Projeto Final - Grupo 1 - Metroid

Luca Heringer Megiorin [*] João Pedro G.C.M. Antonow [†] Caleb Martim de Oliveira [‡] Ítalo Braga de Paulo [§]
Luis Augusto da Silveira Cavalcanti[¶]

Universidade de Brasília, 17 de setembro de 2024

Figura 1: *Jogo do Grupo 1 - Samus e Zoomers*

RESUMO

Metroid (1986 – Nintendo) é um jogo feito para o *Nintendo Entertainment System*. O objetivo do projeto foi recriar tal jogo em Assembly com a arquitetura RISC-V por meio dos emuladores RARS [4] e FPGRARS [3] e da simulação na DE1-SOC [5]. Nessa disciplina de Organização e Arquitetura de Computadores do Departamento de Ciência da Computação da Universidade de Brasília, o foco principal do aplicativo foi familiarizar e introduzir os alunos à lógica em baixo nível com a ISA RISC-V RV32IMF. A base do projeto consistiu em emular o jogo de forma mais fiel possível à versão original de Metroid, de forma que foram desenvolvidas uma série de técnicas por meio de efeitos visuais, otimizações via hardware, uma jogabilidade verossímil à versão do original, adequada à interface do teclado conectável a FPGA, e diversas técnicas de programação em RISC-V que fossem adequadas ao gerenciamento de arquivos e minimizassem o uso de instruções e/ou o poder de processamento para executá-las.

Palavras-chave: OAC · Assembly · RISC-V · Metroid · FPGA · DE1-SOC · Renderização

1 INTRODUÇÃO

Metroid é um jogo de ação-aventura desenvolvido em 1986 pela Nintendo. O jogador, que visualiza o estilo visual sidescrolling 2D nativo do jogo, controla a personagem Samus Aran - *Samusu Aran (サムス・アラン)*, que se move em apenas duas dimensões e ataca com armas especiais em um cenário com inimigos, sendo seu objetivo sobreviver e passar para o próximo cenário, desbravando o mapa; para isso, devem ser utilizadas habilidades especiais e as armas da Samus, de forma que essas possibilitarão a entrada/saída a novos locais e capacitarão a Samus a derrotar inimigos que possam prejudicá-la em sua jornada.

[*]231003390@aluno.unb.br
[†]221006351@aluno.unb.br
[‡]221017060@aluno.unb.br
[§]221002049@aluno.unb.br
[¶]231003415@aluno.unb.br

2 FUNDAMENTAÇÃO TEÓRICA E TRABALHOS RELACIONADOS

A fundamentação dos efeitos visuais e sonoros baseou-se nos tutoriais e vídeos criados pelos monitores de Organização e Arquitetura de Computadores, em especial o "Livrão de OAC"[10], para melhores conhecimentos da ISA RISC-V e melhores noções de implementação de jogos, e o website The Spriters Resourcer [6] , para obtenção adequada de *sprites* necessárias para o funcionamento do jogo

A fundamentação teórica em termos de implementação de código consistiu em análise de códigos antigos dos melhores projetos anteriores e experiência prévia de implementação de um jogo na disciplina Introdução a Sistemas Computacionais, com o jogo Gauntlet [8] de 2023/1; as maiores inspirações para o desenvolvimento do projeto foram o jogo Celeste [9] De 2021/1 e o jogo Castlevania [12] de 2021/2. Grande parte das mecânicas novas e técnicas de renderização foi implementada de forma original - mediante erros e acertos - pelo integrante do grupo Luca Heringer, que também liderou o projeto e propôs um efetivo planejamento para que o trabalho fosse exitoso.

Em termos de conhecimento da ISA RV32IMF, o trabalho foi fundamentado com os conhecimentos adquiridos ao longo das aulas da disciplina de Organização e Arquitetura de Computadores e Introdução aos Sistemas Computacionais, além de constante acompanhamento do livro Guia Prático RISC-V [1], de forma a minimizar o uso de instruções e maximizar a performance, legibilidade e reúso de código.

3 METODOLOGIA

O objetivo central foi criar a versão mais verossímil possível do jogo Metroid, dentro das limitações da DE1-SOC e atendendo aos requisitos do projeto, de forma que as técnicas de renderização, fí-

sica e a modularização em si do código foram pensadas visando a otimização de instruções, memória e fluidez na renderização, fatores esses que influenciaram no tempo e esforço demandados para o desenvolvimento do projeto, mas se mostraram assertivos, vide o resultado final exitoso.

