Escuela
Politécnica
Superior

Playdate game development in C

Bachelor's degree in Multimedia Engineering

Bachelor's Thesis

Author:
Alberto Benavent Ramón
Supervisor:
Francisco José Gallego Duran

Universitat d'Alacant
Universidad de Alicante

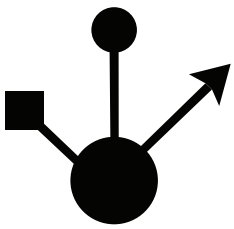
July 2021

Playdate game development in C

Development and documentation of a videogame for the Playdate console

Author
Alberto Benavent Ramón

Supervisor
Francisco José Gallego Duran
Ciencia de la Computación e Inteligencia Artificial



Bachelor's degree in Multimedia Engineering

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, July 2021

Abstract

Playdate is a new handheld console developed by Panic that will be launched in 2021. Its objective is offering a unique and surprising experience for videogame enthusiasts, and for that reason, it presents unconventional characteristics: in addition to the common directional and action buttons, it features a reflective monochromatic screen that draws only in pure black and white, an accelerometer, and a crank on its side that acts as a controller.

This Bachelor's Thesis will explore the videogame design possibilities stemming from such a peculiar hardware through the creation of several prototypes, covering all supported programming languages and evaluating them. In terms of performance, the Playdate falls on the modest side; for that reason, the focus will be in low-level programming to obtain the best performance possible. The knowledge acquired during this first phase will be applied to the development of a full game in C, "TinySeconds".

"TinySeconds" is a 2D platformer game where each level must be completed under 2.5 seconds. In addition to that limit, to progress to the next world the player must complete all the levels of the previous one in succession under an overarching time limit. This makes the game a frenetic experience with great replayability, as it invites the player to practice and improve their completion times. In addition to the time limit, different types of obstacles add variety to the levels making use of characteristics unique to the console, such as the crank.

In addition to documenting the development of these projects, a C programming tutorial for Playdate will be included, teaching the basic principles of configuring the programming environment in Windows and developing a sample program. This chapter has the aim of supplying the lack of documentation about C programming for Playdate in a Windows platform, as the official manual is centered around the Lua language in Mac environments.

Resumen

Playdate es una nueva consola portátil desarrollada por Panic que será lanzada al mercado en 2021. Su objetivo es ofrecer una experiencia distinta y sorprendente a entusiastas de los videojuegos, y por ello, presenta características poco convencionales: además de los habituales botones direccionales y de acción, tiene una pantalla monocroma reflectante en blanco y negro puros, acelerómetro, y una manivela en el lado que sirve como controlador.

En esta memoria, se explorarán las posibilidades de diseño de videojuegos que ofrece un hardware tan peculiar mediante la creación de diversos prototipos, cubriendo los diferentes lenguajes de programación que soporta y realizando una evaluación de los mismos. A nivel de hardware es una consola de potencia modesta, por lo que se optará por la programación a bajo nivel para buscar el mejor rendimiento posible. Este conocimiento adquirido será después aplicado al desarrollo de un juego completo en C, "TinySeconds".

"TinySeconds" es un videojuego de plataformas en vista lateral donde cada nivel debe ser completado en menos de 2,5 segundos. Además, para poder progresar de un mundo al siguiente, los niveles de un mismo mundo deben ser superados consecutivamente en un tiempo limitado. Esto dota al juego de un ritmo frenético y de gran rejugabilidad al invitar a los jugadores a practicar para mejorar sus tiempos. Además de la limitación temporal, diversos tipos de obstáculos añaden variedad a los niveles utilizando características propias de la consola como la manivela.

Además de documentar el desarrollo de estos proyectos, se desarrollará un tutorial de programación en C para Playdate, instruyendo los principios básicos de configuración del entorno de programación en Windows, y desarrollando un programa de ejemplo. Este capítulo nace para suplir la falta de documentación oficial sobre programación en C para la consola en un entorno Windows, ya que los recursos oficiales se centran en el lenguaje Lua y entornos Mac.

Acknowledgments

This work would not have been possible without the support and affection I received from my environment during its development.

I would like to thank my supervisor Francisco José Gallego for his guidance during the development of this Bachelor's Thesis, and for sharing the passion and knowledge he holds for videogames with his students.

To the friends I have made during university and the lifelong ones I brought along: thank you for making me enjoy these past five years as much as I have. You have all been a constant source of joy and support, and I cannot wait to live many more adventures together.

Thanks to my family for listening to my ramblings about the development of this thesis, for being the most loving and supporting, and for starring in my happiest memories; you have shaped me into the person I am today.

So, entertaining stories, fun game systems... These already exist in this world.
I want to see what is beyond that wall.
Whatever you wanna call it,
it's the space where no one has entered yet.

Yoko Taro.

Contents

List of Figures

Listings

1. Introduction

"Developing for Playdate" is an introduction to software development for the upcoming handheld console Playdate, written before its public launch during the developer preview.

The contents of this Bachelor's Thesis are intended to be a reference point for future developers interested in this hardware, as well as a chronicle of my prototypes, experiments, and learning process, all culminating in the development of a full game.

Most of the contents are centered around programming in the C language with the objective of gaining low-level knowledge of the hardware, consciously developing from this perspective to maximize performance, and transmitting the lessons learned from this experience to the reader. It also aims to cover the less documented area of C language development in Windows for the console, as most of the available resources are centered around Lua programming and Mac environments.

Each of the prototypes and demos created will strive to explore the device's strengths and limitations, finding new design opportunities in them, and incorporating them into gameplay. "TinySeconds", the main game developed in this Bachelor's Thesis, will benefit from the experience gained in the prototypes phase to design engaging gameplay and innovative interactions tailored to the hardware's features.

"TinySeconds" is a 2D platforming game with puzzle elements centered around completing levels within a short time limit. This mechanic asks of the player quick reaction times and boosts replayability by challenging them to complete levels and worlds in the least amount of restarts.

In addition to this chronicle, the thesis includes a chapter written like a conventional tutorial, which will guide newcomers to the console through the first steps of C development for Playdate in Windows. This chapter emphasizes the main ways of achieving performance on the device and includes exercises to practice and expand the concepts explained in it.

This thesis also narrates the experience of creating games during a hardware's production phase, a process which included features and specifications being revealed or changed during development, as well as the reporting of bugs and errors contributing to the console's Quality Assurance (QA).

2. Justification and objectives

When the Playdate console was announced in may 2019, I instantly became enamored with the simplicity and freshness of its proposal; oftentimes, creativity is boosted by limitation, and while the Playdate is a console of modern sensibilities, it still is restricted hardware when compared to modern consoles or PC. Its ability to be programmed in a low-level language, C, was an opportunity to apply the knowledge acquired studying Multimedia Engineering, which made me consider it a perfect fit for my Bachelor's Thesis.

In the summer of 2020, I got the chance to participate in the Playdate Developers Preview, a program that granted me access to the console and SDK before launch. I realized there was very little documentation for the C API and figured that my Bachelor's Thesis could be a helpful resource for other developers after me.

So, I decided to develop my Bachelor's Thesis around researching and developing for the Playdate and writing useful documentation for developers interested in C coding for this new console.

The list of objectives for this thesis is as follows:

- Analyze the Playdate console in regards to software, hardware, SDK and documentation.
- Create small videogame prototypes while learning to develop for this console.
- Design and implement a complete game that makes use of the Playdate's characteristics.
- Test the game with real users and iterate based on the received feedback.
- Develop learning resources for C programming for Playdate.

3. Theoretical framework

3.1. Playdate

The Playdate (3.1) is an upcoming portable console created by Panic; a software development company specialized in Mac applications with prior experience in the videogame industry as a publisher for the titles “Firewatch” and “Untitled Goose Game”. It was first announced on May 22nd, 2019[1], alongside the launch of its official website (`https://play.date`).

Figure 3.1: Playdate console model

Instead of competing for mainstream attention, the Playdate is aimed towards independent developers and enthusiasts. A collection of more than 24 games made by prominent figures in the game development scene (such as Keita Takahashi, Bennett Foddy, and Chuck Jordan) comes bundled with the purchase. The involvement of renowned creators, plus the device being an open platform to develop and publish games for, sparked significant interest among its target audience[2].

Swedish firm Teenage Engineering designed the hardware aspect of the console, and some of its standout characteristics are its 1-bit black and white screen and the crank on the right side of the device, which functions as a controller.

[1] Playdate reveal tweet, `https://twitter.com/playdate/status/1131307504116174848`

[2] Over 70,000 mailing list sign-ups in the first 24 hours and thousands of developer emails (@playdate, 2019)

3.1.1. Hardware specifications

The screen used is a SHARP Memory LCD display, which combines matrix technology with a one-bit memory circuit embedded into every pixel, so image information is retained once it is written (SHARP, n.d.). In addition to the screen being already very energy efficient, this per-pixel memory allows for further energy saving and refresh rates above 50 Hz when draw calls are optimized to render only the changing portions of the screen. Another distinctive characteristic is the highly reflective quality of the display, which makes it suitable to play under direct sunlight; on the other hand, the impossibility of adding backlight to this type of screen makes it unfit for poor lighting conditions. With a 400 x 240 px resolution, and considering the device's small size, the image appears crisp and well-defined.

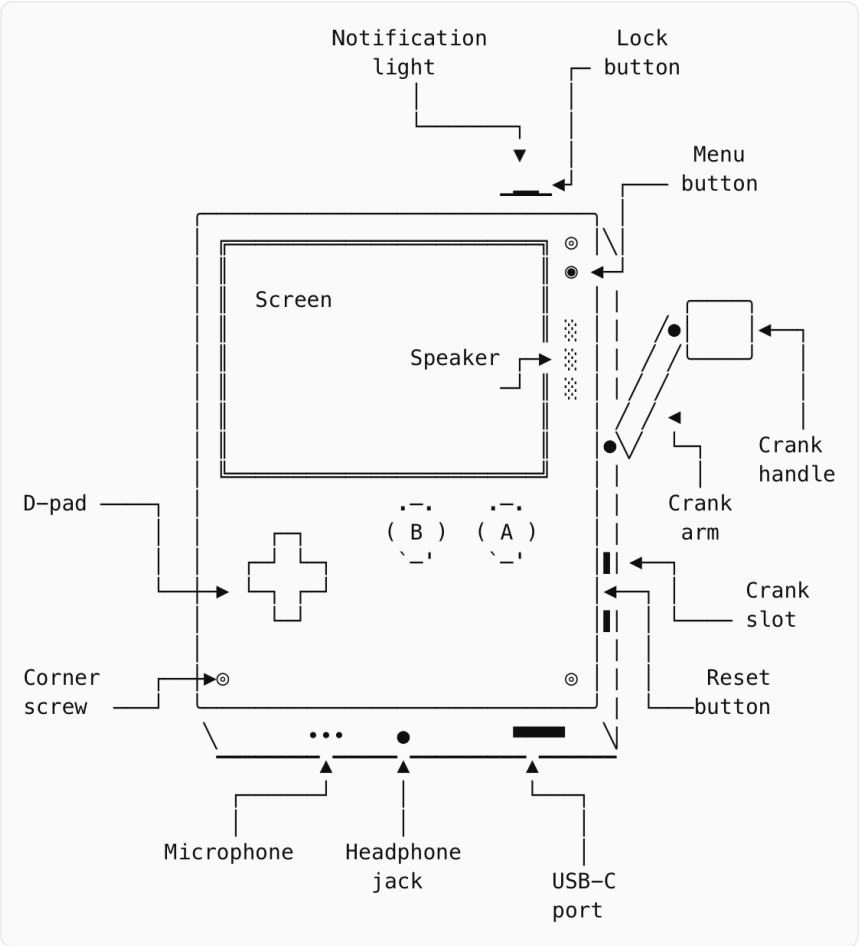


Figure 3.2: Hardware elements diagram (Panic, 2020a)

Regarding input, the Playdate has an eight-way D-Pad, two buttons labeled A and B, a pause menu button, a lock button, an accelerometer, a microphone, and most importantly, the crank. The crank is attached to a rotary encoder and can be queried during gameplay to obtain its current angle and acceleration. It is also collapsible and uses a magnet switch to

detect if it is stowed (Lun, 2020).

A complete list of the specifications:

- **Dimensions**: 76mm x 74mm x 9mm.
- **Display**: 2.7-inch, 400 × 240 (173 ppi) Sharp Memory LCD.
- **Refresh rate**: Up to 50Hz for full-screen drawing, higher when drawing on less pixel rows.
- **CPU**: 180 MHz Cortex M7
- **Memory**: 16 MB of external RAM plus 320 KB of on-board RAM.
- **Storage**: 4 GB.
- **Connectivity**: Wi-Fi (b/g/n) @ 2.4 GHz, Bluetooth 4.2, USB-C, headphone jack.
- **Mass**: 86 grams.

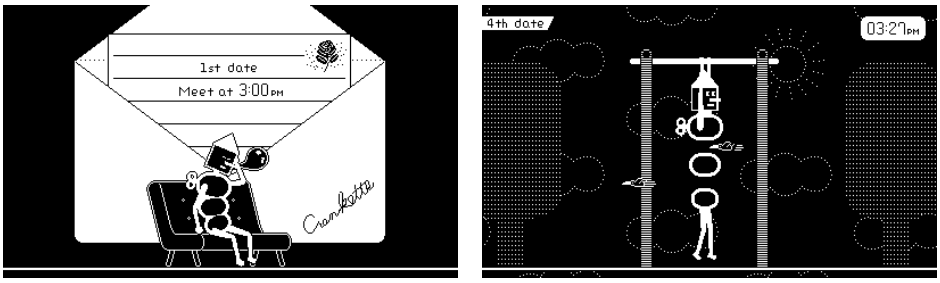
3.2. State of the art

3.2.1. Playdate games

3.2.1.1. *Crankin's Time Travel Adventure*

Crankin's Time Travel Adventure is a game developed by Keita Takahashi, creator of the *Katamari* saga, in collaboration with Panic. It was the first game to be revealed when the Playdate was first announced, and the one used in the early demos, promotional material, and website.

This game is controlled exclusively with the crank, which advances or rewinds time. The main character falls asleep and is late for a date, and the player must protect him on his way to his appointment. Some hazards are unaffected by the alteration of the flow of time, so the player must avoid harm by rewinding to a moment where the main character can't be hit by them. There is also a time limit independent from the rewinding, which prevents users playing in a too cautious way.



(a) Story premise **(b)** Avoiding obstacles

Figure 3.3: Crankin's time-travelling adventure screenshots

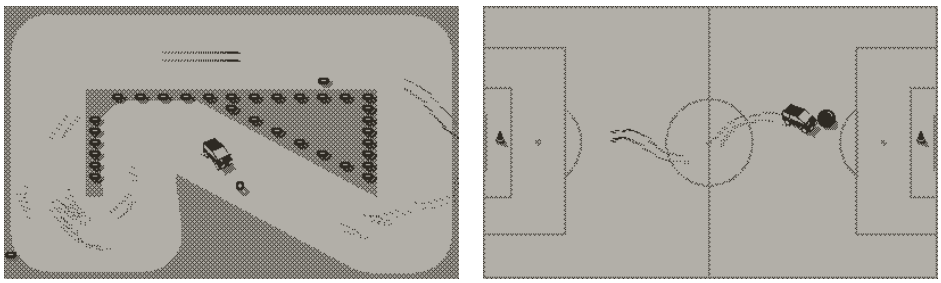
3.2.1.2. *Daily Driver*

Daily Driver is a top-down driving game created by developer Matt Septhon. It features a wide selection of cars and similar vehicles with different physics and appearances.

The cars are pre-rendered images of 3D objects created in *OpenSCAD*, a Computer-Aided Design (CAD) program that allows for creating models by scripting using its own description language. The parts of the model are assigned pure red, green, or blue colors and then rendered from 32 angles around them to obtain a sampled 360º view. Then, the resulting images are batch-processed using *ImageMagick*, an open-source image processing library, separating them into RGB channels and assigning black, white, or a dithering pattern to each channel.

Further into development, additional renders were added for each vehicle to reflect the turning of the wheels and the weight shift on the vehicle. Shadows are implemented by flattening the 3D models of the vehicles along the vertical axis and rendering them for each one of the vehicle sprites.

Gameplay-wise, the cars are controlled with the A button or up arrow for acceleration, the B button or down arrow for breaks, and the crank for drifting. The levels present a variety of challenges in which the player must race on a track, be careful not to hit obstacles, play a soccer game, or collect coins, among other win conditions.



(a) Race track mode **(b)** Soccer mode

Figure 3.4: Daily Driver screenshots

3.2.1.3. *PlayMaker*

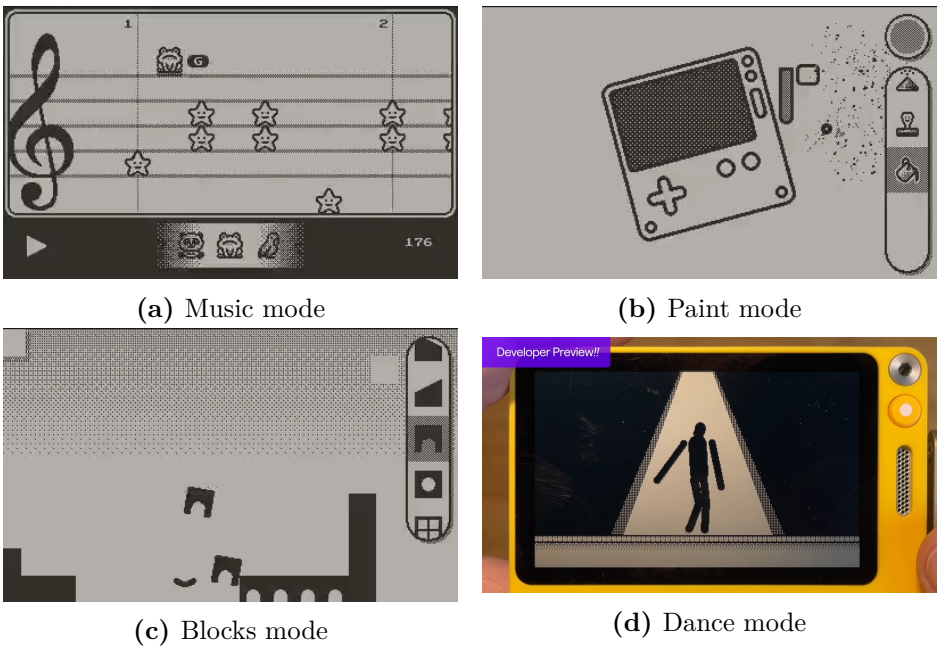
PlayMaker is a creativity toy suite developed by Dustin Mierau. It features the modes music, paint, blocks, and dance, with possible additional ones not yet revealed.

The music mode works similarly to a music box, where the player can place notes on a pentagram choosing their timbre and pitch, and then play the music back by cranking. The tempo depends on how fast the player turns the crank, also allowing to play songs backwards.

The paint mode is a simple bitmap editor, with several painting tools such as a brush with stroke dynamics, a pencil for fine details, a bucket for color filling, an eraser, and a spray painting tool. It features file import and export, allowing the user to upload .gif images to the device and use them in-game.

The blocks mode allows the player to build structures with blocks of various shapes, like boxes and rooftops. Then, the construction can be brought down with an explosion.

Finally, the dance mode consists of a ragdoll figure that reacts to accelerometer movement and crank input, making it twitch and "dance" comically. It is implemented using the library Box2D, using rigid bodies connected with joints to create each part of the character. With the directional pad, the doll can be moved from side to side of the screen, with a theater spotlight following it as it moves.



(a) Music mode

(b) Paint mode

(c) Blocks mode

(d) Dance mode

Figure 3.5: PlayMaker screenshots

3.2.2. Other games

Because no games had been released on Playdate at the time of conceptualizing it, *TinySeconds* takes inspiration from games released before it in other systems. Here is a list of games that have shaped our game in one way or another:

3.2.2.1. *Super Mario 3D World*

Super Mario 3D World is a 3D platforming game developed by Nintendo and released for the Wii U in November 2013. It was the second installment of the *Super Mario 3D* series, which translates the level design philosophies of the classic 2D *Super Mario* games to a 3D perspective. This game served as inspiration for some of the special blocks in our game, namely the toggle blocks and the spring blocks.

Toggle blocks in *TinySeconds* behave similarly to the "Beep Blocks" (fig. 3.6) from *Super Mario 3D World* in that they have two states, solid and intangible, and are often found in the same level with blocks on their opposite state. In the Nintendo game, these blocks change state at a fixed rhythm, while in our game the player controls their state using the crank. This behavior where the player controls this type of block can be compared to "Red-Blue

Panels" (fig. 3.7) from the same game, which toggle their state every time the player jumps.

(a) Pink blocks are solid **(b)** Blue blocks are solid

Figure 3.6: Beep Blocks in Super Mario 3D World

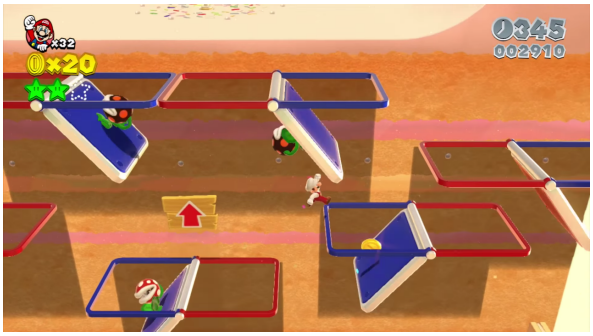


Figure 3.7: Red-Blue panels changing mid-jump

The spring blocks implemented in *TinySeconds* are a common mechanic in platforming games and also appear in *Super Mario 3D World* as the "Mushroom Trampoline" blocks (fig. 3.8). These blocks propel the player in the direction the block is pointing at.

