirmativo, se pinta totalmente la sección en patrón de ajedrez (si la suma de las coordenadas del píxel es impar, se pinta, si no, se deja en blanco). Si no hay una pared enfrente, se dibujan las paredes o pasillos a la izquierda y derecha.

En el proceso de dibujar columnas en la pantalla, se emplean diversas subrutinas para colorear los píxeles de la pantalla en negro o en una alternancia de blanco y negro. Estas subrutinas reciben parámetros que indican la posición X relativa al borde izquierdo de la pantalla, la altura de la línea a dibujar y la posición final X del dibujo, en caso de ser necesario trazar un rectángulo. Un ejemplo de esta situación se presenta al dibujar pasillos laterales en lugar de paredes.

El resultado de este modo de mostrado en pantalla se puede ver en la Ilustración *16*.

Ilustración 16. Captura de la visión tridimensional obtenida, vista desde el emulador *Wabbitemu*

El mostrado de la visión 3D se genera a una velocidad suficiente como para no ralentizar el bucle del juego, a pesar de realizar una extensa sucesión de escrituras de píxeles individuales en cada ejecución de la subrutina.

5.2.5 Generación de laberintos

Esta subrutina se desarrolló tras finalizar y comprobar la de visión 3D para poder comprobar que los laberintos se generaban correctamente.

El mayor problema de esta función reside en que está originalmente escrita en lenguaje BASIC, por lo que se tuvo que traducir, manteniendo toda la funcionalidad. Para este fin se consultó la guía de "Términos y sentencias más usuales", incluida en el embalaje del Sinclair ZX81 (Anexo 2: Guía rápida de las instrucciones BASIC del Sinclair ZX81).

De la misma manera, se tuvo que implementar una rutina generadora de números pseudoaleatorios que realizase la función de *RND()* en BASIC.

El algoritmo de generación de laberintos creado por Malcolm Evans se caracteriza por una serie de pasos secuenciales y aleatorios. En primer lugar, todas las celdas del laberinto se convierten en paredes (_MW), creando así una estructura inicial homogénea. A continuación, se entra en un bucle en el que se selecciona de manera aleatoria una dirección (norte, sur, este u oeste) y una longitud de pasillo (de 1 a 6 unidades). A partir de estas variables, se inserta un pasillo desde la última posición conocida en la dirección seleccionada. Cabe destacar que esta posición inicial se encuentra en la esquina sureste del laberinto, en las coordenadas (14,16).

Se tratan de introducir 800 celdas pasillo en dicho bucle. Si bien esta cantidad supera el número de celdas del laberinto, se espera que muchos pasillos se sobrescriban en el proceso, lo que hace que el número de pasillos sea adecuado para crear un laberinto complejo y variado.

Con el fin de depurar esta subrutina se implementó una rutina para imprimir el laberinto convirtiendo cada celda en un píxel. Se podría utilizar como indicador de carga en la generación del laberinto, pero su uso ralentiza exponencialmente la tarea de crear de laberintos.

5.2.6 Implementación de la lógica completa del videojuego

Como se ha mencionado en el apartado de Lógica modificada, fue posible reutilizar casi en su totalidad el código del videojuego original relacionado con la lógica del videojuego, es decir, los movimientos del monstruo, la velocidad de los movimientos, y la utilización de los *flags* para controlar el avance del juego.

Otro ejemplo de código modificado es el tiempo de retardo para la ejecución del bucle principal, para que se realice con la subrutina *WAIT_HALFSEC*, que recibe un parámetro en el registro B. Dicho parámetro tiene un rango de valores entre 0 y 255, y a medida que este valor aumenta, se produce un mayor retardo, llegando a un tiempo máximo aproximado de medio segundo.

5.3 Implementación de ray casting

Una vez convertido el juego original a la Texas Instruments TI-84 Plus, se procede a la mejora de los gráficos 3D por medio de *ray casting*. Para facilitar la corrección de errores y la adición de posibles mejoras en el programa original, se decide implementar el trazado de rayos como una opción de renderizado que el usuario puede seleccionar cuando inicia el programa, pulsando la tecla *MODE* para utilizar el renderizado original o pulsando la tecla *DEL* para jugar con el modo ray casting. Esto se marca poniendo a 1 el *flag* del primer bit de la variable *RENDERFLAGS*.

Una vez hecha esta bifurcación, el programa sigue su flujo original, pero a la hora de ejecutar las acciones del *GAMELOOP*, se comprueba que el *flag* que indica si el renderizado es por *ray casting* o no, y si es así, realiza varias acciones de manera distinta a la original.

El movimiento del jugador en el modo ray casting implica mover su posición con más exactitud, lo que significa que tendrá una posición dentro de la celda sobre la que esté. Esto se almacena en las variables *TEMPPX* y *TEMPPY*, que guardan, en 2 Bytes, la posición de jugador con una resolución de 512 unidades por cada celda.

5.3.1 Funcionamiento del ray casting

Se evaluó implementar el algoritmo de proyección de rayos utilizando Análisis Diferencial Digital [36], o DDA por sus siglas en inglés, pero se determinó que el *ray cast* por pasos era más simple de implementar. Esta subrutina de proyección de rayos se basa en el programa de demostración desarrollado por Hans Törnqvist [37].

El funcionamiento de este método de renderizado es muy simple: se proyectan rayos con punto de inicio igual a la posición precisa del jugador, en un arco de amplitud igual al FOV (campo de visión) utilizado. El número de rayos

proyectados en ese rango definirá la resolución de visión del entorno. Por cada rayo, se entra en un bucle, aumentándose su posición X e Y en un valor o paso predefinido, aumentando la variable que almacena su distancia, comprobando paralelamente si la nueva posición se encuentra en una celda pared. Si es así, se procede a dibujar el rayo en base a la distancia que se ha calculado. Si continúa en una celda pasillo, se continúa aumentando el rayo. En la Ilustración *17* se representan 3 rayos proyectados a modo de ejemplo. Los puntos verdes indican la posición de cada rayo tras cada paso. Así, cada rayo, de izquierda a derecha, tendrá una longitud de 4, 5 y 2 unidades de distancia.

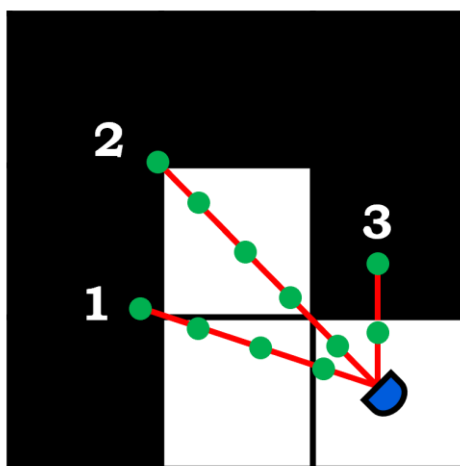


Ilustración 17. Representación gráfica de la proyección de tres rayos por pasos

La visualización del entorno con únicamente estos tres rayos quedaría tal y como se muestra en la Ilustración *18*. Se resalta en rojo la pared en la que intersecan los rayo 1 y 2, para clarificar la interpretación. La altura de la columna representada es inversamente proporcional a la distancia del rayo proyectado. Esta relación se realiza con una tabla de consulta o "*look-up table*" predefinida, en la cual se asigna a cada distancia una altura de columna, con una progresión de valores no lineal, para corregir el posible efecto ojo de pez que ocurriría si fuera lineal.

