UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

THIAGO AULER DOS SANTOS

Biblioteca para Desenvolvimento de Jogos para o Nintendo DS

Trabalho de Graduação.

Prof. Dr. Marcelo de Oliveira Johann
Orientador

Porto Alegre, novembro de 2009.

AGRADECIMENTOS

Agradeço primeiramente ao Pai Maior, por todos os desafios vencidos e graças alcançadas; à minha família que sempre soube me apoiar em todos os momentos, tenham sido eles fáceis ou difíceis; e à minha princesa, por sua compreensão e carinho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

CD	Compact Disk
CGI	Computer-Generated Imaginary
CPU	Central Processing Unit
DS	Developers' System ou Dual Screen
ESRB	Entertainment Software Rating Board
FPS	Frames per Second
GBA	GameBoy Advance
GPU	Graphical Processing Unit
HD	Hard Disk
LCD	Liquid Crystal Display
LED	Light Emitting Diode
MMORPG	Multi Massive Online Role Playing Game
NDS	Nintendo DS
NES	Nintendo Entertainment System
OAM	Object Attribute Memory
PC	Personal Computer
PSP	PlayStation Portable
RAM	Random Access Memory
ROM	Read Only Memory
SDHC	Secure Digital High Capacity
SDK	Software Development Kit
SNES	Super Nintendo Entertainment System
VRAM	Video RAM

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Nas últimas décadas, percebe-se um avanço considerável na temática de jogos eletrônicos. Mesmo depois de enfrentar algumas crises, hoje este é um dos setores mais produtivos e rentáveis no mundo. Junto a isso, as atenções estão voltadas aos consoles portáteis, devido à sua mobilidade e qualidade em desempenho satisfatória.

Este trabalho se enfoca no desenvolvimento de software para o *Nintendo DS*, mostrando suas principais virtudes e apontando suas limitações. Também propõe uma implementação de uma biblioteca que objetiva facilitar o desenvolvimento de jogos voltados para o *Nintendo DS*.

A partir deste quesito, a biblioteca *FLib* foi criada. Sendo ela orientada a objetos e tendo uma arquitetura simples e concisa, esconde do desenvolvedor todas as preocupações com detalhamentos de *hardware*, permitindo que jogos 2D sejam gerados de forma totalmente descomplicada.

Testes realizados demonstraram que a *FLib* além de consumir o mínimo de memória ainda possui um desempenho ótimo, tornando-a ideal para o uso. Também são comparadas uma pequena aplicação que não utiliza a *FLib* e outra que a utiliza, mostrando em números as vantagens que a mesma proporciona.

Apesar das várias dificuldades encontradas, de a *FLib* não possuir um escopo mais abrangente, e de também estar longe de ser a solução definitiva para a criação de jogos, ainda sim, evita garantidamente muita retrabalho para o desenvolvedor.

Palavras-Chave: biblioteca, Nintendo, DS, desenvolvimento, jogos, portáteis

Library for Nintendo DS Game Development

ABSTRACT

In the latter decades, it's perceivable a considerable advance on electronic games thematic. Even after some declines, now this is the most productive and rentable sector of industry in the world. Along with that, the spotlight goes to the portable consoles, due to their mobility and satisfactory quality.

This work focuses on software development for Nintendo DS, showing its mains virtues and pointing its limitations. Also propose an implementation of a library, whose objective is to facilitate the game development for Nintendo DS.

After this requirement, the FLib library was created. Being object-oriented and having a simple and concise architecture, it hides all the details concerning the hardware from developer, letting 2D games being generated in a totally uncomplicated way.

Performed tests demonstrate that FLib consumes low memory and has an optimal performance, being ideal to the use. Also, an application using Flib is compared with another that doesn't use it, showing in numbers all the advantages that FLib offers.

In spite of all difficulties, FLib's straight scope and not intending to be a definitive solution in game development, FLib can make a game development a lot easier.

Keywords: library, Nintendo, DS, development, games, portables

1 INTRODUÇÃO

Nas últimas décadas, percebe-se um avanço considerável na temática de jogos eletrônicos, que começou como brincadeira e hoje é um dos setores mais produtivos e rentáveis no mundo.

O desenvolvimento de jogos inclui praticamente quase todos os ramos de pesquisa da computação: processamento de imagens, processamento de sons, inteligência artificial, otimização, etc. Tornando-a assim, uma das áreas mais fartas em conhecimentos.

Este trabalho tem como objetivo estudar o *hardware* do *Nintendo DS* e implementar uma biblioteca para desenvolvimento de jogos para esse sistema. Concebido nos primeiros anos do século XXI, o *Nintendo DS* é um *handheld* que inova no quesito interatividade, e este trabalho busca explorar essas funcionalidades.

Nos dois primeiros capítulos, é apanhado o histórico do videogame mostrando as principais características e tecnologias de cada época e tentando mostrar uma tendência para as gerações vindouras.

O terceiro capítulo busca mostrar todas as característica e minúcias do *hardware* do *Nintendo DS*. Também evidencia os aspectos mais relevantes para o desenvolvimento de jogos para esta plataforma.

Por fim, os últimos capítulos propõem a implementação de uma biblioteca baseada em orientação a objetos, que por sua vez, almeja facilitar o desenvolvimento de jogos para o *Nintendo DS*. A biblioteca a ser discutida chama-se *FLib* e objetiva basicamente o domínio dos jogos 2D. Em seguida, uma discussão sobre o desempenho e limitação da *FLib*, bem como, as principais vantagens em seu uso.

2 HISTÓRIA DO VIDEOGAME

Ao longo dos muitos anos de existência dos consoles de videogames, uma classificação foi concebida para que melhor se entendesse sua evolução. Tendo a primeira geração iniciada em 1972 e atualmente estando em sua sétima geração, sua história é remontada muitos anos antes.

Apesar de muitas controvérsias, várias pessoas defendem que o início do videogame se deu no dia 18 de outubro de 1958 quando "*Tennis for Two"* (figura 2.1), um jogo de tênis desenvolvido em um pequeno computador analógico que gerava curvas num tubo de raios catódicos de um osciloscópio, foi pela primeira vez exibido (GETTLER). E desde então, o videogame é um setor da indústria que não para de crescer.

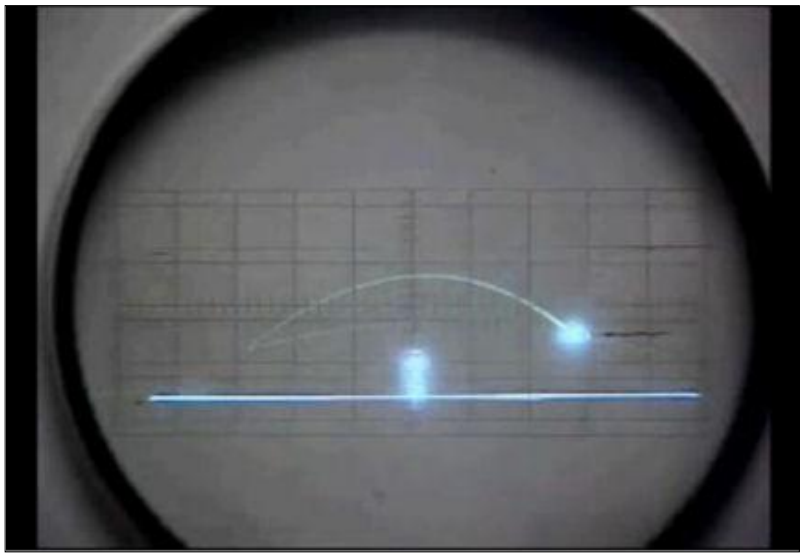

Figura 2.1: *Tennis for Two* na tela de um osciloscópio

2.1 Primeira Geração

A primeira geração teve seu início na década de 70 quando o primeiro *console* plugável à televisão, o *Magnavox Odyssey*, foi lançado. O *Odyssey* foi uma concretização de um projeto iniciado em 1966, conhecido como "*Brown Box*", que fora desenvolvido pelo engenheiro Ralph Baer (BAER, 2005).

O *Odyssey* e seus poucos similares, na época, foram os únicos que representaram a primeira geração até o ano de 1977. A característica mais marcante num *console* dessa geração era a ausência de microprocessador, ou seja, cada elemento do jogo era composto por circuitos lógicos discretos.

Nesta mesma época, surge *Pong* (figura 2.2), um jogo no estilo pingue-pongue, e que acaba se tornando a primeira máquina *arcade* popular. As máquinas *arcade* começam a aparecer como uma forma barata de diversão, pois o usuário jogava uma ou mais partidas apenas inserindo moedas nelas.

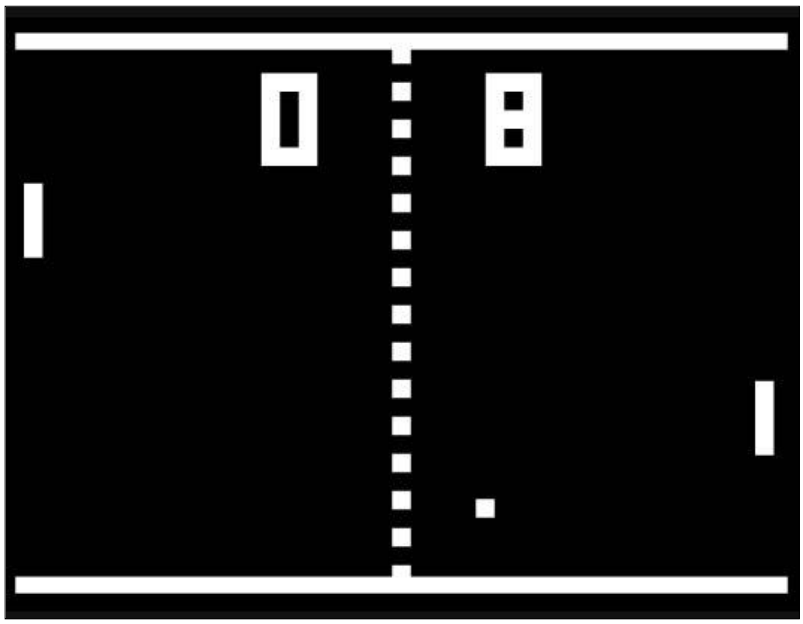

Figura 2.2: *Pong*, o primeiro *arcade* de sucesso

2.2 Segunda Geração

A segunda geração inicia em 1976 com uma idéia marcante, *Fairchild Channel F* é lançado, sendo o primeiro *console* a possuir um microprocessador e permitir o uso de cartuchos que continham ROMs (WIKIPEDIA). Os cartuchos possuíam então, o código compilado de um jogo arbitrário, o que dava uma liberdade sem precedentes aos programadores da época.

Um ano depois, a *Atari* lança o seu console baseado em CPU: o *Atari 2600*. Ao mesmo tempo, ela se prepara e põe a venda nove jogos, o que acaba por tornar o *Atari 2600* o videogame mais popular da segunda geração (WIKIPEDIA).

É nessa época, que surgem os jogos lendários como *Space Invaders* (1978), *Asteroids* (1979) e *Pac Man* (1980). Em compensação, o ano de 1983 é marcado pelo fim da segunda geração devido à baixíssima qualidade dos jogos lançados ficando conhecido como o ano do *crash* dos videogames (RETROSPACE).

2.3 Terceira Geração

Em julho de 1983, dá-se o início da terceira geração com o lançamento do *console* da *Nintendo*, o *Famicon*; e após alguns meses, o *Famicon* é lançado no mercado americano sob o nome de NES (*Nintendo Entertainment System*).

A *Sega*, empresa conhecida por seus jogos em *arcade*, entra para o mercado de *consoles* de terceira geração com o *Master System*, originando assim a saga de rivalidade entre *Nintendo* e *Sega*.

Apesar do fato dos processadores dos consoles de terceira geração também serem de 8-*bits* como os da segunda geração, é importante salientar que foi deste ponto em diante que a contagens de *bits* do console começou a ter relevância.

As principais contribuições desta geração foram a padronização dos gêneros de jogos e a criação das franquias mais importantes do mundo dos videogames. Destas franquias destaca-se "*Sonic, the Hedgehog*", "*Super Mario Bros*.", "*Final Fantasy*" e "*The Legend of Zelda*", sendo que estas continuam em expansão e em pleno vigor até os dias atuais.

2.4 Quarta Geração

Desta vez, quem saiu na frente foi a *Sega*, com o *Mega Drive* e seu processador 16-*bits*, mas não muito depois, a *Nintendo* lança o SNES (*Super Nintendo Entertainment System*). A disputa entre as duas gigantes culminava com contratos de exclusividade com as produtoras de jogos, assim, podiam reservar franquias de modo a tentar vencer a concorrência.

Durante esta geração, na década de 1990, surgiram jogos que até hoje causam polêmica. É o caso de *Doom* e *Mortal Kombat* (figura 2.3) que estão entre os responsáveis pela criação da ESRB (*Entertainment Software Rating Board*) devido a cenas extremas de violência e sangue (WIKIPEDIA).

Figura 2.3: *Mortal Kombat* e violência explícita

Com o crescente aumento de jogos para *consoles* e jogos para computadores pessoais, as máquinas de *arcade* começam a perder espaço no mercado. Mas é aproveitando esta deixa, que a SNK adapta o *hardware* de suas máquinas *arcade* para seu console conhecido como *NeoGeo*.