All the mechanics mentioned made their first appearance in *Super Mario Galaxy 2* and continued being used in the *Super Mario 3D* saga.

3.2.2.2. *Rhythm Heaven*

Rhythm Heaven is a saga of rhythm games developed by Nintendo and created by Japanese music producer Mitsuo Terada, better known by his stage name Tsunku. The games consist of many different minigames where the player performs rhythmical actions synchronized to a music track.

At the start of each minigame, a tutorial (3.9) explains its main mechanic and the rhythm pattern it will be based around; some of the minigames use syncopated rhythms, some use audio cues that telegraph actions, and some are based around repetition, among other variations. Then, the mechanic is put in practice in a song, and the player's performance is rated at the end of the minigame.

Minigames are grouped in columns in the main menu (fig. 3.10) and are unlocked sequen-

Figure 3.8: Players bouncing on Mushroom Trampoline blocks

tially once the previous ones have been cleared with an "OK" rank or higher. Then, at the end of each group, a special "Remix" stage is unlocked. This stage does not have a tutorial, and combines the mechanics of the previous minigames in one song with higher difficulty. The "Remix" stages serve as an opportunity for the player to challenge themselves and apply the knowledge acquired up until that point.

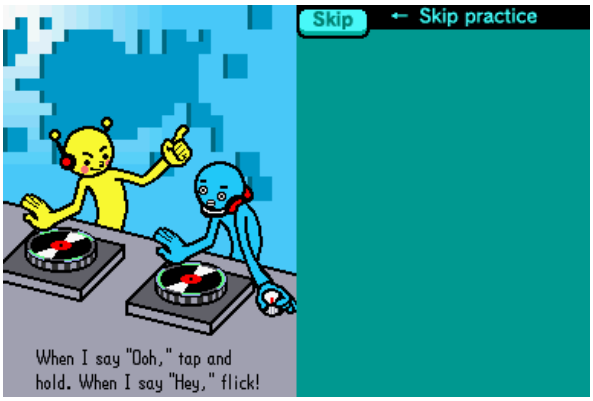


Figure 3.9: Minigame tutorial

"Remix" levels were the main inspiration for *TinySeconds*' structure: each world is made of platforming sections constrained to a single screen and a short timer that resets the level, but to complete each world the player is challenged to clear all the levels in a row without letting the timer run out. That way, completing a world gives a feeling of mastery, as well as an adrenaline rush as the player avoids failure the closer to the end they get.

3.2.2.3. *BOXBOY!*

BOXBOY! is a puzzle-platformer game series developed by HAL Laboratory and published by Nintendo for the Nintendo 3DS system. The player controls a character that can produce boxes and uses them to solve puzzles. The boxes are created stuck to the player, which makes them useful to hang off ledges or as a shield, and can then be dropped on the ground, which can activate switches and other kinds of mechanics.

BOXBOY! was the main inspiration for *TinySeconds*' art style with its mostly solid black

Figure 3.10: Minigame selection screen

or white aesthetic, which values readability above everything else. With *TinySeconds*' main mechanic being the short time frame in which the player must solve each level, platforms and mechanics must be instantly recognizable. Plus, the colors used match the restrictions of the Playdate screen, which makes it an easy comparison.

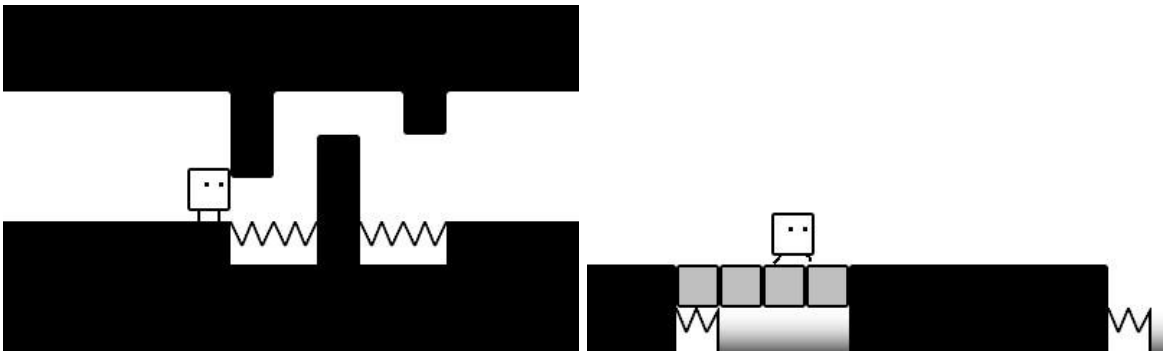


Figure 3.11: *BOXBOY!*'s monochromatic artstyle

3.2.2.4. *Minit*

Minit is an independent videogame developed by Dominik Johann, Jukio Kallio, Jan Willem Nijman, and Kitty Calis. At its core, the game is a retro action-adventure role-playing game (RPG), but its main hook is that after a timer of one minute the player returns to the last visited checkpoint. Progression depends on finding shortcuts, understanding the world, and completing quests, as well as reaching new checkpoints in different areas.

This game was a notable reference when conceptualizing *TinySeconds* because of its time limit aspect, even though the games pertain to different genres. It also features a 1-bit art style akin to the Playdate's capabilities, which made it instantly spring to mind when looking for inspiration.

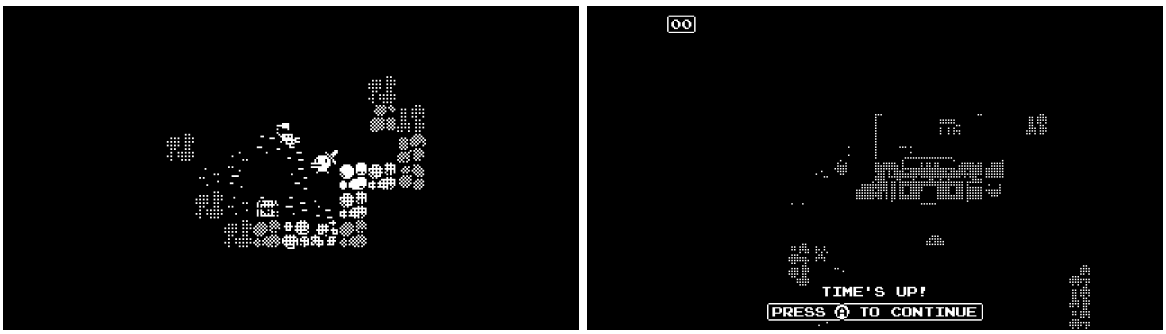


Figure 3.12: Minit screenshots

3.2.3. Conclusion

Our game *TinySeconds* is innovative in the Playdate market, as there are no announced games with comparable mechanics that could be competitors in the fast-paced arcade platformer category. It also introduces a new way of using the crank by collapsing it behind the device, limiting its range to the backside of the console. This position allows for one of the fastest uses of the crank as it can be flicked like a switch without it going outside the player's reach, fixing the common problem of the crank and buttons being difficult to use simultaneously.

It is also special in that it is programmed in C, while the more widely adopted programming language for the console is Lua. C is one of the most low-level languages, with memory being managed manually and code compiling directly to assembly. Lua games, on the contrary, are garbage collected and run on a virtual machine. This difference greatly boosts the game's performance in C, for example, when reading JSON files compared to the same operation in Lua. Even if the end-user is oblivious to the programming language used, good performance will always be welcome, and the development chronicle of the game will be valuable to future Playdate C programmers.

4. Methodology

This project follows an iterative methodology based on prototypes. The development time is divided into iterations that build upon the previous ones, meaning a core implementation of all functionalities will quickly be in place, and enhancements and polish will be added over it in waves. In the first stages of the project, the aim of the iterations will not be to advance the main game but to build quick demos as a way of learning and documenting the usage of the Playdate SDK.

Each iteration is divided into three phases:

1. **Planning**. The first step in each iteration is to establish the objectives that will be pursued during its duration. These must be short-term, concrete goals achievable in a single iteration, which in our case means four weeks of development time. Tasks that extend over one iteration's length should be broken down into smaller objectives, specifying which part of them will be completed in the current term.

2. **Development**. Naturally the longest part of each iteration, in which work is put towards reaching the goals decided in the planning phase. This involves programming the demos or game, and oftentimes stumbling upon errors or impediments that may slow down or change the curse of development. Although undesirable, these diversions can be valuable from a learning perspective and will be collected and reviewed in the third phase of the iterative process.

3. **Analysis and documentation**. Reaching the end of the iteration, some time will be devoted to analyzing and summarizing the lessons learned during the development phase and reviewing performance in terms of having achieved the proposed goals.

Outside of this structure lies the creation of chapter 5, which serves as a guide for new Playdate developers interested in C coding and does not follow the main project's development cycles.

5. Working with Playdate in C

This chapter will be a beginner's guide for developing for Playdate in C. We will cover every step from configuring the C tools for coding and compiling in Windows to creating a simple asteroids game.

This guide assumes a medium understanding of the C programming language. Most concepts will be easy to follow with general programming knowledge, but we will utilize some characteristics specific to C, such as pointers.

The version of the Playdate SDK used in this tutorial will be release 1.0.8, which can be downloaded from official sources (at the time of writing, the Playdate Developer Forums[1]).

5.1. Setting up the environment

Before we get started, some configuration must be done for developing for Playdate in Windows. We will be using the free multipurpose text editor Visual Studio Code, developed by Microsoft, due to its many extensions, ease of use, and task support.

Download Visual Studio Code for Windows[2]. Then open it, and in the sidebar, select the extensions panel. Search for the following extensions and install them:

- C/C++ extension by Microsoft: offers C language support and code completion.
- CMake Tools extension by Microsoft: integrates the compiling pipeline we will be using into the editor.

Once this is done, download and install CMake[3]. CMake is a collection of build tools that will generate the files required by a build system to compile our games. Speaking of which, download Ninja[4] and decompress the zip file, taking note of the directory you extract it to. Ninja is a small, low-level build system focused on fast build times. It relies on CMake for creating the build files for it.

The Playdate has an ARM Central Processing Unit (CPU), so we will need to install a suitable C compiler for this architecture. Download the GCC ARM Toolchain[5] and extract the files as we did with Ninja, taking note of its path.

Once everything is installed, we will create user environment variables to easily reference the necessary paths to these tools. An important thing to note: when writing the paths in the environment variables use forward slashes (/) or escaped backslashes, but not single backslashes.

[1] SDK 1.0.8 download page: `https://devforum.play.date/t/playdate-sdk-1-0-8/1468`
[2] Visual Studio Code download page: `https://code.visualstudio.com/Download`
[3] CMake download page: `https://cmake.org/download/`
[4] Ninja download page: `https://github.com/ninja-build/ninja/releases`
[5] GCC ARM Toolchain download page: `https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads/9-2019-q4-major`

Open the control panel and search for the "Change my environment variables" option. Click on it, and under "user variables", click the "New" button. This way, create a variable named PLAYDATE_SDK that stores the path to the unzipped Playdate SDK folder. Create another variable named PLAYDATE_ARM_GCC and set it to the path to the GCC ARM Toolchain. Finally, create or append to the variable PATH the path to the Ninja build tools.

Next, we are going to adapt the CMake configuration file included with the Playdate SDK for Windows. From the Playdate SDK folder, go to C_API/buildsupport and create the file arm_patched.cmake. Open it with a text editor, and paste the contents of listing 5.1. [6]

Listing 5.1: arm_patched.cmake

```
#
# Toolchain
#

set(CMAKE_SYSTEM_NAME Generic)
set(CMAKE_SYSTEM_PROCESSOR ARM)
set(TOOLCHAIN_PREFIX arm-none-eabi-)
if (WIN32)
    set(TOOLCHAIN_POSTFIX .exe)
else()
    set(TOOLCHAIN_PREFIX "")
endif()
set(TOOLCHAIN_DIR $ENV{PLAYDATE_ARM_GCC})

set(CMAKE_TRY_COMPILE_TARGET_TYPE STATIC_LIBRARY)

set(CMAKE_C_COMPILER ${TOOLCHAIN_DIR}/bin/${TOOLCHAIN_PREFIX}gcc${↩
    ↪ TOOLCHAIN_POSTFIX})
set(CMAKE_CXX_COMPILER ${TOOLCHAIN_DIR}/bin/${TOOLCHAIN_PREFIX}g++${↩
    ↪ TOOLCHAIN_POSTFIX})
set(CMAKE_ASM_COMPILER ${CMAKE_C_COMPILER})

set(CMAKE_OBJCOPY ${TOOLCHAIN_DIR}/bin/${TOOLCHAIN_PREFIX}objcopy${↩
    ↪ TOOLCHAIN_POSTFIX} CACHE INTERNAL "objcopy tool")
set(CMAKE_SIZE_UTIL ${TOOLCHAIN_DIR}/bin/${TOOLCHAIN_PREFIX}size${↩
    ↪ TOOLCHAIN_POSTFIX} CACHE INTERNAL "size tool")

set(CMAKE_FIND_ROOT_PATH ${TOOLCHAIN_DIR}/bin)
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)

set(TOOLCHAIN armgcc)

MESSAGE(STATUS "arm_patched.cmake loaded")
```

5.1.1. Creating a template

Let's create a template we can reuse to build our future projects. For this, we are going to duplicate the Hello World example that comes bundled with the SDK and modify it for Windows and Visual Studio Code. From the Playdate SDKdirectory, navigate to C_API/Examples and duplicate the "Hello World" folder.

[6] Teaching CMake is outside the scope of this tutorial, which is instead centered around Playdate specific development.

Open the folder we just copied and delete the *.nova*, *.xcodeproj*, and *Makefile* files, as they are relative to other editors and build systems we will not be using. We must modify the contents of the *CMakeLists.txt* file to adapt it to the Windows platform. This file tells CMake the location of our source files, the name of the executable we want to build, the version of CMake we want to use, and where to find the CMake files provided by Panic with the SDK. Replace the contents of the file with the following:

Listing 5.2: CMakeLists.txt

```
cmake_minimum_required(VERSION 3.14)
set(CMAKE_C_STANDARD 11)

# Initial Variables
set(SDK $ENV{PLAYDATE_SDK})

# Game Name Customization
set(PLAYDATE_GAME_NAME c_template)
set(PLAYDATE_GAME_DEVICE c_template_DEVICE)

# Source files
file(
    GLOB_RECURSE
    SOURCE_FILES
    src/*.c
)

# Configure project
project(${PLAYDATE_GAME_NAME} C ASM)
add_executable(${PLAYDATE_GAME_DEVICE} ${SDK}/C_API/buildsupport/setup.c ${SOURCE_FILES})

# Make sure we get rid of any existing builds on clean
set_property(DIRECTORY APPEND PROPERTY ADDITIONAL_MAKE_CLEAN_FILES "../${PLAYDATE_GAME_NAME}.pdx" "../Source/pdex.bin")

include(${SDK}/C_API/buildsupport/playdate_game.cmake)
```

Create a new folder at the root of the project called *.vscode*. This directory will hold configuration files that Visual Studio Code will read and use. Inside it, create the file *cmake-kits.json* and populate it with the following:

Listing 5.3: cmake-kits.json

```
[
  {
    "name": "Playdate Device",
    "toolchainFile": "${env:PLAYDATE_SDK}/C_API/buildsupport/arm_patched.cmake"
  }
]
```

This defines a new CMake target that uses the *arm_patched.cmake* file we made in the

previous section.

As a last optional step, we can create tasks for launching common commands easily from the editor. In the *.vscode* folder, create a new *tasks.json* file and add the following contents:

Listing 5.4: tasks.json

```
{
    // See https://go.microsoft.com/fwlink/?LinkId=733558
    // for the documentation about the tasks.json format
    "version": "2.0.0",
    "tasks": [
        {
            "label": "Deploy to Playdate",
            "type": "shell",
            "command": "${env:PLAYDATE_SDK}/bin/pdutil install ${workspaceFolder}/${workspaceFolderBasename}.pdx"
        },
        {
            "label": "Run on Playdate",
            "type": "shell",
            "command": "${env:PLAYDATE_SDK}/bin/pdutil run /Games/${workspaceFolderBasename}.pdx",
            "dependsOn": [
                "Deploy to Playdate"
            ],
            "problemMatcher": [],
            "group": {
                "kind": "build",
                "isDefault": true
            }
        },
        {
            "label": "Mount Playdate",
            "type": "shell",
            "command": "${env:PLAYDATE_SDK}/bin/pdutil datadisk"
        }
    ]
}
```

This defines a "Run on Playdate" task that installs and launches the game in the device by pressing Ctrl+Shift+B, a "Deploy to Playdate" task, which installs the executable in the console but does not launch it, and a "Mount Playdate" task that will open the console's file system in Windows File Explorer[7].

5.1.2. Structure of a Playdate project

Take a look at the template project we configured in the previous section. A typical Playdate C project will have the following structure:

[7] Note that for the first two tasks to work the project's root folder must be named exactly the value of the `PLAYDATE_GAME_NAME` variable in the *CMakeLists.txt* file, and not contain spaces.

- The *build* directory, which stores the CMake and Ninja intermediate build files. Typically, there will be no need to edit or add files in this folder manually.
- The *Source* directory, which contains files that will be packaged alongside our game. This is where images and sound files must be stored, as well as any additional files our game may require (save files, tilemap JSON files, etc.).
- The *src* directory, where we create the source files containing the code of our game. Here is where most of the development happens. Every Playdate project will have in this folder a *main.c* file, which holds the update loop that will execute every frame and the *eventHandler* function, which allows us to react to different types of callbacks such as the game starting, the console locking or unlocking, or the pause menu being opened.
- A *CMakeLists.txt*, which configures CMake for our project, and where we will specify its name and package name.

When we compile our game, an additional folder will appear at the root directory: the .pdx file. This is the package that gets installed on the device and contains the built binaries for all the assets and code.

Open the project folder in Visual Studio Code. If the CMake extension is enabled, a pop-up should appear asking if we want to configure CMake using the *CMakeLists.txt* file. Select "Yes", and then the "Playdate Device" toolkit option on the following drop-down. You can now open the CMakeLists.txt file and change the `PLAYDATE_GAME_NAME` and `PLAYDATE_GAME_DEVICE` variables to the name you want your project to have; the configuration files will update automatically upon saving.

With this last step, we have successfully set up the programming environment.

5.2. Hello World

Let's examine the simplest possible Hello World code.

Listing 5.5: Basic Hello World main.c

```c
#include "pd_api.h"

static int update(__attribute__((unused)) void *ud) {
    return 1; // 1 means refresh the screen, 0 means no refresh.
}

int eventHandler(PlaydateAPI *playdate, PDSystemEvent event, __attribute__((unused)) uint32_t arg) {
    if (event == kEventInit) {
        playdate->system->setUpdateCallback(update, NULL);

        playdate->graphics->clear(kColorWhite);
        playdate->graphics->drawText("Hello World!", strlen("Hello World!"), kASCIIEncoding, 100, 100);
    }

    return 0;
}
```

Here we can see the two obligatory functions: `update()` and `eventHandler()`. When the game starts, the `kEventInit` is received in the `eventHandler()`, and we use it to perform

any required initialization actions. First of all, we indicate to the Playdate Application Programming Interface (API) the function we are going to be using as our update function. Then, we clear the screen with the color white and write the text "Hello World!" at the position x = 100, y = 100.

The update method simply returns 1: an important thing to know is that if the update function returns 0, no drawing is performed on that frame. We need the update method to return 1, or else the text will not be drawn.

Duplicate the template project we created in section 5.1.2, and substitute the contents of the *main.c* file with the code in listing 5.5. Compile the project by using the CMake keybinding F7. You can also compile by navigating to the CMake icon in the left sidebar and clicking the "Build all projects" icon. Now connect your Playdate device and deploy the .pdx, either by using the shortcut Ctrl+Shift+B, the Playdate Simulator, or the pdutil.exe commands, these last two included in the Playdate SDK download.

Once the game is launched in your Playdate, you should see this result:

Figure 5.1: Hello World!

5.2.1. Some improvements

Let's start by defining an alias for the Playdate API: we will create a static pointer of type `PlaydateAPI` and call it `pd`. This will not affect the code's behavior, but it is a common practice in Playdate development and allows us to write less. The `pd` pointer needs to be assigned to the value `playdate` in the `kEventInit` event. We can now substitute all references to the `playdate` variable by this shorthand alias.

Now let's move the drawing function calls to the update method. Even though calling them in the `kEventInit` works, that event should be reserved for initialization purposes, while drawing is usually done at the end of the update method. We can also add an frames per second (fps) indicator with a single line of code using the function `pd->system->drawFPS(x, y)`.

As you can see from testing on the device, the default text font is very thin, with only 1px of font weight. We can change the font to a bold one by using `pd->graphics->loadFont()` and `pd->graphics->setFont()`.

All these changes together leave us with the following *main.c*:

Listing 5.6: Hello World with improvements, main.c

```c
#include "pd_api.h"

static PlaydateAPI *pd = NULL;
static LCDFont *font;

static int update(__attribute__((unused)) void *ud) {
    pd->graphics->clear(kColorWhite);
    pd->graphics->drawText("Hello World!", strlen("Hello World!"), kASCIIEncoding, 100, 100);

    pd->system->drawFPS(0, 0);

    return 1;
}

int eventHandler(PlaydateAPI *playdate, PDSystemEvent event, __attribute__((unused)) uint32_t arg) {
    if (event == kEventInit) {
        pd = playdate;
        pd->system->setUpdateCallback(update, NULL);

        font = pd->graphics->loadFont("/System/Fonts/Asheville-Sans-14-Bold.pft", NULL);
        pd->graphics->setFont(font);
    }

    return 0;
}
```

5.2.2. On framerate

As you may have noticed, the fps counter we added in the last section does not go over 30fps even though we are only showing a string of text with no additional calculations. This is because the screen's refresh rate is capped to 30Hz by default, but this limit can be modified using the function `pd->display->setRefreshRate(float rate)`. Setting the rate parameter to 0 gives us an unlocked framerate, making the screen update at the highest frequency it can.

