Universidade do Porto

Faculdade de Engenharia

FEUP

Electronic dice

“Dicequake”

Diogo Pereira Poças Oliveira

João Vitor Ferreira Andrade

Relatório do Trabalho Prático realizado no âmbito da Unidade Curricular
“Arquitetura de Computação Embarcada”, do
Mestrado em Engenharia Eletrotécnica e Computadores

16-11-2025

Declaramos que o presente trabalho/relatório é da nossa autoria e não foi utilizado previamente noutro curso ou unidade curricular, desta ou de outra instituição. As referências a outros autores (afirmações, ideias, pensamentos) respeitam escrupulosamente as regras da atribuição, e encontram-se devidamente indicadas no texto e nas referências bibliográficas, de acordo com as normas de referenciação. Temos consciência de que a prática de plágio e autoplágio constitui um ilícito académico.

In "Código Ético de Conduta Académica", art.14, Universidade do Porto.

1 Resumo

Neste trabalho iremos descrever o processo da implementação de um dado eletrónico. Iremos abordar aspetos como comunicação, lógica de máquina de estados e escrita num display.

Palavras-chave: Máquina de estados; Raspberry Pi Pico; I2C; MPU6500

Índice

1 Resumo 1
Índice 1
2 Introdução 1
2.1 Hardware/Componentes e esquema do circuito 1
2.2 I2C 2
3 Máquina de Estados 3
3.1 Estado Begin, Pressing e Menu 6
3.2 Estados Calibração 8
3.3 Estados de Jogo e definição de parâmetros 10
4 Conclusão e aspetos a melhorar 13
Referências 13

2 Introdução

O objetivo deste trabalho é conceber e implementar um sistema embebido capaz de simular o lançamento de dados físicos, utilizando uma plataforma de baixo custo baseada na Raspberry Pi Pico. A proposta enquadra-se na unidade curricular de Arquitetura de Computação Embarcada e pretende consolidar conceitos de comunicação digital, leitura de sensores, desenho de interfaces simples com o utilizador e modelação de comportamento através de máquinas de estados finitos.

Nas subsecções deste capítulo vamos abordar o hardware utilizado neste projeto, o esquema do circuito montado, o protocolo de comunicação utilizado (I2C)

2.1 Hardware/Componentes e esquema do circuito

O hardware utilizado é composto pelos seguintes elementos principais:

- **Raspberry Pi Pico** – placa com microcontrolador RP2040;
- **Sensor inercial MPU6500** – módulo que integra um acelerómetro e um giroscópio, utilizado para detetar a agitação do sistema e para medir a aceleração e velocidade angular em 3 eixos;
- **Mostrador OLED SSD1306 -** ecrã monocromático que constitui a interface visual do sistema;
- **Módulo de botões** – conjunto de oito botões com resistências de pull-up internas.

O esquema do circuito é o seguinte:

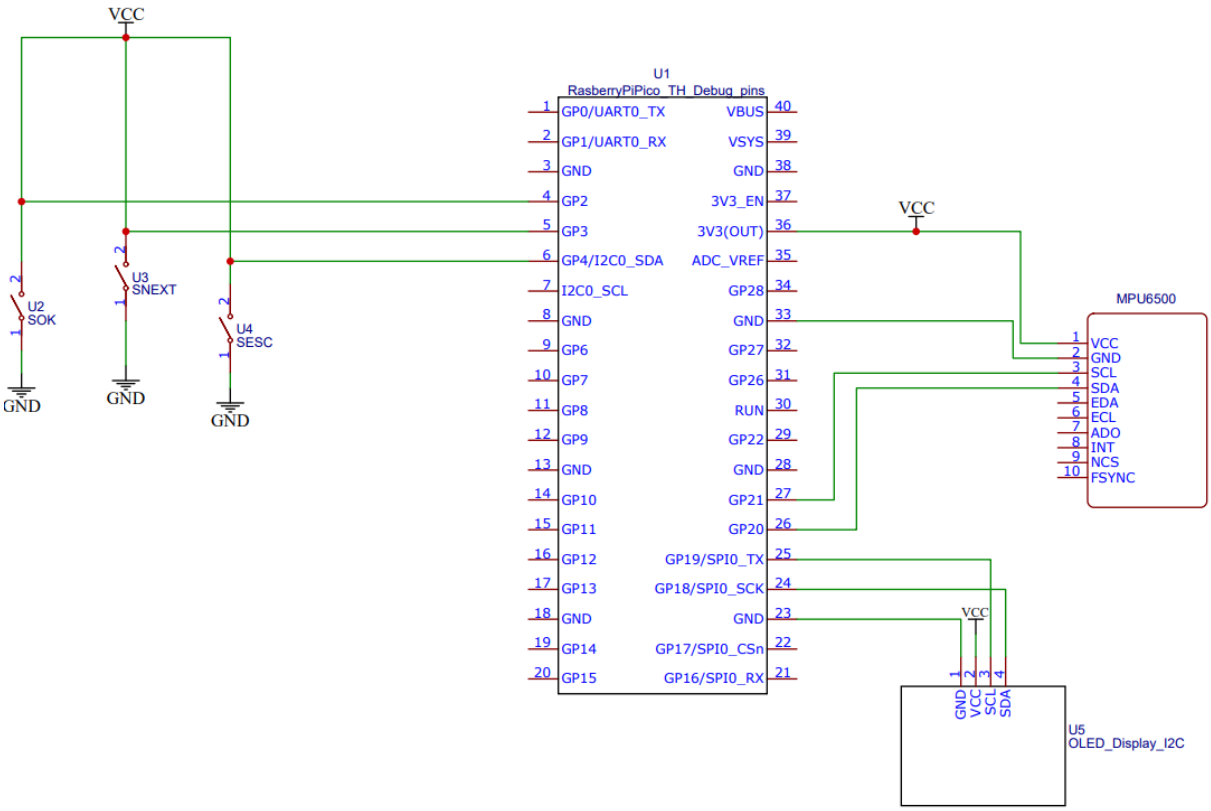


Figura 1 - Esquemática do circuito (CAD)

Observando o esquema podemos notar as ligações características no protocolo de comunicação I2C, e que a lógica dos pinos 2, 3 e 4, dos respetivos botões SOK, SNEXT e SESC, devem ser lidos com lógica inversa, pois estão diretamente conectados à alimentação, e no momento que eles estão pressionados, a tensão dos pinos é baixa.

2.2 I2C

A comunicação entre a Raspberry Pi Pico, a MPU6500 e o mostrador SSD1306 é feita através do protocolo I2C. Como se observa no esquemático (Figura X), ambos os módulos dispõem de duas ligações dedicadas, SDA e SCL, que são conectadas aos respetivos pinos I2C da Pico. Estas linhas funcionam como terminais de comunicação série bidirecional.

- SCL (Serial Clock) transporta o sinal de relógio gerado pelo master;
- SDA (Serial Data) transporta os dados entre o master e os dispositivos slave.

O protocolo I2C utiliza uma topologia de barramento partilhado, onde múltiplos dispositivos podem coexistir nas mesmas duas linhas, sendo identificados por endereços próprios. A Raspberry Pi Pico atua como mestre, iniciando todas as comunicações e controlando o sinal de relógio, enquanto a IMU e o OLED operam como escravos, respondendo apenas quando o seu respetivo endereço é chamado. Esta relação pode ser observada na figura 2.

