Laboratório de processadores: PCS3732

Projeto Low Level Tetris

João Pedro Arroyo	12550991	joao.arroyo@usp.br
Tiago Mariotto Lucio	12550556	tiagolucio@usp.br
Vinicius Viana de Paula	12550650	viniciusviana@usp.br

17 de agosto de 2024

Sumário

1 Contexto e motivação

O contexto base do projeto é definido, em linhas gerais, pelo conteúdo abordado na disciplina laboratório de processadores (PCS3732): ISA e estrutura do processador ARM, programação em *assembly* para processadores ARM, linguagem C (com compilação para ARM) e uso da Raspberry Pi. Diante destas possibilidades, posto que o grupo, desde o início, pensou em desenvolver alguma aplicação com um grau de complexidade relativamente alto, as ideias foram pautadas na programação em C e, eventualmente, no uso da Raspberry Pi.

Ocorreu que, no processo de levantamento de ideias, os projetos elencados que não utilizavam a placa foram avaliados pelo grupo como pouco relacionados à disciplina. Em geral, pensou-se em algumas emulações (como a de um sistema de memória, de um *job scheduler* com multiprogramação ou de um SO completo), porém após melhor considerar percebeu-se que (i) tais simulariam programas fantasma e seriam pouco fidedignas e (ii) o único motivo para fazer a emulação e não o sistema em si (e implementar na Raspberry Pi, por exemplo) era a dificuldade elevada do segundo caso. Dessa forma, motivou-se a escolha de um projeto que fizesse uso da Raspberry, rodando sem qualquer sistema operacional. Assim, buscou-se unir a maior facilidade de se programar em C em vez de *assembly* com uma maior proximidade à disciplina ao usar a Raspberry.

Com base no contexto da disciplina e das decisões tomadas pelo grupo explicadas acima, a primeira ideia em que se investiu um maior tempo foi a elaboração de um *driver* USB em si. Porém, após análise mais cuidadosa, o grupo concluiu que seria inviável considerando o escopo do projeto da disciplina. Assim, como consequência, juntou-se a ideia do tetris a um escopo modificado: implementar o jogo na linguagem C, porém carregando-o na Raspberry, e utilizar teclado e monitor a partir de bibliotecas prontas que tivessem os respectivos *drivers* implementados. Tal possibilidade tornaria viável o incremento de dispositivos externos ao projeto, necessários para permitir a jogabilidade e visualização da aplicação, assim agregando valor, porém sem a necessidade de um trabalho incompatível com o projeto (i.e a elaboração dos *drivers* em si). A figura 1 apresenta um diagrama que sistematiza a ideia base do projeto.

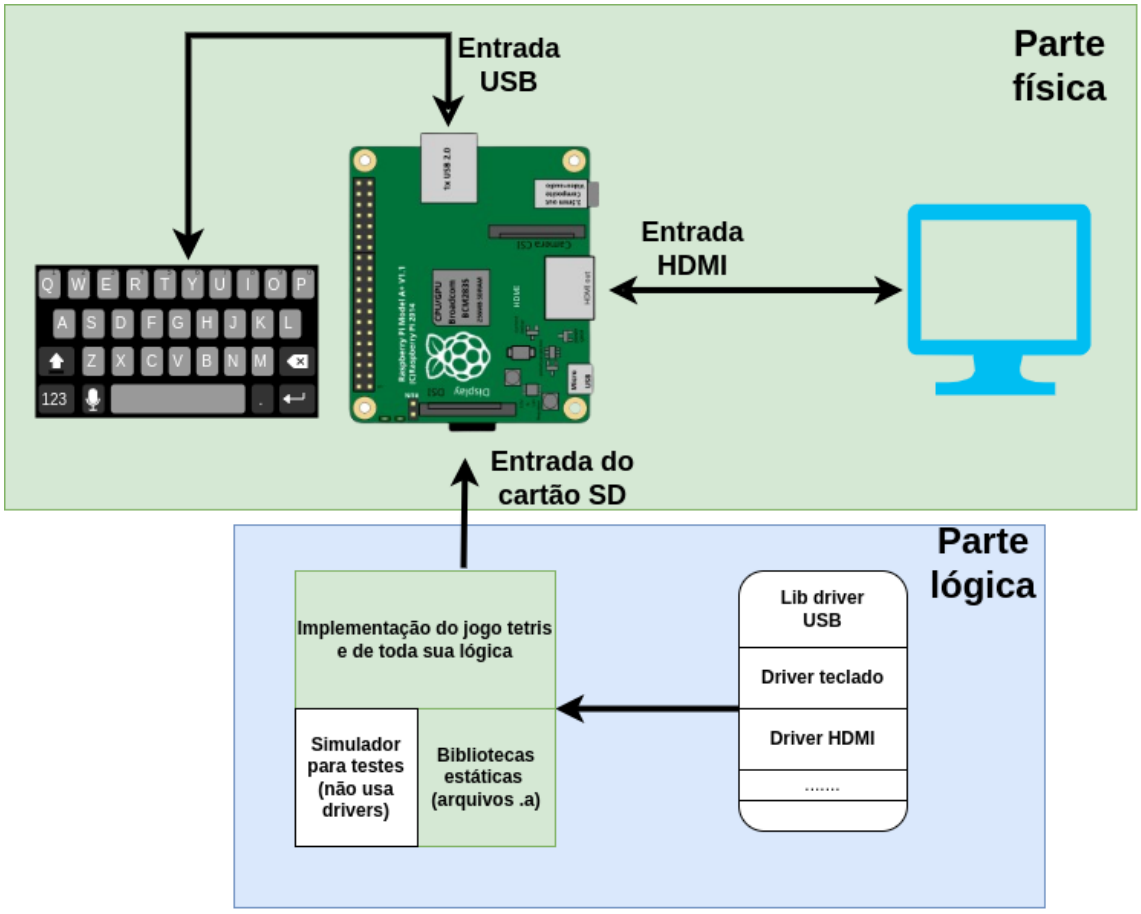

Figura 1: Diagrama com os componentes básicos necessários para o desenvolvimento do projeto.

Em seguida, detalham-se e reiteram-se alguns dos pontos principais, justificando-se decisões de projeto.

1.1 Uso de um *driver* USB para a Raspberry Pi

Para possibilitar a interação do jogador utilizando o teclado, a alternativa considerando o desenvolvimento na Raspberry Pi *bare metal* é a utilização de um *driver* USB. Desse modo, as interações de movimentação dos blocos, de rotação, entre outras, tornam-se possíveis por meio das teclas definidas.

Considerando a inclusão de um *driver* USB para possibilitar a utilização do teclado, o objetivo inicial foi a identificação de possíveis *drivers* já existentes, desenvolvidos especialmente para o uso *bare metal* na Raspberry Pi. Um projeto de um *driver* USB é muito extenso (mesmo adotando grandes simplificações), de forma que se torna adequado tomar um projeto já desenvolvido como referência.

1.2 Simulador

Desde a primeira semana em que se definiu o projeto, o grupo notou que haveriam duas principais frentes de trabalho. Uma seria no desenvolvimento da lógica do jogo em si (que como será abordado em seções adiante possui suas próprias complexidades) e outra seria em encontrar um *driver* que atendesse as necessidades do grupo e em como incorporá-lo ao projeto, isto é, possibilitar que seja corretamente carregado na Raspberry Pi.

