UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

LUCAS ZISCHLER

# RENDERIZAÇÃO 3D EM SISTEMAS EMBARCADOS UTILIZANDO RISC-V ASSEMBLY

APUCARANA
2022

LUCAS ZISCHLER

# RENDERIZAÇÃO 3D EM SISTEMAS EMBARCADOS UTILIZANDO RISC-V ASSEMBLY

## 3D Rendering in embedded systems utilizing RISC-V assembly

Trabalho de conclusão de curso apresentado como requisito para obtenção do título de Bacharel em Engenharia Elétrica da Universidade Tecnológica Federal do Paraná (UTFPR).

Orientador: Bruno de Nadai Nascimento
Universidade Tecnológica Federal do Paraná

Coorientador: Carlos Matheus Rodrigues de Oliveira
Universidade Tecnológica Federal do Paraná

APUCARANA
2022

**LUCAS ZISCHLER**

# RENDERIZAÇÃO 3D EM SISTEMAS EMBARCADOS UTILIZANDO RISC-V ASSEMBLY

Trabalho de conclusão de curso apresentado como requisito para obtenção do título de Bacharel em Engenharia Elétrica da Universidade Tecnológica Federal do Paraná (UTFPR).

Data de aprovação: 13/Junho/2022

---

Bruno de Nadai Nascimento
Doutorado
Universidade Tecnológica Federal do Paraná

---

Carlos Matheus Rodrigues de Oliveira
Doutorado
Universidade Tecnológica Federal do Paraná

---

Vinícius Dário Bacon
Doutorado
Universidade Tecnológica Federal do Paraná

---

André Luiz Tinassi D'Amato
Doutorado
Universidade Tecnológica Federal do Paraná

**APUCARANA**
**2022**

Este espaço eu dedico para os meus gatos, ao Faísca, ao Neguinho, à Neguinha, aos filhotes Florzinha, Bob, e em especial, à Lindinha por sempre me fornecer companhia no desenvolvimento deste trabalho, à Pandora e seus filhotes, Frajola e Paçoca, ao Figo, aos filhotes, Carvão, Fumaça, Tigor, Abigail, Emily, Nebula, e Tesla, à Pérola, e ao Tom.

Dedico também aos meus cachorros, ao Bili, à Lili, à Mel, ao Rick, à Amora, e ao Banzé. E em especial, à Pedrita, por ser o motivo principal deste trabalho existir, e por me fornecer a motivação necessária para seguir em frente apesar das diversas dificuldades encontradas. Espero que este trabalho seja um lembrete de seu nome.

*Muito obrigado.*

## AGRADECIMENTOS

Agradeço ao professor coorientador deste projeto, Carlos Matheus Rodrigues de Oliveira, por fornecer a ideia inicial que viria a culminar neste trabalho. Agradeço também, em especial, o professor orientador, Bruno de Nadai Nascimento, pelo auxílio no desenvolvimento deste trabalho, por lidar com as burocracias relacionadas, e pela confiança concedida a mim.

“*O desenvolvimento progressivo do homem é vitalmente dependente da invenção. Ela é o produto mais importante de seu cérebro criativo. Seu propósito final é o domínio completo da mente sobre o mundo material, a subordinação das forças da natureza às necessidades humanas. Esta é a difícil tarefa do inventor, que geralmente é incompreendido e não recebe recompensa. Porém, ele encontra ampla compensação no agradável exercício de suas capacidades e por saber que pertence à classe excepcionalmente privilegiada sem a qual o homem teria perecido há muito na dura luta contra impiedosos elementos.*”

**(TESLA, Nikola, 1919)**.

## RESUMO

O desenvolvimento de tecnologias de renderização tridimensional vem, prioritariamente, focando na melhora de qualidade de imagem, com viés reduzido à otimização energética e econômica dos dispositivos que a processam. Neste trabalho, busca-se a otimização deste meio ao aplicar técnicas de renderização 3D a um sistema embarcado de baixo consumo energético, utilizando-se, em sua base, o conjunto de instruções de arquitetura RISC-V. A arquitetura de um processador influência no desenvolvimento de seu software, por ser o que fornecerá as ferramentas de construção de algorítimos. O RISC-V apresenta diversas otimizações em sua estrutura que são uteis na programação de alta eficiência, acessíveis de forma direta pela linguagem assembly. Para a renderização tridimensional de um objeto, são necessárias as conversões de seus pontos em coordenadas tridimensionais para um plano bidimensional. Estas conversões são realizadas pelas matrizes de transformação, e realizarão as transformações necessárias para projetar o modelo em um cubo de projeção. Com pontos em coordenadas bidimensionais, seus valores podem ser rasterizados para a projeção em pixels do modelo.



**Palavras-chave**: RISC-V; Assembly; Renderização 3D; Sistemas Embarcados.

## ABSTRACT

The development of tridimensional rendering technologies is, primarily, focused on the image's quality improvement, with less regard to the energetic and economic optimization of the processing devices. In this work, it is pursued otimizing this medium. Using 3D rendering techniques in an embedded system with low energy consumption while using the RISC-V instruction set architecture as a foundation. The processor's architecture has an influence on the development of the software, because it is the one that gives the tools for the algorithm's development. RISC-V has a wide range of optimizations in its structure, useful for high-efficiency programming and accessible directly via assembly language. For an object's tridimensional rendering, it is necessary to convert the points from a tridimensional to a bidimensional plane. Those conversions are realized by the transformatrion matrices, which realize the necessary transformations in order to project the model into a projection cube. With the points in bidimensional coordinates, those values can be rasterized into the model's pixel projection.



**Keywords**: RISC-V; Assembly; 3D Rendering; Embedded Systems.

# LISTA DE ILUSTRAÇÕES


Figura 1 – Rasterização com um disco de Nipkow em um televisor CRT . . . . . 18
Figura 2 – Componentes de uma instrução assembly . . . . . 33
Figura 3 – Testes de contagem de instruções e tamanho de programa de diferentes ISAs 38
Figura 4 – Renderização de modelos 3D em *desktop* utilizando OpenGL . . . . . 41
Figura 5 – Pipeline de renderização do OpenGL ES . . . . . 41
Figura 6 – Demonstração do uso de dados contidos nos *vertexes* . . . . . 42
Figura 7 – Formas de mapeamento para valores de UV fora dos limites . . . . . 43
Figura 8 – Aplicação de iluminação especular . . . . . 45
Figura 9 – Transformação da matriz de modelo para o mundo . . . . . 47
Figura 10 – Transformação da matriz de projeção . . . . . 50
Figura 11 – Rasterização de polígonos à tela . . . . . 53
Figura 12 – Retas construídas pelo algoritmo de Bresenham . . . . . 53
Figura 13 – Preenchimento pelo algoritmo *scan line* . . . . . 56
Figura 14 – Placa de desenvolvimento Maix Dock . . . . . 63
Figura 15 – Diagrama de conexão de dispositivos por SPI . . . . . 66
Figura 16 – Fluxograma da configuração do *display* ST7789V . . . . . 69
Figura 17 – Pinos do cartão micro SD via comunicação SPI . . . . . 70
Figura 18 – Fluxograma da configuração do cartão SD . . . . . 74
Figura 19 – Fluxograma da leitura de um arquivo em um sitema FAT32 . . . . . 82
Figura 20 – Exemplo de um arquivo de uma imagem no formato .ppm . . . . . 85
Figura 21 – Fluxograma da função `_start` . . . . . 88
Figura 22 – Fluxograma da função `initBSP` . . . . . 90
Figura 23 – Fluxograma da função `setupPLL` . . . . . 91
Figura 24 – Fluxograma da função `tftSetup` . . . . . 93
Figura 25 – Fluxograma da função `sdSetup` . . . . . 93
Figura 26 – Fluxograma da função `tftWriteDMA` . . . . . 95
Figura 27 – Fluxograma da função `sdSPIwrite` . . . . . 97
Figura 28 – Fluxograma da função `sdSPIread` . . . . . 97
Figura 29 – Fluxograma da função `readFile` . . . . . 100
Figura 30 – Fluxograma da função `openOBJModel` . . . . . 103
Figura 31 – Fluxograma da função `openPPMTex` . . . . . 104
Figura 32 – Renderização de linhas do modelo . . . . . 108
Figura 33 – Identificação dos *buffers* de trabalho por cor . . . . . 108
Figura 34 – Aplicação da iluminação no modelo . . . . . 109
Figura 35 – Fluxograma das etapas de *loop* do algoritmo . . . . . 110
Figura 36 – Rotação de um modelo com diferença de 1 s entre quadros . . . . . 111

Figura 37 – Variações dos modelos e texturas utilizados para testes . . . . . 114
Figura 38 – Tempo de inicialização entre modelos . . . . . 115
Figura 39 – Tempo de leitura do cartão SD . . . . . 115
Figura 40 – Variação da taxa de quadros entre modelos . . . . . 116

# LISTA DE TABELAS

Tabela 1 – Armazenamento dos pontos flutuantes de acordo com a IEEE 754 . . . . . 28
Tabela 2 – Formato das instruções de 32 bits do RISC-V . . . . . 31
Tabela 3 – Formato das instruções compactas de 16 bits do RISC-V . . . . . 32
Tabela 4 – Hierarquias de memórias em um computador em 2015 . . . . . 59
Tabela 5 – Formato do comando SD . . . . . 71
Tabela 6 – Formato da resposta dos comandos SD . . . . . 71
Tabela 7 – Formato da resposta de leitura de dados do cartão SD . . . . . 74
Tabela 8 – Campos de tempo, e data de uma entrada curta . . . . . 81
Tabela 9 – Organização interna dos arquivos do projeto . . . . . 87
Tabela 10 – Contagem das instruções das funções . . . . . 113

# LISTA DE QUADROS


Quadro 1 – Instruções bases e algumas extensões da arquitetura RISC-V . . . . . 27
Quadro 2 – Alguns CSRs definidos na ISA do RISC-V . . . . . 29
Quadro 3 – Convenções da ABI do RISC-V para registradores . . . . . 33
Quadro 4 – Lista de alguns mnemônicos de instruções base do RISC-V . . . . . 35
Quadro 5 – Lista de comandos do *display* ST7789V referentes ao trabalho . . . . . 67
Quadro 6 – Código de cores do *display* ST7789V . . . . . 68
Quadro 7 – Funções do registrador de controle do *display* ST7789V . . . . . 68
Quadro 8 – Lista de comandos do protocolo SD referentes ao trabalho . . . . . 71
Quadro 9 – Significado dos bits de estado do cartão SD . . . . . 72
Quadro 10 – Valores do OCR do cartão SD . . . . . 73
Quadro 11 – Definição das regiões do MBR . . . . . 77
Quadro 12 – Definição das regiões da entrada da partição . . . . . 77
Quadro 13 – Definição das regiões do setor de inicialização da partição . . . . . 78
Quadro 14 – Regiões da entrada longa de um arquivo . . . . . 80
Quadro 15 – Regiões da entrada curta de um arquivo . . . . . 81

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| CRT | Tubo de raios catódicos, de *Cathode Ray Tube* em inglês |
| LCD | Display de cristal líquido, de *Liquid Crystal Display* em inglês |
| CAD | Design auxiliado por computador, de *Computer-Aided Design* em inglês |
| GPU | Unidade de processamento gráfico, de *Graphical Processing Unit* em inglês |
| CPU | Unidade central de processamento, de *Central Processing Unit* em inglês |
| CNC | Controle Numérico Computadorizado |
| IHM | Interface Homem-Máquina |
| CISC | Computador de conjuto de instruções complexas, de *Complex Instruction Set Computer* em inglês |
| RISC | Computador de conjuto de instruções reduzido, de *Reduced Instruction Set Computer* em inglês |
| ISA | Conjuto de instruções de arquitetura, de *Instruction Set Architecture* em inglês |
| opcode | Código de operação, de *operational code* em inglês |
| hart | *Hardware thread* |
| CSR | Registrador de controle e estado, de *Control Status Register* em inglês |
| ABI | Interface de aplicação binária, de *Application Binary Interface* em inglês |
| RAM | Memória de acesso aleatório, de *Random-Access Memory* em inglês |
| API | Interface de programação de aplicação, de *Application Programming Interface* em inglês |
| pixel | Elemento de imagem, de *picture element* em inglês |
| FOV | Campo de visão, de *Field Of View* em inglês |
| FET | Transistor de efeito de campo, de *Field-Effect Transistor* em inglês |
| DRAM | RAM dinâmica, de *Dynamic RAM* em inglês |
| SRAM | RAM estática, de *Static RAM* em inglês |

| | |
|---|---|
| DMA | Acesso direto à memória, de *Direct Memory Access* em inglês |
| OoO | Fora de ordem, de *Out-of-Order* em inglês |
| TFT | Transistor de película fina, de *thin-film-transistor* em inglês |
| SYSCTL | Controlador do sistema, de *System Controller* em inglês |
| PLL | Malha de captura de fase, de *Phase-Locked Loop* em inglês |
| FPIOA | Matriz de campo programável de entrada/saída, de *Field Programmable Input/Output Array* em inglês |
| GPIO | Interface de propósito geral de entrada/saída, de *General Purpose Input/Output Interface* em inglês |
| SPI | Interface de periférico serial, de *Serial Peripheral Interface* em inglês |
| CRC | Verificação cíclica de redundância, de *Cyclic Redundancy Check* em inglês |
| OCR | Registrador de operação e condição, de *Operation Condition Register* em inglês |
| FAT | Tabela de alocação de arquivos, de *File Allocation Table* em inglês |
| BSP | Pacote de apoio à placa, de *Board Support Package* em inglês |
| FIFO | Primeiro a entrar, primeiro a sair, de *First In, First Out* em inglês |
| EOF | Fim do arquivo, de *End of File* em inglês |
| LSB | Bit menos significativo, de *Least Significant Bit* em inglês |
| MSB | Bit mais significativo, de *Most Significant Bit* em inglês |

# LISTA DE SÍMBOLOS

| | |
|---|---|
| $U$ | Valor horizontal para o mapeamento de textura |
| $V$ | Valor vertical para o mapeamento de textura |
| $L$ | Coeficiente de iluminação |
| $\vec{P}$ | Define o vetor posição tridimensional de um objeto, vértice, ou fragmento. |
| $R$ | Intensidade da cor vermelha do elemento. |
| $G$ | Intensidade da cor verde do elemento. |
| $B$ | Intensidade da cor azul do elemento. |
| $\boldsymbol{q}$ | Quaternião. |
| $\theta$ | Angulo de rotação. |
| $\varphi$ | Angulo de campo de visão. |

## SUMÁRIO


**1 INTRODUÇÃO . . . . . 17**
1.1 Motivação . . . . . 20
1.2 Objetivos . . . . . 21
1.2.1 Objetivos gerais . . . . . 21
1.2.2 Objetivos específicos . . . . . 21
1.3 Organização do trabalho . . . . . 22

**2 FUNDAMENTAÇÃO TEÓRICA . . . . . 23**
2.1 Arquitetura RISC-V . . . . . 23
2.1.1 Registradores . . . . . 28
2.1.2 Instruções . . . . . 30
2.1.3 Linguagem assembly . . . . . 31
2.1.4 Comparação com outras arquiteturas . . . . . 37
2.2 Renderização gráfica . . . . . 39
2.2.1 Vertexes . . . . . 41
2.2.1.1 Posição . . . . . 42
2.2.1.2 Mapa UV . . . . . 43
2.2.1.3 Normal . . . . . 44
2.2.2 Matrizes de transformação . . . . . 46
2.2.2.1 Matriz do modelo . . . . . 46
2.2.2.1.1 *Escala* . . . . . 47
2.2.2.1.2 *Rotação* . . . . . 47
2.2.2.1.3 *Translação* . . . . . 48
2.2.2.2 Matriz de visualização . . . . . 49
2.2.2.3 Matriz de projeção . . . . . 49
2.2.2.3.1 *Field of view* . . . . . 49
2.2.2.3.2 *Distância de corte* . . . . . 50
2.2.2.3.3 *Cone de projeção* . . . . . 51
2.2.3 Rasterização . . . . . 52
2.2.3.1 Algoritmo de Bresenham . . . . . 52
2.2.3.2 Algoritmo scan line . . . . . 55
2.3 Segmentação de memória . . . . . 56
2.3.1 Ordenação de memória fraca RVWMO . . . . . 60

**3 METODOLOGIA . . . . . 63**
3.1 Hardware . . . . . 63

3.2 Microcontrolador K210 . . . 64
3.2.1 Periféricos . . . 64
3.3 Protocolos de comunicação . . . 65
3.3.1 SPI . . . 66
3.3.1.1 Display TFT . . . 67
3.3.1.2 Protocolo SD . . . 69
3.4 Armazenamento de dados . . . 74
3.4.1 Sistema de arquivos FAT . . . 75
3.5 Arquivos externos . . . 82
3.5.1 Arquivo para modelos 3D .obj . . . 83
3.5.2 Arquivo para texturas .ppm . . . 83

**4 DESENVOLVIMENTO . . . 86**
4.1 Configurações de inicialização . . . 86
4.1.1 Entry point . . . 87
4.1.2 Board support package . . . 88
4.1.3 Registradores de system control . . . 90
4.2 Configurações de dispositivos externos . . . 90
4.2.1 Display TFT . . . 91
4.2.2 Cartão SD . . . 92
4.3 Comunicação . . . 93
4.3.1 Comunicação com o display . . . 93
4.3.2 Decodificação do cartão SD . . . 96
4.3.2.1 Verificação CRC . . . 97
4.4 Decodificação da FAT . . . 98
4.5 Leitura de arquivos . . . 98
4.5.1 Arquivos .obj . . . 99
4.5.2 Arquivos .ppm . . . 102
4.6 Renderização do modelo . . . 104
4.6.1 Matrizes de transformação . . . 105
4.6.2 Rasterização de polígonos . . . 107
4.6.3 Loop de renderização . . . 109

**5 ANÁLISE E DISCUSSÃO DOS RESULTADOS . . . 111**

**6 CONCLUSÃO . . . 118**
6.1 Aprimoramentos futuros . . . 118
6.2 Considerações finais . . . 119

**REFERÊNCIAS . . . 120**

# 1 INTRODUÇÃO

A evolução dos meios de interação entre seres humanos e computadores vem evoluindo drasticamente nas últimas décadas, e a utilização de telas se tornou algo natural e corriqueiro da nossa sociedade. A aplicação de renderização 3D se estende a diversas áreas de atuação, desde o entretenimento até aplicações médicas (TIEDE et al., 1990). A nossa necessidade, como indivíduos que vivem em um mundo tridimensional, de representar o universo nas nossas mãos e olhos é antiga. Esculturas em barro e argila precedem a civilização social do modo que conhecemos (GOREN; SEGAL, 1995). Pinturas providenciam uma facilidade para converter nossas experiencias em objetos. Porém, com o surgimento da fotografia no século 18, nossa capacidade de representar o mundo tendera a ser confinada em um plano bidimensional, pelo menos durante esta época.

A capacidade de câmeras de atrelar um momento a um objeto era inequivocamente a forma mais fácil de representar o mundo. Pinturas podem ser feitas de forma rápida, dependendo do artista, mas não possuem a mesma facilidade do simples clique de um botão, mesmo que necessite de um tempo em salas escuras para revelação de uma foto. E esculturas, apesar de conseguirem um realismo sinistro à realidade, como as belas obras de Michelangelo, levam anos para atingirem o realismo, e, até poucas décadas atrás com a maquinização das industrias, eram incapaz de saírem das poucas unidades que o artista fizera.

Entrando no século 19, com os irmãos Lumière na França, a cinematografia nasce e prospera, cimentando nosso mundo a segunda dimensão, mas agora em movimento. A arte do tridimensional é voltada para o maquinário, sendo que nesta época a revolução industrial prospera em diversos países ao redor do mundo. Buscando a padronização de produtos e de máquinas, as peças utilizadas necessitavam da tridimensionalidade, com engenheiros prezando nos detalhes de cada uma de suas três dimensões. Com essas revoluções o mundo se conforma, 2D representa as nossas experiencias, a arte, e a fantasia, e 3D os motores, máquinas, e a nossa realidade. O foco em desenhos tridimensionais foi definido pelo matemático francês Monge (1798), que formalizou o desenho técnico, amplamente utilizado por engenheiros.

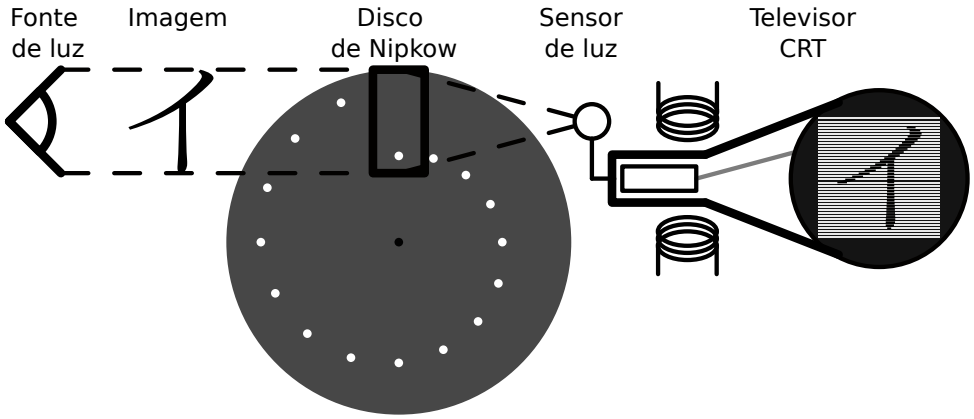
E em meados do século 19, uma nova tecnologia desenvolvida em Bonn na Alemanha revolucionaria por séculos nossa forma de representar imagens. O físico alemão Plücker (1858), desenvolveu o denominado tubo de Geißler, um tubo de vidro, em que seu ar foi removido, e acrescentado pequenas quantidades de gases específicos, com dois eletrodos em suas extremidades. Geißler foi um físico, proficiente em trabalhos com vidro, que desenvolveu a bomba de vácuo utilizada por Plücker, além de outros instrumentos, devido a isto lhe foi atribuído nome ao tubo. Com uma alta tensão aplicada em seus terminais, os elétrons atravessam o tubo de Geißler ionizando o gás interno, e produzindo fótons ao saltar pelo material. A coloração varia dependendo do elemento presente, pela variação da energia na camada de valência destes materiais. A baixa pressão interna é necessária para diminuir a colisão dos elétrons, e permitir

passagem pelo tubo.

Após os experimentos de Plücker, o físico britânico Crookes (1879) comprovou a influência de campos magnéticos nos raios, fazendo-os se curvarem dentro do tubo, permitindo direcionar o raio de elétrons. E com esta tecnologia, o físico alemão Braun (1909), desenvolveu o primeiro protótipo de um tubo de raios catódicos, ou *Cathode Ray Tube* (CRT) em inglês, utilizando o mesmo principio do tubo de Geißler, mas acrescentando uma tela coberta de fósforo ao final do raio, e eletroímãs ao seu redor. Braun também sugeriu a utilização do dispositivo como tela, feito possibilitado posteriormente por sua descoberta.

A partir desta invenção o mundo mudaria, com a capacidade de criar imagens em tempo real, desafiando qualquer outro meio de arte, ou tecnologia. Porém seu uso era inicialmente apenas vetorial, ou seja, as formas eram compostas somente de linhas, e esta tecnologia prevaleceu em uso por décadas, em osciloscópios, e em computadores durante a década de 70. Contudo, o inventor polonês Nipkow (1885) desenvolvia uma nova forma de gravar imagens digitalmente em tempo real, utilizando um disco com diversos furos formando uma espiral, e com uma lente direcionando a imagem, denominado disco de Nipkow. A medida que este disco roda, a intensidade da luz que atravessa os furos varia, dependendo do objeto a sua frente, funcionando como uma câmera digital, sendo a imagem rasterizada ponto a ponto. O engenheiro 高柳健次郎 (1928), lê-se Takayanagi Kenjirō em rōmaji, utilizou esta tecnologia para representar imagens com um televisor CRT, forçando o raio a andar em 40 linhas horizontais paralelas equidistantes, com intensidade dependendo da informação amostrada por um disco de Nipkow, e com isso se cria a rasterização de imagens em televisores CRT, e a primeira transmissão de uma imagem real para um televisor eletrônico. Uma ilustração do experimento de Takayanagi está presente na Figura 1.

Figura 1 – Rasterização com um disco de Nipkow em um televisor CRT




Fonte: Ilustração do trabalho de Takayanagi (高柳健次郎, 1928)

Com isto, se observa o começo da tecnologia de rasterização de imagens em telas, que evoluíram de CRTs preto e branco para incluir variação de cores e outros meios de reprodução, como displays de cristal líquido, *Liquid Crystal Display* (LCD) em inglês. Estas telas, que

utilizamos com frequência nos dias de hoje, nos permitem observar o mundo, porém, confinados ainda a sua bidimensionalidade. Mas com novos desenvolvimentos em software e hardware durante o século 20, computadores passaram a possuir capacidade matemática suficiente para representar modelos tridimensionais em uma tela bidimensional, realizando o complexo processo de transformação de coordenadas para algo possível de reproduzir em 2D.

Um dos primeiros casos de utilização tridimensional em computadores é o do programa *Sketchpad* desenvolvido pelo cientista da computação estadunidense Sutherland (1964), e o que lhe trouxe o prêmio Turing em 1988. *Sketchpad* providenciava uma interface capaz de ser interagida pelo usuário por uma caneta de luz, e possuía a capacidade de criar formas bidimensionais. Porém, apenas era possível utilizar pontos, linhas, e semicírculos, o que se tornaria ideal para peças mecânicas sem preenchimento interno, e diagramas científicos, mas tinha menor interesse no meio artístico. O software também permitia relações entre formas, e com isto, aplicando transformações matemáticas internamente, foi possível reproduzir formas tridimensionais em tempo real em uma tela bidimensional.

A partir deste ponto, renderização 3D passaria a ter grande importância em computação. Engenheiros utilizariam esta ferramenta para produção de peças e diagramas, com a popularização do que é denominado de design auxiliado por computador, ou *Computer-Aided Design* (CAD) em inglês. Com a capacidade de preenchimento, no salto de vetores para pixels, artistas começaram a utilizar esta ferramenta para aprimorar obras cinematográficas, chegando a grandes empresas, como Pixar Animation Studios, contribuírem diretamente para o ramo da computação com suas técnicas de renderização (APODACA; MANTLE, 1990). E hoje, com jogos 3D iterativos, sentimos como este meio de arte fosse natural, já que é deste modo que vemos o mundo, porém, não podemos nos esquecer das complexas operações que são realizadas por dentro dos chips de silício para simular esta naturalidade.

Devido a complexidade atribuída a este tipo de renderização, dispositivos específicos existem nos computadores e celulares modernos, chamados de unidades de processamento gráfico, ou *Graphics Processing Unit* (GPU) em inglês. Estes dispositivos possuem arquiteturas diferentes do que as unidades centrais de processamento, ou *Central Processing Unit* (CPU) em inglês, estão acostumadas. GPUs são construídas com renderização em mente, possuindo diversos núcleos voltados para cálculos matemáticos, o que é necessário quando precisamos converter milhões de polígonos em posições tridimensionais para uma tela plana. Para redução de custo e de área de utilização do silício, diversas instruções presentes em uma CPU são descartadas para a GPU.

Para CPUs de computadores de mesa se utiliza, geralmente, instruções x86, desenvolvidas originalmente para o microprocessador 8086 da Intel, ou ARM para dispositivos móveis, porém estas possuem muitas instruções, que seriam inutilizadas em uma utilização eficiente de uma GPU. Atualmente não a um padrão interno nas GPUs como nas CPUs. Contudo novas arquiteturas podem ser utilizadas neste caso, como o conjunto de instruções aberto RISC-V, desenvolvido na universidade de Berkeley por Waterman et al. (2011). Baseado na ideia de

simplificação e modularidade, RISC-V possui a capacidade de ser utilizado em diversas áreas inacessíveis para arquiteturas complexas como x86 e até arquiteturas ARM.

Essa vontade humana de representar o mundo tridimensional em objetos tangíveis, e de poder guardar experiências no tempo, nos trouxe complexos dispositivos visuais capazes de recriar mundos em tempo real. Buscamos sempre o que nos é semelhante, a linguagem binária de computadores esta longe de nos ser legível, porém, devido ao avanço da tecnologia e criatividade humana estes dispositivos conseguem nos mostrar um mundo similar ao nosso. Para continuar a evolução da interação entre computadores e humanos, e melhorar nossa capacidade de criar mundos precisamos aprimorar esta tecnologia.

## 1.1 Motivação

A utilização prática da renderização 3D pode ser observada no dia a dia. De filmes, jogos, a propagandas, basicamente toda forma de entretenimento visual possui utilização de modelagem 3D. Na área de engenharia, modelagem de peças, e diagramas são essenciais para todo o setor. Em pesquisas cientificas, modelos 3D de moléculas, organismos, planetas, e outras estruturas facilitam o estudo dos profissionais. Em medicina, representações 3D de órgãos e estruturas ósseas são utilizadas desde seus primórdios, e máquinas de ressonância magnética também apresentam amplo uso.

As GPUs necessitam de processadores de alta densidade de transistores, contudo, devido a uma complexa *supply-chain* distribuída ao redor do globo, estes tipos de dispositivos são frágeis do ponto de vista de distribuição, como foi demonstrado pela perturbação deste mercado causado durante a pandemia de SARS-CoV-2. Acabando por causar escassez de GPUs e outros dispositivos em todo o mundo devido a uma imprevisão de demanda (KING; WU; POGKAS, 2021).

A complexidade de uma GPU a volta para dispositivos mais completos, como celulares, computadores pessoais, e consoles. Porém, nem sempre a GPU opera em seu limite, muitas vezes o potencial deste dispositivo não é utilizado, mas sem outra opção de baixo custo, em certos casos, acabam-se por utilizar equipamentos superdimensionados. Contudo, fora do mercado das GPUs dedicadas existe também uma necessidade de renderização 3D, como podemos ver pelo mercado de realidade aumentada, que possui uma taxa de crescimento anual composta de 35%, e em muitos casos precisa apenas de uma renderização de poucos modelos simples, para incrementar alguma atividade do usuário (TECHNAVIO, 2021).

Além disto, muitos dispositivos podem ter sua usabilidade aprimorada com uma visualização 3D em tempo real, como máquinas de Controle Numérico Computadorizado (CNC), e impressoras 3Ds. Máquinas que trabalham com objetos tridimensionais podem ser amplamente aprimoradas com uma Interface Homem-Máquina (IHM) que apresente capacidade de renderização 3D. E a renderização 3D de baixo consumo também podem ser utilizadas em celulares e notebooks, para reduzir o consumo elétrico do dispositivo, visto que muitas atividades do dia a dia não possuem a necessidade de renderizar modelos complexos.

Certas pesquisas mostram resultados nesta área de atuação, como o exemplo de Tine et al. (2021), onde é desenvolvido uma extensão para a arquitetura do RISC-V de modo a desenvolver uma microarquitetura *open-source* base para placas de vídeo em geral, com suporte para linguagens de alto nível via OpenCL e *shaders* para OpenGL. Outro exemplo neste contexto é o trabalho desenvolvido por Zhou, Jin e Xiang (2020), onde a extensão para o RISC-V é voltada para renderização em baixo consumo energético, porém neste caso, com uma implementação voltada para chips de inteligência artificial.

Ambos os trabalhos prévios, contudo, atuam com a ampliação do conjunto de instruções, ou seja, há modificação direta do hardware, o que apresenta certas complexidades de implementação. Por conta disto, este trabalho busca tratar do tema apenas pelas ferramentas providas nativamente pelo RISC-V, porém, possibilitando assim, uma análise sobre possíveis modificações na arquitetura com base na estrutura do algoritmo.

## 1.2 Objetivos

Esta seção é segmentada em duas partes, os objetivos gerais, que apresentam uma visão geral dos pontos a serem traçados, e específicos, que detalham mais profundamente os caminhos a serem tomados.

### 1.2.1 Objetivos gerais

Neste trabalho é proposto a implementação de renderização 3D de modelos simples, em um dispositivo de baixo consumo energético, assim compactando esta ferramenta para conjunto de algoritmos mínimos. Deste modo, podendo atingir certos setores do mercado em que a renderização 3D é buscada de forma simples, com baixo consumo energético e custo de implementação.

O algoritmo será realizado com base na linguagem RISC-V assembly, utilizando assim o seu conjunto de instruções respectivo. Como o RISC-V possui instruções simples e compactas, os algoritmos de renderização são adaptados para estes limites. Esta simplificação sendo realizada na base da tecnologia de renderização, desconstruindo a implementação de interfaces de programações em seus componentes base de renderização.

### 1.2.2 Objetivos específicos

Será utilizado um microcontrolador com arquitetura RISC-V embutida internamente, de modo a buscar o baixo custo e consumo energético proposto. Busca-se também uma forma de comunicação exterior para permitir a disposição da renderização a uma tela. Com o objetivo de tornar o algoritmo mais genérico, será também utilizado uma forma de trabalhar com formatos de arquivos editáveis em computador, para que se possa facilmente alterar os modelos renderizados, sem serem atrelados ao código.

A renderização buscada não necessita ser de alta qualidade, pois este não é um ponto buscado. Contudo, a capacidade de identificação do modelo na tela é necessária. Em contraponto, a taxa de renderização necessita também ser confortável o suficiente para a visão humana.

Os resultados finais sendo dados pela comprovação da utilidade da construção de algoritmos de renderização 3D pelas ferramentas diretas do RISC-V, de modo que esta tecnologia seja implementada em dispositivos de baixo consumo e processamento como microcontroladores.

## 1.3 Organização do trabalho

Este trabalho foi segmentado em quatro capítulos principais, com a introdução de modo a providenciar uma entrada ao texto, e a conclusão oferecendo uma breve resolução dos conteúdos tratados.

O corpo do texto inicia-se com a fundamentação teórica, onde será discutido a progressão histórica das tecnologias utilizadas, e uma explicação detalhada sobre os conceitos tratados, em conjunto das componentes teóricas do trabalho a ser desenvolvido, incluindo os equacionamento, pseudo algoritmos, e entre outros assuntos. É seguido uma sequência dos conteúdos mais abrangentes para os mais específicos. Iniciando com explicação da arquitetura RISC-V, seguido das equações e algoritmos da renderização e rasterização.

Seguindo é apresentado as metologias utilizadas para o trabalho. Neste ponto são apresentados diversas tecnologias e métodos que são necessários para o compreendimento do desenvolvimento. Sua progressão é realizada a partir dos meios físicos utilizados, descrito pelo hardware e dispositivos externos, seguindo para as informações de protocolo, finalizando com a descrição da organização interna da segmentação de arquivos.

O desenvolvimento prossegue, apresentando um detalhamento das etapas da construção do trabalho físico final. Nesta é apresentada uma progressão do hardware, até as etapas puramente matemáticas, representadas pela renderização e rasterização do modelo. Nesta etapa o desenvolvimento do algoritmo é abordado de forma mais gráfica, evitando-se assim uma abordagem muito detalhista de difícil compreensão.

A análise e discussão dos resultados finaliza o corpo do texto, dispondo de forma visual os objetivos alcançados. São realizadas análises qualitativas do resultado obtido, para a verificação do cumprimento dos objetivos propostos. Além disto, é feito uma análise qualitativa breve, verificando a satisfatoriedade do trabalho finalizado.

A introdução apresenta breves textos para situar o leitor no trabalho, providenciando uma forma de se imergir no contexto do trabalho. A conclusão apresenta um texto de finalização, com pontos de melhoria, e uma visão geral breve dos resultados. Apêndices apresentam complementações ao texto que são demasiadamente grandes, ou seguem tangentes ao texto. Em anexos, são apresentados documentos de terceiros que, com os devidos créditos fornecidos, serão utilizados para aprofundar o contexto do texto, e possivelmente resolver questões que podem ocorrer durante a leitura deste trabalho.

# 2 FUNDAMENTAÇÃO TEÓRICA

Com a necessidade humana de representar o mundo, chegamos à atualidade, com vasta utilização de renderização em dispositivos cada vez mais complexos. Com isto, este trabalho se insere, propondo uma nova forma de lidar com as tecnologias existentes.

O desenvolvimento deste projeto requer um conhecimento prévio de certas áreas. Atualmente não existe uma vasta literatura envolvendo estes temas de forma agrupada, por conta disto é necessário uma discussão dos temas de forma individual. Todas as seções são precedidas de um contexto histórico, para avaliação da evolução da tecnologia e do estado da arte. A descrição não será vasta, porém, aprofundada naqueles tópicos que forem essenciais para o projeto. A explicação detalhada providenciará o conhecimento necessário para o desenvolvimento do projeto.

Devido a independência das seções, a ordem foi escolhida com base na abrangência e importância no projeto. Caso temas possuam relação contextual, o texto segue de forma linear, se aprofundando nas nuanças a medida que este se progride.

Um dos assuntos centros do trabalho é a arquitetura RISC-V. É explicado o seu conceito de forma geral, assim como o detalhamento das instruções e registradores. Seguindo se explica sobre as operações de renderização gráfica, como o equacionamento matemático das transformações, e as etapas de rasterização. Após isto, é finalizado com uma análise sobre segmentação de memória, com as considerações de manuseio realizadas no contexto do RISC-V.

## 2.1 Arquitetura RISC-V

Um dos itens que define um computador é a sua arquitetura, porque é a responsável pela sua estrutura de operação. Um computador é um dispositivo lógico, que se resume em realizar tarefas dependendo de certos comandos, sendo estes, definidos em sua arquitetura. Hoje em dia, dado a linguagens de programação compiláveis entre plataformas, a arquitetura é geralmente desconsiderada nos trabalhos realizado, contudo, era item fundamental a poucas décadas atrás. A evolução da computação é recente, tendo apenas destaque após o meio do século 20, sendo sua origem datada do início do século 19, com os teares de Jacquard e sua utilização de cartão perfurados.

Computadores são máquinas utilizadas para cálculos matemáticos e lógicos, e manuseio de informação, e surgiu desta necessidade. O estadunidense Hollerith (1894) desenvolveu o primeiro organizador de dados digitais para o censo dos Estados Unidos. Utilizando a ideia dos cartões perfurados do tear de Jacquard, relés eletromecânicos, e contadores mecânicos o usuário inseria os cartões perfurados na máquina, e, ao acionar o dispositivo, seria acrescentado em um cada item das categorias em que o indivíduo representado no cartão se enquadrava. O cartão era dividido em regiões, e cada uma representava uma categoria. A lógica de cada

item era booleana, ou seja, cada ponto somente podia representar dois valores, verdadeiro ou falso. Neste caso o *clock* do sistema era definido pelo usuário, porque ele acionava a máquina, e o único comando possível era a soma. Apesar de sua operacionalidade limitada comparada com os dias de hoje, foi o suficiente para reduzir o tempo de contagem do censo estadunidense drasticamente.

Nesta época os computadores avançaram para realizar outras operações matemáticas, porém, sempre necessitavam de ligação de contatos manualmente para seleção de operação, ou eram construídos para apenas um propósito. Contudo, em 1941, o engenheiro alemão Konrad Zuse desenvolveu o primeiro computador eletrônico programável, denominado Z3. Este pode ser dito como o primeiro computador de uso geral desenvolvido. Este computador possuía uma arquitetura similar a de von Neumann, com uma unidade central de processamento, isolada da memória, e isolada dos dispositivos de entrada e saída. O Z3 utilizava lógica binária, que atualmente nos parece a norma, mas na época era a exceção com diversos outros computadores adotando a lógica decimal em seus sistemas. E além disto, apresentava operações com números flutuantes, sendo o primeiro e único da época a utilizá-los. Os números flutuantes eram armazenados em um número de 22 bits, sendo destes, 1 para o sinal, 7 para o expoente, e 14 para a fração, sendo a fração sempre armazenada na forma normalizada, ou seja, com o primeiro digito antes da vírgula sendo sempre 1, podendo este ser omitido. Esta forma de armazenamento de números flutuantes é a mesma utilizada atualmente na maioria dos processadores, seguindo as normas IEEE 754, apenas com quantidades diferentes de bits (ROJAS, 1997).

O que definia a máquina Z3 especial eram seus códigos de operação, que permitiam o computador de realizar operações pré definidas em uma fita perfurada sem intervenção do usuário. As nove possíveis operações eram de comunicação com o teclado, mostrar resultado em um *array* de lâmpadas, armazenamento e carregamento de memória, e cinco operações matemáticas, contudo, este computador não possuía operações condicionais, essenciais para a operação de computadores modernos. As operações matemáticas possíveis eram multiplicação, divisão, raiz quadrada, adição, e divisão. Cada uma destas instruções devia estar presente na fita perfurada para ser realizada pelo computador. As operações aritméticas são sempre realizadas entre os dois únicos registradores, com o resultado armazenado sempre no primeiro. Carregamentos da memória operam sempre no segundo registrador, com exceção do primeiro carregamento. O funcionamento básico é similar aos computares modernos, apenas variando a quantidade de registradores e operações.

Zuse desenvolveu um dispositivo capaz de facilitar cálculos matemáticos, permitindo a programação automática de operações aritméticas. Isto se manteve como norma seguindo adiante. Atualmente é extremamente necessário essa auto programação dos dispositivos. Contudo, com a modernização dos computadores programáveis, o desenvolvimento destes códigos para operação evoluiu-se em seu próprio ramo.

Com o avanço da eletrônica, e a redução dos circuitos, os computadores puderam ser reduzidos gradativamente, até chegarem nos complexos dispositivos que se encontram

em basicamente todos os produtos eletrônicos da atualidade. Muitas arquiteturas específicas surgiram e foram esquecidas ao tempo, contudo, com a rápida comercialização de computadores na década de 70, uma certa necessidade de padronização nos dispositivos se tornou aparente. Como os códigos de operação e registradores dependiam da arquitetura do dispositivo, os programas somente rodariam nos dispositivos para os quais os programadores os compilavam. Isto acabou limitando o alcance dos programadores, e prendia usuários a certas empresas que pudessem dar suporte para seus respectivos computadores. Essa forma de operação não era comercialmente viável, e pouco desejável. Com isto, a Intel desenvolveu o 8086, que possuía certa retrocompatibilidade com outros processadores anteriores da mesma linha. O dispositivo apresentava arquitetura Von Neumann, e foi utilizado em diversos produtos na época devido a sua versatilidade (INTEL, 1979). Esta arquitetura acabaria se tornando o padrão no ramo, se denominando x86 devido a sua origem, e mantendo sempre a retrocompatibilidade em mente. As instruções eram de 16 bits de comprimento originalmente, expandindo para 32 bits, e para os subsequentes 64 bits seguindo a evolução dos equipamentos, sendo comumente denominado x86-64 para diferenciar-se de versões anteriores.

Até um período recente, as arquiteturas dos computadores tenderam para um conjunto de instruções complexas, denominados *Complex Instruction Set Computer* (CISC) em inglês, com o x86 fazendo parte deste conjunto. Esta tendência foi possível graças ao aumento da frequência de processamento e redução do tamanho do transistor, o que acabou permitindo que os fabricantes acrescentassem um maior número de instruções no dispositivo para reduzir o tamanho dos programas na memória, e permitir uma mais rápida execução. Sistemas CISCs apresentavam uma vasta gama de opções para os compiladores, porém isto se refletia na quantidade de transistores utilizados para armazenar todas as possíveis instruções, e estes sistemas somente tiveram um crescimento viável enquanto a microeletrônica acompanhava essa necessidade. Esta evolução dos dispositivos foi ditada por Moore (1965), e se manteve próxima a realidade até recentemente, onde a redução no tamanho dos transistores começa a apresentar problemas na operação da CPU a medida que se aproxima da escala atômica. Para manter a evolução dos CISCs, com o limite da microeletrônica, o tamanho dos processadores aumentou, criando chips de alto consumo elétrico, devido a grande quantidade de transistores e área de silício necessária. Com o rápido crescimento de dispositivos móveis, este consumo se apresentava inviável. E com o surgimento de processamento paralelo, múltiplos processos simples podiam ser executados em conjunto sem travar o dispositivo. A partir destas necessidades uma alternativa dos dispositivos CISCs obteve grande espaço no mercado, denominado de computador de conjunto de instruções reduzidas, *Reduced Instruction Set Computer* (RISC) em inglês.

