DESENVOLVIMENTO PARA MEGA DRIVE NA ERA 4K: CONHECIMENTO E NOSTALGIA

Elai Cris Motta de Assis Corrêa – ecmac@cin.ufpe.br
Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Recife – PE – Brasil

RESUMO

Apenas em 2020, a indústria de videogames obteve uma receita global de 179.7 bilhões de dólares, dentre computadores, consoles e dispositivos móveis. Apesar desse mercado lucrativo e do atual poder computacional, ainda há uma comunidade de entusiastas dos jogos antigos, desenvolvidos para hardware agora obsoleto. Algumas dessas pessoas não apenas jogam, elas também produzem novos conteúdos para consoles como Sega Mega Drive e Super Nintendo. Portanto, o objetivo deste trabalho é construir um jogo retrô e no processo, compreender as motivações e os desafios envolvidos, assim como analisar o impacto das ferramentas e tecnologias atuais.

Palavras-chave: videogames; desenvolvimento de jogos; Sega Mega Drive

ABSTRACT

Just in 2020, the videogame industry obtained a global revenue of 179.7 billion dollars, among computers, consoles and mobile devices. Despite the lucrative market and the current computational power, there still is a community of enthusiasts for old games, developed for now obsolete hardware. Some of these people not only play, they also produce new content for these consoles like the Sega Mega Drive and the Super Nintendo. Therefore, the objective of this paper is to build a retro game and in the process, understand the motivations and challenges behind, as well as analyse the impact of current tools and technologies.

Keywords: videogames; game development; Sega Mega Drive

1 INTRODUÇÃO

Devido às suas demandas tecnológicas, a área de jogos eletrônicos impulsiona os avanços de hardware, software e design. Na última década, consoles, PCs e dispositivos móveis passaram por diversas inovações, o que facilitou o desenvolvimento, trazendo: fidelidade visual impressionante, lógica de jogo complexa, excelência em desempenho e design cativante (Pashkov, 2021).

Apesar disso, o conceito de jogos retrô não está desaparecendo. Ele foi, na verdade, incorporado ao mainstream. Jogos como Super Mario Wii e Final Fantasy VII estavam em destaque na E3 de 2009. Mas eles não são verdadeiramente jogos retrô, apenas jogos clássicos, com gameplay clássico. Por outro lado, ainda existe uma comunidade focada em jogos e sistemas antigos, com emuladores ou hardware original, não só jogando, mas também desenvolvendo. (Fulton; Fulton, 2010).

Super Mario 64, lançado em 1996 para o Nintendo 64, é um exemplo de jogo antigo considerado favorito para speedruns, ou seja, finalizá-lo o mais rápido possível (Heubl, 2021). Em 1 de Janeiro de 2011, o recorde mundial para 120 estrelas caiu rapidamente de 02:07:40 para 01:52:40. Desde então, observa-se uma queda frequente de alguns segundos. O recorde atual é de 01:37:50, obtido em Fevereiro desde ano (Speedrun.com).

Em 2006, logo após o lançamento do Wii, a Nintendo publicou o Virtual Console, uma biblioteca paga composta apenas por jogos que foram desenvolvidos para sistemas com tecnologias agora obsoletas. Apenas nos primeiros meses, 10 milhões de downloads foram vendidos. Desde então, várias empresas do mercado de jogos eletrônicos também passaram a oferecer serviços similares, relançaram ou compilaram seus jogos descontinuados (Heineman, 2014). Um exemplo mais recente é o Atari Flashback Classics, lançado em 2018 para o Nintendo Switch.

Visto que já há interesse de ambas as partes, empresas e usuários, em manter o acesso aos jogos antigos, por que é menos comum o interesse em desenvolver novos jogos para os consoles obsoletos? Desde 2009, Krikzz1[1] vende, para diversos consoles, cartuchos chamados de EverDrive, que leem os arquivos ROM – uma cópia em arquivo da memória somente leitura do cartucho – salvos em um cartão de memória. Assim, jogadores podem possuir diversos títulos em um único cartucho. O EverDrive também pode ser considerado um fator motivacional para o desenvolvimento de jogos para os consoles como Sega Mega Drive e Super

1 https://krikzz.com

Nintendo. Ele possibilita que um hardware obsoleto seja capaz de executar software desenvolvido na atualidade, trazendo uma experiência mais fiel que a emulação.

Os objetivos deste trabalho são: Entender as motivações para o desenvolvimento, na atualidade, de jogos para consoles da era 16-bits; Mapear os desafios encontrados e os possíveis impactos positivos da utilização de tecnologias e ferramentas não existentes nas décadas de 1980 e 1990; E, por fim, implementar um jogo simples, Resta Um, para o Sega Mega Drive, aplicando assim os conhecimentos obtidos.

O capítulo 2 fará uma visão geral sobre o Mega Drive e o desenvolvimento de jogos para ele. O capítulo 3 descreverá brevemente o jogo Resta Um e comentará sobre o 33, uma implementação web. O capítulo 4 documentará o gerenciamento e a implementação do *Hungry Cans* para o Mega Drive. O capítulo 5 fará um comparativo entre os casos de desenvolvimento. Por fim, este trabalho será concluído com sugestões de melhorias futuras e uma visão geral da experiência.

2 SEGA MEGA DRIVE E DESENVOLVIMENTO DE JOGOS

O Sega Mega Drive, lançado no Japão em 1988, foi um dos primeiros consoles 16-bits do mercado. Chegou à América do Norte em 1989, sendo chamado de Sega Genesis por razões legais. Em 1990, chegou e à Europa, Oceania e ao Brasil. Com mais de 29 milhões de unidades vendidas, tornou-se o console mais bem-sucedido da Sega, em parte com a ajuda do Sonic como mascote, para concorrer com a Nintendo e o personagem Mario. (Centre for Computing History)

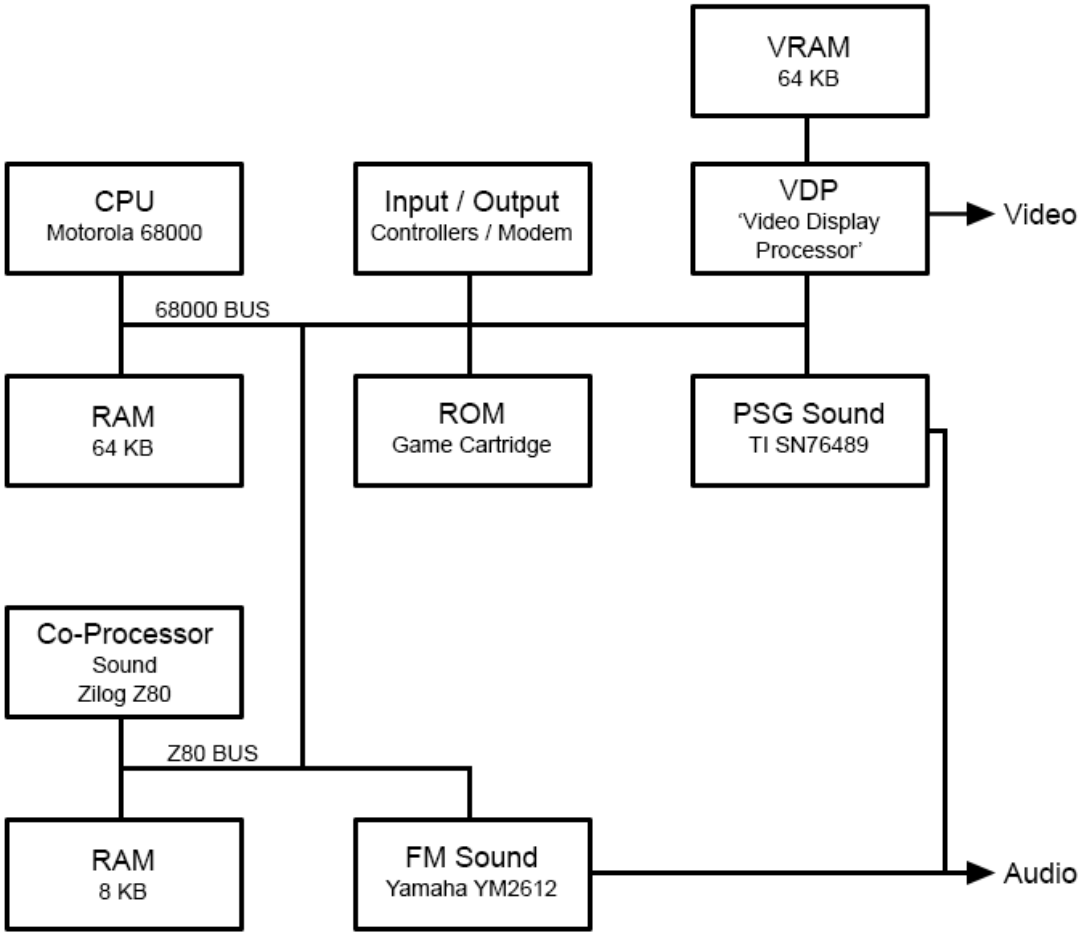
2.1 Arquitetura do Mega Drive

Antes de começar o processo de desenvolvimento de um jogo para o Mega Drive, é necessário entender sua arquitetura. A CPU é um Motorola 68000, um dos processadores mais populares dos anos 1980, presente em computadores pessoais como Apple Macintosh, Commodore Amiga, Sharp X68000 e Atari ST. Também era muito utilizado em sistemas de jogos arcade da Sega e outras empresas. Ao final dos anos 1980, o custo do 68000 já estava baixo o suficiente para ele ser incluso no console. Duas das vantagens foram a familiaridade dos desenvolvedores com a CPU e a facilidade de portar os jogos arcade já existentes (Harrison, 2021).

