UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
COODERNAÇÃO DE ENGENHARIA ELÉTRICA
GRADUAÇÃO EM ENGENHARIA ELÉTRICA

LUCAS ZISCHLER

RENDERIZAÇÃO 3D EM SISTEMAS EMBARCADOS UTILIZANDO RISC-V ASSEMBLY

TRABALHO DE CONCLUSÃO DE CURSO

APUCARANA
2022

LUCAS ZISCHLER

RENDERIZAÇÃO 3D EM SISTEMAS EMBARCADOS UTILIZANDO RISC-V ASSEMBLY

3D Rendering in embedded systems utilizing RISC-V assembly

Monografia de Trabalho de Conclusão de Curso de graduação, apresentado à disciplina de Trabalho de Conclusão de Curso, em Graduação em Engenharia Elétrica da Universidade Tecnológica Federal do Paraná – UTFPR – Câmpus Apucarana, como requisito parcial para obtenção do título de Bacharel em Engenharia Elétrica.

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

Este Trabalho de Conclusão de Curso foi apresentado em 13 de Junho de 2022 como requisito parcial para a obtenção do título de Bacharel em Engenharia Elétrica. O candidato foi arguido pela Banca Examinadora composta pelos abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

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

ZISCHLER, Lucas. **Renderização 3D em sistemas embarcados utilizando RISC-V assembly**. 2022. 175 f. Trabalho de Conclusão de Curso, Graduação em Engenharia Elétrica, Universidade Tecnológica Federal do Paraná. Apucarana, 2022.

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

ZISCHLER, Lucas. **3D Rendering in embedded systems utilizing RISC-V assembly**. 2022. 175 f. Trabalho de Conclusão de Curso, Graduação em Engenharia Elétrica, Universidade Tecnológica Federal do Paraná. Apucarana, 2022.

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 19
Figura 2 – Componentes de uma instrução assembly 33
Figura 3 – Testes de contagem de instruções e tamanho de programa de diferentes ISAs 39
Figura 4 – Renderização de modelos 3D em *desktop* utilizando OpenGL 42
Figura 5 – Pipeline de renderização do OpenGL ES 42
Figura 6 – Demonstração do uso de dados contidos nos *vertexes* 43
Figura 7 – Formas de mapeamento para valores de UV fora dos limites 44
Figura 8 – Aplicação de iluminação especular 46
Figura 9 – Transformação da matriz de modelo para o mundo 48
Figura 10 – Transformação da matriz de projeção 50
Figura 11 – Rasterização de polígonos à tela 53
Figura 12 – Retas construídas pelo algoritmo de Bresenham 54
Figura 13 – Preenchimento pelo algoritmo *scan line* 56
Figura 14 – Placa de desenvolvimento Maix Dock 64
Figura 15 – Diagrama de conexão de dispositivos por SPI 67
Figura 16 – Fluxograma da configuração do *display* ST7789V 70
Figura 17 – Pinos do cartão micro SD via comunicação SPI 71
Figura 18 – Fluxograma da configuração do cartão SD 75
Figura 19 – Fluxograma da leitura de um arquivo em um sitema FAT32 83
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` 92
Figura 25 – Fluxograma da função `sdSetup` 93
Figura 26 – Fluxograma da função `tftWriteDMA` 95
Figura 27 – Fluxograma da função `sdSPIwrite` 96
Figura 28 – Fluxograma da função `sdSPIread` 97
Figura 29 – Fluxograma da função `readFile` 99
Figura 30 – Fluxograma da função `openOBJModel` 102
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 114
Figura 39 – Tempo de leitura do cartão SD 115
Figura 40 – Variação da taxa de quadros entre modelos 116
Figura 41 – Corte transversal do cone de projeção 128
Figura 42 – Implementação da verificação por CRC em hardware 132

LISTA DE TABELAS

Tabela 1 – Instruções bases e algumas extensões da arquitetura RISC-V 28
Tabela 2 – Armazenamento dos pontos flutuantes de acordo com a IEEE 754 29
Tabela 3 – Alguns CSRs definidos na ISA do RISC-V 30
Tabela 4 – Formato das instruções de 32 bits do RISC-V 32
Tabela 5 – Formato das instruções compactas de 16 bits do RISC-V 32
Tabela 6 – Convenções da ABI do RISC-V para registradores 34
Tabela 7 – Lista de alguns mnemônicos de instruções base do RISC-V 35
Tabela 8 – Hierarquias de memórias em um computador em 2015 60
Tabela 9 – Lista de comandos do *display* ST7789V referentes ao trabalho 68
Tabela 10 – Código de cores do *display* ST7789V 69
Tabela 11 – Funções do registrador de controle do *display* ST7789V 69
Tabela 12 – Formato do comando SD 71
Tabela 13 – Lista de comandos do protocolo SD referentes ao trabalho 72
Tabela 14 – Formato da resposta dos comandos SD 72
Tabela 15 – Significado dos bits de estado do cartão SD 73
Tabela 16 – Valores do OCR do cartão SD 74
Tabela 17 – Formato da resposta de leitura de dados do cartão SD 75
Tabela 18 – Definição das regiões do MBR 77
Tabela 19 – Definição das regiões da entrada da partição 78
Tabela 20 – Definição das regiões do setor de inicialização da partição 79
Tabela 21 – Regiões da entrada longa de um arquivo 80
Tabela 22 – Regiões da entrada curta de um arquivo 81
Tabela 23 – Campos de tempo, e data de uma entrada curta 82
Tabela 24 – Organização interna dos arquivos do projeto 87
Tabela 25 – Contagem das instruções das funções 113

LISTA DE ALGORITMOS

Algoritmo 1 – Exemplo de código assembly 36
Algoritmo 2 – Aplicação do algoritmo de Bresenham 55
Algoritmo 3 – Aplicação do algoritmo *scan line* 57
Algoritmo 4 – Exemplo de ambiguidade na ordenação da memória 61
Algoritmo 5 – Algoritmo de geração do valor de verificação para a entrada longa . . . 81
Algoritmo 6 – Exemplo de um modelo 3D armazenado em `.obj` 84
Algoritmo 7 – Inicialização da seção `.bss` 89
Algoritmo 8 – Liberação do núcleo 1 89
Algoritmo 9 – Alteração do valor de saída de uma GPIO de alta velocidade 92
Algoritmo 10 – Aplicação da verificação CRC por tabela 133

LISTA DE 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.
θ	Angulo de rotação.
φ	Angulo de campo de visão.

SUMÁRIO

1 INTRODUÇÃO 18
1.1 MOTIVAÇÃO 21
1.2 OBJETIVOS 22
1.2.1 Objetivos Gerais 22
1.2.2 Objetivos Específicos 22
1.3 ORGANIZAÇÃO DO TRABALHO 23
2 FUNDAMENTAÇÃO TEÓRICA 24
2.1 ARQUITETURA RISC-V 24
2.1.1 Registradores 29
2.1.2 Instruções 31
2.1.3 Linguagem Assembly 33
2.1.4 Comparação com Outras Arquiteturas 38
2.2 RENDERIZAÇÃO GRÁFICA 40
2.2.1 Vertexes 42
2.2.1.1 Posição 43
2.2.1.2 Mapa UV 44
2.2.1.3 Normal 45
2.2.2 Matrizes de Transformação 46
2.2.2.1 Matriz do modelo 47
2.2.2.1.1 *Escala* 47
2.2.2.1.2 *Rotação* 48
2.2.2.1.3 *Translação* 49
2.2.2.2 Matriz de visualização 49
2.2.2.3 Matriz de projeção 50
2.2.2.3.1 *Field of view* 50
2.2.2.3.2 *Distância de corte* 51
2.2.2.3.3 *Cone de projeção* 52
2.2.3 Rasterização 53
2.2.3.1 Algoritmo de Bresenham 54
2.2.3.2 Algoritmo scan line 56
2.3 SEGMENTAÇÃO DE MEMÓRIA 58
2.3.1 Ordenação de Memória Fraca RVWMO 61
3 METODOLOGIA 64
3.1 HARDWARE 64
3.2 MICROCONTROLADOR K210 65
3.2.1 Periféricos 65

3.3 PROTOCOLOS DE COMUNICAÇÃO . . . 66
3.3.1 SPI . . . 67
3.3.1.1 Display TFT . . . 67
3.3.1.2 Protocolo SD . . . 70
3.4 ARMAZENAMENTO DE DADOS . . . 75
3.4.1 Sistema de Arquivos FAT . . . 76
3.5 ARQUIVOS EXTERNOS . . . 82
3.5.1 Arquivo Para Modelos 3D .obj . . . 83
3.5.2 Arquivo Para Texturas .ppm . . . 84
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 . . . 93
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 . . . 100
4.5.2 Arquivos .ppm . . . 103
4.6 RENDERIZAÇÃO DO MODELO . . . 104
4.6.1 Matrizes de Transformação . . . 105
4.6.2 Rasterização de Polígonos . . . 106
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

Apêndices 124
APÊNDICE A Quaterniões . . . 125

A.1 Matriz De Rotação 126
APÊNDICE B Dedução Da Matriz De Projeção 128
APÊNDICE C Cyclic Redundancy Check (CRC) 131
APÊNDICE D Algoritmo 134

Anexos 166
ANEXO A Instruções RISC-V 167
ANEXO B Diretivos assembly 175

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.

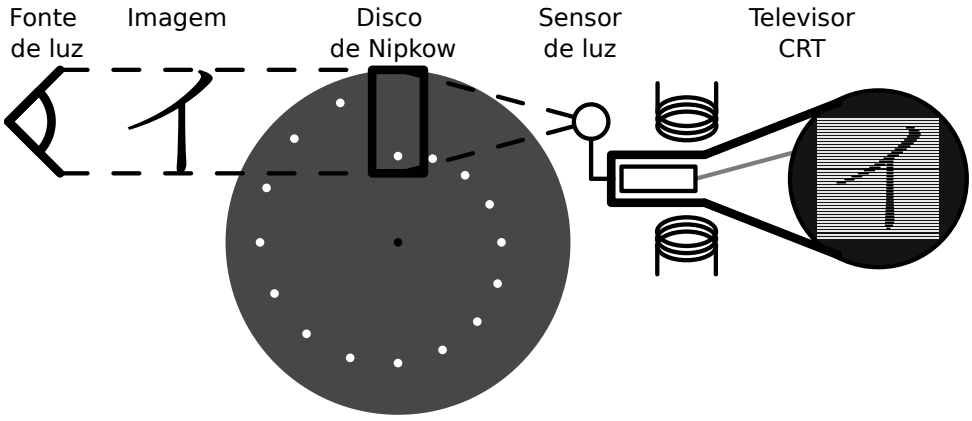


Figura 1 – Rasterização com um disco de Nipkow em um televisor CRT
Fonte: Ilustração do trabalho de Takayanagi (高柳健次郎, 1928)

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 Tabela 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.

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*.

Tabela 1 – Instruções bases e algumas extensões da arquitetura RISC-V
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 2, 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.

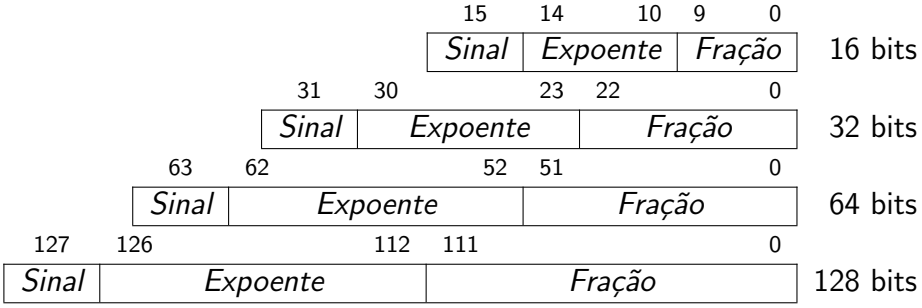


Tabela 2 – Armazenamento de pontos flutuantes de acordo com a IEEE 754
Fonte: (IEEE, 2008)

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 Tabela 3 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.

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*

Tabela 3 – Alguns CSRs definidos na ISA do RISC-V
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 4, 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

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

Tabela 4 – Formato das instruções de 32 bits do RISC-V
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 5 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.

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

Tabela 5 – Formato das instruções compactas de 16 bits do RISC-V
Fonte: (WATERMAN; ASANOVIĆ, 2021)

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 "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.

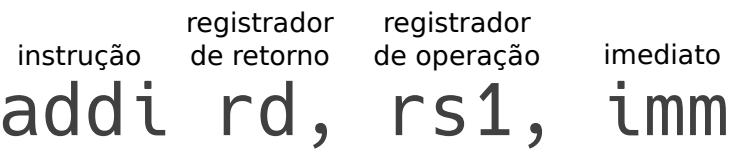


Figura 2 – Componentes de uma instrução assembly
Fonte: Autoria Própria

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 4. Isto permite uma otimização do compilador, em relação a uma operação do tipo `rd=rs1+imm`, porque nesse caso o operador 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 Tabela 6 mostra as convenções de utilização e nomeação dos registradores de inteiros e pontos flutuantes.

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

Tabela 6 – Convenções da ABI do RISC-V para registradores
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 Tabela 7, sendo alguns diferentes das instruções em si, decompostos em outras instruções antes da geração do código de máquina. Uma lista completa de mnemônicos para um processador *RV64GC* está presente no Anexo A.

Inst	Parâmetros	Descrição
	RV32I	
`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`

Tabela 7 – Lista de alguns mnemônicos de instruções base do RISC-V

Fonte: (WATERMAN; ASANOVIĆ, 2021)

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

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.

Algoritmo 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
```

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 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 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. Uma lista de diretivos da linguagem assembly está presente no Anexo B.

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.

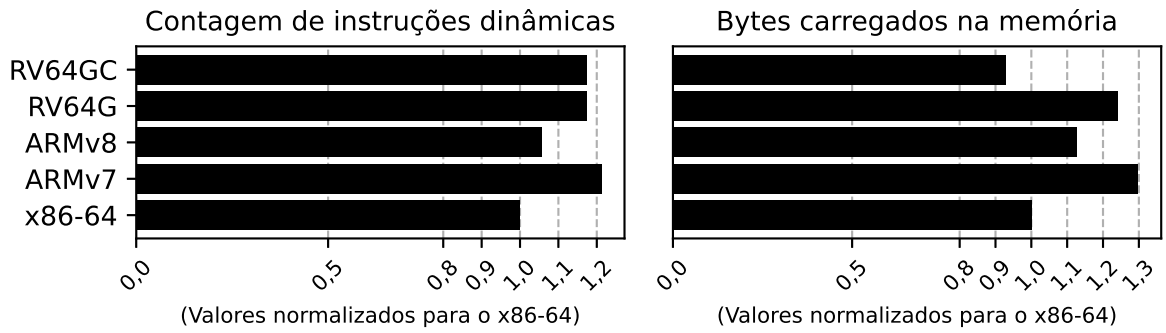


Figura 3 – Testes de contagem de instruções e tamanho de programa de diferentes ISAs
Fonte: Realizada a média aritmética dos resultados de Celio et al. (2016)

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

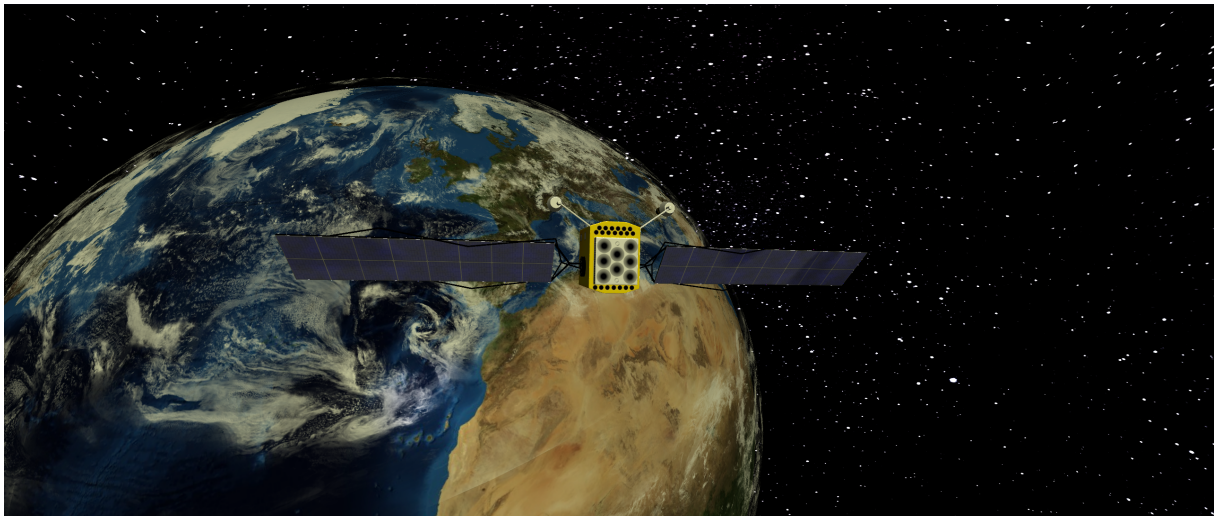


Figura 4 – Renderização de modelos 3D em *desktop* utilizando OpenGL
Fonte: Autoria própria

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*.

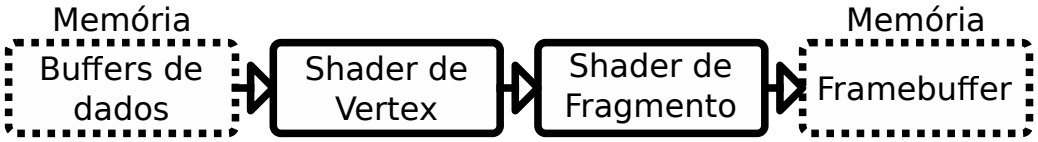


Figura 5 – Pipeline de renderização do OpenGL ES
Fonte: (SEGAL; AKELEY, 2019)

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

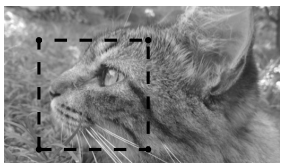
ser passados fora de um *vertex*, como em técnicas de tesselação, em que novos polígonos são gerados com base em texturas específicas. Por conta destes fatores, se evitará realizar esta ligação entre essas palavras.

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).

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



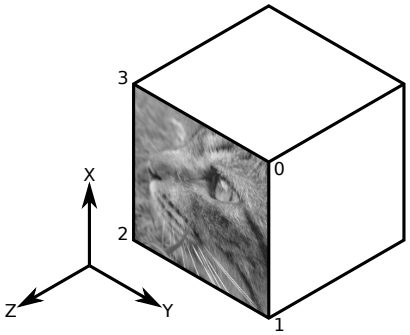


Figura 6 – Demonstração do uso de dados contidos nos *vertexes*
Fonte: Autoria própria

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

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).

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

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.

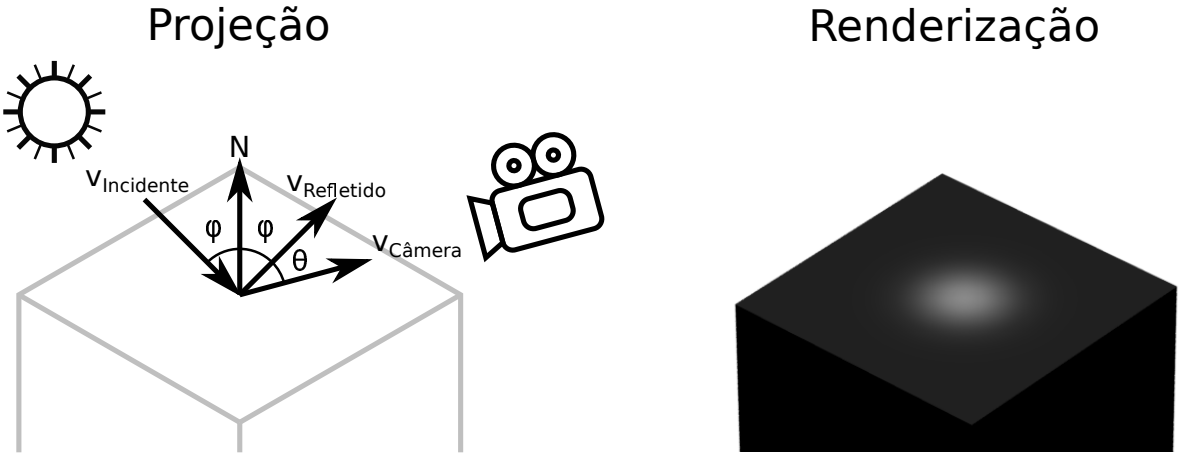


Figura 8 – Aplicação de iluminação especular
Fonte: Autoria própria

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.

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

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.

$$\begin{gathered}(x,y,z,w)^T_{Tela} = M_P M_V M_M (x,y,z,w)^T_{Modelo} \\ M_P, M_V, M_M \in \mathbb{R}^{4\times4}\end{gathered} \tag{6}$$

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.

$$\begin{gathered}(x,y,z,w)^T_{Mundo} = M_M = M_T M_R M_E (x,y,z,w)^T_{Modelo} \\ M_T, M_R, M_E \in \mathbb{R}^{4\times4}\end{gathered} \tag{7}$$

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

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

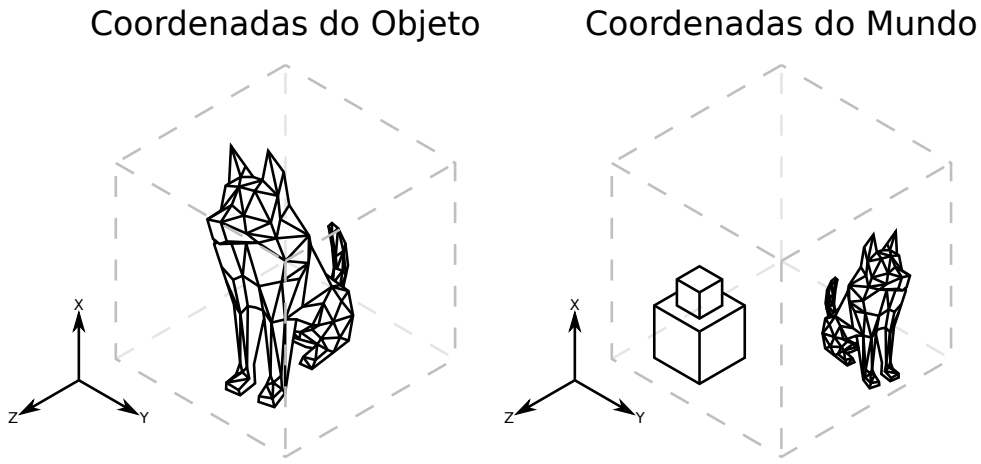


Figura 9 – Transformação da matriz de modelo para o mundo
Fonte: Autoria Própria

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 estudo mais detalhado sobre, e a comprovação detalhada da obtenção da matriz de rotação pode ser lido no Apêndice A.

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.

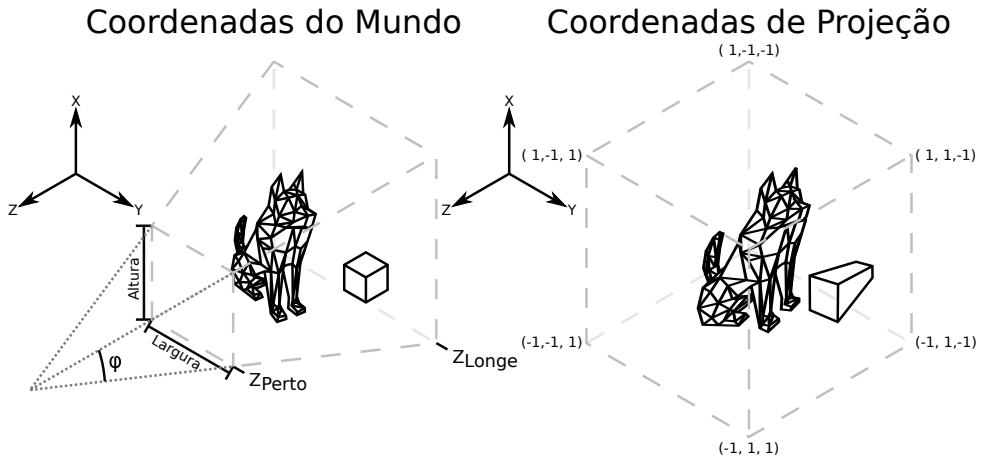


Figura 10 – Transformação da matriz de projeção
Fonte: Autoria Própria

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 "φ". 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 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. Uma análise em detalhe sobre os termos da matriz de projeção é dado no Apêndice B.

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

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.

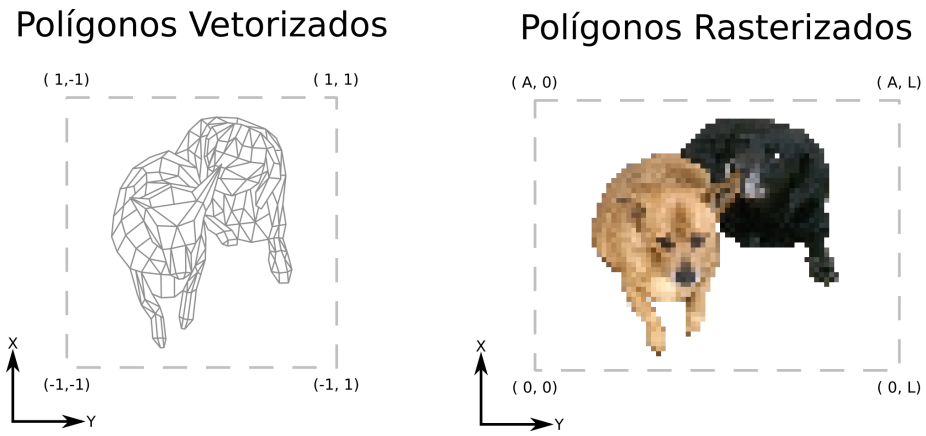


Figura 11 – Rasterização de polígonos à tela
Fonte: Autoria Própria

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 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.

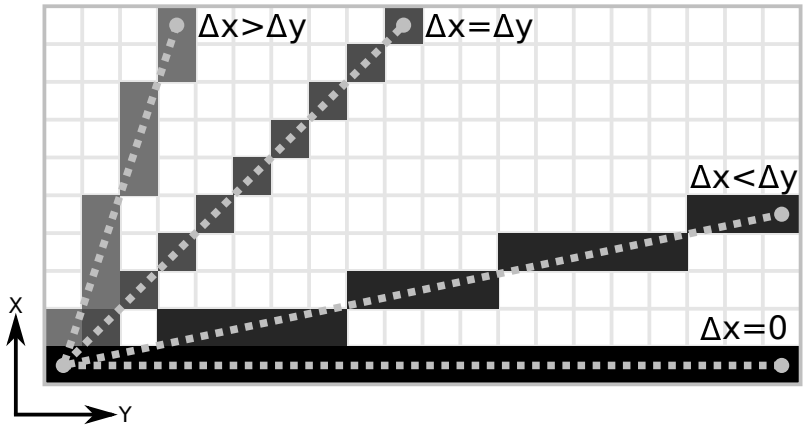


Figura 12 – Retas construídas pelo algoritmo de Bresenham
Fonte: Autoria Própria

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

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

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 Δer. 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.

Algoritmo 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: $x \leftarrow x + sinal$; $erro \leftarrow erro - 1$;

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.

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.

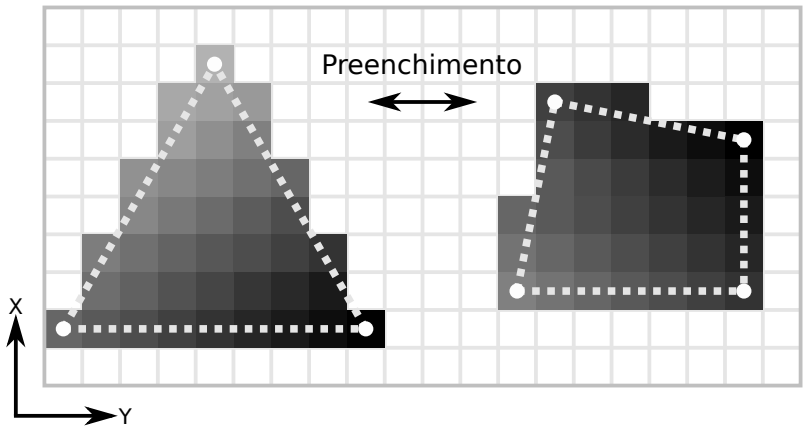


Figura 13 – Preenchimento pelo algoritmo *scan line*
Fonte: Autoria Própria

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 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.

Algoritmo 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**$(x,y) \leftarrow valor$;

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, 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 8.

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

Tabela 8 – Hierarquias de memórias em um computador em 2015
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.

Algoritmo 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;
```

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.

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.

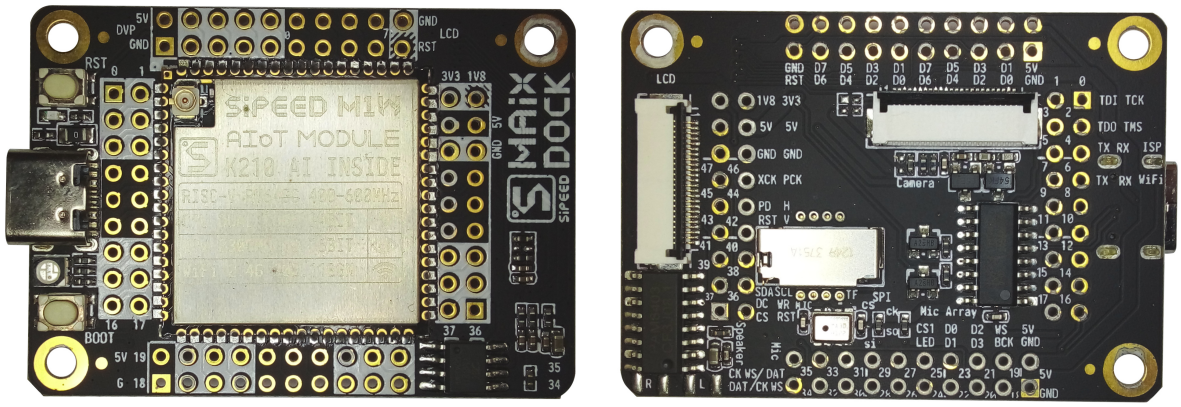


Figura 14 – Placa de desenvolvimento Maix Dock
Fonte: Autoria Própria

Em sua parte superiora, é posicionado o módulo M1W, desenvolvido também pela Sipeed (2019a). Este módulo contem o microcontrolador K210, sendo responsável pelo proces-

samento, 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.

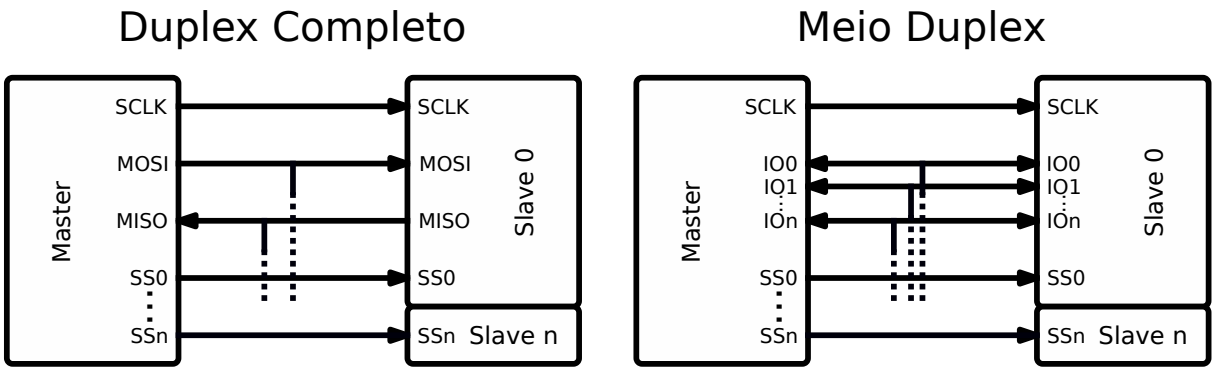


Figura 15 – Diagrama de conexão de dispositivos por SPI
Fonte: Autoria Própria

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 Tabela 9.

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*

Tabela 9 – Lista de comandos do *display* ST7789V referentes ao trabalho
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 Tabela 10. 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.

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

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

Tabela 10 – Código de cores do *display* ST7789V
Fonte: (SITRONIX, 2014)

`MEMORY_ACCESS_CTL`. Dentro deste, diversos parâmetros de operação podem ser ajustados. A função e seu respectivo bit podem ser vistos na Tabela 11.

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		

Tabela 11 – Funções do registrador de controle do *display* ST7789V
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.

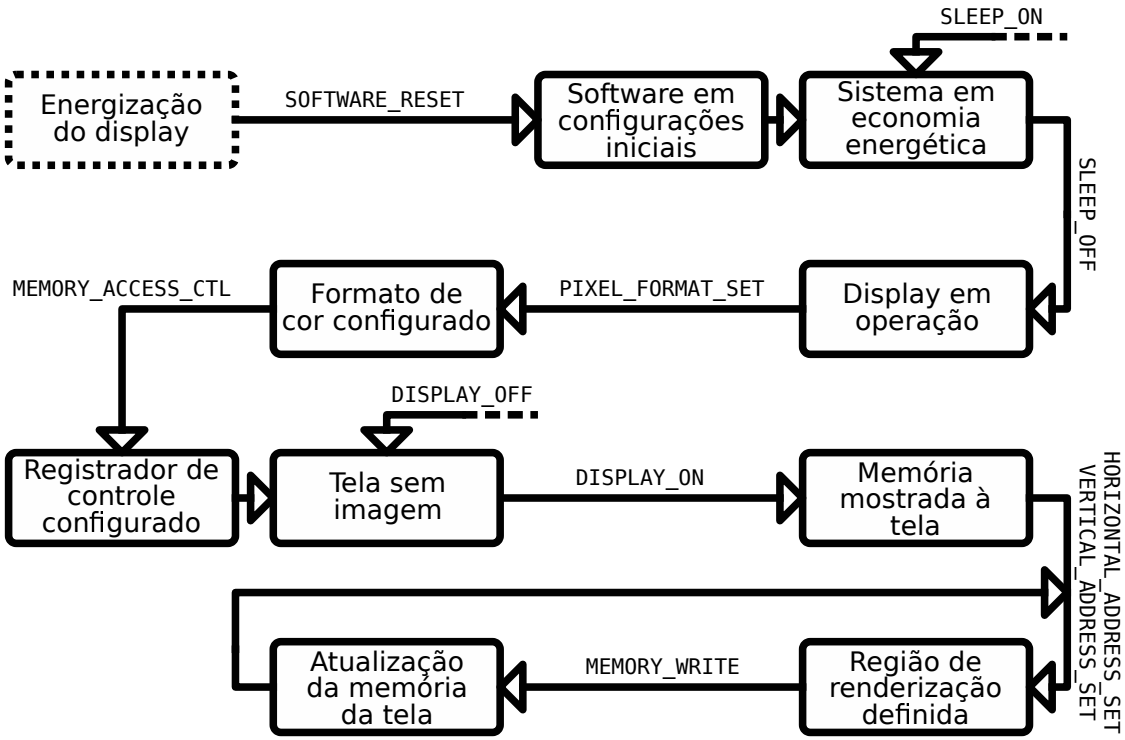


Figura 16 – Fluxograma da configuração do *display* ST7789V

Fonte: Autoria Própria

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.

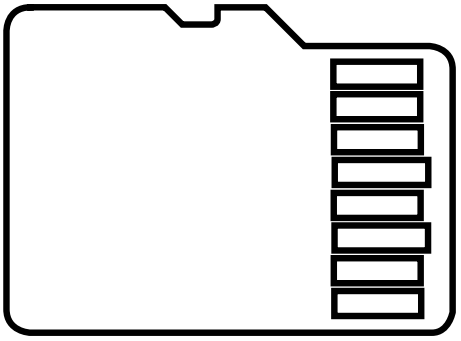
Desde sua implementação, diversas tecnologias diferentes foram implementadas no protocolo SD, contudo, todas mantém ainda suporte legado aos demais dispositivos. Deste modo é necessário várias ramificações do programa para utilizar de forma eficiente o cartão inserido.

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.



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

Figura 17 – Pinos do cartão micro SD via comunicação SPI
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 12. 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.

47	46	45 … 40	39 … 8	7 … 1	0
0	*1*	*Índice do Comando*	*Argumento*	*CRC7*	*1*

Tabela 12 – Formato do comando SD
Fonte: (SD Card Association, 2018)

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 $x^7 + x^3 + 1$. Este método é abordado em mais detalhe no Apêndice C.

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 Tabela 13. 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.

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

Tabela 13 – Lista de comandos do protocolo SD referentes ao trabalho
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 14.

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
0	*0*	*Índice*	*0x00000* – *PCIe* – *Tensão aceita* – *Verificação*	*CRC7*	*1*	R7

Tabela 14 – Formato da resposta dos comandos SD
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 Tabela 15. Para garantir a prontidão do cartão, é recomendado repetir o comando `CMD0` até que se obtenha êxito na resposta do estado.

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

Tabela 15 – Significado dos bits de estado do cartão SD
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 Tabela 16. 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.

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		

Tabela 16 – Valores do OCR do cartão SD
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 17. 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.

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

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

Tabela 17 – Formato da resposta de leitura de dados do cartão SD
Fonte: (SD Card Association, 2018)

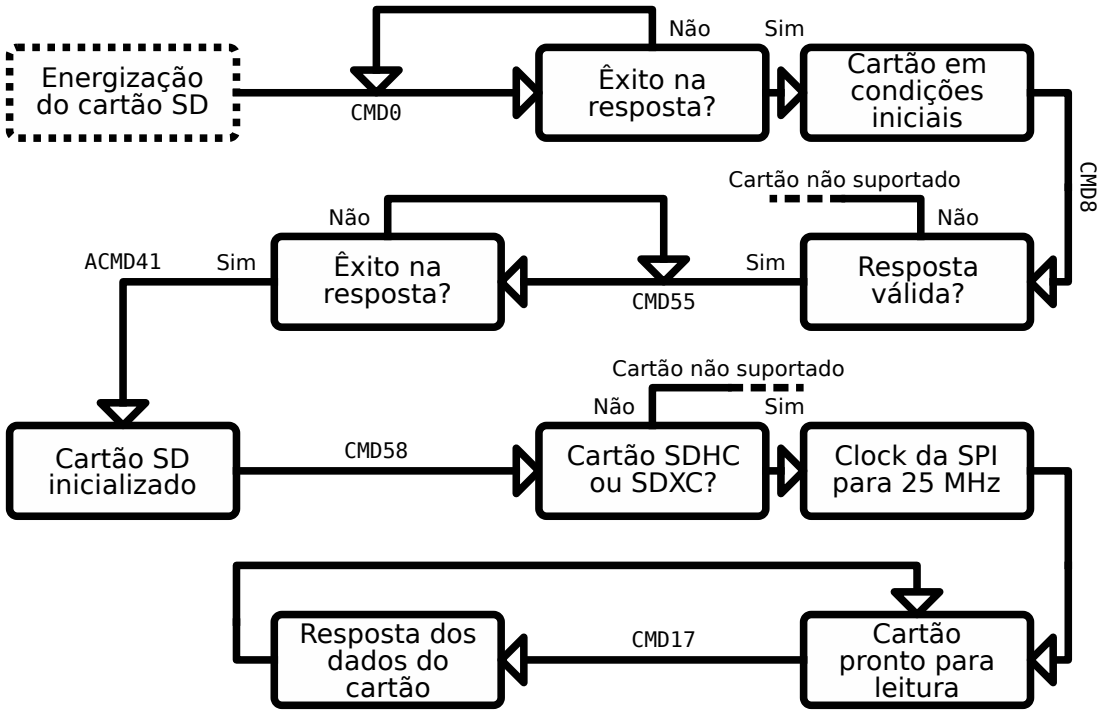


Figura 18 – Fluxograma da configuração do cartão SD
Fonte: Autoria Própria

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 Tabela 18. 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.

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]

Tabela 18 – Definição das regiões do MBR
Fonte: (MICROSOFT, 2005)

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 Tabela 19. 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 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.

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

Tabela 19 – Definição das regiões da entrada da partição
Fonte: (MICROSOFT, 2005)

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 Tabela 20.

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 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*.

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]

Tabela 20 – Definição das regiões do setor de inicialização da partição
Fonte: (MICROSOFT, 2005)

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 Tabela 21.

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

Tabela 21 – Regiões da entrada longa de um arquivo
Fonte: (CARRIER, 2005)

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

Algoritmo 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 Tabela 22.

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

Tabela 22 – Regiões da entrada curta de um arquivo
Fonte: (CARRIER, 2005)

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.

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 23. 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.

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

Tabela 23 – Campos de tempo, e data de uma entrada curta
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.

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.