Devido a complexidade, muitas das instruções de um CISC não são utilizadas, como mostra Akshintala et al. (2019), de 853 mnemônicos no conjunto de instruções, 12 destes são utilizados em 89% dos casos, sendo somente o mnemônico MOV, utilizado para movimentar valores entre registradores, compondo 37,8% das instruções de um programa em média. O RISC foi desenvolvido por Patterson e Sequin (1982), com o objetivo de reduzir a complexidade

das arquiteturas. A ideia original do RISC era de rodar todas as instruções em um ciclo de *clock*, manter todas as instruções de mesmo tamanho, remover operações entre dados na memória e mantê-las apenas entre registradores, e permitir integração a compiladores para linguagens de alto nível. RISCs se fixaram no mercado a partir da família de arquiteturas ARM, que implementou as técnicas de design RISC e expandiu-as para um dispositivo comercialmente viável. O primeiro ARM foi desenvolvido pela cientista da computação Sophie Wilson e Steve Furber e apresentava uma arquitetura de 32 bits, com capacidade de operar três milhões de instruções por segundo, com um consumo típico de 0.1 W devido a simplicidade de sua arquitetura (ACORN, 1986). Contudo, o mercado de arquiteturas é extremamente resistente a alterações, porque novas arquiteturas não possuem vastas quantidades de softwares compatíveis como o x86, e para os programadores verem viabilidade em portarem seus programas para estes novos sistemas é necessário uma grande quantidade de possíveis usuários. Por conta disto, o ARM somente teve presença no mercado com a revolução dos *smartphones*, que providenciou uma nova chance de inovação até o estabelecimento de um novo status quo. Hoje em dia os sistemas ARM predominam no mercado de *smartphones*, microcontroladores, e outros sistemas embarcados, com uma gradativa adoção no mercado de notebooks, porém o mercado de *desktops* é definido ainda pela arquitetura x86.

Apesar do ARM adotar a metodologia RISC, e representar uma melhora da arquitetura x86, ainda apresenta um vasto conjunto de instruções de arquitetura, *Instruction Set Architecture* (ISA) em inglês. Em seu conjunto base do ARMv6 de 32 bits existem cerca de 146 instruções, variando em uma certa margem para as outras versões. Com adicionais 72 instruções Thumb, que são um subconjunto das instruções ARM utilizando 16 bits, reduzindo espaço na memória. Cada instrução necessita de seu próprio código de operação, *operational code* (opcode) em inglês, então muitas instruções podem reduzir o campo para utilização de dados. Estes valores podem acrescentar caso necessite de extensões, como pontos flutuantes e instruções vetoriais (ARM, 2005).

Desenvolvido por Waterman et al. (2011), com o auxilio do professor da Universidade de Berkeley e desenvolvedor do RISC, David Patterson, foi desenvolvida a quinta versão do conjunto de instruções reduzido denominado RISC-V. Inicialmente o foco era acadêmico, então a ISA foi desenvolvida com um conjunto simples de instruções, apenas para o ensino de arquiteturas. Após interesse externo, desenvolvedores do RISC-V tornaram a tecnologia em um produto viável para o mercado, criando a RISC-V Foundation, responsável por manter a arquitetura e divulgá-la para o público.

Um dos aspectos principais do RISC-V é o fato de ser *open source*, ou seja, o seu uso é livre e disponibilizado em sua totalidade. Esta forma de desenvolvimento *open source* apresenta certas diferenças da de softwares, devido a impossibilidade de atualizar o hardware de um chip já fabricado. Outra característica é a sua modularidade. As instruções são divididas em extensões acrescentadas a um conjunto base, como presente na Quadro 1, sendo cada uma destas para um propósito específico. Cada extensão é desenvolvida separadamente, sendo sujeita a *peer*

*review* pública. Isto permite que cada conjunto de instruções seja o mais eficiente possível, levando anos até ser propriamente finalizado. Cada chip utiliza o seu próprio conjunto de instruções, anexando as letras em sequência. Para manter retrocompatibilidade e comunicação entre dispositivos cada extensão é congelada no tempo após definido pela comunidade como pronto. Após a ratificação poucas mudanças que não alterem o hardware ainda são possíveis. Existem *opcodes* não reservados para extensões oficiais, podendo ser utilizados de forma livre por cada desenvolvedor, apenas tendo em mente que outros dispositivos também são livres para interpretar estes códigos em suas próprias instruções.

Quadro 1 – Instruções bases e algumas extensões da arquitetura RISC-V

| **Conjunto de instruções bases** | | |
|---|---|---|
| **Base** | **Status** | **Descrição** |
| RV32I | Ratificado | Conjunto base de instruções de inteiros de 32 bits. |
| RV64I | Ratificado | Conjunto base de instruções de inteiros de 64 bits. |
| RV32E | Rascunho | Conjunto base de instruções para sistemas embarcados. (Similar ao RV32I, porém possui apenas 16 registradores) |
| RV128I | Rascunho | Conjunto base de instruções de inteiros de 128 bits. |
| **Extensões** | | |
| **Código** | **Status** | **Descrição** |
| M | Ratificado | Instruções de multiplicação e divisão de inteiros. |
| A | Ratificado | Instruções atômicas para sincronização e operação entre *threads*. |
| F | Ratificado | Registradores e instruções de pontos flutuantes de precisão única em norma com a IEEE 754-2008. |
| D | Ratificado | Registradores e instruções de pontos flutuantes de precisão dupla em norma com a IEEE 754-2008. |
| Q | Ratificado | Registradores e instruções de pontos flutuantes de precisão quádrupla em norma com a IEEE 754-20008. |
| Zfh | Rascunho | Registradores e instruções de pontos flutuantes de meia precisão em norma com a IEEE 754-2008. |
| C | Ratificado | Instruções compactas de 16 bits. |
| V | Rascunho | Registradores e instruções para operações de dados simultaneamente em paralelo. |
| Zicsr | Ratificado | Instruções e registradores de controle e estado. |
| Zifence | Ratificado | Instruções de sincronização entre acessos à memória. |
| G | - | Utilizado de abreviação para o conjunto *IMAFDZicsr_Zifencei*. |

Fonte: (WATERMAN; ASANOVIĆ, 2021)

Certas extensões necessitam que outras estejam presentes, como o *F* que necessita do *Zicsr*, e *D* que necessita do *F*. O código *G* não é uma extensão por si só, mas representa um conjunto de extensões geralmente utilizado em conjunto para programação de propósito geral, o código é uma abreviação do inglês *general-purpose*. Além disto, o RISC-V apresenta divisão de privilégios, com acesso a registradores e instruções dependendo do nível em que o software rodará. Os níveis são divididos, em ordem crescente de privilégio, *User*, *Supervisor*, *Hypervisor* e *Machine*. O uso dos quatro níveis não é obrigatório, porém, modo *Machine* sempre estará

presente. O modo *Hypervisor* é voltado para a virtualização de máquinas, para providenciar um nível de isolação entre os sistemas operacionais e a máquina (WATERMAN; ASANOVIĆ, 2019).

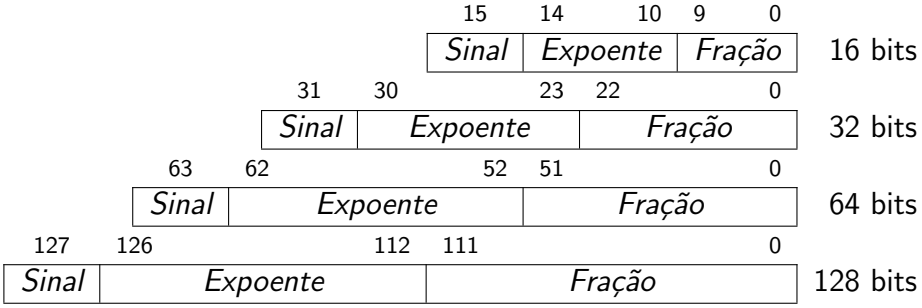
### 2.1.1 Registradores

Desde o computador Z3 de Zuse os registradores vem sendo utilizado como pequenas memórias em que o trabalho é realizado. Devido ao fato de arquiteturas CISC realizarem operações entre endereços da memória, sua quantidade de registradores é baixa. Sistemas RISCs utilizam geralmente vários registradores, isto evita acesso intensivo à memória, que geralmente leva vários ciclos de *clock* para ser completado.

Os conjuntos bases do RISC-V geralmente apresentam 32 registradores de uso geral, com exceção do *RV32E* que possui 16. O tamanho dos registradores depende da base utilizada, para o *RV32I*, existe 32 registradores de inteiros de 32 bits, 32 de 64 bits para o *RV64I*, e 32 de 128 bits para o *RV128I*. Cada conjunto de registrador é único para cada *hardware thread* (hart) O registradores de inteiros são definidos como `x0` até `x31`. O `x0` possui seus bits fixos a 0, feito desta forma para descartar certas instruções e comprimir a ISA. Os demais registradores podem ser livremente utilizados para qualquer propósito, porém, será visto na Subseção 2.1.3 que estes são segmentados em diferentes grupos definidos pelo compilador. Há ainda o registrador do contador de programa incluído no conjunto base, o (`pc`), do inglês *program counter*. O `pc` é somente acessível por instruções específicas.

Certas extensões acrescentam registradores ao núcleo, como as instruções de pontos flutuantes. A extensão *F* acrescenta 32 registradores de 32 bits de uso geral para operações de pontos flutuantes, a extensão *D* expande estes para 64 bits, e *Q*, para 128 bits. Há ainda a extensão *Zfh*, que permite operação de ponto flutuante de 16 bits, porém depende de *F* e não altera o número de bits dos registradores. Os valores são armazenados como mostrados na Tabela 1, de acordo com a norma IEEE 754. Os registradores são nomeados de `f0` até `f31`. Além disto, para pontos flutuantes é necessário a extensão *Zicsr*, porque é necessário o uso do registrador `fscr`, acessível por instruções específicas.

Tabela 1 – Armazenamento de pontos flutuantes de acordo com a IEEE 754




Fonte: (IEEE, 2008)

A extensão *Zicsr* acrescenta diversos registradores adicionais e instruções para acessá-los. Os registradores de controle e estado, *Control Status Registers* (CSRs) em inglês, são responsáveis por fornecer informações do hardware, e utilizado para interrupções e exceções. As instruções envolvendo estes registradores suportam endereços de 12 bits, ou seja, existem 4096 possíveis CSRs. Contudo, cada endereço é segmentado em 3 partes, os bits 11 e 10 definem as permissões de leitura e escrita ([`0b00`, `0b01`, `0b10`]: *read/write*, [`0b11`]: *read-only*), os bits 9 e 8 definem o privilégio, e os demais bits não possuem definições. Alguns destes CSRs estão presentes na Quadro 2 com suas respectivas descrições. Parte dos CSRs estão pré-definidos na ISA do RISC-V, alguns reservados, porém existem ainda regiões para livre definição do fabricante.

Quadro 2 – Alguns CSRs definidos na ISA do RISC-V

| **User** | | | |
|---|---|---|---|
| **Endereço** | **Permissão** | **Mnemônico** | **Descrição** |
| 0x000 | Read/Write | `ustatus` | Estado do *User* |
| 0x003 | Read/Write | `fscr` | Configurações e exceções de pontos flutuantes |
| 0x004 | Read/Write | `uie` | Registrador de ativação de interrupções |
| 0x005 | Read/Write | `utvec` | Endereço da função de gerenciamento de *trap* |
| 0x042 | Read/Write | `ucause` | Motivo da *trap* do *User* |
| 0x044 | Read/Write | `uip` | Interrupção pendente do *User* |
| 0xC00 | Read-Only | `cycle` | Contador de ciclo |
| 0xC01 | Read-Only | `time` | Contador de tempo |
| **Supervisor** | | | |
| **Endereço** | **Permissão** | **Mnemônico** | **Descrição** |
| 0x100 | Read/Write | `sstatus` | Estado do *Supervisor* |
| 0x105 | Read/Write | `stvec` | Endereço da função de gerenciamento de *trap* |
| 0x142 | Read/Write | `scause` | Motivo da *trap* do *Supervisor* |
| 0x180 | Read/Write | `satp` | Endereço de tradução e proteção da memória |
| **Machine** | | | |
| **Endereço** | **Permissão** | **Mnemônico** | **Descrição** |
| 0x300 | Read/Write | `mstatus` | Estado da *Machine* |
| 0x301 | Read/Write | `misa` | Base e extensões presentes |
| 0x304 | Read/Write | `mie` | Registrador de ativação de interrupções |
| 0x305 | Read/Write | `mtvec` | Endereço da função de gerenciamento de *trap* |
| 0x342 | Read/Write | `mcause` | Motivo da *trap* da *Machine* |
| 0x344 | Read/Write | `mip` | Interrupção pendente da *Machine* |
| 0xF11 | Read-Only | `mvendorid` | ID do fornecedor |
| 0xF14 | Read-Only | `mhartid` | ID do *hart* |

Fonte: (WATERMAN; ASANOVIĆ, 2019)

Quando ocorre alguma exceção ou interrupção o sistema redireciona o código para o gerenciador de *trap* do respectivo respectivo. Cada privilégio pode definir sua própria função no registrador `xtvec`. Quando ocorrida a *trap*, o registrador `xcause` indicará o motivo da divergência, podendo assim o núcleo lidar com a situação.

### 2.1.2 Instruções

A utilização de instruções é o que define um computador programável. As instruções Em sistemas CISCs o objetivo é de providenciar o maior número possível de instruções específicas para o compilador, reduzindo o seu espaço na memória, e visando aumentar a velocidade de operação. Em sistemas RISCs o objetivo é um conjunto reduzido e mínimo de instruções, mantendo todas de mesmo tamanho, e rodá-las cada uma em um único ciclo de *clock*.

As instruções em um RISC operam com 3 componentes, um código indicador da instrução, registradores, e um valor arbitrário denominado de imediato. Apenas o código de operação é item obrigatório. Dos registradores, geralmente são diferenciados entre operandos e resultantes, podendo estes serem os mesmos. Um estudo em detalhe sobre instruções de forma geral pode ser lido em Stallings (2002).

O RISC-V apresenta um conjunto mínimo de instruções possíveis. A base *RV32I* possui apenas 40 instruções, desenvolvida para uma implementação mínima, contudo podendo operar suficientemente por si só. A base *RV64I* adiciona 15 instruções, e *RV128I* adiciona mais 15, para um total de 70 neste caso. As demais extensões apresentam algumas instruções específicas. A extensão *Zicsr* acrescenta 6, a *Zifencei*, 1. A *M* acrescenta 8 para inteiros de 32 bits e mais 5 caso utilize inteiros de 64 bits, a *A* adiciona 11 para 32 bits e mais 11 para 64 bits, a *F* e *D* acrescentam cada uma 26 para inteiros de 32 bits e mais 4 para de 64 bits. Um computador de propósito geral utilizando a ISA *RV32G* possui no total 118 instruções únicas, e *RV64G* possui 159, ambos com capacidade de realizar operação em pontos flutuantes. Além disto, a extensão *C*, que é recomendada para providenciar uma redução do tamanho dos programas na memória, possui 40 instruções (WATERMAN; ASANOVIĆ, 2021).

As instruções do RISC-V são de 32 bits e possuem um campo de *opcode* de 7 bits, com exceção do das instruções compactas que são de 16 bits com 2 bits de *opcode*. Os bits 0 e 1 do *opcode* são destinados para as instruções compactas, em uma instrução comum estes bits são `0b11`, ou seja, existem $2^5$ *opcodes* disponíveis para as demais instruções. Contudo, várias instruções possuem campo de função, que reduz o uso dos bits do *opcode*.

Os campos dos registradores e *opcodes* tendem a ficarem sempre na mesma posição, podendo aplicar otimizações a nível de hardware. As instruções são divididas em 7 formatos diferentes como mostrado na Tabela 2, com exceção das instruções compactas, que apresentam uma divisão diferente.

Os campos de `rsn` representam os registradores de trabalho, que serão utilizados para realizar a operação. O campo de `rd` define o registrador de armazenamento do resultado. Os campos de *funct* são utilizados para definir a função da operação, de modo a permitir várias instruções no mesmo *opcode*. Os campos de *imm* define o imediato, a região onde um valor arbitrário pode ser alocado. Alguns destes campos estão segmentados, porém, isto é para otimização no hardware, para os bits do imediato estarem na mesma posição entre formatos diferentes de instruções.

O formato *B* é utilizado nas instruções de *branching*, e como as instruções do RISC-V

Tabela 2 – Formato das instruções de 32 bits do RISC-V

| 31 27 | 26 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 | **Tipo** |
|---|---|---|---|---|---|---|---|
| *funct7* | | rs2 | rs1 | *funct3* | rd | *opcode* | R |
| rs3 | *funct2* | rs2 | rs1 | *funct3* | rd | *opcode* | R4 |
| imm[11:0] | | | rs1 | *funct3* | rd | *opcode* | I |
| imm[11:5] | | rs2 | rs1 | *funct3* | imm[4:0] | *opcode* | S |
| imm[12\|10:5] | | rs2 | rs1 | *funct3* | imm[4:1\|11] | *opcode* | B |
| imm[31:12] | | | | | rd | *opcode* | U |
| imm[20\|10:1\|11\|19:12] | | | | | rd | *opcode* | J |

Fonte: (WATERMAN; ASANOVIĆ, 2021)

são múltiplos de 2 bytes, o bit menos significativo pode ser desconsiderado, porque todo salto deverá ser para uma instrução inteira. O formato *J* inicia a partir do bit 1 pelo mesmo motivo, porém neste caso é para instruções de *jump*. O formato *U* é para instruções que operam com a parte superior de valores de 32 bits, por isto o imediato está definido de 12 a 31.

Uma das extensões mais importantes é a de instruções compactas *C*. Como já visto anteriormente, boa parte dos programas são compostos por poucas instruções únicas. Dependendo dos registradores e imediatos utilizados, as instruções podem ser reduzidas para seu equivalente de 16 bits, permitindo uma redução da memória utilizada pelo programa. Esta redução de tamanho é possível para instruções que operam com imediatos pequenos, registradores de operação igual ao de resultado, e operações com zero. Certas operações com zero somente são permitidas por instruções compactas, para evitar confusões com a instrução *no operation* (`nop`).

Contudo, para estas instruções é necessário uma nova forma de se arranjar os valores, e com o menor espaço, acabam sendo necessários mais formatos de instruções. A Tabela 3 mostra como os campos são organizados. Percebe se que os campos `rsn'` e `rd'` são de 3 bits, permitindo apenas operação entre os registradores `x8` a `x15`, que são os mais comumente utilizado. Neste caso os bits do imediato variam em cada instrução.

### 2.1.3 Linguagem assembly

O nível das instruções em representação binária é denominado de código de máquina, e é o que o processador ira conseguir traduzir em sua *lookup table*. Contudo, as instruções são dificilmente analisadas neste nível, sendo preferível a conversão a mnemônicos para compreensão humana. Por conta disto códigos não são diretamente lidos pelos computadores, dependendo de programas compiladores para realizar esta tradução para a máquina. Diversos níveis de compiladores podem ser utilizados, alguns compilando para linguagens de programação intermediárias antes da conversão para a linguagem alvo.

A primeira linguagem de programação de alto nível foi desenvolvida por Zuse (1948), e denominada *Plankalkül*, vindo das palavras em alemão "*der Plan*" e "*der Kalkül*", significando

Tabela 3 – Formato das instruções compactas de 16 bits do RISC-V

| 15 14 13 12 | 11 10 9 7 | 6 5 | 4 2 | 1 0 | **Tipo** |
|---|---|---|---|---|---|
| *funct4* | `rd/rs1` | `rs2` | | *opcode* | CR |
| *funct3* imm | `rd/rs1` | imm | | *opcode* | CI |
| *funct3* | imm | `rs2` | | *opcode* | CSS |
| *funct3* | imm | | `rd'` | *opcode* | CIW |
| *funct3* | imm | `rs1'` | imm | `rd'` *opcode* | CL |
| *funct3* | imm | `rs1'` | imm | `rs2'` *opcode* | CS |
| *funct6* | | `rd/rs1'` | funct2 | `rs2'` *opcode* | CA |
| *funct3* | imm | `rs1'` | imm | *opcode* | CB |
| *funct3* | imm | | | *opcode* | CJ |

Fonte: (WATERMAN; ASANOVIĆ, 2021)

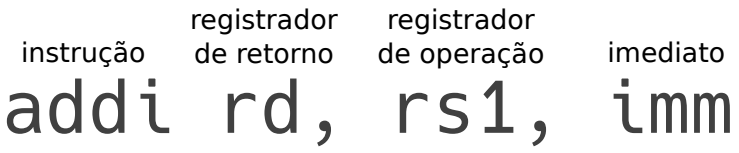
"cálculo planejado". O objetivo foi traduzir códigos de máquinas para identificadores, representados por letras. Após isto, Wilkes, Wheeler e Gill (1951) utilizam o termo *assembler* para definir programas para a junção de seções em um algoritmo único, no contexto de estudos e aplicações para o computador britânico EDSAC. Com isto a linguagem assembly foi definida, sendo uma forma mais simples na época de se escrever programas, mas requisitando de um software para realizar esta tradução. A partir disto outras linguagens foram surgindo, a fim de simplificar a interpretação humana, como Fortran, Cobol, e C.

Atualmente, a linguagem assembly é raramente recomendada como linguagem para uso direto de desenvolvedores. Com diversas linguagens de mais fácil interpretação humana, assembly está mais próximo da máquina do que destas. Apesar de ser de mais difícil compreensão humana, geralmente a sua tradução é de um para um para o código de máquina, permitindo um maior controle e ajuste das operações realizadas, com a capacidade de utilizar todas as ferramentas disponíveis pelo processador. Por conta disto, apesar de compiladores modernos, a eficiência do programa tende a reduzir quanto mais longe a linguagem é da máquina. Como analisado por Leiserson et al. (2020) com uma multiplicação matricial, em *Python* leva cerca de 25,5 mil segundos para executar o cálculo, em comparação com 0,41 segundos em um programa que utiliza as instruções vetoriais avançadas da arquitetura x86, ou seja, uma diferença de cerca de 62 mil vezes ao utilizar as corretas operações providas pela CPU, operações que podem fugir do escopo dos compiladores.

A linguagem assembly é definida de forma diferente para cada arquitetura, e pouco padronizada por normas. Contudo, existem certas similaridades entre as sintaxes atuais. Na Figura 2 se percebe um exemplo de uma instrução.

Esta instrução da Figura 2 se refere especificamente ao RISC-V, contudo, para demais arquiteturas a sintaxe pode ser parcialmente compreendida. As instruções geralmente são de 3 letras, afim de manter a simplicidade. Percebe-se como a instrução é segmentada de forma similar ao exato inverso dos formatos visto na Tabela 2. Isto permite uma otimização do compilador, em relação a uma operação do tipo `rd=rs1+imm`, porque nesse caso o operador

Figura 2 – Componentes de uma instrução assembly




Fonte: Autoria Própria (2022)

se encontra no meio da instrução, necessitando de reorganização.

O RISC-V assembly apresenta uma lista maior de mnemônicos do que de instruções na sua ISA, porém estes são convertidos em suas respectivas instruções no momento da compilação, otimização que ocorre também para as instruções que podem ser reduzidas para o seu formato compacto.

As instruções agem sempre nos devidos registradores, logo instruções de aritmética de pontos flutuantes opera somente com os registradores de pontos flutuantes. Certas instruções permitem operações de conversão entre inteiros e pontos flutuante, e estas especificam os registradores possíveis. Para melhor organização dos registradores certos padrões são organizados pela interface de aplicação binária, *Application Binary Interface* (ABI) em inglês. A Quadro 3 mostra as convenções de utilização e nomeação dos registradores de inteiros e pontos flutuantes.

Quadro 3 – Convenções da ABI do RISC-V para registradores

| **Inteiros** | | |
|---|---|---|
| **Valor** | **Mnemônico** | **Descrição** |
| x0 | zero | Todos os bits deste registrador são sempre zero |
| x1 | ra | Endereço de retorno da função |
| x2 | sp | Ponteiro para o topo do *stack* |
| x3 | gp | Ponteiro global |
| x4 | tp | Ponteiro do *thread* |
| x5-x7 | t0-t2 | Registradores temporários |
| x8-x9 | s0-s1 | Registradores salvos pelo chamante da função |
| x10-x17 | a0-a7 | Registradores de argumento |
| x18-x27 | s2-s11 | Registradores salvos pelo chamante da função |
| x28-x31 | t3-t6 | Registradores temporários |
| **Pontos flutuantes** | | |
| **Valor** | **Mnemônico** | **Descrição** |
| f0-f7 | ft0-ft7 | Registradores temporários |
| f8-f9 | fs0-fs1 | Registradores salvos pelo chamante da função |
| f10-f17 | fa0-fa7 | Registradores de argumento |
| f18-f27 | fs2-fs11 | Registradores salvos pelo chamante da função |
| f28-f31 | ft8-ft11 | Registradores temporários |

Fonte: (RISC-V, 2021)

Os registradores salvos pelo chamante, por convenção, são armazenados no *stack* antes

de saltar para a função. É importante isto, porque geralmente não se conhece os registradores utilizados pela função, podendo sobrescrever os dados. Como convenção, a função chamada deve salvar o `ra` no *stack* para permitir o retorno ao chamante. Os valores de `gp` e `tp` são setados apenas no início de um novo programa ou *thread*, utilizados para guardar as variáveis de seus respectivos níveis. O ponteiro do topo do *stack* deve ser atualizado quando acrescentar ou retirar valores.

Os mnemônicos das instruções podem operar com os valores dos registradores ou com os seus mnemônicos, notando-se que os registradores de ponto flutuante podem possuir o mesmo valor, separando-os apenas pelo formato da instrução utilizada. Certas regras adicionais, fora do escopo das especificações da ABI, podem ser aplicadas dependendo do compilador utilizado. No momento, o principal e um dos únicos compiladores existentes é o GNU Compiler[1], então as suas considerações serão utilizadas. Uma lista de alguns mnemônicos utilizáveis dos conjuntos base está presente na Quadro 4, sendo alguns diferentes das instruções em si, decompostos em outras instruções antes da geração do código de máquina.

Para instruções de saltos e *branchs* o valor de *offset* não precisa ser implicitamente utilizado. O compilador interpreta certos símbolos e realiza o cálculo necessário para o valor de *offset* a se utilizar. O símbolo pode ser dado como nomes específicos ou como números. Caso utilize números, deve se identificar se o símbolo está a frente ou atrás da instrução, indicando com `f` ou `b` respectivamente. Além disto, o diretivo `.globl` identifica para o compilador que este endereço pode ser referenciado em outros arquivos, podendo até ser utilizados em C ou C++, com o indicador `extern`. Este símbolo é armazenado em uma tabela de endereços denominada de *global offset table*, esta pode ser acessada, durante a montagem do código de máquina, permitindo que o endereço da função seja obtido para utilização em qualquer programa *linkado* ao arquivo de referência do símbolo. Uma explicação detalhada sobre a programação assembly para RISC-V pode ser vista em Shakti (2020).

Códigos assembly também possuem segmentações no algoritmo em seções, utilizadas para haver uma diferenciação de função no programa. Esta separação é importante de ser feita, porque em código de máquina não há diferença entre dados e instruções, podendo ocorrer problemas caso o endereço errado seja lido. Existem diversas seções possíveis de serem utilizadas, algumas das mais utilizadas são a seção `.text` onde se define seção a ser lida como código, `.data` como região de armazenamento de dados, e `.rodata` que armazena dados de apenas leitura. Um exemplo de programa assembly está presente no Algoritmo 1, sendo o carácter "#" indicador de um comentário.

Outra seção comumente utilizada é a `.bss`, que é o símbolo de início de bloco, ou *block starting symbol* em inglês. Nesta seção as variáveis são definidas mas não são alocados valores. Em comparação, nas seções de `.data` os valores são salvos no código binário do programa, aumentando seu tamanho, porque os dados são previamente definidos. Na seção `.bss` o código possui somente uma indicação de alocação, com informação do tamanho da

[1] Disponível em: https://github.com/riscv-collab/riscv-gnu-toolchain

Quadro 4 – Lista de alguns mnemônicos de instruções base do RISC-V

| **RV32I** | | |
|---|---|---|
| **Inst** | **Parâmetros** | **Descrição** |
| add | rd, rs1, rs2 | Adiciona rs1 a rs2 e armazena em rd |
| addi | rd, rs1, imm | Adiciona rs1 ao valor imm e armazena em rd |
| and | rd, rs1, rs2 | Realiza o AND de rs1 e rs2 e armazena em rd |
| auipc | rd, imm | Adiciona o valor sem sinal de imm ao pc e o copia em rd |
| beq | rs1, rs2, off | Se rs1 = rs2, adiciona off ao pc |
| beqz | rs1, off | Converte para beq rs1, x0, off |
| bgt | rs1, rs2, off | Converte para blt rs2, rs1, off |
| blt | rs1, rs2, off | Se rs1 < rs2, adiciona off ao pc |
| bltu | rs1, rs2, off | Se rs1 ≤ rs2, adiciona off ao pc |
| jal | rd, off | Copia o endereço da próxima instrução para rd e acrescenta off ao pc |
| lb | rd, off(rs1) | Copia um byte da memória rs1+off e o armazena em rd |
| li | rd, imm | Converte para lui e/ou addi, dependendo do tamanho de imm |
| lui | rd, imm | Carrega o valor sem sinal de imm em rd e o desloca 12 bits à esquerda |
| mv | rd, rs1 | Converte para addi rd, rs1 |
| nop | | Converte para addi x0, x0, 0 |
| sb | rs2, off(rs1) | Armazena um byte de rs2 na memória rs1+off |
| sw | rs2, off(rs1) | Armazena 4 bytes de rs2 na memória rs1+off |
| sll | rd, rs1, rs2 | Desloca rs1 rs2 bits à esquerda e armazena em rd |
| **RV64I** | | |
| addiw | rd, rs1, imm | Adiciona rs1 a imm, trunca em 32 bits, e armazena em rd |
| addw | rd, rs1, rs2 | Adiciona rs1 a rs2, trunca em 32 bits, e armazena em rd |
| ld | rd, off(rs1) | Copia 8 bytes da memória rs1+off e os armazenam em rd |
| lw | rd, off(rs1) | Copia 4 bytes da memória rs1+off e os armazenam em rd |
| sd | rs2, off(rs1) | Armazena 8 bytes de rs2 na memória rs1+off |

Fonte: (WATERMAN; ASANOVIĆ, 2021)

área a ser reservada na memória de acesso aleatório, ou *Random-Access Memory* (RAM) em inglês. Geralmente estes valores são alocados no *heap* da memória, em oposição ao *stack*, contudo, o software é livre para alocar as posições. Para esta alocação de dados também existe o diretivo `.comm`, que pode ser utilizado em outras seções e permite criar um objeto no `.bss` sem declarar a seção especificamente, apenas definindo o símbolo e o tamanho do objeto. Existem ainda seções para alocação de dados para cada *thread*, com os identificadores `.tbss` e `.tdata`, porém o software é responsável por suas alocações.

O diretivo `.align` é utilizado para garantir o alinhamento dos bytes no programa. Se específica o número de bytes em múltiplo de 2 a ser preenchido. Isto previne que a próxima instrução ou dado caia em um endereço que não seja múltiplo do valor definido, item importante para as instruções, que precisam estar em endereços divisíveis por 2 bytes no RISC-V. Também as

**Algorithme 1** Exemplo de código assembly

```
1: .section .rodata # Seção de dados read-only
2:   simbolo_de_dados:
3:     .ascii "Hello World!\0"
4:
5: .section .text # Seção de código
6: .align 2 # Alinha o endereço para 4 bytes
7:   .globl simbolo_global
8:   simbolo_de_funcao:
9:     addi sp, sp, -16
10:    sd ra, 8(sp) # É boa prática armazenar o ra no stack
11:    la t0, simbolo_de_dados
12:    li t2, 0x6F
13:  1:
14:    lbu t1, 0(t0)
15:    c.addi t0, 1 # "c."força o uso da instrução compacta
16:    beq t1, t2, 1f
17:    bnez t1, 1b
18:  1: # Símbolos numéricos podem ser repetidos
19:    mv a0, t1
20:    ld ra, 8(sp)
21:    addi sp, sp, 16
22:    ret # Função retorna para o endereço salvo em ra
```

instruções de acesso a memória levantarão uma exceção quando acessam um dado desalinhado ao seu respectivo tamanho, como por exemplo, uma instrução `ld` somente pode carregar valores de endereços múltiplos de 8 bytes.

Todas as informações de seções e diretivos são utilizadas como intermediário para construção do código pelo montador. Os arquivos resultantes podem ser *linkados* por outros programas, quando compilados como livrarias, apenas se atentando para colisão de símbolos, porque pode ocorrer da utilização do mesmo nome para símbolos globais. A função de início do código é definida pelo símbolo de entrada, sendo este normalmente definido pelo `_start`, mas podendo ser alterado no campo de endereço de entrada no software de montagem. Na construção do código de máquina a função no símbolo de entrada é posicionada no endereço 0x0 do programa. Demais composições de montagem do código pelo RISC-V estão presentes em sua ABI (RISC-V, 2021).

A linguagem assembly é o que permite o primeiro nível de comunicação humana com o processador, sendo as operações convertidas de forma quase que direta para o código de máquina. Certas arquiteturas providenciam um sistema complexo de ser trabalhado neste nível, sendo preferível utilizar linguagens de alto nível. A simplicidade do RISC-V facilita o desenvolvimento neste nível, possibilitando uma simples lista de instruções e materiais didáticos providos da fundação, permitindo que se opere de forma mais eficiente com as ferramentas do processador.

### 2.1.4 Comparação com outras arquiteturas

Um dos pontos a ser analisados quando se trata de qualquer tecnologia, é como se relaciona com o estado da arte. As arquiteturas presentes hoje se resumem a x86, no lado dos sistemas CISCs, e a ARM, no lado dos RISCs. A prioridade em sistemas CISCs é a compressão do código e aceleração da execução, e tenta alcançar isto com providenciando um extenso conjunto de instruções específicas para ser utilizadas pelo compilador. Contudo, isto acaba levando a chips maiores e maior gasto de energia. Os RISCs priorizam a simplicidade e padronização, com um conjunto simplificado de instruções, que operem em um único ciclo de *clock* e utilize de forma eficiente o espaço dos chips. Porém menos instruções disponíveis podem resultar em maiores programas para realizarem a mesma operação. A diferença entre RISC-V e x86 é mais clara do que entre ARM, contudo, o RISC-V adere mais a simplicidade da ISA, sendo um melhor representativo dos RISCs.

Em relação aos sistemas CISCs, um dos principais fatores de distinção entre RISCs é o manuseio de memória. Os sistemas CISCs possuem geralmente um registrador para cada propósito, e o resto das operações é realizada na memória, porém, acesso a memória é extremamente lento em relação ao *clock* do sistema. Por conta disto, foi-se implementado memórias cache de diversos níveis para permitir acesso mais rápido a secções de código e variáveis comumente utilizadas. Mas a memória cache tende a ser mais escassa para o computador, sendo presente geralmente no próprio chip da CPU, seu espaço deve ser reduzido. Por conta disto processadores modernos investem em algoritmos de previsão para sempre manter a memória cache com dados úteis, evitando acessos a RAM. Mas nem sempre isto é possível, muitas instruções buscam endereços inexistentes no cache, necessitando de acesso a RAM, forçando o núcleo a esperar ou divergir da tarefa atual. E demais problemas surgem caso múltiplos núcleos operem sobre o mesmo endereço, podendo colidir em *race conditions*. Além disto, CISCs possuem tempo de operação e tamanho de instrução variável, dificultando a organização do pipeline. Os RISCs visam trabalhar com os dados de forma mais eficiente, mantendo-os nos registradores, esses individuais para cada núcleo, sem permitir operação direta a memória. Isto acelera a operação de programas que atuam com diversas variáveis. E as instruções em um RISC são de um único ciclo, apenas com algumas exceções, isto torna o programa mais simples e otimiza a operação do processador (STALLINGS, 2002).

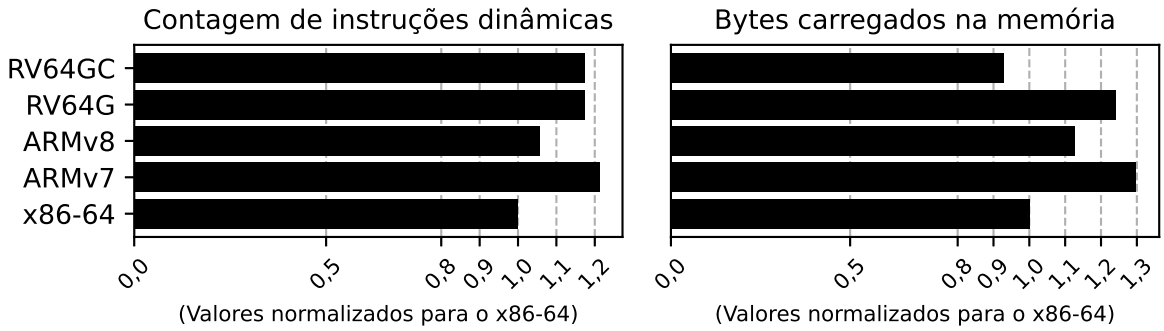
Para análise da performance de uma ISA específica, é preciso analisar como estas podem ser avaliadas. É comumente utilizado a proposta de Emer e Clark (1984) para esta análise, em que é definido que a arquitetura de um computador apenas tem controle das instruções, sendo o resto fatores dependentes do programa ou do dispositivo. Isto pode ser visto em (1), denominada lei de ferro da performance do processador.

$$\frac{\text{Tempo}}{\text{Programa}} = \frac{\text{Instruções}}{\text{Programa}} \times \frac{\text{Ciclos de clock}}{\text{Instrução}} \times \frac{\text{Tempo}}{\text{Ciclos de clock}} \tag{1}$$

O conjunto de instruções só consegue alterar os termos de “Instruções por Programa”,

e a microarquitetura somente o de "Ciclos de *clock* por Instrução". O x86 visou reduzir o primeiro termo, e os RISCs, o segundo. Porém, é difícil realizar uma análise no segundo termo, porque este termo depende da implementação do chip e não altera suas instruções, então não são fixos para nenhum conjunto de instruções. Então considera-se os RISCs com o segundo termo próximo de um, devido a sua filosofia de projeto, mas é um termo somente possível de ser avaliado quantitativamente entre implementações de arquiteturas. Considerando isto, o primeiro termo pode avaliar diretamente as ISAs de forma quantitativa. O x86 possui um vasto conjunto de instruções contudo, como visto anteriormente, a relação entre quantidade de instruções e a redução do programa não é linear, porque um conjunto pequeno de instruções únicas preenche grande parte das operações. O RISC-V possui ainda menos instruções únicas, e instruções mais simples que o ARM. Testes realizados por Celio et al. (2016) mostram a diferença entre instruções dinâmicas e tamanho de programas entre diferentes ISAs, e o resultado esta presente na Figura 3. Instruções dinâmicas são as executadas pelo processador na hora de execução do código.

Figura 3 – Testes de contagem de instruções e tamanho de programa de diferentes ISAs




Fonte: Realizada a média aritmética dos resultados de Celio et al. (2016)

Dos resultados da Figura 3 certos pontos podem ser verificados. Primeiramente, o conjunto de instruções compactas *RV64GC* providência um código mais compacto do que o do x86, apesar de possuir mais instruções. Isto é devido ao x86 possuir instruções de tamanho variável e podendo atingir 15 bytes em certas instruções (INTEL, 2021), em comparação, o RISC-V possui instruções de 4 bytes e 2 bytes somente. Nota-se também como as ISAs RISCs apresenta maior contagem de instruções do que o x86. E o RISC-V apresenta maior contagem de instruções do que o ARM.

As diferenças entre ARM e RISC-V não são predominantes, mas um dos fatores que se destaca são suas filosofias de desenvolvimento. ARM é um produto comercial e é vendido como tal, providenciando consultoria e desenvolvimento para clientes interessados, seu foco é providenciar um produto viável para o mercado. O RISC-V é livre e gratuito, pensado na eficiência e simplicidade, a ISA é divida em compactas extensões, e o seu desenvolvimento é guiado por *peer review* e aberto a qualquer indivíduo.

## 2.2 Renderização gráfica

A foto e o filme aumentaram drasticamente nossa abrangência do mundo, nos permitindo ver acontecimentos do redor da Terra, que podem ser transportados em um rolo de filme. Estas tecnologias revolucionaram a arte e a ciência. Ao passar dos séculos esta evolução culminou nas telas de televisores e computadores, e isto permitiu que momentos fossem transmitidos em tempo real, permitindo não apenas a captura de imagens no passado, mas, a partir deste momento, também do presente. E com a evolução da computação gráfica, pode-se criar mundos fantasiosos similares aos nossos, e interagir com esses instantaneamente.

Para permitir esta capacidade de manipulação de objetos fictícios nos dias atuais, certos marcos tiveram que ser alcançados. A partir da capacidade de renderização bidimensional de imagens, o ser humano ainda buscava formas de integrar modelos tridimensionais nas telas. O primeiro a realizar este feito de uma forma ampla foi Sutherland (1964), com o programa Sketchpad. Com o cálculo de operações matemáticas de relações entre linhas, representações de pontos tridimensionais foram mostradas em uma tela de duas dimensões. Esta tecnologia de renderização permitiu um uso mais eficiente dos computadores, que podiam realizar simulações e cálculos matemáticos complexos, mas previamente conseguiam apenas mostrar os resultados em uma lista impressa de valores, tornando para o indivíduo interpretá-los. Gráficos e formas geométricas começaram a transmitir informações de forma mais clara.

Certos itens criados da computação gráfica se integraram ao nosso cotidiano, como a imagem do cursor do mouse e a ideia das "janelas" dos computadores. Devido ao seu vasto uso, esta tecnologia já é, de certo modo, natural. Atualmente saber interagir com interfaces gráficas não é um diferencial, e sim, a norma, e processamento gráfico desenvolveu em seu próprio ramo. Contudo, na década de 60 esta tecnologia era restringida para os poucos computadores que tinham capacidade de processá-la, e aos programadores que sabiam utilizá-la. Na década de 70 seu uso havia se espalhado para diversas empresas que viam seu potencial, utilizando a tecnologia para fins de desenvolvimentos de produtos e desenho técnico. Com isto surgiu diversas empresas para suprir essa demanda de computadores com *displays* CRTs. Esta utilização foi de grande parte da industria de circuito integrados, que se utilizou da tecnologia para desenvolver circuitos de larga escala (MACHOVER, 1978).

Os softwares de computação gráfica na época se desenvolveram de forma independente. A maioria se utilizava no principio básico de armazenar objetos na memória, e utilizar algoritmos que executassem a cada atualização do monitor para criar uma imagem com a informação presente, contudo, obtinham os resultados por métodos diferentes. Isto causava problemas para usuários que precisassem transferir arquivos entre máquinas. A primeira interface de programação de aplicação, ou *Application Programming Interface* (API), a padronizar a renderização gráfica foi o sistema CORE, desenvolvido pelo Comitê de Planejamento e Uniformização Gráfico da Association for Computing Machinery (MICHENER; DAM, 1978). Esta API permitia a manipulação e transformação de imagens 2D, e de objetos 3D. A renderização tridimensional era

feita pela decomposição do objeto em formas primitivas, presentes em coordenadas normalizadas, que então seriam transformadas para o *display*, pelas transformações de visualização, com a capacidade de corte de polígonos dependendo da posição da câmera virtual.

