Adventures of Lolo

César Augusto B. R. Lacerda Felipe Dantas Borges Gustavo Pierre Starling

University of Brasília, Dept. of Computer Science, Brazil

Abstract

Este trabalho tem como objetivo a criação de um jogo baseado em ”Adventures of Lolo” na ISA RISC-V utilizando o simulador RARS, a fim de criar familiaridade com a linguagem Assembly tanto para complementar o conteúdo visto ao longo do semestre quanto como preparação para as disciplinas relacionadas nos semestres seguintes.

1 Introdução

Adventures of Lolo é um jogo de puzzle lançado em 1989 para o console NES. Suas mecânicas são simples e facilmente compreensíveis, o que o torna um bom jogo para se recriar em Assembly, uma linguagem de baixo nível em que conceitos e mecânicas complexas são mais difíceis de se aplicar e requerem maior experiência com a linguagem e o simulador.

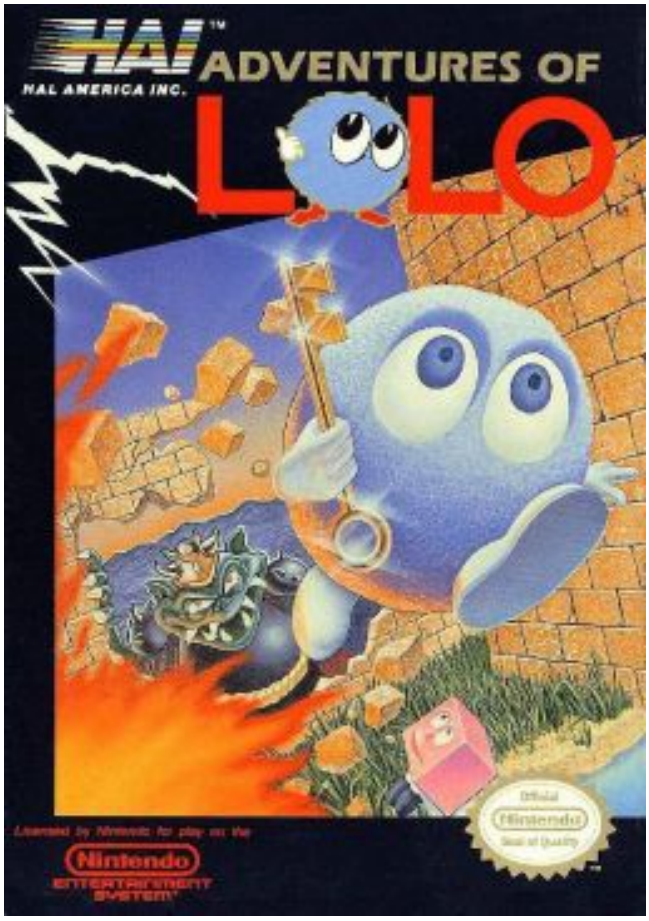

Figure 1: Capa do jogo *Adventures of Lolo*

2 Metodologia

Decidimos logo no começo usar o jogo original apenas como uma base e modificar diversos de seus aspectos para que não se tornasse apenas mais uma cópia. Trocamos o personagem principal, os inimigos e alguns outros elementos ao longo do jogo para que ele fizesse mais sentido no contexto da disciplina que estamos cursando.

2.1 Menu e animação

Optamos por fazer o menu de início, para nos familiarizarmos com o RARS e suas ferramentas. Assim, as primeiras implementações foram o sistema de password e o menu, com uma seta para selecionar uma das duas opções utilizando um registrador para armazenar a opção selecionada.

Com o que entendemos na criação do menu, se tornou muito mais simples imprimir o personagem e fazê-lo andar na fase sem que ele deixasse um ”rastro”.

Armazenamos sua posição atual e sempre que as teclas de andar são pressionadas, essa posição é carregada e utilizada para preencher o bloco 16x16 em que o personagem se encontra com a sprite do chão no frame 1, em seguida soma-se 8 a essa posição e imprime o personagem lá.

Ocorre então uma troca de frames, repetindo esse processo para cada tecla pressionada e alternando os frames com sprites diferentes do personagem para criar uma animação.