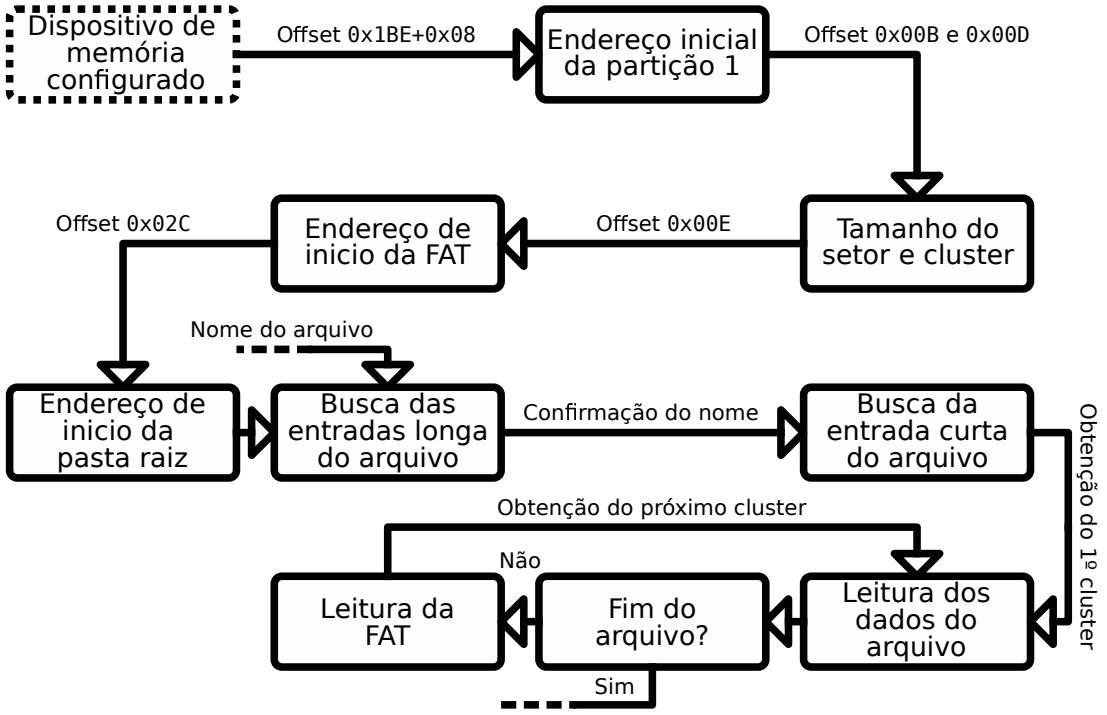


Figura 19 – Fluxograma da leitura de um arquivo em um sitema FAT32
Fonte: Autoria Própria

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.

Algoritmo 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
```

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.

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.

Arquivo .ppm

```
P3
8 12
255
0 0 77 0 0 77 0 0 77 0 0 77 0 0 77 0 0 77 0 0
77 0 0 77 0 0 77 0 0 77 0 0 255 87 87 255 87
87 255 87 87 255 0 0 77 0 0 77 0 0 77 0 255 0
0 0 255 0 0 255 87 87 255 188 255 188 188
255 188 0 0 77 0 0 189 0 255 0 0 255 0 0 0
255 0 255 0 0 255 0 188 255 188 188 255 188
0 0 189 0 255 0 0 255 0 0 0 255 0 0 255 0 255
0 0 255 0 188 255 188 0 0 189 0 0 255 0 255 0
0 255 0 0 0 255 0 255 0 0 255 0 87 87 255
0 0 189 0 0 189 0 144 0 0 0 255 0 0 255 0 255
0 0 0 255 0 0 255 0 0 77 238 238 238 0 0 189
0 0 189 0 0 255 0 0 255 209 209 209 209 209
209 0 0 77 0 0 77 238 238 238 238 238 238
238 238 238 112 112 112 176 176 176 209
209 209 0 0 77 0 0 77 0 0 77 0 0 77 0 0 77 112
112 112 176 176 176 176 176 176 0 0 77 0 0
77 0 0 77 0 0 77 0 0 77 112 112 112 112 112
112 176 176 176 0 0 77 0 0 77 0 0 77 0 0 77 0
0 77 0 0 77 112 112 112 112 112 112
```

Imagem

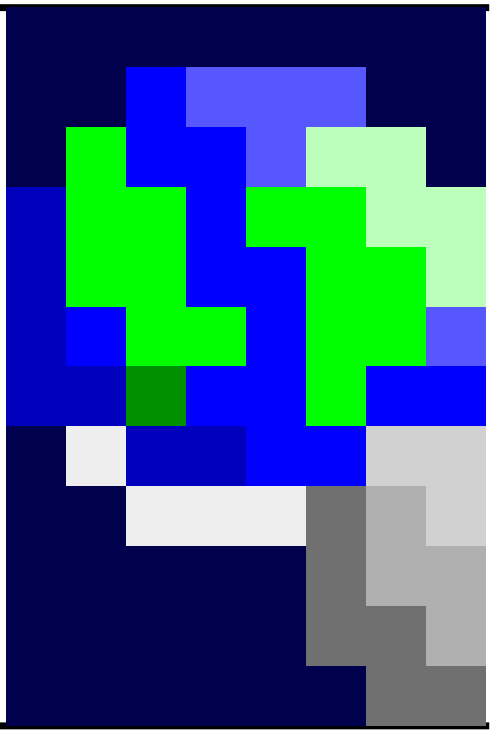


Figura 20 – Exemplo de um arquivo de uma imagem no formato .ppm

Fonte: Autoria Própria

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.

O algoritmo por completo pode ser visto no Apêndice D. 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

[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

microcontrolador K210 também se enquadra neste caso. Com um compilador dedicado de 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 24.

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

Tabela 24 – Organização interna dos arquivos do projeto
Fonte: Autoria Própria

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.

[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

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 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.

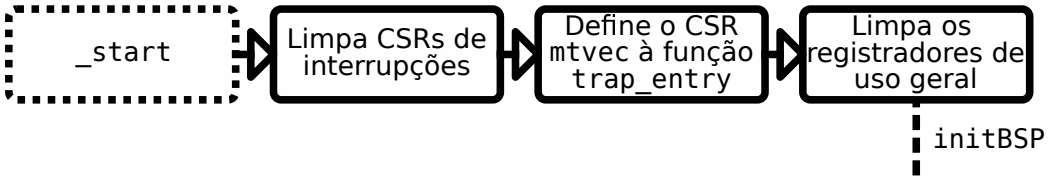


Figura 21 – Fluxograma da função `_start`
Fonte: Autoria Própria

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.

Algoritmo 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
```

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 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.

Algoritmo 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.

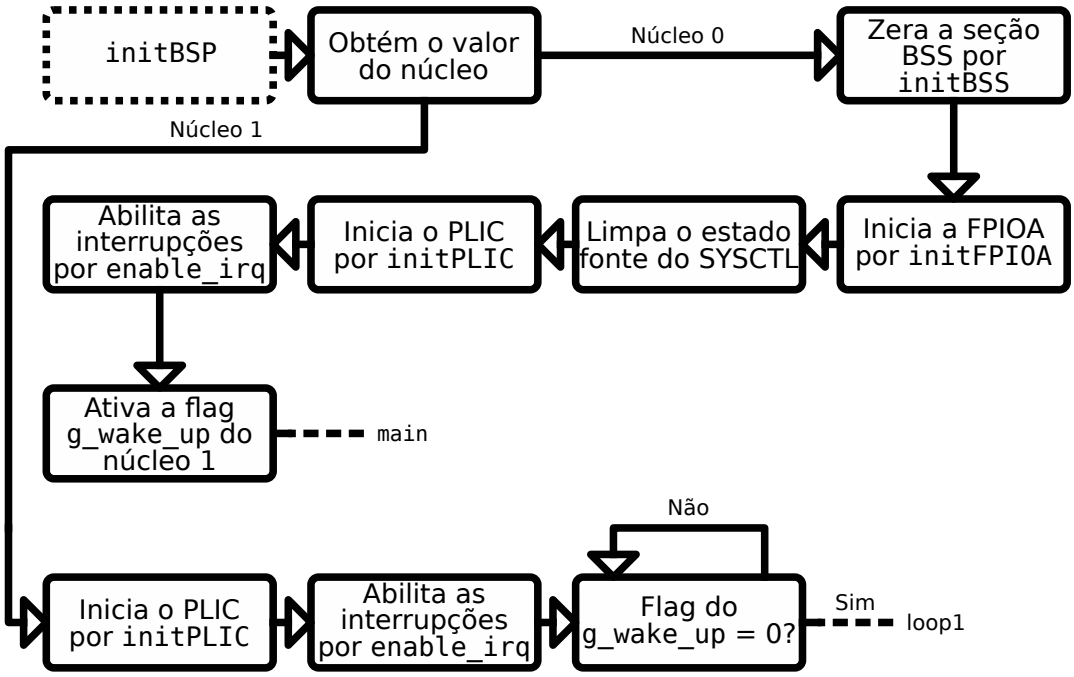


Figura 22 – Fluxograma da função `initBSP`

Fonte: Autoria Própria

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.

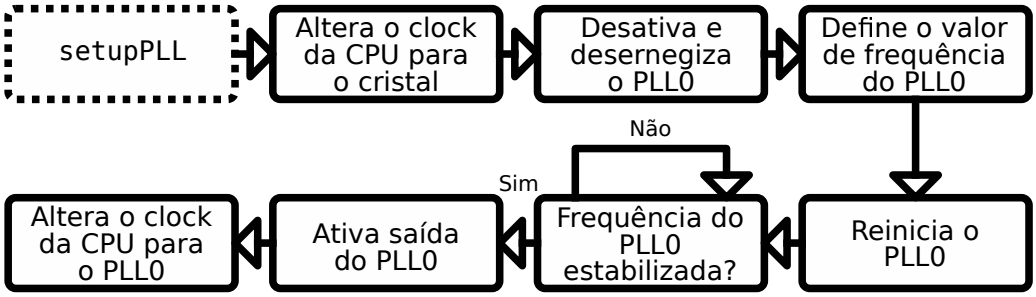


Figura 23 – Fluxograma da função `setupPLL`
Fonte: Autoria Própria

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

Algoritmo 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.

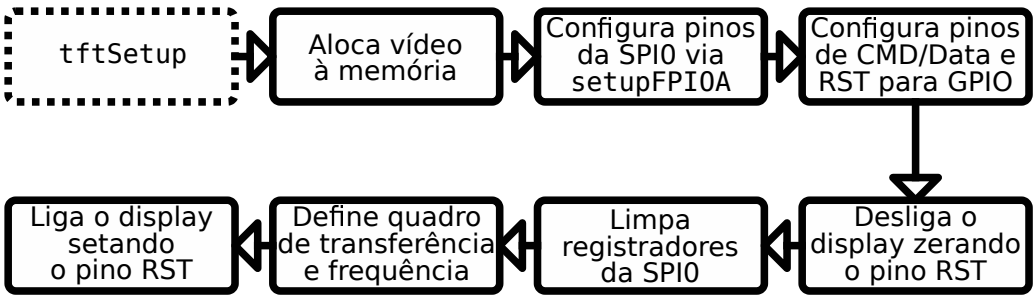


Figura 24 – Fluxograma da função `tftSetup`

Fonte: Autoria Própria

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.

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

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.

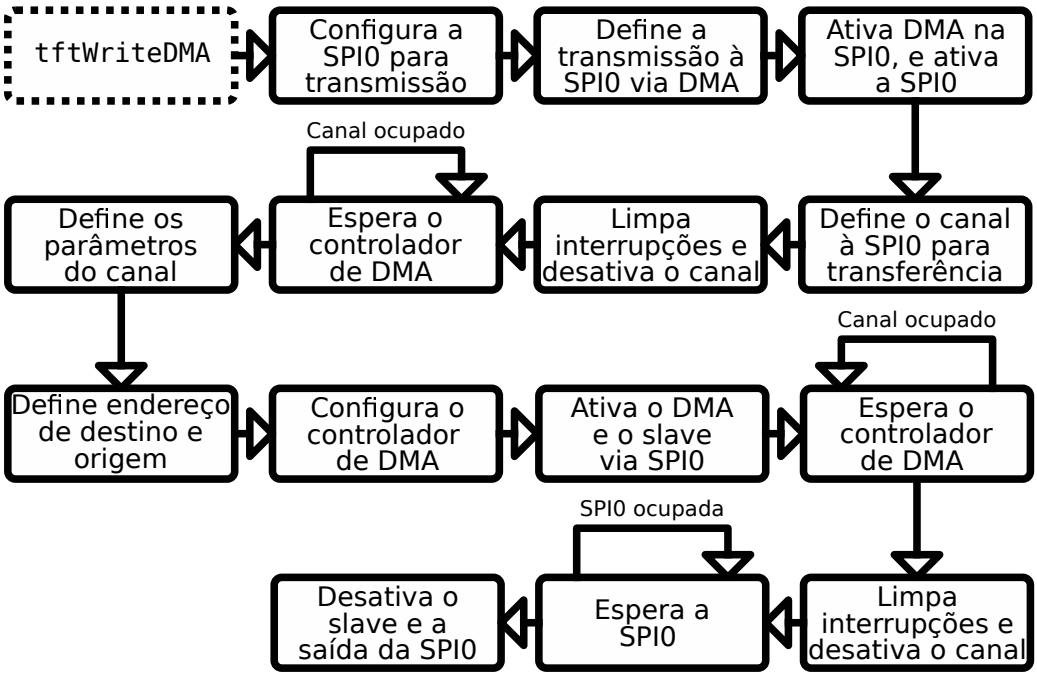


Figura 26 – Fluxograma da função `tftWriteDMA`
Fonte: Autoria Própria

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

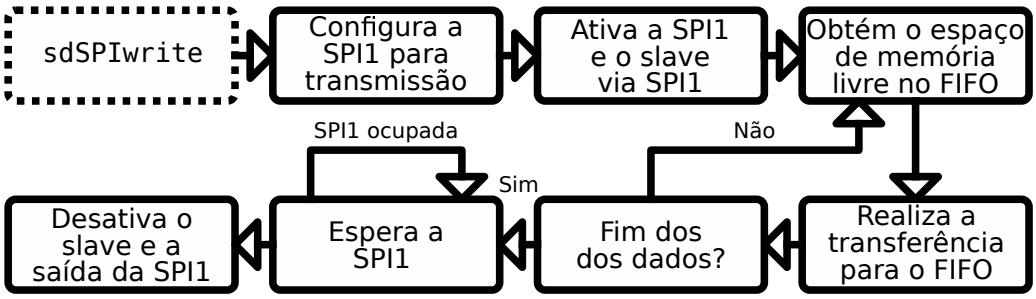


Figura 27 – Fluxograma da função `sdSPIwrite`
Fonte: Autoria Própria

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 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.

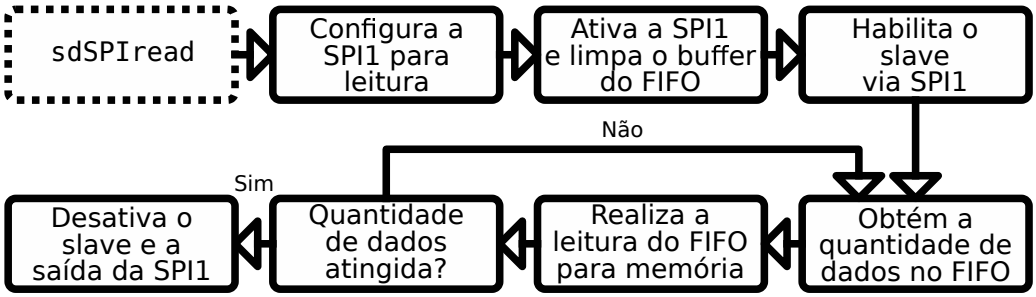


Figura 28 – Fluxograma da função `sdSPIread`
Fonte: Autoria Própria

4.3.2.1 Verificação CRC

A estrutura de um comando via protocolo SD foi previamente decomposto na Tabela 12. 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 visto no Apêndice C.

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.

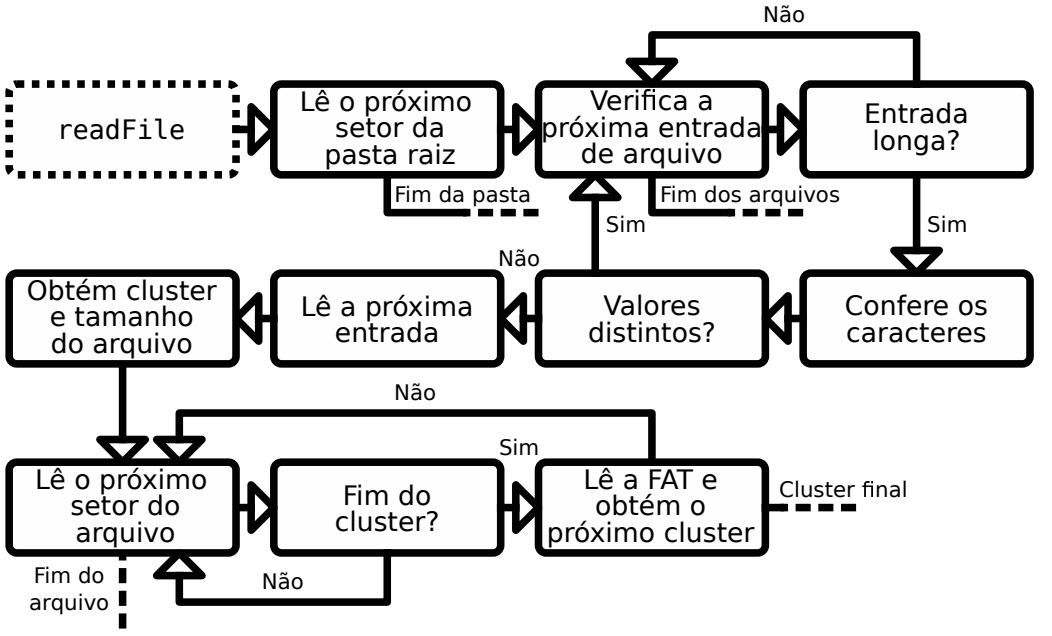


Figura 29 – Fluxograma da função `readFile`
Fonte: Autoria Própria

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 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.

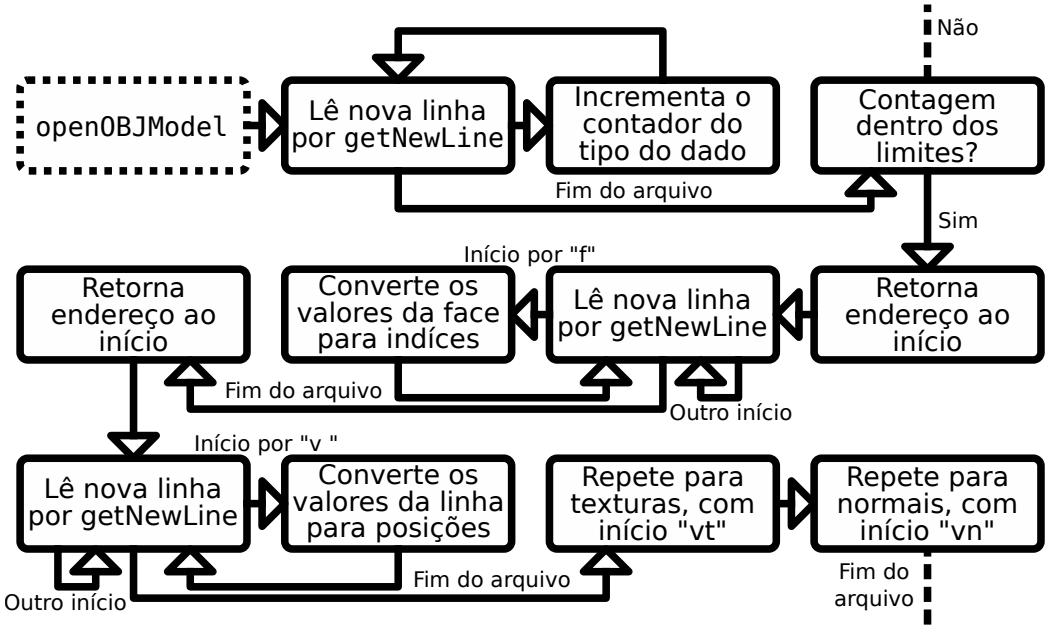


Figura 30 – Fluxograma da função `openOBJModel`

Fonte: Autoria Própria

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 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.

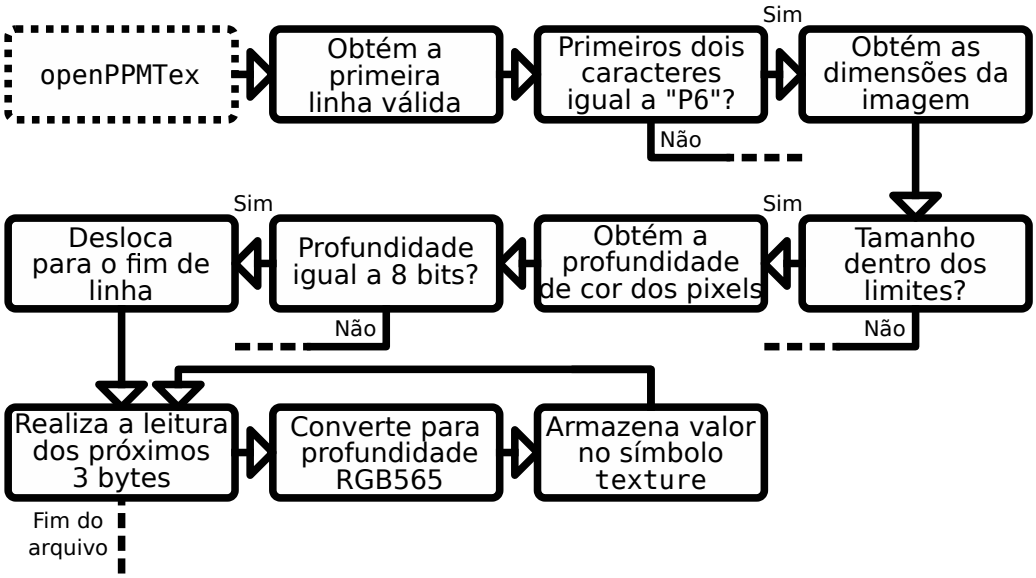


Figura 31 – Fluxograma da função `openPPMTex`
Fonte: Autoria Própria

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 conside-

rado 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.

Com isto, as linhas são preenchidas, utilizando a função `fillPolygon`. Nesta é

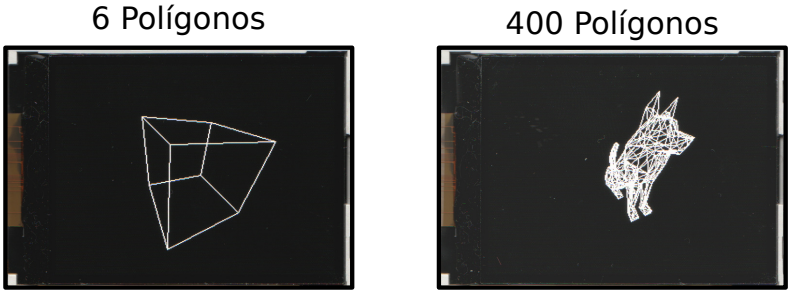


Figura 32 – Renderização de linhas do modelo
Fonte: Autoria Própria

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.

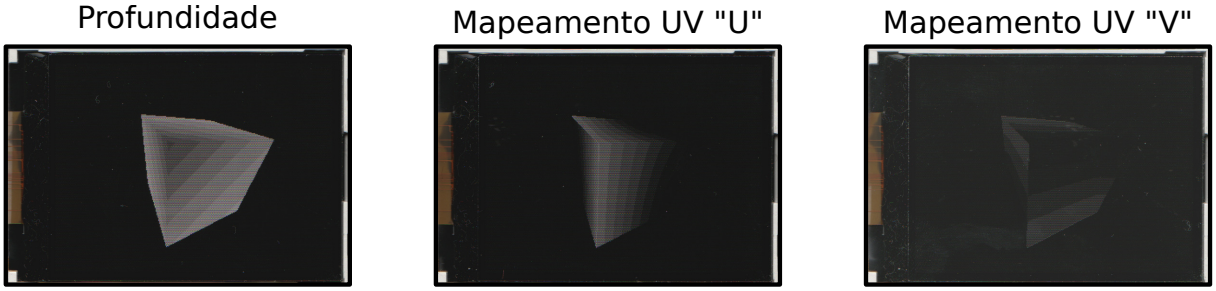


Figura 33 – Identificação dos *buffers* de trabalho por cor
Fonte: Autoria Própria

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.

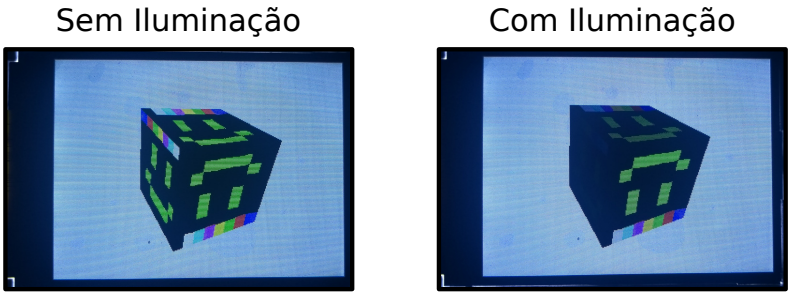


Figura 34 – Aplicação da iluminação no modelo
Fonte: Autoria Própria

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.

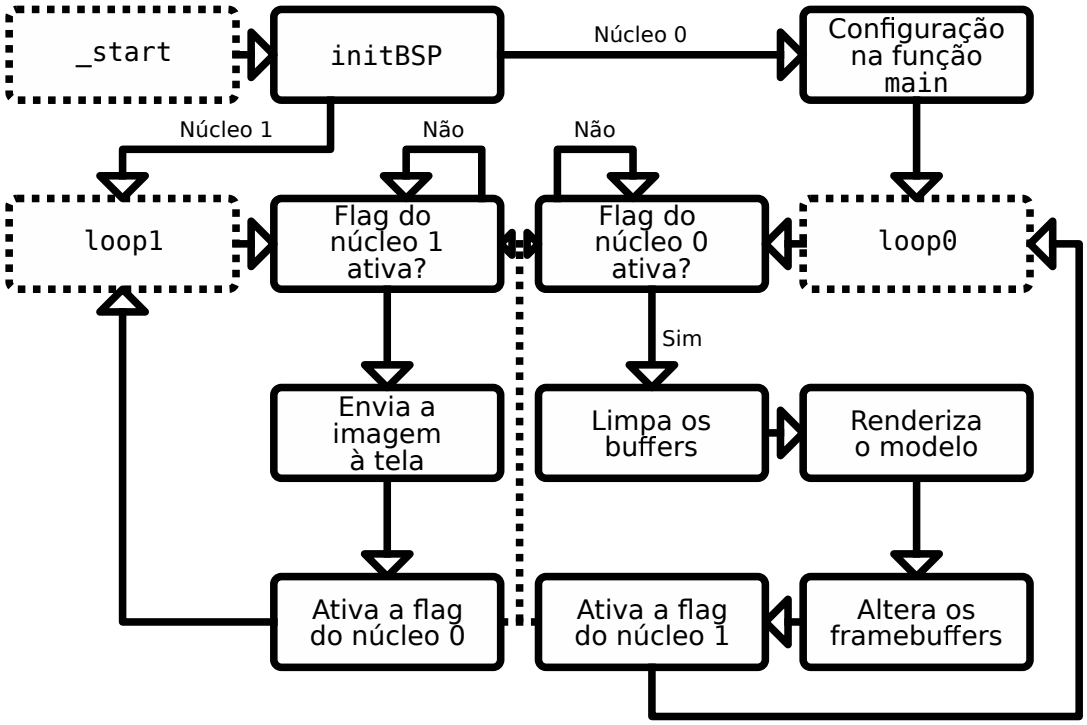


Figura 35 – Fluxograma das etapas de *loop* do algoritmo

Fonte: Autoria Própria

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.

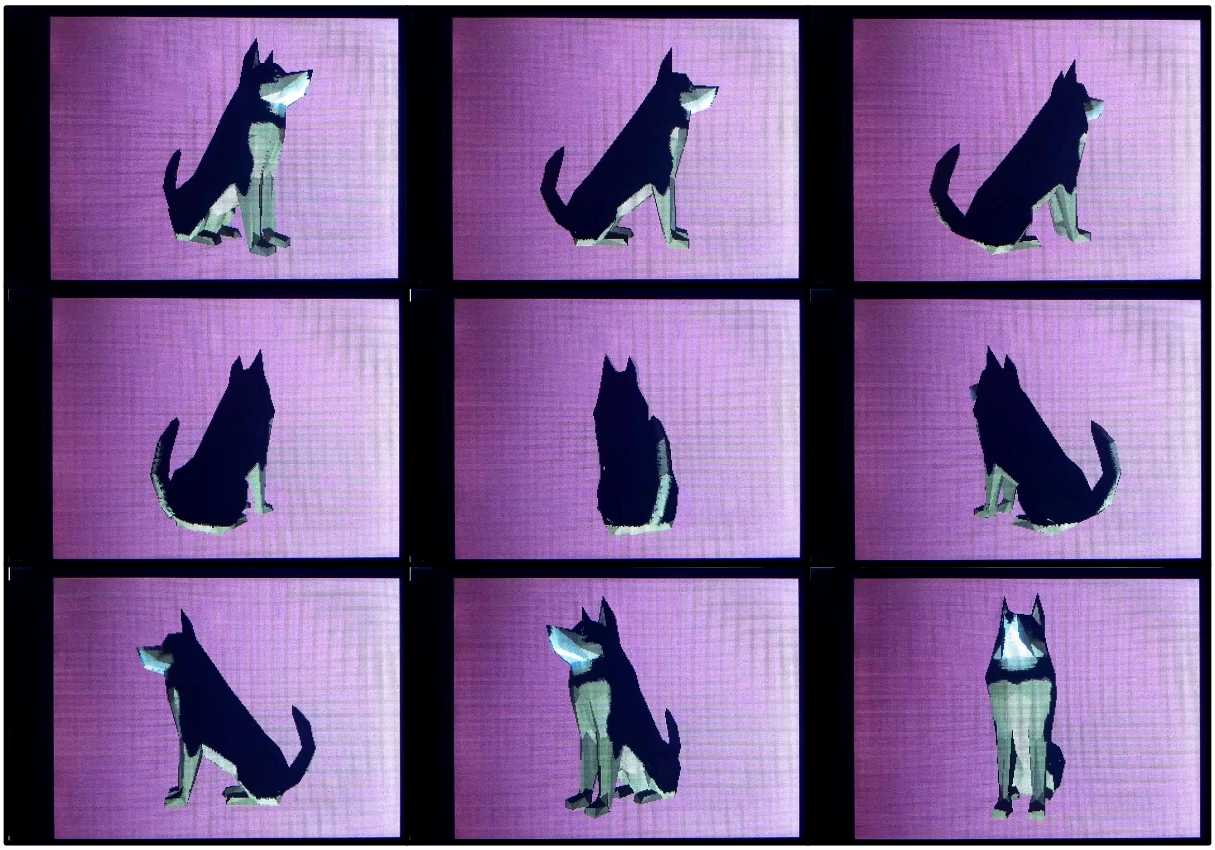


Figura 36 – Rotação de um modelo com diferença de 1 s entre quadros
Fonte: Autoria Própria

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 25. 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 25. 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 alterando-se os nomes buscados definidos em algoritmo. Deste modo, para uma comparação

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

Tabela 25 – Contagem das instruções das funções
Fonte: Autoria Própria

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 na Figura 38. Neste caso, é contabilizado o tempo que um LED auxiliar demora pra ligar,

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%

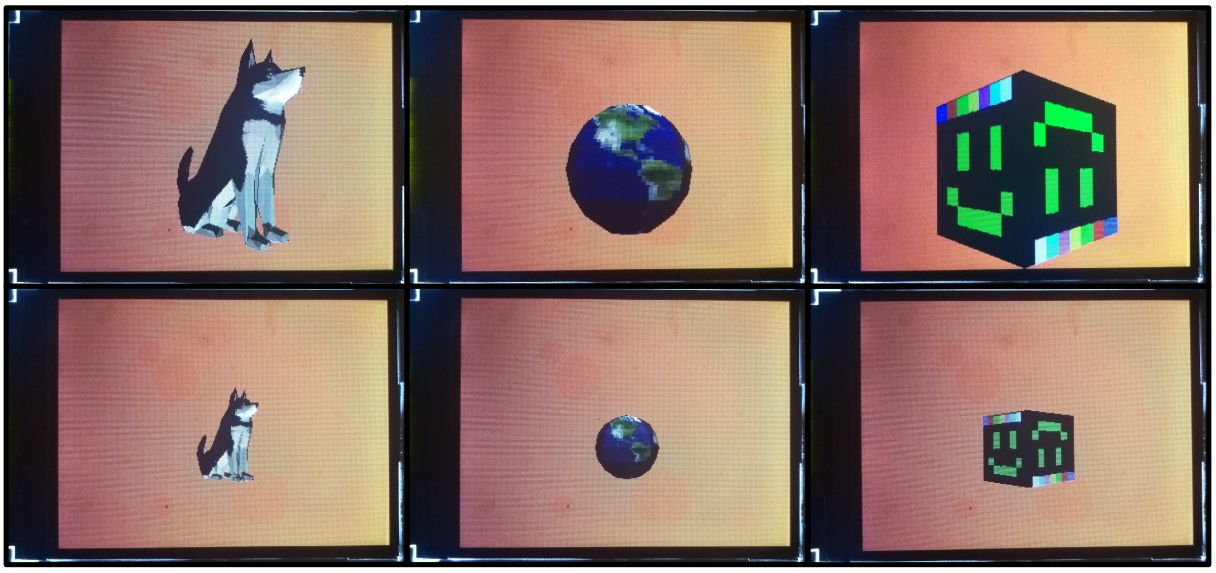


Figura 37 – Variações dos modelos e texturas utilizados para testes
Fonte: Autoria Própria

sendo este, ligado no momento que o algoritmo entra na etapa de *loop*. Antes da energização própriamente 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.

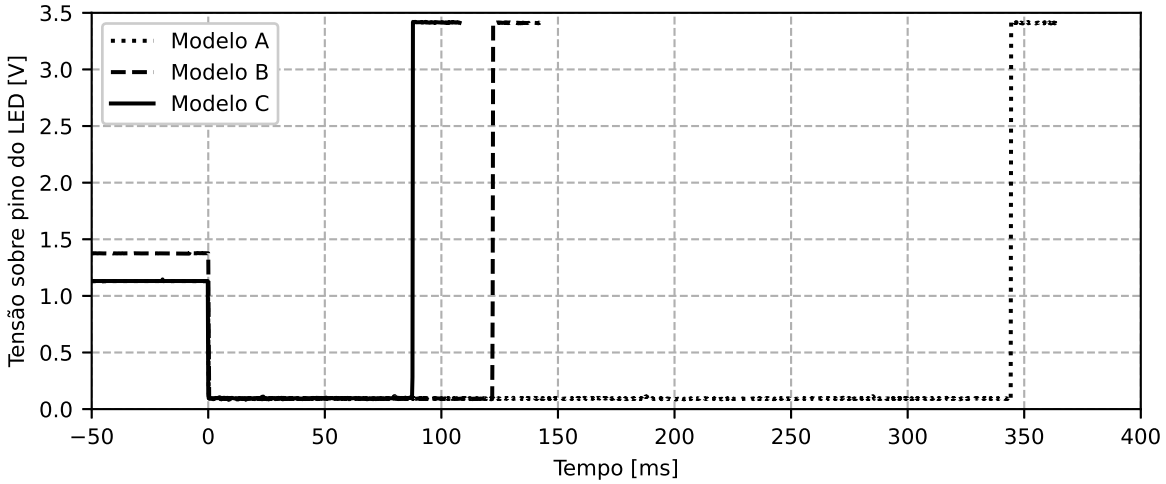


Figura 38 – Tempo de inicialização entre modelos
Fonte: Autoria Própria

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 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.

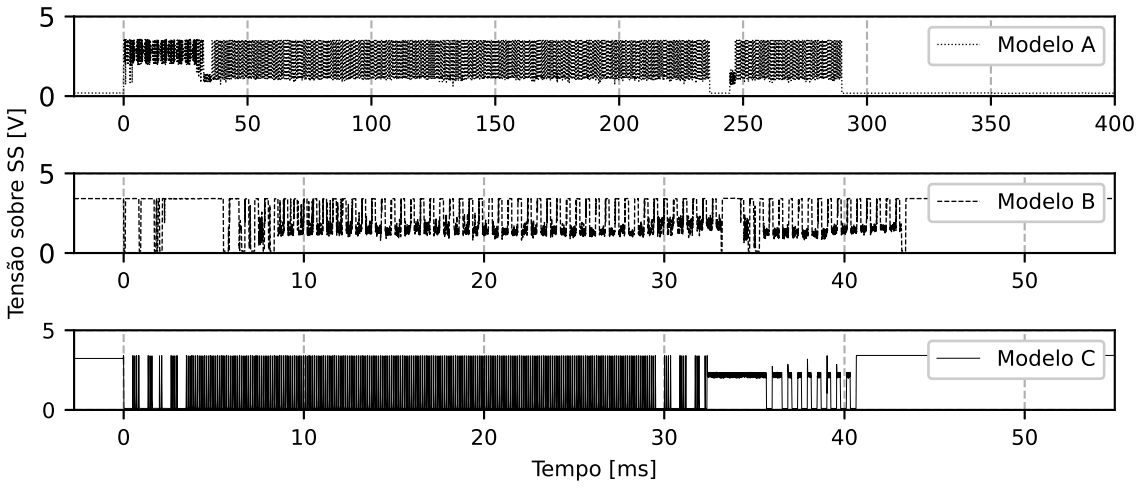


Figura 39 – Tempo de leitura do cartão SD
Fonte: Autoria Própria

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.

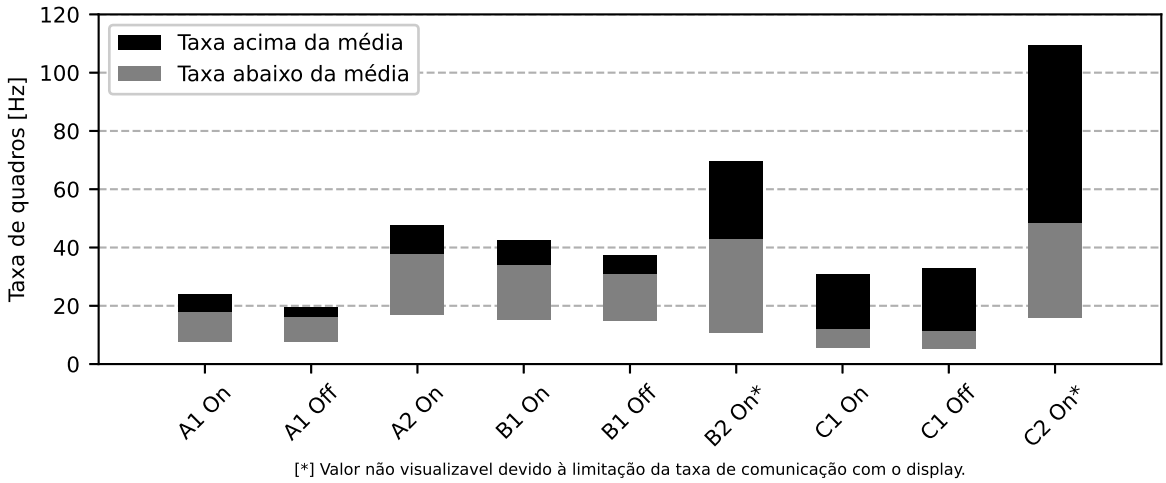


Figura 40 – Variação da taxa de quadros entre modelos
Fonte: Autoria Própria

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 27.

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 62.

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 26.

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 20.

ARM. **ARM architecture reference manual**. Cambridge, 2005. Citado na página 27.

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 56.

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 58.

BRAUN, K. F. Electrical oscillations and wireless telegraphy. **Nobel Lecture, December**, v. 11, n. 1909, p. 226–245, 1909. Citado na página 19.

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 54.

CANAAN. **K210 Datasheet**. Hangzhou, 2018. Citado na página 65.

CARRIER, B. **File system forensic analysis**. Upper Saddle River: Addison-Wesley Professional, 2005. Citado 3 vezes nas páginas 80, 81 e 82.

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 39.

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 46.

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 19.

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 38.

FOLEY, J. D. **Computer Graphics**: Principles and practice. 2. ed. Boston: Addison-Wesley, 1996. Citado na página 47.

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 58.

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 18.

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 2 vezes nas páginas 48 e 125.

HARRIS, S. L.; HARRIS, D. **Digital design and computer architecture**: Arm edition. 2. ed. São Francisco: Morgan Kaufmann, 2015. Citado na página 60.

HOLLERITH, H. The electrical tabulating machine. **Journal of the Royal Statistical Society**, JSTOR, v. 57, n. 4, p. 678–689, 1894. Citado na página 24.

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 29.

INTEL. **Intel 8086 Family User's Manual October 1979**. Santa Clara, 1979. 208 p. Citado na página 26.

INTEL. **Intel 64 and IA-32 Architectures Software Developer's Manual**. Santa Clara, 2021. 4778 p. Citado na página 39.

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 21.

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 62.

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 33.

MACHOVER, C. A brief, personal history of computer graphics. **Computer**, IEEE, v. 11, n. 11, p. 38–45, 1978. Citado na página 40.

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 59.

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 40.

MICROSOFT. **Microsoft FAT Specification**. Redmond, 2005. Citado 4 vezes nas páginas 76, 77, 78 e 79.

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 18.

MOORE, G. E. Cramming more components onto integrated circuits. **Electronics**, McGraw-Hill Education, v. 38, n. 8, p. 4, 1965. Citado na página 26.

