Sussar

Maxwell Oliveira Dos Reis Wilson Oliveira Guimarães Neto

University of Brasília, Dept. of Computer Science, Brazil

Abstract

Esse trabalho tem como objetivo a criação baseado no Pulsar (1981 - Sega) utilizando o simulador RARs em conjunto com a ISA do RISC-V. Os conceitos utilizados para a criação desse jogo foram adquiridos no decorrer da disciplina de Introdução a sistemas computacionais e servem, também, para se ter um noção de como é escrever um código complexo em baixo nível.

1 Introdução

Pulsar é um jogo desenvolvido pela empresa SEGA. Foi publicado no ano de 1981 e possui uma mecânica simples, onde o jogador controla um tanque e precisa coletar chaves para avançar de fase. Porém, o tanque gasta combustível ao se locomover e precisa passar por diversos tipos de inimigos durante sua trajetória.

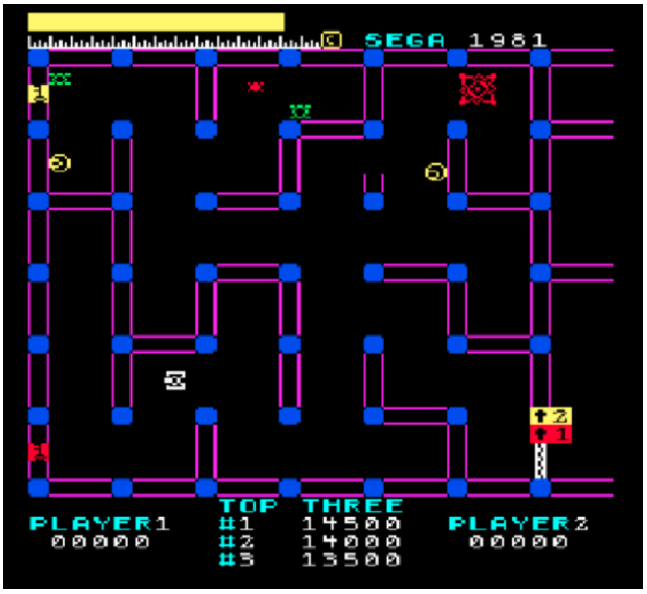


Figure 1: Gameplay do jogo

A Figura 8 mostra a visão do jogador em uma das fases.

Na imagem é possivel notar a quantidade variada de inimigos e de chaves. Além disso, há uma barra na parte superior da tela que representa o combustivel e um menu com informações na parte de baixo. O jogo também conta com algumas paredes que podem sumir ou aparecer, modificando, assim, o layout da fase temporariamente.

Na seção 2 será apresentada a metodologia utilizada. A seção 3 apresenta os resultados obtidos. A seção 4 conclui este trabalho.

2 Metodologia

Varias ferramentas foram usadas para o desenvolvimento deste projeto, dentre elas temos o python para a geração de alguns dos sprites utilizados, o Rars para executar o código criado, os tutoriais e exemplos disponibilizados pela disciplina e etc.

No primeiro momento deste projeto, criamos um programa que gerava um personagem e andava pelo mapa. Com ele, começamos a adicionar as lógicas de colisão. Porém, pela falta de planejamento do código, decidimos abandonar este arquivo e começar outro do zero.

Após este evento, criamos outro projeto e escrevemos um código que fazia a mesma coisa, porém era bem mais otimizado, de manutenção mais fácil e capaz de receber os módulos que seriam adicionados no futuro. Uma das ideias utilizadas para este novo código foi a de criar várias funções que se comunicariam prioritariamente por *flags* e estariam em arquivos próprios. Por exemplo, se a *flag winFlag* valer 1, significa que a função que carrega a tela de vitória do jogo deve ser acionada.

Com isso feito, passamos apenas a criar mais funções em arquivos separados e apenas importá-las no arquivo principal do jogo. Deixando, assim, o código principal mais organizado, facil de entender e limpo.

3 Resultados Obtidos

3.1 Resultado Final

O jogo foi desenvolvido e obtivemos um resultado satisfatório, tendo conseguido implementar a lógica da colisão em obstáculos, movimentação e ataque dos inimigos, o sistema de combustivel, mais de uma fase e etc.

Figure 2: Capa do jogo

Além de implementar toda esta lógica, de nada agradaria um jogador se a parte visual não ficasse no mesmo nível. Pensando assim, decidimos implementar um visual do jogo baseado nos personagens de outro jogo chamado *Among Us*, fazendo o personagem principal ser um *crewmate* e os inimigos serem *impostors*.

