Renato dos Santos Cerqueira

Felipe Pedrosa Martinez

Ferramentas e métodos para o desenvolvimento de um jogo de plataforma 2D para o Nintendo DS

Rio de Janeiro - RJ, Brasil

13/07/2011

Renato dos Santos Cerqueira

Felipe Pedrosa Martinez

Ferramentas e métodos para o desenvolvimento de um jogo de plataforma 2D para o Nintendo DS

Monografia apresentada para obtenção do Grau de Bacharel em Ciência da Computação pela Universidade Federal do Rio de Janeiro.

Orientador:
Adriano Joaquim de Oliveira Cruz

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
INSTITUTO DE MATEMÁTICA
UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

Rio de Janeiro - RJ, Brasil

13/07/2011

Resumo

O objetivo deste trabalho é fazer um *engine* de jogo para o *videogame Nintendo DS*^{TM} e também um editor de fases e configurações, como seria feito numa equipe de desenvolvimento de um jogo comercial, dando a possibilidade aos *designers* de fazerem seus *sprites* e criarem as fases com eles, sem que fosse necessário mexer com códigos.

Abstract

The objective of this paper is to make a game engine to the Nintendo DS^{TM}system and a level and configurations editor, as it would be done in a development team in a comercial game, giving designers the possibility to make their sprites and create their levels without touching actual source code.

Dedicatória

Agradecimentos

Sumário

Lista de Figuras

1 Introdução

"*Quote 1*"
Author1

Neste capítulo são apresentados o objetivo desta monografia e a estrutura da mesma.

1.1 Objetivo deste trabalho

A área de jogos eletrônicos é uma área relativamente nova, se comparada a outros ramos da ciência da computação. No entanto, por mais que a área de jogos compreenda conceitos importantes de computação, ela engloba outros que fogem ao seu escopo. Nesse sentido, para o desenvolvimento de um jogo eletrônico, além de conhecimentos de linguagens de programação, algoritmos e estruturas de dados, interface humano-computador e inteligência artificial, é preciso considerar também os aspectos subjetivos que dizem respeito a um jogo de forma geral, seja ele eletrônico ou não. O *gameplay*, ou a mecânica do jogo, ou seja, a forma com que o jogo vai se desenrolar e o seu funcionamento, é um exemplo claro de que o seu desenvolvimento e sua concepção vão muito além do que pode ser programado, compilado e executado em um computador.

É possível observar o crescimento da área de jogos eletrônicos se notarmos que o primeiro exemplo conhecido aparece em 1947, quando Thomas T. Goldsmith Jr. e Estle Ray Mann introduziram sua patente para um Dispositivo de Entretenimento usando Tubo de Raios Catódicos em [Goldsmith Jr. et al. 1948].

Com a tecnologia cada vez mais aprimorada, conceitos básicos que definiam os jogos daquela época vêm sido substituídos por formas cada vez mais rebuscadas. Novos elementos de jogabilidade vêm sido incorporados à mecânicas clássicas, e estas evoluem por si só, dadas as cada vez menores limitações e as constantes inovações de hardware e interação com o jogador. Isso tudo se reflete no desenvolvimento de novas áreas de jogos: onde antes havia poucos estilos bem definidos, como corrida, luta e aventura, encontramos novos nichos como por exemplo jogos casuais, sociais e de *multiplay* massivo.

Jogos produzidos na época do Atari 2600 e Magnavox Odyssey, por exemplo, eram criações de um único desenvolvedor e em geral, concluídos em períodos curtos de tempo, sem nem ao menos identificá-lo. Em razão disso, era comum a aparição de *Easter Eggs*, recursos que os desenvolvedores utilizavam para assinar a sua criação, mesmo que de uma forma não convencional [Katie Salen e Eric Zimmerman 2006].

Hoje em dia, a indústria de jogos conta com super-produções que com uma quantidade quase infindável de desenvolvedores, designers, criadores de fases, dentre outros tantos profissionais trabalhando cada um com sua especialidade.

Neste trabalho, pretendemos nos inserir no contexto desses times de criação, sob a ótica do desenvolvedor, que, através da criação de ferramentas, facilita a interação entre as demais equipes do time de desenvolvimento.

Assim, criaremos um jogo em duas dimensões, do tipo plataforma para o videogame portátil Nintendo DS. Para isso, desenvolvemos uma *engine*, responsável por controlar todas as partes envolvidas no funcionamento do jogo como áudio, vídeo, estruturas de dados, controle de personagens, fases e itens.

Parte do nosso objetivo foi tornar esta *engine* simples e de fácil reutilização. Isto foi feito através de ferramentas de criação e configuração dos componentes pertinentes ao mundo do jogo.

1.2 Contextualização

Conforme dito anteriormente, criaremos ferramentas para desenvolver jogos em duas dimensões do tipo de plataforma. O questionamento imediato a partir desta proposta é: o que é exatamente um jogo de plataforma em duas dimensões?

Basicamente, estes são jogos onde a movimentação do personagem principal se dá verticalmente ou horizontalmente, sem considerar a aproximação ou o distanciamento deste ao jogador. Conforme o personagem se desloca pelo mundo, este vai se revelando a ele, e é comum que apresente inimigos e obstáculos, como uma fileira de espinhos ou buracos sem fundo. Em geral, o personagem principal tem um objetivo que o motiva a atravessar o mundo que o é apresentado, podendo ou não saltar ou possuir alguma habilidade especial, ter ou não acesso a alguma arma ou item, e enfrentar algum grande inimigo no final de tudo. Esta é uma definição bastante aberta, e é nela que nos inspiramos para escrever o *engine* que será executado no Nintendo DS. Para ilustrar, podemos pensar em exemplos clássicos, como Super Mario World para o Super Nintendo, ou Sonic The Hedgehog para o Sega Mega Drive. É certo que nem todos os jogos plataforma são em duas dimensões. Entretanto, neste trabalho, o foco será exclusivamente nos jogos de plataforma em duas dimensões, para facilitar o entendimento e o desenvolvimento de soluções.

Nessa modalidade de jogos, um recurso muito utilizado pelos desenvolvedores são *tiles* - ladrilho, em inglês - figuras de tamanho pré-definido, que compõem todo o mundo. Além disso, nesse tipo de jogo, todos os objetos costumam ser feitos a partir da combinações de *sprites*, usados mais de uma vez e em diferentes posições. Os *sprites* são responsáveis por representar possíveis inimigos, por exemplo. De forma mais simples, é possível dizer que enquanto os *tiles* compõem o mundo, os *sprites* o habitam.

Dentro desse contexto, o objetivo é criar uma *engine* que leia um arquivo de configuração, onde estarão detalhadas informações sobre quais são as imagens que compõem o personagem

principal (ou personagens, no caso de haver uma escolha); informações sobre os inimigos (como por exemplo a qual tipo de movimento que cada um obedece); descrição dos itens (como qual efeito ele causa no personagem principal ou nos inimigos) e informações sobre as fases (dentre elas o posicionamento dos elementos que as compõem).

1.3 Estrutura da monografia

No primeiro capítulo, introduzimos o nosso trabalho, discursando sobre o nosso objetivo e nossas metas ao realizá-lo. Dentre os pontos que citamos, comentamos brevemente sobre o cenário atual do mundo de desenvolvimento de jogos, de forma a ambientar o leitor com as circunstâncias sob as quais o desenvolvimento é dado. Assim, fica clara a importância do trabalho e o impacto que visamos ter com ele.