2.5 Quinta Geração

Na quinta geração, destacam-se dois fatores importantes: inovação com jogos totalmente em 3D (exemplo pode ser visualizado na figura 2.4) e a transição da mídia de cartuchos para CD-ROMs. Os principais destaques dessa época foram o *Sega Saturn*, o *Nintendo 64* e o *PlayStation* da *Sony*. A *Atari* também tentou uma investida, mas fracassou com o lançamento do *Atari Jaguar*, saindo de uma vez por todas do mercado de consoles de videogame.

A principal batalha desta época foi o cartucho contra o CD. Um era mais eficiente, rápido e mais seguro contra pirataria. O outro sofria com os tempos de carga dos jogos, mas podia armazenar muitas vezes mais informação, o que possibilitou a introdução de filmes CGI (*Computer-Generated Imaginary*) dentro dos jogos.

Figura 2.4: *Tomb Raider*, totalmente modelado em 3D

2.6 Sexta Geração

Em novembro de 1998, a *Sega* se antecipa e lança o primeiro console de sexta geração, o *Dreamcast*. Infelizmente, devido às baixas vendas e ao lançamento do *PlayStation 2*, a *Sega* sai do ramo de *consoles* e se dedica apenas à criação de jogos.

A *Nintendo* se rende ao disco óptico e cria o *GameCube*, com seu novo *console*, obteve um relativo sucesso.

Nesse ponto, a *Microsoft* decide entrar no ramo de *consoles* e projeta seu exclusivo videogame, o *Xbox*. O *Xbox* mais parecia um PC comum colocado dentro de uma caixa preta, pois continha um *Pentium III*, uma placa de vídeo da *NVIDIA*, 64MB de RAM, um HD de 8 ou 10 GB e ainda por cima rodava um sistema operacional baseado no *kernel* do *Windows 2000*.

A qualidade gráfica subiu muito comparada com a geração anterior, mas o ponto determinante desta geração foi a conectividade. Tomada pela onda os MMORPGs (*Multi Massive Online Role Playing Game*), essa geração inova pela conectividade com a *Internet* e a possibilidade de se jogar *multiplayer* com qualquer pessoa do planeta que também esteja conectada.

Nos últimos suspiros desta geração, o *PlayStation 2* continua sendo o *console* mais vendido de toda a história dos videogames, com seus mais de 138 milhões de unidades vendidas desde 2000 (WIKIPEDIA). É também deste ponto em diante, que o PC deixa de ser o centro das atenções no mundo dos jogos, que pode ser percebido através dos apenas 14% de participação dos jogos de PC na indústria de *games* (BRIGHTMAN, 2009).

2.7 Sétima Geração

A sétima geração dos videogames iniciou no final do ano de 2004 e contempla os *consoles* que atualmente se digladiam. A primeira empresa a lançar seu *console* no

mercado foi a *Microsoft* com o *Xbox 360*, e um ano depois é seguida da *Sony* com o *PlayStation 3*, que a propósito, foi o *console* com preço de lançamento mais elevado da história (WIKIPEDIA). Poucos dias depois, a *Nintendo* coloca no mercado o *Wii*, um *console* que inova no quesito interação utilizando o *Wiimote*, uma espécie de controle remoto capaz de interpretar os movimentos do jogador.

Apesar das negativas críticas que o *Wii* vinha recebendo antes do seu lançamento, ele surpreende a todos sendo o console com a vendagem mais rápida até hoje vista (WIKIPEDIA). Parte desse sucesso se dá também devido a possibilidade de se jogar jogos lançados anteriormente, inclusive jogos para consoles concorrentes como *Mega Drive* e *NeoGeo*. Esse recurso é conhecido como *Virtual Console* e os jogos são comprados através do *Wii Shop Channel*; em apenas um ano foram comprados mais de 10 milhões de títulos para o *Virtual Console* (WIKIPEDIA).

Para o final do ano de 2010, a *Microsoft* planeja um dispositivo que atualmente é chamado pelo codinome "*Project Natal*". Esse dispositivo promete ter funcionalidades avançadas como captura total de movimentos, reconhecimento facial e de voz, o que pretende uma maior imersão do jogador.

2.8 Mercado

É inegável que a indústria de jogos eletrônicos vem crescendo desde a sua concepção. Mesmo que tenham ocorridas algumas depressões na indústria, o impacto sofrido sempre foi superado em poucos anos.

A disseminação dos jogos eletrônicos se deu basicamente em meados da década de 90, este foi o ponto em que o *boom* começou. E esse estouro trouxe suas conseqüências para o novo século de uma maneira muito marcante.

Foi no ano de 2003, que o rendimento obtido pela indústria de jogos eletrônicos (levando-se em conta todas as plataformas de videogame juntas) excedeu a marca de onze bilhões de dólares. Esse número mostra um fato interessante, a indústria de jogos teve um rendimento superior ao obtido por *Hollywood*, que teve uma bilheteria de filmes na margem dos nove bilhões de dólares (SHAH, 2005).

Não obstante, esse rendimento obtido pela indústria de videogames não para de crescer, chegando aos 40 bilhões de dólares no ano de 2007. Percebe-se então que em pouco mais de dez anos de dominação no mercado, esta indústria segue quase que imune às diversas crises financeiras enfrentadas pelo mundo. Outro ponto interessante é que se estima que dentro de alguns anos, a indústria de videogames ultrapasse a indústria musical em rendimentos obtidos (ANDERSON, 2007).

3 HISTÓRIA DOS CONSOLES PORTÁTEIS

3.1 Os primórdios

Os *consoles* portáteis, ou *handhelds* como são comumente conhecidos, tiveram um início bastante difícil, basicamente devido à complexidade de se conseguir circuitos integrados minúsculos e que consumissem pouca energia. A sua história inicia com um portátil que foi pouquíssimo conhecido e que teve uma vida de apenas dois anos. O *Microvision* da *Milton Bradley Company* debuta em novembro de 1979 e acaba em 1981 devido ao seu elevado custo (WIKIPEDIA), mas, no entanto já introduzia algo ousado, ele possuía cartuchos intercambiáveis.

Ainda na segunda geração dos videogames, a Nintendo lança em 1980 o "*Game & Watch*" (figura 3.1), que era um *handheld* mais limitado que o *Microvision*. Ele não tinha a capacidade de trocar de cartuchos, assim, para cada jogo existia um "*Game & Watch*" diferente, mesmo assim, obteve bastante sucesso, o que lhe garantiu uma vida até o ano de 1991 e dezenas de títulos criados, entre eles "*Donkey Kong*" e "*Super Mario Bros*.".

Figura 3.1: *Game & Watch* inspirou o design do NDS

3.2 Ascensão dos consoles portáteis

Em 21 de abril de 1989 a *Nintendo* lança o produto que se tornou o *console* mais vendido de sua história, o *GameBoy* (WIKIPEDIA). O *GameBoy* possuía uma tela LCD preta e branca e consumia pouca energia, essa era a aposta dos projetistas da *Nintendo*; pois, mesmo sendo um *hardware* modesto, era um grande avanço se comparado ao antecessor *Microvision*.

Em contrapartida, a *Atari* e a *Sega* preparam-se para dar suas investidas no mundo dos portáteis, valendo-se de um recurso que inovou para o ano de 1989. O *Atari Lynx* e o *Sega Game Gear* além de terem um *hardware* mais potente comparado ao *GameBoy*, possuíam telas LCD coloridas, recurso esse que buscava bater a iniciativa da *Nintendo*. Infelizmente, um *hardware* potente e tela colorida vinham com um preço: o altíssimo consumo de energia. Para uma época em que não existiam baterias recarregáveis eficientes, o ponto alto acabou tornando-se um ponto negativo.

Mesmo assim, a *Nintendo* precisava de um jogo de sucesso para o seu desbotado portátil. Foi então que surge o famoso *Tetris*. Ele foi o grande responsável pelo estrondoso aumento de vendas dos portáteis, todos o queriam jogar onde quer que estivessem. A dupla *GameBoy* e *Tetris* rendeu à *Nintendo* 25 milhões de unidades vendidas em apenas três anos (WIKIPEDIA).

Outras empresas também tentaram entrar no ramo de portáteis, onde podemos destacar a *Tiger Electronics* com o seu *Game.com*, que foi o primeiro console portátil a disponibilizar uma tela LCD sensível ao toque. Também a SNK lança o *NeoGeo Pocket*, um *handheld* de tela colorida mas que sofreu pela baixa diversidade de jogos para o sistema.

Novamente a *Nintendo* sai na frente em vendas quando faz uma atualização do *GameBoy*. Surge então o *GameBoy Color* com um *hardware* não muito mais sofisticado do que seu antigo *handheld*, mas desta vez contava com uma tela colorida. Juntamente, a *Nintendo* planejou o lançamento de mais um jogo que se tornou a maior febre de todos os tempos, *PokéMon* (figura 3.2). Assim, até 31 de março de 2005, foram vendidos mais de 118 milhões de unidades de *GameBoy* e *GameBoy Color* (WIKIPEDIA).

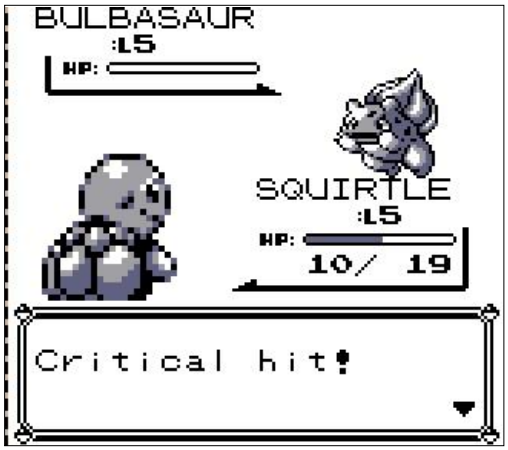

Figura 3.2: *PokéMon Blue*

3.3 Novo milênio

Durante a sexta geração surgem poucos *consoles* portáteis, sendo basicamente dominada pela *Nintendo*. No ano de 2001, é lançado *o GameBoy Advance* (GBA) que nada mais é do que uma evolução do antigo *GameBoy*. O GBA fez grande sucesso porque uma de suas principais características era um poderoso *hardware* que era capaz de ter vários jogos de SNES portados para ele, isso sem perder a retro-compatibilidade com os jogos de *GameBoy* e *GameBoy Color*.

Nas versões posteriores do GBA, foram inclusas baterias recarregáveis e *backlight*. O *backlight* é um conjunto de LEDs que se situam atrás da tela LCD, tornando possível assim o jogo num ambiente escuro.

Juntamente nessa geração, começam a aparecer os *handhelds* voltados para o desenvolvimento de softwares amadores; como possuem *hardwares* mais sofisticados do que seus concorrentes, podem ser vistos como computadores de mão. Apesar de seu nicho restrito, eles têm uma boa aceitação e sucesso; entre eles temos a linha GP32X, o *OpenPandora* e o *Dingoo*.

Antes mesmo de surgirem os primeiros *consoles*, a sétima geração inicia com os *handhelds*. Os *consoles* portáteis de última geração são representados basicamente por duas frentes: a *Nintendo* com o seu *Nintendo DS* que possui um *hardware* mais modesto se comparado ao concorrente, mas possui duas telas LCD, sendo que uma delas é sensível ao toque; e a *Sony*, que esbanja potência com o seu *PlayStation Portable* (PSP).

A disputa de mercado entre as duas marcas já vem se prolongando desde o ano de 2004. E desde então, pode-se observar um comportamento inédito no mercado: as vendas de *consoles* portáteis superam em número as vendas de consoles convencionais conforme pode ser visto na tabela 3.1.

Tabela 3.1: Comparação de vendas na 7ª geração

Fabricante	**Modelo**	**Unidades Vendidas**
Nintendo	Nintendo DS	107,75 milhões
Sony	PSP	55,9 milhões
Nintendo	Wii	52,62 milhões
Microsoft	Xbox 360	30,2 milhões
Sony	PlayStation 3	24,6 milhões

Fonte: WIKIPEDIA

4 NINTENDO DS

O *Nintendo DS* (simplesmente abreviado para DS ou NDS) é o *console* portátil mais recente da *Nintendo* e pertence à sétima geração dos videogames sendo o sucessor do consagrado *GameBoy*. O primeiro modelo do NDS foi lançado no dia 21 de novembro de 2004 e foi substituído pelo *DSLite* no dia 2 de março de 2006 (WIKIPEDIA). Atualmente, o NDS está em seu terceiro modelo conhecido como DSi (figura 4.1) e conta com mais formas de interação homem-computador.

Figura 4.1: *Nintendo DSi*, o modelo mais recente

Quando em tempo de projeto, o novo console da *Nintendo* recebeu o codinome de *Nintendo DS* que significava "*Developers' System*", mas logo em seguida seu codinome foi mudado para *Nitro*. Quando um protótipo foi preparado para ser exibido em público, seu codinome voltou a ser "*Nintendo DS*" e acabou ficando como a marca final do produto. Deste então, DS ficou referenciado como "*Dual Screen*" devido às duas telas LCD que o console contém.

4.1 Hardware

Apesar do *console* possuir uma arquitetura de *hardware* extremamente complexa, um diagrama esquemático bastante simplificado foi montado conforme pode ser visto na figura 4.2.

