A Última Chance de Mário - A Jornada de OAC

Maxwell Oliveira dos Reis [*] João Gilberto de Oliveira Teixeira, [†] Lucas Sala Cruz [‡]
Emanuel de Oliveira Barbosa [§] Débora Venturelli Machado [¶]

Universidade de Brasília, 24 de julho de 2023

RESUMO

O projeto "A Última Chance de Mário - A Jornada de OAC"é uma versão modificada do jogo Super Mario Bros., adaptada para a ISA RISC-V da linguagem assembly e para o cotidiano do aluno da Universidade de Brasília. A história, contada dentro do jogo através de telas de narração, consiste na tarefa do aluno Mário de conseguir cumprir com seu prazo do projeto de OAC mesmo tendo dormido nas aulas presenciais e não compreendido corretamente todas as instruções do projeto. O jogo apresenta um background móvel que se mexe de acordo com a movimentação do personagem Mário, essa por sua vez guiada pelas teclas W, A e D, associadas ao pulo e à movimentação lateral de Mário. O personagem Mário possui um status de Vida, o qual é perdido quando ele entra em contato com o (por vezes) tóxico Suco do RU, um dos inimigos da saga. A única arma de Mário consiste em seu próprio pulo e sua força de vontade em conseguir entregar o projeto e não decepcionar seus colegas e professores.

Palavras-chave: Organização e arquitetura de computadores · Assembly RISC-V · Super Mario Bros · Desenvolvimento de jogos

1 INTRODUÇÃO

O clássico Super Mario Bros., desenvolvido em 1985 pela multinacional Nintendo Co., Ltd. para NES, foi usado como base fundamental para o desenvolvimento do jogo A Última Chanced de Mário - A Jornada de OAC. Os sprites, o mapa e a ideia de movimentação, entre outros, foram fundamentalmente criados a partir do Super Mario Bros. Nesse contexto, utilizamos da linguagem Assembly em sua ISA RISC-V para adaptar o jogo para a plataforma RARS e suas ferramentas de reprodução gráfica, sonora e de utilização do teclado.

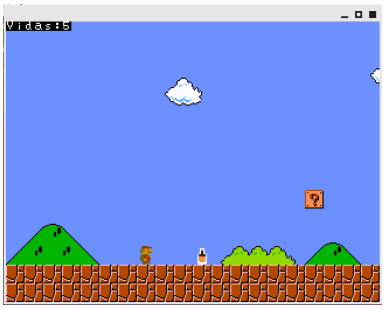

Figura 1: *Primeiro frame do jogo*

[*]221002100@aluno.unb.br
[†]211036070@aluno.unb.br
[‡]211010477@aluno.unb.br
[§]211010403@aluno.unb.br
[¶]190086238@aluno.unb.br

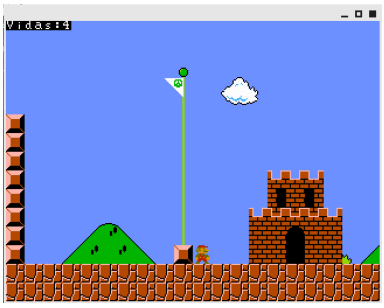

Figura 2: *Frame proximo a chegada*

2 CONTEXTO E TELAS

Assim que o jogo é iniciado, existem 7 frames com texto que servem para explicar o contexto do jogo ao jogador. Em resumo, Mario é um aluno de Lamar e deseja cumprir com os prazos do trabalho passado, porém Mario dormiu em algumas aulas e perdeu informações úteis sobre os prazos!

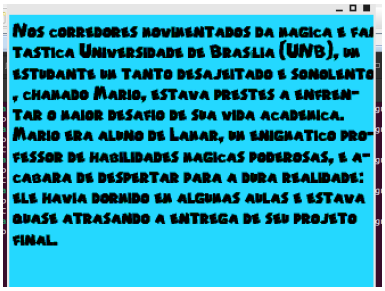

Figura 3: *Primeiro frame apresentado a história*

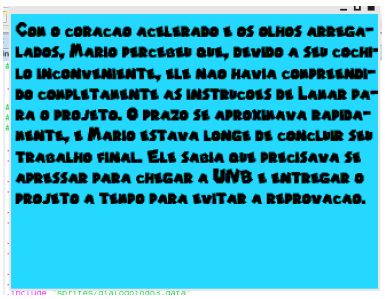

Figura 4: *Primeiro frame apresentado a história*

Além da história de Mario, o jogo também conta com um inimigo que é citado na história: O Suco do RU! Segundo a lenda, ele pode

ou não ser bom. Na implementação utilizada no jogo, ao tomar um Suco do RU, o personagem principal perde uma vida!

