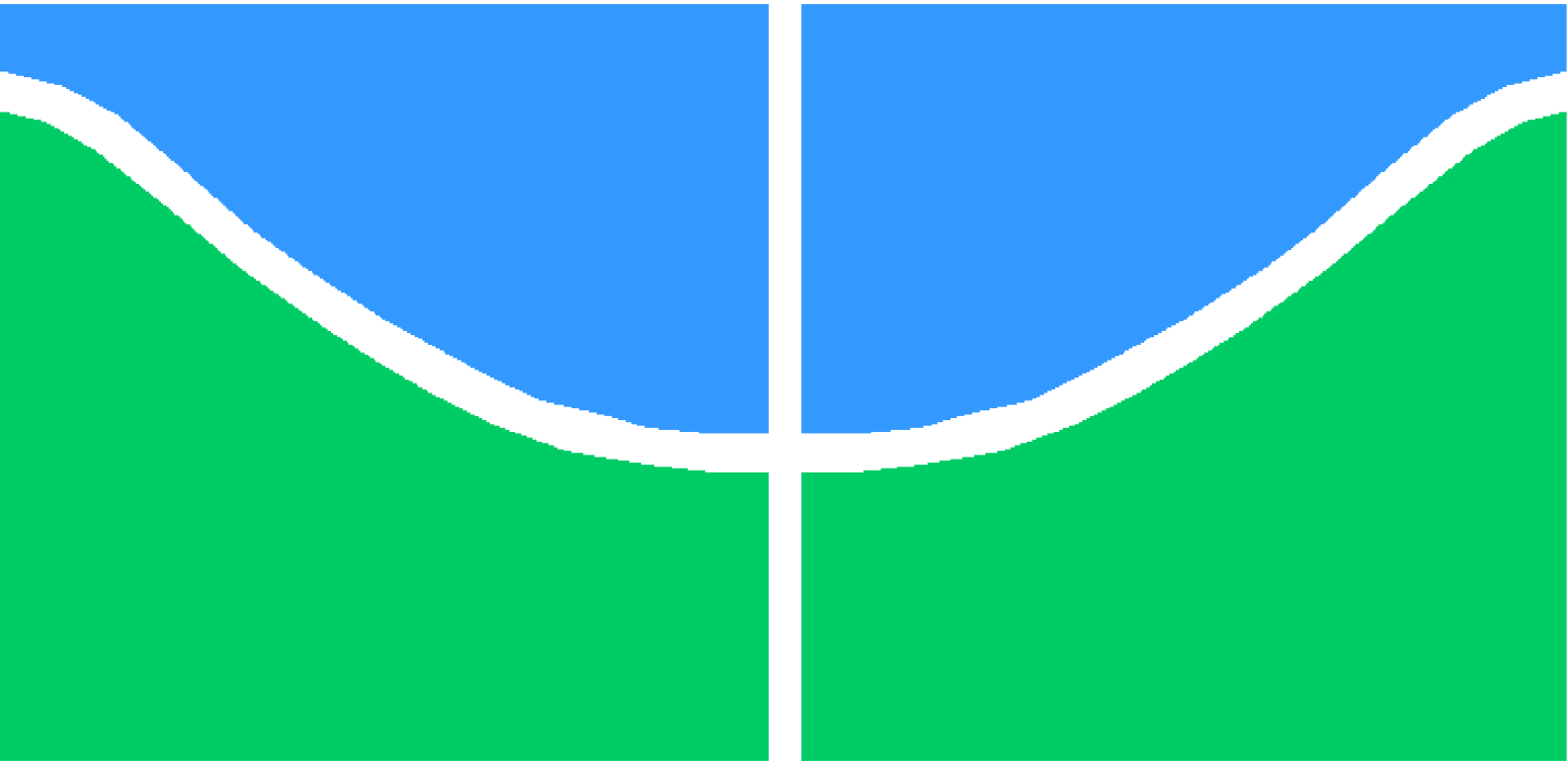

Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

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

Autores: Igor Ribeiro Barbosa Duarte e Vítor 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-
46 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 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.