O primeiro aspecto a ser notado é a presença de dois processadores: um é o ARM7TDMI (33MHz) e o outro é o ARM946E-S (67MHz). Cada processador desempenha uma tarefa específica na execução de um *software*, enquanto o primeiro é responsável pela comunicação com os periféricos de entrada e saída do videogame; o

segundo executa as instruções do *software* propriamente dito e também se comunica com a *engine* gráfica.

Para que o ARM7 possa enviar e receber informações do ARM9, uma memória compartilhada de 32KB é utilizada. Além disso, existe uma memória RAM (4MB) de uso geral que é utilizada basicamente para carregar os recursos visuais e sonoros para que possam ser exibidos ou executados.

O código executável fica armazenado numa memória conhecida como ROM e pode possuir um tamanho máximo de 256MB. A memória ROM é comumente conhecida como cartucho de jogo.

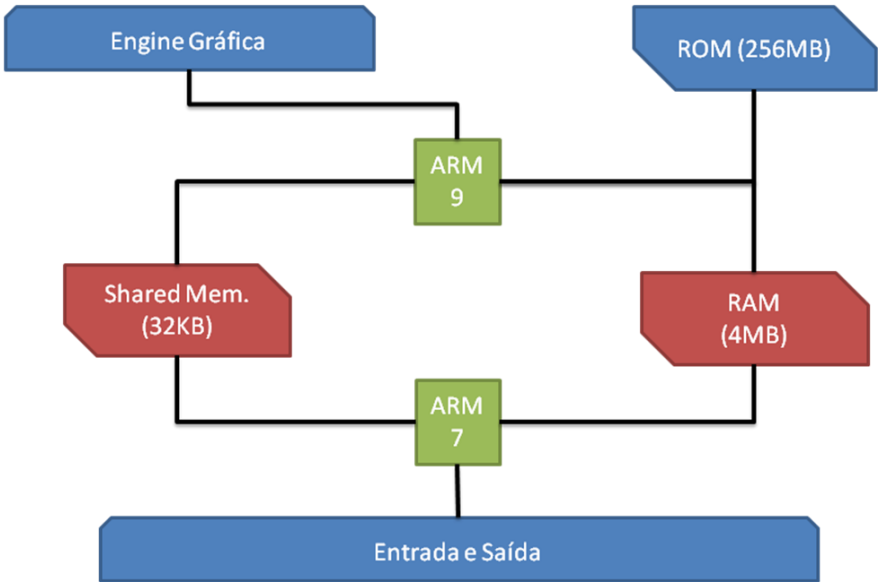

Figura 4.2: *Overview* do *Hardware*

4.1.1 Periféricos de Entrada e Saída

O *Nintendo DS* possui vários mecanismos para a entrada e a saída de dados. Já que se trata de um *console*, obviamente o NDS possui vários botões que são usados para controlar o jogo. A configuração de botões veio herdada do famoso SNES, consistindo de um botão direcional e mais seis botões de disparo conhecidos como A / B / X / Y / L / R, e ainda dois botões de controle: *START* e *SELECT*.

O maior destaque fica a cargo da película *touchscreen* que se localiza na tela inferior do *console*. A película além de capturar a coordenada em que a tela foi tocada, também mede a pressão do toque, permitindo uma maior precisão ao gesto do jogador.

Para a entrada e saída de áudio, o NDS conta com um microfone embutido, que pode ser utilizado em reconhecimento de voz; e dois alto-falantes que permitem um som estéreo em 16 canais.

O NDS vem com uma placa de rede sem fio embutida. A conexão *wireless* é o padrão IEEE 802.11b, permitindo que o NDS acesse a *Internet*; e também implementa o protocolo de comunicação NiFi (*Nintendo WiFi*) possibilitando a comunicação *ad-hoc* entre vários *consoles* NDS.

4.2 Motores Gráficos

O *Nintendo DS* tem dois motores (*engines*) gráficos, sendo um destinado para cada uma das telas do *console*. Os dois motores são classificados em motor principal (*main engine*) e motor secundário (*sub engine*) e são associados às respectivas telas (principal e secundária). É importante ressaltar que a classificação da tela é dada pela sua associação com o motor e não pela sua posição física, ou seja, a tela superior pode ser tanto a tela principal ou a tela secundária dependendo apenas de qual motor ela está associada no momento.

Apesar de ambos os motores serem bastante semelhantes e possuírem uma ótima renderização 2D em diversos modos de vídeo, o motor principal ainda conta com algumas outras funcionalidades, onde se destaca a renderização 3D.

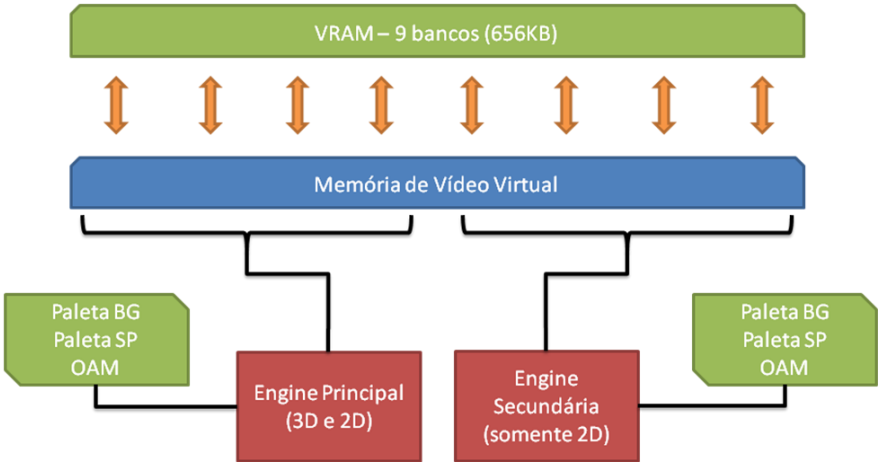

Figura 4.3: *Overview* dos motores gráficos

4.2.1 Memória de Vídeo

O NDS possui uma memória dedicada para vídeo chamada de VRAM. Esta memória possui um tamanho de 656KB, sendo dividida em nove blocos nomeados de A até I.

Como pode ser observado na figura 4.3, os motores gráficos “enxergam” apenas uma memória única chamada de Memória de Vídeo Virtual. Essa memória virtual não existe fisicamente e por esta razão, os blocos da VRAM devem ser mapeados a ela.

O uso da memória é determinado pelo intervalo da mesma que está sendo acessada, ou seja, a Virtual VRAM é divida em regiões e cada região possui a sua finalidade conforme a tabela 4.2.

Tabela 4.1: Virtual VRAM e suas regiões

End. Inicial	Tamanho	Finalidade
0x6000000	512KB	Memória de *Background* Principal
0x6200000	128KB	Memória de *Background* Secundário
0x6400000	256KB	Memória de *Sprites* Principal
0x6600000	128KB	Memória de *Sprites* Secundário

Os tipos de mapeamento são: Memória de *Background* que é utilizada para armazenar *tilesets*, mapas de *tiles* e também *bitmaps*; e Memória de *Sprite* que é utilizada exclusivamente para os gráficos dos *sprites*.

4.2.2 Sistema de Cores

A unidade fundamental de qualquer motor gráfico é o *pixel*, e o *pixel* é determinado pelo sistema de cores. Basicamente, qualquer cor suportada pelo *Nintendo DS* é descrita por 16 *bits* no formato ABGR como mostra a figura 4.4.

Figura 4.4: Formato de cor ABGR (16 *bits*)

Assim, para utilizar uma cor, basta usar a seguinte máscara: "`abbbbbggggrrrrr`", onde "`a`" é o *bit alpha*, "`bbbbb`" são 5 *bits* de intensidade de azul, "`ggggg`" são 5 *bits* de intensidade de verde e "`rrrrr`" são 5 *bits* de intensidade de vermelho. Como 5 *bits* permite a variação de 0 a 31, temos então um total de 32768 cores diferentes.

Para evitar um grande uso da memória de vídeo, são utilizadas paletas de cores, de modo que, um *pixel* não é mais determinado por 16 *bits*, mas sim, por 4 ou 8 *bits*. O *hardware* do NDS provê uma porção de memória específica para o uso de quatro paletas. São elas a paleta do *background* principal, a do *background* secundário, a dos *sprites* principal e dos *sprites* secundário.

Cada paleta possui 256 posições, portanto cada *pixel* pode ser determinado por apenas 8 *bits*. Além disso, a paleta pode ser dividida em 16 sub-paletas de 16 cores cada, e como conseqüência, cada *pixel* é descrito por 4 *bits* somente.

4.2.3 Renderização 2D

A renderização 2D pode ser feita basicamente de duas maneiras: modo *bitmap* ou modo texto. Mas independentemente do modo gráfico escolhido, o *Nintendo DS* trabalha com composição de imagens (exemplo na figura 4.5).

Figura 4.5: Cena do jogo *Chrono Trigger* (composição de *backgrounds* e *sprites*)

A composição final da tela é dada pela superposição de quatro imagens chamadas de *backgrounds* (figuras 4.6, 4.7 e 4.8). Cada *background* possui um valor de prioridade que varia de 0 a 3, assim, o *background* 3 é a camada mais inferior, enquanto que o *background* 0 é a camada mais superior.

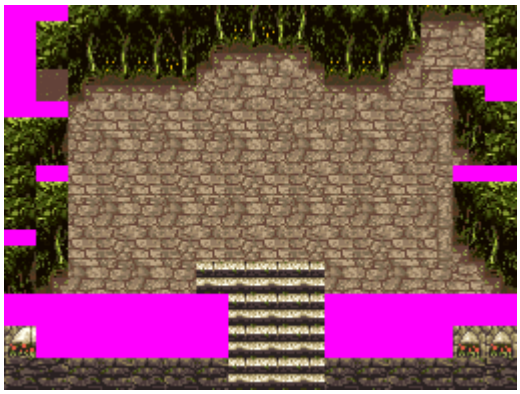

Figura 4.6: Camada de *background* 3

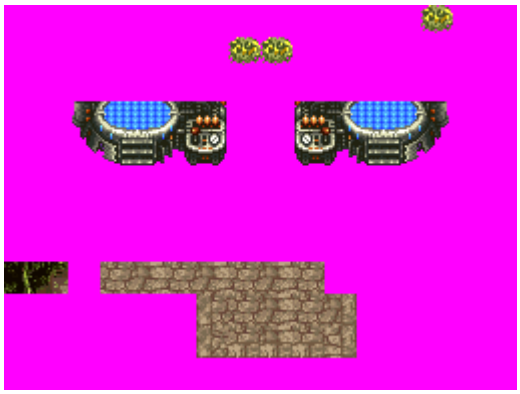

Figura 4.7: Camada de *background* 2

Figura 4.8: Camada de *background* 1

Além disso, existe uma camada especial nesta composição onde são desenhados os *sprites* (figura 4.9). *Sprites* são elementos gráficos animados que podem se mover livremente pelo *background*, e geralmente são compostos pelos personagens e itens dos jogos.

Um ponto importante que deve ser ressaltado é que todos os elementos gráficos (com isso entende-se os *backgrounds* em modo *bitmap* ou modo texto, os *sprites* e também a renderização 3D) são suportados por *hardware*, tornando a computação de cada quadro do jogo muito mais rápida.

Outro ponto que é igualmente relevante é que apesar da tela possuir dimensões físicas de 256x192 *pixels*, todo o processamento e gerenciamento de memória se dão sobre uma dimensão simétrica de 256x256 *pixels*.

Figura 4.9: Camada de *sprites*

4.2.3.1Background em Modo Bitmap

Este modo gráfico é o mais simples, pois a imagem formada no *background* nada mais é do que um mapa de *bits*, ou seja, cada *pixel* da imagem é diretamente descrita por um conjunto de *bits* dentro do mapa de *bits*.

O modo *bitmap* é o único modo que além da possibilidade de utilizar uma paleta de cores para descrever um *pixel*, também permite utilizar o sistema de cores ABGR diretamente. A comparação entre os modos pode ser vista na tabela 4.3.

Tabela 4.2: Modos *Bitmap*

	***Bitmap* 4**	***Bitmap* 8**	***Bitmap* 16**
Sistema de cores	Paleta de cores	Paleta de cores	Modo direto ABGR
Qtd de cores	16 cores	256 cores	32768 cores
Bits por pixel	4 *bits*	8 *bits*	16 *bits*
Tamanho (256x256)	32KB	64KB	128KB

4.2.3.2Background em Modo Texto

O modo texto é o modo que descreve um *background* usando um mapeamento de *tiles*, e esse processo se realiza em dois passos.

Primeiramente é alocada na memória de vídeo uma porção que será usada para armazenar os gráficos dos *tiles*. O *hardware* do *Nintendo DS* define um tamanho fixo de *tile* em 8x8 *pixels*. Para o conjunto desses *tiles* dá-se o nome de *tileset*, além disso, os *tiles* são numerados de 0 até n-1 para um conjunto de n *tiles*. Cada *tileset* pode conter no máximo 1024 *tiles*.

Na figura 4.10, pode ser visto um exemplo de *tileset* com os respectivos índices de seus *tiles*, no entanto, os *tiles* estão com um tamanho de 16x16 *pixels* para uma melhor visualização.

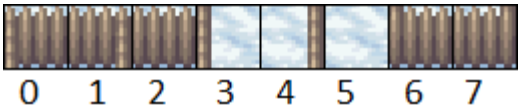

Figura 4.10: Exemplo de *tileset*

No modo texto, o *tile* é a unidade básica e é formado por 64 *pixels* que podem ser descritos usando uma paleta de 16 cores ou uma paleta de 256 cores. Quando se utiliza uma paleta de 16 cores, cada *tile* ocupará um total de 32 *bytes* dentro do *tileset*, enquanto que ao utilizar uma paleta de 256 cores, ocupará 64 *bytes*.

