UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

DESENVOLVIMENTO E IMPLEMENTAÇÃO DE FONTES DE TEXTO DE ALTA DEFINIÇÃO PARA O HARDWARE ODROID-SHOW

TRABALHO DE GRADUAÇÃO

Guilherme Recchi Cardozo

Santa Maria, RS, Brasil

2015

DESENVOLVIMENTO E IMPLEMENTAÇÃO DE FONTES DE TEXTO DE ALTA DEFINIÇÃO PARA O HARDWARE ODROID-SHOW

Guilherme Recchi Cardozo

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de

Bacharel em Ciência da Computação

Orientador: Prof. Dr. Cesar Tadeu Pozzer

403
Santa Maria, RS, Brasil

2015

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

DESENVOLVIMENTO E IMPLEMENTAÇÃO DE FONTES DE TEXTO DE ALTA DEFINIÇÃO PARA O HARDWARE ODROID-SHOW

elaborado por
Guilherme Recchi Cardozo

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Cesar Tadeu Pozzer, Prof. Dr.
(Presidente/Orientador)

Giovani Rubert Librelotto, Prof. Dr. (UFSM)

Osmar Marchi dos Santos, Prof. Dr. (UFSM)

Santa Maria, 9 de Dezembro de 2015.

AGRADECIMENTOS

Agradeço à minha família pelo suporte oferecido ao longo de minha vida.
Aos novos e velhos amigos pelas experiências - boas e ruins - compartilhadas.
Ao professor Cesar Pozzer pela orientação e a todos os demais professores do curso.

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

DESENVOLVIMENTO E IMPLEMENTAÇÃO DE FONTES DE TEXTO DE ALTA DEFINIÇÃO PARA O HARDWARE ODROID-SHOW

AUTOR: GUILHERME RECCHI CARDOZO
ORIENTADOR: CESAR TADEU POZZER
Local da Defesa e Data: Santa Maria, 9 de Dezembro de 2015.

Odroid-Show é um hardware compatível com Arduino que possui uma tela LCD de 2,2 polegadas e um microcontrolador de 8 bits. A fonte de texto disponível nesse dispositivo torna-se menos legível a medida que seu tamanho aumenta. Para tratar esse problema, o presente trabalho propõe a criação de fontes para serem utilizadas na tela LCD do mesmo. Tal otimização poderá ser aproveitada em diversas aplicações que necessitam de elementos textuais, dentre essas, o presente projeto visa desenvolver tecnologia que será utilizada para a modernização de um DSET (Dispositivos de Simulação de Engajamento Tático). Os resultados obtidos mostram fontes mais legíveis e personalizáveis, abrindo margem para trabalhos futuros que deem continuidade a essa pesquisa.

Palavras-chave: Sistemas Embarcados. Fontes Truetype. Fontes Bitmap.

ABSTRACT

Undergraduate Final Work
Undergraduate Program in Computer Science
Federal University of Santa Maria

DESENVOLVIMENTO E IMPLEMENTAÇÃO DE FONTES DE TEXTO DE ALTA DEFINIÇÃO PARA O HARDWARE ODROID-SHOW

AUTHOR: GUILHERME RECCHI CARDOZO
ADVISOR: CESAR TADEU POZZER
Defense Place and Date: Santa Maria, December 9th, 2015.

The Odroid-Show is an Arduino compatible hardware which has a 2.2 inches LCD screen and an 8-bit microcontroller unit. The text font available on that device becomes less readable as its size increases. To address this problem, this thesis proposes the creation of fonts to be used on this hardware. Such optimization can be used in several applications that require textual elements, among these, this project aims to develop technology that will be used for the modernization of a TESS (Tactical Engagement Simulation System). The results show more readable and customizable fonts, making room for future work, giving continuity on this research.

Keywords: Embedded Systems. TrueType Fonts. Bitmap Fonts.

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

Atualmente existem diversos sistemas embarcados baseados em Arduino. Tais hardwares são utilizados no desenvolvimento de diversos de projetos, desde objetos de uso diário até complexos instrumentos científicos (Arduino, 2015).

Alguns dispositivos baseados em Arduino possuem tela LCD, como é o caso do Odroid-Show. No entanto, devido às limitações de memória, o Odroid-Show dispõe de um sistema de fonte de texto de baixa resolução (ver figura 1.1), o qual não modifica o número - mas sim o tamanho - dos pixels, baseando-se no tamanho especificado da fonte, resultando assim na apresentação de textos de baixa legibilidade. Apenas uma opção de fonte é oferecida pelo dispositivo, impossibilitando a personalização do texto. É então apropriado desenvolver um novo sistema de fontes de alta definição para o Odroid-Show, permitindo a escolha de diferentes tipos, tamanhos e estilos de fonte.

O Exército Brasileiro emprega sistemas embarcados como parte de projetos de adestramento de tropas no Centro de Avaliação de Adestramento do Exército (CAAdEX). Essa Organização Militar faz uso de 42 Dispositivos de Simulação de Engajamento Tático (DSET) para os Carros de Combate Leopard 1A5, os quais possuem impressoras para apresentar os dados das simulações (Blog do Exército Brasileiro, 2015). No projeto de modernização dos DSETs, cada impressora será substituída por um Odroid-Show e os dados das simulações serão exibidos em sua tela LCD, implicando em necessidade por um sistema de fontes mais legíveis e personalizáveis.

O Odroid-Show apresenta severas limitações de memória e processamento (ver subseção 3.1.2). Logo, é necessário que o novo sistema de fontes seja implementado de forma a suprir tais limitações. Para suprir tal necessidade pode-se fazer uso de compressão de dados sobre as fontes geradas.

O presente trabalho será utilizado no projeto DSET. No entanto, a otimização do sistema de fonte do Odroid-Show é útil para quaisquer aplicações que necessitem exibir texto na tela do mesmo.

1.1 Objetivos do Trabalho

Desenvolver e implementar um formato de arquivo de fontes de texto, baseando-se em caracteres representados por bitmaps, para utilização no hardware Odroid-Show. Devido às

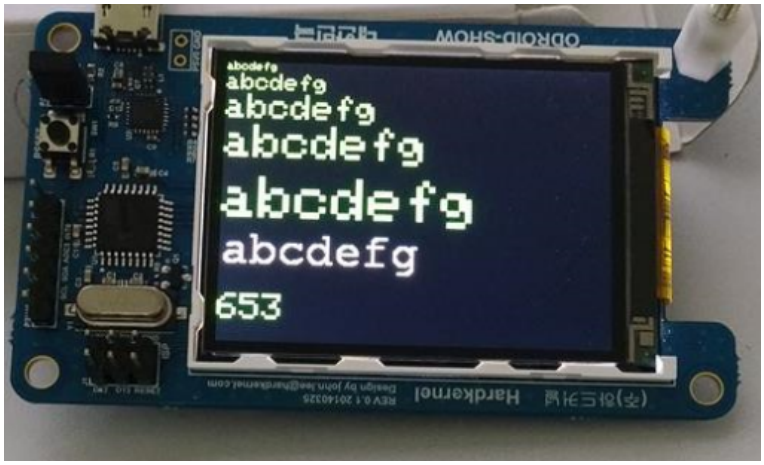

Figura 1.1 – Fonte original do Odroid-Show em diversos tamanhos. Em branco, o teste de criação de uma nova fonte.

grandes limitações de memória e processamento, o modelo de fonte deverá ser suficientemente otimizado para que a mesma possa ser enviada ao dispositivo via comunicação serial ou armazenada no mesmo.

1.2 Contextualização e Motivação

O Odroid-Show é um hardware compatível com Arduino que possui uma tela TFT LCD (*thin film transistor liquid crystal display*) de 2,2 polegadas (240x320 pixels) e um microcontrolador 8-bits.

Uma tela de 15 polegadas, com resolução 1024x768 pixels, possui aproximadamente 85 PPI de densidade de pixels, enquanto uma tela de 2,2 polegadas com 240x320 pixels dispõe de aproximadamente 180 PPI de densidade de pixels. Conclui-se então que a tela do Odroid-Show possui qualidade considerável para representar caracteres e não foram encontrados trabalhos que explorassem tal otimização para o dispositivo até o momento. Além disso, o presente trabalho visa gerar tecnologia para a modernização de um DSET.

Muitos sistemas embarcados possuem dispositivos periféricos conectados ao microcontrolador a fim de expandir suas capacidades (RUSSELL, 2010). O Odroid-Show se comunica com outros periféricos através de comunicação serial. Logo, pode-se desenvolver um protocolo de comunicação serial para envio de fontes de texto para o Odroid-Show.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo tem por objetivo fazer a apresentação dos conceitos teóricos envolvidos no projeto.

2.1 Comunicação Serial e USART

A comunicação serial, ao contrário da comunicação paralela, utiliza um único canal de transmissão. A comunicação ocorre através do envio e recebimento de bytes que são transmitidos bit a bit. O Odroid-Show, assim como outros sistemas baseados em Arduino, utiliza o circuito de comunicação USART (*Universal Synchronous/Asynchronous Receiver/Transmitter*) o qual permite a transformação entre as formas serial e paralela. O USART provê uma linha de comunicação para recebimento de dados (RX) e para transmissão (TX). No Odroid-Show as linhas RX e TX estão conectadas ao circuito integrado de conversão USB-to-UART, através do qual o software desenvolvido é baixado para o dispositivo. No caso do Odroid-Show, o USART opera sob o protocolo RS-232, responsável por especificar diversos aspectos de comunicação, dentre eles conectores, tensão, ordenamento de bits e taxa de transferência de bits (*bit rate*). O sistema USART não utiliza sinal de *clock* para realizar a sincronização durante a comunicação. Cada dispositivo deve aguardar por bits em sua linha RX. Quando o bit inicial é encontrado, o dispositivo sabe que a comunicação iniciou. Ambos os dispositivos (transmissor e receptor) devem concordar em utilizar um *bit rate* comum, caso contrário, o receptor decodificará os dados recebidos de maneira errada.