Figura 5: *Primeiro tipo de suco*

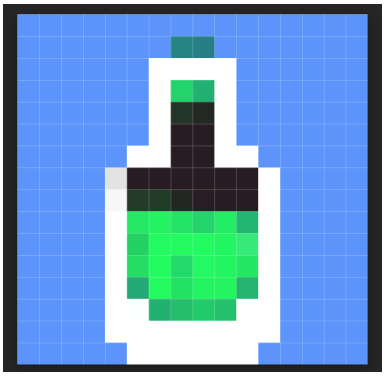

Figura 6: *Segundo tipo de suco*

Por fim, o jogo também conta com duas outras telas: Uma para quando o jogo termina em vitória e outro quando termina em derrota.

Figura 7: *Tela de vitória (creditos dos criadores)*

Figura 8: *Tela de derrota*

3 CÓDIGO DO PROJETO

O jogo roda com sua função principal chamada GameLoop. Essa função é responsavel por fazer tudo acontecer na ordem escrita. Portanto, o seu código é minusculo, apenas chamando outras funções.

Listing 1: *Função principal do projeto*

```
        .text
        GameLoop:
                    call LifeDisplay
                    call Midi
                    call Jump
                    call Tec
                    call CheckDamage
                    call CheckWin
                      j GameLoop
```

Mesmo com a função principal do jogo sendo pequena, o código total é imenso, contando com 23 arquivos de códigos.

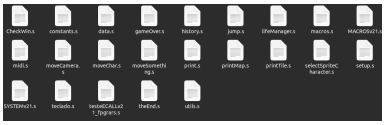

Figura 9: *Arquivos de código do jogo*

Além destes arquivos específicos para fazerem o jogo funcionar, também há outros códigos escritos em python que foram utilizados para auxiliar na criação do projeto. Das ferramentas citadas, temos o TileMaker.py e o CreateMapHitBox.py. O TileMaker é responsável por receber uma matriz com cores e gerar um mapa onde cada cor dada vira uma tile completa. O CreateMapHitBox.py é mais complexo e será melhor explicado mais adiante.

3.1 LIFEDISPLAY

Das funções citadas, o LifeDisplay é é responsável por manter escrito na tela a string "Vidas: x", onde x representa a quantidade de vidas atual do jogador. Esta função utiliza dos ecalls 104 e 101 para desenhar uma string e um int na tela, respectivamente.

3.2 MIDI

A função Midi é responsável por manter a música do jogo tocando sem que isso atrapalhe a gameplay. Ou seja, a musica não pode 'travar' o jogo para tocar. Isso foi feito salvando na memória a nota atual que está sendo tocada e até quando esta deve continuar tocando. Com isso, basta comparar o tempo atual com o tempo salvo para saber se já é necessário tocar a próxima nota. Esta utiliza o ECALL 31 chamado "MidiOut".

Outro ponto importante desta função é que após um tempo o som do jogo pode se tornar cansativo para o jogador, pois é repetitivo e pode ser que o jogador demore um tempo consideravel para terminar o jogo. Sendo assim, também foi implementada uma função que liga/desliga o som do jogo, basta pressionar a tecla 'M'.

3.3 JUMP

O Jump é responsável por duas funções primordiais do jogo: O Salto do Mário e a Gravidade que puxa o personagem para baixo. Estas duas funcionalidades serão melhor explicadas mais a frente.

3.4 TEC

A função Tec é responsável por verificar o teclado do jogador. Caso haja uma tecla pressionada, ele irá verificar se existe alguma ação a ser tomada. Por exemplo, caso o jogador aperte 'd', o teclado irá chamar a função responsável por mover o personagem para a direita. Mas caso a tecla pressionada seja 'i', nada irá ocorrer, já que não há ação para esta tecla.