O segundo passo constitui no mapeamento dos *tiles* para compor um cenário. Novamente, outra porção de memória é alocada para acomodar as matrizes que comporão uma determinada imagem. Cada *background* é formado por uma matriz 32x32 chamada de "mapa de *tiles*" ou *tilemap*. Cada valor indexado dessa matriz tem um tamanho de 16 *bits* e representa um *tile* do *tileset*, assim, cada *tilemap* possui um tamanho de 2KB. Um exemplo de cenário gerado através de um *tilemap* pode ser visualizado a figura 4.11.

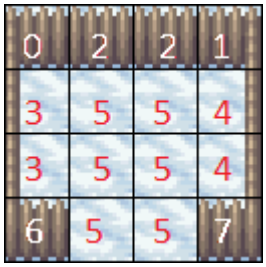

Figura 4.11: Exemplo de mapa de *tiles*

Outro recurso que o *hardware* do *Nintendo DS* disponibiliza são transformações afins sobre um *background*. Graças à matriz de transformação é possível obter efeitos como cisalhamento, rotação, aumento ou diminuição da escala e também um *scrolling* nos eixos x e y.

4.2.3.3Sprites

Sprites são pequenos objetos gráficos que podem ser manipulados independentemente uns dos outros e podem ser utilizados em conjunto com *backgrounds*. Os *sprites* também são formados por *tiles* como acontece com os *backgrounds*, mas, no entanto, ocupam outra região da memória de vídeo, não permitindo compartilhamento de gráficos com os b*ackgrounds*.

A utilização de *sprites* também se sucede em duas etapas distintas, onde a primeira é a carga dos gráficos e a segunda é o ajuste da *Object Attribute Memory*, conhecida simplesmente por OAM.

Na primeira etapa, deve-se alocar uma porção de memória especialmente para os *sprites*, onde os gráficos serão carregados também na forma de *tileset*.

Figura 4.12: Exemplo de *sprite* na memória de vídeo

Conforme pode ser visto na figura 4.12, os *tiles* são armazenados lado a lado, não sendo informação suficiente para que se possa exibir o *sprite* corretamente na tela. Para tanto, existe o OAM, que nada mais é do que uma tabela que possui 128 entradas e onde cada entrada corresponde a um *sprite* independente com seus parâmetros específicos. O *hardware* do NDS fornece duas tabelas OAM, uma para cada motor gráfico.

Na segunda etapa, a tabela OAM deve ser convenientemente preenchida. As informações relativas a cada *sprite* são as seguintes:

- Identificador único que varia de 0 a 127.
- Ponteiro para o primeiro *tile* que está na memória de vídeo.
- Profundidade de cores, que pode ser 16 *bits* por *pixel* ou a utilização de paletas. Assim, o *hardware* saberá o tamanho em *bytes* de cada *tile* do *sprite*.
- Tamanho do *sprite*, que pode variar de 8x8 *pixels* a 64x64 *pixels*. Somente assim o *hardware* saberá exatamente quantos *tiles* devem ser lidos da memória e como devem ser agrupados na tela.
- Posicionamento x e y do *sprite* na tela. É permitido posicionar o *sprite* fora dos limites da tela.
- Também é possível ajustar espelhamento vertical ou horizontal, e aplicar transparência.

Na figura 4.13, pode ser visto um *sprite* montado a partir do *tileset* exibido na figura anterior.

Figura 4.13: Exemplo de *sprite* montado

Novamente, o *hardware* provê mecanismos para transformações afins sobre os *sprites*. Embora seja bastante poderoso, existe uma limitação de apenas 32 *sprites* que podem sofrer tipos diferentes de transformações afins.

Juntamente com a OAM para os *sprites* regulares, existem mais 32 entradas para os *sprites* passíveis de transformações conhecidos como *rotsets*. Cada *rotset* possui um identificador único e atributos que modificam o *sprite* regular associado a ele. Assim, apesar do *hardware* ser limitado quanto ao número total de *rotsets*, é possível associar quantos *sprites* regulares se queira a um *rotset* específico; em contrapartida, um *sprite* regular pode ser associado a um e somente um *rotset* por vez.

4.2.4 Renderização 3D

Para o uso de desenhos em 3D, o motor gráfico utiliza um de seus *backgrounds* para receber as projeções da *engine* gráfica. Dentre as funcionalidades do motor em modo 3D, podemos salientar as seguintes:

- Transformação e iluminação: todas as operações relativas ao modo 3D são realizadas pelo motor, deixando a CPU livre para outras operações.
- Mapeamento de textura e *alpha blending*: é capaz de aplicar a textura em algum polígono e também aplicar transparência se desejado.

- *Anti-aliasing*: minimiza as distorções nas imagens devido ao tamanho reduzido da tela.
- *Cel shading*: também conhecido como "*toon shading*", simula desenho feito à mão.
- *Z-buffering* e *fog*: gerenciamento da profundidade e permite também aplicação de efeito de neblina.

Além disso, existem alguns detalhes que devem ser levados em conta: está disponível um total de 512KB para o armazenamento de texturas com no máximo 1024x1024 *pixels* de resolução; e pode-se renderizar 6144 vértices por quadro (ou 2048 triângulos por quadro), numa taxa de 60 quadros por segundo.

4.3 Desenvolvimento de Software

O desenvolvimento de *software* para *consoles* de videogames, em geral pode conseguido através de duas alternativas.

A primeira alternativa é apenas conseguida por empresas de médio a grande porte que já são especializadas no desenvolvimento de jogos. No caso da *Nintendo*, ela exige que a empresa possua alguns anos de experiência na área e dois ou mais jogos lançados no mercado. Somente assim a empresa possui os requisitos para se tornar uma desenvolvedora licenciada e por conseqüência poderá vender seus produtos desenvolvidos. A maioria das empresas como a *Sony* e *Microsoft* também trabalham desta maneira, exigindo certo nível de experiência e comprometimento de suas futuras licenciadas.

Tais requisitos por sua vez, trazem algumas facilidades, como *hardware* especial para a depuração do *software* a ser desenvolvido, bibliotecas e SDKs oficiais das empresas de videogames e também ferramentas que buscam automatizar e otimizar todo o processo de desenvolvimento do *software*.

A segunda alternativa é a opção que resta para os amadores e entusiastas na área de desenvolvimento de jogos para algum videogame e como não possuem nenhum tipo de suporte das empresas de videogames, o trabalho para o desenvolvimento costuma ser demorado e complexo. O *homebrewing*, como o próprio nome indica, é a "fabricação" de jogos em casa, ou seja, são pessoas que buscam aprender e desenvolver de forma não profissional algum *software* para a plataforma que costuma jogar seus outros jogos.

No caso de *homebrewing* para o *Nintendo DS*, apesar de ser uma tarefa árdua, existem alguns *softwares* que buscam facilitar de forma pontual o desenvolvimento dos *softwares*, como podem ser vistos a seguir.

4.3.1 Compilador ARM

Um grupo de desenvolvedores conhecido como *devkitPro* disponibiliza um conjunto de ferramentas conhecida como *toolchain* que tem por objetivo transformar o código fonte em um objeto executável no *hardware* alvo. Atualmente, o *devkitPro* dispõe de *toolchain* para as plataformas *Nintendo DS*, *Sony PSP*, *Nintendo Wii*, *Nintendo GBA* e outras.

Esse *toolchain* já contém todos os ajustes necessários para que o objeto executável gerado seja totalmente compatível com o *hardware* do *Nintendo DS*. Os principais passos que o *toolchain* executa são os seguintes:

1. Compilação: esta é a primeira etapa, onde o código fonte é transformado em código *assembly*. E em seguida esse código é transformado em arquivo objeto pelo *assembler*. Nesse ponto, nenhuma decisão é feita sobre onde residirá a memória de dados e a memória de código.
2. Ligação: o *linker* então é responsável por determinar exatamente onde cada função e variável irá residir na memória, levando em conta todos os arquivos objeto que serão usados para formar o arquivo executável. Para que o *linker* possa mapear corretamente os endereçamentos é necessário existir um *layout* da memória física do Nintendo DS que o *toolchain* já fornece.
3. Junção: o processo de compilação e ligação é realizado duas vezes, um para cada executável que rodarão nos processadores ARM7 e ARM9 respectivamente. No entanto, um software para o NDS não pode ser dois executáveis, então é realizado o processo de junção que agrega os binários ARM7 e ARM9 em um único arquivo executável. Neste ponto, os dados do cabeçalho da ROM são inseridos, como nome do *software* e ícone.
4. Polimento: a junção gera um executável completo para o Nintendo DS em formato .elf, mas no entanto, esse formato contém informações que o NDS não consegue executar. Na última etapa do *toolchain*, são removidas do executável todas as informação desnecessárias para que o NDS execute-o corretamente.

4.3.2 Biblioteca básica, o libnds

Paralelamente ao *toolchain* da *devkitPro*, é necessário uma biblioteca conhecida como *libnds*. Essa é uma biblioteca de baixo nível que fornece ao programador todos os ponteiros para as memórias e registradores que o *Nintendo DS* utiliza.

Desta maneira, o *libnds* permite que programadores não precisem memorizar um incontável número de endereçamentos para poderem trabalhar em seus *softwares*. Um exemplo disso seria tentar acessar o banco A da VRAM: ao invés de utilizar explicitamente o endereço `((u16*) 0x6800000)`, basta usar o *define* VRAM_A gerado pelo *libnds*.

Além dos endereçamentos, o *libnds* provê uma série de *structs* que permitem um fácil acesso aos recursos do *hardware*, bem como várias funções básicas que buscam trazer certo grau de abstração ao *hardware*. Um exemplo de função que o *libnds* fornece é o `vramSetBankA()`, esta função é utilizada para mapear o banco A da VRAM para algum dos endereçamentos da Memória de Vídeo Virtual.

4.3.3 Outras ferramentas

Somente o *toolchain* e o *libnds* não são suficientes para que se possa desenvolver alguma coisa para o Nintendo DS. Um último passo são os recursos gráficos e sonoros que serão utilizados no *software* a ser programado. Para que tais recursos possam ser utilizados, estes devem ser convertidos em um formato que possa ser lido pelo *hardware* do NDS. Em geral, os recursos devem ser convertidos para um formato *raw*,

ou seja, um formato que não possui nenhum tipo de cabeçalho ou informação adicional, apenas um conjunto de *bits* forma o som ou a imagem propriamente dito.

As duas principais representantes nesse setor são o *mmutil* e o *grit*. O *mmutil* é a ferramenta responsável por converter um arquivo de som arbitrário para o formato utilizável pelo NDS, além disso, o *mmutil* gera um arquivo *header* (.h) que conterá todos os ponteiros para os respectivos arquivos e também um arquivo objeto (.o) e que poderão ser facilmente integrados em qualquer projeto.

O *grit*, por sua vez, tem como tarefa analisar imagens buscando padrões de semelhança para poder gerar os *tilesets* e os *tilemaps* dos arquivos de imagens de entrada, e também permite gerar saídas no formato *bitmap* utilizadas pelo NDS. O *grit* é utilizado também para gerar os *tiles* para os *sprites* e além de tudo isso, ele busca otimizar a memória de vídeo, gerando arquivos que ocupam muito pouco espaço na ROM. Este utilitário também gera arquivos *header* e objeto (.h e .o respectivamente) para que possam ser integrados aos projetos.

4.3.4 Emulação

Infelizmente não existe uma maneira simples de depuração de código para quem desenvolve *homebrews*, no entanto, pode-se contar com a ajuda de emuladores. Existem poucos emuladores de DS que realmente funcionam, sendo que a maioria deles são projetos abandonados. Dentre os que executam satisfatoriamente uma ROM de DS estão o *no$gba* e o *desMume*.

O *no$gba* conta com alta performance e compatibilidade, mas é um projeto que está praticamente descontinuado. Existe juntamente a esse emulador, um programa de depuração de jogos que pode ser bastante útil aos desenvolvedores, mas, no entanto, é um depurador pago e seu autor não o vende mais.

O *desMume* é um emulador para *Nintendo DS* que também possui alto grau de compatibilidade e performance. Mas o que mais importa nesse emulador são algumas utilidades que ele dispõe, onde se pode ver o comportamento de várias porções de memória ao longo do tempo.