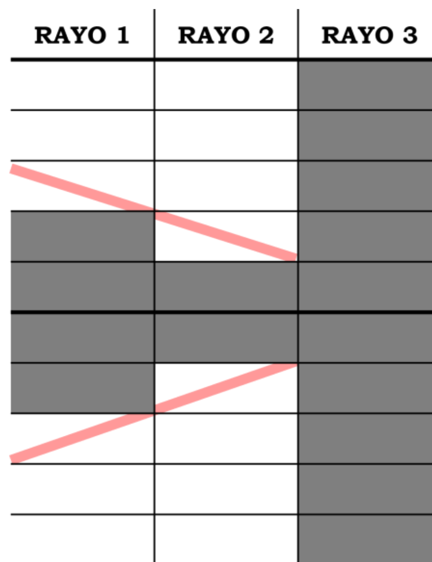


Ilustración 18. Mostrado de los tres rayos proyectados a modo de ejemplo

Con una mayor resolución de trazado de rayos, se consigue una imagen que es más fácilmente entendible, como la captura del resultado final en la Ilustración *19*. En la imagen de la izquierda se han añadido guías en rojo para comprender mejor la profundidad del dibujo.

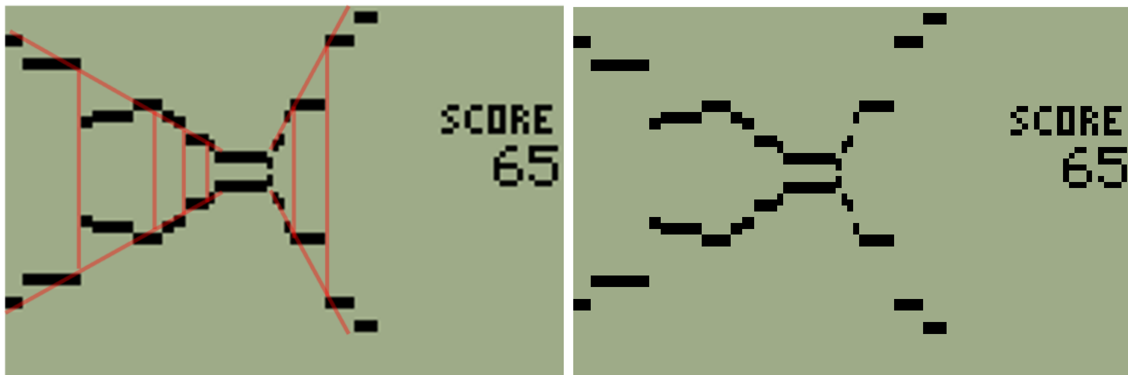


Ilustración 19. Capturas de la visión tridimensional obtenida con ray casting, vista desde el emulador Wabbitemu

Esta técnica de proyección de rayos es muy fácil de implementar si se tienen funciones que calculan funciones trigonométricas, pero el procesador Z80 no tiene esta capacidad. Por eso, se tuvo que implementar estas dos funciones, adaptándolas a la equivalencia de 360º a 512 unidades, realizada para estandarizar los ángulos.

Además, un inconveniente que presenta esta técnica con respecto al DDA es que en situaciones en las que se encuentran dos celdas pared en diagonal, es posible que el rayo atraviese la zona en la que son adyacentes, como se puede ver en la Ilustración *20*. En este caso, no presenta un gran problema puesto que el mapa o laberinto no crea muchas situaciones similares a estas, y si lo hace, lo más probable es que haya una pared detrás, minimizando el efecto.

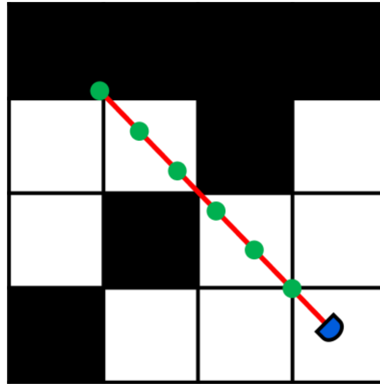


Ilustración 20. Problema del trazado de rayos con paredes en diagonal

Es por esto por lo que, en ciertas esquinas, el rayo atraviesa la primera pared y su distancia aumenta ligeramente, haciendo que haya una discrepancia con los puntos adyacentes. Esta situación se puede ver en la Ilustración *21*. El jugador se encuentra frente a un callejón sin salida, en el que las celdas contiguas a la primera pared frente al jugador son celdas pasillo.

Ilustración 21. Captura de la interfaz del videojuego en la que se ve un pasillo sin salida, vista desde el emulador Wabbitemu

Una forma de solucionar el problema sería imponer un paso de aumento de rayo ínfimo, para comprobar al más mínimo detalle si el rayo ha atravesado una pared, pero no es una solución eficiente ni viable. La mejor solución sería utilizar DDA, cuya forma de operar previene este fallo [36].

En la Ilustración *22* se representa la técnica de trazado de rayos aplicada al caso de *3D Monster Maze*, siendo el semicírculo azul la posición del jugador y las líneas en rojo más gruesas los ángulos máximos de trazado de rayos. En rojo transparente se muestra el área que cubierta por los rayos trazados desde el jugador.

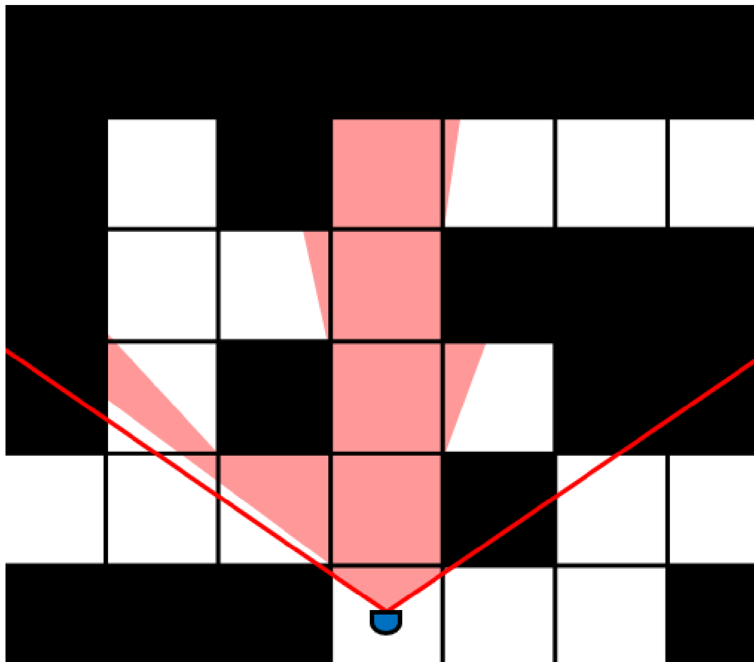


Ilustración 22. Representación gráfica de los rayos trazados desde el jugador (figura azul)