Listing 2: *Função que realiza a leitura do teclado*

```
.text
Tec:

li t1,0xFF200000      # carrega o endereco de
    controle do KDMMIO
lw t0,0(t1)           # Le bit de Controle
    Teclado
andi t0,t0,0x0001     # mascara o bit menos
    significativo
beq t0,zero,FIM           # Se nao ha tecla
    pressionada entao vai para FIM
lw t2,4(t1)           # le o valor da tecla tecla

li t0,'d'
li a0,8           # add em x para ir pra
    direita
li a1,0           # add em y pra ir pra direita
li a2,1           # nova direcao: Direita
beq t2,t0, MoveSomething  # 'd', move para
    direita

li t0,'a'
li a0,-8          # add em x para ir pra esq

li a1,0           # add em y para ir pra esq
li a2,0           # nova direcao: Esquerda
beq t2,t0, MoveSomething  # se tecla
    pressionada for 'a', move pra esquerda

li t0, 'w'            # inicia o processo do pulo
beq t2, t0, StartJump

li t0, 'm'            # ativa/desativa o audio do
     jogo
beq t2, t0, InvMuteMidi

li t0, 'k'            # printa certos
    registradores na tela (DEBUG)
beq t2, t0, Utils

FIM:  ret             # retorna
```

3.5 CHECKDAMAGE

Essa função é responsável por verificar se o jogador está tocando em algo que causa dano ao personagem. Nesse caso ele irá perder uma vida e voltar ao início do jogo

3.6 CHECKWIN

Por fim, essa função verifica se o personagem ganhou o jogo e chama a função de vitória.

4 DIFICULDADES DO JOGO E SOLUÇÕES

A realização do projeto contava com algumas implementações que não eram triviais e que necessitariam de ideias para solucioná-las.

4.1 MOVIMENTAÇÃO DO BACKGROUND

Para melhor jogabilidade, se foi definido a seguinte regra: O background só se move quando o jogador está no meio da tela, aperta para ir para a direita, e ainda há background para ser renderizado. Em qualquer outro caso quem se move é o personagem.

Dado essas condições, quando o background iria se mover, o que acontece é que o personagem fica parado e o background move, dando a sensação ao jogador de que o personagem que se moveu.

Para fazer isso, foi notado a diferença entre o background antes e depois de um movimento.

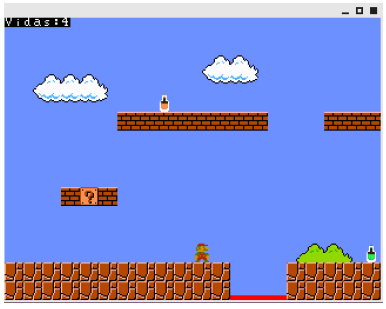

Figura 10: *Background antes do movimento*

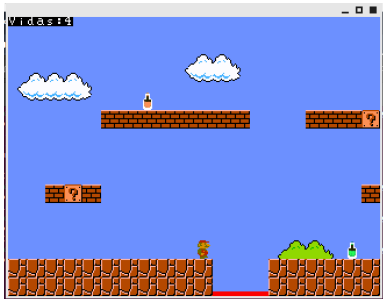

Figura 11: *Background depois do movimento*

Note que a unica coluna diferente no background 'depois' do movimento é a ultima coluna. Todas as outras já estavam renderizadas na tela antes!

Com isso, a ideia de ter o background movel foi basicamente fazer com que cada pixel do BitMap display recebesse o valor do pixel que está na mesma linha, mas 4 colunas a frente. Após isso, todo o background iria se mover 4 pixels para a esquerda, mas a ultima coluna não mudaria. Para isso, há um for que itera APENAS pela ultima coluna e a atualiza de acordo com o mapa dado.

Porém, ainda há um problema não resolvido. No bitmap é facil saber qual o endereço da ultima coluna, porém como saber qual coluna do mapa renderizar, dado que o mapa se move e é bem maior que o bitmap?

Para resolver esse problema, foi criado uma variavel na memória chamado de 'camera'. Sempre que o personagem se move, a coordenada no eixo x dele é modificada. Porém, sempre que o background se move, a camera que é modificada. Sempre que o background vai para a direita, a camera é incrementada com o valor que seria para a coordenada x do personagem.

Com isso, para saber qual a coordenada do mapa está na ultima coluna, basta ver os a coordenada x do personagem somada a 'camera' que representaria um 'offset' de quantas vezes o background se moveu para a direita.

4.2 Salto e Gravidade

A Gravidade foi implementada tentando simular como acontece no mundo real, sendo uma força que puxa o jogador para baixo. Nesse caso, a cada loop do jogo a gravidade faz com que o jogador tenha sua altura diminuída em 1 quando o personagem está no ar se ele não estiver pulando. Se há um bloco logo abaixo dos pés do jogador, então essa força não age.

