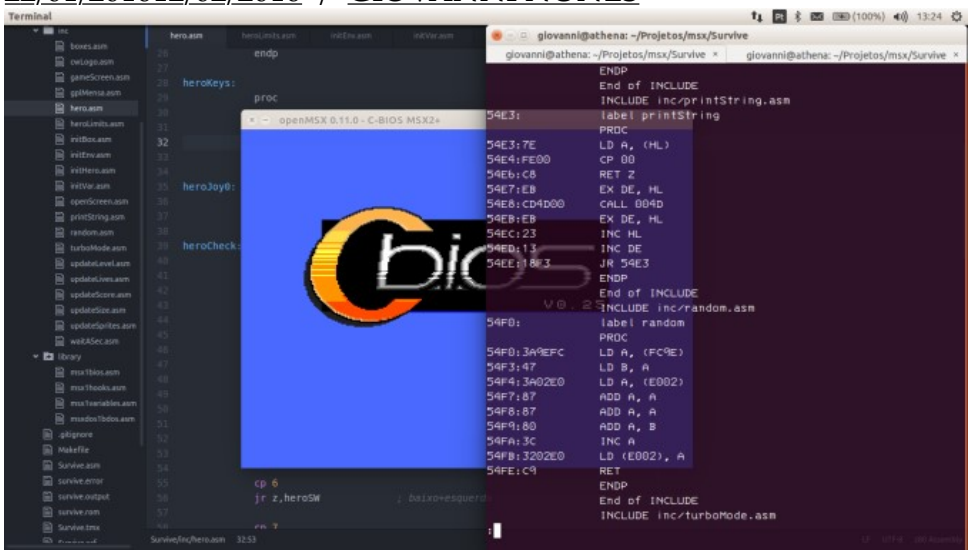
Escrevendo um jogo para MSX – parte 1

22/01/201612/02/2016 / GIOVANNI NUNES



E agora para algo completamente diferente e em comemoração ao sexto aniversário do Retrocomputaria (http://www.retrocomputaria.com.br/) (ocorrido no último 20 de janeiro) uma série de postagens detalhando o desenvolvimento de um jogo para **MSX** a partir de um descendente (in)direto das antigas workstations (https://en.wikipedia.org/wiki/Workstation) baseadas em UNIX: um notebook rodando **Linux**. 🙂

Óbvio que minha ideia não é produzir o "*Guia definitivo sobre desenvolvimento de jogos em plataformas clássicas*" ou mesmo o "*Grande tutorial unificado de programação assembly para computadores MSX*", é coisa bem mais modesta. É detalhar o processo de criação de um jogo para **MSX** feito (quase que totalmente) do zero, indicando as ferramentas, os passos necessários e até dando algumas dicas práticas.

O jogo

A partir da minha primeira experiência de animação de *sprites* na tela resolvi seguir adiante e fazer logo um *demake* do Survival (https://www.ouya.tv/game/Survival/) — jogo de visual retro escrito por **Mason Barry** (o **Rubix**) para o **OUYA** mas também disponível no Google Play (https://play.google.com/store/apps/details?id=com.joseki.androidsurvival) para dispositivos **Android**.

No jogo você basicamente deve "comer" os quadrados menores ou de igual tamanho que vão cruzando a tela ao mesmo tempo que evita ser comido pelos maiores. Conforme se alimenta, crescerá e quando alcançar o tamanho máximo o jogo recomeçará em uma nova fase e com maior velocidade.

O jogo no MSX

Transferindo o conceito do jogo para o **MSX**, temos cinco padrões de *sprites* em modo ampliado (o tamanho de 16×16 é duplicado para 32×32) e como são caixas ninguém perceberá a enganação.

Também são cinco objetos na tela:

- O "herói" que você move pela tela usando as setas de cursor ou os *joysticks* e
- As quatro "caixas" que cruzam a tela da esquerda para a direita ou da direta para a esquerda em diversas velocidades.