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. A fim de lidar com esse problema, o objetivo deste trabalho é portar o jogo *Traveling Will*, desenvolvido originalmente para PC, para o *console* portátil *Nintendo Game Boy Advance*, em conjunto com o desenvolvimento de uma *game engine* que auxilie no processo de reescrita. Até o momento, foi implementada uma versão parcial da *engine* contendo os módulos de input e vídeo, assim como um protótipo do menu do jogo.

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. In order to deal with these issues, this work aims to port the game Traveling Will, originally developed for PC, to the portable console Nintendo Game Boy Advance, alongside with the development of a game engine, which will assist the rewriting process. Until the present date, a partial version of this engine, containing input and video modules, and a prototype of the game menu were implemented.

Key-words: *porting, electronic games, nintendo game boy advance, performance.*

Lista de ilustrações

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

Contextualização

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 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 (LEWIS; JACOBSON, 2002), 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.

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;
- Metodologia, onde serão definidos os procedimentos a serem realizados para o porte do jogo;
- Resultados, onde serão apresentados os resultados obtidos até o momento; e
- Considerações Finais do 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 próxima seção 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.

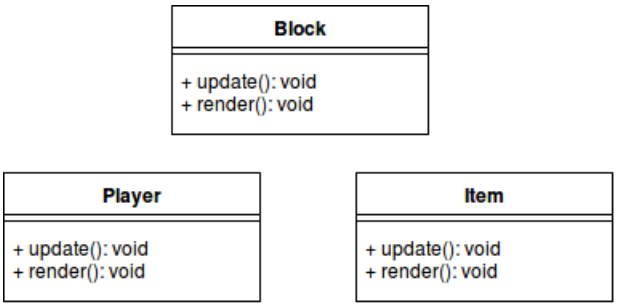

Figura 1 – Exemplo de arquitetura monolítica. Fonte: *Autores*.

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.

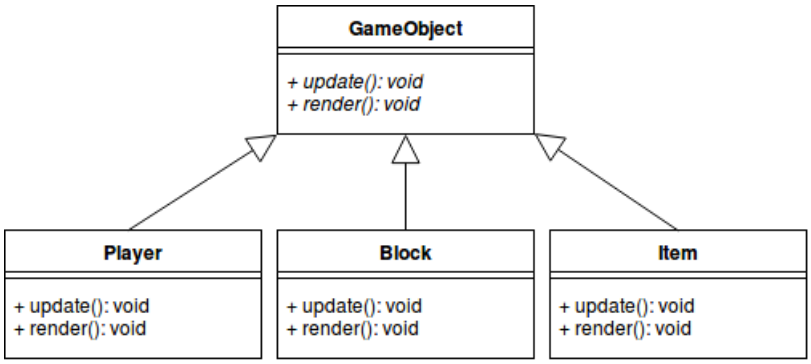

Figura 2 – Exemplo de arquitetura reutilizável. Fonte: *Autores*.

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).

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) (HARBOUR, 2003).

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, de acordo com o Korth (2007), 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 (HAPP, 2002).

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 (HAPP, 2002).

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 (HAPP, 2002).

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 (HAPP, 2002)

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 (HAPP, 2002).

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 (HAPP, 2002).

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 (HAPP, 2002). A *OBJ Attributes Memory* (OAM) possui 1 *KByte* de tamanho e é utilizada para controlar as *sprites* do GBA (HAPP, 2002). 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 (HAPP, 2002).

[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).

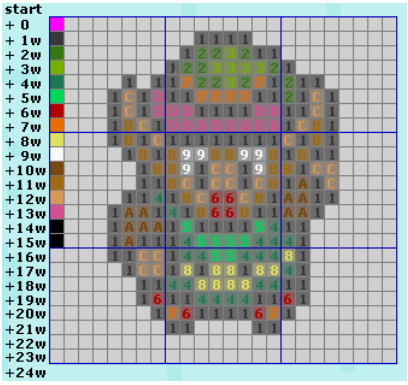

Figura 3 – Exemplo de *bitmap* 24×24. Fonte: (VIJN, 2013).