NEIDER, J.; DAVIS, T.; WOO, M. **OpenGL programming guide**. Reading: Addison-Wesley, 1993. v. 478. Citado na página 43.

NIPKOW, P. Der telephotograph und das elektrische teleskop. **Elektrotechnische Zeitschrift**, v. 6, p. 419–425, 1885. Citado na página 19.

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 26 e 60.

PETERSON, W. W.; BROWN, D. T. Cyclic codes for error detection. **Proceedings of the IRE**, IEEE, v. 49, n. 1, p. 228–235, 1961. Citado na página 131.

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 45.

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 18.

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 34 e 37.

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 25.

SD Card Association. **SD Specifications**: Part 1 sdio simplified specification. San Ramon, 2018. Citado 3 vezes nas páginas 71, 72 e 75.

SD Card Association. **SD Specifications**: Part 1 physical layer simplified specification. San Ramon, 2020. Citado 4 vezes nas páginas 70, 71, 73 e 74.

SEGAL, M.; AKELEY, K. **The OpenGL Graphics System**: A specification (version 1.0). Mountain View, 1994. 172 p. Citado na página 41.

SEGAL, M.; AKELEY, K. **The OpenGL Graphics System**: A specification (version 4.6 (core profile)). Beaverton, 2019. 850 p. Citado 2 vezes nas páginas 41 e 42.

SHAKTI. **RISC-V Assembly Language Programmer Manual**. Chenai, 2020. 138 p. Citado 2 vezes nas páginas 36 e 175.

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 41.

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

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

SITRONIX. **ST7789V Datasheet**. Hangzhou, 2014. Citado 4 vezes nas páginas 66, 67, 68 e 69.

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 31 e 38.

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 20 e 40.

TECHNAVIO. Augmented reality and virtual reality market by technology and geography - forecast and analysis 2021-2025. Technavio, p. 120, 2021. Citado na página 21.

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 18.

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 22.

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 29 e 30.

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 5 vezes nas páginas 28, 31, 32, 35 e 62.

WATERMAN, A. et al. The risc-v instruction set manual, volume i: Base user-level isa. EECS Department, UC Berkeley, v. 116, 2011. Citado 3 vezes nas páginas 20, 27 e 167.

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 33.

WU, X. An efficient antialiasing technique. **Acm Siggraph Computer Graphics**, ACM New York, v. 25, n. 4, p. 143–152, 1991. Citado na página 56.

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 22.

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 33.

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 19 e 53.

Apêndices

APÊNDICE A – Quaterniões

Quaterniões são valores complexos compostos de um elemento escalar e três componentes complexos. Foram formulados originalmente por Hamilton (1848), para estudos no plano tridimensional. Considera-se um quaternião como $\boldsymbol{q} = q_s + q_x\hat{\boldsymbol{i}} + q_y\hat{\boldsymbol{j}} + q_z\hat{\boldsymbol{k}}$, onde os termos $\hat{\boldsymbol{i}}$, $\hat{\boldsymbol{j}}$, e $\hat{\boldsymbol{k}}$ representam dimensões externas ao plano real, e ortogonais entre si. O plano dos quaterniões pode ser representado pelo símbolo $\mathbb{H}$, de plano de Hamilton. Os valores de q_s, q_x, q_y, e q_z são reais. De forma simplificada, a definição de quaternião é dado na Equação (18).

$$\boldsymbol{q} = q_s + q_x\hat{\boldsymbol{i}} + q_y\hat{\boldsymbol{j}} + q_z\hat{\boldsymbol{k}},\ q_s,q_x,q_z \in \mathbb{R},\ \boldsymbol{q} \in \mathbb{H},\ 1 \perp \hat{\boldsymbol{i}} \perp \hat{\boldsymbol{j}} \perp \hat{\boldsymbol{k}} \tag{18}$$

Uma soma ou subtração de quaterniões é realizado entre cada eixo de forma independente como visto na Equação (19).

$$\boldsymbol{q} + \boldsymbol{q'} = (q_s + q'_s) + (q_x + q'_x)\hat{\boldsymbol{i}} + (q_y + q'_y)\hat{\boldsymbol{j}} + (q_z + q'_z)\hat{\boldsymbol{k}} \tag{19}$$

Como este elemento é de quatro dimensões, análises gráficas somente são possíveis com o corte transversal entre planos. Contudo, a relação entre os eixos pode ser dado como na Equação (20).

$$\begin{gathered}\hat{\boldsymbol{i}}\hat{\boldsymbol{j}} = -\hat{\boldsymbol{j}}\hat{\boldsymbol{i}} = \hat{\boldsymbol{k}} \quad \hat{\boldsymbol{j}}\hat{\boldsymbol{k}} = -\hat{\boldsymbol{k}}\hat{\boldsymbol{j}} = \hat{\boldsymbol{i}} \quad \hat{\boldsymbol{k}}\hat{\boldsymbol{i}} = -\hat{\boldsymbol{i}}\hat{\boldsymbol{k}} = \hat{\boldsymbol{j}} \\ \hat{\boldsymbol{i}}^2 = \hat{\boldsymbol{j}}^2 = \hat{\boldsymbol{k}}^2 = \hat{\boldsymbol{i}}\hat{\boldsymbol{j}}\hat{\boldsymbol{k}} = -1\end{gathered} \tag{20}$$

O produto entre dois quaterniões pode ser dado pelo produto de Hamilton, e é realizado pela distributiva dos elementos, como presente na Equação (21). Nota-se que a ordem dos produtos é relevante.

$$\begin{aligned}\boldsymbol{qq'} =& (q_s + q_x\hat{\boldsymbol{i}} + q_y\hat{\boldsymbol{j}} + q_z\hat{\boldsymbol{k}})(q'_s + q'_x\hat{\boldsymbol{i}} + q'_y\hat{\boldsymbol{j}} + q'_z\hat{\boldsymbol{k}}) \\ \boldsymbol{qq'} =& q_sq'_s - q_xq'_x - q_yq'_y - q_zq'_z \\ &+(q_sq'_x + q_xq'_s + q_yq'_z - q_zq'_y)\hat{\boldsymbol{i}} \\ &+(q_sq'_y - q_xq'_z + q_yq'_s + q_zq'_x)\hat{\boldsymbol{j}} \\ &+(q_sq'_z + q_xq'_y - q_yq'_x + q_zq'_s)\hat{\boldsymbol{k}}\end{aligned} \tag{21}$$

O conjugado de $\boldsymbol{q}$ é denotado por $\boldsymbol{q}^*$. Dado um $\boldsymbol{q} = q_s + q_x\hat{\boldsymbol{i}} + q_y\hat{\boldsymbol{j}} + q_z\hat{\boldsymbol{k}}$, o seu conjugado é dado por $\boldsymbol{q}^* = q_s - q_x\hat{\boldsymbol{i}} - q_y\hat{\boldsymbol{j}} - q_z\hat{\boldsymbol{k}}$. Aplicando o quaternião e seu conjugado na Equação (21), se obterá um número puramente real. Aplicando a raiz quadrada a este resultante, se obtém o módulo de $\boldsymbol{q}$ como visto na Equação (22). Um quaternião de módulo um é denominado de quaternião unitário.

$$\|\boldsymbol{q}\| = \sqrt{\boldsymbol{qq}^*} = \sqrt{q_s^2 + q_x^2 + q_y^2 + q_z^2} \tag{22}$$

Com o módulo, pode-se obter o recíproco do quaternião, ou seja, o quaternião que resultará em 1 em seu produto. Uma multiplicação por um quaternião unitário pelo seu conjugado resultará em 1, porque ambos possuem módulo unitário, e anularão suas partes complexas. Com isto, o reciproco do quaternião $\boldsymbol{q}$ é dado pelo conjugado dividido pelo quadrado do módulo, e é denotado por $\boldsymbol{q}^{-1}$. Sendo um quaternião $\boldsymbol{q}$, seu recíproco é dado por $\boldsymbol{q}^{-1} = \boldsymbol{q}^*/\|\boldsymbol{q}\|^2$. A comprovação deste valor pode ser visto na Equação (23).

$$\boldsymbol{q}\boldsymbol{q}^{-1} = \boldsymbol{q}\frac{\boldsymbol{q}^*}{\|\boldsymbol{q}\|^2} = \frac{q_s^2 + q_x^2 + q_y^2 + q_z^2}{\|\boldsymbol{q}\|^2} = 1 \tag{23}$$

A.1 Matriz De Rotação

O produto de Hamilton é equivalente ao produto vetorial, de modo que produz um ponto perpendicular ao plano de ambos produtos, como visto na Equação (20). Isto pode ser utilizado para translação de pontos no espaço. Como exemplo, suponhamos um quaternião $\boldsymbol{q} = q_x\hat{\boldsymbol{i}}$, e outro $\boldsymbol{q'} = q'_x\hat{\boldsymbol{i}} + q'_y\hat{\boldsymbol{j}} + q'_z\hat{\boldsymbol{k}}$. O produto entre estes dois é igual a $\boldsymbol{qq'} = -q_xq'_x - q_xq'_z\hat{\boldsymbol{j}} + q_xq'_y\hat{\boldsymbol{k}}$. Percebe-se como o resultante é perpendicular a ambos, e foi rotacionado do eixo (0,1,1,1) para o (-1,0,-1,1). Contudo, isto não é prático para movimentos tridimensionais, porque há a mudança para um quarto eixo, que não possuí utilidade.

Para isto, pode ser aplicado o recíproco $\boldsymbol{q}^{-1} = -q_x\hat{\boldsymbol{i}}/q_x^2$ ao produto, atentando-se à ordem. Com isto se obtém $\boldsymbol{qq'q}^{-1} = q'_x\hat{\boldsymbol{i}} - q'_y\hat{\boldsymbol{j}} - q'_z\hat{\boldsymbol{k}}$. O que mostra um mapeamento de (0,1,1,1) a (0,1,-1,-1), sem um quarto eixo. Percebe-se que isto é equivalente a uma rotação de 180° ao redor do eixo $\hat{\boldsymbol{i}}$, e esta é a utilização de quaterniões em rotações tridimensionais.

De forma genérica, este produto sanduíche é dado na Equação (24), com $\boldsymbol{q}_p = x\hat{\boldsymbol{i}} + y\hat{\boldsymbol{j}} + z\hat{\boldsymbol{k}}$ representando um vetor posição mapeado a um quaternião.

$$\begin{aligned}
\boldsymbol{q}\boldsymbol{q}_p\boldsymbol{q}^{-1} &= (q_s + q_x\hat{\boldsymbol{i}} + q_y\hat{\boldsymbol{j}} + q_z\hat{\boldsymbol{k}})(x\hat{\boldsymbol{i}} + y\hat{\boldsymbol{j}} + z\hat{\boldsymbol{k}})(q_s + q_x\hat{\boldsymbol{i}} + q_y\hat{\boldsymbol{j}} + q_z\hat{\boldsymbol{k}}) \\
\boldsymbol{q}\boldsymbol{q}_p\boldsymbol{q}^{-1} &= ([x(q_s^2 + q_x^2 - q_y^2 - q_z^2) + 2y(q_xq_y - q_sq_z) + 2z(q_xq_z + q_sq_y)]\hat{\boldsymbol{i}} \\
&\quad +[2x(q_xq_y + q_sq_z) + y(q_s^2 - q_x^2 + q_y^2 - q_z^2) + 2z(q_yq_z - q_sq_x)]\hat{\boldsymbol{j}} \\
&\quad +[2x(q_xq_z - q_sq_y) + 2y(q_yq_z + q_sq_x) + z(q_s^2 - q_x^2 - q_y^2 + q_z^2)]\hat{\boldsymbol{k}})/\|\boldsymbol{q}\|^2
\end{aligned} \tag{24}$$

Se percebe como o resultante e a entrada deste produto pode ser representado de forma vetorial. Com isto, uma matriz pode ser utilizada para conversão entre estes valores. Considerando que $\boldsymbol{q_p}$ representa um vetor posição tridimensional de entrada, a saída sera representada pelas componentes resultantes dos eixos $\hat{\boldsymbol{i}}$, $\hat{\boldsymbol{j}}$ e $\hat{\boldsymbol{k}}$. Com isto se obtém a matriz resultante do produto sanduíche na Equação (25).

$$\begin{bmatrix} x_{Res} \\ y_{Res} \\ z_{Res} \end{bmatrix} = \frac{1}{\|\boldsymbol{q}\|^2}\begin{bmatrix} (q_s^2 + q_x^2 - q_y^2 - q_z^2) & 2(q_xq_y - q_sq_z) & 2(q_xq_z + q_sq_y) \\ 2(q_xq_y + q_sq_z) & (q_s^2 - q_x^2 + q_y^2 - q_z^2) & 2(q_yq_z - q_sq_x) \\ 2(q_xq_z - q_sq_y) & 2(q_yq_z + q_sq_x) & (q_s^2 - q_x^2 - q_y^2 + q_z^2) \end{bmatrix}\begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{25}$$

Como notado no exemplo prévio, para ocorrer uma rotação de 180° ao redor de $\hat{\boldsymbol{i}}$, o $\boldsymbol{q}$ deve ser $q_x\hat{\boldsymbol{i}}$. Utilizando $\boldsymbol{q} = q_s + q_x\hat{\boldsymbol{i}}$ percebe-se que a função resultante de $\boldsymbol{q'}$ será de $(q'_x(q_s^2+q_x^2)\hat{\boldsymbol{i}} - [q'_y(q_s^2-q_x^2) - 2q'_zq_xq_s]\hat{\boldsymbol{j}} - [q'_z(q_s^2-q_x^2)+2q'_yq_xq_s]\hat{\boldsymbol{k}})/(q_s^2+q_x^2)$. Tornando $q_s = q_x$, teremos que $\boldsymbol{q'} = q'_x\hat{\boldsymbol{i}} - q'_z\hat{\boldsymbol{j}} + q'_y\hat{\boldsymbol{k}}$, uma rotação de 90° no sentido anti-horário ao redor do eixo $\hat{\boldsymbol{i}}$. Mantendo $q_s^2+q_x^2$ constante, e alterando valores, perceberá a rotação do ponto. Aplicando valores, será verificado que q_s e q_x podem ser substituídos por $cos(\theta/2)$ e $sen(\theta/2)$ respectivamente, sendo " θ " o angulo de rotação.

Esta análise pode se estender aos demais eixos, e se obterá que o quaternião para rotação pode ser representado como dado na Equação (26), em que $\hat{\mathbf{r}}$ representa o vetor unitário do eixo de rotação.

$$\boldsymbol{q}_R = cos(\theta/2) + sen(\theta/2)(r_x\hat{\boldsymbol{i}} + r_y\hat{\boldsymbol{j}} + r_z\hat{\boldsymbol{k}}),\ \hat{\boldsymbol{r}} \in \mathbb{R}^3,\ \|\hat{\boldsymbol{r}}\| = 1 \tag{26}$$

Se $\hat{\mathbf{r}}$ é unitário, $\boldsymbol{q}_R$ será unitário. Se comprova isto pela Equação (27).

$$\|\boldsymbol{q}_R\| = \sqrt{cos^2(\frac{\theta}{2}) + sen^2(\frac{\theta}{2})(r_x^2+r_y^2+r_z^2)} = \sqrt{cos^2(\frac{\theta}{2}) + sen^2(\frac{\theta}{2})\|\hat{\boldsymbol{r}}\|^2} = 1 \tag{27}$$

Substituindo a Equação (26) na Equação (25), se obtém o produto da rotação em função do eixo e do ângulo de rotação, a matriz resultante é dada na Equação (28). Percebe-se como os valores ficam em função diretamente de " θ ".

$$\begin{bmatrix} 1-(1-cos(\theta))(r_y^2+r_z^2) & sen(\theta)(r_xr_y - r_z) + r_xr_y & sen(\theta)(r_xr_z + r_y) + r_xr_z \\ sen(\theta)(r_xr_y + r_z) + r_xr_y & 1-(1-cos(\theta))(r_x^2+r_z^2) & sen(\theta)(r_yr_z - r_x) + r_yr_z \\ sen(\theta)(r_xr_z - r_y) + r_xr_z & sen(\theta)(r_yr_z + r_x) + r_yr_z & 1-(1-cos(\theta))(r_x^2+r_y^2) \end{bmatrix} \tag{28}$$

Pode-se analisar como os componentes da diagonal principal dependem apenas de eixos de rotação perpendiculares. A diagonal principal mostra como o deslocamento da variável em seu próprio eixo é visto, por isto acabam por ser representadas em função de um cosseno. Os demais elementos mostram como os eixos perpendiculares são mapeados, por conta disto, suas componentes dependem do seno do ângulo.

Apesar da matriz da Equação (28) ser a mais intuitiva para análise, não é a mais computacionalmente viável, devido a repetição de senos e cossenos. Por conta disto, a matriz de rotação pode ser obtida pela Equação (25), com algumas simplificações realizadas pelo fato do quaternião de rotação $\boldsymbol{q}_R$ ser unitário. A matriz de rotação resultante a ser utilizada é dada pela Equação (29).

$$\boldsymbol{q}_R = cos(\frac{\theta}{2}) + sen(\frac{\theta}{2})(r_x\hat{\boldsymbol{i}} + r_y\hat{\boldsymbol{j}} + r_z\hat{\boldsymbol{k}}) = q_s + q_x\hat{\boldsymbol{i}} + q_y\hat{\boldsymbol{j}} + q_z\hat{\boldsymbol{k}}, \boldsymbol{q}_R \in \mathbb{H}$$

$$\begin{bmatrix} x_{Rotacionado} \\ y_{Rotacionado} \\ z_{Rotacionado} \end{bmatrix} = \begin{bmatrix} 1-2(q_y^2+q_z^2) & 2(q_xq_y - q_zq_s) & 2(q_xq_z + q_yq_s) \\ 2(q_xq_y + q_zq_s) & 1-2(q_x^2+q_z^2) & 2(q_yq_z - q_xq_s) \\ 2(q_xq_z - q_yq_s) & 2(q_yq_z + q_xq_s) & 1-2(q_x^2+q_y^2) \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{29}$$

APÊNDICE B – Dedução Da Matriz De Projeção

A matriz de projeção é a responsável pelo mapeamento das coordenadas do mundo para o cubo de projeção, que será utilizado na fase gráfica de rasterização dos objetos. É geralmente a última transformação a ser aplicada nos objetos. Os seus componentes podem ser divididos entre as transformações dos eixos "x" e "y", e as do eixo "z".

Uma ilustração de um corte transversão do cone de projeção pode ser visto na Figura 41. A análise será realizada para o eixo "x", contudo pode ser realizada de forma análoga para o "y".

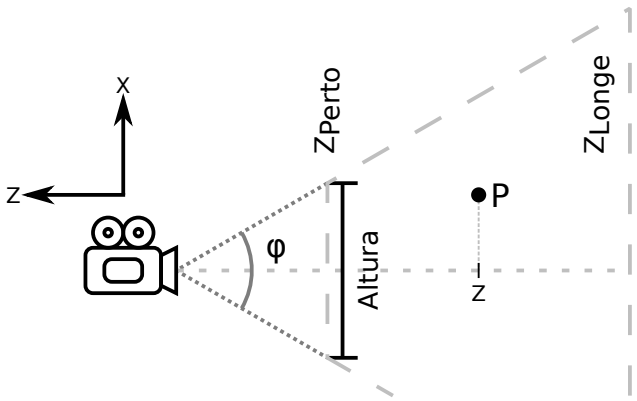


Figura 41 – Corte transversal do cone de projeção
Fonte: Autoria Própria

Traçando um feixe da câmera ao ponto $\vec{P}$ se nota um ponto de intersecção com o plano z_{perto}, este ponto será a projeção do vértice na tela. Para obter esta função de mapeamento, pode se realizar equacionamentos mapeando pontos conhecidos, e depois utilizar disto para criar uma função para um "x" qualquer. Um ponto no topo do cone dado por x_h a uma distância z será mapeado à posição 1 no cubo de projeção, e este valor pode ser obtido pela análise dos triângulos, como dado na Equação (30). O "φ" é divido por dois, porque a análise é feita utilizando triângulos retângulos partindo do eixo central, dividindo a câmera em uma região superiora e inferiora.

$$\frac{x_h}{z} = \frac{\frac{altura}{2}}{-z_{perto}} = -tan(\frac{\varphi}{2})x_h = -z \cdot tan(\frac{\varphi}{2}) \tag{30}$$

De modo equivalente, pode-se obter um segundo ponto para construção da equação da reta. Neste caso se considera um x_0 que se situa no eixo central, em $x = 0$. Este será mapeado ao ponto 0 do cubo de projeção.

Com isto, pode-se obter uma equação de modo $x_{Projetado} = c_0 x_{Mundo} + c_1$. Percebe-se que c_1 será igual a zero e pode ser desconsiderado, e c_0 será igual ao inverso de x_h para se obter o mapeamento de 0 a 1 na projeção. A partir disto se obtém a Equação (31). Esta equação é uma aproximação linear, na realidade efeitos como a curvatura da lente e difusão devido a raios adjacentes existem, contudo, para aplicações simples, é suficientemente preciso.

$$x_{Projetado} = \frac{x_{Mundo}}{x_h} = \frac{1}{-z \cdot tan(\frac{\varphi}{2})} x_{Mundo} \tag{31}$$

Em relação ao eixo "y", a equação obtida será análoga, contudo, deve-se haver uma compensação de compressão ou dilatação para levar em consideração as dimensões da tela, porque o cubo de projeção resultante apresenta as mesmas dimensões em todos os eixos. Geralmente os *displays* possuem alturas maiores do que larguras. É recomendável manter a menor dimensão fixa para reduzir o corte de objetos, então neste caso será comprimido os valores da maior dimensão no mundo para obter uma maior quantidade de pontos dentro do cubo de projeção. Por conta disto será multiplicado um termo de $altura/largura$ pelo termo de x_{Mundo}. Assim a conversão de x_{Mundo} e y_{Mundo} é dado na Equação (32).

$$(x,y)_{Projetado} = \frac{1}{-z \cdot tan(\frac{\varphi}{2})} \left(\frac{altura}{largura}, 1 \right) \cdot (x,y)_{Mundo} \tag{32}$$

A divisão entre os elementos não pode ser inclusa em uma multiplicação de matriz, contudo, o valor de $-z$ será alocado ao valor de "w", que dividirá todo o vetor resultante. Esta multiplicação afetará também o valor de "z", porque certas GPUs são otimizadas para divisão vetorial, e realizam a divisão entre todos os elementos simultaneamente, e dependendo da API, pode até não ser possível essa divisão de modo individual.

A transformação do "z" não utiliza do FOV, somente dos planos de corte z_{Perto} e z_{Longe}. Deve-se atentar ao sinal do eixo, que diminui a medida que se distancia da câmera, então os planos são situados em valores negativos. Neste caso, deseja-se mapear $-z_{Perto}$ a 1, e $-z_{Longe}$ a -1, porém, como haverá uma posterior divisão por $-z$, estes pontos deverão ser mapeados para $-z_{Perto} \rightarrow z_{Perto}$ e $-z_{Longe} \rightarrow -z_{Longe}$. Esta relação pode ser comprovado na Equação (33).

$$\left.\frac{z_{Perto}}{-z}\right|_{z=-z_{Perto}} = 1 \qquad \left.\frac{-z_{Longe}}{-z}\right|_{z=-z_{Longe}} = -1 \tag{33}$$

Supondo um ponto z_{Mundo} a ser mapeado em $z_{Projetado}$, a relação de ganho pode ser obtida dividindo pela distância limites de cada coordenada, como dado na Equação (34).

$$\frac{z_{Projetado}}{z_{Longe} + z_{Perto}} \propto \frac{z_{Mundo}}{z_{Longe} - z_{Perto}} \tag{34}$$

A equação de transformação resultante será da forma $z_{Projetado} = c_2 z_{Mundo} + c_3$. O valor de c_2 pode ser obtido pela Equação (34) como $(z_{Perto} + z_{Longe})/(z_{Longe} - z_{Perto})$, porque define a relação entre a contração ou dilatação entre os sistemas. O valor de c_3 pode ser obtido aplicando os limites dos planos, que possuem valores de conversão conhecidos, como dado na Equação (35).

$$-z_{Perto}\frac{z_{Perto} - z_{Longe}}{z_{Perto} + z_{Longe}} + c_3 = z_{Perto} \qquad -z_{Longe}\frac{z_{Perto} - z_{Longe}}{z_{Perto} + z_{Longe}} + c_3 = -z_{Longe} \tag{35}$$

Isolando c_3 de um dos limites dados na Equação (35), se obtém a relação de deslocamento de $(2 \cdot z_{Longe} z_{Perto})/(z_{Longe} - z_{Perto})$. Com isto se obtém a relação de conversão entre os dois sistemas de coordenadas pela Equação (36).

$$z_{Projetado} = \frac{1}{-z_{Mundo}} \left(\frac{z_{Longe} + z_{Perto}}{z_{Longe} - z_{Perto}} z_{Mundo} + \frac{2 \cdot z_{Longe} z_{Perto}}{z_{Longe} - z_{Perto}} \right) \tag{36}$$

O valor de "w" é mapeado ao valor de $-z_{Mundo}$ para ser utilizado posteriormente para a divisão do vetor resultante, e o seu valor anterior é descartado.

Com as estas funções de mapeamento, a matriz de projeção resultante é dada na Equação (37).

$$\begin{bmatrix} x_{Projetado} \\ y_{Projetado} \\ z_{Projetado} \\ 1 \end{bmatrix} = \frac{1}{-z} \begin{bmatrix} x_{Res} \\ y_{Res} \\ z_{Res} \\ -z \end{bmatrix} = \frac{1}{-z} \begin{bmatrix} \frac{1}{tan(\frac{\varphi}{2})} \frac{altura}{largura} & 0 & 0 & 0 \\ 0 & \frac{1}{tan(\frac{\varphi}{2})} & 0 & 0 \\ 0 & 0 & \frac{z_{Longe}+z_{Perto}}{z_{Longe}-z_{Perto}} & \frac{2 \cdot z_{Longe} z_{Perto}}{z_{Longe}-z_{Perto}} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \tag{37}$$

APÊNDICE C – Cyclic Redundancy Check (CRC)

Durante a transmissão de dados, diversos ruídos providos do ambiente podem afetar a integridade das informações. Em transmissões de dados binários isto ocorre ao alternar os valores do bi-estado de um único ou um conjunto de bits em uma sequência. Para evitar que as informações sejam interpretadas erroneamente no dispositivo receptor, certas técnicas de detecção ou correção são utilizadas, em que se acrescenta à sequência de dados, informações adicionais sobre a disposição binária dos dados.

Uma destas técnicas de detecção de erros, é a de CRC. Esta técnica foi proposta por Peterson e Brown (1961), e se baseia em uma detecção de erro de forma cíclica, utilizando cada bit da informação de forma sequencial para gerar um código de verificação.

De forma matemática, o código de verificação é obtido pela divisão polinomial dos dados, pelo polinômio gerador. O formato do polinômio gerador varia dependendo do padrão de erro que se é buscado. Geralmente é aplicado um polinômio esparso, para detecção de erros de rajada, que ocorrem comumente em transmissões de alta frequência.

O tamanho do código de verificação é a quantidades de bits que devem ser acrescentados ao final da mensagem, e este valor geralmente é sinalizado junto ao método de verificação, como *CRCn*, em que *n* é o tamanho do código de verificação.

Considerando uma operação genérica, em aritmética binária, em que uma mensagem de dados a ser transmitida é representada pela sequência $A_N A_{N-1} ... A_1 A_0$, e um polinômio gerador dado pela equação $\sum_{m=0}^{M} b_m x^m$. Com um código de verificação do tamanho M, a operação de divisão polinomial é realizada como na Equação (38), em que a mensagem é acrescida de M bits com valor 0 aos bits menos significativos, ou *Least Significant Bits* (LSBs) em inglês, e convertida para uma equação polinomial, com cada bit representando um índice específico.

$$\sum_{m=0}^{M-1} C_m x^m = \frac{\sum_{n=0}^{N} A_n x^{n+M} + \sum_{m=0}^{M-1} 0 \cdot x^m}{\sum_{m=0}^{M} b_m x^m} \tag{38}$$

No final, a mensagem é acrescida dos bits de verificação à sua direita, acabando na sequência $A_N A_{N-1} ... A_1 A_0 C_M C_{M-1} ... C1C0$. A verificação pode ser realizada de mesmo modo, caso se tenha conhecimento do polinômio gerador. Contudo, para a verificação, a sequência é dividida como está, com a mensagem e o código de verificação, sem acréscimo de variáveis. Este processo pode ser visto na Equação (39), e caso o valor da divisão retorne 0, a sequência não possui erros, e foi transmitida com êxito, caso contrário, foi detectado um erro, e os dispositivos devem repetir a transferência.

$$0 = \frac{\sum_{n=0}^{N} A_n x^{n+M} + \sum_{m=0}^{M-1} C_m x^m}{\sum_{m=0}^{M} b_m x^m} \tag{39}$$

Apesar do equacionamento matemático envolver divisões longas de polinomiais, o método CRC é preferível em diversas formas de verificação pela sua simplicidade de implementação. As divisões polinomiais em aritmética binária podem ser traduzidas em operações *XOR* sequenciais, o que permite uma implementação direta via hardware.

No cálculo por operações *XOR*, o polinômio é convertido para um número binário, considerando seus índices b_m como os respectivos bits. A partir desta conversão, é realizado uma operação *XOR* com os bits mais significativos da sequência, ou *Most Significant Bits* (MSBs) em inglês. A partir disto, o polinômio convertido é deslocado à direita, e a operação se repete, até que todos os valores tenham sido computados, e o resultante é o código de verificação. Este é o mesmo processo de divisão por subtração, contudo, devido a natureza da aritmética binária, a etapa de subtração produz o mesmo resultado de uma *XOR* bit a bit caso o resultante seja positivo, o que sempre ocorre neste caso. A operação de verificação realiza o mesmo processo, contudo, com os bits de verificação no final da sequência, como visto anteriormente.

Uma forma de materializar isto é por *buffers* de memória, que podem ser implementados diretamente nas portas da transmissão de dados, e permite a criação e verificação dos bits em tempo real, sem gasto computacional adicional. Uma implantação desta forma pode ser visto na Figura 42. Os valores com 1 no polinômio representam os pontos em que a operação *XOR* deve ser realizada, e o MSB indica quando a operação deve ser realizada, por conta disto, ele é realimentado para os pontos do polinômio. Após o final da mensagem, os bits armazenados no *buffer* é o código CRC, ou o resultado da verificação, dependendo da operação realizada.

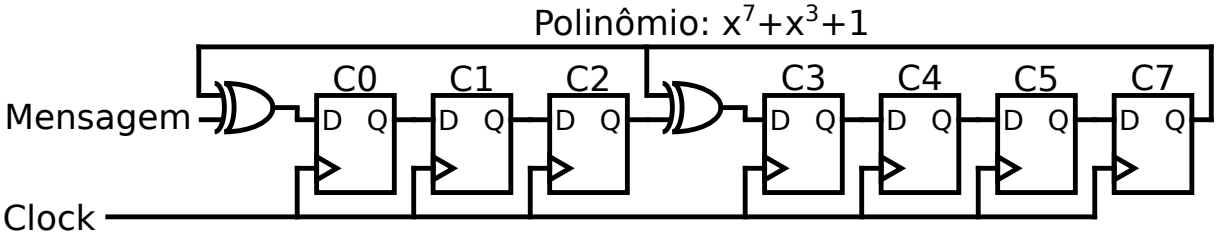


Figura 42 – Implementação da verificação por CRC em hardware
Fonte: Autoria Própria

Uma peculiaridade da verificação cíclica, é que o próximo estado do código de verificação para o próximo bit de uma sequência, depende somente do código de verificação atual, e do novo bit a ser acrescido. Por conta disto, os valores resultantes dos códigos podem ser previamente calculados em tabelas, e realizar as operações em segmentos maiores do que 1 bit. Para aplicação via software do CRC, esta aplicação em software é preferível. Contudo, deve-se atentar que quanto maior o tamanho do segmento, maior o tamanho da tabela. Uma aplicação disto pode ser visto no Algoritmo 10.

Algoritmo 10 Aplicação da verificação CRC por tabela

Input Tamanho do segmento: N; Tamanho do código: M; Sequência: MSG;
Geração da Tabela:
1: **for** $n \in [0,N]$ **do**
2: $Verificador \leftarrow n$;
3: **for** $m \in [0,M]$ **do**
4: **if** $Verificador \wedge 2^M$ **then**
5: $Verificador \leftarrow (Verificador << 1) \veebar Polinômio$;
6: **else**
7: $Verificador \leftarrow Verificador << 1$;
8: $TabelaCRC[n] \leftarrow$**Mod**($Verificador$, M);
Cálculo do CRC:
9: $Verificador \leftarrow 0$;
10: **for** $n \in [0,N]$ **do**
11: $Verificador \leftarrow Verificador << 8 \veebar TabelaCRC[Verificador >> 8 \veebar MSG[n]]$;
12: $Verificador \leftarrow$**Mod**($Verificador$, M);

APÊNDICE D – Algoritmo

crt.S

```
1   .section .text.start,
    "ax", @progbits
2   .globl _start
3   _start:
4    j 1f
5    .word 0xdeadbeef
6   1:
7    csrw mideleg, 0
8    csrw medeleg, 0
9    csrw mie, 0
10   csrw mip, 0
11   la t0, trap_entry
12   csrw mtvec, t0
13
14   li x1, 0
15   li x2, 0
16   li x3, 0
17   li x4, 0
18   li x5, 0
19   li x6, 0
20   li x7, 0
21   li x8, 0
22   li x9, 0
23   li x10,0
24   li x11,0
25   li x12,0
26   li x13,0
27   li x14,0
28   li x15,0
29   li x16,0
30   li x17,0
31   li x18,0
32   li x19,0
33   li x20,0
34   li x21,0
35   li x22,0
36   li x23,0
37   li x24,0
38   li x25,0
39   li x26,0
40   li x27,0
41   li x28,0
42   li x29,0
43   li x30,0
44   li x31,0
45
46   li t0, 0x6000
47   csrs mstatus, t0
48
49   fssr x0
50   fmv.w.x f0, x0
51   fmv.w.x f1, x0
52   fmv.w.x f2, x0
53   fmv.w.x f3, x0
54   fmv.w.x f4, x0
55   fmv.w.x f5, x0
56   fmv.w.x f6, x0
57   fmv.w.x f7, x0
58   fmv.w.x f8, x0
59   fmv.w.x f9, x0
60   fmv.w.x f10,x0
61   fmv.w.x f11,x0
62   fmv.w.x f12,x0
63   fmv.w.x f13,x0
64   fmv.w.x f14,x0
65   fmv.w.x f15,x0
66   fmv.w.x f16,x0
67   fmv.w.x f17,x0
68   fmv.w.x f18,x0
69   fmv.w.x f19,x0
70   fmv.w.x f20,x0
71   fmv.w.x f21,x0
72   fmv.w.x f22,x0
73   fmv.w.x f23,x0
74   fmv.w.x f24,x0
75   fmv.w.x f25,x0
76   fmv.w.x f26,x0
77   fmv.w.x f27,x0
78   fmv.w.x f28,x0
79   fmv.w.x f29,x0
80   fmv.w.x f30,x0
81   fmv.w.x f31,x0
82
83   .option push
84   .option norelax
85   la gp, __global_pointer$
86   .option pop
87   la tp, _end + 63
88   and tp, tp, -64
89   csrr a0, mhartid
90
91   add sp, a0, 1
92   sll sp, sp, 15
93   add sp, sp, tp
94
95   j initBSP
96
97   .globl trap_entry
98   .type trap_entry,
     @function
99   .align 2
100 trap_entry:
101  addi sp, sp, -64*8
102
103  sd x1, 1*8(sp)
104  sd x2, 2*8(sp)
105  sd x3, 3*8(sp)
106  sd x4, 4*8(sp)
107  sd x5, 5*8(sp)
108  sd x6, 6*8(sp)
109  sd x7, 7*8(sp)
110  sd x8, 8*8(sp)
111  sd x9, 9*8(sp)
112  sd x10, 10*8(sp)
113  sd x11, 11*8(sp)
114  sd x12, 12*8(sp)
115  sd x13, 13*8(sp)
116  sd x14, 14*8(sp)
117  sd x15, 15*8(sp)
118  sd x16, 16*8(sp)
119  sd x17, 17*8(sp)
120  sd x18, 18*8(sp)
121  sd x19, 19*8(sp)
122  sd x20, 20*8(sp)
123  sd x21, 21*8(sp)
124  sd x22, 22*8(sp)
125  sd x23, 23*8(sp)
126  sd x24, 24*8(sp)
127  sd x25, 25*8(sp)
128  sd x26, 26*8(sp)
129  sd x27, 27*8(sp)
130  sd x28, 28*8(sp)
131  sd x29, 29*8(sp)
132  sd x30, 30*8(sp)
133  sd x31, 31*8(sp)
134
135  fsw f0, ( 0 + 32)*8(sp)
136  fsw f1, ( 1 + 32)*8(sp)
137  fsw f2, ( 2 + 32)*8(sp)
138  fsw f3, ( 3 + 32)*8(sp)
139  fsw f4, ( 4 + 32)*8(sp)
140  fsw f5, ( 5 + 32)*8(sp)
141  fsw f6, ( 6 + 32)*8(sp)
142  fsw f7, ( 7 + 32)*8(sp)
143  fsw f8, ( 8 + 32)*8(sp)
144  fsw f9, ( 9 + 32)*8(sp)
145  fsw f10,( 10 + 32)*8(sp)
146  fsw f11,( 11 + 32)*8(sp)
147  fsw f12,( 12 + 32)*8(sp)
148  fsw f13,( 13 + 32)*8(sp)
149  fsw f14,( 14 + 32)*8(sp)
150  fsw f15,( 15 + 32)*8(sp)
151  fsw f16,( 16 + 32)*8(sp)
152  fsw f17,( 17 + 32)*8(sp)
153  fsw f18,( 18 + 32)*8(sp)
154  fsw f19,( 19 + 32)*8(sp)
155  fsw f20,( 20 + 32)*8(sp)
156  fsw f21,( 21 + 32)*8(sp)
157  fsw f22,( 22 + 32)*8(sp)
158  fsw f23,( 23 + 32)*8(sp)
159  fsw f24,( 24 + 32)*8(sp)
160  fsw f25,( 25 + 32)*8(sp)
161  fsw f26,( 26 + 32)*8(sp)
162  fsw f27,( 27 + 32)*8(sp)
163  fsw f28,( 28 + 32)*8(sp)
164  fsw f29,( 29 + 32)*8(sp)
165  fsw f30,( 30 + 32)*8(sp)
166  fsw f31,( 31 + 32)*8(sp)
167
168  csrr a0, mcause
169  mv a2, sp
170  add a3, sp, 32*8
171  add a3, sp, 32*8
172  bgez a0, .handle_syscall
173 .handle_irq:
174  jal handleIRQ
175  j .restore
176 .handle_syscall:
177  jal handleSyscall
178 .restore:
179  csrw mepc, a0
180  ld x1, 1*8(sp)
181  ld x2, 2*8(sp)
182  ld x3, 3*8(sp)
183  ld x4, 4*8(sp)
184  ld x5, 5*8(sp)
185  ld x6, 6*8(sp)
186  ld x7, 7*8(sp)
```

```
187  ld x8, 8*8(sp)
188  ld x9, 9*8(sp)
189  ld x10, 10*8(sp)
190  ld x11, 11*8(sp)
191  ld x12, 12*8(sp)
192  ld x13, 13*8(sp)
193  ld x14, 14*8(sp)
194  ld x15, 15*8(sp)
195  ld x16, 16*8(sp)
196  ld x17, 17*8(sp)
197  ld x18, 18*8(sp)
198  ld x19, 19*8(sp)
199  ld x20, 20*8(sp)
190  ld x21, 21*8(sp)
201  ld x22, 22*8(sp)
202  ld x23, 23*8(sp)
203  ld x24, 24*8(sp)
204  ld x25, 25*8(sp)
205  ld x26, 26*8(sp)
206  ld x27, 27*8(sp)
207  ld x28, 28*8(sp)
208  ld x29, 29*8(sp)
209  ld x30, 30*8(sp)
210  ld x31, 31*8(sp)
211
212  flw f0, ( 0 + 32)*8(sp)
213  flw f1, ( 1 + 32)*8(sp)
214  flw f2, ( 2 + 32)*8(sp)
215  flw f3, ( 3 + 32)*8(sp)
216  flw f4, ( 4 + 32)*8(sp)
217  flw f5, ( 5 + 32)*8(sp)
218  flw f6, ( 6 + 32)*8(sp)
219  flw f7, ( 7 + 32)*8(sp)
210  flw f8, ( 8 + 32)*8(sp)
221  flw f9, ( 9 + 32)*8(sp)
222  flw f10,( 10 + 32)*8(sp)
223  flw f11,( 11 + 32)*8(sp)
224  flw f12,( 12 + 32)*8(sp)
225  flw f13,( 13 + 32)*8(sp)
226  flw f14,( 14 + 32)*8(sp)
227  flw f15,( 15 + 32)*8(sp)
228  flw f16,( 16 + 32)*8(sp)
229  flw f17,( 17 + 32)*8(sp)
230  flw f18,( 18 + 32)*8(sp)
231  flw f19,( 19 + 32)*8(sp)
232  flw f20,( 20 + 32)*8(sp)
233  flw f21,( 21 + 32)*8(sp)
234  flw f22,( 22 + 32)*8(sp)
235  flw f23,( 23 + 32)*8(sp)
236  flw f24,( 24 + 32)*8(sp)
237  flw f25,( 25 + 32)*8(sp)
238  flw f26,( 26 + 32)*8(sp)
239  flw f27,( 27 + 32)*8(sp)
240  flw f28,( 28 + 32)*8(sp)
241  flw f29,( 29 + 32)*8(sp)
242  flw f30,( 30 + 32)*8(sp)
243  flw f31,( 31 + 32)*8(sp)
244
245  addi sp, sp, 64*8
246  mret
247
248 .section .data
249 .align 3
250
251 .global g_wake_up
252 g_wake_up:
253  .dword 1
254  .dword 0
255
256 .section ".tdata.begin"
257 .globl _tdata_begin
258 _tdata_begin:
259
260 .section ".tdata.end"
261 .globl _tdata_end
262 _tdata_end:
263
264 .section ".tbss.end"
265 .globl _tbss_end
266 _tbss_end:
```