A resolução do jogo é a padrão do **MSX** (256×192 pixeis — 32×24 caracteres) e assim como no jogo original só precisarei de dois sons:

- Um para a morte do herói e
- Outro para troca de nível.

Também preciso de um nome então, em um surto de assustadora originalidade, vou chamá-lo de "Survive"! 🙂

Pensando a ação

Antes de começar a escrever qualquer coisa que se pareça com código, gráficos ou sons é bom levantar questões importantes (as "premissas do projeto") a respeito do que será realmente feito (isto é "entregue"):

i. Qual a linguagem de programação?

Poderia ser escrito em **MSX-BASIC** com ajuda do MSX-BASIC kun (http://www.generation-msx.nl/software/ascii/msx-basic-kun/721/) para acelerar algumas partes mas optei por fazer diretamente em *assembly* de Z80 (https://en.wikipedia.org/wiki/Zilog_Z80) — preciso exercitar um pouco.

ii. Quais os requisitos mínimos?

O jogo será feito para rodar em um **MSX** de primeira geração com 8KiB[1] de RAM e 16KiB de VRAM. Ou seja, dado um **MSX** nesta configuração o jogo rodará normalmente.

Minha única consideração até o momento é não me preocupar em fazê-lo compatível com os processadores de vídeo da **Toshiba**, os T6950.

[1] O valor correto aqui é **8** e não **4**, ou seja, o mínimo de memória RAM exigido para um computador **MSX** de primeira geração são 8KiB e não 4KiB. Foram poucos os modelos que saíram assim com tão pouca memória, um deles é o PV-7 (http://www.old-computers.com/museum/computer.asp?st=1&c=576) da **CASIO**.

iii. Qual o formato do executável?

Um programa feito para rodar diretamente na RAM, seja a partir do **MSX-DOS** ou mesmo do **MSX-BASIC**, poderá fazer "coisas (https://en.wikipedia.org/wiki/Self-modifying_code)" que um programa rodando na ROM não conseguirá. E um programa de **MSX-DOS** "enxerga" um computador diferente do que um programa executando pelo **BASIC**.

Esta parte eu explicarei depois, por enquanto o que vale a pena é saber que o jogo será uma imagem de cartucho, um ".rom".

iv. Desenvolvimento cruzado ou *in loco*?

Nos "velhos tempos" alguém (muito provavelmente eu) bradaria que somente o cartucho do **Mega Assembler** seria mais que suficiente. Mas sejamos práticos: o desenvolvimento será feito em hardware atual, usará um emulador para testar o código e, em caso de dúvida, verificar-se-á em um **MSX** real.

Além do mais nunca aprendi a usar o **Mega-Assembler** mesmo! 🙂

Outras questões podem ser levantadas e também resolvidas mas estas são as que impactam com maior intensidade o desenvolvimento.

Caixa de ferramentas

O que você precisa usar para fazer um jogo de **MSX**? Já que o desenvolvimento será feito diretamente em um "computador comum e corrente" (meu notebook) estas aqui são as ferramentas que vou usar:

- Atom (http://www.atom.io/) — Não deveria indicar um editor de textos mas resolvi dar uma chance para este sujeito aqui, só é necessário instalar o pacote language-z80asm (https://atom.io/packages/language-z80asm) para o correto realce de sintaxe.
- GIMP (http://www.gimp.org/) — Editor de imagens *bitmap*. A ideia é desenhar aqui e depois exportar ou desenhar no **MSX**.
- openMSX (http://openmsx.org/) — Emulador de **MSX** para testar o código e como o jogo será um "cartucho" a configuração padrão, usando a C-BIOS (http://cbios.sourceforge.net/), é suficiente.
- Pasmo (http://pasmo.speccy.org/) — Montador *assembler* que utilizo, comecei a usar porque já gerava automaticamente binários de **MSX** mas hoje é por pura inércia.

- Tiled Map Editor (http://www.mapeditor.org/) — Programa específico para o desenho de mapas e cenários em duas ou três dimensões.
- TinySprite (http://msx.jannone.org/tinysprite/tinysprite.html) — Excelente editor de *sprites* feito pelo Rafael Jannone (http://www.jannone.org/). É em JavaScript e roda direto pelo navegador, nem precisa ser instalado.