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)

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)

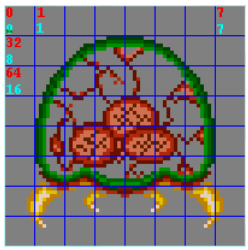

Figura 4 – Exemplo de *sprite* dividida em tiles. Fonte: (VIJN, 2013).

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).

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

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 (HARBOUR, 2003). 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* (KORTH, 2007).

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* (KORTH, 2007).

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 (KORTH, 2007).

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 (KORTH, 2007).

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 manutenível);

[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)

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 descrita nas próximas seções e está dividida em Ferramentas de Desenvolvimento e Metodologia de Desenvolvimento.

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 será 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, basicamente, do *kit* de desenvolvimento devkitARM.

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.

Para o ajuste das imagens do jogo para a resolução de tela do GBA, será 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/>

2.1.2 Ambiente de teste

2.1.2.1 Emulador

Para a realização de testes com os executáveis gerados pelo *devkitARM* está sendo utilizado um emulador de *Game Boy Advance*, chamado ***VisualBoyAdvance-M***[3].

2.1.2.2 *Console*

O *console* que está sendo utilizado como ambiente de testes real é um ***Nintendo DS***[4], que possui um *slot* para cartuchos de *Game Boy Advance*. Neste trabalho está sendo utilizado um cartucho especial onde é possível escrever arquivos executáveis diretamente nele.

Para a escrita dos arquivos executáveis neste cartucho é utilizado o dispositivo ***EZFlash II***[5]. 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***[6], 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* conterá uma classe que irá representar um objeto do jogo (do inglês, *game object*). Esta classe ficará responsável por conter o comportamento genérico de um objeto dentro do jogo (podendo este ser um personagem, uma plataforma, um item coletável, etc.). Ele será representado da seguinte maneira:

[3] *VisualBoyAdvance-M*, disponível em <https://github.com/visualboyadvance-m/visualboyadvance-m>
[4] *Nintendo DS*, disponível em <https://www.nintendo.com/consumer/systems/selectds.jsp>
[5] *EZ Flash II*, disponível em <http://www.ezfadvance.com/cards/EZ-Flash_2.htm>
[6] Sistema operacional da *Microsoft*, disponível em <https://support.microsoft.com/pt-br/help/14223/windows-xp-end-of-support>

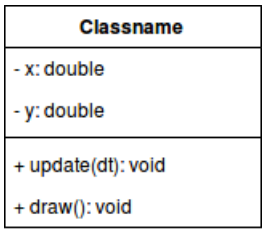

Figura 6 – Modelagem inicial da classe *GameObject*. Fonte: *Autores*.

Onde o método ***update()***, que é puramente virtual, trata de qualquer atualização do objeto a cada frame e o método ***draw()***, também puramente virtual, é responsável por renderizar o objeto na posição (x, y) a cada frame.

É importante frisar que os métodos ***update()*** e ***draw()*** 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* a serem implementados são: módulo de vídeo, módulo de áudio, módulo de física e módulo de input.

2.2.1.1 Módulo de vídeo

O módulo de vídeo será responsável por renderizar qualquer tipo de imagem, animação e texto existente. Ele conterá as seguintes funcionalidades:

- Renderizar uma imagem de fundo;
- Renderizar uma *sprite*;
- Renderizar uma animação (como uma série de *sprites*);
- Movimentar horizontalmente uma imagem de fundo (*horizontal scroll*);
- Renderizar textos com uma determinada cor;
- Remover uma imagem, *sprite*, texto ou animação que esteja renderizado da tela;
- Atualizar a renderização a cada frame, levando em consideração as posições x e y do *sprite*, animação ou texto.

Deve-se lembrar que as imagens e textos que serão renderizados já estarão ajustados para a resolução e formato de cores corretos do GBA.

2.2.1.2 Módulo de áudio

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

- Iniciar a execução de um efeito sonoro;
- Pausar a execução de um efeito sonoro;
- Parar a execução de um efeito sonoro;
- Iniciar a execução de uma música de fundo;
- Pausar a execução de uma música de fundo;
- Parar a execução de uma música de fundo.

2.2.1.3 Módulo de física

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

- Simular, opcionalmente, a ação da gravidade em objetos do jogo;
- Detectar, opcionalmente, colisões entre objetos do jogo.