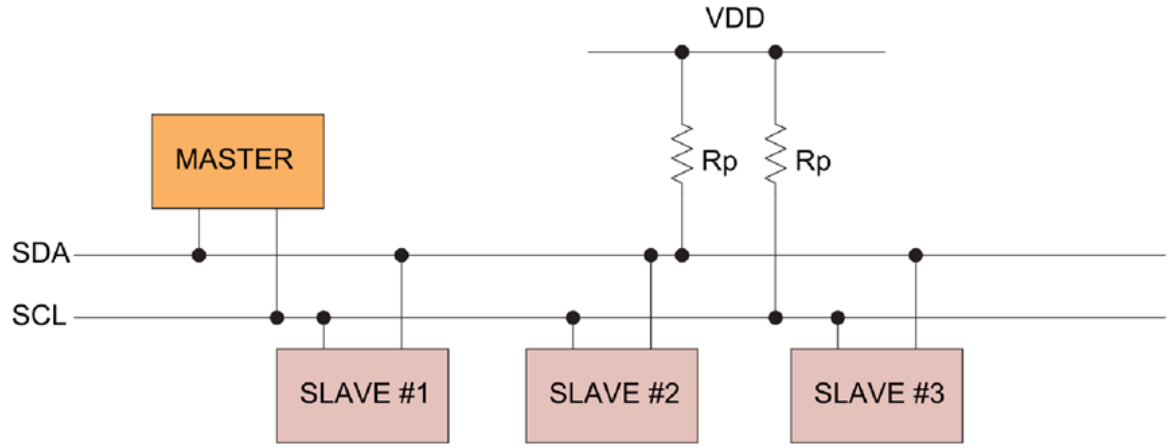


Figura 2 – Ligações protocolo I2C

3 Máquina de Estados

Usamos máquina de estados para controlar o comportamento do display e a lógica do código. Nas subsecções deste capítulo iremos entrar em detalhe sobre a implementação da máquina de estados no código, das funções e transições de cada estado de maneira a dar uma visão geral do projeto.

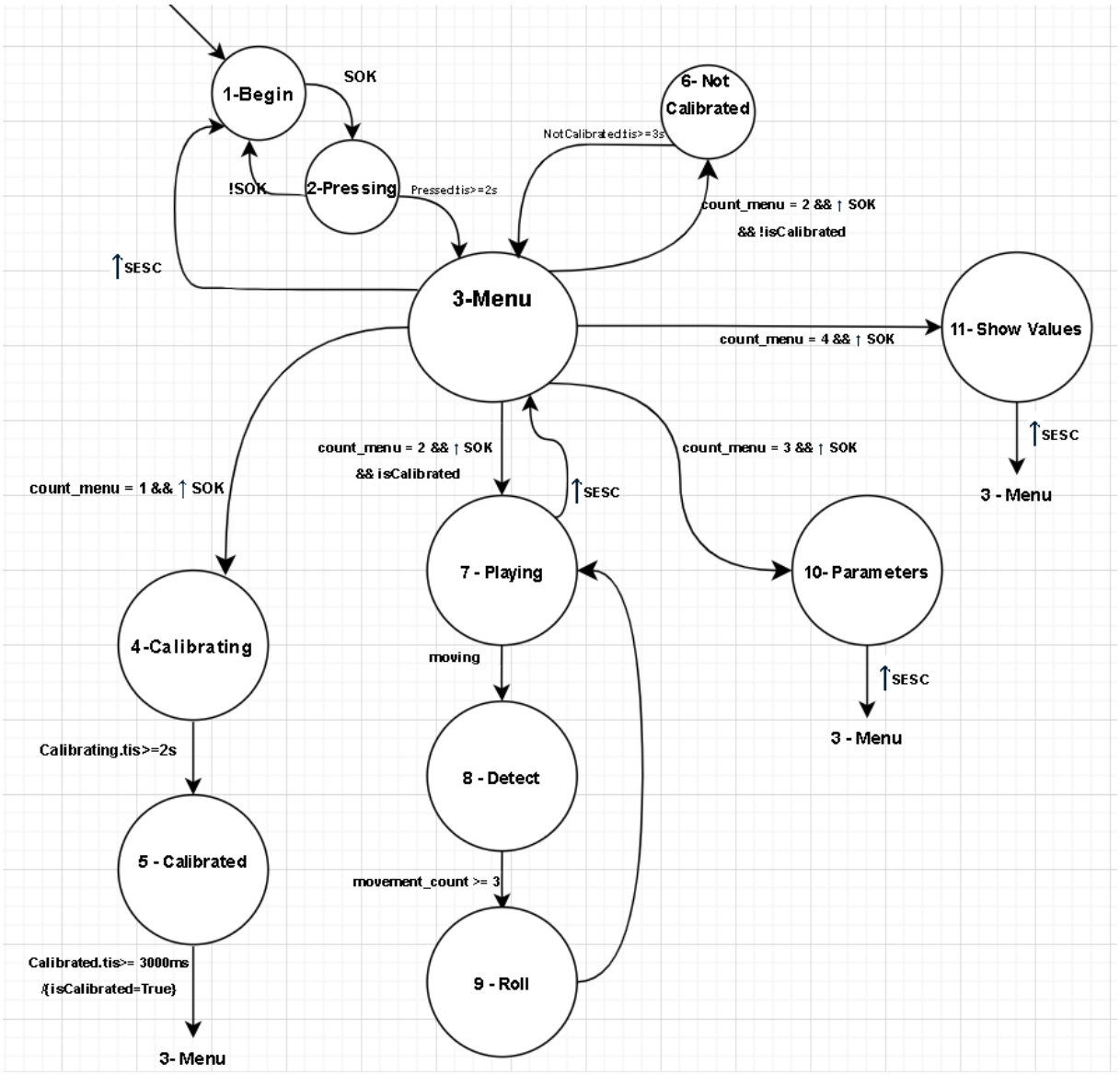


Figura 3 - Máquina de estados

A implementação dos diferentes estados e das transições foi possível através da criação de alguns elementos structs e funções.