3.1 Modularização

Para que o projeto fosse modular e com suas frações funcionando, na maior parte, independentemente, decidiu-se que esse seria dividido em vários módulos, dentre eles: *collision.s*, *data.s*, *enemies.s*, *entities_op.s*, *helpers.s*, *input_fpga.s*, *map_op.s*, *music++.s*, *physics.s*, *player_attacks.s*, *render.s*, *setup.s*, *status_operations.s*, além dos arquivos *main.s*, *MACROSv24.s* e *SYSTEMv24.s*. Essencialmente, os arquivos *helpers.s* e *data.s* condensam os demais, sendo referência para as diretivas *.text* e *.data*, respectivamente. Os arquivos estão bem documentados, fator que melhora a legibilidade e revisão de código. Além disso, faz-se mencionar modificações no *SYSTEMv24.s*, fator abordado na seção 3.10.

3.2 Renderização

3.2.1 Desenvolvimento

O módulo de renderização desenvolvido ao longo do projeto tem como base o procedimento de renderização de *sprites* – *RENDER* e *RENDER_WORD* – e o procedimento de renderização do mapa – *RENDER_MAP*, haja vista o uso diferenciado para cada um e a proposta do grupo em generalizar os procedimentos com o intuito de promover o reúso de código.

3.2.2 Sprite

O procedimento de renderização – *RENDER* (realizado pixel pixel) e *RENDER_WORD* (4 pixels por vez) – se trata de um procedimento que recebe o endereço de uma sprite, as cooredenadas iniciais da renderização (X e Y), sua altura e largura, o status da sprite (irá modificar o endereço inicial da sprite, deslocando-o baseado na altura da sprite dada), que é usado para a animação de *sprites*, e caso deve ou não renderizar a sprite cortada, e a renderiza no *bitmap display* e no monitor via *VGA* pelo *Computador RISC-V*. Esse procedimento é reutilizado diversas vezes ao longo do módulo de renderização e é o sustentáculo do jogo e da experiência de Metroid.

3.2.3 Mapa

O procedimento de renderização do mapa – *RENDER_MAP* – invoca o procedimento de renderização de sprites – *RENDER_WORD* – diversas vezes, com cada chamada envolvendo um *tile*. Os *tiles* são distribuídos em matrizes do mapa a partir de cores (em ordem crescente, a partir de 0), de modo que, era possível acessar o endereço de um *tile* sem usar inúmeras comparações (apenas era necessário adicionar o valor do *tile* multiplicado por seu tamanho de 256 bytes). A partir das coordenadas do mapa, o seu offset, e mediante o tamanho da tela do jogo (16 vezes maior que a matriz), verifica-se o *tile* correspondente à posição na tela e é iniciada a renderização do *tile*, até que toda a tela seja renderizada. Os dados correspondentes à posição do mapa e o tamanho da tela estão presentes no módulo *data.s*.

Figura 2: *Mapa Utilizado no Jogo (junção de mapas originais)*

3.2.4 Personagem

A partir da referência [6], foram retiradas as *sprites* dos personagens, que foram separadas em diferentes conjuntos de imagens, posicionadas no sentido vertical, para que a próxima *sprite* da animação fosse obtida apenas ao modificar o campo do *status* da *sprite* no procedimento de renderização presente em 3.2.2. Assim, sempre que a personagem se movesse, a troca entre uma sprite e outra geraria uma sensação fluida de movimentação.

Figura 3: *Algumas das sprites usadas para a Samus*

3.2.5 Rastro

A limpeza de rastro fora implementada por meio da localização das posições antigas da personagem/dos inimigos e renderizando a parte do mapa correspondente. Entretanto, após analizar que o procedimento *RENDER_WORD* era custoso para ser chamado para cada entidade no jogo, foi decidido que renderizar o mapa inteiro a cada loop do jogo era suficiente para apagar o rastro de cada *sprite*.