Todas as ferramentas listadas acima rodam tranquilamente em versões razoavelmente atuais de **Linux**, **Mac OS X** e até **Windows**.

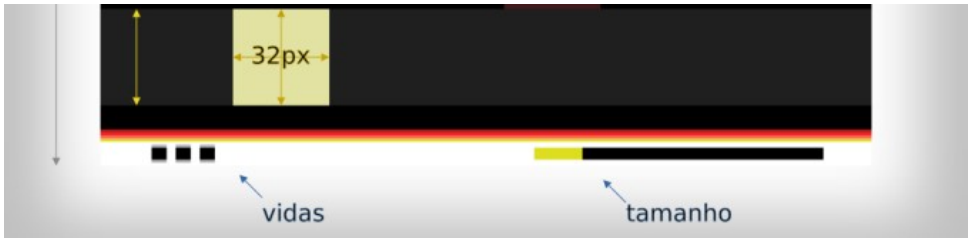
Mas antes de fechar a caixa de ferramentas dois itens importantes para ter a disposição:

- Fudeba Assembler : Manual de Referência da Arquitetura MSX(http://foca.sourceforge.net/) — O título é autoexplicativo, trata-se de um manual que abrange *assembly* de Z80, do essencial dos outros processadores do **MSX**.das rotinas e chamadas da **MSX-BIOS**, **MSX-DOS 1.x** e UZIX (http://uzix.sourceforge.net/index.php?lang=pt) e, claro, do próprio **Fudeba Assembler**.
- Z80 Instruction Set (http://clrhome.org/table/#) — Tabela online com as instruções do Z80, serve para você descobrir que não existe **LD HL,DE** e que precisará fazer **LD H,D** seguido de **LD L,E**! 🙂

Reparou na quantidade de coisas necessárias para se fazer um "simples" joguinho?

Esboçando o jogo

Tratando-se de um jogo a parte mais importante sempre acaba sendo a visual (os gráficos), então esbocei (não necessariamente de imediato) algo assim:



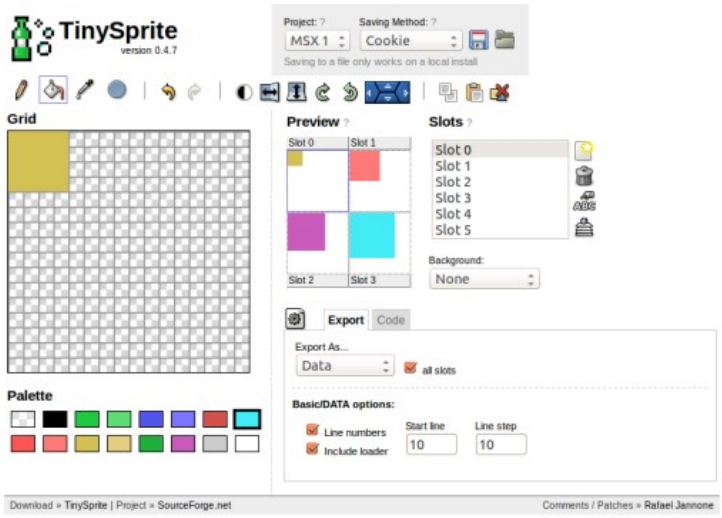
(https://giovannireisnunes.files.wordpress.com/2016/01/survive-1_conceito.png)

Como no jogo original a tela é quase autoexplicativa, pontuação e nível atual na parte superior em informações numéricas e quantidade de vidas (um quadrado para cada) e tamanho corrente (ajudando a indicar pelas cores quais "caixas" podem ser comidas) na parte inferior. O uso da borda na cor branca é para simular a moldura lateral da área do jogo e manter a área do jogo retangular, como no original.