2.2.1.4 Módulo de *input*

O módulo de *input* é responsável por receber qualquer pressionamento de qualquer um dos 10 botões e teclas do GBA.

2.2.2 Desenvolvimento do jogo

A implementação do jogo vai seguir o seguinte modelo de classes:

2.2.2.1 Objetos do jogo

O personagem principal, os itens coletáveis, plataformas, portais (para acesso e saída das fases) e elementos de HUD (*heads-up display*) serão representados como *game objects*.

Abaixo se encontra a modelagem inicial dos objetos do jogo.

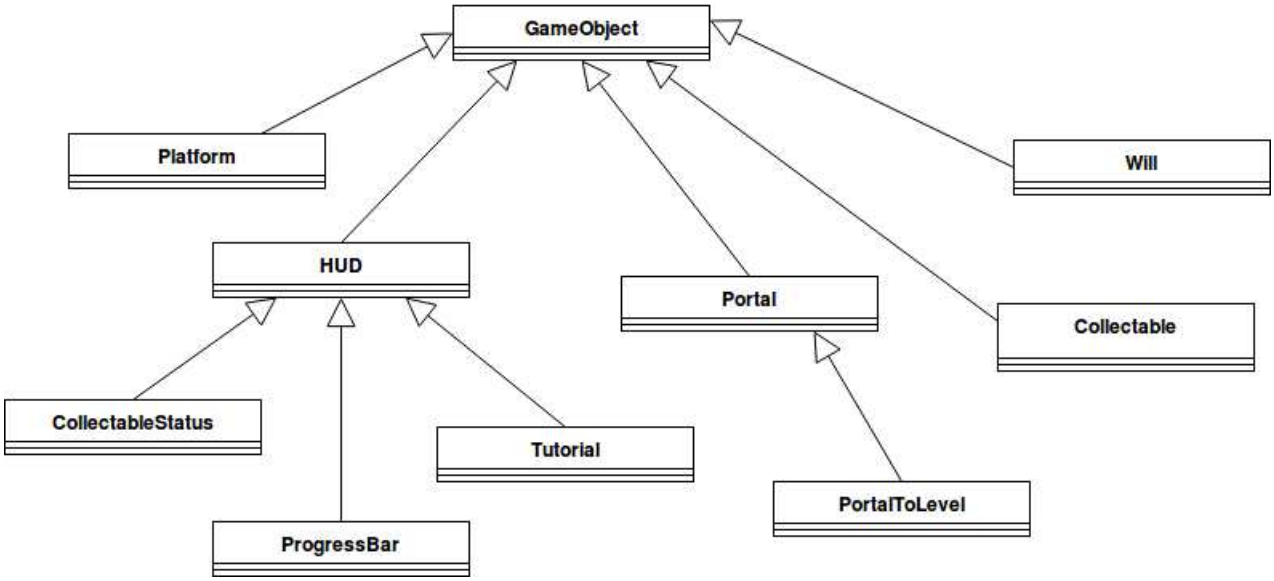

Figura 7 – Modelagem inicial dos objetos do jogo. Fonte: *Autores*.

2.2.2.2 Níveis

A classe *Level* contém a generalização de um nível no jogo. No jogo, os menus (principal e de seleção de fases), *cutscenes*, fases e telas de vitória e derrota são representados como níveis do jogo.

Abaixo se encontra a modelagem inicial dessas classes.

