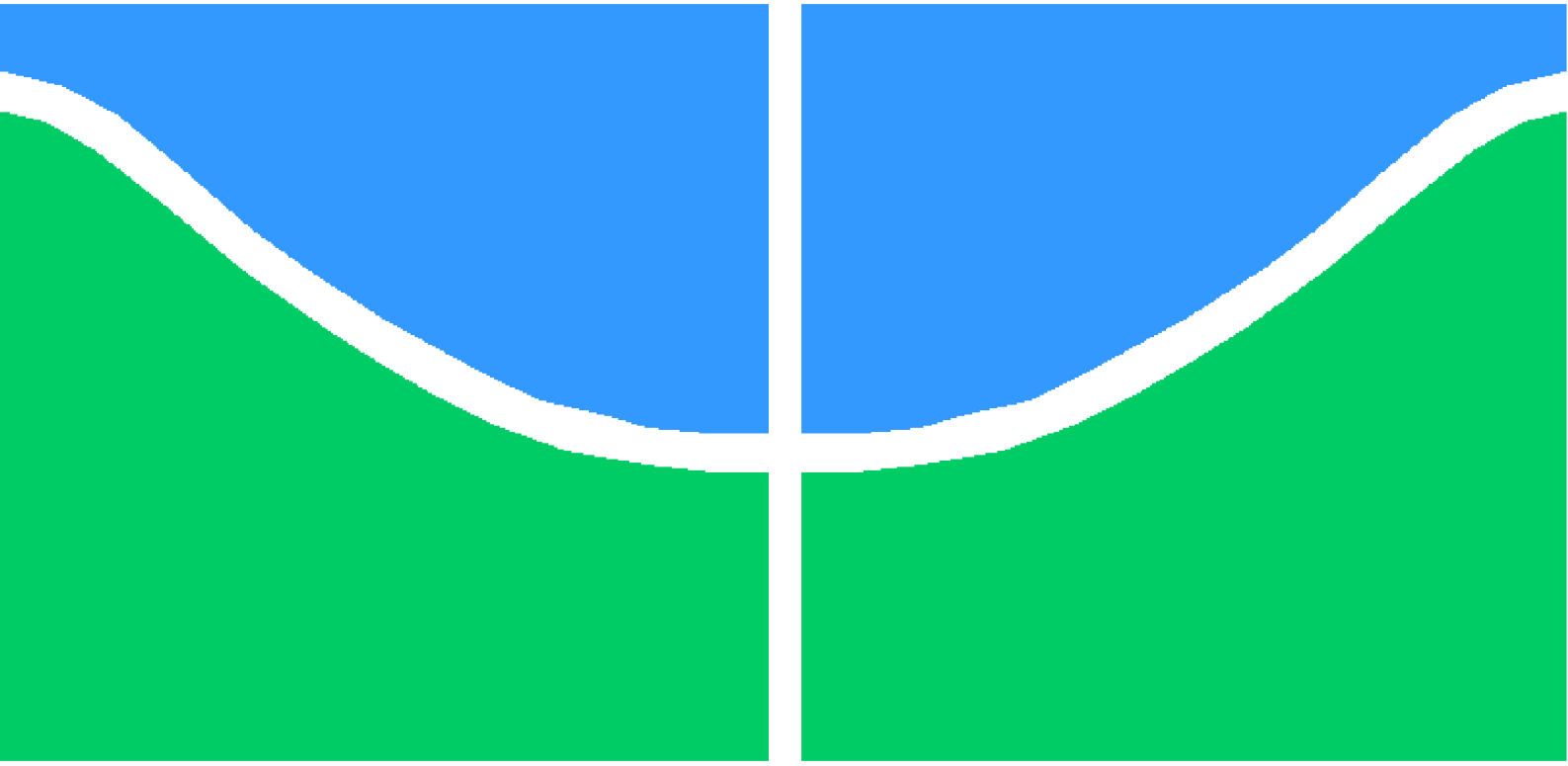


Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

Porte do jogo *Traveling Will* para *Nintendo Game Boy Advance*

Autores: Igor Ribeiro Barbosa Duarte e
Vitor Barbosa de Araujo
Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF
2018

Igor Ribeiro Barbosa Duarte e Vítor Barbosa de Araujo

Porte do jogo *Traveling Will* para *Nintendo Game Boy Advance*

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Coorientador: Prof. Matheus de Sousa Faria

Brasília, DF

2018

Igor Ribeiro Barbosa Duarte e Vítor Barbosa de Araujo
Porte do jogo *Traveling Will* para *Nintendo Game Boy Advance*/ Igor Ribeiro Barbosa Duarte e Vítor Barbosa de Araujo. – Brasília, DF, 2018-
84 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2018.

1. Porte. 2. Game Boy Advance. I. Prof. Dr. Edson Alves da Costa Júnior. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Porte do jogo *Traveling Will* para *Nintendo Game Boy Advance*

CDU 02:141:005.6

Igor Ribeiro Barbosa Duarte
Vítor Barbosa de Araujo

Porte do jogo *Traveling Will* para *Nintendo Game Boy Advance*

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Prof. Dr. Edson Alves da Costa Júnior
Orientador

Prof. Dr. Carla Silva Rocha Aguiar
Convidado 1

Prof. Matheus de Sousa Faria
Convidado 2

Brasília, DF
2018

"Algo que pode ser feito a qualquer momento não é feito nunca."

(RUBIN, Gretchen)

Resumo

Jogos atuais tendem a ter diversos problemas de performance que não são notados devido à grande disponibilidade de recursos presentes nas plataformas atuais. Em plataformas antigas, por outro lado, problemas como esse podem causar impactos significativos à execução dos jogos desenvolvidos. Este trabalho consiste no porte do jogo *Traveling Will*, desenvolvido originalmente para PC, para o *console* portátil *Nintendo Game Boy Advance*. Como resultado deste trabalho, o porte do jogo foi realizado com sucesso, e contém seis níveis jogáveis, assim como o jogo original, além do menu principal e menus de finalização dos níveis.

Palavras-chaves: Porte. Jogos eletrônicos. *Nintendo Game Boy Advance*. Performance.

Abstract

Modern games tend to have some performance issues that aren't noticed due to the great amount of resources available on current platforms. On the other hand, those issues can cause significant impacts to games developed for old platforms. This work aims to port the game Traveling Will, originally developed for PC, to the portable console Nintendo Game Boy Advance. As a final result of this work, the game was succesfully ported to the Game Boy Advance, and contains six playable levels, a main menu and victory and defeat screens, just like the original game.

Keywords: *Porting. Electronic games. Nintendo Game Boy Advance. Performance.*

Lista de ilustrações

Lista de Códigos

Lista de tabelas

Lista de abreviaturas e siglas

GBA	*Game Boy Advance*
PC	Computador pessoal (do inglês, *Personal Computer*)
HUD	*Heads-Up Display*
HID	Dispositivo de interface humana (do inglês, *Human Interface Device*)
RAM	Memória de acesso aleatório (do inglês, *Random Access Memory*)
ROM	Memória somente de leitura (do inglês, *Read-Only memory*)
I/O	Entrada/Saída (do inglês, *Input/Output*)
KByte	Conjunto de 1024 *bytes*
CPU	Unidade Central de Processamento (do inglês, *Central Processing Unit*)

Sumário

Introdução

Desenvolver jogos eletrônicos pode ser uma tarefa complicada, especialmente quando não se tem suporte financeiro, visibilidade e reconhecimento, o que é o caso de muitos desenvolvedores que sonham em seguir carreira nessa área. Em vista dessa limitação de recursos, a estratégia adotada envolve montar equipes pequenas para desenvolver jogos com baixo custo de produção. Os produtos finais deste cenário são conhecidos como jogos eletrônicos independentes, ou, como são popularmente chamados, jogos *indie*.

Jogos *indie* são jogos criados por organizações independentes com recursos limitados operando fora da indústria convencional de publicação de jogos (IUPPA, 2010). Essas organizações, pelo fato de terem recursos limitados, tendem a operar com um número baixo de funcionários. Segundo dados de 2016 da Entertainment Software Association (2017), das quase 2500 empresas de jogos sediadas nos Estados Unidos, 99,7% são consideradas empresas de pequeno porte, sendo que 94,57% foram fundadas domesticamente. Além disso, 91,4% das empresas americanas de jogos empregam 30 funcionários ou menos.

Com investimentos baixos e poucas pessoas envolvidas, essas empresas geralmente optam por modelos de negócio de baixo risco. Seguindo esse modelo, essas empresas tendem a utilizar *game engines* para produzir seus jogos. Segundo Lewis e Jacobson (2002), *game engines* são coleções de módulos de código de simulação que não ditam, diretamente, o comportamento ou ambiente do jogo. Elas incluem módulos para tratar o *input*, *output*, física e comportamentos gerais do jogo, isto é, são construídas de forma genérica. Sendo assim, a utilização de *game engines* no desenvolvimento de jogos independentes traz certas vantagens, como diminuir custos que seriam gerados ao se desenvolver comportamentos e mecânicas gerais, além de evitar retrabalho e facilitar a publicação do jogo para diversas plataformas.

Apesar de tais benefícios, o uso dessas ferramentas pode não ser recomendado, por exemplo, quando performance é um fator essencial no jogo. Jogos são complicados e, em alguns casos, possuem mecânicas e características muito específicas. Em casos assim, a utilização dessas ferramentas gera a necessidade do uso de *workarounds*, que podem gerar desperdícios de processamento e memória e, consequentemente, prejudicar a performance do jogo.

Sendo assim, como prosseguir quando se deseja desenvolver um jogo onde a performance é indispensável (por exemplo, quando se desenvolve jogos para plataformas antigas, onde não há memória ou armazenamento abundantes, ou quando é necessária a utilização máxima de recursos de uma plataforma, como em jogos com gráficos robustos)? Nestes casos, é preciso trabalhar com diversas otimizações como, por exemplo, gerenciamento

eficiente de memória, utilização de algoritmos customizados, otimização de imagens e recursos do jogo, dentre outros.

Um cenário possível onde tais limitações de recursos ocorrem é o desenvolvimento de jogos para *Game Boy Advance*. Este *console* possui memória e poder de processamento limitados quando comparado a *consoles* atuais, o que o torna um bom candidato a objeto de comparação e avaliação de performance de jogos.

A pergunta de pesquisa deste trabalho é "*É possível portar o jogo Traveling Will, desenvolvido para PC pelos autores deste trabalho, para o Nintendo Gameboy Advance, no contexto de um trabalho de conclusão de curso, com performance e jogabilidade próximos da versão para computador?*"

Objetivos

O objetivo geral deste trabalho é reescrever o jogo *Traveling Will*[1], desenvolvido originalmente para PC na disciplina de Introdução aos Jogos Eletrônicos, para o *Nintendo Gameboy Advance*.

Os objetivos específicos são:

- comprimir imagens e músicas do jogo original para reduzir o uso de memória;
- criar módulos para renderização de imagens e texto;
- criar módulos para manipulação de *inputs* dos botões e carregamento de áudio;
- criar módulos para detecção de colisões e manipulação de eventos;
- criar métodos para carregamento do *level design* das fases do jogo;
- executar e testar o jogo desenvolvido na plataforma escolhida.

Estrutura do trabalho

Este trabalho está dividido em quatro capítulos: Fundamentação Teórica, onde serão apresentados os conceitos que serão necessários para o completo entendimento do trabalho; Metologia, onde serão definidos os procedimentos a serem realizados para o porte do jogo; Resultados, onde serão apresentados os resultados deste trabalho e Considerações Finais, onde serão apresentados a conclusão e sugestões de funcionalidades e melhorias para este trabalho.

[1] Jogo musical de plataforma 2D, disponível em <https://github.com/lmanaslu/traveling_will>

1 Fundamentação Teórica

1.1 Desenvolvimento de jogos

Antes de se falar como se dá o desenvolvimento de jogos eletrônicos, é necessário definir o que é um jogo, quais os componentes que fazem parte de um jogo e como um jogo é estruturado.

Segundo Koster (2014), um jogo pode ser definido como uma experiência interativa que dá ao jogador uma série de padrões com desafios cada vez mais difíceis ao ponto que ele eventualmente os domine. Jogos diferentes contém temáticas diferentes, mecânicas diferentes e são executados em plataformas diferentes. Porém, a construção desses diferentes jogos tendem a ser bastante semelhante. Levando em consideração a semelhança de construção de jogos eletrônicos, Dave Roderick diz que "um jogo é somente um banco de dados em tempo real com um *front-end* bonito" (ROLLINGS; MORRIS, 2004).

Na seção 1.1.1 serão detalhados os componentes principais que compõem jogos eletrônicos.

1.1.1 Componentes de um jogo

De forma geral, um jogo é composto pelos seguintes módulos: vídeo, áudio, física, *input*, gerenciamento de recursos e eventos.

- **Módulo de vídeo:** módulo responsável por realizar a renderização de imagens, animações e textos. O vídeo de um jogo é composto por toda a parte de *gameplay* (onde o usuário está efetivamente jogando o jogo) e a parte de *front-end*, que, segundo Gregory (2014, p. 39), é composta pelo *heads-up display* (HUD), menus e qualquer interface de usuário presente no jogo.
- **Módulo de áudio:** módulo responsável por realizar o gerenciamento de áudio no jogo. O áudio de um jogo é composto por músicas de fundo e efeitos sonoros.
- **Módulo de física:** módulo responsável por simular a física do mundo real dentro do ambiente do jogo. A principal responsabilidade de um módulo de física é realizar a detecção de colisões entre objetos do jogo (personagens, cenários, chão, etc.)
- **Módulo de entrada:** módulo responsável por coletar *inputs* advindos do jogador por meio de teclado, mouse, controles, etc.. Também chamado de módulo HID (*human interface device*), este pode ser simples ao ponto de somente coletar o pressionamento de botões, ou mais complexo, chegando a "detectar pressionamento de

vários botões, sequências de botões pressionados e gestos, utilizando acelerômetros, por exemplo" (GREGORY, 2014, p. 43)

- **Módulo de gerenciamento de recursos:** Segundo Gregory (2014, p. 35), este módulo é responsável por prover uma interface (ou conjunto de interfaces) unificada para acessar todo e qualquer tipo de recurso do jogo. Recurso do jogo, nesse caso, é qualquer imagem, *script*, fonte, áudio, mapa localizado nos arquivos do jogo, dentre outros.

- **Módulo de rede:** não necessariamente utilizado em todos os jogos, porém importante ao ponto de ser citado, este módulo é responsável por prover suporte à comunicação (local ou *online*) entre diversos jogadores em um mesmo jogo.

Além dos componentes gerais, todo jogo contém um mecanismo que é responsável pelo controle do estado atual do jogo, conhecido como *game loop*. Castanho et al. (2016) caracterizam a estrutura de um *game loop* por quatro tipos de tarefas (utilizando como exemplo uma partida do jogo *Tetris*):

1. tarefas que serão executadas somente uma vez quando a partida iniciar como, por exemplo, zerar a quantidade de pontos e escolher uma peça inicial;

2. tarefas que serão executadas repetidamente (enquanto a partida não acabar) como, por exemplo, coletar inputs do jogador, verificar se o jogador completou uma linha no jogo, verificar se o jogador atingiu o topo da tela e atualizar a posição da peça que está caindo;

3. tarefas que serão executadas quando situações ou eventos específicos ocorrerem como, por exemplo, atualizar os pontos quando uma linha é completada;

4. tarefas que serão executadas somente uma vez quando o jogo estiver pronto para ser finalizado.

1.1.2 Arquitetura de um jogo

De acordo com Gregory (2014), a separação entre um jogo e sua engine é uma linha turva. Isso se dá principalmente pelo modo como é modelada a arquitetura do jogo. Caso seja utilizada uma arquitetura monolítica, a engine e o jogo serão um só, isto é, não haverá separação clara entre componentes genéricos e específicos do jogo. Por outro lado, caso o jogo utilize uma arquitetura sólida, esta separação é evidente.

1.1.2.1 Arquitetura monolítica

A primeira abordagem trata da construção do jogo utilizando um design de arquitetura monolítico. Nesta arquitetura cada objeto do jogo contém toda a sua implementação de mecânicas de renderização de gráficos, detecção de colisões, física, inteligência artificial, etc. Este modelo não é adequado para jogos mais complexos, pois, de acordo com Plummer (2004), o código não é reutilizável porque as mecânicas estão altamente acopladas com o comportamento dos objetos e, além disso, o design não é flexível, prejudicando a manutenção do código, pois qualquer modificação pode afetar o sistema inteiro. A figura 1 apresenta um exemplo desta arquitetura.

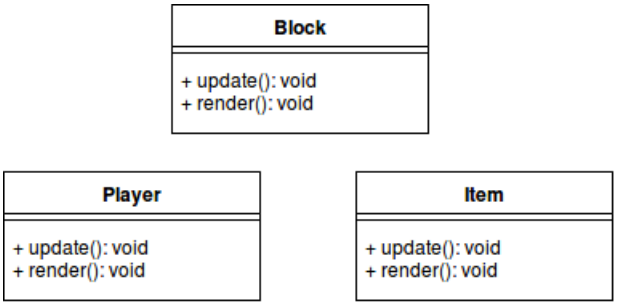


Figura 1 – Exemplo de arquitetura monolítica.

1.1.2.2 Arquitetura reutilizável

Esta arquitetura tem como objetivo principal desacoplar mecânicas gerais (como as já citadas acima) de objetos e funcionalidades específicas do jogo.