Na seção *Contextualização*, explicamos o foco do trabalho, restringindo e deixando bem definido o seu escopo. A partir disso, fica claro o que nos foi pertinente durante o desenvolvimento dele.

A seção seguinte é esta, na qual apresentamos um breve resumo sobre o que é abordado em cada parte desta monografia.

O segundo capítulo aborda os aspectos relativos ao desenvolvimento para o console que escolhemos. Primeiramente, comentamos o motivo desta escolha: o porquê resolvemos desenvolver nossas ferramentas para um console portátil, com as limitações e as particularidades do Nintendo DS.

Logo em seguida, damos ao leitor uma breve descrição sobre as especificações do console. É importante comentar as limitações que esta escolha nos ofereceu e as possibilidades que ela nos permite. Para isso, detalhamos o hardware e assim o leitor pode ter uma ideia do que ele é capaz.

No fim deste capítulo, explicamos como o desenvolvimento de jogos é dado para o Nintendo DS e condições normais. Nesta seção, descrevemos como os grandes desenvolvedores produzem seus jogos e os desafios que os pequenos desenvolvedores enfrentam, além dos artifícios que utilizamos para realizar este trabalho sob tais circunstâncias.

No terceiro capítulo, escrevemos sobre as ferramentas desenvolvidas: como cada uma se encaixa no fluxo de trabalho para a criação de um jogo completo, do início ao fim.

Neste capítulo, comentamos sobre como o usuário utilizaria a BGTool, e como a ferramenta executa cada uma das funções que são de sua responsabilidade. Em seguida, é feito o mesmo

tendo como foco a segunda ferramenta desenvolvida: a SpriteTool.

Finalizamos o terceiro capítulo com uma explicação sobre como a interação entre as ferramentas acontece, e quais foram as nossas preocupações ao escolher como este tipo de comunicação é dada.

O quarto capítulo, intitulado"*A Engine*", aborda alguns desafios que temos ao desenvolver um jogo. Estes desafios até então não tinham sido abordados, porém achamos importante pontuá-los: detectar colisões, construir o mundo do jogo e encapsular os *outputs* das nossas ferramentas em um jogo completo.

Na seção 4.2, descrevemos brevemente um *loop* genérico de jogo, de forma a ambientar o leitor que não está acostumado a programar jogos eletrônicos, comentando algumas particularidades genéricas que envolvem este nicho de programação. Em seguida, comentamos como são tratadas as colisões e por fim, como é dado o uso da *engine* e algumas configurações associadas. Explicitamos também, algumas estruturas importantes encontradas no desenvolvimento de um jogo com o uso dela.

Depois de desenvolver as ferramentas, criamos um jogo completo para ilustrar tanto a sua utilização quanto a participação num fluxo de trabalho de um jogo completo. No capítulo 5, descrevemos como as ferramentas foram aplicadas para a construção deste e comentamos o resultado final.

Por fim, concluímos este trabalho comentando alguns pontos importantes, metas atingidas e trabalhos futuros.

2 *Desenvolvendo para o Nintendo DS*

"*Quote 2*"
Author 2

Neste capítulo são apresentados a motivação para desenvolver para a plataforma, as suas especificações e como desenvolver.

2.1 Motivação

Há poucos trabalhos desenvolvidos para plataformas diferentes do PC, seja pelo mais restrito número de usuários, menor versatilidade de ferramentas e recursos associados ou mesmo pela maior dificuldade de encontrar documentação relativa ao desenvolvimento. A motivação para este trabalho é, dadas essas circunstâncias, criar um facilitador para o desenvolvimento de jogos para um console real, desenvolvendo ferramentas para fomentar uma maior quantidade de trabalhos para a plataforma, apesar das dificuldades.

2.2 Especificações

O foco deste trabalho se dá nas duas primeiras iterações do Nintendo DS: o Nintendo DS original, lançado no final de 2004, que normalmente é chamado de "DS Phat"; e sua segunda iteração, o Nintendo DS Lite, lançado no meio de 2006.

Ambos possuem um hardware muito parecido, sendo as suas maiores diferenças a estética, o contraste e iluminação da tela. Nesta seção, são detalhadas as especificações de cada um:

Nintendo DS:
Peso: 275 gramas
Dimensões: 148.7mm x 84.7mm x 28.9mm
Telas: Duas telas, ambas com 3 polegadas, de LCD TFT (*Thin-Film Transistor*) com 18-bit de cor, resolução de 256x192. As duas telas tem um espaço entre elas de aproximadamente 21mm. A tela inferior possui *touchscreen* resistivo, que responde a um ponto de pressão por vez. No caso de vários pontos da tela serem pressionados simultaneamente, ela responde na posição média destes pontos.
Além disso, a tela possui iluminação traseira, que pode ser desligada por software.
Entradas: Duas entradas de cartucho, uma para o formato de cartucho do seu antecessor, o Gameboy Advance(GBA), na parte inferior,e uma para seu próprio formato de cartucho na parte superior.
Processadores: Dois processadores ARM, um ARM946E-S a 67MHz, que é o processador principal, responsável pelo *loop* de jogo e renderização de vídeo e um coprocessador ARM7TDMI a 33MHz, responsável pelo som, *wi-fi*, e que quando em modo GBA, diminui seu *clock* para 16MHz, e passa a ser responsável pelo processamento principal.
Memória: 4MiB de RAM principal, expansíveis através da entrada de Gameboy Advance. No entanto, essa expansão só foi usada em jogos oficiais pelo Opera Browser.

Wireless: Conexão IEEE802.11b, compatível com encriptação por WEP. WPA e WPA2 não são suportadas. Essa conexão também pode ser usada para comunicação entre consoles.

Nintendo DS Lite:
Peso: 218 gramas
Dimensões: 133mm x 73.9mm x 21.87mm
Telas: Mesma especificação que o original, no entanto, possui quatro níveis de iluminação que podem ser reguladas por software. A tela também possui um contraste melhor que o original. Existem algumas outras menores diferenças, como a mudança do chip que controla o *Wireless*, o controlador da *touchscreen* também é ligeiramente diferente, mas para fins práticos, o resto da configuração é igual ao original.

2.3 Desenvolvimento

O desenvolvimento oficial no DS segue os mesmos moldes do desenvolvimento em consoles diversos: a Nintendo possui um time interno chamado de Intelligent Systems Co. Ltd. responsável pelas ferramentas de desenvolvimento para os consoles da empresa, dentre eles, o Nintendo DS. De outro lado, desenvolvedores independentes ou empresas de desenvolvimento, interessados em criar jogos para o console, pode entrar em contato com a empresa em busca das ferramentas necessárias, que incluem uma unidade especial de desenvolvimento, que se conecta ao computador por meio de uma porta USB. Justamente por ser voltada para desenvolvimento, dentre outras particularidades, esta unidade possui o dobro de memória de uma unidade normal, de forma a facilitar o *debug*. No entanto, para adquiri-las, é necessário assinar um Acordo de Confidencialidade (do inglês, *Non-Disclosure Agreement*), que não permite que se comente ou escreva sobre esses *kits* de desenvolvimento. Por causa disto, é difícil encontrar mais detalhes acerca de seu funcionamento.

Em geral, pequenas empresas de desenvolvimento não têm acesso ao *kit* de desenvolvimento oficial, seja por causa dos altos custos para obter o hardware ou pela dificuldade de adquirir as licenças. Desta forma, a publicação de jogos para consoles do tipo ainda é rara no mercado atual de jogos.