bsp.S

```
1   .section .text
2   .align 2
3
4   .include "inc/mmap.inc"
5   .include "inc/sysctl.inc"
6   .include "inc/csr.inc"
7   .include "inc/plic.inc"
8
9   .globl initBSP
10  initBSP:
11   addi sp, sp, -16
12   sd ra, 8(sp)
13
14   # Get core Hart id
15   csrr t0, mhartid
16   bne x0, t0, 1f
17
18   # If core 0 initialize
    this section
19   # Zero out BSS
20   jal ra, initBSS
21
22   # Initialize FPIOA
23   jal ra, initFPIOA
24
25   # Reset sysctl source
    status
26   lui t0,
    SYSCTL_BASE_ADDR_LUI
27   addi t0, t0
    SYSCTL_RST_SST
28   ld t1, 0(t0)
29   li t2,
    SYSCTL_RST_CLR_MASK
30   or t1, t1, t2
31   sd t1, 0(t0)
32
33   # Init Platform-Level
    Interrupt Controller
    (PLIC)
34   #jal ra, initPlic
35
36   # Enable Global
    Interrupts
37   jal ra, enable_irq
38
39   # Enable Core 1 Machine-
    mode Software Interrupts
    (MSIP)
40   # in the Coreplex-Local
    INTerrupts (CLINT)
41   lui t0,
    CLINT_BASE_ADDR_LUI
42   li t1, 0x1
43   lwu t2, 0x4(t0)
44   or t2, t2, t1
45   sw t2, 0x4(t0)
46
47   # Free core 1
48   la t0, g_wake_up
49   addi t0, t0, 8
50   li t1, 1
51  2:
52   lr.d t2, (t0)
53   sc.d t2, t1, (t0)
54   bne t2, x0, 2b
55
56   # Send core 0 to main
    function
57   j main
58
59  1:
60   # If core 1 initialize
    this section
61   # Init Platform-Level
    Interrupt Controller
    (PLIC)
62   #jal ra, initPlic
63
64   # Enable Global
64  Interrupts
65   jal ra, enable_irq
66
67   # Wait for core 0 setup
68   la t0, g_wake_up
69  2:
70   ld t1, 0x8(t0)
71   beq t1, x0, 2b
72
73   # Send core 1 to loop
    function
74   j loop1
75
76   ld ra, 8(sp)
77   addi sp, sp, 16
78   ret
79  # end
80
81  initBSS:
82   addi sp, sp, -16
83   sd ra, 8(sp)
84
85   la t0, _bss # BSS memory
    start
86   la t1, _ebss # BSS
    memory end
87
88   # Zero out BSS memory
    region
89  1:
90   sd x0, 0(t0)
91   addi t0, t0, 8
92   blt t0, t1, 1b
93
94   ld ra, 8(sp)
95   addi sp, sp, 16
96   ret
97  # end
98
99   .globl enable_irq
100 enable_irq:
101  addi sp, sp, -16
102  sd ra, 8(sp)
103
104  # Enable external
    machine interrupt
    requests
105  li t0,
    MIE_EXTRN_M_EN_MASK
```

```
106  csrs mie, t0
107
108  # Set status of the
    Machine Interrupt Enable
    status register
109  li t0,
    MSTATUS_MIE_MASK
110  csrs mstatus, t0
111
112  ld ra, 8(sp)
113  addi sp, sp, 16
114  ret
115 # end
116
117 .globl initPlic
118 initPlic:
119  addi sp, sp, -16
120  sd ra, 8(sp)
121
122  # Get core Hart id
123  csrr t0, mhartid
124  li t1, 80
125  mul t1, t1, t0
126
127  # Disable all interrupts
    for this core
128  lui t0,
    PLIC_BASE_ADDR_LUI
129  add t1, t1, t0
130  li t1,
    PLIC_ENABLE_OFFSET
131  add t1, t1, t0
132
133  mv t2, x0
134  li t3, PLIC_NUM_SOURCES
135  srli t3, t3, 3
136 1:
137  add t4, t1, t2
138  #sb x0, 0(t4)
139  #sb x0, 1(t4)
140  #sb x0, 2(t4)
141  #sb x0, 3(t4)
142  sw x0, 0(t4)
143  addi t2, t2, 4
144  blt t2, t3, 1b
145
146  # Set sources priorities
    to zero on first setup
147  la t1, plic_init_flag
148  lbu t1, 0(t1)
149  bne x0, t1, 1f
150
151  lui t0,
    PLIC_BASE_ADDR_LUI
152  li t1,
    PLIC_SRC_PRIORITIES_OFFSET
153  add t1, t1, t0
154
155  mv t2, x0
156  li t3, PLIC_NUM_SOURCES
157  slli t3, t3, 2
158 1:
159  add t4, t1, t2
160  # This must be stored in
    1 byte, in order to avoid
    misalignment
161  sb x0, 0(t4)
162  addi t2, t2, 1
163  blt t2, t3, 1b
164
165  la t1, plic_init_flag
166  li t2, 1
167  sb t2, 0(t1)
168 1:
169  # Set the target
    threshold to zero
170  csrr t0, mhartid
171  li t1, 0x1000
172  mul t1, t1, t0
173  lui t0,
    PLIC_BASE_ADDR_LUI
174  add t0, t0, t1
175  sw x0, 0(t0)
176
177  # Clear the target
    claims
178  lwu t1, 0x4(t0)
179  li t2, 100
180 1:
181  lwu t1, 0x4(t0)
182  bne x0, t1, 2f
183  addi t2, t2, -1
184  bge t2, x0, 1b
185 2:
186
187  # Enable external
    machine interrupt
    requests
188  li t0,
    MIE_EXTRN_M_EN_MASK
189  csrs mie, t0
190
191  ld ra, 8(sp)
192  addi sp, sp, 16
193  ret
194 # end
195
196 .section .data
197 plic_init_flag:
198  .word 0
```

crc.S

```
1   .section .text
2   .align 1
3
4   .globl crc7get
5   # a0 - Data Address
6   # a1 - Length
7   crc7get:
8    addi sp, sp, -16
9    sd ra, 8(sp)
10
11   mv t0, x0
12   mv t1, x0
13   mv t2, a0
14   la t3, table_crc7
15  1:
16   lbu t4, 0(t2)
17
18   slli t5, t1, 1
19   xor t4, t4, t5
20   add t4, t4, t3
21   lbu t1, 0(t4)
22
23   addi t2, t2, 1
24   addi t0, t0, 1
25   bltu t0, a1, 1b
26
27   slli t1, t1, 1
28   ori t1, t1, 0x1
29   add t0, a0, a1
30   sb t1, 0(t0)
31
32   ld ra, 8(sp)
33   addi sp, sp, 16
34   ret
35  #end
36
37  .section .rodata
38  .align 1
39
40  table_crc7:
41   .byte 0x00, 0x09, 0x12,
    0x1b, 0x24, 0x2d, 0x36,
    0x3f, 0x48, 0x41, 0x5a,
    0x53, 0x6c, 0x65, 0x7e,
    0x77
42   .byte 0x19, 0x10, 0x0b,
    0x02, 0x3d, 0x34, 0x2f,
    0x26, 0x51, 0x58, 0x43,
    0x4a, 0x75, 0x7c, 0x67,
    0x6e
43   .byte 0x32, 0x3b, 0x20,
    0x29, 0x16, 0x1f, 0x04,
    0x0d, 0x7a, 0x73, 0x68,
    0x61, 0x5e, 0x57, 0x4c,
    0x45
44   .byte 0x2b, 0x22, 0x39,
    0x30, 0x0f, 0x06, 0x1d,
    0x14, 0x63, 0x6a, 0x71,
    0x78, 0x47, 0x4e, 0x55,
    0x5c
45   .byte 0x64, 0x6d, 0x76,
    0x7f, 0x40, 0x49, 0x52,
    0x5b, 0x2c, 0x25, 0x3e,
    0x37, 0x08, 0x01, 0x1a,
    0x13
46   .byte 0x7d, 0x74, 0x6f,
    0x66, 0x59, 0x50, 0x4b,
    0x42, 0x35, 0x3c, 0x27,
    0x2e, 0x11, 0x18, 0x03,
    0x0a
47   .byte 0x56, 0x5f, 0x44,
    0x4d, 0x72, 0x7b, 0x60,
    0x69, 0x1e, 0x17, 0x0c,
    0x05, 0x3a, 0x33, 0x28,
    0x21
48   .byte 0x4f, 0x46, 0x5d,
    0x54, 0x6b, 0x62, 0x79,
    0x70, 0x07, 0x0e, 0x15,
    0x1c, 0x23, 0x2a, 0x31,
    0x38
49   .byte 0x41, 0x48, 0x53,
    0x5a, 0x65, 0x6c, 0x77,
    0x7e, 0x09, 0x00, 0x1b,
    0x12, 0x2d, 0x24, 0x3f,
    0x36
50   .byte 0x58, 0x51, 0x4a,
    0x43, 0x7c, 0x75, 0x6e,
    0x67, 0x10, 0x19, 0x02,
    0x0b, 0x34, 0x3d, 0x26,
    0x2f
51   .byte 0x73, 0x7a, 0x61,
    0x68, 0x57, 0x5e, 0x45,
    0x4c, 0x3b, 0x32, 0x29,
    0x20, 0x1f, 0x16, 0x0d,
    0x04
52   .byte 0x6a, 0x63, 0x78,
    0x71, 0x4e, 0x47, 0x5c,
    0x55, 0x22, 0x2b, 0x30,
    0x39, 0x06, 0x0f, 0x14,
```

```
    0x1d
53   .byte 0x25, 0x2c, 0x37,
    0x3e, 0x01, 0x08, 0x13,
    0x1a, 0x6d, 0x64, 0x7f,
    0x76, 0x49, 0x40, 0x5b,
    0x52
54   .byte 0x3c, 0x35, 0x2e,
    0x27, 0x18, 0x11, 0x0a,
    0x03, 0x74, 0x7d, 0x66,
    0x6f, 0x50, 0x59, 0x42,
    0x4b
55   .byte 0x17, 0x1e, 0x05,
    0x0c, 0x33, 0x3a, 0x21,
    0x28, 0x5f, 0x56, 0x4d,
    0x44, 0x7b, 0x72, 0x69,
    0x60
56   .byte 0x0e, 0x07, 0x1c,
    0x15, 0x2a, 0x23, 0x38,
    0x31, 0x46, 0x4f, 0x54,
    0x5d, 0x62, 0x6b, 0x70,
    0x79
```

fat.S

```
1   .section .text
2   .align 1
3
4   .include "inc/fat.inc"
5
6   .comm fatBuffer, 514
7   .comm fatTableBuffer, 514
8
9   .globl openFAT
10  openFAT:
11   addi sp, sp, -16
12   sd ra, 8(sp)
13
14   # MBR -------
15   # Get Master Boot Record
16   la a0, fatBuffer
17   li a1, 0
18   jal ra, sdSectorRead
19
20   # Verify if it is a
    valid FAT system
21   lbu t0, 510(a0)
22   lbu t1, 511(a0)
23   slli t0, t0, 8
24   or t0, t0, t1
25   li t1, FAT_BR_SIGNATURE
26   beq t0, t1, 2f
27   li a0, 1
28   j 1f # Not a FAT system
29  2:
30
31   # Get partition values
32   li t0, PARTITION_START
33   li t1, PARTITION_NUMBER
34   li t2,
    PARTITION_ENTRY_SIZE
35   mul t1, t1, t2
36   add t0, t0, t1 #
    Partition entry offset
37   # Partition Start
38   li t1,
    PARTITION_START_OFF
39   add t1, t1, t0
40   add t2, t1, a0
41   lbu t3, 0(t2)
42   lbu t4, 1(t2)
43   slli t4, t4, 8
44   or t3, t3, t4
45   lbu t4, 2(t2)
46   slli t4, t4, 16
47   or t3, t3, t4
48   lbu t4, 3(t2)
49   slli t4, t4, 24
50   or t3, t3, t4
51   la t1, partition_start
52   sw t3, 0(t1)
53   bne x0, t3, 2f
54   li a0, 1
55   j 1f # Partition doesn't
    exist
56  2:
57   # Partition Size
58   li t1,
    PARTITION_SIZE_OFF
59   add t1, t1, t0
60   add t2, t1, a0
61   lbu t3, 0(t2)
62   lbu t4, 1(t2)
63   slli t4, t4, 8
64   or t3, t3, t4
65   lbu t4, 2(t2)
66   slli t4, t4, 16
67   or t3, t3, t4
68   lbu t4, 3(t2)
69   slli t4, t4, 24
70   or t3, t3, t4
71   la t1, partition_size
72   sw t3, 0(t1)
73   bne x0, t3, 2f
74   li a0, 1
75   j 1f # Partition doesn't
    exist
76  2:
77
78   # Verify partition type
79   # [Must be FAT32 LBA]
80   li t1, PARTITION_ID_OFF
81   add t1, t1, t0
82   add t2, t1, a0
83   li t0,
    PARTITION_FAT32_LBA
84   lbu t1, 0(t2)
85   beq t0, t1, 2f
86   li a0, 1
87   j 1f # Type unsuported
88  2:
89
90   # Boot Sector --
91   # Get Partition Boot
    Sector
92   la t0, partition_start
93   lwu a1, 0(t0)
94   la a0, fatBuffer
95   jal ra, sdSectorRead
96
97   # Check Sector Size
98   # [Must be 512]
99   li t0, BOOT_BPS_OFF
100  add t0, t0, a0
101  lbu t1, 1(t0)
102  slli t1, t1, 8
103  lbu t0, 0(t0)
104  or t0, t0, t1
105  li t1, FAT_SECTOR_SIZE
106  beq t0, t1, 2f
107  li a0, 1
108  j 1f # Wrong sector size
109 2:
110
111  # Get Sectors per
    Clusters
112  li t0, BOOT_SPC_OFF
113  add t0, t0, a0
114  lbu s0, 0(t0)
115  la t0, cluster_size
116  sw s0, 0(t0)
117
118  # Get Reserved Area Size
119  li t0, BOOT_RSVD_SIZE
120  add t0, t0, a0
121  lbu t1, 1(t0)
122  slli t1, t1, 8
123  lbu t0, 0(t0)
124  or t0, t0, t1
125
126  # FAT Table Start
127  la t1, partition_start
128  lwu s1, 0(t1)
129  add t0, t0, s1
130  la t1, fat_table_start
131  sw t0, 0(t1)
132
133  # FAT Table Count
134  li t0, BOOT_FAT_CPS
135  add t0, t0, a0
136  lbu s2, 0(t0)
137  la t0, fat_table_count
138  sw s2, 0(t0)
139
140  # FAT Table Size
141  li t0, BOOT_FAT_SIZE
142  add t0, t0, a0
143  lbu t1, 0(t0)
144  lbu t2, 1(t0)
145  slli t2, t2, 8
146  or t1, t1, t2
147  lbu t2, 2(t0)
148  slli t2, t2, 16
149  or t1, t1, t2
150  lbu t2, 3(t0)
151  slli t2, t2, 24
152  or s3, t1, t2
153  la t0, fat_table_size
154  sw s3, 0(t0)
155
156  # Root Start
157  li t0, 2
158  mul t0, t0, s0
159  mul t1, s2, s3
160  add t1, t1, s1
161  add t1, t1, t0
162  la t0, root_start
163  sw t1, 0(t0)
164
165  # FAT Table ---
166  la t0, fat_table_start
167  lwu a1, 0(t0)
168  la a0, fatBuffer
169  jal ra, sdSectorRead
170
171  # Get root folder size
    in clusters
172  mv t0, x0
173  li t2, 4
174  li t3, 0xF8
175  li a1, 0xFFF
176 2:
177  addi t0, t0, 1
```

```
178  addi t1, t0, 1
179  mul t1, t1, t2
180  add t1, t1, a0
181  lbu t1, 0(t1)
182  addi a1, a1, -1
183  bne x0, a1, 3f
184  blt t1, t2, 2b
185 3:
186  la t1, root_folder_size
187  sw t0, 0(t1)
188
189  li a0, 0 # Successful
    FAT setup
190 1:
191  ld ra, 8(sp)
192  addi sp, sp, 16
193  ret
194 #end
195
196 .globl readFile
197 # a0 - File Name
198 # a1 - Result Address
199 readFile:
200  addi sp, sp, -48
201  sd ra, 8(sp)
202  sd a0, 16(sp)
203  sd a1, 24(sp)
204
205  # Find file
206  la a0, fatBuffer
207  la a1, root_start
208  lwu a1, 0(a1)
209  jal ra, sdSectorRead
210  li a1, 0
211  la t0, cluster_size
212  lwu t0, 0(t0)
213  la t1, root_folder_size
214  lwu t1, 0(t1)
215  mul s1, t0, t1
216  li s0, 0
217 1:
218  # Check if is Long File
    Entry
219  add t0, a1, a0
220  lbu t1, 0xB(t0)
221  li t2, 0xF
222  bne t1, t2, 3f # Not a
    Long File Entry
223  # Check if it is the
    first entry
224  lbu t1, 0(t0)
225  and t1, t1, t2
226  li t2, 1
227  bne t1, t2, 3f # Not
    first entry
228  li t1, 0
229  li t2, 1
230 4:
231  ld t3, 16(sp)
232  add t3, t3, t1
233  lbu t3, 0(t3) # File
    name
234  add t4, t0, t2
235  lbu t4, 0(t4) # FAT
    buffer
236
237  bne t3, t4, 3f # Wrong
    file
238  beq t3, x0, 2f # File
    exist
239
240  addi t1, t1, 1
241  addi t2, t2, 2
242
243  # Skip gaps
244  li t3, 11
245  bne t2, t3, 5f
246  addi t2, t2, 3
247 5:
248  li t3, 26
249  bne t2, t3, 5f
250  addi t2, t2, 2
251 5:
252  li t3, 32
253  blt t2, t3, 4b
254  j 2f
255
256  # Check next entry
257  li t3, 512
258  addi a1, a1, 32
259  # Read new sector if
    needed
260  blt a1, t3, 5f
261  addi s0, s0, 1
262  blt s0, s1, 6f
263  li a0, 1
264  j 1f # File doesn't
    exist
265 6:
266  la a1, root_start
267  lwu a1, 0(a1)
268  add a1, a1, s0
269  jal ra, sdSectorRead
270  li a1, 0
271 5:
272  j 1b
273 3:
274  li t0, 512
275  addi a1, a1, 32
276  blt a1, t0, 1b
277  # Read new sector
278  addi s0, s0, 1
279  blt s0, s1, 4f
280  li a0, 1
281  j 1f # File doesn't
    exist
282 4:
283  la a1, root_start
284  lwu a1, 0(a1)
285  add a1, a1, s0
286  jal ra, sdSectorRead
287  li a1, 0
288  j 1b
289 2:
290
291  # Get Short File Entry
    Info
292  addi a1, a1, 32
293  add t0, a1, a0
294  # File first cluster
295  lbu t1, 27(t0)
296  slli t1, t1, 8
297  lbu t2, 26(t0)
298  or t1, t1, t2
299  sw t1, 32(sp)
300  # File size in Bytes
301  lbu t1, 31(t0)
302  slli t1, t1, 24
303  lbu t2, 30(t0)
304  slli t2, t2, 16
305  or t1, t1, t2
306  lbu t2, 29(t0)
307  slli t2, t2, 8
308  or t1, t1, t2
309  lbu t2, 28(t0)
310  or t1, t1, t2
311  sw t1, 36(sp)
312  li t2, FAT_MAX_FILE_KIB
313  blt t1, t2, 2f
314  li a0, 1
315  j 1f # File is too big
316 2:
317  # Get short name
    checksum
318  li t1, 0
319  li t2, 0
320 2:
321  andi t3, t1, 1
322  li t4, 0
323  beq t3, x0, 4f
324  li t4, 0x80
325 4:
326  srli t3, t1, 1
327  add t3, t3, t4
328  add t4, t2, t0
329  lbu t4, 0(t4)
330  add t1, t3, t4
331  andi t1, t1, 0xFF
332
333  li t3, 11
334  addi t2, t2, 1
335  blt t2, t3, 2b
336 3:
337  sw t1, 40(sp)
338
339  # Get FAT Table
340  la a0, fatTableBuffer
341  la a1, fat_table_start
342  lwu a1, 0(a1)
343  jal ra, sdSectorRead
344
345  # Read File
346  li s0, 0
347  li s1, 0
348  la s2, cluster_size
349  lwu s2, 0(s2)
350 2:
351  li s3, 0
352 4:
353  # Read sector data
354  la a0, fatBuffer
355  la a1, root_start
356  lwu a1, 0(a1)
357  lwu t0, 32(sp) # File
    first cluster
358  addi t0, t0, -2
359  mul t0, t0, s2
360  add a1, a1, t0
361  add a1, a1, s3
362  jal ra, sdSectorRead
363
364  # Pass data to result
    address
365  ld t0, 24(sp)
366  li t1, 0
367 5:
368  add t2, t1, a0
369  ld t2, 0(t2) # Load data
    from buffer
370  add t3, s1, t0
```

```
371  sd t2, 0(t3) # Pass data
    to result
372
373  lwu t2, 36(sp) # File
    size
374  addi s1, s1, 8
375  blt t2, s1, 3f #
    Completed read
376
377  li t2, 512
378  addi t1, t1, 8
379  blt t1, t2, 5b
380
381  addi s3, s3, 1
382  blt s3, s2, 4b
383
384  # Get next cluster
385  lwu t0, 32(sp) # File
    first cluster
386  la t1, fatTableBuffer
387  slli t0, t0, 2
388  add t0, t0, t1
389  lbu t1, 3(t0)
390  li t2, 0x0F
391  beq t1, t2, 3f # No more
    clusters allocated
392  slli t1, t1, 24
393  lbu t2, 2(t0)
394  slli t2, t2, 16
395  or t1, t1, t2
396  lbu t2, 1(t0)
397  slli t2, t2, 8
398  or t1, t1, t2
399  lbu t2, 0(t0)
400  or t1, t1, t2
401
402  sw t1, 32(sp) # Next
    cluster
403  j 2b
404 3:
405  li a0, 0 # Successful
     read
406 1:
407  ld ra, 8(sp)
408  addi sp, sp, 48
409  ret
410 #end
411
412 .section .data
413 .align 4
414
415 partition_start:
416  .word 0
417 partition_size:
418  .word 0
419 cluster_size:
420  .word 0
421 fat_table_start:
422  .word 0
423 fat_table_count:
424  .word 0
425 fat_table_size:
426  .word 0
427 root_start:
428  .word 0
429 root_folder_size:
430  .word 0
```

fpiogpio.S

```
1   .section .text
2   .align 2
3
4   .include "inc/mmap.inc"
5   .include "inc/fpioa.inc"
6   .include "inc/gpio.inc"
7   .include "inc/pinmap.inc"
8   .include "inc/sysctl.inc"
9
10  .globl initFPIOA
11  initFPIOA:
12   addi sp, sp, -16
13   sd ra, 8(sp)
14
15   # Enable central clock
    bus
16   lui t0,
    SYSCTL_BASE_ADDR_LUI
17   addi t1, t0,
    SYSCTL_CBUS_EN
18   lwu t2, 0(t1)
19   li t3, CBUS_APB0_MASK
20   or t2, t2, t3
21   sw t2, 0(t1)
22
23   # Enable FPIOA clock
24   addi t1, t0,
    SYSCTL_PERI_EN
25   lwu t2, 0(t1)
26   li t3, PERI_FPIOA_MASK
27   or t2, t2, t3
28   sw t2, 0(t1)
29
30   # Configure multiplex
    tie
31   lui t0,
    FPIOA_BASE_ADDR_LUI
32   addi t1, t0,
    FPIOA_TIE_EN_OFF
33   addi t2, t0,
    FPIOA_TIE_VAL_OFF
34   li t3, FPIOA_TIE0
35   sw t3, 0(t1)
36   sw t3, 0(t2)
37   li t3, FPIOA_TIE1
38   sw t3, 4(t1)
39   sw t3, 4(t2)
40   li t3, FPIOA_TIE2
41   sw t3, 8(t1)
42   sw t3, 8(t2)
43   li t3, FPIOA_TIE3
44   sw t3, 12(t1)
45   sw t3, 12(t2)
46   li t3, FPIOA_TIE4
47   sw t3, 16(t1)
48   sw t3, 16(t2)
49   li t3, FPIOA_TIE5
50   sw t3, 20(t1)
51   sw t3, 20(t2)
52   li t3, FPIOA_TIE6
53   sw t3, 24(t1)
54   sw t3, 24(t2)
55   li t3, FPIOA_TIE7
56   sw t3, 28(t1)
57   sw t3, 28(t2)
58
59   ld ra, 8(sp)
60   addi sp, sp, 16
61   ret
62  # end
63
64  .globl setupFPIOA
65  # a0 - Pin Number
66  # a1 - Configuration
    Value
67  setupFPIOA:
68   addi sp, sp, -16
69   sd ra, 8(sp)
70
71   # Skip if out of range
72   li t0, PINS_MAX
73   bltu t0, a0, 1f
74
75   # Configure FPIOA
76   li t0, FPIOA_BASE_ADDR
77   slli t1, a0, 0x2
78   c.add t0, t1
79   sw a1, 0(t0)
80
81  1:
82   ld ra, 8(sp)
83   addi sp, sp, 16
84   ret
85  # end
86
87  .globl setupGPIO
88  # a0 - GPIO Number
89  # a1 - Direction
90  setupGPIO:
91   addi sp, sp, -16
92   sd ra, 8(sp)
93
94   # Skip if out of range
95   li t0, GPIO_MAX
96   bltu t0, a0, 1f
97
98   li t0, GPIO_BASE_ADDR
99   lw t1, GPIO_DIR_OFF(t0)
100  li t2, 1
101  sll t2, t2, a0
102  not t2, t2
103  and t1, t1, t2
104  mv t2, a1
105  sll t2, t2, a0
106  or t1, t1, t2
107  sw t1, GPIO_DIR_OFF(t0)
108
109 1:
110  ld ra, 8(sp)
111  addi sp, sp, 16
112  ret
113 # end
114
115 .globl setupGPIOHS
116 # a0 - GPIOHS Number
117 # a1 - Direction
118 setupGPIOHS:
119  addi sp, sp, -16
120  sd ra, 8(sp)
121
122 # Skip if out of range
123 li t0, GPIOHS_MAX
124 bltu t0, a0, 1f
125
126  li t0, GPIOHS_BASE_ADDR
127  li t1, 1
128  sll t1, t1, a0
129
130  # Set Input REG
131  lw t2, GPIOHS_IN_EN(t0)
132  not t3, t1
133  and t2, t2, t3
```

```
134  mv t3, a1
135  not t3, t3
136  andi t3, t3, 0x1
137  sll t3, t3, a0
138  or t2, t2, t3
139  sw t2, GPIOHS_IN_EN(t0)
140
141  # Set Output REG
142  lw t2, GPIOHS_OUT_EN(t0)
143  not t3, t1
144  and t2, t2, t3
145  mv t3, a1
146  sll t3, t3, a0
147  or t2, t2, t3
148  sw t2, GPIOHS_OUT_EN(t0)
149
150 1:
151  ld ra, 8(sp)
152  addi sp, sp, 16
153  ret
154 # end
155
156 .globl outputGPIO
157 # a0 - Mask
158 # a1 - Output
159 outputGPIO:
160  addi sp, sp, -16
161  sd ra, 8(sp)
162
163  li t0, GPIO_BASE_ADDR
164  lw t1, GPIO_OUT_OFF(t0)
165  not t2, a0
166  and t1, t1, t2
167  and t2, a0, a1
168  or t1, t1, t2
169  sw t1, GPIO_OUT_OFF(t0)
170
171  ld ra, 8(sp)
172  addi sp, sp, 16
173  ret
174 #end
175
176 .globl outputGPIOHS
177 # a0 - Mask
178 # a1 - Output
179 outputGPIOHS:
180  addi sp, sp, -16
181  sd ra, 8(sp)
182
183  li t0, GPIOHS_BASE_ADDR
184  lw t1,
    GPIOHS_OUT_VAL(t0)
185  not t2, a0
186  and t1, t1, t2
187  and t2, a0, a1
188  or t1, t1, t2
189
190  sw t1,
    GPIOHS_OUT_VAL(t0)
191
192  ld ra, 8(sp)
193  addi sp, sp, 16
194  ret
195 #end
```

irq.S

```
1   .section .text
2   .align 8
3
4   .include "inc/mmap.inc"
5   .include "inc/sysctl.inc"
6   .include "inc/csr.inc"
7   .include "inc/plic.inc"
8
9   .globl handleSyscall
10  handleSyscall:
11   addi sp, sp, -16
12   sd ra, 8(sp)
13
14   # Lock core in a loop
15  1:
16   nop
17   j 1b
18  #end
19
20  .globl handleIRQ
21  handleIRQ:
22   addi sp, sp, -16
23   sd ra, 8(sp)
24
25   # Lock core in a loop
26  1:
27   nop
28   j 1b
29  #end
30
31  .globl handleIrqMExt
32  handleIrqMExt:
33   addi sp, sp, -32
34   sd ra, 8(sp)
35
36   # Verify if interrupt
    bit is set
37   li t0,
    MIE_EXTRN_M_EN_MASK
38   csrr t1, mip
39   bne t0, t1, 1f
40
41   # Get core Hart id
42   csrr t0, mhartid
43
44   # Get primitive
    interrupt enable flag
45   csrr t1, mie
46   sd t1, 16(sp)
47
48   # Get current Interupt
    ReQuest (IRQ) number
49   lui t1,
    PLIC_BASE_ADDR_LUI
50   li t2, 0x1000
51   mul t2, t2, t0
52   add t1, t1, t2
53   lw t2, 0x4(t1)
54
55   # Get primitive IRQ
    threshold
56   lw t3, 0(t1)
57   sd t3, 24(sp)
58
59   # Set new IRQ threshold
60   lui t3,
    PLIC_BASE_ADDR_LUI
61   li t4,
    PLIC_SRC_PRIORITIES_OFFSET
62   add t3, t3, t4
63   slli t2, t2, 2
64   add t3, t3, t2 # PLIC
    IRQ Source Priority
65   lw t3, 0(t3)
66   sw t3, 0(t1)
67
68   # Disable software and
    timer interrupts
69   li t3,
    MIE_SOFTW_M_EN_MASK
70   ori t3, t3,
    MIE_TIMER_M_EN_MASK
71   csrc mie, t3
72
73   # Enable global
    interrupt
74   li t3, MSTATUS_MIE_MASK
75   csrs mstatus, t3
76
77  1:
78   ld ra, 8(sp)
79   addi sp, sp, 32
80   ret
81  #end
```

main.S

```
1   .section .rodata
2   .align 4
3
4   .include "inc/files.inc"
5
6   .section .text
7   .align 4
8
9   .include "inc/colors.inc"
10  .include "inc/fpioa.inc"
11  .include "inc/gpio.inc"
12  .include "inc/pinmap.inc"
13
14  .comm frame_buffer0,
    320*240*4
15  .comm frame_buffer1,
    320*240*4
16  .comm texture, 300*300*2
17  .comm ready_flag, 4
18  .comm fb_select, 4
19
20  .globl main
21  main:
22   addi sp, sp, -32
23   sd ra, 8(sp)
24
25   # Setup PLL
26   jal ra, setupPLL
27
28   # Setup LEDs for
    Debugging purposes
29   li a0, PIN_LED_G
30   li a1, PIN_LED_G_CONFIG
31   jal ra, setupFPIOA
32   li a0, GPIOHS_LED_G
33   li a1, GPIO_OUTPUT
34   jal ra, setupGPIOHS
35   li a0, PIN_LED_B
36   li a1, PIN_LED_B_CONFIG
37   jal ra, setupFPIOA
38   li a0, GPIOHS_LED_B
39   li a1, GPIO_OUTPUT
40   jal ra, setupGPIOHS
41
42   # Initialize TFT display
43   jal ra, tftInitialize
44   la a0, frame_buffer0
45   li a1, COLOR_LOADING
46   jal ra, clearScreen
47   la a0, frame_buffer0
```

```
48   jal ra,
    tftRefreshDisplay
49
50   # Initialize SD card
51   li t0, 0x3 # Timeout
    tries
52   sw t0, 24(sp)
53  1:
54
55   jal ra, sdInitialize
56   beq x0, a0, 1f # SD card
    initialized
57
58   lwu t0, 24(sp)
59   addi t0, t0, -1
60   sw t0, 24(sp)
61   bge t0, x0, 1b
62
63   li a1, COLOR_ERROR_SD
64   bne x0, a0, failure #
    Failed to initialize SD
    card
65  1:
66
67   # Open FAT
68   jal ra, openFAT
69   li a1, COLOR_ERROR_FAT
70   bne x0, a0, failure #
    Failed to open FAT
71
72   # Load the ppm texture
73   la a0, ppm_file
74   la a1, frame_buffer0
75   jal ra, readFile
76   li a1, COLOR_INVALID_PPM
77   bne x0, a0, failure #
    Failed to read file
78
79   la a0, frame_buffer0
80   la a1, texture
81   jal ra, openPPMTex
82   li a1, COLOR_PROBLEM_PPM
83   bne x0, a0, failure #
    File incompatible
84
85   # Load the obj model
86   la a0, obj_file
87   la a1, frame_buffer0
88   jal ra, readFile
89   li a1, COLOR_INVALID_OBJ
90   bne x0, a0, failure #
    Failed to read file
91
92   la a0, frame_buffer0
93   jal ra, openOBJModel
94   li a1, COLOR_PROBLEM_OBJ
95   bne x0, a0, failure #
    File incompatible
96
97   la t0, ready_flag
98   li t1, 1
99   sb t1, 0(t0)
100  sb t1, 1(t0)
101
102 #end
103
104 loop0:
105
106  # TODO: Change flags to
    fence
107  la t0, ready_flag
108 1:
109  lbu t1, 1(t0)
110  beq x0, t1, 1b
111
112  la t0, ready_flag
113  sb x0, 0(t0)
114
115  # DEBUG: Utilized to
    measure framerate
116  # Turn off LED
117  li t0, 1
118  slli a0, t0,
    GPIOHS_LED_G
119  mv a1, a0
120  jal ra, outputGPIOHS
121
122  # Clear frame buffer
123  la t0, fb_select
124  lwu t1, 0(t0)
125  la a0, frame_buffer0
126  beq t1, x0, 1f
127  la a0, frame_buffer1
128 1:
129  #li a1, 0x03
130  li a1, 0xAAAA
131  jal ra, clearScreen
132
133  # Render model
134  la t0, fb_select
135  lwu t1, 0(t0)
136  la a0, frame_buffer0
137  beq t1, x0, 1f
138  la a0, frame_buffer1
139 1:
140  la a1, texture
141  jal ra, renderModel
142
143  # Change frame buffer
144  la t0, fb_select
145  lwu t1, 0(t0)
146  xori t1, t1, 0x1
147  sw t1, 0(t0)
148
149  # DEBUG: Utilized to
    measure framerate
150  # Turn on LED
151  li t0, 1
152  slli a0, t0,
    GPIOHS_LED_G
153  mv a1, x0
154  jal ra, outputGPIOHS
155
156  la t0, ready_flag
157  li t1, 1
158  sb t1, 0(t0)
159
160  j loop0
161 #end
162
163 .globl loop1
164 loop1:
165
166  la t0, ready_flag
167 1:
168  lbu t1, 0(t0)
169  beq x0, t1, 1b
170
171  la t0, ready_flag
172  sb x0, 1(t0)
173
174  # DEBUG: Utilized to
    measure time
175  # Turn off LED
176  li t0, 1
177  slli a0, t0,
    GPIOHS_LED_B
178  mv a1, a0
179  jal ra, outputGPIOHS
180
181  # Update display
182  la t0, fb_select
183  lwu t1, 0(t0)
184  la a0, frame_buffer1
185  beq t1, x0, 1f
186  la a0, frame_buffer0
187 1:
188  jal ra,
    tftRefreshDisplay
189
190  # DEBUG: Utilized to
    measure time
191  # Turn on LED
192  li t0, 1
193  slli a0, t0,
    GPIOHS_LED_B
194  mv a1, x0
195  jal ra, outputGPIOHS
196
197  la t0, ready_flag
198  li t1, 1
199  sb t1, 1(t0)
200
201  j loop1
202 #end
203
204 # a1 - Color
205 failure:
206  # Clear frame buffer
207  la a0, frame_buffer0
208  jal ra, clearScreen
209
210  # Update display
211  la a0, frame_buffer0
212  jal ra,
    tftRefreshDisplay
213
214  # Lock
215 1:
216  nop
217  j 1b
218 #end
```