El código final de la subrutina de trazado de rayos se adjunta a continuación.

```
DRAW3DVIEW_RC:
    XOR A
    LD (INCANGLEDIV),A
    ; Ángulo inicial de proyectado de rayos (playerAngle - (FOV / 2))
    LD HL,(PDD)
    LD BC,FOV/2
    ADD HL,BC

    CALL CLAMPANGLE ; Fit angle into [0,512] range
    LD (CASTINGANGLE),HL

    ; Iniciar a 0 el contador de la columna a pintar en pantalla
    XOR A
    LD (SCREENCOLUMN),A

RAYCASTCOLUMNLOOP:
        ; Iniciar posición del rayo a la del jugador
        LD HL,(PDX)
        LD (RAYX),HL ; Posición rayo en X
        LD HL,(PDY)
        LD (RAYY),HL ; Posición rayo en Y

        ; Obtener pasos trigonométricos
        LD HL, (CASTINGANGLE)
        CALL GETTRIGS

        ; Iniciar a 0 la distancia del rayo actual
        XOR A
```

```
        LD (RAYLENGTH), A

RAYCASTINGLOOP:
        ; Actualizar coordenada X e Y
        LD HL, (RAYX)
        LD A, (COS)
        CALL STEP
        LD (RAYX), HL
        LD HL, (RAYY)
        LD A, (SIN)
        CALL STEP
        LD (RAYY), HL

        ; Aumentar distancia o longitud del rayo
        LD A,(RAYLENGTH)
        INC A
        LD (RAYLENGTH), A

        ; Guardar posición entera, no precisa
        LD HL, (RAYX)
        CALL REMOVEFPM
        LD (TEMPPDX), A
        LD HL, (RAYY)
        CALL REMOVEFPM
        LD (TEMPPDY), A
        ; Comprobar si el rayo se encuentra en una celda pared
        CALL CHECKHIT
        OR A ; Comprobar si retorna 0
        JR Z,RAYCASTINGLOOP

        LD A,(RAYLENGTH) ; Altura de la columna
        LD HL,HEIGHT
        LD B,0
        LD C,A
        ADD HL, BC
        LD A,(HL)
        LD C,A
        CP (GAME_H/2)+1
        JR Z,OUTOFDISPLAY
        LD A,(GAME_H/2)+1
        SUB C
        LD C,A

        LD A,(SCREENCOLUMN) ; Posición X en pantalla
        LD B,A

       ; Comprobar si se encuentra en la celda salida
        PUSH BC
        CALL CHECK_HIT_EXIT
        OR A ; Comprobar si retorna 0
        POP BC
        JR Z,NORMALDRAW_RC
        CALL DRAWVLINE_RAY_EXIT
        JP OUTOFDISPLAY
```

```
NORMALDRAW_RC:
        CALL DRAWVLINE_RAY

OUTOFDISPLAY:
        ; Actualizar ángulo del rayo
        LD HL, (CASTINGANGLE)
        DEC HL
        DEC HL ; x2 el FOV

        ; Saltar columnas cuyo índice es divisible entre 5
        ; Así se eliminan 28 angles para encajar la vista en la pantalla
        LD A,(INCANGLEDIV)
        CP A,4
        JP NZ,ANGLENODIVBY4
        LD A,0
        INC HL
ANGLENODIVBY4:
        INC A
        LD (INCANGLEDIV),A
        CALL CLAMPANGLE
        LD (CASTINGANGLE), HL

        ; Aumentar la columna y repetir el proceso
        LD A, (SCREENCOLUMN)
        INC A
        LD (SCREENCOLUMN), A
        CP GAME_W
        JP NZ, RAYCASTCOLUMNLOOP

    CALL DISPSCREEN ; Mostrar la vista
    RET
```

Como se puede apreciar, la subrutina es muy compacta, incluso más que la del renderizado original. Cabe destacar que la obtención de los valores trigonométricos se hace a través de otras funciones, que acceden a una tabla de consulta con los valores precalculados, para mejorar la eficiencia.

A continuación, se detalla el funcionamiento de la subrutina paso a paso:

1. Se inicializa la variable *INCANGLEDIV* a 0, la cual sirve para saltarse el dibujado de algunos rayos y así conseguir que el amplio FOV utilizado para emular el del juego original quepa en los 64 píxeles de ancho del área de juego en la pantalla.

2. Se realiza una serie de cálculos para convertir la dirección del jugador (*PDD*) ajustándola en base al campo de visión (*FOV*) para obtener el primer ángulo de trazado (*CASTINGANGLE*). Esto implica restarle al ángulo del jugador actual el *FOV* entre 2. En el diagrama de la Ilustración *23* se aprecia esta operación sobre el ángulo de dirección del jugador.

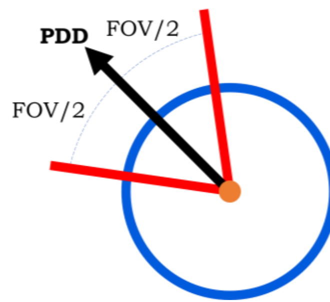


Ilustración 23. Campo de visión del jugador con respecto a su dirección

3. Se restablece el contador de columnas (*SCREENCOLUMN*) a cero. Este parámetro indica la posición en la coordenada X de la pantalla en la que se dibujará la línea vertical que se requiera.

4. Se inicia un bucle (*RAYCASTCOLUMNLOOP*) que recorre todas las columnas de la pantalla.

   Dentro del bucle se realizan las siguientes acciones:

   1. Se copian las coordenadas X e Y del jugador en las variables de ubicación del rayo (*RAYX* y *RAYY*).

   2. Se obtienen los valores trigonométricos correspondientes al ángulo de trazado (*CASTINGANGLE*) mediante la función auxiliar *GETTRIGS*.

   3. Se actualizan las coordenadas X e Y del rayo sumándoles los pasos correspondientes según los valores trigonométricos.

   4. Se incrementa la longitud del rayo (*RAYLENGTH*) en uno.

   5. Se guarda la ubicación del rayo sin formato de punto flotante en variables temporales (*TEMPPX* y *TEMPPY*).

   6. Se verifica si el rayo ha impactado en una pared mediante la llamada a la función *CHECKHIT*.

   7. Se verifica si el rayo ha impactado en la salida mediante la llamada a la función *CHECK_HIT_EXIT*.

   8. Si se ha producido impacto, se procede a pintar la columna correspondiente en la pantalla mediante la llamada a la función *DRAWVLINE_RAY*. La altura de la columna se calcula en base a la longitud del rayo.

   9. Se actualiza el ángulo de trazado (*CASTINGANGLE*) incrementándolo en una unidad, para pintar la siguiente columna de la pantalla.