Além do Motorola 68000 com seus 7.6MHz, o Mega Drive possui também um outro processador Zilog Z80 3.5MHz, servindo principalmente de controlador de som

para as placas Yamaha YM2612 e Texas Instruments SN76489 (Copetti). A YM2612 é um sintetizador por modulação de frequência, com 6 canais de som (Genesis Sound Software Manual, 1992). A SN76489 é um gerador programável de som, com 4 canais programáveis: 3 de tons e 1 de ruído (Engineering staff of Texas Instruments Semiconductor Group).

Figura 1: Arquitetura do Mega Drive



Fonte: Harrison, 2021

De memória, o 68000 possui acesso a uma RAM 64KB para armazenar dados de uso geral e o Z80 contém 8KB para operações de som. Para o processamento dos gráficos, o Mega Drive possui o Video Display Processor – VDP. O VDP roda a 13MHz e suporta várias resoluções, de acordo com a região: até 320x240 para PAL e até 320x224 para NTSC. Os conteúdos gráficos são distribuídos em 3 regiões de memória: VRAM, VSRAM e CRAM. A VRAM, Video RAM, possui 64KB e armazena a maioria dos dados gráficos. A VSRAM, Vertical

Scroll RAM, possui 80B e armazena apenas os dados relacionados a scroll vertical. A CRAM, Color RAM, possui 128B e armazena as 4 paletas de cores, cada com 16 cores, que podem ser escolhidas dentre 512 cores permitidas (Copetti).

2.2 Sega Genesis Development Kit – SGDK

Mantido pelo francês Stéphane Dallongeville desde 2006[2], o SGDK é um SDK gratuito que possibilita o desenvolvimento de jogos para o Mega Drive em C. Ele contém as bibliotecas e algumas ferramentas necessárias para a compilação dos recursos. Para gerar a imagem ROM, é utilizado o compilador GCC com a toolchain m68k-elf e a biblioteca *libgcc*. Também é possível programar com a própria linguagem assembly do Motorola 68000, mas não é necessário. (SGDK, 2022).

Como dito em sua wiki no GitHub, o SDGK utiliza um *makefile* genérico para compilar projetos. Por isso, a estrutura dos diretórios deve seguir um padrão específico:

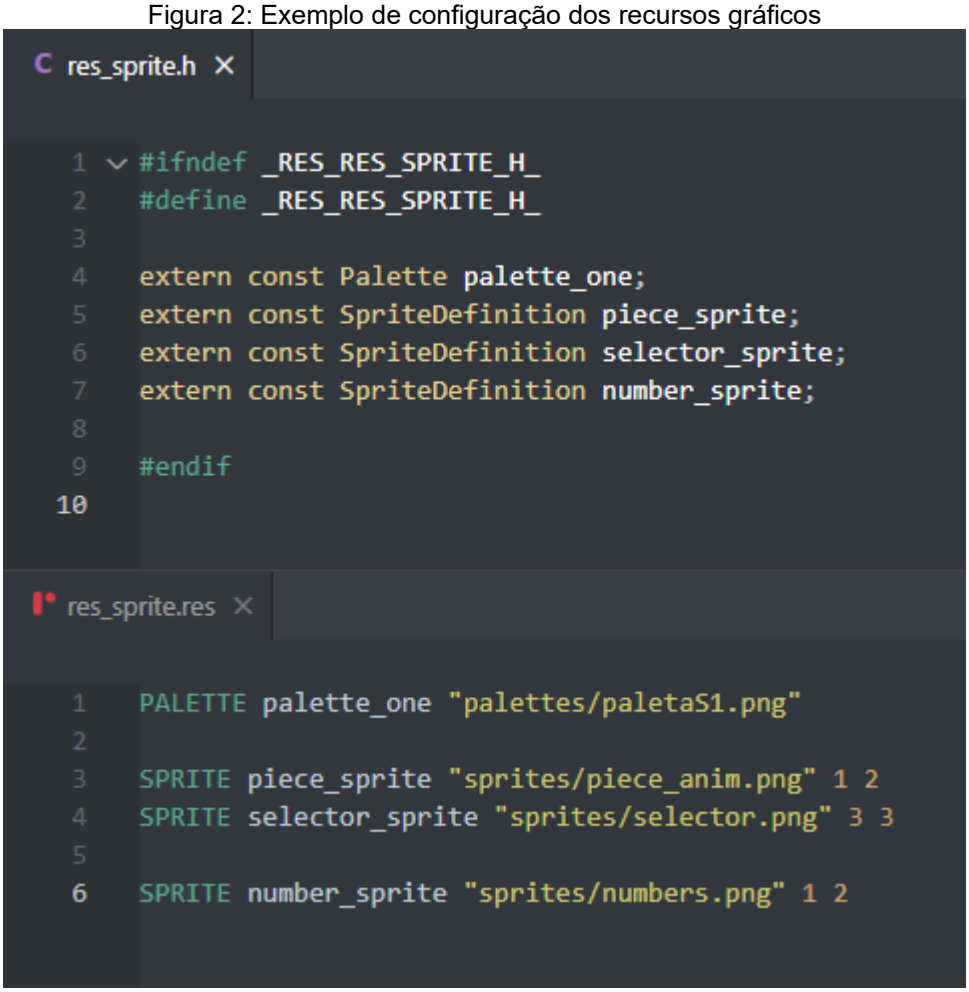
- `src` – arquivos-fonte (subdiretórios até 2 níveis, ou seja, *`src/dir1/dir2/*`*)
- `inc` – arquivos *include* (cabeçalhos, por exemplo, `inc/modulo.h`)
- `res` – arquivos de recursos (por exemplo, imagens e áudio)
- `out` – arquivos de saída (gerados automaticamente)

As imagens podem ser PNG ou BMP e devem estar no modo de cores indexadas. Os arquivos de áudio podem ser XGM ou VGM, que deve ter sido programado especificamente para Mega Drive. Também é possível utilizar arquivos WAV, mas apenas para efeitos sonoros, já que não haveria memória suficiente para um WAV de longa duração.

Para que os recursos sejam compilados, é utilizada a ferramenta *rescomp*, também desenvolvida por Dallongeville. A documentação da ferramenta define o formato em que esses recursos devem ser configurados, em arquivos `res/*.res`. Ao ser chamada, rescomp gera cabeçalhos para cada um dos arquivos `.res`, que declaram os recursos como constantes. Então, para que os arquivos-fonte utilizem as imagens e áudios, esses cabeçalhos devem ser inclusos.

2 http://gendev.spritesmind.net/forum/viewtopic.php?f=19&t=14

Figura 2: Exemplo de configuração dos recursos gráficos



res_sprite.h

```
#ifndef _RES_RES_SPRITE_H_
#define _RES_RES_SPRITE_H_

extern const Palette palette_one;
extern const SpriteDefinition piece_sprite;
extern const SpriteDefinition selector_sprite;
extern const SpriteDefinition number_sprite;

#endif

```

res_sprite.res

```
PALETTE palette_one "palettes/paletaS1.png"

SPRITE piece_sprite "sprites/piece_anim.png" 1 2
SPRITE selector_sprite "sprites/selector.png" 3 3

SPRITE number_sprite "sprites/numbers.png" 1 2
```

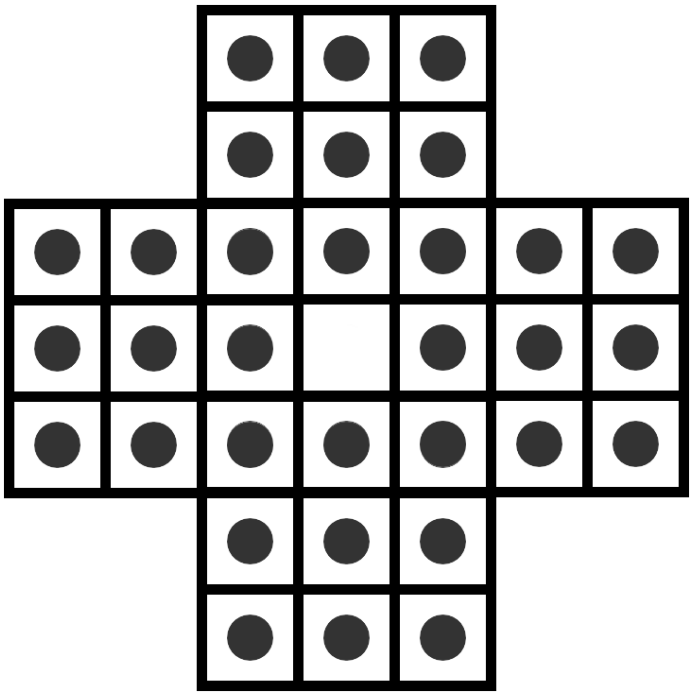
O SGDK é estruturado em várias bibliotecas, cada uma com sua finalidade. Alguns exemplos importantes são: `pal`, manipulação das paletas de cores; `sprite_eng`, manipulação e operações com as sprites; `joy`, leitura do joystick; `map`, gerenciamento dos gráficos de fundo (planos); `xgm`, utilização do driver de som (SGDK, 2022).