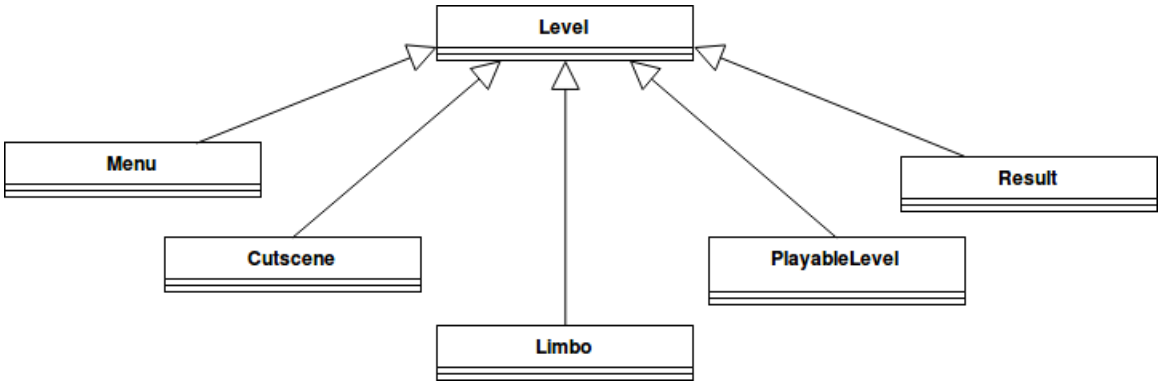

Figura 8 – Modelagem inicial dos níveis do jogo. Fonte: *Autores*.

2.2.2.3 Ajuste de recursos do jogo

Os recursos como imagens, áudios e arquivos de fonte do jogo precisarão ser editados para que possam ser carregados em memória e utilizados no GBA. No caso das imagens, por exemplo, serão 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.

2.3 Cronograma de desenvolvimento

Abaixo se encontra o cronograma de desenvolvimento das tarefas planejadas para a conclusão do trabalho.

TAREFAS	JULHO	AGOSTO	SETEMBRO	OUTUBRO	NOVEMBRO	DEZEMBRO
Finalizar desenvolvimento do módulo de input	X	X				
Finalizar desenvolvimento do módulo de vídeo	X	X	X			
Desenvolver módulo de áudio	X	X	X			
Desenvolver módulo de física	X	X	X			
Finalizar implementação do menu		X	X			
Implementar rolagem infinita do background dos níveis			X	X		
Implementar mecanismo de renderização das plataformas			X	X		
Implementar movimentos do personagem			X	X		
Implementar mecanismo de renderização dos coletáveis			X	X		
Implementar telas de finalização do nível				X		
Carregar níveis a partir do level design				X	X	
Implementar seletor de fases (Limbo)				X	X	
Implementar opções do menu					X	X
Implementar tutorial					X	X

Figura 9 – Cronograma de desenvolvimento. Fonte: *Autores*.

3 Resultados Parciais

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 2.1.2.2 do capítulo 2). Para isso, a principal ferramenta utilizada foi a *libtonc*[1], que nessa versão inicial fez o papel de engine do jogo.

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

Figura 10 – Jogo original sendo executado em um PC. Fonte: *Autores*.

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

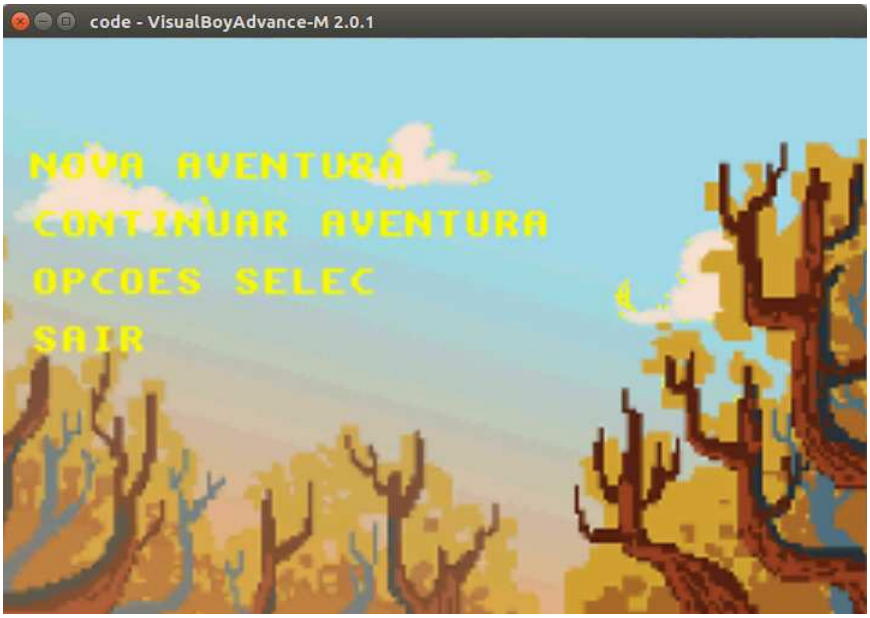

Figura 11 – Protótipo sendo executado no emulador de GBA. Fonte: *Autores*.