Para os fins do presente trabalho, cabe recorrer ao desenvolvimento não oficial, apelidado de *Homebrew*. A cadeia de ferramentas usada é chamada devkitArm, que serve para compilar programas em C e C++ para consoles com processadores ARM. Ao contrário do desenvolvimento oficial, não é possível produzir cartuchos de jogos *Homebrew*. Para testarmos, no entanto, po-

Figura 2.1: Uma imagem do kit de desenvolvimento oficial do Nintendo DS original

demos fazer uso de emuladores ou de um Nintendo DS com uso de um *flash cartridge*, um cartucho que possui uma memória *flash*, como um *pendrive*, e que pode ser apagada e reescrita várias vezes. Este cartucho não é suportado pela Nintendo.

A partir do devkitArm, foi desenvolvida a biblioteca libnds, feita especificamente para o hardware do DS. Essa biblioteca inclui funções de baixo nível que acessam todo o hardware: som, video, *wireless*, controle, dentre outros. O desenvolvimento nela apresenta dificuldades dado o baixo nível de suas instruções. Muitas vezes é necessário o uso de endereços de memória, cópias diretas da memória principal para a memória de vídeo, dentre outros artifícios não usuais na programação atual. Dado este cenário, baseadas na libnds, surgiram algumas bibliotecas em um nível mais alto. Para o desenvolvimento deste trabalho, foi escolhida a PALib, uma biblioteca que visa facilitar o desenvolvimento de jogos em duas dimensões que usem *sprites* e mapas de fundo.

Ainda assim, o uso dessas ferramentas não é trivial. Diferente de um programa convencional para *desktop*, as suas funcionalidades se apresentam em formas menos claras para o desenvolvedor: incluir ou gerar imagens, por exemplo, implica em um fluxo de trabalho diferente do esperado por um time de desenvolvimento convencional.

O conjunto de ferramentas desenvolvidos neste trabalho tem como objetivo tornar mais transparente e intuitivo o desenvolvimento no cenário apresentado. As ferramentas desenvolvidas são, em sua maioria, visuais pa1ra que o seu uso se dê de uma forma simplificada.

Para desenvolver as ferramentas foi usado o QT, um *Framework* multi-plataforma que permite o uso de interfaces gráficas em C++, além de possuir classes que facilitam o acesso a

tecnologias como XML, acesso a diversos formatos de arquivos de imagem, dentre outras possibilidades. A escolha foi feita para que as ferramentas funcionem em diversos sistemas operacionais sem precisar de mudanças no código.

3 *As Ferramentas*

"*It's dangerous to go alone! Take this.*"

Old Man (Zelda)

Neste capítulo são apresentadas as ferramentas e o seu método de uso.

3.1 A criação do jogo

Um jogo necessita de algumas partes constituintes, como por exemplo, suas fases e cenários, seus itens, inimigos e personagem principal. Com o uso das ferramentas apresentadas neste trabalho, o usuário executa os seguintes passos:

Em primeiro lugar, o usuário cria os cenários usando a BGTool, onde ele desenha os cenários em que o jogo vai se passar. Esses cenários ficam organizados logicamente em fases, e essas fases, no produto final, aparecerão encadeadas.

Em seguida, é necessário importar as figuras dos personagens, inimigos e itens. Isso é feito a partir da SpriteTool.

3.2 BGTool: A ferramenta de criação de cenários

A BGTool foi criada com o objetivo de permitir que o usuário importe seus *tiles* pré-existentes e a partir deles, desenvolva um novo cenário. Além disso, a ferramenta também é responsável por exportar esse cenário para o formato do Nintendo DS.

Ao começar um novo projeto, o usuário pode escolher quantas camadas de *background* o cenário que está sendo criado terá. O hardware do DS suporta até cinco camadas de *background*, no entanto, usar todas elas não é compulsório e o número de camadas a serem usadas fica a critério do desenvolvedor. Com as várias camadas, porém, a gama de recursos para o desenvolvimento do jogo aumenta significantemente. O desenvolvedor pode, por exemplo, fazer uso de Paralaxe, uma técnica em que as camadas mais distantes da câmera se movem mais devagar, dando uma ilusão de profundidade e imersão.

Por uma combinação de exigências do hardware do DS e das bibliotecas usadas, uma camada de *background*, na verdade, é implementada como um mapa de *tiles*. Isto significa dizer que as imagens que compõem o *background* são compostas de imagens menores, de tamanho fixo, que podem ou não ser rotacionadas ou espelhadas na vertical ou horizontal. E estas imagens menores são chamadas de *tiles*, conforme mostrado anteriormente.

Portanto, além do número de camadas, o usuário também especifica o tamanho dessas. A única limitação quanto ao seu tamanho é ter suas dimensões múltiplas de 8. Assim, neste trabalho, utilizamos *tiles* de tamanho 8x8 *pixels*, o mais simples de ser utilizado no DS.

Ao importar os *tiles*, o usuário pode usá-los para editar o *background* da fase que está construindo. Uma paleta é gerada automaticamente pela ferramenta a partir dos *tiles* que o