2.2 Tabela ASCII

O sistema de fontes proposto por este trabalho faz uso apenas dos caracteres imprimíveis da tabela ASCII (*American Standard Code for Information Interchange*). A tabela é responsável por armazenar o código numérico para representação de caracteres. Contém 128 caracteres, dos quais 95 são imprimíveis e estão nas posições de 32 (caractere espaço) até a posição 126 (caractere til) (ASCIITable.com, 2015).

2.3 Caracteres e Glifos

Atualmente quaisquer pessoas que utilizam computadores ou telefones celulares fazem uso, conscientemente ou não, de fontes de texto. Diferentes tipos de mídias escritas - desde livros a websites - usam fontes de texto específicas como recurso tipográfico para melhor se comunicar com seu público alvo.

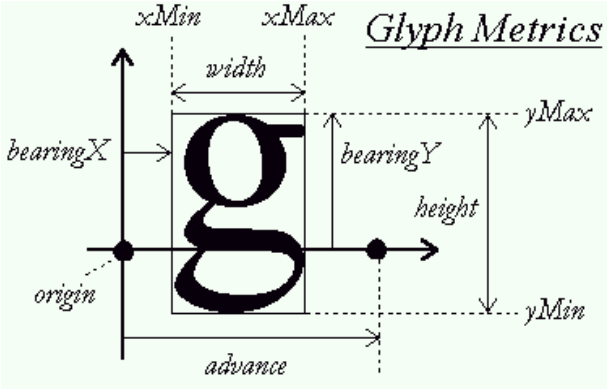

Figura 2.1 – Métricas de glifo definidas pela biblioteca FreeType 2.

Sendo as fontes uma parte fundamental do presente trabalho, cabe aqui definir os elementos básicos que as constituem. HARALAMBOUS; HORNE (2007) propoem que fontes são constituídas basicamente por caracteres e glifos (2.1). Definiram glifo como a imagem de um símbolo utilizado em um sistema de escrita ou notação, como as utilizadas em um alfabeto, em um conjunto de ideogramas ou em notação musical, por exemplo. Caractere foi definido como a descrição linguística ou lógica de uma classe equivalente de glifos. Ou seja, no caso de um alfabeto, um caractere representa uma letra, enquanto um glifo se refere especificamente a forma utilizada para representar tal letra (ver figura 2.2).

Figura 2.2 – Diversos glifos representando o caractere "a".

2.4 Formatos de Fontes

Existem basicamente duas abordagens de implementação para definir caracteres em fontes: *raster* (*bitmap*) ou vetores. Um caractere guardado como *bitmap* é representado como um

array de duas dimensões, onde cada pixel é representado por 1 bit. Por outro lado, a abordagem vetorial utiliza um conjunto de linhas e curvas de Bezier para definir o contorno do caractere.

Ao longo dos anos, diversos formatos de arquivos de fontes foram propostos, a seguir são apresentadas as características de alguns desses formatos.

- **RAW**: De acordo com HARALAMBOUS; HORNE (2007) é o formato de fonte bitmap mais simples que existe. Fontes RAW contém apenas o bitmap de cada caractere, não possuem cabeçalho, a largura dos caracteres é fixa em 8 pixels e existem exatamente 256 glifos. Sendo assim, para encontrar o i-ésimo caractere em um arquivo RAW multiplica-se i x 8 x h, onde h representa a altura do mesmo, sendo definida por T/(8x256), onde T é o tamanho do arquivo.

- **PXL**: Um formato semelhante ao RAW, porém, contém 128 glifos de largura variável e guarda uma tabela de dados métricos ao final do arquivo.

- **CPI**: A Microsoft introduziu o formato *Code Page Information*(CPI) durante a era MS-DOS, no qual três tamanhos de caracteres (8x8, 14x8 e 16x8 pixels) eram oferecidos, cada um contendo 256 glifos. A fonte era dividida em sessões, cada uma delas divididas em três sub-sessões, uma para cada tamanho de caractere. Os caracteres eram representados por sequências de bytes, como possuiam largura fixa de 8 pixels, cada byte representava uma linha de um caractere.

- **FNT**: Outro formato introduzido pela Microsoft foi o FNT, no qual caracteres eram formados por colunas de pixels onde a largura da coluna era definida por um parâmetro no cabeçalho do arquivo e um caractere consistia de uma ou mais colunas, dependendo de sua largura.

- **PSF**: O formato PSF adotado pelo Linux em 1994 possuía um cabeçalho de 32 bytes contendo o número de caracteres, tamanho da fonte e largura máxima de caractere. O número de bytes necessário para armazenar um caractere dependia do tamanho da fonte e da largura do mesmo.

- **FONT**: A Apple desenvolveu o formato FONT para ser utilizado nos seus computadores Macintosh, o qual consistia de uma grande imagem bitmap na qual todos os caracteres estavam concatenados, a qual era conhecida como *global image*. Utilizava-se uma tabela de ponteiros indicando a origem dos caracteres e o cabeçalho do arquivo informava o

tamanho da fonte, largura da *global image*, largura máxima de caractere, entre outros parâmetros.

- **BDF**: A Adobe Systems desenvolveu o formato *Bitmap Distribution Format* (BDF), o qual é utilizado como base para praticamente todas fontes Unix (HARALAMBOUS; HORNE, 2007). Escrito puramente em ASCII, cada arquivo de fonte é dividido em propriedades e sessões de caracteres. As propriedades contém nome da fonte, tamanho e outros parâmetros, enquanto cada caractere guardado é definido em linhas de texto, utilizando-se uma linha de texto por linha de pixels do caractere em questão. A linha de pixels é codificada como um valor hexadecimal, por exemplo a linha 00110100 seria codificada como 0x34. Além disso, os caracteres podem ser representados em escala de cinza com 2, 4 ou 8 bits por pixel (representando 4, 16 ou 256 tons de cinza).

- **GF**: Uma abordagem semelhante ao BDF é utilizada pelo formato *Generic Font* (GF), embora a forma do caractere não seja escrita em ASCII, utiliza-se uma série de operadores seguidos de um ou mais valores. Os operadores indicam começo do caractere (*boc*), nova linha (*newrow*), pintar (*paint*) e fim do caractere (*eoc*). O operador nova linha especifica quantos pixels são deixados em branco ao seguir para próxima linha. Em seguida, o operador pintar é utilizado para desenhar pixels pretos.

- **TrueType**: Durante os anos de 1987 a 1989 a linguagem PostScript, introduzida pela Adobe, era a linguagem padrão para integrar sistemas operacionais e impressoras. Diante disso, Microsoft e Apple se uniram para desenvolver um novo sistema de renderização de fontes, o qual veio a ser conhecido como TrueType. Em 1991 a Apple liberou uma versão em *plugin* de TrueType para seu Mac OS 6, enquanto apenas em 1992 a Microsoft viria a incorporar o sistema TrueType em seu Windows 3.1 (HARALAMBOUS; HORNE, 2007).

  Um glifo TrueType é definido por linhas e curvas. O formato faz uso de tabelas para guardar informações da fonte como métricas, mapeamento entre caracteres e glifos, nomes, dados de bitmap e ponteiros para os glifos, entre outros.

SUMÁRIO DO CAPÍTULO

Este capítulo abordou fundamentos importantes para o desenvolvimento do presente trabalho, como o funcionamento da comunicação serial em sistemas Arduino, a diferença entre caractere e glifo, a definição dos 95 caracteres para os quais serão gerados glifos e peculiaridades de diferentes formatos de arquivos de fontes de texto.

3 ARQUITETURA DA SOLUÇÃO

O presente trabalho propõe desenvolver um novo sistema de fonte para o Odroid-Show. Este capítulo visa expor a arquitetura, as ferramentas e abordagem de criação para o novo sistema. Ademais, muitas das seguintes seções visam expor e discutir problemas que justificam a escolha das soluções.

3.1 Ferramentas

A seguir são apresentadas as ferramentas utilizadas no desenvolvimento do presente trabalho.

3.1.1 FreeType 2

FreeType 2 é uma biblioteca de software escrita em linguagem C, desenvolvida para renderização de fontes. Provê uma interface de acesso aos arquivos de fonte, gerando bitmaps dos glifos com redução de serrilhado, fazendo uso de uma escala de 256 tons de cinza e suportando os formatos *TrueType*, *OpenType*, *Type1*, *CID*, *CFF*, *Windows FON/FNT*, *X11*, *PCF* entre outros (FreeType 2, 2015).

Além de prover acesso ao bitmap de glifos, a FreeType 2 também provê acesso a informações métricas como distância necessária entre glifos, largura e altura dos mesmos. Após inicializar a biblioteca deve-se fornecer um arquivo de fonte (Courier por exemplo, o qual ocupa aproximadamente 700 Kb, ou outro) como mostra o código 3.1.

```
#include <ft2build.h>
#include FT_FREETYPE_H

FT_Library library;

error = FT_Init_FreeType(&library);
if(error)
{
        /* erro ao inicializar a biblioteca FreeType */
}

error = FT_New_Face( library, "./fonts/Courier.ttf",0,&face );

if ( error == FT_Err_Unknown_File_Format )
{
    /* o arquivo de fonte foi aberto e lido, mas
    seu formato nao e suportado */
}
else if ( error )
```

```
{
    /* o arquivo de fonte nao pode ser aberto, lido ou
    esta corrompido */
}
```

Código 3.1 – Inicialização da biblioteca FreeType 2.