Nas sequentes décadas de 80 e 90, a API de renderização PHIGS se tornou predominante. O PHIGS se baseia no CORE, porém providencia diversas ferramentas para a programação de software. A sua principal diferença era a capacidade de alocar hierarquias para os objetos, permitindo uma melhor manipulação de cenas. Esta hierarquia foi acompanhada em APIs modernas, devido a sua utilidade para renderização de modelos complexos, com diversas partes que devem se mover de forma independente porém conectadas a um elemento base. O processo de renderização pelo PHIGS carregava os modelos em uma estrutura central, onde seriam carregados em uma área de trabalho para serem utilizados em cena. Para isto, se utiliza quatro etapas de transformação, primeiramente o objeto é transformado para suas coordenadas no "mundo", em seguida era transformado para coordenadas adequadas para visualização, com isto operações de mapeamento e *clipping* ocorrem e transformações específicas do dispositivo são aplicadas, e o objeto aparece na tela (SHUEY; BAILEY; MORRISSEY, 1986).

Atualmente o estado da arte se encontra em algumas poucas APIs, com destaque para o OpenGL. Desenvolvida por Segal e Akeley (1994), o OpenGL é uma API que consiste em um conjunto de ferramentas para processamento gráfico com capacidade de extensões, para providenciar funções adicionais. As extensões e as versões do OpenGL disponíveis dependem da GPU utilizada, porque esta API deve ser implementada na microarquitetura do dispositivo para ser suportada.

Apesar de APIs antigas possuírem suporte para monitores vetoriais, com a presença de *displays* de LCD, as APIs novas são voltadas para rasterização de *pixels*, termo derivado de *picture element* em inglês. *Pixels* são elementos finitos de uma decomposição de uma imagem, em comparação com as linhas de uma renderização vetorial, que podiam ser criadas em CRTs. O OpenGL também é voltado para renderização rasterizada em *pixels*.

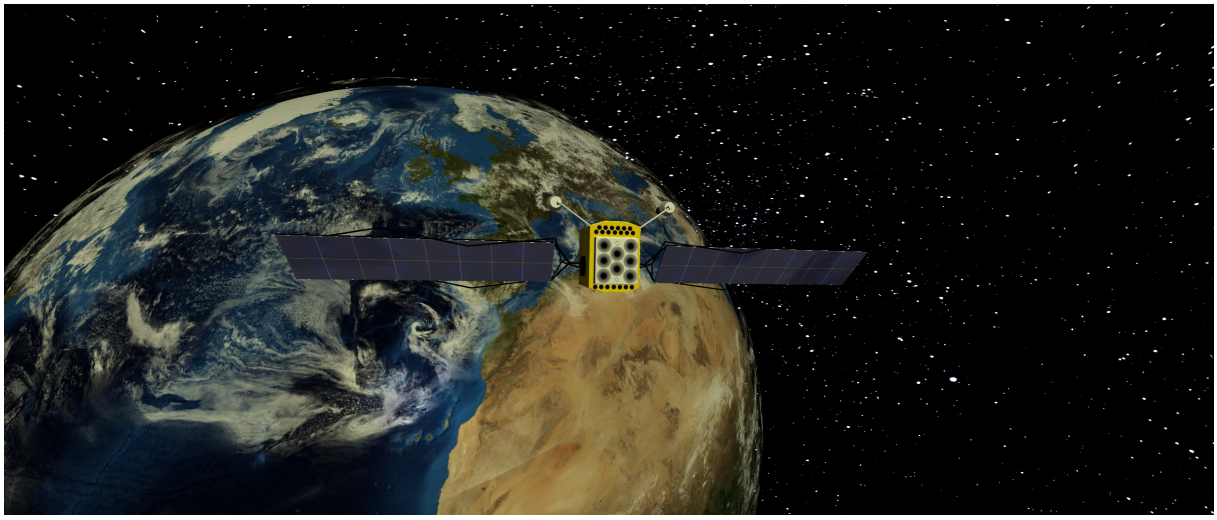
O OpenGL possui uma pipeline para transformação dos dados, dentro dessas, as transformações tridimensionais e rasterizações são realizadas. Estas transformações ocorrem nos *shaders*, que são algoritmos programáveis pelo desenvolvedor, com auxílio de funções providas pelo OpenGL. Estes contudo podem realizar tarefas não necessariamente relacionadas à renderização, sendo apenas códigos que rodam na GPU. Em versões prévias, esta programação era realizada em assembly, porém versões mais atuais se utilizam de sintaxes similar a C (SEGAL; AKELEY, 2019).

Uma aplicação de renderização básica para *desktop* utilizando esta API[2] pode ser visto na Figura 4 para se ter uma base dos objetivos buscados. Percebe-se recursos como aplicação de texturas, e iluminação direcional, itens que serão abordados seguindo esta seção.

O OpenGL possui também versões otimizadas para sistemas embarcados, pela linha OpenGL ES. Neste caso somente as etapas de renderização mínimas são implementadas. Como

[2]Disponível em: https://github.com/luczis/Satellite-In-Orbit-OpenGL

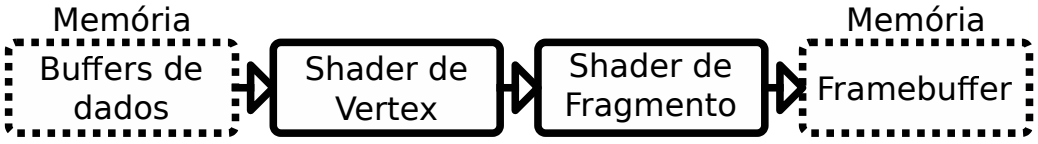
Figura 4 – Renderização de modelos 3D em *desktop* utilizando OpenGL



Fonte: Autoria própria (2022)

visto na Figura 5, os dados são inicialmente armazenados na memória, a partir disto, os *shaders* realizam as transformações necessárias para a projeção no *framebuffer*, que pode então ser mostrado à tela. O *shader* de *vertex* é a região do código responsável pelas operações nas posições do objeto, rodando uma vez para cada *vertex* do modelo. Este é o responsável pelas transformações tridimensionais necessárias. O *shader* de fragmento realiza a rasterização da imagem, aplicando as texturas e a iluminação no modelo, além disto, mapeia os *pixels* da imagem resultante na região de memória definida para o *framebuffer*.

Figura 5 – Pipeline de renderização do OpenGL ES




Fonte: (SEGAL; AKELEY, 2019)

As etapas de operação do OpenGL ES podem então ser decompostas, para uma análise do funcionamento do algoritmo de renderização tridimensional. Sendo as transformações realizadas pelo *shader* de *vertex* dadas na Subseção 4.6.1. E as funções relacionadas ao *shader* de fragmento são apresentadas na Subseção 2.2.3.

### 2.2.1 Vertexes

A palavra *vertex*, traduzida do inglês para o português, significa vértice, contudo, será evitado utilizar este termo para não haver a associação com posições geométricas. O *vertex* pode possuir além de informação da posição dos vértices no espaço, como informações de mapeamento de textura, informações de vetores normais, informações de materiais e cores, e

qualquer outro dado pertinente necessário ao programa. Trata-se de um vetor de dados que é enviado para a memória da GPU para ser trabalhado. Os próprios vértices podem também ser passados fora de um *vertex*, como em técnicas de tesselação, em que novos polígonos são gerados com base em texturas específicas. Por conta destes fatores, se evitará realizar esta ligação entre essas palavras.

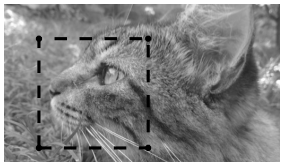
Para fins deste trabalho, serão considerados três itens geralmente utilizados para renderização tridimensional, a posição dos vértices do modelo, as coordenadas UVs utilizadas para texturas, e as direções do vetor normal. Uma demonstração do uso destes valores pode ser visto na Figura 6. Um guia detalhado sobre a utilização de *vertexes* é detalhado por Neider, Davis e Woo (1993).

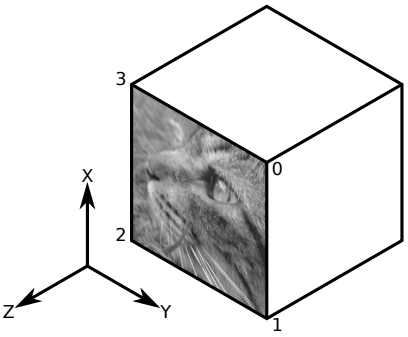
Figura 6 – Demonstração do uso de dados contidos nos *vertexes*

Dados dos vertexes

| Índice | Posição | | | UV | | Normal | | |
|---|---|---|---|---|---|---|---|---|
| | X | Y | Z | U | V | X | Y | Z |
| 0 | 1,0 | 1,0 | 1,0 | 0,52 | 0,80 | 0,0 | 0,0 | 1,0 |
| 1 | -1,0 | 1,0 | 1,0 | 0,52 | 0,08 | 0,0 | 0,0 | 1,0 |
| 2 | -1,0 | -1,0 | 1,0 | 0,12 | 0,08 | 0,0 | 0,0 | 1,0 |
| 3 | 1,0 | -1,0 | 1,0 | 0,12 | 0,80 | 0,0 | 0,0 | 1,0 |

Textura






Fonte: Autoria própria (2022)

#### 2.2.1.1 Posição

Modelos 3Ds são formados geralmente por uma superfície de interior vazio. Esta superfície é mapeada em um conjunto de vértices às suas respectivas posições. Certas técnicas podem reconstruir a estrutura utilizando os vértices de forma indireta. Contudo, a forma mais simples é de construir polígonos com três ou mais vértices, e renderizar com base nestes. Por conta disto, nesta técnica, formas curvas são representadas pela decomposição em faces planas, não sendo verdadeiramente reconstruídas a seu formato original.

Para renderizar um objeto, são passados nos *vertexes* as informações de posições dos vértices. Estas informações geralmente são convertidas por matrizes de transformação antes de serem utilizadas para rasterização do modelo. Porém, as informações dos modelos são armazenadas normalizadas ao modelo, com o centro do modelo em zero. Para renderização 3D, estes pontos são armazenados em três dimensões, porém, certas APIs conseguem utilizar formas 2D.

Os valores de posição são geralmente os que sofrem mais manipulações matemáticas antes da renderização, porque são os que formam o modelo tridimensional, e precisam ser mapeados para a tela.

O mesmo vértice de posição pode, e geralmente é, utilizado por mais de um polígono. As informações sobre quais posições a serem utilizadas são armazenadas em um *buffer* de indexação, que complementa o *buffer* de *vertexes* para renderização do modelo.

#### 2.2.1.2 Mapa UV

Os objetos 3D apenas com posição não podem ser rasterizados a tela, porque não possuem informações de cores. Cores lisas podem ser atribuídas para a superfície, e podem ser assimiladas nos *vertexes*, contudo, isto somente gera o realismo necessário caso o número de vértices sejam altos, a ponto do olho humano não detectar as faces individuais do objeto. Para providenciar a ilusão de realidade, sem aumentar a qualidade do modelo, certas técnicas podem ser empregadas. Uma das mais comuns e mais utilizadas é o mapeamento UV, que consiste em projetar imagens 2D em um modelo 3D.

Esta técnica apresenta certos defeitos, como engrandecimento de *pixels* dependendo da posição da câmera, porém, é uma das formas mais eficazes de simular complexidade. E para ocorrer este mapeamento, dois valores reais, geralmente de 0 a 1, são assimilados aos *vertexes*. Estes valores são denominados de "U" e "V", para diferenciar com os valores de "x", "y" e "z", utilizados para a posição, e sendo a origem da terminologia.

Estes valores são então mapeados a uma imagem 2D de *pixels*, denominada "textura". O processo de cálculo pode ser feito como em (2), porém, técnicas mais avançadas de projeção podem ser utilizadas para melhorar a qualidade da imagem, e lidar com efeitos de distorção.

$$\begin{gathered}(U_{Pixel}, V_{Pixel}) = (round(U \cdot Largura_{Pixel}), round(V \cdot Altura_{Pixel})) \\ U_{Pixel}, V_{Pixel} \in \mathbb{N},\ \ (0{,}0) \leq (U{,}V) < (1{,}1)\end{gathered} \tag{2}$$

Valores UV maiores ou iguais a 1 e menores que 0 devem ser mapeados a esta faixa por alguma função. Algumas das formas de limitação de UV são apresentadas na Figura 7.

Figura 7 – Formas de mapeamento para valores de UV fora dos limites


Fonte: Autoria própria (2022)

Os valores de UV são geralmente fixos para cada vértice, independente das face que será renderizada, exceto quando há um salto na textura. Quando ocorre isto diferentes posições das texturas são mapeadas a um mesmo vértice.

#### 2.2.1.3 Normal

Os valores da normal são utilizados para modelos 3D, e representam os valores do vetor normal a face do polígono. Este valor pode ser utilizado para certas simulações físicas, testes de colisões, e iluminação. Este dado é um vetor unitário de três dimensões, que apontam perpendicularmente à face. Como este valor é intrínseco ao polígono, ele se repete para os *vertexes* que o compõe.

Este valor pode ser calculado a partir dos pontos da face, contudo de forma ambígua com dois possíveis sentidos. Este vetor define onde é o exterior do modelo, e isto pode ser utilizado para técnicas de *culling*, onde não se renderiza polígonos vistos do lado interno, para providenciar aumento de performance. A definição do exterior também é relevante para se aplicar as acelerações necessárias em caso de colisão, e para cálculos de iluminação.

Em geral a iluminação é divida em três componentes, a iluminação ambiente, que está presente independente da orientação da normal, a iluminação difusa, que depende da posição da luz e da orientação da face, e a iluminação especular, que leva em consideração a posição da câmera. Este modo de reflexão é nomeado de modelo de Phong, devido ao trabalho de Phong (1975).

Para a iluminação ambiente, apenas uma constante multiplica os valores de cores da face do polígono. Este valor é único para todas as faces do modelo.

Para o cálculo da iluminação difusa, é realizado o produto escalar entre a normal e o vetor unitário da distância entre o ponto de luz e o fragmento a ser iluminado. Em fontes de iluminação unidirecional este vetor unitário pode ser simplificado pelo negativo da direção de iluminação. O valor é calculado para cada fragmento da face, contudo, este cálculo pode ser realizado apenas uma única vez para cada face plana do modelo, utilizando o centro geométrico, para otimizações. O seu valor é dado por (3). O valor do coeficiente deve ser sempre positivo para não reduzir a contribuição de outros componentes iluminantes.

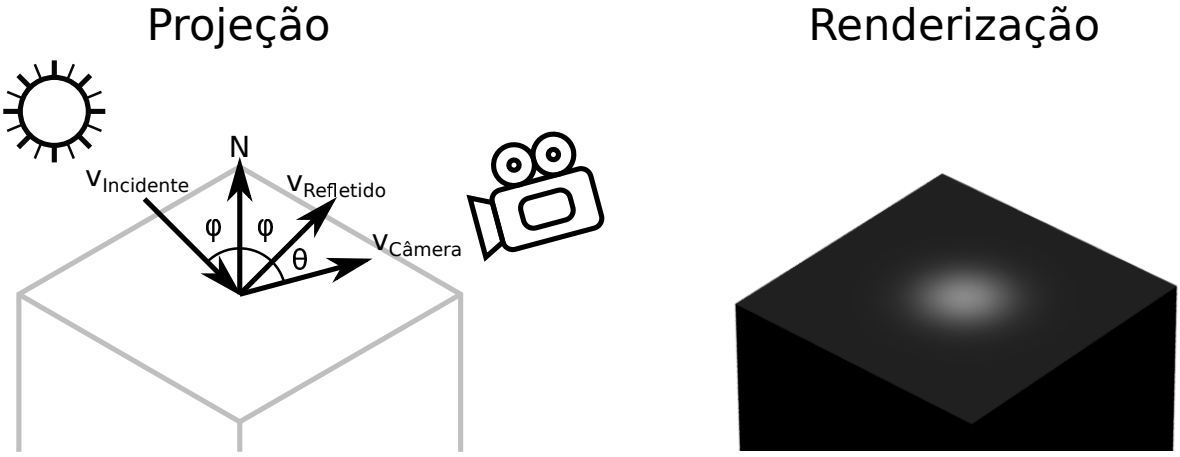
$$L_{Difuso} = max\left(0,\hat{N}\cdot\left[\frac{\vec{P}_{Luz}-\vec{P}_{Fragmento}}{|\vec{P}_{Luz}-\vec{P}_{Fragmento}|}\right]\right) \tag{3}$$

A iluminação especular considera, além da normal e posição do ponto de luz, a posição da câmera, o que acaba por produzir um efeito reflexivo no objeto. Neste caso é realizado a reflexão do vetor de iluminação incidente, e realizado o produto escalar entre o resultante e o vetor de distância entre o fragmento e a câmera. E com o resultado do produto escalar, é elevado a um número maior do que um, geralmente múltiplo de 2, para criar o efeito pontual da iluminação. A sua equação é dada por (4).

$$\hat{\boldsymbol{v}}_{Refletido} = \left( \frac{\vec{P}_{Luz} - \vec{P}_{Fragmento}}{|\vec{P}_{Luz} - \vec{P}_{Fragmento}|} \right)$$
$$L_{Especular} = max \left( 0, \hat{\boldsymbol{v}}_{Refletido} \cdot \left[ \frac{\vec{P}_{Câmera} - \vec{P}_{Fragmento}}{|\vec{P}_{Câmera} - \vec{P}_{Fragmento}|} \right] \right)^{x}, \; x \in \mathbb{R} \; x > 1 \tag{4}$$

A aplicação da iluminação especular pode ser visto na Figura 8. Percebe-se como há a presença de um ponto de iluminação predominante no centro da face renderizada, criando um efeito de reflexão no objeto.

Figura 8 – Aplicação de iluminação especular




Fonte: Autoria própria (2022)

Os coeficientes são então somados, e o resultante multiplica as cores originais do fragmento. Cada método é atribuído um peso, com esses valores dependendo do efeito desejado. A equação do fragmento pelo modelo de Phong é dado em (5). Neste caso os valores de cores são dados normalizados, porém, estes valores no momento de renderizar à tela, serão valores inteiros que dependem da profundidade de cor utilizada. Os coeficientes apresentam um efeito de sombreamento, porque geralmente estão na faixa de 0 e 1, contudo, isto cria uma iluminação das regiões não escurecidas.

$$(R,G,B)_{Iluminado} = (K_A \cdot L_{Ambiente} + K_D \cdot L_{Difuso} + K_E \cdot L_{Especular}) \, (R,G,B)_{Original}$$
$$(0,0,0) \leq (R,G,B) < (1,1,1), \;\; 0 \leq K_A + K_D + K_E \leq 1 \tag{5}$$

Há técnicas para utilizar arquivos de textura para alterar a direção da normal para cada fragmento do modelo, desenvolvida por Cignoni et al. (1998), e denominada de *normal mapping*. Nesta textura valores de cores representam variações no vetor normal, e é utilizado para aplicação mais detalhada de iluminação, providenciando sensação de profundidade e complexidade com uma quantidade reduzida de vértices.

As direções das normais são dadas em relação ao modelo, porém, para se manter em consonância com o mundo, operações de rotação devem ser aplicadas, a medida que a face rotaciona.

### 2.2.2 Matrizes de transformação

Os valores de posição são dados inicialmente em relação ao modelo, ou seja, o centro (0,0,0) é posicionado no centro do objeto. Porém, para projeção de cenas com diversos modelos é necessário o mapeamento deste objeto para coordenadas diferentes. As coordenadas globais são denominadas de coordenadas de mundo, porque representam todos os objetos que existem na cena virtual.

São utilizadas geralmente três matrizes de transformação, a de transformação do modelo, que é responsável pelo mapeamento entre as coordenadas do objeto e mundo, a de visualização, que está relacionada com a movimentação da câmera, e a de projeção, que converterá as coordenadas 3D para a tela. Estas matrizes são quadradas de tamanho quatro por quatro, por conta disto, é acrescentado um valor "w" ao vetor posição. Esta variável permitirá o deslocamento como será visto na Subsubseção 2.2.2.1, entre outros usos. As matrizes são concatenadas como visto em (6), com isto, são convertidos de coordenadas do modelo para coordenadas rasterizáveis em tela. Neste caso serão considerados as posições como vetores coluna, porém a análise pode ser realizada de forma análoga para linha, com a rotação dos elementos das matrizes pela diagonal principal.

$$\begin{gathered}(x,y,z,w)^T_{Tela} = M_P M_V M_M (x,y,z,w)^T_{Modelo} \\ M_P, M_V, M_M \in \mathbb{R}^{4\times 4}\end{gathered} \tag{6}$$

As transformações em 2D são análogas a 3D, contanto, com algumas simplificações, por conta disto não serão abordadas. Para informações sobre transformações 2D tão como 3D recomenda-se a leitura de Foley (1996).

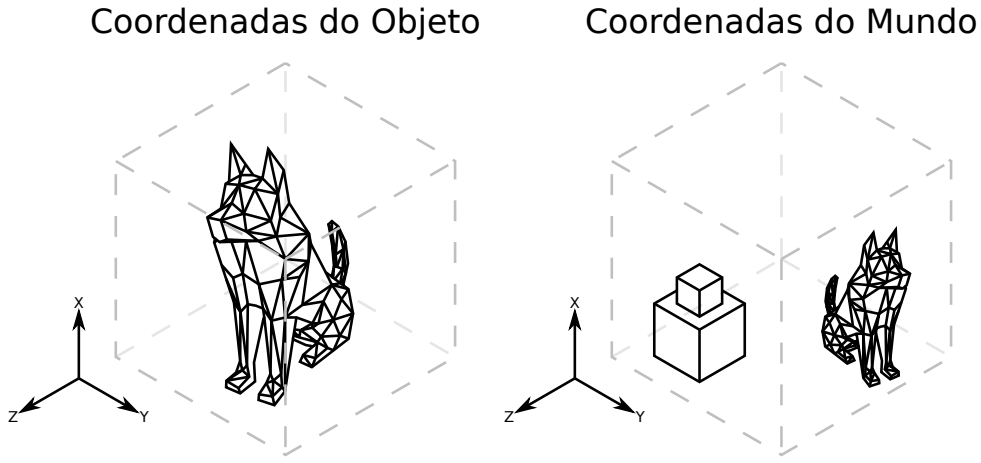
#### 2.2.2.1 Matriz do modelo

A matriz do modelo é a responsável pela transformação de coordenadas do objeto para o as coordenadas do mundo. Esta pode ser decomposta em três elementos, a conversão de escalas, que engrandecerá ou reduzirá o modelo, a rotação, e a translação, que deslocará o modelo nas coordenadas do mundo. Estes componentes são compostos de matrizes de tamanho quatro por quatro, que multiplicam as coordenadas do modelo. A transformação de escala, rotação a partir do centro de origem, e translação nas coordenadas do mundo é dada como em (7). Estas matrizes podem trocar de posição, e podem ser multiplicadas por outras matrizes adicionais, dependendo da movimentação do modelo desejado.

$$\begin{gathered}(x,y,z,w)^T_{Mundo} = M_M = M_T M_R M_E (x,y,z,w)^T_{Modelo} \\ M_T, M_R, M_E \in \mathbb{R}^{4\times 4}\end{gathered} \tag{7}$$

A aplicação desta matriz de modelo pode ser vista na Figura 9.

Figura 9 – Transformação da matriz de modelo para o mundo




Fonte: Autoria Própria (2022)

#### 2.2.2.1.1 *Escala*

A conversão de escala é realizada pela simples multiplicação dos pontos. Esta transformação é realizada ao redor da origem, convergindo ou afastando os vértices de (0,0,0). Esta matriz pode ser obtida multiplicando a matriz identidade pelo vetor de escala ($E_x$,$E_y$,$E_z$). O resultado da multiplicação da matriz escala pelo vértice pode ser visto em (8).

$$\begin{bmatrix} x \cdot E_x \\ y \cdot E_y \\ z \cdot E_z \\ w \end{bmatrix} = \begin{bmatrix} E_x & 0 & 0 & 0 \\ 0 & E_y & 0 & 0 \\ 0 & 0 & E_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \tag{8}$$

Uma matriz de translação pode ser aplicada previamente para deslocar o centro de escala do objeto.

#### 2.2.2.1.2 *Rotação*

A rotação do objeto é o item de maior intensidade computacional na matriz do modelo. A rotação é realizada ao redor de um vetor definido, por um ângulo. Para este cálculo, é necessário a análise por quaterniões primeiramente, e com isto a matriz de rotação pode ser deduzida.

Quaterniões são números complexos de quadro componentes, sendo um escalar, e três complexos, definido como $\boldsymbol{q} = q_s + q_x\hat{\boldsymbol{i}} + q_y\hat{\boldsymbol{j}} + q_z\hat{\boldsymbol{k}}$, originalmente propostos por Hamilton (1848) para análise no espaço tridimensional.

Um vetor posição tridimensional pode ser representado em forma de quaternião como $\boldsymbol{q}_p = x\hat{\boldsymbol{i}} + y\hat{\boldsymbol{j}} + z\hat{\boldsymbol{k}}$. Com este quaternião de posição, será utilizado um quaternião de rotação dado como $\boldsymbol{q}_R = cos(\theta/2) + sen(\theta/2)(r_x\hat{\boldsymbol{i}} + r_y\hat{\boldsymbol{j}} + r_z\hat{\boldsymbol{k}})$, onde " θ " define o ângulo de rotação, e $\hat{\mathbf{r}}$, o vetor unitário do eixo de rotação. A partir disto pode ser realizado o produto sanduíche dado em (9), onde $\boldsymbol{q}^{-1}$ denota o recíproco do quaternião. Nota-se que neste caso, como os quaterniões de rotação são unitários, o recíproco é igual ao conjugado complexo. O quaternião resultante será equivalente a rotação de $\boldsymbol{q}_p$ em volta de $\hat{\mathbf{r}}$.

$$\boldsymbol{q}_{pRotacionado} = \boldsymbol{q}_R\boldsymbol{q}_p\boldsymbol{q}_R^{-1},\ \boldsymbol{q}_R,\boldsymbol{q}_p \in \mathbb{H} \tag{9}$$

Contudo, a multiplicação por quaterniões não é conveniente em um conjunto de produto de matrizes. Por conta disto, (9) pode ser decomposta no produto de Hamilton, e isolado os elementos para uma matriz três por três que multiplica um vetor de posição de três elementos. Como é necessário a manutenção do "w", está matriz é substituída a uma matriz identidade de tamanho quatro. A transformação resultante pode ser visto em (10).

$$\begin{gathered}\boldsymbol{q}_R = cos(\frac{\theta}{2}) + sen(\frac{\theta}{2})(r_x\hat{\boldsymbol{i}} + r_y\hat{\boldsymbol{j}} + r_z\hat{\boldsymbol{k}}) = q_s + q_x\hat{\boldsymbol{i}} + q_y\hat{\boldsymbol{j}} + q_z\hat{\boldsymbol{k}}, \boldsymbol{q}_R \in \mathbb{H} \\ \begin{bmatrix} x_{Rotacionado} \\ y_{Rotacionado} \\ z_{Rotacionado} \\ w \end{bmatrix} = \begin{bmatrix} 1-2(q_y^2+q_z^2) & 2(q_xq_y - q_zq_s) & 2(q_xq_z + q_yq_s) & 0 \\ 2(q_xq_y + q_zq_s) & 1-2(q_x^2+q_z^2) & 2(q_yq_z - q_xq_s) & 0 \\ 2(q_xq_z - q_yq_s) & 2(q_yq_z + q_xq_s) & 1-2(q_x^2+q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}\end{gathered} \tag{10}$$

A rotação é aplicada sempre ao redor do ponto (0,0,0). Para rotacionar o objeto ao redor de outro centro, é necessário realizar uma translação prévia das posições, e depois realizar uma translação de mesmo valor, de sentido oposto, para retomar o modelo a posição original.

#### 2.2.2.1.3 *Translação*

A translação do modelo é comumente realizada após a escala do objeto, para se considerar os valores em medidas do mundo. Esta translação é realizada adicionando valores ao vetor posição dos vértices. Esta matriz pode ser construída pela soma do vetor coluna ($T_x$,$T_y$,$T_z$) à ultima coluna da matriz identidade. O resultado pode ser visto em (11). Percebe-se como os valores de traslação são multiplicados pelo "w", por conta disto, é geralmente definido-o como 1, quando se deseja transladar o vetor. Em situações onde o vetor deve se manter na origem, como no caso dos vetores normais, o valor de "w" é definido como 0.

$$\begin{bmatrix} x + w\cdot T_x \\ y + w\cdot T_y \\ z + w\cdot T_z \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \tag{11}$$

#### 2.2.2.2 Matriz de visualização

A matriz de visualização é composta da mesma forma que a do modelo, contudo, é utilizada para a câmera. Em uma cena cada objeto utilizará a sua própria matriz de modelo, contudo, para deslocamento da câmera não é necessário o deslocamento de todos estes. Esta operação envolveria recalcular todas as matrizes de modelo. Para simplificação, é acrescentado uma segunda matriz de transformação, contudo, para a câmera, que será aplicado em todos os objetos da cena. Com isto, apenas uma matriz necessita ser recalculada.

Para esta operação deve-se atentar que o movimento da câmera é em sentido oposto aos modelos, então deve se calcular o deslocamento e ângulo de rotação como negativos, além da escala ser o inverso. Fora este ponto, as equações são as mesmas das vistas na Subsubseção 2.2.2.1.

#### 2.2.2.3 Matriz de projeção

Para a visualização tridimensional de objetos em um *display* 2D, é necessário certas transformações nestes planos que consideram as particularidades da visão. As formas reais de mapeamento 3D para 2D são restritas aos olhos humanos e a câmeras fotográficas, e em ambos estes casos a visualização é feita de uma forma cônica, com os feixes de fótons convergindo em um elemento central. Em uma renderização computacional, de modo a manter o realismo, deve ser simulado este efeito de visualização pontual.

Na prática, esta transformação converterá elementos de um volume piramidal a um cubo, que possui altura e largura nas dimensões da tela. A profundidade será utilizada apenas para verificação de sobreposição de polígonos, não havendo possibilidade de utilização para o *display*.

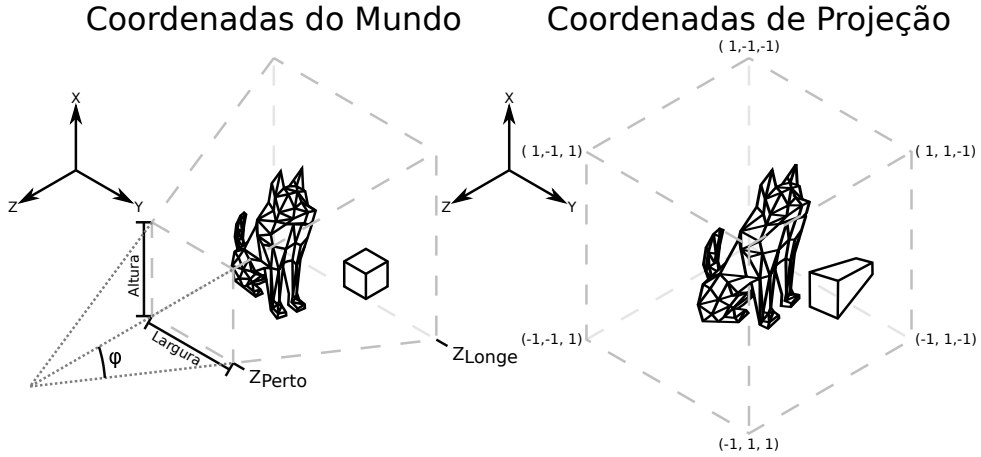
A matriz de projeção consiste do campo de visão, e da distância de corte. Com estes valores é criado um cone de projeção que é transformado para um cubo pela matriz. A aplicação desta matriz de projeção pode ser vista na Figura 10.

##### 2.2.2.3.1 *Field of view*

O campo de visão, ou *Field Of View* (FOV) em inglês, é o que gera a sensação de profundidade na imagem, porque gera uma distorção do tamanho dos objetos em relação a distância. Com um campo de projeção na forma de um cubo, ou seja, linhas paralelas de visão, a renderização é dada no formato isométrico, com isto os polígonos possuem o mesmo tamanho independente de suas distâncias. Modelos isométricos são úteis para a renderização de peças físicas, para permitir uma melhor análise entre as dimensões, contudo, para reprodução realística de modelos, é necessário utilizar linhas centradas à câmera pontual.

O FOV é dado como ângulo, e neste caso será dado como "$\varphi$". Para conversão de um ponto $\vec{p}$ do plano do mundo para o ponto de visão as coordenadas em "x" e "y" devem ser convergidas ao centro a medida que "z" diminui, como visto na Figura 10. Estes valores são

Figura 10 – Transformação da matriz de projeção




Fonte: Autoria Própria (2022)

mapeados ao plano de corte frontal, denominado de $z_{Perto}$, e o resultante pode ser dado pela análise dos triângulos, o que resulta em (12). Nota-se que neste caso o "z" é negativo devido a convenção utilizada para os eixos.

$$(x,y)_{Projetado} = \frac{1}{-z \cdot tan(\frac{\varphi}{2})}(x,y)_{Mundo} \tag{12}$$

Nota-se que o termo "z" não pode ser diretamente aplicado a matriz, porque não há uma simples operação matemática para divisão pela multiplicação matricial. Este valor será atribuído ao "w", e a divisão será realizada no momento de rasterização.

#### 2.2.2.3.2 *Distância de corte*

Em uma projeção ideal, não existe distância limite dos objetos exceto, fora dos limites do ângulo de alcance da câmera. Na prática, são utilizados planos limites de distância, para o início e fim da variável "z" a ser projetada.

O limite minimo é dado pelo plano próximo e denominado de $z_{Perto}$. Este se estende a uma distância da câmera onde nenhum polígono será renderizado. Objetos muito próximos à câmera acabarão por ocupar grande parte do espaço da tela, então não são viáveis de renderização. Como o eixo "z" se distancia da câmera negativamente, este plano é presente no valor de $z = -z_{Perto}$.

O limite máximo é dado pelo plano distante e denominado de $z_{Longe}$. Este se inicia a partir de uma certa distância da câmera, a partir da onde, nenhum polígono será renderizado. Na realidade isto ocorre naturalmente devido a partículas atmosféricas ofuscarem gradativamente objetos distantes. Contudo, neste caso será realizado um corte abrupto dos polígonos, porque qualquer polígono além deste plano será mapeado no exterior do cubo de projeção e será descartado. Este plano se encontra em $z = -z_{Longe}$. Existem certas técnicas que ofuscam objetos em função da distância, contudo não serão aqui abordadas.

Deve-se atentar de que ambos os limites devem estar em valores negativos, porque, neste caso, a câmera é situada em (0,0,0) e voltada para $-z$. Caso maior do que zero, o objeto será espelhado nos eixos "x" e "y", como visto em (12).

Para o mapeamento do eixo "z" para as coordenadas de projeção, é necessário o mapeamento destes planos para os valores limites de 1 e -1 do cubo de projeção. Neste caso será mapeado $-z_{Perto}$ para o limite de 1, e $-z_{Longe}$ para -1. Estes valores podem divergir dependendo do método utilizado para projeção, neste caso é considerado 1 enfrente a câmera, e menores valor de $z_{Projetado}$ representará um objeto mais distante. Também, como visto anteriormente, o vetor posição será dividido por "z" na fase de renderização. Esta operação pode ser realizada somente para "x" e "y", porém, certas arquiteturas de GPUs possuem capacidade de operação simultânea de vetores, principalmente de vetores de três posições, por conta disto, é considerado que "z" passará por uma compressão futura. Com isto, a equação resultante do mapeamento é dado em (13).

$$z_{Projetado} = \frac{1}{-z}\left(\frac{z(z_{Longe} + z_{Perto}) + 2 \cdot z_{Longe} z_{Perto}}{z_{Longe} - z_{Perto}}\right) \tag{13}$$

#### 2.2.2.3.3 *Cone de projeção*

Com estes termos pode-se projetar a matriz de conversão do cone de projeção. Percebe-se como as coordenadas do mundo relevantes se apresentam dentro de uma pirâmide retangular que são mapeados para um cubo. Como os *displays* geralmente apresentam dimensões retangulares, é necessário realizar uma compressão ou dilatação em um dos eixos "x" ou "y", pela relação de dimensão. Então a matriz completa de projeção é dado como em (14). Atentando-se ao termo "w" que será descartado nesta fase.

$$\begin{bmatrix} x_{Res} \\ y_{Res} \\ z_{Res} \\ -z \end{bmatrix} = \begin{bmatrix} \frac{1}{tan(\frac{\varphi}{2})}\frac{altura}{largura} & 0 & 0 & 0 \\ 0 & \frac{1}{tan(\frac{\varphi}{2})} & 0 & 0 \\ 0 & 0 & \frac{z_{Longe}+z_{Perto}}{z_{Longe}-z_{Perto}} & \frac{2 \cdot z_{Longe} z_{Perto}}{z_{Longe}-z_{Perto}} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \tag{14}$$

A partir destes valores é realizado a divisão do vetor por "w", e se obtém os valores projetados, como dado em (15).

$$(x,y,z,1)^T_{Projetado} = \frac{1}{w}(x,y,z,w)^T_{Res} = \frac{1}{-z}(x_{Res},y_{Res},z_{Res},-z)^T \tag{15}$$

As coordenadas de projeção se encontrarão dentro de um cubo de tamanho dois por dois por dois, centrado na origem. A partir desta conversão, é realizado corte dos polígonos que se apresentam fora desta região. Os polígonos exteriores não serão renderizados à tela podendo ser descartados a partir deste ponto. E, caso não haja cálculos físicos a serem influenciados por um modelo completamente fora da região, esse pode ser temporariamente descartado da cena, de modo a aprimorar a performance.

### 2.2.3 Rasterização

Após as matrizes de transformação, o objeto pode ser decomposto em conjuntos de seus respectivos vértices, de modo a formar uma face poligonal. Os vértices são armazenados na memória em um *buffer* de posições, de mesmo modo da textura, e normais. Estes elementos são armazenados de forma individual, de modo que necessite de informações adicionais para construção do polígono. Os dados utilizados para construção dos *vertexes* são dados por *buffers* de índices, que são compostos por vetores de números inteiros sem sinal, que indicam as posições de posição, textura, e normal a serem transferidas aos *shaders*.

Um polígono deve possuir pelo menos três *vertexes* para sua construção. A utilização de um ou dois *vertexes* pode ser utilizada para criação de pontos ou linhas, respectivamente, e serão utilizados como processo parcial para criação dos polígonos, como será visto na Subsubseção 2.2.3.1. De três a quatro *vertexes* é o recomendado, porque são geralmente os formatos providenciáveis pela API.

Com o polígono formado, após a etapa de transformação, o objeto pode ser rasterizado em tela. A primeira forma de rasterização eletrônica foi dada por 高柳健次郎 (1928), onde se foi projetado imagens em posições finitas na tela. Esta mesma técnica é utilizada para elementos fictícios tridimensionais. Os polígonos são projetados linha a linha em um *framebuffer*, que será então mostrado ao *display* na atualização da tela.

O termo rasterização vem do alemão "*das Raster*", que significa "grade". Isto se deve aos modelos serem posicionados a uma grade de *pixels*, que representa o *display*. Na fase de rasterização os limites "x" e "y" do cubo de projeção é convertido para os limites da tela. Com isto, valores de ponto flutuantes são discretizados para valores inteiros, criando um efeito de serrilhamento nos objetos. O efeito da rasterização pode ser visto na Figura 11. Certos monitores de CRT antigos tinham a capacidade de projeção vetorial, contudo, esta tecnologia não permitia uso de múltiplas cores. Por conta disto, e da popularização dos *displays* de LCD, a tecnologia de rasterização se cimentou como o de facto meio de renderização à tela.

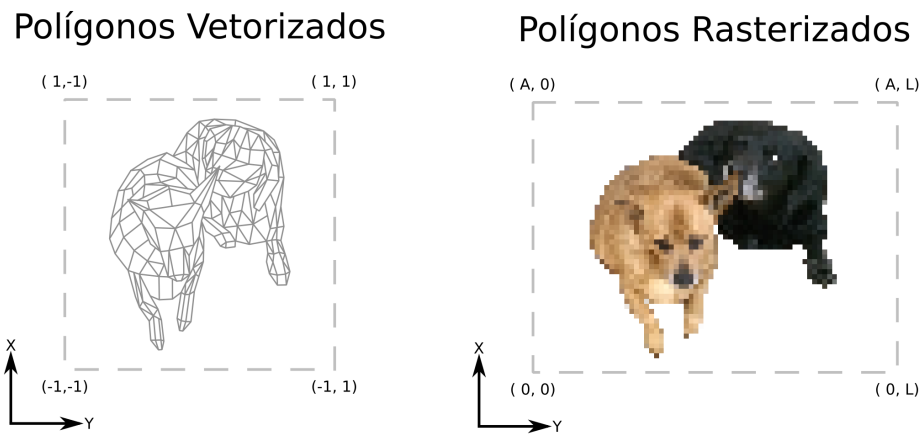
#### 2.2.3.1 Algoritmo de Bresenham

Os valores dos índices especificam também a ordem de renderização dos *vertexes*. Isto será utilizado para renderização de polígonos, porque estes são primeiramente rasterizados por suas bordas. De modo incremental as linhas limites dos polígonos são rasterizadas ao *framebuffer* de dois em dois *vertexes*.

Para ligação destes pontos certas operações matemáticas devem ser realizadas, para permitir a criação da linha, em formato de *pixels*, utilizando a posição dos dois *vertexes*. O método mais intuitivo é a ligação pela obtenção da equação da linha. Porém, este método é computacionalmente intensivo, porque requere o uso de divisão, para encontrar a angulação da reta, e posterior multiplicação para obtenção da linha em si, e isto em cada iteração.

Certos algoritmos podem ser utilizados para a obtenção da linha. Um dos mais

Figura 11 – Rasterização de polígonos à tela

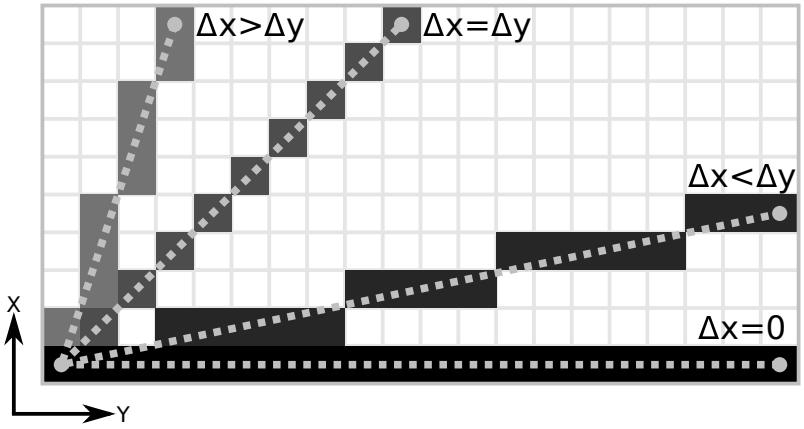



Fonte: Autoria Própria (2022)

utilizados é o criado por Bresenham (1965) onde é utilizado apenas uma sucessiva série de somas e condicionais. Multiplicações tendem a levar mais ciclos para completar do que soma, e necessitam de mais transistores para operarem. Por conta disto, é viável a utilização deste método, denominado de algoritmo de Bresenham.