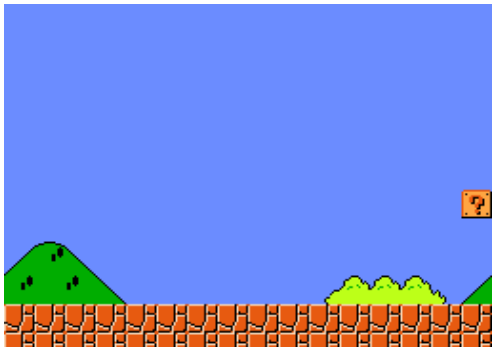

Figura 3.1: A imagem antes de ser carregada

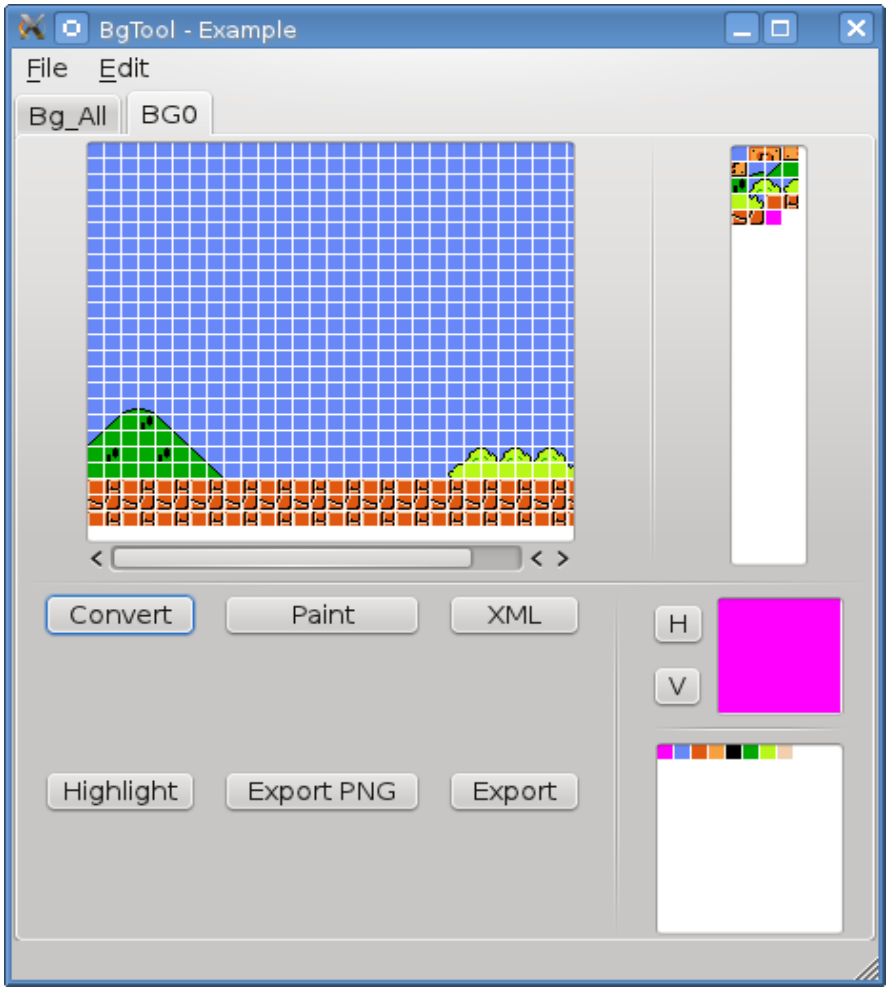

Figura 3.2: A imagem após a abertura na ferramenta

usuário carrega. Em seguida, a ferramenta se encarrega de fazer a importação de uma imagem e transformá-la em *tiles*, dividindo-a em partes de 8x8 *pixels* e eliminando as possíveis repetições. Como o DS suporta espelhamento na horizontal e na vertical de um *tile*, então é feita uma verificação se o *tile* em questão é igual a um dos *tiles* existentes, em quaisquer das configurações: sem espelhamento, com espelhamento vertical, horizontal ou ambos.

Os *tiles* são armazenados como imagens, sendo assim não há um algoritmo de *hash* dis-

ponível no QT que pudesse ser utilizado para organizá-los. A implementação de tal algoritmo foge ao escopo deste trabalho e por isso, é feita uma busca exaustiva pelo vetor de imagens comparando as imagens geradas com as que já existem. Embora esse seja um processo relativamente custoso, com complexidade O(n) para cada inserção, dado o tamanho da entrada de dados, este custo pode ser ignorado.

Figura 3.3: Alguns *tiles* depois de separados

Figura 3.4: Configurações possíveis para um *tile*

Com os *tiles* já importados, o usuário pode construir os mapas que compõem a fase. A ferramenta separa as diversas camadas de fundo em abas, para a edição separada de cada uma. Porém, ao construir uma fase, é possível ver uma composição destas camadas em uma imagem de *background* de forma a prover um *feedback* do ao usuário de como ficaria a versão final dela.

Ao salvar o projeto, suas informações são armazenadas em um arquivo XML com um formato bem definido, numa área delimitada destinada à BGTool. Lá, ficam organizadas informações relativas às diversas fases tais como suas camadas. Isto será explicado com mais detalhes na seção 3.4.

A qualquer momento, o usuário pode exportar o que já está pronto para o DS, esse é um processo transparente no qual são gerados os arquivos necessários para a execução da fase criada por ele. São criados três arquivos, um arquivo Pal, que contém a paleta, um arquivo Tile que contém os *tiles*, indexados pela paleta e um arquivo Map, que contém o mapa, ou *background*, indexado pelos *tiles*.

Os três arquivos são binários e obedecem a um formato delimitado pela PALib e suas funções, que coordenam o carregamento de cenários.

Uma breve explicação desses formatos é a seguinte:

- Arquivo Pal

  Neste arquivo, temos informações sobre a paleta usada no arquivo. São descritas todas as

cores contidas no arquivo. Precisam haver 256 cores, mesmo que o arquivo seja inflado de cores falsas. O Formato usado no arquivo é A1B5G5R5, o que significa que há um bit para o caso da cor ser transparente (alpha channel), 5 para o verde, 5 para o azul e 5 para o vermelho. A primeira cor no arquivo é a cor usada como transparência. (A cor padrão sendo o Magenta)

- Arquivo Tiles
  Neste arquivo, ficam os *tiles*, cada um de 8x8 *pixels*. Cada pixel do *tile* referencia uma cor do arquivo Pal. Neste arquivo, os tiles são escritos em blocos (vetores) de 64 bits.

- Arquivo Map
  Neste arquivo, fica o mapa do *background*, ele referencia os tiles. Os bits 0-9 indicam o tile, o bit 10 indica se é espelhado na horizontal e o bit 11 indica se é espelhado na vertical. Os bits restantes indicam qual paleta está sendo usada para o mapa em questão, eles não são usados nos modos em que só há uma paleta, ou seja, modos de 256 cores.

- Arquivo .c
  Este arquivo constrói a *struct* que será usada no programa, informa o tipo do mapa, o seu tamanho, os ponteiros para as regiões onde ficaram os três arquivos acima depois do processo de *link*, tamanho do tiles em bytes e tamanho do mapa em bytes.

No momento em que o usuário carrega uma imagem, ela é convertida para um formato de 256 cores. Devido as limitações de cor, o ideal seria que o usuário já carregasse imagens com apenas 256 cores, desta forma não há perda ou alteração de cores da imagem original, ou no caso dessa perda ser necessária, ela acontece com o controle do usuário.

Pelo mesmo motivo, caso mais de uma imagem seja utilizada na composição de um mapa, o usuário deve fazer com que as imagens compartilhem uma mesma paleta.

Depois dessa conversão inicial, todas as ações no programa são feitas usando a paleta gerada na fase de carregamento. Na exportação, a paleta é exportada diretamente, sendo simplesmente colocada no arquivo binário.

Os *tiles*, quando exportados, também não necessitam de preocupação adicional. Sua conversão para o formato binário envolve apenas obter o índice da cor de cada pixel de cada *tile* e escrevê-lo no arquivo binário.

A geração do mapa, ou *background*, é a tarefa mais complexa. Internamente na ferramenta, o mapa é armazenado como uma matriz de estruturas do tamanho do cenário, onde cada estrutura armazena o índice do *tile* daquela posição e também se naquela posição este tile está

espelhado na vertical, na horizontal ou ambos.

Obtidas essas informações, o arquivo é convertido para o formato discutido acima e exportado para o arquivo binário.

Listing 3.1: Função que exporta o mapa

```
void cBackground::export_map_to_ds()
{
  QString map = "../gfx/bin/" + m_name + "_Map.bin";

  //exporting map
  QFile file_map(map);
  if(file_map.open(QIODevice::WriteOnly | QFile::Truncate))
  {
    QDataStream out(&file_map);
    for(int i(0); i < m_map_matrix.size(); ++i)
    {
      for(int j(0); j < m_map_matrix[i].size(); ++j)
      {
        unsigned char tile_index;
        unsigned char tile_flipping;
        sMapInfo tile = m_map_matrix[i][j];
        tile_index = (unsigned char) tile.m_tile_index;
        tile_flipping = ((tile.m_tile_index & (1024+512)) >> 8);
        if(tile.m_tile_flipping == VERTICAL_AND_HORIZONTAL_FLIPPING)
        {
          tile_flipping |= 4 + 8; // bit 3 e 4
        }
        if(tile.m_tile_flipping == HORIZONTAL_FLIPPING)
        {
          tile_flipping |= 4;
        }
        if(tile.m_tile_flipping == VERTICAL_FLIPPING)
        {
          tile_flipping |= 8;
        }
        out.writeRawData((const char *)&tile_index, 1);
        out.writeRawData((const char *)&tile_flipping, 1);
      }
    }
  }
  file_map.close();
}
```

Por fim, depois de exportados, esses arquivos estarão prontos para serem usados num estágio posterior da cadeia de desenvolvimento e inseridos no jogo final.