2.2 Colisão

Com o mapa impresso e o personagem andando, o problema seguinte era a colisão. Inicialmente, pensamos em verificar os limites do mapa e blocos em que deveria haver colisão utilizando matrizes correspondentes ao mapa, mas logo pensamos em uma solução que seria mais abrangente e utilizada em mais casos: Usar a posição do personagem que é salva para sua movimentação e com base nela, de acordo com a direção em que ele anda, verificar a sprite que está imediatamente em sua frente.

Essa verificação consiste em analisar a cor de dois pixels específicos na sprite que variam de acordo com a direção, um pixel na primeira metade da sprite e um pixel na segunda pois o personagem anda meio sprite por vez. Essa cor é comparada com valores de cor armazenados que correspondem a sprites específicas que ativam certos algoritmos. Em sua forma mais básica, esse sistema realiza a verificação e se os pixels tiverem uma cor diferente da cor estipulada para a sprite do chão, o personagem não anda.

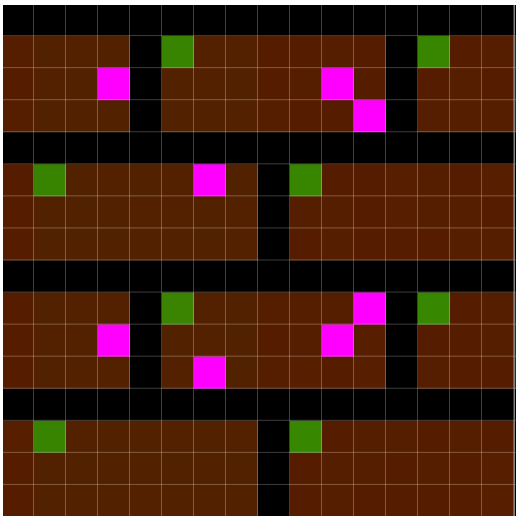

Figure 2: As posições dos pixels analisados estão em rosa

Esse algoritmo acabou por ser uma parte crucial do projeto pois sua versatilidade permitiu que ele fosse usado em todas as interações do jogo, bastando apenas especificar um procedimento a ser executado caso os pixels analisados fossem de uma cor específica. Isso foi aplicado para empurrar caixas, coletar corações, entrar em portas e também em um dos inimigos, que será comentado em detalhes mais adiante.

Para as caixas, por exemplo, o algoritmo de colisão verifica a colisão do personagem com a caixa. Se isso ocorrer, a caixa atual é

apagada, o personagem anda e a caixa é impressa novamente em sua nova posição, se mantendo imediatamente na frente do personagem, o que faz com que ele efetivamente ”empurre” a caixa. Além disso, a colisão para a caixa também verifica a posição futura para a qual a caixa será empurrada, para que ela não ultrapasse obstáculos. Isso faz com que, caso a caixa esteja encostada em uma parede, ela não possa mais ser empurrada naquela direção.

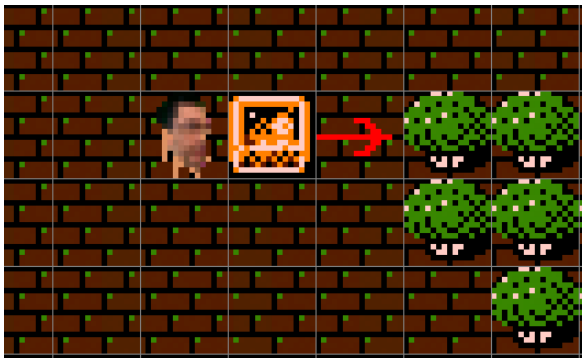

Figure 3: Demonstração do personagem empurrando a caixa

Figure 4: A caixa colide com objetos do mapa

2.3 Interface MIDI e música

Para integrar músicas e efeitos sonoros no projeto, foi utilizada a interface MIDI do RARS, que, apesar de suas limitações, pode ser aplicada de formas interessantes e com resultados satisfatórios se utilizada com um algoritmo inteligente.

O algoritmo que fizemos para usar a interface MIDI possui duas macros principais: Uma utilizada para tocar músicas e outra utilizada para sons individuais, como efeitos sonoros. Ao entrar no menu ou iniciar uma fase, é utilizada a macro para iniciar a música, que carrega a sequência de notas da música armazenada no .data guardando a posição no tempo em que a música está a cada nota tocada.

Esse loop sempre retorna para a função que recebe input pela interface KDMMIO. Caso nenhuma tecla seja pressionada, ele prossegue para a próxima nota da música. Caso uma tecla seja pressionada, ele executa todas as ações necessárias e ao voltar, carrega a posição em que estava na música para continuar de onde parou.