O algoritmo de Bresenham consiste em incrementar um dos eixos unitariamente, e incrementar uma variável de erro, que causara o aumento do eixo perpendicular a partir de certo valor. Com isto, uma série de *pixels* pode ser traçada aproximando os pontos da linha contínua. O efeito desta aproximação e aplicação do algoritmo é visto na Figura 12. Percebe-se que os valores são medidos a partir do meio do *pixel*, isto será relevante para definir qual as coordenadas que devem ser preenchidas.

Figura 12 – Retas construídas pelo algoritmo de Bresenham




Fonte: Autoria Própria (2022)

Para a construção da reta primeiramente é discretizado o valor das posições para coordenadas de tela, utilizando uma função de arredondamento ou truncamento. A partir disto

são obtidos os valores de incremento, sendo $\Delta x$ e $\Delta y$, e o valor de incremento do erro, dado por $\Delta er$. Para $\Delta x > \Delta y$, o cálculo destes valores estão presentes em (16). Estes termos são calculados somente uma vez por reta.

$$\begin{gathered} \Delta x = x_1 - x_0 \quad \Delta y = y_1 - y_0 \quad \Delta er = \frac{\Delta x}{\Delta y} \\ \Delta y,\Delta x,x_0,x_1,y_0,y_1 \in \mathbb{N},\ \Delta er \in \mathbb{R} \end{gathered} \tag{16}$$

A partir dos termos iniciais, se inicia o laço para preenchimento dos *pixels*. Para uma linha de $\Delta x > \Delta y$, o termo em "x" é acrescentado de forma unitária em cada iteração, enquanto o termo em "y" se mantém constante. A cada iteração a variável de erro, responsável pelo acompanhamento da diversão, é incrementada em $\Delta er$. A partir do instante em que este valor ultrapassa o limite de 0,5, o termo em "y" é incrementado unitariamente. Quando ocorre este incremento é importante que seja subtraído 1 da variável de erro. Este ciclo se repete até o limite das coordenadas.

Para retas onde $\Delta x < \Delta y$ o procedimento é similar, contudo, nesta progressão o termo em "y" é incrementado em cada iteração, enquanto o termo em "x", somente no estouro da variável de erro. E neste caso a variável de erro é invertida de modo $\Delta er = \Delta x/\Delta y$. Em casos de inclinações negativas, o incremento é de -1. Também caso o ponto de início no eixo de incremento seja em uma posição superior ao de término, as posições de início e fim devem ser invertidas. A implementação destas e demais condições pode ser visto no Algoritmo 2.

Em casos especiais como $\Delta x = 0$, e $\Delta y = 0$, o algoritmo pode divergir para iterações simplificadas, de modo que apenas um termo necessitará ser incrementado.

É importante de notar que o algoritmo de Bresenham cria linhas serrilhadas, porque o seu preenchimento se dá somente somente em um único *pixel* por iteração. Em telas de baixa resolução este efeito se torna aparente. E esta forma binária de lidar com o preenchimento cria efeitos de *aliasing* em visualizações de menor resolução que a da própria imagem. Para lidar com isto outras técnicas podem ser utilizadas como a de Wu (1991). Porém, a fim de manter a simplicidade, não serão aqui detalhadas.

No caso dado no Algoritmo 2, a linha será preenchida com uma constante dada pelo valor de preenchimento. Contudo, isto somente é utilizado caso se deseje a criação de objetos monocromáticos como na criação de *wireframes*. Na prática é assimilado um valor para cada ponto da reta, de modo que durante as iterações do algoritmo de Bresenham, o valor seja interpolado, incrementando em um $\Delta valor$ durante o preenchimento. Nota-se que quando ocorrer inversão das coordenadas de posição, este valor deve ser também invertido.

Este valor de preenchimento não é necessariamente a cor da reta, podendo ser simplesmente uma variável a incrementar durante a progressão da linha. São utilizados também os valores de profundidade, obtidos do termo em "z" da posição do vértice, e os termos de UV, para mapeamento da textura. Estes valores serão interpolados em *buffers* individuais de modo a que se definam os valores para as bordas do polígono a ser preenchido.

**Algorithme 2** Aplicação do algoritmo de Bresenham

**Input** Coordenadas da linha: $x_0, y_0, x_1, y_1$; Valor de preenchimento: $valor$
1: $\Delta x \leftarrow x_1 - x_0$; $\Delta y \leftarrow y_1 - y_0$;
2: **if** $\Delta y < \Delta x$ **then**
3: **if** $y_0 > y_1$ **then**
4: **Swap**($x_0$,$x_1$); **Swap**($y_0$,$y_1$);
5: **goto** *Linha 1*;
6: $\Delta er \leftarrow \|\Delta y/\Delta x\|$;
7: $sinal \leftarrow$ **if** $\Delta y > 0$ **then** $1$ **else** $-1$;
8: **for** $x \in [x_0, \cdots, x_1]$; $y \leftarrow y_0$; $erro \leftarrow 0$ **do**
9: **Buffer**($x$,$y$)$\leftarrow valor$; $erro \leftarrow erro + \Delta er$;
10: **if** $erro > 0{,}5$ **then**
11: $y \leftarrow y + sinal$; $erro \leftarrow erro - 1$;
12: **else**
13: **if** $y_0 > y_1$ **then**
14: **Swap**($x_0$,$x_1$); **Swap**($y_0$,$y_1$);
15: **goto** *Linha 1*;
16: $\Delta er \leftarrow \|\Delta x/\Delta y\|$;
17: $sinal \leftarrow$ **if** $\Delta x > 0$ **then** $1$ **else** $-1$;
18: **for** $y \in [y_0, \cdots, y_1]$; $x \leftarrow x_0$; $erro \leftarrow 0$ **do**
19: **Buffer**($x$,$y$)$\leftarrow valor$; $erro \leftarrow erro + \Delta er$;
20: **if** $erro > 0{,}5$ **then**
21: $x \leftarrow x + sinal$; $erro \leftarrow erro - 1$;

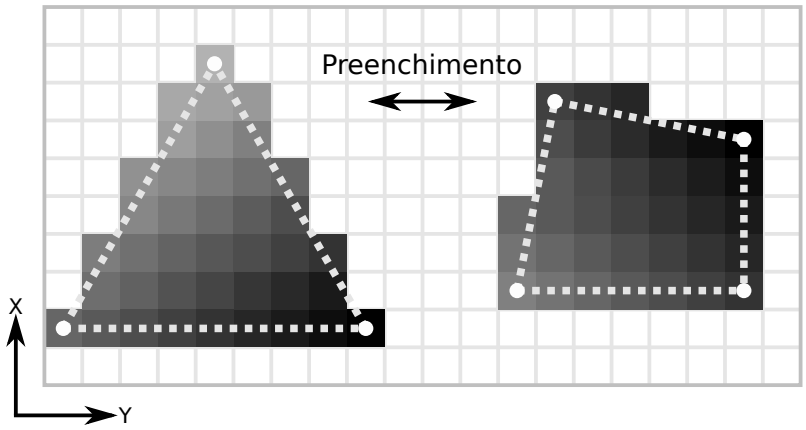
#### 2.2.3.2 Algoritmo scan line

A definição das bordas dos polígonos são de certo modo suficiente para definir uma estrutura, contudo, sem o preenchimento não consegue se transmitir volume e detalhe do objeto. Por conta disto certas técnicas foram desenvolvidas para preenchimento, sendo uma das primeiras a lidar com as diferenças de tonalidade de modo eficiente, proposta por Bouknight (1969). Este algoritmo se assemelha ao processo de rasterização por escaneamento de linhas utilizada pelos televisores de CRT, por isto pode ser encontrado na literatura pela nomeação de algoritmo *scan line*. O efeito deste algoritmo pode ser visto na Figura 13. Neste caso é criado um gradiente no eixo "y" entre os valores de borda criados pelo algoritmo de Bresenham.

A ideia do algoritmo *scan line* é a de obter uma região limítrofe, dentro de onde as bordas do polígono se encontram, e ir de linha a linha escaneando os *pixels* do *buffer*. Quando se encontra um valor de borda preenchido, ele mantém o preenchimento até encontrar com a próxima borda. Quando o polígono é convexo, o preenchimento é finalizado neste ponto, contudo, em polígonos côncavos, o algoritmo continua a progressão até colisão com a próxima borda, a partir deste ponto retoma o preenchimento, e segue desta forma alternada até as coordenadas máximas dos pontos. Quando ocorre preenchimento em sequência na reta de Bresenham, deve ser desconsiderado este *pixel*, porque não representa uma nova borda.

Caso os polígonos sejam de três ou quatro pontos, em nenhum caso haverá polígonos

Figura 13 – Preenchimento pelo algoritmo *scan line*




Fonte: Autoria Própria (2022)

convexos na renderização. Para três pontos isto se dá de modo trivial. E para quatro pontos, isto se da de modo intuitivo, como os pontos da face do modelo estarão sempre em um mesmo plano, de modo a formar um polígono plano, não há como este ser côncavo. Como a face é plana, todos os pontos devem ser ligáveis por uma reta, incluindo pontos opostos da face, e isto deve se manter independente da posição dos pontos, e como a rotação produz uma translação dos pontos na tela bidimensional, independente também de rotação. Por conta disto, a única instância em que ocorre intersecção de linhas pelas bordas em uma face de quatro pontos é no caso de uma visão perpendicular do polígono.

Inicialmente no preenchimento de uma linha, os valores de preenchimento das bordas são desconhecidos, por conta disto, o algoritmo deve escanear duas vezes a mesma linha. Na primeira passada se obtém os valores de preenchimento das bordas e os limites, a partir disto é possível definir os limites de operação e valor de incremento. Na segunda passada o procedimento é similar ao algoritmo de Bresenham, contudo sem incremento em eixos perpendiculares. Este processo se repete até todo o polígono ser preenchido. A representação deste processo pode ser visto no Algoritmo 3. Neste caso o escaneamento é no eixo "y", valores levemente diferentes podem ser obtidos caso se invertam os eixos, porém, caso sejam iguais os valores de preenchimento para os mesmos vértices, esta diferença pode ser desconsiderada.

## 2.3 Segmentação de memória

Os computadores são segmentados em seções para operações e manuseio de dados, representado pelas CPU, e organizado de acordo com sua respectiva ISA. Contudo, as informações que o processador utiliza estão presentes em compartimentos específicos para este fim denominados de memória. Este componente do computador pode ser tratado como uma própria seção, necessitando de suas próprias considerações, e com seu próprio ramo de estudo. Em aplicações gerais de programação este termo pode ser visto como uma extensão da CPU,

**Algorithme 3** Aplicação do algoritmo *scan line*

**Input** Limites do polígono: $x_{min}, y_{min}, x_{max}, y_{max}$
1: **for** $x \in [x_{min}, \cdots, x_{max}]$ **do**
2: **for** $y \in [y_{min}, \cdots, y_{max}]$; $y_{inicial} \leftarrow NaN$; $y_{final} \leftarrow NaN$ **do**
3: **if Buffer**$(x,y) \neq 0$ **then**
4: **if** $(y_{inicial} = NaN) \vee (y_{inicial} = y - 1)$ **then**
5: $y_{inicial} \leftarrow x$;
6: **else**
7: $y_{final} \leftarrow y$;
8: **if** $(y_{inicial} \neq NaN) \wedge (y_{final} \neq NaN)$ **then**
9: $valor \leftarrow$**Buffer**$(x, y_{inicial})$;
10: $\Delta valor \leftarrow$ (**Buffer**$(x, y_{final}) - valor_{inicial})/(y_{final} - y_{inicial})$;
11: **for** $y \in ]y_{inicial}, \cdots, y_{final}[$ **do**
12: $valor \leftarrow valor + \Delta valor$; **Buffer**$(x,y) \leftarrow valor$;

sendo uma região para armazenamento das informações trabalhadas. Contudo, uma análise no escopo de instruções individuais se torna um ponto que possui considerações específicas, que são geralmente filtradas na tradução pelo compilador. Sendo assim, um estudo deste ramo agrega ao entendimento da construção de algoritmos em linguagem assembly.

Este ramo sendo sempre presente na evolução dos processadores, contudo, com uma vasta gama de tecnologias, visando cada uma o aprimoramento de alguns pontos específicos. As memórias podem ser classificadas de acordo com a sua capacidade de armazenamento por custo, taxa de transferência de dados, tempo de acesso, método de acesso, volatilidade, e capacidade de sobrescrever informações.

Devido a diversidade de tecnologias, não há uma evolução linear neste ramo de tecnologia, contudo, alguns pontos podem ser enfatizados. Um dos marcos deste tipo de tecnologia é o armazenamento em núcleos magnéticos, desenvolvido por Forrester (1951). Neste caso é utilizado um núcleo toroidal com alta retenção de densidade de fluxo, criando assim uma curva de histerese de formato retangular. Deste modo um bit pode ser armazenado no magnetismo residual.

O desenvolvimento de armazenamento magnético continuou na década de 50, onde a empresa IBM desenvolveu o armazenamento em fitas magnéticas, onde um filme plástico é revestido com material magnetizável (BRADSHAW; SCHROEDER, 2003). Com isto, vasta quantidades de informações puderam ser armazenadas em dispositivos relativamente pequenos e de baixo custo. Contudo, a velocidade de leitura e escrita era baixa, isto pode ser atribuído em parte a leitura ser sequencial, causando que saltos nos endereços dos dados necessitassem da passagem por todos os bits em seu meio. Este se manteve um problema na evolução desta área, onde a capacidade de armazenamento se manteve inversamente proporcional a velocidade de leitura e escrita.

Na década de 60, foi desenvolvido por Robert Dennard uma tecnologia de armazenamento de informações no campo elétrico de transistores de efeito de campo, ou *Field-Effect*

*Transistors* (FETs) em inglês. Permitindo assim o armazenamento na velocidade dos transistores, ou seja, em tempo similar ao processador. Além disto, o acesso é possível de forma aleatória, sendo possível o salto entre endereços distintos pelo fato das informações serem armazenadas em grades de FETs. Esta tecnologia predominou no uso das memórias RAM dinâmicas, ou *Dynamic RAM* (DRAM) em inglês (KLEIN, 2016). Esta tecnologia permite um acesso rápido aos dados, podendo transmitir informações com certo paralelismo, contudo o seu armazenamento é volátil, e seu custo por bit é mais caro do que outras tecnologias.

Novas técnicas de armazenamento não volátil estático foram inseridas no mercado. Uma destas tecnologias é a memória flash, desenvolvida por Masuoka et al. (1984), e utiliza um dispositivo FET com um *gate* intermediário que pode ser carregado com elétrons, criando uma resistência ao campo elétrico produzido pelo *gate* principal dependendo do valor da carga. Além disto, o carregamento da camada intermediária se mantém mesmo com a desenergização do dispositivo, permitindo o uso desta tecnologia para armazenamento externo ao processador.

Por conta dos diferentes custos e desempenhos, os computadores modernos apresentam diversos métodos de armazenamento. Cada dado é armazenado em um nível dependendo de sua importância para o processador. As informações são inicialmente armazenadas nas memórias externas, de baixo custo. Quando é visto a necessidade, suas informações são deslocadas para a memória de trabalho, onde os seus endereços são disponibilizados para requisições do processador. Após trabalho nestas informações pode haver a movimentação no sentido reverso para a memória de baixo custo, ou apenas são desalocados os endereços e sobrescritas por novos dados.

As memórias de trabalho são geralmente predominadas pela DRAM, contudo, novas tecnologias atribuem regiões no interior da CPU para armazenamento de dados utilizados, criando uma camada adicional. Esta região é denominada de memória *cache*, e durante o processamento, quando ocorre a requisição de acesso para um certo endereço da memória, seu valor é buscado na *cache* primariamente. Caso o valor não seja encontrado, ocorre uma requisição para a memória principal, onde geralmente é criado também uma cópia do respectivo bloco em que o endereço se contém para a memória *cache*.

Contudo, caso o processador possua o endereço dos dispositivos de armazenamento de forma direta, cada cópia à memória *cache* necessitaria de alteração nos endereços definidos no próprio algoritmo que esta sendo executado. Isto se torna custoso para a CPU. Por conta disto, o endereço requisitado é fictício, ou seja, não é equivalente ao endereço na memória principal ou *cache*. Para isto é realizado um mapeamento de endereços, onde um dispositivo independente do setor de processamento mantém uma tabela de tradução entre o processador e os dispositivos de armazenamento. Caso o mapeamento seja de 1 byte a 1, a tabela necessitaria de um tamanho similar a soma de todas as memórias de trabalho. Por conta disto, os dados são armazenados em blocos, podendo um arquivo ser segmentado em diversas regiões não congruentes. Por conta disto, as tabelas de mapeamento são relações somente entre os endereços de inícios de blocos de memória, permitindo assim ser de um tamanho reduzido, e são armazenadas,

geralmente na própria memória *cache* ou principal do dispositivo. Endereços superiores ao início de um bloco podem ser mapeados de forma linear a partir do endereço definido na tabela. Para esta tecnologia são comumente utilizado RAM estáticas, ou *Static RAM* (SRAM) em inglês, devido à sua rápida velocidade de leitura e escrita.

Além disto, as próprias memórias de armazenamento externas, geralmente, possuem uma segmentação denominada de *swap* para dados comumente requisitados, criando assim mais um nível de abstração. Porém, estas memórias de acesso em massa não possuem acesso direto para o processador, necessitando de uma requisição específica para um dispositivo de entrada/saída responsável por esta comunicação.

A grande variedade de tecnologias neste ramo torna difícil o estudo de números específicos, contudo, uma análise geral destes parâmetros de performance podem ser vistos na Tabela 4.

Tabela 4 – Hierarquias de memórias em um computador em 2015

| **Memória** | **Tecnologia** | **Custo por GB** | **Tempo de acesso** | **Largura de banda** |
|---|---|---|---|---|
| *Cache* | SRAM | $5.000,00 | 0,5 ns | 25+ GB/s |
| Principal | DRAM | $7,00 | 10-50 ns | 10 GB/s |
| Externa | SSD | $0,40 | 20.000 ns | 0,5 GB/s |
| | HDD | $0,05 | 5.000.000 ns | 0,75 GB/s |

Fonte: (HARRIS; HARRIS, 2015)

Analisando o tempo de acesso pode-se notar que acesso a dados na memória podem ser tarefas ociosas para o processador. Dependendo do nível, o acesso pode ser em poucos ciclos de *clock*, contudo, a medida que se passam as camadas, o processador pode esperar os dados por milhões de ciclos. Em fato, caso não existam divergências para ocupar a CPU com outras atividades de processamento, a espera por acesso de dados pode ocupar a maior porcentagem do tempo do núcleo. Por conta disto são implementados diversos algoritmos nos próprios dispositivos de gestão de memória para prever e manter dados importantes nos níveis mais baixos da memória, sobrescrevendo informações de menor prioridade. Contudo, apesar das otimizações, acesso as memórias mais lentas tendem a ocorrer, porém com frequência reduzida. Estes sendo um dos motivos para implementação de maior número de registradores na arquitetura RISC, como definido por Patterson e Sequin (1982), permitindo um acesso em tempo inferior a 1 ciclo de *clock* aos dados nesta região.

O desacoplamento dos dispositivos de memória para a CPU acaba por permitir também uma certa independência de operação. Por conta disto, existem técnicas de acesso direto à memória, ou *Direct Memory Access* (DMA) em inglês, onde dispositivos externos a unidade de processamento conseguem acessar informações sem necessidade de transferência pela CPU. Isto pode ser realizado quando o barramento de dados é compartilhado entre a CPU e demais periféricos. Isto acaba permitindo uma maior otimização do processamento, evitando a ocupação do núcleo com tarefas de translação de dados.

Este atraso na leitura reflete somente em uma perda de performance, caso processado o algoritmo em um único núcleo. Porém, o mesmo não pode ser garantindo em operações envolvendo paralelismo de acessos na mesma memória, podendo ocorrer problemas de adicionais de sincronização.

### 2.3.1 Ordenação de memória fraca RVWMO

Durante o processamento de um algoritmo em um único núcleo, caso o programa seja executado diversas vezes em instantes diferentes, o resultado das alterações na memória será sempre o mesmo. Isto se deve ao fato do acesso exclusivo a memória ao processo. Por conta disto as alterações realizadas na memória, mesmo que levem diferentes tempos de acesso entre iterações, é sempre realizada na sequência em que o algoritmo determina. Isto não excluí a alteração das ordem das instruções, em fato, processadores modernos modificam a ordem do algoritmo durante o *runtime*. Permitindo assim a ocupação do tempo de processamento durante instruções multicíclicas. Contudo, é importante enfatizar que esta operação somente é realizada sobre registradores e endereços de memória independentes, ou seja, endereços de leitura de um dado específico é realizado sempre antes de uma operação de escrita. Este tipo de otimização é denominada de execução fora de ordem, ou Out-of-Order (OoO) em inglês.

Contudo, em operações de múltiplos núcleos paralelos não há garantia da ordem de acesso a um endereço. Neste caso os registradores são individuais para cada núcleo, contudo, a memória é geralmente compartilhada. O primeiro problema resulta do tempo de acesso aos dados, ocorrendo também em execução sequencial do algoritmo. Neste caso a requisição de leitura de um núcleo pode ocorrer durante uma operação de escrita de outro, por conta disto o resultado se torna incerto, podendo ocorrer a escrita antes ou depois da leitura. E este problema se expande quando se considera tempos de leitura diferentes de escrita. Ou uma requisição para escrita pode ser realizada por diversos núcleos ao mesmo instante, criando uma incerteza do valor corretamente armazenado.

O segundo problema é criado devido a execução OoO. Em operações sequências é possível ter uma estimação da ordem em que cada núcleo se comunica com a memória, permitindo uma previsão de pontos de colisão. Contudo, em operações OoO, os núcleos podem agrupar leituras e escritas em ordens específicas, a fim de otimizar a utilização do barramento de dados.

Um exemplo de ambiguidade nos valores do endereço pode ser visto no Algoritmo 4.

Em execução OoO a escrita e leitura da memória podem ocorrer em ordens diferentes no núcleo 1, de mesmo modo que as duas primeiras operações do núcleo 2. No momento em que o valor for sobrescrever `rsA`, o núcleo deve ter calculado previamente `rsB`, e `rsC`. Porém, não há como saber se a se a leitura ou escrita do Endereço A ocorrerá primeiro, devido a estarem em núcleos diferentes. Neste caso não há como estimar o valor de `rsA` que será repassado à função no núcleo 1. Esta incerteza tende a aumentar ao incrementar o paralelismo do processo.

**Algorithme 4** Exemplo de ambiguidade na ordenação da memória

**Núcleo 1:**
1: **Escrita**(*Endereço B*) ←rsB;
2: rsA ←**Leitura**(*Endereço A*);
3: **Função**(rsA);
**Núcleo 2:**
4: rsB ←**Operação Aritmética**(rsA,rsB);
5: rsC ←**Operação Lógica**(rsA,rsD);
6: rsA ←**Operação Aritmética**(rsB,rsC);
7: **Escrita**(*Endereço A*) ←rsA;

Este tema de estudos é denominado de ordenação de memória. Atualmente cada ISA aplica sua própria técnica de resolução, podendo variar entre versões. Contudo, um dos métodos mais diretos de resolução é a técnica de consistência sequencial desenvolvida por Lamport (1979). Neste caso não é utilizado execução OoO, e o algoritmo define uma ordem global entre todos os *threads* de sequência de instruções. Isto traz o beneficio da consistência, porém a performance de processamento é reduzida.

Técnicas similares a consistência sequencial, que buscam o rigor na ordem de acesso, são denominadas de "ordenações fortes". Em contrastes, "ordenações fracas" representam um sistema com rigidez reduzida para as ordens de acesso a memória, permitindo um nível de sincronização entre núcleos, porém com capacidade de execução OoO nos *threads* (ADVE; HILL, 1990).

O RISC-V apresenta especificações para implementação do modelo de memória, se baseando na ordenação de memória fraca, com sigla RVWMO. Este modelo tende ser de modo geral relaxado, contudo, existem diversas regras de ordenação. As instruções, de modo geral, não são influenciadas por esta especificação, com exceção de instruções de leitura e escrita, atômicas, e *fences*. O conjunto base do RISC-V possui uma instrução de *fence*, e as instruções atômicas são providas pela extensão *A* (WATERMAN; ASANOVIĆ, 2021).

O RVWMO permite a execução de um algoritmo desde que sejam cumpridos três axiomas. O primeiro é o de valor carregado, onde uma leitura de um endereço deve retornar o valor previamente escrito pelo último armazenamento na ordem global ou na ordem do programa. Isto significa que não é necessário uma consistência global entre os *harts*, mas é necessário respeitar a ordem de acessos dita pelo programa.

O segundo é o axioma de atomicidade, onde no caso de utilização de instruções de carregamento e armazenamento atômicas, `lr` e `sc` respectivamente, e uma instrução de armazenamento intermediária no mesmo endereço do carregamento, o armazenamento atômico deve ser realizado após o intermediário, e não deve ocorrer sobrescrita por outro *hart* neste meio tempo. Isto garante que o valor desejado seja armazenado, impedindo execução OoO, e *race conditions*.

E o terceiro axioma é o do progresso, que dita que operações de memórias após um *loop* infinito devem se manter nesta ordem. Impedindo assim acesso à memória durante

períodos imprevisíveis de estado.

Além disto existem regras específicas para diversas condições de operação, sempre respeitando os três axiomas. Em geral, as operações de acesso a memória com dependência não devem ser alteradas em ordem, mesmo esta sendo de forma indireta por registradores intermediários. Operações atômicas garantem a consistência entre acessos, realizada pela requisição de um endereço por um *acquire*, até que um *release* seja estabelecido, contudo, isto também significa que requisições de outros *harts* podem retornar falhas, devendo se atentar a isto durante a programação. E requisições explicitas de sincronização pelo meio da instrução `fence`.

As instruções atômicas mantêm um endereço da memória reservado para um único *hart*. Isto é realizado pelo meio de bits de *acquire* e *release* definidos em bits na própria instrução. Para acesso à memória de forma reservada se utiliza das instruções `lr` e `sc`, contudo, além disto, há formas de realizar simples operações de forma direta. As instruções com início de `amo` são utilizadas para operações aritméticas e lógicas simples. Nos parâmetros são definidos os registradores de operação, de retorno, e o endereço de memória, contudo, o valor resultante é replicado no registrador de retorno e no endereço de memória do argumento, permitindo uma leitura e escrita na mesma instrução. Neste caso o *acquire* é requisitada na mesma instrução do *release*, porém, a reserva na memória dura durante o tempo de processamento da instrução.

A sincronização por meio de um `fence` permite a fixação da ordenação global em um ponto do programa específico. No momento em que esta instrução for encontrada no código, o *hart* não realizará modificações na ordem de instruções em lados opostos deste ponto. Contudo, deve ser especificado os limites desta restrição, podendo ser *read*, *write*, ou *read-write* em seus parâmetros, sendo definidos de forma separada para os predecessores, e sucessores. Além disto, todas as operações dentro do limite ditado no parâmetro serão finalizadas antes da continuação dos acessos. Esta sincronização é realizada na ordem global, por conta disto todos os *harts* deveram realizar esperar as realizações das operações requisitadas na memória para prosseguirem. A sincronização local de *harts* pode ser realizada por meio da instrução *fence.i*, provida na extensão *Zifence*.

# 3 METODOLOGIA

Para se obter os objetivos buscados, é necessário formalizar as ferramentas e tecnologias a serem utilizadas. Este capítulo não se refere ao algoritmo, sendo relevante somente durante as etapas de desenvolvimento, contudo, certas ordens de configurações referentes aos protocolos são detalhadas, de modo que sejam implementadas no código.

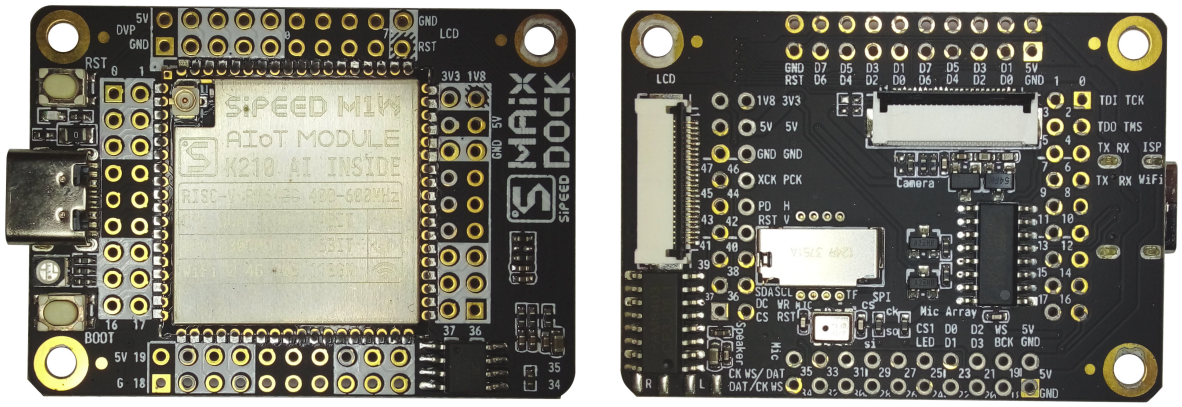
A ordem deste capítulo segue do hardware ao software. Iniciando com o detalhamento do hardware utilizado, com os detalhes do processador, e os protocolos de comunicação. E seguindo com o detalhamento do manuseio para armazenamento dos dados, e detalhamento dos arquivos a serem utilizados.

## 3.1 Hardware

Devido a busca da utilização da ISA do RISC-V, o dispositivo utilizado para o desenvolvimento deve possuir esta arquitetura de forma intrínseca ou ser capaz de simulá-la. Devido ao recente surgimento do RISC-V, a utilização desta não é ampla, contudo, existem implementações a disposição no mercado.

A placa de desenvolvimento Maix Dock, desenvolvida pela empresa Sipeed (2019b), possui os requisitos necessários, podendo ser vista na Figura 14. O Maix Dock possui alguns itens auxiliares, sendo os de interesse para este trabalho o conector para um *display* LCD de transistor de película fina, ou *thin-film-transistor* (TFT) em inglês, e uma entrada para um cartão micro SD. Possuindo também o chip de comunicação USB-Serial CH340, utilizado para comunicação e programação do microcontrolador pelo computador.

Figura 14 – Placa de desenvolvimento Maix Dock




Fonte: Autoria Própria (2022)

Em sua parte superiora, é posicionado o módulo M1W, desenvolvido também pela

Sipeed (2019a). Este módulo contem o microcontrolador K210, sendo responsável pelo processamento, e um microcontrolador ESP8285, responsável pela comunicação sem fio via padrão IEEE 802.11.b/g/n. Além disto, há uma memória flash para armazenamento do programa, e circuito de alimentação.

## 3.2 Microcontrolador K210

O componente que realiza o processamento no módulo é o microcontrolador K210, desenvolvido pela empresa Canaan (2018). Em seu interior, há uma CPU com dois núcleos utilizando a ISA do RISC-V, especificamente, *RV64GC*. Deste modo pode-se realizar operações de pontos flutuantes, além de instruções atômicas, e há uma redução do tamanho do código pelas instruções compactas. As extensões presentes são suficientes para o desenvolvimento do trabalho buscado, além dos núcleos providenciarem capacidade de paralelismo. A CPU tem a capacidade de operar a 400 MHz.

Pelo fato de ser um microcontrolador, diversos componentes adicionais são também inclusos. Entre estes, está presente um processador dedicado para rede neurais, para aplicações de inteligência artificial embarcada. Contudo, como o objetivo deste trabalho é a utilização do RISC-V, este processador não será utilizado.

Em seu interior o microcontrolador possui uma memória SRAM de 8 MiB, sendo 6 MiB destes para a CPU principal, e 2 MiB para o processador de rede neural. Contudo, desativando o processador de rede neural, pode-se alocar toda a memória para a CPU, porém com um tempo de acesso maior para dados armazenados nesta região. Além disto, próximo à ambos os núcleos existem duas memórias *cache* de 32 KiB para cada, sendo um destes pares para instruções e outro para dados.

É ainda disponível para a CPU gerenciadores de interrupções internas e externas, independentes para ambos os núcleos.

### 3.2.1 Periféricos

Em adição ao processador, existem diversos periféricos que auxiliam nas execuções das tarefas, e interagem com componentes externos. Estes são configurados salvando valores específicos em seus respectivos registradores, sendo esses acessíveis por endereços reservados da memória.

Alguns destes periféricos podem ser destacados, tendo relevância para este trabalho. O controlador do sistema, ou *System Controller* (SYSCTL) em inglês, é responsável por gerenciar principalmente o *clock* da CPU e periféricos. O K210 possui um oscilador interno de 26 MHz, contudo, pode ser configurado para ser regido por osciladores externos. O SYSCTL configura ainda três controladores de malha de captura de fase, ou *Phase-Locked Loop* (PLL) em inglês. Cada PLL pode obter um *clock* de até 800 MHz, com valores distintos para cada, com base no mesmo oscilador. Cada periférico está conectado a um barramento de um PLL específico.

Além disto, cada periférico possui um divisor de frequência para redução do *clock* do PLL, configurado pelo SYSCTL. Além do *clock*, o SYSCTL mantém a sincronia do DMA.

O controlador de DMA, é um periférico que permite o acesso direto de periféricos para a memória, além de comunicação direta entre periféricos. Para isto, existem 6 canais internos de comunicação cruzada que são gerenciados por este controlador.

O K210 possui também uma matriz de campo programável de entrada/saída, ou *Field Programmable Input/Output Array* (FPIOA) em inglês. A FPIOA permite a comunicação entre os periféricos e os pinos de saída do chip, permitindo um mapeamento de qualquer conjunto das 255 funções disponíveis para os 48 pinos físicos. Permite, deste modo, o controle dos níveis de corrente de saída, e tensões de entrada.

A interface de propósito geral de entrada/saída, ou *General Purpose Input/Output Interface* (GPIO) em inglês, possui uma segmentação em dois barramentos, com velocidades diferentes. De um total de 40 GPIOs, 32 são definidas como de alta velocidade, e 8 padrões.

A interface de periférico serial, ou *Serial Peripheral Interface* (SPI) em inglês, é o que permite a comunicação com diversos dispositivos externos ao chip. No total, o K210 possui quatro SPIs, sendo três destas como *master*, e uma como *slave*. A comunicação de dados por SPI pelo K210 pode ser realizada de 1 a 8 fios. As três primeiras SPIs possuem comunicação em até 25 MHz, e a quarta, de até 100 MHz. Além disto, a SPI pode se comunicar internamente via DMA.

## 3.3 Protocolos de comunicação

Além do processamento, certas tarefas necessitam ser realizadas por dispositivos externos. No caso deste trabalho, é necessário uma forma de visualização da renderização, além de uma forma de armazenamento dos modelos 3D e texturas.

Devido ao conector presente na placa de desenvolvimento Maix Dock, um *display* externo pode ser conectado. Neste caso será utilizado o *display* LCD TFT ST7789V, desenvolvido pela Sitronix (2014). A sua comunicação é realiza por SPI, permitindo uma comunicação de dados de até 8 fios paralelos. Além disto, o *display* possui um protocolo de comunicação próprio, com os comandos especificados em seu *datasheet*.

Neste *display* TFT o resultado da renderização será mostrado, contudo, além disto é necessário a leitura dos arquivos. Os arquivos podem ser diretamente carregados ao programa, contudo, isto acaba por aumentar o tamanho do arquivo, e necessita de remontagem do código para refletir alterações.

Por conta isto, é utilizado um armazenamento externo, onde os arquivos são lidos, e as informações relevantes são salvas na memória SRAM. Desta forma permite uma flexibilidade no manuseio dos arquivos utilizados. Neste caso é utilizado um cartão micro SD, pela presença do conector na placa, e pela sua grande presença no mercado. O cartão micro SD possui um protocolo próprio, contudo, sua comunicação pode ser realizada por SPI de um fio de dado. Existe também um método próprio de comunicação para este tipo de dispositivo, que permite

paralelismo de dados, contudo, necessita de hardware específico.

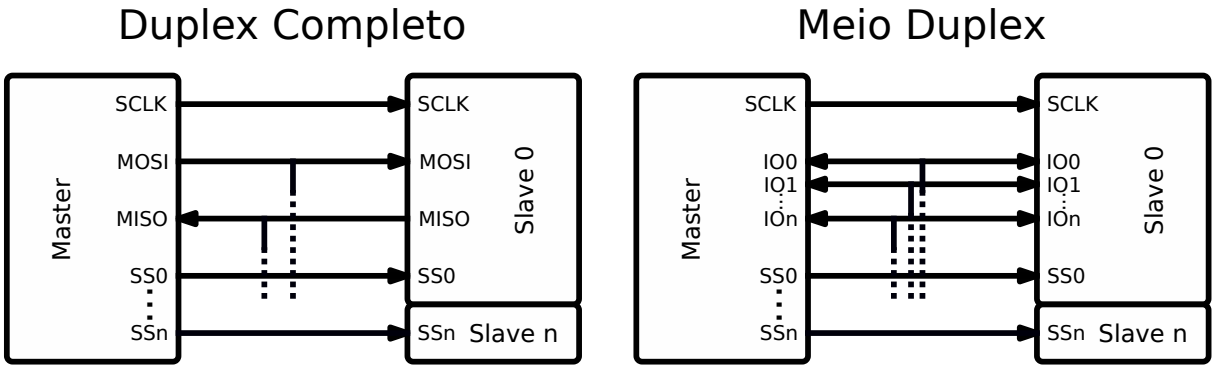
### 3.3.1 SPI

Na utilização com dispositivos diferentes, certa comunicação deve ser estabelecida entre eles. Para isto é necessário que o mesmo método de comunicação seja conhecido e entendido por todos presentes. Existem diversos métodos estabelecidos, contudo, a comunicação por SPI é a relevante para este trabalho.

Quando é necessário uma comunicação direta entre dois dispositivos, a simples transferência de dados entre os pinos conectados pode ser o suficiente. Contudo, quando é necessário uma comunicação entre diversos dispositivos, a quantidade de pinos dedicados podem não ser suficiente. Por conta disto certos métodos de comunicação utilizam barramentos de dados, compartilhados por todos os dispositivos presentes. A SPI se utiliza deste artifício.

Contudo, para controlar a ordem de utilização do barramento, um dispositivo controla o fluxo dos dados, e é denominado de *master*, sendo os demais, *slaves*. O dispositivo *master* precisa ainda de um pino dedicado para cada periférico, porém isto acaba por reduzir a utilização total do chip, em comparação com a comunicação individual de dados. Além disto, o *master* define o *clock* de comunicação entre os dispositivos. Estas conexões podem ser vistas de forma gráfica pela Figura 15.

Figura 15 – Diagrama de conexão de dispositivos por SPI




Fonte: Autoria Própria (2022)

O *clock* é passado pelo pino SCLK, os dados do *master* ao *slave*, pelo pino MOSI, os dados do *slave* ao *master*, pelo pino MISO, e os pinos de seleção são definidos pelo SSn. Esta forma de conexão é denominada de duplex completo, e é a configuração mais comum da SPI, podendo neste caso receber e transmitir dados no mesmo ciclo de *clock*.

Contudo, a SPI pode ser configurada para uma comunicação bidirecional, neste caso, a comunicação é realizada por um meio duplex. Pelas linhas de comunicação de dados serem compartilhadas, os dados podem ser somente recebidos ou enviados de forma não simultânea. Fora os pinos de dados, os demais se mantém idênticos ao duplex completo.

#### 3.3.1.1 Display TFT

O *display* TFT ST7789V, da Sitronix (2014), possui certos comandos específicos para configuração. Aqui serão abordados os itens relevantes ao trabalho, podendo itens adicionais serem conferidos no *datasheet*.

Além dos pinos padrões de uma SPI, de *clock*, de seleção, e 8 linhas de dados bidirecionais, o *display* possui pinos adicionais específicos. Entre estes está um pino de *reset*, que é ativo em baixa, além de um pino de especificação de dado/comando, 1 representando dado, e 0 representando um comando. Estes pinos devem operados em conjunto com a SPI, contudo, por um periférico adicional, como a GPIO.

Para operação é necessário realizar uma sequência de configuração, após a energização do dispositivo. Primeiramente é necessário *resetar* o dispositivo pelo pino de *reset*, colocando a saída como 0, enquanto a SPI é configurada, após isto, é retomado à 1, e inicia o procedimento por software.

Os comandos possuem 8 bits, porém os dados variam em tamanho. Uma lista dos comandos relevantes para esta aplicação, com suas respectivas descrições, pode ser visto na Quadro 5.

Quadro 5 – Lista de comandos do *display* ST7789V referentes ao trabalho

| **Comando** | **Valor** | **Descrição** |
|---|---|---|
| SOFTWARE_RESET | 0x01 | Software retorna a seus valores iniciais |
| SLEEP_ON | 0x10 | Entra em modo de economia energética |
| SLEEP_OFF | 0x11 | Sai do modo de economia energética |
| DISPLAY_OFF | 0x28 | Interrompe a ligação entre memória e tela |
| DISPLAY_ON | 0x29 | Mostra o conteúdo da memória à tela |
| HORIZONTAL_ADDRESS_SET | 0x2A | Define a região horizontal de renderização |
| VERTICAL_ADDRESS_SET | 0x2B | Define a região vertical de renderização |
| MEMORY_WRITE | 0x2C | Escreve na memória do *display* |
| MEMORY_ACCESS_CTL | 0x36 | Escreve no registrador de controle do *display* |
| PIXEL_FORMAT_SET | 0x3A | Define o formato binário do *pixel* |

Fonte: (SITRONIX, 2014)

O primeiro comando a ser enviado é o *reset* do software, pelo comando SOFTWARE_RESET. Após isto, é necessário verificar que o dispositivo não está em rotina de *sleep*, sendo neste estado, seu oscilador interno e circuito de gerenciamento de *display* desativado. Para sair desta rotina, se utiliza o comando SLEEP_OFF. Deste modo o dispositivo está apto a receber configurações específicas de imagem.

O *display* possui opções da coloração do *pixel*, em relação ao valor binário enviado. Este pode ser segmentado em três diferentes tamanhos de até 18 bits. Valores menores tratam os valores recebidos em tabelas de *look-up* para 18 bits. Os diferentes valores para este parâmetro podem ser visto na Quadro 6. Após enviar o comando PIXEL_FORMAT_SET, deve ser enviado a

informação de cores, contendo o valor binário da profundidade de cor duplicado, de modo que os 8 bits sejam preenchidos.

Quadro 6 – Código de cores do *display* ST7789V

| **Profundidade** | **R** | **G** | **B** | **Cores Possíveis** | **Valor Binário** |
|---|---|---|---|---|---|
| 12 bits | 4 | 4 | 4 | 4095 | 0b0011 |
| 16 bits | 5 | 6 | 5 | 65535 | 0b0101 |
| 18 bits | 6 | 6 | 6 | 262143 | 0b0110 |

Fonte: (SITRONIX, 2014)

A orientação da imagem na tela pode ser ajustada. Para realizar isto é necessário acessar o registrador de controle do *display*. Este acesso para escrita é realizado pelo comando `MEMORY_ACCESS_CTL`. Dentro deste, diversos parâmetros de operação podem ser ajustados. A função e seu respectivo bit podem ser vistos na Quadro 7.