10. Se realiza un control de bucle para verificar si se ha recorrido todas las columnas de la pantalla.

5. Una vez finalizado el bucle, se realiza una llamada a la función *DISPSCREEN* para mostrar el resultado del renderizado en la pantalla y se retorna de la subrutina.

En resumen, partiendo de la base de que es código ensamblador y no se tienen instrucciones para realizar operaciones complejas, es un código muy compacto que cumple su cometido sin ningún problema y de forma muy eficiente. Las subrutinas auxiliares también son relativamente simples.

5.3.2 Movimiento del jugador dentro de la celda

Al mismo tiempo que se implementó la subrutina de mostrar la vista 3D con *ray casting*, se modificó ligeramente la función que mueve al jugador y asigna valores a su orientación (*PD*), para que también actualizase la **orientación precisa** del jugador (*PDD*), asignando valores fijos para norte, sur, este y oeste. De esta manera, se mantiene la técnica de movimiento del juego original, pero no consigue aprovechar del todo este método de renderizado.

Una vez finalizada la renderización por medio de *ray casting*, era necesario implementar el movimiento libre del jugador dentro de cada celda del laberinto. Para esto, se creó otra función que ejecutaría los movimientos sólo cuando se estuviera en modo *ray casting*. En esta subrutina, las teclas de las flechas de la calculadora incrementan o decrementan el valor de PDD, así como el valor de *PDX* o *PDY* según se requiera. Después, **se ajusta el valor** de *PD* o *PX/PY* al número entero que corresponda.

Para el giro, el ajuste se realiza manualmente con unos rangos de 128 unidades para cada orientación (ya que la orientación se mide en un círculo de 512 unidades). Para el movimiento, cada celda equivale a 512 unidades de alto por 512 unidades de ancho, por lo que, si el número en *PDX* o *PDY* va a ser incrementado a un valor divisible entre 512, se actualiza la posición entera *PX* o *PY*.

Este cambio sí que requiere una modificación mucho más profunda de la lógica del videojuego, así que se optó por hacer las mínimas modificaciones para mantener su esencia. Se creó un bucle dentro de las acciones que se ejecutan en el bucle de juego, el cual se ejecuta 4 de cada 5 veces que se ejecuta el bucle de juego. De esta forma se puede conseguir que, en modo *ray casting*, se sondee la tecla pulsada y se mueva el jugador varias veces antes de ejecutar el resto del bucle, además de diferenciar la subrutina a la que se llama para hacer el movimiento del jugador.

Otras modificaciones necesarias para el correcto funcionamiento del modo *ray casting* tuvieron que ver con el mostrado de Rex. En el original, se calculaba la posición en la que imprimirlo en base al tamaño del *sprite*. Pero con el giro libre, esto no puede ser así. Para subsanar esto, se modificó la subrutina que muestra el Sprite de Rex para que tuviera en cuenta el giro del jugador (variable PDD), y

se ajuste la posición en el eje horizontal en la que se comienza a imprimir el *sprite* de Rex.

Ilustración 24. Rex apareciendo en la vista dibujada con trazado de rayos, vista desde el emulador Wabbitemu

Con esta implementación, se permite el giro libre del jugador a cualquier ángulo, y el movimiento libre en línea recta a través de todo el laberinto, con el inconveniente de que se interrumpe este movimiento cada segundo para poder ejecutar el bucle principal, que mueve a Rex, actualiza la puntuación y muestra el mensaje de estado del monstruo, entre otras acciones.

6 Evaluación

Se ha obtenido un **producto muy completo**, que posee la gran mayoría de las características que el juego original, además de añadir la posibilidad de cambiar el modo de renderización al inicio, entre el método original y la versión avanzada con *ray casting*.

Las modificaciones realizadas al código no han afectado a la funcionalidad o rendimiento del programa final y han **facilitado la mejora de gráficos**. Con motivo de simplificar el programa y poder cumplir los objetivos, se han realizado una serie de compromisos como, por ejemplo, simplificando el mostrado de la salida con respecto a la representación del juego original, o la impresión de los *sprites* de Rex, que se realiza con una función firmware de la calculadora en vez de copiarla al buffer de la pantalla.

Es interesante comprobar que el **rendimiento** de la versión con trazado de rayos es mejor que la versión original, además de que es más versátil el trazado de rayos que el renderizado original.

Es altamente probable que la diferencia en el rendimiento entre la versión original y la versión con *ray casting* se derive del hecho de que en la primera versión se requiere la representación de un mayor número de píxeles en la pantalla para exhibir las paredes, lo que ocasiona una ralentización en el procesamiento de cuadros o *frames*. En cambio, en la versión con *ray casting*, solo se efectúa la representación de los bordes superior e inferior de las paredes.

Este aspecto también es esclarecedor en cuanto al inconveniente que supone la imposibilidad de acceder directamente a la memoria de la pantalla LCD. Dicha limitación implica la necesidad de almacenar temporalmente la pantalla en un búfer en la memoria, lo que provoca una **disminución en la velocidad de visualización de imágenes**, ya que estas deben ser mostradas mediante la llamada a una función firmware específica. Asimismo, esto no es un inconveniente en la parte estética, ya que actúa de **doble búfer** para mostrar al usuario la imagen ya renderizada, en lugar de que visualice en tiempo real la escritura de píxeles en la pantalla.

En cuanto al resultado final de la versión con proyección de rayos, para lograr una implementación óptima del ray casting que permita un **movimiento fluido y constante del jugador**, se requerirían una serie de modificaciones sustanciales en todo el programa. Estas modificaciones involucrarían ajustes en el manejo de la posición del jugador, de Rex y de cómo se realiza el bucle de juego. Dado el nivel de cambios necesarios, no sería factible realizar estas modificaciones en el **mismo programa** existente.

Una opción más viable sería desarrollar un **nuevo programa**, específicamente diseñado para utilizar el método de renderizado con ray casting, y que las comprobaciones para imprimir los mensajes de estado de Rex, su movimiento o la actualización de la puntuación se realizasen de una forma más eficiente que permitiese no interrumpir el movimiento del jugador.

En último lugar, realizando una comparación de gráficos en modo de renderizado original con el juego original en Sinclair ZX81, se comprueba que

se ha mantenido la esencia de este, y la experiencia de juego es prácticamente idéntica.

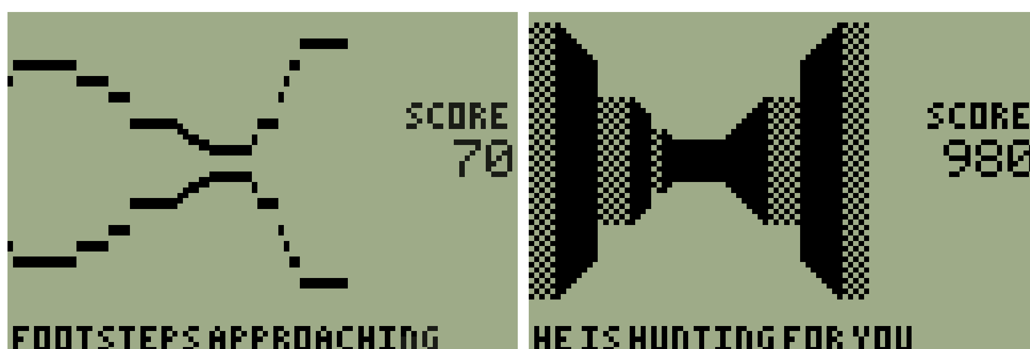


Ilustración 25. Comparación entre la versión normal y la versión ray cast. Ambas ejecutadas con el emulador Wabbitemu

7 Resultados y conclusiones

Se han cubierto los objetivos que se habían planteado inicialmente en este proyecto:

1. Se ha realizado un **análisis detallado** del juego original y su funcionamiento, estudiando el código fuente original e investigando sobre este.