3 RESTA UM E O 33

Sua origem é incerta, mas desde o século XVIII, o jogo Resta Um é um passatempo apreciado. Atualmente, destacam-se dois tabuleiros: o inglês, com 33 casas e o francês, com 37 casas. Em ambos casos, o objetivo é o mesmo: deixar uma única peça no tabuleiro. Para isso, movem-se as peças, na vertical ou na horizontal, pulando uma única peça e retirando-a. A peça movida deve cair na casa imediatamente após a peça retirada. Apesar de ser o mais comum, não é obrigatório que a casa vazia seja iniciada no meio (Monte Neto, 1990). Caso chegue-se a um

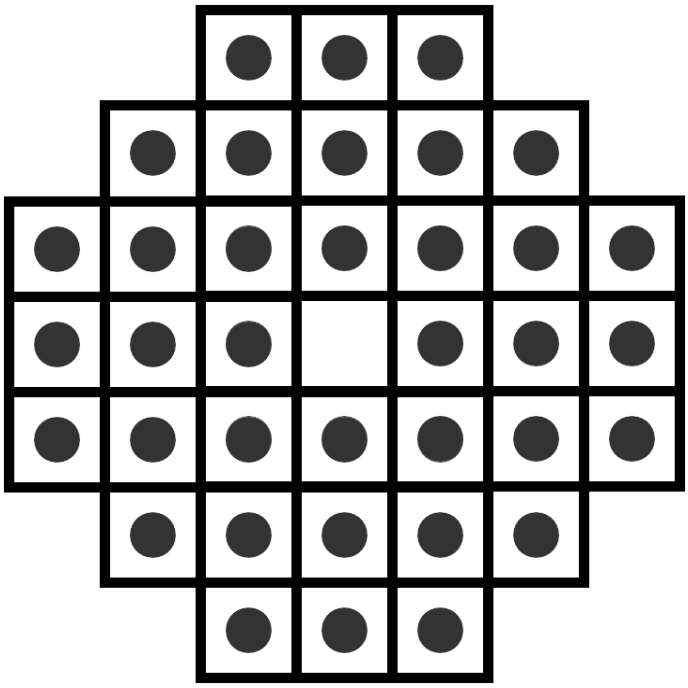
estado com mais de uma peça restante, mas nenhuma movimentação válida seja possível, configura-se a derrota.

Figura 3: Tabuleiro inglês



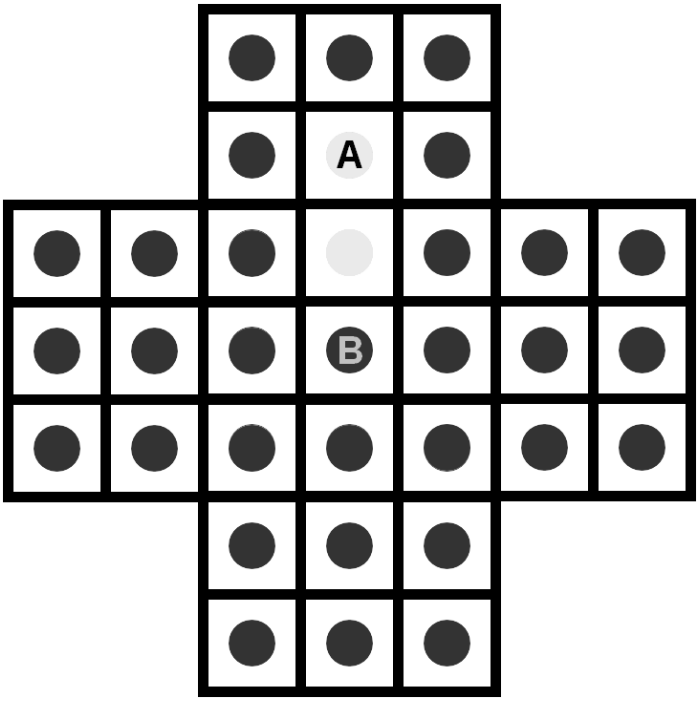
Fonte: Elaboração própria

Figura 4: Tabuleiro francês



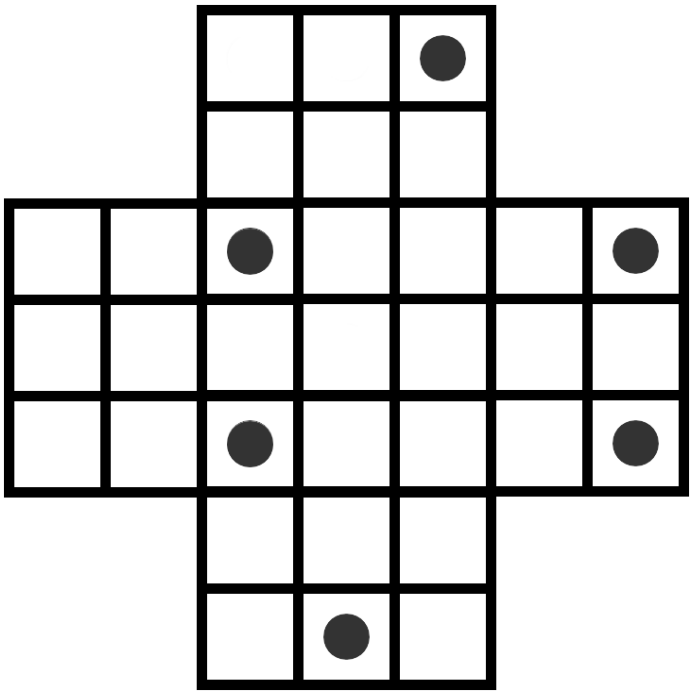
Fonte: Elaboração própria

Figura 5: Exemplo de movimentação válida (A → B)



Fonte: Elaboração própria

Figura 6: Exemplo de derrota



Fonte: Elaboração própria

Em Junho de 2021, publicamos a primeira versão do 33, uma implementação web de Resta Um, desenvolvida em Angular. Ela utilizava o tabuleiro inglês e era bem simples: possuia apenas um botão de reiniciar, um switch para tema escuro ou claro e não detectava automaticamente vitória ou derrota. Para disponibilizá-lo na internet, foi usado o Google Firebase (33, 2021). Após essa experiência, com a disponibilidade de um Sega Mega Drive III e um cartucho EverDrive, surgiu a oportunidade de reimplementarmos o jogo, desta vez para o console.