Add `pd->display->setRefreshRate(0)` in the kEventInit section of the eventHandler, compile, and test on the device to see how many frames per second we obtain.

Now you will be getting around 50 fps, which may seem high, but is still not the maximum capability of the Playdate. We just encountered a limitation caused by hardware: a fullscreen redraw of the display cannot be performed faster than 50 Hz. Drawing on the display is done on a pixel row basis, meaning that only the affected lines of the screen will be updated. If you look at our code, you will notice we are doing a `pd->graphics->clear()` each frame, filling every pixel row with white color before redrawing the text. Remove this line, compile, and check the framerate on the device.

You should now see the indicator at 99 fps, which is the maximum value it can display, meaning the actual value could be even higher. As a matter of fact, framerates over 100 Hz are possible on the Playdate using selective drawing techniques.

The takeout of this experiment should be the importance of optimizing draw calls and rendering only the required regions of the screen. Even though the device is capable of such high framerates, they come at an energy cost, which in a portable console means a decreased battery life. Most of the times, a 30 fps framerate will suffice for a good experience, with 50 fps mode as a good option for certain effects or fast-paced types of games.

5.2.3. Bouncing around

Re-add the `pd->graphics->clear()` line at the beginning of the update function. We are going to make our hello world more interesting by making the text bounce across the screen, like in the C example included with the SDK.

Declare the following global variables before the update function:

Listing 5.7: Hello World movement variables, main.c

```
int textWidth;
int textHeight;
int x;
int y;
int stepX = 1;
int stepY = 2;
```

First, we need to know the dimensions of the "Hello World!" text to determine when one of its sides is touching the border of the screen and invert the sense of its movement. We know the text height from the font we specified in the line `loadFont()`, which is "Asheville-Sans-14-Bold.pft", meaning it is 14 pixels tall. For calculating the width, the Playdate SDK has its own method, `pd->graphics->getTextWidth()`. Knowing this, initialize the `textWidth` and `textHeight` variables just after the `setFont()` method in the `eventHandler`.

The variables `x` and `y` store the position of the text. We want it to start in the center of the screen, so on each axis the position must be the size of the screen minus the size of the text , then divided by two. The Playdate API has two constants for the width and height of the screen: `LCD_COLUMNS` and `LCD_ROWS`. They are simply the values 400 and 240, respectively, which is the resolution of the display, but using standard constants makes the code more readable.

Now we need to update the text's position each frame by adding `stepX` to the `x` variable, and `stepY` to the `y` variable. Finally, if the text goes offscreen we need to switch the sense of the movement on each axis: this happens when the position value for that axis is smaller than 0, or greater than the size of the screen minus the text size for that axis.

Adding these changes to the code results in the following main.c:

Listing 5.8: Bouncing Hello World, main.c

```
#include "pd_api.h"

static PlaydateAPI *pd = NULL;
static LCDFont *font;

int textWidth;
int textHeight;

int x;
int y;
int stepX = 1;
int stepY = 2;

static int update(__attribute__((unused)) void *ud) {
    x += stepX;
    y += stepY;

    if (x < 0 || x > LCD_COLUMNS - textWidth)
        stepX = -stepX;
```

```
20
21      if (y < 0 || y > LCD_ROWS - textHeight)
22          stepY = -stepY;
23
24      // Rendering
25      pd->graphics->clear(kColorWhite);
26      pd->graphics->drawText("Hello World!", strlen("Hello World!"), kASCIIEncoding, x, y);
27      pd->system->drawFPS(0, 0);
28
29      return 1;
30  }
31
32  int eventHandler(PlaydateAPI *playdate, PDSystemEvent event, __attribute__((unused)) uint32_t arg) {
33      if (event == kEventInit) {
34          pd = playdate;
35          pd->system->setUpdateCallback(update, NULL);
36
37          // Font setup
38          font = pd->graphics->loadFont("/System/Fonts/Asheville-Sans-14-Bold.pft", NULL);
39          pd->graphics->setFont(font);
40
41          // Text dimensions setup
42          textWidth = pd->graphics->getTextWidth(font, "Hello World!", strlen("Hello World!"), ↩
                ↪ kASCIIEncoding, 0);
43          textHeight = 14;
44          x = (400 - textWidth) / 2;
45          y = (240 - textHeight) / 2;
46      }
47
48      return 0;
49  }
```

Compile and deploy the program to the console and watch as the text bounces around the screen, just like in the classic DVD player screensavers. Nostalgia!

Figure 5.2: Hello World! bouncing across the screen.

5.2.4. Crank it up

As you know, one of the defining characteristics of the Playdate is its crank input. Let's incorporate it into our example by using it to fast-forward or reverse the movement of the text. We only need to change two lines of our code to add this functionality; but first, let's understand how the Playdate SDK handles crank input.

The Playdate API has three methods related to the crank:

- `int pd->system->isCrankDocked()`, which returns 1 if the crank is docked and 0 if it is undocked.
- `float pd->system->getCrankAngle()`, which gives us the current angle the crank is at, starting at 0^o at the top and increasing clockwise as viewed from the right side of the device.
- `float pd->system->getCrankChange()`, which returns the angle change of the crank since the last time the function was called. This is the best option for what we want to do, as we only need to know the speed and direction of the crank between update calls.

We want the text to bounce as before and to increase or decrease the velocity based on the crank's speed. To achieve this, when adding the step variables to the text's position, we will also add the step variables multiplied by the value returned by `pd->system->getCrankChange()`.

Listing 5.9: Adding crank control, main.c

```
static int update(__attribute__((unused)) void *ud) {
    x += stepX + stepX * pd->system->getCrankChange();
    y += stepY + stepY * pd->system->getCrankChange();
    [...]
}
```

Compile and deploy the project, and test the result: moving the crank forward makes the "Hello World!" text move faster, while cranking in the opposite direction makes it move backwards. Much more interactive!

5.2.5. Extra steps

Working on the example we built on this chapter, try to implement some of the following improvements, or challenge yourself with any modification you can imagine:

- We saw in section 5.2.2 how clearing the screen every frame limits the refresh rate to 50fps. Can you modify our rendering process to erase only the required portion of the screen? Take a look at the geometry drawing functions in the official manual, Inside Playdate with C by Panic (2020b), or come up with your own solution.
- Draw a background under the text instead of the blank screen we have now. The Playdate has a memory display, meaning that pixels keep their value until a different draw call is performed over them. Knowing this, can you implement the previous optimization, and only redraw the part of the background necessary to erase the text between frames?
- Experiment with drawing modes such as XOR, OR, and how they affect the text when moving over a background image.
- Add some background music.

6. Development

6.1. Iteration 0 - Getting to know the Playdate

The first couple of months in possession of the hardware were dedicated to learning and understanding the console, as well as the structure and philosophy behind the SDK. At the same time, with the purpose of being extensive in this thesis and covering every major way of developing for the Playdate, prototypes were made in Lua, C, C++, and the Pulp game creation tool. Thanks to this investigative process, a broad understanding of each language's pros and cons was acquired, which helped cement C as the language of choice for developing the main game.

For full details about every prototype created during this phase, refer to the *Appendix A*.

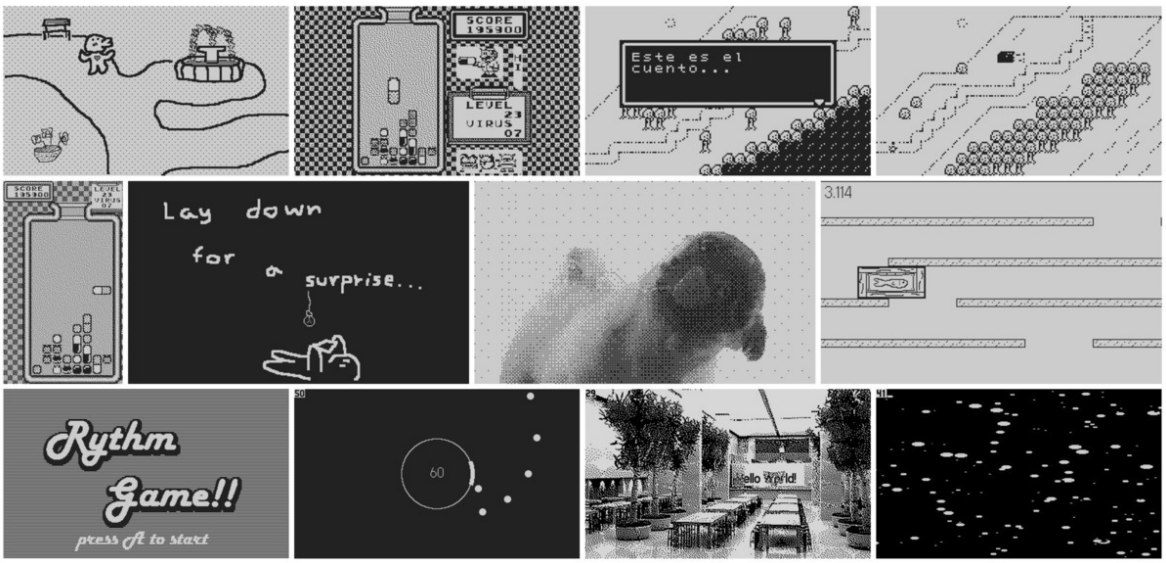


Figure 6.1: Screenshots from all developed prototypes.

6.1.1. Iteration 0.1 - Lua

One of the first recommendations Playdate developer Panic made during a coding-oriented livestream was for experienced programmers to "check the Lua interface first, you can get some decent performance out of it, and it is much simpler than writing to the C interface" (Frank, 2020, min. 4:02). This suggestion seemed sound, and so the first prototypes were written using the Lua SDK.

Even without previous experience with the language, the learning curve was moderate. The

Lua interface proved to be clear and extensive, going further than the basics with an out-of-the-box implementation of many common game features such as tilemaps, image effects, z-buffering, and collision detection.

Hello World. The first experiment was a modification of the sample code from Inside Playdate, the official manual by Panic (2020a), and served to get an understanding of drawing images on the display, using the sprite functionality included in the SDK, simple input handling, and audio playing.

Dr. Mario Mock-up. Based on that first project, I quickly implemented a mock-up of how the Nintendo Entertainment System (NES) classic Dr. Mario would feel like on Playdate. Here, the pill is freely moved using the directional pad and spun by turning the crank. The accelerometer is used to detect if the device is sideways and, in that case, switch to a vertical layout.

Lay down surprise. The third experiment used the same concept of determining device orientation with the accelerometer to show an animated gif of a dog when the display faces the ground. This explored animating sprites with the built-in sprite functions and applying some of the real-time image effects included in the SDK.

Tilting Microgame. Following these demos, a lengthier prototype was developed again centered around accelerometer input. The result was a minigame where the goal was to slide a box through a randomly generated maze by tilting the device from side to side. The box had simple physics implemented by using rectilinear accelerated motion equations. This demo used a game state framework provided by another Playdate developer in the official forums, Nic Magnier.

6.1.2. Iteration 0.2 - C and C++

After gaining familiarity with the Lua SDK, development shifted towards studying the C interface again through the making of various prototypes.

Hello World C. The first project was a modification of the Hello World C sample project that is distributed with the SDK. In it, the string of text "Hello World" bounces around the screen in a similar fashion to old DVD player logos. Expanding upon this simple demo, I incorporated a background image and made the text render in NXOR draw mode (that means, the pixels of the text that overlap black ones get their color inverted). The text is erased by drawing only the necessary rectangle of the background image over it, which improves performance by avoiding full-screen draw calls.

Hello World C++. The same example was implemented in C++. With this version, the focus of the experiment was compiling and running C++ code on Playdate, as it is not an officially supported language. By studying the example included in the SDK and modifying the CMake configurations, the demo was successfully compiled and run on the device.

A big limitation that diminishes the advantages C++ could bring to Playdate development became apparent during this process: the console lacks a C++ standard library implementation. Still, there are useful features of the language that do not require the standard library to function, such as classes, inheritance, or templates.

Some time was spent understanding this problem and exploring possible solutions. The possibilities explored were the following: defining the missing system-level symbols and operations, a solution used in other ARM metal processors; modifying a standard library implementation and tailoring it to the device, which went above the scope of this thesis; and

finally, avoiding the use of standard library functionality or developing custom implementations for specific classes instead. The last solution proved to be the most feasible, but in the end, this diminished the appeal of the language, and C++ was discarded in favor of C as this project's main programming language.

6.1.3. Iteration 0.3

Rhythm Game. Following those last weeks of C and C++ development, I returned to Lua to quickly prototype a rhythm game. In the vein of classic musical titles such as the *Guitar Hero* series, *Osu!*, or Japanese arcade machines, this game consists of a series of falling notes synchronized to a song that the player must hit on the beat. This detour from the C language was taken to prioritize speed and ease of development and center the weight of the prototype on game design, interaction design, and having a closed product.

None of the prototypes since the early Dr. Mario proof of concept had used the crank input at all, which is arguably the most iconic feature of the console. Exploring and using the characteristics that make the Playdate unique is one of the main objectives of this thesis; it was time for the crank to play a central role in the experience, so this game's concept was conceived around it.

The gameplay is as follows: a song plays in the background, a circle occupies the center of the screen, and players control an arc that moves along it matching the current angle of the crank. Using this arc, the player must catch dots representing "notes" that fall towards the center of the circle. For the game to feel satisfying, these notes must be synchronized with the music and arranged representing characteristics of it, such as beat, voices, and overall energy.

I implemented a simple state machine to handle switching between the menu and gameplay portions of the game. This was done via a class called `GameManager`, which holds a Lua table referencing the logic and rendering functions of the current state. Changing between states is done by calling `GameManager.changeState()`, with the update and render functions plus an optional init function as parameters. When this method is invoked, it stores the functions in the `GameManager` table and then executes the `init` function once.

In this prototype version of the game, there are only three game states. The first one is a loading state, which, in a full version, would be used for loading assets when opening the game. Currently, all this state does is instantly change to the next one, which is the menu state. In the menu state, players are greeted with a title screen and music. In a full version, other menu options would appear, implemented in their own game states; but as of now, simply pressing the A button in the menu switches to the in-game state, in which the gameplay starts.

Note patterns needed to be designed by hand, a process that would benefit from having audio playback and a timeline and waveform visualization. Audacity, an open-source sound editing program, met all of those requirements and allowed for tagging specific points of an audio file, making it perfect for the job. A simple parser was written to translate Audacity tags containing time and angle information to in-game notes. More information about this and other aspects of the prototype can be found in the appendix A.

6.2. The game: TinySeconds

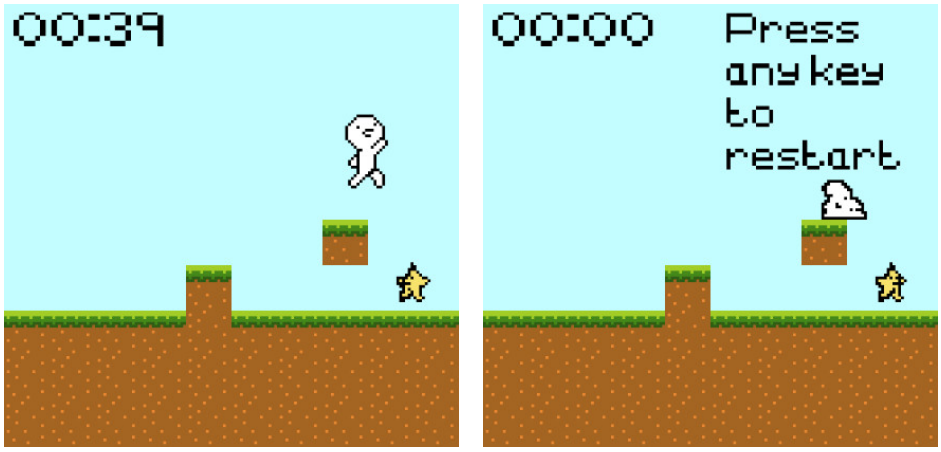
Those first months dedicated to little prototypes proved very useful for quickly learning to develop for Playdate, covering various programming languages and areas of development. At last, it was time to start working on a bigger project, the game whose development will be covered in the rest of this chapter. As said before, each iteration will be divided between planning, developing, and conclusions; but first, an introduction to the game.

6.2.1. The concept

TinySeconds will be a side-view platformer where the player must get to the goal in one second or less, focusing on high-speed gameplay, sharp controls, and quick reaction times. Levels will be single-screen and drawn using tilemaps.

Throughout the levels, several obstacles and special mechanics will present a challenge to the player and add variety to the gameplay. Because levels are very short in duration, players will have to sequentially complete a series of them without losing in order to advance to the next batch.

Prior to development, a prototype was made in the proprietary game engine Unity3d, which can be played in-browser[1]. See fig. 6.2.



(a) Playing **(b)** Game over

Figure 6.2: Unity prototype

6.3. Iteration 1 - Setting up the foundation

The tasks planned for this iteration were:

- Implementing an ECS engine in C as the base game structure. Then, use it to implement a simple starfield effect demo.
- Expanding upon this engine by allowing components to be detached from entities.

[1]One second prototype playable at `https://abramaran.itch.io/one-second`

- Investigating Tup as a possible build system for Playdate games.

6.3.1. Introduction to Entity Component System (ECS)

An Entity Component System (ECS) is an architectural pattern focused on ease of design and cache access optimization. It achieves this by modularizing its parts and decoupling functionality from data, grouping the latter into entities with simple identifiers. From a design perspective, this results in a much simpler and maintainable structure than the hierarchical setups present in most Object Oriented Programming (OOP) game engine patterns.

Components are one or more variables grouped together based on conceptual similarity; for example, position, velocity and acceleration can be grouped in a physics component. They are the building blocks with which entities are constructed.

An entity is usually represented by an identifier, typically a unique number assigned on creation. Their purpose is to mark a series of components as pertaining to a same owner; for example an instance of an enemy, the player, or a game camera.

Systems are methods that provide a specific functionality, such as physics simulation, reading player input, or collision checking. They use and modify the data contained in the components to perform their function. Oftentimes, a system will need more than one type of component pertaining to the same entity to work.

Imagine a typical OOP approach to creating a "Player" class for a game: we would define a new class, add member variables for storing its data like position, velocity, or sprite image, and then add methods to provide functionality using those variables.

Listing 6.1: Example of a player class definition in an Object Oriented Programming (OOP) architecture

```
class Player {
  private:
    int x, y;
    float velocityX, velocityY;
    int health;
    Image sprite;

  public:
    move(int x, int y);
    attack();
    die();
};
```

In an ECS architecture, however, we would not create a new class to represent the Player; we would create a new entity, usually a simple identifier, and add the necessary components to it. Then, during the update cycle, each system would provide a piece of functionality by retrieving all the components of the types they need and operating on them, oftentimes oblivious to who owns them.

Listing 6.2: Example of creating a player entity in an ECS architecture

```
void initialize() { // Game initialization
    int entityID = createEntity();
    addComponent<PhysicsComponent>(entityID);
    addComponent<HealthComponent>(entityID);
}

void update() { // Main loop
    inputSystem.update();
```

```
    physicsSystem.update();
    attackSystem.update();
    healthSystem.update();
}
```

6.3.2. Simplified version of the ECS

The ECS pattern was the architecture of choice for structuring TinySeconds's game engine; its focus on speed and cache optimization is important on a limited machine as is the Playdate, and its design simplicity is a welcome characteristic. The challenge was now to develop this type of engine using C, a language missing features like templates or interfaces which are commonly used in ECS implementations.

Because of these difficulties, it was decided to simplify the ECS structure for the first version of the engine: every entity would have a component of each type stored inside them, whereas in a full implementation components should be decoupled from entities and stored separately from them. Then, instead of systems iterating over all components of the required types, they would iterate over all entities. In a normal setup this could reduce cache access efficiency, but because the data contained in our components is very small, all entities fit completely in cache. This simplified version of an ECS structure was modeled after a series of instructional livestreams by Durán (2020).

The project's code is divided between the main loop, the entity manager, and systems.

The *main.c* file performs the necessary initialization operations and contains the main loop of the application, which is called every frame. First, all logic systems are updated, including ones that may create new entities; then the rendering system is called, drawing the elements of the game on the screen; lastly, the entity manager is called to destroy the entities that are marked for deletion. See 6.3.

Listing 6.3: main.c: main loop of the game

```
static int update(__attribute__((unused)) void *ud) {
    sys_physics_update();
    sys_generator_update();
    sys_render_update();

    man_entity_deletedead();
    return 1;
}
```

The entity manager in the *entity.c* file defines the entity struct, entity types, and all the components. It also manages the creation and destruction of entities and executes systems on all of them. A commented overview of the entity class can be read in listing 6.4.

Listing 6.4: The entity manager's header file

```
// Type and status of the entity.
typedef enum entity_type {
    invalid,
    default_type,
    star,
    dead
} entity_type;
```

```
8
9 // An entity has an entity_type and one component of each type.
10 typedef struct Entity_t {
11     entity_type type;
12     int x, y;
13     int vx;
14     unsigned int wx, wy;
15 } Entity_t;
16
17 // Array that holds all entities
18 static Entity_t m_entities[MAX_ENTITIES];
19
20 // Create a new entity
21 Entity_t *man_entity_create();
22
23 // Mark entity as dead
24 void man_entity_set4destruction(Entity_t *dead_e);
25
26 // Delete entities marked as dead
27 void man_entity_deletedead();
28
29 // Execute a system on all entities
30 void man_entity_forall(void (*ptrfunc)(Entity_t *));
31
32 // How many entities can still be created
33 unsigned int man_entity_freespace();
```

Each system defines its update function and an optional initialization method. When calling its update function, a system needs to execute its functionality once for every entity and operate on its components. One way of doing this could be to pass the entity array to the system and iterate over them in it, but this would result in repeated code as every system shares this necessity. To avoid this, we use a programming principle called "inversion of control": instead of the entity manager passing the entities to the system, the system sends it an update function for a single entity. Then, the entity manager calls that function once per entity passing it as parameter so that the system can access its components.