3.1.2 Odroid-Show

O Odroid-Show (figura 3.1) é um hardware compatível com Arduino, o quadro 3.1 apresenta as especificações técnicas do dispositivo, esse possui uma tela TFT LCD (*thin film transistor liquid crystal display*) de 2,2 polegadas (240x320 pixels) e um microcontrolador 8-bits ATMEL ATmega328P funcionando a 16Mhz. ATmega328P é um microcontrolador 8-bits, de arquitetura AVR. De acordo com (RUSSELL, 2010), a arquitetura AVR é um aprimoramento da arquitetura RISC (*Reduced Instruction Set Computing*). Tal microcontrolador possui 32KBytes de memória ISP (*In-System Programmable*) flash para armazenar programas, 1KByte de memória EEPROM (não volátil) a qual é utilizada pela IDE do Arduino como *boot-loader* para carregar programas guardados na memória flash, 2Kbytes de SRAM para guardar variáveis em tempo de execução e se comunica através de porta serial com *Baud rate* de até 500.000 bps (0,5Mbps) (Odroid.com, 2015). O *boot-loader* do Odroid-Show utiliza USART (*Universal Synchronous/Asynchronous Receiver/Transmitter*) configurado para o padrão de comunicação RS-232 via conversor USB-serial.

A tela do Odroid-Show suporta 16 bits de cor RGB 565 (5 bits para vermelho, 6 para verde e 5 para azul). É possível repintar a tela completamente ou parcialmente. Sua taxa de atualização atinge aproximadamente 4 fps (*frames per second*) em tela cheia. A memória de vídeo é não volátil, i.e., se nada for desenhado, a última imagem mostrada se mantém na tela.

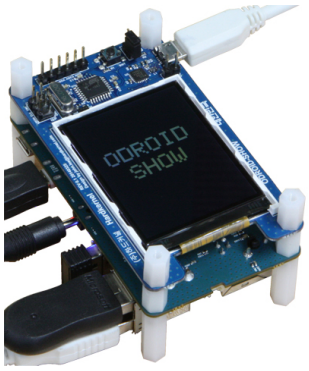

Figura 3.1 – O hardware Odroid-Show

Quadro 3.1 – Odroid-Show: Especificações Técnicas

	Odroid-Show
MCU	ATmega328P 16Mhz
Memória ISP	32 Kbytes
Memória EEPROM	1 Kbyte
Memória SRAM	2 Kbytes
Baud rate	até 0,5 Mbps
Buffer de comunicação serial	64 bytes
LCD	2,2 240x320 TFT-LCD SPI 8MHz interface
Taxa de atualização ao repintar a tela	aproximadamente 4 fps
Frequência	70Hz
Cores	RGB-565

3.1.3 A IDE Arduino

A IDE (*Integrated Development Environment*) Arduino é utilizada para desenvolver software para o Odroid-Show e outros dispositivos baseados em Arduino. Programas desenvolvidos na IDE são chamados *sketches*, podem ser escritos em linguagem C e são salvos em arquivos com a extensão ".ino". Para rodar um programa basta realizar a compilação e upload da *sketch*. É preciso definir qual porta - serial ou USB - está sendo utilizada e o modelo de dispositivo em questão, como mostra a figura 3.2.

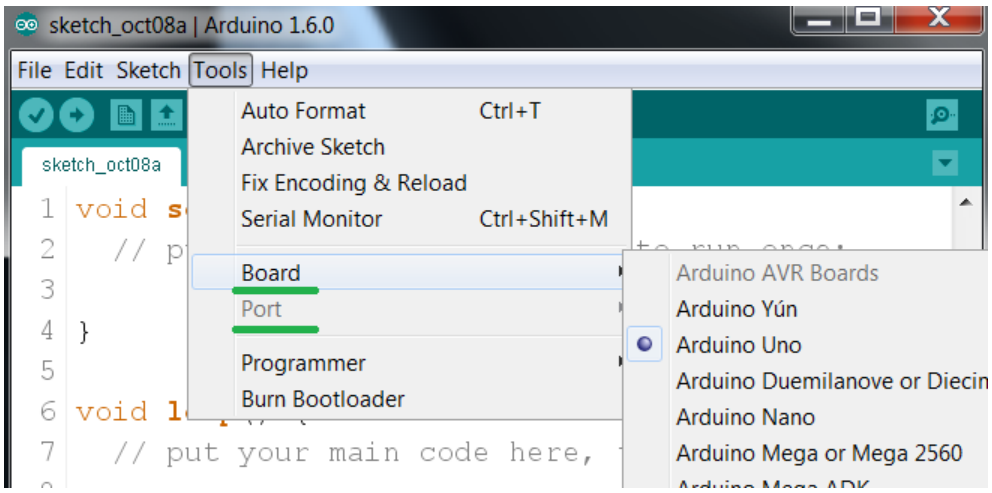

Figura 3.2 – IDE Arduino: menu de seleção de dispositivo e abaixo seleção de porta serial/USB.

3.2 Armazenamento vs Envio de Glifos

Esta seção discute sobre as possibilidades de armazenamento ou envio de glifos para o Odroid-Show.

Para representar a forma de um caractere, técnicas de obtenção de contorno do mesmo podem ser utilizadas. Muitas delas representam a forma de um caractere através de vários tipos de aproximações do contorno, incluindo polígonos e curvas splines. Embora tais técnicas atinjam alto nível de compressão, são muito complexas para utilização em sistemas embarcados (GLOBACNIK; ZALIK, 2010).

GLOBACNIK; ZALIK (2010) propôs a utilização de código de cadeia (*chain code*) para definir o contorno de glifos, fazendo uso de códigos pré-definidos para denotar a direção na qual está o próximo pixel pertencente ao contorno do caractere. Após a definição dos glifos, é utilizada compressão Huffman sobre os mesmos.

Pensando nas limitações de memória apresentadas na subsessão 3.1.2, optou-se por enviar os glifos como bitmap para o Odroid-Show via comunicação serial, possibilitando que o dispositivo leia os bytes recebidos para formar os glifos, os quais após desenhados na tela não precisam ser armazenados. Dessa forma evita-se o armazenamento da árvore de decodificação Huffman no dispositivo, resultando em melhor aproveitamento de memória.

3.3 Raster vs Vetores

Fontes vetoriais permitem a geração de glifos reescaláveis, fazendo com que seu uso seja independente da resolução do dispositivo em foco. No entanto, o uso de vetores e curvas resulta em alta demanda por processamento, normalmente restringindo sua utilização em sistemas embarcados (GLOBACNIK; ZALIK, 2010).

A abordagem raster define cada glifo como um bitmap, o que por um lado resulta em glifos de resolução fixa, por outro oferece baixa complexidade. O processamento necessário para leitura de um bitmap consiste basicamente na verificação da cor de cada pixel. Para fins de compressão pode-se definir a cor do pixel com apenas 1 bit, representando *background* ou *foreground*.

3.4 Abordagem Utilizada

A biblioteca FreeType 2 (subseção 3.1.1) permite gerar bitmaps para glifos de fontes TrueType, tornando possível contornar os problemas de personalização e resolução fixa em fontes raster (seção 3.3). Para tanto, basta definir o tamanho da fonte e estilo (normal, negrito, itálico) antes de gerar o bitmap de cada glifo.

Dadas as justificativas apresentadas nas seções 3.2 e 3.3, propõe-se o uso da biblioteca FreeType 2 para definir tamanho e estilo, seguido da geração de um bitmap para cada glifo da fonte em questão. Os glifos serão armazenados em um arquivo de saída, o qual posteriormente será utilizado como entrada no programa responsável por enviar a fonte para o Odroid-Show.

3.5 Modelos de Arquivo para Armazenamento das Fontes Geradas

Fazendo uso da abordagem exibida na seção 3.4 são gerados 95 bitmaps, os quais representam caracteres imprimíveis da tabela ASCII (seção 2.2). As fontes geradas por este trabalho são monoespaçadas, isto é, todos os caracteres possuem as mesmas dimensões. Os bits empregados na representação de pixels são armazenados como *array* de bytes, os quais são salvos em arquivo no formato ASCII. Todos os *arrays* de bytes iniciam na posição superior esquerda do glifo e terminam na posição inferior direita.

A seguir são propostos dois modelos de arquivo para armazenamento da fonte gerada.

3.5.1 Modelo Bit a Bit

Nesse modo de representação os glifos são definidos bit a bit. Bit 0 representa pixel em *background* e bit 1 em *foreground*. Os bits são agrupados em um *array* de bytes.

Por exemplo, para gerar o glifo 'W' lê-se o bitmap pixel a pixel atribuindo 1 para pixel de *foreground* e 0 para *background*. Em seguida, os bits são agrupados em bytes. Logo, os primeiros bytes da figura 3.3(a) são *00000000, 00000000, 00000000* e *00110001*, os quais representam *0, 0, 0 e 49* em decimal.

O arquivo que representa a fonte contém:

- **Cabeçalho**: uma linha indicando nome, tamanho utilizado ao gerar glifos com a FreeType 2, dimensões da fonte e modo de representação (bit a bit).
- **Número de bytes**: uma linha indicando quantos bytes são necessários para representar

cada glifo.

- **Arrays de bytes**: cada linha subsequente representa um glifo, os quais são escritos byte a byte (em representação decimal) separados por espaço.

A figura 3.4 apresenta as primeiras linhas de uma fonte gerada pelo modelo bit a bit. Observe que os bytes estão representados por números decimais e que o primeiro glifo é uma sequência de zeros, pois representa o caractere espaço, que contém apenas pixels de *background*.