Quadro 7 – Funções do registrador de controle do *display* ST7789V

| **Bit** | **Descrição** | **Função em 0** | **Função em 1** |
|---|---|---|---|
| 7 | Ordenação de linha | Cima a baixo | Baixo a cima |
| 6 | Ordenação de coluna | Esquerda a direita | Direita a esquerda |
| 5 | Coordenadas da tela | Linha X, e coluna Y | Coluna X, e linha Y |
| 4 | Direção de *refresh* em Y | Cima a baixo | Baixo a cima |
| 3 | Ordem de cor | RGB | BGR |
| 2 | Direção de *refresh* em X | Esquerda a direita | Direita a esquerda |
| 1:0 | Sem uso | | |

Fonte: (SITRONIX, 2014)

Após as configurações iniciais, o *display* pode ser comandado para mostrar imagens à tela. Isto é realizado com o comando `DISPLAY_ON`. Neste caso a imagem pode ser transferida da memória interna do *display* aos *pixels*. Este estado pode ser desativado pelo comando `DISPLAY_OFF`.

A região de renderização da tela pode ser diferente das dimensões da tela. Isto deve ser definido sempre que desejado atualização dos *pixels* na tela. Para a região da tela na direção horizontal, o que seria as coordenadas de "X" em relação a tela, é utilizado o comando `HORIZONTAL_ADDRESS_SET`. Para a região vertical, o que seria no sentido de "Y", o comando é `VERTICAL_ADDRESS_SET`. O ponto inicial é dado pelas coordenadas 0,0, e o ponto máximo é dado por 239,319, possuindo assim, o *display* ST7789V, dimensões de tela de 240x320.

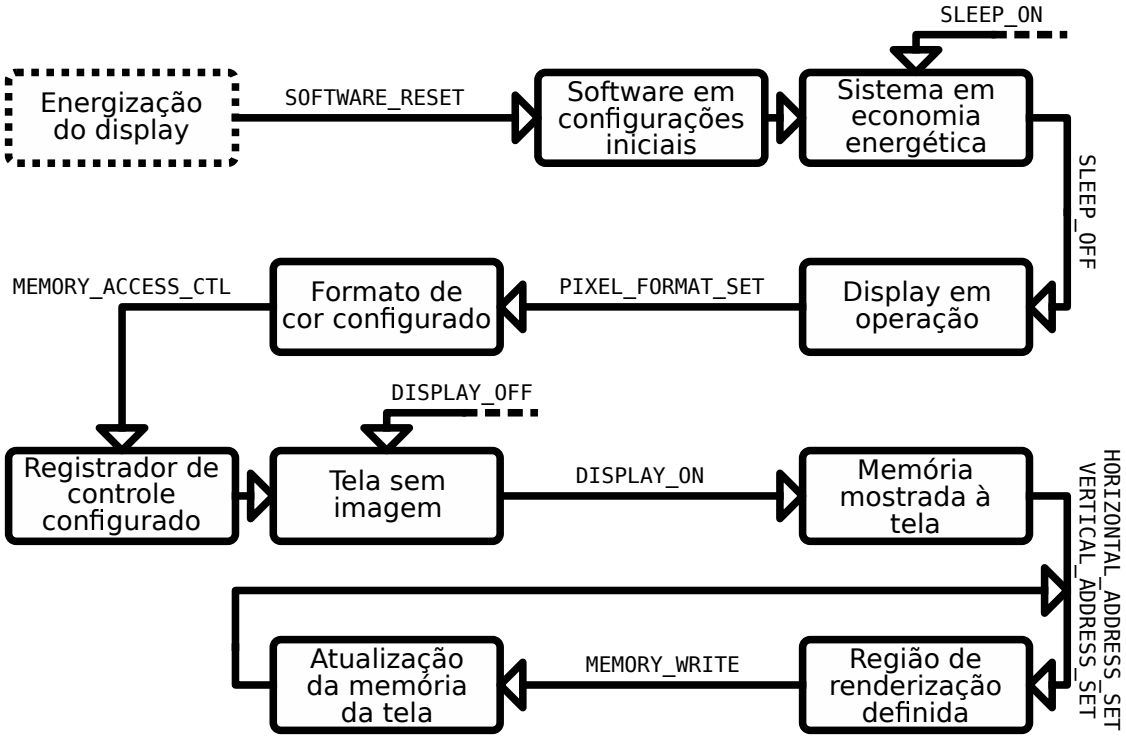
A região de renderização dos *pixels* são definidos por 2 registradores, para ambas as dimensões. Deve-se notar também que a transferência de dados é dada por 8 bits por ciclo de *clock*. Para que ambas as dimensões sejam abrangidas, os registradores comportam 16 bits. Deste modo, após que o comando de definição de coordenadas é enviado, seja horizontal ou

vertical, é primeiro definido os valores iniciais da região, e depois os valores finais, ambos em 16 bits.

Com a região definida, os dados de cores podem ser enviados à memória do *display*. Isto é realizado pelo comando `MEMORY_WRITE`. Após isto, os valores de cores são enviados, até que toda a região seja preenchida. A conversão binária às cores, é definida pelo código de cores previamente estabelecido. O sentido de preenchimento é dada pelas direções de *refresh*, definidas no registrador de controle.

Um fluxograma baseado nestas etapas de configuração pode ser visto na Figura 16.

Figura 16 – Fluxograma da configuração do *display* ST7789V




Fonte: Autoria Própria (2022)

#### 3.3.1.2 Protocolo SD

Outra forma de comunicação que se utiliza da SPI, é o protocolo SD. Este é utilizado para a comunicação do dispositivo com cartões micro SD, porém, não é restrito somente a estes. O cartão micro SD é um dispositivo utilizado para o armazenamento de dados de forma compacta, sendo prevalente em equipamentos portáteis, como câmeras e celulares.

A origem deste protocolo vem do MultiMediaCard, utilizado como padrão no mercado de câmeras digitais. Após isto, seu uso foi englobado nos cartões de memória SD. Ambos estes sendo dispositivos de armazenamento externo não-volátil, utilizando tecnologias de dispositivos semicondutores, mais comumente, baseados na memória *flash*. Sendo atualmente, um padrão para diversos equipamento eletrônicos.

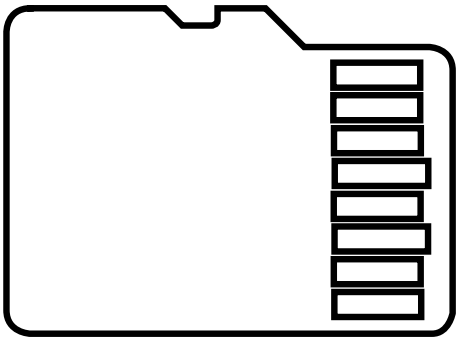
Desde sua implementação, diversas tecnologias diferentes foram implementadas no protocolo SD, contudo, todas mantém ainda suporte legado aos demais dispositivos. Deste modo é necessário várias ramificações do programa para utilizar de forma eficiente o cartão inserido.

Atualmente, todas as informações sobre dispositivos SD são mantidos nas especificações da SD Card Association (2020). Existindo diversas complexidades envolvendo o dispositivo, será abordado apenas o material relevante a este trabalho. E isto esta limitado ao escopo da comunicação com um cartão micro SD, via SPI, a fim de acessar a memória para leitura de dados.

Os cartões SD são divididos também em tamanho. Os cartões de capacidade padrão (SDSC) permitem armazenamento de até 2 GB, os de alta capacidade (SDHC) armazenam de 2 a 32 GB, os de capacidade estendida (SDXC), de 32 GB a 2 TB, e os de ultra capacidade (SDUC), de 2 a 128 TB. O tamanho do cartão influenciará no método de segmentação dos blocos, como será visto na Seção 3.4, por conta disto, será focado neste trabalho somente os cartões do tipo SDHC.

A comunicação via SPI pelo cartão micro SD pode ser vista na Figura 17. Deve-se notar que a alimentação por VDD, para este dispositivo SD, deve ser estar na faixa de 2,7 a 3,6 V.

Figura 17 – Pinos do cartão micro SD via comunicação SPI



| Pino | Função |
|---|---|
| 1 | Reservado |
| 2 | SS |
| 3 | MOSI |
| 4 | VDD |
| 5 | CLK |
| 6 | GND |
| 7 | MISO |
| 8 | Reservado |

Fonte: SD Card Association (2020)

Após o dispositivo SD conectado, o procedimento de configuração por software pode ser inicializado. Os comandos são de 6 bytes, com regiões para o índice, um argumento, e o código de verificação, o formato pode ser visto na Tabela 5. Como a comunicação é serial com 1 linha, a transmissão se inicia com o bit mais significativo, ou *Most Significant Bit* em inglês. Ou seja, se inicia do bit mais longe do zero possível, que seria o bit menos significativo, ou *Least Significant Bit* em inglês.

Para garantir que erros de comunicação não ocorram no meio do caminho, todo comando é complementado de uma verificação, neste caso, realizado pelo método de detecção de erro linear de verificação cíclica de redundância, ou *Cyclic Redundancy Check* (CRC) em inglês. No caso do comando, este valor é de 7 bits, e utiliza o polinômio $x^7 + x^3 + 1$.

Tabela 5 – Formato do comando SD

| 47 | 46 | 45 — 40 | 39 — 8 | 7 — 1 | 0 |
|---|---|---|---|---|---|
| *0* | *1* | *Índice do Comando* | *Argumento* | *CRC7* | *1* |

Fonte: (SD Card Association, 2018)

Uma lista de comandos existe, permitindo comunicação de dispositivos além dos cartões de memória. Contudo, para este trabalho, os comandos relevantes podem ser vistos na Quadro 8. O valor do índice está indicado no próprio comando. Os comandos que são do formato `ACMD`, são ditos comandos de aplicação específica. Contudo, não ha diferença de formato entre estes e comandos comuns, possuindo duplicidade em seus índices.

Quadro 8 – Lista de comandos do protocolo SD referentes ao trabalho

| **Comando** | **Descrição** | **Argumento** |
|---|---|---|
| `CMD0` | Retoma o cartão ao estado inicial | |
| `CMD8` | Verifica a versão do cartão e tensões suportadas | Tensões disponíveis, e padrão de verificação |
| `CMD12` | Requisita o fim de uma leitura de múltiplos blocos | |
| `CMD17` | Leitura de um bloco do cartão | Endereço do bloco |
| `CMD18` | Leitura de blocos subsequentes | Endereço do bloco inicial |
| `CMD55` | Próximo comando a ser interpretado como de aplicação específica | |
| `CMD58` | Requisita o registrador OCR do cartão | |
| `ACMD41` | Requisita a inicialização interna do cartão SD | Suporte do dispositivo para SDHC e SDXC |

Fonte: (SD Card Association, 2018)

Após o comando, o dispositivo retorna uma resposta, dependendo do que for requisitado. Estas respostas são agrupadas em certos formatos diferentes, e podem ser vistas na Tabela 6.

Tabela 6 – Formato da resposta dos comandos SD

| 47 | 46 | 45 — 40 | 39 — 8 | 7 — 1 | 0 | **Tipo** |
|---|---|---|---|---|---|---|
| *0* | *0* | *Índice* | *Estado do cartão* | *CRC7* | *1* | R1 |
| 135 | 134 | 133 — 128 | 127 — 1 | | 0 | |
| *0* | *0* | *111111* | *Registrador CID ou CSD* | | *1* | R2 |
| 47 | 46 | 45 — 40 | 39 — 8 | 7 — 1 | 0 | |
| *0* | *0* | *111111* | *Registrador OCR* | *1111111* | *1* | R3 |
| *0* | *0* | *Índice* | *Registrador RCA* | *CRC7* | *1* | R6 |

| 47 | 46 | 45 — 40 | 39 — 22 | 21 | 20 | 19 — 16 | 15 — 8 | 7 — 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|---|
| *0* | *0* | *Índice* | *0x00000* | *PCIe* | *Tensão aceita* | | *Verificação* | *CRC7* | *1* | R7 |

Fonte: (SD Card Association, 2018)

A frequência da SPI para o cartão pode ser alterada em uma certa faixa, com certas versões providenciando um aumento na velocidade de troca de dados. Contudo, a fim de manter suporte para dispositivos legados, o *clock* da SPI deve ser iniciado na frequência de 100 KHz. Após verificação de compatibilidade, este valor pode ser aumentado.

Na inicialização do software, o primeiro comando a ser enviado é o `CMD0`. Este comando força o cartão a retomar a sua condição inicial, e se manter à espera de comandos. A sua resposta é no formato de R1, indicando se o cartão pode ser acessado no momento. Caso o cartão se encontre disponível sem problemas, todos os bits de estado são retornados como 0, podendo ser visto cada valor na Quadro 9. Para garantir a prontidão do cartão, é recomendado repetir o comando `CMD0` até que se obtenha êxito na resposta do estado.

Quadro 9 – Significado dos bits de estado do cartão SD

| **Bit** | **Descrição** | **Bit** | **Descrição** |
|---|---|---|---|
| 31 | Argumento fora do alcance | 18:17 | Reservado |
| 30 | Endereço não alinhado requisitado | 16 | Erro na escrita do registrador CSD |
| 29 | Erro no tamanho do bloco | 15 | Região protegida pulada |
| 28 | Erro na sequência de apagamento | 14 | Sem uso do corretor de erro |
| 27 | Blocos para apagar inválidos | 13 | Cancelada ordem de apagamento |
| 26 | Tentativa de apagar área protegida | 12:9 | Estado atual do cartão |
| 25 | Cartão está travado | 8 | Cartão pronto para aceitar dados |
| 24 | Falha ao (des)travar o cartão | 7 | Reservado |
| 23 | Erro ao verificar CRC | 6 | Bit para extensões |
| 22 | Comando ilegal | 5 | Comando interpretado como `ACMD` |
| 21 | Falha no corretor de erro | 4 | Reservado |
| 20 | Falha no controlador do cartão | 3 | Erro na sequência de autenticação |
| 19 | Falha no controlador do cartão | 2:0 | Reservado |

Fonte: (SD Card Association, 2020)

Com o cartão disponível, se pode obter informações deste, para que se possa configurar a velocidade do *clock* do dispositivo. Para isto, é primeiramente utilizado o comando `CMD8`, em que é enviado como argumento as tensões disponíveis do dispositivo *master*, em junção com um código de verificação. Após isto, o cartão retorna com a resposta no formato R7, em que é retornado o valor do código de verificação, em conjunto com uma confirmação da tensão suportada. Além disto, caso seja obtida uma resposta válida, indica que o cartão é da versão 2 em diante, o que o agrupa nas classes mínimas de SDHC ou SDXC.

Após isto, pode ser enviado o comando de inicialização do cartão SD, pelo `ACMD41`. Para enviar um comando de aplicação específica, é necessário primeiramente enviar uma requisição para o cartão, via comando `CMD55`, podendo ser repetido até obter êxito, não havendo duplicidade de índice para este comando. A resposta do `CMD55` se dá via R1, e caso o cartão não esteja ocupado, o próximo comando enviado será interpretado como comando de aplicação específica.

O comando `ACMD41` é enviado com o argumento de um bit, indicando o suporte do dispositivo *master* para cartões SDHC e SDXC, sendo 1 no caso de suporte, e 0 caso contrário.

Após configuração do dispositivo, o comando `CMD58`, realiza o requerimento das disposições do cartão SD. A resposta é dada por R3, que retorna o registrador de operação e condição, ou *Operation Condition Register* (OCR) em inglês.

O OCR fornece diversas informações de compatibilidade do cartão, como faixas de tensão e tecnologia de capacidade. Uma lista dos valores podem ser vistos na Quadro 10. Contudo, caso se busque a capacidade SDHC ou SDXC, o bit 30, de estado da capacidade do cartão, será definido como 1. Isto permite que a velocidade de transmissão via SPI atinga, no mínimo, 25 MHz, além da faixa para o método de segmentação desejado. É desejável verificar também o bit de alimentação do cartão, para garantir que não houveram falhas durante a inicialização do dispositivo.

Quadro 10 – Valores do OCR do cartão SD

| **Bit** | **Descrição** | **Bit** | **Descrição** |
|---|---|---|---|
| 31 | Cartão propriamente alimentado | 21 | Suporta 3,3 a 3,4 V |
| 30 | Estado da capacidade do cartão | 20 | Suporta 3,2 a 3,3 V |
| 29 | Capacidade de UHS-II | 19 | Suporta 3,1 a 3,2 V |
| 28 | Reservado | 18 | Suporta 3,0 a 3,1 V |
| 27 | Suporte para 2 TB | 17 | Suporta 2,9 a 3,0 V |
| 26:25 | Reservado | 16 | Suporta 2,8 a 2,9 V |
| 24 | Capacidade para 1,8 V | 15 | Suporta 2,7 a 2,8 V |
| 23 | Suporta 3,5 a 3,6 V | 14:0 | Reservado |
| 22 | Suporta 3,4 a 3,5 V | | |

Fonte: (SD Card Association, 2020)

Com o dispositivo propriamente configurado, a frequência de *clock* pode ser aumentada, para garantir uma taxa de transmissão maior de dados.

A memória no interior do cartão SD é segmentada em blocos, e que será a quantidade elementar de leitura. Geralmente estes blocos são de 512 bytes, porém, podem ser alterados entre a faixa de 1 a 2048 bytes, via `CMD16`. Contudo, é recomendo se manter ao padrão de 512 bytes. O índice da memória segmentada por blocos é denominada de setores.

Para efetuar a leitura de um endereço específico, é necessário requisitar o bloco em que se situa como um todo. Por conta disto o valor do endereço deve ser arrendondado para a primeira posição múltipla do tamanho do bloco, caso contrario, um aviso de endereço desalinhado é retornado. Isto pode ser obtido também, tratando a memória como setores ao invés de endereços, multiplicando o valor do setor requisitado pelo tamanho do bloco. Garantindo isto, o endereço pode ser enviado como argumento para o comando `CMD17`.

Contudo, o `CMD17` irá somente ler um único setor na memória. Para aumentar a taxa de leitura, o protocolo providência o comando `CMD18`, que lerá os subsequentes blocos, até um comando de finalização, dado pelo `CMD12` seja enviado.

Após o comando, o dispositivo deve se manter no aguardo da resposta do cartão, que enviará uma resposta de estado R1, e caso êxito, enviará os dados, precedido de um *token* de dados, com o valor 0xFE, seguido do bloco, e finalizado com um código de verificação CRC de 16 bits, com polinômio $x^{16} + x^{12} + x^5 + 1$. Para um tamanho de bloco de 512 bytes, a resposta dos dados pode ser vista na Tabela 7. No caso de múltiplos setores, via `CMD18`, a resposta de estado não é repetida, porém, o *token* de dados e a verificação CRC estão presentes em cada bloco.

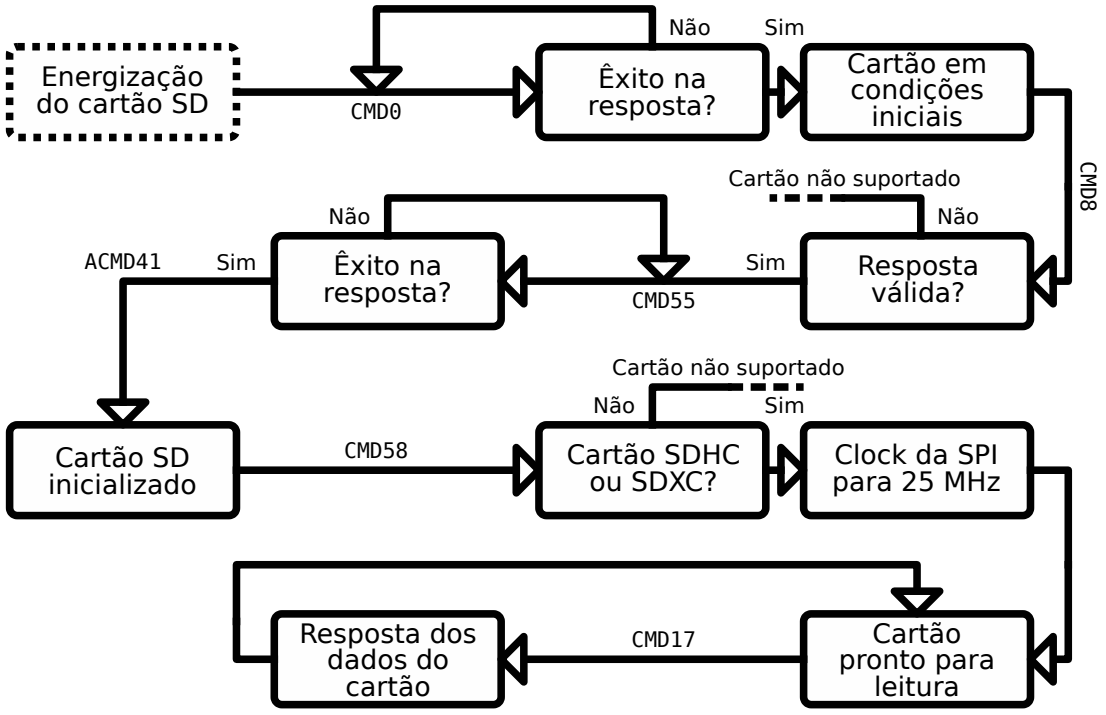
Tabela 7 – Formato da resposta de leitura de dados do cartão SD

| 535 – 528 | 527 – 16 | 15 – 0 |
|---|---|---|
| *11111110* | *Bloco de dados* | *CRC16* |

Fonte: (SD Card Association, 2018)

Estas etapas de configuração são apresentadas em formato de fluxograma na Figura 18.

Figura 18 – Fluxograma da configuração do cartão SD




Fonte: Autoria Própria (2022)

## 3.4 Armazenamento de dados

A memória RAM de um dispositivo permite que certos dados sejam armazenados para serem processados pela CPU. Contudo, seu uso é limitado, e não há acesso a esta, fora via instruções da CPU. Isto acaba dificultando que sejam armazenados informações

relevantes diretamente na RAM. Uma forma de enviar os dados pode ser feita via protocolos de comunicação a outro dispositivo que supra essa informação em tempo real, porém, este método pode acabar por ser custoso, e dificultando a repetibilidade do algoritmo, por necessitar sempre de um equipamento externo.

Por conta disto, é preferível a utilização de um dispositivo de armazenamento externo, que possa ser configurado por um computador distinto do microcontrolador. Neste caso, este dispositivo é um cartão micro SD. Contudo, para armazenamento de arquivos, o computador realiza diversas operações de configuração e organização da memória interna. Nisto são definidos metadados dos arquivos, e em quais regiões da memória seus valores binários são alocados.

Neste trabalho, se busca armazenar arquivos contendo as informações relevantes para o modelo tridimensional a ser renderizado. As informações destes arquivos são definidas por metadados de alocação, que segmentam a memória bruta em blocos para a alocação dos dados. Esta segmentação é necessária, porque permitem que grandes arquivos sejam armazenados de forma não continua, o que pode se fazer necessário quando arquivos menores são espalhados no contínuo dos endereços da memória devido à liberação de espaço de outros arquivos durante apagamentos. Este processo acaba por se comportar, basicamente, como um mapeamento de endereços da memória real, para aqueles relativos ao arquivo.

### 3.4.1 Sistema de arquivos FAT

Existem diversos métodos reconhecidos para a alocação das regiões da memória, denominados de sistemas de arquivos. Entre estes, está o sistema de arquivo de tabela de alocação de arquivos, ou *File Allocation Table* (FAT) em inglês.

O uso inicial do sistema FAT foi previsto para disquetes, contudo, se mantém em utilização devido a sua simplicidade de implementação. Seu ponto de destaque é o suporte em quase todo equipamento que realize comunicação com dispositivos de armazenamento externo. Possui suporte em todos os sistemas operacionais predominantes, além de diversos eletrônicos portáteis que necessitem de uma memória externa. Isto acaba por permitir a fácil transferência de dados entre equipamentos, sem problemas de compatibilidade, ou necessidade de conversão. Por conta disto, seu uso se tornou o padrão para dispositivos de memória removíveis, como Pen Drives e cartões SD.

Contudo, boa parte dos benefícios que tornam o FAT atraente, acabam por se tornarem prejudiciais em certos aspectos. A simplicidade do seu método de armazenamento significa que os metadados dos arquivos são mínimos, como a falta do atributo de proprietário do arquivo. Além disto, o FAT não aplica compressão nos arquivos, o que acaba por aumentar o tempo de transferência de dados, e acaba por ocupar um espaço maior na memória do dispositivo.

A segmentação no interior da tabela de alocação é realizada em *clusters*. Cada arquivo é alocado em um número inteiro de *clusters*, arredondado para cima. Por conta disto certa quantidade de memória no *cluster* final pode ser ocupada por espaços não alocados. Contudo, o benefício de segmentação dos arquivos é priorizado, neste caso. Com os arquivos segmentados,

o sistema pode armazenar grandes quantidades de arquivos intercalados, e após apagar uma quantidade destes da memória, o espaço pode ser preenchido por novos arquivos sem corromper os demais, garantindo que cada *cluster* ocupe somente seu espaço.

Cada *cluster* é definido por um agrupamento de setores, podendo ser compostos em até 128 destes. E como já visto na Subsubseção 3.3.1.2, estes setores são, em si, compostos de 512 bytes geralmente. Este valor pode ser alterado, como será visto a frente, porém, a fim de manter a compatibilidade, é tipicamente mantido em 512 bytes.

As especificações do sistema de arquivos FAT pode ser visto em mais detalhe em Microsoft (2005). Contudo, algumas de suas especificações serão aqui descritas.

O sistema FAT é divido em certas versões, sendo o principal ponto de variação o valor do endereço máximo alocável em sua tabela. A versão FAT12 permite a criação de $2^{12}$ *clusters*, o que permite um armazenamento teórico máximo de $2^{28}$ bytes, ou 256 MB, considerando setores de 512 bytes. A versão seguinte de FAT16 segue o mesmo padrão, porém, por especificação, seu valor máximo de *clusters* é de 64 setores, o que permite uma alocação máxima de $2^{31}$ bytes, ou 2 GB. A última versão nesta linha, o FAT32, é limitada por especificação em 32 setores por *cluster*, contudo sua limitação máxima é no número de setores, permitindo a alocação de $2^{32}$ setores, o que reflete em $2^{41}$ bytes, ou 2 TB de limite. Contudo, uma certa quantidade de *clusters* deve ser reservada para as informações do sistema, reduzindo estes valores na prática.

Além da variação na quantidade máxima de alocação de memória, certas variações na estrutura dos metadados também são presentes. Por conta disto, neste trabalho será focado somente na versão FAT32, por permitir dispositivos de armazenamento em uma grande faixa de valores presentes atualmente no mercado. Além disto, o tamanho do setor será considerado pelo padrão de 512 bytes, ou 0x200 em hexadecimal.

A forma de segmentação realizada pelo FAT32 é composta de três faixas principais. A primeira, posicionada no endereço 0x00, é o registro mestre de inicialização, ou *Master Boot Record* (MBR). Esta região é do tamanho de um único setor, e armazena as informações necessárias para o dispositivo inicializar. As segmentações da região do MBR pode ser vista na Quadro 11. Este setor está presente em outros sistemas de arquivos, não sendo restritos ao FAT. O sistema de arquivo em si, é aplicado individualmente para cada partição do disco, porém é relevante o conhecimento desta região, para a ordem de progressão para acesso de um arquivo na memória. O MBR é, porém, um método não utilizado em memórias de armazenamento em massa, contudo, devido a sua simplicidade, ainda é utilizado em dispositivos removíveis.

A primeira região do código de inicialização pode ser lida como código binário pelo computador, caso não exista um sistema operacional em execução, realizando certas operações breves para estabelecer um ambiente de operação. A partir disto quatro partições podem ser alocadas, podendo os valores de cada uma destas entradas ser visto na Quadro 12. E no final do MBR existe uma assinatura de inicialização, definida pelo valor 0x55AA, indicando o final do setor.

A primeira região define se a partição é inicializável, ou seja, se deve ser lida como

Quadro 11 – Definição das regiões do MBR

| Offset | Tamanho | Descrição |
|---|---|---|
| 0x000 | 446 B | Código de inicialização |
| 0x1BE | 16 B | Entrada da partição 1 |
| 0x1CE | 16 B | Entrada da partição 2 |
| 0x1DE | 16 B | Entrada da partição 3 |
| 0x1EE | 16 B | Entrada da partição 4 |
| 0x1FE | 2 B | Assinatura de inicialização [0x55AA] |

Fonte: (MICROSOFT, 2005)

Quadro 12 – Definição das regiões da entrada da partição

| Offset | Tamanho | Descrição |
|---|---|---|
| 0x00 | 1 B | Partição inicializável |
| 0x01 | 3 B | Endereço do primeiro setor cilindro-cabeça da partição |
| 0x04 | 1 B | Identificação da partição |
| 0x05 | 3 B | Endereço do último setor cilindro-cabeça da partição |
| 0x08 | 4 B | Endereço lógico do primeiro setor da partição |
| 0x0C | 4 B | Números de setores na partição |

Fonte: (MICROSOFT, 2005)

código de máquina para inicialização do sistema. Caso o valor seja 0x80, a partição é inicializável, e 0x00 significa uso apenas para armazenamento de dados. O endereçamento via cilindro-cabeça não é comumente utilizado, mas se mentém por compatibilidade legado com dispositivos antigos. O identificador de partição define o tipo do sistema de arquivos da partição, e estes valores são previamente tabelados. Referente a utilização neste caso, o código de identificação do FAT32 buscado é 0x0C, que especifica, além disto, que é um dispositivo com alocação de blocos lógicos, por ser buscado a utilização de um cartão SD, em contra partida do cilindro-cabeça.

O endereço de início da partição, e para onde deve ser prosseguido, caso se deseje acessar esta partição. E o número de setores delimita o tamanho total da partição. Como estes valores são de 4 bytes, ou 32 bits, uma partição não pode começar a partir de 2 TB na memória, e ter um tamanho maior que este. Deve se atentar também que os valores armazenados nas regiões de metadados do FAT32 são armazenados como *little-endian*, ou seja, os maiores valores do número são armazenados nos maiores valores de memória. Como geralmente os endereços são lidos das menores para as maiores posições, deve ser dedicada atenção a este item, para evitar problemas na leitura.

Após o redirecionamento do endereço via MBR, o processo se encontra em uma partição em que o sistema FAT32 esta presente. O primeiro setor de uma partição FAT32 é o setor de inicialização. As informações deste setor podem ser vistas na Quadro 13.

Caso a partição seja inicializável, o primeiro endereço deste setor será lido como código de máquina, e redirecionará o processo para a rotina de inicialização. Caso lido como

Quadro 13 – Definição das regiões do setor de inicialização da partição

| **Offset** | **Tamanho** | **Descrição** |
|---|---|---|
| 0x000 | 3 B | Instrução de salto para código de inicialização |
| 0x003 | 8 B | Texto ASCII de identificação do fabricante |
| 0x00B | 2 B | Tamanho do setor em bytes |
| 0x00D | 1 B | Tamanho do *cluster* em setores |
| 0x00E | 2 B | Tamanho da região reservada em setores |
| 0x010 | 1 B | Quantidade de FATs |
| 0x011 | 4 B | Reservado |
| 0x015 | 1 B | Definição do tipo de dispositivo de armazenamento |
| 0x016 | 8 B | Reservado |
| 0x020 | 4 B | Número de setores na partição |
| 0x024 | 4 B | Tamanho de uma FAT em setores |
| 0x028 | 2 B | Bandeiras de identificação de extensão |
| 0x02A | 2 B | Versão do FAT32 presente |
| 0x02C | 4 B | *Cluster* da pasta raiz |
| 0x030 | 2 B | Setor da estrutura de informações do sistema de arquivos |
| 0x032 | 2 B | Setor da cópia do código de inicialização |
| 0x034 | 14 B | Reservado |
| 0x042 | 1 B | Assinatura de inicialização estendida |
| 0x043 | 4 B | Identificação serial do dispositivo |
| 0x047 | 11 B | Texto ASCII do nome dado ao dispositivo |
| 0x052 | 8 B | Texto ASCII do nome do sistema de arquivo |
| 0x05A | 420 B | Código de inicialização |
| 0x1FE | 2 B | Assinatura de inicialização [0x55AA] |

Fonte: (MICROSOFT, 2005)

somente dispositivo de armazenamento, esta região pode ser ignorada. As regiões de texto no código padrão americano para o intercâmbio de informações, ou *American Standard Code for Information Interchange* (ASCII) em inglês, servem apenas finais informativos, não havendo crucialidade de preenchimento ou leitura. O tamanho do setor, como já previamente comentado, é geralmente definido como 512 bytes, mas outros valores podem ser definidos.

O tamanho do *cluster* é o que criará a unidade de segmentação dos arquivos. A memória é internamente segmentada nestes *clusters*, e a alocação destes por cada arquivo é definida na FAT. Não há um valor completamente otimizado para este tamanho, *clusters* menores permitirão uma maior segmentação dos arquivos, evitando que os *clusters* finais sejam subutilizados, contudo, necessitam de uma FAT maior, e há um esforço computacional maior para mapeamento de todas as regiões, e como já visto, a transferência de dados de um cartão SD aumenta caso os dados sejam contíguos. Este valor é definido como um múltiplo de 2, e no caso do FAT32, pode variar de 1 à 128 setores por *cluster*.

O tamanho da região reservada marca a região inicial da partição. A partir deste valor, as FATs são alocadas de forma continua, até o início dos dados armazenados.

O mapeamento dos *clusters* são definidos na FAT. Desta forma se obtém informações

dos *clusters* utilizados pelo arquivo, disponíveis para uso, e regiões corrompidas da memória. Esta tabela ocupa, por si, uma região da memória, e, de modo a garantir a segurança destas informações, são criadas réplicas da FAT em regiões subsequentes. A quantidade de FATs informa quantas destas tabelas estão disponíveis.

A definição do tipo de dispositivo informa se a memória é do tipo removível, ou se é um disco dedicado. O número de setores na partição geralmente possui o mesmo valor do obtido na entrada da partição, e identifica o tamanho da partição, como visto anteriormente.

O *Cluster* da pasta raiz informa onde os arquivos em si se iniciam na memória. Os valores dos *clusters* são começados a contar a partir do início da partição. A partir disto, a pasta raiz pode ser localizada, e os arquivos podem ser buscados dados seus endereços relativos.

O setor da estrutura de informações serve para providenciar dados adicionais do dispositivo, e o de cópia do código de inicialização providencia redundância contra corrompimento de memória. E a assinatura de inicialização marca o fim do setor inicial.

Ao acessar as FATs, as informações da ocupação dos *clusters* são obtidas. No caso do FAT32, a quantidade máxima teórica de *clusters* é de $2^{32}$-1. Por conta disto, para cada *cluster* na tabela, 4 bytes são reservados. A tabela de alocação não providencia informação dos arquivos por si só, contudo, cada *cluster* alocado nesta tabela referencia o seu subsequente. Ou seja, nos 4 bytes referentes ao *cluster* na tabela, é indicado o índice do próximo *cluster* do arquivo, porém alguns valores são reservados, sendo interpretados como valores especiais.

Um valor de 0x00000000 indica que o *cluster* esta livre para alocação, podendo assim ser sobrescrito por qualquer dado que se deseja armazenar. Valores de 0xFFFFFFF0 são interpretados como fim da cadeia de *clusters*. O valor de 0xFFFFFFF7 significa uma região com defeito, que não deve ser utilizada. E 0xFFFFFFF8 indica um *cluster* reservado. E com estas informações, as segmentações do arquivo podem ser unidas, juntando todas os dados necessários.

Contudo, arquivos são geralmente definidos por endereços relativos, para mais fácil interpretação humana. São definidos nomes para os arquivos, e estes podem ser armazenados dentro de pastas. Isto se segue até que haja um vetor de caracteres relacionando o arquivo desejado à pasta raiz do dispositivo. O endereço desta pode ser obtido no setor de inicialização.

A pasta raiz se comporta como um arquivo, necessitando de *clusters* alocados até que todos seus dados sejam preenchidos na memória. Esta pasta guarda as informações necessárias para acessar demais pastas e arquivos, como informações de *clusters* e nome. O método de estrutura de uma pasta comum e da pasta raiz no FAT32 não difere, por conta disto, a análise subsequente é dada de modo geral.

Uma entrada em uma pasta no sistema FAT pode ser dada de duas formas, entrada longa, ou entrada curta. Entradas longas são utilizadas quando o nome do arquivo possui muitos caracteres, e não cabe em uma curta. O formato de uma entrada longa pode ser visto na Quadro 14.

A entrada longa pode ser concatenada, criando um arquivo com dezenas de caracteres.

Quadro 14 – Regiões da entrada longa de um arquivo

| Offset | Tamanho | Descrição |
|---|---|---|
| 0x00 | 1 B | Valor da sequência, realiza *OR* com 0x40 caso seja o último |
| 0x01 | 10 B | Caracteres 1 à 5 da entrada, por Unicode de 16 bits |
| 0x0B | 1 B | Atributo da entrada [0x0F] |
| 0x0C | 1 B | Reservado |
| 0x0D | 1 B | Soma de verificação |
| 0x0E | 12 B | Caracteres 6 à 11 da entrada, por Unicode de 16 bits |
| 0x1A | 2 B | Reservado |
| 0x1C | 4 B | Caracteres 12 à 13 da entrada, por Unicode de 16 bits |

Fonte: (CARRIER, 2005)

Cada entrada deve ser então identificada na ordem. Isto é dado no valor de sequência, sendo no final desta, o valor realizado *OR* lógico com 0x40. Por conta disto, há um limite teórico de 0x3F, ou 63 entradas longas, o que reflete em 819 caracteres para um único arquivo. Além disto, caso o valor de sequência seja 0x00, isto indica um espaço de entrada não alocado, e 0xE5 indica uma entrada apagada.

Os caracteres do arquivo são armazenados no formato Unicode de 16 bits, com cada um ocupando 2 bytes, e armazenados no formato *little-endian*. O atributo de verificação varia dependendo do tipo do arquivo na entrada curta, porém, para entrada longa este valor é sempre de 0x0F.

O valor de verificação é o mesmo para todas as entradas longas, e é obtido pela soma dos caracteres da entrada curta, rotacionando o valor resultante. A operação pode ser vista no Algoritmo 5.

**Algorithme 5** Algoritmo de geração do valor de verificação para a entrada longa

1: *Verificação* $= 0$;
2: **for** $i \in [0,11]$ **do**
3: *Verificação* $\leftarrow$(**if** *Verificação*$\wedge$0x01 **then** 0x80 **else** 0)$+$(*Verificação*$>>1$);
4: *Verificação* $\leftarrow$*Verificação* $+$ *Caracteres*$[i]$;

A entrada curta permite arquivos com no máximo 8 caracteres em ASCII, com espaço para 3 caracteres de formato de arquivo. As entradas longas referenciam a esta, e é onde as informações de metadados do arquivo estão presentes. O formato desta entrada pode ser visto na Quadro 15.

Os caracteres do nome do arquivo são convertidos para os seus valores ASCII em letra maiúscula. Após isto, caso o arquivo possua um nome maior do que 8 caracteres, o dispositivo deve criar uma entrada curta com um processo de compressão do nome. Após isto, caso se deseja se manter o nome longo, devem ser criadas entradas longas até que todos os caracteres sejam armazenados. Este processo deve ocorrer também caso exista caracteres que não pertençam à tabela ASCII.

Quadro 15 – Regiões da entrada curta de um arquivo

| **Offset** | **Tamanho** | **Descrição** |
|---|---|---|
| 0x00 | 1 B | Caractere 1 do nome do arquivo em ASCII, e estado de alocação |
| 0x01 | 7 B | Caracteres 2 à 8 do nome do arquivo em ASCII |
| 0x08 | 3 B | Caracteres 1 à 3 da extensão do arquivo em ASCII |
| 0x0B | 1 B | Atributo da entrada |
| 0x0C | 1 B | Reservado |
| 0x0D | 1 B | Centésimos de segundos do tempo de criação |
| 0x0E | 2 B | Hora, minuto, e segundo de criação |
| 0x10 | 2 B | Data de criação |
| 0x12 | 2 B | Data de acesso |
| 0x14 | 2 B | Os 2 bytes mais significativos do índice do *cluster* |
| 0x16 | 2 B | Hora, minuto, e segundo de modificação |
| 0x18 | 2 B | Dia de modificação |
| 0x1A | 2 B | Os 2 bytes menos significativos do índice do *cluster* |
| 0x1C | 4 B | Tamanho do arquivo em bytes |

Fonte: (CARRIER, 2005)

A extensão do arquivo pode também ser armazenado, possuindo um campo específico para os três caracteres de extensão. Neste caso, o ponto pode ser ignorado, mantendo se apenas o nome e a extensão do arquivo na entrada curta. Contudo, como a extensão pode ser tratada apenas como uma continuação do nome do arquivo, na entrada longa seus caracteres são armazenados de forma normal, incluindo o ponto prévio.

O primeiro caractere do nome curto pode indicar também o estado de alocação do arquivo. O significado dos valores é o mesmo da entrada longa.

O atributo da entrada define funções especiais do arquivo, com exceção do valor 0x0F, que representa uma entrada longa, como já visto previamente. O atributo 0x01 indica um arquivo de somente leitura. O valor 0x02 indica um arquivo escondido. 0x04 refere-se a um arquivo do sistema. 0x08 é um identificador de volume. 0x10 é uma pasta. E 0x20 é um arquivo normal.

Os campos de tempo e de data são subdivididos internamente, como visto na Tabela 8. O menos passo possível no campo do tempo é de 2 segundos. O valor do ano se soma a 1980, e como todos os valores são lidos como inteiros positivos, o ano máximo que um arquivo no sistema FAT pode ter indicado é 2107. O campo de centésimos de segundos varia de 0 à 199, e acrescenta uma precisão ao tempo de criação.

Tabela 8 – Campos de tempo, e data de uma entrada curta

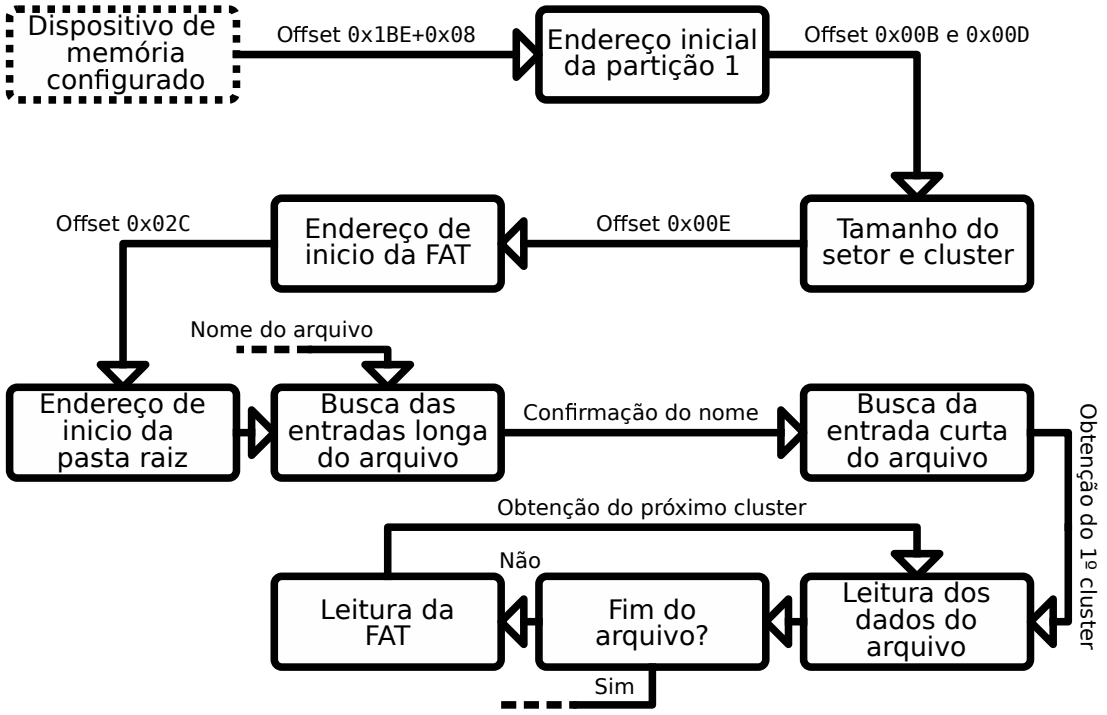
| 15 11 | 10 9 8 5 | 4 0 | **Tipo** |
|---|---|---|---|
| *Hora [0-23]* | *Minuto [0-59]* | *Segundo [0-29]* | Tempo |
| *Ano [0-127]* | *Mês [1-12]* | *Dia [1-31]* | Data |

Fonte: (CARRIER, 2005)

Para ser redirecionado para o endereço dos valores binários do arquivo, os segmentos do índice do *cluster* são unidos. Com este índice, e o tamanho do *cluster* obtido anteriormente, o endereço inicial em relação ao começo da partição é obtido, e além disto, os *clusters* seguintes podem ser obtidos pela FAT. E com o tamanho do arquivo, o programa pode realizar a leitura até o endereço final com informações relevantes.

