Projeto Aplicativo de Organização e Arquitetura de Computadores

Arthur Menezes Botelho [*] Hiago Sousa Rocha [†] Iasmin de Queiroz Freitas [‡] Luiz Felipe Ducat [§]

Universidade de Brasília, 19 de Janeiro de 2025

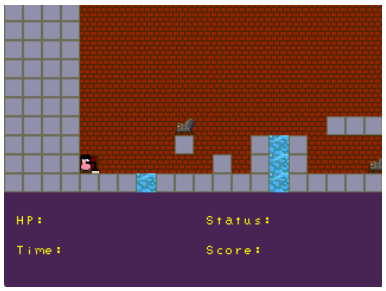

Figura 1: *Imagem de teaser (opcional)*

RESUMO

Kirby's Adventures é um jogo publicado em 1993 para a plataforma *Nintendo Entertainment System* (NES). Este relatório descreve a implementação de um jogo escrito em Assembly RISC-V e inspirado em *Kirby's Adventures* — tal como especificado em [1] — para a disciplina de Organização e Arquitetura de Computadores. Para tanto, foram utilizados os simuladores de sistemas RISC-V RARS e FPGRARS, bem como um processador RISC-V implementado numa placa FPGA.

Palavras-chave: OAC · Assembly IRSC-V · Template

1 INTRODUÇÃO

Em *Kirby's Adventure*, o jogador controla o personagem Kirby, que deve atravessar *Dream Land* para reparar a *Star Rod*, destruída pelo King Dedede. O jogo utiliza um estilo de plataforma 2D e foi o primeiro da franquia Kirby a utilizar a habilidade de cópia, que permite ao personagem jogável copiar as habilidades de certos inimigos, numa mecânica similar a de outros jogos em que o jogador pode utilizar *power ups* para modificar suas habilidades.

2 FUNDAMENTAÇÃO TEÓRICA E TRABALHOS RELACIONADOS

Esta seção descreve a base de algumas ideias e tecnologias utilizadas na implementação do jogo.

2.1 A ARQUITETURA RISC-V

A arquitetura RISC-V é uma arquitetura de conjunto de instruções (ISA, do inglês *Instruction Set Architecture*) de código aberto e baseada no conceito de Computação com Conjunto de Instruções Reduzido (RISC, do inglês Reduced Instruction Set Computing). Ela foi desenvolvida na Universidade da Califórnia, Berkeley, a partir de 2010, com o objetivo de ser uma arquitetura simples, modular, eficiente e livre de royalties, permitindo sua adoção em uma ampla gama de aplicações, desde dispositivos embarcados até supercomputadores. RISC-V é a ISA utilizada na disciplina de Organização e Arquitetura de Computadores e o Assembly requerido por [**?**] para implementação do jogo.

2.2 PLACAS FPGA

As placas FPGA (*Field-Programmable Gate Array*, ou Matriz de Portas Programáveis em Campo) são dispositivos de hardware reconfiguráveis que permitem a implementação de circuitos digitais personalizados. Diferente dos processadores tradicionais, que executam instruções de software de forma sequencial, as FPGAs são projetadas para implementar circuitos lógicos diretamente em hardware, o que as torna adequadas para projetos de prototipação e pesquisa e desenvolvimento, em que a fabricação de circuitos customizados não é viável. A disciplina de Organização e Arquitetura de Computadores faz uso extensivo de placas FPGA para implementação de diversas organizações (organização tal como compreendida em [2]) da ISA RISC-V, e a execução do jogo em um processador implementado numa placa FPGA também é um requerimento de [1].

2.3 ITERAÇÕES PASSADAS DA DISCIPLINA

Tendo sido ofertada diversas vezes no passado, a produção de alunos da disciplina é extensa e variada e foi importante para a construção do trabalho atual. Em particular, o simulador de RISC-V [3], o repositório git [4] e as diversas contribuições de jogos listadas nele — em particular [5].

3 METODOLOGIA

Para a implementação do jogo, foi decidido atender aos requisitos especificados em [1], mas divergindo da temática original de *Kirby's Adventure* em favor de uma temática própria introduzida pelo grupo. As subseções a seguir exploram as decisões feitas com respeito à implementação do jogo.

3.1 RENDERIZAÇÃO

A renderização consiste num mecanismo em que os mapas, ao invés de serem carregados inteiros na memória, são desenhados tile a tile utilizando sprites que juntos compõem a totalidade do mapa. Durante toda a duração do jogo, a posição do jogador é mantida na memória. Essa é a posição em relação ao início do mapa. Essa

[*] 231003362@aluno.unb.br
[†] 221002049@aluno.unb.br
[‡] 190108665@aluno.unb.br
[§] 231035400 @aluno.unb.br

posição é utilizada para calcular um offset em relação ao personagem para decidir quais partes do mapa renderizar. Isso é necessário pois a tela mostrada é menor do que a totalidade do mapa. Então a decisão do que efetivamente renderizar é feita utilizando a posição do personagem subtraída por um offset, para renderizar o mapa no e ao redor do personagem.