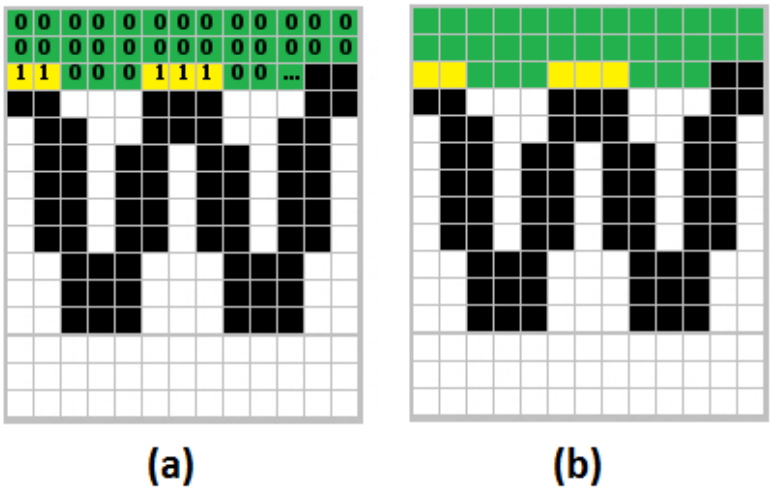

Figura 3.3 – (a) leitura do bitmap no modo bit a bit, (b) leitura do bitmap no modo alternado

```
1  Font: ./resources/courbd.ttf Size 16. Character size 16x23px. Mode
   Bit a Bit 46 bytes
2  46
3  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0
4  0 0 0 0 1 192 3 192 3 192 3 192 1 192 1 192 1 192 1 192 1 192 1 128 1
   128 0 0 0 0 0 0 3 192 3 192 3 192 0 0 0 0 0 0 0 0
5  0 0 0 0 31 120 31 120 31 120 30 120 30 120 14 120 14 112 0 0 0 0 0 0
   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Figura 3.4 – Primeiras linhas de um arquivo gerado pelo modelo bit a bit.

3.5.2 Modelo Alternado

Esse modelo define glifos como *array* de bytes. No *array* são alternados bytes que representam sequência de pixels em *background* e sequência de pixels em *foreground*.

Por exemplo, para gerar o glifo 'W' lê-se o bitmap e conta-se quantos pixels de *background* existem em sequência até encontrar o primeiro pixel de *foreground*, em seguida, conta-

se quantos pixels de *foreground* existem em sequência até encontrar o próximo pixel de *background*. Assim sucessivamente alterna-se entre sequências de *background* e *foreground*. Cada um desses contadores é armazenado em um byte. Logo, em representação decimal, os primeiros bytes gerados na figura 3.3(b) seriam *26,2,3,3,3*.

O arquivo que representa a fonte contém:

- **Cabeçalho**: uma linha indicando nome, tamanho utilizado ao gerar glifos com a FreeType 2, dimensões da fonte e modo de representação (alternado).
- **Dimensões da fonte**: uma linha indicando largura e altura da fonte, separadas por espaço.
- **Número de bytes e arrays de bytes**: duas a duas, as linhas subsequentes representam respectivamente número de bytes necessários para armazenar o glifo e array de bytes do mesmo.

A figura 3.5 apresenta as primeiras linhas de uma fonte gerada pelo modelo alternado. Observe que a terceira linha do arquivo indica o número de bytes usados pelo caractere espaço. Em seguida, a quarta linha contém "253 0 69", que significa 253 pixels em *background*, 0 pixels em *foreground* seguidos por 69 pixels em *background*.

```
1  Font: ./resources/courbd.ttf Size 16. Character size 14x23px. Mode
   Alternado
2  14 23 pixels
3  3
4  253 0 69
5  25
6  33 3 11 3 11 3 11 3 11 3 11 3 12 2 12 2 12 2 12 2 40 2 12 2 104
7  25
8  46 2 2 3 7 2 2 3 7 2 2 3 7 2 2 2 8 2 3 1 8 2 3 1 200
```

Figura 3.5 – Primeiras linhas de um arquivo gerado pelo modo de armazenamento alternado.

3.6 Fluxograma da Solução

Esta seção apresenta o fluxo de informação entre os diferentes programas envolvidos neste trabalho. Tal processo é representado visualmente pela figura 3.6. A seguir são apresentados os programas que fazem parte da solução:

- **MyFont**: programa responsável por gerar os arquivos de fonte, seja no modo bit a bit ou alternado. *Entrada*: arquivo de fonte monoespaçada TrueType. *Saída*: arquivo de fonte gerado no modo selecionado (bit a bit ou alternado).

- **Modo_Bit_a_Bit**: sketch do Odroid-Show para receber fontes no modo bit a bit.
- **Modo_Alternado**: sketch do Odroid-Show para receber fontes no modo alternado.
- **SerialCom**: programa responsável por ler o arquivo de fonte e enviar glifos para o Odroid-Show.

Deve-se utilizar o programa MyFont para gerar a fonte, em seguida deve-se realizar o upload da sketch correta para o Odroid-Show. Por fim, o programa SerialCom é utilizado para realizar o envio das fontes para o Odroid-Show.

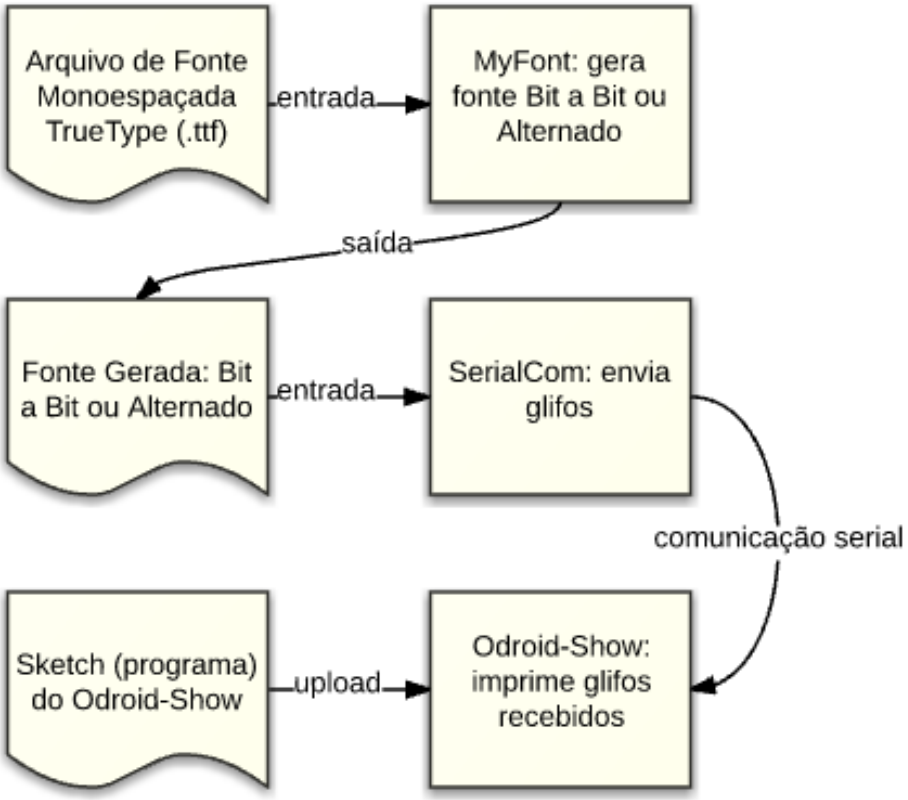

Figura 3.6 – Fluxograma exibindo os processos de geração e envio de fontes.

SUMÁRIO DO CAPÍTULO

Este capítulo abordou tópicos essenciais para compreensão das decisões tomadas no desenvolvimento do novo sistema de fontes, bem como tópicos fundamentais para aqueles que desejam dar continuidade ao presente trabalho. Foi abordado o funcionamento da biblioteca FreeType 2, a arquitetura do Odroid-Show, o funcionamento da IDE do Arduino, as diferenças entre fontes raster e vetoriais, além dos motivos que resultaram na preferência pelo envio de

glifos ao invés do armazenamento dos mesmos. Ademais foram definidos dois modelos de arquivos de fonte, os modelos Bit a Bit e Alternado.

4 IMPLEMENTAÇÃO

Este capítulo expõe a implementação e uso dos diferentes programas que fazem parte do presente trabalho.

4.1 Geração de Glifos Através da FreeType 2

O código 4.1 apresenta o método "*Font::genFreetypeAlphabet()*", responsável por acessar e armazenar os bitmaps dos glifos da fonte TrueType. Para tanto, percorrem-se as posições imprimíveis da tabela ASCII e, para cada uma delas gera-se um bitmap referente ao glifo daquele caractere. É importante observar que ocorre compactação no laço *for* mais interno ao reduzir o valor do pixel que era representado em 8-bits para apenas 1-bit. Para uma representação visual dos glifos que foram gerados, basta utilizar o método "*Font::printFreetypeOnConsole()*".

Os glifos gerados pelo método "*Font::genFreetypeAlphabet()*"ocupam a menor largura e altura possíveis como mostra a figura 4.1. O glifo referente ao caractere 't' possui altura e largura diferentes do glifo referente ao caractere 'p'.

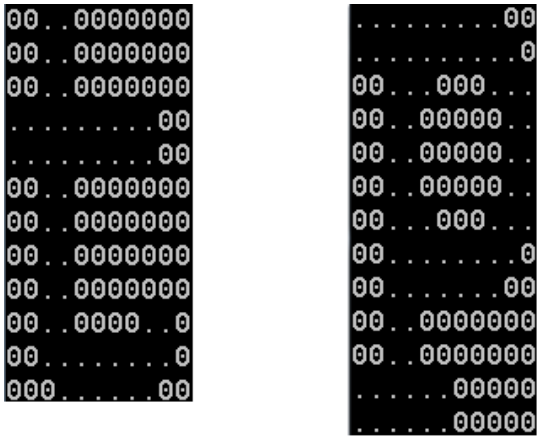

Figura 4.1 – Dois glifos, com dimensões diferentes, gerados pelo método Font:: genFreetype-Alphabet().

Para alinhar corretamente os glifos através de uma mesma *baseline* e evitar a necessidade de armazenar informações de deslocamento horizontal e vertical, o método "*Font::genFontAlphabet()*", exposto no código 4.2, acrescenta novas linhas e colunas de pixels aos glifos, deixando-os todos com as mesmas dimensões e corretamente alinhados, como pode-se verificar na figura 4.2.

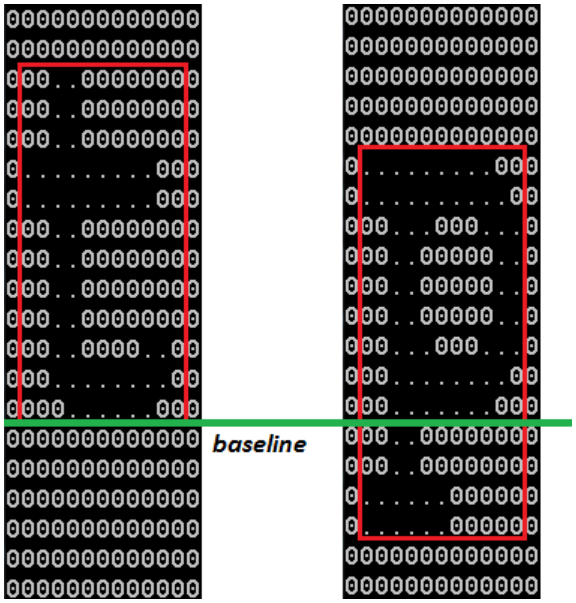

Figura 4.2 – Glifos com novas linhas e colunas de pixels, permitindo alinhamento correto em uma mesma *baseline*.

A largura dos glifos é equivalente a largura do glifo mais largo gerado pelo método "*Font:: genFreetypeAlphabet()*". A altura é dada pela maior distância entre *baseline* e topo do glifo, somada a maior distância encontrada entre *baseline* e restante abaixo da baseline.