Segundo Rollings e Morris (2004), uma arquitetura bem implementada facilita a flexibilidade e reutilização de código, e mecânicas que tendem a mudar bastante durante o desenvolvimento do jogo estão atrás de uma interface consistente.

Rollings e Morris (2004) definem esse modelo como "Arquitetura Sólida" (do inglês, *Hard Architecture*). Uma arquitetura sólida pode ser definida como um framework relativamente genérico que não necessariamente depende do tipo de jogo que é produzido utilizando-o, e é conciso e confiável em relação a suas interfaces, trazendo robustez à sua implementação. A figura 2 apresenta um exemplo desta arquitetura.

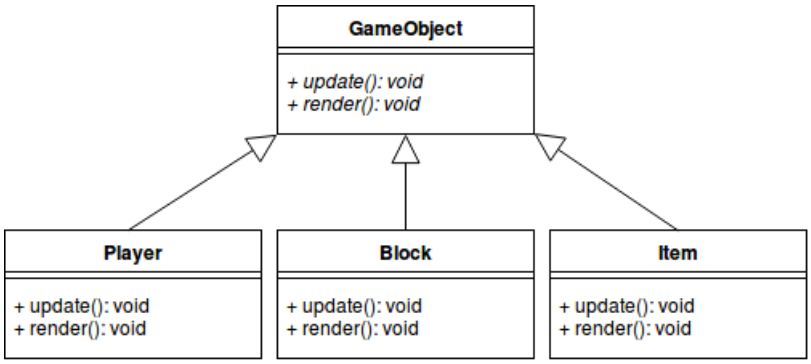


Figura 2 – Exemplo de arquitetura reutilizável.

No modelo de arquitetura sólido, as mecânicas gerais do jogo estão definidas à parte e são providas interfaces para a utilização destas mecânicas. A partir dessa interface, as funcionalidades específicas do jogo são construídas. Esse conjunto de funcionalidades específicas do jogo é chamado de "Arquitetura Flexível" (do inglês, *Soft Architecture*). Rollings e Morris (2004) definem arquitetura flexível como "uma arquitetura de um domínio específico, geralmente não reutilizada entre projetos diferentes, construída em cima da arquitetura sólida e fazendo o uso de seus serviços providos".

1.2 *Game Boy Advance*

O *Game Boy Advance* é um vídeo game portátil lançado em 2001 pela Nintendo, possuindo tela com resolução de 240 × 160 *pixels* e até 32768 cores possíveis, sistema de som estéreo PCM e processador ARM de 32 bits com memória incorporada (NINTENDO, 2018).

O detalhamento de cada tipo de memória, que seguirá até o fim desta seção, tem como referência (HAPP, 2002).

1.2.1 Memória

O *Game Boy Advance* possui várias semelhanças com um PC como por exemplo processador, memória *RAM*, ferramentas para entrada de dados e placa mãe (HARBOUR, 2003). Ao desenvolver jogos para ambas as plataformas, porém, há uma nítida diferença em se tratando do controle do hardware por parte do programador. Ao programar em um PC, o sistema operacional fornece uma série de funções que facilitam o acesso ao hardware, enquanto no GBA, o acesso ao hardware é feito acessando diretamente uma determinada posição de memória (registrador).

Korth (2007), no *GBATek*, classifica a memória utilizável como memória interna geral, memória interna do display e memória externa. A memória interna geral é dividida em *System ROM* (BIOS), *On-Board Work RAM*, *On-chip Work RAM* e *I/O Registers*.

Já a memória interna do display divide-se em *Pallete RAM*, *Video RAM* (VRAM) e *OBJ Attributes Memory* (OAM). Por fim, a memória externa é formada por 4 *Game PAK ROM's* e uma *Game PAK SRAM*. Por sua vez, o *CowBite Virtual Especifications* trata essa última região de memória de forma mais geral, como *Cart RAM*, e explica que ela pode ser utilizada como SRAM, *Flash ROM* ou EEPROM.

A *System ROM* possui 16 *KBytes* de tamanho e contém a BIOS do sistema. Essa região de memória pode ser usada somente para escrita e qualquer tentativa de leitura resultará em falha.

A *On-Board Work RAM*, citada pelo *GBATek*, é tratada como *External Work RAM* (EWRAM) pelo *CowBite Virtual Specifications*. Ela possui 256 *KBytes* de tamanho e é utilizada para inserir código e dados do jogo. Se um cabo *multiboot* estiver presente quando o console for iniciado, a BIOS irá detectá-lo e automaticamente deverá transferir o código binário para essa região.

A *On-Chip Work RAM* é tratada pelo o *Cowbite Virtual Specifications* como *Internal Work RAM* (IWRAM) e possui 32 *KBytes* de espaço. Dentre as RAM's do GBA, essa é a mais rápida. Levando em consideração que seu barramento possui 32 *bits* de tamanho, enquanto o da *System ROM* e da EWRAM possuem apenas 16 *bits*, é recomendado que o código ARM[1] de 32 *bits* seja utilizado aqui, deixando o código *THUMB*[2] para ser utilizado na *System ROM* e EWRAM

A *I/O RAM*, citada anteriormente como *I/O Registers*, possui 1 *KByte* de extensão e é utilizada para acesso direto à memória, controle dos gráficos, do áudio e de outras funções do GBA.

A *Pallete RAM* possui 1 *KByte* de tamanho e tem como função armazenar as cores de 16 *bits* necessárias quando se deseja utilizar paletas de cores. Ela possui duas áreas: uma para *backgrounds* e outra para *sprites*. Cada uma dessas áreas pode ser utilizada como uma única paleta de cores ou como 16 paletas de 16 cores cada.

A *Video RAM* (VRAM) possui 96 *KBytes* de espaço e é onde devem ser armazenados os dados gráficos do jogo para que possam ser mostrados na tela do GBA. A *OBJ Attributes Memory* (OAM) possui 1 *KByte* de tamanho e é utilizada para controlar as *sprites* do GBA. Por fim, a *Cart RAM* pode ser utilizada como *SRAM*, *Flash ROM* ou *EEPROM*. Essa região é utilizada principalmente para salvar os dados do jogo.

[1] *A32 instructions, known as Arm instructions in pre-Armv8 architectures, are 32 bits wide, and are aligned on 4-byte boundaries. A32 instructions are supported by both A-profile and R-profile architectures.* (ARM, 2018)

[2] *The T32 instruction set, known as Thumb in pre-Armv8 architectures, is a mixed 32- and 16-bit length instruction set that offers the designer excellent code density for minimal system memory size and cost.* (ARM, 2018)

1.2.2 Renderização de Vídeo

Segundo Vijn, o GBA possui 3 tipos de representação de gráficos:

> *All things considered, the GBA knows 3 types of graphics representations: bitmaps, tiled backgrounds and sprites. The bitmap and tiled background (also simply known as background) types affect how the whole screen is built up and as such cannot both be activated at the same time.* (VIJN, 2013, p. 38)

O GBA possui 3 modos de vídeo baseados em *bitmaps* e a principal diferença entre eles está no fato de utilizarem ou não paletas de cores, no número de *bits* utilizados para representar as cores e na resolução (HARBOUR, 2003). Nesses modos, a memória de vídeo funciona como se fosse um grande *bitmap*, de tal forma que cada pixel da tela é representado por uma posição na memória (VIJN, 2013). A figura 3 apresenta um exemplo de *bitmap*.

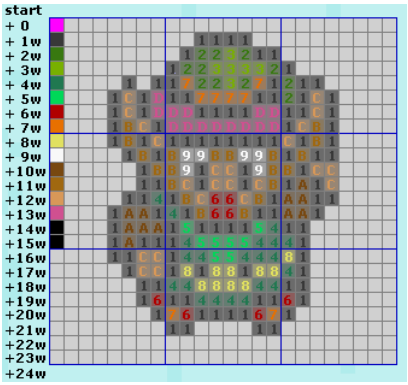


Figura 3 – Exemplo de *bitmap* 24×24. Fonte: (VIJN, 2013).

O GBA também possui 3 modos de vídeo baseados em *tiles* e a principal diferença entre eles está na quantidade de *backgrounds* que podem ser utilizados em cada um dos três modos e nas operações (rotação/*zoom*) que podem ser aplicadas ou não em cada um deles (HARBOUR, 2003). Nesses modos, são construídos *tiles* de 8×8 *bits* para que posteriormente possam ser utilizados em *tilemaps*, que por sua vez serão utilizados para renderizar os objetos necessários (VIJN, 2013). A figura 4 apresenta um exemplo de *sprite* dividida em *tiles*.

Há ainda a camada reservada para as *sprites*: "*Sprites are small (*8×8 *to* 64×64 *pixels) graphical objects that can be transformed independently from each other and can be used in conjunction with either bitmap or background types.*" (VIJN, 2013, p. 38)

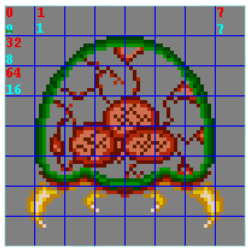


Figura 4 – Exemplo de *sprite* dividida em tiles. Fonte: (VIJN, 2013).

1.2.3 Tratamento de Input

O GBA possui 4 teclas direcionais e 6 botões, cujos estados podem ser acessados por meio dos 10 primeiros *bits* do registrador localizado no endereço `0x4000130` (KORTH, 2007). Há ainda um outro registrador, no endereço `0x4000132`, que permite escolher quais pressionamentos de teclas geram interrupções (HAPP, 2002). A figura 5 apresenta teclas e botões do GBA.

Figura 5 – Imagem da frente do GBA mostrando teclas e botões. Fonte: (NINTENDO, 2018).

O detalhamento de cada canal de áudio, que seguirá até o fim desta seção, tem como referência (KORTH, 2007).

1.2.4 Tratamento de Áudio

O GBA fornece 4 canais de áudio utilizados para reproduzir tons e ruídos (KORTH, 2007). Apesar dos 4 canais de áudio, o console não possui um *mixer* embutido, o que faz com que os programadores precisem escrever seus próprios *mixers* ou utilizar bibliotecas de terceiros para tal propósito. Sem um *mixer*, apenas um áudio pode ser tocado por vez, o que é chamado de reprodução assíncrona (HARBOUR, 2003).

O primeiro canal de áudio do GBA é responsável pelo tom e pelo *sweep*, ele possui um registrador para controle do *sweep*, um para controle da frequência, que também permite reiniciar o áudio que está sendo tocado e um registrador para controlar o volume do áudio, o padrão de onda, o *envelope Step-Time* e o *envelope Direction*.

O segundo canal de áudio funciona de forma similar ao primeiro e também é responsável pelo controle do tom. Ele não possui, porém, um registrador para controle do *sweep* ou do *tone envelope*.

O terceiro canal de áudio é responsável pela saída de onda e pode ser utilizado para reproduzir áudio digital. Ele também pode ser utilizado para reproduzir tons normais a depender da configuração dos registradores. O canal em questão possui um registrador para controle da RAM de onda, um para controle do comprimento e volume do áudio e um para controle da frequência, permitindo também reiniciar o áudio que está sendo tocado.

Por fim, o quarto canal é responsável pelo ruído. Ele pode ser utilizado para reproduzir ruído branco[3], o que pode ser feito alternando randomicamente a amplitude entre alta e baixa em uma dada frequência. Também é possível modificar a função do gerador randômico de tal forma que a saída de áudio se torne mais regular, o que fornece uma capacidade limitada de reproduzir tons ao invés de ruído. Esse canal possui um registrador para controlar o volume do áudio, o *Envelope Step-Time* e o *Envelope Direction* e um registrador para controle da frequência, permitindo também reiniciar o áudio.

1.3 Porte de jogos

Na engenharia de software, porte é definido como "mover um sistema entre ambientes ou plataformas" (FRAKES; FOX, 1995). No contexto de jogos eletrônicos a definição é um pouco mais específica e, segundo Carreker (2012), é o processo de converter código e outros recursos desenvolvidos para uma plataforma específica para outra plataforma, que, nesse caso, representa algum tipo de console ou computador.

Segundo Horna e Wawro (2014), o processo de porte de um jogo para uma plataforma específica consiste, geralmente, em três passos:

1. primeiramente deve-se conseguir executar o jogo na plataforma de destino (pelo menos compilar, com chamadas *stub*[4] para a biblioteca de gráficos). Este processo tende a ser o mais difícil, pois há diversos problemas com bibliotecas específicas da plataforma de destino;

2. adicionar o suporte à biblioteca de gráficos da plataforma de destino, criando uma interface comum para todas as plataformas (de modo a manter o código mais ma-

[3] *A musician thinks of white noise as a sound with equal intensity at all frequencies within a broad band. Some examples are the sound of thunder, the roar of a jet engine, and the noise at the stock exchange.* (KUO, 1996)

[4] *Stub is a dummy function that assists in testing part of a program. A stub has the same name and interface as a function that actually would be called by the part of the program being tested, but it is usually much simpler.* (DALE; WEEMS, 2004)

nutenível);

3. após adicionar o suporte à bliblioteca de gráficos, é necessário otimizar a performance do jogo, principalmente otimizações que levam em conta a performance da CPU, que podem afetar a execução do jogo de maneiras difíceis de antecipar.

Realizar o porte de jogos para diferentes plataformas pode ser um trabalho desafiador. Diversas dificuldades podem ser encontradas durante o processo de porte, geralmente relacionadas ao retrabalho de se reescrever o código do jogo e adaptá-lo para a plataforma de destino. Tomando como exemplo o porte do jogo *Fez* para a plataforma *PlayStation*, Horna diz que os principais desafios ao realizar o porte desse jogo foram a falta de suporte da linguagem no qual o jogo foi escrito e a performance dos gráficos após o porte:

> *The most difficult challenge we had was that the original Fez game was written in C#, and there was no C# support for PlayStation platforms when we started the port.*
>
> *If we continued using C#, we would be hitting CPU performance problems, as even the original game on the Xbox 360 experienced some slowdowns. On the other hand, converting the game code (and Monogame itself) to C++ was a very long and tedious task, but it would offer us some unique optimization opportunities. As we wanted to achieve the best possible port, we finally opted for the C++ way.*
>
> *Another big problem was graphics performance. Although it might look like a simple 2D game, Fez worlds have a very complex 3D structure (pictured) – there could even be pixel-sized polygons. We were using the PC version as a base for the port, and graphics performance was not a big problem on that platform as current video cards could already handle that workload quite easily, but the shaders and geometry caused us some performance trouble when running on previous-gen or portable consoles. So we had to rewrite some parts of the drawing code and optimize a few shaders to particularly fit each console.* (HORNA; WAWRO, 2014)

2 Metodologia

A metodologia deste trabalho está dividida em duas seções: a seção 2.1 apresenta as ferramentas que serão utilizadas para realizar a compilação e manipulação dos recursos do jogo (como áudio e imagens) e a seção 2.2 aborda como se darão o desenvolvimento da *engine* e do jogo.

2.1 Ferramentas de desenvolvimento

Para a realização do porte do jogo para *Game Boy Advance*, são necessários dois ambientes principais: um ambiente de desenvolvimento onde seja possível implementar o jogo e exportar o binário executável para o console e um ambiente para testar o executável gerado, sendo esse físico ou emulado.

2.1.1 Ambiente de desenvolvimento

O jogo foi reescrito utilizando a linguagem C++, na versão 11, pois provê uma série de recursos e estruturas não presentes na linguagem C que facilitarão o desenvolvimento do jogo.

O ambiente de desenvolvimento utilizado para a implementação do jogo consiste do *kit* de desenvolvimento devkitARM, ferramentas para manipulação das imagens e áudio do jogo.

2.1.1.1 *devkitPro* e *devkitARM*

O `devkitPro`[1] é uma organização que provê conjuntos de ferramentas para desenvolvimento de jogos em diversos consoles da *Nintendo*, como *Nintendo GBA*, *Nintendo Wii*, *Nintendo Switch*, dentre outros.

Dentre esses conjuntos de ferramentas encontra-se o *devkitARM*, *toolchain* que contém o ambiente de desenvolvimento necessário para realizar a compilação do código escrito em C/C++ para a arquitetura de processadores ARM existente no GBA, citado na Seção 1.2 do Capítulo 1.

2.1.1.2 Manipulação de imagens e áudio

Para o ajuste das imagens do jogo para a resolução de tela do GBA, foi utilizada a ferramenta de manipulação de imagens GIMP[2], versão 2.8.

[1] *devkitPro*, disponível em <https://devkitpro.org/>
[2] GNU Image Manipulation Program, disponível em <https://www.gimp.org/>

Para realizar a conversão das imagens para um formato legível no GBA, foi utilizada uma biblioteca C chamada GRIT[3], versão `0.8.15`.

Para a manipulação do áudio do jogo foram utlizadas três ferramentas: `avconv`[4] `13_dev0-1601-g56f5018` e `librosa`[5] `0.6.2` para *downsampling* das músicas; e `modplug tracker`[6] `1.27.11.00` para realizar a conversão das músicas para `.mod`.

2.1.2 Ambiente de teste

Para a realização de testes com os executáveis gerados pelo *devkitARM* foi utilizado o emulador de *Game Boy Advance VisualBoyAdvance-M*[7], versão `2.0.1`.

O *console* utilizado como ambiente de testes real é um *Nintendo DS*[8], que possui um *slot* para cartuchos de *Game Boy Advance*. Neste trabalho foi utilizado um cartucho especial onde é possível escrever arquivos executáveis diretamente nele.