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

3.2 Desenvolvimento da *engine*

Após a finalização do protótipo inicial, foi iniciado o desenvolvimento da *engine* que irá substituir a *libtonc* na versão final do jogo. Ela irá padronizar a utilização dos recursos providos pelo GBA e irá conter um módulo de vídeo, áudio, *input*, física, dentre outros.

Até o momento, o módulo de *input* e parte do módulo de vídeo foram implementados.

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* no valor guardado por esse registrador. Sempre que um botão é apertado, 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 expotente o *bit* em questão

(4, nesse exemplo). Abaixo é possível vizualizar no código do módulo de *input* a definição das constantes que representam os botões, assim como a função utilizada para checar o estado de cada um deles:

```
#ifndef INPUT_H
#define INPUT_H

#include <stdbool.h>
#include "base_types.h"

#define BUTTON_A 1
#define BUTTON_B 2
#define BUTTON_SELECT 4
#define BUTTON_START 8
#define BUTTON_RIGHT 16
#define BUTTON_LEFT 32
#define BUTTON_UP 64
#define BUTTON_DOWN 128
#define BUTTON_R 256
#define BUTTON_L 512

#define N_BUTTON 10

int pressed_state[N_BUTTON];

void check_buttons_states();
bool pressed(int button);

#endif
```

Cabeçalho do módulo de input. Fonte: *Autores*.

```
#include "input.h"

volatile unsigned int *buttons_mem = (volatile unsigned int *)0x04000130;

```

```
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 }
```

Código fonte do módulo de input. Fonte: *Autores*.

Abaixo é possível visualizar um teste implementado para checar o pressionamento dos botões do GBA. Para cada botão pressionado um *pixel* vermelho aparece na tela. Na imagem abaixo, os botões B, R, *LEFT*, *RIGHT* e *START* estão sendo pressionados simultaneamente.

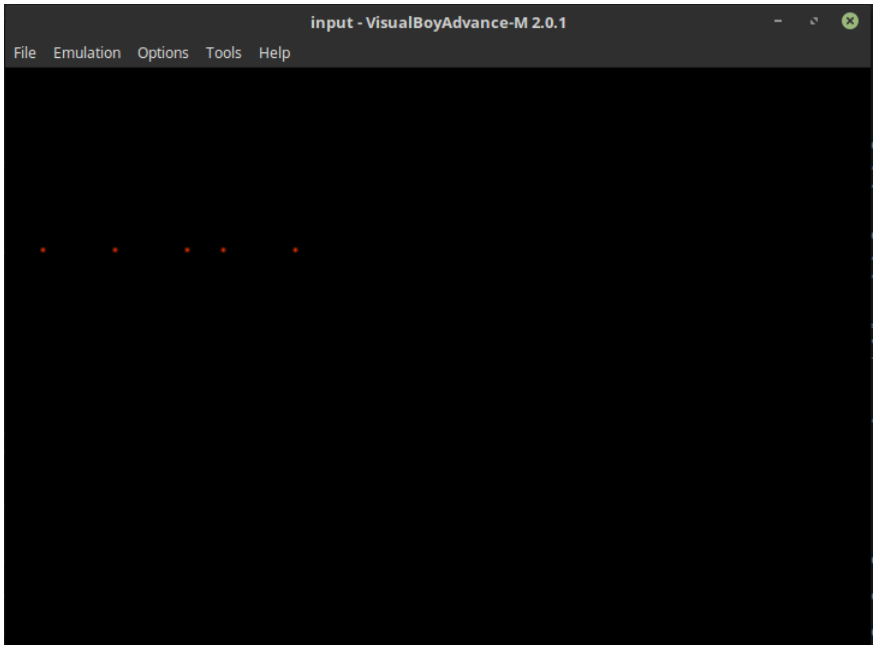

Figura 12 – Teste de pressionamento de botões no emulador. Fonte: *Autores*.