3.3 SpriteTool: A ferramenta de criação de *sprites*

A SpriteTool foi criada com o objetivo de, a partir de simples imagens, gerar *sprites* para o desenvolvimento de um jogo completo, executável no Nintendo DS. O objetivo de tal ferramenta é criar um ambiente para o usuário, no qual ele não tenha que lidar com as particularidades do desenvolvimento que não sejam pertinentes à criação do *sprite* em si.

Além de permitir a portabilidade das imagens para o Nintendo DS, transformando-as em *sprites* do jogo no formato reconhecido pelo console, a SpriteTool gera um arquivo de projeto no formato XML, que pode ser reconhecido pelas outras ferramentas desenvolvidas neste trabalho. A integração entre estas ferramentas é feita respeitando protocolos pré-estabelecidos de escrita e leitura nestes arquivos, de forma transparente para o usuário. A seção 3.4 aborda em detalhes a estrutura do formato escolhido e como cada uma das ferramentas faz uso dele para armazenar e resgatar as informações pertinentes ao desenvolvimento do jogo.

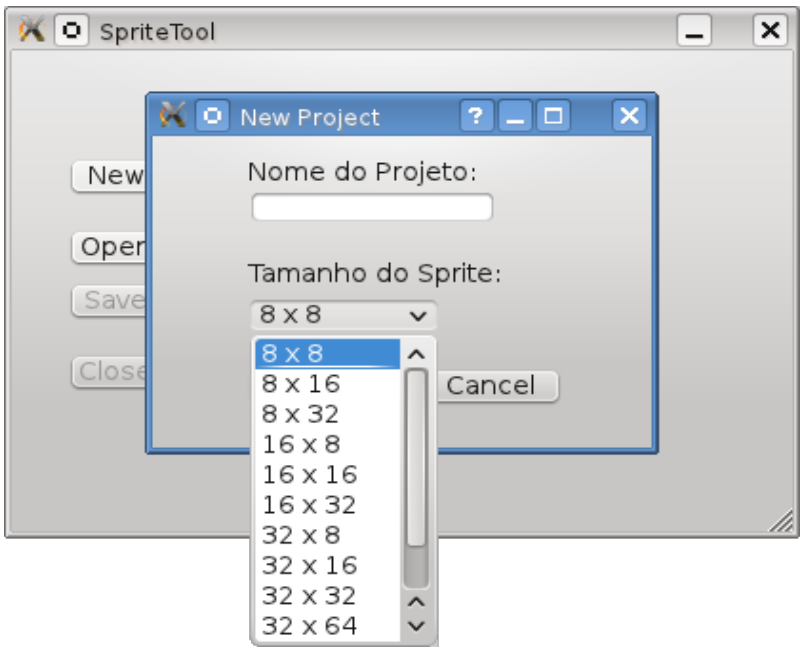

Figura 3.5: Janela de criação de novo *sprite*

Ao começar um novo projeto, uma instância de *sprite* é criada, mesmo que ainda não tenha sido adicionada nenhuma imagem. É importante salientar que o termo “sprite” neste escopo,

engloba uma série de conceitos que vão além de um simples arquivo de imagem. São eles: informações relativas ao tamanho da imagem, uma paleta de cores, informações sobre transparência e, por fim, os *frames* do *sprite* – que são as imagens em si. Algumas destas informações são requeridas para a criação do projeto de *sprite*, como o nome do *sprite* e as dimensões das imagens que servirão como *frames* da animação deste. O programa oferece uma gama de formatos de *sprite* para o usuário escolher, respeitando as restrições das bibliotecas e do hardware do DS sobre as imagens que podem ser usadas como *sprites*.

Criado um novo projeto de *sprite*, o usuário tem a possibilidade de adicionar novas imagens a ele. Além de permitir que o usuário busque uma imagem compatível com o tamanho já definido, o programa cria uma pasta de projeto e mantém nesta pasta uma cópia de cada imagem adicionada a ele, eliminando o uso de caminhos específicos para os arquivos fontes das imagens selecionadas. Assim, o programa tem uma maior portabilidade de uso. De forma análoga, as imagens relativas aos *frames* deletados do projeto são removidas desta pasta, que encapsula apenas os arquivos necessários para o funcionamento do SpriteTool.

Com a ferramenta, ainda é possível pré-visualizar a animação do *sprite*, provendo ao usuário uma ideia do seu comportamento dentro do jogo. Ainda que esta visualização não ofereça suporte aos eventos normalmente associados com cada parte da animação, como por exemplo o personagem pular ao ser apertado o botão de pulo, ela é um *feedback* importante para a definição da animação. Os frames são mostrados de forma contínua, na sequencia em que foram adicionados ao projeto, e isto permite o usuário verificar o quão fluídos os movimentos do personagem estão.

Depois de adicionar as imagens que vão compor o *sprite* e de validar o resultado da animação, o fluxo de trabalho para o usuário termina salvando o projeto e exportando para o formato compatível com as especificações de *sprite* do Nintendo DS e da PALib. Todo o processamento feito ao executar estas duas funções é transparente a ele.

Ao exportar, o SpriteTool traduz as informações do *sprite* para o formato que poderá ser usado no desenvolvimento do jogo dentro da *engine*. Para que isto aconteça, são necessários dois arquivos: nomedoprojeto_Pal.bin e nomedoprojeto_Sprite.bin

O primeiro arquivo contém as informações relativas à paleta de cores daquele *sprite*. Isto deve ser feito para cada *frame* contido no projeto, e acontece da seguinte forma:

Como o formato de cor aceito pelo Nintendo DS é o formato A1B5G5R5, citado em 3.2, a cor de cada pixel da imagem é pré-processada de forma a se adequar ao padrão. Depois disso, é preciso verificar se a cor já existe na paleta que está sendo criada. Para tal, é feito o uso de

uma tabela *hash*, tanto pela sua praticidade quanto pelo seu desempenho. Se a cor ainda não existe na paleta, ela é adicionada. Vale ressaltar que por convenção, a cor magenta representa a transparência e é a primeira cor a ser adicionada na paleta.

Após o processamento de todos os *frames* do projeto, o arquivo nomedoprojeto_Pal.bin está completo e pronto para uso na *engine*.

O segundo arquivo que é gerado nesta etapa, contem as informações referentes aos *frames*. Nele é descrito como cada *frame* é organizado, baseando-se nos índices das cores referenciadas na paleta do projeto. O formato no qual isto deve ser feito, exige que o programa itere os *frames* em blocos de 64 pixels (8 de altura por 8 de largura). O fim do processamento se dá junto ao término dos *frames* do projeto de *sprite*.

Com estes dois arquivos criados, a *engine* é capaz de reconhecer o *sprite* e animá-lo conforme conveniente.

Listing 3.2: Exportando o *sprite* para o Nintendo DS:

```
//exporting sprite
QFile file_spr(spr);

if(file_spr.open(QIODevice::WriteOnly | QFile::Truncate))
{
  QDataStream out(&file_spr);

  int paletteBias = 0;

  for(int it(0); it < m_sprite.size(); ++it)
  {
    QImage img = pImg.at(it);

    for(int i(0); i < m_sprite.get_height()/8; ++i)
    {
      for(int j(0); j < m_sprite.get_width()/8; ++j)
      {
        for(int ii(0); ii < 8; ++ii)
        {
          for(int jj(0); jj < 8; ++jj)
          {
            unsigned char color_index;

            if(img.pixelIndex(8*j+jj,8*i+ii) == 0)
              color_index = 0;
```

```
            else
              color_index = img.pixelIndex(8*j+jj,8*i+ii) + paletteBias;

            out.writeRawData((const char *)&color_index,1);
          }
        }
      }
    }

    paletteBias += img.colorTable().size();
  }
}

file_spr.close();
```

Para o desenvolvimento desta etapa, foi usado um editor hexadecimal de forma a facilitar a visualização dos arquivos gerados pela ferramenta e verificar a resposta que eles tinham na *engine*.