Para a escrita dos arquivos executáveis neste cartucho foi utilizado o dispositivo *EZFlash II*[9]. Como essa é uma versão antiga do produto, é necessário instalar um cliente para *upload* dos arquivos para o cartucho. Este cliente só possui compatibilidade com *Windows XP*[10], fazendo com que seja necessário instalar uma máquina virtual com o sistema operacional.

2.2 Metodologia de desenvolvimento

2.2.1 Desenvolvimento da *Engine*

Para contribuir com uma arquitetura mais manutenível, foi optado por desacoplar a *engine* do jogo em si. A *engine* ficará responsável por implementar os módulos genéricos do jogo, enquanto que o jogo em si conterá as funcionalidades mais específicas.

A *engine* contém uma classe que representa um objeto do jogo (do inglês, *game object*). Esta classe foi responsável por conter o comportamento genérico de um objeto dentro do jogo (podendo este ser um personagem, uma plataforma, um item coletável, etc.). Ele foi representado de acordo com a figura 6:

[3] *GBA Raster Image Transmogrifier*, disponível em <http://www.coranac.com/projects/grit/>
[4] `avconv`, disponível em <https://github.com/libav/libav>
[5] `librosa`, disponível em <https://github.com/librosa/librosa>
[6] *Open Modplug Tracker*, disponível em <http://openmpt.org/features>
[7] *VisualBoyAdvance-M*, disponível em <https://github.com/visualboyadvance-m/visualboyadvance-m>
[8] *Nintendo DS*, disponível em <https://www.nintendo.com/consumer/systems/selectds.jsp>
[9] *EZ Flash II*, disponível em <http://www.ezfadvance.com/cards/EZ-Flash_2.htm>
[10] *Windows XP*, disponível em <https://support.microsoft.com/pt-br/help/14223/windows-xp-end-of-support>

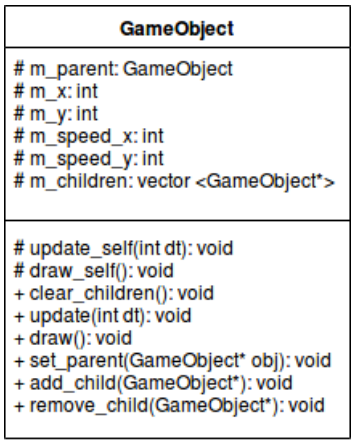


Figura 6 – Modelagem da classe *GameObject*.

Onde os métodos `update_self()` e `draw_self()`, abstratos, tratam de atualizar o objeto a cada frame e renderizar o objeto na posição `(x, y)`, respectivamente.

É importante frisar que os métodos `update_self()` e `draw_self()` são abstratos e, considerando que o jogo foi portado utilizando C++, estes métodos devem ser puramente virtuais, pois isso garante que qualquer classe que venha a extender de `GameObject` seja obrigada a implementer suas próprias rotinas específicas de atualização e renderização.

Além da classe `GameObject`, os principais componentes da *engine* implementados foram: módulo de vídeo, módulo de áudio, módulo de física e módulo de *input*.

2.2.1.1 Módulo de vídeo

O módulo de vídeo foi responsável por renderizar qualquer tipo de imagem e animação existente. Ele contém as seguintes funcionalidades:

- renderizar de um a até quatro imagens de fundo;
- renderizar uma ou mais *sprites*;
- renderizar uma animação (como uma série de *sprites*);
- movimentar horizontalmente qualquer imagem de fundo (*horizontal scroll*);
- remover uma imagem, *sprite* ou animação que esteja renderizado da tela;
- atualizar a renderização a cada frame, levando em consideração as posições x e y da *sprite* ou animação; e
- configurar a prioridade de exibição de texturas (animações ou *sprites*)

Deve-se lembrar que as imagens renderizadas foram ajustadas para a resolução e formato de cores corretos do GBA.

2.2.1.2 Módulo de áudio

O módulo de áudio foi responsável por executar, no momento correto, qualquer música de fundo e efeito sonoro do jogo. Ele contém as seguintes funcionalidades:

- iniciar a execução de uma música de fundo; e
- parar a execução de uma música de fundo.

2.2.1.3 Módulo de física

O módulo de física teve como principal responsabilidade a detecção de colisões entre objetos do jogo. Ele contém as seguintes funcionalidades:

- simular, opcionalmente, a ação da gravidade em qualquer objeto do jogo;
- detectar, opcionalmente, colisões entre todos os objetos do jogo; e
- detectar, opcionalmente, colisões entre um objeto e todos os outros objetos do jogo.

2.2.1.4 Módulo de *input*

O módulo de *input* foi responsável por receber qualquer pressionamento de qualquer um dos 10 botões e teclas do GBA. Ele tem como funcionalidades:

- detectar se determinado botão foi pressionado; e
- detecter se determinado botão está sendo pressionado.

2.2.2 Desenvolvimento do jogo

A implementação do jogo foi baseada em classes e foi modelado como mostrado na figura 7. O personagem principal, os itens coletáveis e plataformas foram representados como *game objects* e também herdam da classe `Collidable`, para que possam colidir entre si. Já os níveis do jogo são representados pela classe `TWLevel` e esta, por sua vez, herda da classe `Level`, que contém a generalização de um nível no jogo. Por fim, a classe `TWGame` tem como responsabilidade iniciar e realizar a transição entre os níveis do jogo.

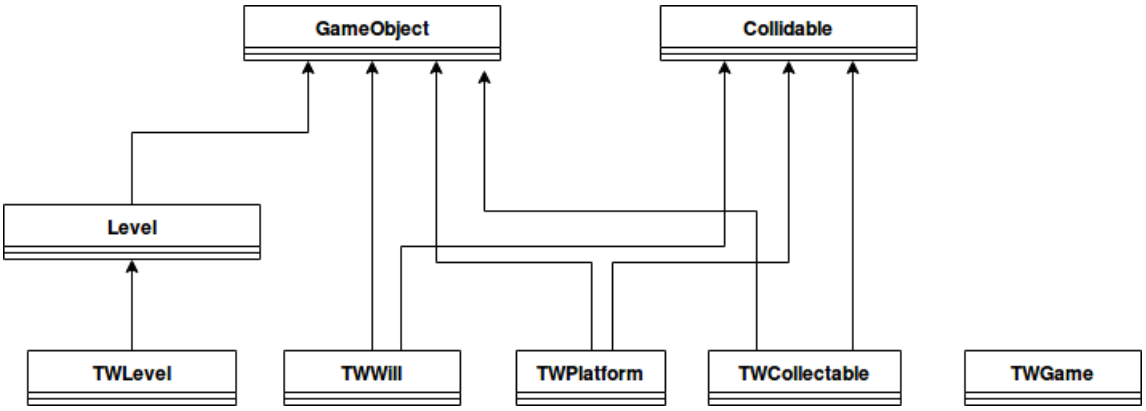


Figura 7 – Modelagem dos objetos do jogo.

Os recursos como imagens e músicas precisaram ser editados para que pudessem ser carregados em memória e utilizados no GBA. No caso das imagens, por exemplo, foram modificadas características como dimensões e quantidade de cores a fim de diminuir seu tamanho para que possam ser convertidas para um formato utilizável no GBA. Já no caso dos arquivos de áudio, a principal mudança realizada foi a diminuição da frequência das músicas, como será explicado na Seção 3.3.4.

3 Resultados

Este capítulo irá apresentar os resultados obtidos durante a realização deste trabalho. Serão mostrados os prótotipos iniciais do jogo, a *engine* desenvolvida para a construção do jogo e, por fim, as telas do jogo final desenvolvido.

3.1 Protótipo inicial

A fim de atestar a viabilidade do porte do jogo *Traveling Will*, desenvolvido inicialmente para PC, para a plataforma *Nintendo Game Boy Advance*, foi feita uma versão funcional do menu original do jogo, já testada em um *Nintendo DS* (como explicado na Seção **??** do Capítulo 2). Para isso, a principal ferramenta utilizada foi a *libtonc*[1], que nessa versão inicial fez o papel de *engine* do jogo.

Na figura 8 é possível comparar o menu principal do jogo original com o protótipo implementado sendo executado em um emulador de *Game Boy Advance*:

Este protótipo inicial contém apenas a tela de menu, com um *background* desenhado ao fundo e quatro botões clicáveis, porém sem nenhuma ação após o clique. Para alterar o botão selecionado, basta utilizar as setas do teclado.

O protótipo desenvolvido está disponível no seguinte repositório: <https://github.com/traveling-will-gba/traveling-will-prototype>

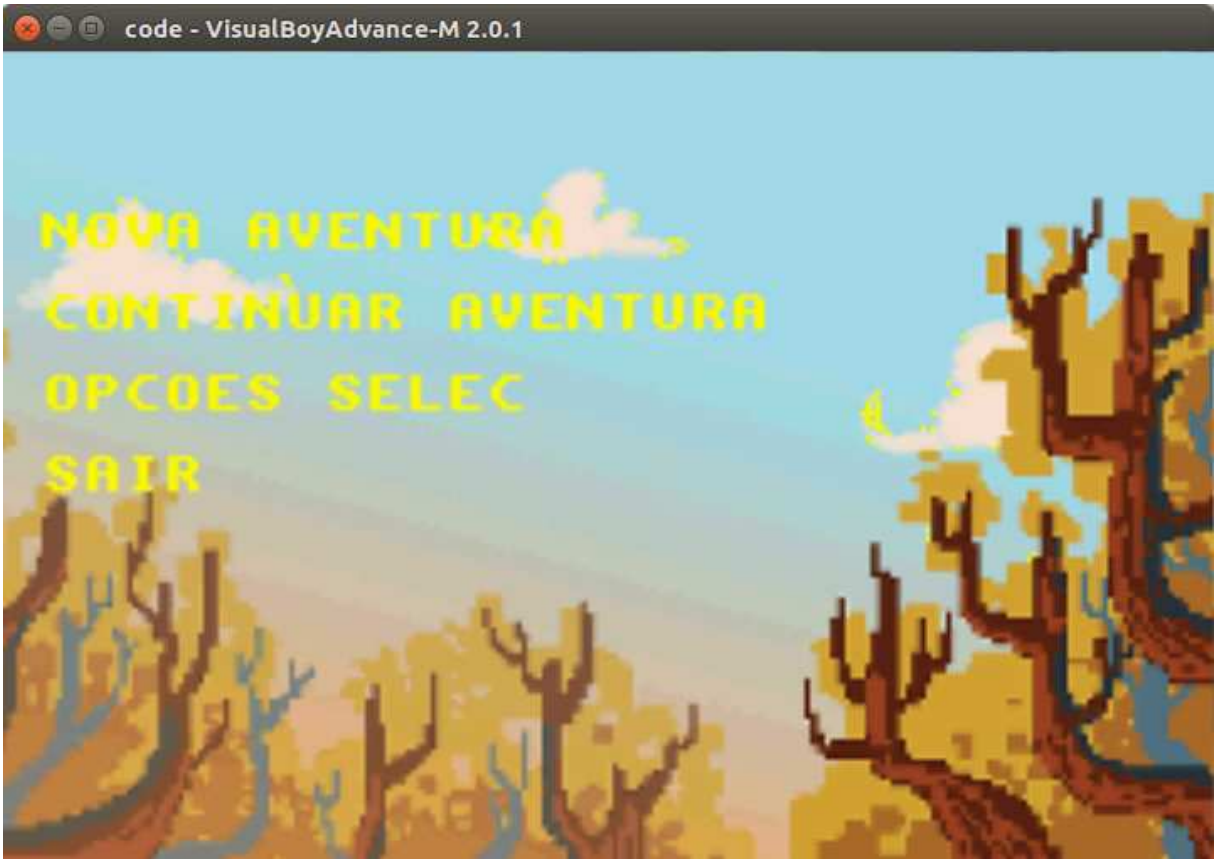
3.2 Desenvolvimento da *engine*

Logo após a finalização do protótipo, foi iniciado o desenvolvimento da *engine* responsável por substituir a *libtonc* e gerenciar os recursos do jogo. Esta *engine* contém os seguintes módulos: *input*, vídeo, gerenciador de memória, áudio e física. Além disso, a *engine* contém abstrações para os níveis (*levels*) e objetos do jogo (*game objects*) e um módulo utilitário para funções genéricas relacionadas ao *hardware* do GBA.

Os módulos de vídeo, *input*, física e as abstrações para níveis e objetos do jogo foram desenvolvidos tendo como base a *ijengine*[2], *engine* desenvolvida para a disciplina de Introdução aos Jogos Eletrônicos, que tem como foco a criação de jogos para PC utilizando *C++*. Ela contém módulos de vídeo, áudio, manipulação de eventos, física, *input*, dentre outros, e contém uma interface para a utilização de diferentes bibliotecas gráficas, como

[1] *libtonc*, disponível em <http://www.coranac.com/files/tonc-code.zip>
[2] *ijengine*, disponível em <https://github.com/fgagamedev/ijengine>

(a) Jogo original sendo executado em um PC.



(b) Protótipo sendo executado no emulador de GBA.

Figura 8 – Comparação entre o jogo original e o protótipo no emulador.

Código 3.1 – Cabeçalho do módulo de *input*.

```
1  #ifndef INPUT_H
2  #define INPUT_H
3
4  #include <stdbool.h>
5  #include "base_types.h"
6
7  #define BUTTON_A 1
8  #define BUTTON_B 2
9  #define BUTTON_SELECT 4
10 #define BUTTON_START 8
11 #define BUTTON_RIGHT 16
12 #define BUTTON_LEFT 32
13 #define BUTTON_UP 64
14 #define BUTTON_DOWN 128
15 #define BUTTON_R 256
16 #define BUTTON_L 512
17 #define N_BUTTON 10
18
19 int pressed_state[N_BUTTON];
20
21 void check_buttons_states();
22 bool pressed(int button);
23
24 #endif
```

SDL[3] e OpenGL[4].

3.2.1 Módulo de *input*

Os estados dos botões do GBA ficam salvos em um registrador. Cada um desses estados é representado por um *bit* do valor guardado por esse registrador. Sempre que um botão é pressionado, o GBA automaticamente troca o valor guardado nesse registrador de tal forma que o *bit* que representa o botão em questão passe a possuir valor 0. De forma similar, quando o botão é solto, o valor contido no *bit* em questão é modificado para 1, seu valor padrão. Sendo assim, a checagem dos estados pode ser realizada facilmente utilizando *bitmasks*.

Por exemplo, caso se deseje checar um botão representado pelo *bit* 2 (com a contagem começando em 0), basta pegar o resultado do *AND* binário entre o valor guardado no registrador e a potência de 2 que possui como expoente o *bit* em questão (4, nesse exemplo). No código 3.1 é possível visualizar a definição das constantes que representam os botões, assim como a função utilizada para checar o estado de cada um deles.

A figura 9 apresenta o teste implementado para checar o pressionamento dos botões do GBA. Para cada botão pressionado um *pixel* vermelho aparece na tela. Nesta figura,

[3] *Simple DirectMedia Layer*, disponível em <https://www.libsdl.org>
[4] OpenGL, disponível em <https://www.opengl.org>

Código 3.2 – Código-fonte do módulo de *input*.

```
1  #include "input.h"
2
3  volatile unsigned int *buttons_mem = (volatile unsigned int *) 0x04000130;
4
5  void check_buttons_states() {
6      for(int i = 0; i < N_BUTTON; i++) {
7          pressed_state[i] = !((*buttons_mem) & (1 << i));
8      }
9  }
10
11 bool pressed(int button) {
12     return pressed_state[button];
13 }
```

os botões B, R, *LEFT*, *DOWN* e *START* estão sendo pressionados simultaneamente. O código 3.3 apresenta este teste.

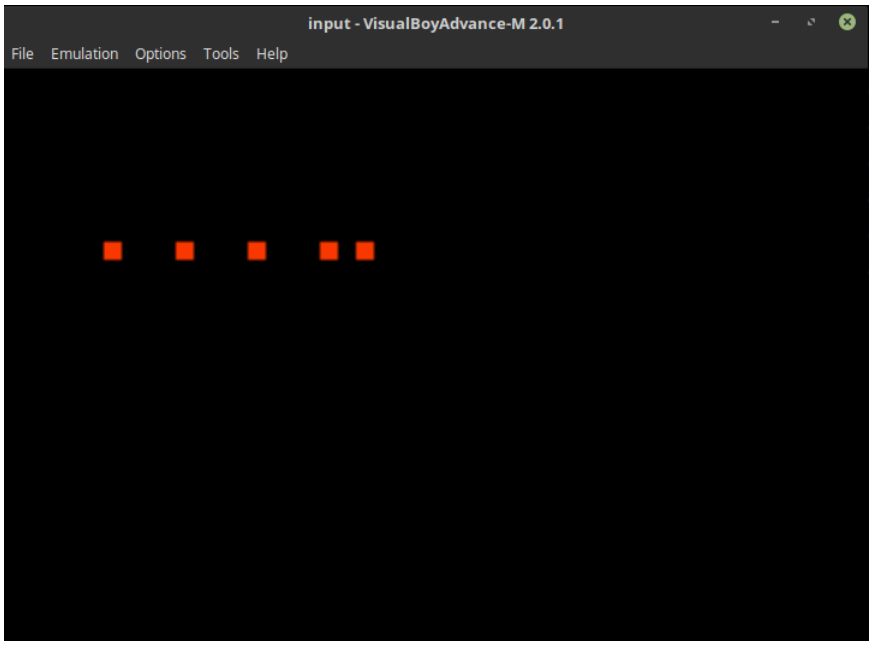


Figura 9 – Teste de pressionamento de botões no emulador.

O código deste projeto se encontra no seguinte endereço: <https://github.com/traveling-will-gba/gbengine/tree/dev/test/input>.