Listing 6.5: Example of a system: Physics system

```
1 // Called once per entity from the entity manager
2 void sys_physics_update_one_entity(Entity_t *e) {
3     int16_t newx = e->x + e->vx;
4     if (newx < -e->wx) {
5         man_entity_set4destruction(e);
6         e->vx = 0;
7     } else {
8         e->x = newx;
9     }
10 }
11
12 // Public function called from the main loop
13 void sys_physics_update() {
14     man_entity_forall(sys_physics_update_one_entity); // Pass the function to the entity manager
15 }
```

After finishing development on this first version of the engine, it was time to put it to the test. For this, I implemented a graphical effect similar to the one from the instructional videos by Durán (2020), consisting of a series of stars that move across the screen. The illusion of depth is created by reducing the size and speed of the stars the further away they are from the

camera. Running this demo on the console gave excellent performance, achieving an average of 43 fps for 1000 simultaneous entities. Screenshot in fig. 6.3.

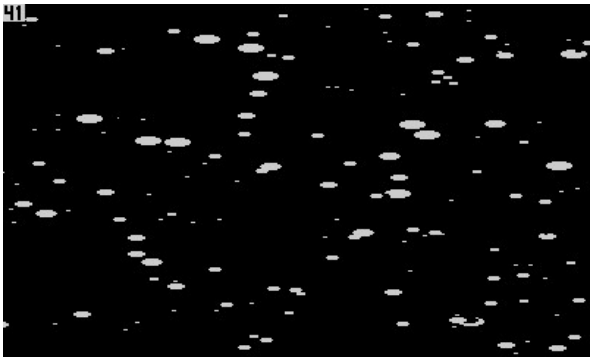


Figure 6.3: ECS Starfield effect

6.3.3. Full ECS implementation

Even though the simplified version of the ECS was functional and performant, I decided to evolve it into a more complete ECS implementation with components being decoupled from entities. Components still needed to be subsequent in memory to optimize cache, and for the `man_component_forall(Component component)` function to work some sort of polymorphism would be needed. The process for this evolution of the ECS can be read about in appendix A.

A functioning full ECS engine was successfully implemented, and so it was time to test it with the same example as before: the star field graphical effect. The results were dissapointing; performance was significantly reduced with framerates averaging 12 fps. Profiling tools for C games on Playdate are, at the time of writing, limited to simple console prints, which makes searching for a culprit difficult. Still, the source of this loss of performance could be attributed to the small size of the Playdate code and data caches.

In this simple example there are only two types of components: Physics, and Size. Each of them is enclosed in a more generic Component struct along with a type enum to provide polymorphism, and the entity id of their owner. Adding the size in bits of its members, we see that each instance of a Component occupies 112 bits, assuming integers are 32 bit. The data cache of the console can contain up to 4096 bytes of information, which equates 32768 bit. From this, we can see that only 292 components would fit in the data cache. Systems usually need more than one component type to function, and component arrays of different types are stored sequentially in memory. Each component array allocates sufficient memory for the maximum number of components, in this case, 1000. Knowing all this, it is plain to see that two components of different types will almost never be close enough in memory for them to be loaded in cache at the same time. In decoupling components from their entity, the engine has lost the cache speed boost that its simplified version benefited from.

6.3.4. Conclusions

After comparing the performance of the rudimentary and complete versions of the ECS engine, the former was decided as the base for the game. The lesson learned is that the most orthodox solution is not always the best; design choices should not be motivated by dogma or theoretical correctness, but by the needs and characteristics of each specific project. Over-engineering and premature optimization are common mistakes among software engineers, and so a balance between correctness and simplicity must be found.

As mentioned in the planning section, some tests were carried out regarding the Tup build system. While it is a well-designed and innovative build system, featuring fast compile times and intuitive usage, it is not a good fit for this thesis's project. Officially, Playdate games are built using CMake and make, and so the time investment required to translate the scripts, CMake rules, and makefiles to Tup configurations outweighs the rewards.

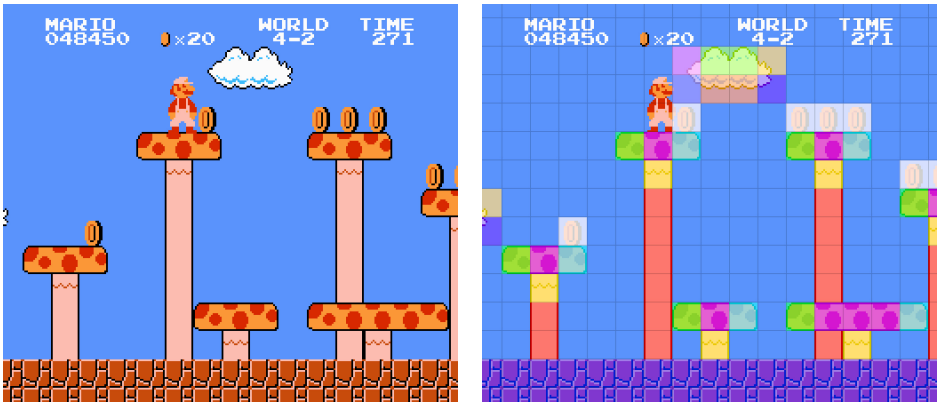
6.4. Iteration 2 - Tilemaps and movement

The objectives set for this iteration were the following:

- Implementing level loading via tilemaps.
- Implement platforming mechanics: player movement.
- Develop a minimum viable product with a win state, a goal the player can achieve.

6.4.1. Tilemaps

Tilemaps are a method of creating videogame maps and levels using small tiles that form the walls, floors, and corners, instead of unique drawings for the whole level. It was the most popular approach during the early days of the medium, as storage was tight and not many graphics could be bundled into a game. Some notable examples of tile-based games from that era are Super Mario Bros. or The Legend of Zelda, both for the NES.



(a) Screenshot of Super Mario Bros. **(b)** Repeated tiles in the same color

Figure 6.4: Example of a tilemap in Super Mario Bros.

Even though data storage availability is not usually a concern in modern days, tilemaps are still widely used, as they offer many advantages. First of all, they offer a very economical way of creating graphics, as only a small number of reusable drawings are needed to form the scenery and platforms. They also allow for quick design iteration on a map, as doing the necessary modifications is as quick as changing a few tiles. Another advantage is the spatial division of levels in rows and columns, which can be used (and will be in our game) to optimize collisions by checking only the tiles surrounding the player. In cases where collision can be less precise, such as RPGs games, the map can be divided between solid and walkable tiles, making collision checking as simple as reading a boolean from the level matrix.

The open-source tilemap editor Tiled will be used for the creation of all levels and tilesets in this game. More information about this program can be found in annex C.

The Playdate screen has a resolution of 240x400 px. If we find all divisors for both of those sizes and select the common ones, we obtain the square tile sizes that can perfectly cover the whole screen. The Head-Up Display (HUD), such as lives, score, time, or other info displayed graphically, usually takes up part of the screen, so other tile sizes that leave a margin in one of the axes can also be useful. The square tile sizes that fill one or both of the Playdate screen axis are the following:

- Square tile sizes that cover the screen: 1, 2, 4, 5, 8, 10, 16, 20, 40, and 80 px.
- Sizes that fit screen width but leave a margin on the height: 25, 50, 100, and 200 px.
- Sizes that fit screen height but leave a margin on the width: 3, 6, 12, 15, 24, 30, 48, 60, and 120 px.

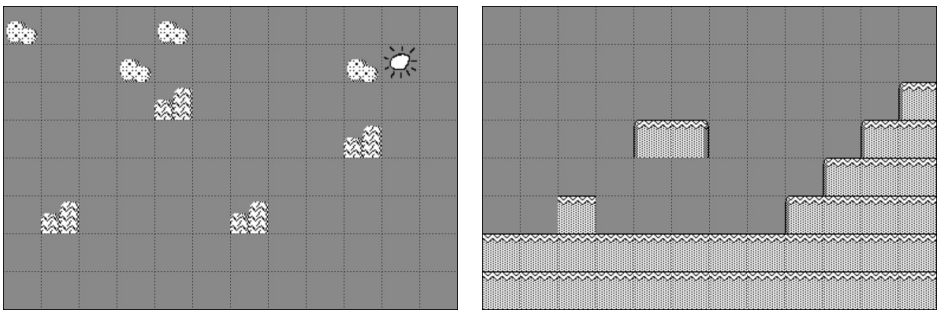
In the end, the tile size selected for this game was 32x32 px. As the screen is not divisible by those dimensions, we are left with margins in both the width and height axes. This is covered by adding an extra row of tiles at the bottom of the map that will be only half-visible. The height margin will be used to draw a simple HUD for the timer of the level.

One useful feature of Tiled is the ability to have several tilemap layers, allowing for depth effects or dividing tiles between collidable and not, among many other uses. In our case, maps will have a foreground layer, the one representing platforms the player can walk on and collide with, and a background layer, used for decorations and other non-collidable graphics. See fig. 6.5.

6.4.2. JSON

Once created or modified, the tilemaps are exported from Tiled in JSON format and saved with the rest of the game files. The Playdate SDK provides a JSON parser and writer, which will be used for loading the levels at runtime: the class `json_decoder` inside *pd_api.h*.

The `json_decoder` class works by setting handlers for each of the actions we are interested in managing; these are function pointers that can be set at creation by using C99's designated initializers like in the code snippet 6.6. When creating a `json_decoder` it is obligatory to implement the `decodeError` handler, while the rest are optional. Unused handlers in a `json_decoder` must explicitly be initialized to NULL.



(a) Foreground **(b)** Background

Figure 6.5: Division in layers of a tilemap

Listing 6.6: Initializing a *json_decoder* object using C99's designated initializers

```
json_decoder my_decoder = {.decodeError = handler_decodeError,
                           .didDecodeArrayValue = handler_didDecodeArrayValue,
                           .didDecodeSublist = NULL,
                           .didDecodeTableValue = handler_didDecodeTableValue,
                           .shouldDecodeArrayValueAtIndex = NULL,
                           .shouldDecodeTableValueForKey = handler_shouldDecodeTableValueForKey,
                           .willDecodeSublist = NULL};
```

The tile distribution that forms the level is represented in the JSON file as an array of tile identifiers (IDs). The `didDecodeArrayValue` handler, which fires after parsing a JSON array, is implemented for storing this data in the level array. Before reading this or any other value, the `shouldDecodeTableValueForKey` handler is called; here, it is implemented to increment the tilemap layer number. The `didDecodeTableValue` handles other variables that are packaged in the JSON file alongside the tile distribution data, like the dimensions of the tilemap and tileset, the pixel size of the tiles, or the name of the tilemap layer about to be read.

Once the `json_decoder` is created, the JSON file is opened using the SDFile class included in the Playdate SDK and then passed to the decoder.

Listing 6.7: Opening a file using the Playdate SDK and passing it to the decoder

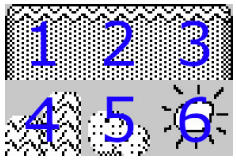
```
// Reading handler for the json_decoder
int readfile(void *readud, uint8_t *buf, int bufsize) {
    return pd->file->read((SDFile *)readud, buf, bufsize);
}


SDFile *file = pd->file->open(jsonName, kFileRead);
 if (file == NULL) {
    pd->system->error("Couldn't open file %s", jsonName);
}

pd->json->decode(&my_decoder, (json_reader){.read = readfile, .userdata = file}, NULL);
```

6.4.2.1. JSON decoder bug

A bug affecting the `json_decoder` was found in version 0.12.0 of the Playdate SDK while developing the JSON file loading for this project: returning 0 in the `shouldDecodeTableValueForKey`↩

↪ and `shouldDecodeArrayValueAtIndex` methods should skip the reading of the value paired with the current key, but using this feature caused a crash in the application. A bug report was submitted to the official GitLab repositories, and fixed in the next SDK release. More information about this issue can be found in annex B.1.

6.4.3. Drawing the tilemap



(a) Tileset with tile IDs overlaid

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
 0, 0, 0, 0, 1, 3, 0, 0, 0, 0, 1, 2,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 2,
 0, 0, 2, 0, 0, 0, 0, 0, 1, 2, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

(b) Representation of the tilemap as a 1-dimensional array of IDs

Figure 6.6: Numbering and distribution of tiles in a tileset and tilemap

Once the tilemap is read and stored as tile IDs in an array (fig. 6.6b), it is time to render the level. For each layer of the tilemap, ordered from back to front, we iterate the tilemap IDs, determine the portion of the tileset texture that corresponds to that tile, and draw it in its correspondent row and column of the screen.

For determining the portion of the tileset to be drawn, we start from the knowledge that Tiled assigns tiles their IDs based on their position in the tileset, starting at 1 at the top-left tile and going from left to right (6.6a). Knowing the width in pixels of a tile and the number of columns in the tileset, we can obtain the offset in pixels on the x and y coordinates, named (u, v) respectively by convention, by using the following equations[2]:

$$u = tile\_width \times ((tile\_id - 1) \bmod tileset\_columns) \tag{6.1a}$$

$$v = tile\_width \times (\lfloor (tile\_id - 1) \div tileset\_columns \rfloor) \tag{6.1b}$$

Next, the Playdate function `playdate->graphics->setClipRect()` is used to select the area of the screen that we will be drawing to; in this case, a square the size of a tile at the position corresponding to the current row and column of the tilemap. Finally, the tileset image is drawn offsetting its position by the (u, v) offset we calculated, so that the part of the image corresponding to the tile to be drawn fills the clip rectangle. You can think of this process as cutting a window in a sheet of paper and sliding an image underneath it, so that the desired portion of it is visible through the window.

[2]The floor operation in 6.1b can be omitted because of C's default integer division behavior, which discards decimals by flooring the result. Implementations in other languages or using different data types must include it for the formula to work.

Listing 6.8: Drawing the tile

```
playdate->graphics->setClipRect(x, y, width, height);
playdate->graphics->drawBitmap(tileset, x - u, y - v, kBitmapUnflipped);
```

6.4.3.1. ClipRect bug

Version 0.12.0 of the SDK introduced a bug in the creation of clip rectangles, which I stumbled upon at this point of development. The size of the rectangle was being affected by its position: the resulting dimensions were the specified size plus the value of the position in the same axis. For example, a rectangle with position (3, 6) and size (10, 20) would end up having the dimensions (13, 26). I filed a bug report with a demonstration program and source code, and it was soon patched. More information about this process can be found in annex B.2.

6.4.4. Player movement

In this iteration, basic player movement and input reading were also implemented. Movement is handled in the physics system: the player's position is incremented if the right button on the D-Pad is pressed, or decremented if pressing the left button. When starting a jump, the entity variable `vy` (velocity in the y axis) is set to a high number, as in platformer games jumps feel better if they are explosive, not accelerated; then, each frame for the duration of the jump, the player moves `vy` pixels and the variable is decremented. The jump ends when the player is back at the y position it started at.

The same method used to crop the tiles from the tileset was used to change the player sprite depending on the action performed (moving left or right, standing still, and jumping). Sprites are stored in a sprite sheet, which is a single image containing different frames of animation instead of them being in separate images. See fig. 6.7.

Figure 6.7: Example of player sprite sheet

6.4.5. Conclusions

Most of the objectives laid out for this iteration were achieved, save for the last one (obtaining a first closed product with a win state). This was due to the delays caused by the bugs encountered when developing the level loading and drawing. Still, the implemented features are a big part of the game, and so this iteration proved fruitful.

In addition to the tasks explained in this section, a prototype was made using Pulp, a visual game creation tool developed by Panic. This was done with the objective of gaining a whole perspective of the possibilities of Playdate game development, a goal that has been stated several times in this thesis. More information about this prototype can be found in annex A.4.

6.5. Iteration 3 - Collision

The objectives set for this iteration were the following:

- Implement collisions with the solid tiles.
- Modify player movement to respond to these collisions.
- Implement changing between levels.
- Add a win state, a goal the player can achieve, to have a minimum viable product.

6.5.1. Collision

Now that level loading and player movement were implemented, it was time to program collisions with the environment. As its name implies, a platforming game is based around movement, with the player jumping on different platforms with precision to avoid gaps and other hazards. Because of this, it is essential to have a robust collision system. This means implementing a system that allows the player to stand on top of different terrain heights and prevents it from moving inside walls or ceilings.

The approach taken consists of the following steps: first, determining which tiles are overlapping the player sprite; then, iterating those tiles in the tilemap array and checking if they are solid or an empty space; lastly, if the tile is solid, calculating from which direction the player entered the tile and undoing the overlap in that direction.

Determining which tiles overlap the player begins with translating its position from pixel coordinates to tile coordinates, as in row and column of the tilemap:

$$(column, row) = (\lfloor x \div tile\_width \rfloor, \lfloor y \div tile\_height \rfloor) \tag{6.2}$$

Next, the minimum number of tiles that the player will overlap based on its dimensions is calculated by dividing the size of the player sprite by the size of the tile on each axis, rounding up in cases where this division can have decimals.

Most of the time the player will not be aligned with the tilemap grid, as its movement is independent from it. This must be accounted for by checking collision on an additional row or column of tiles for the misaligned axis. A way to determine this is by using the module of the division in formula 6.2: if the module is 0, the player is perfectly aligned with the tiles; if not, the number of overlapping tiles for that axis is incremented by 1. In summary, the number of tiles that must be checked for collision is calculated with the formula 6.3.

$$\begin{aligned}(tiles_x, tiles_y) = (&\lceil sprite\_width \div tile\_width \rceil + x \mod tile\_width, \\ &\lceil sprite\_width \div tile\_height \rceil + y \mod tile\_height)\end{aligned} \tag{6.3}$$

The next step is to iterate the tiles that overlap the player. For each one, the tilemap array is checked to determine if they are a solid or empty tile. If a tile is solid, its overlap in pixels for each axis is the difference between the positions of the tile and the player, subtracted to the player's size in that axis.

$$\begin{aligned}(overlap_x, overlap_y) = (&player\_width - |player_x - tile_x|,\\ &player\_height - |player_y - tile_y|)\end{aligned} \tag{6.4}$$

For undoing the collision, the player will be pushed away from the tile only on the axis with the smallest overlap:

- In the case of this being the horizontal axis: if the player's x position is smaller than the tile's, the player is colliding from the left side and is moved $overlap_x$ pixels in that direction; otherwise, the player is entering from the right side, and is moved the same amount to the right.

- For the vertical axis: if the player's y position is smaller than the tile's, the player is coming from above and is moved $overlap_y$ pixels up; otherwise, the player is colliding from below, and is moved the same amount down.

6.5.2. Delta time

Up until now, the player's velocity was tied to the game's frame rate, as its position was incremented each update call by a fixed amount. This approach can work in some cases, but it is not the best solution, as an eventual frame rate drop would cause the action to slow down perceivably. Instead, most time-related features such as player movement, animations, or damage over time ailments should be based on timers independent from frame rate.

The usual way to do this is with what is commonly called Delta Time: the time elapsed between each update call. This consists of a simple system called first thing in the update loop which stores two variables: `DeltaTime`, and `last_time`. The Playdate SDK has a function to retrieve the current time in milliseconds, measured from an arbitrary point in time: `playdate->system->getCurrentTimeMilliseconds()`. On each update, the system stores in the `DeltaTime` variable the current time minus `last_time`, which stores the timestamp of the last time the system was called. This way, the system obtains the elapsed time between update calls. It finally updates `last_time` with the current time, preparing it for the next update. `DeltaTime` is a public global variable that other systems can use to their needs.

6.5.3. Updated player movement

Until now, the player could only move and jump at one height, as if it stood on flat ground, because there was no collision detection to tell it if it had landed on a platform. That way, all jumps ended at the same height. Having implemented collision, it was necessary to update the physics system, which controls the player's movement, to allow the player to stand on different ground levels.

The way this is implemented is by adding "gravity": if the player is not jumping, its

position on the y axis is incremented each frame[3], allowing it to fall off ledges. The ability to stand on higher platforms comes as a result of the collision system correcting overlaps with platforms, so no additional programming was required to allow for this.

Another notable change was the usage of the *DeltaTime* (converted to seconds) to determine the amount the player needed to move each frame, making its speed based in pixels per second instead of being a fixed amount per update.

6.5.4. Conclusions

Even though the implementation of a first collision system is a significant milestone for this project, testing revealed that the current way of undoing overlaps presents poor results in certain situations. Platforms can be made of more than one tile, but the method employed treats each tile as if it were an individual platform, leading to collisions being solved by pushing the player inside the adjacent tile (which in turn pushes them further, resulting in a strange teleportation). This most notably occurs when colliding with a platform from below.

Still, the game's main mechanics are present in their basic form, which is fundamental for progressing development. Two of the objectives for this iteration weren't met: the implementation of level change, and a way to finish a level. This slowdown can be attributed to the underestimated complexity of collision detection, plus the lack of debugging or simulation tools for C Playdate games in Windows at the time of writing. These unmet goals and the problems with the collision system will be addressed in a future iteration.

6.6. Iteration 4 - Enter the game loop

The objectives established for this iteration were the following:

- Implement win and lose states.
- Change between levels.
- Add a static hazard such as spikes or lava that restarts the level when the player touches them.
- Add a simple patrolling enemy.