O processo de leitura de um arquivo dentro do sistema FAT32 pode ser visto no fluxograma da Figura 19.

Figura 19 – Fluxograma da leitura de um arquivo em um sitema FAT32




Fonte: Autoria Própria (2022)

## 3.5 Arquivos externos

Os dados necessários para se renderizar um modelo tridimensional são suas posições de vértices, as coordenadas de mapeamento UV, e os seus vetores normais. Além disto, para a textura, é necessário as informações de colorações dos *pixels*, e o tamanho da imagem.

Estes dados podem ser supridos de forma direta para o processador, contudo, é mais prático enviar arquivos com compatibilidade para computadores. Deste modo os arquivos podem ser criados e editados por ferramentas externas.

Devido a velocidade de leitura dos arquivos não ser algo crítico, porque apenas necessita de uma única conversão para a memória RAM do dispositivo, a escolha dos formatos é dada pela compatibilidade com softwares externos, além da simplicidade de organização interna.

Para os modelos tridimensionais os arquivos do formato "`.obj`" permitem um armazenamento simples das informações necessárias, além da grande compatibilidade por softwares de modelamento. E para as imagens, o formato "`.ppm`" apresenta os valores dos *pixels* em um formato sem compressão, e permite edição via software de edição de imagem. Como não há geralmente uma nomenclatura oficial para cada formato de arquivo, estes serão aqui referenciados pelos suas extensões.

### 3.5.1 Arquivo para modelos 3D .obj

O arquivo do formato `.obj` permite o armazenamento de diversos parâmetros de um modelo tridimensional, e formas geométricas. Seu nome deriva de "arquivo de objeto". Neste trabalho será focado somente nas propriedades 3D relevantes.

O `.obj` é escrito em texto ASCII simples, podendo ser lido de forma fácil por diversas ferramentas computacionais. Cada linha indica um parâmetro do modelo, então seu conteúdo é facilmente divisível, contudo, a ordem dos parâmetros é relevante.

No início de cada linha, um conjunto de caracteres indica o tipo do parâmetro. O parâmetro "`o`" indica o início de um objeto, e é seguido de um texto ASCII o nomeando. O caractere "`v`" indica a posição de um vértice, seguido de três valores não inteiros separados por espaço, indicando as coordenadas tridimensionais dos pontos. O "`vt`" indica coordenadas do mapa UV, e também são representados por valores não inteiros separados por espaço, porém somente com dois itens. E "`vn`" representa os valores tridimensionais de vetores normais.

Estes valores se repetem quantas vezes forem necessário para representar todas os parâmetros do modelo, contudo, não é obrigatório a representação de duplicas em seus valores, a fim de evitar consumo desnecessário de memória. Para a reconstrução do modelo, os valores de posição, UV, e normal devem ser combinados para criar as faces do modelo. Para isto é utilizado o parâmetro "`f`", que aparece uma vez para cada face do modelo. Nestes os índices dos valores prévios são agrupados, com início em um. O índice é obtido pela posição do parâmetro no arquivo, contudo, contando apenas entre parâmetros similares. Para cada vértice de uma face os parâmetros são indicados no arquivo como `v/vt/vn`, sendo substituído cada um por seus respectivos índices. Os parâmetros de cada vértice são então separados por espaço, e sua contagem é do tamanho do polígono que está representando, o que tende a ser geralmente três ou quatro pontos.

Um exemplo de um modelo em `.obj` pode ser visto no Algoritmo 6.

### 3.5.2 Arquivo para texturas .ppm

Para armazenamento de imagens existem diversos formatos que são amplamente utilizados. Contudo, boa parte dos formatos atuais utilizam várias etapas de compressão, o que dificulta a leitura do arquivo. Como a textura será armazenada de forma completa, sem compressão, na memória RAM do dispositivo, por simplicidade, é buscado um formato que armazene as informações de mesmo modo, para facilitar a transferência de dados.

**Algorithme 6** Exemplo de um modelo 3D armazenado em `.obj`

```
o Quadrado3D
v -1.0 -1.0 0.0
v  1.0 -1.0 0.0
v  1.0  1.0 0.0
v -1.0  1.0 0.0
vt 0.0 1.0
vt 1.0 1.0
vt 1.0 0.0
vt 0.0 0.0
vn 0.0 0.0 1.0
f 1/4/1 2/3/1 3/2/1
f 1/4/1 3/2/1 4/1/1
```

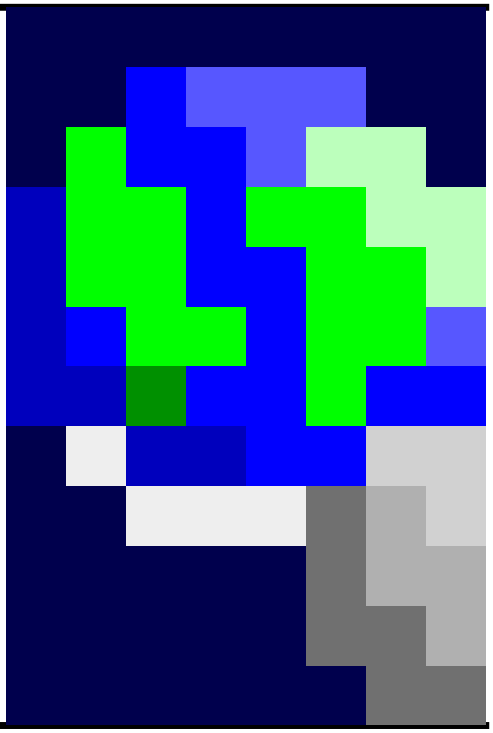
Por conta disto o formato utilizado é o `.ppm`. Nesta extensão os *pixels* da imagem são representados de modo inteiro, sem compressão. O formato `.ppm` é uma extensão antiga, que ainda possui suporte de ferramentas atuais pela sua simplicidade. Seu formato é um de um conjunto, sendo `.pbm` para representação de imagens puramente preto e brancas, `.pgm` para imagens com escalas de cinza de 8 bits em ASCII e 16 bits em valor binário, e `.ppm` com profundidade de 8 bits em ASCII e 16 bits em binário por canal de cor RGB. Neste caso, o formato `.ppm` será adotado, contudo, seu formato de organização é similar aos demais.

As primeiras linhas do arquivo podem ser lidas por uma ferramenta de edição de texto, porque as informações são providas em formato de texto ASCII. Na primeira linha, dois caracteres em ASCII identificam o formato de composição dos dados. Para uma imagem de cor de formato `.ppm` apenas dois valores se aplicam. Os caracteres `P3` identificam que os valores na seção de dados da imagens devem ser lidos como caracteres ASCII, e `P6` significa que os valores estão armazenados como valores binários.

As linhas que representam um comentário são indicadas pelo caractere "#". A próxima linha válida indica as dimensões da imagem em texto ASCII, com a dimensão no eixo "x" primeiramente, seguido da dimensão em "y", sendo os valores separados por espaços. E a última linha antes dos dados indica o valor máximo de cada cor.

A partir disto, nas próximas linhas válidas, os valores dos *pixels* da imagem são dispostos. Os valores devem ser interpretados como partindo do canto superior esquerdo, progredindo no sentido do eixo "x", e em "y" quando atingir os limites das dimensões da figura. Caso os valores estejam armazenados em ASCII, os valores devem ser separados por espaços ou quebras de linha. Um exemplo de um arquivo neste formato pode ser visto na Figura 20.

Figura 20 – Exemplo de um arquivo de uma imagem no formato .ppm

| Arquivo .ppm | Imagem |
|---|---|
| P3<br>8 12<br>255<br>0 0 77 0 0 77 0 0 77 0 0 77 0 0 77 0 0 77 0 0<br>77 0 0 77 0 0 77 0 0 77 0 0 255 87 87 255 87<br>87 255 87 87 255 0 0 77 0 0 77 0 0 77 0 255 0<br>0 0 255 0 0 255 87 87 255 188 255 188 188<br>255 188 0 0 77 0 0 189 0 255 0 0 255 0 0 0<br>255 0 255 0 0 255 0 188 255 188 188 255 188<br>0 0 189 0 255 0 0 255 0 0 0 255 0 0 255 0 255<br>0 0 255 0 188 255 188 0 0 189 0 0 255 0 255 0<br>0 255 0 0 0 255 0 255 0 0 255 0 87 87 255<br>0 0 189 0 0 189 0 144 0 0 0 255 0 0 255 0 255<br>0 0 0 255 0 0 255 0 0 77 238 238 238 0 0 189<br>0 0 189 0 0 255 0 0 255 209 209 209 209 209<br>209 0 0 77 0 0 77 238 238 238 238 238 238<br>238 238 238 112 112 112 176 176 176 209<br>209 209 0 0 77 0 0 77 0 0 77 0 0 77 0 0 77 112<br>112 112 176 176 176 176 176 176 0 0 77 0 0<br>77 0 0 77 0 0 77 0 0 77 112 112 112 112 112<br>112 176 176 176 0 0 77 0 0 77 0 0 77 0 0 77 0<br>0 77 0 0 77 112 112 112 112 112 112 |  |

Fonte: Autoria Própria (2022)

# 4 DESENVOLVIMENTO

A partir das bases teóricas e metódicas estabelecidas, o objetivo do trabalho pode ser construído. Para o auxilio no desenvolvimento, certos softwares computacionais foram utilizados para a formulação do trabalho, estes serão propriamente nomeados durante o capítulo.

A progressão textual das seções não segue necessariamente a evolução temporal do desenvolvimento. Devido a certas partes serem desenvolvidas em paralelo, e necessidade de retorno para algoritmos prévios para correção de erros, uma progressão cronologia dificultaria a leitura e entendimento do texto. Por conta disto, este capítulo adota uma divisão dos pontos mais próximos ao hardware até as fases totalmente aritméticas da renderização do modelo.

Os arquivos do trabalho estão também livremente disponíveis online[1].

## 4.1 Configurações de inicialização

O processador é um dispositivo capaz de realizar operações simples, em uma ordem sequencial. Esta ordem é dada pela sequência de instruções no código binário carregado no dispositivo. Este código binário pode ser carregado no dispositivo de certas formas diferentes, como por uma memória de apenas leitura inserida como componente físico na placa de circuito. Contudo, para microcontroladores, o método mais comumente utilizado é pelo envio do código por uma conexão física com um outro computador de maior complexidade.

Neste caso, o microcontrolador K210 é conectado fisicamente pela placa a uma memória flash, e este conjunto possui ligação a um chip para comunicação ao computador via protocolo USB. O código binário gerado no computador pode ser então descarregado ao microcontrolador. Cada marca de microcontrolador geralmente utiliza um software individual que realiza esta transferência, neste caso este é dado pela ferramenta de código livre Kflash[2] fornecida pelos próprios desenvolvedores do microcontrolador.

Para a criação do código binário é recomendado a utilização de uma linguagem de programação, e um software de compilação complementar. Neste caso a linguagem de programação é dada pelo RISC-V assembly. Os arquivos de código assembly são identificáveis pela extensão "`.s`", contudo, são apenas arquivos de texto ASCII, podendo ser utilizado qualquer editor de texto para desenvolvê-los. Neste caso o programa utilizado para criação e edição dos algoritmos foi o editor de código livre Vim[3].

Para a compilação do código, devido as particularidades de endereçamento e de comunicação com o computador, cada microprocessador geralmente apresenta compiladores específicos, geralmente derivados de softwares voltados para a respectiva arquitetura. O microcontrolador K210 também se enquadra neste caso. Com um compilador dedicado de

[1] Disponível em: https://github.com/luczis/Renderizacao-3D-RISC-V-Assembly
[2] Disponível em: https://github.com/sipeed/kflash_gui
[3] Disponível em: https://www.vim.org

código aberto, dado pela *toolchain* Kendryte para RISC-V[4], derivado do compilador GNU para o RISC-V de modo geral.

Devido aos diversos arquivos presentes de desenvolvimento, a compilação e o *link* do código de forma individual pode se tornar custoso. Por conta disto, existem softwares que são responsáveis por automatizar este processo, que neste caso são representados pelas ferramentas de código livre Make[5] e CMake[6]. O arquivo CMake utilizado é fornecido pelos desenvolvedores do K210, dados nos exemplos de projetos, apenas modificando algumas linhas para se adequar ao trabalho desenvolvido.

Para organização da ordem de alocação dos arquivos de código binário, além de definição das regiões e endereços de memória são dados em um arquivo de *linkagem*. Este arquivo é dado pela extensão "`.ld`", e é utilizado o fornecido previamente pelos desenvolvedores.

A organização interna do projeto pode ser visto na Tabela 9.

Tabela 9 – Organização interna dos arquivos do projeto

**Renderizacao3D_ASM**
- **build**
  - *Arquivos compilados*
  - Makefile
- **lds**
  - kendryte.ld
- **lib**
  - CMakeLists.txt
  - entry.S
- **src**
  - **inc**
    - *Arquivos de variáveis*
  - *Arquivos assembly*
- BuildCode.sh
- CMakeLists.txt

Fonte: Autoria Própria (2022)

### 4.1.1 Entry point

As instruções realizadas pelo processador são dadas de forma sequencial, contudo, existe um ponto de partida para qualquer algoritmo. Este código é desenvolvido para ocupar o minimo de espaço necessário, e realiza apenas operações básicas de configurações que geralmente são fundamentais para a operação do dispositivo. Como o contador de programa geralmente se inicia nulo, a primeira instrução lida é a presente na memória 0x0. O símbolo geralmente dado para esta posição é de `_start`, podendo ser redirecionado para este o código caso se deseje reiniciar o algoritmo.

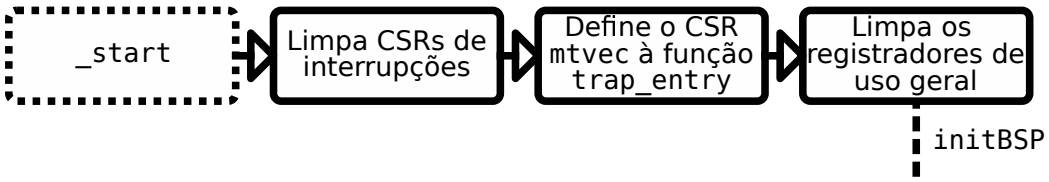
O arquivo que contém este código assembly esta em uma pasta individual, definido pelo nome "entry.S". Inicialmente o algoritmo limpa os registradores de uso geral e CSRs, além disto, aloca *flags* indicando quais núcleos estão ativos, armazenada no símbolo `g_wake_up`. A

[4]Disponível em: https://github.com/kendryte/kendryte-gnu-toolchain
[5]Disponível em: https://www.gnu.org/software/make
[6]Disponível em: https://cmake.org

partir disto, o algoritmo é redirecionado para a inicialização específica do microcontrolador, por uma instrução de *jump* para o símbolo `initBSP`.

Esta seção do código pode ser vista em forma de fluxograma na Figura 21.

Figura 21 – Fluxograma da função `_start`




Fonte: Autoria Própria (2022)

### 4.1.2 Board support package

Para cada microcontrolador uma rotina específica de instruções deve ser executada. Esta rotina é definida pelo código do pacote de apoio à placa, ou `Board Support Package` (BSP) em inglês. Em comparação com as demais linguagens de programação, esta é a rotina que redireciona o algoritmo para a função `main`, em um código escrito em C, a partir da onde o programador tem acesso ao dispositivo. A partir desta parte, o código é modificado para obter o resultado final desejado no trabalho.

Primeiramente o processador obtém o número do núcleo pelo CSR `mhartid`. Isto é importante, porque neste caso, os núcleos são segmentados para funções diferentes. Caso o núcleo seja o 1, é realizado um salto para a seção de inicialização específica, caso contrário, a sequência continua.

O núcleo 0 é o responsável por inicializar a região da memória `.bss`. Como comentado previamente, esta região da memória deve ser zerada e reservada, permitindo que símbolos de dados sejam definidos nesta região. Os endereços iniciais e finais são previamente definidos pelo compilador e *linker*. A partir disto, é apenas necessário alocar zeros para estes limites. Esta função pode ser vista no Algoritmo 7.

Após a inicialização do `.bss`, a FPIOA é inicializada para permitir que o processador interaja com o mundo externo. Nesta operação o *clock* central para os periféricos é também inicializado. Também é ativado a interrupção externas da FPIOA para os periféricos de SPI0 e SPI1, que serão utilizadas posteriormente para comunicação com os dispositivos externos.

Seguindo o algoritmo, o estado fonte do SYSCTL é limpa para que o periférico de gerenciamento possa ser iniciado. O controlador de interrupções de nível de plataforma, ou *Platform-Level Interrupt Controller* (PLIC) em inglês, é ativado no interior da função `initPLIC`. Em sequência, as interrupções são ativadas.

O núcleo 1 deve ser então configurado. Primeiramente as interrupções de nível *Machine* são ativadas para o núcleo 1. E como o núcleo 1 é energizado ao mesmo tempo que o 0, acaba

**Algorithme 7** Inicialização da seção `.bss`

```
1: .initBSS:
2:   addi sp, sp, -16
3:   sd ra, 8(sp)
4:
5:   la t0, _bss # Inicio da memoria BSS
6:   la t1, _ebss # Fim da memoria BSS
7:
8:   # Zera a região BSS da memoria
9: 1:
10:  sd x0, 0(t0)
11:  addi t0, t0, 8
12:  blt t0, t1, 1b
13:
14:  ld ra, 8(sp)
15:  addi sp, sp, 16
16:  ret
17: # end
```

por seguir o mesmo caminho de operação, partindo do *Entry Point* 0x0, contudo, os núcleos se divergem no início do BSP.

Nesta divergência, o núcleo 0 segue as rotinas descritas, porém o núcleo 1 apenas ativa seu respectivo controlador de interrupção pela função *initPLIC*. A partir disto, o núcleo 1 se mantém em um *loop* a espera da liberação da *flag* respectiva no símbolo `g_wake_up`.

O núcleo 0, quando terminado suas configurações, deve liberar a *flag* para o núcleo 1, contudo, como esta operação envolve um registrador que esta sendo lido por ambos os núcleos no mesmo tempo, é importante que seja utilizado instruções atômicas, para evitar colisões de acesso. Esta seção do código é dado no Algoritmo 8.

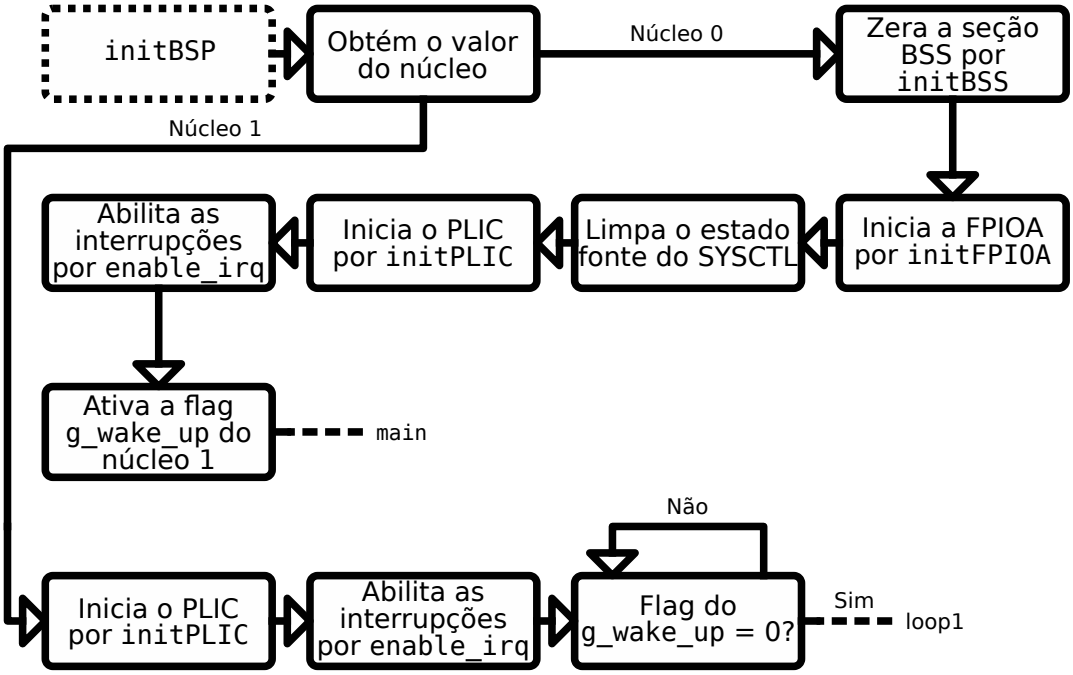
**Algorithme 8** Liberação do núcleo 1

```
1:   # Libera o nucleo 1
2:   la t0, g_wake_up
3:   addi t0, t0, 8
4:   li t1, 1
5: 2:
6:   lr.d t2, (t0)
7:   sc.d t2, t1, (t0)
8:   bne t2, x0, 2b
```

Após a configuração inicial dos núcleos, o algoritmo é redirecionado para configurações específicas dos periféricos. Enquanto o núcleo 0 realiza as demais configurações na função `main`, o núcleo 1 é redirecionado para sua função de renderização `loop1`, onde deverá esperar a sua respectiva *flag*.

O fluxograma do código está presente na Figura 22.

Figura 22 – Fluxograma da função `initBSP`




Fonte: Autoria Própria (2022)

### 4.1.3 Registradores de system control

Os registradores de controle do sistema são os responsáveis por organizar a operação dos periféricos, e principalmente, reger os diversos *clocks* presentes, para que todos os elementos necessários trabalhem em sincronia.

A primeira configuração a ser feita é a definição das frequências dos *clocks*, dadas pelos PLLs do sistema. O PLL principal, que está conectado à CPU, é o PLL0. Inicialmente é importante desconectar a CPU do PLL0, e conectá-la diretamente ao cristal oscilador. A partir disto, a saída do PLL0 é desligada, e sua alimentação é desligada. O novo valor de frequência é então armazenado no registrador específico. Neste caso, a frequência é de 400 MHz.

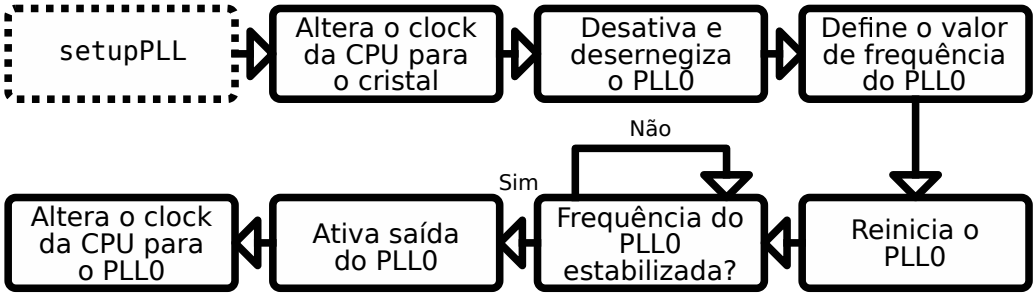
Após isto, o PLL0 é novamente alimentado, reiniciado, e o algoritmo espera até a frequência se estabilizar no valor desejado. Assim a saída pode ser conectada novamente na CPU, e nos demais periféricos que utilizem este barramento. O fluxograma desta função pode ser visto na Figura 23.

## 4.2 Configurações de dispositivos externos

Além do microcontrolador, exitem conectados a este um cartão micro SD e um *display* TFT. Após a configuração base da CPU, a configuração destes pode ser realizada. O principal fator desta etapa, é a configuração dos componentes periféricos do microcontrolador, que serão os responsáveis por permitir esta comunicação com os dispositivos externos.

O K210 apresenta diversos periféricos, contudo, nem todos possuem utilização neste

Figura 23 – Fluxograma da função `setupPLL`




Fonte: Autoria Própria (2022)

caso. Para este trabalho, se utiliza os periféricos de SPI, para comunicação com os dispositivos externos, o controlador de DMA, para otimização da transmissão de dados via SPI, e o GPIO, que permite a comunicação dos componentes internos com o mundo físico.

As operações de configuração dos periféricos estão presentes nas funções `tftSetup` e `sdSetup`, para o *display* TFT e cartão SD, respectivamente. Estas não possuem sequencialidade direta no código, contudo, textualmente, são discutidas de forma direta, para após ser tratado do aspecto da comunicação ligada puramente ao software.

### 4.2.1 Display TFT

O microcontrolador K210 possui certas peculiaridade para a comunicação com o *display*. O K210 possui a capacidade de conexão direta com uma câmera de vídeo, e está, para fins de otimização, pode ser conectada diretamente para o *display*, alterando apenas alguns *pixels* da imagem, o que acaba evitando o trafego de todos os dados pela memória e CPU. Contudo, neste caso isto não é utilizado. Por conta disto, o caminho da conexão de vídeo é redirecionado para a memória. Isto é realizado ao setar o bit 10 do registrador de configurações miscelâneas do SYSCTL.

Para a configuração do *display* TFT deve ser também abordado a comunicação, que é realizada via SPI. Para a SPI, o primeiro passo é configurar os pinos de saída físicos pela FPIOA.

Todas as operações de alocação de pinos são realizadas pela função `setupFPIOA`, em que é passado o número do pino, e o valor de configuração de 32 bits a ser armazenado. O número do pino reflete o *offset* do endereço base da FPIOA. O valor a ser armazenado apresenta diversos campos de configuração, regulando itens como definição de entrada ou saída, corrente de saída, e seleção de canal.

Na seleção de canal cada função interna do microcontrolador com capacidade de comunicação externa é alocada. No caso do *display* TFT, estas funções são a de pino de seleção, e *clock*, ambos da SPI0, sendo os demais 8 pinos de dados já alocados por hardware, sem necessidade do FPIOA. Além disto, é anexado dois pino às GPIOs, sendo um para a

informação de dado ou comando, e um outro para *reset*. Neste caso, as GPIOs utilizadas são do barramento de alta velocidade.

Após as configurações dos pinos, o *display* é *resetado* pela saída especificada. Isto é feito pela função `outputGPIOHS`, ao alterar os bits da região de memória específica, e que pode ser vista no Algoritmo 9.

**Algorithme 9** Alteração do valor de saída de uma GPIO de alta velocidade

```
1: .globl outputGPIOHS
2: # a0 - Mascara
3: # a1 - Saida
4: outputGPIOHS:
5:   addi sp, sp, -16
6:   sd ra, 8(sp)
7:
8:   li t0, GPIOHS_BASE_ADDR
9:   lw t1, GPIOHS_OUT_VAL(t0)
10:  not t2, a0
11:  and t1, t1, t2
12:  and t2, a0, a1
13:  or t1, t1, t2
14:
15:  sw t1, GPIOHS_OUT_VAL(t0)
16:
17:  ld ra, 8(sp)
18:  addi sp, sp, 16
19:  ret
20: #end
```

O *display* é mantido desligado enquanto a SPI é configurada. O primeiro passo é a limpeza dos registradores para evitar configurações salvas em outras compilações. Cada módulo de SPI do microcontrolador possui um subdivisor interno, que é conectado ao barramento do PLL0. Neste caso a frequência desejada é de 10 MHz, então este divisor é definido como 40. E o tamanho do quadro de transferência é definido como 8 bits, ou seja, a quantidade de dados que é transmitida em um ciclo de *clock* da SPI0. Então, o pino de *reset* é setado novamente, e o *display* se inicia. O dispositivo está pronto para receber comandos.

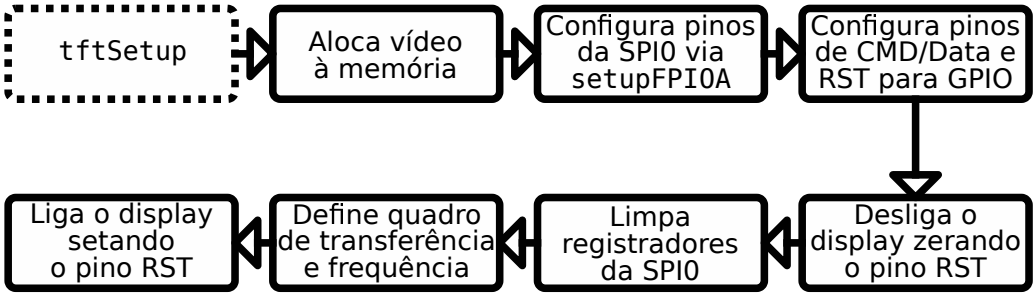
O fluxograma desta parte do código é visto na Figura 24.

### 4.2.2 Cartão SD

A configuração do dispositivo de armazenamento externo, neste caso um cartão micro SD, é realizada de modo similar ao *display*. A comunicação é realizada de mesmo modo, pela SPI. Contudo, para este dispositivo é utilizado o SPI1, o que permite que os dispositivos operem em paralelo.

Os pinos são configurados via FPIOA, e neste caso, os pinos de dados MISO e MOSI são também definidos.

Figura 24 – Fluxograma da função `tftSetup`




Fonte: Autoria Própria (2022)

A configuração do SPI é feita de modo similar, limpando os registradores prévios, e definindo a frequência de operação. Neste caso, a frequência se inicia em 100 KHz, como visto na Subsubseção 3.3.1.2, por razões de compatibilidade. Para isto, é utilizado um fator de divisão de 2000. O quadro de transferência é mantido como 1 bit, o que significa que o SPI está operando em condições padrões, com um barramento MISO em paralelo a um MOSI.

O fluxograma desta etapa de configuração pode ser vista na Figura 25.

Figura 25 – Fluxograma da função `sdSetup`


Fonte: Autoria Própria (2022)

## 4.3 Comunicação

A configuração dos periféricos é necessária para que a CPU possa interagir com os dispositivos externos. Contudo, cada dispositivo apresenta uma forma diferente de se comunicar, utilizando seu próprio protocolo, sendo a SPI apenas o meio. Para que seja possível a configuração do *display* e do cartão SD, é necessário o desenvolvimento de um algoritmo com capacidade de compreensão de seus respectivos protocolos, para que as informações sejam transmitidas de forma correta.

### 4.3.1 Comunicação com o display

O *display* TFT ST778V conectado ao microcontrolador possui um protocolo de comunicação específico, como já descrito na Subsubseção 3.3.1.1, e seu método de comunicação abordado é unidirecional, do microcontrolador para o *display*.

Devido a grande quantidade de dados que são enviados para o preenchimento da tela, esta comunicação é realizada de forma direta da memória RAM à saída SPI, sem necessidade

de ocupação da CPU. Contudo, para controle do fluxo de dados acaba sendo necessário a utilização do controlador de DMA, que gere o uso dos barramentos de dados. Além disto, caso a CPU necessite acessar um endereço próximo na RAM ao mesmo tempo que a SPI, pode acabar acarretando em uma espera para a tarefa, mesmo com canais distintos. Uma solução para este problema é visto ao abordar as etapas de renderização na Seção 4.6.

Para o envio de dados do microcontrolador ao *display* a função `tftWriteDMA` é utilizada. Nesta é definido o endereço e tamanho da memória à enviar, se o *display* deve ler os valores como dados ou comando, e o tamanho dos dados a ser enviado entre 8 ou 16 bits.

Esta definição de tamanho é importante, porque os comandos para o *display* são de 8 bits, contudo, os *pixels* serão da profundidade RGB565, ou seja, 5 bits para vermelho, 6 bits para verde, e 5 bits para azul, o que totaliza 16 bits. Como o DMA, neste caso, é definido para enviar dados discretos de 32 bits, os endereços enviados para a saída da SPI0 devem ser múltiplos de 4 bytes. Por conta disto, este valor do tamanho dos dados é definido para a SPI0 reconhecer a quantidade de bits a desconsiderar do valor transferido para seu *buffer*.

Com estas informações, a SPI0 é configurado para modo de transmissão. A alteração no registrador de configuração é realizada toda vez que a função de escrita para o *display* é requisitada. Isto é necessário para garantir que o formato dos dados seja corretamente definido. Após isto, é definido o fluxo de dados via DMA para a SPI0, e esta interface é ativa em seguida. Com a SPI0 ativa, o periférico se mantem pronto e a espera para os dados vindos pelo DMA.

Como internamente o controlador de DMA possui conexão com diversas portas para os dispositivos internos, o caminho desejado deve ser também configurado. Para a realização da transferência, o canal definido para o acesso da SPI0 é configurado para realizar o *handshake* com o controlador de DMA. Este *handshake* é realizado com a porta de transferência da SPI0, para garantir que a transmissão seja realizada quando o periférico estiver disponível, para evitar a lotação de um canal de modo desnecessário. A se notar que para a recepção de dados via SPI, outra porta deve ser utilizada.

O registrador de interrupções do canal de DMA é limpo, e sua saída é requisitada a ser desabilitada. Este requerimento é feito limpando o bit do canal desejado no registrador de ativação do canal de DMA. O bit permanece ativo enquanto o DMA esta em outra transferência, e neste ponto o processador deve esperar o controlador de DMA zerar o bit desejado. Neste caso, o algoritmo se mantém em um *loop* até a liberação do canal.

Com o canal liberado, é necessário atualizar os parâmetros do canal para a nova transferência. A transferência é definida como da memória para o periférico, via controlador de DMA. Nota-se que a transferência pode ser realizada também entre dois periféricos, e entre dois diferentes endereços da memória. O *handshake* é definido entre um dispositivo acessível da CPU, neste caso a RAM, para um externo, neste caso a SPI0. E a fonte e o destino são definidos no mesmo canal de DMA, neste caso, o canal 3.

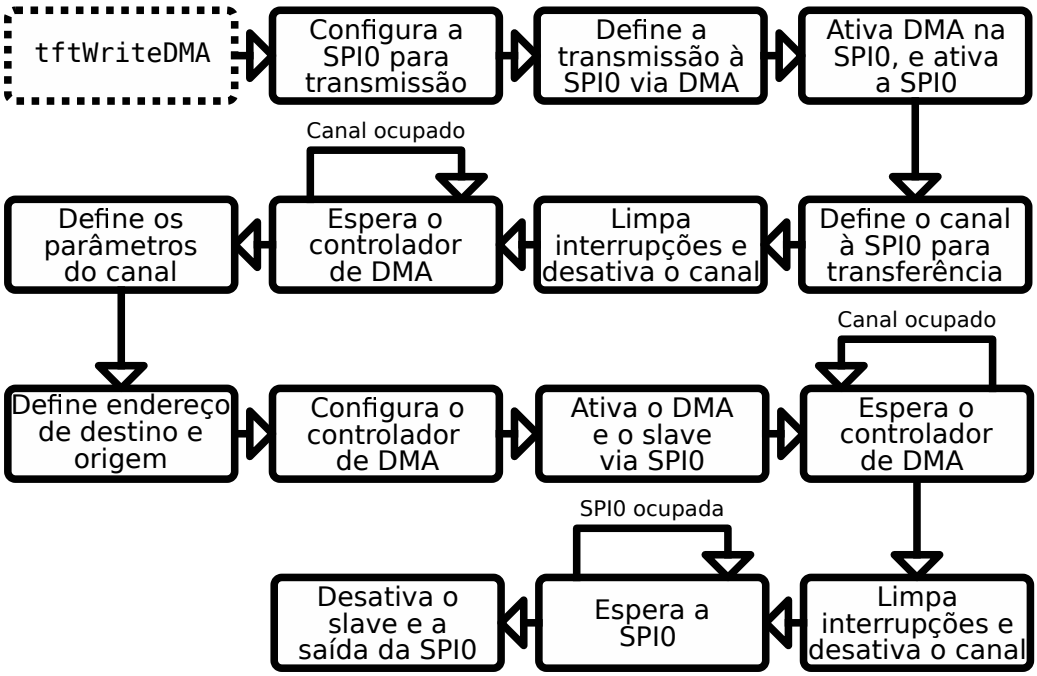
O endereço de origem é então definido, que neste caso é definido como parâmetro da função. Juntamente, é definido o endereço de destino, sendo este o *buffer* de dados da SPI0.

Após isto, é definidos os parâmetros para o controlador de DMA. O destino, ou seja, a SPI0, é definida como mestre da transferência. O endereço de origem é definido como incremental, e o de destino, definido como fixo. O incremento na RAM é necessário para transmitir todos os dados que são armazenados em endereços sequenciais, porém o *buffer* da SPI possui endereço fixo, e serve apenas como ponto temporário de redirecionamento para os pinos de saída. O tamanho da transferência é definido como 32 bits para destino e origem, restringido a este pelo tamanho fixo do *buffer* da SPI. E a transferência é configurada para ser realizada em *burst* de 4 dados. Por fim, é definido o tamanho da transferência, dado como parâmetro da função.

O controlador DMA é ativo, seguido do canal definido e do *slave* da SPI0, que neste caso é o *display* TFT. Deste modo a transferência DMA é realizada. Neste ponto o núcleo pode ser redirecionado para outra tarefa, contudo, neste caso como o núcleo 0 é responsável apenas pela transferência do *framebuffer* para o *display* foi decidido mantê-lo em espera.

Com a transferência concluída, a interrupção do controlador de DMA é limpa. A saída da SPI0 e o selecionador de *slave* são então limpos, e a função é finalizada. O fluxograma desta função é visto na Figura 26.

Figura 26 – Fluxograma da função `tftWriteDMA`




Fonte: Autoria Própria (2022)

Para a configuração do *display* TFT é seguido a ordem previamente vista na Subsubseção 3.3.1.1. A região de renderização é definida como 320x240 *pixels*, sendo estes os limites da tela. As dimensões da região de renderização são armazenadas em 32 bytes alocados no símbolo `tft_display_size`, devido a necessidade deste valor em todo ciclo de renderização. Para o envio dos comandos é utilizado um endereço de 4 bytes predefinido, com o símbolo `tft_cmd`.

O *framebuffer* é armazenado em dois símbolos diferentes, para permitir uma operação paralela dos núcleos, sendo estes o `frame_buffer0` e `frame_buffer1`. Cada um ocupa um espaço de 307200 bytes, alocando 4 bytes para cada um dos 76800 *pixels*, devido ao tamanho de transferência, totalizando 600 KB em ambos os *framebuffers*. A região entre *pixels* é posteriormente utilizada na etapa de renderização. Estes símbolos citados foram definidos na seção `.bss` pelo diretivo `.comm`, então estes valores de ocupação são refletidos somente na memória RAM.

### 4.3.2 Decodificação do cartão SD

Para a comunicação com o cartão SD a SPI é utilizada. Contudo, neste caminho os dados transmitidos e recebidos devem obedecer um protocolo de códigos. A comunicação, diferentemente do *display*, é realizada de forma bidirecional.

As funções utilizadas são a `sdSPIwrite` e `sdSPIread`, respectivamente, para a escrita e leitura em relação ao microcontrolador. Além disto, para o envio de dados e comandos é necessário enviar o CRC da mensagem.

Para a escrita, a operação é similar ao *display*. A comunicação neste caso é realizada utilizando a SPI1. Com isto, a primeira operação é a definição dos parâmetros da SPI1, e defini-la para transmissão. Neste caso a transmissão é feita por um único caminho de ida e outro de volta, o MOSI e MISO respectivamente, então as configurações não são alteradas. O que acaba sendo modificado no registrador de controle é a definição de transmissão do microcontrolador para o cartão.

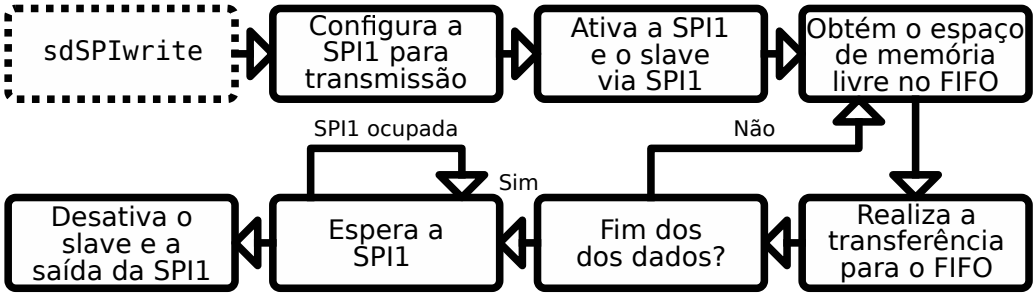
Após a definição dos parâmetros, a SPI1 é ativada, seguindo do *slave* do cartão. É obtido também o valor livre do *buffer* de memória "primeiro a entrar, primeiro a sair", ou *First In, First Out* (FIFO) em inglês. Assim os dados são transmitidos para o endereço do FIFO até a sua lotação. Caso não haja memória livre, o algoritmo repete a leitura de espaço até ser possível a armazenagem. Caso o tamanho do FIFO seja maior do que a quantidade de dados a armazenar, o algoritmo restringe para o menor valor. Enquanto é realizado a transferência para o FIFO, a SPI1 transmite os dados para o cartão SD.

Quando finalizado a transferência total para o FIFO, é esperado a SPI1 finalizar a transmissões de dados restantes. Com isto a SPI1 e o selecionador de *slave* são desativados. O envio de dados pode ser visto na Figura 27

Para a leitura, os parâmetros da SPI1 é definido de forma similar, apenas com a identificação de recebimento de dados. Além disto, é definido no registrador especifico o tamanho de dados a ler.

Com isto a SPI1 é ativa. O *buffer* do FIFO também é limpo, para evitar a interpretação errada de valores previamente armazenados. O *slave* é então habilitado. Durante esta operação o algoritmo opera em *loop*, verificando a lotação do FIFO de recebimento, e a medida que os dados são recebidos, o processador carrega os valores no endereço definido nos parâmetros da função. Caso a quantidade de dados armazenados no FIFO seja maior do que o requisitado, a transferência é restringida no menor valor. Isto é um caso que não deve ocorrer, porque a SPI

Figura 27 – Fluxograma da função `sdSPIwrite`

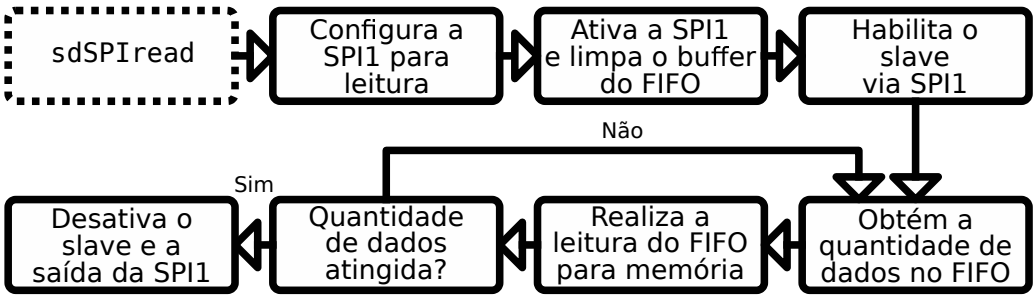



Fonte: Autoria Própria (2022)

deve desligar o *slave* em tempo suficiente, porém, em altas frequências de transferência, pode acabar sendo lido o canal MISO por alguns ciclos além.