Ocorre que para de fato paralelizar o trabalho deve-se permitir um desenvolvimento com o menor acoplamento possível entre tais tarefas. Assim, surge a necessidade de que, de alguma forma, possa-se testar a lógica do jogo sem precisar-se da raspbery Pi, ou seja, apenas compilando o código C. Com relação ao teclado, a alteração é simples e consiste apenas em trocar métodos do *driver* pela leitura com `scanf`. Por outro lado, a questão gráfica em C exige um maior trabalho, e será detalhada na parte de soluções.

1.3 Bibliotecas estáticas

Como comentado, usou-se um *driver* existente, porém o código do *driver* em si é escrito em C. Logo, deve-se de alguma forma ser capaz de gerar arquivos das bibliotecas estáticas utilizadas (formato `.a`). Além disso, no meio do desenvolvimento do projeto, notou-se que outras bibliotecas poderiam ser adicionadas para simplificar o desenvolvimento. Destaca-se que o maior esforço quanto a este ponto consiste em descobrir como gerar o arquivo corretamente para que funcione ao carregar-se o código na Raspberry.

1.4 Placa Raspberry, monitor e teclado

Como já explicitado anteriormente, são necessários uma placa Raspberry Pi, um monitor com conexão HDMI e um teclado quanto a componentes físicos para que o projeto possa ser posto em funcionamento. Um dos integrantes do grupo utilizou sua própria Raspberry Pi 3, e no laboratório usaram-se as Raspberry Pi 2 disponíveis. Nota-se que, em ambos os casos, tem-se a entrada USB e HDMI, bastando ter-se os outros componentes. O resto da operação fica por responsabilidade do *driver*.

2 Objetivos

Nesta seção, explicitam-se os objetivos do projeto e mapeiam-se os requisitos definidos pelo grupo. Em primeiro lugar, descreve-se sucintamente o objetivo como: desenvolver o jogo tetris (com mecânicas fiéis ao original) e implementá-lo na Raspberry Pi, utilizando-se os dispositivos externos - teclado e monitor - para tal. Já de um ponto de vista didático, pode-se citar:

- Melhor entendimento do carregamento de códigos C na Raspberry Pi.
- Melhor compreensão acerca da conexão com os dispositivos externos, por meio da prática com funções do *driver*.
- Familiarização com um *driver* USB para a Raspberry Pi *bare metal*, possibilitando diferentes gamas de projetos para a placa.
- Conhecimento de um projeto de *driver*.
- Prática de progamação em C e de lógica de programação em geral com o desenvolvimento do jogo.
- Utilização de diferentes bibliotecas estáticas, possibilitando aprendizado prático no processo de linkagem.

Já com relação aos requisitos, faz-se uma separação. Primeiro, listam-se os requisitos considerados cruciais, isto é, que englobam o MVP (produto mínimo viável):

- Desenvolvimento da lógica básica do jogo tetris, envolvendo: definição dos diferentes tipos de peças e sua geração, queda das peças a uma dada velocidade, colisões horizontal e vertical (com blocos e com o limite do *frame*), rotação das peças e colisão em rotação, sistema que limpa as linhas quando completas e desloca as demais peças para baixo.
- Implementação de multiprogramação para gerenciar escrita pelo teclado (movimentação das peças) e processamento do jogo em si ao mesmo tempo.
- Definição de um *driver* USB, apropriado para o uso na Raspberry, a ser utilizado no projeto (com suporte a teclado).
- Uso do *framebuffer* da Raspberry Pi, possibilitando o uso de sua GPU para desenhar na tela utilizando a interface HDMI.
- Integração do *driver* à placa Raspberry Pi, resolvendo-se a questão das bibliotecas estáticas.
- Definição de biblioteca para simulação e implementação da estrutura base para se poder testar via código C o jogo tetris sem precisar da Raspberry.
- Integração do jogo (inicialmente desenvolvido à parte) com *driver* USB, interface HDMI e a placa Raspberry.
- Compilação automatizada do projeto, facilitando o processo de geração dos arquivos de imagem e disco.
- Disponibilidade para as Raspberry Pi 2 e 3.

Por fim, listam-se os requisitos vistos como desejáveis, mas não necessários. Nota-se que, destes, alguns foram definidos apenas conforme o decorrer do projeto, levando-se em conta o prazo restante e o que poderia ser implementado.

- Uso de bibliotecas estáticas adicionais, para possibilitar o uso das funções `rand` e `srand`, e do tipo `bool`.
- Implementação de aleatoriedade do tipo de peça e das cores das peças.
- Melhoria quanto à paleta de cores: descobrir como gerar cores diversas, utilizando o próprio projeto do *driver* USB (que apresenta definições de ambiente adicionais e externas ao *driver*, nesse caso visando a impressão de pixels na tela), em vez de usar apenas as poucas pré prontas.
- Implementação de movimento para baixo (tecla `s`) que acelera a queda.

- Sistema de níveis conforme o jogador completa um certo número de linhas.
- Alterar a velocidade de queda conforme o jogador sobe de nível, dificultando o jogo.
- Implementar *score* e estatísticas, mostrando, no monitor, o nível, quantas peças de cada tipo apareceram, qual é a próxima peça, qual a pontuação atual e quantas linhas já foram fechadas, além de uma aba de histórico.

Vale notar que, embora a quantidade de requisitos desejáveis possa parecer grande à primeira vista, na prática os gargalos principais mapeados encontraram-se quase que inteiramente nos requisitos do MVP, de modo que cada requisito desejável pôde ser implementado em menos tempo, eventualmente sendo mais trabalhoso, porém não se atingindo nenhum bloqueio de desenvolvimento.

3 Modelo do Problema

Nesta seção, explicam-se os principais gargalos identificados no desenvolvimento e como o grupo os mapeou, para então explicar-se a solução na seção 4. Analisando a imagem 1, pode-se identificar a necessidade de se possibilitar a comunicação via USB, o desenho na tela utilizando a interface HDMI, e o desenvolvimento da parte lógica do jogo. Inicialmente, considerando a necessidade de integração com o teclado, para possibilitar as interações do jogador, serão apresentadas as dificuldades encontradas para a escolha do *driver* USB a ser utilizado pelo projeto. Em sequência, é discutida a utilização do *framebuffer* da GPU da Raspberry Pi para se possibilitar a integração com o monitor. Finalmente, discorre-se acerca da lógica do tetris, identificando-se os principais desafios encontrados.

3.1 Driver USB para a Raspberry Pi

Para a integração do *driver* USB, visando possibilitar o uso de um teclado para as interações do jogador, foram identificados três principais desafios:

- Complexidade de um projeto já existente de *driver*;
- Compatibilidade com diferentes modelos de Raspberry Pi e com o próprio teclado;
- Depuração e identificação dos problemas que surgiram nas tentativas de uso do *driver* para a utilização do teclado.

3.1.1 Complexidade do projeto já existente

Analisando possíveis projetos de *driver* USB para Raspberry Pi, foram estudadas três alternativas. Além da compreensão em termos gerais de cada um dos projetos, geralmente foi preciso efetuar a geração de um arquivo de biblioteca estática (`.a`), além de ajustes pontuais visando a compatibilidade com o modelo de Raspberry Pi utilizado. O projeto adotado, o USPi[7], apresenta duas bibliotecas estáticas, assim como explicado em 7.1.1, e felizmente também apresenta alguns códigos de exemplo, de forma que favoreceu a verificação da conexão do teclado com a Raspberry Pi.