```
1  //Atraves da Freetype gera bitmaps de glifos para todas as posicoes
       imprimiveis da tabela ASCII
2  void Font::genFreetypeAlphabet() {
3
4     for (int ch = ASCII_ALPHABET_BEGIN; ch < ASCII_ALPHABET_BEGIN +
          NUM_CHARS; ch++)
5     {
6        //Carrega glifo do nosso caractere
7        if (FT_Load_Glyph(face, FT_Get_Char_Index(face,
8           (char) ch), FT_LOAD_DEFAULT))
9           throw std::runtime_error("FT_Load_Glyph failed");
10
11       //Move a face do glifo para um objeto FT_Glyph
12       FT_Glyph glyph;
13       if (FT_Get_Glyph(face->glyph, &glyph))
14          throw std::runtime_error("FT_Get_Glyph failed");
15
16       //Converte o glifo para bitmap
17       FT_Glyph_To_Bitmap(&glyph, ft_render_mode_normal, 0, 1);
18       FT_BitmapGlyph bitmap_glyph = (FT_BitmapGlyph)glyph;
19
20       //Referencia para facilitar o acesso ao bitmap
21       FT_Bitmap& bitmap = bitmap_glyph->bitmap;
22
23       //vetor para armazenamento do glifo atual:
24       //1 para pixel de foreground, 0 para background
25       vector<int> currentCharacter;
26
```

```
27          /****
28          Percorre o bitmap pixel a pixel armazenando 1 para
29          foreground e 0 para background. Ignora a informacao de
                antialiasing: se valor do pixel for maior que 100 entao eh
                foreground, se nao eh background. Resulta em compactacao de
                8-bits/pixel para 1-bit/pixel.
30          ****/
31          for (int j = 0; j <bitmap.rows; j++)
32          {
33             for (int i = 0; i < bitmap.width; i++)
34             {
35                unsigned char aux = bitmap.buffer[i + bitmap.width*j];
36
37                if (aux > 100)
38                {
39                   //foreground
40                   currentCharacter.push_back(1);
41                }
42                else
43                {
44                   //background
45                   currentCharacter.push_back(0);
46                }
47             }
48          }
49          //Armazena glifo atual
50          freetypeAlphabet.push_back(currentCharacter);
51          //Armazena largura do glifo atual
52          freetypeWidths[ch - ASCII_ALPHABET_BEGIN] = bitmap.width;
53          //Armazena altura do glifo atual
54          freetypeHeights[ch - ASCII_ALPHABET_BEGIN] = bitmap.rows;
55          //Armazena distancia da baseline ateh topo do glifo
56          freetypeGlyphs_Top[ch - ASCII_ALPHABET_BEGIN] = bitmap_glyph->
                top;
57       }
58    }
```

Código 4.1 – Método Font::genFreetypeAlphabet(), responsável por gerar glifos a partir da FreeType 2.

```
1     //Gera os glifos para nossa fonte
2     void Font::genFontAlphabet()
3     {
4        int maxTop = findMax(freetypeGlyphs_Top, NUM_CHARS);
5
6        //largura: mesma do glifo mais largo
7        width = findMax(freetypeWidths, NUM_CHARS);
8        //altura: maior top + maior distancia entre baseline e parte
             abaixo da baseline
9        height = maxTop + abs(findMin(freetypeGlyphs_Top, NUM_CHARS) -
             freetypeHeights[minTopIndex()]);
10
11       int h_ = height;
12       int w_ = width;
13
14       for (int c = 0; c < NUM_CHARS; c++){
15
16          //numero de colunas extras
```

```
17          int extraCols = (w_ - freetypeWidths[c]) / 2;
18          //numero de linhas extras
19          int extraRows = maxTop - freetypeGlyphs_Top[c];
20
21          //glifo atual
22          vector<int> currentChar;
23          for (int i = 0; i < h_; i++){
24             for (int j = 0; j < w_; j++){
25                int aux = extraCols + freetypeWidths[c];
26                int aux2 = extraRows + freetypeHeights[c];
27                //se j estiver na area correta, guarda info do caractere
28                if ((i >= extraRows) && (i < aux2) && (j >= extraCols) &&
                       (j < aux)){
29                   currentChar.push_back(freetypeAlphabet[c][((i -
                          extraRows)* freetypeWidths[c]) + (j - extraCols)]);
30                }
31                //se nao, coloca 0
32                else{
33                   currentChar.push_back(0);
34                }
35             }
36          }
37          //armazena glifo gerado
38          fontAlphabet.push_back(currentChar);
39       }
40    }
```

Código 4.2 – Método Font::genFontAlphabet(), responsável por gerar glifos para classe Font.

4.2 Geração da Fonte no Modo Bit a Bit

Após a inclusão de novas linhas e colunas de pixels, os novos glifos são traduzidos para o modelo Bit a Bit. Armazena-se bit a bit a informação de *background* e *foreground*, preenchendo o byte atual, representado no código 4.3 como "*alphabetA[c][byte]*". Para tanto, empregam-se sucessivas operações *shift* para realizar o deslocamento do bit 1 para a esquerda na variável "*unsigned char mask*". Em seguida, armazena-se "*alphabetA[c][byte]* |= *mask*", *i.e.*, o byte atual *alphabetA[c][byte]*, originalmente composto por bits zeros, preserva seu estado original e armazena apenas um bit de cada vez através da operação *OR*, a qual altera apenas a posição ocupada pelo único bit 1 contido em *mask*.

Entre as linhas 15 e 22 do método "*Font::genModeAAlphabet()*", supondo que os dois primeiros pixels a serem lidos sejam pixels de *foreground*, a condição no *if* da linha 15 será verdadeira e, durante as duas iterações de leitura dos pixels, acontecerá o seguinte processo:

- Primeiro pixel:
  - *mask* = 00000001 « (7 - bit) = 00000001 « (7 - 0) = 10000000

  - alphabetA[c][byte] = alphabetA[c][byte] OR mask = 00000000 OR 10000000 = 10000000

- Segundo pixel:

  - mask = 00000001 « (7 - bit) = 00000001 « (7 - 1) = 01000000
  - alphabetA[c][byte] = alphabetA[c][byte] OR mask = 10000000 OR 01000000 = 11000000

A variável *bit*, na linha 21, sempre contém um valor inteiro no intervalo fechado [0,7]. Essa variável é utilizada para auxiliar no deslocamento que ocorre na linha 17.

Utiliza-se a variável *byte*, na linha 22, para armazenar o índice do *array* que armazena o glifo. Como a iteração ocorre bit a bit, a variável *byte* só é incrementada a cada 8 iterações.

