Guerreiro vs Dragão: Projeto de Jogo em Assembly MIPS

Bruno Alves - 147938
Reberth Kelvin Santos de Siqueira - 141589

18 de dezembro de 2025

Resumo

Este relatório detalha a implementação de "Guerreiro vs Dragão", um jogo de estratégia em turnos desenvolvido como trabalho final para a disciplina de Arquitetura e Organização de Computadores, ministrada pelo Prof. Dr. Fabio Augusto Menocci Cappa no segundo semestre de 2025 na Universidade Federal de São Paulo (UNIFESP). O projeto demonstra conceitos avançados de programação em assembly MIPS, incluindo arquitetura modular, renderização gráfica e lógica de jogo complexa. Uma característica única deste jogo é a mecânica de "Dívida de Juros Compostos", que serve como uma condição de vitória alternativa ao combate tradicional baseado em HP.

Sumário

1 Introdução

"Guerreiro vs Dragão"é um jogo de batalha gráfico em turnos onde o jogador controla um guerreiro lutando contra um dragão. O projeto foi projetado para demonstrar as capacidades da linguagem Assembly MIPS em lidar com lógica, aritmética e E/S mapeada em memória para gráficos.

O objetivo principal é derrotar o dragão reduzindo seus Pontos de Vida (HP) a zero, ou alternativamente, acumular um "Contador de Dívida"de 10.000 através de uma mecânica de juros compostos, efetivamente dominando o inimigo com estratégia econômica.

2 Arquitetura do Sistema

O projeto segue uma arquitetura modular para garantir a manutenibilidade e organização do código. A base de código é dividida em vários módulos funcionais:

- **main.asm**: O ponto de entrada da aplicação. Gerencia o loop principal do jogo, verifica as condições de fim de jogo (Vitória/Derrota) e gerencia a lógica de turnos de alto nível.
- **data.asm**: Serve como o repositório central para todas as variáveis de estado do jogo (HP, dívida, contadores de turno), constantes (cores, endereços de memória) e strings de texto (mensagens de UI, perguntas do quiz).
- **battle.asm**: Contém a lógica central de combate. Implementa as funções para os ataques do jogador, comportamento da IA do dragão, cálculos de dano e os algoritmos de juros compostos.
- **quiz.asm**: Implementa um sub-sistema educacional. Gerencia a habilidade "Quiz", lidando com a seleção de perguntas, validação de respostas e aplicação de recompensas especiais por respostas corretas.
- **rendering.asm**: O motor gráfico. Lida com escritas diretas no display mapeado em memória (0x10040000) para renderizar o céu, chão, sprites (Guerreiro e Dragão) e barras de HP dinâmicas.

3 Mecânicas de Jogo

3.1 Sistema de Combate

O combate é baseado em turnos. O jogador tem acesso a seis ações distintas:

1. **Ataque Normal**: Dano padrão com uma taxa de acerto equilibrada (80%), dano: 10-19 HP (25 crítico).
2. **Espada (Sword)**: Um movimento tático que atordoa o dragão, fazendo-o perder um turno, mas não causa dano direto. Taxa de acerto: 80%.
3. **Flanco (Flank)**: Um ataque de alto risco e alta recompensa com 40% de chance de acerto crítico. Dano: 15-24 HP (30 crítico).

4. **Lança (Lance)**: Uma postura defensiva que causa menos dano (5-9 HP, 15 crítico), mas aumenta a evasão do jogador para o próximo turno.

5. **Quiz**: Uma habilidade especial que aciona uma pergunta sobre arquitetura de computadores.

6. **Estus Flask**: Um item consumível em referência ao jogo Dark Souls. Regenera 25 HP por turno durante 2 turnos. O jogador começa com 2 frascos.

O Dragão atua como o oponente de IA com quatro comportamentos aleatórios (25% cada):

- **Sopro de Fogo**: Ataque padrão com dano de 25-40 HP (60 crítico). Taxa de acerto: 65% com 15% de chance de crítico.
- **Pisar (Stomp)**: Atordoa o jogador, reduzindo a dívida em 5%. Sem dano direto.
- **Voar (Fly)**: Aumenta a evasão do dragão, tornando-o mais difícil de acertar. Requer 50+ para acertar no próximo turno.
- **Inferno**: Um ataque devastador que causa 45-65 HP de dano com 80% de taxa de acerto, ignorando a maioria dos bônus de evasão do jogador.

3.2 Sistema de Dívida de Juros Compostos

Uma mecânica única envolvendo um "Contador de Dívida".

- **Crescimento**: Cada vez que o jogador acerta um golpe, o contador de dívida cresce 10% mais um valor base de 100.
- **Redução**: Quando o dragão atinge o jogador, a dívida é reduzida em 5%, simulando um revés.
- **Vitória**: Se o contador de dívida atingir 10.000, o jogador vence imediatamente via "Vitória por Juros Compostos".
- **Inspiração**: Esta mecânica foi inspirada nas habilidades do personagem Knuckle Bine, do anime *Hunter x Hunter*, onde o acumulo de "juros"de aura leva à derrota do oponente.