3.2.6 Fonte

A partir do site [7], foi possível adquirir as imagens para cada letra usada nas *ecalls* para print no *bitmap display* na fonte parecida com a do jogo original. Por meio do uso do software *Krita* [14], as imagens foram cortadas e convertidas [11] para *.data* e, a partir disso, transformadas no padrão presente no arquivo *SYSTEMv24.s*, no qual foi adicionado o argumento *a5* às *ecalls* de *print* no *bitmap display*. Alguns dos resultados podem ser vistos abaixo:

Figura 4: *Tela Inicial*

Figura 5: *Game Over*

3.3 Samus

3.3.1 Atributos

A personagem Samus, protagonista do jogo Metroid, essencialmente possui como atributos a vida/*health points*, o *status* da jogadora em questão(número da *sprite*, orientação vertical/horizontal, estado de ataque etc.), as posições da Samus em relação ao *bitmap display* e à matriz do mapa, e os offsets da Samus em relação às suas coordenadas da matriz, que são atualizadas conforme a movimentação do jogador.

3.3.2 Tiro

Para que a Samus desfira tiros, basta que seja pressionada a tecla 'K', quando a Samus estiver em pé, de forma que isso liberará o projétil assegurado pela Samus e poderá ter um alcance que venha a impactar a vida de um inimigo e/ou desbloquear uma porta para uma nova parte do mapa.

Figura 6: *Samus - Tiro*

3.3.3 Morph Ball

MorphBall é uma habilidade que a Samus é capaz de desbloquear para adaptar a sua armadura para uma esfera quase perfeita[15]. A Samus permanece girando para a direção que está olhando, e, em razão do seu tamanho reduzido, consegue acessar novas partes do mapa e/ou acessar lugares anteriormente inacessíveis. Outra característica importante é que somente nesse estado é possível efetuar o lançamento de bombas, um item especial que causa danos em inimigos e quebra certos tipos de *tiles*. A habilidade de evocar a forma de *Morph Ball* é desbloqueada pela Samus após a personagem coletar a *Maru Mari* no início da primeira sessão do jogo. Para que a Samus entre nesse estágio, basta apertar a tecla 'S' quando estiver no solo.

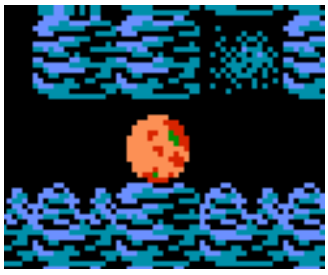

Figura 7: *MorphBall*

3.3.4 Bomba

A bomba é um item especial que, ao ser adquirido no mapa 6, serve para a Samus como arma para ataques contra inimigos e/ou para quebrar *tiles* especiais que impeçam a passagem da Samus. Para que ela seja ativa, basta que a Samus esteja na forma *Morph Ball* e aperte a tecla 'K'.

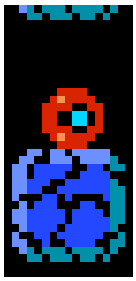

Figura 8: *Bomba ativa*

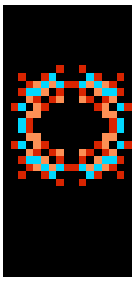

Figura 9: *Explosão*

3.4 Movimentação, colisão e interação

3.4.1 Movimentação dos Personagens

O módulo *collision.s* e *physics.s* trabalham conjuntamente para permitir a movimentação da Samus, com o intuito de verificar os *tiles* que podem estar no caminho da Samus a cada ciclo de *Game Loop* e permitir a movimentação no eixo livre de obstáculos. A Samus pode se mover no eixo vertical e horizontal, podendo pular - com baixa impulsão ou alta impulsão, dependendo do tempo em que a tecla foi pressionada - e também se transformar em *MorphBall*, apenas podendo sair deste estado caso não haja - *tiles* bloqueando acima dela. Os inimigos possuem comportamento análogo, cada um com sua mecânica particular, como descrito na seção 3.5.

3.4.2 Colisão com o Ambiente

Para a colisão da Samus e dos demais personagens,projéteis ou bombas com os elementos do jogo, foi desenvolvido um módulo específico para lidar com todos os tipos de colisão, o *collision.s*, que performará uma série de verificações a cada iteração do *Game Loop*, fazendo com que, em caso positivo para colisões, forneça o resultado esperado para o respectivo contato.

3.4.3 Interação com os Itens e Inimigos

O jogo em questão permite a interação da protagonista com uma série de elementos do jogo, dentre eles os 'itens especiais', que desbloqueiam habilidades especiais da Samus, as 'recompensas de inimigos', em que, ao derrotar o Zoomer, a Samus consegue coletar itens adicionam 5 *health points* em sua vida.