3.2 Músicas e sons

Os sons e as músicas para o jogo utilizam uma biblioteca de sons baseada em [5]. Em particular, as músicas consistem numa estrutura especificando parâmetros como instrumento e quantidade notas, além de uma sequência de pares indicando duração (em milissegundos) e a nota (pitch), que é a a maneira como a chamada de sistema 31 espera receber uma entrada. Para converter arquivos midi para a maneira como a chamada de sistema espera receber a entrada, a ferramenta Midi2RiscV[6] foi largamente utilizada.

3.3 Controles

Os mecanismos de controle utilizados na lógica do jogo são essencialmente baseados em estados. Por exemplo, o arquivo *player_data.s* contém diversas informações a respeito do estado atual do jogador, como a direção para a qual ele aponta, sua posição em relação ao chão (se seu estado está no chão ou no ar), e o estado das suas habilidades.

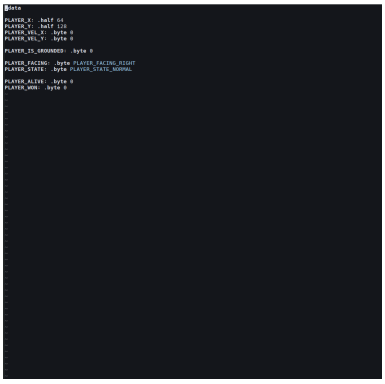

Figura 2: *Arquivo de estados do jogador*

Os estados mencionados são então utilizados para diversas funções; por exemplo, a escolha na renderização de sprites:

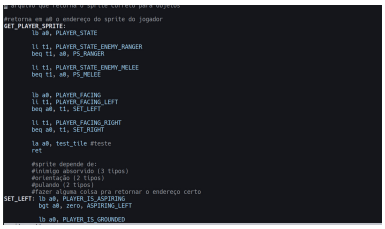

Figura 3

3.4 Leitura do teclado

Para leitura do teclado, o programa utiliza um mecanismo que lê constantemente um buffer que recebe as entradas do teclado. Isso é feito em loop. Uma opção alternativa seria implementar interrupção, o que eliminaria a necessidade checar constantemente o valor recebido no buffer — complicando, no entanto, a implemetação.

3.5 A máquina de estados

O jogo como um todo consiste numa máquina de estados de quatro estados: o estado inicial, configuração, fase e final. O estado inicial consiste na inicialização o endereço dos frames de desenhos na memória, toca a música inicial e chama o estado de configuração. O estado de configuração lê na memória qual é a fase atual e carrega os dados da fase — como mapa, a posição dos inimigos, tipos de inimigos, inicializa a posição do jogador etc — a partir da memória de mapas/etc para a fase/mapa atual. Além disso, ele escolhe a música com base no estado anterior (se você morreu ou se está começando o jogo agora). O estado de fase consiste no jogo em si. Esse estado gerencia a lógica de entrada e saída do jogador, a IA dos inimigos, lógica de colisões e de movimentação. Em qualquer momento dado esse estado pode: vencer a fase, perder a fase ou continuar na fase. Se voc perde, a máquina de estados retorna ao estado de configuração. Se você vence e essa é a última fase, a máquina de estados prossegue para o estado final do jogo. Caso contrário, a máquina de estados retorna ao estado de configuração, mas mudando o índice da fase, de modo a configurar a próxima fase. O estado final toca a música de vitória do jogo e entra em loop no estado atual.

4 Resultados Obtidos

O jogo obtido atende parcialmente aos requisitos descritos em [1]. A lógica de renderização lida corretamente com o deslocamento horizontal e com os limites impostos pela memória, implementa som e música e colisão parcial. A arquitetura baseada na máquina de estados atende corretamente à necessidade de lidar com estados do jogo, como a transição entre múltiplas fases e diferentes músicas. No entanto, a colisão entre inimigos e sólidos no cenário não pôde ser obtida em tempo hábil, bem como outros requisitos, como o mecanismo de habilidades e projéteis.

5 Trabalhos futuros

Um trabalho para o futuro seria atender integralmente aos requisitos do trabalho, bem como melhorar a legibilidade e performance do código, uma vez que a velocidade de execução no momento é relativamente lenta na placa FPGA.

Referências

[1] *Projeto Aplicativo — Kirby's Adventure*. Disponível em: https://tinyurl.com/especificacao. Acesso em: 19 de Janeiro de 2024

[2] Patterson, David A. & Hennessy, John. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, RISC-V Edition, 2017.

[3] Riehter, Leonardo. *FPGRARS*. Disponível em: https://tinyurl.com/fpgrars. Acesso em: 19 de Janeiro de 2025

[4] Lisboa, Victor Hugo França. *LAMAR - Learning Assembly for Machine Architecture and RISC-V*. Disponível em https://tinyurl.com/fpgrars. Acesso em: 19 de Janeiro de 2025.

[5] Santos, Eduardo Freire & Leite, Ruan Petrus Alves. *Mage Embler*. Disponível em: https://tinyurl.com/leitesantos. Acesso em: 19 de Janeiro de 2025.

[6] Megiorin, Luca. *Midi2RiscV*. Disponível em: https://github.com/Luke0133/Midi2RiscV. Acesso em: 19 de Janeiro de 2025.