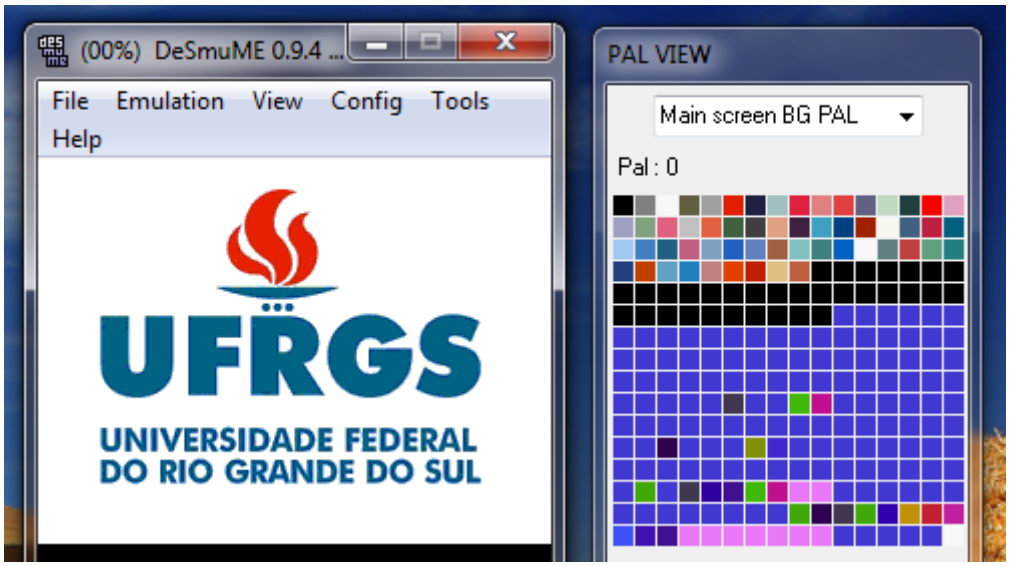

Figura 4.14: Visualizador de paletas de cores

Como podem ser vistas nas figuras 4.14 e 4.15, o emulador permite visualizar as paletas atualmente carregadas e ainda visualizar a memória de vídeo.

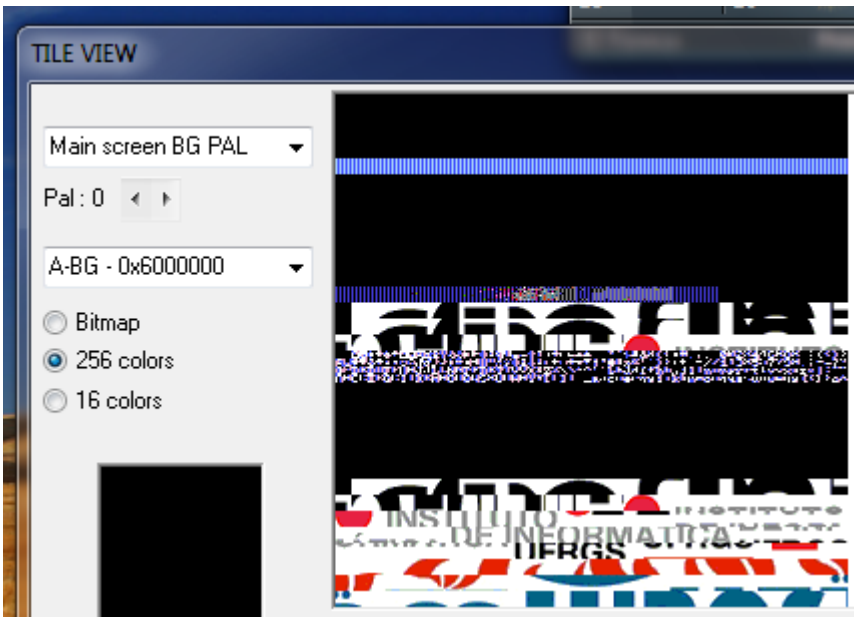

Figura 4.15: Visualizador da memória de vídeo

O *desMume* também conta com visualizadores para memória RAM, registradores, mapas de *tiles*, *sprites*, matrizes de transformação, mapeamento de luzes e visualizador do estado da *engine* de som.

4.3.5 Estrutura de um jogo

O desenvolvimento de um jogo envolve alguns passos diferenciados em relação ao desenvolvimento de outros tipos de *software*, e essa diferença se dá basicamente pela natureza dinâmica e altamente interativa que os jogos propõem.

A estrutura básica começa com duas partes essenciais. Primeiro é a inicialização que ajusta os motores gráficos, ajusta as memórias para que o funcionamento seja o esperado, basicamente é um passo vital que prepara o *hardware* para o seu devido uso. A segunda parte é o núcleo do jogo, que consiste em um laço infinito e é onde ocorrem as constantes atualizações entre um quadro e outro durante toda a execução do jogo.

O mecanismo preferencialmente utilizado no desenvolvimento de jogos é a máquina finita de estados. Assim, a cada iteração do laço principal, deve-se verificar em que estado o jogo se encontra para então as corretas atualizações serem feitas. Um fato interessante a ser notado é que a máquina de estados não possui um estado final, o que significa que na produção de um jogo para videogame deve-ser ter em mente que o jogo estará sendo executado infinitamente. Isto é válido porque a execução só é terminada quando o jogador desliga o aparelho.

Na figura 4.16, pode ser visualizado um exemplo de uma máquina de estados que pode ser utilizada como modelo para qualquer jogo a ser desenvolvido.

A cada quadro, ou seja, a cada iteração do laço infinito, deve ser feita uma série de passos que variam de acordo com a natureza do jogo em questão, embora os passos principais sejam: leitura dos comandos entrados pelo usuário, atualização do estado do

jogo baseada nessa leitura e finalmente o redesenho do quadro a partir das atualizações feitas.

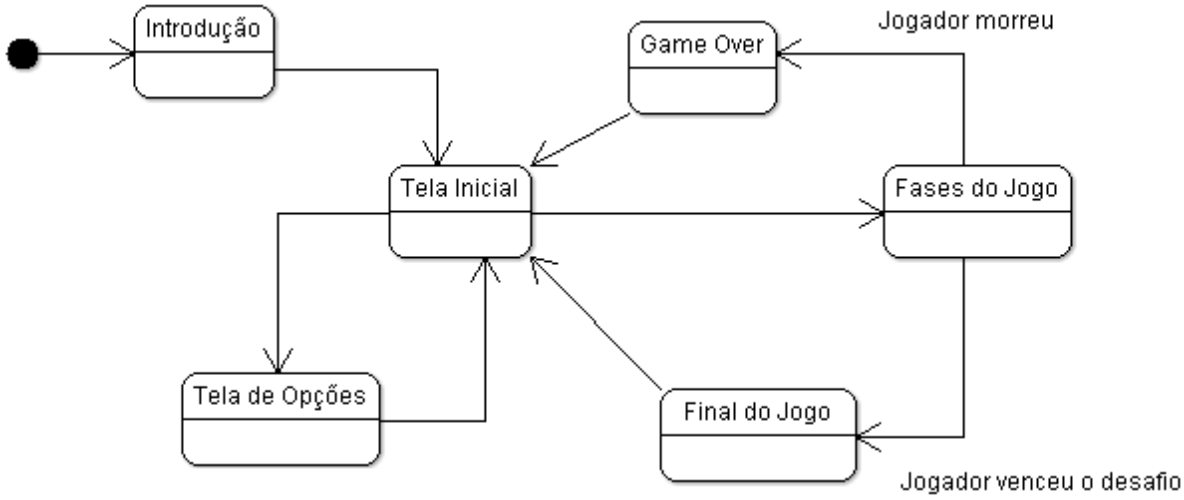

Figura 4.16: Exemplo básico de uma máquina de estados para um jogo

Por fim, um aspecto de extrema relevância na criação de um jogo é o tratamento da taxa de quadros, também conhecido como *framerate*. Mas diferentemente de como aconteceria no desenvolvimento de um jogo para computador, por exemplo, no *Nintendo DS* o *framerate* é fixo em 60Hz, isso acaba tirando a responsabilidade do desenvolvedor de sincronizar as ações do jogo com o tempo.

5 BIBLIOTECA FLIB

A *FLib* nasceu da necessidade de existir uma biblioteca de alto nível que fosse capaz de abstrair todos os aspectos físicos do *hardware* e tornasse o foco apenas no *software* a ser desenvolvido. Assim, aquele que utilizar a biblioteca *FLib* não precisará lidar diretamente com todos os tramites de ajustes de registradores, alocações de memória e gerência de recursos, pois a *FLib* é capaz de suprir todas essas necessidades de maneira simples e eficaz.

A biblioteca *FLib* parte de um princípio simples: asserções. Ou seja, uma série de pré-ajustes é efetuada de maneira que possa ser utilizada a maior parte de recursos do *Nintendo DS* da forma mais transparente possível. E, no entanto, permite também aos desenvolvedores mais experientes, um desenvolvimento mais baixo nível se estes acharem necessário.

5.1 Aspectos gerais

Como mencionado anteriormente, a principal motivação de uma nova biblioteca é que exista uma total abstração do *hardware* e suporte a criação de jogos de maneira extremamente simples. Mas isso não é tudo, a *FLib* busca toda essa facilidade valendo-se da orientação a objetos, o que torna o seu uso ainda mais fácil, e para tanto, a linguagem utilizada é o C++.

A biblioteca é dividida em quatro partes lógicas: classes de suporte, gerenciamento de entrada, gerenciamento de vídeo e construção de cena. Estas divisões se inter-relacionam de maneira absolutamente transparente ao desenvolvedor. Através de mensagens entre um objeto e outro a cada quadro, tem-se todos os objetos atualizados e consistentes para que possam ser extraídas todas as informações e ajustados todos os parâmetros. Apesar de a *FLib* ser compatível da criação com jogos tanto em 2D como em 3D, ela é principalmente focada na ambientações 2D.

O funcionamento da biblioteca se inicia ao se instanciar a classe *FLib* (diagrama na figura 5.1). A partir deste momento, podem ser utilizados todos os benefícios que ela proporciona. Um ponto importante que deve ser notado: no laço principal do programa deve ser chamado o método `Update()` desse objeto. Esse método é responsável por sincronizar todas as outras classes da biblioteca a cada quadro.

Ainda na parte principal da biblioteca temos outras classes importantes como a *FMusic*, *FSoundEffect* e *FMath* (diagrama na figura 5.1). As duas primeiras fazem parte do motor de som e são responsáveis por carregar um recurso sonoro (música ou efeito de som) para a memória e executá-la, assim que o objeto é descartado, ele se responsabiliza por remover o recurso da memória. Seus principais métodos são `Play()` e `Stop()` responsáveis por tocar e parar o som respectivamente.

E por fim, existe a *FMath*, que possui algumas funções básicas de geometria analítica para que possam ser usados para cálculos de coordenadas de forma rápida e eficiente. Todos os cálculos efetuados se utilizam de ponto fixo, pois o *Nintendo DS* não possui suporte a cálculos em ponto flutuante, além disso, os cálculos para seno e cosseno utilizam uma tabela de busca pré-calculada para evitar queda de desempenho.

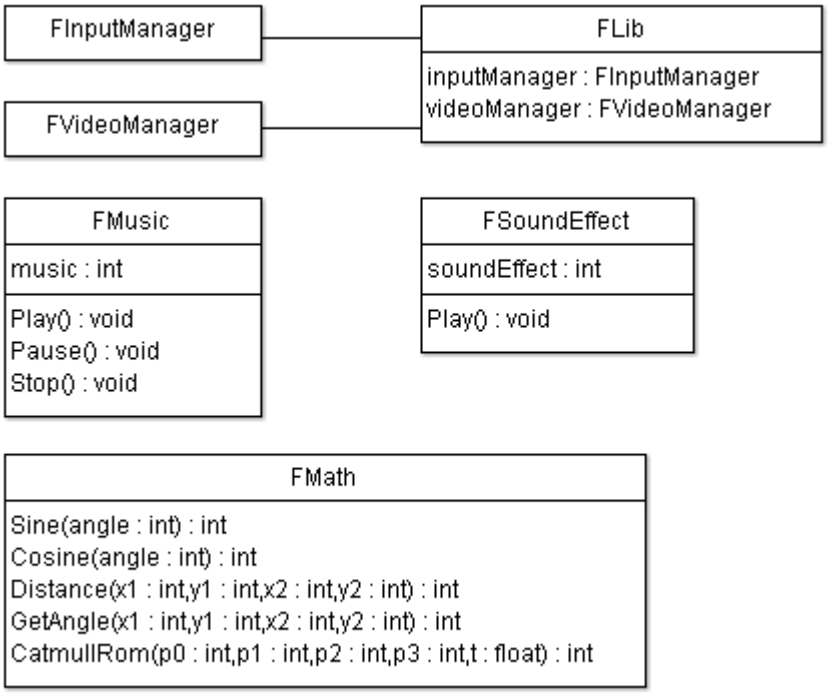

Figura 5.1: Diagrama de Classes – *Flib*

5.2 Gerenciamento de Entrada

O gerenciamento de entrada (diagrama na figura 5.2) trata da leitura de três mecanismos de entrada principais. Assim existe a classe *FInputManager* que é atualizada a cada quadro e provê uma maneira fácil de ler o estado do *Pad*, da *Touchscreen* e do *Lid*.

O objeto *Pad* consiste nos botões do *Nintendo DS*, que são *Up* / *Down* / *Left* / *Right*, A / B / X / Y / L / R e *START* / *SELECT*. Para cada um desses botões, existe um campo booleano para indicar se o botão está sendo apertado (*pressed*), soltado (*released*) ou segurado (*held*).

O objeto *Stylus* trata da leitura das informações da tela de toque, dentre as informações estão os booleanos apertado, soltado ou segurado e também as coordenadas X e Y no momento da leitura. O campo *Lid* do *FInputManager* diz se a tela do NDS está aberta ou fechada.