2. Se ha desarrollado una **adaptación del código del juego original** a la arquitectura de la calculadora gráfica, teniendo en cuenta sus limitaciones y refactorizando partes de código incompatibles, como las relacionadas con la interacción con los periféricos o la memoria.

3. Se ha incorporado la técnica del **trazado de rayos** en la adaptación del juego original en la TI-84 Plus. Esto mejora la visualización del laberinto con respecto a la versión original.
4. Se han propuesto una serie de desarrollos para **mejorar la experiencia de juego**, como la mejora de los controles aportada en el modo de renderizado con *ray casting*, o el propio nuevo modo de renderizado.

5. Finalmente se ha podido comprobar su **funcionamiento correcto y estable** en calculadoras gráficas emuladas y físicas.

Se ha concluido que el desarrollo de aplicaciones 3D en ensamblador tiene muchas **limitaciones**, y conlleva una cantidad de trabajo que no compensa frente a lenguajes de más alto nivel como C o Java, ya que existen compiladores para estos lenguajes que generan código máquina extremadamente eficiente.

Aunque algunos programadores como Chris Sawyer lograron crear grandes videojuegos 3D en ensamblador, como por ejemplo *RollerCoaster Tycoon* 1 y 2 [38], actualmente no hay razones para usar este lenguaje salvo que el sistema objetivo solo admita ensamblador y no existan compiladores o alternativas.

En lo personal, el desarrollo de un proyecto de gran envergadura como la creación de un videojuego en 3D en lenguaje ensamblador ha contribuido significativamente a mi comprensión de este lenguaje y de otros de nivel superior. Esto se debe a que la mayoría de los programas, quitando las numerosas capas de abstracción, se ejecutan en ensamblador. Al dedicar una considerable cantidad de tiempo a este proyecto, uno se familiariza con un enfoque orientado al funcionamiento interno de la computadora y se busca constantemente formas de mejorar la eficiencia. Además, se adquiere una mejor comprensión del papel fundamental que desempeña la memoria RAM en los programas informáticos, así como su gestión y manipulación mediante instrucciones concretas, al igual que es posible hacerlo en el lenguaje de programación *C* [39], por ejemplo.

Finalmente, resulta pertinente señalar que la elaboración de un videojuego como el presente en lenguaje ensamblador **no constituye una forma eficiente**

de desarrollo en la actualidad, y se debería recurrir a esta opción solo en casos en los que no se disponga de alternativas en lenguajes de más alto nivel. Es importante destacar que las calculadoras más recientes tienen la capacidad de ejecutar programas en lenguajes como *C* o *Python*, lo cual ofrece un enfoque más accesible y práctico para el desarrollo de aplicaciones.

7.1 Líneas de trabajo futuro

Como cualquier videojuego, es posible que se encuentre algún comportamiento incorrecto, fallo o *bug*, por lo que será necesario *parchearlos* y actualizarlo.

En cuanto al propio videojuego, se podrían **mejorar los flujos de ejecución**, ya que no son intuitivos y no son muy controlables. Así, se podría utilizar una herramienta de *profiling* (perfilador de programas) para analizar las subrutinas más costosas y optimizarlas.

Para la parte de trazado o proyección de rayos, una mejora relevante podría ser la conversión de esta subrutina para que utilice **Análisis Diferencial Digital** (DDA), como se ha mencionado en el apartado Funcionamiento del ray casting, para mejorar su rendimiento y arreglar los problemas de la técnica utilizada, expuestos en el mismo apartado.

Resultaría sumamente interesante realizar la conversión del juego original a una **calculadora gráfica de mayor capacidad**, como las ofrecidas por Texas Instruments que están basadas en el procesador Motorola 68000, como la familia *TI-89*, *TI-92* o *TI Voyage 200*. Asimismo, también sería relevante considerar aquellas calculadoras basadas en el nuevo procesador Zilog eZ80, como la TI-84 Plus CE. Estas calculadoras más avanzadas brindarían una plataforma óptima para aprovechar al máximo las capacidades gráficas y de procesamiento, permitiendo así una experiencia de juego mejorada y más sofisticada, añadiendo incluso la posibilidad de crear programas a color.

Por último, se puede partir de este trabajo como inspiración para portar o **convertir juegos antiguos a otras plataformas**, ya sean convencionales como un ordenador o cualquier otro aparato con pantalla y algún método de interacción.

8 Análisis de impacto

El impacto potencial de los resultados obtenidos con la conversión del videojuego *3D Monster Maze* a una calculadora gráfica se realiza desde diferentes perspectivas: personal, empresarial, social, económica, medioambiental y cultural. Se ha tratado de identificar los beneficios esperados, así como de los posibles efectos perjudiciales.

En lo que respecta al impacto personal, la realización de este TFG ha resultado en una experiencia altamente enriquecedora. A través del proyecto, se ha adquirido nuevos conocimientos y habilidades en áreas como la programación y la electrónica. Además, se ha tenido la oportunidad de aplicar conocimientos previos y habilidades adquiridas en un proyecto concreto y de gran envergadura, lo que me ha permitido consolidar y mejorar las habilidades prácticas.

Asimismo, el proyecto ha brindado la posibilidad de experimentar con diversas herramientas y técnicas, así como de desarrollar habilidades fundamentales como el pensamiento crítico, la resolución de problemas y la toma de decisiones. Esta experiencia ha sido de gran valor para el desarrollo personal, ya que ha proporcionado una comprensión más profunda de los videojuegos y su impacto en la cultura y la sociedad.

En términos de utilidad, este proyecto ha permitido aplicar los conocimientos y habilidades adquiridos a lo largo de la formación académica a un proyecto visual, práctico y concreto. También ha proporcionado una experiencia valiosa que seguramente resultará beneficiosa en el futuro profesional. La motivación generada por este proyecto ha sido significativa y la satisfacción obtenida al completar este desafío ha sido enriquecedora.

En cuanto al impacto socioeconómico, la publicación de juegos reconocidos en este tipo de plataformas contribuye al aumento del uso de dichos dispositivos, así como a la popularización del videojuego convertido, e incluso puede incrementar el valor de reventa de las calculadoras. Además, se brinda acceso a la experiencia de juego a un mayor número de personas.