Figura 7: Jogo 33 (modo escuro)

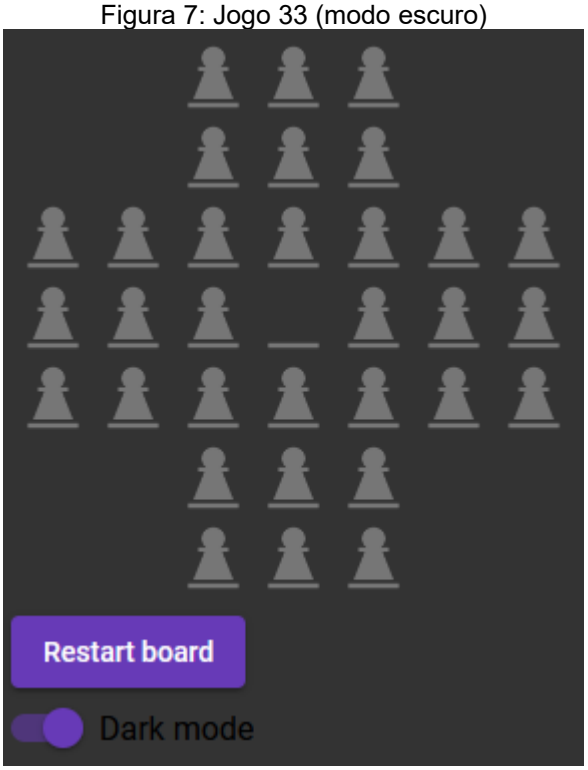
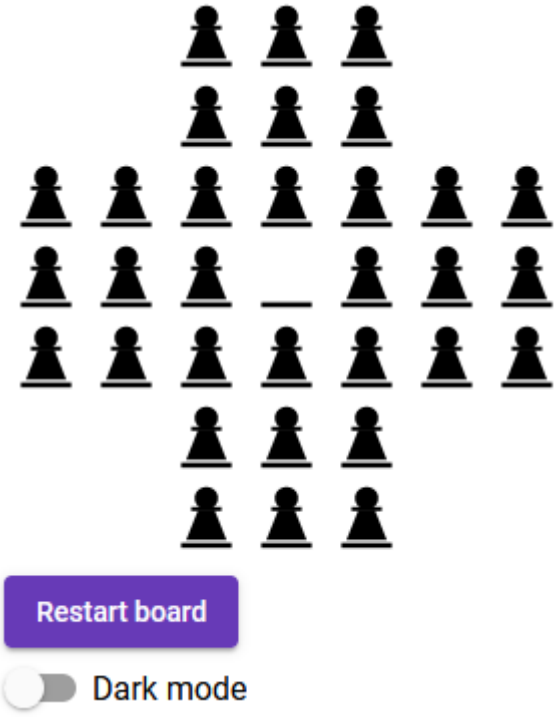


Figura 8: Jogo 33 (modo claro)



4 HUNGRY CANS

Neste capítulo, falaremos sobre a metodologia e o gerenciamento do projeto, conceitos de qualidade de código e a implementação do Hungry Cans, um Resta Um para o Sega Mega Drive que desenvolvemos.

4.1 Metodologia e gerenciamento do projeto

Inicialmente, fizemos uma revisão bibliográfica sobre a arquitetura do Mega Drive e o funcionamento de cada componente de hardware. Esse estudo proporcionou o entendimento das limitações gráficas. Em seguida, realizamos a prototipação de alguns sprites e planos para representar o tabuleiro e as peças. Logo após, iniciamos o desenvolvimento do jogo, utilizando o SGDK, com a linguagem C.

Os arquivos gráficos, planos, sprites e paletas, foram desenhados utilizando os programas gratuitos mtPaint[3] e PhotoFiltre 7[4]. Outras tecnologias utilizadas foram a IDE Visual Studio Code, o emulador Gens KMod[5], o GDB para *debugging*, Git para controle de versão e GitHub para armazenar o repositório e gerenciamento de issues.

Ao início do desenvolvimento, seguimos os tutoriais disponíveis na wiki do SGDK. Também estudamos a documentação do SDK e os projetos-exemplo disponibilizados em seu repositório. E então, foi iniciada a primeira atividade: exibir na tela os gráficos prototipados, utilizando a branch `gfx`.

Assim que foi possível utilizar o Gens KMod para emular a ROM e movimentar o seletor na tela, a branch `gfx` foi mergeada à main e criada a tag `0.0.1`, demarcando o primeiro estado de funcionamento. A partir deste ponto, passou a ser melhor aproveitado o gerenciamento de *issues* do GitHub: cada branch corresponde a uma issue, que é mergeada à main após ser finalizada. Uma exceção foi a issue #22, que foi resolvida junto a #21, então não possuiu branch. Sempre que houve um avanço significativo com um conjunto de issues, foi criada uma nova tag.

Após os testes com o emulador, utilizamos o cartucho EverDrive para também rodar o jogo no Mega Drive III.

---

3 https://mtpaint.sourceforge.net
4 http://www.photofiltre-studio.com/pf7-en.htm
5 https://segaretro.org/Gens_KMod

Tabela 1: Histórico de issues resolvidas

Issue	ID	Tipo	Tag
Escolher uma peça a ser movida	#3	*enhancement*	`0.1.0`
Pular uma peça	#7	*enhancement*	
Adicionar música de fundo	#8	*enhancement*	`0.2.0`
Adicionar chamada aos efeitos sonoros de erro	#12	*enhancement*	`0.3.0`
Exibir tempo decorrido	#4	*enhancement*	
Correções e melhorias gráficas (incluindo #22)	#21	*bug*	
Ação de pular está removendo peça já oculta	#22	*bug*	
Melhorias SOLID e DRY	#17	*enhancement*	`1.0.0`
Correções de posicionamento, seleção e movimentação	#25	*bug*	
Verificar se jogo acabou	#10	*enhancement*	

Também após a tag `0.1.0`, passamos a priorizar alguns conceitos de engenharia de software, como levantamento de features e princípios de qualidade de código. As *issues* do tipo *enhancement* foram todas criadas na mesma semana. *Issues* referentes a *bugs* foram abertas assim que eles foram encontrados, no decorrer do desenvolvimento.

4.2 Qualidade de código

Para facilitar o desenvolvimento e a manutenção do código, foram aplicados alguns princípios de qualidade de código: SOLID, DRY e KISS.

Apesar de serem voltados para a programação orientada a objetos, dois dos princípios SOLID foram proveitosos: Single Responsability – SRP e Open-Closed – OCP. O SRP determina que uma classe deve ter uma única responsabilidade, um único motivo para ser modificada. O OCP afirma que entidades devem estar abertas para extensão, mas fechadas para modificação. Ou seja, quando requisitos mudam, é melhor adicionar código novo, não alterar o que já funciona. Assim, evita-se uma cascata de mudanças (Singh; Hassan, 2022). No *Hungry Cans*, eles foram refletidos pela estrutura modularizada, de acordo com as *features*.

DRY é um acrônimo para “don’t repeat yourself” – não se repita. Já KISS significa “keep it simple, stupid” – mantenha-o simples (Baghel, 2018). Sempre que possível, funções foram criadas para evitar repetição de código e melhorar a

legibilidade. Variáveis, funções e módulos possuem nomes que refletem suas finalidades.

4.3 Implementação

Seguindo a estrutura padrão dos projetos SGDK, o repositório do *Hungry Cans* possui os 4 diretórios necessários: inc, out, res e src. Com o uso do Git, então também estão presentes o diretório .git e o arquivo .gitignore.

O projeto está organizado em cinco módulos: *audio*, *board*, *game_timing*, *stats* e *verifier*. Cada um possui um arquivo de cabeçalho – `inc/*.h` – e um de código fonte – `src/*.c` – com suas respectivas implementações. Além disso, o diretório `src` também possui o arquivo `main.c`, o ponto de início da execução do jogo. Ele define o tamanho da tela, inicializa as paletas de cores e os planos, processa as ações do joystick e realiza as chamadas às funções dos módulos.

Todas as imagens em `res/sprites` e `res/planes` utilizam paletas indexadas de 16 cores, cujas representações estão em `res/palettes`.