3.2.2 Módulo de vídeo

O módulo de vídeo é responsável pelo controle do modo de vídeo, dos *backgrounds* e da renderização das *sprites*.

Para gerenciar a renderização das *sprites* foi desenvolvida uma classe chamada *Texture*. Ela foi planejada de forma a não apenas copiar os dados da imagem para a

Código 3.3 – Código de teste de *input*.

```
#include "video.h"
#include "input.h"

#define RED 0x0000FF

unsigned short *vid_mem = (unsigned short *)0x6000000;

int main() {
    reset_dispcnt();
    set_video_mode(3);
    set_background_number(2);

    while(1) {
        check_buttons_states();

        for(int i=0;i<=9;i++){
            if (pressed(i)) {
                vid_mem[50 * 240 + i * 10] = RED;
            } else {
                vid_mem[50 * 240 + i * 10] = 0;
            }
        }
    }

    return 0;
}
```

região de memória apropriada, mas também permitir a animação das *sprites* de uma textura e controlar os metadados das imagens renderizadas no jogo.

A seguir, será explicado, com o auxílo de trechos do código, o funcionamento dos principais elementos dessa classe. No código 3.4 é possível visualizar o primeiro construtor. Ele recebe os ponteiros para a paleta de cores e para o vetor de *tiles* utilizados pela imagem, os tamanhos das regiões de memória alocadas pra cada um desses ponteiros e a quantidade de *bits* por *pixel* que a imagem utiliza. Cada um desses atributos é guardado na própria classe, e, já nesse construtor, são chamados os métodos `set_sprite_pal` e `set_sprite`, responsáveis por copiá-los para as regiões apropriadas. Por fim, ainda nesse construtor, são atribuídos os índices do *tile base* e da paleta de cores utilizada pela imagem. O cálculo desses índices será explicado logo adiante, nos tópicos dedicados aos métodos `set_sprite_pal` e `set_sprite`.

O segundo construtor funciona de forma similar ao anterior, com a diferença de que em vez de receber todos os atributos da imagem, ele recebe apenas um ponteiro para outra textura. Esse construtor serve para quando se deseja renderizar réplicas de uma mesma textura. Utilizar ele permite que tais texturas compartilhem a paleta de cores e o vetor de *tiles*, fazendo-se necessário alocar espaço apenas para os metadados, que são diferentes pra cada textura. Veja o código 3.5

Código 3.4 – Construtor da classe `Texture`.

```
1  Texture::Texture(uint32_t num_sprites, const uint16_t *pallete, uint32_t
       pallete_len,
2      const unsigned int *tiles, uint32_t tiles_len, enum bits_per_pixel bpp
           = _8BPP)
3  {
4      this->pallete = pallete;
5      this->pallete_len = pallete_len;
6      this->pallete_id = 0;
7      this->bpp = bpp;
8      this->num_sprites = num_sprites;
9      this->num_tiles = tiles_len / ((bpp == _4BPP) ? 32 : 64);
10     this->tiles_per_sprite = num_tiles / num_sprites;
11     this->tiles = tiles;
12     this->tiles_len = tiles_len;
13
14     memory_manager = MemoryManager::get_memory_manager();
15
16     set_sprite_pal();
17     set_sprite();
18     oam_entry = memory_manager->alloc_oam_entry();
19
20     metadata.tid = tile_base * ((bpp == _4BPP) ? 1 : 2);
21     metadata.pb = pallete_id;
22 }
```

Código 3.5 – Construtor por cópia da classe `Texture`.

```
1  Texture::Texture(const Texture *texture)
2  {
3      this->num_sprites = texture->num_sprites;
4      this->pallete = texture->pallete;
5      this->pallete_len = texture->pallete_len;
6      this->tiles = texture->tiles;
7      this->tiles_len = texture->tiles_len;
8      this->bpp = texture->bpp;
9      this->pallete_id = texture->pallete_id;
10     this->num_tiles = texture->num_tiles;
11     this->tiles_per_sprite = texture->tiles_per_sprite;
12     this->tile_base = texture->tile_base;
13
14     memory_manager = MemoryManager::get_memory_manager();
15
16     oam_entry = memory_manager->alloc_oam_entry();
17
18     metadata.tid = tile_base * ((bpp == _4BPP) ? 1 : 2);
19     metadata.pb = pallete_id;
20 }
```

Código 3.6 – Função de alocação da paleta de cores de uma textura.

```
1 bool Texture::set_sprite_pal() {
2     volatile uint8_t *teste = memory_manager->alloc_texture_palette(32);
3     mem16cpy(teste, pallete, 32);
4
5     this->pallete_id = (teste - (volatile uint8_t *)0x05000200) / 32;
6
7     return true;
8 }
```

Código 3.7 – Função de alocação das *sprites* de uma textura.

```
1 bool Texture::set_sprite() {
2     volatile struct tile *teste = memory_manager->alloc_texture(tiles_len);
3
4     mem16cpy((volatile struct tile *)teste, tiles, tiles_len);
5     tile_base = teste - memory_manager->base_texture_mem();
6
7     return true;
8 }
```

Código 3.8 – Função de cópia dos metadados das texturas.

```
1 void Texture::update_metadata() {
2     mem16cpy(oam_entry, &metadata, sizeof(struct attr));
3 }
```

O método `set_sprite_pal` é responsável por chamar o método `alloc_texture_pal` da classe `MemoryManager` passando como parâmetro o tamanho da paleta de cores a ser alocada. Como resultado, ele irá receber o endereço de memória aonde a paleta deverá ser guardada. Após fazer a cópia da paleta, é preciso calcular o índice da paleta escolhida na memória, já que esse é um dos metadados necessários para renderização da imagem a 4 *bits* por *pixel*. Como a região reservada para as paletas de cores no *GBA* é dividida em regiões de 32 *bytes*, para realizar tal cálculo basta pegar a diferença entre o início da região escolhida para a cópia e o início da região reservada para as paletas e dividir por 32. Como uma imagem renderizada a 8 *bits* por *pixel* ocupa toda a região reservada para as paletas, o cálculo do índice não é necessário para uma imagem que utilize 256 cores.

O método `set_sprite` funciona de forma similar ao `set_sprite_pal`, tendo como diferença relevante apenas o cálculo do índice do *tile_base* da imagem, que é feito subtraindo o início da região escolhida para a cópia e o início da região reservada para os *tiles*. Essa diferença ocorre porque diferentemente da paleta de cores, cada unidade do vetor que representa a `tile_mem` no nosso código é, de fato, um *tile*.

O método `update_metadata` apenas copia os metadados para a *OAM* (*Object Attributes Memory*).

Por fim, o método `update` calcula qual a próxima sprite a ser renderizada e atribui o primeiro *tile* dela como `tile_id`. Esse processo é o que permite a animação das *sprites* do jogo. O cálculo é feito somando o `tile_id` atual com a quantidade de *tiles* por *sprite* da textura que está sendo renderizada. Vale ressaltar que os índices dos *tiles* são contabilizados sempre de 32 em 32 *bytes*, mesmo que a imagem utilize 8 *bits* por *pixel* e por isso o número de *tiles* por *sprite* é multiplicado por 2 quando o número de *bits* por *pixel* da textura é 8.

Para a renderização dos *backgrounds*, foi desenvolvida uma classe que recebe ponteiros para a paleta de cores, para o vetor de *tiles* e para o mapa de *tiles* utilizados pelo *background*, assim como os tamanhos das regiões alocadas pra cada um desses ponteiros. Assim que é instanciada, essa classe calcula qual o melhor *charblock* e o melhor *screenblock* para guardar os *tiles* e o mapa de *tiles*, respectivamente. Vale ressaltar que os *charblocks* e *screenblocks* compartilham a mesma região de memória e precisam de um espaço contíguo na memória do *GBA* para que o *background* seja renderizado corretamente. Por esse motivo não é recomendado apenas copiá-los para a memória do *GBA* de forma sequencial, já que isso poderia causar um *overlap* entre um *charblock* e um *screenblock*, e também poderia preencher a memória do *GBA* de forma não-ótima, o que pode fazer com que não caibam todos os *backgrounds* necessários para uma ou mais fases do jogo.

3.2.3 Módulo gerenciador de memória

Para melhor utilização dos recursos providos pelo GBA foi desenvolvido um módulo que trata do gerenciamento de memória dos objetos do jogo. A principal responsabilidade deste módulo é garantir que a alocação e desalocação de recursos seja feita de forma eficiente e segura.

3.2.3.1 Funcionamento do gerenciador de memória

O modelo de gerenciamento escolhido para ser utilizado neste projeto foi o modelo de gerenciamento com partições variáveis (TANENBAUM, 2010). Esse modelo se faz necessário pois os recursos a serem alocados durante a execução do jogo contém tamanhos distintos (uma sprite ocupa menos espaço que um *background* que, por sua vez, ocupa menos espaço do que uma música). Além disso, é importante citar que este gerenciador aloca os recursos contiguamente na memória, isto é, ele sempre irá optar pelo primeiro espaço disponível a partir do começo da memória.

Utilizando esse modelo como base, o gerenciador funciona da seguinte maneira:

- o construtor do gerenciador de memória é responsável por inicializar todos os ponteiros correspondentes aos endereços dos registradores de *backgrounds*, texturas e atributos das texturas;

Código 3.9 – Código de alocação de paletas para *backgrounds* e texturas.

```
1  volatile uint8_t *MemoryManager::alloc_palette(bitset<512>& used,
2      volatile uint8_t *palette, size_t size)
3  {
4      int used_size = used.size();
5
6      for (size_t i = 0; i < used_size; i++) {
7          // if this position is taken, skip it
8          if (used[i] == true) {
9              continue;
10         }
11
12         uint32_t available_pal_len = used_size - i;
13
14         // if there is no space to allocate this pallete, skip it
15         if (size > available_pal_len) {
16             continue;
17         }
18
19         // mark positions from i to size as used
20         for (size_t j = 0; j < size; j++) {
21             used[i + j] = true;
22         }
23
24         memory_map[palette + i] = size;
25
26         return palette + i;
27     }
28
29     return NULL;
30 }
```

- quando ocorre uma chamada de alocação, o gerenciador procura pelo primeiro espaço disponível na memória correspondente para alocar tal recurso. Caso não ache nenhum espaço disponível, retorna um endereço nulo;

- após achar um endereço disponível, é verificado se há espaço suficiente para alocar o recurso. Caso não haja espaço suficiente, um endereço nulo é retornado;

- após garantir que há um espaço disponível e com espaço suficiente para alocação, a posição deste endereço no vetor auxiliar (responsável por indicar se determinado endereço está disponível ou não) é ligada (indicando que o espaço foi ocupado) e é retornado o endereço desta posição para o cliente (no caso, o jogo) realizar a cópia do recurso.

O código 3.9 apresenta a função responsável por realizar a alocação de paletas (tanto para *backgrounds* quanto para texturas). Segundo Korth (2007), as paletas possuem 256 entradas de 2 *bytes*, totalizando 512 *bytes*.

3.2.3.2 Gerenciamento seguro de memória

Para garantir que novos recursos sejam alocados e que nenhum objeto já carregado na memória seja sobrescrito ou corrompido, qualquer chamada que envolva a alocação de novos recursos passa por um processo onde é verificado se a memória que será populada contém espaço suficiente realizar tal alocação.

Para realizar esta verificação, o gerenciador possui um mapa que associa um endereço na memória com o tamanho que o objeto alocado neste endereço ocupa. Deste modo, quando há uma chamada de alocação com tamanho `size` e uma posição `i` está disponível e o tamanho da partição em *bytes* é dado por `t`, as posições de `i` até

$$i + \frac{size}{t} - 1 \tag{3.1}$$

são marcadas como utilizadas.

3.2.3.3 Gerenciamento eficiente de memória

As estratégias utilizadas para garantir um gerenciamento eficiente de memória envolvem a utilização de *singleton* para a criação do objeto gerenciador de memória e utilização de estruturas de dados que permitam configurar o tamanho a ser utilizado.

Para que haja um único objeto encarregado de gerenciar a alocação de recursos no jogo, a classe `MemoryManager` foi modelada utilizado o padrão de projeto *Singleton*. Com este padrão de projeto, cada chamada de criação de uma nova instância do gerenciador de memória passa por uma verificação, que checa se já existe uma instância desta classe (por meio de uma variável dentro da própria classe que armazena uma instância de `MemoryManager` ou nulo). Caso haja, esta instância é retornada. Caso contrário, é criada uma nova instância. O código 3.10 exemplifica o *singleton* (trechos de código desta classe foram omitidos para melhor foco no padrão de projeto):

Código 3.10 – *Singleton* do gerenciador de memória.

```
1  class MemoryManager {
2      private:
3          MemoryManager *instance;
4      public:
5          MemoryManager *get_memory_manager() {
6              if (!instance) {
7                  instance = new MemoryManager();
8              }
9              return instance;
10         }
11 };
```

Código 3.11 – `Struct` com `bitfields` para atributos das texturas.

```
struct attr {
    // attr0
    uint8_t y;
    uint8_t om : 2;
    uint8_t gm : 2;
    uint8_t mos : 1;
    uint8_t cm : 1;
    uint8_t sh : 2;
    // attr1
    uint16_t x : 9;
    uint8_t aid : 5;
    uint8_t sz : 2;
    // attr2
    uint16_t tid : 10;
    uint8_t pr : 2;
    uint8_t pb : 4;
    // attr3
    uint16_t filler;
};
```

Código 3.12 – *Bitsets* para checagem de disponibilidade na memória.

```
// backgrounds and textures have 512 bytes each for palette memory
bitset<512> background_palette_used;
bitset<512> texture_palette_used;
// max number of charblocks that can be stored in VRAM
bitset<4> charblock_used;
// max number of screenblocks that can be stored in VRAM
bitset<32> screenblock_used;
```

A fim de não utilizar memória desnecessariamente, em alguns pontos do jogo foram utilizadas estruturas que permitem configurar a quantidade de *bits* que podem ser utilizados por cada variável. O primeiro exemplo mostra uma `struct` que armazena os atributos das texturas, onde cada atributo só precisa de um pequeno número de *bytes*. Já o segundo exemplo demonstra a utilização de `bitsets` (que mantém a disponibilidade de cada posição do endereço de memória) com tamanho fixo correspondente ao tamanho total da respectiva região de memória (neste exemplo, `Video RAM`).

3.2.4 Módulo de Física

O módulo de física é responsável por checar continuamente se os objetos estão colidindo e caso estejam chamar as funções responsáveis por lidar com a colisão para cada objeto. Para realizar tal procedimento esse módulo guarda uma lista de objetos que devem ser considerados na checagem e um ponteiro para o objeto alvo. Dessa forma, sempre que algum objeto colide com o objeto alvo, o método `on_collision` do alvo e do objeto com o qual ele colidiu é chamado.

A *engine* desenvolvida está disponível no seguinte repositório: <https://github.com/traveling-will-gba/gbengine>

3.2.5 Módulo Utilitário

Este módulo contém funções utilitárias que estão relacionadas diretamente com *hardware* do GBA.

A primeira delas é a função `print`. Esta função é necessária pois, diferentemente de um programa executado em um PC, a ROM não possui redirecionamento de *output* para uma saída padrão. Após diversas buscas em fóruns de desenvolvimento para GBA, foi encontrada uma solução que permite realizar o redirecionamento do *output buffer* para a saída padrão (*stdin*). A solução consiste em um código *Assembly* que contém uma chamada de sistema `vbaprint`. O código 3.13 apresenta a implementação *Assembly* mencionada.

Código 3.13 – Código *Assembly* para a função `vbaprint`

```
1 .arm
2 .global vbaprint
3 .type   vbaprint STT_FUNC
4 .text
5 vbaprint:
6     swi 0xFF0000
7         bx lr
```

Após isso é necessário declarar a função `vbaprint` em um cabeçalho em C. O código 3.14 apresenta este cabeçalho.

Código 3.14 – Cabeçalho da função `vbaprint`

```
1 #ifndef VBAPRINT_H
2 #define VBAPRINT_H
3
4 extern "C" void vbaprint(const char *message);
5
6 #endif
```

A implementação da função `print` utiliza o mesmo modelo de implementação da função `printf` da linguagem C. Ela recebe como parâmetros uma string e uma notação que indica que esta função recebe um número variável de parâmetros. Esses parâmetros variáveis são guardados em uma variável do tipo `va_list`. Para realizar o *parse* da *string* recebida é utilizada a função `vsnprintf`, que recebe a lista de parâmetros, faz a resolução das variáveis na *string* e retorna a string com as substituições feitas. Após isso, é chamada a função `vbaprint` que se encarrega de escrever o *buffer* recebido na saída padrão. O código 3.15 apresenta a implementação da função `print`.

Código 3.15 – Implementação da função `print`

```
1  int print(const char *fmt, ...) {
2      va_list args;
3      va_start(args, fmt);
4
5      char buffer[4096];
6
7      int rc = vsnprintf(buffer, sizeof(buffer), fmt, args);
8
9      vbaprint(buffer);
10     va_end(args);
11
12     return rc;
13 }
```

A próxima função utilitária é a `mem16cpy`. Esta função funciona de forma similar à função `memcpy` da linguagem C, porém copia os dados de uma área de memória para outra 2 *bytes* por vez. Isso se faz necessário pois a `memcpy` original faz cópias *byte* a *byte* quando o alinhamento da memória não está compatível com o tamanho da palavra (32 *bits*, no caso da arquitetura ARM) na arquitetura utilizada e a VRAM não aceita inserções feitas 1 *byte* por vez (VIJN, 2008).