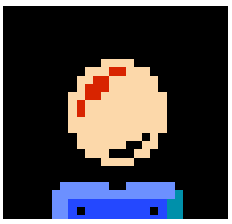

Figura 10: *Maru Mari*

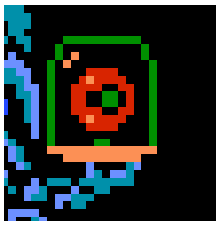

Figura 11: *Bomba*

3.4.4 Física

Para a física do jogo, foi necessária a utilização de instruções de ponto flutuante, tanto para a Samus quanto para o Ridley, assim como a trajetória dos tiros do Ridley. Empiricamente, verificou-se que a utilização de tais instruções conferia uma precisão dos movimentos e trajetórias muito maior que de instruções RV32IM, fator que poderia ter influenciado no baixo desempenho do jogo, caso não fossem feitas otimizações via hardware, como demonstrado na seção 3.9.

Figura 12: *Samus - Pulo*

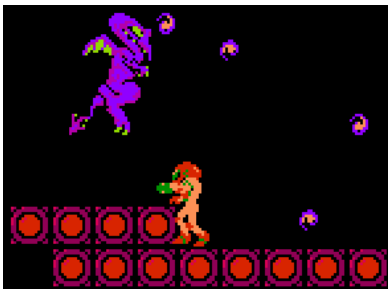

Figura 13: *Ridley - Pulo*

3.5 Inimigos e projéteis

Foram implementados 3 tipos distintos de inimigos, cada um com uma IA diferente - como requerido no projeto -, que são capazes de ter sua cor alterada quando feridos pela Samus (aqueles que podem receber dano). São eles:

3.5.1 Inimigo 1 - Ripper

O primeiro tipo de inimigo é o Ripper, que tem seu funcionamento emulado em uma máquina de estados simples, mas fiel ao jogo, que faz com que o inimigo se mova no eixo horizontal até encontrar um obstáculo e, nesse momento, inverte sua orientação no mesmo eixo. A samus, contato com esse inimigo, leva um dano de 6 *health points*. Existem dois tipos de Ripper: ambos possuem o mesmo funcionamento, distinguindo-se basicamente em suas cores. Não é possível matar esse inimigo, ou seja, nenhum dano ou arma especial é capaz de afetá-los.

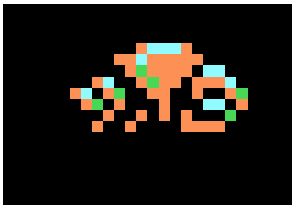

Figura 14: *Ripper*

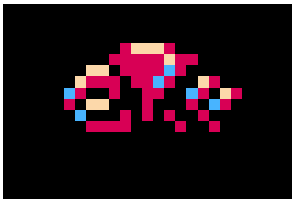

Figura 15: *Red Ripper*

3.5.2 Inimigo 2 - Zoomer

O segundo inimigo, o Zoomer, comporta-se via uma máquina de estados mais sofisticada, cuja movimentação é feita nos eixos horizontal e vertical e, em caso de fim de um *tile* em determinado eixo, ele continuará percorrendo-o no eixo contrafio. Fora isso, ele é capaz de retirar 6 *health points* da vida da Samus. Existem dois tipos de Zoomer: a versão original e a versão variante vermelha, que diferenciam-se na quantidade de tiros necessários para abater o inimigo; para que o zoomer seja abatido, são necessários 6 tiros da Samus ou um ataque especial com a bomba. Para a versão vermelha, são necessários 12 tiros, ou 2 ataques com a bomba. Quando um Zoomer é eliminado, ele deixa para trás um item que, caso a Samus o pegue, concede-a 5 *health points*.

Figura 16: *Zoomer*

Figura 17: *Zoomer Red*

3.5.3 Inimigo Final - Ridley

Por fim, o último inimigo e o 'chefão do jogo' é o Ridley, que tem sua movimentação baseada em uma máquina de estados, que é regida por um contador responsavel por fazê-lo pular quando chega a zero, e, a qualquer momento, são gerados números aleatórios que podem lançar projéteis de plasma com velocidade aleatórias, visando atingir a Samus. O ataque do Ridley é feito a partir de instruções de ponto flutuante - *Single Precision IEEE 754* - com velocidade randômica, a fim de emular melhor o lançamento dos projéteis (apenas 5 podem estar na tela ao mesmo tempo) e se assemelhar mais com o jogo original. Em termos da mecânica para derrotá-lo,são necessários 30 tiros ou qualquer combinação entre bombas e tiros que resulte em 30 *health points* no total.