Figura 9: Hungry Cans rodando no Gens KMod

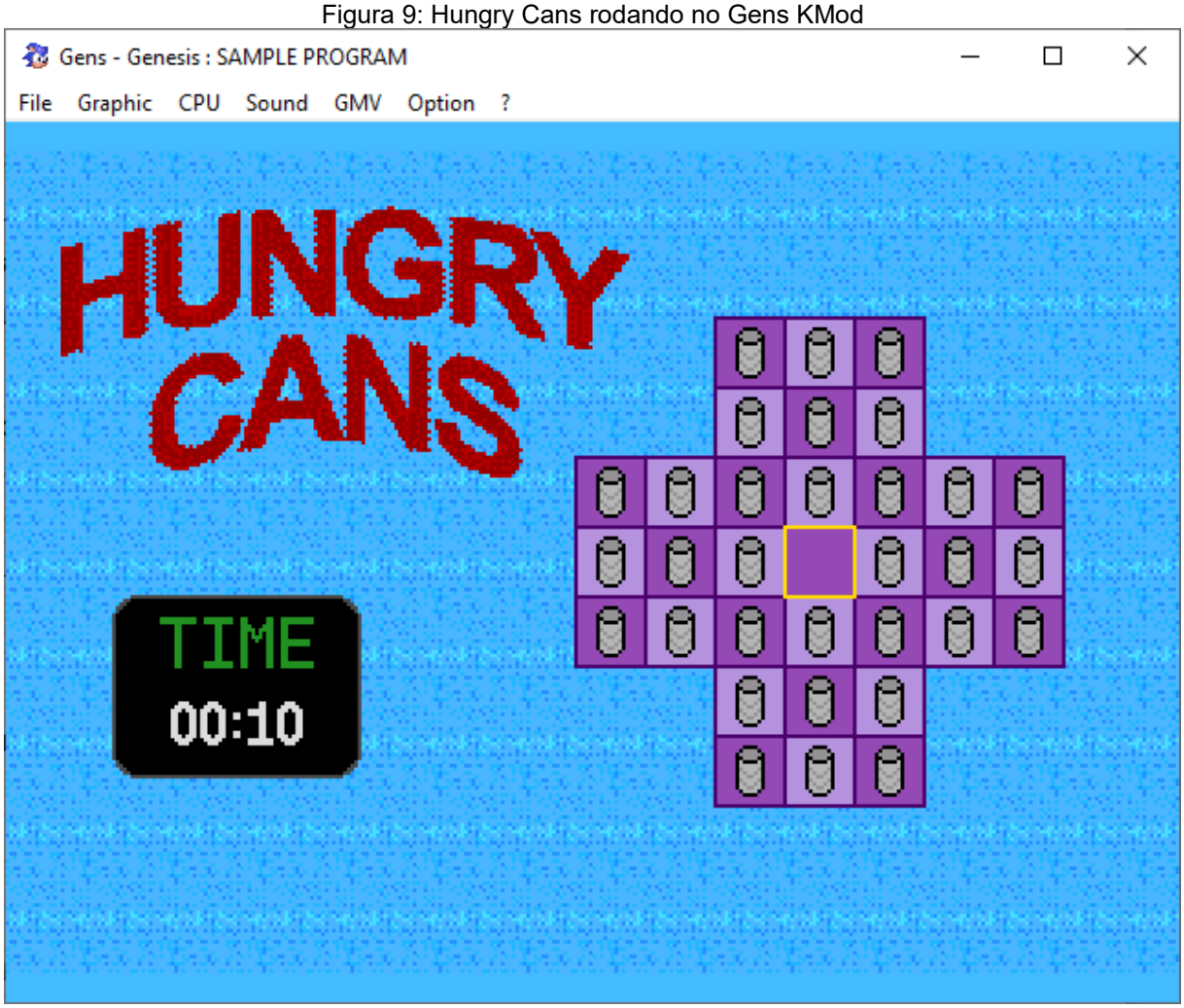


Tabela 2: Descrição dos módulos do projeto

Arquivo de cabeçalho	Descrição
`inc/audio.h`	Música de fundo e efeitos sonoros
`inc/board.h`	Manipulação do tabuleiro
`inc/game_timing.h`	Gerenciamento do tempo de jogo
`inc/stats.h`	Manipulação dos gráficos do contador
`inc/verifier.h`	Verificação do estado de jogo (Em andamento, vitória, derrota)

4.3.1 Música e Efeitos Sonoros

Tabela 3: Constantes do módulo “audio”

Constante	Descrição
`#define BGM_LEN 2`	Quantidade de músicas de fundo
`#define EDGE_ERROR_ID 64`	ID para o efeito sonoro para erro do seletor
`#define MOVE_ERROR_ID 65`	ID para o efeito sonoro para erro de movimentação da peça

Tabela 4: Funções do módulo "audio"

Função	Descrição
`void MUSIC_init()`	Iniciar a execução das músicas de fundo
`void MUSIC_next()`	Tocar a próxima música de fundo
`void MUSIC_pause()`	Pausar a execução da música de fundo
`void MUSIC_resume()`	Retomar a execução da música de fundo
`void SFX_init()`	Iniciar a execução de efeitos sonoros
`void SFX_edgeError()`	Tocar o efeito sonoro para erro no seletor
`void SFX_moveError()`	Tocar o efeito sonoro para erro de movimentação da peça

Para a execução da música de fundo e os efeitos sonoros, foi utilizada a biblioteca *xgm* do SGDK. A função `XGM_startPlay(const u8 *song)` executa o áudio passado como parâmetro. A função `XGM_pausePlay()` pausa a execução, `XGM_resumePlay()` retoma o áudio pausado e `XGM_stopPlay()` interrompe. A função `XGM_setLoopNumber(s8 value)` define quantas vezes o áudio em execução será repetido. `XGM_setPCM(const u8 id, const u8 *sample, const u32 len)` é utilizada para identificar os arquivos *WAV* de efeitos sonoros e `XGM_startPlayPCM(const u8 id, const u8 priority, const u16 channel)` para executá-los.

Para manter uma lista das músicas de fundo, é utilizado um array de ponteiros para arrays de bytes, `const u8[]`, que contém os dados dos áudios a serem executados pelas placas de som. Os arrays de bytes são gerados pelo *rescomp* e declarados em `res/res_sound.h`. Os arquivos originais de áudio foram obtidos na internet e estão localizados no diretório `res/audio`.

As músicas de fundo foram programadas por John Tay[6] e disponibilizadas em seu canal do YouTube[7], com download gratuito dos arquivos originais via Google Drive. Os efeitos sonoros foram compostos por BloodPixelHero[8].

4.3.2 Tabuleiro

Tabela 5: Constantes do módulo "board"

Constante	Descrição
`#define OFFSET_X 161`	Abscissa do posicionamento do tabuleiro na tela
`#define OFFSET_Y 161`	Ordenada do posicionamento do tabuleiro na tela
`#define DIST 19`	Distância entre as peças do tabuleiro (19x19 pixels)
`#define MIDDLE_X 213`	Abscissa da posição central (3,3)
`#define MIDDLE_Y 103`	Ordenada da posição central (3,3)
`#define LEN 7`	Tamanho do tabuleiro (7x7)

Tabela 6: Estruturas do módulo "board"

Estrutura	Variável	Descrição
`typedef struct COORD`	`s16 x`	Abscissa do posicionamento da Sprite na tela
	`s16 y`	Ordenada do posicionamento da Sprite na tela
	`Sprite* piece`	Ponteiro para a Sprite a ser exibida
`typedef struct POS`	`s16 i`	Abscissa da posição no tabuleiro (0 a 6)
	`s16 j`	Ordenada da posição no tabuleiro (0 a 6)

Tabela 7: Funções do módulo "board"

Função	Descrição
`void BOARD_init()`	Inicializar o tabuleiro com as peças
`void BOARD_selectorUp()`	Mover o seletor para cima
`void BOARD_selectorDown()`	Mover o seletor para baixo
`void BOARD_selectorLeft()`	Mover o seletor à esquerda
`void BOARD_selectorRight()`	Mover o seletor à direita
`void BOARD_pick()`	Selecionar a peça a ser movida
`void BOARD_unpick()`	Cancelar a seleção da peça

6 https://twitter.com/johntayjinf
7 https://www.youtube.com/c/JohnTayMusicOrSomethingLikeThat
8 https://freesound.org/people/BloodPixelHero

Função	Descrição
`void BOARD_drop()`	Realizar a movimentação da peça selecionada
`bool BOARD_hasPiece(s16 i, s16 j)`	Verificar se posição contém peça visível

O tabuleiro escolhido foi o inglês, com a casa vazia no meio. Para armazenar seu estado de uma maneira que facilite a movimentação das peças, é utilizado um array 2D: `COORD board[7][7]`. Como uma matriz 7x7 possui 49 posições e o tabuleiro inglês possui 33 posições, as extremidades não podem ser utilizadas. Por isso, elas são demarcadas com `COORD { x = 0, y = 0, *sprite = 0 }` e ignoradas durante a inicialização das peças.

Para cada uma das outras posições, é chamada a função `Sprite* SPR_addSprite(const SpriteDefinition *spriteDef, s16 x, s16 y, u16 attribut)` da biblioteca *sprite_eng* do SGDK. Ela renderiza uma sprite de acordo com uma SpriteDefinition, que tenha sido declarada em `res/res_sprite.h` e retorna um ponteiro para a sprite gerada, armazenado em `COORD board[i][j]`.