```
1  //Gera glifos do modo Bit a Bit
2  void Font::genModeAAlphabet()
3  {
4     for (int c = 0; c < NUM_CHARS; c++)
5     {
6        int bit = 0, byte = 0, cont = 0;
7        unsigned char mask = 0;
8        //le o bitmap do caractere
9        for (int y = 0; y < height; y++)
10       {
11          for (int x = 0; x < width; x++)
12          {
13             int pos = (y*width) + x;
14             //se for foreground, coloca um bit 1
15             if (fontAlphabet[c][pos] == 1)
16             {
17                mask = 1 << (7 - bit);
18                alphabetA[c][byte] |= mask;
19             }
20             cont++;
21             bit = cont % 8;
22             byte = cont / 8;
23          }
24       }
25    }
26 }
```

Código 4.3 – Método Font::genModeAAlphabet(), responsável por gerar glifos no modo Bit a Bit.

4.3 Geração da Fonte no Modo Alternado

O código 4.4 expõe o método de geração dos glifos no modo Alternado. O método consiste em ler os bitmaps armazenados em "*fontAlphabet[c]*", na linha 18, e então traduzi-los

para o modelo Alternado.

O processo de tradução dos glifos fundamenta-se na contagem e armazenamento de sequências de pixels de *background* e *foreground*. Tal processo equivale a uma máquina de estados finitos e está resumido na figura 4.3.

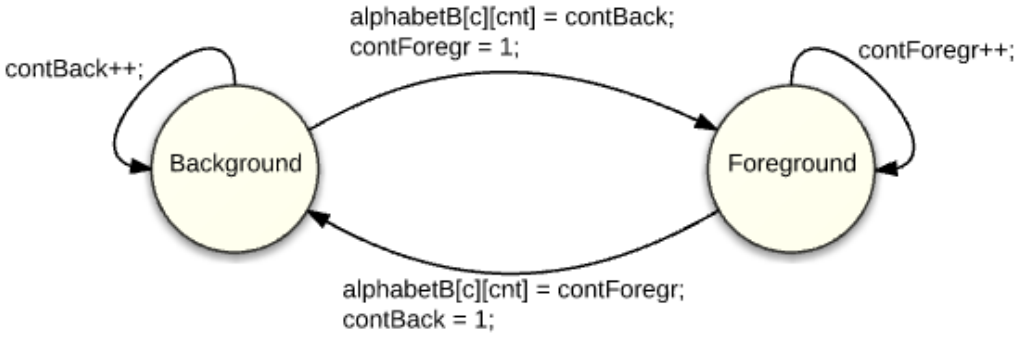

Figura 4.3 – Máquina de estados finitos representando uma versão resumida do método Font::genModeBAlphabet().

```
1
2
3   //Gera glifos do modo Alternado
4   void Font::genModeBAlphabet()
5   {
6      //background pixel (flag)
7      bool flagBack = true;
8      //contadores de background e foreground
9      int contBack = 0, contForegr = 0, cnt = 0;
10
11     for (int c = 0; c < NUM_CHARS; c++)
12     {
13        for (int i = 0; i < height; i++)
14        {
15           for (int j = 0; j < width; j++)
16           {
17              //Background pixel
18              if (fontAlphabet[c][(i*width) + j] == 0)
19              {
20                 //Continua contagem de pixel de background
21                 if (flagBack == true)
22                 {
23                    contBack++;
24                    //254 e 255 reservados para protocolo
25                    if (contBack == 254)
26                    {
27                       alphabetB[c][cnt] = 253;
28                       alphabetB[c][cnt + 1] = 0;
29                       contBack = 1;
30                       cnt += 2;
31                    }
32                 }
33                 //Trocou estado para pixels de background
34                 else
35                 {
```

```
36                      flagBack = true;
37                      alphabetB[c][cnt] = contForegr;
38                      contBack = 1;
39                      cnt++;
40                  }
41
42              }
43              else
44              {
45                  //Trocou estado para pixels de foreground
46                  if (flagBack == true)
47                  {
48                      flagBack = false;
49                      alphabetB[c][cnt] = contBack;
50                      contForegr = 1;
51                      cnt++;
52                  }
53                  //Continua contagem de pixels de foreground
54                  else
55                  {
56                      contForegr++;
57                      //254 e 255 reservados para protocolo
58                      if (contForegr == 254)
59                      {
60                          alphabetB[c][cnt] = 253;
61                          alphabetB[c][cnt + 1] = 0;
62                          contForegr = 1;
63                          cnt += 2;
64                      }
65                  }
66              }
67          }
68
69      }
70      //Armazena o ultimo pixel
71      if (flagBack == true)
72      {
73          alphabetB[c][cnt] = contBack;
74          cnt++;
75      }
76      else
77      {
78          alphabetB[c][cnt] = contForegr;
79          cnt++;
80      }
81      //Armazena length do glifo
82      charSizesAlph_B[c] = cnt;
83
84      cnt = 0;
85      contForegr = 0;
86      contBack = 0;
87      flagBack = true;
88  }
89 }
```

Código 4.4 – Método Font::genModeBAlphabet(), responsável por gerar glifos no modo Alternado.

4.4 Utilização do Programa MyFont

O programa MyFont possui entrada e saída definidas na seção 3.6. A pasta "MyFont/-resources"guarda arquivos de entrada (fontes monoespaçadas TrueType) os quais podem ser utilizados para gerar fontes no modo Bit a Bit e Alternado, outros arquivos de fontes TrueType monoespaçadas podem ser adicionados. Ao executar o código 4.5 gera-se os arquivos de saída "Courier New_Bold_36_Alternado.txt"e "Courier New_Bold_36_Bit_a_Bit.txt", os quais referem-se às fontes Courier Bold, tamanho 36 em seus respectivos modos de geração. Esses arquivos ficam armazenados no diretório raiz da solução.

O código 4.6 expõe todos os métodos públicos definidos pela classe Font.

```
1  int main(){
2     Font *myFont;
3     /****Tratamento de erros ao criar a fonte****/
4     try{
5        myFont = new Font("./resources/CourierBold.ttf", 36);
6     }
7     catch (std::runtime_error& e){
8        std::cout << e.what() <<  std::endl;
9        system("Pause");
10       exit(EXIT_FAILURE);
11    }
12    system("Pause");
13    return EXIT_SUCCESS;
14 }
```

Código 4.5 – Exemplo de geração de fonte Courier Bold no modo Alternado.

```
1
2     //Construtor. Recebe path da fonte (.ttf) e tamanho
3     Font(const char *fontPath, unsigned int size);
4
5     //Armazena em "width" e "height" as dimensoes da fonte
6     void getDimensions(int *width, int *height);
7
8     //Armazena glifo de "c" em "arr[]" no modo Bit a Bit
9     void getCharBit_a_Bit(char c, unsigned char arr[], int *length);
10
11    //Armazena glifo de "c" em "arr[]" no modo Alternado
12    void getCharAlternado(char c, unsigned char arr[], int *length);
13
14    //Armazena glifo de "c" em "arr[]" (no formato textura GL_RGBA)
15    void getCharGL(char c, unsigned char arr[], unsigned char r,
16       unsigned char g, unsigned char b, unsigned char a);
17
18    //Imprime um glifo no console (feito de zeros e pontos)
19    void print(char c);
20    //Imprime no console todos os glifos gerados
21    void printAlphabet();
22    //Imprime no console os glifos originais da FreeType 2
```

```
23 ||    void printFreetypeOnConsole();
```

Código 4.6 – Métodos públicos definidos pela classe Font.

4.5 Envio de Glifos para o Odroid-Show

O programa SerialCom é responsável por enviar glifos para o Odroid-Show. Para utilizá-lo, deve-se definir a porta na qual o Odroid-Show está conectado como mostra o código 4.7. Além disso, é necessário fornecer um arquivo de fonte do modo Bit a Bit ou Alternado. No código 4.7 utiliza-se o arquivo "Courier New_Bold_36_BIT_A_BIT.txt", armazenado no diretório raíz do programa SerialCom.