A complexidade deste tipo de projeto pode ser facilmente verificada analisando o repositório do projeto de cada *driver* USB para a Raspberry Pi. Por exemplo, o projeto adotado apresenta 247 arquivos, considerando os 45 reunidos nos exemplos. O USPi está apresentado em termos gerais na seção 7.1, com destaque ao que é utilizado no projeto do Low Level Tetris.

3.1.2 Compatibilidade

Inicialmente, pode-se considerar a dificuldade em termos da compatibilidade com os diferentes modelos de Raspberry Pi. Em um primeiro momento, é necessário que o *driver* USB a ser utilizado seja compatível com a Raspberry Pi a ser utilizada, neste caso, a Raspberry Pi 2 e a Raspberry Pi 3. Por exemplo, um dos *drivers* estudados foi desenvolvido pensando-se no caso de uso de uma Raspberry Pi Zero, de modo que os ajustes necessários para que pudesse atender outros modelos seriam inviáveis no contexto deste projeto.

Considerando por exemplo o uso do endereço de periféricos, que é diferente entre os modelos de Raspberry Pi, é essencial identificar os modelos cobertos pelo *driver* USB utilizado.

Tomando este exemplo dos endereços físicos, comparando entre BCM2835 (utilizado nas Raspberry Pi Zero e Raspberry Pi 1, por exemplo) e BCM2837 (utilizado na Raspberry Pi 3 modelo B)[4]: para o chip Broadcom BCM2835 os endereços físicos variam de 0x20000000 a 0x20FFFFFF para periféricos[8], enquanto para o chip BCM2837, eles variam de 0x3F000000 a 0x3FFFFFFF[1].

No geral, também é necessário identificar a versão ARM utilizada e a CPU da Raspberry Pi, reforçando o fato de que a implementação do *driver* é dependente do modelo da Raspberry Pi e, a menos que tenha sido planejado para lidar com múltiplos modelos, existe a possibilidade do projeto do *driver* USB não atender a Raspberry Pi para a qual se deseja utilizá-lo.

Tais características são passadas como *flags* ao compilador utilizado no projeto. Como exemplo para a utilização da Raspberry Pi Zero, estas características poderiam ser passadas como: `-march=armv6` e `-mtune=arm1176jzf-s`; já para a Raspberry Pi 2, Estas *flags* teriam os valores `-march=armv7-a -mtune=cortex-a7`.

Além disso, considerando que o *driver* USB desenvolvido é simplificado, e que diferentes teclados podem apresentar algumas funcionalidades adicionais que resultam na necessidade de algumas peculiaridades em seus *drivers*, é possível que o *driver* funcione para alguns teclados, e outros não (este caso, inclusive, foi observado e está registrado na seção 7.1.2).

3.1.3 Depuração

Adicionalmente, outra grande dificuldade ao se utilizar quaisquer dos *drivers* USB para a Raspberry Pi estudados foi a depuração e identificação dos problemas que surgiram nas tentativas de uso do *driver* para a utilização do teclado. Isso porque a única alternativa para depuração, obtida sem maiores ajustes na biblioteca estática do *driver* (em uma das alternativas, existia a opção de configuração `DEBUG` que possibilitaria facilitar o processo, mas dependeria de mais ajustes), seriam os *logs* visualizados após a inicialização da Raspberry Pi com o código designado. As imagens 2 e 3, abaixo, apresentam estes registros:

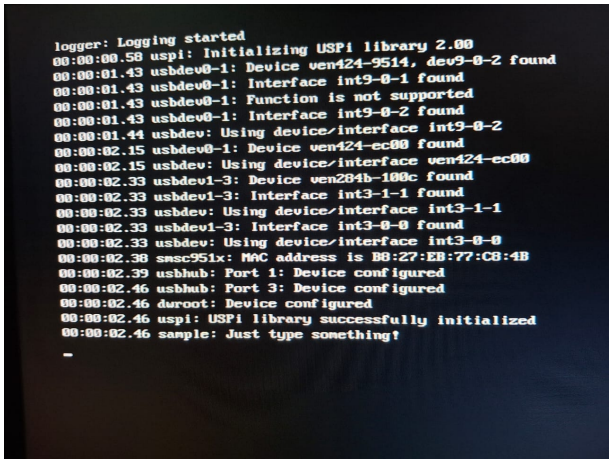

Figura 2: *Log* para o caso em que o teclado é encontrado e é possível escrever na tela.

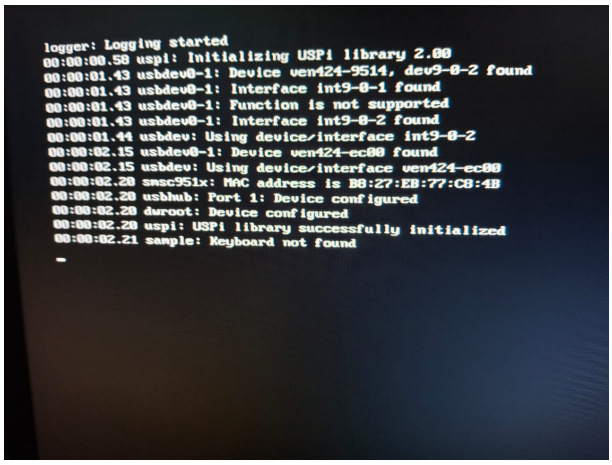

Figura 3: Demonstração do *log* para o caso em que o teclado não é identificado (não há teclados conectados ou o teclado é incompatível).

Assim, considerando o caso em que o teclado não é encontrado, não há como identificar possíveis problemas. Inclusive, considerando que o *driver* USB desenvolvido é simplificado, assim como mencionado em 3.1.2, teclados incompatíveis não são reconhecidos e este é o erro que será explicitado nos *logs* de inicialização.

3.2 Integração com o monitor

Para a integração com o monitor utilizando interfaceamento HDMI, com a finalidade de se exibir o jogo na tela, identificaram-se os seguintes principais problemas:

- Compreender métodos básicos usados para desenhar na tela e criar métodos para desenhar formas mais complexas.
- Entender como são definidas as proporções da tela para que se possa fazer um desenho de tamanho e proporções adequadas ao monitor.
- Como gerar uma paleta de cores mais diversa.
- Compreender a utilização do *framebuffer* da GPU da Raspberry Pi para os desenhos na tela.

Na implementação das definições de ambiente adicionais e externas ao *driver* USB, no repositório do USPi, detalhadas na seção 7.1.1, já está contida a utilização de funções que se encarregam da impressão dos *pixels* designados na tela. O uso de tais funções será melhor apresentado na seção 4.2 e nos apêndices.

3.3 Desenvolvimento da lógica do tetris

Um dos dois componentes principais do projeto é o código do jogo em si, que deve ser possuir o comportamento esperado do tetris. Abaixo listam-se os principais desafios encontrados na realização desta tarefa.