Como a posição (3,3) deve ser vazia, então ao início do jogo, a sprite em board[3][3] deve estar ocultada. Para ocultar e reexibir sprites, é utilizada a função `SPR_setVisibility(Sprite *sprite, SpriteVisibility value)` do sprite_eng. Nas mesmas coordenadas da tela que a posição vazia, também é inicializada a *sprite* do seletor, para auxiliar a movimentação pelo tabuleiro. Os posicionamentos das sprites das peças na tela são gerados a partir dos valores de `OFFSET_X`, `OFFSET_Y` e `DIST`:

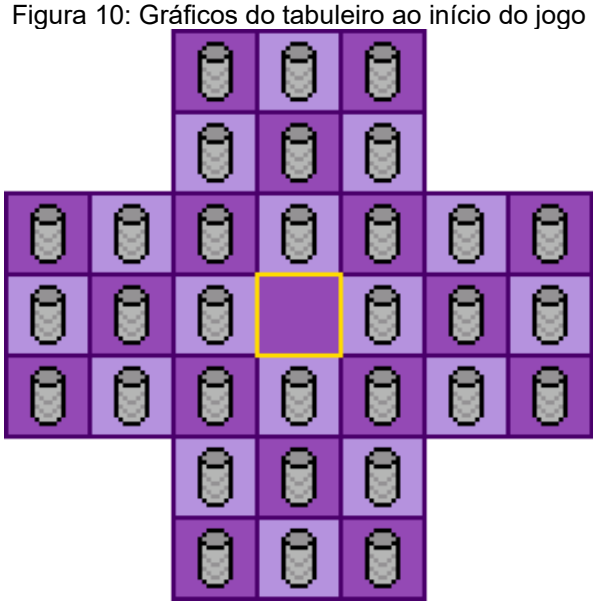
- `board[i][j].x = (DIST * i) + OFFSET_X`
- `board[i][j].y = OFFSET_Y – (DIST * j)`

Tabela 8: Representação do estado inicial de `COORD board[7][7]`

{0, 0, 0}	{0, 0, 0}	**{1, 1, 0}**	**{1, 1, 0}**	**{1, 1, 0}**	{0, 0, 0}	{0, 0, 0}
{0, 0, 0}	{0, 0, 0}	**{1, 1, 0}**	**{1, 1, 0}**	**{1, 1, 0}**	{0, 0, 0}	{0, 0, 0}
{1, 1, 0}	**{1, 1, 0}**	**{1, 1, 0}**	**{1, 1, 0}**	**{1, 1, 0}**	**{1, 1, 0}**	**{1, 1, 0}**
{1, 1, 0}	**{1, 1, 0}**	**{1, 1, 0}**	**{1, 1, 0}**	**{1, 1, 0}**	**{1, 1, 0}**	**{1, 1, 0}**
{1, 1, 0}	**{1, 1, 0}**	**{1, 1, 0}**	**{1, 1, 0}**	**{1, 1, 0}**	**{1, 1, 0}**	**{1, 1, 0}**
{0, 0, 0}	{0, 0, 0}	**{1, 1, 0}**	**{1, 1, 0}**	**{1, 1, 0}**	{0, 0, 0}	{0, 0, 0}
{0, 0, 0}	{0, 0, 0}	**{1, 1, 0}**	**{1, 1, 0}**	**{1, 1, 0}**	{0, 0, 0}	{0, 0, 0}

Para mover o seletor, utiliza-se as setas do joystick. O botão “A” realiza a ação de escolher uma peça a ser movida. É necessário que a posição do seletor contenha uma sprite visível. O botão “C” cancela essa escolha. O botão “B” aplica a movimentação da peça. Para isso, as seguintes condições são necessárias: Há uma peça escolhida e ela não é a mesma do seletor; A sprite na posição do seletor está oculta; O quadrado da distância entre as duas posições é igual a 4; Existe uma peça visível entre as duas posições.

Figura 10: Gráficos do tabuleiro ao início do jogo



4.3.3 Tempo de jogo

Tabela 9: Constantes do módulo "game_timing"

Constante	Descrição
`#define PAL_INC 50`	Frequência em Hz para a região PAL
`#define NTSC_INC 60`	Frequência em Hz para a região NTSC
`#define WIN 1`	Indicador de vitória
`#define LOSE 0`	Indicador de derrota

Tabela 10: Funções do módulo "game_timing"

Função	Descrição
`void TIMER_update()`	Atualizar o contador
`void GAME_start()`	Iniciar contador a partir de zero

Função	Descrição
`void GAME_pause()`	Pausar o jogo e a contagem
`void GAME_resume()`	Retomar o jogo e a contagem
`void GAME_end(bool win)`	Encerrar jogo e a contagem
`bool GAME_isPaused()`	Informar se o jogo está pausado
`bool GAME_isOver()`	Informar se o jogo acabou
`bool GAME_isWon()`	Informar se foi vitória

Para contar o tempo decorrido, primeiro é necessário saber se o sistema está rodando em PAL ou NTSC. A cada atualização de gráficos, o contador vblankCount é atualizado. Assim, a cada 50 ou 60 ciclos, o contador é atualizado em 1 segundo.

Ao iniciar o jogo, a variável interna `vblankCount` é inicializada com zero. Se o jogo for pausado, a atualização de gráficos não incrementará seu valor até que seja retomado. Se o jogo terminar, o contador é finalizado e só reinicializará com um nova partida.

4.3.4 Gráficos do contador

Tabela 11: Constantes do módulo "stats"

Constante	Descrição
`#define CLOCK_X 44`	Abscissa do posicionamento do contador na tela
`#define CLOCK_Y 148`	Ordenada do posicionamento do contador na tela
`#define DIGITS_AMOUNT 4`	Quantidade de dígitos do contador
`#define DIGIT_LEN 8`	Largura do dígito em pixels
`#define COLON_OFFSET 5`	Deslocamento dos dígitos após o “:”

Tabela 12: Funções do módulo "stats"

Função	Descrição
`void TIMER_initGfx()`	Inicializar os gráficos do contador
`void TIMER_updateGfx()`	Atualizar os gráficos do contador

Responsável por inicializar e atualizar os gráficos do contador de tempo decorrido. A função `TIMER_initGfx()` inicializa os gráficos com quatro zeros.

Quando chamada em `game_timing.c`, a função `TIMER_updateGfx()` atualiza os sprites.

Figura 11: Sprites dos dígitos

Figura 12: Gráficos do contador na tela

Os sprites numéricos se comportam como uma animação. Ao chamar a função `SPR_nextFrame(Sprite *sprite)` do `sprite_eng`, passando a sprite do dígito a ser atualizado, o próximo número em ordem crescente será exibido. Para “9”, o último *frame*, a função reinicia a animação, exibindo o frame “0”. Sendo assim, para que o contador funcione corretamente, as atualizações são feitas da seguinte forma:

- A cada segundo, o dígito que representa a unidade dos segundos é incrementado normalmente, de “0” a “9”
- A cada 10 segundos, o dígito que representa a dezena dos segundos é incrementado, exceto seja “5”, então será manualmente retornado a “0”
- A cada 60 segundos, o dígito que representa a unidade dos minutos é incrementando normalmente, de “0” a “9”
- A cada 600 segundos, o dígito que representa a dezena dos minutos é incrementado. Como o limite de tempo do jogo é 1 hora, ele não passará do *frame* “5”

4.3.5 Verificação de estado

Tabela 13: Constantes do verificador de estado

Constante	Descrição
`#define MAX_PIECES 32`	Quantidade inicial de peças no tabuleiro
`#define EMPTY_START 3`	Posição inicial da casa vazia (3,3)

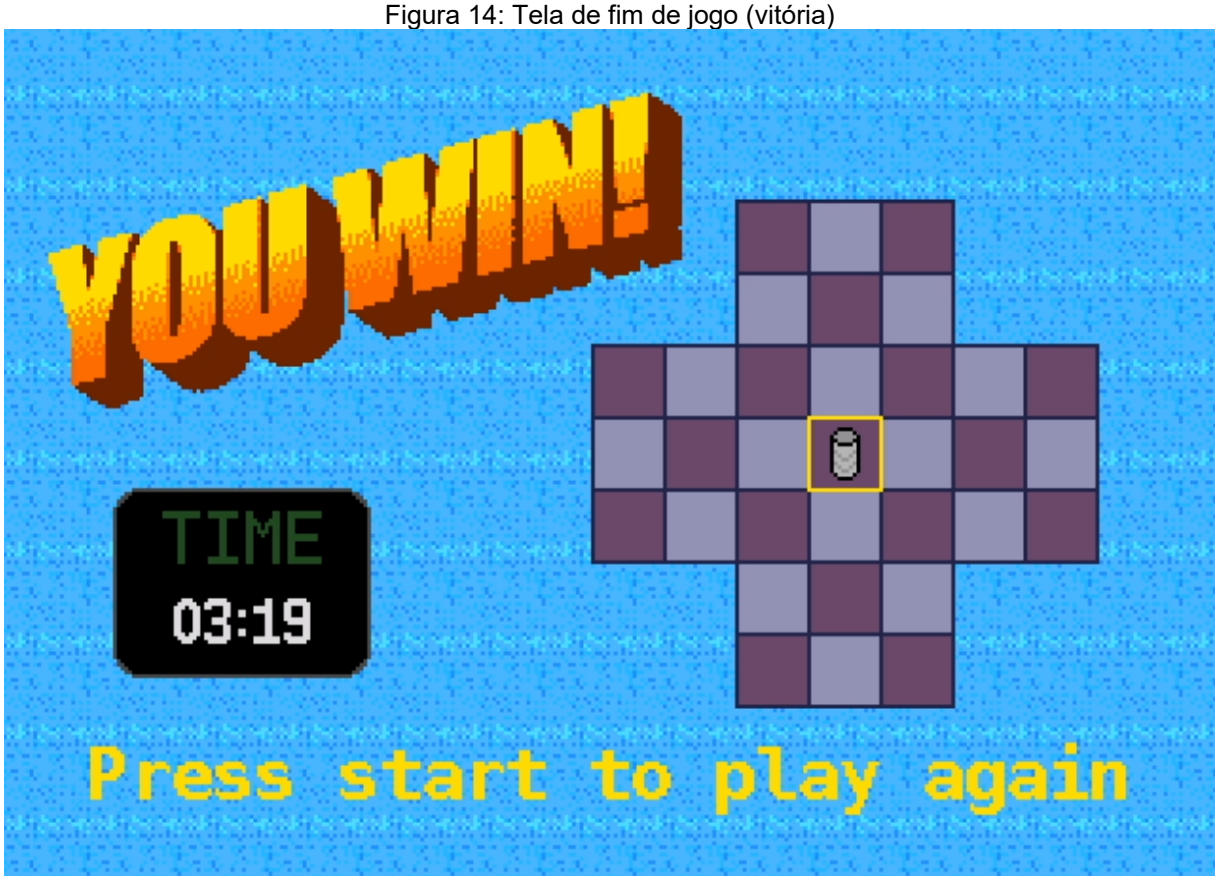
Tabela 14: Funções do verificador de estado