O sombreado em cada caixinha indica como é o desenho dos *sprites* na **VRAM**, originalmente coloquei todas centralizadas mas rapidamente percebi da besteira que estava fazendo, principalmente na parte de restringir a movimentação do "herói" pela tela. As listras horizontais são só a indicação dos "caminhos" que as "caixas" percorrerão.

Desenho dos *sprites*

Começando pelo mais simples, ou seja, os *sprites*. Como são apenas quadrados a tarefa de desenhá-los é pouco trabalhosa e exige a coordenação motora necessária para produzir linhas retas com o mouse.



(https://giovannireisnunes.files.wordpress.com/2016/01/survive-1_sprites.png)

Ao todo são cinco *sprites* diferentes representando quadrados nos tamanhos de 4×4 (amarelo ouro), 8×8 (vermelho claro), 10×10 (magenta), 12×12 (ciano), 14×14 (verde claro) e 16×16 (amarelo claro). Bom lembrar que os *sprites* serão ampliados, logo com as dimensões duplicadas e que o "herói" utilizará os mesmos gráficos.

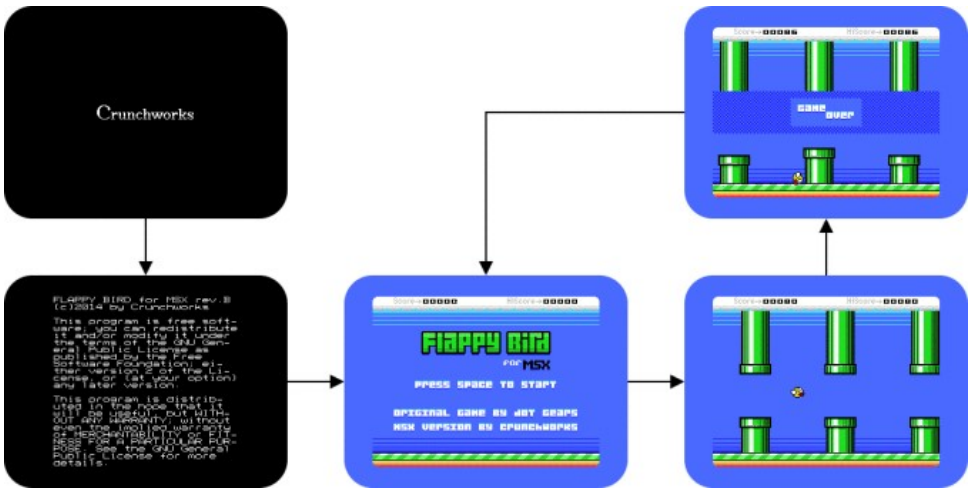
O **TinySprite** exporta os desenhos em um formato que o **Pasmo** reconhece normalmente. Aqui os gráficos da caixa em 4×4 (que será 8×8 na tela):

```
db 0xF0,0xF0,0xF0,0xF0,0x00,0x00,0x00,0x00
db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
db 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
```

Aliás os gráficos do Flappy Bird (https://giovannireisnunes.wordpress.com/meu-software/flappy-bird-para-msx/) também foram desenhados nele!