6.6.1. Trigger system

Generally speaking, there are two categories in which the response to two entities overlapping in a game can fall: one of them is collision, which simulates interactions between physical bodies by correcting the intersection between them, and usually, adding the appropriate reaction forces; the other one is triggers, which execute a function upon entering overlap. Triggers are a fundamental feature in platforming games, as they can be used to detect when the player reaches the goal of the level, collects items, or touches hazards or enemies that inflict damage, among many other uses and mechanics.

[3]By default, in Playdate games the origin of the coordinate system is located in the upper-left corner of the screen, so y values increase towards the lower edge of the screen.

In this iteration, triggers were implemented and used for changing level upon touching the goal and restarting the level if the player touches a hazard tile. For this, the trigger system was created, and its update function added to the main loop. This system must be updated after the boundingTiles system, as it depends upon the tile coordinates and bounding box calculated in it.

The trigger system is called for every entity but acts only on those of the types that must react upon contact with the player, in our case, `goal_type` and `enemy_type`. First, the system must determine if the player and trigger entity are overlapping, which will be true if the following condition is met: for each axis, the entity's position is bigger than or equal to the player's, but smaller than the player's position plus the size of its bounding box on that axis.

If the player and the trigger entity are overlapping, the system returns an enum value based on the required response: `triggered_none`, `triggered_goal`, or `triggered_death`. In the main update loop, a switch statement is done on this return value, and the necessary actions are performed for each case.

6.6.2. Time limit

There was one defining mechanic of the game, the one that gives it its name "Tiny Seconds", which had not yet been implemented: the time limit. In Tiny Seconds, the goal is to reach the end of each level in a very small amount of time, originally one second, focusing the gameplay on agile, fast platforming.

For this mechanic to be effective, the time limit had to be just enough to reach the goal, but not more, so as to give a small margin of error but make the game feel tight and give an adrenaline rush to the player. The originally planned limit of one second proved to be too short, and after some tests, the chosen value for the timer was 2.5 seconds. There also needed to be a very readable way of transmitting the amount of time left: instead of displaying a numeric counter, a gauge in the form of a vertical bar on the right side of the screen was used. This display uses contrasting black and white colors for the bar and background, so that even without directly looking at it, the player can sense how much time is left out of the corner of their eye. Also, although less important, with the bar being at the right side of the screen where the goal usually is at, the person playing will follow the player sprite as it moves towards it, and the timer will be in focus in the most crucial last tenths of a second as the player is about to finish the level and the time is about to run out.

The timer system is one of the simplest of the game: it substracts the `DeltaTime` value from the time limit on each update, and returns true if there is time left. If the timer is smaller than 0, it returns false instead, making the main update loop know that the level must be reset. It also has a method called `sys_timer_reset()` that assigns the maximum value to the time limit, which is called on level resets by the update loop.

For rendering the timer bar, a new utility class was created called HUD. This class has a render method that reads the time left from the timer system, divides it by the maximum time limit, and draws the vertical gauge multiplying its total height by the percentage obtained.

6.6.3. Reading objects from the tilemap

Now that the game had goals, hazards, and the player spawn, with more elements coming in the future, it was clear that a better way of placing these objects needed to be set in place.

A very widespread way of handling this problem was chosen as the solution: placing these objects as tiles in the tile editor, and at the time of loading a level, identifying these tiles and performing the necessary actions (such as creating the pertinent entities or setting the goal and player spawn position).

The tile IDs of these special tiles were saved in constants. For those that required spawning or setting the position of a unique entity, in this case the goal and spawn tiles, a global variable was created in the *tilemap.h* file to store their position and tile coordinates.

In the JSON reader, the `didDecodeArrayValue()` handler was modified to perform a switch statement on the ID of the tile being read, performing the necessary operations in the cases of the special tiles. Even though this may seem costly, switch statements are very optimized by the compiler, especially when compared to if-else statements because cases within a switch statement do not rely on the previous ones. There was no noticeable increase in level loading time by adding this step.

For the static hazard and goal tiles, their tile index in the tilemap array is translated to tile coordinates using the following formula:

$$(column, row) = (index \mod tilemap\_columns, \lfloor index \div tilemap\_columns \rfloor) \tag{6.5}$$

Then, for the player spawn, its tile index is translated to position in pixel coordinates with the following formula:

$$\begin{aligned}(player_x, player_y) = (&index \mod tilemap\_columns \times tile\_width,\\ &\lfloor index \div tilemap\_columns \rfloor \times tile\_width - tile\_width)\end{aligned} \tag{6.6}$$

6.6.4. Level restart

When the player touches a hazard or the timer reaches zero, the level needs to be restarted. This is done by calling a simple method in the main.c file which moves the player to its spawn position, the goal to its position, and calls `sys_timer_reset()`. Setting the goal's position is done because the restart method is also called when changing levels.

6.6.5. Level change

Implementing loading the next level when the player touches the goal was quite straightforward: as explained in the trigger system subsection, if the player is overlapping the goal the trigger system returns the `triggered_goal` enum value to the main loop, which in turn calls its `loadAndDrawMap()` method. This method asks the entity manager to delete all entities that are tagged `enemy_type` as hazards are unique to each level; calls `util_tilemap_loadLevel()` passing the path to the next tilemap file as parameter; renders it into a new fully white bitmap; and finally, calls its `restart()` method to reset the timer and the position of the goal and player spawn.

The path to each level's JSON file is stored in an array in main.c in the order they need to appear. Then, the index in the array of the current level is stored in a counter variable. When loading the next tilemap, the counter is incremented, and the path at the next index of the array is passed to the tilemap loader.

The only difficulty found during development of this feature was with C's string manipulation. Until now, the path to the tileset image was obtained by reading it from the tilemap file, where it figures under the "image" field. Tiled exports this path as a relative route, which means that the string "/media/" must be prepended to it so that the Playdate hardware can find the file.

This worked well when the tilemap loading function was only called once, but on consecutive calls the tileset image path was being appended to itself, making the route incorrect. Assigning a value before calling the concatenation function `strcat()` as an attempt of resetting the variable did not make any change. Some time was dedicated to investigating this issue, but knowing that all tilemaps shared the same tileset, it was decided to statically set the route to the image and tackle this problem in the future, if it becomes necessary.

6.6.6. Conclusions

This was a very fruitful iteration, where the remaining core features were implemented. The only objective that was not fulfilled was adding a moving enemy, but in exchange, the timer and its HUD were implemented. The ability of reading special tiles that spawn entities or other objects is also a significant step forward in the foundation of the game, as it is a feature shared between all interactive entities in the level, and will make level design easier.

6.7. Iteration 5

The objective for this iteration was to design and implement new gameplay mechanics to add variety in level design. In addition to that, some areas of the code needed a refactor to improve readability and avoid repeated code. One case of this was conversions between coordinate systems, which were done throughout the project with code repetition and bad legibility.

6.7.1. Toggle blocks

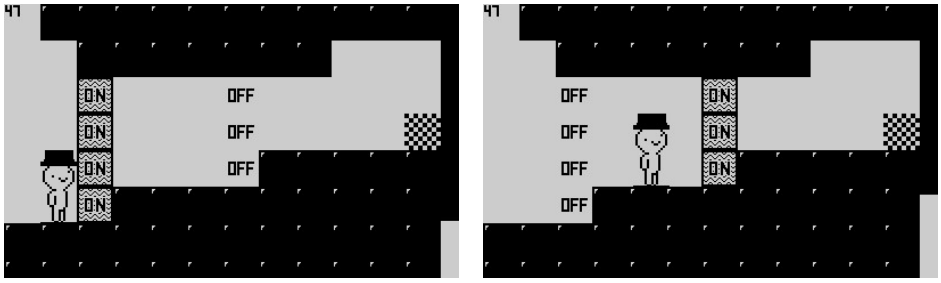
Designing interesting mechanics that took advantage of the Playdate's hardware was challenging due to the fast-paced nature of this game. Every level must be beaten in less than 2.5 seconds, which means that the player will almost surely be pressing the arrow keys constantly to get to the goal in time. This impacts mechanics using the crank, because they usually require changing the grip to grab it. Precise movements with the crank are also difficult in such a short time, and fast cranking shakes the device too much, which makes moving the character at the same time difficult.

The accelerometer is also restrained by the fact that shaking the device blurs the screen, making the player unable to keep track of what is happening. This effect is aggravated by the unlit SHARP technology which depends on light reflecting off the screen for good visibility.

After some experimentation, an unusual interaction was discovered: stowing the crank behind the device instead of extending it completely, so that the handle acts as a stop and gives the crank only half of its range. This way, the crank can be used by flicking it up and down like a switch, an interaction that can be performed with one of the free fingers on the right hand without losing hold of the buttons.

Figure 6.8: Collapsing the crank behind the device to create a switch interaction

This interaction is used for a new type of tile that becomes solid or intangible depending on the crank's position: the toggle block. The range of the crank is divided in two at the middle, and if the crank's angle changes region, the toggle blocks change to their opposite state, working as a switch.



(a) Crank over 270º angle **(b)** Crank under 270º angle

Figure 6.9: Puzzle involving opposing toggle blocks

The state of the toggle blocks is not tied to one of these two regions: on level load they start in the state assigned to them in the level editor, and switch when the crank changes region.

Toggle blocks are implemented as a new type of entity: `toggle_type`, which has a boolean variable called `toggle_on` for storing its initial state. When creating the levels in the tilemap editor Tiled, the initial state is represented using two different types of tile: one for blocks that start enabled, and another one for the opposite case.

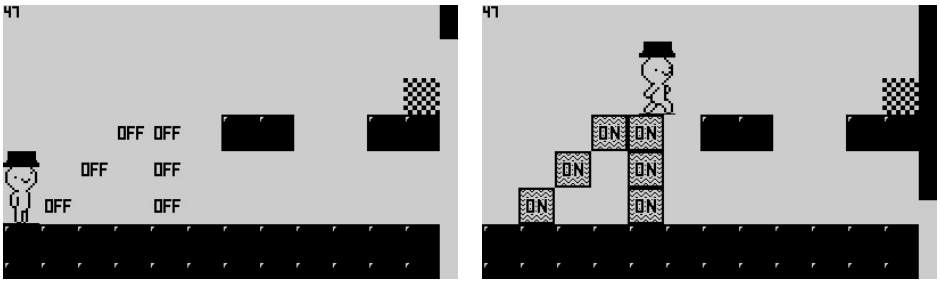
A new system class was created for handling this mechanic: the toggle system. It contains a boolean to store the crank's initial position on each level change. On the update method, it performs the following check: if the crank's initial position equals the crank's current position and the entity's initial state was enabled, the block will be ON and will be solid; otherwise, the block is OFF and intangible.

The approach taken to enabling or disabling collision with the toggle blocks was simple: modifying the tilemap's "Ground" layer, which has collision, by adding or removing a solid block underneath the entity. To do this, a helper function was implemented called `getTilePointer(tileCoords, layerName)`. This function selects the tilemap layer with the

specified name and returns a pointer to the position in the tile array which corresponds to the coordinates passed by parameter. Changing the type of tile is only a matter of writing a different tile ID on that position.

Rendering these blocks was simple; the `render_update_one_entity()` function in the render system was modified to have a switch statement, which handles drawing the player or the toggle blocks. The toggle system sets the entity's sprite sheet coordinates to the ON or OFF sprite when its state changes, so rendering them is as simple as drawing that region of the sprite sheet.

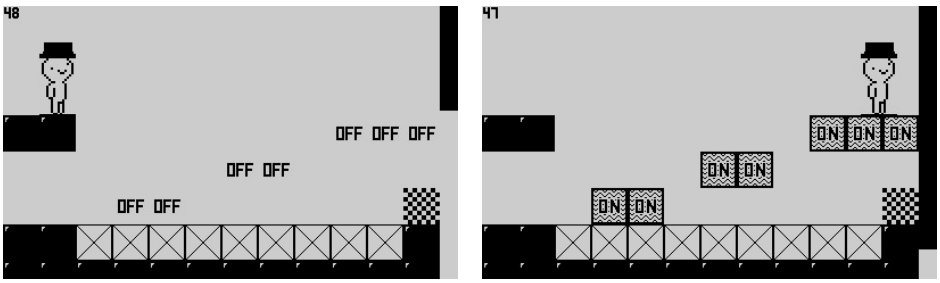
The result is an engaging game mechanic, simple enough to be understood at a glance, but allowing for many design challenges. The blocks can have opposing toggle states, as seen in fig. 6.9, making the player alternate between states to reveal different level layouts. In other cases, the player must quickly reveal a structure to reach a high place, like in fig. 6.10. In fig. 6.11, the player must reveal the toggle blocks to jump over the spikes, but quickly disable them afterward to reach the goal.



(a) The toggle blocks start disabled and there is no way to climb to the goal

(b) Enabling the blocks reveals a staircase

Figure 6.10: Puzzle involving hidden structures



(a) The player must enable the blocks to cross

(b) The goal is under the blocks, so they need to be disabled again

Figure 6.11: Puzzle involving quick coordination for enabling and disabling the blocks

6.7.2. Conversion functions

There are three spatial representations in this project, which are pixel coordinates, tile coordinates, and tilemap arrays. Up until this point, the required conversions between the

systems were performed inline in the code, even when some of them were done identically in several places of the project.

In this iteration, a new utility class was created to address this repeated code, providing functions to perform these common operations. The functions implemented are converting between pixel and tile coordinates using the formula 6.2, between tilemap array index and tile coordinates with the formula 6.5, and between tilemap array index and pixel coordinates with the formula 6.6.

6.7.3. Conclusions

This iteration was fruitful thanks to the implementation of the toggle blocks which is a differentiating mechanic, and a new batch of levels using them was added. Still, more mechanics need to be designed and implemented in the following iterations to add variety to the game.

Refactoring the unit conversions was also beneficial to improve legibility and code maintenance.

6.8. Iteration 6

The objectives for this iteration were improving the collision and physics system, adding a victory state to be able to complete a run of the game, and implementing a new mechanic.

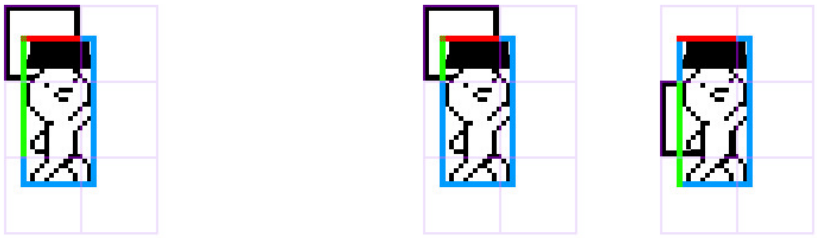
6.8.1. Improved collision system

There were some critical bugs at this point of development in the collision system relative to how overlaps between the player and the environment were being corrected. The most notable issue came when colliding with a platform from below, which caused the player to be pushed out of the screen in a span of a few frames due to it getting stuck inside solid tiles.

The first step in solving this issue was to trace how the current system was handling collisions in search of a mistake that could cause the problem. In-game testing seemed to suggest that collisions with a tile above the player were always being corrected horizontally instead of vertically. Reviewing the formula 6.4 that calculates the overlap between the player and a tile, the issue became apparent: using the sprite's height to calculate the correction of the overlap was producing wrong results when the tile being checked was in the top row of the bounding box. This is because the player sprite is two tiles tall, while the correction on that case needs to be in the realm of one tile's height. Because of this, the correction on the x axis was always smaller than the y axis, and as the collision system undoes the smallest of the two, the y axis was never being selected. See a visual representation of this issue in figure 6.12.

To solve this problem, the correction formula was modified to use the minimum between the player's height and the tile row in the bounding box multiplied by the tile's height. The updated correction (or overlap) formula is as follows:

$$\begin{aligned}(overlap_x, overlap_y) = (&\min(player_{width}, bbox\ row \cdot tile\ size) - |player_x - tile_x|,\\ &\min(player_{height}, bbox\ row \cdot tile\ size) - |player_y - tile_y|)\end{aligned} \tag{6.7}$$



(a) Old formula **(b)** New formula

Figure 6.12: The old method 6.12a produced an excessive y axis overlap correction.
Green: y axis correction Red: x axis correction
Blue: sprite dimensions Purple: bounding box

Another problem was collision outside the bounds of the level. This happened, for example, when jumping on a high platform, which can involve the player passing the upper border of the screen. That situation resulted in undefined behavior, as collisions were checked with data outside the bounds of the tilemap array.

The way this was solved was simple: in the loop that iterates the tiles of the bounding box to check collision for each of them, if the tile coordinates correspond to somewhere outside the bounds of the screen, no collision management is done. With this change, the player can pass the upper bound in a jump without unexpected collisions. On the horizontal bounds, instead of solving the problem through collision, the player is simply prevented from moving past the coordinates that correspond to the limits of the screen.

6.8.2. Game state management

Most videogames pass through a series of different states during their execution, such as menus, victory or game-over screens, gameplay sections, level selector screens, et cetera. Each of these portions of the game usually are self-contained, behave in a distinctly different way from the others (e.g. the input and available actions in a menu screen are separate from the ones that control the player during gameplay), and can transition between each other. Because of these characteristics, the most common way of handling this by implementing a Finite State Machine (FSM).

The general characteristics of the FSM pattern are as follows: "You have a fixed set of states that the machine can be in. [...] The machine can only be in one state at a time. [...] A sequence of inputs or events is sent to the machine. [...] Each state has a set of transitions, each associated with an input and pointing to a state." Nystrom (2014).

There are many sophisticated ways of implementing this pattern, but it is important to remember the current needs and restrictions of the project and the console. The C language does not have the OOP capabilities most implementations rely on, such as interfaces, inheritance and polymorphism. Additionally, Playdate development entails prioritizing performance and working with a restrictive amount of code cache. Because of these reasons, the chosen implementation of the FSM is very simple: using a switch statement on a `currentState` variable, which holds one of a series of states defined in an enumeration. Code for the imple-

mentation used in this project can be found in annex D.

In this iteration the game had two states: the first one for when playing the levels, and a second victory state for displaying a congratulations screen (6.13) when getting to the goal in the last level.

Figure 6.13: Image displayed in the victory game state

6.8.3. Improved player physics

Up until this point the physics in charge of jumps were very basic: unlike horizontal movement, the jump was not using the delta time, which meant its speed depended on the update frequency. It consisted of a counter set at the initial speed of the jump, which on each frame was used to move the player that amount of pixels, and was decremented by one unit. There was also a crude implementation of gravity, which was simply moving the player downwards by three pixels every frame.

In this iteration a new implementation was done, which incorporates delta time to untie the physics from the refresh rate, and uses an approximation of the linear uniformly accelerated movement equations. These equations use the entity variable `vy`, which stores the player's velocity on the vertical axis.

First, a new field in the entity was created called `airborne_time`, which counts the time elapsed since the player was in contact with ground. This timer and the player's `vy` are reset when a collision is undone in the y axis. If the collision occurs with a platform above the player, the timer is set to a slightly higher value than 0 to give a small bounce to the impact before falling back to the ground. Resetting this values also fixed an existing problem with the past implementation, which was that when landing on a higher platform no jump could be initiated until the jump counter reached its final value, creating some frames of unresponsiveness.

Then, on the physics system, gravity is applied to the player's vertical velocity using the equation 6.8. Finally, the player's position is calculated with the formula 6.9, converting the units from meters to pixels.

$$v_y = v_y + 9.8 \times airborne\_time \tag{6.8}$$

$$position_y = position_y + v_y \times \left(\frac{16\ pixel}{1\ meter}\right) \times delta\_time \tag{6.9}$$

A jump starts when the player presses the A button and the `airborne_time` timer is less than 0.2 seconds. This small window of time where the user can jump while airborne is known in game design as "coyote time", and makes the controls feel more responsive by being a little bit permissive with the user's reflexes. The way the jump is initiated is simple: the player's `vy` variable is set to the initial speed of the jump.

6.8.4. Conclusions

Although one of the objectives was not met, which was the addition of a new mechanic, this iteration elevated the game's feel and responsiveness, as well as fixed some very present bugs carried along since the first iterations. The game state implementation will also be used in the future with the addition of menus and other possible states.

6.9. Iteration 7

The objective for iteration 7 was to implement a new mechanic and create new levels fleshing out the ones already implemented.

6.9.1. Vector2f

A new type was created mimicking the `Vector2i` struct already defined in our project. As mentioned before, the Vector2 structs hold two numbers saved under the fields x and y. Defining this type of structure is common practice in game development since many variables go in pairs, such as position in coordinate systems, texture coordinates, or physics values in 2D environments. The new type differs from the existent one in that its values are stored as floating-point instead of integers.