O código 3.16 apresenta a implementação da função `mem16cpy`.

Código 3.16 – Implementação da função `mem16cpy`

```
1  void mem16cpy(volatile void *dest, const void *src, size_t n)
2  {
3      if (n & 1) {
4          print("Size must be even");
5      }
6
7      for (int i = 0; i < n / 2; i++) {
8          *(((volatile uint16_t *)dest) + i) = *(((uint16_t *)src) + i);
9      }
10 }
```

A última função deste módulo utilitário é a `vsync` (*vertical synchronization*). Ela tem como responsabilidade servir como um mecanismo para garantir a atualização do jogo durante o período de VBlank (período onde há uma pausa na atualização vertical da tela do jogo). O código 3.17 apresenta a implementação do `vsync`.

Código 3.17 – Implementação da função `vsync`

```
1  void vsync() {
2      while(REG_VCOUNT >= 160);
3      while(REG_VCOUNT < 160);
4  }
```

3.2.6 Abstrações de níveis e objetos do jogo

Além de implementar módulos relacionados a entrada (*input*) e saída (vídeo, áudio, entre outros), a *engine* também implementa abstrações para os níveis e para os objetos do jogo.

Os *game objects* são a base para qualquer elemento utilizado no jogo. Um *game object* possui uma posição `(x, y)` no espaço do jogo, possui velocidade horizontal e vertical, possui *game objects* filhos (que formam uma hierarquia de *game objects*) e tem como responsabilidades atualizar seu próprio estado, se desenhar na tela e atualizar os seus filhos.

Pelo fato de o mecanismo de renderização ser, de certa forma, automático (basta carregar as imagens na memória e especificar os metadados que elas já são renderizadas na tela), a função de se desenhar na tela não é utilizada. Porém, a atualização e controle dos filhos é importante para o comportamento correto desses objetos. O código 3.18 apresenta a classe `GameObject`.

Código 3.18 – Classe `GameObject`

```
1  #ifndef GAME_OBJECT_H
2  #define GAME_OBJECT_H
3
4  #include <vector>
5  #include <stdint.h>
6
7  using std::vector;
8
9  class GameObject {
10     protected:
11         GameObject *m_parent;
12         int m_x, m_y;
13         int m_speed_x, m_speed_y;
14         vector <GameObject *> m_children;
15
16         GameObject *parent() const { return m_parent; }
17
18         virtual void update_self(uint64_t dt) = 0;
19         virtual void draw_self() = 0;
20
21     public:
22         void update(uint64_t dt);
23         void draw();
24
25         void set_parent(GameObject *obj) { m_parent = obj; }
```

```
26          void add_child(GameObject *);
27          void remove_child(GameObject *);
28  };
29
30  #endif
```

Com a definição e implementação dos *game objects*, a caracterização da abstração dos níveis do jogo se torna bem simples. De forma geral, um nível é um *game object* que, além de possuir todas as propriedades já explicitadas acima, possui uma variável que indica se o nível foi finalizado e outra que guarda o próximo nível a ser renderizado. O código 3.19 apresenta a implementação da classe `Level`.

Código 3.19 – Classe `Level`

```
1   #ifndef LEVEL_H
2   #define LEVEL_H
3
4   #include "game_object.h"
5
6   #include <stdio.h>
7
8   class Level : public GameObject {
9       protected:
10          int m_x, m_y;
11          bool m_done;
12          int m_next;
13  };
14
15  #endif
```

3.3 Desenvolvimento do jogo

De acordo com o cronograma apresentado na versão anterior, as funcionalidades do jogo original que foram planejadas para a realização deste trabalho são:

- F1: implementação do menu principal do jogo
- F2: implementação da rolagem infinita dos *backgrounds*
- F3: implementação do mecanismo de renderização das plataformas
- F4: implementação da movimentação do personagem principal
- F5: implementação do mecanismo de renderização dos itens coletáveis
- F6: implementação das telas de finalização dos níveis

- F7: carregamento dos níveis a partir do *level design*
- F8: implementação do seletor de fases
- F9: implementação das opções do menu
- F10: implementação do tutorial

Da lista acima, as funcionalidades `F8`, `F9` e `F10` foram removidas da realização do projeto.

O desenvolvimento do jogo foi marcado por quatro fases principais (que ocorreram simultaneamente): adaptação das imagens do jogo, construção dos níveis do jogo, transição entre os níveis do jogo e adaptação das músicas do jogo.

3.3.1 Adaptação das imagens do jogo

Esta fase foi marcada principalmente pelo entendimento de como imagens são carregadas e interpretadas pelo GBA.

Primeiramente, era necessário conhecer a resolução das imagens no jogo original e no jogo a ser portado para que pudesse ser estabelecida uma proporção a ser utilizada na adaptação das imagens. Para isso, foram utilizadas as imagens de *background*, pois elas ocupam todo o espaço da janela do jogo original. A altura dos *backgrounds* no jogo original é *480px* e, sabendo que a tela do GBA possui *160px*, é possível estabelecer um fator de conversão bastante preciso, seguindo a fórmula

$$\frac{480_{px}}{160_{px}} = \frac{3}{1} \tag{3.2}$$

Portanto, a proporção das imagens do jogo original para o jogo a ser portado é de `3:1`.

Após realizar o redimensionamento das imagens para o tamanho correto, foi necessário descobrir como utilizar essas imagens no GBA. Diferentemente de sistemas mais modernos, o GBA não carrega imagens de fato, e sim um código C que contém as informações da imagem, como paleta de cores, *tiles* e o mapeamento desses *tiles* na imagem. Para realizar a conversão da imagem para este código, foi utilizada a ferramenta GRIT (*GBA Raster Image Transmogrifier*). Com essa ferramenta é possível converter a imagem utilizando uma série de parâmetros, como quantidade de bits por pixel da imagem, formato de redução de *tiles*, formato de saída do arquivo gerado (C, *Assembly*, entre outros), altura e largura, em *tiles* (conjuntos de *8px* x *8px*), da imagem , dentre outras opções. Para a conversão das *sprites* do jogo, o seguinte comando foi utilizado:

Código 3.20 – Comando para conversão das imagens em código

```
$ grit nome-da-imagem.png -gB4 -ftc -Mw2 -Mh4
```

Nesse comando, o código relativo à imagem é gerado considerando que cada *pixel* da imagem utiliza 4 *bits* (**-gb4**), exportando a imagem como código C (**-ftc**) e definindo a largura e a altura da imagem como sendo, respectivamente, *16px* (**-Mw2**) e *32px* (**-Mh4**). O resultado da execução dessa instrução é um *header* e um código-fonte correspondentes à imagem, contendo uma paleta com as cores utilizadas e um vetor de *tiles*, assim como mostrado nos códigos 3.21 e 3.22.

Código 3.21 – Cabeçalho da parte superior da imagem da plataforma da primeira fase.

```
//{{BLOCK(level1_plat0)

//=======================================================================
//
//  level1_plat0, 16x32@4,
//  Transparent palette entry: 17.
//  + palette 16 entries, not compressed
//  + 8 tiles Metatiled by 2x4 not compressed
//  Total size: 32 + 256 = 288
//
//  Time-stamp: 2018-11-28, 22:18:37
//  Exported by Cearn's GBA Image Transmogrifier, v0.8.15
//  ( http://www.coranac.com/projects/#grit )
//
//=======================================================================

#ifndef GRIT_LEVEL1_PLAT0_H
#define GRIT_LEVEL1_PLAT0_H

#define level1_plat0TilesLen 256
extern const unsigned int level1_plat0Tiles[64];

#define level1_plat0PalLen 32
extern const unsigned short level1_plat0Pal[16];

#endif // GRIT_LEVEL1_PLAT0_H

//}}BLOCK(level1_plat0)
```

Código 3.22 – Código fonte da parte superior da imagem da plataforma da primeira fase.

```
//{{BLOCK(level1_plat0)

//=======================================================================
//
```

```
5  //  level1_plat0, 16x32@4,
6  //  Transparent palette entry: 17.
7  //  + palette 16 entries, not compressed
8  //  + 8 tiles Metatiled by 2x4 not compressed
9  //  Total size: 32 + 256 = 288
10 //
11 //  Time-stamp: 2018-11-28, 22:18:37
12 //  Exported by Cearn's GBA Image Transmogrifier, v0.8.15
13 //  ( http://www.coranac.com/projects/#grit )
14 //
15 //================================================================
16
17 const unsigned int level1_plat0Tiles[64] __attribute__((aligned(4)))
       __attribute__((visibility("hidden")))=
18 {
19     0xCCCCCCCC,0xAAACCCCC,0x888AACCB,0x89988BBA,0x99889888,0x99898221,0
           x98822222,0x81122444,
20     0xCCCCCCCC,0xCCCCCCBA,0xCCCCCA89,0xBB888989,0x78888988,0x25889988,0
           x44225889,0x22444168,
21     0x22244223,0x44444444,0x44444444,0x44444444,0x44444444,0x44444444,0
           x44444444,0x44444444,
22     0x44223432,0x44444244,0x44444444,0x44444444,0x44444444,0x44444444,0
           x44444444,0x44444444,
23     0x44444444,0x44444444,0x44444444,0x43344334,0x24422222,0x42222222,0
           x22222222,0x22222222,
24     0x44444444,0x44444444,0x44444444,0x33444344,0x44223432,0x22444244,0
           x22222222,0x11222222,
25     0x22222222,0x21122222,0x11111111,0x11111111,0x11111111,0x11111111,0
           x11111111,0x11111111,
26     0x11222222,0x11111122,0x11111111,0x11111111,0x11111111,0x11111111,0
           x11111111,0x11111111,
27 };
28
29 const unsigned short level1_plat0Pal[16] __attribute__((aligned(4)))
       __attribute__((visibility("hidden")))=
30 {
31     0x0002,0x0889,0x088B,0x088D,0x10D0,0x092D,0x09AD,0x0A50,
32     0x0A90,0x1290,0x0AD0,0x1332,0x1B74,0x0000,0x0000,0x0000,
33 };
34
35 //}}BLOCK(level1_plat0)
```

Já para a conversão das imagens dos *backgrounds* foi utilizado o seguinte comando:

Código 3.23 – Comando para conversão dos *backgrounds* em código

```
1 $ grit nome-da-imagem.png -gB4 -ftc -mRtf -mp0
```

Assim como na imagem anterior, o código é gerado em C considerando que cada *pixel* da imagem utiliza 4 *bits*. Além disso, o código é gerado utilizando redução completa de *tiles* (**-mRtf**) e, diferentemente das sprites, onde o índice a ser utilizado na paleta de cores é especificado apenas no código, é necessário especificar, já nessa instrução (**-mp0**), o índice da paleta de cores que será utilizada no código para guardar as cores da imagem. Vale ressaltar que isso é necessário apenas caso a imagem esteja sendo convertida a 4 *bits* por *pixel*, já que a 8 *bits* por *pixel* é possível utilizar apenas uma única paleta de cores, assim como já explicado na seção 1.2.1.

Nesse ponto da explicação é necessário saber diferenciar 3 coisas: o *background* do GBA, as imagens que irão compor o *background* do jogo e o que de fato vai ser mostrado na tela. O *background* do GBA é todo o espaço disponível para o mapa do jogo, ele determina os limites do jogo. Se ele tiver *512px* de largura, os limites horizontais do mapa irão de 0 a 512. Esse será o espaço disponível para carregamento de todo e qualquer elemento do jogo, seja ele *sprite* ou imagem de fundo. Porém, nem tudo que estiver carregado no *background* do GBA será necessariamente mostrado na tela, afinal a tela do GBA possui apenas *240px* de largura e *160px* de altura. O que será de fato mostrado dependerá do que será explicado à frente.

Nos modos de vídeo baseados em *tiles*, o GBA possui um total de 4 *backgrounds*, sendo que existem restrições explícitas para o tamanho de cada um deles. As dimensões aceitas são as seguintes: 256x256, 512x256, 256x512, 512x512. Cada um desses *backgrounds* possui um registrador por meio do qual é possível escolher suas dimensões. A escolha das dimensões é feita por meio de uma *flag* atribuída diretamente no registrador `REG_BGxCNT`. Cada um desses *backgrounds* pode possuir um tamanho diferente, desde que siga as restrições já citadas.

Para passar a ideia de que o personagem está se movimentando durante o jogo, as imagens dos *backgrounds* se movem horizontalmente em velocidades diferentes, técnica chamada *parallax*. Fazer isso convertendo a imagem em uma maior, a replicando em um editor de imagens para posteriormente carregá-la por completo no código provavelmente funcionaria em uma plataforma atual. Porém, no GBA, onde a memória é extremamente limitada, não seria possível nem mesmo carregar tal imagem e mesmo em uma plataforma atual muita memória seria gasta desnecessariamente. Por isso, para realizar essa movimentação, os *backgrounds* do jogo são reproduzidos mudando o ponto onde a renderização começa, sendo que sempre que o *background* chega ao fim, a renderização continua do início. E aqui é importante ressaltar que é o fim do *background* do GBA e não das imagens, por isso é importante que as imagens possuam os mesmos tamanhos dos *backgrounds* em que estão sendo renderizadas nos eixos aonde a movimentação ocorre. Por exemplo, em um *background* de 512x256, em que a imagem de fundo se move somente no sentido horizontal, é necessário que essa imagem de fundo possua *512px* de largura, porém para a

altura, basta que a imagem possua a altura da tela, já que ela não será movimentada na vertical e por isso os *pixels* da parte de cima nunca serão vistos.

Na versão original do jogo, o movimento dos *backgrounds* também é realizado dessa forma, porém tendo que manualmente (no código), manipular os índices onde a renderização inicia para passar a ideia de que o *background* é infinito. O GBA facilita esse processo, pois possui registradores que sinalizam o ponto onde acontece o início da renderização da imagem e quando a imagem chega ao fim, ele automaticamente emenda o fim com o início. Dessa forma, se a imagem possuir *256px* de largura, basta que o valor contido no registrador seja continuamente incrementado, sem que seja necessário resetar o valor quando a imagem chegar no seu limite. Outra vantagem que o GBA proporciona ao realizar essa movimentação dos *backgrounds* é o fato de que não há tamanho mínimo para a interseção entre o fim e o início da imagem. Na versão original, por exemplo, em que a resolução era de *852px x 480px*, era necessário que os *852px* finais da imagem fossem iguais aos *852px* iniciais, pois assim, quando o índice do fundo fosse resetado, isso não seria vísivel para quem está jogando, passando a ideia de continuidade. Esse fator facilitou até mesmo a edição das imagens, fazendo com que o *background* pudesse ser mais diverso, sem que fosse necessário gastar grande parte do fim da imagem apenas replicando o início. No código 3.24, é possível visualizar uma amostra de como é feita a atualização dos índices. É importante ressaltar que esse efeito da renderização continuar no início acontece para qualquer textura, e não apenas com os mapas. Sendo assim, uma *sprite* cujas extremidades ultrapassem algum dos limites do *background* aonde ela está sendo renderizada (o mais à frente) terá a parte que ultrapassou o *background* renderizada no lado oposto.

Código 3.24 – Método responsável por atualizar os índices de renderização do *background*

```
1  void Background::update_self(uint64_t dt) {
2      if (dt % frames_to_skip == 0) {
3          m_x += m_speed_x;
4          m_y += m_speed_y;
5      }
6
7      switch(this->background_id) {
8          case 0:
9              REG_BG0HOFS = m_x;
10             REG_BG0VOFS = m_y;
11             break;
12         case 1:
13             REG_BG1HOFS = m_x;
14             REG_BG1VOFS = m_y;
15             break;
16         case 2:
17             REG_BG2HOFS = m_x;
```

```
18              REG_BG2VOFS = m_y;
19              break;
20          case 3:
21              REG_BG3HOFS = m_x;
22              REG_BG3VOFS = m_y;
23              break;
24          default:
25              print("Invalid background id\n");
26              break;
27      }
28  }
```

Dessa forma, sempre que a variável `m_x`, que representa o ponto no eixo X onde começa a renderização do *backgrounds* na tela, é atualizado, o registrador correspondente também é atualizado. O mesmo se aplica à variável `m_y`, que representa o ponto no eixo Y.

Na versão original do jogo, as imagens das plataformas possuíam *20px* de largura e *400px* de altura. Porém, o GBA possui explícitas limitações em relação às dimensões das imagens a serem renderizadas. Sendo assim, é necessário que as imagens estejam em uma das dimensões mostradas na tabela 1. Tendo em vista que a altura máxima de uma *sprite* no GBA é *64px*, foi necessário dividir a imagem em várias para que fosse possível atingir a altura necessária das plataformas no jogo. As dimensões escolhidas para essas novas imagens foram *16px* de largura por *32px* de altura, pois assim a largura ficava mais próxima da largura original, preservando ao máximo o *level design* original. Já a altura de *32px* foi escolhida tendo em vista que é a altura máxima que pode ser utilizada com largura *16px*, como mostrado na tabela 1.

8x8	16x16	32x32	64x64
16x8	32x8	32x16	64x32
8x16	8x32	16x32	32x64

Tabela 1 – Dimensões válidas para renderização de *sprites* no GBA

Devido a essa adaptação das plataformas, a classe `TWPlatform` possui um número variável de texturas diferentemente das classes que representam os demais objetos do jogo, que possuem uma única textura. As plataformas podem utilizar uma textura (caso das plataformas que representam o chão, pois apenas a parte mais superior das plataformas é mostrada) ou por 4 texturas, podendo atingir uma altura máxima de *128px*, correspondendo a 80% da altura da tela do GBA. O código 3.25 apresenta o trecho responsável por carregar as texturas e o código 3.26 o trecho responsável por calcular as alturas de cada uma das texturas.