Função	Descrição
`void VERIFIER_init()`	Inicializar verificador
`void VERIFIER_update(POS from, POS mid, POS to)`	Atualizar verificador

Para verificar o estado do jogo, é mantida uma lista das casas vazias do tabuleiro. A cada movimentação de peças, a lista é atualizada. Se a lista tiver 32 casas vazias, então resta apenas uma peça e o jogo foi ganho. Senão, a lista é varrida, buscando uma movimentação válida. Se não for encontrada, o jogo foi perdido.

Figura 13: Tela de fim de jogo (derrota)

Figura 14: Tela de fim de jogo (vitória)



5 ESTUDO COMPARATIVO

5.1 Casos da “era 16-bits” (1987 a 1996)

Segundo a Polygon (2015 apud Stuart, 2014), quando perguntado em uma entrevista sobre como os desenvolvedores criavam jogos para o Mega Drive, um designer de produtos da Sega, Masami Ishikawa, afirmou:

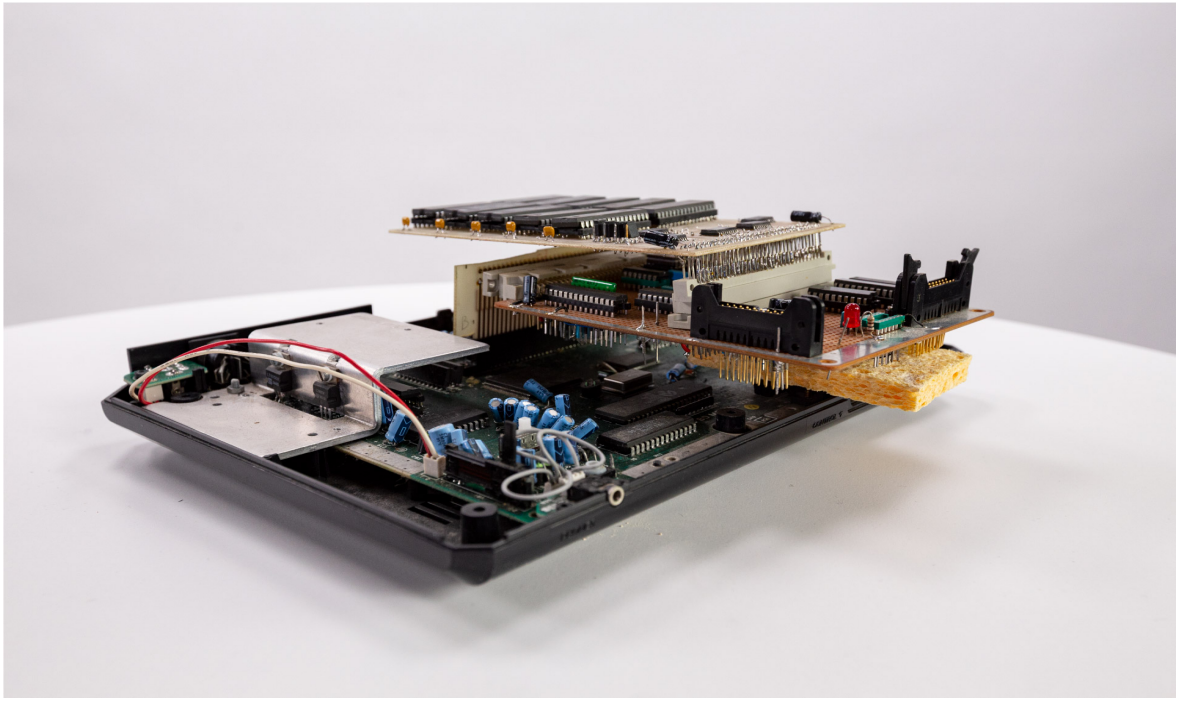
> *Até onde eu sei, eles estavam utilizando emuladores ZAX Z80 para desenvolver os programas dos jogos. Não era nada como os ambientes de desenvolvimentos de software dos dias atuais, que são equipados com bibliotecas e SDKs. Recordo que programadores estavam estudando os códigos-fonte dos programas de teste que eu fiz para debugging a fim de desenvolver cada função. Isso porque os programadores da CS[9] não tinham experiência com o 68000 – até aquele ponto eles tinham trabalhado com o Z80 apenas.*

Ishikawa também comentou sobre o processo de criação dos hardwares, dizendo: “O processo não era como é hoje – nós não perguntávamos as opiniões dos desenvolvedores de software. Nós simplesmente tínhamos uma reunião de mão única quando terminávamos a elaboração das especificações.”

9 Sega CS R&D ou Sega Consumer Research and Development Department

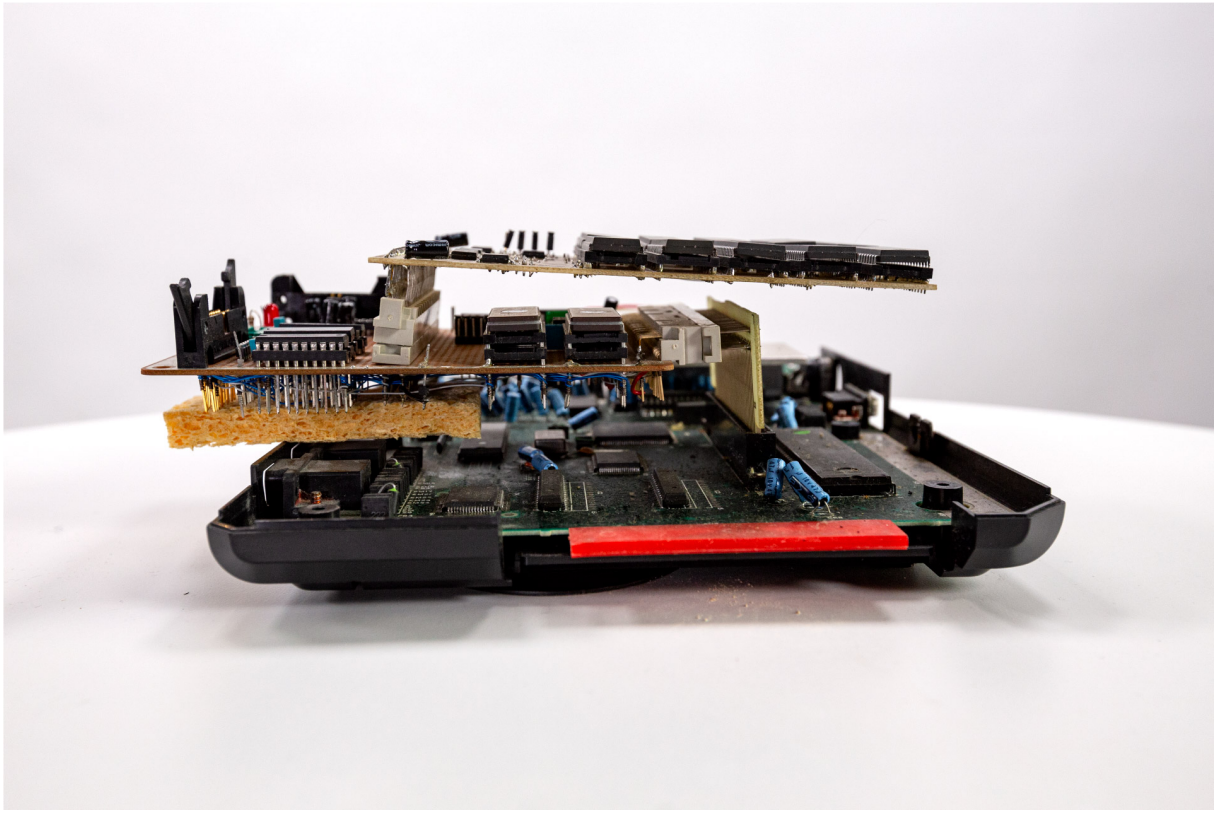
Nestenius (2022) também relatou sua experiência. No início dos anos 90, ele construiu, do zero, o seu próprio kit de hardware de desenvolvimento para o Mega Drive, após muitas pesquisas, tentativas e erros, engenharia reversa e compras de materiais. Ele afirma que sabia que esse projeto seria bem sucedido, pois ele já estava bem familiarizado com o 68000, tanto a nível de chip, quanto à linguagem assembly da CPU. Ele também possuia alguma familiaridade com o Z80. Nestenius possuia um Atari ST, e era nele que a programação acontecia. Ele comentou que utilizava uma IDE chamada Turbo Assembler, que o possibilitou de enviar o código assembly diretamente ao kit. Ele relata que, por causa da faculdade, não conseguiu realmente desenvolver um jogo. Mas foi capaz de exibir alguns sprites na tela e controlá-los com o joystick.