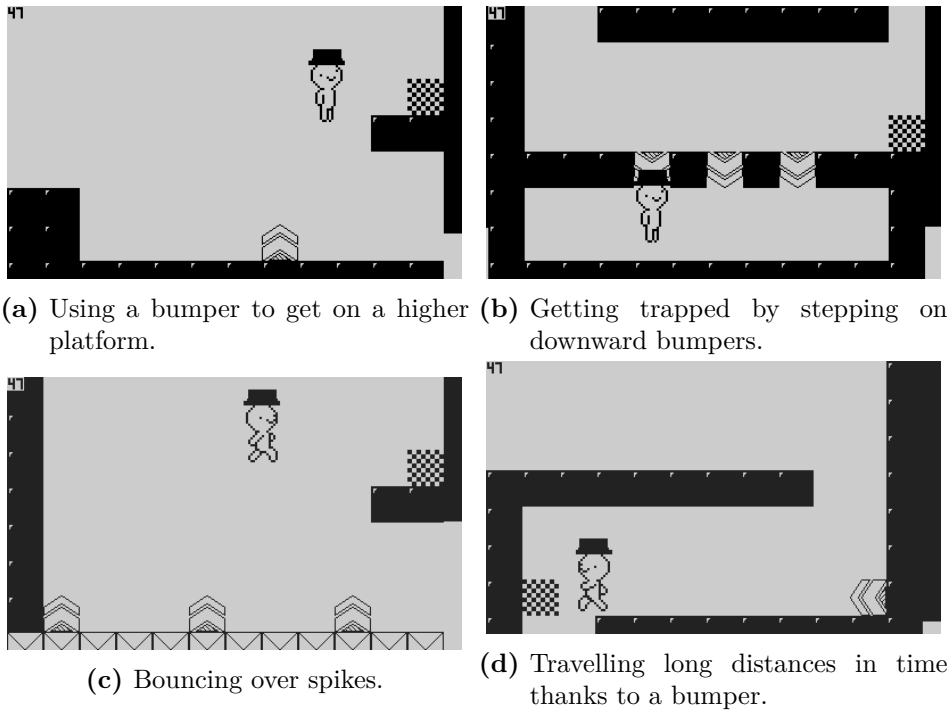
6.9.2. Bumpers

The levels created up until this point were restricted by the distance that the player can walk in 2.5 seconds, with the mechanics focusing on manipulating the environment with the toggle blocks. To break with this limitation, the next mechanic needed to affect the player's movement, allowing for levels where the player traveled greater distances or reached higher ground than what the jump affords them.

The mechanic designed with this in mind were the bumpers, a special type of tile that adds an instantaneous velocity to the player in the direction it is pointing at. Some of the uses for this mechanic are spring-like platforms that propel the player upwards making it bounce; turbo pads, accelerating the player forward or backward horizontally; traps that force the player into a dead end; or diagonal bumpers that launch the player forwards and upwards at the same time. See fig. 6.14 for level screenshots.

A new type of entity was created called `bumper_type`, along with a new `Vector2f` variable called `bumperForces` which stores the velocity vector that will be applied to the player upon contact. Also, a new variable was added to the player to store its velocity on the x axis.

There are eight tiles, one for each direction a bumper can have (left, right, up, down, and diagonals). They are drawn as an arrow pointing towards where the force will be applied.



(a) Using a bumper to get on a higher platform.

(b) Getting trapped by stepping on downward bumpers.

(c) Bouncing over spikes.

(d) Travelling long distances in time thanks to a bumper.

Figure 6.14: Bumper levels

When loading the map, depending on the tile ID a direction in the form of a normalized vector is multiplied to the modulus of a bumper's velocity, obtaining its `bumperForces`.

The interaction with the bumpers begins in the *trigger system*: as explained before, this system detects overlap between the player and special tiles (like the goal or spikes) and performs the required actions. Here, a new case was added so that when the player overlaps a bumper, its `bumperForces` variable is added to the player velocity using vector arithmetic (adding the components of the same axis).

Then, at the end of the *physics system*, the player's position on the x axis is modified to account for its velocity on that axis added by the bumper. If the user is pressing an arrow key and the bumper is afflicting an x axis velocity opposed to that movement, the instantaneous velocity from the input is subtracted from the player's `vx` variable, which only stores the velocity inflicted by bumpers.

Finally, to account for friction, in the collision system the player's `vx` is diminished every time a collision is undone towards the top of a platform, meaning the player is standing on ground.

6.9.3. Conclusion

Implementing the bumpers provided a versatile tool, expanding the level design possibilities and allowing for longer and more complex levels. Thanks to this addition, the second world (as in collection of levels) was crafted, increasing playtime significantly.

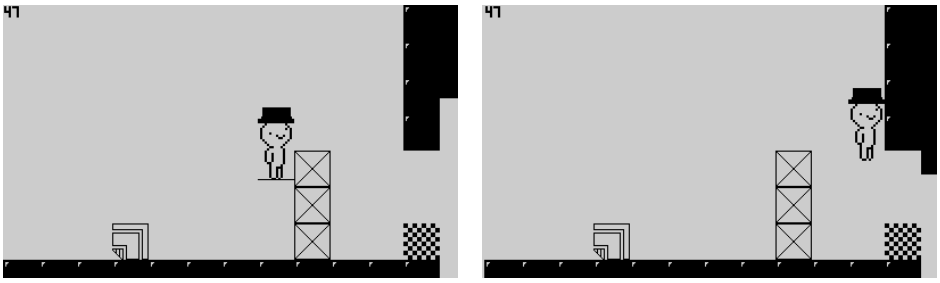
6.10. Iteration 8

The objectives for this iteration were developing an overworld map, improving the bumpers, improving the state machine, and adding a main menu.

6.10.1. Improved bumpers

In their first iteration, bumpers added velocity to the player on each frame they were overlapping them. This sometimes caused movements too big, which were not the desired effect. To fix this, a new boolean variable was added to the entities called `bumperTouchedPlayer`: when the player is not overlapping the bumper, the variable is false. If they start overlapping, the trigger system applies the velocity and then sets it to true, so as to not apply it again on the next update. Once the player stops touching the tile, the trigger system sets the boolean back to false. This means that the velocity will only be added upon commencing the overlap with the bumper.

To add more depth to the mechanic, stepping on a bumper now resets the jump timer, allowing the player to initiate another jump and gain more impulse than just by falling on top of it. This requires timing the jump button press precisely, which raises the skill ceiling of the game. A comparison can be seen in fig. 6.15.



(a) Regular bounce **(b)** Bounce plus jump

Figure 6.15: Jumping after bouncing off a bumper.

6.10.2. New state machine

Although it seemed appropriate to use a simple state machine up to this point, the complexity of the game states grew in this iteration with the addition of menus and the overworld view. For this reason, a new version of the state machine was created.

First, inside the *utils* directory, a new class was added. This file defines a struct called `State`, which stores pointers to a `state_update` function and to a `state_init` function. It also exposes a global variable of this type called `currentState`, which will hold the pointers to the active update and init functions. Lastly, it provides a function called `changeToState(State newState)`, which receives a State parameter, copies it to the `currentState` variable, and if the pointer to the initialization function is not NULL, it calls it.

A new folder called *state* was created to store the classes that represent each state. Each one of them must have at least an update function, and can have an optional initialization

function. For each one of the states, a variable needs to be created in the *State.h* file, so as to avoid circular dependencies between files when states change between them.

In this iteration, the following states existed:

6.10.2.1. State menu

Shows the title screen with the option to start the game and to go to settings. No settings are implemented yet, but the extensibility point is created. The user can cycle through the menu options by pressing the right or up arrows of the D-Pad, go to the previous ones with the left and down arrows, and select one option with the A button.

6.10.2.2. State in game

All the logic related to the gameplay state was moved from the *main.c* file to its own state file. One addition was a pause menu option to allow the player to exit a level to the overworld at any time; the Playdate SDK allows adding custom options to the pause menu by using the function `playdate->system->addMenuItem()`. Choosing to exit the level via this option calls the `changeToState()` function passing the *overworld* state as parameter.

When the player beats the last level of a world, the state is changed to the *victory* state.

6.10.2.3. State overworld

This state shows a map with the levels available in one world, similar to how classic *Super Mario Bros.* games display them. As in the *menu* state, the player moves through the options using the D-Pad, and selects a level using the A button. Only levels that the player has already beaten can be selected, except for the first level of the world. Selecting a level calls the `state_ingame_setCurrentLevel()` function so that the player appears in the chosen level, and then changes to the *ingame* state.

6.10.2.4. State victory

Displays a victory screen congratulating the player for having completed all the levels (see fig. 6.13). If the A button is pressed, the current state is changed to the *overworld* state.

6.10.3. Menu hotspot system

States that operate like a menu, like the *overworld* and *menu* states, share a similar set of functionalities. These can be summarized in being able to cycle through a set of options with the directional pad, display a bitmap to mark the active option, and perform an action when pressing the A button on one of them.

This common functionality was implemented as a system to make it reusable between menu states: the `menu_hotspot` system. In addition to the obligatory update function, this system has a configuration function to initialize its member variables when changing state.

A new type of struct was declared in this file called `Hotspot`, which represents an option of the menu. It has as members a `Vector2i` variable marking the position of the selection cursor when this hot spot is active, and a function to be called when it gets selected.

When entering a state that uses this system, it must call the `sys_menu_hotspot_config()` function with the following parameters:

- An array with the `Hotspot` structs present in this menu.
- The length of the hot spots array.
- A pointer to the image (using the Playdate SDK type `LCDBitmap`) that will be drawn as the background. In the current implementation, this system does not render text or any other graphics for the options, and they must be baked into this background image or drawn by the state in any other way.
- A pointer to the image that will be drawn as the cursor.
- A `Vector2i` with the pivot offset for the cursor image, to make its center be a different relative coordinate than the default upper-left corner.

All of these parameters are stored in static variables inside the class, and are used to provide the system's functionality. The index of the currently active option from the array is also stored in a variable.

In the system's update function, `pd->system->getButtonState()` is used to query the device's input. This function returns the buttons currently down, pressed, and released over the previous update cycle. Then, based on the pressed buttons, a set of operations is performed. If the A button is pressed and the current hot spot's pointer to function is set, that function is called. Else, if the right or up arrows are pressed, the current hot spot index is incremented or set to zero if there are no more options after the current one. Lastly, if both of these conditions fail and the left or down arrow are pressed, the current hot spot index is decremented or set to the last value of the array if the active index is the first position.

The update method ends by drawing the background image, and then the cursor image at the position of the currently active hot spot minus the pivot offset.

6.10.4. User testing and design changes

During this iteration, some informal user testing was carried out with family and friends. The reactions were positive, with players finding the game to be challenging but engaging. The less experienced players found completing the levels difficult at first, but were enjoying the game and persevered to the end.

Something that became apparent was that the original plan to lock progression to the next world until the player completed all levels of the current one without failing once was too harsh of a win condition, and would keep less experienced players from being able to play the game. One solution would be to lower the difficulty of the levels, but that was not the issue as the testers enjoyed the challenge.

In the end, what was decided was to allow failure, either by the timer running out or by hitting hazards, and center the progressing condition in an overarching timer from the first level to the last one. This way, there is still a motivation for the player to improve their skills without gatekeeping less experienced users. The timer was represented using a new system: the flying clock system.

6.10.5. Flying clock system

This newly added system provides a representation of the timer under which players must complete each world to progress to the next one.

There needed to be a clear differentiation between a single level's timer and the overarching one of the world. Adding more HUD could confuse players and clutter the screen with information, when this is a game that needs focused and readable visuals.

Instead of a gauge, the solution found was to add a "ghost" character that flies across the levels of the world, enticing the players to race it or try to keep up with it. When arriving at the last level, the ghost will wait a few cycles, and then disappear; if the player reaches it before that happens, they will have cleared the world and the next one will be unlocked. If not, they will have to try again from the first level.

This system has a timer separate from the timer system's, as the player can touch a hazard and restart the level without affecting the flying clock's progression. This timer accumulates the `deltaTime` until it reaches 2.5 seconds, which is the duration of a single level's time limit, and resets afterwards.

The core movement of the flying clock is a linear interpolation from a level's player spawn position to its goal. This information is obtained from the tilemap utils class, which reads it from the level's tilemap file. A new utility class was added with functions to perform the interpolation, discussed further in subsection 6.10.6. To add flair to the movement, the timer value is halved and converted to radians, passed into a sinus function obtaining a number between 0 and 1, multiplied by 5 to obtain a value ranging between 5 and -5, and added to the clock's y axis position. This simple addition makes the clock hover in a sinusoidal motion, greatly improving the visual result.

The clock is drawn in NXOR mode, which is the negated version of a XOR logic gate and can be summarized as follows: if both inputs are the same, the gate returns 1; else, it returns 0. In Playdate graphics, pixels are represented with a bit as they can only take two values: a value of 1 represents a white pixel, and a value of 0 represents a black pixel. Drawing in NXOR mode, then, means that each pixel of the image is compared to the one beneath it in the display; if the underlying pixel is white, the pixel's color is preserved, but if the background pixel is black, the pixel's color is inverted. The resulting effect is that the clock's color gets inverted when passing through walls and tiles.

To simulate the clock traveling through the levels, a variable called `clockLevel` keeps track of the level the clock is currently at. This variable is incremented every time the clock's timer is reset, which coincides with the clock's interpolated position reaching the goal. Then, the clock is only rendered if it is at the same level as the player.

6.10.6. Linear interpolation

One of the most common operations in game development is linear interpolation, which obtains intermediate values between an initial and final number using a linear function.

It uses the following formula 6.10, with a being the initial value, b being the final value, and t being a number from 0 to 1 indicating the progress between the two values, with 0 returning the initial value, 1 the final value, and 0.5 the point midway between a and b.

$$x = a + (b - a) \times t \tag{6.10}$$

This function was implemented in a new util file called *lerp*, which is a widely adopted quasi-acronym for linear interpolation.

6.10.7. Different tiles per world

The ground tile now changes depending on the current world, differentiating them and boosting the sensation of progress when the player sees the environment change. It is implemented in the tilemap util class by setting the path to the world's tileset image when loading a tilemap depending on the folder it pertains to.

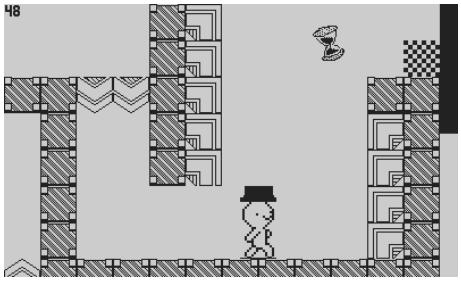


Figure 6.16: Different tiles for world 2

6.10.8. Conclusion

This was one of the most fruitful iterations so far. The application flow is finally complete with the overworld view and the improved state machine. With the clock system, the game's replayability has been greatly improved, inviting players to practice the levels and improve their performance. Finally, there were many other additions that add polish to the experience. The Minimum Viable Product (MVP) is now developed, and all that remains now is a polish iteration to close the product.

6.11. Iteration 9

The ninth and last iteration was dedicated to adding the last mechanics and polishing the existing ones to achieve a well-rounded package, improve the user experience and fix errors.

6.11.1. Saving progress

Up until this point, the player's progress in unlocking the levels was not being persisted, so each time the game was opened the player had to start from the beginning. To solve this, the number of unlocked levels needed to be stored in the file system of the console.

A game's data in the Playdate system is divided between two directories: its .pdx file in the *Games* directory, and a folder named after its package name in the *Data* directory.

The .pdx extension is Playdate's package format, which contains the game's binary file (.bin), a .info file storing metadata such as the SDK version it was built with, and finally the contents of the project's *Source* folder preserving its file structure. When accessing files from code, the *Source* folder acts as the root directory.

Files created in runtime are stored in the *Data* directory, inside the game's folder. When accessing a path from code, both this and the .pdx file's *Source* directory are scanned.

The usual approach to saving data is to create a file in the device, store the information that needs to be persisted inside it, and then read it or modify it in runtime. In the Playdate's case, the save file must be created during execution, as files bundled in the .pdx package cannot be modified.

For a situation that requires saving and loading many variables, the Playdate SDK's JSON parser and writer can be a good fit, as it handles the file's formatting in a clean way and offers reasonable performance in C programs. This tool is used in TinySeconds for loading tilemap information from files, as explained in chapter 6.4.2. However, the requirements for this project are much simpler: the only value needed to restore a player's progress is the number of unlocked levels.

The chosen approach is an unconventional solution that nonetheless covers this project's use cases, which is creating a */saved/* directory inside the game's *Data* folder, and writing an empty file with the number of unlocked levels as its filename. If the data was stored inside the file, more processing and system calls would need to be performed, as the file would need to be listed, then opened, and then its contents parsed. By using the filename to store the variable, we can retrieve its value with only the first step by listing the files in the */saved/* folder.

To write the save file, the path is first created using `pd->system->formatString()`, the Playdate SDK analogue to C's `sprintf()` function which allocates and formats a string, allowing to easily concatenate text and other types of values. In this case, the format is `"/saved/%d"`, where `%d` gets replaced by the variable holding the number of unlocked levels. Then, the */saved/* directory gets deleted using `pd->system->unlink()`, which deletes the files at the provided path. Finally, the new save file is created using `pd->system->mkdir()` passing the formatted path string as parameter.

Reading the save file is even simpler: the function `pd->file->listfiles()` receives a path, a function pointer as callback, and a void pointer to any data we need to access inside the callback. The function calls the callback for every file in the specified path, passing its filename and the void pointer as parameters. This way, a function callback was created that converts the filename to an integer using the C standard library function `atoi()`, and then stores it inside the variable that counts the number of unlocked levels. This variable is passed to the function via the void pointer parameter and cast to an integer pointer inside it.

6.11.2. Drawing the overworld

Even though the overworld state was implemented in the past iteration, the asset used as background in the level selection screen was a placeholder map, with no visual representation of which levels were unlocked and which were not.

A new system was created called drawOverworld, which receives from the overworld state the array of hot spots representing levels and its size, the number of unlocked levels in the current world, and the index of world that is being rendered. The system has a render function which draws an ellipse at each hot spot's position, and then connects them with lines, highlighting the lines between unlocked levels.

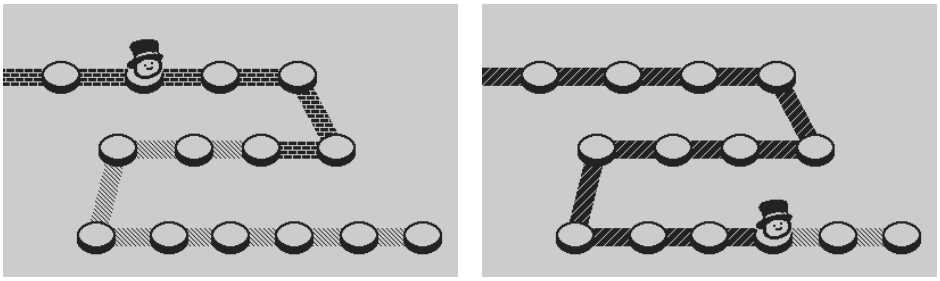
First, two bitmaps are created: one for holding the resulting image, and another one for drawing the level marker once and then reuse it for each hot spot. The level marker is drawn

as a short cylinder in orthographic perspective. It is rendered by drawing two ellipses using `playdate->graphics->fillEllipse()` for the top and the base, and joining their major axis with a rectangle of the same width using `playdate->graphics->drawLine()`. The top ellipse is drawn last and in a different color to improve the faux three-dimensional illusion.

After having created the reusable level marker image, the system iterates the hot spots and draws a line from the current to the next item. If the next level has not been unlocked, the line between it and the current one is drawn in a light pattern; otherwise, it is drawn using a dark pattern. At the end of each iteration of the loop, the level marker bitmap is drawn at the hot spot's position on top of the lines.

Playdate SDK drawing functions that receive a `LCDColor` also accept a `LCDPattern` instead of a solid color; these are defined as an array of 16 bytes, the first eight representing the colors of a row of 8 pixels each, and the last eight the same rows' mask value. To understand the representation of a row as a byte one must look at its binary form: the number 11110000 would mean a row where the first four pixels are white and the last four are black. If the byte represents a mask, a bit value of 1 means a solid pixel, and a value of 0 a transparent pixel.[4]

The first and last elements of the array have a line extend from their position to the leftmost or rightmost edge of the screen, respectively, to represent that progression starts in the past worlds and continues in the next ones. Depending on the world being rendered, a different pattern is used for the paths between unlocked levels.



(a) World one **(b)** World two

Figure 6.17: Programmatically drawing the overworld

6.11.3. Adding music

During this iteration, a music track was composed to serve both as background music and as an audible indicator of the player's performance. The track has a duration of thirty-five seconds, which corresponds to the duration of a world's levels' timers combined ($2.5\ seconds\ per\ level \times 14\ levels = 35\ seconds$).

The music track is used in the flyingClock system in the following way: if the player is in the same level or higher than the clock, the music plays at full volume. If they fall behind, the music's volume descends linearly depending on how many levels of difference are between the player and the clock. The volume change is done smoothly using a linear equation and

[4]This concept can be visualized using the tool `https://ivansergeev.com/gfxp/`, which allows for graphically creating patterns for use in Playdate development.

the clock's timer: first, the x value is calculated as the difference between the player and the clock's levels plus the advanced percentage of the timer. Then, it is substituted in the simplified equation of a line that passes through the coordinates (1, 1) and (3, 0), which means that at one level of difference and no time elapsed the volume will be at full magnitude, and then descend until the music is no longer audible at three levels of difference. See equations 6.11.

$$\begin{aligned} x &= \max(0, clockLevel - currentLevel) + (\frac{accumulator}{timer\ duration}) \\ volume &= \frac{(3-x)}{2} \end{aligned} \tag{6.11}$$

6.11.4. Enforcing the world timer

The last important missing feature was for the clock to block progress to the next level if the overarching timer of the world ran out. There also needed to be a margin of a number of timer iterations once the clock reaches the last level with a visual representation in form of a countdown.

To include these features, the `flyingClock` system was modified. Starting with its initialization function, it now receives the level selected from the overworld as a parameter. If that level is not the first one of the world, the system does nothing, as the clock must be raced from the first level to the last one. A boolean variable called `sys_flyingClock_isGoalOpen` was added to the class, which if true signifies that the player can unlock the next world upon reaching the last goal; if false, progression to the next world is locked. This boolean is only true if the player has started on the first level of the world and the clock's timer has not run out.