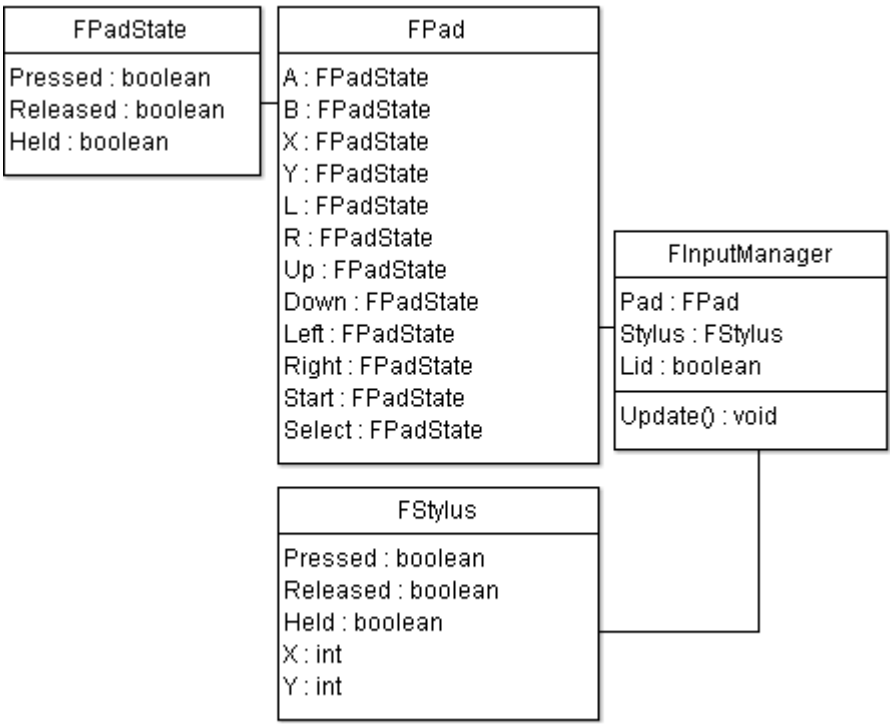

Figura 5.2: Diagrama de Classes – Gerenciamento de Entrada

5.3 Gerenciamento de Vídeo

O gerenciamento de vídeo trata dos ajustes dos modos gráficos, alocação de memória e atualização apropriada da tela a cada quadro. O objeto *FVideoManager* (diagrama na figura 5.3) pode ser inicializado em modo 2D ou em modo 3D, existindo assim algumas diferenças quanto as alocações para os *backgrounds* conforme pode ser visto na tabela 5.1.

Independentemente do número de dimensões desejadas, os modos gráficos disponíveis são o Modo *Bitmap* utilizando 256 cores e o Modo Texto também utilizando 256 cores. Todos os ajustes nos registradores dos motores gráficos ficam ocultos pela biblioteca *FLib*, tornando o uso de gráficos uma tarefa simples.

Tabela 5.1: Diferença dos *backgrounds* nos modos 2D e 3D

	2D	**3D**
Background 0	Console	Projeção 3D
Background 1	Modo Texto	Console
Background 2	Modo Texto	Modo Texto
Background 3	Modo Bitmap	Modo Bitmap

O gerenciamento de vídeo também efetua o mapeamento dos bancos de memória de vídeo (VRAM) para os correspondentes endereçamentos na Memória de Vídeo Virtual. Assim, temos que o banco A será utilizado para os *backgrounds* da *engine* principal e o

banco B, para os *sprites* da *engine* principal, o banco C será utilizado para os *backgrounds* da *engine* secundária e o banco D, para os *sprites* da *engine* secundária.

Ainda nesta parte da biblioteca, temos mais três classes: *FEngine*, *FPalette* e *FConsole* (diagrama na figura 5.3). A *FPalette* é responsável por acessar a respectiva paleta a qual está associada e seu método mais importante é `Load(int*)` que recebe um ponteiro de um vetor de cores previamente criado e preenche a paleta correspondente no *hardware*.

A classe *FConsole* é utilizada para escrever textos na tela. Para tanto, existem dois métodos `Print()` e `Type()`, onde o primeiro desenha as letras na tela em uma só vez e o outro desenha letra a letra como se fosse uma máquina de escrever. Além disso, é possível limpar a tela (`Clear()`), ajustar a cor a ser usada na impressão das letras (`SetColor()`) e ajustar uma janela de impressão (`SetWindow()`), assim o texto pode ser posicionado em qualquer lugar da tela.

Por fim, a classe mais importante destas é a *FEngine* que é responsável pelo gerenciamento dos motores gráficos. Existem, portanto, dois objetos de *FEngine*, um para cada motor.

Juntamente com o gerenciamento gráfico, os objetos de *FVideoManager* e *FEngine* dispõem de um conjunto de métodos para exibir e esconder as imagens das telas e também métodos para efetuar uma transição suave de uma imagem para outra. Para tais efeitos, foram definidos os métodos `Show()`, `Hide()`, `FadeIn()` e `FadeOut()`.

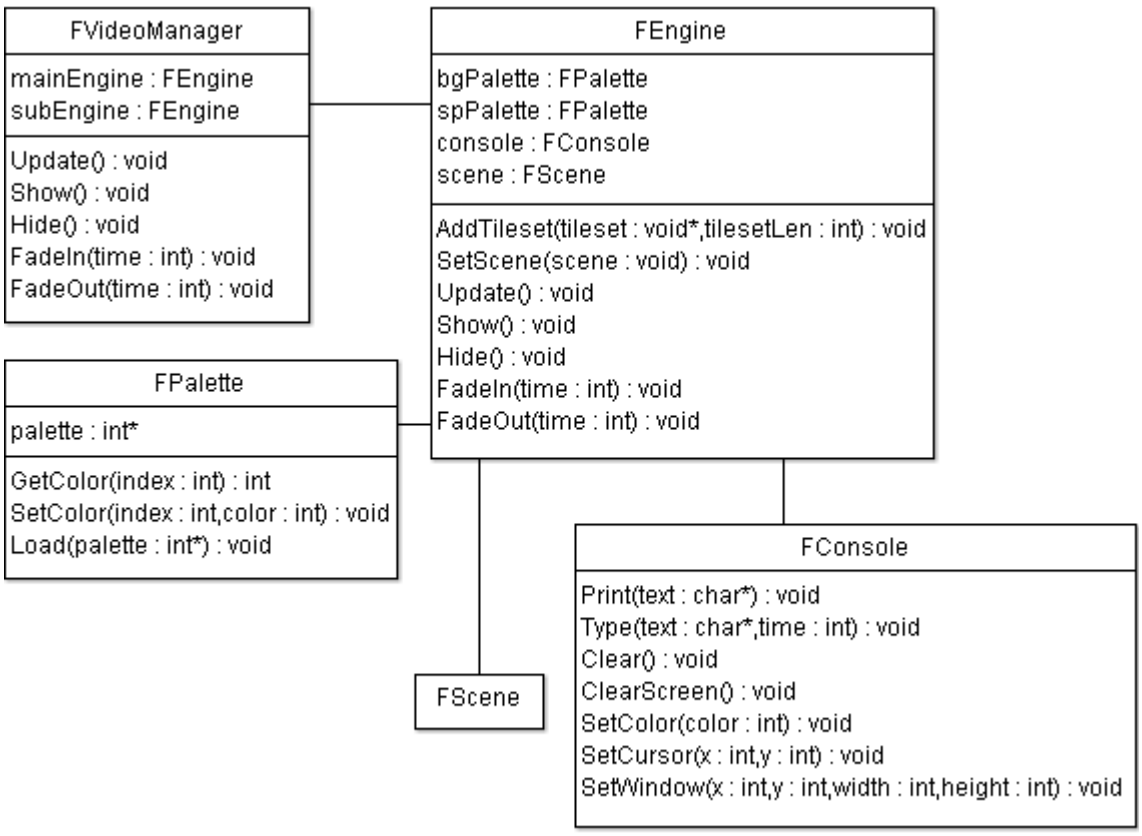

Figura 5.3: Diagrama de Classes – Gerenciamento de Vídeo

5.4 Construindo uma cena

As três partes anteriores da biblioteca *FLib* podem ser vistas como objetos de suporte, ou seja, eles existem e se cooperam sem que seja necessária qualquer interação do desenvolvedor do jogo. No entanto, esta última parte, é a de maior interesse àquele que utiliza a biblioteca *FLib*, pois contem as classes que serão usadas diretamente.

5.4.1 FBackground

A classe *FBackground* (diagrama na figura 5.9) é a classe que cuida dos aspectos de um *background* de uma *engine* a qual ele é associado, sendo que a principal funcionalidade que a *FLib* implementa é o *scrolling*.

O *scrolling* foi implementado da seguinte maneira: conforme a figura 5.4, tem-se um exemplo de cenário que possui um tamanho de 1024x1024 *pixels*, sabe-se que cada mapa de *tiles* possui um tamanho de 256x256 *pixels*, então existem um total de 4x4 mapas de *tiles* nomeados de A até P para formar o cenário do exemplo.

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Figura 5.4: Cenário contendo 4x4 mapas de *tiles*

Para fins de inicialização do exemplo, a tela está posicionada sobre o mapa A. Sabendo que o NDS permite visualizar 256x192 *pixels* na tela e que o *scrolling* é possível, deve-se notar que ao mover a tela horizontalmente de A para B, precisa-se que os mapas A e B estejam contíguos na memória; o mesmo raciocínio é válido se mover a tela verticalmente de A para E. No entanto, existe o caso de mover a tela diagonalmente de A para F, e nesse momento, os cantos adjacentes dos mapas A, B, E e F estarão visíveis na tela. Para que tal efeito possa ocorrer, é necessário que inicialmente os mapas A, B, E e F estejam contíguos na memória de vídeo como pode ser visto na figura 5.5.

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Figura 5.5: Alocação inicial dos mapas na memória

No entanto, esta configuração inicial permite que apenas os mapas A, B, E e F sejam exibidos. O que aconteceria se tentar mover a tela de B para C?

Para que tal efeito possa ocorrer, é necessário que os mapas A e E sejam descartados e em seu lugar sejam carregados os mapas C e G conforme mostra a figura 5.6. Desta maneira, agora é possível visualizar os mapas B, C, F e G sem que haja problemas.

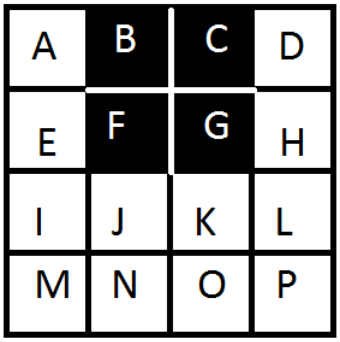

Figura 5.6: *Scrolling* horizontal, realocação de memória

De forma análoga, o *scrolling* vertical funciona da mesma maneira. Sabendo que os mapas B, C, F e G estão alocados contiguamente na memória de vídeo, o movimento da tela de B para F é perfeitamente possível sem qualquer outro ajuste. Mas ao mover a tela do mapa F para o J, uma nova substituição é necessária, ou seja, descartam-se os mapas B e C e em seus lugares alocam-se os mapas J e K como pode ser visualizado na figura 5.7.

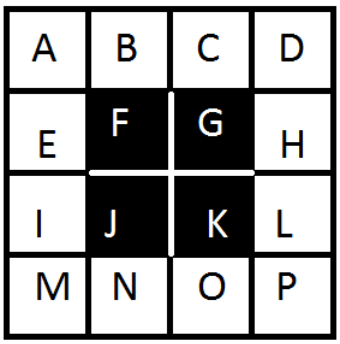

Figura 5.7: *Scrolling* vertical, realocação de memória

E finalmente para o *scrolling* diagonal, os passos são bastante similares aos *scrolling* horizontal e vertical, pois ele é apenas uma composição dos dois.

A implementação do *scrolling* foi feita de duas maneiras, uma chamada de `Scroll()` em que o cenário "trava" ao chegar nos limites de cada um dos quatro lados do cenário; e a outra é a `InfiniteScroll()` em que o cenário não "trava" e o *scroll* fica ativo, apenas repetindo a imagem quando chegar no limite do cenário.

Juntamente a esses métodos, a *FBackground* também dispões dos métodos `Show()` e `Hide()`, assim, os *backgrounds* podem ser ocultados e exibidos isoladamente em relação aos outros *backgrounds* da cena.

5.4.2 FSprite

A segunda classe que pertence à criação do cenário é a classe *FSprite* (diagrama na figura 5.9). A *FSprite* é responsável pelo tratamento dos *sprites* no cenário, atualizando seu posicionamento, rotação e animação.

Uma animação de *sprite* consiste na troca do gráfico do *sprite* (chamado de *frame*) num dado intervalo de tempo. Cada *frame* do *sprite* contém uma variação na animação que pode ser facilmente entendido pelo exemplo da figura 5.8.

Figura 5.8: *Sprite* contendo 12 *frames* de animação

Para que possa ocorrer a ilusão de animação do *sprite*, existem três métodos básicos:

1. É alocado um *sprite* associoado para cada um dos *frames*, assim, se animação precisar de três *frames*, três *sprites* são alocados. A animação acontece ao exibir apenas um *sprite* dentre todos, e a cada intervalo de tempo é feita a exibição de outro *sprite* e ocultando o que estava sendo exibido. A principal desvantagem é que o *Nintendo DS* suporta apenas 128 *sprites* por tela.
2. Apenas um *sprite* é alocado e um *frame* é associado a ele. Os demais *frames* também são copiados para a memória de vídeo. A animação acontece, pois a cada intervalo de tempo, o ponteiro do *frame* do *sprite* é mudado. A desvantagem é que esse método consome uma maior quantidade de memória de vídeo.
3. No último método, apenas um *sprite* é alocado e apenas um *frame* é copiado para a memória de vídeo. A animação nesse método ocorre de uma maneira um pouco diferente, pois a cada intervalo de tempo, um novo *frame* é copiado para memória, substituindo o *frame* anterior. Apesar de poupar memória, esse método possui a desvantagem de precisar efetuar cópias de memória constantemente.