Abaixo se encontra o código fonte escrito para a realização deste teste:

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

```
3
4  #define RED 0x0000FF
5
6  unsigned short *vid_mem = (unsigned short *)0x6000000;
7
8  int main() {
9      reset_dispcnt();
10     set_video_mode(3);
11     set_background_number(2);
12
13     while(1) {
14         check_buttons_states();
15
16         for(int i=0;i<=9;i++){
17             if (pressed(i)) {
18                 vid_mem[50 * 240 + i * 10] = RED;
19             } else {
20                 vid_mem[50 * 240 + i * 10] = 0;
21             }
22         }
23     }
24
25     return 0;
26 }
```

Código fonte do teste de *input*. Fonte: *Autores*.

3.2.2 Módulo de vídeo

O módulo de vídeo deve possuir funções para controle do modo de vídeo, dos *backgrounds* e da renderização das *sprites*. Atualmente o módulo de vídeo desenvolvido trata apenas, de forma limitada, do modo de vídeo e do *background*. Ele permite a renderização de um determinado *background* desde que sejam passados a paleta de cores utilizada, um vetor com os *tiles* utilizados no *background* e um vetor representando o mapa de *tiles* a ser utilizado para renderizar o *background*. O módulo copia cada uma das informações passadas para as regiões de memória apropriadas. Atualmente, o módulo sempre utiliza o *background* 2 e parte do princípio de que o modo utilizado não é baseado em *bitmap*, não permitindo, portanto, o uso de múltiplos *backgrounds*. Além disso, o módulo ainda não

permite definir a posição exata de memória aonde o vetor de *tiles* e o *tilemap* deverão ser escritos, o que inviabiliza o uso de diferentes *tilemaps* e a renderização de fontes de forma correta. Abaixo é possível visualizar a versão atual do código fonte do módulo de vídeo:

```
1  #include "video.h"
2
3  #include <string.h>
4
5  void reset_dispcnt() {
6    REG_DISPCNT = 0;
7  }
8
9  void set_video_mode(int video_mode) {
10   REG_DISPCNT |= video_mode;
11 }
12
13 void set_background_number(int background) {
14   switch (background) {
15     case 0:
16       REG_DISPCNT |= 0x0100;
17     case 1:
18       REG_DISPCNT |= 0x0200;
19     case 2:
20       REG_DISPCNT |= 0x0400;
21     case 3:
22       REG_DISPCNT |= 0x0800;
23   }
24 }
25
26 void set_background(const void *pal, int pal_len, const void *tiles, int tiles_len, con
27   // Load palette
28   memcpy(pal_bg_mem, pal, pal_len);
29   // Load tiles into CBB 0
30   memcpy(&tile_mem[0][0], tiles, tiles_len);
31   // Load map into SBB 31
32   memcpy(&se_mem[31][0], map, map_len);
33
34   REG_BG2CNT = BG_CBB(0) | BG_SBB(31) | BG_8BPP | BG_REG_64x32;
35 }
```

Código fonte do módulo de vídeo. Fonte: *Autores*.

Abaixo é possível visualizar o teste realizado com o módulo de vídeo da nova *engine*.