```
1  int main()
2  {
3     printf("Welcome to the DSET Project!\n\n");
4     //Define porta a ser utilizada
5     Serial *SP = new Serial("COM6");
6     SP->setPort();
7
8     if (SP->IsConnected())
9        printf("We're connected");
10
11    //Da um tempo pro Odroid-Show pensar
12     Sleep(4000);
13
14     //Instancia objeto Font modo Bit a Bit
15     Font myFont("Courier New_Bold_36_BIT_A_BIT.txt", BIT_A_BIT);
16    //Envia um glifo para a posicao especificada
17     myFontB.print("G ",SP,40,40);
18
19     return 0;
20 }
```

Código 4.7 – SerialCom: função main().

4.6 Recebimento de Glifos no Odroid-Show

Para receber glifos no Odroid-Show deve-se realizar o upload da *sketch Modo_Bit_a_Bit.ino* ou *Modo_Alternado.ino* as quais referem-se ao modelo de arquivo utilizado para representação de fontes.

Em ambas as *sketches* emprega-se um array do tipo *unsigned char* como buffer. Dessa forma, armazena-se no buffer o glifo recebido e, em seguida, o mesmo é impresso na tela. O processo é repetido a medida que novos glifos são recebidos.

O tamanho do buffer é definido por *#define MAX_GLYPH_LENGTH*. Como o Odroid-Show possui 2Kbytes de memória para armazenamento de variáveis, sugere-se que o valor de MAX_GLYPH_LENGTH seja alterado conforme a necessidade.

4.7 Protocolo de Comunicação do Modo Alternado

A fim de desenvolver um protocolo de comunicação sucinto, porém suficientemente abrangente para cobrir atributos como posição e dimensões do glifo, definiu-se a seguinte formatação para representação das mensagens:

{Início de Mensagem, Posição (x,y), Dimensões do Glifo, Glifo, Fim de Mensagem}

onde:

- **Começo de Mensagem**: representado pelo byte 254.
- **Posição (x,y)**: 4 bytes representando a posição *(x,y)* onde o glifo será impresso na tela do Odroid-Show.
- **Dimensões do Glifo**: 2 bytes, um indicando a largura e outro a altura do glifo.
- **Glifo**: o array de bytes que representa o glifo.
- **Fim de mensagem**: representado pelo byte 255.

Inicia-se o envio de glifos ao Odroid-Show enviando o byte 254. Ao receber os 4 bytes seguintes, o dispositivo soma os 2 primeiros para obter a posição do glifo no eixo x e faz o mesmo com os 2 últimos para obter a posição no eixo y. Os 2 bytes seguintes são armazenados como largura e altura do glifo. Por fim, recebe-se o array de bytes que representa o glifo e o byte 255 indicando o fim do mesmo.

4.8 Protocolo de Comunicação do Modo Bit a Bit

Devido ao fato de impossibilitar que bytes específicos sejam reservados para comunicação, o modo Bit a Bit limita o desenvolvimento de um protocolo semelhante ao do modo Alternado. Ademais, os quadros e gráficos apresentados no capítulo 5 demonstram o quão inferior o modo Bit a Bit é em relação ao Alternado. Em virtude disso, não foi definido um

protocolo de comunicação para esse modo. É necessário definir na sketch do Odroid-Show a *length* e posição na qual deseja-se imprimir glifos recebidos.

O recebimento de glifos no modo Bit a Bit consiste em armazenar no buffer de comunicação os bytes recebidos e imprimí-los quando a *length* do mesmo é atingida.

SUMÁRIO DO CAPÍTULO

Este capítulo expôs o processo de criação dos glifos através da FreeType 2. Em seguida foi abordado o processo de tradução, para os modos Bit a Bit e Alternado, da informação referente aos bitmaps dos glifos. Também foi definido o protocolo de comunicação e o método de transmissão de glifos para o Odroid-Show. Tais tópicos são fundamentais para aqueles que desejam fazer uso do presente sistema.

5 RESULTADOS

Este capítulo expõe resultados de testes de desempenho das fontes geradas pelos modos Bit a Bit e Alternado. Os mesmos foram analisados em relação aos seus custos de processamento, armazenamento e transmissão via porta serial.

5.1 Dados de Processamento das Fontes

O quadro 5.1 compara o tempo de renderização e o tempo de envio para diferentes modos de fonte. Utilizou-se a fonte Courier New Bold, tamanho 36, 33x48 pixels e a fonte nativa do Odroid-Show, tamanho 5, o qual refere-se a 50 pixels de altura.

O tempo de renderização da fonte nativa do Odroid-Show mostrou-se inferior aos demais modos de fonte. Tal resultado é esperado, pois a definição dos glifos nativos do Odroid-Show é muito inferior à dos modos Bit a Bit e Alternado. Dessa forma, há mais dados a serem processados nos modos Bit a Bit e Alternado do que pela fonte nativa do dispositivo. Um glifo da fonte Courier New Bold 36 no modo Bit a Bit por exemplo, ocupa 199 bytes. No método de renderização do glifo Bit a Bit itera-se sobre cada bit dos 199 bytes. Tal fato implica em tempo de renderização diretamente proporcional a *length* do glifo.

Quadro 5.1 – Testes de desempenho: fonte Courier Bold, tamanho 36, 33x48 pixels.

Teste	**Bit a Bit**	**Alternado**	**Fonte Nativa do Odroid-Show**
Tempo de Renderização de 10 Glifos	144ms	64ms	23ms
Tempo de Envio de 10 Glifos	279ms	123ms	-

5.2 Dados de Armazenamento das Fontes

Os quadros 5.2 a 5.5 mostram o custo em bytes para armazenamento de diferentes tipos e tamanhos de fontes. Cada quadro possui um gráfico associado para melhor visualização.

Observou-se que para cada tipo de fonte, até um determinado tamanho, o modo Bit a Bit apresenta menor custo de armazenamento comparado ao modo Alternado. O tamanho no qual ambos os modos apresentam custo semelhante varia dependendo do modelo e estilo de fonte. No quadro 5.2 observa-se que o modo Bit a Bit é mais eficiente até o tamanho 9, no tamanho 12 ambos os modos possuem custo semelhante e, a partir de então o modelo Alternado

se mostra cada vez mais eficiente a medida que o tamanho da fonte aumenta. Tal resultado é esperado, visto que o modo Alternado opera sobre sequências de pixels de *background* e *foreground*. Dessa forma, quanto maiores essas sequências, maior a compactação obtida pelo modo Alternado.