The biggest change is in the system's update function. Here, a new method was created to calculate the clock's position depending on the situation that it is at. If the clock is on a level previous to the last one, its movement is an interpolation between the player spawn position and the goal, adding a height offset of 1/4 of the player sprite's height so that it floats over the tiles instead of positioning itself on top of them. If the clock is on the last level, this offset is incremented for the goal so that it floats higher over it. Then, if the clock's level counter has incremented past the maximum levels of the world, for a number of iterations the clock will float over the goal and display a countdown from the number of iterations to zero. After the set margin iterations have passed, the clock flies out of the right margin of the screen by setting the goal as its starting position and a point off-screen as end position, it sets the `sys_flyingClock_isGoalOpen` boolean to false, and it is not drawn again afterward.

The countdown text is drawn using the `pd->graphics->drawText()` function at the clock's position minus double the aforementioned height offset. Unlike the clock, the countdown is visible from any level so that the player knows how much time they have left to complete the world.

In the *ingame* state, a condition was added to loading the next level so it only happens if the `sys_flyingClock_isGoalOpen` value is true.

6.11.4.1. Fence system

Having the condition for locking progress to the next world set in place, it was time to graphically represent it to the player. A new system was created called `fence`, which draws an image of a fence if the `sys_flyingClock_isGoalOpen` boolean is false. The fence is a three tiles high bitmap that is drawn before the goal; the player's jump only reaches slightly above two tiles, making this obstacle insurmountable. In the same way as with the *toggle* tiles system, dynamic collisions are implemented by modifying the tilemap array to mark the tiles underneath the fence as solid. This is done using the previously implemented function `util_tilemap_getTilePointer()`, which given a layer name and a row and column returns a pointer to the array element of the tile that is on that layer and position. To simplify the system, the tile coordinates are hardcoded to the place the fence occupies onscreen. This can be improved by converting the fence's position to tile coordinates and incrementing two times the row's value, or by adding a type of tile to the tilemap and parsing it in the tilemap loading functions. The current implementation is fast and clear, so it was chosen over a more sophisticated approach.

6.11.5. Conclusions

The last iteration in the game's development proved to be very fruitful, as it notably improved the quality of the game and the remaining key features were added. The addition of locking progression to the next world based on the overarching timer completes the designed mechanics of the game, and the programmatically drawn overworld adds the visual progress indication and flair needed on that area.

The music track composed for the game follows the player's progression through the levels pushing them to improve their abilities and try to complete the game at its same pace. Of course, locking the next world imposes an unavoidable challenge to the player, but the music and flying clock already hint towards that goal before it is presented.

More informal testing was carried out, its findings being that players accustomed to platforming games were able to progress to the second world in few tries, while less experienced players were frustrated by that limitation. A way of customizing the experience could open the door to more types of public, done in a way that states how the game is designed to play while providing easier options. As of now, difficulty options are outside the scope of this project, and the focus stays in delivering a challenging game that can be conquered through persistence. The *overworld* state is designed as a way to practice specific levels to prepare for doing them on a row, so that aspect is contemplated in the current design.

Considering all this, we can affirm that the game has successfully reached a first finished version.

7. Conclusions

This chapter summarizes the final state of the project, the lessons learned during its development, and a personal evaluation of the overall work.

7.1. State of the game

After nine development iterations, *TinySeconds* has reached a state that can be considered a first finished version, with all core mechanics set in place, twenty-eight levels divided between two worlds, an original music track, and three versatile types of special obstacles that provide many level design opportunities.

Most of the planned mechanics have been included in the final product, although some of them were redesigned through development to improve them, substitute them for better approaches or balance the difficulty of the game. Using an iterative methodology afforded the project this high level of maneuverability: as each iteration builds upon the previous ones, ways that the planned features can be improved are soon identified and development can be pivoted with ease.

Developing *TinySeconds* has been an opportunity to understanding and improving my skills on all areas of game development, from writing the game engine, to designing levels and mechanics, to programming physics and game logic, creating 2D art assets, and finally composing music for it. Working for a new, limited hardware was also a very educative experience, and improved my problem solving abilities as an engineer. This experience, as well as the complete development of the game and prototypes, is documented in this Bachelor's Thesis to serve as reference for future developers.

7.2. Improvements

There are several areas in which *TinySeconds* can be improved and expanded upon. Some of these additions would be important to have before releasing the game to the general public, while others would be welcome but not necessary.

- Improving the art of the sprites, tiles, menus and backgrounds. As a one-person driven development, the game's 2D art is functional but far from the level of polish a professional artist could bring to it. The overworld state and the levels have a white background instead of an image, and while this boosts legibility in such a fast-paced game, some unobtrusive art could add visual flair to them.

- Adding sprite animations. As of now, the player changes its appearance depending on the action it is performing (such as standing still, or walking in one or another direction) by drawing the appropriate portion of its sprite sheet image. An improvement would

be to implement an animation system that cycled through a series of images at a chosen frame rate to instead of a still image per action have an animation.

- Creating new mechanics and worlds that utilize them. Some of the unadded ideas for mechanics were patrolling enemies, moving bumpers, harmful laser beams, portals, or levels that require the player to go to the goal and back to the beginning. There were plans for designing a third world combining bumpers and toggle blocks, but it was discarded due to development time constraints.
- Composing new music and adding sound effects. There is only one music track reused for all the worlds, when ideally each world should have a different music theme. Also, no sound effects are present in the game, which if added would provide additional feedback to the actions on-screen.
- Presenting tutorial screens to the player. The double jump mechanic that appears in the last levels of world two have been difficult to understand for the players that have tested the game. Showing a tutorial screen could help mitigate this issue.
- Difficulty and accessibility settings. There are a series of options that could be added to allow less skilled players or players with an impairment to customize their experience with the game. One of such options would be to increase the overarching timer of the worlds to reduce difficulty. Another one would be the option to let the player skip a level after a number of tries. Finally, the toggle block mechanic could use the B button if the player is uncomfortable or unable to use the crank in its designed configuration. This would avoid players quitting the game due to frustration or inability to progress.

7.3. Learned lessons

Working on this Bachelor's Thesis, on *TinySeconds* and on the smaller games, has been a very educational experience covering all the areas of the development of a game and learning to develop for a new platform.

First, much was learned about the Playdate's architecture and capabilities. Creating and developing the game in a low-level language involved coming to understand the characteristics, strengths, and weaknesses of the console, and analyzing them to identify best practices. These considerations were taken into account when developing the game to optimize it and obtain good performance.

The prototypes phase of the project provided an understanding of the different ways to make games for the console and their advantages and disadvantages. Programming in Lua offered a moderate learning curve thanks to its higher level of abstraction, its garbage-collected memory management, as well as a richer set of features from the Playdate SDK. The C language, on the contrary, was found to be less welcoming to beginners but excelled in the performance department, allowing Playdate games to reach higher frame rates thanks to its lower level of abstraction, its manual control of the system memory, and not running on a virtual machine. All this knowledge gained during the prototype phase was then applied to *TinySeconds*' development, making problem solving much easier.

Developing TinySeconds has also been educational in regards to planning processes, analyzing progress and the current state of the project in each iteration, and tailoring the scope

of a project to the available time. It also has been an opportunity to greatly expand my knowledge of the C language, and how to approach low-level development.

7.4. Personal conclusions

Personally, I feel very satisfied with the work carried out during this project, as the Playdate was uncharted territory for me when I first started writing this Bachelor's Thesis, and now I can say I have a good understanding of the console and how to develop for it.

Taking part of the Developer Preview program has been an amazing experience: seeing the community blossom, the console evolve, and the public's anticipation grow for a year has been really exciting. It has also allowed me to catch a glimpse of Panic's processes while they worked on readying the console for release, and I feel happy to have contributed by reporting bugs and helping test the SDK and hardware.

Also, having been able to complete a first version of the game, TinySeconds, I can be proud of, has given me confidence for undertaking future developments, and made me consider the possibility to further polish the product and ready a version of it to release it as a launch title.

References

Carr, R. (2007, March). *Speed test: Switch vs if-else-if.* Retrieved 03/21/2021, from http://www.blackwasp.co.uk/speedtestifelseswitch.aspx

contributors, S. M. W. (2021a). *Blinking block.* Retrieved 06/16/2021, from https://www.mariowiki.com/Blinking_Block

contributors, S. M. W. (2021b). *Mushroom trampoline.* Retrieved 06/16/2021, from https://www.mariowiki.com/Mushroom_Trampoline

contributors, S. M. W. (2021c). *Super mario 3d world.* Retrieved 06/16/2021, from https://www.mariowiki.com/Super_Mario_3D_World

contributors, W. (2021a). *Boxboy! (video game) — wikipedia, the free encyclopedia.* Retrieved 06/16/2021, from https://en.wikipedia.org/wiki/BoxBoy!_(video_game)

contributors, W. (2021b). *Inversion of control.* Retrieved 02/01/2021, from https://en.wikipedia.org/w/index.php?title=Inversion_of_control&oldid=998512915

contributors, W. (2021c). *Minit (video game).* Retrieved 06/16/2021, from https://en.wikipedia.org/wiki/Minit_(video_game)

contributors, W. (2021d). *Rhythm heaven.* Retrieved 06/16/2021, from https://en.wikipedia.org/wiki/Rhythm_Heaven

Durán, F. J. G. (2020, August). *Gameengine ecs: Starfield effect.* Retrieved 11/11/2020, from https://www.youtube.com/watch?v=ighkMUM9-Ww

Frank, S. (2020, August). *Playdate programming livestream.* Retrieved 01/02/2021, from https://www.twitch.tv/videos/608372277

Lee, N. (2019, August). *Playdate's tiny hand-held with a crank is big on charm.* Retrieved 06/16/2021, from https://www.engadget.com/2019-08-28-playdate-hands-on.html

Lindeijer, T. (2019). *Tiled map editor.* Retrieved 02/07/2021, from https://www.mapeditor.org/

Lun, S. (2020, August). *Shang lun's proglog.* Retrieved 11/05/2020, from https://devforum.play.date/t/shang-luns-proglog/1194/3

Mierau, D. (2021, March). *Playmaker progress log.* Retrieved 06/16/2021, from https://twitter.com/dmierau/status/1372321742828412936

Nystrom, R. (2014). *Game programming patterns.* United States: Genever Benning.

Panic. (2020a, August). *Inside playdate* [Device and Lua SDK Manual].

Panic. (2020b, August). *Inside playdate with c* [C language SDK Manual].

Panic. (2021, June). *Playdate official hardware specifications.* Retrieved 06/10/2021, from `https://play.date/#the_specs`

@playdate. (2019, May). *Playdate initial reception report.* Retrieved 11/05/2020, from `https://twitter.com/playdate/status/1131733213083136001?s=20`

Septhon, M. (2021a, June). *Daily driver: Channelling rgb into 1-bit.* Retrieved 06/16/2021, from `https://blog.gingerbeardman.com/2021/06/05/channelling-rgb-into-1bit/`

Septhon, M. (2021b, May). *Daily driver: Pre-rendering ranger.* Retrieved 06/16/2021, from `https://blog.gingerbeardman.com/2021/05/18/prerendering-ranger/`

SHARP. (n.d.). *Sharp memory lcd technology.* Retrieved 11/03/2020, from `https://www.sharpsma.com/sharp-memory-lcd-technology`

List of Acronyms and Abbreviations

API	Application Programming Interface.
CAD	Computer-Aided Design.
CPU	Central Processing Unit.
ECS	Entity Component System.
fps	frames per second.
FSM	Finite State Machine.
GUI	Graphical User Interface.
HUD	Head-Up Display.
ID	identifier.
JSON	JavaScript Object Notation.
MVP	Minimum Viable Product.
NES	Nintendo Entertainment System.
OOP	Object Oriented Programming.
QA	Quality Assurance.
RPG	role-playing game.
SDK	Software Development Kit.

A. Previous experiments

A.1. Lua

A.1.1. Hello world

This prototype consists of drawing a background image, a sprite that can be moved using the directional pad, and adding background music. Due to its simplicity, the code resides entirely in the main.lua file, which just like its C counterpart, is obligatory on every project. The drawing of the player sprite and the background image is done using the Playdate SDK sprite functions. See fig. A.1.

The player image is loaded using playdate.graphics.image.new(). Then, it is added to a new sprite, its pivot moved from the upper-left corner to its center, and the sprite's add() function is called. This is a crucial step, as it indicates the Playdate SDK sprite module that this sprite must be updated and drawn.

Next, the background image is loaded in the same way, and a callback function is registered in the sprite module to establish it as the scene's background. This step is done by calling playdate.graphics.sprite.setBackgroundDrawingCallback(). The callback function receives the position and size of the sprite, which are used to draw only the necessary portion of the background each time, an important optimization in Playdate games.

Following this, the background music is loaded using playdate.sound.fileplayer.new(), and its play() method is called.

The last part of the demo is the playdate.update() function, where input is handled by calling playdate.buttonIsPressed() for each of the D-Pad keys, and the player is moved in the direction of the ones that are pressed.

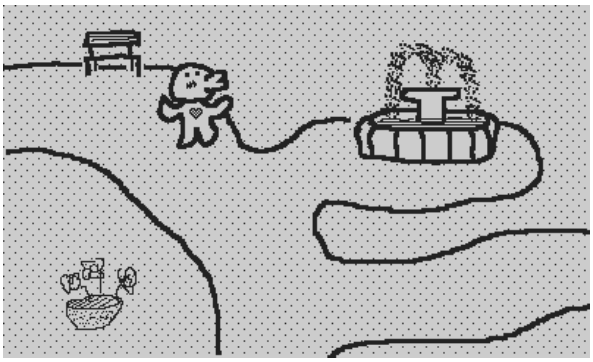


Figure A.1: Hello world Lua

A.1.2. Dr. Mario Mock-up

This was a modification of the first prototype, but with two different layouts depending if the device is oriented in landscape or portrait mode.

In the playdate.update() function, the accelerometer is queried using playdate.readAccelerometer(). By using the test input app preinstalled in the Playdate, one can see that a value of 1.0 on the accelerometer's y axis corresponds to holding the console in landscape mode, while holding it in portrait mode gives a value on -1.0 on the x axis. Using these numbers as reference, the background image is changed in the update method to fit the orientation. See fig. A.2.

As in the previous prototype, the player can move a sprite around the screen, with the added ability to rotate it in 90^{o} increments using the crank. This feature is implemented by using the `playdate.cranked` callback, which fires every time the crank's angle changes. Inside this callback, we query the crank's absolute angle using `playdate.getCrankPosition()`, and depending in which quadrant of its circumference it is currently at, the pill is rotated.

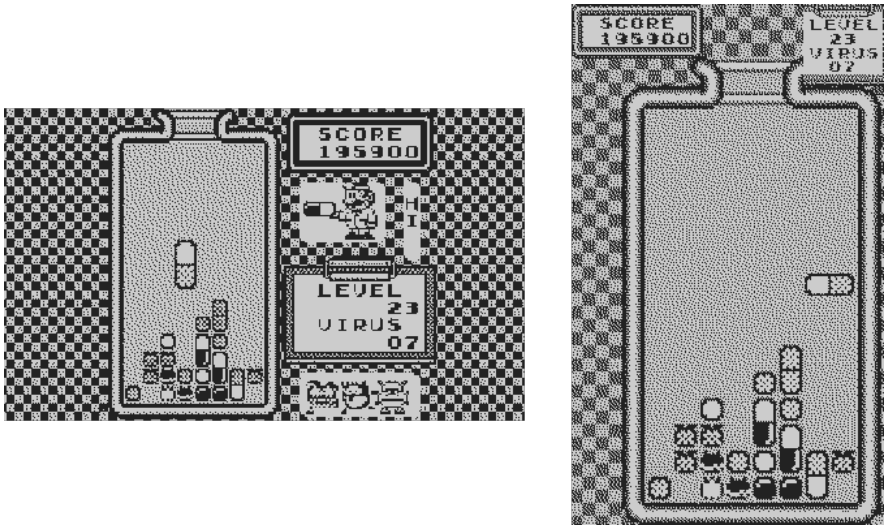


Figure A.2: Dr. Mario Mock-up

A.1.3. Lay down surprise

This demo featured a first animation test and continued exploring the use of the accelerometer to determine device orientation. When the user opens the application, a screen mysteriously prompts them to lay on their backs holding the device over their head. Doing this plays an animation of a pug licking the screen along with a humorous song.

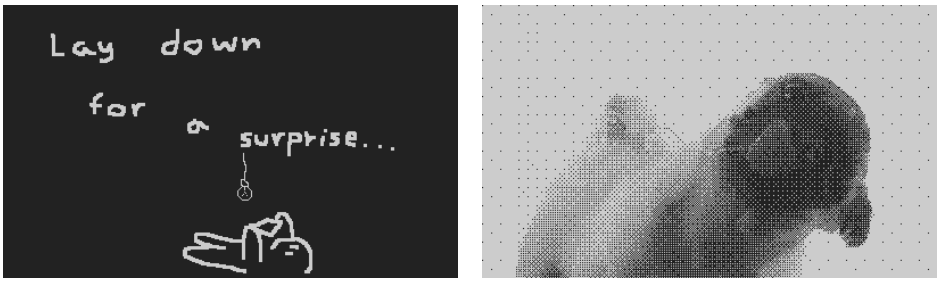
An improvement over the Dr. Mario demo is the use of a timer to discard punctual spikes in the accelerometer, which made the orientation flicker on small movements. This way, the device orientation detection is stable. Each of the two states (instruction image, and dog animation) has a timer associated to it. When the accelerometer enters a state's orientation, the timer increments one unit per frame. If the timer is incremented ten successive frames, it changes to the new state; otherwise, if the accelerometer value changes before that happens, the timer is reset to zero avoiding the flicker.

The Lua Playdate SDK provides some image processing effects that can be used at runtime. In this demo, the `playdate.graphics.image:drawBlurred` effect was used at random intervals to add flair to the instruction image.

The animation was implemented by initializing a `playdate.graphics.imagetable` from a .gif file and creating a `playdate.graphics.animation.loop` from it. These classes are the standard solution for animating a series of images in the SDK, with the ability to specify the delay between frames in the constructor. The animation updates automatically when calling its `draw()` function.

Finally, a cover image, launch sound and animation were added. These assets are displayed in the Playdate menu when the game gets selected, with the animation playing in full screen alongside the sound effect. These elements are set by modifying the *pdxinfo* file at the root of each project. In it, the *imagePath* field needs to be set to the folder inside the *Source* directory that stores the assets for the menu. Inside that path, a folder called *launchImages* contains the frames for the launch animation named by frame number, starting with "1.png". Another field in the *pdxinfo* file called *launchSoundPath* stores the path from the *Source* folder to the custom launch sound effect.

Screenshots in fig. A.3.



(a) Any orientation **(b)** Device facing the ground

Figure A.3: Lay Down Surprise

A.1.4. Tilting microgame

Fellow Playdate developer Nic Magnier suggested in a forum post the idea of creating a crowdsourced collection of microgames in the style of Nintendo's *WarioWare* series. He provided a framework that would envelop the microgames, providing timers, random microgame pulling and win/lose states. For this, I developed a simple game where you must guide a box to the last of a series of floors by tilting the console from side to side. The gaps in the floors are generated randomly in one of four positions, never repeating so the box only falls through one at a time. The floors are drawn using primitives, and I implemented simple physics by using rectilinear accelerated motion equations.

For generating the holes on each floor, the screen is divided into five columns. An array containing numbers from one to five is shuffled at the start of the game, and an element from it is assigned to each one of the floors. If a floor pulls the number 1, its gap will be positioned in the first column of the screen, and so forth. This way, no gaps can be on top of the other, guaranteeing that the box will only fall through one floor at a time.

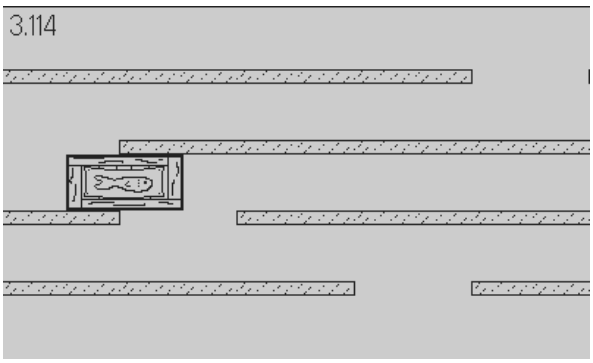


Figure A.4: Tilting microgame

On each frame, the framework calls the microgame's `update()` function, which is divided in three parts: first, the method checks the distance from the box to the gap on its floor, and if it is under a small threshold (5 pixels), it moves the box to the next floor. Then, the `physics_update()` function is called, which updates the movement calculations of the box. Finally, the `render()` function draws all elements on the screen.

The `render()` function draws the ground sprite for each column without a gap and two vertical lines at the sides of the gap to close the floor, as the sprite is made to connect horizontally seamlessly. It also draws the crate at its current position.

Regarding physics, three global variables were created at the top of the program: `mass`, which is the value used as the mass of the crate in physics calculations (set to 1); `gravity`, which represents gravity's acceleration (set to 98); and `force`, which is calculated with formula A.1a, and represents the total force acting on the crate. The `physics_update()` function starts by querying the Playdate's accelerometer using `playdate.readAccelerometer()`. This function returns x, y, and z values between -1 and 1, which are the components of the acceleration unit vector of the console on that axis. The value on the x axis is multiplied by the `force` variable, obtaining the horizontal magnitude of the force. Then, the crate's acceleration is calculated with the formula A.1b, its speed using the formula A.1c[1], and finally, the crate's position is calculated using A.1d. The last value is a multiplier to aproximately adjust the scale of the simulation, its value chosen as a result of testing and adjusting.

$$force = mass \times gravity \tag{A.1a}$$
$$acceleration = force_x \div mass \tag{A.1b}$$
$$speed = acceleration \times deltaTime \tag{A.1c}$$
$$position_x = position_x + speed \times deltaTime \times 200 \tag{A.1d}$$

A.1.5. Rhythm Game

The last Lua project was a musical game designed to focus on the crank input. In this game, the crank controls the angle of a cursor orbiting around the middle of the screen. While a

[1] deltaTime being the elapsed time since the last update call.

song is playing, notes move towards the center of the screen in a preset pattern, and the user's goal is to catch them with the cursor as they enter its reach.