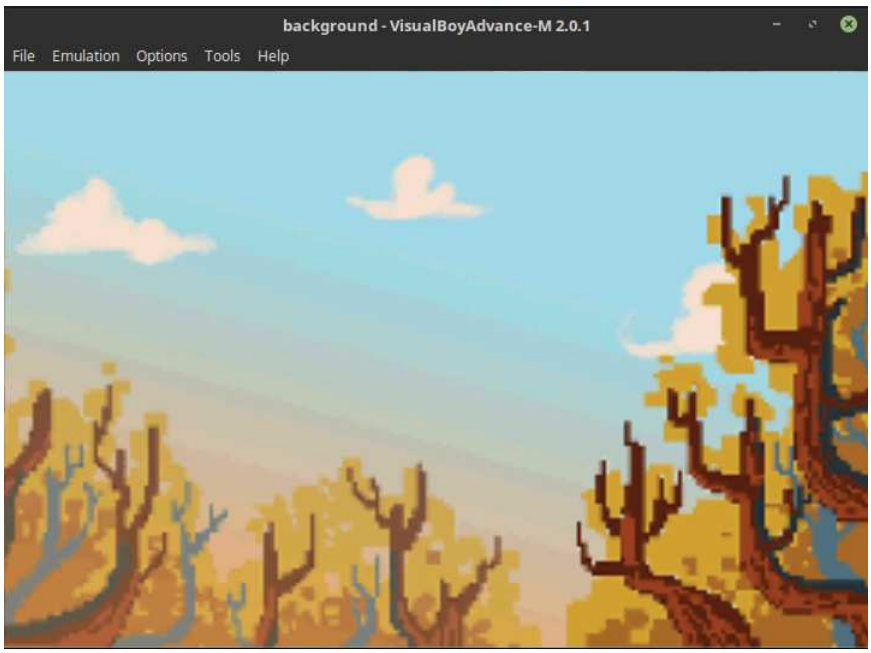

Figura 13 – Demonstração do módulo de vídeo. Fonte: *Autores*.

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

4 Considerações finais

O trabalho desenvolvido até aqui conta com uma *engine* contendo versões parciais do módulo de áudio e vídeo e um protótipo da tela de menu do jogo *Traveling Will* já funcionando em um *Game Boy Advance*. Entre as próximas tarefas estão: a conversão dos recursos do jogo, como imagens e áudios para um formato que possa ser utilizado no GBA, aprimoramento da *engine* para que ela permita trabalhar com todos os modos de vídeo presentes no GBA, utilizando a quantidade de *backgrounds* permitida em cada modo de forma simultânea, implementação dos módulos de física e áudio e, por fim, a reescrita do jogo utilizando a *engine* desenvolvida.

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 23.

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 26.

CASTANHO, C. et al. *Gameplay: ensaios sobre estudo e desenvolvimento de jogos*. [S.l.: s.n.], 2016. ISBN 9789978551530. Citado na página 20.

DALE, N.; WEEMS, C. *Programming in C++*. Sudbury, Mass: Jones and Bartlett Publishers, 2004. ISBN 0763732346. Citado na página 26.

Entertainment Software Association. *Analyzing the american video game industry 2016*. [S.l.], 2017. Citado na página 17.

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 26.

GREGORY, J. *Game engine architecture*. Boca Raton, FL: CRC Press, Taylor & Francis Group, 2014. ISBN 1466560010. Citado 2 vezes nas páginas 19 e 20.

HAPP, T. *CowBite Virtual Hardware Specifications: Unofficial documentation for the CowBite GBA emulator*. [S.l.], 2002. Citado 3 vezes nas páginas 22, 23 e 24.

HARBOUR, J. *Programming the Nintendo Game Boy Advance: The Unofficial Guide*. [S.l.]: Course Technology PTR, 2003. ISBN 978-1931841788. Citado 4 vezes nas páginas 22, 23, 24 e 25.

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 26 e 27.

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 17.

KORTH, M. *GBATEK: Gameboy Advance / Nintendo DS - Technical Info*. [S.l.], 2007. Citado 3 vezes nas páginas 22, 24 e 25.

KOSTER, R. *A theory of fun for game design*. Sebastopol, CA: O'Reilly Media Inc, 2014. ISBN 1449363210. Citado na página 19.

KUO, H.-H. *White noise distribution theory*. [S.l.]: CRC press, 1996. v. 5. Citado na página 25.

LEWIS, M.; JACOBSON, J. Game engines in scientific research - introduction. v. 45, p. 27–31, 01 2002. Citado na página 17.

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 22 e 25.

PLUMMER, J. *A flexible and expandable architecture for computer games*. Dissertação (Mestrado) — Arizona State University, 12 2004. Citado na página 21.

ROLLINGS, A.; MORRIS, D. *Game architecture and design*. Indianapolis, Ind: New Riders, 2004. ISBN 0735713634. Citado 3 vezes nas páginas 19, 21 e 22.

VIJN, J. *Tonc: GBA Programming in rot13*. 2013. Acesso em 03/03/2018. Disponível em: <http://www.coranac.com/tonc/text/toc.htm>. Citado 2 vezes nas páginas 23 e 24.