Listing 3: *Pseudocódigo da gravidade implementada*

```
tile[personagem.x][personagem.y + 1] not in
    blocosSolidos[] && !personagem.jumping:
personagem.y += 1
```

Sendo 'blocosSolidos' um array pré definido contendo todos os blocos que o personagem não pode atravessar enquanto se move. Além disso, como o grid 'cresce para baixo', a coordenada y do jogador deve ser incrementada para que ele no bitmap.

Quanto ao salto, a implementação é quase o contrário da gravidade. Ao iniciar o salto, o personagem começa a subir pixels até uma determinada altura e daí o salto se encerra. Quando o salto se encerra, a gravidade age para que o personagem desça novamente. Além disso, foram implmenetadas técnicas para que o player não possa pular enquanto o personagem estiver pulando e também não possa pular enquanto o personagem não estiver acima de um bloco sólido.

Além do citado quanto ao salto, também há algumas temporizações e valores pré definidos. Como o salto ocorre dentro do loop principal do jogo (que roda inumeras vezes por minuto) foi necessário adicionar uma temporização para as ações do salto. Ou seja, o personagem só aumenta/diminue a sua altura a cada um certo período definido no código. Por fim, também salvamos o quanto o personagem já subiu e a altura máxima do salto atual.

```
24  # Jumping section
25
26  Jumping:                .word 0          # Booleano pra afirmar se o jogador está
27                                           # pulando ou não
28
29  JumpSleepTime:          .word 5          # Tempo entre as chamadas do jump
30
31  LastJumpUpdateTime:     .word 0          # Ultima vez que o pulo foi atualizado
32
33  JumpHeight:             .word 75         # altura máxima do pulo
34  JumpCurrentHeight:      .word 0          # altura que o pulo atual já subiu
35  JumpHeightPerClock:     .word 1          # quantos pixels o pulo aumenta ou a gravidade diminue
36                                           # no y do personagem a cada 'clock' do publo
```

Figura 12: *Variaveis necessárias para o salto*

4.3 Colisão

Como o mapa é imenso com diversas tiles diferentes, implementar qualquer tipo de algoritmo que cheque a cor da tile na posição de destino do personagem não seria fácil. Existem diversos blocos sólidos no mapa, fazendo com que o código ficasse imenso.

Para evitar isso e implementar da melhor forma possível, pensamos no conceito de 'hitbox' dos jogos. São basicamente retângulos em volta de formas mais complicadas que sinalizariam quando a colisão aconteceu ou não.

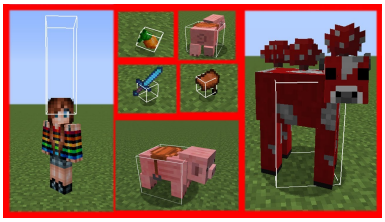

Figura 13: *Exemplo de hitbox no Minecraft*

Tendo essa ideia, ainda havia um problema: Mesmo Super Mario sendo um jogo tão antigo e tão famoso, não havia um 'hitbox' do mapa já pronto na internet. Ao menos não da forma que se precisava. Para contornar esse problema, pensamos em criar o nosso próprio, porém o mapa do mário tem mais de 3300 colunas, cada uma com 240 linhas de pixels, deixando inviavel fazer na mão.

Portanto, a saída que restou foi criar um algoritmo que criasse isso para nós. A ferramenta criada em python se chama 'CreateMapHitbox.py'. A ideia é que ela lê o mapa do jogo como um array de duas dimensões e também lê todos os tiles que representam blocos rígidos do jogo em outro array 'walls'. Assim, itera por todas as tiles do mapa e verifica se é uma 'wall' ou não. Caso seja, modifica essa tile do mapa para a cor preta.

Além de verificar os blocos rígidos no mapa gerado, também foi implementado a mesma ideia para verificar os blocos que causam dano ao jogador. Nesse caso, a implementação trocava este tile por um tile todo em vermelho.