math.S

```
1   .section .text
2   .align 4
3
4   .globl multiplyModel
5   # a0 - vertex
6   multiplyModel:
7    addi sp, sp, -16
8    sd ra, 8(sp)
9
10   la t0, base_matrix
11
12   flw fs0, 0(a0)
13   flw fs1, 4(a0)
14   flw fs2, 8(a0)
15
16   # W
17   flw ft0, 48(t0)
```

```
18   flw ft1, 52(t0)
19   flw ft2, 56(t0)
20   flw ft3, 60(t0)
21   fmul.s ft0, ft0, fs0
22   fmadd.s ft0, ft1, fs1,
    ft0
23   fmadd.s ft0, ft2, fs2,
    ft0
24   fadd.s fs3, ft0, ft3
25
26   # X
27   flw ft0, 0(t0)
28   flw ft1, 4(t0)
29   flw ft2, 8(t0)
30   flw ft3, 12(t0)
31   fmul.s ft0, ft0, fs0
32   fmadd.s ft0, ft1, fs1,
    ft0
33   fmadd.s ft0, ft2, fs2,
    ft0
34   fmadd.s ft0, ft3, fs3,
    ft0
35   fdiv.s ft0, ft0, fs3
36   fsw ft0, 0(a0)
37
38   # Y
39   flw ft0, 16(t0)
40   flw ft1, 20(t0)
41   flw ft2, 24(t0)
42   flw ft3, 28(t0)
43   fmul.s ft0, ft0, fs0
44   fmadd.s ft0, ft1, fs1,
    ft0
45   fmadd.s ft0, ft2, fs2,
    ft0
46   fmadd.s ft0, ft3, fs3,
    ft0
47   fdiv.s ft0, ft0, fs3
48   fsw ft0, 4(a0)
49
50   # Z
51   flw ft0, 32(t0)
52   flw ft1, 36(t0)
53   flw ft2, 40(t0)
54   flw ft3, 44(t0)
55   fmul.s ft0, ft0, fs0
56   fmadd.s ft0, ft1, fs1,
    ft0
57   fmadd.s ft0, ft2, fs2,
    ft0
58   fmadd.s ft0, ft3, fs3,
    ft0
59   fdiv.s ft0, ft0, fs3
60   #fneg.s ft0, ft0
61   fsw ft0, 8(a0)
62
63   ld ra, 8(sp)
64   addi sp, sp, 16
65   ret
66  #end
67
68  .globl scaleModel
69  scaleModel:
70   addi sp, sp, -16
71   sd ra, 8(sp)
72
73   la t0, base_matrix
74   la t1, scale_vector
75
76   # X
77   flw ft0, 0(t1)
78   flw ft1, 0(t0)
79   fmul.s ft0, ft0, ft1
80   fsw ft0, 0(t0)
81   # Y
82   flw ft0, 4(t1)
83   flw ft1, 4*5(t0)
84   fmul.s ft0, ft0, ft1
85   fsw ft0, 4*5(t0)
86   # Z
87   flw ft0, 8(t1)
88   flw ft1, 4*10(t0)
89   fmul.s ft0, ft0, ft1
90   fsw ft0, 4*10(t0)
91
92   ld ra, 8(sp)
93   addi sp, sp, 16
94   ret
95  #end
96
97  .globl rotateModel
98  rotateModel:
99   addi sp, sp, -16
100  sd ra, 8(sp)
101
102  la t0, rotate_angle
103  flw fa0, 0(t0)
104  jal ra, cosSin
105
106  la t0, base_matrix
107  la t1, rotate_axis
108
109  # Quaternion
110  fmv.s fs0, fa0
111  flw fs1, 0(t1)
112  fmul.s fs1, fs1, fa1
113  flw fs2, 4(t1)
114  fmul.s fs2, fs2, fa1
115  flw fs3, 8(t1)
116  fmul.s fs3, fs3, fa1
117
118  # Rotation Matrix
119  fmul.s fs4, fs1, fs1
    #qx2
120  fmul.s fs5, fs2, fs2
    #qy2
121  fmul.s fs6, fs3, fs3
    #qz2
122  fmul.s fs7, fs1, fs2
    #qx.qy
123  fmul.s fs8, fs1, fs3
    #qx.qz
124  fmul.s fs9, fs1, fs0
    #qx.qs
125  fmul.s fs10, fs2, fs3
    #qy.qz
126  fmul.s fs11, fs2, fs0
    #qy.qs
127  fmul.s ft9, fs3, fs0
    #qz.qs
128  li t1, 1
129  fcvt.s.w ft10, t1
130  li t1, 2
131  fcvt.s.w ft11, t1
132  fadd.s ft0, fs5, fs6
133  fmul.s ft0, ft0, ft11
134  fsub.s ft0, ft10, ft0
    # r00
135  fsub.s ft1, fs7, ft9
136  fmul.s ft1, ft1, ft11
    # r01
137  fadd.s ft2, fs8, fs11
138  fmul.s ft2, ft2, ft11
    # r02
139  fadd.s ft3, fs7, ft9
140  fmul.s ft3, ft3, ft11
    # r10
141  fadd.s ft4, fs4 , fs6
142  fmul.s ft4, ft4, ft11
143  fsub.s ft4, ft10, ft4
    # r11
144  fsub.s ft5, fs10, fs9
145  fmul.s ft5, ft5, ft11
    # r12
146  fsub.s ft6, fs8, fs11
147  fmul.s ft6, ft6, ft11
    # r20
148  fadd.s ft7, fs10, fs9
149  fmul.s ft7, ft7, ft11
    # r21
150  fadd.s ft8, fs4 , fs5
151  fmul.s ft8, ft8, ft11
152  fsub.s ft8, ft10, ft8
    # r22
153
154  # Base Matrix
155  flw fs0, 0(t0) # b00
156  flw fs1, 4(t0) # b01
157  flw fs2, 8(t0) # b02
158  flw fs3, 12(t0) # b03
159  flw fs4, 16(t0) # b10
160  flw fs5, 20(t0) # b11
161  flw fs6, 24(t0) # b12
162  flw fs7, 28(t0) # b13
163  flw fs8, 32(t0) # b20
164  flw fs9, 36(t0) # b21
165  flw fs10, 40(t0) # b22
166  flw fs11, 44(t0) # b23
167
168  # Multiplication
169  fmul.s fa5, ft0, fs0
170  fmadd.s fa5, ft1, fs4,
    fa5
171  fmadd.s fa5, ft2, fs8,
    fa5
172  fsw fa5, 0(t0) # c00
173  fmul.s fa5, ft0, fs1
174  fmadd.s fa5, ft1, fs5,
    fa5
175  fmadd.s fa5, ft2, fs9,
    fa5
176  fsw fa5, 4(t0) # c01
177  fmul.s fa5, ft0, fs2
178  fmadd.s fa5, ft1, fs6,
    fa5
179  fmadd.s fa5, ft2, fs10,
    fa5
180  fsw fa5, 8(t0) # c02
181  fmul.s fa5, ft0, fs3
182  fmadd.s fa5, ft1, fs7,
    fa5
183  fmadd.s fa5, ft2, fs11,
    fa5
184  fsw fa5, 12(t0) # c03
185  fmul.s fa5, ft3, fs0
186  fmadd.s fa5, ft4, fs4,
    fa5
187  fmadd.s fa5, ft5, fs8,
    fa5
188  fsw fa5, 16(t0) # c10
```

```
 fmul.s fa5, ft3, fs1
 fmadd.s fa5, ft4, fs5, fa5
 fmadd.s fa5, ft5, fs9, fa5
 fsw fa5, 20(t0) # c11
 fmul.s fa5, ft3, fs2
 fmadd.s fa5, ft4, fs6, fa5
 fmadd.s fa5, ft5, fs10, fa5
 fsw fa5, 24(t0) # c12
 fmul.s fa5, ft3, fs3
 fmadd.s fa5, ft4, fs7, fa5
 fmadd.s fa5, ft5, fs11, fa5
 fsw fa5, 28(t0) # c13
 fmul.s fa5, ft6, fs0
 fmadd.s fa5, ft7, fs4, fa5
 fmadd.s fa5, ft8, fs8, fa5
 fsw fa5, 32(t0) # c20
 fmul.s fa5, ft6, fs1
 fmadd.s fa5, ft7, fs5, fa5
 fmadd.s fa5, ft8, fs9, fa5
 fsw fa5, 36(t0) # c21
 fmul.s fa5, ft6, fs2
 fmadd.s fa5, ft7, fs6, fa5
 fmadd.s fa5, ft8, fs10, fa5
 fsw fa5, 40(t0) # c22
 fmul.s fa5, ft6, fs3
 fmadd.s fa5, ft7, fs7, fa5
 fmadd.s fa5, ft8, fs11, fa5
 fsw fa5, 44(t0) # c23

 ld ra, 8(sp)
 addi sp, sp, 16
 ret
#end

.globl translateModel
translateModel:
 addi sp, sp, -16
 sd ra, 8(sp)

 la t0, base_matrix
 la t1, translate_vector

 # X
 flw ft0, 0(t1)
 flw ft1, 4*3(t0)
 fadd.s ft0, ft0, ft1
 fsw ft0, 4*3(t0)
 # Y
 flw ft0, 4(t1)
 flw ft1, 4*7(t0)
 fadd.s ft0, ft0, ft1
 fsw ft0, 4*7(t0)
 # Z
 flw ft0, 8(t1)
 flw ft1, 4*11(t0)
 fadd.s ft0, ft0, ft1
 fsw ft0, 4*11(t0)

 ld ra, 8(sp)
 addi sp, sp, 16
 ret
#end

.globl cosSin
# fa0 - Angle [deg]
cosSin:
 addi sp, sp, -16
 sd ra, 8(sp)

 fcvt.wu.s t0, fa0

 srli t0, t0, 2
 li t2, 180
 remu t1, t0, t2

 la t2, cos_table
 slli t1, t1, 2
 add t1, t1, t2
 flw fa0, 0(t1) # cos

 addi t0, t0, 135
 li t1, 180
 remu t0, t0, t1

 slli t0, t0, 2
 add t0, t0, t2
 flw fa1, 0(t0) # sin

 ld ra, 8(sp)
 addi sp, sp, 16
 ret
#end


.globl projectModel
projectModel:
 addi sp, sp, -16
 sd ra, 8(sp)

 la t0, fov_const
 la t1, aspect_ratio
 la t2, zconst

 flw fa1, 0(t0)
 flw ft0, 0(t1)
 fmul.s fa0, fa1, ft0
 flw fa2, 0(t2)
 flw fa3, 4(t2)

 la t0, base_matrix

 flw ft0, 4*0(t0)
 fmul.s ft0, ft0, fa0
 fsw ft0, 4*0(t0) # a00
 flw ft0, 4*1(t0)
 fmul.s ft0, ft0, fa0
 fsw ft0, 4*1(t0) # a01
 flw ft0, 4*2(t0)
 fmul.s ft0, ft0, fa0
 fsw ft0, 4*2(t0) # a02
 flw ft0, 4*3(t0)
 fmul.s ft0, ft0, fa0
 fsw ft0, 4*3(t0) # a03

 flw ft0, 4*4(t0)
 fmul.s ft0, ft0, fa1
 fsw ft0, 4*4(t0) # a10
 flw ft0, 4*5(t0)
 fmul.s ft0, ft0, fa1
 fsw ft0, 4*5(t0) # a11
 flw ft0, 4*6(t0)
 fmul.s ft0, ft0, fa1
 fsw ft0, 4*6(t0) # a12
 flw ft0, 4*7(t0)
 fmul.s ft0, ft0, fa1
 fsw ft0, 4*7(t0) # a13

 flw ft0, 4*8(t0)
 flw ft1, 4*12(t0)
 fneg.s ft2, ft0
 fsw ft2, 4*12(t0) # a30
 fmul.s ft0, ft0, fa2
 fmadd.s ft0, ft1, fa3, ft0
 fsw ft0, 4*8(t0) # a20

 flw ft0, 4*9(t0)
 flw ft1, 4*13(t0)
 fneg.s ft2, ft0
 fsw ft2, 4*13(t0) # a31
 fmul.s ft0, ft0, fa2
 fmadd.s ft0, ft1, fa3, ft0
 fsw ft0, 4*9(t0) # a21

 flw ft0, 4*10(t0)
 flw ft1, 4*14(t0)
 fneg.s ft2, ft0
 fsw ft2, 4*14(t0) # a32
 fmul.s ft0, ft0, fa2
 fmadd.s ft0, ft1, fa3, ft0
 fsw ft0, 4*10(t0) # a22

 flw ft0, 4*11(t0)
 flw ft1, 4*15(t0)
 fneg.s ft2, ft0
 fsw ft2, 4*15(t0) # a33
 fmul.s ft0, ft0, fa2
 fmadd.s ft0, ft1, fa3, ft0
 fsw ft0, 4*11(t0) # a23

 ld ra, 8(sp)
 addi sp, sp, 16
 ret
#end

.section .data
.align 4

scale_vector:
 .float 0.35, 0.35, 0.35

.globl rotate_angle
rotate_angle:
 .float 45.0

.globl rotate_axis
rotate_axis:
 .float 0.0, 1.0, 0.0
 #.float 0.7071068, 0.7071068, 0.0
 #.float 0.9961947, 0.0616284, 0.0616284

```

```
379 .globl translate_vector
380 translate_vector:
381  .float 0.0, -0.1, -3.5
382
383 fov_const:
384  # 1/tan(FOV/2)
385  .float 2.0
386
387 aspect_ratio:
388  # height/width
389  .float 0.75
390
391 zconst:
392  # znear = -1; zfar = -5
393  # (znear+zfar)/(znear
    -zfar)
394  .float 1.5
395  # 2*znear*zfar/(znear-
    zfar)
396  .float -2.5
397
398 .section .rodata
399 .align 4
400
401 .globl cos_table
402 cos_table:
403  .float 1.000000,
    0.999391, 0.997564,
    0.994522, 0.990268,
    0.984808, 0.978148,
    0.970296, 0.961262,
    0.951057
404  .float 0.939693,
    0.927184, 0.913545,
    0.898794, 0.882948,
    0.866025, 0.848048,
    0.829038, 0.809017,
    0.788011
405  .float 0.766044,
    0.743145, 0.719340,
    0.694658, 0.669131,
    0.642788, 0.615662,
    0.587785, 0.559193,
    0.529919
406  .float 0.500000,
    0.469472, 0.438371,
    0.406737, 0.374607,
    0.342020, 0.309017,
    0.275637, 0.241922,
    0.207912
407  .float 0.173648,
    0.139173, 0.104528,
    0.069757, 0.034899,
    -0.000000, -0.034899,
    -0.069756, -0.104529,
    -0.139173
408  .float -0.173648,
    -0.207912, -0.241922,
    -0.275637, -0.309017,
    -0.342020, -0.374607,
    -0.406737, -0.438371,
    -0.469472
409  .float -0.500000,
    -0.529919, -0.559193,
    -0.587785, -0.615661,
    -0.642788, -0.669131,
    -0.694658, -0.719340,
    -0.743145
410  .float -0.766044,
    -0.788011, -0.809017,
    -0.829038, -0.848048,
    -0.866025, -0.882948,
    -0.898794, -0.913545,
    -0.927184
411  .float -0.939693,
    -0.951056, -0.961262,
    -0.970296, -0.978148,
    -0.984808, -0.990268,
    -0.994522, -0.997564,
    -0.999391
412  .float -1.000000,
    -0.999391, -0.997564,
    -0.994522, -0.990268,
    -0.984808, -0.978148,
    -0.970296, -0.961262,
    -0.951057
413  .float -0.939693,
    -0.927184, -0.913545,
    -0.898794, -0.882948,
    -0.866025, -0.848048,
    -0.829038, -0.809017,
    -0.788011
414  .float -0.766044,
    -0.743145, -0.719340,
    -0.694658, -0.669131,
    -0.642788, -0.615662,
    -0.587785, -0.559193,
    -0.529919
415  .float -0.500000,
    -0.469472, -0.438371,
    -0.406737, -0.374607,
    -0.342020, -0.309017,
    -0.275637, -0.241922,
    -0.207912
416  .float -0.173648,
    -0.139173, -0.104528,
    -0.069757, -0.034899,
    0.000000, 0.034899,
    0.069757, 0.104528,
    0.139173
417  .float 0.173648,
    0.207911, 0.241922,
    0.275637, 0.309017,
    0.342020, 0.374607,
    0.406737, 0.438371,
    0.469472
418  .float 0.500000,
    0.529919, 0.559193,
    0.587785, 0.615662,
    0.642788, 0.669131,
    0.694658, 0.719340,
    0.743145
419  .float 0.766044,
    0.788011, 0.809017,
    0.829038, 0.848048,
    0.866025, 0.882948,
    0.898794, 0.913546,
    0.927184
420  .float 0.939693,
    0.951057, 0.961262,
    0.970296, 0.978148,
    0.984808, 0.990268,
    0.994522, 0.997564,
    0.999391
```

model.S

```
1   .section .data
2   .align 8
3
4   # Light position
5   light_pos:
6    .float 0.0, 1.0, -1.0
7
8   light_min:
9    .float 0.2
10
11  light_max:
12   .float 1.0
13
14  .section .text
15  .align 8
16
17  # Count
18  .globl polygon_count
19  .comm polygon_count 4
20  .globl vertice_count
21  .comm vertice_count 4
22  .globl texture_count
23  .comm texture_count 4
24  .globl normal_count
25  .comm normal_count 4
26  # Indexes
27  .globl pos_index
28  .comm pos_index 500*4*4 #
    OBJ_MAX_POLYGONS = 500
29  .globl uv_index
30  .comm uv_index 500*4*4 #
    OBJ_MAX_POLYGONS = 500
31  .globl norm_index
32  .comm norm_index 500*4 #
    OBJ_MAX_POLYGONS = 500
33  # Values
34  .globl pos
35  .comm pos 500*4*3 #
    OBJ_MAX_VERTICES = 500
36  .globl uv
37  .comm uv 500*4*2 #
    OBJ_MAX_UVS = 500
38  .globl norm
39  .comm norm 500*4*3 #
    OBJ_MAX_NORMALS = 500
40  # Vertex buffer
41  .comm vertex 5*16
42  .comm normal_buffer 4*4
43
44  .globl renderModel
45  # a0 - Frame Buffer
46  # a1 - Texture
47  renderModel:
48   addi sp, sp, -40
49   sd ra, 8(sp)
50   sw a0, 16(sp)
51   sw a1, 20(sp)
52
53   # Make Object rotate
54   # Just for testing, may
    be removed
55   la t0, rotate_angle
56   flw fa0, 0(t0)
57   li t1, 4
58   fcvt.s.w ft0, t1
59   li t1, 1
60   fcvt.s.w ft1, t1
61   fdiv.s ft0, ft0, ft1
62   fadd.s fa0, fa0, ft0
63   fsw fa0, 0(t0)
64
65   # Make Object move
66   # Just for testing, may
    be removed
67   la t0, translate_vector
```

```
68   flw fa0, 4(t0)
69   li t1, 1
70   fcvt.s.w ft0, t1
71   li t1, 50
72   fcvt.s.w ft1, t1
73   fdiv.s ft0, ft0, ft1
74   fneg.s ft0, ft0
75   fadd.s fa0, fa0, ft0
76   #fsw fa0, 4(t0)
77
78   la t0, polygon_count
79   lw t0, 0(t0)
80   sw t0, 28(sp)
81   li t1, 0
82   sw t1, 32(sp)
83  1:
84   lw t0, 28(sp)
85   lw t1, 32(sp)
86   bge t1, t0, 2f
87   addi t2, t1, 1
88   sw t2, 32(sp)
89   slli t1, t1, 4
90
91   # Copy vertex
92   la a2, vertex
93   la s0, pos_index
94   add s0, s0, t1
95   la s1, pos
96   la s2, uv_index
97   add s2, s2, t1
98   la s3, uv
99   li t0, 0
100  li t1, 4
101 3:
102  bge t0, t1, 4f
103
104  # Pos
105  slli t2, t0, 2
106  add t2, t2, s0
107  lw t2, 0(t2)
108  li t3, 12
109  mul t2, t2, t3
110  add t2, t2, s1
111
112  # UV
113  slli t3, t0, 2
114  add t3, t3, s2
115  lw t3, 0(t3)
116  li t4, 8
117  mul t3, t3, t4
118  add t3, t3, s3
119
120  # Vertex
121  li t4, 20
122  mul t4, t4, t0
123  add t4, t4, a2
124
125  # Must use lw in order
    to avoid misaligned load
    and store
126  lw t6, 0(t2)
127  sw t6, 0(t4)
128  lw t6, 4(t2)
129  sw t6, 4(t4)
130  lw t6, 8(t2)
131  sw t6, 8(t4)
132  lw t6, 0(t3)
133  sw t6, 12(t4)
134  lw t6, 4(t3)
135  sw t6, 16(t4)
136
137  addi t0, t0, 1
138  j 3b
139 4:
140
141  # Transformation Matrix
142  # Clear to Identity
143  jal ra, cleanBaseMatrix
144  # Transform
145  jal ra, scaleModel
146  jal ra, rotateModel
147
148  # Load Index
149  la t0, norm_index
150  lw t1, 32(sp)
151  addi t1, t1, -1
152  slli t1, t1, 2
153  add t0, t0, t1
154  # Load Normal
155  lw t1, 0(t0)
156  li t0, 12
157  mul t1, t1, t0
158  la t0, norm
159  add t1, t1, t0
160
161  # Must use lw in order
    to avoid misaligned load
    and store
162  la t2, normal_buffer
163  lw t0, 0(t1)
164  sw t0, 0(t2)
165  lw t0, 4(t1)
166  sw t0, 4(t2)
167  lw t0, 8(t1)
168  sw t0, 8(t2)
169  sw x0, 12(t2)
170
171  # Rotate Normal
172  mv a0, t2
173  jal ra, multiplyModel
174  mv t1, a0
175
176  # Light intensity
177  la t0, light_pos
178  # X
179  flw ft0, 0(t0)
180  flw ft1, 0(t1)
181  fmul.s ft3, ft0, ft0
182  fmul.s ft4, ft1, ft1
183  fmul.s ft0, ft0, ft1
184  # Y
185  flw ft1, 4(t0)
186  flw ft2, 4(t1)
187  fmadd.s ft3, ft1, ft1,
    ft3
188  fmadd.s ft4, ft2, ft2,
    ft4
189  fmadd.s ft0, ft1, ft2,
    ft0
190  # Z
191  flw ft1, 8(t0)
192  flw ft2, 8(t1)
193  fmadd.s ft3, ft1, ft1,
    ft3
194  fmadd.s ft4, ft2, ft2,
    ft4
195  fmadd.s ft0, ft1, ft2,
    ft0
196  # Divide by the normal
    and light module
197  fsqrt.s ft3, ft3
198  fsqrt.s ft4, ft4
199  fdiv.s ft0, ft0, ft3
200  fdiv.s ft0, ft0, ft4
201  # Check limits
202  la t0, light_min
203  flw ft1, 0(t0)
204  fmax.s ft0, ft0, ft1 #
    Minimum
205  la t0, light_max
206  flw ft1, 0(t0)
207  fmin.s ft0, ft0, ft1 #
    Maximum
208  # Store
209  fsw ft0, 24(sp)
210
211  # Continue Matrix
    Transform
212  jal ra, translateModel
213
214  # Project
215  jal ra, projectModel
216
217  # Render
218  lwu a0, 16(sp)
219  lwu a1, 20(sp)
220  flw fa0, 24(sp)
221  jal ra, renderSquare
222
223  j 1b
224 2:
225
226  ld ra, 8(sp)
227  addi sp, sp, 40
228  ret
229 #end
230
231 cleanBaseMatrix:
232  addi sp, sp, -16
233  sd ra, 8(sp)
234
235  # Clean Matrix to
    Identity
236  li t0, 1
237  fcvt.s.w ft0, t0
238
239  la t0, base_matrix
240  fsw ft0, 4*0(t0)
241  sw x0, 4*1(t0)
242  sw x0, 4*2(t0)
243  sw x0, 4*3(t0)
244  sw x0, 4*4(t0)
245  fsw ft0, 4*5(t0)
246  sw x0, 4*6(t0)
247  sw x0, 4*7(t0)
248  sw x0, 4*8(t0)
249  sw x0, 4*9(t0)
250  fsw ft0, 4*10(t0)
251  sw x0, 4*11(t0)
252  sw x0, 4*12(t0)
253  sw x0, 4*13(t0)
254  sw x0, 4*14(t0)
255  fsw ft0, 4*15(t0)
256
257  ld ra, 8(sp)
258  addi sp, sp, 16
259  ret
260 #end
```

obj.S

```
1   .section .text
```

```
2   .align 8
3
4   .include "inc/obj.inc"
5   .include "inc/char.inc"
6
7   .globl openOBJModel
8   # a0 - File buffer
9   openOBJModel:
10   addi sp, sp, -16
11   sd ra, 8(sp)
12
13   # Count number of
    vertices, uvs, normals
    polygons
14   mv t0, a0
15   li s0, 0
16   li s1, 0
17   li s2, 0
18   li s3, 0
19  2:
20   jal ra, getNewLine
21   lbu t1, 0(t0)
22   addi t0, t0, 1
23
24   # EOF
25   beq x0, t1, 5f
26
27   # Vertice
28   li t2, CHAR_v
29   bne t1, t2, 4f
30   lbu t2, 0(t0)
31   li t3, CHAR_SPC
32   bne t2, t3, 3f
33   addi s0, s0, 1
34   j 2b
35  3:
36   # UV
37   li t3, CHAR_t
38   bne t2, t3, 3f
39   addi s1, s1, 1
40   j 2b
41  3:
42   # Normal
43   li t3, CHAR_n
44   bne t2, t3, 4f
45   addi s2, s2, 1
46   j 2b
47  4:
48   # Polygon
49   li t2, CHAR_f
50   bne t1, t2, 2b
51   addi s3, s3, 1
52   j 2b
53  5:
54
55   # Verify values
56   # Vertices
57   blt x0, s0, 2f
58   li a0, 1 # Vertice count
    too small
59   j 1f
60  2:
61   li t1, OBJ_MAX_VERTICES
62   bge t1, s0, 2f
63   li a0, 1 # Too many
    vertices
64   j 1f
65  2:
66   # UVs
67   blt x0, s1, 2f
68   li a0, 1 # UV count too
    small
69   j 1f
70  2:
71   li t1, OBJ_MAX_UVS
72   bge t1, s1, 2f
73   li a0, 1 # Too many UVs
74   j 1f
75  2:
76   # Normals
77   blt x0, s2, 2f
78   li a0, 1 # Normal count
    too small
79   j 1f
80  2:
81   li t1, OBJ_MAX_NORMALS
82   bge t1, s2, 2f
83   li a0, 1 # Too many
    Normals
84   j 1f
85  2:
86   # Polygon
87   blt x0, s3, 2f
88   li a0, 1 # Polygon count
    too small
89   j 1f
90  2:
91   li t1, OBJ_MAX_POLYGONS
92   bge t1, s3, 2f
93   li a0, 1 # Too many
    polygons
94   j 1f
95  2:
96   # Store vertice count
97   la t1, vertice_count
98   sw s0, 0(t1)
99   # Store texture count
100  la t1, texture_count
101  sw s1, 0(t1)
102  # Store normal count
103  la t1, normal_count
104  sw s2, 0(t1)
105  # Store polygon count
106  la t1, polygon_count
107  sw s3, 0(t1)
108
109  # Load polygons to
    memory
110  la a1, pos_index
111  la a2, uv_index
112  la a3, norm_index
113  jal ra, loadPolygons
114
115  # Load vertices to
    memory
116  la a1, pos
117  jal ra, loadVertices
118
119  # Load UVs to memory
120  la a1, uv
121  jal ra, loadUVs
122
123  # Load normals to memory
124  la a1, norm
125  jal ra, loadNormals
126
127  li a0, 0 # Setup
    successful
128 1:
129  ld ra, 8(sp)
130  addi sp, sp, 16
131  ret
132 #end
133
134 # a0 - File buffer
135 # a1 - Vertice Index
    Buffer
136 # a2 - UV Index Buffer
137 # a3 - Normal Index
    Buffer
138 loadPolygons:
139  addi sp, sp, -16
140  sd ra, 8(sp)
141
142  mv t0, a0
143  li s0, 0
144 1:
145  jal ra, getNewLine
146  lbu t1, 0(t0)
147  addi t0, t0, 1
148
149  # EOF
150  beq x0, t1, 2f
151
152  # Polygon
153  li t2, CHAR_f
154  bne t1, t2, 1b
155
156  # Vertice 0
157  jal ra, ignoreSpaces
158  li t4, CHAR_FSLASH
159  jal ra,
    getNumberFromFace
160  # Store vertice 0
161  mv t2, s0
162  slli t2, t2, 4
163  add t2, t2, a1
164  sw t1, 0(t2)
165  # Texture 0
166  jal ra,
    getNumberFromFace
167  # Store texture 0
168  mv t2, s0
169  slli t2, t2, 4
170  add t2, t2, a2
171  sw t1, 0(t2)
172  # Normal
173  li t4, CHAR_SPC
174  jal ra,
    getNumberFromFace
175  # Store normal
176  mv t2, s0
177  slli t2, t2, 2
178  add t2, t2, a3
179  sw t1, 0(t2)
180
181  # Vertice 1
182  jal ra, ignoreSpaces
183  li t4, CHAR_FSLASH
184  jal ra,
    getNumberFromFace
185  # Store vertice 1
186  mv t2, s0
187  slli t2, t2, 4
188  add t2, t2, a1
189  sw t1, 4(t2)
190  # Texture 1
191  jal ra,
    getNumberFromFace
192  # Store texture 1
```

```
193  mv t2, s0
194  slli t2, t2, 4
195  add t2, t2, a2
196  sw t1, 4(t2)
197  # Normal
198  li t4, CHAR_SPC
199  jal ra,
    getNumberFromFace
200
201  # Vertice 2
202  jal ra, ignoreSpaces
203  li t4, CHAR_FSLASH
204  jal ra,
    getNumberFromFace
205  # Store vertice 2
206  mv t2, s0
207  slli t2, t2, 4
208  add t2, t2, a1
209  sw t1, 8(t2)
210  # Texture 2
211  jal ra,
    getNumberFromFace
212  # Store texture 2
213  mv t2, s0
214  slli t2, t2, 4
215  add t2, t2, a2
216  sw t1, 8(t2)
217  # Normal
218  li t4, CHAR_SPC
219  jal ra,
    getNumberFromFace
220
221  # Vertify if it is a
    triangle
222  mv t1, t0
223  li t2, CHAR_SPC
224 3:
225  lbu t3, 0(t1)
226  addi t1, t1, 1
227  beq t2, t3, 3b
228
229  # Verify if it is a
    number
230  li t2, CHAR_0
231  blt t3, t2, 2f
232  li t2, CHAR_9
233  blt t2, t3, 2f
234  j 3f
235 2:
236  # If it is a triangle,
    copy the third vertex to
    the fourth
237  mv t2, s0
238  slli t2, t2, 4
239  # Vertice 2 -> Vertice 3
240  add t3, t2, a1
241  lwu t1, 8(t3)
242  sw t1, 12(t3)
243  # Texture 2 -> Texture 3
244  add t3, t2, a2
245  lwu t1, 8(t3)
246  sw t1, 12(t3)
247  j 4f
248 3:
249
250  # Vertice 3
251  jal ra, ignoreSpaces
252  li t4, CHAR_FSLASH
253  jal ra,
    getNumberFromFace
254  # Store vertice 3
255  mv t2, s0
256  slli t2, t2, 4
257  add t2, t2, a1
258  sw t1, 12(t2)
259  # Texture 3
260  jal ra,
    getNumberFromFace
261  # Store texture 3
262  mv t2, s0
263  slli t2, t2, 4
264  add t2, t2, a2
265  sw t1, 12(t2)
266  # Normal
267  li t4, CHAR_SPC
268  jal ra,
    getNumberFromFace
269
270 4:
271  addi s0, s0, 1
272  la t1, polygon_count
273  lwu t1, 0(t1)
274  blt s0, t1, 1b
275 2:
276
277  ld ra, 8(sp)
278  addi sp, sp, 16
279  ret
280 #end
281
282 # a0 - File buffer
283 # a1 - Vertice Buffer
284 loadVertices:
285  addi sp, sp, -16
286  sd ra, 8(sp)
287
288  mv t0, a0
289  li s0, 0
290 1:
291  jal ra, getNewLine
292  lbu t1, 0(t0)
293  addi t0, t0, 1
294
295  # EOF
296  beq x0, t1, 2f
297
298  # Vertice
299  li t2, CHAR_v
300  bne t1, t2, 1b
301  li t2, CHAR_SPC
302  lbu t1, 0(t0)
303  bne t1, t2, 1b
304
305  # Vertice x
306  jal ra, ignoreSpaces
307  jal ra,
    getFloatFromValue
308  # Store vertice x
309  mv t2, s0
310  li t3, 12
311  mul t2, t2, t3
312  add t2, t2, a1
313  fsw fa0, 0(t2)
314
315  # Vertice y
316  jal ra, ignoreSpaces
317  jal ra,
    getFloatFromValue
318  # Store vertice y
319  mv t2, s0
320  li t3, 12
321  mul t2, t2, t3
322  add t2, t2, a1
323  fsw fa0, 4(t2)
324
325  # Vertice z
326  jal ra, ignoreSpaces
327  jal ra,
    getFloatFromValue
328  # Store vertice z
329  mv t2, s0
330  li t3, 12
331  mul t2, t2, t3
332  add t2, t2, a1
333  fsw fa0, 8(t2)
334
335  addi s0, s0, 1
336  la t1, vertice_count
337  lwu t1, 0(t1)
338  blt s0, t1, 1b
339 2:
340
341  ld ra, 8(sp)
342  addi sp, sp, 16
343  ret
344 #end
345
346 # a0 - File buffer
347 # a1 - UV Buffer
348 loadUVs:
349  addi sp, sp, -16
350  sd ra, 8(sp)
351
352  mv t0, a0
353  li s0, 0
354 1:
355  jal ra, getNewLine
356  lbu t1, 0(t0)
357  addi t0, t0, 1
358
359  # EOF
360  beq x0, t1, 2f
361
362  # UV
363  li t2, CHAR_v
364  bne t1, t2, 1b
365  li t2, CHAR_t
366  lbu t1, 0(t0)
367  bne t1, t2, 1b
368  addi t0, t0, 1
369
370  # UV x
371  jal ra, ignoreSpaces
372  jal ra,
    getFloatFromValue
373  # Store UV x
374  mv t2, s0
375  slli t2, t2, 3
376  add t2, t2, a1
377  fsw fa0, 0(t2)
378
379  # UV y
380  jal ra, ignoreSpaces
381  jal ra,
    getFloatFromValue
382  # Store UV y
383  mv t2, s0
384  slli t2, t2, 3
385  add t2, t2, a1
386  fsw fa0, 4(t2)
```

```
387
388  addi s0, s0, 1
389  la t1, texture_count
390  lwu t1, 0(t1)
391  blt s0, t1, 1b
392 2:
393
394  ld ra, 8(sp)
395  addi sp, sp, 16
396  ret
397 #end
398
399 # a0 - File buffer
400 # a1 - Normal Buffer
401 loadNormals:
402  addi sp, sp, -16
403  sd ra, 8(sp)
404
405  mv t0, a0
406  li s0, 0
407 1:
408  jal ra, getNewLine
409  lbu t1, 0(t0)
410  addi t0, t0, 1
411
412  # EOF
413  beq x0, t1, 2f
414
415  # UV
416  li t2, CHAR_v
417  bne t1, t2, 1b
418  li t2, CHAR_n
419  lbu t1, 0(t0)
420  bne t1, t2, 1b
421  addi t0, t0, 1
422
423  # Normal x
424  jal ra, ignoreSpaces
425  jal ra,
    getFloatFromValue
426  # Store Normal x
427  mv t2, s0
428  li t3, 12
429  mul t2, t2, t3
430  add t2, t2, a1
431  fsw fa0, 0(t2)
432
433  # Normal y
434  jal ra, ignoreSpaces
435  jal ra,
    getFloatFromValue
436  # Store Normal y
437  mv t2, s0
438  li t3, 12
439  mul t2, t2, t3
440  add t2, t2, a1
441  fsw fa0, 4(t2)
442
443  # Normal z
444  jal ra, ignoreSpaces
445  jal ra,
    getFloatFromValue
446  # Store Normal z
447  mv t2, s0
448  li t3, 12
449  mul t2, t2, t3
450  add t2, t2, a1
451  fsw fa0, 8(t2)
452
453  addi s0, s0, 1
454  la t1, texture_count
455  lwu t1, 0(t1)
456  blt s0, t1, 1b
457 2:
458
459  ld ra, 8(sp)
460  addi sp, sp, 16
461  ret
462 #end
463
464 # t0 - Temp File buffer
465 getNewLine:
466  addi sp, sp, -16
467  sd ra, 8(sp)
468
469 1:
470  lbu t1, 0(t0)
471  addi t0, t0, 1
472
473  # New line
474  li t2, CHAR_NL
475  bne t1, t2, 2f
476  j 1b
477 2:
478  # EOF
479  beq t1, x0, 2f
480
481  # Useful line
482  li t2, CHAR_v
483  beq t1, t2, 2f
484  li t2, CHAR_f
485  beq t1, t2, 2f
486
487  # Unused line
488 3:
489  lbu t1, 0(t0)
490  addi t0, t0, 1
491
492  # Line ended
493  li t2, CHAR_NL
494  bne t1, t2, 3b
495  addi t0, t0, -1
496  j 1b
497 2:
498  addi t0, t0, -1
499
500  ld ra, 8(sp)
501  addi sp, sp, 16
502  ret
503 #end
504
505 # t0 - Temp File buffer
506 getFloatFromValue:
507  addi sp, sp, -16
508  sd ra, 8(sp)
509
510  # Check sign
511  li s1, 0
512  li t4, CHAR_MINUS
513  lbu t1, 0(t0)
514  bne t1, t4, 2f
515  li s1, 1 # s1 : 0 = + ;
    1 = -
516  addi t0, t0, 1
517 2:
518
519  # Get number left of the
    dot
520  li t4, CHAR_SPC
521  li t5, CHAR_NL
522  li t6, CHAR_DOT
523  li t3, 0
524 3:
525  lbu t1, 0(t0)
526  addi t0, t0, 1
527  beq t1, t4, 1f
528  beq t1, t5, 1f
529  beq t1, t6, 2f
530
531  li t2, CHAR_0
532  sub t1, t1, t2
533  li t2, 10
534  mul t3, t3, t2
535  add t3, t3, t1
536  j 3b
537
538 2:
539  fcvt.s.w ft0, t3
540
541  # Get number right of
    the dot
542  li t3, 0
543  li s2, 1
544 3:
545  lbu t1, 0(t0)
546  addi t0, t0, 1
547  beq t1, t4, 2f
548  beq t1, t5, 2f
549
550  li t2, CHAR_0
551  sub t1, t1, t2
552  li t2, 10
553  mul t3, t3, t2
554  add t3, t3, t1
555  mul s2, s2, t2
556  j 3b
557 2:
558  fcvt.s.w ft1, t3
559  fcvt.s.w ft2, s2
560  fdiv.s ft1, ft1, ft2
561  fadd.s ft0, ft0, ft1
562
563 1:
564  # Change sign
565  beq x0, s1, 1f
566  fneg.s ft0, ft0
567 1:
568  fmv.s fa0, ft0
569
570  ld ra, 8(sp)
571  addi sp, sp, 16
572  ret
573 #end
574
575 # t0 - Temp File buffer
576 # t4 - Stop Character
577 getNumberFromFace:
578  addi sp, sp, -16
579  sd ra, 8(sp)
580
581  li t5, CHAR_NL
582  li t3, 0
583 3:
584  lbu t1, 0(t0)
585  addi t0, t0, 1
586  beq t1, t4, 2f
587  beq t1, t5, 2f
588
589  li t2, CHAR_0
590  sub t1, t1, t2
```

```
591  li t2, 10
592  mul t3, t3, t2
593  add t3, t3, t1
594  j 3b
595 2:
596  mv t1, t3
597  addi t1, t1, -1
598
599  ld ra, 8(sp)
600  addi sp, sp, 16
601  ret
602 #end
603
604 # t0 - Temp File buffer
605 ignoreSpaces:
606  addi sp, sp, -16
607  sd ra, 8(sp)
608
609  li t2, CHAR_SPC
610 1:
611  # Ignore spaces
612  lbu t1, 0(t0)
613  addi t0, t0, 1
614  beq t1, t2, 1b
615
616  addi t0, t0, -1
617
618  ld ra, 8(sp)
619  addi sp, sp, 16
620  ret
621 #end
```

ppm.S

```
1   .section .text
2   .align 1
3
4   .include "inc/ppm.inc"
5   .include "inc/char.inc"
6
7   .globl openPPMTex
8   # a0 - File buffer
9   # a1 - Texture buffer
10  openPPMTex:
11   addi sp, sp, -16
12   sd ra, 8(sp)
13
14   # Check if it is ppm
    file
15   mv t0, a0
16   jal ra, getNewLine
17
18   lbu t1, 0(t0)
19   li t2, CHAR_P
20   beq t1, t2, 2f
21   li a0, 1 # Not a ppm
    file
22   j 1f
23  2:
24   # Get ppm magic number
25   addi t0, t0, 1
26   lbu t1, 0(t0)
27   li t2, CHAR_0
28   sub t1, t1, t2
29   li t2, 6
30   beq t1, t2, 2f
31   li a0, 1 # Unsuported
    ppm
32   j 1f
33  2:
34   # Get image dimensions
35   addi t0, t0, 1
36   jal ra, getNewLine
37   # Width
38   li t3, 0
39   lbu t1, 0(t0)
40   li t2, CHAR_0
41   sub t3, t1, t2
42  2:
43   addi t0, t0, 1
44   lbu t1, 0(t0)
45   li t2, CHAR_SPC
46   beq t1, t2, 3f
47   li t2, 10
48   mul t3, t3, t2
49   li t2, CHAR_0
50   sub t1, t1, t2
51   add t3, t3, t1
52   j 2b
53  3:
54   # Verify dimensions
55   blt x0, t3, 2f
56   li a0, 1 # Invalid width
57   j 1f
58  2:
59   li t2, PPM_MAX_WIDTH
60   bge t2, t3, 2f
61   li a0, 1 # Invalid width
62   j 1f
63  2:
64   la t2, texture_size
65   sh t3, 0(t2)
66
67   # Height
68   addi t0, t0, 1
69   li t3, 0
70   lbu t1, 0(t0)
71   li t2, CHAR_0
72   sub t3, t1, t2
73  2:
74   addi t0, t0, 1
75   lbu t1, 0(t0)
76   li t2, CHAR_NL
77   beq t1, t2, 3f
78   li t2, CHAR_SPC
79   beq t1, t2, 3f
80   li t2, CHAR_HASH
81   beq t1, t2, 3f
82   li t2, 10
83   mul t3, t3, t2
84   li t2, CHAR_0
85   sub t1, t1, t2
86   add t3, t3, t1
87   j 2b
88  3:
89   # Verify dimensions
90   blt x0, t3, 2f
91   li a0, 1 # Invalid
    height
92   j 1f
93  2:
94   li t2, PPM_MAX_HEIGHT
95   bge t2, t3, 2f
96   li a0, 1 # Invalid
    height
97   j 1f
98  2:
99   la t2, texture_size
100  sh t3, 2(t2)
101
102  # Get maximum color
    value
103  jal ra, getNewLine
104  li t3, 0
105  lbu t1, 0(t0)
106  li t2, CHAR_0
107  sub t3, t1, t2
108 2:
109  addi t0, t0, 1
110  lbu t1, 0(t0)
111  li t2, CHAR_NL
112  beq t1, t2, 3f
113  li t2, CHAR_SPC
114  beq t1, t2, 3f
115  li t2, CHAR_HASH
116  beq t1, t2, 3f
117  li t2, 10
118  mul t3, t3, t2
119  li t2, CHAR_0
120  sub t1, t1, t2
121  add t3, t3, t1
122  j 2b
123 3:
124  # Verify value
125  li t2, PPM_COLOR_DEPTH
126  beq t2, t3, 2f
127  li a0, 1 # Unsuported
    value
128  j 1f
129 2:
130
131  # Get image data
132  jal ra, getNewLine
133  la t1, texture_size
134  lhu t2, 0(t1)
135  lhu t3, 2(t1)
136  mul s0, t2, t3
137  li s1, 0
138  li s2, 0xF8
139  li s3, 0xFC
140 2:
141  # Red
142  lbu t1, 0(t0)
143  and t2, t1, s2
144  slli t2, t2, 8
145  # Green
146  addi t0, t0, 1
147  lbu t1, 0(t0)
148  and t3, t1, s3
149  slli t3, t3, 3
150  or t2, t2, t3
151  # Blue
152  addi t0, t0, 1
153  lbu t1, 0(t0)
154  srli t3, t1, 3
155  or t1, t2, t3
156
157  # Store texture
158  slli t2, s1, 1
159  add t2, t2, a1
160  sh t1, 0(t2)
161
162  addi t0, t0, 1
163  addi s1, s1, 1
164  blt s1, s0, 2b
165
166  li a0, 0 # Setup
    successful
167 1:
168  ld ra, 8(sp)
169  addi sp, sp, 16
170  ret
```

```
171 #end
172
173 # t0 - Temp File buffer
174 getNewLine:
175  addi sp, sp, -16
176  sd ra, 8(sp)
177
178 1:
179  lbu t1, 0(t0)
180  addi t0, t0, 1
181
182  # Space
183  li t2, CHAR_SPC
184  bne t1, t2, 2f
185  j 1b
186 2:
187  # New line
188  li t2, CHAR_NL
189  bne t1, t2, 2f
190  j 1b
191 2:
192  # Commented line
193  li t2, CHAR_HASH
194  bne t1, t2, 2f
195 3:
196  lbu t1, 0(t0)
197  addi t0, t0, 1
198
199  # Commented ended
200  li t2, CHAR_NL
201  bne t1, t2, 3b
202  addi t0, t0, -1
203  j 1b
204 2:
205  addi t0, t0, -1
206
207  ld ra, 8(sp)
208  addi sp, sp, 16
209  ret
210 #end
```