Por otro lado, se permite dar una segunda vida a estas calculadoras, las cuales, a pesar de su amplio uso en la actualidad, cuentan con hardware que comienza a quedar obsoleto. Aunque la fabricación de calculadoras y la venta del juego pueden generar residuos electrónicos, la conversión del juego puede tener un impacto positivo en el medio ambiente al proporcionar una alternativa sostenible a los videojuegos modernos, los cuales consumen gran cantidad de energía. Al mismo tiempo, se facilita la reutilización de estas calculadoras, que muchas veces son desechadas cuando dejan de ser utilizadas en el campo de la educación, se previene de convertirlas en residuos electrónicos.

En el contexto de los Objetivos de Desarrollo Sostenible (ODS), la conversión del juego puede contribuir al logro de metas específicas. En primer lugar, se destaca su contribución al ODS 4 (Educación de calidad), al fomentar el desarrollo de habilidades cognitivas y motoras en los usuarios a través de la experiencia de juego proporcionada.

Finalmente, la conversión del videojuego *3D Monster Maze* en una calculadora antigua puede tener un impacto significativo en diferentes contextos, como el personal, empresarial, social, económico, medioambiental y cultural. En cuanto a los beneficios esperados, es interesante destacar la provisión de una experiencia única para los usuarios, la creación de un mercado potencial para la venta de la calculadora y el juego, la posibilidad de que cualquier persona pueda disfrutar de videojuegos históricos y el fomento del desarrollo de habilidades cognitivas y motoras en los usuarios.

Bibliografía

[1] G. d. l. Fuente, «MUSEO INFORMÁTICO de laEscuela de Ingeniería Informática de la Universidad de Valladolid,» Universidad de Valladolid, 31 August 2017. [En línea]. Available: https://museo.inf.uva.es/?0=Z80. [Último acceso: 16 May 2023].

[2] E. O. Duque, «MUSEO INFORMÁTICO de la Escuela de Ingeniería Informática de la Universidad de Valladolid,» Universidad de Valladolid, 31 August 2017. [En línea]. Available: https://museo.inf.uva.es/index.php?0=Sinclair%20ZX81. [Último acceso: 16 May 2023].

[3] D. M. G. Preethichandra, «Z80—The 1970s Microprocessor Still Alive,» *IEEE Micro,* vol. 41, nº 6, pp. 156 - 157, 2021.

[4] M. Evans, «3D Monster Maze,» JK Greye Software, 1982.

[5] W. contributors, «3D Monster Maze,» Wikipedia, The Free Encyclopedia., 6 March 2023. [En línea]. Available: https://en.wikipedia.org/w/index.php?title=3D_Monster_Maze&oldid=1143241614. [Último acceso: 13 April 2023 20:42 UTC].

[6] J.K. Greye Software, «3D Monster Maze (19xx)(J.K. Greye Software),» Archive.org, 9 August 2021. [En línea]. Available: https://archive.org/details/3D_Monster_Maze_19xx_J.K._Greye_Software. [Último acceso: 2023 April 13].

[7] M. Evans, «Trashman,» New Generation Software, 1984.

[8] World of Spectrum, «World of Spectrum June 2017,» June 2017. [En línea]. Available: https://ia800604.us.archive.org/view_archive.php?archive=/1/items/World_of_Spectrum_June_2017_Mirror/World%20of%20Spectrum%20June%202017%20Mirror.zip&file=World%20of%20Spectrum%20June%202017%20Mirror/sinclair/zx81/games-inlays/MonsterMaze3D.jpg. [Último acceso: 13 April 2023].

[9] t. community, «ticalc.org,» ticalc.org, [En línea]. Available: https://www.ticalc.org/pub/83plus/asm/games/. [Último acceso: 13 April 2023].

[10] J. Martin, *Pong 1.2,* 2006.

[11] A. El-Helw, *Tetris Beta,* 1999.

[12] A. Guinamard y R. Siryani , «Doom for Ti 83,» ticalc.org, 19 June 2002. [En línea]. Available: https://www.ticalc.org/archives/files/fileinfo/238/23843.html. [Último acceso: 2 March 2023].

[13] D. Toaster, «Snakecaster: A 3D Nibbles Game (2012),» June 2012. [En línea]. Available: https://www.ticalc.org/archives/files/fileinfo/449/44963.html. [Último acceso: 13 April 2023].

[14] ZX Gaming, «Monster Maze,» [En línea]. Available: https://www.zx-gaming.co.uk/games/monstermaze/default.htm. [Último acceso: 13 April 2023].

[15] D. Curran, «Remaking 3D Monster Maze for the Commodore PET,» Tynemouth Software, 17 July 2022. [En línea]. Available: http://blog.tynemouthsoftware.co.uk/2022/07/remaking-3d-monster-maze-for-the-commodore-pet.html. [Último acceso: 17 April 2023].

[16] Texas Instruments, *TI-84 Plus graphing calculator,* Texas Instruments.

[17] Meriam Library - California State University Chico, *TI-84 Plus Silver Edition.*

[18] M. McFarland, «The unstoppable TI-84 Plus: How an outdated calculator still holds a monopoly on classrooms.,» *Washington Post,* 2014.

[19] G. Hilliard, «&> /dev/null,» 06 October 2021. [En línea]. Available: https://www.thirtythreeforty.net/posts/2021/10/ti-calculator-innovation/. [Último acceso: 12 May 2023].

[20] F. Tołkaczewski, «From Symbolism to Realism. Physical and Imaginary Video Game Spaces in Historical Aspects,» *Homo Ludens,* vol. 1, nº 12, pp. 193--212, 2019.

[21] TVTropes, «TVTropes,» TVTropes, [En línea]. Available: https://tvtropes.org/pmwiki/pmwiki.php/VideoGame/ThreeDMonsterMaze. [Último acceso: 15 April 2023].

[22] P. Farrow, «3D Monster Maze Disassembly,» ZX81 Resource Centre, 2016. [En línea]. Available: http://www.fruitcake.plus.com/Sinclair/ZX81/Disassemblies/MonsterMaze.htm. [Último acceso: February 2023].

[23] Soft Tango, «3D Monster Maze Dissected,» Soft Tango, [En línea]. Available: https://softtangouk.wixsite.com/soft-tango-uk/3d-monster-maze. [Último acceso: 13 April 2023].

[24] S. Holdsworth, «3D Monster Maze,» zx81stuff, [En línea]. Available: http://www.zx81stuff.org.uk/zx81/tape/3DMonsterMaze(NGS). [Último acceso: 18 April 2023].

[25] P. Bourke, «RLE - Run Length Encoding,» August 1995. [En línea]. Available: https://www.dlsi.ua.es/~carrasco/papers/RLE%20-%20Run%20length%20Encoding.html. [Último acceso: 20 April 2023].

[26] Texas Instruments, «TI-83 Plus SDK Documentation,» 28 May 2002. [En línea]. Available: https://education.ti.com/en/guidebook/details/en/830D08FF31804AEAA2F03B8F5E89AD14/83psdk. [Último acceso: 2 April 2023].

[27] I. B. (Imanolea), *Z80 Assembly for Visual Studio Code,* 2016.

[28] K. Martian, *jsTIfied,* Cemetech, 2013.