Figura 18: *Ridley*

3.6 CheatCodes

Haja vista que o projeto em questão é uma emulação fiel ao jogo original, que tem sua dificuldade um pouco elevada, foram desenvolvidos alguns *CheatCodes* que vão ser mapeados nessa seção, tais quais, ao apertar a respectiva tecla, possuem os funcionamentos a seguir :

1. **J** : permite que a samus seja invencível, o que não possibilita perda de health points caso haja algum ataque inimigo. item
2. **[1:7]** : as sete teclas, de 1 a 7, permitem a transição entre um mapa e outro, cada um referenciado conforme a respectiva tecla.
3. **0** : gera dano no jogador.
4. **O** : abrem-se as portas do mapa.

Figura 19: *Samus Invencível*

3.7 Música

Por meio de um conversor de midi novo, criado por um dos componentes do grupo, foi possível converter arquivos midi em sequências com notas e durações, considerando pausas e harmonias, foram obtidas as sequências para tocar a música *"Metroid - Brinstar Theme"* [16]. O arquivo *music.s* havia um protótipo do código de música, se baseando em trabalhos de outros semestres. Porém, após obter o *.data* para a música completa, foi necessário alterar os procedimentos de música para conseguir tocar várias melodias e harmonias ao mesmo tempo. Para isso, foi criado o arquivo *music++.s*, que resolvia esse probleutilizado no módulo *music* do código. Precipuamente, ratifica-se a inspiração no módulo de música do jogo Celeste [9], que, com algumas modificações, serviu de base para a implementação desse requisito no presente projeto.

3.8 Problemas e desafios

O maior problema para o desenvolvimento do projeto foi a complexidade relacionada à introdução de uma mecânica nova no jogo, já que normalmente a maioria das mecânicas do jogo não poderiam ser reutilizadas e envolveram um grande tempo de implementação e esforço referente a *brainstorm* de ideias, além da elevada dificuldade de solução de bugs que viriam a ocorrer em cada mecânica de forma diferenciada.

A renderização, por sua vez, sendo a primeira mecânica do jogo, foi um dos maiores obstáculos também, já que o jogo dependia inteiramente dos efeitos visuais e quaisquer alterações no código ou adições de outras mecânicas no jogo envolviam o desenvolvimento de procedimentos adicionais de renderização, além do esforço de organização dos parâmetros e da integração com o código original, já que o código em *assembly* exigia uma organização adequada e a conciliação com os 32 registradores disponíveis para a implementação foi um fator difícil de ser manejado. O esforço de implementação do módulo de renderização foi extenso e com duração de quase quatro meses. Outra fração do projeto que o grupo considerou como um objetivo de dificuldade alta foi o desenvolvimento da transição fluida entre dois mapas; faz-se notar que essa *feature* do projeto foi desenvolvida ao longo de duas semanas, mas teve êxito no intuito de simular mais adequadamente o jogo original e permitir fluidez na transição entre dois mapas.

3.9 Otimização do RISC-V

As otimizações essencialmente consistiram em criar circuitos IPS para as operações mais utilizadas na ULA e na FPULA, de forma a minimizar os ciclos necessários para a execução de uma instrução. Para isso, foram utilizados artefatos provenientes do *IP Catalog*, componente do *Quartus* que permite a criação e instanciação de circuitos *IP* para o projeto. A otimização permitiu o aumento da frequência do *Clock* usado no projeto, tendo como base o processador uniciclo advindo do Computador RISC-V [13].