Figure 3: Gameplay do jogo

A imagem acima mostra o jogo sendo executado. Aqui é possivel ver que temos dois tipos de inimigos, um deles tem a capacidade de atirar e o outro de roubar combustivel. Caso o jogador seja atingido por um tiro ou ficar sem combustivel, ele perderá uma vida. Caso fique sem vidas, o jogo é encerrado por gameover.

Figure 4: Tela de derrota

Por outro lado, se o jogador for capaz de passar pelas duas fases, então o player conseguiu finalizar o jogo!

Figure 5: Tela de vitoria

Da mesma forma, decidimos investir tempo na parte sonora do jogo, adicionado sons na tela de iniciar, ao atirar e na tela de vitória.

Com isso, acreditamos que o resultado obtido é capaz de entregar uma boa experiência para o jogador, mesmo com as limitações do Rars e a da ISA do Risc-V.

3.2 Alguns dos Problemas enfrentados

3.2.1 Colisao

O primeiro problema enfrentado foi o de como poderíamos adicionar a colisao no jogo. Para resolver, definimos a cor do fundo do jogo como preto. Assim, sempre que algo (personagem, inimigo, tiro) vai se movimentar, este verifica se a cor do bloco de destino é preto. Se for, entao ele pode se mover, pois temos a certeza de ser ”Chao”. Para implementar isso, sempre verificamos a cor do primeiro bit de cada bloco.

Utilizando essa lógica, o personagem principal já conseguia passar por cima da chave e esta sumir do mapa. Entretanto, neste momento as chaves são apenas sobrepostas pela cor do fundo, não contando para abrir o portão.

Na imagem 6 temos como a chave está sendo representada para o jogo. Ela possui o tamanho de 16x16 pixels e cada numero da

```
key00: .word 16, 16
.byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,83,83,83,0,0,0,0,
0,0,0,0,0,0,0,0,74,174,174,174,74,0,0,0,
0,0,0,0,0,0,0,74,174,1,1,83,165,74,0,0,
0,0,0,0,0,0,0,1,174,1,0,1,165,1,0,0,
0,0,0,0,0,0,0,1,165,83,1,1,165,1,0,0,
0,0,0,0,0,0,0,1,255,255,174,165,1,0,0,0,
0,0,0,0,0,0,74,255,91,1,1,1,0,0,0,0,
0,0,0,0,0,83,174,91,1,0,0,0,0,0,0,0,
0,0,0,0,83,174,91,1,0,0,0,0,0,0,0,0,
0,0,0,74,165,83,1,0,0,0,0,0,0,0,0,0,
0,0,1,165,83,92,165,74,0,0,0,0,0,0,0,0,
0,0,1,91,83,165,255,255,74,0,0,0,0,0,0,0,
0,0,0,1,1,83,174,74,0,0,0,0,0,0,0,0,
0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
```

Figure 6: Sprite da chave neste momento

imagem representa uma cor para aquele pixel específico, sendo 0 a cor preta.

3.2.2 Adicionando a pontuação nas chaves

Neste ponto, temos que a chave apenas é sobreposta. A maneira mais facil que encontramos de implementar um ”valor” para ela foi utilizando o segundo bit dessa imagem para representar o ”valor da chave”. Assim, o primeiro bit é usado para checar se podemos passar por cima e o segundo é um valor que iremos retirar da ”vida do portão”. Se a ”vida do portão” for zero, entao ele se abre e permite sua passagem.

```
key00: .word 16, 16
.byte 0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,83,83,83,0,0,0,0,
0,0,0,0,0,0,0,0,74,174,174,174,74,0,0,0,
0,0,0,0,0,0,0,74,174,1,1,83,165,74,0,0,
0,0,0,0,0,0,0,1,174,1,0,1,165,1,0,0,
0,0,0,0,0,0,0,1,165,83,1,1,165,1,0,0,
0,0,0,0,0,0,0,1,255,255,174,165,1,0,0,0,
0,0,0,0,0,0,74,255,91,1,1,1,0,0,0,0,
0,0,0,0,0,83,174,91,1,0,0,0,0,0,0,0,
0,0,0,0,83,174,91,1,0,0,0,0,0,0,0,0,
0,0,0,74,165,83,1,0,0,0,0,0,0,0,0,0,
0,0,1,165,83,92,165,74,0,0,0,0,0,0,0,0,
0,0,1,91,83,165,255,255,74,0,0,0,0,0,0,0,
0,0,0,1,1,83,174,74,0,0,0,0,0,0,0,0,
0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
```

Figure 7: Sprite final da chave

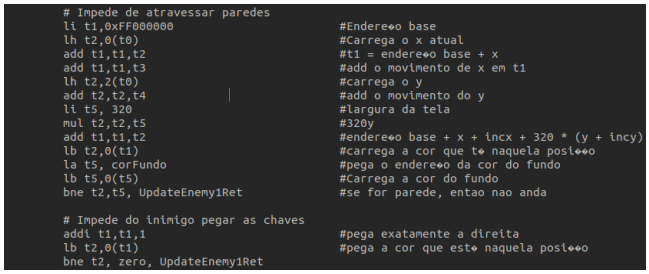
Na imagem acima adicionamos o valor 1 para esta chave. Assim, quando o jogador passar por cima dela, este valor é computado para que o portão se abra.