Código 3.25 – `std::vector` com as texturas das plataformas sendo preenchido.

```
m_textures.push_back(new Texture(1, level1_plat0Pal, level1_plat0PalLen,
    level1_plat0Tiles, level1_plat0TilesLen, _4BPP));
if (not m_is_floor)
{
    m_textures.push_back(new Texture(1, level1_plat1Pal, level1_plat1PalLen
        , level1_plat1Tiles, level1_plat1TilesLen, _4BPP));
    m_textures.push_back(new Texture(1, level1_plat2Pal, level1_plat2PalLen
        , level1_plat2Tiles, level1_plat2TilesLen, _4BPP));
    m_textures.push_back(new Texture(1, level1_plat3Pal, level1_plat3PalLen
        , level1_plat3Tiles, level1_plat3TilesLen, _4BPP));
}
```

Código 3.26 – Cálculo das alturas das texturas utilizadas nas plataformas

```
void TWPlatform::set_y(int y) {
    m_y = y;

    for (size_t i = 0; i < m_textures.size(); i++) {
        m_textures[i]->metadata.y = m_y + platform_height * i;

        if (m_y + platform_height * (i + 1) >= 256) {
            // hide sprite
            m_textures[i]->metadata.om = 2;
        } else {
            m_textures[i]->metadata.om = 0;
        }
    }
}
```

No código 3.25, uma única textura é carregada caso a plataforma faça parte do chão do jogo e 4 plataformas são carregadas caso contrário. A variável `m_is_floor` indica se a plataforma faz ou não parte do chão.

No código 3.26, é possível ver que a variável `m_y` representa o ponto `y` (ponto onde a plataforma começa a ser renderizada na tela) da plataforma em si e por isso é utilizado para calcular o `y` de cada uma das texturas utilizadas na classe.

Ao falar sobre a adaptação do *background*, foi explicado que o GBA automaticamente emenda o fim e o início da tela de fundo, a medida que o ponto onde a renderização da imagem é avançado. O mesmo funciona com as demais imagens do jogo. Sempre que uma imagem ultrapassa as dimensões do *background*, ela começa a ser renderizada no lado oposto. Por exemplo, levando em consideração uma altura de *256px* para o *background* e lembrando que a altura da tela do GBA é de *160px*, se uma imagem de *32px* for renderizada no ponto `y = 150` na tela, os *22px* restantes serão renderizados a partir do topo do *background*, o que não necessariamente significa que esses *pixels* remanescentes serão mostrados na tela, isso pelo fato de serem carregados na tela apenas os *160px* finais da

imagem do *background*. Isso se torna um problema, porém, quando o número de *pixels* que ultrapassam os limites inferiores da tela do GBA são suficientes para alcançar os *160px* finais do *background* e é para isso que serve o trecho que vai da linha 7 à linha 12 no código 3.26. Este trecho esconde a textura caso ela ultrapasse o limite inferior do *background* (independente de chegar ou não ao ponto do restante da imagem ser renderizado na parte de cima da tela) e mostra caso contrário.

3.3.2 Construção dos níveis do jogo

A construção dos níveis do jogo tem como base o *level design* desenvolvido para cada nível do jogo original. Este *level design* possui a seguinte estrutura:

Código 3.27 – Estrutura do *level design* do jogo original

```
1 level_tempo
2 num_platforms num_backgrounds
3 platform_height is_enemy_present [enemy_type enemy_height]
    is_collectable_present [collectable_height]
```

A primeira linha do *level design* contém a velocidade da música que será reproduzida no nível. A linha abaixo contém dois valores: número de plataformas e de *backgrounds* do nível. Após isso, seguem `num_platform` linhas caracterizando cada plataforma: sua altura, se possui um inimigo, tipo e altura do inimigo (caso haja um inimigo), se possui coletável, altura do coletável (caso haja um coletável).

O bloco 3.28 mostra um trecho do level design da primeira fase do jogo original

Código 3.28 – Exemplo do *level design* da fase 1 do jogo original

```
1  100
2  220 3
3  50 0 0
4  50 0 0
5  50 0 0
6  50 0 0
7  50 0 0
8  50 0 0
9  50 0 0
10 115 0 1 155
11 115 0 0
12 115 0 0
13 111 0 0
14 111 0 1 151
15 111 0 0
16 111 0 0
17 111 0 0
18 111 0 1 155
```

```
111 0 0
111 0 0
111 0 0
111 0 1 151
...
```

Para que o *level design* consiga ser lido e utilizado no jogo portado, foi necessário criar um *parser* do *level design* original para um formato legível no GBA. O *parser* tem como responsabilidade ler o arquivo de *level design* e criar vetores de inteiros correspondentes a: altura das plataformas, presença de inimigo, tipo do inimigo, altura do inimigo, presença de coletável e altura do coletável. O código 3.29 apresenta o *script python* criado para realizar o *parser* e os códigos 3.30 e 3.31 mostram o resultado do *parse*.

Código 3.29 – *script python* para realizar o *parse*

```python
import sys

if len(sys.argv) < 3:
    print("File name and level_design path must be passed as arguments")
    sys.exit()

file_name = sys.argv[1]
level_design_path = sys.argv[2]

f = open(level_design_path, "r")

header_file = open("{}.h".format(file_name), "w")

level_tempo = f.readline()
level_len, background_num = f.readline().split()

header_file.write("#ifndef {}_H\n".format(file_name.upper()))
header_file.write("#define {}_H\n\n".format(file_name.upper()))
header_file.write("#define {}_tempo {}\n".format(file_name, level_tempo))
header_file.write("#define {}_len {}\n".format(file_name, level_len))

header_file.write("extern const int {}_platform_heights[{}];\n".format(
    file_name, str(level_len)))
header_file.write("extern const int {}_enemy_present[{}];\n".format(
    file_name, str(level_len)))
header_file.write("extern const int {}_enemy_type[{}];\n".format(file_name,
     str(level_len)))
header_file.write("extern const int {}_enemy_heights[{}];\n".format(
    file_name, str(level_len)))
header_file.write("extern const int {}_collectable_present[{}];\n".format(
    file_name, str(level_len)))
header_file.write("extern const int {}_collectable_heights[{}];\n".format(
```

```
    file_name, str(level_len)))

header_file.write("\n#endif")

heights = []
ep = []
et = []
eh = []
cp = []
ch = []

for line in f.readlines():
    args = line.split()

    platform_height = 0
    enemy_present = False
    enemy_type = 0
    enemy_height = 0
    collectable_present = False
    collectable_height = 0

    if len(args) == 0:
        continue

    platform_height = args[0]
    enemy_present = int(args[1])

    if enemy_present:
        enemy_type = args[2]
        enemy_height = args[3]
    else:
        collectable_present = int(args[2])
        if collectable_present:
            collectable_height = args[3]


    heights.append(str(platform_height))
    ep.append(str("1" if enemy_present else "0"))
    et.append(str(enemy_type))
    eh.append(str(enemy_height))
    cp.append(str("1" if collectable_present else "0"))
    ch.append(str(collectable_height))

header_file.close()

cpp_file = open("{}.c".format(file_name), "w")

```

```
74 cpp_file.write("#include \"{}.h\"\n\n".format(file_name))
75 cpp_file.write("const int {}_platform_heights[{}] = {{ {} }};\n".format(
       file_name, str(len(heights)), ', '.join(heights)))
76 cpp_file.write("const int {}_enemy_present[{}] = {{ {} }};\n".format(
       file_name, str(len(ep)), ', '.join(ep)))
77 cpp_file.write("const int {}_enemy_type[{}] = {{ {} }};\n".format(file_name
       , str(len(et)), ', '.join(et)))
78 cpp_file.write("const int {}_enemy_heights[{}] = {{ {} }};\n".format(
       file_name, str(len(eh)), ', '.join(eh)))
79 cpp_file.write("const int {}_collectable_present[{}] = {{ {} }};\n".format(
       file_name, str(len(cp)), ', '.join(cp)))
80 cpp_file.write("const int {}_collectable_heights[{}] = {{ {} }};\n".format(
       file_name, str(len(ch)), ', '.join(ch)))
81
82 cpp_file.close()
```

Código 3.30 – Código-fonte gerado pelo *level design parser*

```
1  #ifndef LEVEL1_H
2  #define LEVEL1_H
3
4  #define level1_tempo 100
5
6  #define level1_len 220
7  extern const int level1_platform_heights[220];
8  extern const int level1_enemy_present[220];
9  extern const int level1_enemy_type[220];
10 extern const int level1_enemy_heights[220];
11 extern const int level1_collectable_present[220];
12 extern const int level1_collectable_heights[220];
13
14 #endif
```

Código 3.31 – Amostra do *header* gerado pelo *level design parser*

```
1  #include"level1.h"
2
3  const int level1_platform_heights[220] = { 50, 50, 50, 50, 50, 50, 50, 115,
       115, 115, 111, 111, 111, ... };
4  const int level1_enemy_present[220] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, ... };
5  const int level1_enemy_type[220] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, ... };
6  const int level1_enemy_heights[220] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, ... };
7  const int level1_collectable_present[220] = { 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
       0, 1, 0, 0, 0, 1, 0, 0, ... };
```

```
8 const int level1_collectable_heights[220] = { 0, 0, 0, 0, 0, 0, 0, 155, 0,
    0, 0, 151, 0, 0, 0, 155, ... };
```

Em um cenário ideal, o carregamento dos elementos do nível (plataformas, inimigos e coletáveis) seria feito por completo no momento que o nível inicia. Porém, essa abordagem não é possível no *GBA* devido ao fato de que a região de memória que armazena as *sprites* e paletas é bastante limitada e não comporta o carregamento de todas as plataformas. Dessa forma, a solução encontrada envolve a utilização de duas abordagens em conjunto: reutilizar texturas e carregar somente os itens visíveis.

A primeira abordagem envolve utilizar somente uma textura em todas as plataformas. Dessa maneira, somente as *sprites* de uma plataforma e de um coletável são carregados em memória. Essa primeira abordagem é possível utilizando o construtor por cópia da classe `Texture`, explicado na seção 3.2.2. O código 3.32

Código 3.32 – Implementação da reutilização das *sprites* de uma plataforma

```
1  for (int i = 0; i < max_platforms_loaded; i++)
2  {
3      if (i == 0)
4      {
5          platforms[i] = new TWPlatform(level, i * platform_width,
               platform_height[i]);
6          collectables[i] = new TWCollectable(level, i * platform_width +
               platform_width / 2 - collectable_width / 2,
7              collectable_height[i]);
8      }
9      else
10     {
11         // reusing platform and collectable textures
12         platforms[i] = new TWPlatform(i * platform_width, platform_height[i
               ], platforms[0]->textures());
13         collectables[i] = new TWCollectable(i * platform_width +
               platform_width / 2 - collectable_width / 2,
14             collectable_height[i], collectables[0]->texture());
15     }
16
17     platforms[i]->set_collectable(collectables[i]);
18     collectables[i]->set_visibility(collectable_present[i]);
19
20     add_child(platforms[i]);
21     q.push(platforms[i]);
22 }
```

Já a segunda abordagem se trata de carregar somente as plataformas e coletáveis visíveis na tela durante a execução do nível. A largura da tela do *GBA* é 240px e a largura

das plataformas utilizadas nos níveis é 16px. Dados esses dois valores, é possível notar que, caso as plataformas não se mexam, o número máximo de plataformas que podem aparecer simultaneamente na tela é 15.

$$\frac{240}{16} = 15 \tag{3.3}$$

Porém, com as plataformas se movendo esse número sobe para 16. Portanto, só é necessário renderizar 16 plataformas simultaneamente em qualquer momento do nível. Com isso, a solução pode ser dada utilizando uma fila, onde as 16 primeiras plataformas são carregadas em memória e, assim que a próxima for aparecer na tela, a primeira plataforma é removida da fila e inserida novamente ao final, com sua posição e altura modificadas. Assim, o carregamento das plataformas e coletáveis se faz possível utilizando apenas o número máximo de itens visíveis na tela. O código 3.33 apresenta como é feita a checagem da plataforma que está prestes a aparecer e a remoção da primeira plataforma e sua inserção ao fim da fila.

Código 3.33 – Implementação da checagem da plataforma prestes a aparecer

```
1  // neste exemplo, platform_idx corresponde ao indice da plataforma mais a
       direita da tela
2  while (1) {
3      if (platform_idx * platform_width <= m_backgrounds[0]->x() +
           screen_width) {
4          auto plat = q.front();
5          q.pop();
6
7          plat->set_x(platform_idx * platform_width - m_backgrounds[0]->x());
8          plat->set_y(platform_height[platform_idx]);
9
10         plat->collectable()->set_y(collectable_height[platform_idx]);
11         plat->collectable()->set_visibility(collectable_present[
               platform_idx]);
12
13         q.push(plat);
14     } else break;
15
16     platform_idx++;
17 }
```

Por fim, é importante explicitar que a velocidade das plataformas é dada pela velocidade do *background* mais à frente e, que, quando a última plataforma é renderizada, a velocidade tanto dos *backgrounds* quanto da plataforma são zeradas e o personagem principal corre para a direita (na mesma velocidade das plataformas antes de serem zeradas) até sair da tela, indicando o fim do nível.

3.3.3 Transição entre os níveis do jogo

No jogo portado há dois tipos de níveis: jogáveis e não-jogáveis. Os níveis jogáveis são aqueles onde o usuário está realmente jogando o jogo, enquanto os não-jogáveis podem ser representados pelos menus e telas de vitória e derrota.

O fluxo atual do jogo realiza a transição automática entre os níveis da seguinte forma: O jogo sempre começa no menu principal. Quando o jogador pressiona o botão *START* no menu, ele é levado à primeira fase jogável do jogo. Quando a primeira fase acaba, ele é levado à tela de vitória. Na tela de vitória, o jogador é redirecionado ao menu principal caso pressione o botão B e é levado à próxima fase caso pressione o botão A. Na última fase jogável, após terminar o nível e ser levado à tela de vitória, ele é redirecionado ao menu principal do jogo.

O esquema de transição entre os níveis se dá utilizando a propriedade `m_next` da classe `TWLevel`, classe que herda da classe abstrata `Level`. Cada nível é representado por um identificador inteiro, como mostrado no código 3.35.

Código 3.34 – Identificação dos níveis do jogo.

```
1  #define LEVEL_MENU 0
2  #define LEVEL_1 1
3  #define LEVEL_2 2
4  #define LEVEL_3 3
5  #define LEVEL_4 4
6  #define LEVEL_5 5
7  #define LEVEL_6 6
8  #define MENU_VICTORY 7
9  #define MENU_DEFEAT 8
10 #define NEXT_LEVEL 42
```

Quando o nível acaba, a própria classe `TWLevel` decide para onde o jogador vai, utilizando duas dessas variáveis: `LEVEL_MENU`, para redirecionar o jogador ao menu principal e `NEXT_LEVEL`, para informar à classe `TWGame` que o jogador irá para o próximo nível. Sendo assim, a classe `TWGame`, responsável por instanciar os níveis do jogo, ao receber essa informação por meio do método `level->next()`, calcula qual o nível seguinte, altera sua propriedade `current_level` e instancia o novo nível. O código 3.35 apresenta esta implementação.

Código 3.35 – Transição entre os níveis do jogo.

```
1 if (m_level->done()) {
2     if (m_level->next() == NEXT_LEVEL) {
3         current_level = (previous_playable_level + 1) % (LOADED_LEVELS + 1)
              ;
4         if (current_level == 0)
5             current_level++;
```

```
6       }
7       else {
8           previous_playable_level = current_level;
9           current_level = m_level->next();
10      }
11
12      delete m_level;
13
14      bool is_playable = current_level != LEVEL_MENU && current_level !=
            MENU_VICTORY
15          && current_level != MENU_DEFEAT;
16
17
18      m_level = new TWLevel(current_level, is_playable);
19  }
```

A criação das texturas relativas à cada nível é feita utilizando o nível como parâmetro para decidir quais plataformas, coletáveis e *backgrounds* instanciar e renderizar. Isso foi necessário pois cada nível contém uma série diferente de variáveis para plataformas, coletáveis e *backgrounds* e, desta forma, também é possível abranger os menus não-jogáveis, como o menu principal e as telas de finalização de um nível. O código 3.36 apresenta a definição dos *backgrounds* baseados no nível atual passado como parâmetro.