Atingindo o limite de dados requisitados, a SPI1 e o selecionador de *slave* são desativados. Este recebimento pode ser visto na Figura 28. Durante a leitura dos dados do cartão, a leitura de setores é uma operação recorrente, por conta disto, a função `sdSectorRead` compromete o envio de todos os comandos necessários, além da leitura e armazenamento dos dados.

Figura 28 – Fluxograma da função `sdSPIread`




Fonte: Autoria Própria (2022)

#### 4.3.2.1 Verificação CRC

A estrutura de um comando via protocolo SD foi previamente decomposto na Tabela 5. Os campos de índice e argumento tendem a ser valores diretos, ou seja, não é necessário nenhuma operação para defini-los, apenas a necessidade do comando buscado. Contudo, o campo de CRC necessita de uma operação lógica envolvendo os demais valores da mensagem. Por conta disto, é utilizado a função `crc7get` para cada comando enviado ao cartão.

Esta função é apenas chamada no envio, para a leitura não há nenhuma verificação imposta. Isto pode ser realizado pelo valor do CRC7 enviado juntamente a respostas de alguns comandos, ou o CRC16 enviado após um bloco de dados. Contudo, devido ao algoritmo não realizar nenhuma tarefa delicada, e da proximidade do cartão em relação a CPU, assume-se

que não a perda de dados na comunicação. Por conta disto, a verificação é realizada apenas para o aceite dos comandos pelo cartão SD.

Para a otimização da geração do código verificador, uma tabela previamente calculada é utilizada, permitindo assim que a verificação seja realizada em conjunto de 8 bits. Esta tabela pode ser gerada por outra linguagem de programação, sendo um pseudo algoritmo de calculo de um verificador CRC.

## 4.4 Decodificação da FAT

A comunicação com o cartão SD é necessária para a leitura dos arquivos, porém, apenas com este protocolo de interface, não é possível obter os dados relevantes. Devido a segmentação interna da memória pelo sistema FAT, é necessário um encaminhamento até a posição correta dos arquivos. Neste caso, considera-se que o cartão foi formatado em FAT32, sendo impossibilitado a leitura em outro formato.

Como visto previamente, na Subseção 3.4.1, o primeiro setor a ser lido é o 0, e caso corretamente formatado, deve ser a posição do MBR. Utilizando a função `sdSectorRead`, passando o símbolo `fatBuffer`, e o argumento 0, deve ser lido o MBR do cartão. A partir disto, é feito a verificação pela assinatura de inicialização.

Utilizando o *offset*, a entrada da partição 1 é buscada. Deste endereço, é acrescentado um deslocamento para a obtenção do endereço do primeiro setor da partição, e em sequência, o seu tamanho. Além disto é buscado, no respectivo *offset*, o identificador do sistema FAT32.

Com o endereço correto, é feito a leitura do setor de inicialização da partição. O tamanho do setor é verificado, garantindo que seja de 512 bytes, e é obtido a quantidade de setores em um *cluster*. Para obter o início da tabela de alocação, é acrescentado o tamanho da região reservada ao endereço de início. Com isto, o endereço é armazenado para que possa ser facilmente acessado posteriormente. A contagem de tabelas e seus tamanhos são também obtidos.

A posição da pasta raiz é obtida multiplicando o tamanho das tabelas de alocação por seus tamanhos, e somando o endereço à base da partição. O endereço base da pasta raiz é também armazenado. Com a FAT, os *clusters* da pasta raiz são seguidos para obter o seu tamanho máximo, para definir o limite máximo de busca de uma entrada.

Deste modo, com a pasta raiz e a FAT definida, os demais arquivos podem ser obtidos pelo seu nome ASCII. Para isto, uma nova função, `readFile`, é utilizada para obter as informações de um arquivo dado um vetor de caracteres.

## 4.5 Leitura de arquivos

Por causa da organização da FAT32, os dados dos arquivos estão segmentados em diversos blocos internos a memória do cartão, podendo estes serem não sequenciais. Para esta reconstrução, é necessário a utilização da função `readFile`. Em seus argumentos, é passado o

endereço de memória interno do microcontrolador a ser armazenados os dados reconstruídos, e o vetor de caracteres com o nome do arquivo incluindo sua extensão. Por questões de simplicidade, não é possível a escalada de pastas por esta função, permitindo apenas a leitura de arquivos na pasta raiz. Como a etapa de leitura é realiza de arquivos é realizada antes da renderização, o `frame_buffer0` é utilizado como *buffer* temporário dos dados dos arquivos.

Para obter a posição do arquivo, primeiramente é lido o setor da pasta raiz. A primeira entrada é lida, e é buscado o início das entradas longas do arquivo, utilizando como base o vetor de caracteres. Caso a entrada verificada não seja do tipo longa, o algoritmo adiciona um *offset* de 32 bits, o tamanho de uma entrada, e repete a verificação. Como a codificação via Unicode é uma extensão do conjunto ASCII, os caracteres podem ser comparados de forma direta, apenas desconsiderando o segundo byte de cada letra da entrada. Caso todos os caracteres da primeira entrada longa tenham sido lidos sem distinção, o algoritmo é redirecionado para a entrada curta. Neste caso o código foi desenvolvido para apenas ler a primeira entrada longa, por questões de simplicidade. Sendo assim, se assume um nome de arquivo com a sua extensão menor do que 14 caracteres. Caso passado dos 512 bytes do setor, o próximo setor é lido, atentando-se ao tamanho máximo da pasta raiz, caso atingido o seu limite, assume-se a inexistência do arquivo no cartão SD.

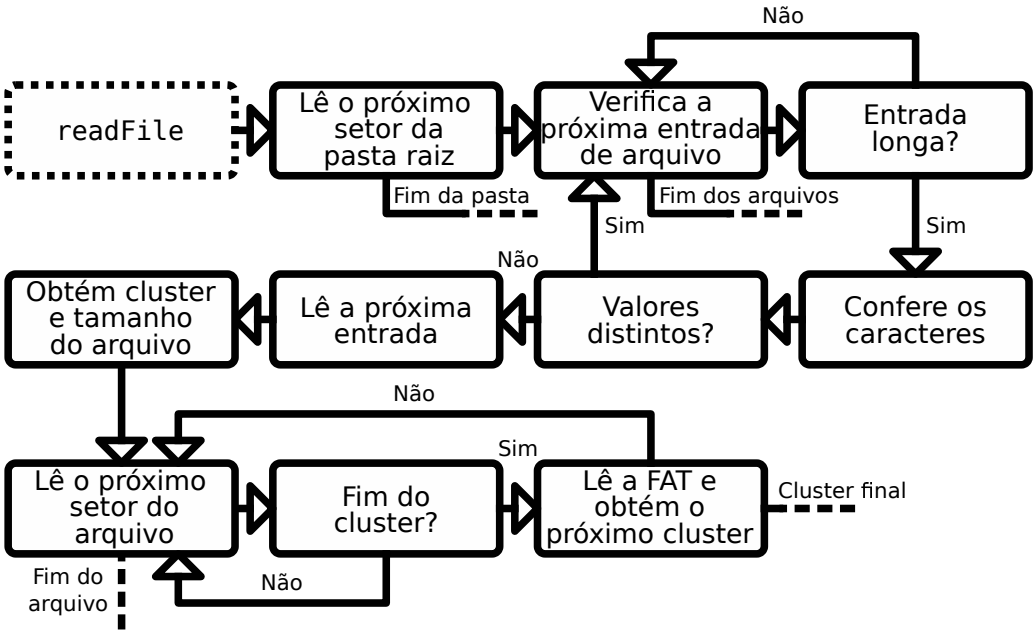
A entrada subsequente da longa, é considerada como a entrada curta. Isto geralmente pode ser dado como verdade, exceto em casos de exclusões de arquivos, e uma superlotação do *cluster* reservado para a pasta raiz, nesta condição as entradas excluídas são sobrepostas por novas, o que pode gerar saltos entre as entradas de um mesmo arquivo. Na entrada curta é extraído o índice do *cluster* inicial do arquivo, pelos 4 bytes segmentados em 2 *offsets*. É obtido também da entrada curta, o tamanho do arquivo. Como os dados a serem lidos são armazenados no `frame_buffer0`, a ocupação máxima possível sem que haja sobreposição de outras regiões da memória é de 300 KB, porém, como o símbolo de `frame_buffer1` é contíguo, isto permite uma região alocável de 600 KB de arquivo. Essa região única da memória é garantida por ambos os símbolos serem alocados em sequência na seção `.bss`, e devido ao fato de não haver gerenciamento dinâmico da memória RAM. Arquivos maiores não são lidos, e a função retorna como falha.

Os setores são lidos do primeiro *cluster*, e armazenados no *buffer* de dados. Caso o tamanho total seja atingido, a função é finalizada. Caso o próximo *cluster* seja necessário, a FAT é lida, e, a partir do índice do primeiro *cluster*, o próximo é encontrado. Com isto, a leitura dos setores é retomada, e isto se repete até atingir o *cluster* final. De modo compacto, a operação de leitura do arquivo pode ser visto na Figura 29.

### 4.5.1 Arquivos .obj

Com os dados do arquivo transferidos para a memória RAM do microcontrolador, as informações relevantes podem ser obtidas. No caso do modelo 3D, estes dados são as informações dos *vertexes*. Contudo, como visto previamente, estes dados não são dispostos de

Figura 29 – Fluxograma da função `readFile`




Fonte: Autoria Própria (2022)

forma direta, necessitando de uma interpretação por algoritmo. Neste caso, esta operação é realizada pela função `openOBJModel`.

A posição da leitura do arquivo é definida pelo valor do endereço, a medida que os caracteres são lidos o valor do endereço é incrementado em um byte, e este valor é passado a diante nas funções, de modo que cada operação inicie a partir do ponto final prévio. Em certos pontos, quando se deseja manter uma posição prévia, o endereço atual é copiado para um registrador temporário, e neste é realizado a operação.

O arquivo `.obj` é disposto em formato de texto ASCII, de modo que cada dado está disposto em uma linha. Isto permite uma fácil segmentação das informações de modo a processar cada valor individualmente. Deste modo, o algoritmo realiza a busca do caractere de quebra de linha "\n", representado pelo hexadecimal 0x0A. Além disto, caso seja encontrado o caractere "\0" de fim do arquivo, *End Of File* (EOF) em inglês, definido pelo hexadecimal 0x00, indica que o arquivo chegou em seu final, e a função acaba.

Inicialmente, é realizado uma contagem da quantidade de vértices, UVs, normais, e polígonos, para garantir que o modelo não seja muito complexo, e possam ser armazenados nos espaços reservados, sendo limitado, neste caso, todos os valores a 500. Para esta contagem é utilizada a função de declaração local `getNewLine`, em que o endereço é deslocado por byte até que se encontre o caractere de fim de linha, EOF, ou caso o primeiro caractere da linha seja "v" ou "f". É importante esta função ser chamada com o endereço do arquivo no início de uma linha, caso contrário, a função pode detectar um caractere como valor de importância, e será considerado na contagem do limite dos modelos.

Ao retornar da função `getNewLine`, o endereço atual do *buffer* do arquivo estará na

posição de início de linha, ou em um caractere de EOF. Deste modo, pode ser definido do que se trata o dado. Para um valor de "v" na primeira posição, caso o caractere seguinte seja um espaço, com valor hexadecimal de 0x20, os valores da linha representam uma posição de um vértice, caso seja "t", indica valores de coordenada UV, e caso seja "n", é valores de um vetor normal. Além disto, caso o primeiro caractere seja "f", os valores da linha indicam uma face de um polígono. Deste modo a contagem de cada um destes itens é realizada até o EOF. Caso os valores estejam dentro dos limites, o código prossegue com sucesso, caso contrário, é retornado um erro.

Com a quantidade de valores definida, o algoritmo retorna o endereço ao início do arquivo para poder obter os respectivos dados. A primeira função de obtenção é a `loadPolygons`, em que os índices das faces são carregados para a memória RAM. A função realiza opera em *loop*, obtendo cada linha do arquivo pela função `getNewLine`. Caso o primeiro caractere da linha seja "f", indica que os valores são de uma face de um polígono, e a função realiza essa leitura. Utilizando a função `ignoreSpaces` o endereço é deslocado para o primeiro caractere diferente de espaço. Caso o arquivo `.obj` esteja propriamente formatado, o caractere após o espaço deverá ser um número que indica o índice de posição do primeiro *vertex* da face.

Para obter o valor em ASCII e convertê-lo para inteiro, a função `getNumberFromFace` é utilizada. Nesta é definido um caractere de parada como parâmetro. Caso seja lido o índice da posição do vértice, ou das coordenadas UV, este caractere é a barra comum, indicado pelo caractere "/", ou 0x2F em hexadecimal. Caso seja lido o índice do vetor normal, este caractere é o espaço. Esta distinção é realizada para evitar que espaços entre os índices adicionados para identações aplicadas por questões estéticas sejam interpretados como separador de números.

A função `getNumberFromFace` realiza então o deslocamento dos endereços, e os caracteres ASCII são lidos. Como em ASCII todos os números de zero à nove são representados como 0x3n em hexadecimal, para conversão de ASCII para inteiro basta subtrair o caractere "0", ou em hexadecimal, 0x30. É utilizado um registrador temporário que é multiplicado por 10 para cada número encontrado em ASCII, e o valor do próximo caractere é somado a este. Isto é repetido até se encontrar o caractere de parada. Com o número final inteiro, o valor é subtraído em 1. Isto é realizado devido ao valor da indexação do `.obj` iniciar em um, porém, para *offset* de endereços, a contagem a partir de zero permite um mapeamento direto.

Esta operação é realizada para todos os índices do primeiro *vertex*, armazenando-os nos símbolos `pos_index`, `uv_index`, e `norm_index`, para os valores de posições dos vértices, posições UVs, e direções das normais, respectivamente. Com isto, é utilizado a função `ignoreSpaces` novamente, para obter o início do próximo *vertex*. Isto se repete novamente, totalizando três *vertexes*. Neste ponto ocorre uma verificação, caso o polígono seja triangular, e não há mais dados na linha, os valores do terceiro *vertex* são copiados para a posição do quarto, o que simplificará a etapa de renderização. Caso possua quatro posições, estes dados são também lidos. O endereço retorna então ao início do *buffer* do arquivo.

Após os índices, os próximos valores a serem carregados são o de posição dos vértices,

pela função `loadVertices`. A função `getNewLine` é chamada até que os dois primeiros caracteres sejam "v" seguido por um espaço. Sendo o caso, a linha é lida, utilizando a função `ignoreSpaces` para deslocar o endereço para o primeiro número, sendo este o valor do eixo "x". Como o número é dado em ASCII, a função `getFloatFromValue` lê os caracteres até um espaço ou nova linha.

A função `getFloatFromValue` verifica se o primeiro valor é um sinal de menos, representado pelo caractere "-", ou 0x2D em hexadecimal. Caso seja, um registrador é tratado como *flag*, para posteriormente negativar o número final. Com isto, o algoritmo entra em um *loop*, deslocando o endereço e obtendo os valores a esquerda do ponto. Para cada caractere de número encontrado o valor do número é somado ao registrador, o qual é multiplicado por 10 em cada iteração. Caso seja encontrado um ponto, representado por 0x2E em hexadecimal, o *loop* é quebrado. O valor inteiro é então convertido a ponto flutuante de 32 bits.

Para a parte a direita do ponto, a operação realizada é a mesma, com a multiplicação por 10 em cada iteração. Contudo, para manter a informação da quantidade de casas deslocadas, um registrador temporário é multiplicado na mesma proporção. Ao fim do valor a parte a direita é dividida pelo registrador temporário, após ambos serem convertidos a ponto flutuante, obtendo a parte fracionária do número, que é adicionada a parte inteira obtida anteriormente. E neste ponto, caso a *flag* de número negativo esteja ligada, é utilizada a instrução `fneg.s` para negar o valor final.

Este valor é armazenado no símbolo `pos`, que guarda as informações de todos as posições de vértices do modelo. Isto é repetido para os valores em "y" e "z", e a função itera todas as linhas até o final do arquivo.

O endereço é retornado ao início do arquivo, e a rotina se repete de forma similar para as coordenadas UVs, e normais. Para os valores UVs a função utilizada é `loadUVs`, neste caso os caracteres buscados são "vt", e se lê os valores das coordenadas "u" e "v", armazenando no símbolo `uv`. Para as normais, a função é `loadNormals`, com a leitura das coordenadas "x", "y" e "z", armazenadas em `norm`.

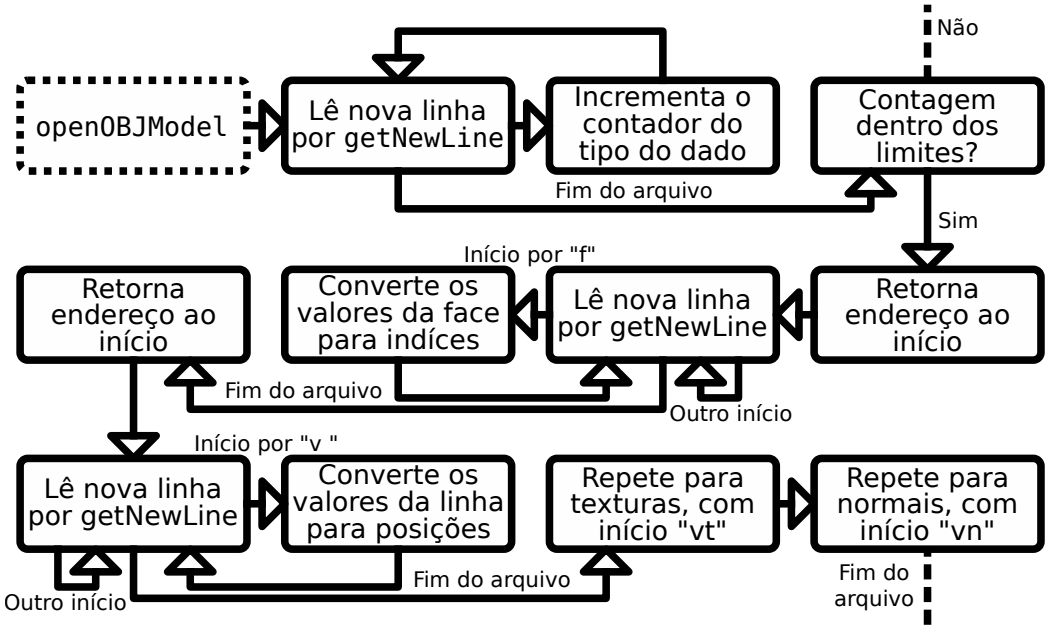
Deste modo todas os dados necessários do modelo são carregados na memória RAM. A função retorna com sucesso, e os dados no *buffer* do arquivo podem ser sobrescritos. O fluxograma desta função é visto na Figura 30.

### 4.5.2 Arquivos .ppm

O arquivo de textura também necessita de certa leitura pelo algoritmo, contudo, os dados relevantes tendem a ser dispostos de forma mais direta. Com o arquivo `.ppm` carregado no *buffer* do arquivo na memória, a conversão pode ser iniciada pela função `openPPMTex`.

A primeira etapa na leitura é a garantia que os dados são realmente de um arquivo `.ppm`. Isto é feito ao verificar que o primeiro caractere da primeira linha não comentada seja "P". Para obter este endereço é utilizado a função de definição local `getNewLine`. Nota-se que esta função, apesar de ter o mesmo símbolo, é definida em outro arquivo do para leitura do

Figura 30 – Fluxograma da função `openOBJModel`




Fonte: Autoria Própria (2022)

arquivo `.obj`, e não possuem o diretivo de alocação global `.global`. Isto resulta em endereços diferentes na compilação, e por conta disto, quanto chamadas dentro dos algoritmos nos arquivos, serão redirecionadas para as suas respectivas posição, e não ocorrerá conflito entre as funções.

A função `getNewLine` do arquivo `.ppm` opera de forma similar a do `.obj`. Contudo, neste caso também é ignorado linhas que iniciem com o caractere "#", ou 0x23 em hexadecimal, além de não ser considerado o caractere de EOF, por ser uma função utilizada apenas no inicio do arquivo, e pelo armazenamento dos valores serem em binário, o que não garante que um byte de valor 0x00 seja relacionado ao caractere ASCII "\0".

Após obter o primeiro caracter, e verificado a sua compatibilidade de arquivo, é realizado a verificação do número de identificação. Neste caso é buscado somente arquivos `.ppm` com dados em binário, para simplificação da leitura dos *pixels*. Por conta disto, caso os primeiros caracteres válidos não sejam "P6", o algoritmo termina sem êxito.

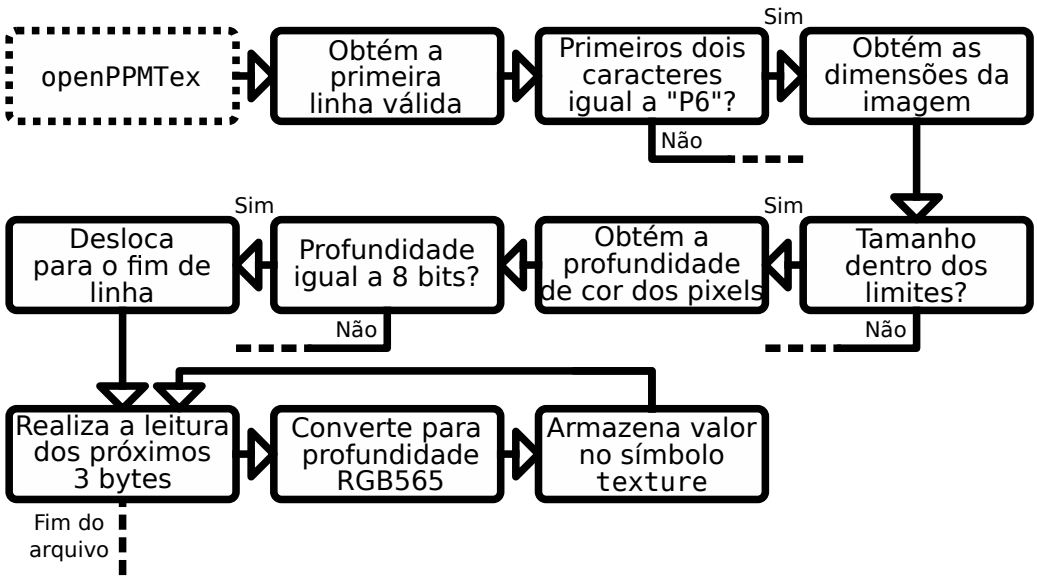
A próxima linha válida deverá apresentar as dimensões da imagem em formato ASCII, separadas com espaços. O primeiros números representam a dimensão da largura, e os demais, a altura. Neste caso ambos são limitados a 100 *pixels*, para caber na região de memória reservada. Após as dimensões, a próxima linha constará a profundidade de cada cor do *pixel*. Como o arquivo é do formato `.ppm`, se espera que este valor seja 255, caso contrário é detectado um erro. Este valor pode ser de diferentes faixas caso busque otimização ou melhor qualidade, porém, acaba por dificultar a interpretação do arquivo, e então não são considerados.

Dado os metadados iniciais, a partir do caractere de quebra de linha da última linha não comentada, são dispostos os valores dos *pixels* em formato binário. Com 8 bits para cada

cor, a leitura é realizada byte a byte. Com os valores de RGB nos registradores, a cor vermelha é limitada aos seus 5 bits superiores e deslocada 8 bits à esquerda, a cor verde é limitada aos seus 6 bits superiores e deslocadas 3 bit à esquerda, e a cor azul é deslocada 3 bit à direita limitando-a nos seus 5 bits superiores. Cada cor é realizada a operação lógica *OR* bit a bit com a prévia. Com isto se obtém um *pixel* de 16 bits com a profundidade RGB565, o formato utilizado pelo *display* TFT. Os valores dos *pixels* são armazenados no símbolo `texture`, com 20 KiB alocados.

Isto é repetido até as dimensões máximas da imagem, como dado nas informações do arquivo. Com isto a função retorna com sucesso. A função `openPPMTex` pode ser vista como fluxograma na Figura 31.

Figura 31 – Fluxograma da função `openPPMTex`




Fonte: Autoria Própria (2022)

## 4.6 Renderização do modelo

Um modelo tridimensional é constituído das suas informações de coordenadas tridimensionais, e das suas informações de textura para o mapeamento UV. Porém, para estas informações serem interpretáveis visivelmente para os indivíduos, é necessário uma conversão para uma tela capaz de disponibilizar uma projeção dos dados. Com as informações do modelo carregadas na memória RAM, o mapeamento pode ser realizado para uma matriz de *pixels* definida no *framebuffer*, que será então projetado à tela.

Com a leitura dos arquivos de modelo e textura realizadas com sucesso, o algoritmo entra nas rotinas de *loop* de renderização. Existem duas funções que são conduzidas em paralelo, `loop0` para o núcleo 0, e `loop1` para o núcleo 1. Para sincronizar a operação de ambos, dois bytes reservados no símbolo `ready_flag` indicam a prontidão de cada núcleo. O núcleo 1 se

mantem em espera durante as etapas de configuração realizadas pelo núcleo 0 esperando esta *flag*.

Após as configurações serem transitadas com êxito, o núcleo 0 entra na sua rotina de *loop*. Este será o responsável pela renderização em si, convertendo o modelo tridimensional para uma projeção bidimensional no *framebuffer* selecionado. Neste caso, são reservado dois *framebuffers* de mesmo tamanho, sendo eles representados pelo símbolo `frame_buffer0` e `frame_buffer1`, a utilização de qual depende do valor armazenado no símbolo `fb_select`. O núcleo 1 fica com a função de transferir a imagem para o *display*, e sempre trabalhará no *framebuffer* alternado do núcleo 0, evitando assim colisões durante o acesso a memória. A cada *loop*, o núcleo 0 alterna o valor armazenado em `fb_select`, sendo informado também para o núcleo 1.

### 4.6.1 Matrizes de transformação

Os valores de posições tridimensionais não podem ser diretamente mostradas para o usuário de forma visual, devido a limitação das telas serem dimensionais. Por conta disto, é necessário um mapeamento destes pontos para coordenadas bidimensionais. Isto é realizado pelas matrizes de transformação.

Estes passos são realizados dentro da função `renderModel`. O algoritmo utiliza a lista de índices obtida do modelo para cada face do modelo. Para cada item desta lista é obtido o índice das posições, e coordenadas UV. Com isto, os valores para a renderização do polígono, armazenando-os no *buffer* temporário de símbolo `vertex`. Neste símbolo são armazenados os quatro *vertexes* necessários para criação de uma face. Os valores da normal são tratados separadamente.

A matriz de transformação é dada no símbolo `base_matrix`. Esta é retornada a uma matriz identidade de tamanho quatro por quatro para cada polígono renderizado. Após isto, são aplicadas as transformações na ordem, a conversão de escala, em que se multiplica os valores da diagonal principal de "x", "y", e "z" por um vetor de escala definido em `scale_vector`.

A próxima transformação realizada na matriz é a rotação do modelo. A rotação é dada pelo eixo de rotação definido em `rotate_axis`, e com o ângulo, em graus, dado em `rotate_angle`. Primeiramente, o quaternião deve ser calculado para o preenchimento da matriz de rotação, contudo, para obtê-lo é necessário a obtenção do cosseno e seno do ângulo de rotação dividido por dois, como dado em (10). Estes são obtidos na função `cosSin`.

Na função `cosSin` o valor dado em ponto flutuante é convertido para valor inteiro. Após isto, é realizado a divisão por quatro, e obtido o resto em relação ao 180 com a instrução `remu`, limitando as faixas de repetição das funções. Funções trigonométricas não são possíveis de obtenção do valor direto por nenhuma instrução do conjunto *RV64GC*, por conta disto, outros métodos devem ser utilizados para obtenção dos resultados. Neste caso foi escolhido a utilização de uma tabela ao invés da operação por suas respectivas séries, isto permite uma maior eficiência do código, necessitando em contrapartida um gasto no tamanho total do

arquivo final. A tabela utilizada esta presente no símbolo `cos_table`, e consta 180 valores em ponto flutuante de precisão única, criados para cada 2° de ângulo, e foi gerada por cálculos prévios. Como o ângulo de rotação é limitado a 360° e o cálculo é realizado em sua metade com discretização de 2° os valores são limitados a 180 números na tabela. O valor do seno é obtido adicionando 270° ao ângulo dividido por dois, sendo uma soma de 135 considerando a indexação da tabela, defasando assim o cosseno para uma senoide. Com isto, é limitado o valor a 180 novamente, e realizado a mesma verificação via tabela.

Com o quaternião calculado, é realizada a multiplicação prévia matricial dos valores obtidos da matriz de rotação com os armazenados na matriz base previamente.

Antes de ser aplicada a translação, a matriz base pode ser aplicada na normal. A normal da face deve ser tratada como um vetor, por conta disto, as operações devem considerá-la com início na origem. Seus valores de coordenadas "x", "y", e "z" são obtidas a partir de seu índice, e então armazenados no símbolo `normal_buffer`. A função `multiplyModel` é responsável por realizar multiplicações da matriz base com um vetor de quatro coordenadas. Deste modo, a normal é redirecionada para sua nova posição, que será utilizada para o cálculo da intensidade luminosa na face.

Por simplicidade, é considerada apenas um ponto de luz armazenado no símbolo `light_pos`. Neste caso é apenas considerado a contribuição do coeficiente $L_{Difuso}$, como visto em (3). Para este cálculo, é considerado uma fonte de luz com uma distância muito maior da origem do que o fragmento, de modo que $\vec{P}_{Fragmento}$ possa ser considerado na origem. É considerado também um limite inferior maior do que zero, para permitir que regiões escuras possuam um leve detalhamento. Deste modo, a intensidade luminosa da face é dada como em (17). Este valor é armazenado para posteriormente ser utilizado na rasterização.

$$(R,G,B)_{Iluminado} = max(L_{min},\hat{N} \cdot \hat{P}_{Luz})(R,G,B)_{Original} \tag{17}$$

A translação é então acrescentada, adicionando os valores do símbolo `translate_vector` na última coluna da matriz base. Com isto, a matriz do modelo está construída.

Como neste caso apenas um objeto é renderizado, a câmera é considerada fixa, e todas as alterações são aplicadas no próprio modelo, sendo assim, a matriz de visualização pode ser tratada como a identidade.

A projeção é a última etapa aplicada na fase de trabalho sobre o *vertex*. Esta transformação é realizada pela função `projectModel`. Para isto, é necessário o cálculo da matriz de projeção, como dado em (14). Devido ao uso de funções trigonométricas, e a necessidade do recálculo continuo destes valores, foi optado por trabalhar com valores pré calculados, e armazenados no próprio código binário do algoritmo. Neste caso foi considerado o símbolo `fov_const` como $1/tan(\varphi/2)$, `aspect_ratio` para $altura/largura$, e `zconst` para ambas os valores das equações envolvendo os planos de corte $z_{Longe}$ e $z_{Perto}$.

Assim, a multiplicação matricial ocorre. Os termos na diagonal principal se comportam apenas como fatores de escala. Contudo, em relação ao eixo "z", a variável é previamente

armazenada antes de sofrer as transformações, e este valor antigo é armazenado no termo "w" novo. Deste modo a matriz de transformação é construída, e pode ser aplicada aos *vertexes* do modelo.

Com a matriz de transformação construída, e os quatro *vertexes* da face prontos, o algoritmo é redirecionado para a etapa de renderização na função `renderSquare`. Neste ponto, cada *vertex* é multiplicado pela matriz base, sendo seus novos valores armazenados nos mesmos *buffers* temporários.

### 4.6.2 Rasterização de polígonos

A etapa de transformação, de uma forma relativa, não exige grandes esforços computacionais do microcontrolador, sendo limitada pelo teto de polígonos possíveis no algoritmo. Porém, a etapa de rasterização envolve cálculos para cada fragmento da imagem, que considerado as dimensões da tela, esta etapa é operada uma vez para cada *pixel*. Por conta disto, as operações neste ponto são trabalhadas buscando as otimizações possíveis.

Dado as dimensões da face definidas pelos *vertexes*, a função `drawSquare` é chamada. De modo a evitar o trabalho sobre os valores dos próprios *vertexes*, 128 bytes do *stack* são reservados para o armazenamento temporário de variáveis. Neste ponto as coordenadas de "x" e "y" são mapeadas do cubo de projeção para as coordenadas da tela. Os demais valores são copiados para o *stack*.

Para a operação de rasterização certos símbolos são reservados para trabalho temporário antes da projeção no *framebuffer*. Os *buffers* de trabalho são definidos pelos símbolos `fbx_buffer`, `fby_buffer`, e `depth_buffer`, que são responsáveis pelo armazenamento temporário dos valores de textura nas coordenadas "U", texturas nas coordenadas "V", e valores de profundidade no cubo de projeção. Para todos estes são alocados um ponto flutuante de precisão dupla para cada *pixel* da tela, totalizando 900 KB de ocupação na memória RAM.

É reservado também o *buffer* principal de profundidade, dado pelo símbolo `depth_grid`. Neste caso é reservado 2 bytes para cada *pixel* da tela. Este valor representa a profundidade discretizada, e a diferença deste *buffer* para os de trabalho, é que este permeia mais do que um polígono, sendo utilizado para definir os *pixels* que devem ser sobrepostos.

Antes de continuar com a renderização, é verificado se os vértices do polígono estão sobrepostos. Isto é feito realizando o mapeamento da profundidade de cada *vertex* para valores discretos de 16 bits, com zero representando a posição mais próximo a câmera. Isto é utilizado para otimização, com polígonos totalmente sobrepostos sendo ignorados durante esta atualização de tela. Nota-se que isto pode resultar na não projeção de polígonos que não são sobrepostos somente em seu centro, contudo, isto ocorre somente em modelos mais complexos com estruturas concavas. De modo geral, a maioria dos polígonos sobrepostos tendem a ser cobertos como um todo, então esta técnica se mantém utilizada.

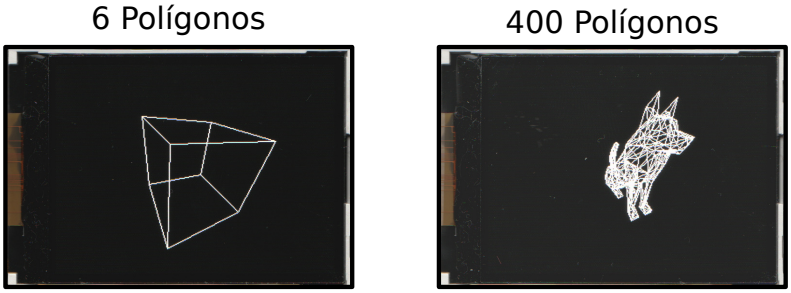
Para a rasterização do polígono, o primeiro passo é a obtenção dos limites máximos que este irá ocupar. Com os quatro valores dos vértices de "x" e "y" discretizado, é obtido o

minimo e o máximo entre estes. Com isto, os *buffers* de trabalho são limpos na região do retângulo definido por estes limites. Nos três, é armazenado o valor de infinito negativo como ponto flutuante de precisão simples, como indicação de *pixel* não preenchido ao invés do zero, que possuí significado tanto no mapeamento de texturas quanto de profundidade.

Com a região de trabalho limpa, são criado linhas utilizando o algoritmo de Bresenham, definindo os limites do polígono. Isto é feito pela função `drawLine`. Isto é realizado em quatro passos, conectando os pontos do *vertex* zero ao três em conjunto de dois. Além da ligação via preenchimento de *pixels*, cada *vertex* possui um valor de ponto flutuante que é interpolado em suas linhas. Isto é repetido para os dois *buffers* de textura, e para o de profundidade.

O efeito do algoritmo de Bresenham, desconsiderando os valores de interpolação pode ser visto na Figura 32.

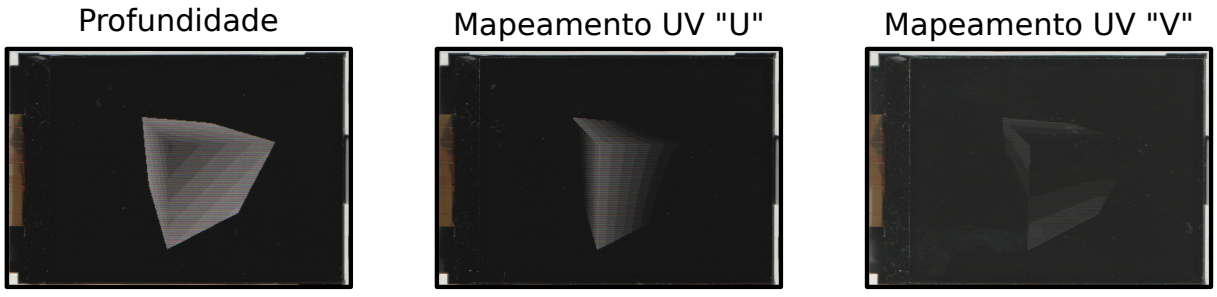
Figura 32 – Renderização de linhas do modelo




Fonte: Autoria Própria (2022)

Com isto, as linhas são preenchidas, utilizando a função `fillPolygon`. Nesta é utilizado o algoritmo *scan line* com preenchimento no eixo "y", para que o interior do polígono seja construído. Novamente, esta operação é repetida para os três *buffers* de trabalho, como visto na Figura 33, sendo neste caso, o mapeamento UV igual em ambos os eixos.

Figura 33 – Identificação dos *buffers* de trabalho por cor




Fonte: Autoria Própria (2022)

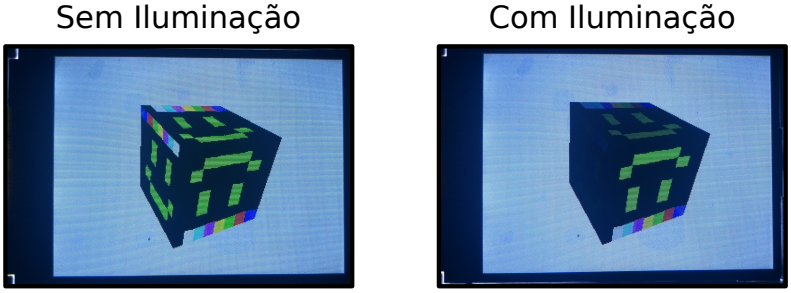
O valor do *buffer* de trabalho de profundidade, é mapeado para valores inteiros, sendo o valor de -1 em "z" mapeado para 0x0000, e 1 para 0xFFFF. Caso o valor esteja fora destes limites o *pixel* é descartado. Caso contrário, é comparado a profundidade do *pixel* com o valor

armazenado em `depth_grid`, dado suas coordenadas na tela. Sendo o novo *pixel* com um valor menor, ele segue para renderização da textura, e seu valor é armazenado em `depth_grid`. Caso o valor seja maior, o algoritmo salta para o próximo *pixel*.

Na coloração do *pixel*, é utilizado os valores armazenados nos *buffers* do mapeamento UV, sendo mapeados de 0 a 1, para 0 ao tamanho máximo da textura em cada eixo. Com os valores inteiros, a coloração da posição específica na memória da textura é carregada ao registrador. Neste ponto, a coloração do modelo para a posição de todos os fragmentos que ocupa pode ser obtida, porém, ainda é necessário a consideração do efeito da iluminação.

Como o valor da intensidade luminosa para a face que está sendo rasterizada já foi obtida, na etapa de manipulação dos *vertexes*. Esta é mapeada de um valor inteiro de 0 a 32 antes da iteração dos *pixels* da face. Para aplicar este valor nas cores do modelo, é necessário a realização em três etapas, sendo uma para cada cor. A cor utilizada na etapa é movida para um novo registrador utilizando uma mascara, dada sua posição no *pixel*. Após isto, é realizado a multiplicação inteira com a cor isolada, e dividido o valor resultante por 32, armazenando o resultado na sua posição original dada a profundidade RGB565. A multiplicação por inteiro oferece um tempo de cálculo menor do que se tratado como ponto flutuante, e a divisão por um valor múltiplo de dois pode ser realizada por deslocamento binário à direita. O efeito da iluminação pode ser visto na Figura 34. Como visto na Subsubseção 2.2.1.3, o coeficiente de iluminação é limitado do intervalo de 0 a 1, por conta disto, a iluminação é aplicada visualmente por um efeito de sombreamento das faces ocultas.

Figura 34 – Aplicação da iluminação no modelo




Fonte: Autoria Própria (2022)

### 4.6.3 Loop de renderização

As etapas de renderização são realizadas para todos os polígonos do modelo, porém todas estas se dão em endereços da memória RAM. Finalizando estes, o algoritmo entra em rotina para enviar o resultado para o *display*, e limpa os *buffers* de memória utilizados.

A etapa realizada pelo algoritmo no núcleo 0, dada pela função `loop0` inicia verificando a prontidão da *flag* para seu respectivo núcleo. Caso seja a primeira vez que a iteração é realizada, ou o núcleo 1 tenha finalizado sua rotina, o código prossegue, limpando a sua respectiva *flag*.

Neste ponto é o *framebuffer* que será renderizado sobre é limpo, armazenando uma cor sólida sobre todos os seus *pixels*. O *buffer* `depth_grid` é também limpo, armazenando o valor máximo de 0xFFFF em toda sua região, criando assim um plano no fundo do cubo de projeção que será utilizado para o corte de *pixels* na etapa de rasterização.

Neste ponto, o modelo é renderizado no *framebuffer*, pela função `renderModel`.

O símbolo indicador do *framebuffer* é alternado. Deste modo é indicado que na próxima etapa de renderização, ambos os núcleos devem trocar o *framebuffer* em que operam.

Ao fim a *flag* indicativa para o núcleo 1 é ativada. O núcleo 0 retorna para o início da função `loop0`.

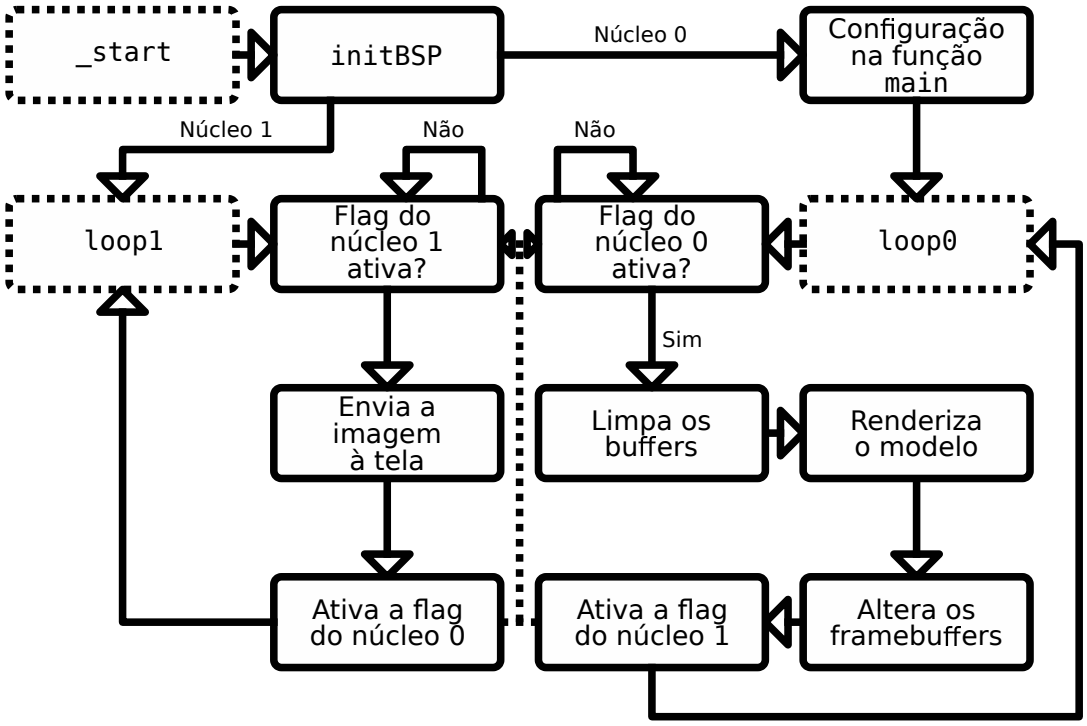
O núcleo 1 inicia de modo similar na função `loop1`, ao aguardar a sua respectiva *flag*. Caso confirmado, é carregado o *framebuffer* oposto ao núcleo 0, e enviado para a tela pela função `tftRefreshDisplay`.