raster.S

```
1   .section .data
2   .align 8
3
4   .globl texture_size
5   texture_size:
6    # Width, Height
7    .half 4, 4
8
9   .section .text
10  .align 1
11
12  .include "inc/image.inc"
13
14  # Main Buffers
15  .comm depth_grid
    W_WIDTH*W_HEIGHT*2
16
17  # Work Buffers
18  .comm fbx_buffer
    W_WIDTH*W_HEIGHT*4
19  .comm fby_buffer
    W_WIDTH*W_HEIGHT*4
20  .comm depth_buffer
    W_WIDTH*W_HEIGHT*4
21
22  # Float constants
23  .equ QNaN, 0x7FC00000
24  .equ NInf, 0xFF800000
25
26  .globl drawSquare
27  # a0 - Frame Buffer
28  # a1 - Texture
29  # a2 - vertex0
30  # a3 - vertex1
31  # a4 - vertex2
32  # a5 - vertex4
33  # fa0 - Light Intensity
34  drawSquare:
35   addi sp, sp, -128
36   sd ra, 8(sp)
37   sw a0, 12(sp)
38   sw a1, 16(sp)
39   sw a2, 20(sp)
40   sw a3, 24(sp)
41   sw a4, 28(sp)
42   sw a5, 32(sp)
43   fsw fa0, 100(sp)
44
45   # Store Positions on
    Stack
46   lwu t0, 20(sp)
47   lwu t1, 24(sp)
48   lwu t2, 28(sp)
49   lwu t3, 32(sp)
50   # X position
51   li a2, W_WIDTH
52   flw fa1, 0(t0)
53   jal ra, floatToDimension
54   sh a1, 36(sp)
55   flw fa1, 0(t1)
56   jal ra, floatToDimension
57   sh a1, 52(sp)
58   flw fa1, 0(t2)
59   jal ra, floatToDimension
60   sh a1, 68(sp)
61   flw fa1, 0(t3)
62   jal ra, floatToDimension
63   sh a1, 84(sp)
64   # Y position
65   li a2, W_HEIGHT
66   flw fa1, 4(t0)
67   jal ra, floatToDimension
68   sh a1, 38(sp)
69   flw fa1, 4(t1)
70   jal ra, floatToDimension
71   sh a1, 54(sp)
72   flw fa1, 4(t2)
73   jal ra, floatToDimension
74   sh a1, 70(sp)
75   flw fa1, 4(t3)
76   jal ra, floatToDimension
77   sh a1, 86(sp)
78   # Depth Value
79   flw fa1, 8(t0)
80   fsw fa1, 40(sp)
81   flw fa1, 8(t1)
82   fsw fa1, 56(sp)
83   flw fa1, 8(t2)
84   fsw fa1, 72(sp)
85   flw fa1, 8(t3)
86   fsw fa1, 88(sp)
87   # UV X Value
88   flw fa1, 12(t0)
89   fsw fa1, 44(sp)
90   flw fa1, 12(t1)
91   fsw fa1, 60(sp)
92   flw fa1, 12(t2)
93   fsw fa1, 76(sp)
94   flw fa1, 12(t3)
95   fsw fa1, 92(sp)
96   # UV Y Value
97   flw fa1, 16(t0)
98   fsw fa1, 48(sp)
99   flw fa1, 16(t1)
100  fsw fa1, 64(sp)
101  flw fa1, 16(t2)
102  fsw fa1, 80(sp)
103  flw fa1, 16(t3)
104  fsw fa1, 96(sp)
105
106  # Verify if all points
    are hidden
107  li t0, 2
108  li t1, 1
109  fcvt.s.w ft0, t0
110  fcvt.s.w ft1, t1
111  fdiv.s ft0, ft1, ft0 #
    0.5f to avoid rounding
112  # TODO: This number is
    not correct, correct
    should be 0x10000
113  # But somehow the zfar
    plane will be too near
114  li t0, 0x1
115  slli t0, t0, 13
116  fcvt.s.w ft2, t0 # for
    depth buffer
117
118  li s0, 36
119 1:
120  mv t0, s0
121  add t0, t0, sp
122
123  # x0, y0
124  la a0, depth_grid
125  lhu a1, 0(t0)
126  lhu a2, 2(t0)
127  flw fa0, 4(t0)
128  fmul.s fa0, fa0, ft0
129  fsub.s fa0, fa0, ft0
130  fmul.s fa0, fa0, ft2
131  fadd.s fa0, fa0, ft0 #
    Avoid rounding
132  fneg.s fa0, fa0
133  fcvt.w.s t0, fa0 # Depth
    to integer
134
135  li t2, W_WIDTH
136  mul t2, t2, a2
137  add t2, t2, a1
138  slli t2, t2, 2
139  srli t1, t2, 1
140  add t1, t1, a0
141  lhu t1, 0(t1)
142
143  li t2, 0xFFFF
144  bge x0, t0, 4f # Cut if
    too far forward
145  bge t0, t2, 4f # Cut if
    too far behind
146  li t2, 0x1F
147  sub t0, t0, t2
148  bge t0, t1, 4f # Cut if
    too far behind another
    pixel
149
150  j 3f # At least one
    vertice is visible
151 4:
```

```
152  addi s0, s0, 16
153  li t0, 88
154  bge t0, s0, 1b
155  j 2f # All vertices are
    hidden
156 3:
157
158  # Get X max and min
159  lhu a1, 36(sp)
160  lhu a2, 52(sp)
161  lhu a3, 68(sp)
162  lhu a4, 84(sp)
163  jal ra, minmaxi # Min
    Max x
164  addi a2, a2, 1
165  sh a1, 104(sp)
166  sh a2, 106(sp)
167  # Get Y max and min
168  lhu a1, 38(sp)
169  lhu a2, 54(sp)
170  lhu a3, 70(sp)
171  lhu a4, 86(sp)
172  jal ra, minmaxi # Min
    Max y
173  addi a2, a2, 1
174  sh a1, 108(sp)
175  sh a2, 110(sp)
176
177  # Clear buffers
178  la a0, depth_buffer
179  la a1, fbx_buffer
180  la a2, fby_buffer
181  la a3, depth_grid
182  lhu t0, 104(sp) # sx
183  lhu t1, 106(sp) # bx
184  lhu t3, 110(sp) # by
185  li t4, NInf
186  li s1, W_WIDTH
187  lhu s2, 108(sp) # sy
188  fmv.w.x ft0, t4
189 1:
190  bgt t0, t1, 3f
191  mv t2, s2
192  mv t2, s2
193 4:
194  bgt t2, t3, 5f
195
196  mul t4, s1, t2
197  add t4, t4, t0
198  slli t4, t4, 2
199
200  add t5, t4, a0
201  fsw ft0, 0(t5)
202  add t5, t4, a1
203  fsw ft0, 0(t5)
204  add t5, t4, a2
205  fsw ft0, 0(t5)
206
207  addi t2, t2, 1
208  j 4b
209 5:
210  addi t0, t0, 1
211  j 1b
212 3:
213
214  # Draw Lines
215  # v0 & v1
216  lhu a1, 36(sp)
217  lhu a2, 52(sp)
218  lhu a3, 38(sp)
219  lhu a4, 54(sp)
220  la a0, depth_buffer #
    Depth buffer
221  flw fa0, 40(sp)
222  flw fa1, 56(sp)
223  jal ra, drawLine
224  la a0, fbx_buffer #
    Texture UV X buffer
225  flw fa0, 44(sp)
226  flw fa1, 60(sp)
227  jal ra, drawLine
228  la a0, fby_buffer #
    Texture UV Y buffer
229  flw fa0, 48(sp)
230  flw fa1, 64(sp)
231  jal ra, drawLine
232  # v1 & v2
233  lhu a1, 52(sp)
234  lhu a2, 68(sp)
235  lhu a3, 54(sp)
236  lhu a4, 70(sp)
237  la a0, depth_buffer #
    Depth buffer
238  flw fa0, 56(sp)
239  flw fa1, 72(sp)
240  jal ra, drawLine
241  la a0, fbx_buffer #
    Texture UV X buffer
242  flw fa0, 60(sp)
243  flw fa1, 76(sp)
244  jal ra, drawLine
245  la a0, fby_buffer #
    Texture UV Y buffer
246  flw fa0, 64(sp)
247  flw fa1, 80(sp)
248  jal ra, drawLine
249  # v2 & v3
250  lhu a1, 68(sp)
251  lhu a2, 84(sp)
252  lhu a3, 70(sp)
253  lhu a4, 86(sp)
254  la a0, depth_buffer #
    Depth buffer
255  flw fa0, 72(sp)
256  flw fa1, 88(sp)
257  jal ra, drawLine
258  la a0, fbx_buffer #
    Texture UV X buffer
259  flw fa0, 76(sp)
260  flw fa1, 92(sp)
261  jal ra, drawLine
262  la a0, fby_buffer #
    Texture UV Y buffer
263  flw fa0, 80(sp)
264  flw fa1, 96(sp)
265  jal ra, drawLine
266  # v3 & v0
267  lhu a1, 84(sp)
268  lhu a2, 36(sp)
269  lhu a3, 86(sp)
270  lhu a4, 38(sp)
271  la a0, depth_buffer #
    Depth buffer
272  flw fa0, 88(sp)
273  flw fa1, 40(sp)
274  jal ra, drawLine
275  la a0, fbx_buffer #
    Texture UV X buffer
276  flw fa0, 92(sp)
277  flw fa1, 44(sp)
278  jal ra, drawLine
279  la a0, fby_buffer #
    Texture UV Y buffer
280  flw fa0, 96(sp)
281  flw fa1, 48(sp)
282  jal ra, drawLine
283
284  # Fill Polygon
285  #lhu a1, 38(sp)
286  #lhu a2, 54(sp)
287  #lhu a3, 70(sp)
288  #lhu a4, 86(sp)
289  #jal ra, minmaxi # Min
    Max y
290  #mv s0, a1
291  #mv s1, a2
292  #lhu a1, 36(sp)
293  #lhu a2, 52(sp)
294  #lhu a3, 68(sp)
295  #lhu a4, 84(sp)
296  #jal ra, minmaxi # Min
    Max x
297  #mv a3, s0
298  #mv a4, s1
299  #addi a2, a2, 1
300  #addi a4, a4, 1
301
302  lhu a1, 104(sp) # sx
303  lhu a2, 106(sp) # bx
304  lhu a3, 108(sp) # sy
305  lhu a4, 110(sp) # by
306  # Depth buffer
307  la a0, depth_buffer
308  jal ra, fillPolygon
309  # Texture UV X buffer
310  la a0, fbx_buffer
311  jal ra, fillPolygon
312  # Texture UV Y buffer
313  la a0, fby_buffer
314  jal ra, fillPolygon
315
316  # Draw Polygon
317  mv t0, a1 # x=sx
318  mv t1, a2 # bx
319  mv t3, a4 # by
320  li s0, W_WIDTH
321  la s1, texture_size
322  lhu s1, 0(s1)
323  li t4, NInf
324  fmv.w.x fa0, t4
325  la t4, texture_size
326  lhu t4, 0(t4)
327  fcvt.s.w fa1, t4
328  la t4, texture_size
329  lhu t4, 2(t4)
330  fcvt.s.w fa2, t4
331  li t4, 2
332  li t5, 1
333  fcvt.s.w fa3, t4
334  fcvt.s.w fa4, t5
335  fdiv.s fa3, fa4, fa3 #
    0.5f to avoid rounding
336  # TODO: This number is
    not correct, correct
    should be 0x10000
337  # But somehow the zfar
    plane will be too near
338  li t4, 0x1
339  slli t4, t4, 13
340  fcvt.s.w fa5, t4 # for
```

```
    depth buffer
341  #li t4, 0x1F
342  #fcvt.s.w ft7, t4
343  lwu a0, 12(sp)
344  lwu a1, 16(sp)
345  la s2, fbx_buffer
346  la s3, fby_buffer
347  la s4, depth_buffer
348  la s5, depth_grid
349  # Light Intensity
350  flw ft0, 100(sp)
351  #li t4, 1 # DEBUG:
352  #fcvt.s.w ft0, t4 #
    Light as 1.0
353  li t4, 0x20
354  fcvt.s.w ft1, t4
355  fmul.s ft0, ft0, ft1
356  fsub.s ft0, ft0, fa3 #
    Avoid rounding
357  fcvt.w.s s7, ft0
358  bge t4, s7, 6f
359  mv s7, t4 # Clamp to max
    width
360 6:
361  bge s7, x0, 1f
362  mv s7, x0 # Clamp to
    zero
363 1:
364  bge t0, t1, 2f
365  mv t2, a3 # y=sy
366 3:
367  bge t2, t3, 4f
368
369  mul s6, s0, t2
370  add s6, s6, t0
371  slli s6, s6, 2
372
373  add t4, s6, s2 # Fbx pos
374  flw ft0, 0(t4)
375
376  feq.s t4, fa0, ft0
377  bne t4, x0, 5f # Only
    write used pixels
378
379  # Check depth_grid
380  add t4, s6, s4 # Depth
    pos
381  flw ft1, 0(t4)
382  fmul.s ft1, ft1, fa3
383  fsub.s ft1, ft1, fa3
384  fmul.s ft1, ft1, fa5
385  fadd.s ft1, ft1, fa3 #
    Avoid rounding
386  fneg.s ft1, ft1
387  fcvt.w.s t4, ft1 # Depth
    to integer
388  bge x0, t4, 5f # Cut if
    too far forward
389  li t5, 0xFFFF
390  bge t4, t5, 5f # Cut if
    too far behind
391  srli t5, s6, 1
392  add t5, t5, s5
393  lhu t6, 0(t5)
394  bge t4, t6, 5f # Cut if
    behind another pixel
395
396  # Store highest depth
397  sh t4, 0(t5)
398
399  #srli t5, t4, 11 #
    DEBUG:
400  #slli t4, t5, 6 #
401  #slli t6, t5, 11 # Show
    Depth
402  #or t5, t4, t5 # as
403  #or t5, t6, t5 # Gray
404  #add t4, s6, a0 #
405  #sh t5, 0(t4) #
406  #j 5f #
407
408  #add t4, s6, a0 # DEBUG:
409  #li t5, 0xFFFF # Show
410  #sh t5, 0(t4) # White
    Only
411  #j 5f #
412
413  fmul.s ft0, fa1, ft0
414  fsub.s ft0, ft0, fa3 #
    Avoid rounding
415  fcvt.wu.s t4, ft0 # UV X
    to integer
416
417  #srli t5, t4, 1 # DEBUG:
418  #mv t5, t4 #
419  #addi t5, t5, 2 #
420  #slli t4, t5, 6 #
421  #slli t6, t5, 11 # Show
    UV X
422  #or t5, t4, t5 # as
423  #or t5, t6, t5 # Gray
424  #add t4, s6, a0 #
425  #sh t5, 0(t4) #
426  #j 5f #
427
428  # Clamp Value
429  la t5, texture_size
430  lhu t5, 0(t5)
431  addi t5, t5, -1
432  bge t5, t4, 6f
433  mv t4, t5 # Clamp to max
    width
434 6:
435  bge t4, x0, 6f
436  mv t4, x0 # Clamp to
    zero
437 6:
438
439  add t5, s6, s3 # Fby pos
440  flw ft0, 0(t5)
441  fsub.s ft0, fa4, ft0
442  fmul.s ft0, fa2, ft0
443  fsub.s ft0, ft0, fa3 #
    Avoid rounding
444  fcvt.wu.s t5, ft0 # UV Y
    to integer
445
446  #srli t5, t5, 1 # DEBUG:
447  #addi t5, t5, 2 #
448  #slli t4, t5, 6 #
449  #slli t6, t5, 11 # Show
    UV Y
450  #or t5, t4, t5 # as
451  #or t5, t6, t5 # Gray
452  #add t4, s6, a0 #
453  #sh t5, 0(t4) #
454  #j 5f #
455
456  # Clamp Value
457  la t6, texture_size
458  lhu t6, 2(t6)
459  addi t6, t6, -1
460  bge t6, t5, 6f
461  mv t5, t6 # Clamp to max
    width
462 6:
463  bge t5, x0, 6f
464  mv t5, x0 # Clamp to
    zero
465 6:
466
467  mul t5, t5, s1
468  add t4, t4, t5
469  slli t4, t4, 1
470  add t4, t4, a1
471
472  lhu t4, 0(t4) # Texture
473  li t6, 0
474  # Blue
475  andi t5, t4, 0x001F
476  mul t5, t5, s7
477  srli t5, t5, 5
478  andi t5, t5, 0x001F
479  or t6, t6, t5
480  # Green
481  andi t5, t4, 0x07E0
482  mul t5, t5, s7
483  srli t5, t5, 5
484  andi t5, t5, 0x07E0
485  or t6, t6, t5
486  # Red
487  li t5, 0xF800
488  and t5, t5, t4
489  mul t5, t5, s7
490  srli t5, t5, 5
491  li t4, 0xF800
492  and t5, t5, t4
493  or t5, t6, t5
494
495  add t4, s6, a0 # Frame
    Buffer Pos
496  sh t5, 0(t4)
497 5:
498  addi t2, t2, 1
499  j 3b
500 4:
501  addi t0, t0, 1
502  j 1b
503 2:
504  ld ra, 8(sp)
505  addi sp, sp, 128
506  ret
507 #end
508
509 # fa1 - Value
510 # a2 - Dimension
511 floatToDimension:
512  addi sp, sp, -24
513  sd ra, 8(sp)
514  sd a2, 16(sp)
515
516  # From float cordinates
    to screen values
517  li t4, 1
518  fcvt.s.w ft0, t4
519  fcvt.s.w ft1, a2
520  fadd.s fa1, fa1, ft0
521  fmul.s fa1, fa1, ft1
522  fcvt.w.s a1, fa1
523
```

```
524  # Clamp Dimensions
525  mv a3, a2
526  li a2, 0
527  slli a3, a3, 1
528  addi a3, a3, -1
529  jal ra, clampi
530  srli a1, a1, 1
531
532  ld a2, 16(sp)
533  ld ra, 8(sp)
534  addi sp, sp, 24
535  ret
536 #end
537
538 # a1 - Value
539 # a2 - Min
540 # a3 - Max
541 clampi:
542  addi sp, sp, -16
543  sd ra, 8(sp)
544
545  bge a1, a2, 1f
546  mv a1, a2
547 1:
548  bge a3, a1, 2f
549  mv a1, a3
550 2:
551  ld ra, 8(sp)
552  addi sp, sp, 16
553  ret
554 #end
555
556 # a1 - v0
557 # a2 - v1
558 # a3 - v2
559 # a4 - v3
560 minmaxi:
561  addi sp, sp, -16
562  sd ra, 8(sp)
563
564  mv t0, a1
565  mv t1, a1
566
567 # Min
568  bge a2, t0, 1f
569  mv t0, a2
570 1:
571  bge a3, t0, 1f
572  mv t0, a3
573 1:
574  bge a4, t0, 1f
575  mv t0, a4
576 1:
577
578 # Max
579  bge t1, a2, 1f
580  mv t1, a2
581 1:
582  bge t1, a3, 1f
583  mv t1, a3
584 1:
585  bge t1, a4, 1f
586  mv t1, a4
587 1:
588
589  mv a1, t0
590  mv a2, t1
591
592  ld ra, 8(sp)
593  addi sp, sp, 16
594  ret
595 #end
596
597 .globl clearScreen
598 # a0 - Frame Buffer
599 # a1 - Color
600 clearScreen:
601  addi sp, sp, -16
602  sd ra, 8(sp)
603
604  li t0, 0
605  li t1, W_WIDTH
606  li t3, W_HEIGHT
607  la t4, depth_grid
608 1:
609  bge t0, t1, 2f
610  li t2, 0
611 3:
612  bge t2, t3, 4f
613
614  mul t5, t1, t2
615  add t5, t5, t0
616
617  slli t6, t5, 2
618  add t6, t6, a0
619  sh a1, 0(t6)
620
621  slli t6, t5, 1
622  add t6, t6, t4
623  li t5, 0xFFFF
624  sh t5, 0(t6)
625
626  addi t2, t2, 1
627  j 3b
628 4:
629  addi t0, t0, 1
630  j 1b
631 2:
632
633  ld ra, 8(sp)
634  addi sp, sp, 16
635  ret
636 #end
637
638 .globl fillPolygon
639 # a0 - Frame Buffer
640 # a1 - X Min Border
641 # a2 - X Max Border
642 # a3 - Y Min Border
643 # a4 - Y Max Border
644 fillPolygon:
645  addi sp, sp, -16
646  sd ra, 8(sp)
647
648  li t0, W_WIDTH # f_y
649  mv t1, a1
650
651  li t2, NInf
652  fmv.w.x fa0, t2
653 1:
654  beq t1, a2, 2f
655
656  li t3, -1 # i_y
657  li t4, -1 # f_y
658
659  mv t2, a3
660 3:
661  blt a4, t2, 4f
662
663  mul t5, t2, t0
664  add t5, t5, t1
665  slli t5, t5, 2
666  add t5, t5, a0
667
668  flw ft0, 0(t5)
669  feq.s t5, fa0, ft0
670  bne t5, x0, 5f # Only
    write used pixels
671
672  bne t3, t4, 6f
673 7:
674  mv t3, t2 # i_y = y
675  j 5f
676 6:
677  mv t5, t2
678  addi t5, t5, -1
679  beq t3, t5, 7b
680
681  mv t4, t2 # f_y = y
682  j 4f
683 5:
684  addi t2, t2, 1
685  j 3b
686 4:
687
688  beq t3, t4, 4f
689
690  mul t5, t3, t0
691  add t5, t5, t1
692  slli t5, t5, 2
693  add t5, t5, a0
694  flw ft0, 0(t5) #
    current_val
695
696  mul t5, t4, t0
697  add t5, t5, t1
698  slli t5, t5, 2
699  add t5, t5, a0
700  flw ft1, 0(t5)
701  sub t5, t4, t3
702  fcvt.s.w ft2, t5
703  fsub.s ft1, ft1, ft0
704  fdiv.s ft1, ft1, ft2 #
    delta_val
705
706  addi t3, t3, 1
707 3:
708  bge t3, t4, 4f
709
710  mul t5, t3, t0
711  add t5, t5, t1
712  slli t5, t5, 2
713  add t5, t5, a0
714
715  fadd.s ft0, ft0, ft1
716  fsw ft0, 0(t5)
717
718  addi t3, t3, 1
719  j 3b
720 4:
721  addi t1, t1, 1
722  j 1b
723 2:
724
725  ld ra, 8(sp)
726  addi sp, sp, 16
727  ret
728 #end
729
730 .globl drawLine
```

```
731 # a0 - Frame Buffer
732 # a1 - x0
733 # a2 - x1
734 # a3 - y0
735 # a4 - y1
736 # fa0 - val0
737 # fa1 - val1
738 drawLine:
739  addi sp, sp, -16
740  sd ra, 8(sp)
741
742  mv s0, a1
743  mv s1, a2
744  mv s2, a3
745  mv s3, a4
746  fmv.s fs0, fa0
747  fmv.s fs1, fa1
748
749  sub t0, s1, s0 # dx =
    x1-x0
750  sub t1, s3, s2 # dy =
    y1-y0
751
752  # Special Cases
753  # Straight Vertical Line
754  bne t0, x0, 1f
755  # y0 must be smaller
    than y1
756  bge s3, s2, 2f
757  mv t0, s2
758  mv s2, s3
759  mv s3, t0
760  fmv.s ft0, fs0
761  fmv.s fs0, fs1
762  fmv.s fs1, ft0
763  sub t1, x0, t1
764 2:
765  fcvt.s.w ft1, t1 # dy
766  fsub.s ft2, fs1, fs0 #
    val1-val0
767  fdiv.s ft1, ft2, ft1 #
    delta_val
768
769  fmv.s ft0, fs0
770  mv t0, s2
771  mv t1, s0
772  li t2, W_WIDTH
773
774 2:
775  blt s3, t0, 3f
776  fadd.s ft0, ft0, ft1 #
    current_val
777
778  mul t3, t0, t2
779  add t3, t3, t1 #
    x+WIN_WIDTH*y
780
781  slli t3, t3, 2
782  add t3, t3, a0
783  fsw ft0, 0(t3)
784
785  addi t0, t0, 1
786  j 2b
787 3:
788  ld ra, 8(sp)
789  addi sp, sp, 16
790  ret
791 1:
792  # Straight Horizontal
    Line
793  bne t1, x0, 1f
794  # x0 must be smaller
    than x1
795  bge s1, s0, 2f
796  mv t1, s0
797  mv s0, s1
798  mv s1, t1
799  fmv.s ft0, fs0
800  fmv.s fs0, fs1
801  fmv.s fs1, ft0
802  sub t0, x0, t0
803 2:
804  fcvt.s.w ft1, t0 # dx
805  fsub.s ft2, fs1, fs0 #
    val1-val0
806  fdiv.s ft1, ft2, ft1 #
    delta_val
807
808  fmv.s ft0, fs0
809  mv t0, s0
810  mv t1, s2
811  li t2, W_WIDTH
812
813 2:
814  blt s1, t0, 3f
815  fadd.s ft0, ft0, ft1 #
    current_val
816
817  mul t3, t1, t2
818  add t3, t3, t0 #
    x+WIN_WIDTH*y
819
820  slli t3, t3, 2
821  add t3, t3, a0
822  fsw ft0, 0(t3)
823
824  addi t0, t0, 1
825  j 2b
826 3:
827  ld ra, 8(sp)
828  addi sp, sp, 16
829  ret
830 1:
831
832  fcvt.s.w ft0, t0
833  fcvt.s.w ft1, t1
834  fdiv.s ft0, ft1, ft0
835  fabs.s ft0, ft0 # der
836
837  li t2, 1
838  fcvt.s.w ft1, t2
839  fle.s t2, ft0, ft1
840  # Line if dy is smaller
    than dx
841  beq t2, x0, 1f
842  # x0 must be smaller
    than x1
843  bge s1, s0, 2f
844  mv t2, s0
845  mv s0, s1
846  mv s1, t2
847  mv t2, s2
848  mv s2, s3
849  mv s3, t2
850  fmv.s ft1, fs0
851  fmv.s fs0, fs1
852  fmv.s fs1, ft1
853  sub t0, x0, t0
854  sub t1, x0, t1
855 2:
856  li t3, 1 # sign
857  bge t1, x0, 2f
858  li t3, -1
859 2:
860  fcvt.s.w ft1, t0 # dx
861  fsub.s ft2, fs1, fs0 #
    val1-val0
862  fdiv.s ft1, ft2, ft1 #
    delta_val
863
864  fmv.w.x ft2, x0
865  fmv.s ft3, fs0
866  li t0, 1
867  li t1, 2
868  fcvt.s.w ft4, t0 # 1.0
869  fcvt.s.w ft5, t1
870  fdiv.s ft5, ft4, ft5 #
    0.5
871  mv t0, s0
872  mv t1, s2
873  li t2, W_WIDTH
874
875  #li t6, 0x1F
876  #fcvt.s.w ft7, t6
877 2:
878  blt s1, t0, 3f
879  fadd.s ft2, ft2, ft0 #
    er
880  fadd.s ft3, ft3, ft1 #
    current_val
881
882  mul t4, t1, t2
883  add t4, t4, t0 #
    x+WIN_WIDTH*y
884
885  #fmul.s ft6, ft7, ft3
886  #fcvt.wu.s t5, ft6
887
888  slli t4, t4, 2
889  add t4, t4, a0
890  fsw ft3, 0(t4)
891
892  addi t0, t0, 1
893  fle.s t5, ft5, ft2
894  beq t5, x0, 4f
895  add t1, t1, t3
896  fsub.s ft2, ft2, ft4
897 4:
898  j 2b
899 3:
900  ld ra, 8(sp)
901  addi sp, sp, 16
902  ret
903
904  # Line if dx is smaller
    than dy
905 1:
906  # y0 must be smaller
    than y1
907  bge s3, s2, 2f
908  mv t2, s0
909  mv s0, s1
910  mv s1, t2
911  mv t2, s2
912  mv s2, s3
913  mv s3, t2
914  fmv.s ft1, fs0
915  fmv.s fs0, fs1
916  fmv.s fs1, ft1
917  sub t0, x0, t0
```

```
925  fsub.s ft2, fs1, fs0 # val1-val0
926  fdiv.s ft1, ft2, ft1 # delta_val
927
928  fmv.w.x ft2, x0
929  fmv.s ft3, fs0
930  li t0, 1
931  li t1, 2
932  fcvt.s.w ft4, t0 # 1.0
933  fcvt.s.w ft5, t1
934  fdiv.s ft5, ft4, ft5 # 0.5
935  fdiv.s ft0, ft4, ft0
936  mv t0, s2
937  mv t1, s0
938  li t2, W_WIDTH
939
940  #li t6, 0x1F
941  #fcvt.s.w ft7, t6
942 2:
943  blt s3, t0, 3f
944  fadd.s ft2, ft2, ft0 # er
945  fadd.s ft3, ft3, ft1 # current_val
946
947  mul t4, t0, t2
948  add t4, t4, t1 # x+WIN_WIDTH*y
949
950  #fmul.s ft6, ft7, ft3
951  #fcvt.wu.s t5, ft6
952
953  slli t4, t4, 2
954  add t4, t4, a0
955  fsw ft3, 0(t4)
956
957  addi t0, t0, 1
958  fle.s t5, ft5, ft2
959  beq t5, x0, 4f
960  add t1, t1, t3
961  fsub.s ft2, ft2, ft4
962 4:
963  j 2b
964 3:
965
966  ld ra, 8(sp)
967  addi sp, sp, 16
968  ret
969 #end
```

render.S

```
1   .section .text
2   .align 8
3
4   # Base Matrix
5   .comm base_matrix 4*4*4
6
7   .globl renderSquare
8   # a0 - Frame Buffer
9   # a1 - Texture
10  # a2 - Vertexes
11  # fa0 - Light Intensity
12  renderSquare:
13   addi sp, sp, -32
14   sd ra, 8(sp)
15   sw a0, 12(sp)
16   sw a1, 16(sp)
17   sw a2, 20(sp)
18   fsw fa0, 24(sp)
19
20   # Multiply Vertexes
21   lwu a0, 20(sp)
22   jal ra, multiplyModel
23   lwu a0, 20(sp)
24   addi a0, a0, 20
25   jal ra, multiplyModel
26   lwu a0, 20(sp)
27   addi a0, a0, 40
28   jal ra, multiplyModel
29   lwu a0, 20(sp)
30   addi a0, a0, 60
31   jal ra, multiplyModel
32
33   # Draw Square
34   lwu a0, 12(sp)
35   lwu a1, 16(sp)
36   lwu a2, 20(sp)
37   addi a3, a2, 20
38   addi a4, a2, 40
39   addi a5, a2, 60
40   flw fa0, 24(sp)
41   jal ra, drawSquare
42
43   ld ra, 8(sp)
44   addi sp, sp, 32
45   ret
46  #end
```

sd.S

```
1   .section .text
2   .align 1
3
4   .include "inc/mmap.inc"
5   .include "inc/spi.inc"
6   .include "inc/sysctl.inc"
7   .include "inc/pinmap.inc"
8   .include "inc/sd.inc"
9
10  .comm sdFrame, 6
11  .comm sdAFrame, 6
12  .comm sdReadRes, 514
13
14  .globl sdInitialize
15  sdInitialize:
16   addi sp, sp, -16
17   sd ra, 8(sp)
18
19   # Setup SD
20   jal ra, sdSetup
21
22   # CMD 0
23   la a0, sdFrame
24   li t0, SD_CMD0
25   sb t0, 0(a0)
26   sb x0, 1(a0)
27   sb x0, 2(a0)
28   sb x0, 3(a0)
29   sb x0, 4(a0)
30   sb x0, 5(a0)
31   li a1, 5
32   jal ra, crc7get
33   # Send Command
34   li a1, 6
35   jal ra, sdSPIwrite
36   # Wait for response
37   # Response: 0x01 = OK
38   la a0, sdReadRes
39   li a1, 1
40   li a2, 0xFFF
41  2:
42   jal ra, sdSPIread
43   lbu t1, 0(a0)
44   li t2, 0xFF
45   bne t1, t2, 3f
46   addi a2, a2, -1
47   bne a2, x0, 2b
48   li a0, 1
49   j 1f # Failed to read
50  3:
51   # Sync SD Card
52   jal ra, syncData
53
54   # CMD 8
55   la a0, sdFrame
56   li t0, SD_CMD8
57   sb t0, 0(a0)
58   li t1, SD_CMD8_ARG
59   sb t1, 4(a0)
60   srli t1, t1, 8
61   sb t1, 3(a0)
62   srli t1, t1, 8
63   sb t1, 2(a0)
64   srli t1, t1, 8
65   sb t1, 1(a0)
66   sb x0, 5(a0)
67   li a1, 5
68   jal ra, crc7get
69   # Send Command
70   li a1, 6
71   jal ra, sdSPIwrite
72   # Wait for response
73   # Response: 0x01000001nn means SD card V2+ compatible and voltage accepted
74   li a2, 0xFFF
75   la a0, sdReadRes
76   li a1, 1
77  2:
78   jal ra, sdSPIread
79   lbu t1, 0(a0)
80   li t2, 0xFF
81   bne t1, t2, 3f
82   addi a2, a2, -1
83   bne a2, x0, 2b
84   li a0, 1
85   j 1f # Failed to read
86  3:
87   addi a0, a0, 1
88   li a1, 5
89   jal ra, sdSPIread
90
91   # Sync SD Card
92   jal ra, syncData
93
94   # CMD 55
95   la a0, sdFrame
96   li t0, SD_CMD55
97   sb t0, 0(a0)
98   sb x0, 1(a0)
99   sb x0, 2(a0)
100  sb x0, 3(a0)
101  sb x0, 4(a0)
102  sb x0, 5(a0)
103  jal ra, crc7get
104  # ACMD 41
105  la a0, sdAFrame
106  li t0, SD_ACMD41
107  sb t0, 0(a0)
108  li t1, 0x40
```

```
109  sb t1, 1(a0)
110  sb x0, 2(a0)
111  sb x0, 3(a0)
112  sb t1, 4(a0)
113  sb x0, 5(a0)
114  li a1, 5
115  jal ra, crc7get
116
117  # Try to get a response
118  li a2, 0xFF
119 2:
120  # Send Command 55
121  la a0, sdFrame
122  li a1, 6
123  jal ra, sdSPIwrite
124  # Wait for response
125  # Response: 0x01 = OK
126  la a0, sdReadRes
127  li a1, 1
128  li a3, 0xFF
129 4:
130  jal ra, sdSPIread
131  lbu t1, 0(a0)
132  li t2, 0x01
133  beq t1, t2, 5f
134  addi a3, a3, -1
135  bne a3, x0, 4b
136  j 6f # Failed to read
137 5:
138  # Sync SD Card
139  jal ra, syncData
140  la a0, sdFrame
141  li t0, SD_CMD55
142  sb t0, 0(a0)
143
144  # Send Command A41
145  la a0, sdAFrame
146  li a1, 6
147  jal ra, sdSPIwrite
148  # Wait for response
149  # Response: 0x00 = OK
150  la a0, sdReadRes
151  li a1, 1
152  li a3, 0xFF
153 4:
154  jal ra, sdSPIread
155  lbu t1, 0(a0)
156  beq t1, x0, 3f
157  addi a3, a3, -1
158  bne a3, x0, 4b
159  j 6f # Failed to read
160  la a0, sdFrame
161  li t0, 0xFF
162  sb t0, 0(a0)
163 6:
164  # Sync SD Card
165  jal ra, syncData
166  la a0, sdFrame
167  li t0, SD_CMD55
168  sb t0, 0(a0)
169
170  addi a2, a2, -1
171  bne a2, x0, 2b
172  li a0, 1
173  j 1f # Failed to read
174 3:
175  # Sync SD Card
176  jal ra, syncData
177
178  # CMD 58
179  la a0, sdFrame
180  li t0, SD_CMD58
181  sb t0, 0(a0)
182  sb x0, 1(a0)
183  sb x0, 2(a0)
184  sb x0, 3(a0)
185  sb x0, 4(a0)
186  sb x0, 5(a0)
187  jal ra, crc7get
188  # Send Command
189  li a1, 6
190  jal ra, sdSPIwrite
191  # Read Response
192  # Response: [1]R1 + [2-
    5]OCR(MSB first)
193  # If CCS (Card Capacity
    Status) is set, card is
    SDHC or SDXC
194  li a2, 0xFF
195  la a0, sdReadRes
196  li a1, 1
197 2:
198  jal ra, sdSPIread
199  lbu t1, 0(a0)
200  li t2, 0xFF
201  bne t1, t2, 3f
202  addi a2, a2, -1
203  bne a2, x0, 2b
204  li a0, 1
205  j 1f # Failed to read
206 3:
207  addi a0, a0, 1
208  li a1, 4
209  jal ra, sdSPIread
210
211  # Check Response
212  # Verify if CCS and
    power up bit are set
213  li t0, 0xC0
214  lb t1, 0(a0)
215  and t1, t1, t0
216  beq t0, t1, 2f
217  li a0, 1
218  j 1f # Problem with card
219 2:
220  # Sync SD Card
221  jal ra, syncData
222
223  # Increase Clock Rate
224  li t0, SPI1_BASE_ADDR
225  li t1,
    SYSCTL_SPI1_FDIV_HS
226  sw t1,
    SPI_BAUD_RATE_OFF(t0)
227
228  li a0, 0 # Setup
    successful
229 1:
230  ld ra, 8(sp)
231  addi sp, sp, 16
232  ret
233 #end
234
235 syncData:
236  addi sp, sp, -16
237  sd ra, 8(sp)
238
239  # Dummy Data
240  la a0, sdFrame
241  li t0, 0xFF
242  sb t0, 0(a0)
243  # Send Command
244  li a1, 1
245  jal ra, sdSPIwrite
246
247  ld ra, 8(sp)
248  addi sp, sp, 16
249  ret
250 #end
251
252 sdSetup:
253  addi sp, sp, -16
254  sd ra, 8(sp)
255
256  # Setup SPI1 Chip Select
    Pin
257  li a0, PIN_SD_CS
258  li a1, PIN_SD_CS_CONFIG
259  jal ra, setupFPIOA
260
261  # Setup SPI1 Clock Pin
262  li a0, PIN_SD_CLK
263  li a1, PIN_SD_CLK_CONFIG
264  jal ra, setupFPIOA
265
266  # Setup SPI1 MISO Pin
267  li a0, PIN_SD_MISO
268  li a1,
    PIN_SD_MISO_CONFIG
269  jal ra, setupFPIOA
270
271  # Setup SPI1 MOSI Pin
272  li a0, PIN_SD_MOSI
273  li a1,
    PIN_SD_MOSI_CONFIG
274  jal ra, setupFPIOA
275
276  # Configure SPI
277  li t0, SPI1_BASE_ADDR
278  sw x0,
    SPI_ENABLE_OFF(t0)
279  sw x0,
    SPI_SLAVE_EN_OFF(t0)
280  sw x0,
    SPI_INTR_MASK_OFF(t0)
281  sw x0,
    SPI_DMA_CTL_OFF(t0)
282  sw x0,
    SPI_DMA_TDL_OFF(t0)
283  sw x0,
    SPI_DMA_RDL_OFF(t0)
284  sw x0,
    SPI_ENDIAN_OFF(t0)
285  sw x0, SPI_SCTL0_OFF(t0)
286  li t1, SYSCTL_SPI1_FDIV
287  sw t1,
    SPI_BAUD_RATE_OFF(t0)
288  li t1, SPI_WORK_MODE
289  slli t1, t1, 6
290  li t2, SPI_TRANS_M
291  slli t2, t2, 8
292  or t1, t1, t2
293  li t2, 8
294  addi t2, t2, -1
295  slli t2, t2, 16
296  or t1, t1, t2
297  li t2, SPI1_FRM_FMT
298  slli t2, t2, 21
299  or t1, t1, t2
300  sw t1, SPI_CTL0_OFF(t0)
```

```
301
302  ld ra, 8(sp)
303  addi sp, sp, 16
304  ret
305 #end
306
307 .globl sdSectorRead
308 # a0 - Data Address
309 # a1 - Sector
310 sdSectorRead:
311  addi sp, sp, -24
312  sd ra, 8(sp)
313  sd a0, 16(sp)
314
315  # CMD 17
316  la a0, sdFrame
317  li t0, SD_CMD17
318  sb t0, 0(a0)
319  mv t1, a1
320  sb t1, 4(a0)
321  srli t1, t1, 8
322  sb t1, 3(a0)
323  srli t1, t1, 8
324  sb t1, 2(a0)
325  srli t1, t1, 8
326  sb t1, 1(a0)
327  sb x0, 5(a0)
328  li a1, 5
329  jal ra, crc7get
330  li t0, 0xFF
331  sb t0, 6(a0)
332  # Send Command
333  li a1, 7
334  jal ra, sdSPIwrite
335
336  # Wait for response
337  la a0, sdReadRes
338  li a1, 1
339  li a2, 0xFFF
340 2:
341  jal ra, sdSPIread
342  lbu t1, 0(a0)
343  li t2, 0xFF
344  bne t1, t2, 3f
345  addi a2, a2, -1
346  bne a2, x0, 2b
347  li a1, 1
348  j 1f # Failed to read
349 3:
350  # Wait for sector start
    code
351  # Response: 0xFE = OK
352  ld a0, 16(sp)
353  li a2, 0xFFF
354 2:
355  jal ra, sdSPIread
356  lbu t1, 0(a0)
357  li t2, 0xFE
358  beq t1, t2, 3f
359  addi a2, a2, -1
360  bne a2, x0, 2b
361  li a1, 1
362  j 1f # Failed to read
363 3:
364  # Read Sector
365  # Response: [512]Data +
    [2]CRC
366  li a1, 514
367  jal ra, sdSPIread
368
369  # Sync SD Card
370  jal ra, syncData
371
372  ld a0, 16(sp)
373  li a1, 0 # Successful
    read
374 1:
375  ld ra, 8(sp)
376  addi sp, sp, 24
377  ret
378 #end
379
380 .globl sdSPIwrite
381 # a0 - Data Address
382 # a1 - Data Length
383 sdSPIwrite:
384  addi sp, sp, -16
385  sd ra, 8(sp)
386
387  # Set SPI Transmission
    Mode
388  li t0, SPI1_BASE_ADDR
389  lw t1, SPI_CTL0_OFF(t0)
390  li t2, 0x3
391  slli t2, t2, 8
392  not t2, t2
393  and t1, t1, t2
394  li t2, SPI_TRANS_M
395  slli t2, t2, 8
396  or t1, t1, t2
397  sw t1, SPI_CTL0_OFF(t0)
398
399  # Enable SPI
400  li t1, 1
401  sw t1,
    SPI_ENABLE_OFF(t0)
402
403  # Enable SPI Slave
404  li t1, 1
405  slli t1, t1, SPI_SD_SS
406  sw t1,
    SPI_SLAVE_EN_OFF(t0)
407
408  # Loop while have data
    to send
409  mv t1, a0
410  mv t2, a1
411 1:
412  lw t3,
    SPI_TFIFO_LVL_OFF(t0)
413  li t4, 32
414  sub t3, t4, t3
415  li t4, 0
416  bge t2, t3, 2f
417  mv t3, t2
418 2:
419  # Store data until FIFO
    is filled
420  bge t4, t3, 3f
421  lbu t5, 0(t1)
422  sw t5,
    SPI_DATA_FIFO_OFF(t0)
423  addi t1, t1, 1
424  addi t4, t4, 1
425  j 2b
426 3:
427  sub t2, t2, t3
428  blt x0, t2, 1b
429
430  # Wait for SPI
431  li t1, 0x5
432  li t2, 0x4
433 1:
434  lwu t3,
    SPI_STATUS_OFF(t0)
435  and t3, t3, t1
436  bne t3, t2, 1b
437
438  # Disable SPI Slave and
    Output
439  sw x0,
    SPI_SLAVE_EN_OFF(t0)
440  sw x0,
    SPI_ENABLE_OFF(t0)
441
442  ld ra, 8(sp)
443  addi sp, sp, 16
444  ret
445 #end
446
447 .globl sdSPIread
448 # a0 - Data Address
449 # a1 - Data Length
450 sdSPIread:
451  addi sp, sp, -16
452  sd ra, 8(sp)
453
454  # Set SPI Transmission
    Mode
455  li t0, SPI1_BASE_ADDR
456  lw t1, SPI_CTL0_OFF(t0)
457  li t2, 0x3
458  slli t2, t2, 8
459  not t2, t2
460  and t1, t1, t2
461  li t2, SPI_RECEV_M
462  slli t2, t2, 8
463  or t1, t1, t2
464  sw t1, SPI_CTL0_OFF(t0)
465
466  # Set Receive Data
    Length
467  mv t1, a1
468  addi t1, t1, -1
469  sw t1, SPI_CTL1_OFF(t0)
470
471  # Enable SPI
472  li t1, 1
473  sw t1,
    SPI_ENABLE_OFF(t0)
474
475  # Clear FIFO Buffer
476  li t1, 0
477  not t1, t1
478  sw t1,
    SPI_DATA_FIFO_OFF(t0)
479
480  # Enable SPI Slave
481  li t1, 1
482  slli t1, t1, SPI_SD_SS
483  sw t1,
    SPI_SLAVE_EN_OFF(t0)
484
485  # Loop while have data
    to receive
486  mv t1, a0
487  mv t2, a1
488 1:
489  lw t3,
    SPI_RFIFO_LVL_OFF(t0)
```

```
490  li t4, 0
491  bge t2, t3, 2f
492  mv t3, t2
493 2:
494  # Store data until FIFO
    is filled
495  bge t4, t3, 3f
496  lw t5,
    SPI_DATA_FIFO_OFF(t0)
497  sb t5, 0(t1)
498  addi t1, t1, 1
499  addi t4, t4, 1
500  j 2b
501 3:
502  sub t2, t2, t3
503  blt x0, t2, 1b
504
505  # Disable SPI Slave and
    Output
506  sw x0,
    SPI_SLAVE_EN_OFF(t0)
507  sw x0,
    SPI_ENABLE_OFF(t0)
508
509  ld ra, 8(sp)
510  addi sp, sp, 16
511  ret
512 #end
```