O método escolhido na implementação da animação de *sprites* na *FLib*, foi o terceiro método, porque apesar de o método consumir um certo tempo a cada troca de *frame*, esse tempo pode ser atenuado utilizando-se um método conhecido por DMA (*Direct Memory Access*).

Os métodos que a classe *FSprite* provê são os seguintes: `SetXY()` que permite posicionar o *sprite* arbitrariamente na tela, `Center()` que permite posicionar o *sprite* no centro da tela, `Show()` e `Hide()` que permitem ajustar a visibilidade do *sprite*, `SetFrame()` que serve para ajustar o *frame* do *sprite* tornando assim a animação possível, e `Rotate()` que rotaciona o *sprite* a partir do ângulo recebido.

5.4.3 FScene

A última classe e que pode ser considerada a classe principal para o desenvolvimento de um jogo utilizando a biblioteca *FLib* é a *FScene* (diagrama na figura 5.9). A *FScene* é uma classe abstrata (não pode ser instanciada), que possui dois métodos puramente virtuais que são a chave do funcionamento da biblioteca. Esses métodos são: `Load()` e `Update()`.

O desenvolvedor ao utilizar a biblioteca *FLib* deverá criar uma série de classes que deverão herdar de *FScene*. Essas classes filhas já possuem todo o seu funcionamento e integração com a *FLib* implementadas, e apenas os métodos `Load()` e `Update()` devem ser implementadas pelo desenvolvedor de acordo com a necessidade do jogo.

O método `Load()` é o método que é invocado no momento em que a cena é associada à *engine* gráfica, e é nesse método que todas as inicializações pertinentes à cena devem ser feitas. Em geral, em `Load()` são preenchidas as paletas, os *tileset* são carregados na memória e os mapas de *tiles* são associados aos seus respectivos *backgrounds*. Este método é chamado apenas uma vez durante a vida do objeto.

O método `Update()` possui uma abordagem diferenciada, pois este é o método responsável pela atualização da cena e seus objetos. Esse método é invocado a cada quadro do jogo, ou seja, normalmente ele será chamado 60 vezes a cada segundo. Esse método continuará sendo chamado enquanto a cena estiver associada à *engine* gráfica.

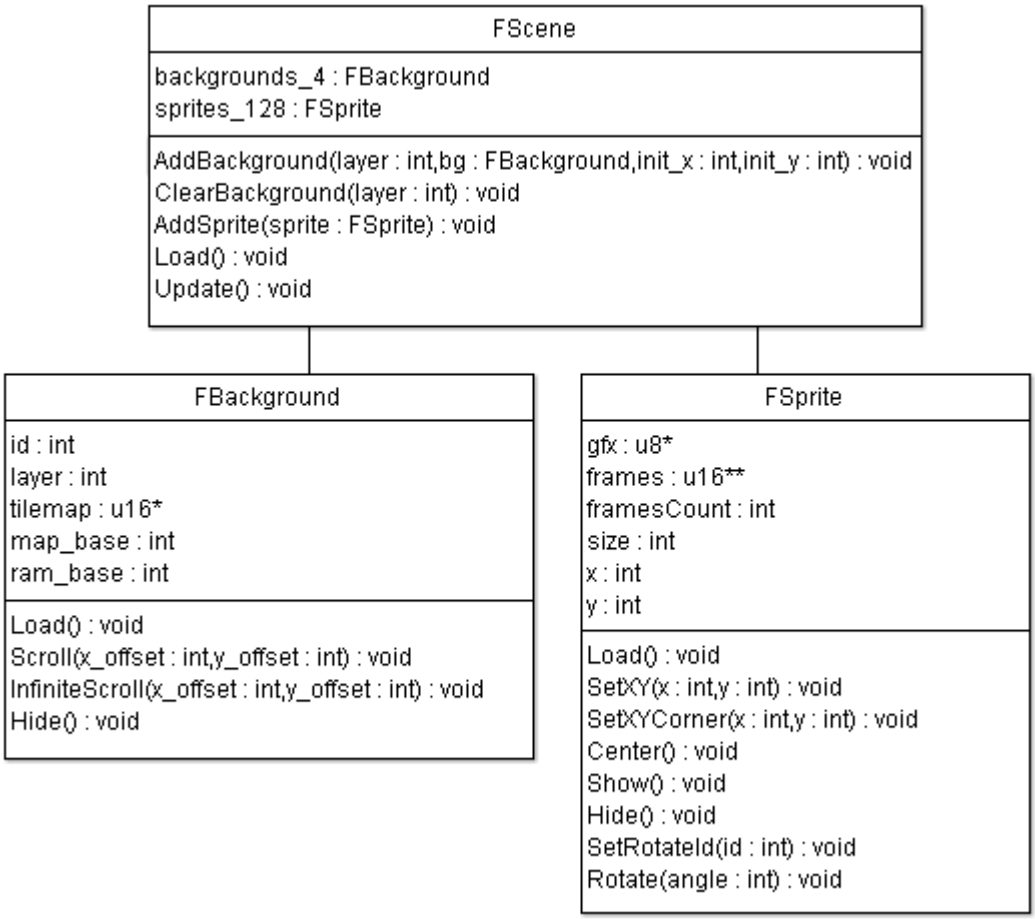

Figura 5.9: Diagrama de Classes – Cena

5.5 Os Cinco Passos para a Criação de um Jogo Utilizando a FLib

Tendo a *FLib* como seu objetivo principal simplificar o desenvolvimento de jogos, é possível com a sua utilização gerar um jogo em apenas cinco passos, os quais estão compreendidos a seguir.

5.5.1 Passo 1 – Inicializar a FLib

Primeiramente deve-se instanciar a classe *FLib*. No momento em que a classe é instanciada, todos os recursos do *hardware* são inicializados e ajustados para que o jogo possa começar corretamente. Segue o código:

```
#include "FLib.h"

int main()
{
   FLib *flib = new FLib();

   [...]

   return 0;
}
```

Nas linhas de código acima apresentado, são salientados os seguintes comandos: `#include "FLib.h"` que é responsável por incluir a biblioteca *FLib* no código fonte para que esta possa ser utilizada; e `new FLib()` que é onde a biblioteca inicia o seu funcionamento.

5.5.2 Passo 2 – Herdar da Classe FScene

Para que a lógica do jogo seja criada, deve-se criar uma classe que seja filha da classe *FScene*. Esta classe gerada terá todos os mecanismos de funcionamento já integrados com a *FLib*, ficando apenas a cargo do desenvolvedor a implementação dos métodos `Load()` e `Update()`. Conforme o código seguinte:

```
class MyScene : public FScene
{
   FBackground *bg1, *bg2;
   FSprite *sp;
   Int timer, frame;
public:
   virtual void Load();
   virtual void Update();
};
```

5.5.3 Passo 3 – Implementar o método Load()

O método `Load()` é executado somente uma vez durante a execução da cena e é responsável pela inicialização dos recursos que serão usados. Dentre os recursos a serem carregados para a memória estão as paletas de cores, o *tileset*, os *tilemaps* e os *sprites*.

```
Void MyScene::Load()
{
   // Ajustando paletas de cores
   engine->GetBgPalette()->Load(bgroundPal);
   engine->GetSpPalette()->Load(spritesPal);

   // Adicionando Tileset
   engine->AddTileset(bgroundTiles, bgroundTilesLen);

   // Adicionando Backgrounds
   bg2 = new FBackground(bground2Map, 2, 1);
   bg1 = new FBackground(bground1Map, 2, 1);
   AddBackground(2, bg2, 0, 0);
   AddBackground(1, bg1, 0, 0);

   // Adicionando Sprite
   sp = new FSprite(spriteTiles, 32, 32);
   AddSprite(sp);
   sp->SetXY(128, 145);
}
```

De acordo com o exemplo acima citado, pode-se observar o uso de algumas das funções de inicialização que a *FLib* disponibiliza através de seu conjunto de objetos. Dentre elas podemos destacar:

- `Load()` do objeto *palette* é utilizada para carregar completamente a paleta de cores a partir de uma matriz pré carregada.
- `AddTileset()` do objeto *engine* é utilizada para carregar o *tileset* dos *backgrounds* a partir de um ponteiro para os gráficos.
- `AddBackground()` do objeto cena é utilizada para associar um *background* à cena.
- `AddSprite()` do objeto cena é utilizada para associar um *sprite* à cena.

5.5.4 Passo 4 – Implementar o método Update()

O método `Update()` é executado uma vez a quadro do jogo, assim, a cada segundo é chamado 60 vezes. Seu principal objetivo é implementar a lógica a ser usada na cena. Um exemplo de implementação pode ser visto no trecho de código a seguir.

Neste exemplo, destacam-se o *scroll* dos *backgrounds* através do método `InfiniteScroll()` e também o método `SetFrame()` que é utilizado para realizar a animação do *sprite*.

```
void MyScene::Update()
{
   // Scrolling backgrounds
   bg2->InfiniteScroll(1, 0);
   bg1->InfiniteScroll(2, 0);

   // Animando o sprite
   if(++timer == 6)
   {
      timer = 0;
      if(++frame == 3)
         frame = 0;
     sp->SetFrame(frame);
   }
}
```

5.5.5 Passo 5 – Associar a Cena à FEngine

Finalmente, o último passo é responsável por associar a instância da classe criada e implementada nos passos anteriores à *engine*. Pode-se fazer a associação tanto ao motor principal quanto ao motor secundário.

```
#include "FLib.h"
#include "MyScene.h"

int main()
{
   [...]

   FVideoManager *vm = flib->GetVideoManager();
   FEngine *me = vm->GetMainEngine();

   MyScene *scene = new MyScene();
   me->SetScene(scene);

   while (true)
      flib->Update();

   return 0;
}
```

Nas linhas apresentadas, destacam-se os seguintes pontos: primeiramente é a inclusão da classe da cena criada através do comando `#include "MyScene.h"`; e em seguida sua instanciação e o comando `SetScene()` do objeto *engine*.

O método `SetScene()` faz a associação da cena à *engine* e faz com que o método `Load()` da cena seja chamado. Em seguida é necessário invocar o `Update()` do objeto *FLib* constantemente, sendo este o método responsável por atualizar todos objetos internos da *FLib* e finalmente chamando o `Update()` do objeto cena.

6 AVALIAÇÃO DA FLIB

6.1 Caso de Uso

Paralelamente ao desenvolvimento da biblioteca *FLib*, um jogo foi implementado utilizando-a. Esse jogo foi criado como avaliação para a disciplina de Projeto em Computação Gráfica e serviu como alavanca no levantamento de requisitos para a biblioteca e também como meio de verificar sua eficiência; ela permitiu que fases do jogo pudessem ser implementadas em apenas algumas dezenas de linhas de código.

O jogo gerado no projeto chama-se "As Incríveis Aventuras de Byteson" (figura 6.1) e é composto basicamente por *minigames*, onde toda a sua programação sucedeu-se através da *FLib*. O seu desenvolvimento tornou-se simples e dinâmico.

Figura 6.1: "As Incríveis Aventuras de Byteson" – Tela Inicial e Tela de Mapa

Para que a *FLib* seja utilizada, um espaço em torno de 100KB na memória RAM será ocupado pela mesma. Felizmente, este *overhead* é compensado pelo conjunto de objetos e abstrações que a biblioteca oferece em suas mais de 1400 linhas de código.

6.2 Desempenho

O desempenho é um dos fatores mais importantes para um jogo e sendo a *FLib* uma biblioteca para desenvolver jogos, esse é um ótimo ponto de medida. Todos os testes foram realizados no emulador *desMume* rodando num *notebook Dell Latitute 131L* (*AMD Semprom* 1.8GHz e 1GB de memória RAM) com o sistema operacional *Microsoft Windows 7 Professional*; e obviamente, o veredito final ficou a cargo do próprio *hardware* do *Nintendo DS*.

Os testes realizados envolveram apenas a utilização dos recursos do *hardware* em si, sem levar em conta a "lógica" do jogo, assim, algumas das variáveis do experimento puderam ser removidas. Dentre os testes de desempenho podemos citar:

1. Camadas de *backgrounds* em uma tela movendo-se com deslocamentos diferentes.
2. Um conjunto de *sprites* em uma tela com movimento e animação.
3. *Backgrounds* movendo-se com deslocamentos diferentes e 128 *sprites* (máximo permitido pela *engine*) com movimento e animação.
4. E finalmente, os mesmos testes anteriores, mas utilizando as duas telas simultaneamente.

Analisando um pouco mais a fundo o que acontece nos testes, três pontos podem ser observados. E esses três pontos buscam estressar a relação de desempenho da *FLib* com o *hardware*.

Ponto 1: Mover os *backgrounds* em velocidades diferentes serve para demonstrar o quão eficaz é a *FLib* no gerenciamento dos registradores dos *backgrounds*, principalmente na questão de *scrolling*.

Ponto 2: Mover e animar *sprites* envolvem dois passos importantes no teste. Um é a manipulação da tabela OAM de forma consistente; e o outro são as constantes cópias de memória para a animação.