3.4 A integração entre as ferramentas

As ferramentas desenvolvidas neste trabalho têm como objetivo conjunto possibilitar o desenvolvimento de jogos no Nintendo DS. Estas ferramentas foram desenvolvidas de forma separada, cada uma sendo responsável por uma parte específica da cadeia de desenvolvimento. Desta forma, se faz a necessário estabelecer um meio de comunicação entre elas. É importante ressaltar que esta forma de comunicação deve ser robusta, para que permita, por exemplo, uma integração futura entre as ferramentas apresentadas neste trabalho, construindo assim um único ambiente de desenvolvimento.

Além disso, seja para fins de interromper momentaneamente o fluxo de trabalho em uma determinada etapa do desenvolvimento, ou guardá-lo para referência futura, fez-se necessário, também, estabelecer um meio de armazenamento da informação usada por cada uma das ferramentas.

Para resolver estas questões, foi escolhido o formato de arquivo XML como um meio intermediário entre as ferramentas, sendo responsável tanto para armazenar informações de configuração de cada uma delas, quanto para estabelecer uma possível comunicação entre elas. Este formato é amplamente usado hoje em dia e foi escolhido pela facilidade de uso e legibilidade que traz consigo.

Os arquivos XML são compostos por diversas *tags*, responsáveis por delimitar um determinado espaço no arquivo onde se encontra a informação referente a elas. Dentro da área delimitada por uma *tag*, existe a possibilidade de criarmos mais um nível de profundidade no arquivo, estabelecendo assim um sistema hierárquico dentro dele. Dessa forma, este tipo de arquivo pode ser entendido como uma árvore, onde cada *tag* representa um nó, e cada nível hierárquico abaixo dele representa um filho. Isto traz a vantagem de introduzir diferentes ramos, cada um contendo um tipo específico de informação, todos no mesmo arquivo. Assim, ferramentas distintas têm suas informações completamente separadas, mas todos os dados relativos a um mesmo projeto se encontram no mesmo arquivo.

Depois de escolhido o formato XML, foi criado um protocolo de escrita e leitura a ser respeitado por todas as ferramentas.

Listing 3.3: Formato de XML de integração entre as ferramentas

```
<?xml version="1.0" encoding="UTF-8"?>
<Projeto name="Projeto 00">
  <BgTool>
    <Levels>
      <Level width="8" height="8" name="Projeto 00">
        <Backgrounds>
          <Background id="0" path="bg0"/>
          <Background id="1" path="bg1"/>
        </Backgrounds>
      </Level>
    </Levels>
  </BgTool>
   <SpriteTool>
     <Sprite name="Mario" sizex="16" sizey="16">
        <Frame id="1" path="frame01.png"/>
        <Frame id="2" path="frame02.png"/>
        <Frame id="3" path="frame03.png"/>
        <Frame id="4" path="frame04.png"/>
        <Frame id="5" path="frame05.png"/>
     </Sprite>
     <Sprite name="Luigi" sizex="16" sizey="16">
        <Frame id="1" path="luigi01.png"/>
     </Sprite>
   </SpriteTool>
</Projeto>
```

4 A Engine

"*Quote 4*"
Author4

Neste capítulo é apresentada a *engine* feita para o Nintendo DS e as decisões tomadas sobre ela.

4.1 Introdução

No capítulo anterior, foi demonstrado como este trabalho resolve a primeira parte do processo de criação do jogo: a criação de seus personagens e *backgrounds*. Ficam então alguns desafios pela frente: a detecção de colisões, a constituição do mundo e a interligação das fases, isto é, o próprio desenrolar do jogo.

Para resolver esses desafios, se faz necessária a criação de alguma ferramenta no Nintendo DS. Não é a intenção deste trabalho apresentar uma *engine* completa, no entanto, é pretendido se criar uma espécie de *engine* que consiga usar-se dos personagens e *backgrounds* criados com as ferramentas previamente apresentadas.

Podemos estabelecer que a *engine* criada é, na verdade, um *template*, que pode (e deve) ser modificado pelo programador para torná-la adequada ao jogo em desenvolvimento.

4.2 O *Loop* de jogo

Podemos encontrar, em bibliotecas de jogos, um *loop* de jogo bastante comum, que é o mesmo que usaremos:

Listing 4.1: Loop de jogo

```
void run()
{
    init();

    for(;;)
    {
        PA_WaitForVBlank();
        render();

        if(!update())
            break;
    }

    cleanup();
}
```

Nesse *loop*, a ideia é que o usuário possa trabalhar com a seguinte separação lógica de código:

Em *init* ficam as tarefas a serem realizadas antes do início do jogo. Lá se encontram o carregamento e organização das imagens e sons a serem usados no jogo.

Em *update* ficam as tarefas a serem realizadas ao fim de cada iteração do *loop*. Normalmente, acelerações advindas da gravidade e de outras forças existentes no ambiente, detecções e tratamento de colisões, avaliação da inteligência artificial, dentre outras.

Em *render* se encontram as rotinas de desenho na tela.

Essa separação, como dito anteriormente, é lógica. Não há restrição a colocar o código de avaliação de inteligência artificial na função de renderização, porém, no formato sugerido o código ganha em organização e legibilidade.

4.3 Colisões

O tratamento de colisões é um grande desafio para qualquer simulação. No caso de jogos, há adicionalmente a necessidade de manter a taxa de atualização com a qual o jogador está acostumado. É necessário, apesar das várias tarefas que precisam ser processadas, que o tempo entre elas seja dividido de um modo que nenhuma delas sofra consideravelmente com a perda de tempo. Como o usuário espera por uma taxa de atualização constante, se uma das tarefas consome todo o tempo entre atualizações, as outras são prejudicadas.

Foi feito uso de um método de colisão *a posteriori*, isto é, após calculadas todas as atualizações dos personagens, cenário e itens, suas novas posições são usadas para verificar a existência de uma colisão. Caso ela tenha acontecido, um passo adicional é executado: o de tomar alguma ação para tratar essa colisão.

No caso de uma colisão com um objeto sólido, como uma parede, o objeto é recolocado em sua última posição válida. No caso da colisão entre o personagem e um inimigo, é feita uma verificação para averiguar qual dos dois foi afetado, e algum tipo de dano pode ser associado a ele. O número de situações possíveis, tal como o tratamento específico a cada uma delas, varia de acordo com o jogo que está sendo desenvolvido e o *gameplay* que se deseja implementar nele.

Para o algoritmo que detecta a colisão, usamos uma implementação do teorema dos eixos separados, como pode ser visto em [Eberly 2008], que dita que dadas duas formas convexas, existe uma linha em que suas projeções vão ser separadas se e somente se eles não estão interceptando. Se uma das formas não for convexa, o teorema não é válido. Entretanto, este caso não é pertinente ao desenvolvimento do jogo e o teorema é aplicável.