A segunda macro toca sons individuais, recebendo com argumentos valores como o volume, a nota e o instrumento que será utilizado. Após ser executada, ela retorna para a função de receber input do teclado para continuar o loop principal do jogo.

Há também uma macro auxiliar utilizada para limpar todos os valores relacionados ao algoritmo de sons quando se passa de um menu para uma fase ou de uma fase para outra, para que a música comece do início sem causar problemas.

2.4 Inimigos e ataques

Implementamos dois tipos de inimigo que foram distribuídos ao longo das cinco fases.

Um deles funciona como um obstáculo, sendo necessário atacá-lo para liberar o caminho.

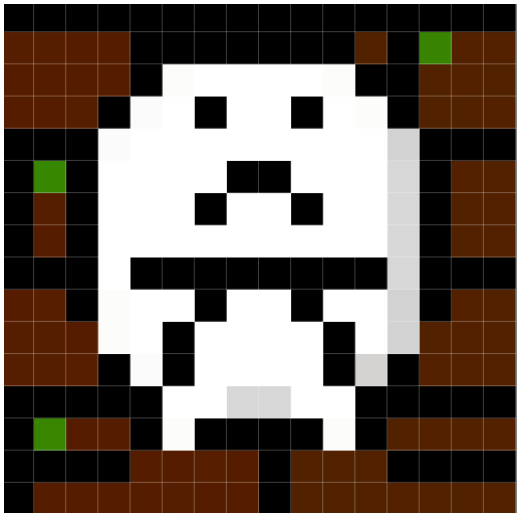

Figure 5: Transistor em depleção, inimigo obstáculo

O outro é um inimigo estático com um alcance de 2 blocos em todas as direções. Ele detecta quando o personagem entra em seu alcance e o mata quando isso acontece. Ao colocar uma caixa em uma das direções do inimigo, ele não ataca mais naquela direção específica, abrindo várias possibilidades de mapas e puzzles interessantes que usam essa mecânica ao máximo.

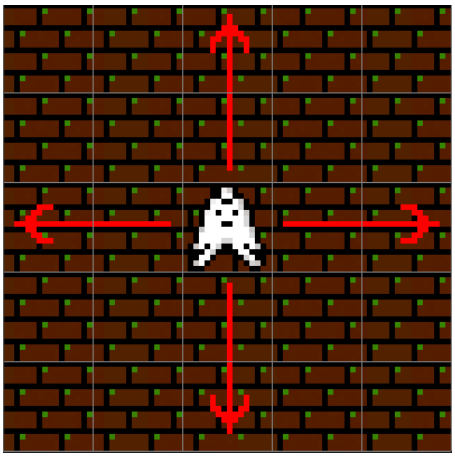

Figure 6: Inimigo ”Gate OR” com setas indicando seu alcance.

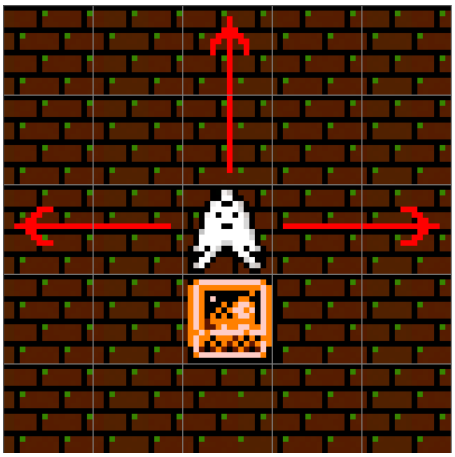

Figure 7: Inimigo ”Gate OR” com um obstáculo impedindo seu ataque.

Discutimos várias formas de aplicar essa mecânica de verificações.

Uma delas consistia em verificar pixel por pixel em todos os blocos em que o inimigo alcançava, mas devido as limitações do RARS isso é extremamente ineficiente e teria um impacto muito grande na performance do jogo. Recorremos então à colisão novamente: Os sprites de chão que estão no alcance do inimigo tem os pixels de verificação da colisão pintados com uma cor discreta, semelhante à do chão. Ao colidir com esse sprite, ele verifica a sprite vizinha que também está no alcance do inimigo. Se houver uma caixa nele, o personagem não morre.