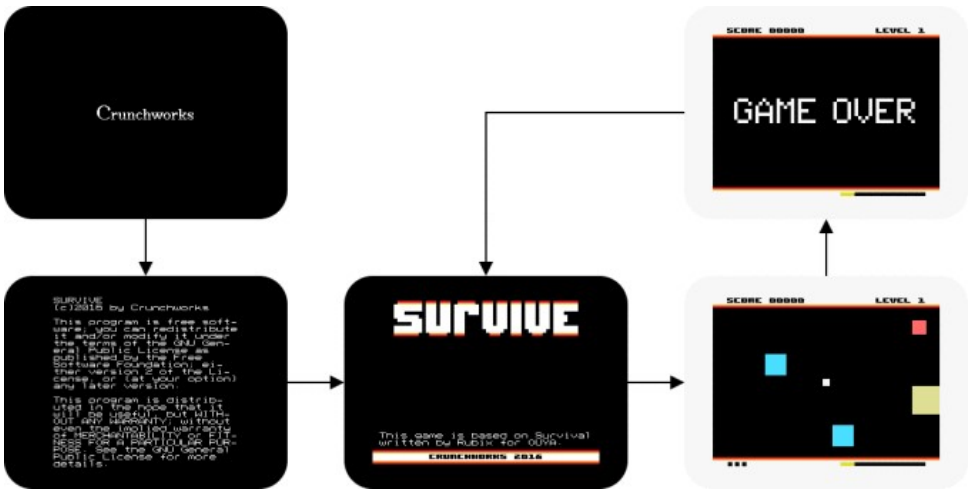
Sequência das telas

Sob o ponto de vista funcional, bastaria entregar a tela principal do jogo e fim da história mas não é bem o caso, é necessário apresentar algumas informações adicionais. Esta é a sequência de telas utilizada no meu **Flappy Bird**:



A tela de **introdução** com o "Crunchworks", a **informação** sobre o licenciamento na GPL (que sou obrigado a informar), a tela de **abertura** do jogo, o **jogo** propriamente dito e outra avisando do **final** dele e que segue para a abertura.

Logo basta fazer parecido:



A tela de abertura conterá bem mais informações do que tem agora (instruções básicas do jogo e a opção de seleção do nível inicial). Aliás, escolha um jogo qualquer de **MSX** e repare que ele também não fugirá muito disto.

Qual o propósito disto?

Colocar a ordem como as coisas serão colocadas na tela ajuda a esboçar o "algoritmo" do jogo, ou seja:

1. Inicializa o ambiente e as variáveis
2. Apresenta tela de introdução
3. Apresenta a informação da licença

4. Exibe a tela de abertura
5. Rotina principal do jogo
6. Rotina de "final do jogo"
7. Volta para o item 4

Também saber o que será exibido, simplifica a tarefa de desenhar os gráficos do jogo porque já se tem ideia do que se precisa fazer — ou seja, nada de perder tempo desenhando castelo, dragão, princesa etc 🙂

Desenho dos caracteres

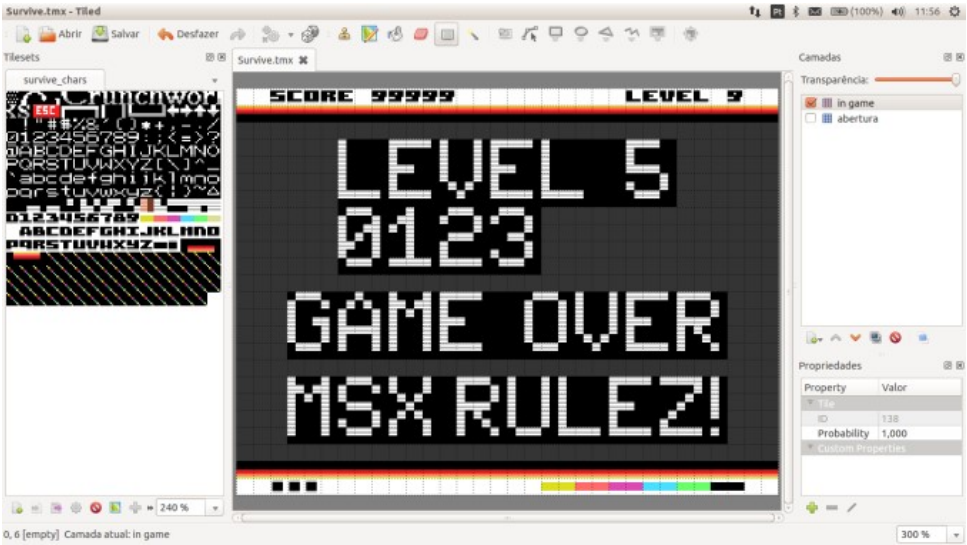
Como optei por "não inventar" utilizei diretamente o **Gimp** para esboçar os gráficos que precisava antes de desenhá-los no **MSX**. E por justamente ter decidido "não inventar" (estou enfatizando, não sendo repetitivo) a tarefa foi rápida e acabou quase um trabalho de copiar e colar.