Listing 4: *Parte principal do script de geração da HitBox*

```
for i in range(0, len(map), tamTile):
    for j in range(0, len(map[0]), tamTile):

        cur_sprite = []

        for k in range(tamTile):
            cur_sprite.append(map[i + k][j : j
                + tamTile])

        isWall = False

        for w in walls:
            if w == cur_sprite:
                isWall = True

        isDamage = False

        for w in damage:
            if w == cur_sprite:
                isDamage = True

        if isWall:

            # if is a wall change the
                cur_sprite on map to 0

            for k in range(tamTile):
                map[i + k][j : j + tamTile] =
                    [0] * tamTile

        elif isDamage:

            # this sprite should damage char
            # paint it as a red

            for k in range(tamTile):
                map[i + k][j : j + tamTile] =
                    [7] * tamTile
```

```
        else:

            # if is not a wall, change the
                cur_sprite to blue color

            for k in range(tamTile):
                map[i + k][j : j + tamTile] =
                    [227] * tamTile
```

Tendo isso preparado, o código gera um segundo mapa, chamado de mapHitBox na implementação, que pode ser visualizado da seguinte forma.

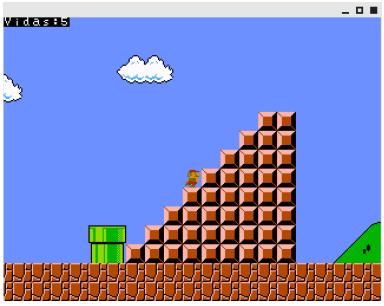

Figura 14: *Frame do mapa original*

Figura 15: *Frame do mapa gerado*

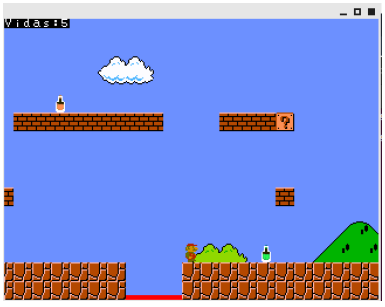

Figura 16: *Frame do mapa original*

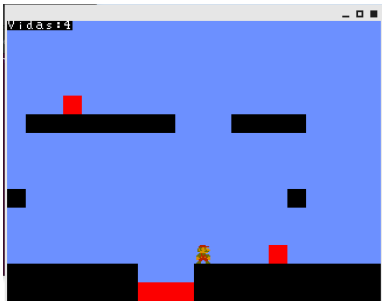

Figura 17: *Frame do mapa gerado*

Com este hitbox fica fácil checar a colisão: Basta ver naquela posição a cor dada é preta. Além disso, para checar se o personagem deve tocou em algum ponto que não devia é só checar se ele está em um ponto de cor vermelha da hitbox.

4.4 Animação de movimento do personagem

Para que o personagem seja estático e simplesmente teleporte para a posição de movimento, foram usadas quatro sprites de movimento. Sempre que o personagem está em uma e se move no eixo x, a próxima é utilizada. Dessa forma, quando o player mantém a tecla 'd' pressionada, o personagem se move enquanto alterna entre os sprites, criando a animação de que está correndo. Essas sprites existem tanto para a direção esquerda quanto para a direita.

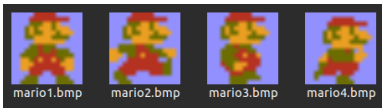

Figura 18: *Animações de movimento*

5 Conclusão

Trabalhar neste projeto trouxe inumeros aprendizados. Programar não é trivial em qualquer linguagem que seja - muito menos em tão baixo nível. É um desafio criar um jogo tão complexo apenas com as ferramentas disponíveis. Porém, é neste momento que ideias brilhantes surgem para resolver problemas complexas. O problema resolvido é grande, mas as ideias são simples. Assim, este projeto contribuiu para que todo o grupo desenvolvesse as técnicas de programção, de pensamento em baixo e também técnicas de desenvolvimento de jogos como colisão, movimentação de background, hitbox. Após inumeros desafios e horas debugando endereços de memória nosso personagem principal Mario finalmente conseguirá entregar seu trabalho final de OAC – E sempre tomando cuidado com o suco do RU.

Referências

[1] Conversor de imagens bmp para .data. Disponível em https://github.com/gss214/Gerenciador-de-Conversao/

[2] Hooktheory: Site com cifra de músicas Disponível em https://www.hooktheory.com

[3] RARS: Risc-v assembler and runtime simulator. Disponível em https://github.com/TheThirdOne/rars

[4] FPGRARS: Fast Pretty Good RISC-V Assembly Rendering System Disponível em https://github.com/LeoRiether/FPGRARS