- Gerenciar a necessidade de se estar o tempo todo checando se o usuário usou o teclado para se movimento e também precisar estar o tempo todo processando as alterações no *grid*, posto que sempre há alguma peça se movendo - lembrando que não há múltiplas *threads* na implementação do grupo para a Raspberry.
- Implementação da colisão vertical, levando-se em conta peças do *grid* bem como a linha mais inferior do *grid*. Deve-se, ainda, implementar o sistema que fixa a peça no *grid* no momento em que ocorrer tal colisão.
- Implementação da colisão horizontal, levando-se em conta os limites do *grid* e outras peças. Nesse caso, não permitir o movimento, porém a peça segue caindo.
- Implementação da rotação das peças e checagem da colisão devido a rotação. Como lidar com o fato de cada peça ter um formato particular? Dessa forma uma função que transforme a posição atual da peça em sua posição rotacionada de 90 graus é muito complicada e depende completamente do tipo de peça, e o mesmo vale para checar a colisão. Será que não há alguma forma simples de implementar isso?
- Diferenciar peças que estejam já fixadas no *grid* da peça que está caindo no momento. Como bônus, como garantir que a peça atualmente caindo não "colida com si mesma"de forma incorreta em rotações?
- Checar se há linhas completas (quantas linhas completas pode haver ao máximo?) apenas no fim de uma rodada. Em caso afirmativo, deve-se mover o *grid* para baixo de forma correspondente à quantidade de linhas fechadas pelo jogador. Como lidar com a necessidade de redesenhar boa parte do tabuleiro?

4 Modelo da Solução

Considerando os principais desafios discorridos na seção 3, a solução proposta para o projeto do jogo está dividida entre as decisões para possibilitar a utilização do teclado, para permitir a integração com o monitor e as decisões do desenvolvimento lógico do jogo em si. Dessa forma, considerando tais decisões de projeto, e sua integração final, obtém-se a versão final do projeto, atendendo o planejamento do diagrama 1.

4.1 Driver USB

Para a conexão do teclado na Raspberry Pi foi utilizado, conforme indicado na seção 3.1, o projeto do USPi como referência, com pequenos ajustes adicionais. A presença dos *logs* de inicialização do *driver* contribuiram para a inclusão de mensagens de depuração durante o desenvolvimento do projeto. O repositório deste projeto[5] desenvolvido para a disciplina apresenta no diretório `uspi` tudo o que é utilizado na integração necessária para as interações do jogador via teclado.

Desse modo, escolhido o *driver* USB adequado para a conexão necessária - e para a Raspberry Pi -, e tomados os devidos ajustes em sua integração com o projeto do jogo, tem-se a solução para o *driver* USB, importante requisito no desenvolvimento deste projeto.

Assim como indicado em 7.1.1, o núcleo do projeto do USPi é a biblioteca `libuspi.a`, contida no diretório `lib`, dentro do diretório `uspi` anteriormente mencionado. Outro diretório interno a este contém tudo o que precisa ser incluído no projeto (`include`) e, o último, contém os arquivos para as definições adicionais de ambiente (`env`).

Tais definições adicionais são desvinculadas do projeto do *driver* em si, mas foram implementadas por padrão para os códigos-exemplo presentes no repositório do USPi. Considerando que no caso de uso deste projeto o *driver* tem como principal papel apenas possibilitar o uso de algumas teclas para a movimentação e rotação de blocos (e reinício do jogo), optou-se por adotar estas definições já implementadas, que estão presentes na biblioteca estática `libuspienv.a`.

Dessa forma, basicamente tornam-se necessárias as duas bibliotecas estáticas já existentes, `libuspi.a` e `libuspienv.a` (com exceção de alguns arquivos de inicialização desta última), presentes em `uspi/lib` e já geradas para a Raspberry Pi 2 e para a Raspberry Pi 3 (a fim de incluir uma versão de depuração, os nomes encontrados nos respectivos diretórios diferem para a biblioteca contendo as definições de ambiente).

Considerando necessária (basicamente) apenas a linkagem destas bibliotecas estáticas, utiliza-se um *Makefile* visando automatizar o processo de geração do arquivo de imagem a ser escrito no cartão SD para posterior escrita na Raspberry Pi, e o seu processo de uso e passagem de parâmetros estão indicados na documentação do repositório.

4.2 Integração com o monitor

Assim como apresentado em 7.1.1, a biblioteca `libuspienv.a` já contém implementações suficientes para que seja possível adicionar desenhos na tela, de modo que a utilização do projeto do USPi facilitou a integração da Raspberry Pi com o monitor.

Para todos os desenhos utilizados no projeto do jogo, é utilizada para cada *pixel* necessário a função `ScreenDeviceSetPixel`, que já efetua o uso do `framebuffer` da GPU para possibilitar estes desenhos na tela, utilizando a interface HDMI.

4.3 Implementação da lógica do jogo

4.3.1 Simulador SDL

Como comentado, ter uma forma de simular o jogo sem a necessidade de usar os *drivers* era um dos pontos principais para desbloquear o desenvolvimento. Logo, a atuação em tal aspecto ocorreu nas primeiras semanas do projeto, em que se optou pelo uso do SDL. Para tal, foram temporariamente incluídos os arquivos necessários contendo o código da biblioteca

4.3.2 Geração de sequências de blocos aleatórias

Para garantir melhor experiência para os jogadores, é necessário que a cada início de jogo as sequências de blocos geradas sejam distintas, e preferencialmente aleatórias.

Nesse sentido, a solução proposta para a inclusão desta aleatoriedade no projeto é a geração de *seeds* aleatórias, vista como necessária para que a sequência de blocos gerados em cada jogo seja distinta e independente.

Para a utilização de tais sequências aleatórias, optou-se por utilizar a função `rand`[3] da `stdlib`. A fim de possibilitar a inclusão da `stdlib`, foi utilizada a `newlib`.

Por outro lado, para gerar a *seed* de forma aleatória, utilizada na função `srand` da `stdlib`, geralmente se utiliza a função `time` para que a *seed* seja imprevisível. A função `time()` retorna a hora atual do calendário como um único valor aritmético[6]. Nesse sentido, é imediato perceber que adotar tal função na Raspberry Pi *bare metal* seria inviável e, por isso, adotou-se o próprio *uptime* da Raspberry Pi.

Para possibilitar o uso das funções definidas no `stdlib.h`, é necessário ter entre os parâmetros, no *Makefile*, passados para o compilador a indicação de um local onde estão presentes os includes, ou seja, onde estão os arquivos-cabeçalhos `.h`. Como o `stdlib.h` também necessita de outros arquivos-cabeçalhos, é adequado já utilizar todos os *includes* do `newlib`, que podem estar presentes, no Linux, em `/usr/include/newlib/`. Ademais, é necessário utilizar alguns arquivos `.h` que são incluídos pelo próprio compilador (no caso, utiliza-se o `arm-none-eabi-gcc`):

- Cabeçalhos da Biblioteca Padrão C (Newlib):
  - A *newlib* fornece a biblioteca padrão C para sistemas embarcados. Ela inclui cabeçalhos essenciais como `stdio.h`, `stdlib.h`, `string.h`, `time.h`, etc. Esses cabeçalhos definem funções padrão, tipos e macros necessários para a programação em C, como *printf*, *malloc*, e *memcpy*.
- Cabeçalhos Específicos da *toolchain* do ARM
  - A *toolchain* do ARM fornece cabeçalhos específicos da arquitetura que definem tipos, macros e funções *inline* que estão intimamente ligados à arquitetura ARM. Esses cabeçalhos podem incluir definições para funções intrínsecas, acesso ao hardware ou outras funcionalidades de baixo nível específicas para processadores ARM. Por exemplo, inclui cabeçalhos como `stdint.h` e `stddef.h`.

Para o uso da `stdlib` é preciso utilizar os tipos definidos no `stdint.h`, para o qual precisa-se incluir algumas definições de tipos.