Código 3.36 – Definição dos *backgrounds* baseado no nível passado.

```
1   void TWLevel::load_backgrounds(int level) {
2       m_backgrounds.clear();
3
4       switch(level) {
5           case LEVEL_MENU:
6               m_backgrounds.push_back(new Background(menuPal, menuPalLen,
                    menuTiles, menuTilesLen, menuMap, menuMapLen, 0, 0, 0, 0, 0)
                    );
7               break;
8           case MENU_VICTORY:
9               m_backgrounds.push_back(new Background(victoryPal,
                    victoryPalLen, victoryTiles, victoryTilesLen, victoryMap,
                    victoryMapLen, 0, 0, 0, 0, 0));
10              break;
11          case MENU_DEFEAT:
12              m_backgrounds.push_back(new Background(defeatPal, defeatPalLen,
                     defeatTiles, defeatTilesLen, defeatMap, defeatMapLen, 0, 0,
                    0, 0, 0));
13              break;
14          case LEVEL_1:
15              m_backgrounds.push_back(new Background(level1_b0Pal,
                    level1_b0PalLen, level1_b0Tiles, level1_b0TilesLen,
```

```
                level1_b0Map, level1_b0MapLen, 0, 0, 0, 1, 0));
16              m_backgrounds.push_back(new Background(level1_b1Pal,
                    level1_b1PalLen, level1_b1Tiles, level1_b1TilesLen,
                    level1_b1Map, level1_b1MapLen, 1, 0, 0, 1, 0));
17              m_backgrounds.push_back(new Background(level1_b2Pal,
                    level1_b2PalLen, level1_b2Tiles, level1_b2TilesLen,
                    level1_b2Map, level1_b2MapLen, 2, 0, 0, 2, 0));
18
19              m_backgrounds[0]->set_frames_to_skip(2);
20              break;
21          case LEVEL_2:
22              m_backgrounds.push_back(new Background(level2_b0Pal,
                    level2_b0PalLen, level2_b0Tiles, level2_b0TilesLen,
                    level2_b0Map, level2_b0MapLen, 0, 0, 0, 1, 0));
23              m_backgrounds.push_back(new Background(level2_b1Pal,
                    level2_b1PalLen, level2_b1Tiles, level2_b1TilesLen,
                    level2_b1Map, level2_b1MapLen, 1, 0, 0, 1, 0));
24              m_backgrounds.push_back(new Background(level2_b2Pal,
                    level2_b2PalLen, level2_b2Tiles, level2_b2TilesLen,
                    level2_b2Map, level2_b2MapLen, 2, 0, 0, 2, 0));
25
26              m_backgrounds[0]->set_frames_to_skip(2);
27              break;
28          // other levels supressed...
29          default:
30              break;
31      }
32
33      for (auto background : m_backgrounds)
34      {
35          add_child(background);
36      }
37  }
```

3.3.4 Adaptação das músicas do jogo

Antes de falar como foi feita a implementação e carregamento das músicas do jogo no GBA, é importante explicar brevemente como funciona o áudio no *console*. O GBA possui 4 canais específicos para áudio, sendo os canais 1 e 2 geradores de ondas quadradas, o canal 3 para reproduzir *samples* e o canal 4 como gerador de gerador de ruído (VIJN, 2013).

A versão inicial do módulo de áudio conseguia reproduzir somente ondas quadradas por uma determinada duração. Para que isso pudesse ser possível, era necessário passar para a função duas listas: uma com as notas a serem reproduzidas e outra com a respectiva

duração das notas.

Para conseguir executar as músicas do jogo original utilizando ondas quadradas, foi implementado um *parser* em cima do *parser* do *level design* do jogo original. O *parser* do *level design* do jogo original gera o tamanho das plataformas e coletáveis a partir das notas de um arquivo *.ly* (arquivo gerado utilizando a biblioteca `lilypond`[5] e que contém as informações da música de um nível). O incremento realizado neste *script* permite capturar a nota e exportá-la para um vetor secundário, juntamente com sua duração. O código 3.37 apresenta o trecho responsável por gerar os vetores com as notas e suas durações. Já o código 3.38 mostra os vetores gerados para a música da primeira fase.

Código 3.37 – *Parser* incrementado para geração das notas e tempos

```
1  void generate_sound_files(string filename, const vector<int> notes, const
       vector<int> tempos) {
2      string header_filename = filename + ".h";
3      FILE* header_file = fopen(header_filename.c_str(), "w");
4
5      int size = notes.size();
6
7      string upper_filename(filename);
8
9      for(auto &c : upper_filename) {
10         c = toupper(c);
11     }
12
13     // write header
14     fprintf(header_file, "#ifndef %s_H\n", upper_filename.c_str());
15     fprintf(header_file, "#define %s_H\n\n", upper_filename.c_str());
16     fprintf(header_file, "#define %s_notes_len %d\n", filename.c_str(),
           size);
17     fprintf(header_file, "#define %s_tempos_len %d\n\n", filename.c_str(),
           size);
18     fprintf(header_file, "extern const int %s_notes[%d];\n", filename.c_str
           (), size);
19     fprintf(header_file, "extern const int %s_tempos[%d];\n", filename.
           c_str(), size);
20     fprintf(header_file, "\n#endif\n");
21
22     fclose(header_file);
23     // end write header
24
25     // write c
26     string c_filename = filename + ".c";
27     FILE* c_file = fopen(c_filename.c_str(), "w");
28     fprintf(c_file, "#include \"%s.h\"\n\n", filename.c_str());
```

[5] *Lilypond*, disponível em <http://lilypond.org/>

```
29      fprintf(c_file, "const int %s_notes[%d] = {\n", filename.c_str(), size)
            ;
30
31      string notes_str = "\t";
32      for(int i=0; i<size; i++) {
33          if (i) notes_str += ", ";
34          notes_str += to_string(notes[i]);
35      }
36
37      fprintf(c_file, notes_str.c_str());
38      fprintf(c_file, "\n};\n\n", filename.c_str(), size);
39      fprintf(c_file, "const int %s_tempos[%d] = {\n", filename.c_str(), size
            );
40
41      string tempos_str = "\t";
42
43      for(int i=0; i<size; i++) {
44          if (i) tempos_str += ", ";
45          tempos_str += to_string(tempos[i]);
46      }
47
48      fprintf(c_file, tempos_str.c_str());
49      fprintf(c_file, "\n};\n", filename.c_str(), size);
50
51      fclose(c_file);
52      // end write c
53  }
```

Código 3.38 – Código gerado com notas e durações.

```
1   // header file
2   #ifndef FASE1_H
3   #define FASE1_H
4
5   #define fase1_notes_len 70
6   #define fase1_tempos_len 70
7
8   extern const int fase1_notes[70];
9   extern const int fase1_tempos[70];
10
11  #endif
12
13  // source code
14  #include "fase1.h"
15
16  const int fase1_notes[70] = {
17          1, 0, 1, 0, 10, 8, 5, 3, 2, 3, 2, 0, 10, 7, 7, 6, 7, 6, 3, 2, 0,
                10, 9, 7, 5, 7, 9, 10, 9, 10, 0, 2, 1, 0, 1, 3, 1, 0, 10, 8, 5,
```

```
            2, 3, 2, 3, 2, 3, 2, 3, 2, 10, 7, 7, 6, 7, 6, 3, 2, 0, 10, 9, 7,
             5, 7, 9, 10, 9, 10, 0, 2
};

const int fase1_tempos[70] = {
        4, 4, 4, 2, 2, 2, 2, 4, 4, 4, 2, 2, 2, 2, 4, 4, 4, 2, 2, 2, 2, 2,
           2, 2, 2, 2, 2, 2, 2, 2, 2, 4, 4, 4, 2, 1, 1, 2, 2, 2, 2, 1, 1,
           1, 1, 4, 4, 2, 1, 1, 2, 2, 4, 4, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2,
           2, 2, 2, 2, 2, 4
};
```

O *parser* incrementado pode ser encontrado no seguinte endereço: <https://github.com/traveling-will-gba/traveling-will/blob/master/res/parser/sound_parser.cpp>

As funções relativas à reprodução de ondas quadradas são: `init()`, responsável por inicializar os registradores relativos ao áudio; `load_from_file()`, responsável por ler os vetores com as notas e durações; `play_note()`, responsável por reproduzir uma nota por um período de tempo; e `play()`, responsável por reproduzir todas as notas. O código 3.39 apresenta estes métodos.

Código 3.39 – Reprodução de notas e durações

```
1  void Sound::init() {
2          // turn sound on
3          REG_SNDSTAT= SSTAT_ENABLE;
4          // snd1 on left/right ; both full volume
5          REG_SNDDMGCNT = SDMG_BUILD_LR(SDMG_SQR1, 7);
6          // DMG ratio to 100%
7          REG_SNDDSCNT= SDS_DMG100;
8          // no sweep
9          REG_SND1SWEEP= SSW_OFF;
10         // envelope: vol=12, decay, max step time (7) ; 50% duty
11         REG_SND1CNT= SSQR_ENV_BUILD(12, 0, 7) | SSQR_DUTY1_2;
12         REG_SND1FREQ= 0;
13 }
14
15 void Sound::load_from_file(int notes_len, const int* notes, int tempos_len,
       const int* tempos) {
16     this->notes = notes;
17     this->notes_len = notes_len;
18     this->tempos = tempos;
19     this->tempos_len = tempos_len;
20 }
21
22 void Sound::play_note(int raw_note, int tempo) {
23         int note = raw_note % 12;
24         int octave = raw_note / 12;
25         REG_SND1FREQ = SFREQ_RESET | SND_RATE(note, octave);
```

```
26
27          for(int i=0; i<8*tempo; i++) {
28                  vsync();
29          }
30  }
31
32  void Sound::play() {
33          for(int i=0; i < notes_len; i++) {
34                  print("note %d tempo %d\n", notes[i], tempos[i]);
35                  play_note(notes[i], tempos[i]);
36          }
37  }
```

Para que fosse possível carregar as músicas no GBA, foi necessário reduzir a frequência dos audios originais, técnica conhecida como *downsampling*. No processo de redução, a qualidade dos sons diminui de forma considerável. Após gerar um novo arquivo *.wav* com a frequência reduzida, o arquivo é convertido para o formato *.mod*, para que este, por sua vez, possa ser interpretado pelo GBA.

O código 3.40 mostra o construtor da classe `Sound`, responsável por inicializar a música e deixá-la pronta para ser tocada; o método `get_sound`, que implementa o padrão *singleton* e retorna a instância da classe; e o método `play`, responsável por tocar a música. Os métodos dessa classe apenas encapsulam funções já existentes na *libgba*.

Código 3.40 – Classe `Sound`

```
1   Sound* Sound::instance;
2
3   Sound::Sound() {
4       irqInit();
5
6       irqSet( IRQ_VBLANK, mmVBlank );
7       irqEnable(IRQ_VBLANK);
8
9       mmInitDefault( (mm_addr)soundbank_bin, 8);
10      mmStart( MOD_MUSIC43K, MM_PLAY_LOOP );
11  }
12
13  Sound *Sound::get_sound() {
14      if (!instance) {
15          instance = new Sound();
16      }
17
18      return instance;
19  }
20
21  void Sound::play() {
```

```
22      mmFrame() ;
23  }
```

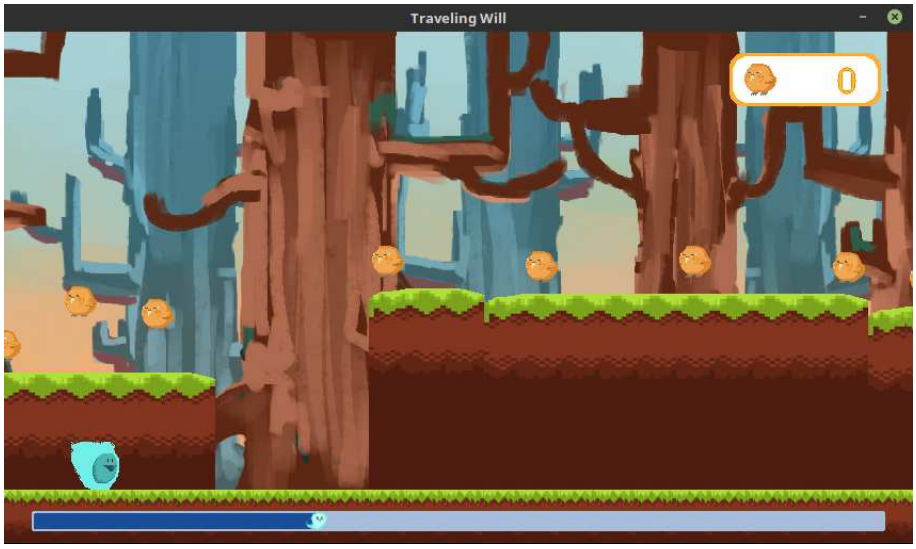
3.3.5 Comparação entre o porte e o jogo original

Esta seção tem como objetivo comparar resultado final do porte do jogo com o jogo original. O jogo portado para GBA pode ser encontrado no seguinte endereço: <https://github.com/traveling-will-gba/gbengine/tree/dev/test/game>.

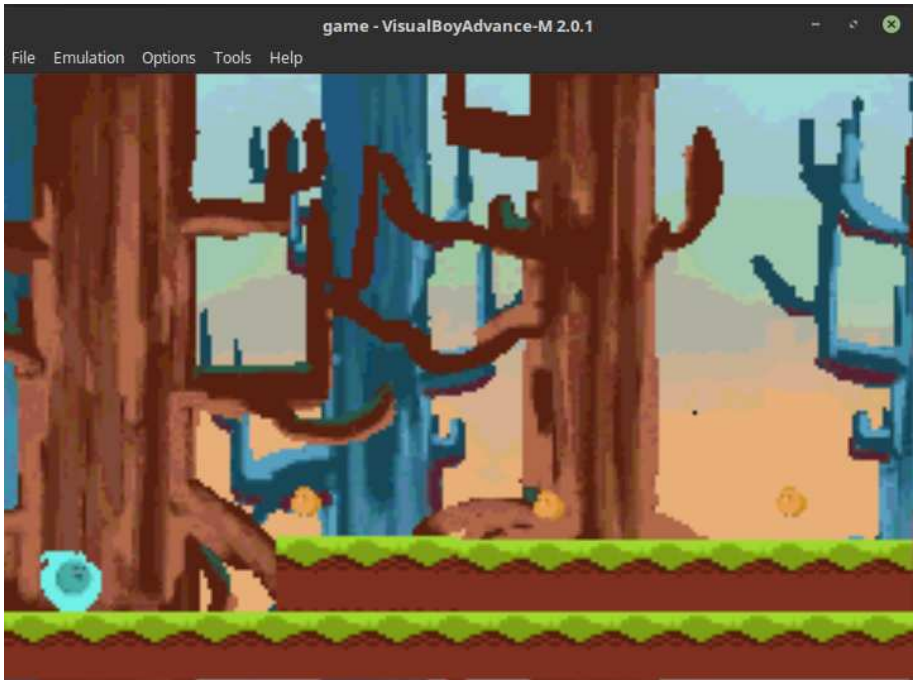
(a) Menu original.

(b) Menu portado.

Figura 10 – Comparação do menu principal.

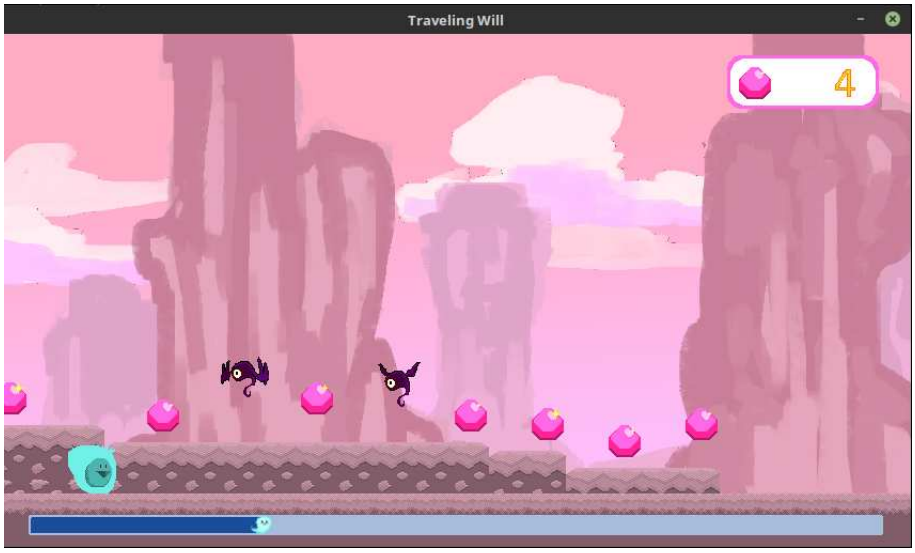


(a) Primeira fase original.



(b) Primeira fase portada.

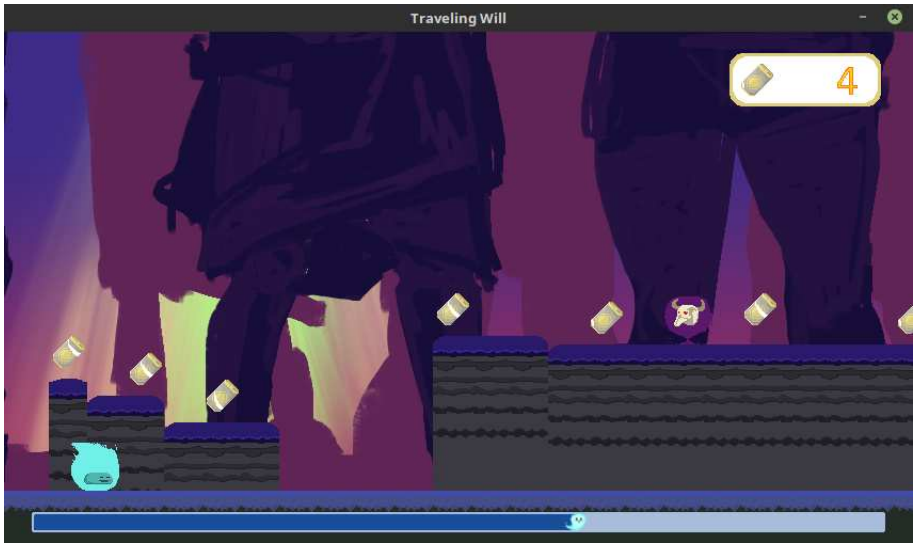
Figura 11 – Comparação da primeira fase.