Figura 15: Kit de desenvolvimento criado por Tore Nestenius



Fonte: Nestenius (2022)

Figura 16: Kit de desenvolvimento criado por Tore Nestenius



Fonte: Nestenius (2022)

5.2 Caso do *Hungry Cans*

Figura 17: Hungry Cans rodando num Mega Drive III Brasileiro

Com o SGDK, o EverDrive e o Gens KMod, algumas das dificuldades no processo de desenvolvimento foram sanadas. Compilar, testar e executar ficou muito mais simples e rápido. Outras tecnologias atuais que também trouxeram um impacto positivo foram: Git, para controle de versão; GitHub, para armazenar o repositório e gerenciamento de *issues*; Ferramentas de pixel art, como o mtPaint, para desenhar os gráficos; DefleMask[10], utilizado por John Tay para programar as músicas.

Apesar dos avanços dos últimos 30 anos trazerem vantagens para o processo, as limitações de hardware, principalmente memória e processamento, não podem ser evitadas. Pouca memória acarreta em baixa resolução, paletas de cores restritivas e limite na quantidade de gráficos que podem ser exibidos ao mesmo

10 https://www.deflemask.com

tempo – 80, segundo Harrison (2021). Por isso, a qualidade dos gráficos foi despriorizada a favor do funcionamento adequado do jogo.

Outra limitação de hardware é a reprodução de áudio. Músicas precisam ter sido feitas especificamente para as placas YM2612 e SN76489. Para produzir conteúdo original, seria necessário ter experiência ou aprender a utilizar algum programa como o DefleMask, o que levaria um certo tempo. E como arquivos WAV grandes são inviáveis, escolhemos utilizar músicas e efeitos sonoros já existentes, que apenas requerem os devidos créditos. Entretanto, foi necessário remover as chamadas às funções de efeitos sonoros, pois utilizando o Gens KMod, apenas ruídos eram reproduzidos. A ROM também foi testada em outro emulador, o BlastEm[11], que demonstrou o mesmo problema.

O tempo total para pesquisa, prototipação e implementação do Hungry Cans foi de aproximadamente 5 meses, com eventuais interrupções. Quanto à implementação do 33, ele levou 3 dias. Como desenvolvemos o 33 em Angular, os gráficos puderam ser facilmente desenhados com HTML e CSS. Além disso, não haviam sons, nem verificação de vitória ou derrota. Entretanto, essas ausências não são o motivo do menor tempo de desenvolvimento. Para a concepção do Hungry Cans, foi preciso um longo período dedicado à pesquisa, e outro à prototipação de gráficos adequados. Com o 33, armazenamento, memória e processamento não eram recursos restritos.

Uma demonstração de vitória do Hungry Cans pode ser vista no YouTube[12]. A ROM da versão 1.0.0 pode ser baixada via Google Drive[13].

6 CONCLUSÃO

O SGDK possibilitou a criação de diversos jogos para o Mega Drive, mesmo 30 anos após seu lançamento. Utilizar C ao invés de assembly foi um dos pontos mais cruciais para a viabilidade do Hungry Cans. A compilação de recursos gráficos e sonoros com o *rescomp* acelerou significativamente o processo.

Para o futuro, possíveis novas funcionalidades são: Tela de créditos; Novas músicas de fundo, autorais ou não; Novos efeitos sonoros, não apenas indicativos de erro; Pausar o jogo; Menu inicial; Permitir escolher entre tabuleiro inglês e francês; Permitir escolher a posição inicial da casa vazia.

11 https://www.retrodev.com/blastem
12 https://youtu.be/bRoIkgFScsk
13 https://drive.google.com/file/d/14yYFiSKX5pAw_tmNZsUwwsfpH4uicwT2

Uma consequência desse estudo foi a percepção de que, nos dias de hoje, desenvolvimento não enfrenta o mesmo nível de restrição de recursos. Um computador pessoal pode executar diversas aplicações e reproduzir várias horas de música ao mesmo tempo que exibe gráficos em resolução 4K – ou até 8K – e se comunica com dispositivos do outro lado do planeta. Da mesma maneira que o estudo da história humana é essencial para entender-se o presente, experimentos com tecnologias obsoletas são relevantes para a valorização do estado da arte. Se não pela pura busca por conhecimento, então pela sensação de nostalgia.

REFERÊNCIAS

PASHKOV, Savelii. **Video Game Industry Market Analysis**: Approaches that resulted in industry success and high demand. 2021.

FULTON, Jeff; FULTON, Steve. Creating an Optimized Post-Retro Game. *In*: The Essential Guide to Flash Games. [*S. l.*: *s. n.*], 2010. cap. 11, p. 473-570.

HEUBL, Ben. The new retro-gaming kings. **Engineering & Technology**, v. 16, n. 6, p.58-60, jul. 2021.

Stats - Super Mario 64 – Speedrun.com. Disponível em: <https://www.speedrun.com/sm64/gamestats>. Acesso em 28 jun. 2022.

HEINEMAN, David S. Public Memory and Gamer Identity: Retrogaming as Nostalgia. **Journal of Games Criticism**, v. 1, jan 2014. Disponível em <http://gamescriticism.org/articles/heineman-1-1>

Sega Mega Drive – Centre for Computing History. Disponível em <http://www.computinghistory.org.uk/det/2201/Sega-Mega-Drive>. Acesso em 25 set. 2022

HARRISON, John. **A Guide to the Graphics of the Sega Mega Drive/Genesis**. Raster Scroll Books. 2021. Disponível em <https://rasterscroll.com/mdgraphics>. Acesso em 25 set. 2022.

COPETTI, Rodrigo. **Mega Drive/Genesis Architecture**. Disponível em <https://www.copetti.org/writings/consoles/mega-drive-genesis>. Acesso em 25 set. 2022.

Sega Corporation. **Genesis Sound Software Manual**, p.4, fev. 1992. Disponível em <https://www.smspower.org/maxim/uploads/Documents/ym2612page4.png>. Acesso em 21 out. 2022.

The Engineering Staff of Texas Instruments Semiconductor Group. **SN 76489 AN**. Disponível em <https://ftp.whtech.com/datasheets%20and%20manuals/Datasheets%20-%20TI/SN76489.pdf>. Acesso em 21 out 2022.

SGDK: Sega Genesis Development Kit. Versão 1.80. Stephane Dallongeville. 2022.

33. Elai Cris M. de A. Corrêa. 2021. Disponível em <https://33.criscorrea.dev.br>. Acesso em 21 out. 2022.

MONTE NETO, Luiz Dal. Jogo – Resta-Um: Características e regras do jogo Resta-Um. **Super Interessante**, [*s. l.*], 30 jun. 1990. Disponível em: <https://super.abril.com.br/comportamento/jogo-resta-um>. Acesso em 21 out. 2022.

SINGH, Harmeet; HASSAN, Syed I. Effect of SOLID Design Principles on Quality of Software: An Empirical Assessment. **International Journal of Scientific & Engineering Research**, [s. l.], v. 6, ed. 4, p. 1321-1324, abr. 2022.

BAGHEL, Arvind S. **Software Design Principles DRY and KISS**. 19 abr. 2018. Disponível em <https://dzone.com/articles/software-design-principles-dry-and-kiss>. Acesso em 24 out. 2022.

Polygon Staff. **How Sega Built The Genesis**. 2015. Disponível em <https://www.polygon.com/features/2015/2/3/7952705/sega-genesis-masami-ishikawa>. Acesso em 23 out 2022.

NESTENIUS, Tore. **How I build my own Mega Drive hardware dev kit from scratch**. 2022. Disponível em <https://nestenius.se/2022/01/18/how-i-built-my-own-sega-mega-drive-hardware-dev-kit-from-scratch>. Acesso em 23 out 2022.