sysctl.S

```
1   .section .text
2   .align 2
3
4   .include "inc/mmap.inc"
5   .include "inc/sysctl.inc"
6
7   .globl setupPLL
8   setupPLL:
9    addi sp, sp, -16
10   sd ra, 8(sp)
11
12   # PLL0 -----
13   li t0, SYSCTL_BASE_ADDR
14   # Change CPU Clock to
    XTAL
15   addi t1, t0,
    SYSCTL_ACLK_CTL
16   ld t2, 0(t1)
17   li t3, ACLK_PLL_EN_MASK
18   not t3, t3
19   and t2, t2, t3
20   sd t2, 0(t1)
21
22   # Disable PLL output
23   addi t1, t0,
    SYSCTL_PLL0_CTL
24   ld t2, 0(t1)
25   lui t3,
    PLL_OUTPUT_MASK_LUI
26   not t3, t3
27   and t2, t2, t3
28   sd t2, 0(t1)
29
30   # Power off PLL
31   ld t2, 0(t1)
32   lui t3,
    PLL_POWER_MASK_LUI
33   not t3, t3
34   and t2, t2, t3
35   sd t2, 0(t1)
36
37   # Set PLL new frequency
    value
38   ld t2, 0(t1)
39   li t3, PLL_FREQ_MASK
40   not t3, t3
41   and t2, t2, t3
42   li t3, SYSCTL_PLL0_FREQ
43   or t2, t2, t3
44   sd t2, 0(t1)
45
46   # Power on PLL
47   ld t2, 0(t1)
48   lui t3,
    PLL_POWER_MASK_LUI
49   or t2, t2, t3
50   sd t2, 0(t1)
51   # Wait
52   li t3, 0xFF
53  1:
54   addi t3, t3, -1
55   bge t3, x0, 1b
56
57   # Reset PLL
58   ld t2, 0(t1)
59   lui t3,
    PLL_RESET_MASK_LUI
60   not t4, t3
61   and t2, t2, t4
62   sd t2, 0(t1)
63   or t2, t2, t3
64   sd t2, 0(t1)
65   # Wait
66   li t3, 0xFF
67  1:
68   addi t3, t3, -1
69   bge t3, x0, 1b
70
71   # Release Reset
72   ld t2, 0(t1)
73   lui t3,
    PLL_RESET_MASK_LUI
74   not t4, t3
75   and t2, t2, t4
76   sd t2, 0(t1)
77
78   # Get lock status, wait
    for PLL to stabilize
79   addi t1, t0,
    SYSCTL_PLL_LOCK
80  1:
81   ld t2, 0(t1)
82   li t3, LOCK_PLL0_MASK
83   and t2, t2, t3
84   beq t2, t3, 2f
85   # Clear slip
86   ld t2, 0(t1)
87   li t3, LOCK_SLIP0_CLEAR
88   or t2, t2, t3
89   sd t2, 0(t1)
90   j 1b
91  2:
92
93   # Enable PLL output
94   addi t1, t0,
    SYSCTL_PLL0_CTL
95   ld t2, 0(t1)
96   lui t3,
    PLL_OUTPUT_MASK_LUI
97   or t2, t2, t3
98   sd t2, 0(t1)
99
100  # Change CPU Clock to
    PLL
101  addi t1, t0,
    SYSCTL_ACLK_CTL
102  ld t2, 0(t1)
103  li t3, ACLK_PLL_EN_MASK
104  or t2, t2, t3
105  sd t2, 0(t1)
106
107  ld ra, 8(sp)
108  addi sp, sp, 16
109  ret
110 # end
```

tft.S

```
1   .section .text
2   .align 2
3
4   .include "inc/mmap.inc"
5   .include "inc/tft.inc"
6   .include "inc/spi.inc"
7   .include "inc/sysctl.inc"
8   .include "inc/dma.inc"
9   .include "inc/pinmap.inc"
10  .include "inc/gpio.inc"
11  .include "inc/image.inc"
12
13  .globl tftInitialize
14  tftInitialize:
15   addi sp, sp, -16
16   sd ra, 8(sp)
17
18   # Setup TFT
19   jal ra, tftSetup
20
21   # Software Reset
22   la a0, tft_cmd
23   li t0, SOFTWARE_RESET
24   sw t0, 0(a0)
25   li a1, 1
26   li a2, 8
27   li a3, 0
28   jal ra, tftWriteDMA
29
30   # Sleep Off
31   la a0, tft_cmd
32   li t0, SLEEP_OFF
33   sw t0, 0(a0)
34   li a1, 1
35   li a2, 8
36   li a3, 0
37   jal ra, tftWriteDMA
38
39   # Set Pixel Format
40   la a0, tft_cmd
41   li t0, PIXEL_FORMAT_SET
42   sw t0, 0(a0)
43   li a1, 1
44   li a2, 8
45   li a3, 0
46   jal ra, tftWriteDMA
47   la a0, tft_cmd
48   li t0, PIXEL_565_16BIT
49   sw t0, 0(a0)
50   li a1, 1
51   li a2, 8
52   li a3, 1
53   jal ra, tftWriteDMA
54
55   # Set Display Direction
```

```
56   la a0, tft_cmd
57   li t0, MEMORY_ACCESS_CTL
58   sw t0, 0(a0)
59   li a1, 1
60   li a2, 8
61   li a3, 0
62   jal ra, tftWriteDMA
63   la a0, tft_cmd
64   li t0, DIR_YX_LR_DU
65   sw t0, 0(a0)
66   li a1, 1
67   li a2, 8
68   li a3, 1
69   jal ra, tftWriteDMA
70
71   # Turn Display On
72   la a0, tft_cmd
73   li t0, DISPLAY_ON
74   sw t0, 0(a0)
75   li a1, 1
76   li a2, 8
77   li a3, 0
78   jal ra, tftWriteDMA
79
80   # Set Display Size
81   # Width
82   li t0, W_WIDTH
83   c.addi t0, -1
84   srli t1, t0, 8
85   andi t1, t1, 0xFF
86   andi t2, t0, 0xFF
87   la t0, tft_display_size
88   sw t1, 0x8(t0)
89   sw t2, 0xC(t0)
90   # Height
91   li t0, W_HEIGHT
92   c.addi t0, -1
93   srli t1, t0, 8
94   andi t1, t1, 0xFF
95   andi t2, t0, 0xFF
96   la t0, tft_display_size
97   sw t1, 0x18(t0)
98   sw t2, 0x1C(t0)
99
100  ld ra, 8(sp)
101  addi sp, sp, 16
102  ret
103 #end
104
105 .globl tftRefreshDisplay
106 # a0 - Frame Buffer
    Address
107 tftRefreshDisplay:
108  addi sp, sp, -24
109  sd a0, 16(sp)
110  sd ra, 8(sp)
111
112  # Set Display Writable
    Area
113  # Width
114  la a0, tft_cmd
115  li t0,
    HORIZONTAL_ADDRESS_SET
116  sw t0, 0(a0)
117  li a1, 1
118  li a2, 8
119  li a3, 0
120  jal ra, tftWriteDMA
121  la a0, tft_display_size
122  li a1, 4
123  li a2, 8
124  li a3, 1
125  jal ra, tftWriteDMA
126  # Height
127  la a0, tft_cmd
128  li t0,
    VERTICAL_ADDRESS_SET
129  sw t0, 0(a0)
130  li a1, 1
131  li a2, 8
132  li a3, 0
133  jal ra, tftWriteDMA
134  la a0, tft_display_size
135  c.addi a0, 0x10
136  li a1, 4
137  li a2, 8
138  li a3, 1
139  jal ra, tftWriteDMA
140
141  # Write Data to Display
    Frame Buffer
142  la a0, tft_cmd
143  li t0, MEMORY_WRITE
144  sw t0, 0(a0)
145  li a1, 1
146  li a2, 8
147  li a3, 0
148  jal ra, tftWriteDMA
149
150  # Send Frame Buffer Data
151  ld a0, 16(sp)
152  li a1, W_WIDTH
153  li t1, W_HEIGHT
154  mul a1, a1, t1
155  li a2, 16
156  li a3, 1
157  jal ra, tftWriteDMA
158
159  ld ra, 8(sp)
160  addi sp, sp, 24
161  ret
162 #end
163
164 tftSetup:
165  addi sp, sp, -16
166  sd ra, 8(sp)
167
168  # Attach Video to Memory
169  li t0, SYSCTL_BASE_ADDR
170  lw t1,
    SYSCTL_MISC_CTL(t0)
171  li t2, 0x400
172  not t3, t2
173  and t1, t1, t3
174  or t1, t1, t2
175  sw t1,
    SYSCTL_MISC_CTL(t0)
176
177  # Setup SPI0 Chip Select
    Pin
178  li a0, PIN_TFT_CS
179  li a1, PIN_TFT_CS_CONFIG
180  jal ra, setupFPIOA
181
182  # Setup SPI0 Clock Pin
183  li a0, PIN_TFT_CLK
184  li a1,
    PIN_TFT_CLK_CONFIG
185  jal ra, setupFPIOA
186
187  # Setup Data/Command Pin
188  li a0, PIN_TFT_DC
189  li a1, PIN_TFT_DC_CONFIG
190  jal ra, setupFPIOA
191  li a0, GPIOHS_TFT_DC
192  li a1, GPIO_OUTPUT
193  jal ra, setupGPIOHS
194
195  # Setup Reset Pin
196  li a0, PIN_TFT_RST
197  li a1,
    PIN_TFT_RST_CONFIG
198  jal ra, setupFPIOA
199  li a0, GPIOHS_TFT_RST
200  li a1, GPIO_OUTPUT
201  jal ra, setupGPIOHS
202
203  # Reset Display
204  li a0, 1
205  slli a0, a0,
    GPIOHS_TFT_RST
206  mv a1, a0
207  jal ra, outputGPIOHS
208  mv a1, x0
209  jal ra, outputGPIOHS
210
211  # Configure SPI
212  li t0, SPI0_BASE_ADDR
213  sw x0,
    SPI_ENABLE_OFF(t0)
214  sw x0,
    SPI_SLAVE_EN_OFF(t0)
215  sw x0,
    SPI_INTR_MASK_OFF(t0)
216  sw x0,
    SPI_DMA_CTL_OFF(t0)
217  sw x0,
    SPI_DMA_TDL_OFF(t0)
218  sw x0,
    SPI_DMA_RDL_OFF(t0)
219  sw x0,
    SPI_ENDIAN_OFF(t0)
220  sw x0, SPI_SCTL0_OFF(t0)
221  li t1, SYSCTL_SPI0_FDIV
222  sw t1,
    SPI_BAUD_RATE_OFF(t0)
223  li t1, SPI_WORK_MODE
224  slli t1, t1, 6
225  li t2, SPI_TRANS_M
226  slli t2, t2, 8
227  or t1, t1, t2
228  li t2, SPI0_FRM_FMT
229  slli t2, t2, 21
230  or t1, t1, t2
231  sw t1, SPI_CTL0_OFF(t0)
232
233  # Turn on Display
234  li a0, 1
235  slli a0, a0,
    GPIOHS_TFT_RST
236  mv a1, a0
237  jal ra, outputGPIOHS
238
239  ld ra, 8(sp)
240  addi sp, sp, 16
241  ret
242 #end
243
244 # a0 - Data Address
245 # a1 - Data Length
```

```
# a2 - Data Bit Length
# a3 - Data/Command Set
tftWriteDMA:
 addi sp, sp, -16
 sd ra, 8(sp)

 # Set DC GPIO Pin
 addi sp, sp, -32
 sd a0, 8(sp)
 sd a1, 16(sp)
 sd a2, 24(sp)
 li t0, 1
 slli a0, t0,
    GPIOHS_TFT_DC
 mv t0, a3
 slli a1, t0,
    GPIOHS_TFT_DC
 jal ra, outputGPIOHS
 ld a2, 24(sp)
 ld a1, 16(sp)
 ld a0, 8(sp)
 addi sp, sp, 32

 # SPI Setup
 li t0, SPI0_BASE_ADDR
 li t1, 0x3 # FF Octal
 slli t1, t1, 21
 mv t2, a2 # Data Bit
    length
 addi t2, t2, -1
 slli t2, t2, 16
 or t1, t1, t2
 li t2, SPI_TRANS_M #
    Transmission Mode
 slli t2, t2, 8
 or t1, t1, t2
 sw t1, SPI_CTL0_OFF(t0)
 # Configure SPI
    Controller
 li t1, 8
 beq a2, t1, 1f
 li t1, 16
 beq a2, t1, 2f
 ld ra, 8(sp)
 addi sp, sp, 16
 ret
1:
 li t1, 2
 j 3f
2:
 li t1, 3
3:
 slli t1, t1, 8
 ori t1, t1, SPI0_IATM
 sw t1, SPI_SCTL0_OFF(t0)
 # Enable DMA on SPI
 li t1, 0x2
 sw t1,
    SPI_DMA_CTL_OFF(t0)
 # Enable SPI
 li t1, 0x1
 sw t1,
    SPI_ENABLE_OFF(t0)

 # Select DMA Channel
    Mode
 li t0, SYSCTL_BASE_ADDR
 lw t1,
    SYSCTL_DMA_SEL(t0)
 li t2, 0x3F
 li t3, 6
 li t4, TFT_DMA_CH
 mul t3, t3, t4
 sll t2, t2, t3
 not t2, t2
 and t1, t1, t2
 li t2, TFT_DMA_SPI0_TX
 sll t2, t2, t3
 or t1, t1, t2
 sw t1,
    SYSCTL_DMA_SEL(t0)

 # Get DMA Base Addresses
 li t0, DMA_BASE_ADDR
 li t1, TFT_DMA_CH
 li t2, DMA_CH_BASE_OFF
 mul t1, t1, t2
 add t1, t1, t0

 # Clear interrupt
 li t2, 0xFFFFFFFF
 sd t2,
    DMA_CH_INT_OFF(t1)

 # Disable DMA Channel
 ld t2, DMA_CH_EN_OFF(t0)
 li t3, 1
 slli t3, t3, TFT_DMA_CH
 not t4, t3
 and t2, t2, t4
 slli t3, t3, 8
 or t2, t2, t3
 sd t2, DMA_CH_EN_OFF(t0)

 # Wait while DMA is idle
 li t2, 1
 slli t2, t2, TFT_DMA_CH
1:
 ld t3, DMA_CH_EN_OFF(t0)
 and t3, t3, t2
 bne t3, x0, 1b

 # Set Channel Parameters
 li t3, 0xF79F0000000F
 not t3, t3
 and t2, t2, t3
 li t3, 0x9
 slli t3, t3, 32
 or t2, t2, t3
 li t3, TFT_DMA_CH
 slli t4, t3, 39
 or t2, t2, t4
 slli t3, t3, 44
 or t2, t2, t3
 sd t2,
    DMA_CH_CONF_OFF(t1)

 # Set Source and
    Destination Address
 sd a0,
    DMA_SRC_ADD_OFF(t1)
 li t2, TFT_SPI_DST_ADD
 sd t2,
    DMA_DST_ADD_OFF(t1)

 # Setup DMA Controller
 ld t2,
    DMA_CH_CTL_OFF(t1)
 li t3, 0x3FFFF55
 not t3, t3
 and t2, t2, t3
 li t3, 0x45244
 or t2, t2, t3
 sd t2,
    DMA_CH_CTL_OFF(t1)

 # Define Block Size
 mv t2, a1
 addi t2, t2, -1
 sd t2,
    DMA_BLOCK_S_OFF(t1)

 # Enable DMA
 ld t2, 0x10(t0)
 ori t2, t2, 0x3
 sd t2, 0x10(t0)

 # Enable Channel
 ld t2, DMA_CH_EN_OFF(t0)
 li t3, 1
 slli t3, t3, TFT_DMA_CH
 or t2, t2, t3
 slli t3, t3, 8
 or t2, t2, t3
 sd t2, DMA_CH_EN_OFF(t0)

 # Enable SPI Slave
 li t2, SPI0_BASE_ADDR
 li t3, 1
 slli t3, t3, SPI_TFT_SS
 sw t3,
    SPI_SLAVE_EN_OFF(t2)

 # Wait while DMA is idle
 li t2, 1
 slli t2, t2, TFT_DMA_CH
1:
 ld t3, DMA_CH_EN_OFF(t0)
 and t3, t3, t2
 bne t3, x0, 1b

 # Clear interrupt
 li t2, 0xFFFFFFFF
 sd t2,
    DMA_CH_INT_OFF(t1)

 # Wait for SPI
 li t0, SPI0_BASE_ADDR
 li t1, 0x5
 li t2, 0x4
1:
 lwu t3,
    SPI_STATUS_OFF(t0)
 and t3, t3, t1
 bne t3, t2, 1b

 # Disable SPI Slave and
    Output
 sw x0,
    SPI_SLAVE_EN_OFF(t0)
 sw x0,
    SPI_ENABLE_OFF(t0)

 ld ra, 8(sp)
 addi sp, sp, 16
 ret
# end

.section .data
.align 4
```

```
432
433 tft_cmd:
434  .word NO_OPERATION
435
436 tft_display_size:
437  .word 0, 0, 0, 0, 0, 0,
    0, 0
```

char.inc

```
1  # ASCII Character
   dictionary
2
3  .equ CHAR_NL, 0x0A
4  .equ CHAR_SPC, 0x20
5  .equ CHAR_HASH, 0x23
6  .equ CHAR_MINUS, 0x2D
7  .equ CHAR_DOT, 0x2E
8  .equ CHAR_FSLASH, 0x2F
9  .equ CHAR_0, 0x30
10 .equ CHAR_9, 0x39
11 .equ CHAR_P, 0x50
12 .equ CHAR_f, 0x66
13 .equ CHAR_n, 0x6E
14 .equ CHAR_t, 0x74
15 .equ CHAR_v, 0x76
```

colors.inc

```
1  # Identification colors
2  # COLOR | ERROR
3  # White | TFT failed to
   initialize
4  # Orange | Loading
5  # Yellow | SD card failed
   to initialize
6  # Green | File system not
   recognized
7  # Cyan | Invalid ppm file
8  # Blue | Problem with ppm
   file
9  # Magenta | Invalid obj
   file
10 # Red | Problem with obj
   file
11
12 .equ COLOR_ERROR_TFT,
   0xFFFF
13 .equ COLOR_LOADING,
   0xFD20
14 .equ COLOR_ERROR_SD,
   0xFFE0
15 .equ COLOR_ERROR_FAT,
   0x07E0
16 .equ COLOR_INVALID_PPM,
   0x07EE
17 .equ COLOR_PROBLEM_PPM,
   0x001F
18 .equ COLOR_INVALID_OBJ,
   0xF81F
19 .equ COLOR_PROBLEM_OBJ,
   0xF800
20
21 # Color list
22
23 .equ COLOR_BLACK, 0x0000
24 .equ COLOR_NAVY, 0x000F
25 .equ COLOR_DARKGREEN,
   0x03E0
26 .equ COLOR_DARKCYAN,
   0x03EF
27 .equ COLOR_MAROON, 0x7800
28 .equ COLOR_PURPLE, 0x780F
29 .equ COLOR_OLIVE, 0x7BE0
30 .equ COLOR_LIGHTGREY,
   0xC618
31 .equ COLOR_DARKGREY,
   0x7BEF
32 .equ COLOR_BLUE, 0x001F
33 .equ COLOR_GREEN, 0x07E0
34 .equ COLOR_CYAN, 0x07FF
35 .equ COLOR_RED, 0xF800
36 .equ COLOR_MAGENTA,
   0xF81F
37 .equ COLOR_YELLOW, 0xFFE0
38 .equ COLOR_WHITE, 0xFFFF
39 .equ COLOR_ORANGE, 0xFD20
40 .equ COLOR_GREENYELLOW,
   0xAFE5
41 .equ COLOR_PINK, 0xF81F
```

csr.inc

```
1  # Values for the Control
   and Status Registers
2
3  # mie Interupt ReQuest
   (IRQ) bits
4  .equ MIE_SOFTW_S_EN_MASK,
   0x002
5  .equ MIE_SOFTW_H_EN_MASK,
   0x004
6  .equ MIE_SOFTW_M_EN_MASK,
   0x008
7  .equ MIE_TIMER_S_EN_MASK,
   0x020
8  .equ MIE_TIMER_H_EN_MASK,
   0x040
9  .equ MIE_TIMER_M_EN_MASK,
   0x080
10 .equ MIE_EXTRN_S_EN_MASK,
   0x200
11 .equ MIE_EXTRN_H_EN_MASK,
   0x400
12 .equ MIE_EXTRN_M_EN_MASK,
   0x800
13
14 # mstatus
15 .equ MSTATUS_MIE_MASK,
   0x008
```

dma.inc

```
1  # Info for DMA functions
2
3  # Base Offsets
4  .equ DMA_CONF_OFF, 0x10
5  .equ DMA_CH_EN_OFF, 0x18
6  .equ DMA_CH_BASE_OFF,
   0x100
7
8  # Channel Offset
9  .equ DMA_SRC_ADD_OFF,
   0x100
10 .equ DMA_DST_ADD_OFF,
   0x108
11 .equ DMA_BLOCK_S_OFF,
   0x110
12 .equ DMA_CH_CTL_OFF,
   0x118
13 .equ DMA_CH_CONF_OFF,
   0x120
14 .equ DMA_CH_INT_OFF,
   0x198
```

fat.inc

```
1  # Info for FAT functions
2
3  # Settings
4  .equ PARTITION_NUMBER, 0
5
6  # Constants
7  # FAT flags
8  .equ FAT_BR_SIGNATURE,
   0x55AA
9  .equ FAT_SECTOR_SIZE,
   0x200
10 .equ FAT_TABLE_EOF,
   0x0FFFFFFF
11 .equ FAT_MAX_FILE_KIB,
   600000
12
13 # Partition Table
14 .equ PARTITION_START,
   0x1BE
15 .equ
   PARTITION_ENTRY_SIZE,
    16
16 .equ PARTITION_ID_OFF,
    0x4
17 .equ PARTITION_START_OFF,
   0x8
18 .equ PARTITION_SIZE_OFF,
   0xC
19 .equ PARTITION_FAT32_LBA,
   0x0C
20
21 # Boot Sector
22 .equ BOOT_BPS_OFF, 0xB
23 .equ BOOT_SPC_OFF, 0xD
24 .equ BOOT_RSVD_SIZE, 0xE
25 .equ BOOT_FAT_CPS, 0x10
26 .equ BOOT_FAT_SIZE, 0x24
```

files.inc

```
1  # Files to be loaded
2
3  ppm_file: .ascii
   "bitboy.ppm"
4   .byte 0
5  obj_file: .ascii
   "bitboy.obj"
6   .byte 0
```

fpioa.inc

```
1  # Info for FPIOA
   functions
2  .equ PINS_MAX, 48
3  .equ FPIOA_MAX, 256
4
5  # Configuration offsets
6  .equ FPIOA_TIE_EN_OFF,
   0xC0
7  .equ FPIOA_TIE_VAL_OFF,
   0xE0
8
9  # Multiplex tie values
```

```
10  # Bits enabled are:
11  # 16 - FUNC_SPI0_ARB
12  # 82 - FUNC_SPI1_ARB
13  .equ FPIOA_TIE0,
    0x00010000
14  .equ FPIOA_TIE1,
    0x00000000
15  .equ FPIOA_TIE2,
    0x00040000
16  .equ FPIOA_TIE3,
    0x00000000
17  .equ FPIOA_TIE4,
    0x00000000
18  .equ FPIOA_TIE5,
    0x00000000
19  .equ FPIOA_TIE6,
    0x00000000
20  .equ FPIOA_TIE7,
    0x00000000
21
22  # Region Mask
23  .equ FPIOA_INPUT_MASK,
    0x00F00000
24  .equ FPIOA_PULL_MASK,
    0x00070000
25  .equ FPIOA_OUTPUT_MASK,
    0x00003000
26  .equ FPIOA_DRIVING_MASK,
    0x00000F00
27  .equ FPIOA_FUNCTION_MASK,
    0x000000FF
28
29  # [23:20] Input
30  .equ FPIOA_ST, 0x800000 #
    Schmitt Trigger
31  .equ FPIOA_II, 0x200000 #
    Input Invert
32  .equ FPIOA_IE, 0x100000 #
    Input Enable
33
34  # [18:16] Pull
35  .equ FPIOA_NP, 0x00000 #
    No Pull
36  .equ FPIOA_PU, 0x10000 #
    Pull Up
37  .equ FPIOA_PD, 0x20000 #
    Pull Down
38  .equ FPIOA_SPU, 0x50000 #
    Strong Pull up
39
40  # [13:12] Output
41  .equ FPIOA_OI, 0x2000 #
    Output Invert
42  .equ FPIOA_OE, 0x1000 #
    Output Enable
43
44  # [11:8] Driving output
    current
45  .equ DS_0, 0x000
46  .equ DS_1, 0x100
47  .equ DS_2, 0x200
48  .equ DS_3, 0x300
49  .equ DS_4, 0x400
50  .equ DS_5, 0x500
51  .equ DS_6, 0x600
52  .equ DS_7, 0x700
53
54  # [7:0] Function select
55  # SPI0
56  .equ CH_SPI0_SS0, 0x0C
57  .equ CH_SPI0_SS1, 0x0D
58  .equ CH_SPI0_SS2, 0x0E
59  .equ CH_SPI0_SS3, 0x0F
60  .equ CH_SPI0_CLK, 0x11
61  # GPIOHS
62  .equ CH_GPIOHS0_SEL, 0x18
63  .equ CH_GPIOHS1_SEL, 0x19
64  .equ CH_GPIOHS2_SEL, 0x1A
65  .equ CH_GPIOHS3_SEL, 0x1B
66  .equ CH_GPIOHS4_SEL, 0x1C
67  .equ CH_GPIOHS5_SEL, 0x1D
68  .equ CH_GPIOHS6_SEL, 0x1E
69  .equ CH_GPIOHS7_SEL, 0x1F
70  # GPIO
71  .equ CH_GPIO0_SEL, 0x38
72  .equ CH_GPIO1_SEL, 0x39
73  .equ CH_GPIO2_SEL, 0x3A
74  .equ CH_GPIO3_SEL, 0x3B
75  .equ CH_GPIO4_SEL, 0x3C
76  .equ CH_GPIO5_SEL, 0x3D
77  .equ CH_GPIO6_SEL, 0x3E
78  .equ CH_GPIO7_SEL, 0x3F
```

gpio.inc

```
1   # Info for GPIO functions
2
3   # Direction
4   .equ GPIO_INPUT, 0x0
5   .equ GPIO_OUTPUT, 0x1
6
7   # GPIO ------------
8   .equ GPIO_MAX, 8
9
10  # Offset
11  .equ GPIO_OUT_OFF, 0x00
12  .equ GPIO_DIR_OFF, 0x04
13  .equ GPIO_IN_OFF, 0x50
14
15  # GPIOHS -----------
16  .equ GPIOHS_MAX, 32
17
18  # Offset
19  .equ GPIOHS_IN_VAL, 0x00
20  .equ GPIOHS_IN_EN, 0x04
21  .equ GPIOHS_OUT_EN, 0x08
22  .equ GPIOHS_OUT_VAL, 0x0C
```

image.inc

```
1   # Info for Image Related
    functions
2
3   # Frame Settings
4   .equ W_WIDTH, 320
5   .equ W_HEIGHT, 240
```

mmap.inc

```
1   # K210 memory map
2
3   .equ GPIOHS_BASE_ADDR,
    0x38001000
4   .equ DMA_BASE_ADDR,
    0x50000000
5   .equ GPIO_BASE_ADDR,
    0x50200000
6   .equ FPIOA_BASE_ADDR,
    0x502B0000
7   .equ SYSCTL_BASE_ADDR,
    0x50440000
8   .equ SPI0_BASE_ADDR,
    0x52000000
9   .equ SPI1_BASE_ADDR,
    0x53000000
10  .equ CLINT_BASE_ADDR,
    0x02000000
11  .equ PLIC_BASE_ADDR,
    0x0C000000
12
13  # 20bit For use with lui
14  .equ
    GPIOHS_BASE_ADDR_LUI,
    0x38001
15  .equ DMAC_BASE_ADDR_LUI,
    0x50000
16  .equ GPIO_BASE_ADDR_LUI,
    0x50200
17  .equ FPIOA_BASE_ADDR_LUI,
    0x502B0
18  .equ
    SYSCTL_BASE_ADDR_LUI,
    0x50440
19  .equ SPI0_BASE_ADDR_LUI,
    0x52000
20  .equ SPI1_BASE_ADDR_LUI,
    0x53000
21  .equ CLINT_BASE_ADDR_LUI,
    0x02000
22  .equ PLIC_BASE_ADDR_LUI,
    0x0C000
```

obj.inc

```
1   # Info for OBJ functions
2
3   # OBJ Settings
4   .equ OBJ_MAX_VERTICES,
    500
5   .equ OBJ_MAX_UVS, 500
6   .equ OBJ_MAX_NORMALS, 500
7   .equ OBJ_MAX_POLYGONS,
    500
```

pinmap.inc

```
1   # Pins utilized
2   .equ PIN_SD_MISO, 26
3   .equ PIN_SD_CLK, 27
4   .equ PIN_SD_MOSI, 28
5   .equ PIN_SD_CS, 29
6   .equ PIN_TFT_CS, 36
7   .equ PIN_TFT_RST, 37
8   .equ PIN_TFT_DC, 38
9   .equ PIN_TFT_CLK, 39
10  .equ PIN_LED_B, 12
11  .equ PIN_LED_G, 13
12  .equ PIN_LED_R, 14
13
14  # GPIOHS utilized
15  .equ GPIOHS_TFT_RST, 0
16  .equ GPIOHS_TFT_DC, 1
17  .equ GPIOHS_LED_B, 2
18  .equ GPIOHS_LED_G, 3
19  .equ GPIOHS_LED_R, 4
20
21  # SPI0 utilized
22  .equ SPI_TFT_SS, 3
23
24  # SPI1 utilized
25  .equ SPI_SD_SS, 3
26
27  # Pin start config
28  .equ PIN_SD_MISO_CONFIG,
    0x80B00747
29  .equ PIN_SD_CLK_CONFIG,
    0x00001753
30  .equ PIN_SD_MOSI_CONFIG,
    0x80003746
31  .equ PIN_SD_CS_CONFIG,
```

```
    0x00001751
32  .equ PIN_TFT_CS_CONFIG,
    0x0000170F
33  .equ PIN_TFT_RST_CONFIG,
    0x00021718
34  .equ PIN_TFT_DC_CONFIG,
    0x00021719
35  .equ PIN_TFT_CLK_CONFIG,
    0x00001711
36  .equ PIN_LED_B_CONFIG,
    0x0002171A
37  .equ PIN_LED_G_CONFIG,
    0x0002171B
38  .equ PIN_LED_R_CONFIG,
    0x0002171C
```

plic.inc

```
1   # Info for PLIC functions
2
3   # Offsets
4   .equ
    PLIC_SRC_PRIORITIES_OFFSET,
    0x000004
5   .equ PLIC_ENABLE_OFFSET,
    0x002000
6   .equ
    PLIC_TARGET_THRS_OFFSET,
    0x200000
7
8   # Constants
9   .equ PLIC_NUM_SOURCES, 65
```

ppm.inc

```
1   # Info for PPM functions
2
3   # PPM Settings
4   .equ PPM_MAX_WIDTH, 300
5   .equ PPM_MAX_HEIGHT, 300
6   .equ PPM_COLOR_DEPTH, 255
```

sd.inc

```
1   # Info for SD functions
2
3   # SD Card Settings
4
5   # SD Commands
6   .equ SD_CMD0, 0x40
7   .equ SD_CMD8, 0x48
8   .equ SD_CMD17, 0x51
9   .equ SD_CMD55, 0x77
10  .equ SD_CMD58, 0x7A
11  .equ SD_ACMD41, 0x69
12
13  # SD Commands Arguments
14  .equ SD_CMD8_ARG,
    0x000001CB
15  .equ SD_ACMD41_ARG,
    0x40000000
```

spi.inc

```
1   # Info for SPI functions
2
3   # SPI Settings
4   .equ SPI_WAIT_CYCL, 0
5   .equ SPI_WORK_MODE, 0
6   .equ SPI_ADDR_LEN, 0
7   .equ SPI_ENDIAN, 0
8   .equ SPI_TRANS_M, 1
9   .equ SPI_RECEV_M, 2
10
11  # SPI0 Settings
12  .equ SPI0_FRM_FMT, 3
13  .equ SPI0_BIT_LEN, 8 #
    Varies 8 or 16
14  .equ SPI0_INS_LEN, 2 #
    Varies 2 or 3
15  .equ SPI0_IATM, 2
16
17  # SPI1
18  .equ SPI1_FRM_FMT, 0
19  .equ SPI1_BIT_LEN, 8
20  .equ SPI1_INS_LEN, 0
21  .equ SPI1_IATM, 0
22
23  # Configuration offsets
24  .equ SPI_CTL0_OFF, 0x00
25  .equ SPI_CTL1_OFF, 0x04
26  .equ SPI_ENABLE_OFF, 0x08
27  .equ SPI_SLAVE_EN_OFF,
    0x10
28  .equ SPI_BAUD_RATE_OFF,
    0x14
29  .equ SPI_TFIFO_LVL_OFF,
    0x20
30  .equ SPI_RFIFO_LVL_OFF,
    0x24
31  .equ SPI_STATUS_OFF, 0x28
32  .equ SPI_INTR_MASK_OFF,
    0x2C
33  .equ SPI_DMA_CTL_OFF,
    0x4C
34  .equ SPI_DMA_TDL_OFF,
    0x50
35  .equ SPI_DMA_RDL_OFF,
    0x54
36  .equ SPI_DATA_FIFO_OFF,
    0x60
37  .equ SPI_SCTL0_OFF, 0xF4
38  .equ SPI_ENDIAN_OFF,
    0x118
39
40  # Region Mask
41  .equ SPI_CTL0_FF_MASK,
    0x0600000
42  .equ SPI_CTL0_DBL_MASK,
    0x01F0000
43  .equ SPI_CTL0_WC_MASK,
    0x000F800
44  .equ SPI_CTL0_IL_MASK,
    0x0000700
45  .equ SPI_CTL0_WM_MASK,
    0x00000C0
46  .equ SPI_CTL0_AL_MASK,
    0x000003C
47  .equ SPI_CTL0_ATM_MASK,
    0x0000003
48
49  #.equ ACLK_PLL_EN_MASK,
    0x00000001
50  #.equ ACLK_APB_SRC_MASK,
    0x00000FF8
51  #.equ
    PLL_OUTPUT_MASK_LUI,
    0x02000
52  #.equ PLL_POWER_MASK_LUI,
    0x00200
53  #.equ PLL_RESET_MASK_LUI,
    0x00100
54  #.equ PLL_FREQ_MASK,
    0x000FFFFF
55  #.equ CTHR_SPI0_MASK,
    0x000000FF
56  #.equ CTHR_SPI1_MASK,
    0x0000FF00
57
58  # APB Source
59  # Between PLL0[0],
    PLL1[1] or PLL2[2]
60  #.equ PLL_APB0_SRC, 0
61  #.equ PLL_APB1_SRC, 0
62  #.equ PLL_APB2_SRC, 0
63
64  # Enable Clock Buses
65  # 0 - CPU
66  # 1 - SRAM0
67  # 2 - SRAM1
68  # 3 - APB0
69  # 4 - APB1
70  # 5 - APB2
71  #.equ CLK_BUS_EN_REG,
    0x003F
72
73  # Enable Peripheral Clock
74  # 0 - ROM
75  # 1 - DMA
76  # 2 - AI
77  # 3 - DVP
78  # 4 - FFT
79  # 5 - GPIO
80  # 6 - SPI0
81  # 7 - SPI1
82  # 8 - SPI2
83  # 9 - SPI3 !It is used
    for the flash, may be
    important!
84  # 10 - I2S0
85  # 11 - I2S1
86  # 12 - I2S2
87  # 13 - I2C0
88  # 14 - I2C1
89  # 15 - I2C2
90  # 16 - UART1
91  # 17 - UART2
92  # 18 - UART3
93  # 19 - AES
94  # 20 - FPIOA
95  # 21 - Timer0
96  # 22 - Timer1
97  # 23 - Timer2
98  # 24 - WDT0
99  # 25 - WDT1
100 # 26 - SHA
101 # 27 - OTP
102 # 28 - Reserved
103 # 29 - RTC
104 # 30 - Reserved
105 # 31 - Reserved
106 #.equ CLK_PERI_EN_REG,
    0x001000E3
107
108 # [18:16] Pull
109 #.equ FPIOA_NP, 0x00000 #
    No Pull
110 #.equ FPIOA_PU, 0x10000 #
    Pull Up
111 #.equ FPIOA_PD, 0x20000 #
    Pull Down
112 #.equ FPIOA_SPU, 0x50000
    # Strong Pull up
113
114 # [13:12] Output
115 #.equ FPIOA_OI, 0x2000 #
    Output Invert
116 #.equ FPIOA_OE, 0x1000 #
```

```
    Output Enable
117
118 # [11:8] Driving output
    current
119 #.equ DS_0, 0x000
120 #.equ DS_1, 0x100
121 #.equ DS_2, 0x200
122 #.equ DS_3, 0x300
123 #.equ DS_4, 0x400
124 #.equ DS_5, 0x500
125 #.equ DS_6, 0x600
126 #.equ DS_7, 0x700
127
128 # [7:0] Function select
129 # GPIOHS
130 #.equ CH_GPIOHS0_SEL,
    0x18
131 #.equ CH_GPIOHS1_SEL,
    0x19
132 #.equ CH_GPIOHS2_SEL,
    0x1A
133 #.equ CH_GPIOHS3_SEL,
    0x1B
134 #.equ CH_GPIOHS4_SEL,
    0x1C
135 #.equ CH_GPIOHS5_SEL,
    0x1D
136 #.equ CH_GPIOHS6_SEL,
    0x1E
137 #.equ CH_GPIOHS7_SEL,
    0x1F
138 # GPIO
139 #.equ CH_GPIO0_SEL, 0x38
140 #.equ CH_GPIO1_SEL, 0x39
141 #.equ CH_GPIO2_SEL, 0x3A
142 #.equ CH_GPIO3_SEL, 0x3B
143 #.equ CH_GPIO4_SEL, 0x3C
144 #.equ CH_GPIO5_SEL, 0x3D
145 #.equ CH_GPIO6_SEL, 0x3E
146 #.equ CH_GPIO7_SEL, 0x3F
```