This project features our first implementation in Lua of a game state machine. In it, game states must have one function for the logic update, another one for render, and an optional init function to be called when changing to that state. Then, a GameManager table stores references to the functions of the active state, which are then used to agnostically call the update and render functions from the main application loop. See figures A.5c, A.5a and A.5b.

Because this type of game requires precise movement and perception, the optimization goal was to hit 50 fps performance. With this in mind, the approach to rendering was divided in two phases: first, drawing all elements as they were on the previous frame with inverted colors to selectively clear the screen, and second, drawing the current frame. As mentioned in chapter 3.1.1, this type of area-based rendering is recommended for Playdate applications instead of a full-screen approach, allowing us to hit higher frame rates and extend the console's battery life.

For a rhythm game to be satisfying, the action must be precisely timed to the music that accompanies it. Generally, this is better achieved with hand-crafted content, so it was important to have a way to easily script at what time and from which angle the "notes" would impact. The chosen route was using Audacity[2], a free and open-source sound editor, as Graphical User Interface (GUI). This way, a label track could be used to represent note impacts specifying the angle as label text, while using the waveform representation and the regular interval labels tool[3] to synchronize them to the music track (see figure A.5e). Finally, a simple text parser was written to convert the labels exported from the audacity project to their in-game representation.

The files exported from Audacity are structured in the following way: for each tag, the time they start, the time they end, their text, and a newline character. Note that tags in Audacity can have a duration, acting as a region marker, even though for our purposes this feature is unused. The parser opens the file using `playdate.file.open()`, and then splits each line using the space character as separator, saving the start value as a new note's timestamp and the text value as its angle.

To make this application feel native, custom system menu fields were used for exiting to the main menu from a song, as well as the default crank alert if it was stowed during gameplay. At one point, partial support for the system-wise upside-down orientation was added, an experimental feature for left-handed players, but it ended up being discarded.

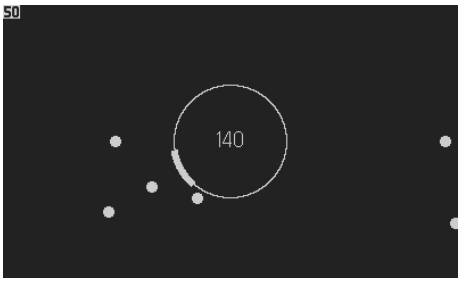
A.2. C

A.2.1. Hello World

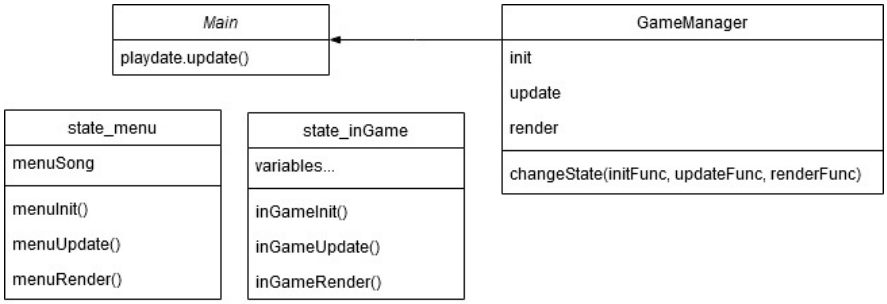
After setting up the C development environment as described in chapter 5, I modified the sample C project that draws a bouncing "Hello World" text around the screen. In my version, I added a background image and modified rendering by drawing only the portion of the image

[2]Official website: `https://www.audacityteam.org`
[3]From the Audacity manual: `https://manual.audacityteam.org/man/regular_interval_labels.html`

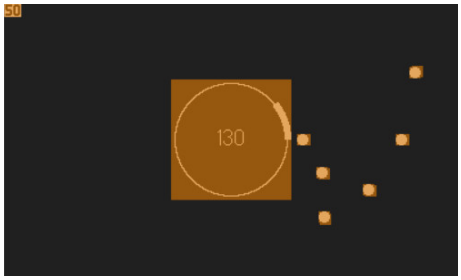
(a) Menu state



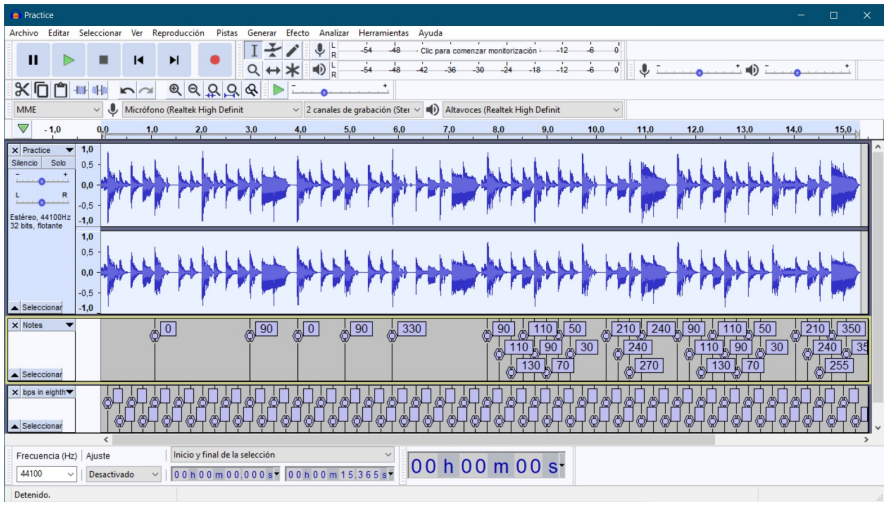
(b) In-game state



(c) Game state machine



(d) Optimized rendering (only highlighted areas are drawn)



(e) Note pattern creation in Audacity

Figure A.5: Rhythm Game

that was under the text on each frame. I also used draw mode NXOR on the text to make it stand out against the background. See screenshot A.6.

Figure A.6: Hello World C

A.2.2. Simplified ECS Starfield effect

Typically, games built in OOP use inheritance to specialize generic classes such as "Actor" or "Enemy" into concrete ones, like specific enemies, items or playable characters. That way, the top classes hold variables and methods common to all derived classes, which allows to generalize methods such as rendering or physics. An ECS approach, in contrast, is an architectural pattern that uses composition instead of inheritance. This means that entities don't hold variables directly; instead, they are simply identifiers linked to components, which are groupings of related data such as physics, health, or transform variables. Then, the game logic is implemented using systems, which are functions that take one or more types of components as input.

The goal with this project was to implement a simple ECS architecture that could serve as a base for future C games. To that end, I followed a series of instructional livestreams by Durán (2020), originally created for Amstrad CPC Z80, and adapted them for the Playdate console. On top of creating the game engine, this series of videos covers how to build a starfield effect, which consists of particles moving from right to left at different speeds to generate the sensation of depth. I further improved this effect by tying particle size to their speed, reinforcing the sensation of fast-moving ones being closer to the camera.

The resulting architecture is not a complete ECS, as components are coupled with entities. This means that every entity has a component of every type associated with them. Nonetheless, it is a useful starting point for C game development, and a good stepping stone for future ECS engine designs. See fig. 6.3.

A.2.3. Full ECS Starfield effect

Following the last project, I decided to implement a full ECS engine, and see if it lead to performance improvements. Components were now separated from the entities, specialized in types and stored accordingly in arrays. Because C is not an OOP language, I emulated polymorphism by defining a C struct for each component type, and then a generic `Component`

struct as a C union of all possible components, along with a type identifier. A union in C is a type of data that can hold one of several types of variable, reserving the size of the biggest one. There is no way to dynamically know what type a union is holding, and that is why the `Component` struct needs the type variable.

All existing components of one type are stored in a ComponentVector struct, which has a type identifier, the `Component` array, and a pointer to the first free position in the array. The component manager owns a ComponentVector variable for each type, and a private function to retrieve the one of the requested type. This is done using a switch statement and returning the pertinent variable.

Listing A.1: component.h class, where the component structs are defined

```
/*****************
 * COMPONENTS
 ****************/
typedef struct PhysicsComponent {
    int16_t x, y;
    int16_t vx;
} PhysicsComponent;

typedef struct SizeComponent {
    uint8_t wx;
    uint8_t wy;
} SizeComponent;

/*****************
 * BOOTSTRAPPING
 ****************/
typedef enum ComponentTypeID {
    type_physics,
    type_size
} ComponentTypeID;

typedef struct Component {
    ComponentTypeID type;
    Entity_id eID;
    union {
        PhysicsComponent physics;
        SizeComponent size;
    };
} Component;

typedef struct ComponentVector {
    ComponentTypeID type;
    Component components[MAX_ENTITIES];
    Component *next_free_component;
} ComponentVector;

// Functions
void man_component_init(void);
void man_component_forall(ComponentTypeID type, void (*ptrfunc)(Component *));
void man_component_destroy(Component *dead_component);
Component *man_component_addcomponent(Entity_id entityid, ComponentTypeID componenttype);
```

The separation of components in arrays of the same type allows to update systems by iterating these component arrays instead of the entities, which typically would improve CPU caching. This improvement would come from the accessed data being sequentially located in memory, allowing the CPU to load that memory portion in a cache of much faster memory

access.

Entities now have an array of pointers to their components so that one system can act on more than one component of an entity. The flow would be the following: the system is called for each component of a type, it accesses the parent entity from that component, and looks for the rest of the required components in that entity.

Listing A.2: entity.h class, the entities now have an array of pointers to their components

```
typedef struct te {
    Entity_id id;
    entity_type type;
    Component *components[MAX_LINKED_COMPONENTS];
    int m_num_components;
} Entity_t;
```

Unfortunately, upon testing the resulting Starfield effect, frame rates had become ~25% slower, dropping from the average 43 fps of the simplified ECS version to an average 12 fps.

A.3. C++

A.3.1. Hello World

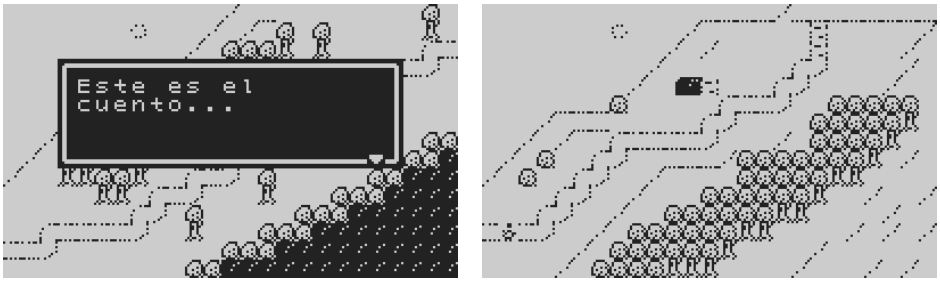
The same as A.2.1, but implemented in C++. The focus of this experiment was to get C++ code running on Playdate, as it is not an officially supported language but I wanted an OOP approach. Using the C configurations, and inspecting the C++ sample project included in the SDK, I modified the CMake configurations, and successfully compiled and run the demo.

A.4. Pulp

A.4.1. Adventure game

In its effort to open game development to beginners, Playdate developer Panic has created a web-based game creation tool called Pulp. Pulp allows users of any skill level to quickly create simple RPG-like tile based games.

Interested in covering all Playdate development possibilities in this bachelor's thesis, I took part of the Pulp beta preview and created a small game as a test. See fig. A.7.



(a) Dialog system **(b)** Tile-based map

Figure A.7: Pulp adventure game

B. Bug reports

Getting a new gaming system before its release to the public is an exciting opportunity, but some roughness in the firmware, SDK and development tools are to be expected. One of the main goals of the Playdate Developers Preview was for the developers to identify bugs and errors before launch so that the Playdate team at Panic could fix them in time.

For this purpose, Panic hosts an issue tracker in their GitLab[1] server, where users can report bugs or feature requests.

During the development of this bachelor's thesis the following bugs were found:

B.1. JSON skipping error

The Playdate C SDK implements its own JSON parser for reading and writing files written in this markup language. The documentation for this feature mentioned the ability to skip JSON key-value pairs individually instead of parsing them.

While writing the code for loading tilemaps in chapter 6.4, I tried to use this feature to speed-up the reading of the map files by not processing unnecessary JSON fields; but then, each time it tried to read the file the program crashed with the error *decode_table expected ','*.

Due to the lack of debugging for C Playdate games in Windows, the source of the error was hard to track down; the JSON file was correct and no comma was missing, so the error message was not being of much help. After some time it became apparent that the JSON parser was interrupting the file reading prematurely, and crashing the whole application.

For testing if this assumption was correct a simplified demo was developed, where the application attempted to open a file, read its contents and print them to the debug console. Further analysis of this demo proved the theory to be correct, and so, a bug was filed in the Playdate GitLab issue tracker.

Here is the full bug report:

B.1.1. Error when skipping a JSON pair in shouldDecodeTableValueForKey()

B.1.1.1. Configuration

- **Version** — Discovered in 0.11.1, present in 0.12.0
- **OS** — Windows

B.1.1.2. Steps

Simple source code to replicate this error is included in BugJSON.zip[2].

[1] GitLab is an online Git source control and project management platform `https://about.gitlab.com/`

[2] `</uploads/135187385976934076 4f58bb4a29d5c9/BugJSON.zip>`

According to the documentation, returning 0 in `shouldDecodeTableValueForKey()` skips the current key-value pair, but this causes an error to pop and stops the json_decoder from further parsing the file.

Listing B.1: skipping JSON pair

```
int util_json_tileset_shouldDecodeTableValueForKey(json_decoder *decoder, const char *key) {
    if (strcmp(key, "donotreadme") == 0) { // Has been tested with simpler conditions too
        return 0; // Skip, causes the error
    } else {
        return 1;
    }
}
```

This has been tested with Tiled-generated JSON files, as well as with simple, handwritten ones. Changing between Windows and Unix line endings doesn't affect the outcome.

B.1.1.3. Expected Results

When returning 0 from `shouldDecodeTableValueForKey()` that key-value pair would be skipped and the json_decoder would continue parsing the JSON.

B.1.1.4. Actual Results

Returning 0 causes the error `decode_table expected ','` at the line that was to be skipped (according to the linenum parameter in `decodeError()`).

B.1.1.5. Frequency

- Always

B.1.1.6. Severity

- Minor

B.1.1.7. Workaround

Not skipping lines and simply ignoring those keys that aren't needed.

B.1.2. Conclusion

After posting the report, a member of the Playdate team expanded on the matter by providing another code example, and the issue was solved in the Playdate 1.0.0 SDK release. At the same time, a working example of using the C JSON parser was added, and more documentation about this feature.

B.2. Clipping rectangle bug

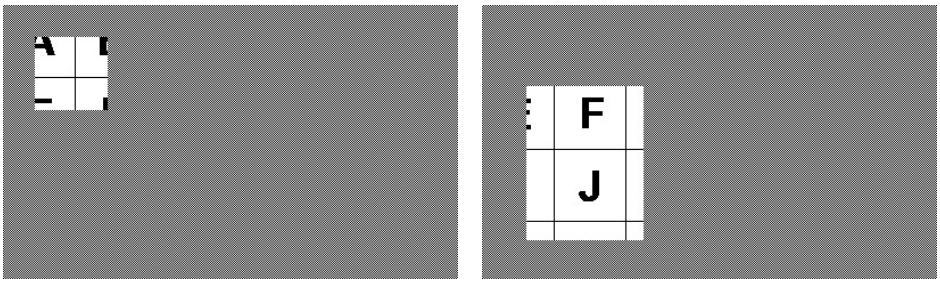
The same week the previous bug happened, still during chapter 6.4, the Playdate 12.0.0 SDK and firmware update was released. It was important to adopt this version as it reworked the

way graphics and bitmaps were drawn from the C SDK, so the previous drawing functions became deprecated and would not work on the final firmware (1.0.0).

Unfortunately, with this update a bug appeared in the way clipping rectangles were created. This broke the drawing of the tilemaps, further slowing down progress in that iteration.

Clipping rectangles are used in my game to select the part of the tilesheet that corresponds to the tile that is being drawn. After some testing, it became clear that the position of the tile was affecting the size of the crop. The x coordinate value was being added to the width of the clipping rectangle, and the same was happening for the y coordinate and the height. Thankfully, this turned out to have an easy workaround: substracting the position of the clipping rectangle to its scale.

For graphically showcasing this effect, I developed a small test application (fig. B.1) comparing the results of drawing a cropped image with the deprecated functions, the new broken functions, and the workaround. It consists of a fullscreen image with a clipping rectangle that bounces around the screen to demonstrate how its position was affecting its dimensions. There was an extra stepped mode to more easily see this dependence.



(a) Expected result **(b)** Bug produces wrong dimensions

Figure B.1: Demo project for the clipping rectangle bug

The full bug report that was posted to GitLab:

B.2.1. Clipping rectangle width/height affected by position

B.2.1.1. Configuration

- **Version** — 12.0.0
- **OS** — Windows

B.2.1.2. Steps

When using `playdate->graphics->setClipRect(x, y, width, height)`, the width and height of the ClipRect are incremented by x and y, respectively. This didn't happen in previous versions of the SDK, or with the deprecated drawing functions.

B.2.1.3. Expected Results

`playdate->graphics->setClipRect(x, y, width, height)` should set a ClipRect of (width, height) dimensions.

B.2.1.4. Actual Results

The ClipRect is of (width + x, height + y) dimensions.

B.2.1.5. Frequency

- Repeatable
- Always

B.2.1.6. Severity

- Major

B.2.1.7. Workaround

Substracting the x position to the width parameter and the y position to the height parameter.

```
c playdate->graphics->setClipRect(x, y, width - x, height - y);
```

B.2.2. Conclusion

No reviews were made to this bug report. The same error was brought up in conversation in the official Playdate Discord server by a fellow developer, and a fix was issued for the SDK 1.0.0 release.

C. Tiled

As by the description on its website, “Tiled is a general purpose tile map editor for all tile-based games, such as RPGs, platformers or Breakout clones” (Lindeijer, 2019). It is a free and open source program, and has the ability to save and load tilemaps in JSON format. Tiled will be used as level editor in this project.

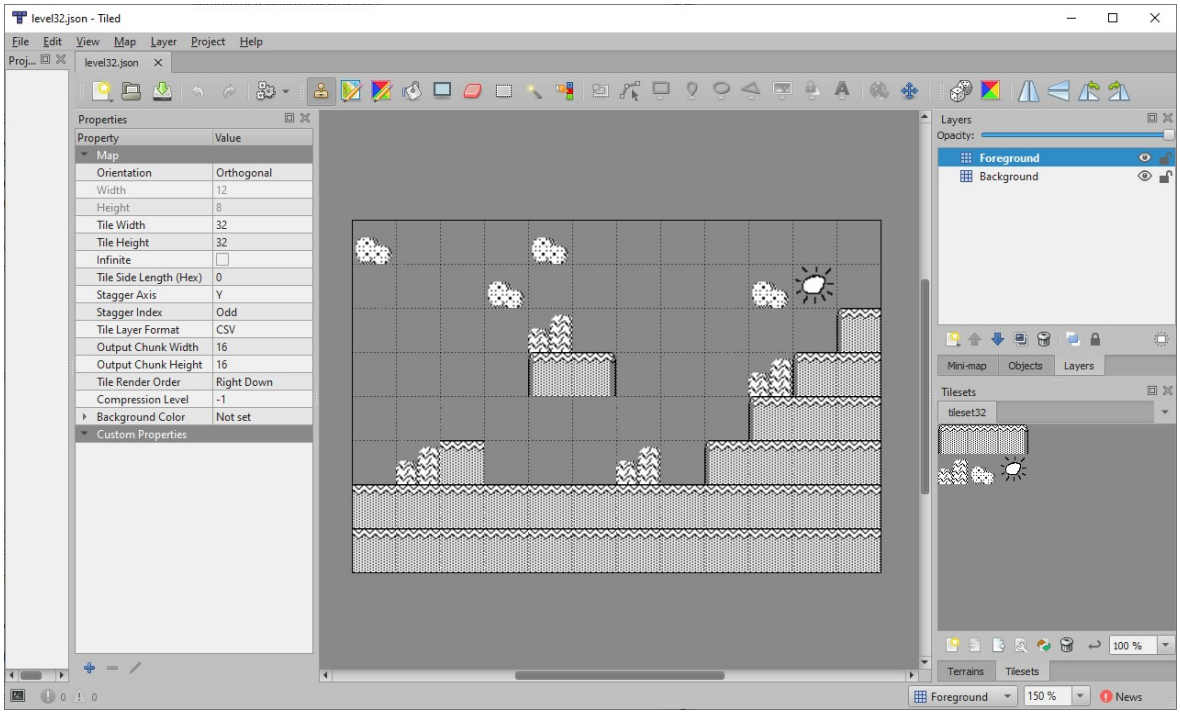


Figure C.1: Tiled interface

D. Simple state machine

The minimalist state machine employed in TinySeconds uses the following code:

Listing D.1: State machine

```
// First, the possible game states are defined.
typedef enum State {
    state_inGame,
    state_victory,
} State;

// This is the variable that indicates which game state is active.
static State currentState;


static int update(void *ud) {
    switch (currentState) {
        case state_inGame:
            inGameUpdate(); // Each state has its corresponding specific update function.
            break;
        case state_victory:
            victoryUpdate();
            break;
        default:
            break;
    }

    return 1;
}
```

Changing to a different game state is done by assigning a different value to the `currentState` variable from inside the update methods.