4.4 Uso e configurações

O desenvolvedor precisa informar para a *engine* quais arquivos compõem as fases e quais arquivos são de personagens e inimigos. Isso deve ser feito através do código; para manter a organização, sugerimos que cada fase tenha um arquivo deste tipo. Por padrão, a *engine* chama uma função *load_resources()*, que pode ser implementada em qualquer arquivo, contanto que ele seja compilado junto ao resto do projeto.

Desse modo, fica a cargo do programador chamar, dentro da função *load_resources()*, funções próprias como por exemplo *load_stage1()* e *load_stage2()*. O controle da *engine* é feito através de estruturas existentes, e é nessas funções que o programador precisa preenchê-las.

Listing 4.2: Estrutura que deve ser povoada para cada fase

```
struct sLevelData
{
  std::vector<const PA_BgStruct *> m_backgrounds;
  std::vector<sEnemyData *> m_enemies;
  std::vector<sItemData *> m_items;
  int m_scrolled;
  int m_world_min_height;
  int m_world_min_width;
  int m_world_max_height;
  int m_world_max_width;
};
```

4.5 Conexão entre fases

As fases são interligadas a partir das estruturas que foram preenchidas na função *load_resources()*. A *engine* liga as fases sequencialmente, e uma vez que o jogador passou de uma fase, não há meio para retornar para ela.

5 O Jogo

"*Quote 5*"
Author5

Neste capítulo é apresentado um jogo criado usando as ferramentas.

5.1 Aplicando as ferramentas

Depois de desenvolvidas as ferramentas e a *engine*, é preciso testá-las. Para tanto, escolhemos desenvolver um clone do jogo Super Mario Bros. original, de modo que possamos testar nossas ferramentas, sem no entanto, termos que nos preocupar em gerar novos gráficos, tarefa esta, que não está no escopo deste trabalho.

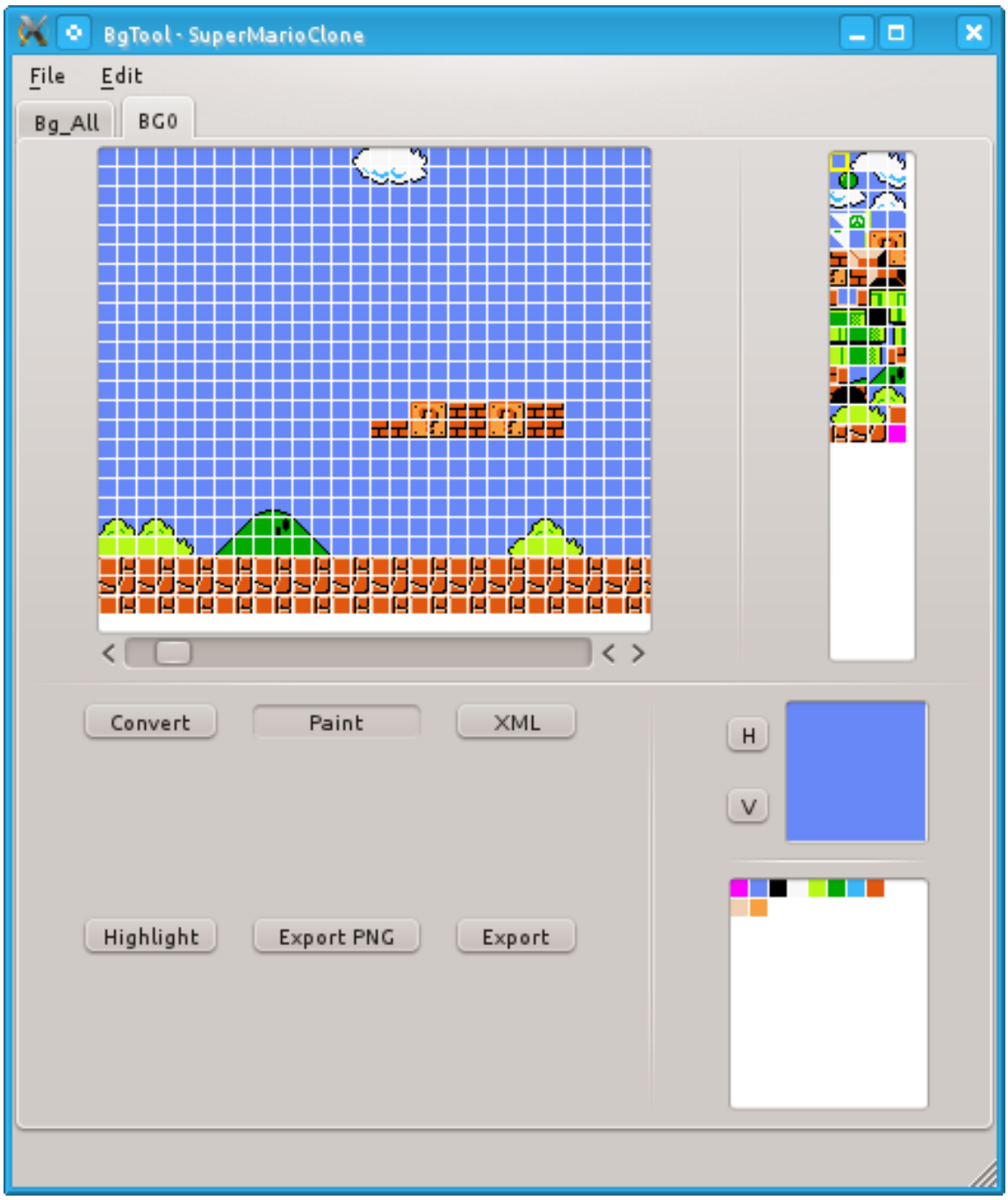

Figura 5.1: Apagando os objetos depois de importar a imagem.

Começamos importando uma imagem da primeira fase do jogo original na BGTool, criando

um mapa da primeira fase, e editamos o mapa para retirar objetos que não são do *background*, como as caixas de interrogação. E usamos a ferramenta para exportar para o DS. Repetimos esse processo para as fases que incluímos neste nosso jogo de demonstração.

O segundo passo é recriar os sprites originais, e converter para o formato aceito pelo DS, dentre eles sprites de inimigos, da caixa de ítens, da bandeira, dentre outros.

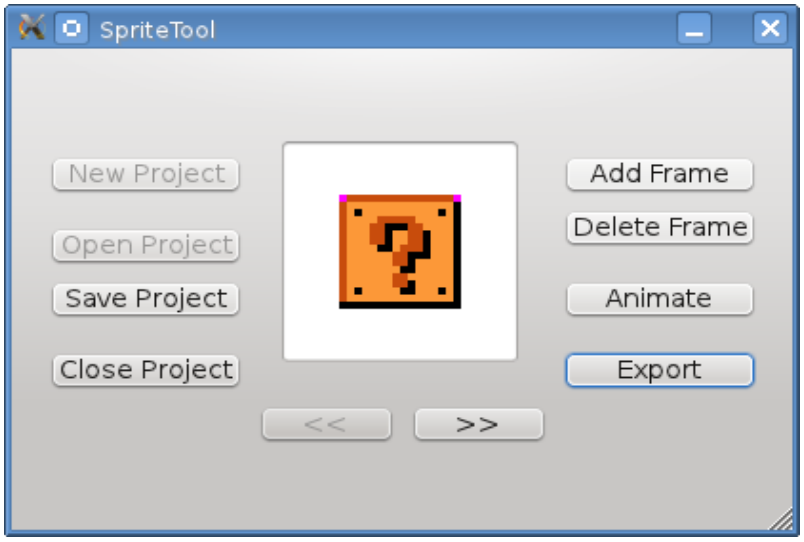

Figura 5.2: Adicionando o *sprite* do bloco de interrogação à SpriteTool.