Além disso, foram necessárias algumas implementações para fornecer as funções básicas de entrada/saída e gerenciamento de memória em sistemas embarcados ou ambientes onde a biblioteca padrão C (`libc`) não pode fornecer essas funcionalidades diretamente, de modo que foram exigidas para o uso da *newlib* no projeto:

- `_write`: Necessária para redirecionar a saída padrão (*stdout*) para um dispositivo específico, como uma UART.
- `_sbrk`: Usada para gerenciar a alocação de memória dinâmica (*heap*). É chamada pelo *malloc* e outras funções de alocação de memória.
- `_close`: Fecha um arquivo.
- `_read`: Lê dados de um arquivo ou dispositivo. Pode ser redirecionado para ler de uma UART ou outro dispositivo de entrada.
- `_lseek`: Move o ponteiro de leitura/escrita em um arquivo.
- `_fstat`: Obtém informações sobre um arquivo.
- `_isatty`: Verifica se um descritor de arquivo refere-se a um terminal.

Considerando o caso de uso neste projeto, estas funções não serão necessárias, então uma implementação simplificada foi adotada para elas, tendo apenas o retorno esperado de cada uma (com algumas exceções, apresentando algumas operações iniciais). No repositório, estas implementações estão presentes na raiz do projeto, no arquivo `syscalls.c`.

Para a geração da *seed* aleatória, optou-se por utilizar o próprio *uptime* da Raspberry Pi, conforme mencionado anteriormente. Isto porque o outro *timer* imediato existente no projeto, que é o tempo desde a inicialização do *Logger* do *driver* USB (USPi), resulta em tempos muito próximos até o início do jogo, e não apresenta uma precisão suficiente para distinguir cada inicialização. O uso do *uptime* está apresentado no arquivo `uptime.c`, também na raiz do projeto. Em termos da implementação deste arquivo, identifica-se o uso dos registradores do contador do *System Timer*, conforme apresentado na listagem abaixo:

```c
#include <stdint.h>
#include "uptime.h"

#if RASPPI == 1
#define SYSTEM_TIMER_BASE 0x20003000
#else
#define SYSTEM_TIMER_BASE 0x3F003000
#endif
#define TIMER_CLO_OFFSET  0x04
#define TIMER_CHI_OFFSET  0x08

uint64_t get_current_time(void) {
    volatile uint32_t *timer_clo = (uint32_t *)(SYSTEM_TIMER_BASE + TIMER_CLO_OFFSET);
    volatile uint32_t *timer_chi = (uint32_t *)(SYSTEM_TIMER_BASE + TIMER_CHI_OFFSET);

    uint32_t low = *timer_clo;
    uint32_t high = *timer_chi;

    return ((uint64_t)high << 32) | low;
}
```

Listing 1: Trecho de código do uptime.c

Neste contexto, o `SYSTEM_TIMER_BASE` é utilizado como endereço base para a obtenção dos *timers*. Apesar de que o projeto está desenvolvido apenas para as Raspberry Pi 2 e 3, incluiu-se o endereço base da Raspberry Pi 1, mantendo consistência com as distinções presentes no *driver*, o que também possibilita futuras implementações do jogo em outros modelos de Raspberry Pi.

A tabela 1, obtida na referência dos periféricos do BCM2837[1], indica os dois registradores utilizados na função *get_ current_ time*, `CLO` e `CHI`, representativos, respectivamente, dos 32 bits menos significativos e mais significativos do contador do *System Timer*.

Mapeamento de Endereços do *System Timer*			
Offset do endereço	**Nome do registrador**	**Descrição**	**Tamanho**
0x0	CS	System Timer Control Status	32
0x4	CLO	System Timer Counter Lower 32 bits	32
0x8	CHI	System Timer Counter Higher 32 bits	32
0xc	C0	System Timer Compare 0	32
0x10	C1	System Timer Compare 1	32
0x14	C2	System Timer Compare 2	32
0x18	C3	System Timer Compare 3	32

Tabela 1: Registradores do *System Timer*

4.3.3 A lógica do jogo em si

Em primeiro lugar, destacam-se dois pontos intrínsecos a se programar em C:

1. Não há programação orientada a objetos: dessa forma, um sistema montado em vários arquivos torna-se menos palpável. Em particular, não se poder utilizar de recursos como a classe torna mais desafiador dividir responsabilidades e manter um código de fácil leitura e escalável. Assim, a despeito dos problemas supracitados, adotou-se uma solução praticamente monolítica, com a lógica completa inicialmente em um único arquivo `main.c`.

2. Não há implementação de estruturas básicas: diferentes de linguagens como python, java e até C++, para uma série de operações relativamente básicas não há qualquer função *built-in* implementada que se possa chamar. Desta forma, foi necessário usar-se de *structs* e métodos criados pelo grupo para se valer de alguns recursos básicos e necessários - como por exemplo a estrutura de dados fila (*queue*). Em um ponto mais relacionado ao item anterior, nota-se que, na parte final do projeto, encapsulou-se parte dessa lógica e de definições em um outro arquivo .c, com as interfaces em um arquivo .h (header), importado na `main.c`. Assim, ao menos de alguma forma, foi possível trazer algum grau de modularização. Como exemplos, citam-se os arquivos `queue.c` e `tile.c`. Detalha-se que o primeiro implementa, de fato, a estrutura de dados fila, usada para gerenciamento da captura dos caracteres digitados pelo usuário na partida. Já o segundo é responsável por uma série de definições acerca das *tiles* (que são as peças, sendo tais um agregado de blocos). Lembra-se que o bloco é a estrutura unitária base do gráfico e das movimentações, e que tal, ainda, é formado por um conjunto de *pixels*. Além das definições, também utiliza métodos para desenhar as *tiles* no *grid* e atualizá-las de posição. Tal é feito via métodos definidos em um terceiro arquivo .c (com seu respectivo .h) à parte: o `graphics.c`. Evidenciando-se a questão arquitetural, vale notar que com essa divisão nenhuma referência direta a métodos de *graphics.c* restou no arquivo `main`, criando-se um isolamento.

Após a consideração dos pontos supracitados, resta a análise do arquivo `main` em si, que engloba toda a lógica do jogo. Na parte inicial, são feitos os diversos *imports* necessários, incluindo arquivos .h criados pelo grupo e outros de fato apenas consumidos - como o *driver* USPi, por exemplo. Em seguida, criam-se algumas das variáveis associadas a um comportamento dinâmico do jogo (ou que se alteram durante a partida), como o número de linhas completas, o nível atual, o *score* máximo, entre outros. Em sequência, podem-se notar diversos métodos auxiliares, e de menor tamanho. Como exemplo, tem-se métodos relativos ao movimento vertical, horizontal e rotacional, e às colisões de tais tipos, além da atualização de informações dinâmicas, como o *score* e a *next tile* - o formato da próxima *tile* a ser colocada no *grid*.

Em seguida, tem-se a primeira função maior. Tal é denominada `save_to_grid` e tem como responsabilidade adicionar a peça ao *grid* assim que houver colisão vertical. No entanto, surge uma dificuldade adicional: é necessário mapear quais linhas (e quantas) estão completas, porque tal implica que todas as *tiles* acima de tal linha devem ser movidas para baixo. Isto é, se em uma rodada foram completas X linhas, das quais Y estão abaixo de uma linha não completa, esta linha deve ser movida Y unidades para baixo - e as linhas completas devem ser apagadas. Vale notar, conforme explicado no início dessa subseção, que já existe certa abstração na main, posto que parte dos métodos é chamada de outros *headers*, como o `draw_grid_block`, definido em `game_screen.c`.