sysctl.inc

```
1   # Info for SYSCTL
    functions
2
3   # PLL frequencies
4   # 0xB4AD0 - 400 MHz
5   # 0xF47D0 - 800 MHz
6   .equ SYSCTL_PLL0_FREQ,
    0xB4AD0
7   .equ SYSCTL_SPI0_FDIV, 40
    # 40 - 10 MHz
8   .equ SYSCTL_SPI1_FDIV,
    2000 # 2000 - 100 kHz
9   .equ SYSCTL_SPI1_FDIV_HS,
    16 # 16 - 25 MHz
10
11  # Configuration offsets
12  .equ SYSCTL_PLL0_CTL,
    0x08
13  .equ SYSCTL_PLL_LOCK,
    0x18
14  .equ SYSCTL_ACLK_CTL,
    0x20
15  .equ SYSCTL_CBUS_EN,
    0x28
16  .equ SYSCTL_PERI_EN,
    0x2C
17  .equ SYSCTL_CTHR_SPI,
    0x3C
18  .equ SYSCTL_MISC_CTL,
    0x54
19  .equ SYSCTL_RST_SST,
    0x60
20  .equ SYSCTL_DMA_SEL,
    0x64
21
22  # Region Mask
23  .equ ACLK_PLL_EN_MASK,
    0x00000001
24  .equ ACLK_APB_SRC_MASK,
    0x00000FF8
25  .equ LOCK_PLL0_MASK,
    0x00000003
26  .equ LOCK_SLIP0_CLEAR,
    0x00000004
27  .equ PLL_OUTPUT_MASK_LUI,
    0x02000
28  .equ PLL_POWER_MASK_LUI,
    0x00200
29  .equ PLL_RESET_MASK_LUI,
    0x00100
30  .equ PLL_FREQ_MASK,
    0x000FFFFF
31  .equ CTHR_SPI0_MASK,
    0x000000FF
32  .equ CTHR_SPI1_MASK,
    0x0000FF00
33  .equ CBUS_APB0_MASK,
    0x00000008
34  .equ PERI_FPIOA_MASK,
    0x00100000
35  .equ SYSCTL_RST_CLR_MASK,
    0x00000001
36
37  # APB Source
38  # Between PLL0[0],
    PLL1[1] or PLL2[2]
39  .equ PLL_APB0_SRC, 0
40  .equ PLL_APB1_SRC, 0
41  .equ PLL_APB2_SRC, 0
42
43  # Enable Clock Buses
44  # 0 - CPU
45  # 1 - SRAM0
46  # 2 - SRAM1
47  # 3 - APB0
48  # 4 - APB1
49  # 5 - APB2
50  .equ CLK_BUS_EN_REG,
    0x003F
51
52  # Enable Peripheral Clock
53  # 0 - ROM
54  # 1 - DMA
55  # 2 - AI
56  # 3 - DVP
57  # 4 - FFT
58  # 5 - GPIO
59  # 6 - SPI0
60  # 7 - SPI1
61  # 8 - SPI2
62  # 9 - SPI3 !It is used
    for the flash, may be
    important!
63  # 10 - I2S0
64  # 11 - I2S1
65  # 12 - I2S2
66  # 13 - I2C0
67  # 14 - I2C1
68  # 15 - I2C2
69  # 16 - UART1
70  # 17 - UART2
71  # 18 - UART3
72  # 19 - AES
73  # 20 - FPIOA
74  # 21 - Timer0
75  # 22 - Timer1
76  # 23 - Timer2
77  # 24 - WDT0
78  # 25 - WDT1
79  # 26 - SHA
80  # 27 - OTP
81  # 28 - Reserved
82  # 29 - RTC
83  # 30 - Reserved
84  # 31 - Reserved
85  #.equ CLK_PERI_EN_REG,
    0x001000E3
86  .equ CLK_PERI_EN_REG,
    0xCCFFFFFF
```

tft.inc

```
1   # Info for TFT functions
2
3   # TFT Settings
4   .equ TFT_DMA_CH, 0x3
5   .equ TFT_DMA_SPI0_TX, 0x1
6   .equ TFT_SPI_DST_ADD,
    0x52000060
7
8   # TFT Commands
9   .equ NO_OPERATION, 0x00
10  .equ SOFTWARE_RESET, 0x01
11  .equ READ_ID, 0x04
12  .equ READ_STATUS, 0x09
13  .equ READ_POWER_MODE,
    0x0A
14  .equ READ_MADCTL, 0x0B
15  .equ READ_PIXEL_FORMAT,
    0x0C
16  .equ READ_IMAGE_FORMAT,
    0x0D
17  .equ READ_SIGNAL_MODE,
    0x0E
18  .equ
    READ_SELT_DIAG_RESULT,
    0x0F
19  .equ SLEEP_ON, 0x10
20  .equ SLEEP_OFF, 0x11
21  .equ PARTIAL_DISPLAY_ON,
    0x12
22  .equ NORMAL_DISPLAY_ON,
    0x13
23  .equ
    INVERSION_DISPLAY_OFF,
    0x20
24  .equ
    INVERSION_DISPLAY_ON,
    0x21
25  .equ GAMMA_SET, 0x26
26  .equ DISPLAY_OFF, 0x28
27  .equ DISPLAY_ON, 0x29
28  .equ
    HORIZONTAL_ADDRESS_SET,
    0x2A
29  .equ
    VERTICAL_ADDRESS_SET,
    0x2B
30  .equ MEMORY_WRITE, 0x2C
31  .equ COLOR_SET, 0x2D
32  .equ MEMORY_READ, 0x2E
```

```
33  .equ PARTIAL_AREA, 0x30
34  .equ
    VERTICAL_SCROL_DEFINE,
    0x33
35  .equ
    TEAR_EFFECT_LINE_OFF,
    0x34
36  .equ TEAR_EFFECT_LINE_ON,
    0x35
37  .equ MEMORY_ACCESS_CTL,
    0x36
38  .equ
    VERTICAL_SCROL_S_ADD,
    0x37
39  .equ IDLE_MODE_OFF, 0x38
40  .equ IDLE_MODE_ON, 0x39
41  .equ PIXEL_FORMAT_SET,
    0x3A
42  .equ
    WRITE_MEMORY_CONTINUE,
    0x3C
43  .equ
    READ_MEMORY_CONTINUE,
    0x3E
44  .equ SET_TEAR_SCANLINE,
    0x44
45  .equ GET_SCANLINE, 0x45
46  .equ WRITE_BRIGHTNESS,
    0x51
47  .equ READ_BRIGHTNESS,
    0x52
48  .equ WRITE_CTRL_DISPALY,
    0x53
49  .equ READ_CTRL_DISPALY,
    0x54
50  .equ
    WRITE_BRIGHTNESS_CTL,
    0x55
51  .equ READ_BRIGHTNESS_CTL,
    0x56
52  .equ
    WRITE_MIN_BRIGHTNESS,
    0x5E
53  .equ READ_MIN_BRIGHTNESS,
    0x5F
54  .equ READ_ID1, 0xDA
55  .equ READ_ID2, 0xDB
56  .equ READ_ID3, 0xDC
57  .equ RGB_IF_SIGNAL_CTL,
    0xB0
58  .equ NORMAL_FRAME_CTL,
    0xB1
59  .equ IDLE_FRAME_CTL, 0xB2
60  .equ PARTIAL_FRAME_CTL,
    0xB3
61  .equ INVERSION_CTL, 0xB4
62  .equ BLANK_PORCH_CTL,
    0xB5
63  .equ
    DISPALY_FUNCTION_CTL,
    0xB6
64  .equ ENTRY_MODE_SET, 0xB7
65  .equ BACKLIGHT_CTL1, 0xB8
66  .equ BACKLIGHT_CTL2, 0xB9
67  .equ BACKLIGHT_CTL3, 0xBA
68  .equ BACKLIGHT_CTL4, 0xBB
69  .equ BACKLIGHT_CTL5, 0xBC
70  .equ BACKLIGHT_CTL7, 0xBE
71  .equ BACKLIGHT_CTL8, 0xBF
72  .equ POWER_CTL1, 0xC0
73  .equ POWER_CTL2, 0xC1
74  .equ VCOM_CTL1, 0xC5
75  .equ VCOM_CTL2, 0xC7
76  .equ NV_MEMORY_WRITE,
    0xD0
77  .equ
    NV_MEMORY_PROTECT_KEY,
     0xD1
78  .equ
    NV_MEMORY_STATUS_READ,
    0xD2
79  .equ READ_ID4, 0xD3
80  .equ
    POSITIVE_GAMMA_CORRECT,
    0xE0
81  .equ
    NEGATIVE_GAMMA_CORRECT,
    0xE1
82  .equ DIGITAL_GAMMA_CTL1,
    0xE2
83  .equ DIGITAL_GAMMA_CTL2,
    0xE3
84  .equ INTERFACE_CTL, 0xF6
85
86  # Pixel Color Format
87  .equ PIXEL_565_16BIT,
    0x55
88  .equ PIXEL_666_18BIT,
    0x66
89
90  # Display Direction
91  .equ DIR_XY_RL_UD, 0x00
92  .equ DIR_YX_RL_UD, 0x20
93  .equ DIR_XY_LR_UD, 0x40
94  .equ DIR_YX_LR_UD, 0x60
95  .equ DIR_XY_RL_DU, 0x80
96  .equ DIR_YX_RL_DU, 0xA0
97  .equ DIR_XY_LR_DU, 0xC0
98  .equ DIR_YX_LR_DU, 0xE0
```

Anexos

ANEXO A – Instruções RISC-V

Neste anexo estão listadas as instruções RISC-V pertencentes ao conjunto base de inteiros de 64 bits, em junção com as extensões *MAFDZicsr_Zifencei_C*, ou *RV64GC* de forma compacta. Estas extensões são comumente utilizadas em conjunto, por isto estão aqui listadas. O detalhamento das instruções pode ser visto em Waterman et al. (2011).

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`	Calcula o *AND* bit a bit de `rs1` e `rs2` e armazena em `rd`
`andi`	`rd, rs1, imm`	Calcula o *AND* bit a bit de `rs1` e `imm` 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`
`bge`	`rs1, rs2, off`	Se `rs1` $\geq$ `rs2`, adiciona `off` ao `pc`
`bgeu`	`rs1, rs2, off`	Se `rs1` $\geq$ `rs2` sem o sinal, adiciona `off` ao `pc`
`bgez`	`rs1, off`	Converte para `bge rs1, x0, off`
`bgt`	`rs1, rs2, off`	Converte para `blt rs2, rs1, off`
`bgtu`	`rs1, rs2, off`	Converte para `bltu rs2, rs1, off`
`bgtz`	`rs2, off`	Converte para `blt x0, rs2, off`
`ble`	`rs1, rs2, off`	Converte para `bge rs2, rs1, off`
`bleu`	`rs1, rs2, off`	Converte para `bgeu rs2, rs1, off`
`blez`	`rs2, off`	Converte para `bge x0, rs2, off`
`blt`	`rs1, rs2, off`	Se `rs1` $<$ `rs2`, adiciona `off` ao `pc`
`bltu`	`rs1, rs2, off`	Se `rs1` $<$ `rs2` sem o sinal, adiciona `off` ao `pc`
`bltz`	`rs1, off`	Converte para `blt rs1, x0, off`
`bne`	`rs1, rs2, off`	Se `rs1` $\neq$ `rs2`, adiciona `off` ao `pc`
`bnez`	`rs1, off`	Converte para `bne rs1, x0, off`
`call`	`rd, sym`	Escreve o endereço da próxima instrução para `rd` e armazena `sym` ao `pc`. Converte para `auipc rd, offsetHi` e depois `jalr rd, offsetLo(rd)`
`ebreak`		Levanta uma exceção de *breakpoint* para o *debugger*
`ecall`		Realiza solicitação de execução ao ambiente de suporte
`fence`	`pred, succ`	Processa acessos a memória em haver para sincronia de *threads*, garantindo que o mesmo valor seja observado
`j`	`off`	Converte para `jal x0, off`
`jal`	`rd, off`	Copia o endereço da próxima instrução para `rd` e acrescenta `off` ao `pc`
`jalr`	`rd, off(rs1)`	Copia o endereço da próxima instrução para `rd` e acrescenta `rs1+off` com o LSB significativo omitido ao `pc`
`jr`	`rs1`	Converte para `jalr x0, 0(rs1)`
`la`	`rd, sym`	Copia o endereço de `sym`, presente na *global offset table*, e o armazena em `rd`. Converte para `auipc rd, offsetHi` e `[lw ou ld] rd, offsetLo(rd)`

RV32I

Inst	Parâmetros	Descrição
lb	rd, off(rs1)	Copia um byte da memória rs1+off e o armazena em rd
lbu	rd, off(rs1)	Copia um byte sem sinal da memória rs1+off em rd
lh	rd, off(rs1)	Copia 2 bytes da memória rs1+off e o armazena em rd
lhu	rd, off(rs1)	Copia 2 bytes sem sinal da memória rs1+off em rd
li	rd, imm	Converte para lui(s) e/ou addi(s), dependendo do tamanho de imm
lla	rd, sym	Copia o endereço local de sym, e o armazena em rd. Converte para auipc rd, offsetHi e addi rd, rd, offsetLo
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
neg	rd, rs2	Converte para sub rd, x0, rs2
nop		Converte para addi x0, x0, 0
not	rd, rs1	Converte para xori rd, rs1, -1
or	rd, rs1, imm	Calcula o *OR* bit a bit de rs1 e rs2 e armazena em rd
ori	rd, rs1, imm	Calcula o *OR* bit a bit de rs1 e imm e armazena em rd
ret		Converte para jalr x0, 0(x1)
sb	rs2, off(rs1)	Armazena um byte de rs2 na memória rs1+off
seqz	rd, rs1	Converte para sltiu rd, rs1, 1
sgtz	rd, rs2	Converte para slt rd, x0, rs2
sh	rs2, off(rs1)	Armazena 2 bytes 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
slli	rd, rs1, imm	Desloca rs1 imm bits à esquerda e armazena em rd
slt	rd, rs1, rs2	Se rs1 $<$ rs2, armazena 1 em rd, e 0 caso contrário
slti	rd, rs1, imm	Se rs1 $<$ imm, armazena 1 em rd, e 0 caso contrário
sltiu	rd, rs1, imm	Se rs1 $<$ imm sem sinal, armazena 1 em rd, e 0 caso contrário
sltu	rd, rs1, rs2	Se rs1 $<$ rs2 sem sinal, armazena 1 em rd, e 0 caso contrário
sltz	rd, rs1	Converte para slt rd, rs1, x0
slnez	rd, rs2	Converte para sltu rd, x0, rs2
sra	rd, rs1, rs2	Desloca rs1 rs2 bits à direita, preenchendo o espaço vazio com o MSB de rs1, e armazena em rd
srai	rd, rs1, imm	Desloca rs1 imm bits à direita, preenchendo o espaço vazio com o MSB de rs1, e armazena em rd
srl	rd, rs1, rs2	Desloca rs1 rs2 bits à direita e armazena em rd
srli	rd, rs1, rs2	Desloca rs1 rs2 bits à direita e armazena em rd
sub	rd, rs1, rs2	Subtrai rs2 de rs1 e armazena em rd
tail	sym	Armazena sym no pc. Converte para auipc x6, offsetHi e jalr x0, offsetLo(x6)
wfi		Mantém o processador em um baixo consumo de energia enquanto espera por interrupções
xor	rd, rs1, rs2	Calcula o *XOR* bit a bit de rs1 e rs2 e armazena em rd
xori	rd, rs1, imm	Calcula o *XOR* bit a bit de rs1 e imm 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

RV64I

Inst	Parâmetros	Descrição
`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`
`lwu`	`rd, off(rs1)`	Copia 4 bytes sem sinal da memória `rs1+off` em `rd`
`sd`	`rs2, off(rs1)`	Armazena 8 bytes de `rs2` na memória `rs1+off`
`sext.w`	`rsd, rs1`	Converte para `addiw rd, rs1, 0`
`slliw`	`rd, rs1, imm`	Desloca `rs1` `imm` bits à esquerda, trunca em 32 bits, para `rd`
`sllw`	`rd, rs1, rs2`	Desloca `rs1` `rs2` bits à esquerda, trunca em 32 bits, para `rd`
`sraiw`	`rd, rs1, imm`	Desloca os 32 LSBs de `rs1` `imm` bits à direita, preenchendo o espaço vazio com o bit 31 de `rs1`, para `rd`
`sraw`	`rd, rs1, rs2`	Desloca os 32 LSBs de `rs1` `rs2` bits à direita, preenchendo o espaço vazio com o bit 31 de `rs1`, e armazena em `rd`
`srliw`	`rd, rs1, imm`	Desloca `rs1` `rs2` bits à direita, trunca em 32 bits, para `rd`
`srlw`	`rd, rs1, rs2`	Desloca `rs1` `rs2` bits à direita, trunca em 32 bits, para `rd`
`subw`	`rd, rs1, rs2`	Subtrai `rs2` de `rs1`, trunca em 32 bits, para `rd`
	M	
`div`	`rd, rs1, rs2`	Divide `rs1` por `rs2`, arredondando para 0, para `rd`
`divu`	`rd, rs1, rs2`	Divide `rs1` por `rs2` sem sinal, arredondando para 0, para `rd`
`divuw`	`rd, rs1, rs2`	Divide os 32 LSBs de `rs1` pelos 32 LSBs de `rs2` sem sinal, arredonda para 0, trunca em 32 bits, para `rd`
`divw`	`rd, rs1, rs2`	Divide os 32 LSBs de `rs1` pelos 32 LSBs de `rs2`, arredonda para 0, trunca em 32 bits, para `rd`
`mul`	`rd, rs1, rs2`	Multiplica `rs1` por `rs2` e armazena em `rd`
`mulh`	`rd, rs1, rs2`	Multiplica `rs1` por `rs2` e armazena a metade superior do resultado em `rd`
`mulhsu`	`rd, rs1, rs2`	Multiplica `rs1` com sinal por `rs2` sem sinal e armazena a metade superior do resultado em `rd`
`mulhu`	`rd, rs1, rs2`	Multiplica `rs1` por `rs2` sem sinal e armazena a metade superior do resultado em `rd`
`mulw`	`rd, rs1, rs2`	Multiplica `rs1` por `rs2`, trunca em 32 bits, para `rd`
`rem`	`rd, rs1, rs2`	Divide `rs1` por `rs2`, e armazena o resto em `rd`
`remu`	`rd, rs1, rs2`	Divide `rs1` por `rs2` sem sinal, e armazena o resto em `rd`
`remuw`	`rd, rs1, rs2`	Divide os 32 LSBs de `rs1` pelos 32 LSBs de `rs2` sem sinal, arredondando para 0, trunca em 32 bits, com o resto para `rd`
`remw`	`rd, rs1, rs2`	Divide os 32 LSBs de `rs1` pelos 32 LSBs de `rs2`, arredonda para 0, trunca em 32 bits, com o resto para `rd`
	A	
`amoadd.d`	`rd,rs2,(rs1)`	Adiciona `rs2` ao valor de 8 bytes em `rs1` para `rd`
`amoadd.w`	`rd,rs2,(rs1)`	Adiciona `rs2` ao valor de 4 bytes em `rs1` para `rd`
`amoand.d`	`rd,rs2,(rs1)`	Calcula o *AND* bit a bit de `rs2` ao valor de 8 bytes em `rs1` e armazena em `rd`
`amoand.w`	`rd,rs2,(rs1)`	Calcula o *AND* bit a bit de `rs2` ao valor de 4 bytes em `rs1` e armazena em `rd`
`amomax.d`	`rd,rs2,(rs1)`	Copia o maior de `rs2` ou o valor de 8 bytes em `rs1` em `rd`
`amomax.w`	`rd,rs2,(rs1)`	Copia o maior de `rs2` ou o valor de 4 bytes em `rs1` em `rd`
`amomaxu.d`	`rd,rs2,(rs1)`	Copia o maior de `rs2` ou o valor de 8 bytes em `rs1` sem sinal em `rd`

A		
Inst	**Parâmetros**	**Descrição**
amomaxu.w	rd,rs2,(rs1)	Copia o maior de rs2 ou o valor de 4 bytes em rs1 sem sinal em rd
amomin.d	rd,rs2,(rs1)	Copia o menor de rs2 ou o valor de 8 bytes em rs1 em rd
amomin.w	rd,rs2,(rs1)	Copia o menor de rs2 ou o valor de 4 bytes em rs1 em rd
amominu.d	rd,rs2,(rs1)	Copia o menor de rs2 ou o valor de 8 bytes em rs1 sem sinal em rd
amominu.w	rd,rs2,(rs1)	Copia o menor de rs2 ou o valor de 4 bytes em rs1 sem sinal em rd
amoor.d	rd,rs2,(rs1)	Calcula o *OR* bit a bit de rs2 ao valor de 8 bytes em rs1 e armazena em rd
amoor.w	rd,rs2,(rs1)	Calcula o *OR* bit a bit de rs2 ao valor de 4 bytes em rs1 e armazena em rd
amoswap.d	rd,rs2,(rs1)	Copia o valor de 8 bytes em rs1 para rs2 e rd
amoswap.w	rd,rs2,(rs1)	Copia o valor de 4 bytes em rs1 para rs2 e rd
amoxor.d	rd,rs2,(rs1)	Calcula o *XOR* bit a bit de rs2 ao valor de 8 bytes em rs1 e armazena em rd
amoxor.w	rd,rs2,(rs1)	Calcula o *XOR* bit a bit de rs2 ao valor de 4 bytes em rs1 e armazena em rd
lr.d	rd,(rs1)	Copia os 8 bytes de rs1 para rd e reserva este endereço
lr.w	rd,(rs1)	Copia os 4 bytes de rs1 para rd e reserva este endereço
sc.d	rd,rs2,(rs1)	Armazena 8 bytes de rs2 em rs1 caso tenha uma reserva, retorna 0 em rd em êxito
sc.w	rd,rs2,(rs1)	Armazena 4 bytes de rs2 em rs1 caso tenha uma reserva, retorna 0 em rd em êxito
F		
fabs.s	rd, rs1	Converte para fsgnjx.s rd, rs1, rs1
fadd.s	rd, rs1, rs2	Adiciona rs1 a rs2 e armazena em rd
fclass.s	rd, rs1	Indica a classe do número flutuante simples rs1 em rd
fcvt.l.s	rd, rs1	Converte o número flutuante simples em rs1 para o inteiro de 8 bytes em rd
fcvt.lu.s	rd, rs1	Converte o número flutuante simples em rs1 para o inteiro de 8 bytes sem sinal em rd
fcvt.s.l	rd, rs1	Converte o número inteiro de 8 bytes em rs1 para o flutuante simples em rd
fcvt.s.lu	rd, rs1	Converte o número inteiro de 8 bytes sem sinal em rs1 para o flutuante simples em rd
fcvt.s.w	rd, rs1	Converte o número inteiro de 4 bytes em rs1 para o flutuante simples em rd
fcvt.s.wu	rd, rs1	Converte o número inteiro de 4 bytes sem sinal em rs1 para o flutuante simples em rd
fcvt.w.s	rd, rs1	Converte o número flutuante simples em rs1 para o inteiro de 4 bytes em rd
fcvt.wu.s	rd, rs1	Converte o número flutuante simples em rs1 para o inteiro de 4 bytes sem sinal em rd
fdiv.s	rd, rs1, rs2	Divide rs1 por rs2 e armazena em rd
feq.s	rd, rs1, rs2	Se rs1 = rs2, armazena 1 em rd, e 0 caso contrário

F		
Inst	**Parâmetros**	**Descrição**
fle.s	rd, rs1, rs2	Se rs1 $\leq$ rs2, armazena 1 em rd, e 0 caso contrário
flt.s	rd, rs1, rs2	Se rs1 $<$ rs2, armazena 1 em rd, e 0 caso contrário
flw	rd, off(rs1)	Copia 4 bytes da memória rs1+off e armazena em rd
fmadd.s	rd,rs1,rs2,rs3	Multiplica rs1 e rs2 e soma com rs3 para rd
fmax.s	rd, rs2, rs1	Copia o maior de rs1 ou rs2 em rs1 em rd
fmin.s	rd, rs2, rs1	Copia o menor de rs1 ou rs2 em rs1 em rd
fmsub.s	rd,rs1,rs2,rs3	Multiplica rs1 e rs2 e subtrai rs3 para rd
fmul.s	rd, rs1, rs2	Multiplica rs1 por rs2 e armazena em rd
fmv.s	rd, rs1	Converte para fsgnj.s rd, rs1, rs1
fmv.w.x	rd, rs1	Move o valor binário de 4 bytes de rs1 para o de ponto flutuante rd
fmv.x.w	rd, rs1	Move o valor binário de ponto flutuante de rs1 para o de 4 bytes de rd
fneg.s	rd, rs1	Converte para fsgnjn.s rd, rs1, rs1
fnmadd.s	rd,rs1,rs2,rs3	Multiplica -rs1 e rs2 e subtrai rs3 para rd
fnmsub.s	rd,rs1,rs2,rs3	Multiplica -rs1 e rs2 e soma com rs3 para rd
frcsr	rd	Converte para csrrs rd, fcsr, x0
frflags	rd	Converte para csrrs rd, fflags, x0
frrm	rd	Converte para csrrs rd, frm, x0
fscsr	rd	Converte para csrrw rd, fcsr, x0
fsflags	rd	Converte para csrrw rd, fflags, x0
fsgnj.s	rd, rs1, rs2	Cria um valor de ponto flutuante com o expoente e significando de rs1 e o sinal de rs2 para rd
fsgnjn.s	rd, rs1, rs2	Cria um valor de ponto flutuante com o expoente e significando de rs1 e o sinal oposto de rs2 para rd
fsgnjx.s	rd, rs1, rs2	Cria um valor de ponto flutuante com o expoente e significando de rs1 e o sinal *XOR* de rs1 rs2 para rd
fsqrt.s	rd, rs1	Calcula a raiz quadrada de rs1 e armazena em rd
fsrm	rd	Converte para csrrw rd, frm, x0
fsub.s	rd, rs1, rs2	Subtrai rs2 de rs1 e armazena em rd
fsw	rs2, off(rs1)	Armazena o ponto flutuante simples rs2 na memória rs1+off
D		
fabs.d	rd, rs1	Converte para fsgnjx.d rd, rs1, rs1
fadd.d	rd, rs1, rs2	Adiciona rs1 a rs2 e armazena em rd
fclass.d	rd, rs1	Indica a classe do número flutuante duplo rs1 em rd
fcvt.d.l	rd, rs1	Converte o número inteiro de 8 bytes em rs1 para o flutuante duplo em rd
fcvt.d.lu	rd, rs1	Converte o número inteiro de 8 bytes sem sinal em rs1 para o flutuante duplo em rd
fcvt.d.w	rd, rs1	Converte o número inteiro de 4 bytes em rs1 para o flutuante duplo em rd
fcvt.d.wu	rd, rs1	Converte o número inteiro de 4 bytes sem sinal em rs1 para o flutuante duplo em rd
fcvt.l.d	rd, rs1	Converte o número flutuante duplo em rs1 para o inteiro de 8 bytes em rd

D		
Inst	**Parâmetros**	**Descrição**
fcvt.lu.d	rd, rs1	Converte o número flutuante duplo em rs1 para o inteiro de 8 bytes sem sinal em rd
fcvt.w.d	rd, rs1	Converte o número flutuante duplo em rs1 para o inteiro de 4 bytes em rd
fcvt.wu.d	rd, rs1	Converte o número flutuante duplo em rs1 para o inteiro de 4 bytes sem sinal em rd
fdiv.d	rd, rs1, rs2	Divide rs1 por rs2 e armazena em rd
feq.d	rd, rs1, rs2	Se rs1 = rs2, armazena 1 em rd, e 0 caso contrário
fld	rd, off(rs1)	Copia 8 bytes da memória rs1+off e armazena em rd
fle.d	rd, rs1, rs2	Se rs1 $\leq$ rs2, armazena 1 em rd, e 0 caso contrário
flt.d	rd, rs1, rs2	Se rs1 $<$ rs2, armazena 1 em rd, e 0 caso contrário
fmadd.d	rd,rs1,rs2,rs3	Multiplica rs1 e rs2 e soma com rs3 para rd
fmax.d	rd, rs2, rs1	Copia o maior de rs1 ou rs2 em rs1 em rd
fmin.d	rd, rs2, rs1	Copia o menor de rs1 ou rs2 em rs1 em rd
fmsub.d	rd,rs1,rs2,rs3	Multiplica rs1 e rs2 e subtrai rs3 para rd
fmul.d	rd, rs1, rs2	Multiplica rs1 por rs2 e armazena em rd
fmv.d	rd, rs1	Converte para fsgnj.d rd, rs1, rs1
fmv.d.w	rd, rs1	Move o valor binário de ponto flutuante de rs1 para o de 4 bytes de rd
fmv.w.d	rd, rs1	Move o valor binário de 4 bytes de rs1 para o de ponto flutuante rd
fneg.d	rd, rs1	Converte para fsgnjn.d rd, rs1, rs1
fnmadd.d	rd,rs1,rs2,rs3	Multiplica -rs1 e rs2 e subtrai rs3 para rd
fnmsub.d	rd,rs1,rs2,rs3	Multiplica -rs1 e rs2 e soma com rs3 para rd
fsd	rs2, off(rs1)	Armazena o ponto flutuante duplo rs2 na memória rs1+off
fsgnj.d	rd, rs1, rs2	Cria um valor de ponto flutuante com o expoente e significando de rs1 e o sinal de rs2 para rd
fsgnjn.d	rd, rs1, rs2	Cria um valor de ponto flutuante com o expoente e significando de rs1 e o sinal oposto de rs2 para rd
fsgnjx.d	rd, rs1, rs2	Cria um valor de ponto flutuante com o expoente e significando de rs1 e o sinal *XOR* de rs1 rs2 para rd
fsqrt.d	rd, rs1	Calcula a raíz quadrada de rs1 e armazena em rd
fsub.d	rd, rs1, rs2	Subtrai rs2 de rs1 e armazena em rd
Z_{icsr}		
csrc	csr, rs1	Converte para csrrc x0, csr, rs1
csrr	rd, csr	Converte para csrrs rd, csr, x0
csrci	csr, imm	Converte para csrrci x0, csr, imm
csrcc	rd, csr, rs1	Copia o registrador csr para rd, e escreve o *AND* bit a bit do registrador csr e o complemento de rs1 em csr
csrrci	rd, csr, imm	Copia o registrador csr para rd, e escreve o *AND* bit a bit do registrador csr e o complemento de imm em csr
csrrs	rd, csr, rs1	Copia o registrador csr para rd, e escreve o *OR* bit a bit do registrador csr e rs1 em csr
csrrsi	rd, csr, imm	Copia o registrador csr para rd, e escreve o *OR* bit a bit do registrador csr e imm em csr

Inst	Parâmetros	Descrição
	Z_{icsr}	
csrrw	rd, csr, rs1	Copia o registrador csr para rd, e escreve rs1 em csr
csrrwi	rd, csr, imm	Copia o registrador csr para rd, e escreve imm em csr
csrs	csr, rs1	Converte para csrrs x0, csr, rs1
csrsi	csr, imm	Converte para csrrsi x0, csr, imm
csrw	csr, rs1	Converte para csrrw x0, csr, rs1
csrwi	csr, imm	Converte para csrrwi x0, csr, imm
mret		Retorno de exceção em modo de *Machine*
rdcycle	rd	Converte para csrrs rd, cycle, x0
rdcycleh	rd	Converte para csrrs rd, cycleh, x0
rdinstret	rd	Converte para csrrs rd, instret, x0
rdinstreth	rd	Converte para csrrs rd, instreth, x0
rdtime	rd	Converte para csrrs rd, time, x0
rdtimeh	rd	Converte para csrrs rd, timeh, x0
sret		Retorno de exceção em modo de *Supervisor*
	$Z_{ifencei}$	
fence.i		Utilizada para a sincronia de instruções e fluxo de dados, aguardando e realizando operações de armazenamento previamente requisitadas
	C	
c.add	rd, rs2	Equivalente a add rd, rd, rs2
c.addi	rd, imm	Equivalente a addi rd, rd, imm
c.addi16sp	imm	Equivalente a addi x2, x2, imm$\ll$4
c.addi4spn	rd', imm	Equivalente a addi rd'+8, x2, imm$\ll$2
c.addiw	rd, imm	Equivalente a addiw rd, rd, imm
c.and	rd', rs2'	Equivalente a and rd'+8, rd'+8, rs2'+8
c.addw	rd', rs2'	Equivalente a add rd'+8, rd'+8, rs2'+8
c.andi	rd', imm	Equivalente a andi rd'+8, rd'+8, imm
c.beqz	rs1', off	Equivalente a beq rs1'+8, x0, off
c.bnez	rs1', off	Equivalente a bne rs1'+8, x0, off
c.ebreak		Equivalente a ebreak
c.fld	rd',imm(rs1')	Equivalente a fld rd'+8, imm(rs1'+8)
c.fldsp	rd',imm(x2)	Equivalente a fld rd'+8, imm(x2)
c.flw	rd',imm(rs1')	Equivalente a flw rd'+8, imm(rs1'+8)
c.flwsp	rd',imm(x2)	Equivalente a flw rd'+8, imm(x2)
c.fsd	rs2',imm(rs1')	Equivalente a fsd rs2'+8, imm(rs1'+8)
c.fsdsp	rs2',imm(x2)	Equivalente a fsd rs2'+8, imm(x2)
c.fsw	rs2',imm(rs1')	Equivalente a fwd rs2'+8, imm(rs1'+8)
c.fswsp	rs2',imm(x2)	Equivalente a fwd rs2'+8, imm(x2)
c.j	off	Equivalente a jal x0, off
c.jal	off	Equivalente a jal x1, off
c.jalr	rs1	Equivalente a jalr x1, 0(rs1)
c.jr	rs1	Equivalente a jr x0, 0(rs1)
c.ld	rd',imm(rs1')	Equivalente a ld rd'+8, imm(rs1'+8)
c.ldsp	rd',imm(x2)	Equivalente a ld rd'+8, imm(x2)
c.li	rd, imm	Equivalente a addi rd, x0, imm
c.lui	rd, imm	Equivalente a lui rd, imm

C

Inst	Parâmetros	Descrição
c.lw	rd',imm(rs1')	Equivalente a lw rd'+8, imm(rs1'+8)
c.lwsp	rd',imm(x2)	Equivalente a lw rd'+8, imm(x2)
c.mv	rd, rs2	Equivalente a add x0, rs2
c.or	rd', rs2'	Equivalente a or rd'+8, rd'+8, rs2'+8
c.sd	rs2',imm(rs1')	Equivalente a sd rs2'+8, imm(rs1'+8)
c.sdsp	rs2',imm(x2)	Equivalente a sd rs2'+8, imm(x2)
c.slli	rd, imm	Equivalente a slli rd, rd, imm
c.srai	rd', imm	Equivalente a srai rd'+8, rd'+8, imm
c.srli	rd', imm	Equivalente a srli rd'+8, rd'+8, imm
c.sub	rd', rs2'	Equivalente a sub rd'+8, rd'+8, rs2'+8
c.subw	rd', rs2'	Equivalente a subw rd'+8, rd'+8, rs2'+8
c.sw	rs2',imm(rs1')	Equivalente a sw rs2'+8, imm(rs1'+8)
c.swsp	rs2',imm(x2)	Equivalente a sw rs2'+8, imm(x2)
c.xor	rd', rs2'	Equivalente a xor rd'+8, rd'+8, rs2'+8

ANEXO B – Diretivos assembly

Neste anexo estão listados os diretivos utilizáveis na programação assembly do RISC-V. Diretivos funcionam como indicadores para a construção do código de máquina, fornecendo funções de intermediação e simplificação. Alguns diretivos representam pseudo operações que serão removidas durante a montagem do código. As informações detalhadas sobre os diretivos pode ser encontrada em Shakti (2020).

Diretivo	Parâmetros	Descrição
`.2byte`	`val1,val2,...`	Define um conjunto de valores de 16 bits não alinhados
`.4byte`	`val1,val2,...`	Define um conjunto de valores de 32 bits não alinhados
`.8byte`	`val1,val2,...`	Define um conjunto de valores de 64 bits não alinhados
`.align`	`size`	Alinha a próxima instrução para $2^{\texttt{size}}$ bytes
`.ascii`	`"texto"`	Define um texto ASCII
`.asciz`	`"texto"`	Define um texto ASCII seguido de um carácter nulo
`.balign`	`size`	Alinha a próxima instrução para `size` bytes
`.bss`	`sym,len,ali`	Seção para dados não iniciados, de símbolo `sym`, a serem alocados em execução, de `len` bytes alinhados em $2^{\texttt{ali}}$
`.byte`	`val1,val2,...`	Define um conjunto de valores de 8 bits não alinhados
`.comm`	`sym, len`	Define um símbolo `sym` a ser alocado `len` bytes em `.bss`
`.common`	`sym, len`	Equivalente a `.comm`
`.data`		Seção para dados *read-write*
`.equ`	`sym, expr`	Define o valor do símbolo `sym` para a expressão `expr`
`.double`	`val1,val2,...`	Define um conjunto de flutuantes duplos alinhados
`.dword`	`val1,val2,...`	Define um conjunto de valores de 64 bits alinhados
`.file`	`nome`	Inicia um arquivo lógico
`.float`	`val1,val2,...`	Define um conjunto de flutuantes simples alinhados
`.global`	`sym`	Define um símbolo global a alocar na *global offset table*
`.globl`	`sym`	Equivalente a `.global`
`.half`	`val1,val2,...`	Define um conjunto de valores de 16 bits alinhados
`.incbin`	`"file"`	Incluí o arquivo `file` como sequência binária
`.include`	`"file"`	Incluí o arquivo `file` prévio a compilação
`.indent`	`"texto"`	Utilizado para adicionar *tags* a um arquivo compilado
`.local`	`sym`	Define um símbolo visível localmente
`.option`	`arg`	Define uma opção do RISC-V
`.p2align`	`size`	Equivalente a `align`
`.rodata`		Seção para dados *read-only*
`.section`	`nome`	Define início da seção `nome`
`.size`	`sym, sym`	Define o tamanho de um símbolo. Gerado pelo compilador
`.string`	`"texto"`	Equivalente a `.asciz`
`.text`		Seção para código. Gera uma exceção caso dados tentem ser armazenados nela
`.type`	`sym, tipo`	Define um símbolo para o tipo `tipo`
`.word`	`val1,val2,...`	Define um conjunto de valores de 32 bits alinhados
`.zero`	`len`	Define um conjunto de zeros de `len` bytes