Quadro 5.2 – Courier Bold: comparativo de custos de armazenamento para os modos Bit a Bit e Alternado.

Tamanho da fonte	**Dimensões (pixels)**	***Length* de glifo no modo Bit a Bit (bytes)**	**Maior *length* de glifo Alternado (bytes)**	**Alfabeto Bit a Bit (bytes)**	**Alfabeto Alternado (bytes)**
6	6x10	8	25	760	1287
9	*8x12*	*13*	*33*	*1235*	*1605*
12	*11x16*	*23*	*41*	*2185*	*2184*
15	13x20	33	57	3135	2986
18	16x24	49	67	4655	3468
21	19x29	69	81	6555	4268
24	22x33	91	93	8645	5014
27	24x36	109	103	10355	5458
30	27x40	136	115	12920	6127
40	35x53	232	161	22040	8395
50	46x68	392	209	37240	11049
60	55x82	564	261	53580	13641
70	62x93	721	295	68495	15585
80	73x108	986	351	93670	18705
90	82x121	1241	397	117895	21319
100	91x135	1536	445	145920	24129

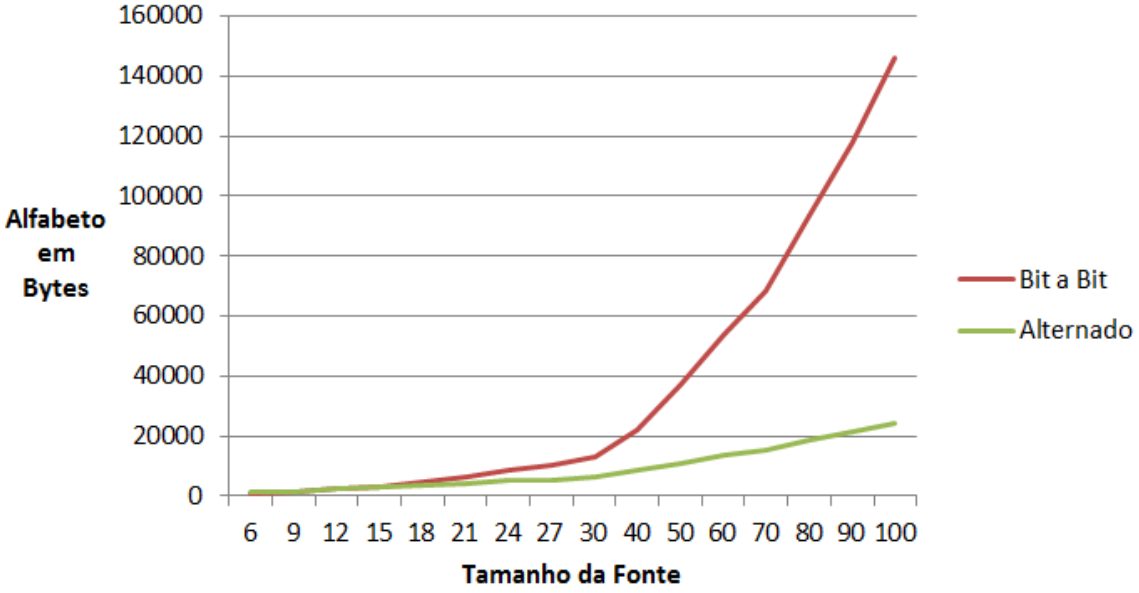

Figura 5.1 – Gráfico do espaço de armazenamento ocupado pela fonte fonte Courier Bold nos modos Bit a Bit e Alternado.

Quadro 5.3 – Ubuntu Mono Bold: comparativo de custos de armazenamento para os modos Bit a Bit e Alternado.

Tamanho da fonte	**Dimensões (pixels)**	***Length* de glifo no modo Bit a Bit (bytes)**	**Maior *length* de glifo Alternado (bytes)**	**Alfabeto Bit a Bit (bytes)**	**Alfabeto Alternado (bytes)**
6	6x6	5	19	475	1123
9	8x9	10	31	950	1743
12	9x15	17	45	1615	2379
15	11x15	21	53	1995	2925
18	14x22	39	75	3705	3575
21	15x26	49	89	4655	4331
24	17x28	60	95	5700	4637
27	19x32	77	107	7315	5289
30	21x34	90	121	8550	5915
40	27x48	163	165	15485	8281
50	33x59	244	209	23180	10345
60	40x71	356	259	33820	12769
70	46x82	472	305	44840	14863
80	54x95	642	353	60990	17347
90	60x106	796	393	75620	19569
100	66x117	966	437	91770	21851

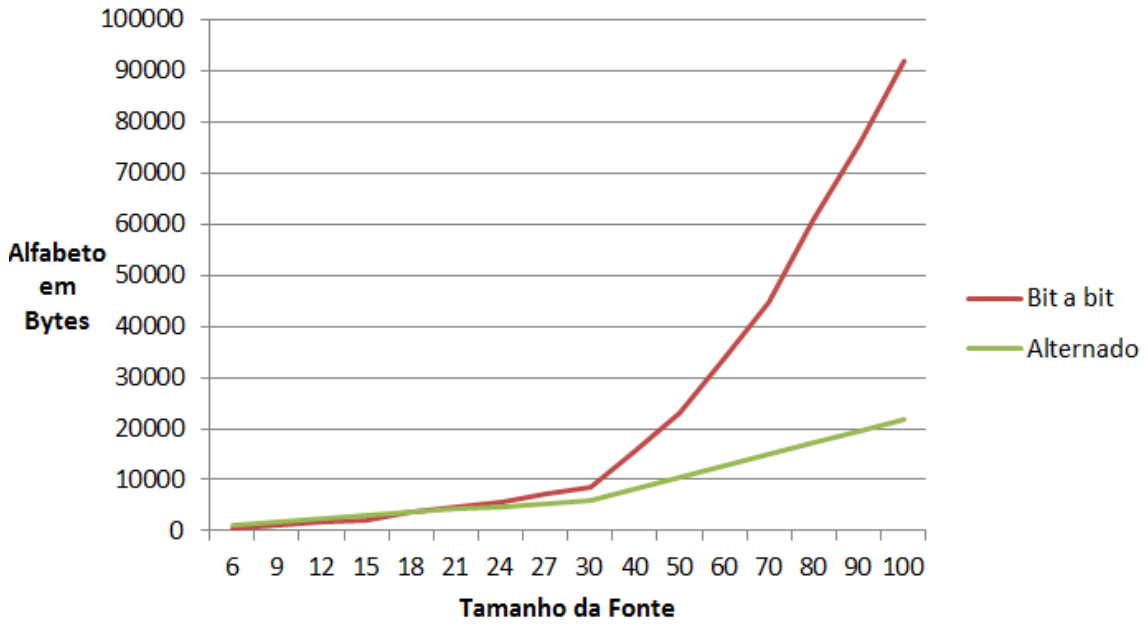

Figura 5.2 – Gráfico do espaço de armazenamento ocupado pela fonte fonte Ubuntu Mono Bold nos modos Bit a Bit e Alternado.

Quadro 5.4 – Consolas Regular: comparativo de custos de armazenamento para os modos Bit a Bit e Alternado.

Tamanho da fonte	**Dimensões (pixels)**	***Length* de glifo no modo Bit a Bit (bytes)**	**Maior *length* de glifo Alternado (bytes)**	**Alfabeto Bit a Bit (bytes)**	**Alfabeto Alternado (bytes)**
6	5x8	6	21	570	1209
9	8x12	13	45	1235	1955
12	10x16	21	53	1995	2585
15	12x20	31	77	2945	3227
18	14x24	43	109	4085	3901
21	16x28	57	117	5415	4597
24	19x32	77	137	7315	5221
27	21x36	95	161	9025	5895
30	23x40	116	177	11020	6671
40	29x52	189	231	17955	8806
50	37x66	306	301	29070	11415
60	45x80	451	359	42845	14133
70	51x92	587	421	55765	16461
80	59x106	782	485	74290	19205
90	67x120	1006	551	95570	22149
100	73x132	1205	603	114475	24557

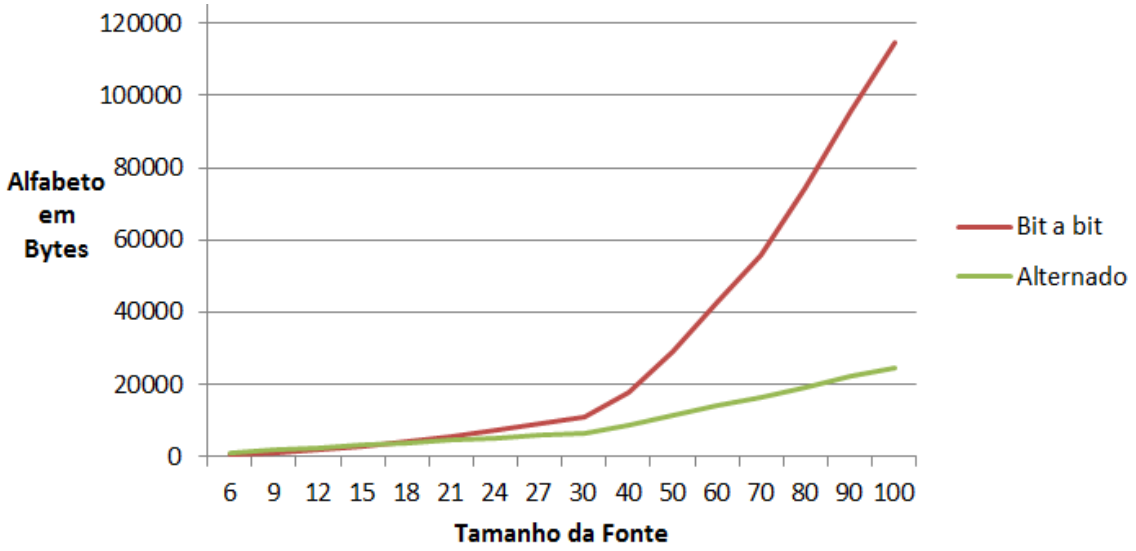

Figura 5.3 – Gráfico do espaço de armazenamento ocupado pela fonte fonte Consolas Regular nos modos Bit a Bit e Alternado.

Quadro 5.5 – Lucida Console Regular: comparativo de custos de armazenamento para os modos Bit a Bit e Alternado.

Tamanho da fonte	**Dimensões (pixels)**	***Length* de glifo no modo Bit a Bit (bytes)**	**Maior *length* de glifo Alternado (bytes)**	**Alfabeto Bit a Bit (bytes)**	**Alfabeto Alternado (bytes)**
6	5x6	4	15	380	795
9	9x10	12	43	1140	2107
12	11x13	18	63	1710	2763
15	13x16	27	71	2565	3327
18	15x19	36	93	3420	3911
21	18x22	50	107	4750	4587
24	20x25	63	119	5985	5329
27	23x29	84	139	7980	6161
30	25x32	101	149	9595	6797
40	33x42	174	203	16530	9081
50	41x53	272	253	25840	11419
60	48x63	379	307	36005	13755
70	56x73	512	361	48640	16159
80	66x85	702	417	66690	18883
90	72x95	856	469	81320	21311
100	79x104	1028	521	97660	23675

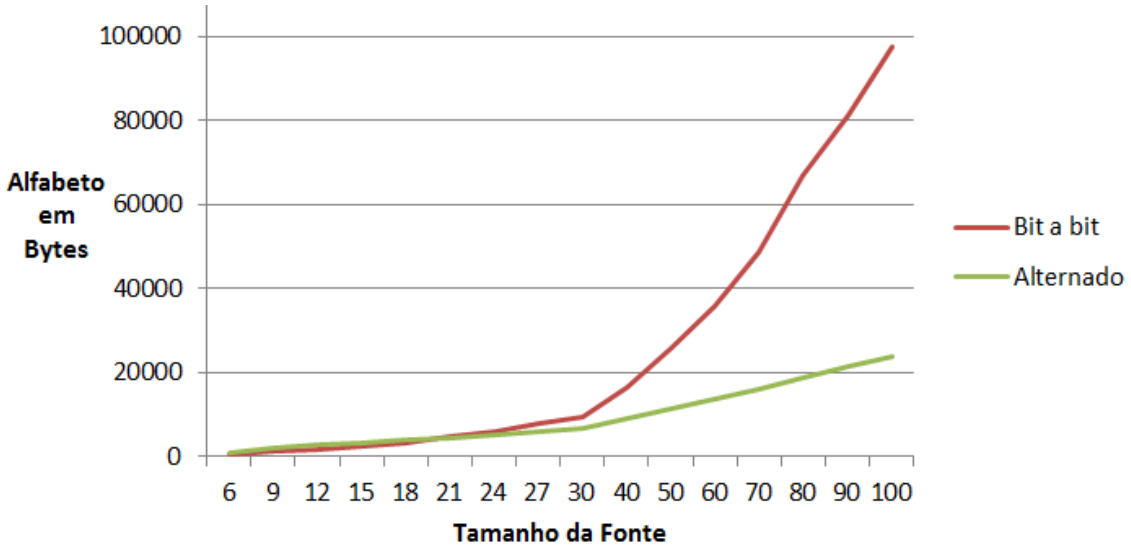

Figura 5.4 – Gráfico do espaço de armazenamento ocupado pela fonte fonte Lucida Console Regular nos modos Bit a Bit e Alternado.

6 CONCLUSÃO

Uma vez que os objetivos de desenvolver e implementar um formato de arquivo de fonte de texto, baseado em glifos representados por bitmaps, respeitando as limitações de memória e processamento existentes no Odroid-Show, foram todos atingidos, pode-se considerar que o presente trabalho alcançou os resultados desejados. Foi desenvolvido e implementado um sistema de fonte de alta definição para o Odroid-Show. Ademais, pôde-se realizar diversos testes e comparações de desempenho para armazenamento, renderização e transmissão de glifos.

O processo de geração e envio de fontes bitmap compactadas, criadas a partir de fontes vetoriais, é a principal contribuição dessa pesquisa, pois possibilita o uso eficiente de glifos com definição próxima a de modelos vetoriais e utiliza o reduzido custo de renderização oferecido por fontes bitmap.

Apesar dos objetivos terem sido alcançados, este trabalho abre margens para implementação de novas funcionalidades e otimizações para o sistema. Em relação a novas funcionalidades sugere-se a inclusão de fontes não monoespaçadas, além de métodos para rotacionar e colorir glifos. Em termos de otimizações, é possível estudar outras formas de compactação e compará-las aos modelos propostos.

REFERÊNCIAS

Arduino. **Arduino Website**. https://www.arduino.cc, Acesso em Agosto de 2015.

ASCIITable.com. **ASCII Table and Description**. http://www.asciitable.com/, Acesso em Agosto 2015.

Blog do Exército Brasileiro. **A Simulação como Ferramenta no Adestramento da Tropa**. http://eblog.eb.mil.br/index.php/noticias/4128-a-simulacao-como-ferramenta-no-adestramento-da-tropa, Acesso em Agosto 2015.

FreeType 2. **FreeType 2 Documentation**. http://freetype.org, Acesso em Agosto 2015.

GLOBACNIK, T.; ZALIK, B. An Efficient Raster Font Compression for Embedded Systems. **Pattern Recognition Journal**, New York, NY, USA, v.43, n.12, p.4137–4147, Dec. 2010.

HARALAMBOUS, Y.; HORNE, P. S. **Fonts & Encodings**. [S.l.]: O'Reilly Media, Inc., 2007.

Odroid.com. **Odroid-Show Hardware**. http://odroid.com/dokuwiki/doku.php?id=en:show_hardware, Acesso em Agosto 2015.

RUSSELL, D. **Introduction to Embedded Systems**: using ansi c and the arduino development environment. [S.l.]: Morgan and Claypool Publishers, 2010.