(a) Segunda fase original.

(b) Segunda fase portada.

Figura 12 – Comparação da segunda fase.



(a) Terceira fase original.

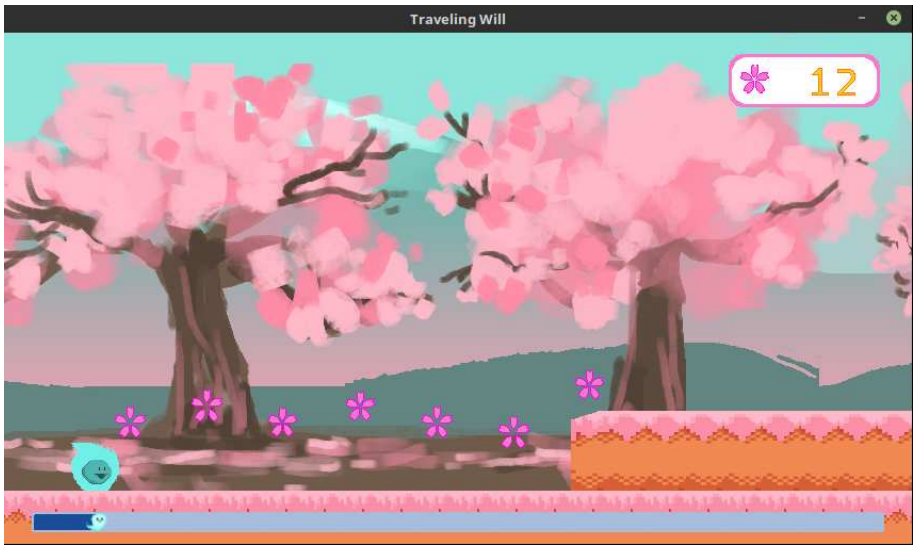
(b) Terceira fase portada.

Figura 13 – Comparação da terceira fase.

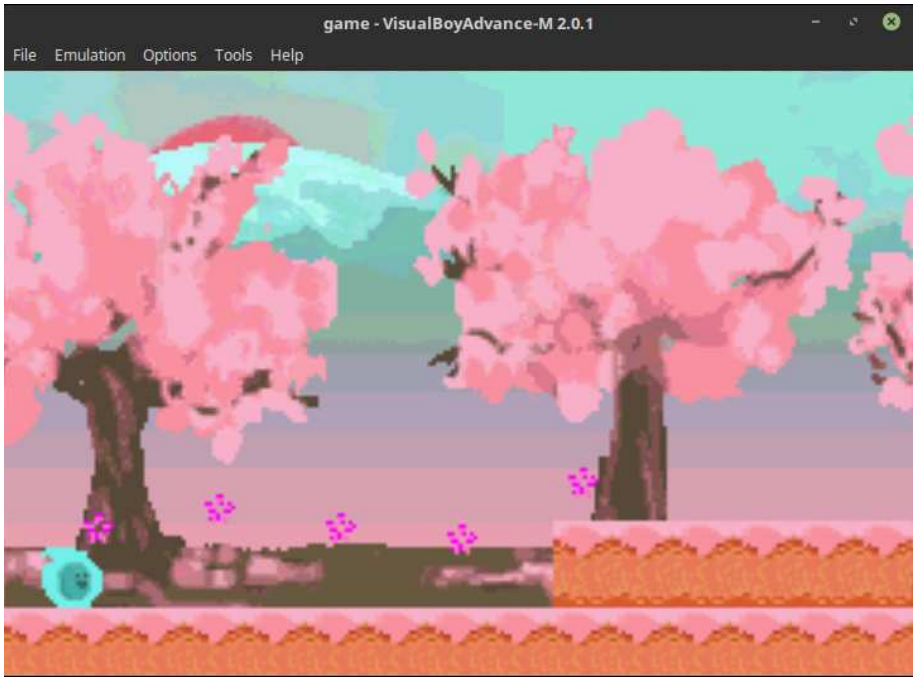
(a) Quarta fase original.

(b) Quarta fase portada.

Figura 14 – Comparação da quarta fase.

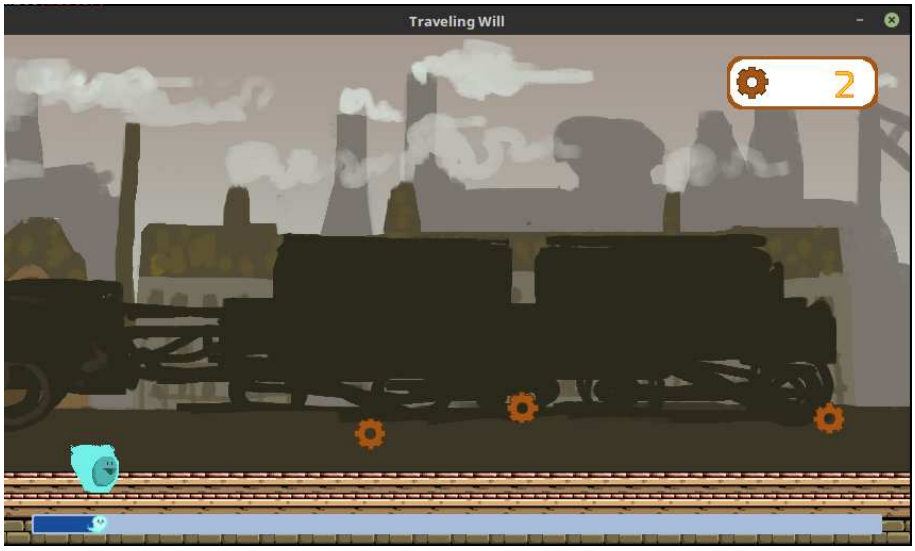


(a) Quinta fase original.

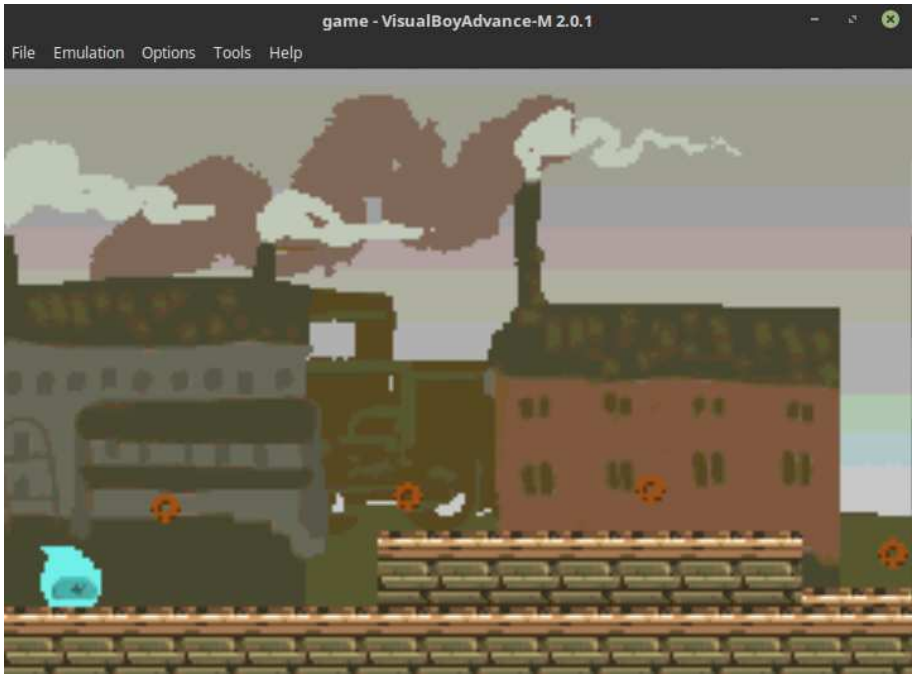


(b) Quinta fase portada.

Figura 15 – Comparação da quinta fase.



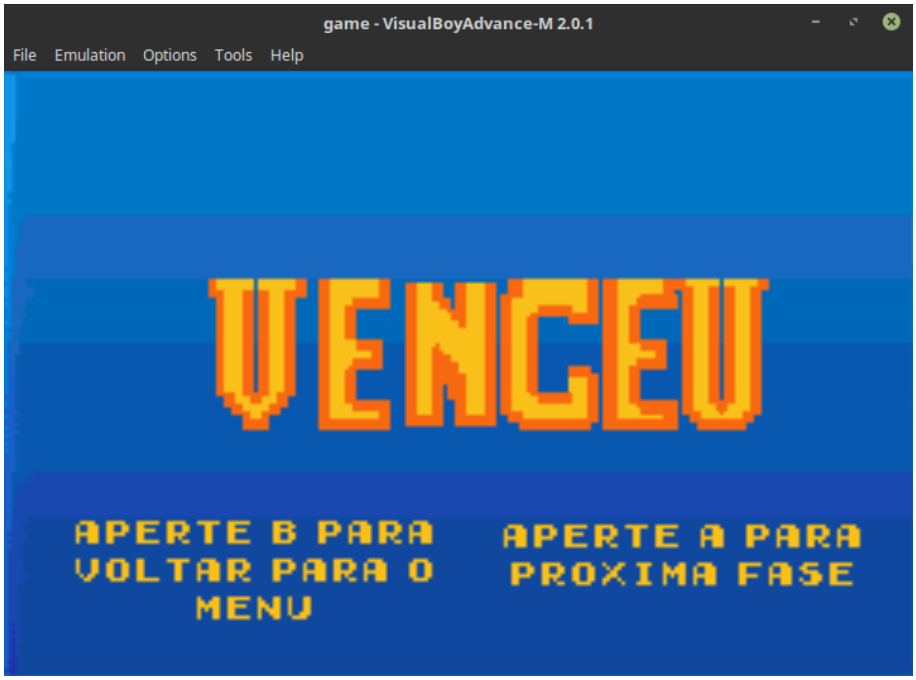
(a) Sexta fase original.



(b) Sexta fase portada.

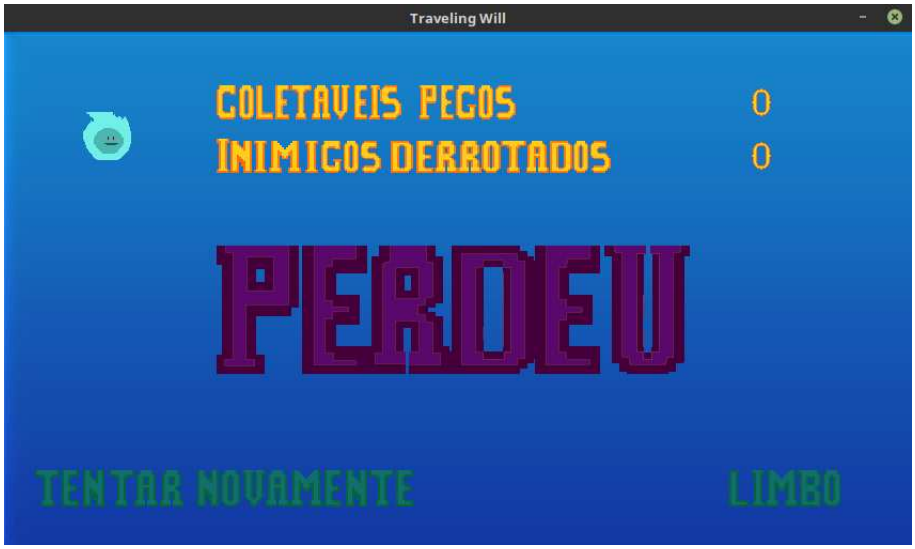
Figura 16 – Comparação da sexta fase.

(a) Menu de vitória original.



(b) Menu de vitória portado.

Figura 17 – Comparação do menu de vitória.



(a) Menu de derrota original.

(b) Menu de derrota portado.

Figura 18 – Comparação do menu de derrota.

4 Considerações finais

Durante o desenvolvimento da *engine* e do jogo, em diversos momentos bastante tempo foi empregado tentando entender detalhes da especificação do *hardware* do GBA, como a definição da paleta de cores dos *backgrounds* e *sprites* serem bastante diferentes, o *overlap* entre *charblocks* e *screenblocks* na região de memória VRAM, os diferentes canais de áudio que serviam para propósitos bem diferentes, a impossibilidade de se utilizar uma ferramenta de depuração de código, como `gdb`[1], entre outros. Esses impedimentos nos permitiram aprofundar nosso conhecimento em relação a como o *hardware* do GBA funciona e como desenvolvedores de jogos para plataformas mais antigas resolviam problemas difíceis.

Por fim, como resultado final do trabalho, a pergunta de pesquisa pôde ser respondida afirmativamente, significando que foi possível portar o jogo *Traveling Will*, desenvolvido originalmente para PC, para o *Nintendo Gameboy Advance*, no contexto de um trabalho de conclusão de curso, com performance e jogabilidade próximos da versão para computador.

4.1 Trabalhos futuros

Como sugestões de trabalhos futuros, têm-se:

- Melhoria do módulo de áudio para permitir carregar efeitos sonoros e pausar músicas durante a execução do jogo;
- Implementação do carregamento e utilização de fontes no jogo;
- Adição de elementos de HUD e seleção de fases;
- Possibilidade de salvar o estado do jogo em memória; e
- Implementação de um desfragmentador de memória na classe `MemoryManager`.

[1] *GNU Debugger*, disponível em <https://www.gnu.org/software/gdb/>

Referências

ARM. *A32 and T32 Instruction Sets*. 2018. Acesso em 03/03/2018. Disponível em: <https://developer.arm.com/products/architecture/instruction-sets/a32-and-t32-instruction-sets>. Citado na página 27.

CARREKER, D. *The game developer's dictionary : a multidisciplinary lexicon for professionals and students*. Boston, MA: Course Technology, 2012. ISBN 978-1-4354-6081-2. Citado na página 30.

CASTANHO, C. et al. *Gameplay: ensaios sobre estudo e desenvolvimento de jogos*. [S.l.: s.n.], 2016. ISBN 9789978551530. Citado na página 24.

DALE, N.; WEEMS, C. *Programming in C++*. Sudbury, Mass: Jones and Bartlett Publishers, 2004. ISBN 0763732346. Citado na página 30.

Entertainment Software Association. *Analyzing the american video game industry 2016*. [S.l.], 2017. Citado na página 21.

FRAKES, W. B.; FOX, C. J. Sixteen questions about software reuse. *Commun. ACM*, ACM, New York, NY, USA, v. 38, n. 6, p. 75–ff., jun. 1995. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/203241.203260>. Citado na página 30.

GREGORY, J. *Game engine architecture*. Boca Raton, FL: CRC Press, Taylor & Francis Group, 2014. ISBN 1466560010. Citado 2 vezes nas páginas 23 e 24.

HAPP, T. *CowBite Virtual Hardware Specifications: Unofficial documentation for the CowBite GBA emulator*. [S.l.], 2002. Citado 2 vezes nas páginas 26 e 29.

HARBOUR, J. *Programming the Nintendo Game Boy Advance: The Unofficial Guide*. [S.l.]: Course Technology PTR, 2003. ISBN 978-1931841788. Citado 3 vezes nas páginas 26, 28 e 29.

HORNA, M. A.; WAWRO, A. *What exactly goes into porting a video game? BlitWorks explains*. 2014. Acesso em 17/06/2018. Disponível em: <https://www.gamasutra.com/view/news/222363/What_exactly_goes_into_porting_a_video_game_BlitWorks_explains.php>. Citado 2 vezes nas páginas 30 e 31.

IUPPA, N. *End-to-end game development : creating independent serious games and simulations from start to finish*. Burlington, MA: Focal Press, 2010. ISBN 978-0-240-81179-6. Citado na página 21.

KORTH, M. *GBATEK: Gameboy Advance / Nintendo DS - Technical Info*. [S.l.], 2007. Citado 3 vezes nas páginas 26, 29 e 45.

KOSTER, R. *A theory of fun for game design*. Sebastopol, CA: O'Reilly Media Inc, 2014. ISBN 1449363210. Citado na página 23.

KUO, H.-H. *White noise distribution theory*. [S.l.]: CRC press, 1996. v. 5. Citado na página 30.

LEWIS, M.; JACOBSON, J. Game engines in scientific research - introduction. v. 45, p. 27–31, 01 2002. Citado na página 21.

NINTENDO. *Game Boy Advance*. 2018. Acesso em 03/03/2018. Disponível em: <https://www.nintendo.pt/A-empresa/Historia-da-Nintendo/Game-Boy-Advance/Game-Boy-Advance-627139.html>. Citado 2 vezes nas páginas 26 e 29.

PLUMMER, J. *A flexible and expandable architecture for computer games*. Dissertação (Mestrado) — Arizona State University, 12 2004. Citado na página 25.

ROLLINGS, A.; MORRIS, D. *Game architecture and design*. Indianapolis, Ind: New Riders, 2004. ISBN 0735713634. Citado 3 vezes nas páginas 23, 25 e 26.

TANENBAUM, A. *Sistemas operacionais modernos*. Rio de Janeiro (RJ: Prentice-Hall do Brasil, 2010. 70-72 p. ISBN 8576052377. Citado na página 44.

VIJN, J. *memcpy and memset replacements for GBA/NDS*. 2008. Acesso em 01/12/2018. Disponível em: <http://www.coranac.com/2008/01/tonccpy/>. Citado na página 49.

VIJN, J. *Tonc: GBA Programming in rot13*. 2013. Acesso em 03/03/2018. Disponível em: <http://www.coranac.com/tonc/text/toc.htm>. Citado 3 vezes nas páginas 28, 29 e 67.