Neste ponto a chave já está funcionando para a interação com o jogador. Porém temos que lembrar que existem os inimigos andando pelo mapa! Como o primeiro bit é zero e representa a cor do chão, existe a possibilidade do inimigo roubar a chave.

3.2.3 Impedindo os inimigos de roubarem as chaves

Para impedir que os inimigos sejam capazes de roubar as chaves de forma simples no código, nós modificamos um pouco a condição para que o inimigo veja uma posição como ”disponível”. Antes, bastava o primeiro bit da cor ser 0, agora nós verificamos também se o segundo bit (representado na Figura do item anterior) também

vale 0. Ou seja, caso os dois primeiros bits sejam iguais a 0, o inimigo é entende como bloco disponível para se mover. Caso contrário, ele entende que há algo ocupando aquele espaço.



```
# Impede de atravessar paredes
li t1,0xFF000000                    #Endere◆o base
lh t2,0(t0)                         #Carrega o x atual
add t1,t1,t2                        #t1 = endere◆o base + x
add t1,t1,t3                        #add o movimento de x em t1
lh t2,2(t0)                         #carrega o y
add t2,t2,t4                        #add o movimento do y
li t5, 320                          #largura da tela
mul t2,t2,t5                        #320y
add t1,t1,t2                        #endere◆o base + x + incx + 320 * (y + incy)
lb t2,0(t1)                         #carrega a cor que t◆ naquela posi◆◆o
la t5, corFundo                     #pega o endere◆o da cor do fundo
lb t5,0(t5)                         #Carrega a cor do fundo
bne t2,t5, UpdateEnemy1Ret          #se for parede, entao nao anda

# Impede do inimigo pegar as chaves
addi t1,t1,1                        #pega exatamente a direita
lb t2,0(t1)                         #pega a cor que est◆ naquela posi◆◆o
bne t2, zero, UpdateEnemy1Ret
```

Figure 8: Codigo final para os inimigos verificarem se um bloco está ocupado ou nao

3.3 Movimentação dos inimigos

3.3.1 Como gerar o movimento

Outro problema a se pensar foi em como fazer os inimigos se movimentarem pelo mapa. Tínhamos algumas opções, a primeira delas foi dar uma direção inicial para o inimigo e fazer ele seguir nesta direção enquanto puder. Assim que batesse em alguma barreira, ele mudaria sua direção em sentido horário e continua seguindo. O problema dessa técnica é que o padrão da movimentação de cada inimigo será sempre o mesmo! Este fato foi crucial para decidirmos não utilizar esta abordagem.

Assim, a tecnica utilizada para fazer a movimentação dos personagens foi na gerando numeros aleatórios. A ideia era gerar um numero aleatório em um intervalo, e pegar este numero modulo 4, então iriamos olhar a direção que ele representava e fazer o personagem mudar de direção.

3.3.2 Consequencia dos movimentos

Utilizando a ideia acima, os inimigos já seriam capazes de se mover. Porém, outro problema surge quando chamamos a função dele se mover a cada loop do jogo. Como o jogo roda muito rápido, entao o inimigo com uma velocidade absuda, aumentando muito a dificuldade do jogador acertar ele com um ataque. Para corrigir isso, adicionamos uma variavel que serviria de ”sleep” para os inimigos, deixando eles se moverem apenas 1 vez a cada uma certa quantidade de loops. Isso balanceou o jogo.

4 Conclusão

Ao fim do projeto, só nos resta a experiência que ficará conosco. Criar um jogo em uma linguagem de programação de alto nível como Python, por exemplo, seria uma atividade bem mais fácil. Porém, ao nos arriscarmos diretamente no baixo nível ganhamos a experiência de como era programar jogos nas décadas 80 e 90.

Por fim, conseguir entregar este resultado é muito satisfatório para nós que não tínhamos conhecimento prévio antes do início do semestre e nos arriscamos a aprender.

References

TheThirdOne. Rars: Risc-v assembler and runtime simulator.

Wikipedia. Pulsar (video game)

Internet archive. Pulsar