[29] . T. DUPONCHELLE y B. MOODY, *TilEm2,* http://lpg.ticalc.org/, 2012.

[30] S. Putt, C. Shappell y J. Montelongo, *Wabbitemu,* http://wabbitemu.org/, 2006.

[31] K. 'Akuyou', «ChibiAkumas,» Learn Assembly Programming... With ChibiAkumas!, 2020. [En línea]. Available: https://www.chibiakumas.com/z80/ti83.php.

[32] Microsoft, *Paint 3D,* Microsoft, 2015.

[33] J. Wingbermuehle, «Ion v1.6,» [En línea]. Available: https://www.ticalc.org/archives/files/fileinfo/130/13058.html. [Último acceso: 27 April 2023].

[34] Learn @ Cemetech, «z80:Intro to Flash Applications,» [En línea]. Available: https://learn.cemetech.net/index.php?title=Z80:Intro_to_Flash_Applications. [Último acceso: 27 April 2023].

[35] J. Pemberton, *Direct Input Routines for the 83/83+,* 2002.

[36] L. Vandevenne, «Lode's Computer Graphics Tutorial - Raycasting,» 2004. [En línea]. Available: https://lodev.org/cgtutor/raycasting.html. [Último acceso: Mayo 2023].

[37] H. Törnqvist, *Raycast demo!,* 2001.

[38] C. Sawyer, «Chris Sawyer Games FAQ,» 2003. [En línea]. Available: http://www.chrissawyergames.com/faq3.htm. [Último acceso: 05 May 2023].

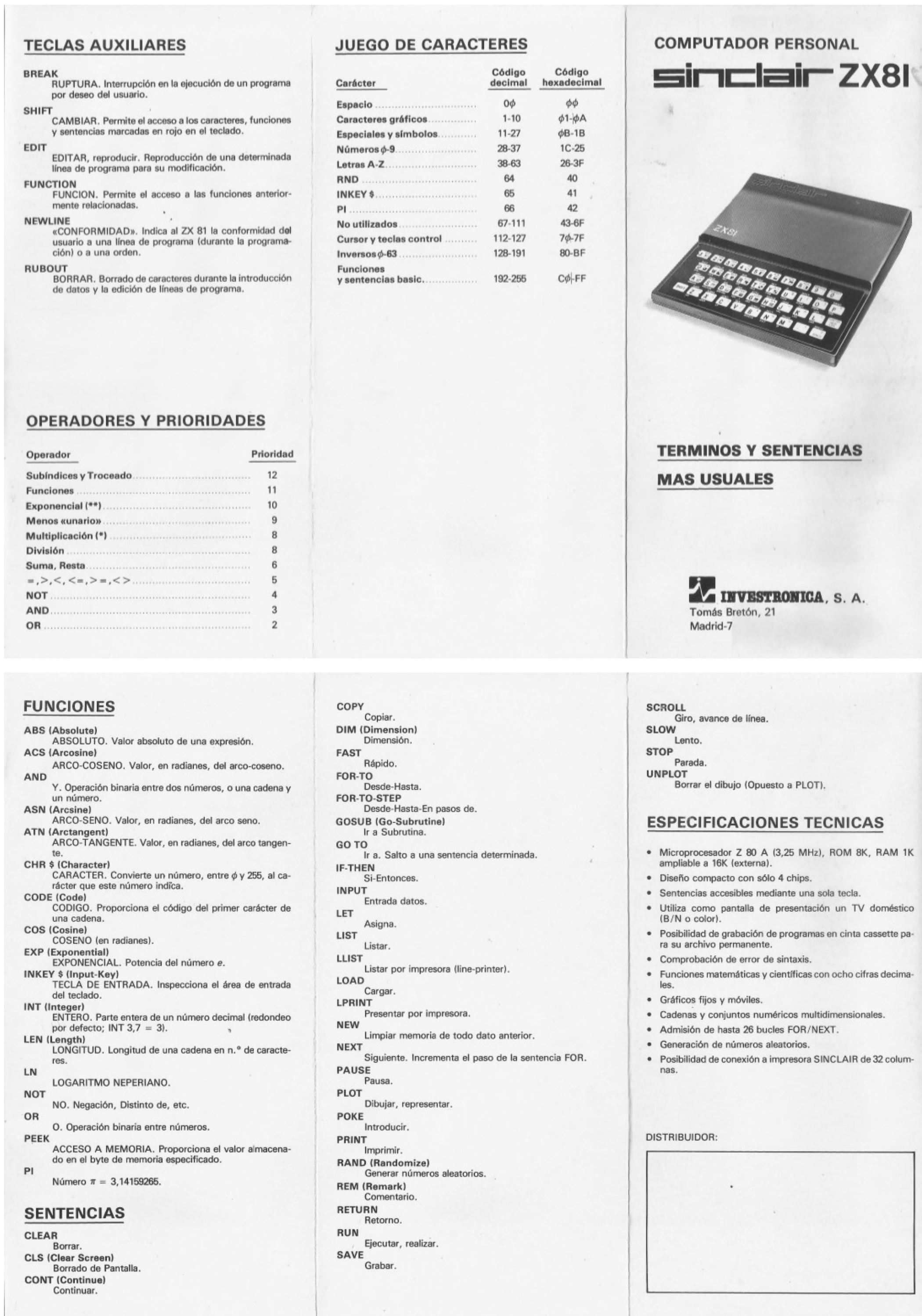
[39] Linux Documentation, «Linux man page,» Linux, [En línea]. Available: https://linux.die.net/man/3/malloc. [Último acceso: 18 May 2023].

[40] P. Jones, «3D Monster Maze - Spectrum Computing,» Spectrum Computing, [En línea]. Available: https://spectrumcomputing.co.uk/entry/28617/ZX81/3D_Monster_Maze. [Último acceso: 13 April 2023].

Anexo 1: Código del transcriptor de imagen PNG a bits (Java)

```
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;
import java.io.File;
import java.io.IOException;

public class ImageToBinary {
  public static void main(String[] args) {
    // Read the image file
    BufferedImage image = null;
    try {
      image = ImageIO.read(new File("/REX/3dmm_rex8.png"));
    } catch (IOException e) {
      System.out.println("Error reading image file");
      System.exit(1);
    }

    // Get image dimensions
    int width = image.getWidth();
    int height = image.getHeight();
    System.out.println(".DB "+height+","+width);

    int countLine = 0;

    // Iterate over all pixels
    for (int y = 0; y < height; y++) {
      for (int x = 0; x < width; x++) {
        int color = image.getRGB(x, y);
        int red = (color >> 16) & 0xff;
        int green = (color >> 8) & 0xff;
        int blue = color & 0xff;
        // If the pixel is not white (255, 255, 255), set it to 0
        // Otherwise, set it to 1
        int binary = (red == 255 && green == 255 && blue == 255) ? 0 : 1;

        if(countLine==0) System.out.print(".DB %");
        else if(countLine%8 == 0) System.out.print(",%");
        System.out.print(binary);
        countLine++;
      }
      //Add padding to unfinished final bytes (happens if image width is
not divisible by 8)
      if (countLine%8 != 0){
        for(;countLine%8 != 0;countLine++) System.out.print("0");
      }
      countLine = 0;
      System.out.println();
    }
  }
}
```

Anexo 2: Guía rápida de las instrucciones BASIC del Sinclair ZX81



TECLAS AUXILIARES

BREAK
RUPTURA. Interrupción en la ejecución de un programa por deseo del usuario.