3.10 SYSTEMv24 - BUG

A implementação na fpga revelou ao grupo que havia um erro na implementação do código SYSTEMv24, especificamente na label *Random2.DE1*, em que a chamada a um procedimento era feita, mas o registrador *ra* não era salvo na pilha e, portanto, perdia-se o endereço de retorno do código. Para isso, bastou a seguinte modificação, com a inserção na pilha:

```
Random2.DE1:  li  t0, LFSR  # carrega endereco do LFSR
  lw  a0, 0(t0) # le a word em a0
  addi sp,sp,-4
  sw ra,0(sp)
  jal   __umodsi3
  #remu   a0,a0,a1  # numero entre 0 e a1
  lw ra,0(sp)
  addi sp,sp,4
  ret     # retorna
```

Listing 1: *Random2.DE1*

4 Resultados obtidos

O jogo está funcionando de maneira consistente no FPGRARS [3] e na DE1-SOC e é possível jogar e concluir o jogo sem grandes entraves. A divisão de frequência do Clock mínima para o jogo funcionar plenamente é 6, usando as chaves *SW[2:0]*), mas a frequência de *Clock* pode ser dividida por valores maiores. As principais características do jogo original, como movimentação em duas dimensões em estilo plataforma, uso de projéteis para atacar os inimigos, obtenção de itens para liberar novos cenários estão presentes na versão final do projeto em questão, requerendo elevado esforço e tempo para implementação e revisões de código, além da testagem em larga escala.

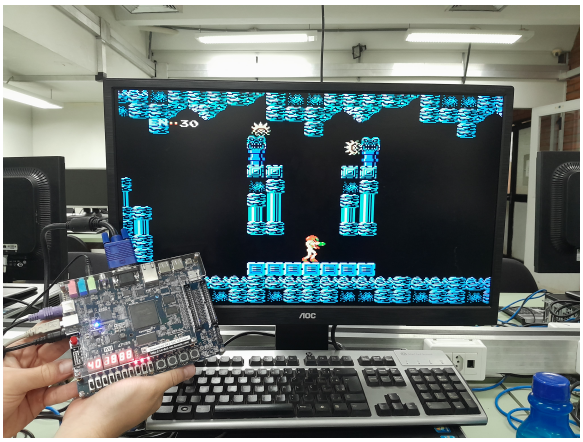

Figura 20: *Jogo implementado na FPGA*

5 Conclusões e Trabalhos futuros

Como os resultados obtidos atendem os requisitos exigidos pelo projeto da discplina de Arquitetura e Organização de Computadores, pode-se considerar o projeto como bem sucedido. Os membros do grupo se familiarizaram com a linguagem em baixo nível e aprimoraram suas habilidades ao decorrer do projeto.

Referências

[1] Patterson, David A., and Waterman, Andrew. *The RISC-V Reader: An Open Architecture Atlas*. First edition. Strawberry Canyon LLC, 2017. ISBN 978-0-9992491-1-6.

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

[3] Riether, Leonardo. *FPGRARS*. Disponível em: https://t.ly/kF-2w. Acesso em: 19 de abril de 2024.

[4] The Third One. *RARS*. Disponível em: https://t.ly/158he. Acesso em: 19 de abril de 2024.

[5] Altera. *DE1-SOC*. Disponível em: https://shorturl.at/XVdst. Acesso em: 19 de abril de 2024.

[6] Dazz & Petie. *The Spriters Resource*. Disponível em: https://shorturl.at/OW6vX. Acesso em: 21 de abril de 2024.

[7] *Font2Bitmap*. Disponível em: https://shorturl.at/q5qYn. Acesso em: 26 de abril de 2024.

[8] Heringer, Luca. *The Assembly Gauntlet, ISC Final Project*. Disponível em: https://t.ly/ezrDG. Acesso em: 22 de abril de 2024.

[9] Lisboa, Victor H; Paturi, Davi; Schweizer, Ana. *Projeto Final - Celeste*. Disponível em: https://shorturl.at/djSwt. Acesso em: 23 de abril de 2024.

[10] de Paula, Thiago T. *Livrão de OAC*. Disponível em: https://shorturl.at/PiR4w. Acesso em: 22 de abril de 2024.

[11] Lisboa, Victor H. *Gerenciador de Conversão - bmp → .data*. Disponível em: https://t.ly/_sW0e. Acesso em: 22 de abril de 2024.

[12] Fernandes, Filipe. *Projeto Final - Castlevania*.

[13] Lamar, Marcus V. *Computador RISC-V*.

[14] Ettrich, Matthias. *Krita*. Disponível em: https://krita.org/. Acesso em: 25 de abril de 2024.

[15] Suit, Brandon. *Strategy Guide - Metroid* Disponível em: https://tinyurl.com/luke1333. Acesso em: 5 de maio de 2024.

[16] *Kingdom Hearts Insider - Metroid* https://tinyurl.com/midiaoac Acesso em: 20 de maio de 2024.