Ponto 3: Uso das duas telas através de diferentes classes *FScene* que estão associadas a diferentes *engines* gráficas.

O resultado obtido foi o mesmo para todos os testes realizados: nenhuma perda de desempenho de forma aparente, ou seja, apesar de existir entre o *software* e o *hardware* uma camada bastante complexa chamada de *FLib*, esta se mostrou bastante competente e rápida, mesmo usando quase o limite em desenho 2D que o *hardware* do NDS disponibiliza.

Apenas para se ter uma idéia, o teste completo para a animação de *sprites* envolveu trocas de memória constantes na ordem de 196608 bytes (192KB) por frame.

A título de curiosidade, para chegar nesse valor basta multiplicar: 64 bytes (tamanho do tile) × 12 (tiles por *sprite*) × 128 (*sprites* por tela) × 2 (nº de telas).

6.3 Comparação

Também foi realizada uma comparação sob vários aspectos de uma aplicação simples medindo o uso e o não uso da *FLib* para desenvolver esta aplicação. Para tal comparação, não foi levado em conta a complexidade do *software*, mas sim, o uso de diferentes pontos que comumente são utilizados ao criar um jogo. Deste modo, a

aplicação gerada exibe duas imagens *bitmap*, uma em cada tela, lê as teclas que o usuário aperta e exibe estas informações na tela através do uso do *console* de texto. Um exemplo da execução pode ser vista na figura 6.2.

Figura 6.2: Aplicação de Teste

Conforme os resultados obtidos na tabela 6.1, pode-se perceber as vantagens que existem em utilizar a *FLib*, nas quais podemos destacar o "tempo de desenvolvimento" e o "número de linhas de código". Nota-se que para a aplicação de teste, a diferença foi bastante grande, demonstrando que a *FLib* realmente supera em valores esses quesitos. É importante ressaltar também que a diferença do "tempo de desenvolvimento" e do "número de linhas de código" tende a aumentar conforme a aumenta-se a complexidade do jogo.

No entanto, para o "tamanho final do jogo" ficou maior ao utilizar a *FLib*, tal fato já era de se esperar, pois conforme já mencionado, a *FLib* ocupa em torno de 100KB do espaço. Vale notar que o tamanho da *FLib* será sempre fixo independentemente da complexidade do jogo, assim, a medida que a complexidade do jogo aumenta essa diferença no "tamanho final do jogo" tende a diminuir.

Tabela 6.1: Teste de Comparação

	Sem FLib	**Com FLib**
Tempo de Desenvolvimento (TD)	20 minutos	10 minutos
TD estimado para um iniciante	45 – 60 minutos	20 – 30 minutos
Número de Linhas de Código	30 linhas	18 linhas
Tamanho Final do Jogo	250KB	279KB

6.4 Limitações

A *FLib* mostrou-se bastante competente em trazer ao desenvolvedor final uma facilidade na implementação de jogos juntamente com uma abstração total do *hardware*. Com a *FLib*, a mente criativa pode ficar totalmente voltada para o jogo a ser criado, livrando de alguns percalços comuns ao desenvolvimento para o *Nintendo DS*.

Infelizmente, tais simplicidades acabam implicando em certas limitações que podem ser relevantes para um desenvolvedor de NDS experiente. Tais limitações aparecem principalmente ao se tentar fazer um tratamento "automático" de um *hardware* tão complexo como é o do DS.

Dentre as limitações atuais mais importantes, pode-se citar:

- Para a utilização de cores, a *FLib* se utiliza apenas das paletas de 256 cores.
- Os modos gráficos dos *backgrounds* são fixos, ou seja, é um *background* em modo *bitmap*, outros dois em modo texto e o último para o *console* de texto.
- Não se pode trocar a ordem de renderização dos *backgrounds*, e a projeção 3D sempre ocorre sobre todos os *backgrounds*.
- Uma vez que o modo 3D é habilitado, a *FLib* não prove mecanismos para desabilitá-lo.
- A ordem de execução dos objetos das *engines* gráficas está ordenada para primeiro a principal e depois a secundária.
- A lógica deve ser dividida entre as classes *FScene* que estão associadas às *engines*, a comunicação entre elas fica a cargo do desenvolvedor.
- A animação da impressão do texto não compartilha o tempo com as outras animações, ou seja, ao usar o método *Type* da classe *FConsole*, as animações do jogo ficam pausadas até o término deste método.
- Não pode exibir um texto com partes dele em diferentes cores.
- A soma total do tamanho de todos os recursos não pode ultrapassar os 4MB, pois não está sendo utilizado um *filesystem*.
- E finalmente, não oferece suporte aos novos recursos do *Nintendo DSi*.

É importante salientar que tais limitações não impedem de maneira alguma o desenvolvimento de uma ampla gama de possibilidades de jogos, mas somente impõem algumas configurações preestabelecidas. No entanto, estas são as limitações atuais, ou seja, para futuras versões, estas seriam um prato cheio para alterações e expansões.

7 CONCLUSÃO

O sucesso do mercado de jogos pode passar despercebido aos olhos menos atentos, mas ao ser feita uma rápida análise de sua expansão ao longo dos anos é indubitável que a indústria de jogos eletrônicos sofreu um aquecimento tão grande que em 2007 teve um rendimento mundial de 41,9 bilhões de dólares.

Juntamente com o avanço tecnológico, equipamentos mais potentes, de menor tamanho e preço podem ser produzidos. E assim, o Nintendo DS não fugiu a regra, permitido que jogos em duas telas e de altíssima qualidade pudessem ser criados.

E a partir da necessidade de se desenvolver jogos amadores com facilidade, a biblioteca *FLib* foi concebida. Buscando auxiliar e otimizar o desenvolvimento de jogos amadores, através de um conjunto de classes que implementam todos os trâmites de comunicação do *software* com o *hardware*.

As principais dificuldades encontradas ao longo do desenvolvimento da biblioteca *FLib* foram as seguintes:

- A maioria dos tutoriais e exemplos disponíveis possui erros conceituais e muita contradição de um autor para outro;
- A falta de documentação da *libnds* também contribui para um atraso no entendimento necessário de seu funcionamento, assim, a principal fonte de informação foi o próprio código-fonte da *libnds*;
- Alguns pequenos *bugs* presentes na versão estável da *libnds* acabaram forçando que a *FLib* os corrigisse de uma maneira elegante e efetiva;
- A falta de plataformas de testes e principalmente depuração, forçando um trabalho manual e minucioso para solucionar os problemas que apareciam.

Como foi mostrado através das análises propostas, seu uso é indubitavelmente vantajoso, ou seja, apesar de existir uma insignificante perda no tamanho final do arquivo do jogo, o ganho em tempo de programação e em número de linhas de código é muito grande. E todo esse ganho ainda garante um desempenho impecável e um futuro bastante promissor.

Deste modo, existe um grande potencial para expansões futuras, características que podem tornar a criação de jogos ainda mais fácil. Dentre as atualizações que merecem destaque estão:

- Maior suporte a ambientações 3D, trazendo recursos que torne o seu desenvolvimento tão simples quanto já é em 2D.

- Facilitar o desenvolvimento de cenários que ocupem as duas telas, ou seja, prover mecanismos que integrem de maneira transparente o uso da memória e o processamento de ambas as telas.
- Codificação de todas as transformações afins que o *hardware* disponibiliza tanto para *backgrounds* quanto para *sprites*.
- Tornar dinâmico o chaveamento dos modos gráficos entre os *backgrounds* bem como suas alocações de memória, para que haja um rendimento ainda maior dos recursos.
- Permitir utilizar todos os modos de cores.
- Utilizar mecanismos de *filesystem*, permitindo assim que jogos maiores de 4MB possam desenvolvidos.
- Suporte à utilização de redes *Wifi*, tornando possível um jogo *multiplayer*.
- Suporte à leitura dos dados entrados pelo microfone.

Apesar das várias dificuldades encontradas, de a *FLib* não possuir um escopo mais abrangente, e de também estar longe de ser a solução definitiva para a criação de jogos, ainda sim, evita garantidamente muita retrabalho para o desenvolvedor.

REFERÊNCIAS

ANDERSON, John. Who Really Invented the Video Game? **Creative Computing Video & Arcade Games**, [S.l.], v.1, n.1, p.8, spring 1983. Disponível em: <http://www.atarimagazines.com/cva/v1n1/inventedgames.php>. Acesso em: set. 2009.

GETTLER, Joe. **The First Video Game?: Before 'Pong', There War 'Tennis for Two'**. [S.l: s.n.]. Disponível em: <http://www.bnl.gov/bnlweb/history/higinbotham.asp>. Acesso em: set. 2009.

POLSSON, Ken. **A Brief Timeline of Video Game Systems**. [S.l.: s.n.], 2009. Disponível em: <http://vidgame.info/mini.htm>. Acesso em: set. 2009.

BAER, Ralph H. **Videogames: In the Beginning**. [S.l.]: Rolenta Press, 2005. 280 p.

DOUBLE, Chris. **Homebrew Nintendo DS Development**. [S.l.: s.n.], 2005. Disponível em: <http://www.double.co.nz/nintendo_ds/>. Acesso em: set. 2009.

DOVOTO. **Dev-Scene: NDS Tutorials**. [S.l: s.n]. Disponível em: <http://dev-scene.com/NDS/Developers>. Acesso em: set. 2009.

AMERO, Jaeden. **Introduction to Nintendo DS Programming**. [S.l.: s.n.], 2007. Disponível em: <http://patater.com/manual>. Acesso em: set. 2009.

BOUNDEVILLE, Oliver. **A Guide to Homebrew Development for the Nintendo DS**. [S.l.: s.n.], 2008. Disponível em: <http://osdl.sourceforge.net/main/documentation/misc/nintendo-DS/homebrew-guide/HomebrewForDS.html>. Acesso em: set. 2009.

EBBESEN, Anders Vind. **Game Programming Crash Course**. [S.l.: s.n.]. Disponível em: <http://www.webbesen.dk/gba/>. Acesso em: set. 2009.

VIJN, Jasper. **TONC**. [S.l: s.n.], 2008. Disponível em: <http://www.coranac.com/tonc>. Acesso em: set. 2008.

PECKHAM, Matt. **Forecast: Video Game Market Will Soar to $57 Billion by 2009**. [S.l.]: PCWorld, 2008. Disponível em: <http://blogs.pcworld.com/gameon/archives/007189.html>. Acesso em: nov. 2009.

ANDERSON, Nate. **Video Gaming to Be Twice as Big as Music by 2011**. [S.l]: ars technical, 2007. Disponível em: <http://arstechnica.com/gaming/news/2007/08/gaming-to-surge-50-percent-in-four-years-possibly.ars>. Acesso em: nov. 2009.

SHAH, Nik; HAIGH, Charles. **The Video Game Industry: An Industry Analysis, from a VC Perspective**. 2005. 42 f. Dissertação (MBA) – Tuck School of Business at Dartmouth, Hanover. Disponível em: <http://mba.tuck.dartmouth.edu/digital/Programs/MBAFellowsProgramArchive/05_shah.pdf>. Acesso em: nov. 2009.

BRIGHTMAN, **James. PC Game Sales Bring US Industry to $18.85 Billion in '07**. [S.l.]: GameDaily, 2008. Disponível em: < http://www.gamedaily.com/articles/news/pc-game-sales-bring-us-industry-to-1885-billion-in-07>. Acesso em: nov. 2009.

RETROSPACE. História dos Video Games em 40 Capítulos. [S.l.]. Disponível em: <http://outerspace.terra.com.br/retrospace/>. Acesso em: set. 2009.

WIKIPEDIA. Homebrew Programmers Guide to the Nintendo DS. [S.l]. Disponível em: <http://www.dspassme.com/programmers_guide/tutorial/>. Acesso em: set. 2009.

WIKIPEDIA. How to Make a Bouncing Ball Game. [S.l.]. Disponível em: <http://ekid.nintendev.com/bouncy/>. Acesso em: set. 2009.

VGSALES. Video Game Industry. [S.l.]. Disponível em: <http://vgsales.wikia.com/wiki/Video_game_industry>. Acesso em: nov. 2009.

WIKIPEDIA. History of Video Games. [S.l]. Disponível em: <http://en.wikipedia.org/wiki/History_of_video_games>. Acesso em: set. 2009.

WIKIPEDIA. Golden Age of Video Arcade Games. [S.l.]. Disponível em: <http://en.wikipedia.org/wiki/Golden_age_of_video_arcade_games>. Acesso em: set. 2009.

WIKIPEDIA. Video Game Console. [S.l.]. Disponível em: <http://en.wikipedia.org/wiki/Video_game_console>. Acesso em: set. 2009.

WIKIPEDIA. List of Best-Selling Game Consoles. [S.l.]. Disponível em: <http://en.wikipedia.org/wiki/List_of_best-selling_game_consoles>. Acesso em: set. 2009.

DEVKITPRO. [S.l.]. Disponível em: <http://www.devkitpro.org>. Acesso em: ago. 2009.

DESMUME. [S.l.]. Disponível em: <http://www.desmume.com>. Acesso em: ago. 2009.

GRIT. [S.l]. Disponível em: <http://conarac.com/projects/grit/>. Acesso em: ago. 2009.

MMUTIL. [S.l]. Disponível em: <http://www.maxmod.org>. Acesso em: ago. 2009.