A função seguinte é a responsável por mover o bloco com base no comando. Seu funcionamento consiste em, com base no caractere, checar a colisão **para a posição que seria ocupada** pela *tile* e então movê-la - novamente com base no caractere. A função seguinte é a `run`, que implementa a lógica de mais alto nível do jogo. Em primeiro lugar, verifica-se se o limite superior vertical foi atingido, situação em que o jogador perde e segue para o reinício. Em seguida, tem-se um *loop* que move a peça para baixo bloco a bloco e checar pela colisão vertical, quando uma nova peça é disparada. Além disso, são checados os movimentos do usuário e aplicados - quando não houver colisão. Por fim, as demais funções fazem o *setup* da parte gráfico e do teclado para o jogo, além de lidarem com o reinício.

5 Protótipo

Nesta seção, detalha-se o protótipo desenvolvido, mostrando-se o resultado final atingido pelo grupo e suas características.

5.1 Parte física

Tal qual projetado de partida, o projeto exige 3 componentes físicos principais: um teclado, um monitor e a Raspberry Pi (com suporte às versões 2 e 3). Quanto ao teclado, vale destacar que alguns modelos podem possuir problemas de compatibilidade. Assim como citado na seção 7.1.2, houve um caso em que um certo modelo de teclado foi um impeditivo para o correto funcionamento com o *driver* em questão. Já quanto ao monitor, não há nenhuma restrição em específico, basicamente qualquer dispositivo que possa se conectar via interface HDMI pode ser utilizado para se visualizar o jogo - como exemplos, no próprio vídeo do projeto usaram-se diferentes monitores e, inclusive, o projetor da sala do laboratório. A figura 4 detalha estas conexões, sendo um registro da configuração no momento em que se gravou o resultado final. Vale notar que o cabo menor, no canto superior direito, corresponde apenas à alimentação de energia da placa.

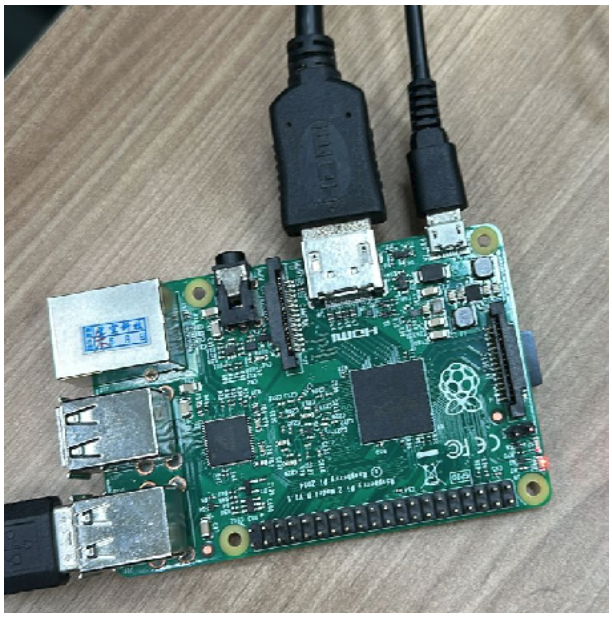

Figura 4: Conexões na Raspberry Pi

5.2 Features implementados e interface gráfica

Primeiro, quanto às *features* desenvolvidas para a versão final, destacam-se:

- Movimentação horizontal com tecla 'a' para a esquerda e tecla 'd' para a direita) e rotação ($+-90$ graus, ou seja, no máximo 4 rotações diferentes por peça) com a tecla 'w'.
- Possibilidade de recomeçar o jogo, usando-se a tecla 'r'.
- Possibilidade de se completar múltiplas linhas por vez (até 4, que é a maior dimensão possível de uma peça no eixo vertical, se devidamente rotacionada).
- Atualização do número de linhas completas *on time* e também do nível, aumentando-se a velocidade conforme o jogador progride.
- Blocos gerados aleatoriamente (e com cores também aleatoriamente geradas).
- *Frame* de estatísticas, contendo quantas peças de cada tipo foram usadas em uma certa rodada.
- *Score* atualizado *on time* para o jogador, bem como *top score* da mesma forma - porém este atualizado a cada vez que o jogo termina.

Além da questão funcional e dos requisitos da lógica do jogo adicionais (não presentes no MVP), desenvolveu-se também a interface gráfica do jogo de modo a aproximar-se o máximo possível do tetris original. Note-se abaixo, por meio das figuras 5 e 6, que implementou-se o *background* da forma mais fiel possível ao original. Além disso, cada *frame* (caixa de texto/figura) foi projetado para ter as mesmas dimensões do original e o mesmo texto. Tal procedimento foi bastante trabalhoso, posto que envolveu entender como a interface HDMI lidava, no mais baixo nível, com a representação de caracteres - isto é, os métodos para se desenhar cada letra, pixel a pixel. Assim, pôde-se utilizá-los da forma adequada para gerar os textos requeridos.

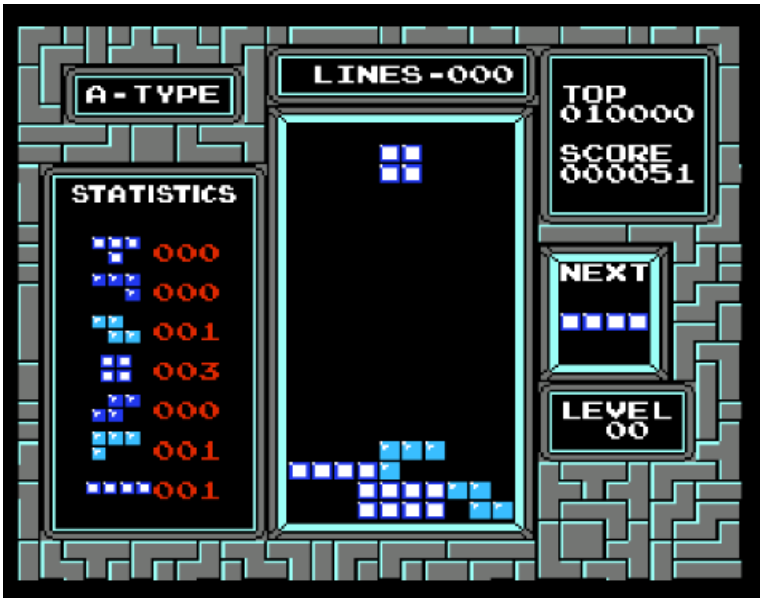

Figura 5: Interface do tetris original, usada como base.