SHIFT
CAMBIAR. Permite el acceso a los caracteres, funciones y sentencias marcadas en rojo en el teclado.

EDIT
EDITAR, reproducir. Reproducción de una determinada línea de programa para su modificación.

FUNCTION
FUNCION. Permite el acceso a las funciones anteriormente relacionadas.

NEWLINE
«CONFORMIDAD». Indica al ZX 81 la conformidad del usuario a una línea de programa (durante la programación) o a una orden.

RUBOUT
BORRAR. Borrado de caracteres durante la introducción de datos y la edición de líneas de programa.

OPERADORES Y PRIORIDADES

Operador	Prioridad
Subíndices y Troceado	12
Funciones	11
Exponencial (**)	10
Menos «unario»	9
Multiplicación (*)	8
División	8
Suma, Resta	6
=,>,<,<=,>=,<>	5
NOT	4
AND	3
OR	2

JUEGO DE CARACTERES

Carácter	Código decimal	Código hexadecimal
Espacio	0ϕ	ϕϕ
Caracteres gráficos	1-10	ϕ1-ϕA
Especiales y símbolos	11-27	ϕB-1B
Números ϕ-9	28-37	1C-25
Letras A-Z	38-63	26-3F
RND	64	40
INKEY $	65	41
PI	66	42
No utilizados	67-111	43-6F
Cursor y teclas control	112-127	7ϕ-7F
Inversos ϕ-63	128-191	80-BF
Funciones y sentencias basic	192-255	Cϕ-FF

COMPUTADOR PERSONAL

sinclair ZX81

TERMINOS Y SENTENCIAS MAS USUALES

INVESTRONICA, S. A.
Tomás Bretón, 21
Madrid-7

FUNCIONES

ABS (Absolute)
ABSOLUTO. Valor absoluto de una expresión.

ACS (Arcosine)
ARCO-COSENO. Valor, en radianes, del arco-coseno.

AND
Y. Operación binaria entre dos números, o una cadena y un número.

ASN (Arcsine)
ARCO-SENO. Valor, en radianes, del arco seno.

ATN (Arctangent)
ARCO-TANGENTE. Valor, en radianes, del arco tangente.

CHR $ (Character)
CARACTER. Convierte un número, entre ϕ y 255, al carácter que este número indica.

CODE (Code)
CODIGO. Proporciona el código del primer carácter de una cadena.

COS (Cosine)
COSENO (en radianes).

EXP (Exponential)
EXPONENCIAL. Potencia del número *e*.

INKEY $ (Input-Key)
TECLA DE ENTRADA. Inspecciona el área de entrada del teclado.

INT (Integer)
ENTERO. Parte entera de un número decimal (redondeo por defecto; INT 3,7 = 3).

LEN (Length)
LONGITUD. Longitud de una cadena en n.º de caracteres.

LN
LOGARITMO NEPERIANO.

NOT
NO. Negación, Distinto de, etc.

OR
O. Operación binaria entre números.

PEEK
ACCESO A MEMORIA. Proporciona el valor almacenado en el byte de memoria especificado.

PI
Número π = 3,14159265.

SENTENCIAS

CLEAR
Borrar.

CLS (Clear Screen)
Borrado de Pantalla.

CONT (Continue)
Continuar.

COPY
Copiar.

DIM (Dimension)
Dimensión.

FAST
Rápido.

FOR-TO
Desde-Hasta.

FOR-TO-STEP
Desde-Hasta-En pasos de.

GOSUB (Go-Subrutine)
Ir a Subrutina.

GO TO
Ir a. Salto a una sentencia determinada.

IF-THEN
Si-Entonces.

INPUT
Entrada datos.

LET
Asigna.

LIST
Listar.

LLIST
Listar por impresora (line-printer).

LOAD
Cargar.

LPRINT
Presentar por impresora.

NEW
Limpiar memoria de todo dato anterior.

NEXT
Siguiente. Incrementa el paso de la sentencia FOR.

PAUSE
Pausa.

PLOT
Dibujar, representar.

POKE
Introducir.

PRINT
Imprimir.

RAND (Randomize)
Generar números aleatorios.

REM (Remark)
Comentario.

RETURN
Retorno.

RUN
Ejecutar, realizar.

SAVE
Grabar.

SCROLL
Giro, avance de línea.

SLOW
Lento.

STOP
Parada.

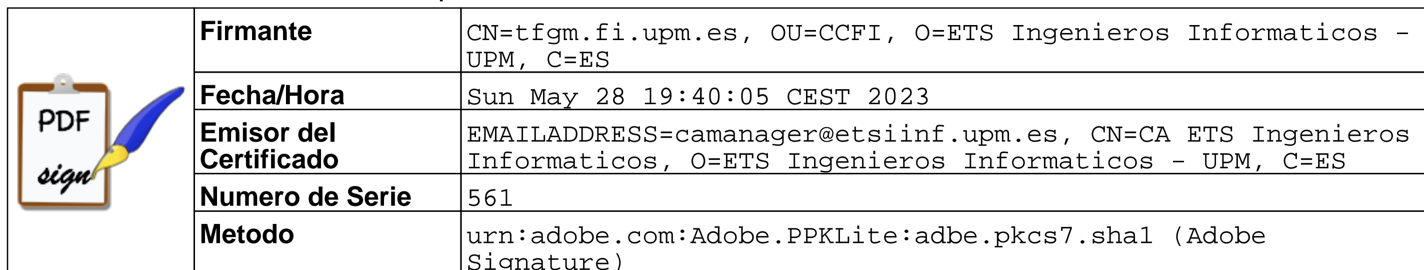
UNPLOT
Borrar el dibujo (Opuesto a PLOT).

ESPECIFICACIONES TECNICAS

- Microprocesador Z 80 A (3,25 MHz), ROM 8K, RAM 1K ampliable a 16K (externa).
- Diseño compacto con sólo 4 chips.
- Sentencias accesibles mediante una sola tecla.
- Utiliza como pantalla de presentación un TV doméstico (B/N o color).
- Posibilidad de grabación de programas en cinta cassette para su archivo permanente.
- Comprobación de error de sintaxis.
- Funciones matemáticas y científicas con ocho cifras decimales.
- Gráficos fijos y móviles.
- Cadenas y conjuntos numéricos multidimensionales.
- Admisión de hasta 26 bucles FOR/NEXT.
- Generación de números aleatorios.
- Posibilidad de conexión a impresora SINCLAIR de 32 columnas.

DISTRIBUIDOR:

Este documento esta firmado por



	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Sun May 28 19:40:05 CEST 2023
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)