3.3 Quiz Educacional

O jogo integra conteúdo educacional diretamente na jogabilidade. A ação "Quiz"apresenta perguntas aleatórias sobre Arquitetura de Computadores (ex: sobre ULA, RAM, Barramentos).

- **Resposta Correta**: Aplica a fórmula de juros compostos 5 vezes instantaneamente, fornecendo um grande impulso para a condição de vitória por dívida.
- **Resposta Errada**: Penaliza o jogador com perda de HP.

4 Balanceamento e Dinâmica de Dificuldade

Durante o desenvolvimento iterativo, foi implementado um sistema de balanceamento estratégico para criar dinâmica de complexidade no jogo:

4.1 Ajustes de Dificuldade

- **Dano do Dragão**: Aumentado de 20-35 para 25-40 HP, tornando o inimigo mais ameaçador.
- **Chance de Crítico**: Aumentada de 5% para 15%, criando momentos de risco maior.
- **Ataque Inferno**: Novo ataque devastador (45-65 HP) com 80% de acerto, força a tomada de decisões tática.
- **Estus Flasks Limitadas**: Reduzidas de 3 para 2, criando dilema estratégico entre usar agora ou guardar.
- **Duração da Poção**: Reduzida de 3 para 2 rodadas, criando janela de vulnerabilidade.

Esta configuração cria decisões tática reais: o jogador deve escolher entre atacar agressivamente, defender-se com Lance, ou gastar seus recursos limitados de Estus Flask preventivamente.

5 Implementação Técnica

5.1 Arquitetura Modular Melhorada

Para facilitar a manutenção e reutilização de código, a arquitetura foi reorganizada em arquivos especializados:

- **macros.asm**: Define macros globais como `draw_rectangle` que são reutilizadas em todo o projeto.
- **include_all.asm**: Arquivo centralizador que garante a ordem correta de inclusão de módulos, evitando problemas de símbolos não encontrados.

A ordem de carregamento é crítica: macros → dados → sprites → lógica de batalha → quiz → renderização.

5.2 Motor Gráfico

O jogo roda em um display de 256x256 pixels com profundidade de cor de 32 bits. O módulo `rendering.asm` usa endereçamento de memória eficiente para desenhar os pixels.

- **Cálculo de Endereço**: $Base + (Y \times 256 + X) \times 4$. A multiplicação por 256 é otimizada usando um deslocamento lógico à esquerda (`sll`) de 8 bits.
- **Sprites**: Sprites são armazenados com cabeçalhos de largura e altura, e o loop de renderização ignora a cor de transparência (0x00000000) para sobrepor os personagens no fundo.

5.3 Geração de Números Aleatórios

O jogo utiliza extensivamente a chamada de sistema (syscall) 42 para gerar números aleatórios para determinar:

- Sucesso do ataque (cálculos de Acerto/Erro).
- Acertos críticos.
- Escolhas da IA do Dragão.
- Seleção de perguntas do Quiz.

5.4 Ambiente de Teste

O projeto foi desenvolvido e validado utilizando o simulador MARS (MIPS Assembler and Runtime Simulator). Para a saída gráfica, foi utilizada a ferramenta *Bitmap Display* incluída no simulador, com as seguintes configurações específicas para garantir a visualização correta:

- **Unit Width in Pixels**: 1
- **Unit Height in Pixels**: 1
- **Display Width in Pixels**: 256
- **Display Height in Pixels**: 256
- **Base address for display**: 0x10040000 (heap)

6 Desafios e Decisões de Projeto

Durante o desenvolvimento do projeto, foram identificados desafios técnicos significativos, especialmente relacionados à renderização gráfica. Um dos principais obstáculos foi o desenho do *bitmap*, onde inicialmente tentou-se utilizar um endereço de memória estático para a manipulação dos pixels. No entanto, para o correto funcionamento com a ferramenta de display gráfico do simulador, deveríamos ter utilizado o endereço de memória da *heap* (dinâmica). Essa divergência causou dificuldades iniciais na exibição correta das sprites e cores na tela, exigindo uma refatoração do código de renderização para apontar para o endereço base correto (0x10040000).

Além disso, devido à complexidade inerente ao desenvolvimento em baixo nível com Assembly, foi tomada a decisão de priorizar a profundidade e robustez das mecânicas de jogo — como o sistema de combate, as perguntas do quiz e o cálculo de juros compostos — em detrimento de uma apresentação visual mais elaborada. O foco principal foi garantir que a lógica do jogo funcionasse perfeitamente, mantendo os gráficos funcionais, porém simples, para assegurar a entrega de um sistema estável e livre de bugs críticos.

7 Conclusão

O projeto "Guerreiro vs Dragão"cria com sucesso uma aplicação interativa e envolvente usando linguagem Assembly de baixo nível. Ele demonstra que lógica complexa, design de software modular e interfaces gráficas podem ser efetivamente implementados mesmo sem abstrações de alto nível, fornecendo insights profundos sobre arquitetura de computadores e operações em nível de máquina.