Durante o jogo, quando a caixa é posicionada em frente ao inimigo para tampar seu ataque ela fica eletrizada, não podendo mais ser movida. Essa mecânica traz um nível maior de profundidade, fazendo o jogador pensar sobre seus movimentos para evitar precisar reiniciar a fase desnecessariamente e perder uma vida.

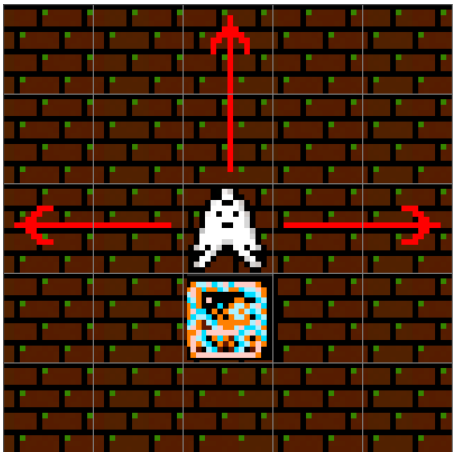

Figure 8: Caixa eletrizada ao entrar em contato com o inimigo

3 Problemas enfrentados

Grande parte dos problemas que encontramos ao longo do projeto foram simples e fáceis de resolver, na maior parte das vezes sendo causados por não conhecermos a fundo como o RARS e a linguagem Assembly funcionam. Ao longo do tempo, porém, começou a surgir o erro que mais nos atrapalhou durante os estágios finais do trabalho: *Branch target word address beyond range*.

Com o passar do tempo o código do jogo ficou cada vez maior e as limitações de um simulador de um processador RISC-V de 32 bits se tornaram mais aparentes. Quando se lida com um projeto deste tamanho, a quantidade de bits utilizada para armazenar endereços nas instruções de jump e branch rapidamente se mostraram insuficientes para cobrir toda a extensão do código, que ao final do trabalho já somava em torno de três mil linhas, considerando os arquivos auxiliares de macros. A cada nova função, fase ou caso de colisão que adicionávamos, esse erro surgia novamente.

Sempre que esse erro surgiu, ele nos tomou um tempo considerável para buscar o que estava causando ele, e o que poderia resolver. A solução mais utilizada foram saltos intermediários dentro do código, levando uma instrução para mais perto do "*.include*" com o arquivo onde a macro que foi chamada estava. Essa solução foi apelidada de "trampolim", e em alguns casos precisou ser usada até três vezes para que as instruções alcancassem suas macros.

O RARS não indica exatamente qual instrução causou o erro de branch, apenas indica uma linha, mas não fica claro o que essa linha significa. Isso dificultou muito o processo de depuração, e nos obrigou a repensar a estrutura do código e a aplicação dos algoritmos diversas vezes.

4 Resultados

No fim, apesar das dificuldades de estarmos utilizando uma linguagem nova um pouco mais rudimentar e das limitações do RARS, o projeto deu um resultado satisfatório, com todas as partes funcionais e um gameplay loop relativamente agradável e próximo do original. As mudanças feitas também se encaixaram bem no estilo do jogo e somam à experiência. Durante todo o projeto, o maior inimigo foi o tempo. Se houvesse mais tempo, teríamos adicionado mais funcionalidades e revisado as já existentes para garantir que o jogo funcionasse da melhor forma possível, além da possibilidade de mais fases e um maior grau de complexidade nelas.

5 Conclusões

Ao fim do projeto, o que ficou mais claro para o grupo foi a diferença que uma engine faz no processo de criar um jogo. Ela é um facilitador em todos os aspectos e salva muito tempo por tomar conta das partes mais rudimentares e da fundação do jogo. Fazer a base do jogo e as mecânicas mais simples, como movimentação e colisão foram as que tomaram mais tempo, desconsiderando os erros. Por outro lado, percebemos o quanto é possível fazer com ferramentas que aparentam ser tão simples, tudo depende da capacidade de solução de problemas. Aprendemos bastante com o processo, e saímos com muito mais conhecimento acerca do funcionamento da linguagem, do RARS e da própria ISA RISC-V. Esse conhecimento é útil mesmo quando lidando com linguagens de alto nível, uma vez que saber como as coisas funcionam "por baixo dos panos" é crucial para otimizar e pensar em soluções mais limpas para uma grande variedade de problemas em qualquer situação.

Figure 9: O trabalho foi um sucesso!

References

Wikipedia, Adventures of Lolo

HookTheory, TheoryTab DB

Retrogames.cz