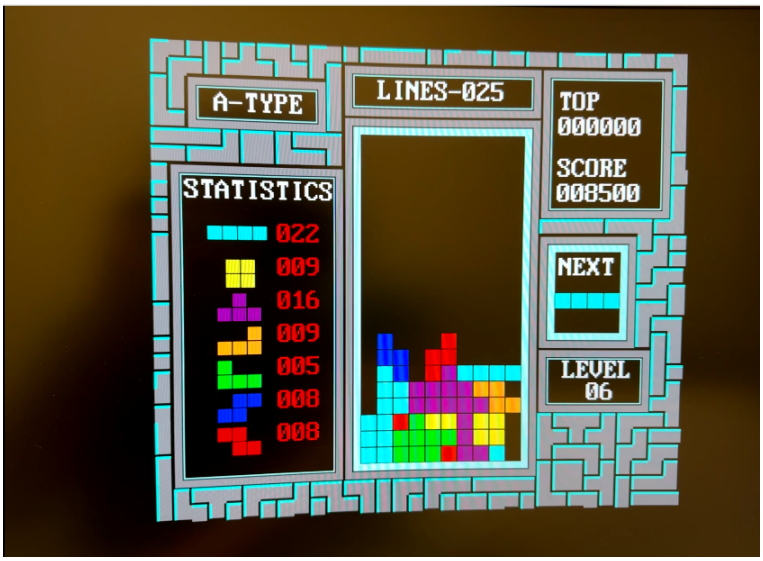

Figura 6: Interface gráfica finalizada

Por fim, destaca-se a tela de reinício na figura 7, em que é indicado que o jogador pode apertar r para recomeçar. Além disso, há um QR code na tela com um link para o repositório do grupo.

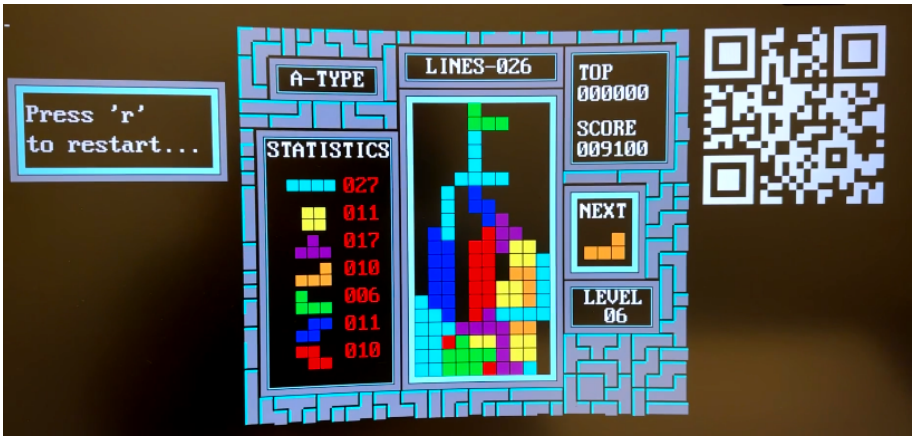

Figura 7: Tela após perder o jogo, com possibilidade de reinício e QR code.

6 Considerações Finais

Finalmente, considerando o desenvolvimento completo do projeto e a integração das principais partes - lógica do jogo e conexões com dispositivos externos -, pode-se validar a proposta inicial, apresentada na seção 1: foi possível a construção de um projeto com um grau de complexidade alto, envolvendo o uso da linguagem C e da Raspberry Pi. Pode-se atribuir grande relevância aos aprendizados na integração do *driver* USB - apropriado para o desenvolvimento *bare metal* na Raspberry Pi - e na integração de outras bibliotecas estáticas, para possibilitar a geração aleatória de sequência de blocos para o jogo, além do desenvolvimento do jogo em si. Tal parte está intimamente relacionada ao conteúdo tratado na matéria, e portanto justifica a escolha do projeto.

A respeito do uso do *driver* do projeto USPi, destaca-se a importância em se utilizar projetos que apresentam compatibilidade para diferentes modelos de Raspberry Pi, possibilitando maior flexibilidade ao projeto, além de incluir nitidamente as diferenças no desenvolvimento para cada um destes modelos. Ademais, outro ponto relevante é a integração de um projeto de tal complexidade, possibilitando aprendizados em termos de boas práticas de projeto, automações na geração dos arquivos a serem escritos no cartão SD para a Raspberry Pi, e definições para possibilitar o uso desta placa. Outra atribuição essencial a este tipo de projeto, em termos de flexibilidade, é o fato de ser possível utilizar este *driver* para a integração com outros dispositivos: *mouses*, *GamePads/Joysticks*, dispositivos de armazenamento em massa USB, controlador Ethernet, MIDI, entre outros. Foi possível verificar o funcionamento adequado deste *driver* para a conexão de um *mouse* na Raspberry Pi, conforme indicado na seção 7.1.2, o que torna possível o seu uso em outros projetos que envolvam a necessidade deste tipo de integração.

O uso de bibliotecas estáticas, por sua vez, possibilitou a compreensão do funcionamento do processo de adição de bibliotecas nos projetos: como funciona o processo das inclusões, com os arquivos-cabeçalhos; identificação de contextos em que é necessário efetuar implementações próprias de tipos e funções, como foi o caso exposto na seção 4.3.2; e como tais inclusões das bibliotecas no projeto devem ser passadas para o *linker*, garantindo que o projeto gerado tenha o comportamento esperado com as inclusões necessárias.

Já com relação à implementação do jogo em si, há uma intersecção com a questão das bibliotecas estáticas usadas e o *driver* USPi, na medida em que para programar algumas partes do código (como por exemplo a questão das múltiplas cores ou das dimensões do *grid*) é necessário um entendimento de como tais funções estão implementadas no nível hierarquicamente abaixo - isto é, no arquivo .a. Ademais, qualquer parte relacionada ao teclado ou ao desenho na tela requereu a chamada de funções do *driver*, então mesmo tratando-as como uma caixa preta ainda é necessário compreender a parte em que há intersecção (ou seja, ao menos as assinaturas das funções), fornecendo uma visão geral sobre a implementação de parte das funcionalidades do *driver*. Por fim, destacam-se os próprios desafios enfrentados na implementação do jogo. Nota-se que pensar em formas mais espertas de implementar algumas partes poupou uma grande quantidade de trabalho em determinadas situações. Por exemplo, quanto à rotação, um ponto observado é que criar uma função que mapeasse a peça em sua versão rotacionada em 90 graus da posição atual seria muito complicado, mas tal seria completamente resolvido definindo-se as 4 possíveis rotações de cada peça e usando um incremento unitário módulo 4 a cada vez que houvesse rotação. Já quanto à colisão, nota-se que uma implementação ingênua poderia considerar erroneamente colisões da peça consigo mesma (por exemplo, suponha um movimento para a direita de uma peça em 'L'), caso houvesse blocos vizinhos horizontalmente na mesma peça ou em outros casos com geometria inconveniente. Tal é facilmente resolvido ao checar-se a colisão em uma matriz grid e e adicionar-se a peça a tal matriz apenas no momento em que houver colisão vertical. Assim, nota-se que a implementação do jogo pode parecer mais simples do que fato é à primeira vista, sendo que a maior dificuldade encontrou-se nos detalhes.

Por fim, vale a pena observar que foram exercidas habilidades de gerenciamento de projeto por parte do grupo. Considerando-se as limitações do projeto (principalmente quanto a tempo), tomou-se a correta decisão de se procurar por um *driver* pronto e compatível e focar os esforços em compreender como transformá-lo em algo compreensível à Raspberry Pi. Dessa maneira, pôde-se suprir os requisitos necessários ao jogo (quanto a receber *input* de algum dispositivo e mostrar o *grid* em um monitor), agregando valor ao projeto com um dispêndio de tempo reduzido - ainda que significativamente elevado. Além disso, a decisão de se usar um simulador (SDL2) ao longo do desenvolvimento do projeto foi útil de um ponto de vista de vazão de trabalho, posto que permitiu ao grupo se dividir em duas frentes de desenvolvimento pouco acopladas.