Em seguida, criamos os arquivos de código fonte que informam para a *engine* sobre os *backgrounds*, personagens, inimigos e tudo que criamos nos passos anteriores.

Listing 5.1: Função que carrega a primeira fase

```
void load_level1(std::vector<sLevelData *>& level_data_vector)
{
  sLevelData *level1_data = new sLevelData();
  level1_data->m_world_min_height = 0;
  level1_data->m_world_min_width = 0;
  level1_data->m_world_max_height = level1_0.height*multiplier;
  level1_data->m_world_max_width = level1_0.width*multiplier;
  level1_data->m_scrolled = 0;
  level1_data->m_backgrounds.push_back(&level1_0);

  sItemData *itembox1 = new sItemData();
  itembox1->m_x = 30;
  itembox1->m_y = 9;
  itembox1->m_sprite_pointer = itembox_Sprite;
  itembox1->m_palette_pointer = itembox_Pal;
  itembox1->set_size(OBJ_SIZE_16X16);
  itembox1->m_item_type = ITEMBOX;
```

```
  level1_data->m_items.push_back(itembox1);

  (... Outros itens inseridos do mesmo modo ...)

  sEnemyData *goomba1 = new sEnemyData();
  goomba1->m_x = 30;
  goomba1->m_y = 3;
  goomba1->m_sprite_pointer = goomba_Sprite;
  goomba1->m_palette_pointer = goomba_Pal;
  goomba1->set_size(OBJ_SIZE_16X16);
  goomba1->m_enemy_movement = STRAIGHT_FALL;

  level1_data->m_enemies.push_back(goomba1);

  (... Outros inimigos inseridos do mesmo modo ...)

  level_data_vector.push_back(level1_data);
}
```

Por fim, ficamos com algo assim:

Figura 5.3: O jogo rodando na engine, após ter suas imagens importadas para o DS.

6 Conclusões e trabalhos futuros

"*Nada se cria, nada se perde, tudo se transforma.*"

Lavousier

Neste capítulo, são apresentadas as conclusões e alguns trabalhos futuros ...

6.1 Conclusões

O mercado de jogos está finalmente chegando em sua maturidade. Jogos feitos por pequenos grupos de desenvolvedores estão alcançando sucesso tão grande quanto grandes franquias, criadas por grandes empresas. Como exemplo disso, podem-se citar jogos como Minecraft, cuja maior parte foi criada por apenas um desenvolvedor, que há pouco tempo alcançou a marca de mais de 2 milhões de cópias vendidas. [Markus Persson 2011]

Com esse cenário, podemos notar que é importante tentar reduzir o tempo gasto mexendo em código fonte, para que os desenvolvedores possam se focar naquilo que realmente faz diferença em um jogo: a originalidade e o seu *gameplay*.

A maior dificuldade nesse tipo de projeto, como citado anteriormente, é o acesso às funções de hardware dos consoles. Entretanto, é possível perceber um movimento das grandes empresas em direção a consoles com desenvolvimento mais aberto, como por exemplo o Xbox Live Indie Games, onde o desenvolvedor usa o XNA Game Studio para desenvolver e recebe 70% do lucro do lançamento do seu jogo, depois de alguns passos que conferem ao jogo a possibilidade de ser lançado. Ainda assim, há um custo anual para o desenvolvedor de $99 USD. [Microsoft 2011]

As ferramentas desenvolvidas nesse trabalho têm como objetivo ser apenas uma ideia do que os autores pensam ser um caminho para reduzir esse gasto, fazendo com que seja possível desenvolver jogos bem diferentes com pequenas alterações de código. Embora com essas ferramentas não seja possível publicar um jogo, há a possibilidade de um desenvolvedor individual utilizá-las para desenvolver seu protótipo de jogo, ou para um jogo que não tenha intenção de ser vendido na plataforma.

Podemos dizer que ainda há muito a ser feito para essas ferramentas. Poderíamos, por exemplo, escrever mais funções que exportem para diferentes cadeias de ferramentas, de diferentes sistemas. Não nos limitando, então, somente ao Nintendo DS. Deste modo, poderia ser desenvolvido um jogo de diversas plataformas com um mínimo de trabalho.

Além disso, certas tarefas podem ser mais automatizadas, como a geração do arquivo que carrega os *backgrounds* e personagens na *engine* do DS. Como tem um formato bem definido, ele poderia ser gerado automaticamente por uma terceira ferramenta, desde que conseguisse ler o formato XML definido neste trabalho.

A colisão precisa ser definida manualmente na *engine*, mas isso poderia ser feito diretamente na BGTool, ao importar os *tiles*. Assim, o artista que estivesse criando os *backgrounds* poderia já escolher o tipo de colisão dos seus trabalhos.

A interface da BGTool ainda pode ser melhorada, para facilitar o seu uso. Com operações de pintura em grupo, preenchimento de regiões por difusão (*flood-fill*), e outras operações comuns em ferramentas de desenho.

Mais ferramentas poderiam ser criadas, cada uma orientada a partes da cadeia de desenvolvimento que não foram abordadas. Um módulo para tratar da inteligência artificial dos inimigos é um exemplo, possibilitando o usuário a definir para cada *sprite* de inimigo um padrão de movimentação já previamente estabelecido.

Uma outra funcionalidade do Nintendo DS, que é a conexão *wireless*, não foi aproveitada da form a que poderia. Um módulo para estabelecer a comunicação entre dois consoles distintos poderia ser criado aproveitando a facilidade de comunicação que o aparelho oferece.

Parte do objetivo deste trabalho é justamente fomentar o desenvolvimento desse tipo de ferramenta, e assim aquecer o desenvolvimento de jogos apesar de todas as dificuldades que ainda são encontradas na área. As possibilidades de trabalhos futuros são inúmeras e além de nos inspirarem, fazem de nós pessoas mais realizadas por atingir esta meta. Este trabalho foi desenvolvido para a obtenção de grau em Bacharel em Ciências da Computação, entretanto, nossas aspirações como autores vão além: impactar o cenário atual de desenvolvimento de jogos e servir de referência para tantos outros interessados nesta área que ainda carece muito de incentivo.

Referências Bibliográficas

[Eberly 2008]EBERLY, D. *"Intersection of Convex Objects: The Method of Separating Axes"*. 2008. Disponível em: <http://www.geometrictools.com/Documentation/MethodOfSeparatingAxes.pdf>.

[Goldsmith Jr. et al. 1948]Goldsmith Jr., Thomas T., Ray e Mann Estle. *Cathode-ray tube amusement device*. December 1948. 2455992. Disponível em: <http://www.google.com/patents?vid=2455992>.

[Katie Salen e Eric Zimmerman 2006]Katie Salen; Eric Zimmerman. Adventure as a video game. adventure for the atari 2600. [1983-84]. In: *The Game Design Reader: A Rules of Play Anthology*. [S.l.]: MIT Press, 2006. p. 690–713.

[Markus Persson 2011]Markus Persson. *Minecraft Stats*. 2011. Disponível em: <http://www.minecraft.net/stats.jsp>.

[Microsoft 2011]Microsoft. *App Hub: How it works*. 2011. Disponível em: <http://create.msdn.com/en-us/home/about/how_it_works>.

ANEXO A – Ferramentas utilizadas

A.1 QT