Para o *display* TFT é enviado os comandos de definição de área a se escrever, que neste caso é todo a região visível, e logo em seguida enviados os dados do *framebuffer* com uma escrita à memória.

Com a imagem enviada à tela, o núcleo 1 ativa a *flag* do núcleo 0, e retorna para o início da função `loop1`.

Uma descrição visual das operações que ocorrem no microcontrolador pode ser vista na Figura 35.

Figura 35 – Fluxograma das etapas de *loop* do algoritmo



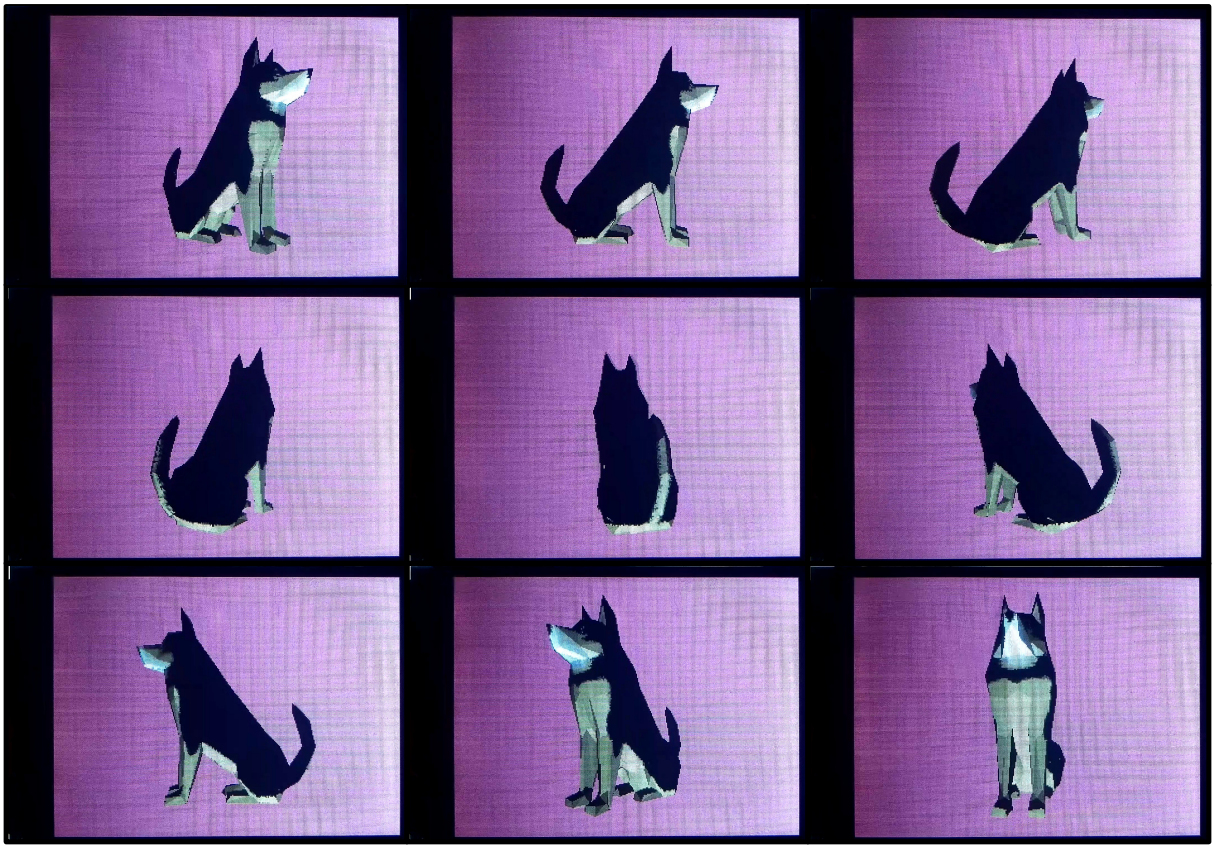

Fonte: Autoria Própria (2022)

# 5 ANÁLISE E DISCUSSÃO DOS RESULTADOS

A partir do desenvolvimento via algoritmo, a imagem é projetada para a tela, e o resultado buscado é obtido. Neste ponto é possível realizar uma análise qualitativa e uma medição quantitativa do trabalho final.

Como não há configurações com dispositivos de entrada de comando externos, o modelo somente pode se mover com as informações previstas no código. Para fins de visualização do tempo de renderização foi definido uma alteração do angulo de rotação de 4° por atualização do quadro. O efeito deste para a disposição do modelo pode ser visto na Figura 36.

Figura 36 – Rotação de um modelo com diferença de 1 s entre quadros



Fonte: Autoria Própria (2022)

O resultado pode ser analisado de diferentes formas, contudo, em quesito do tema proposto, de renderização de modelos tridimensionais utilizando programação em RISC-V assembly, o objetivo foi atingido. Porém é possível a renderização de somente um único modelo com uma única textura em um dado instante.

A replicação para outras placas do mesmo modelo garante que o desenvolvimento se baseia puramente no algoritmo, sem alterações do hardware. O cartão SD contudo, apesar de poder ser utilizado de diversas marcas e modelos, deve ser garantido a formatação prévia via

FAT32, o que, dependendo do tamanho do dispositivo, não é possível.

Após a compilação do algoritmo via ferramentas GCC, antes da construção do arquivo binário, o código é construído em arquivo ELF. Neste ponto, todas as seções são divididas, e as instruções são decompostas a construção direta do código de máquina. Com auxílio de ferramentas do próprio GCC, este arquivo pode ser decomposto em texto ASCII, para análise das instruções utilizadas por cada símbolo, com valores direto dos endereços de memória ocupados.

Uma análise que pode ser realizada é a contagem de instruções de certas rotinas, observando a seção `.text`. Este valor não oferece uma análise quantitativa de performance, especialmente devido a instruções de saltos e acessos à memória, porém reflete a complexidade de cada função. Estes valores podem ser vistos na Tabela 10. Nota-se que estes valores são de igual tamanho ou maiores as funções escritas em código, devido a certas instruções se decomporem em diversas. Instruções de `nop` são utilizadas geralmente para alinhamento, e preenchem espaços entre funções.

O código de máquina final apresenta 14.720 bytes. Destes, desconsiderando preenchimentos pelo compilador, 11.744 bytes são utilizados pelas funções na seção `.text`, 1008 bytes por `.rodata`, e 572 bytes por `.data`. Além disto, são alocados 6 bytes para a seção `.fini_array` que está relacionada com a finalização do código. Demais bytes são de regiões de espaço entre símbolos e seções, preenchidas por zero, alocadas pelo compilador.

Com a contagem de instruções e a ocupação das funções na seção `.text` pode-se analisar o efeito da extensão *C*, de instruções compactas, no tamanho final do código. Caso todas as instruções fossem do tamanho padrão de 4 bytes, a ocupação total das instruções seria de 14.156 bytes, utilizando os valores da Tabela 10. Para uma redução de 2.412 bytes, é necessário a utilização de 1.206 instruções de 2 bytes. Deste modo, percebe-se que 34,077% das instruções puderam ser reduzidas, gerando uma redução de 16,386% do tamanho do arquivo binário final.

Além da memória alocada no arquivo binário, a seção `.bss` reserva 1.905.472 bytes na memória RAM durante a execução do código. Destes, 180.000 bytes são para informações do modelo, e 1.689.600 bytes para *buffers* relacionados ao tamanho da tela. A ocupação das informações do modelo preenchem boa parte da memória, contudo, a maior parte está relacionada ao armazenamento de informações atreladas aos fragmentos. Isto acaba se dando pela grande quantidade de *pixels* presentes, qualquer informações atreladas a estes necessita de vasto espaço na memória. Este ponto deve ser atento caso necessário a utilização de resoluções maiores.

Considerando o arquivo binário mais os símbolos de alocação em execução, dos 8 MiB disponíveis do microcontrolador K210, 1.920.192 bytes, ou 24,002%, da memória RAM é ocupada. Este valor desconsidera as alterações do tamanho do *stack* e regiões reservadas entre os símbolos de `.bss`.

Devido a generalidade do código, pode-se alterar os modelos e texturas utilizadas

Tabela 10 – Contagem das instruções das funções

| **Função** | **Instruções** | **Função** | **Instruções** |
|---|---|---|---|
| _start | 86 | renderModel | 145 |
| trap_entry | 69 | cleanBaseMatrix | 25 |
| .handle_irq | 2 | openOBJModel | 90 |
| .handle_syscall | 1 | loadPolygons | 97 |
| .restore | 66 | loadVertices | 42 |
| deregister_tm_clones | 10 | loadUVs | 34 |
| __do_global_dtors_aux | 21 | loadNormals | 43 |
| initBSP | 37 | getNewLine(.obj) | 22 |
| initBSS | 12 | getFloatFromValue | 47 |
| enable_irq | 10 | getNumberFromFace | 19 |
| initPlic | 52 | ignoreSpaces | 10 |
| crc7get | 22 | openPPMTex | 126 |
| openFAT | 156 | getNewLine(.ppm) | 22 |
| readFile | 174 | drawSquare | 374 |
| initFPIOA | 43 | floatToDimension | 19 |
| setupFPIOA | 11 | clampi | 9 |
| setupGPIO | 17 | minmaxi | 21 |
| setupGPIOHS | 26 | clearScreen | 27 |
| outputGPIO | 12 | fillPolygon | 59 |
| outputGPIOHS | 12 | drawLine | 165 |
| handleSyscall | 4 | renderSquare | 28 |
| handleIRQ | 4 | sdInitialize | 176 |
| handleIrqMExt | 31 | syncData | 11 |
| main | 73 | sdSetup | 49 |
| loop0 | 47 | sdSectorRead | 54 |
| loop1 | 29 | sdSPIwrite | 43 |
| failure | 8 | sdSPIread | 42 |
| multiplyModel | 48 | setupPLL | 74 |
| scaleModel | 21 | tftInitialize | 80 |
| rotateModel | 114 | tftRefreshDisplay | 50 |
| translateModel | 21 | tftSetup | 64 |
| cosSin | 20 | tftWriteDMA | 143 |
| projectModel | 70 | *Total* | 3.539 |

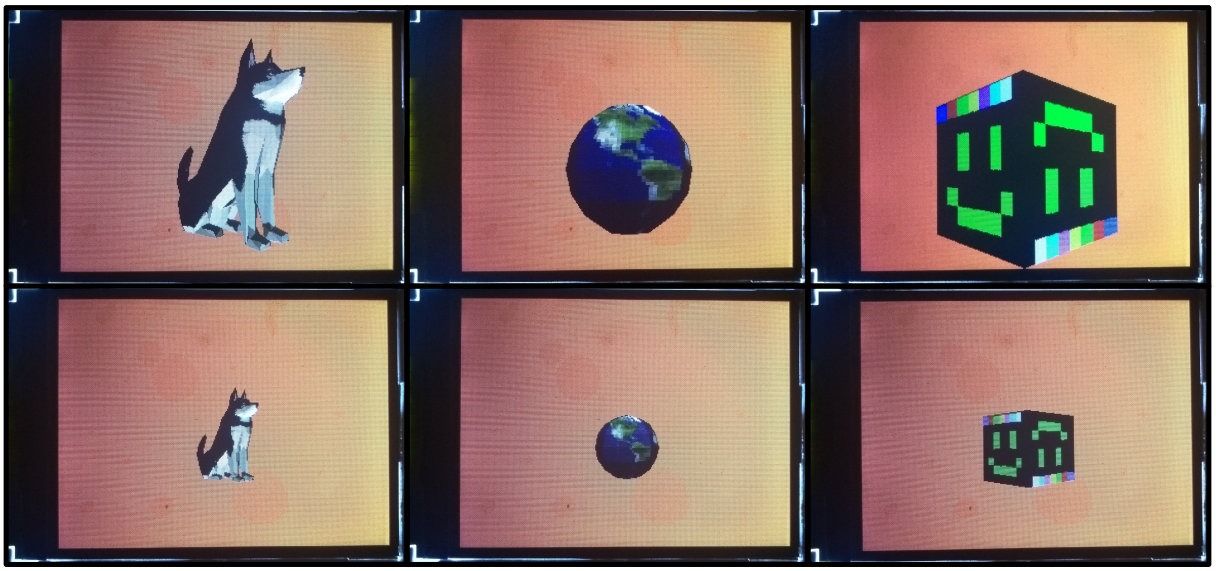
Fonte: Autoria Própria (2022)

alterando-se os nomes buscados definidos em algoritmo. Deste modo, para uma comparação de eficiência e tempo, pode ser realizado uma análise com diversos arquivos diferentes. Estes diversos modelos e texturas estão presentes na Figura 37. Neste caso, é abordado três casos de dimensões diferentes, um modelo com alta contagem de polígonos e maior resolução de textura, um caso médio para ambos os arquivos, e um caso de dimensões mínimas. É também variado a escala no modelo na tela, para comparar a diferença do tamanho da região rasterizavel, e como esta é proporcional ao tempo de renderização.

A comparação entre os tempos de inicialização dos diversos modelos pode ser visto

Figura 37 – Variações dos modelos e texturas utilizados para testes

| **Modelo A**: | **Modelo B**: | **Modelo C**: |
|---|---|---|
| 400 polígonos | 64 polígonos | 6 polígonos |
| [38,8 KiB] | [7,2 KiB] | [466 B] |
| **Textura:** | **Textura:** | **Textura:** |
| 250x250 pixels | 100x78 pixels | 8x8 pixels |
| [183,2 KiB] | [22,9 KiB] | [249 B] |
| **Escala 1:** 35% | **Escala 1:** 90% | **Escala 1:** 90% |
| **Escala 2:** 17,5% | **Escala 2:** 45% | **Escala 2:** 45% |



Fonte: Autoria Própria (2022)

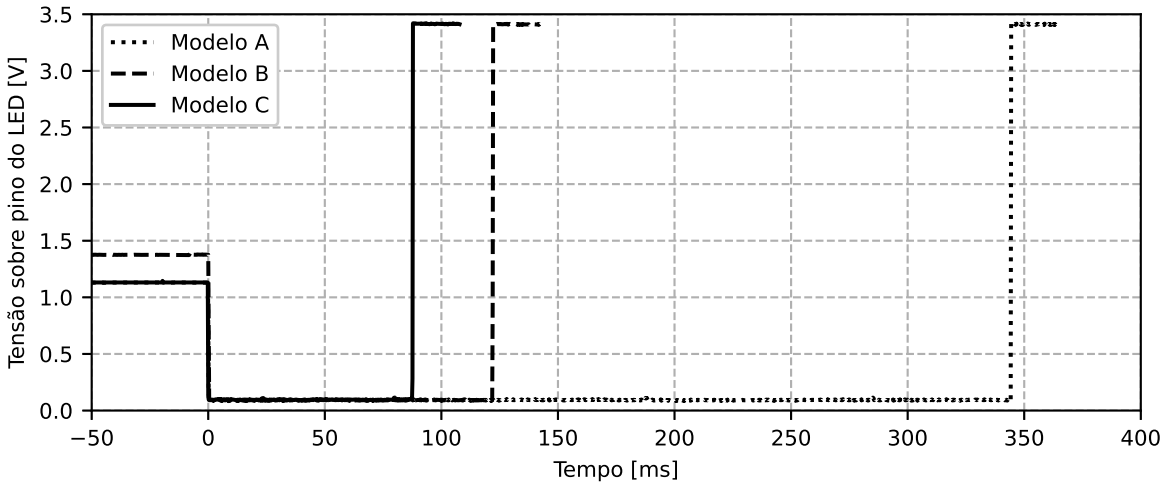
na Figura 38. Neste caso, é contabilizado o tempo que um LED auxiliar demora pra ligar, sendo este, ligado no momento que o algoritmo entra na etapa de *loop*. Antes da energização propriamente do microcontrolador, a sua saída se mantém em um estado intermediário, por conta disto, pode ser medido esta diferença temporal entre os níveis de tensão do pino.

Como a escala do modelo não influencia no tempo de inicialização, porque a inicialização consiste em grande parte da inicialização do cartão SD, e configuração de periféricos, esta não é analisada na Figura 38. Percebe-se como quando maior o tamanho total dos arquivos, mais lendo tende a ser a inicialização. Neste caso, a inicialização com o modelo "A" toma 344,3±0,05 ms, do modelo "B", 122,2±0,1 ms, e do modelo "C", 87,8±0,1 ms.

A variação dos modelos, durante a inicialização, ocorre na etapa da leitura do cartão SD. Para isto, pode ser analisado o pino de seleção do cartão SD. Durante a sua variação são períodos em que o cartão SD esta realizando alguma forma de comunicação com o microcontrolador. Esta comunicação é realizada em três etapas. A primeira destas é para a configuração do cartão SD até a obtenção do endereço da pasta raíz e da tabela de alocação. A segunda é para a leitura do arquivo de textura. A terceira ocorre na leitura do arquivo do modelo 3D.

A Figura 39 apresenta a variação de tensão deste pino com o tempo. No modelo "A" é

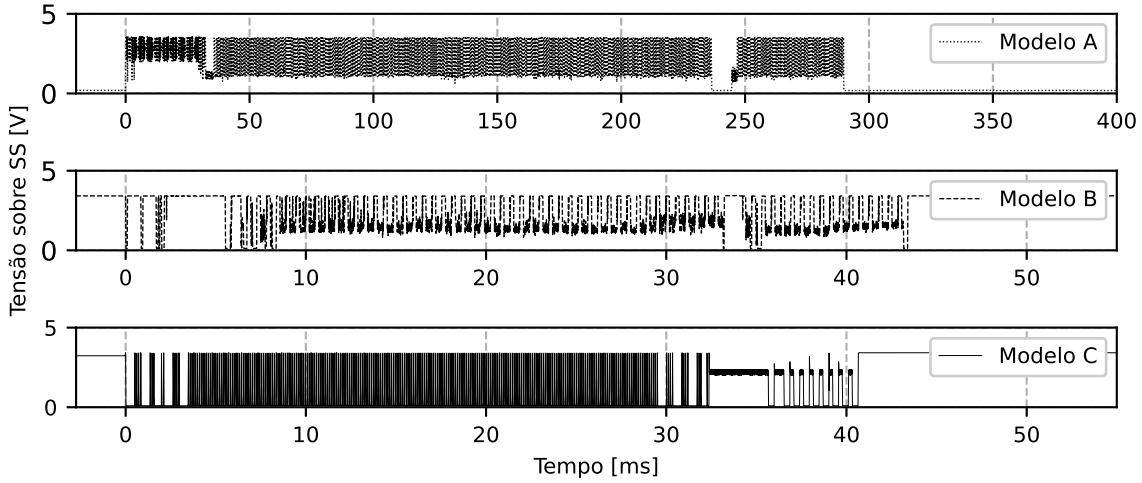
Figura 38 – Tempo de inicialização entre modelos




Fonte: Autoria Própria (2022)

possível ver a distinção entre as etapas, o que trás uma taxa de transmissão mínima que cada arquivo é transferido. A primeira etapa demora 32,35±0,025 ms, a segunda, 200,75±0,025 ms, e a terceira, 43,05±0,025 ms. Utilizando o tamanho dos arquivos, a taxa de transferência durante a transmissão da textura foi de no mínimo 912,578±0,227 KiB/s, e 901,278±1,05 KiB/s. Ao se notar que este tempo incluí diversas outras funções como a leitura das entradas na pasta raíz, além da leitura necessitar de no mínimo um setor para transmitir, o que causa uma taxa maior aparente para arquivos maiores, por isto, é estabelecido como taxa mínima.

Figura 39 – Tempo de leitura do cartão SD



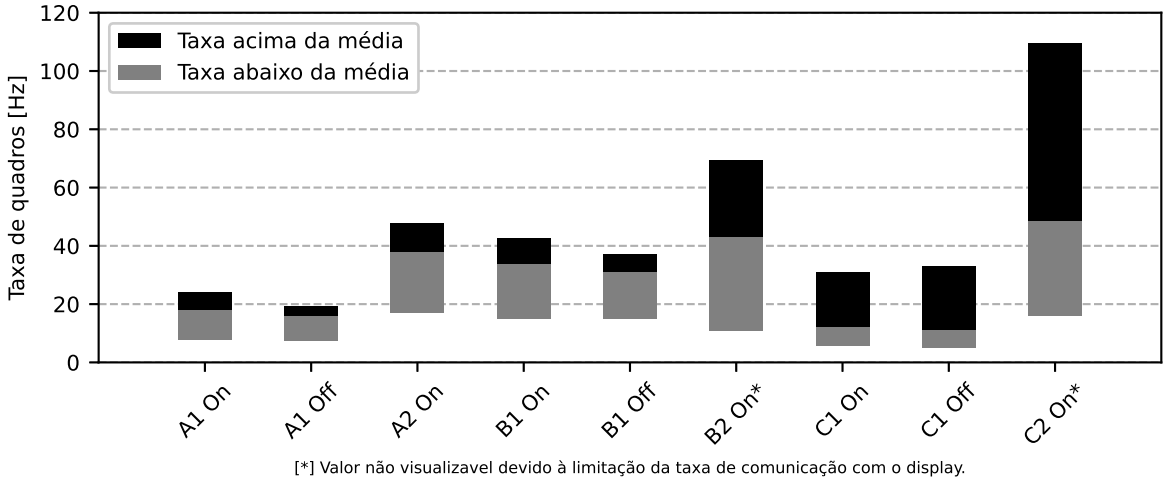

Fonte: Autoria Própria (2022)

Nos demais modelos a distinção entre etapas não é completamente aparente. Porém,

é possível realizar uma comparação de tempo total entre os modelos. O modelo "A" leva 289,85±0,025 ms para comunicação com o cartão SD, o modelo "B" leva 43,405±0,0025 ms, e o modelo "C", 40,650±0,0025 ms. Isto da uma taxa de transferência miníma de 765,913±0,132 KiB/s, 693,468±0,799 KiB/s, e 17,589±0,002 KiB/s respectivamente. Percebe-se como quanto maior o tamanho dos arquivos, maior a taxa de transferência mínima, e como já comentado, isto é devido as demais funções que necessitam de execução em conjunto com a leitura.

Durante a renderização um ponto a ser análisado é a taxa de renderização dos quadros. Neste caso, esta medida foi realizada via um pino auxiliar que atualiza seu estado após a renderização do quadro, e convertido esta medida de tempo em frequência. Os valores obtidos podem ser vistos na Figura 40, onde a barra mostra a taxa de variação entre quadros, e a linha central indica a média da taxa de renderização. Além da escala, foi também comparado as taxas com e sem o corte prévio dos polígonos renderizados.

Figura 40 – Variação da taxa de quadros entre modelos




Fonte: Autoria Própria (2022)

Em relação aos modelos, nota-se que o modelo "A", com uma maior contagem de polígonos, tende a utilizar um tempo de renderização maior. Contudo, o modelo "B" possuí uma maior taxa de renderização do que o modelo "C", apesar de uma maior taxa de polígonos. Isto se dá porque a renderização também está atrelado ao tamanho da região que será renderizada, e como visto na Figura 37, o modelo "C" ocupa uma região maior em tela do que o "B".

Além disto, nota-se com a escala maior, a taxa de renderização é menor. Isto se dá novamente pela região de rasterização em si, porque não há diferença entre a quantidade de operações de um mesmo modelo durante o trabalho sobre os vértices, apenas em suas etapas de rasterização.

O efeito do corte de polígonos prévio pode também ser visto, ao aumentar a taxa de quadros dos modelos "A" e "B" quando aplicado. Contudo, o modelo "C" apresenta uma redução na taxa. Isto ocorre porque o corte dos polígonos necessita de operações adicionais, e

há uma taxa reduzida da quantidade de polígonos pulada quando o modelo apresenta uma baixa contagem destes. Para modelos maiores esta técnica obtém a necessidade buscada.

O microcontrolador, com o acréscimo do display, consome um total de 154,5$\pm$0,05 mA durante a operação. Retirando o display, o consumo do microcontrolador é de 100,7$\pm$0,05 mA. Este valor não leva em consideração o cartão SD, porque este é utilizado somente durante a inicialização, e após isto não é selecionado novamente. Esta corrente considera todos os periféricos utilizados internamente ao microcontrolador, pela impossibilidade de se obter uma corrente interna direta à CPU. Com a alimentação em 3,3 V, isto indica um consumo de 332,31$\pm$0,165 mW para o microntrolador.

A capacidade de renderização com as ferramentas do RISC-V foi comprovada, e se obteve um resultado satisfatório. Além disto, o consumo é baixo, com corrente na casa da centena de mili ampère. Os objetivos estabelecidos foram alcançados.

Em relação a utilização de modificações no próprio hardware, um ponto que deve ser abordado é a implementação das funções de rasterização de forma otimizada. Devido a alta contagem de instruções como `drawSquare` e `drawLine`, cria-se um afunilamento dos dados neste ponto. Como as funções de rasterização são também extensivamente chamadas, por sua necessidade de trabalhar na escala da contagem de *pixels* na tela, implementações paralelas seriam ideiais. Os *pixels* nos *framebuffers* não dependem dos valores adjascentes quando os *buffers* de trabalho já estiverem preenchidos, o que seria uma situação para implementação de uma solução de processamento paralelo.

Um dos problemas notados tanto nas fases do *vertexes* quando dos fragmentos, foi a limitação da quantidade de registradores. Neste caso, ao trabalhar em RISC-V assembly, foi possível contornar as normas estabelecidas na ABI, e utilizar registradores para funções não pré-determinadas. Isto não seria possível em uma linguagem de mais alto nível, permitindo apenas o trabalho sobre os registradores reservados como temporários, supondo o seguimento das convenções. Em casos que os registradores temporários estivessem todos ocupados, necessitaria de armazenamento na memória *cache*, o que aumentaria o tempo de processamento utilizado. Contudo, com a futura extensão *V*, um processador utilizando RISC-V terá maior espaço para armazenamento nos registradores. Além disto, a extensão *V* é prevista para realização de processamento paralelo de dados, o que pode evitar certas repeticições e laços utilizados neste algoritmo.

# 6 CONCLUSÃO

A evolução humana através dos anos nos trouxe cada vez mais próximos da projeção do virtual ao mundo real. O desejo da representação do inimaginável, projetado nas tecnologias que nos cercam. Neste trabalho foi discutido os caminhos para obtenção deste objetivo, e apresentado em um resultado real, mostrando assim a tangibilidade deste meio.

A tecnologia do RISC-V providência novas ferramentas para tratar a implementação de algoritmos, além de diferentes meios de se tratar arquiteturas computacionais. Sua implementação para tarefas de alto consumo computacional pode ser decomposto em sua simplicidade de utilização. A linguagem assembly provida em conjunto apresenta um sistema simples de compreensão, buscando, além disto, otimização em suas instruções. Estes motivos podem ser trazidos como justificativa para a implementação do RISC-V assembly no projeto.

As técnicas de renderização e rasterização utilizadas, apesar de simplórias, apresentam um resultado satisfatório. O objetivo de alto realismo não é alcançado, porém, a projeção de modelos simples com processador de baixo poder computacional, comparado com as ferramentas geralmente utilizadas para estas tarefas, é comprovado.

Deste modo este dispositivo pode ser aplicado para aplicações de renderização de baixo consumo, como realidade aumentada e produtos que teriam sua utilidade aprimorada com visualização 3D. O baixo consumo e simplicidade permite a alimentação diretamente via bateria, e nestes casos que a renderização não necessita de alto realismo, estes métodos de apresentação podem ser utilizados.

De modo geral os objetivos desejados foram alcançados de modo satisfatório. Providenciando, além disto, uma forma diferente do tratamento de renderização, visando um baixo consumo ao invés de um realismo aumentado.

## 6.1 Aprimoramentos futuros

Apesar do objetivo final ter sido alcançado, diversas ramificações da estrutura desenvolvida apresentam pontos de melhora.

Um destes pontos é a subutilização dos núcleos. O núcleo 1, como dedicado para a transferência via DMA, não realiza tarefa neste meio termo, apesar da conexão poder ser realiza ser interferência do processador, porque é guiada por um periférico. Neste ponto, o desejável é a utilização do núcleo 1 para técnicas de renderização, diminuindo o sobrecarregamento sobre o núcleo 0. Além disto, ao aplicar técnicas de paralelismo, o sistema pode ser escalado para um conjunto maior de núcleos, gerando assim ciclos de renderização mais rápidos.

Durante a decodificação das entradas nas pastas FAT alguns pontos podem ser melhorados. Neste caso é assumido que as entradas curtas são precedidas diretamente das respectivas entradas longas, e isto não é sempre o caso. Em um sistema com uma pasta sem

alocação de arquivos, ao acrescentar um arquivo à memória, estas entradas tendem a seguir esta sequência, porém, ao renomear e excluir arquivos com o tempo, com contagem diferentes de entradas longas, esta ordem pode ser quebrada. Geralmente os leitores de cartão reconhecem isto, e realizam uma varredura completa entre as entradas, com a soma de verificação para garantir a conexão. Neste caso esta verificação não é realizada, e a varredura é feita de modo sequencial, até atingir o final das entradas longas corretas, assumindo-se a próxima entrada como entrada curta do arquivo.

Durante a leitura dos valores do arquivo `.obj`, é realizado várias iterações no arquivo. Durante a contagem de valores totais, e outras iterações para cada tipo diferente de valor. Como esta verificação ocorre apenas uma vez para cara energização do dispositivo, este ponto foi mantido deste modo. Contudo, é um tempo que poderia ser reduzido caso sejam implementados mais ramificações internas nesta fase do algoritmo, para a segmentação dos valores e contagem total em uma única passagem.

Na fase de renderização a matriz é recalculada para cada polígono. Este é um ponto que, como visto anteriormente, se mantém constante durante todos os polígonos da cena para um único modelo. Por conta disto, este ponto pode ser retirado algumas ramificações acima, e evitar se manter nas iterações dos polígonos.

As funções `drawLine` e `fillPolygon` são de computação intensiva, porque estão trabalhando no nível do fragmento. Neste caso estas funções são utilizadas três vezes cada para um único polígono, uma vez para cada *buffer* de trabalho. Uma otimização a ser implementada, que necessitaria de implementação de ramificações internas nestas funções, é a rasterização de todos os *buffers* de uma única vez. Como as bordas limites dos polígonos se mantém entre estas, somente alterando os valores aplicados aos gradientes, durante a criação das linhas e preenchimento dos polígonos estes podem ser realizados ao mesmo tempo, apenas alterando o *buffer* na projeção de cada *pixel*.

## 6.2 Considerações finais

Este trabalho visou apresentar técnicas de renderização e rasterização em um processador simples, comparado com os geralmente utilizados para estas tarefas. Estes foram pontos concluídos. Certos pontos podem ser melhorados, porém, de modo geral, o resultado é satisfatório.

Esta monografia apresenta a jornada de desenvolvimento ao ponto final traçado, e busca apresentar o conhecimento de uma forma construtiva para que demais leitores, caso seja de vontade, progridam no desenvolvimento de suas próprias atividades. A presentação do texto buscou métodos concisos e lineares de progressão, gerando uma evolução gradativa a aqueles que sintam a necessidade de absorção de algum dos temas.

# REFERÊNCIAS


ACORN. **ARM hardware reference manual**. Cambridge, 1986. Citado na página 26.

ADVE, S. V.; HILL, M. D. Weak ordering—a new definition. **ACM SIGARCH Computer Architecture News**, ACM New York, NY, USA, v. 18, n. 2SI, p. 2–14, 1990. Citado na página 61.

AKSHINTALA, A. et al. X86-64 instruction usage among c/c++ applications. **Proceedings of the 12th ACM International Conference on Systems and Storage**, p. 68–79, 2019. Citado na página 25.

APODACA, A. A.; MANTLE, M. Renderman: Pursuing the future of graphics. **IEEE Computer Graphics and Applications**, IEEE, v. 10, n. 4, p. 44–49, 1990. Citado na página 19.

ARM. **ARM architecture reference manual**. Cambridge, 2005. Citado na página 26.

BOUKNIGHT, W. An improved procedure for generation of half-tone computer graphics presentations. **Coordinated Science Laboratory Report no. R-432**, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1969. Citado na página 55.

BRADSHAW, R.; SCHROEDER, C. Fifty years of ibm innovation with information storage on magnetic tape. **IBM Journal of Research and Development**, IBM, v. 47, n. 4, p. 373–383, 2003. Citado na página 57.

BRAUN, K. F. Electrical oscillations and wireless telegraphy. **Nobel Lecture, December**, v. 11, n. 1909, p. 226–245, 1909. Citado na página 18.

BRESENHAM, J. E. Algorithm for computer control of a digital plotter. **IBM Systems journal**, IBM, v. 4, n. 1, p. 25–30, 1965. Citado na página 53.

CANAAN. **K210 Datasheet**. Hangzhou, 2018. Citado na página 64.

CARRIER, B. **File system forensic analysis**. Upper Saddle River: Addison-Wesley Professional, 2005. Citado 2 vezes nas páginas 80 e 81.

CELIO, C. et al. The renewed case for the reduced instruction set computer: Avoiding isa bloat with macro-op fusion for risc-v. 2016. Citado na página 38.

CIGNONI, P. et al. A general method for preserving attribute values on simplified meshes. **IEEE Proceedings Visualization'98**, IEEE, p. 59–66, 1998. Citado na página 45.

CROOKES, W. The bakerian lecture.—on the illumination of lines of molecular pressure, and the trajectory of molecules. **Philosophical Transactions of the Royal Society of London**, The Royal Society London, n. 170, p. 135–164, 1879. Citado na página 18.

EMER, J. S.; CLARK, D. W. A characterization of processor performance in the vax-11/780. **ACM SIGARCH Computer Architecture News**, ACM New York, v. 12, n. 3, p. 301–310, 1984. Citado na página 37.

FOLEY, J. D. **Computer Graphics**: Principles and practice. 2. ed. Boston: Addison-Wesley, 1996. Citado na página 46.

FORRESTER, J. W. Digital information storage in three dimensions using magnetic cores. **Journal of Applied Physics**, American Institute of Physics, v. 22, n. 1, p. 44–48, 1951. Citado na página 57.

GOREN, Y.; SEGAL, I. On early myths and formative technologies: A study of pre-pottery neolithic b sculptures and modeled skulls from jericho. **Israel Journal of Chemistry**, Wiley Online Library, v. 35, n. 2, p. 155–165, 1995. Citado na página 17.

HAMILTON, W. R. On quaternions; or on a new system of imaginaries in algebra. **The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science**, Taylor & Francis, v. 33, n. 219, p. 58–60, 1848. Citado na página 47.

HARRIS, S. L.; HARRIS, D. **Digital design and computer architecture**: Arm edition. 2. ed. São Francisco: Morgan Kaufmann, 2015. Citado na página 59.

HOLLERITH, H. The electrical tabulating machine. **Journal of the Royal Statistical Society**, JSTOR, v. 57, n. 4, p. 678–689, 1894. Citado na página 23.

IEEE. **IEEE Std 754™-2008 (Revision of IEEE Std 754-1985), IEEE Standard for Floating-Point Arithmetic**. New York, 2008. 70 p. Citado na página 28.

INTEL. **Intel 8086 Family User's Manual October 1979**. Santa Clara, 1979. 208 p. Citado na página 25.

INTEL. **Intel 64 and IA-32 Architectures Software Developer's Manual**. Santa Clara, 2021. 4778 p. Citado na página 38.

KING, I.; WU, D.; POGKAS, D. How a chip shortage snarled everything from phones to cars. Bloomberg, 2021. Disponível em: <https://www.bloomberg.com/graphics/2021-semiconductors-chips-shortage>. Citado na página 20.

KLEIN, D. The history of semiconductor memory: From magnetic tape to nand flash memory. **IEEE Solid-State Circuits Magazine**, IEEE, v. 8, n. 2, p. 16–22, 2016. Citado na página 58.

LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess programm. **IEEE transactions on computers**, IEEE Computer Society, v. 28, n. 09, p. 690–691, 1979. Citado na página 61.

LEISERSON, C. E. et al. There's plenty of room at the top: What will drive computer performance after moore's law? **Science**, American Association for the Advancement of Science, v. 368, n. 6495, 2020. Citado na página 32.

MACHOVER, C. A brief, personal history of computer graphics. **Computer**, IEEE, v. 11, n. 11, p. 38–45, 1978. Citado na página 39.

MASUOKA, F. et al. A new flash e 2 prom cell using triple polysilicon technology. In: IEEE. **1984 International Electron Devices Meeting**. [S.l.], 1984. p. 464–467. Citado na página 58.

MICHENER, J. C.; DAM, A. v. Functional overview of the core system with glossary. **ACM Computing Surveys (CSUR)**, ACM New York, v. 10, n. 4, p. 381–387, 1978. Citado na página 39.

MICROSOFT. **Microsoft FAT Specification**. Redmond, 2005. Citado 3 vezes nas páginas 76, 77 e 78.

MONGE, G. **Géométrie descriptive. Lecons données aux écoles normales, l'an 3 de la République; par Gaspard Monge**. Paris: Baudouin, imprimeur du corps legislatif et de l'institut naional, 1798. Citado na página 17.

MOORE, G. E. Cramming more components onto integrated circuits. **Electronics**, McGraw-Hill Education, v. 38, n. 8, p. 4, 1965. Citado na página 25.

NEIDER, J.; DAVIS, T.; WOO, M. **OpenGL programming guide**. Reading: Addison-Wesley, 1993. v. 478. Citado na página 42.

NIPKOW, P. Der telephotograph und das elektrische teleskop. **Elektrotechnische Zeitschrift**, v. 6, p. 419–425, 1885. Citado na página 18.

PATTERSON, D. A.; SEQUIN, C. H. Risc i: A reduced instruction set vlsi computer. **Computer**, IEEE Computer Society, v. 15, n. 09, p. 8–21, 1982. Citado 2 vezes nas páginas 25 e 59.

PHONG, B. T. Illumination for computer generated pictures. **Communications of the ACM**, ACM New York, v. 18, n. 6, p. 311–317, 1975. Citado na página 44.

PLÜCKER, J. **Über die Einwirkung des Magneten auf die elektrischen Entladungen in verdünnten Gasen**. **Annalen der Physik und Chemie**, v. 179, p. 88–106, 1858. Citado na página 17.

RISC-V. **RISC-V ABIs Specification**. Suíça, 2021. 37 p. Disponível em: <https://github.com/riscv-non-isa/riscv-elf-psabi-doc>. Citado 2 vezes nas páginas 33 e 36.

ROJAS, R. Konrad zuse's legacy: The architecture of the z1 and z3. **IEEE Annals of the History of Computing**, IEEE, v. 19, n. 2, p. 5–16, 1997. Citado na página 24.

SD Card Association. **SD Specifications**: Part 1 sdio simplified specification. San Ramon, 2018. Citado 2 vezes nas páginas 71 e 74.

SD Card Association. **SD Specifications**: Part 1 physical layer simplified specification. San Ramon, 2020. Citado 3 vezes nas páginas 70, 72 e 73.

SEGAL, M.; AKELEY, K. **The OpenGL Graphics System**: A specification (version 1.0). Mountain View, 1994. 172 p. Citado na página 40.

SEGAL, M.; AKELEY, K. **The OpenGL Graphics System**: A specification (version 4.6 (core profile)). Beaverton, 2019. 850 p. Citado 2 vezes nas páginas 40 e 41.

SHAKTI. **RISC-V Assembly Language Programmer Manual**. Chenai, 2020. 138 p. Citado na página 34.

SHUEY, D.; BAILEY, D.; MORRISSEY, T. P. Phigs: A standard, dynamic, interactive graphics interface. **IEEE Computer Graphics and Applications**, IEEE, v. 6, n. 8, p. 50–57, 1986. Citado na página 40.

SIPEED. **Sipeed M1W Datasheet**. Shenzhen, 2019. Citado na página 64.

SIPEED. **Sipeed MaixDock Datasheet**. Shenzhen, 2019. Citado na página 63.

SITRONIX. **ST7789V Datasheet**. Hangzhou, 2014. Citado 3 vezes nas páginas 65, 67 e 68.

STALLINGS, W. **Arquitetura e organização de computadores**: Projeto para o desempenho. 5. ed. São Paulo: Pearson Education do Brasil, 2002. Citado 2 vezes nas páginas 30 e 37.

SUTHERLAND, I. E. Sketchpad a man-machine graphical communication system. **Simulation**, Sage Publications Sage CA: Thousand Oaks, CA, v. 2, n. 5, p. R–3, 1964. Citado 2 vezes nas páginas 19 e 39.

TECHNAVIO. Augmented reality and virtual reality market by technology and geography - forecast and analysis 2021-2025. Technavio, p. 120, 2021. Citado na página 20.

TIEDE, U. et al. Investigation of medical 3d-rendering algorithms. **IEEE computer graphics and applications**, IEEE, v. 10, n. 2, p. 41–53, 1990. Citado na página 17.

TINE, B. et al. Vortex: Extending the risc-v isa for gpgpu and 3d-graphics. In: **MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture**. [S.l.: s.n.], 2021. p. 754–766. Citado na página 21.

WATERMAN, A.; ASANOVIĆ, K. **The RISC-V Instruction Set Manual Volume II: Privileged Architecture**. Berkeley, 2019. 91 p. Disponível em: <https://riscv.org/technical/specifications/>. Citado 2 vezes nas páginas 28 e 29.

WATERMAN, A.; ASANOVIĆ, K. **The RISC-V Instruction Set Manual Volume I: Unprivileged ISA**. Berkeley, 2021. 244 p. Disponível em: <https://riscv.org/technical/specifications/>. Citado 6 vezes nas páginas 27, 30, 31, 32, 35 e 61.

WATERMAN, A. et al. The risc-v instruction set manual, volume i: Base user-level isa. EECS Department, UC Berkeley, v. 116, 2011. Citado 2 vezes nas páginas 19 e 26.

WILKES, M. V.; WHEELER, D. J.; GILL, S. **The Preparation of Programs for an Electronic Digital Computer**: With special reference to the edsac and the use of a library of subroutines. Reading: Addison-Wesley Press, 1951. Citado na página 32.

WU, X. An efficient antialiasing technique. **Acm Siggraph Computer Graphics**, ACM New York, v. 25, n. 4, p. 143–152, 1991. Citado na página 54.

ZHOU, Y.; JIN, X.; XIANG, T. Risc-v graphics rendering instruction set extensions for embedded ai chips implementation. In: **Proceedings of the 2020 2nd International Conference on Big Data Engineering and Technology**. [S.l.: s.n.], 2020. p. 85–88. Citado na página 21.

ZUSE, K. Über den allgemeinen plankalkül als mittel zur formulierung schematisch-kombinativer aufgaben. **Archiv der Mathematik**, Hopferau, v. 1, n. 6, p. 441–449, 1948. Citado na página 31.

高柳健次郎. Television の實驗. **電氣學會雜誌**, 一般社団法人 電気学会, v. 48, n. 482, p. 932–942, 1928. Citado 2 vezes nas páginas 18 e 52.