7 Apêndices

7.1 USPi

7.1.1 Breve introdução ao repositório do USPi

O repositório do USPi[7] apresenta em seu `README.md` as principais instruções a respeito de seu uso.

Especificamente, em termos dos diretórios presentes na raiz do projeto, é apresentado o propósito de cada um, com destaque para os seguintes diretórios:

- `include`: arquivos-cabeçalhos do USPi;
- `lib`: código-fonte do USPi;
- `sample`: alguns códigos-exemplo;
- `env`: ambiente utilizado pelos códigos-exemplos (não necessário diretamente pela biblioteca do USPi).

Vale destacar brevemente a importância dos dois últimos diretórios.

Inicialmente, considerando o diretório `sample`, identifica-se entre seus diretórios exemplos de caso de uso do *driver* para integração de diferentes dispositivos, entre eles, o teclado. Nesse sentido, o uso deste projeto possibilitou a verificação de seu funcionamento e compatibilidade com a Raspberry Pi sem a necessidade de ajustes iniciais no projeto, apenas indicando o modelo adequado da placa.

Ademais, a respeito do diretório `env`, identifica-se as definições adicionais de ambiente, conforme indicado acima, que possibilitam o funcionamento adequado das implementações de cada exemplo no diretório `sample`. Como o código do teclado é utilizado para identificar a tecla pressionada, para posterior impressão na tela, adotou-se no desenvolvimento do projeto para a disciplina este mesmo conjunto de arquivos definidos no diretório `env`.

Outro ponto relevante a respeito deste diretório é o fato de que ele já inclui implementações suficientes para possibilitar a integração da Raspberry Pi com um monitor. A função `ScreenDeviceSetPixel` implementado no arquivo `screen.c`, presente em `env/lib`, lida com a parte de determinação dos pixels, a fim de possibilitar a inclusão de desenhos na tela. Ele necessita de um dispositivo de tela, para o qual é vinculado o `framebuffer`, assim como pode ser visto no arquivo `screen.h` que define os tipos utilizados nesta função.

Para a geração dos arquivos de imagem para escrita no cartão SD, basta utilizar o comando `./makeall` na raiz do projeto, passando os parâmetros adicionais quando necessário (para indicar o modelo da Raspberry Pi, arquitetura quando coexistem as arquiteturas de 32 e 64 bits, entre outros), e a imagem correspondente será gerada no diretório de cada um dos projetos dos códigos-exemplo.

Finalmente, considerando o exemplo da conexão com o teclado presente no repositório, indicado abaixo, pode-se identificar o uso dos *logs* para verificar a conexão. Além disso, a função `KeyPressedHandler`, utilizada nesse exemplo para a impressão da tecla pressionada (com o uso da função `ScreenDeviceWrte`), serve como referência para as atualizações necessárias para o projeto, sendo apenas necessário indicar no corpo desta função o que deve ser feito após o pressionamento de uma tecla.

```c
//
// main.c
//
#include <uspienv.h>
#include <uspi.h>
#include <uspios.h>
#include <uspienv/util.h>

static const char FromSample[] = "sample";

static void KeyPressedHandler (const char *pString)
{
    ScreenDeviceWrite (USPiEnvGetScreen (), pString, strlen (pString));
}

int main (void)
{
    if (!USPiEnvInitialize ())
    {
        return EXIT_HALT;
    }

    if (!USPiInitialize ())
    {
        LogWrite (FromSample, LOG_ERROR, "Cannot initialize USPi");

        USPiEnvClose ();

        return EXIT_HALT;
    }

    if (!USPiKeyboardAvailable ())
    {
        LogWrite (FromSample, LOG_ERROR, "Keyboard not found");

        USPiEnvClose ();

        return EXIT_HALT;
    }

    USPiKeyboardRegisterKeyPressedHandler (KeyPressedHandler);

    LogWrite (FromSample, LOG_NOTICE, "Just type something!");

    // just wait and turn the rotor
    for (unsigned nCount = 0; 1; nCount++)
    {
        USPiKeyboardUpdateLEDs ();

        ScreenDeviceRotor (USPiEnvGetScreen (), 0, nCount);
    }

    return EXIT_HALT;
}
```

Listing 2: Arquivo keyboard.c, presente nos exemplos de código de uso do driver

7.1.2 Utilização dos *logs* de inicialização do USPi

Vale ressaltar que a escolha deste *driver*, por se tratar de um projeto contendo *logs* de inicialização, foi adequada para minimizar a ausência de formas de depuração para a integração do projeto com o teclado (e também com o monitor), e possibilitou a inclusão de mais *logs* durante o processo de desenvolvimento, além de permitir a identificação da incompatibilidade com o teclado em si. A primeira tentativa de conexão efetuada, com um teclado do modelo `Redragon Sani K581`, não funcionou. A mensagem de depuração indicou que não foi encontrado nenhum teclado, assim como apresentado na imagem 3. Por se tratar de um *driver* USB, o caminho adotado foi efetuar o teste com o *mouse*. Como o USPi também continha um código de exemplo para o uso de *mouses*, este teste foi facilmente efetuado, assim como apresentado na imagem 8.

Assim, foi possível concluir que o problema estava na conexão com aquele teclado em específico, haja

vista que as outras duas possíveis alternativas seriam problema de compatibilidade na Raspberry Pi utilizada (nesta alternativa, a integração do *mouse* possivelmente não funcionaria) e má implementação das funções necessárias para a integração do teclado e/ou visualização das teclas pressionadas (improvável, considerando a relevância do projeto na comunidade de Raspberry Pi). Após efetuar a substituição do teclado, foi possível visualizar a conexão esperada entre o segundo modelo e a Raspberry Pi.

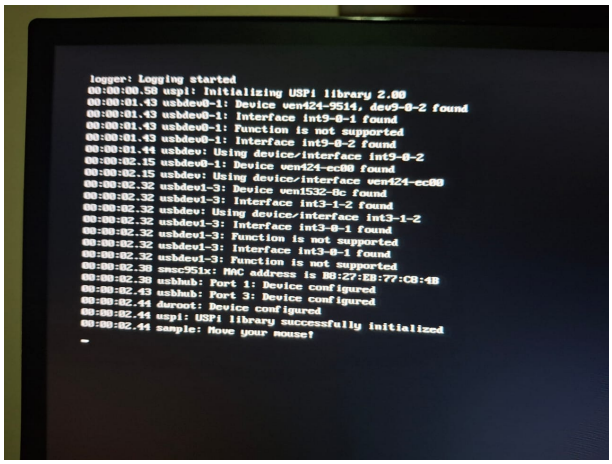

Figura 8: Demonstração do *log* para o caso de conexão de um *mouse*.

Referências

[1] *Broadcom Corporation. BCM2837 ARM Peripherals.*

[2] *Newlib.* https://sourceware.org/newlib/.

[3] *Functions srand and rand in C.* https://www.ibm.com/docs/el/i/7.4?topic=functions-srand-set-seed-rand-function.

[4] *Processors - Raspberry Pi Documentation.* https://www.raspberrypi.com/documentation/computers/processors.html.

[5] *Low Level Tetris.* https://github.com/TiagoMLucio/low-level-tetris.

[6] *Time function in C.* https://www.geeksforgeeks.org/time-function-in-c/.

[7] Rene Stange. *USPi - A bare metal USB driver.* https://github.com/rsta2/uspi.

[8] *Broadcom Corporation. BCM2835 ARM Peripherals.*