A primeira metade dos 256 caracteres (0-127) peguei do **Flappy Bird**, os blocos semigráficos (128-143) para desenhar os títulos no Retromania (https://www.msx.org/es/node/40682#comment-204435) — que segue a mesma ordem do MC6847 (https://en.wikipedia.org/wiki/Motorola_6847). A parte específica do jogo (144-194) eu adaptei de algo que já tinha e o restante (195-255) ficou sem uso.

Para um pouco de luz sobre este assunto recomendo a leitura das partes 1 (http://www.retrocomputaria.com.br/2013/04/07/msx-as-sete-faces-da-screen-1-parte-1/), 2 (http://www.retrocomputaria.com.br/2013/04/09/msx-as-sete-faces-da-screen-1-parte-2/) e 3 (http://www.retrocomputaria.com.br/2013/04/30/msx-as-sete-faces-da-screen-1-parte-3/) do meu "MSX: As sete faces da SCREEN 1" publicado no **Retrocomputaria** em abril de 2013.

Finalizando com o Tiled

A função básica do Tiled (http://www.mapeditor.org/) é editar mapas mas por conta dos recursos dele o utilizei para fazer o título "SURVIVE" e alguns outros com os blocos semigráficos.



(https://giovannireisnunes.files.wordpress.com/2016/01/survive-1_tiled1.png)

A tabela de caracteres foi montada em uma imagem de 128×128 pixeis (16×16 caracteres), o tamanho de cada bloco como 8×8 pixeis (1 caractere) o tamanho da "tela" definido como 32×24 caracteres e o **Tiled** soube como se virar com o resto.

Como ele exporta os mapas em formatos como **CSV** ou **JSON** fica fácil jogar o trabalho diretamente no código:

```
openScreenCredits:
  db 190,190,190,190,190,190,190,190,190,190,190,190
  db 190,190,190,190,190,190,190,190,190,190,190,190
  db 160,160,160,160,160,160,160,160,163,178,181,174
  db 178,171,179,160,146,144,145,150,160,160,160,160
  db 193,193,193,193,193,193,193,193,193,193,193,193
  db 193,193,193,193,193,193,193,193,193,193,193,193
```

Ainda é necessário alguma formatação manual. como colocar os "db" e quebrar as linhas para facilitar a leitura mas não nada comparado a fazer a mesma coisa manualmente.

Tem prévia?

Tem sim! É claro que comecei a escrever isto bem depois do desenvolvimento iniciado, logo o estado atual está (mais ou menos) assim:

Comparando com aquele primeiro exemplo (https://giovannireisnunes.wordpress.com/2016/01/04/movendo-sprites-no-msx/) é possível perceber que a rotina de animação está diferente, sem (muitos) glitches (http://orig06.deviantart.net/d29c/f/2012/338/4/a/i_tell_herglitch_glitch_glitch_glitch_make_me_rich_by_limey404-d5n1j0e.gif)… 🙂

Fim da primeira parte

Encerrando a primeira parte por aqui, a próxima não será assim tão extensa mas será um pouco mais técnica e terá até algum código pois é preciso fazer as coisas caberem corretamente na memória do **MSX**.

Programação

ASSEMBLY , **JOGOS** , **MSX** , **RETROCOMPUTARIA** , **SURVIVE** , **Z80**

5 comentários sobre “Escrevendo um jogo para MSX – parte 1”

1. Pingback: Escrevendo um jogo para MSX usando Linux - Peguei do

2. ***Alexandre de Arruda Paes***
   25/01/2016 ÀS 14:03
   Sensacional meu caro !!!! Saudade daqueles áureos tempos de MegaAssembler!

3. ***Alfredo Jr.***
   25/01/2016 ÀS 14:30
   Sensacional o post.
   Parabéns pela iniciativa e vamos aguardar as outras partes. =)

4. Pingback: Escrevendo um jogo para MSX – parte 2 | giovannireisnunes

5. Pingback: Survive — Colisão de sprites II | giovannireisnunes

Os comentários estão desativados.