Cada estado foi "conseguido" através da criação do struct fsm_t:

```
typedef struct {
  int state, new_state;
  unsigned long tes, tis;
} fsm_t;
```

As variáveis tes e tis, servem para cálculo do tempo do estado, enquanto que as variáveis state e new_state servem para transição entre estados e para possibilitar as saídas respetivas de cada estado.

A função set_state, quando é chamada, trata de reiniciar o tempo de um estado, bem como implementar a transição entre estados.

```
void set_state(fsm_t &fsm, int new_state) {
  if (fsm.state != new_state) {
    fsm.state = new_state;
    fsm.tis = 0;
    fsm.tes = millis();
  }
}
```

No caso dos butões, criamos o struct button_t :

```
typedef struct {
  bool value;
  bool prev_value;
  bool re;
} button_t;
```

Criamos três instâncias para cada botão (SOK,SNEXT e SESC). As variáveis value e prev_value servem para guardar o estado atual e anterior de cada butão. A variável "re" é um booleano que serve para implementar um flanco ascendente. Deve ser verdadeiro, se e apenas se, "prev_value" for 0 e "value" for 1. Este comportamento pode ser observado na função update_buttons que é chamada recorrentemente.

```
static inline void update_buttons(void) {

  SOK.prev_value = SOK.value;
  SOK.value = !digitalRead(SOK_BUT);
  SOK.re = (!SOK.prev_value && SOK.value);

  SESC.prev_value = SESC.value;
  SESC.value = !digitalRead(SESC_BUT);
  SESC.re = (!SESC.prev_value && SESC.value);

  SNEXT.prev_value = SNEXT.value;
  SNEXT.value = !digitalRead(SNEXT_BUT);
  SNEXT.re = (!SNEXT.prev_value && SNEXT.value);
}
```

A função print_display() também é chamada recorrentemente para escrevermos no display conforme o estado atual.

3.1 Estado Begin, Pressing e Menu

Os três primeiros estados correspondem à fase inicial de interação com o utilizador. A transição entre estes estados é feita com base na leitura do botão SOK. Uma das especificações do guião é que devemos aceder ao menu, apertando no botão SOK durante 2 segundos. Para isto, criamos uma variável "Pressed", que está ativa quando o botão SOK está permido. Quando esta etapa estiver ativa durante 2 segundos, então atingimos o estado do Menu.

No estado "Begin", o display apresenta uma mensagem de boas-vindas e indica ao utilizador como alcançar o Menu como pode ser observado na figura x

Figura 4 – Display do estado "Begin"

A função print_display() é responsável pela escrita no display.

```
//BEGIN DISPLAY
if (fsm.state == fsm_begin) {
  display.clearDisplay();
  display.setTextColor(SSD1306_WHITE);
  display.setTextSize(2);
  display.setCursor(8, 10);
  display.println("DiceQuake!");
  display.setTextSize(1);
  display.setCursor(2, 32);
  display.println("Welcome to DiceQuake!");
  display.setCursor(0, 44);
  display.println("Press SOK for 2s to ");
  display.setCursor(0,52);
  display.println("go to menu!");
  display.display();
}
```

Quando o utilizador alcança o menu, o display exibe os estados que se podem atingir através do botão SOK. Para conseguirmos atingir vários estados através de um botão, utilizamos um contador, "count_menu" que é incrementado cada vez que o botão SNEXT é premido.

```
// Avançar opção no menu (NEXT rising edge)
if (fsm1.state == fsm_menu && SNEXT.re) {
  count_menu = (count_menu % 4) + 1;
}
```

As transições do menu para os diferentes estados segue a lógica do excerto do código abaixo (existem mais transições, que não são mencionadas por motivos de repetição).

```
// Menu -> Begin (ESC rising edge)
if (fsm1.state == fsm_menu && SESC.re) {
  set_state(fsm1, fsm_begin);
}
// Menu -> Parameters (count_menu==3, OK rising edge)
if (fsm1.state == fsm_menu && count_menu == 3 && SOK.re) {
  set_state(fsm1, fsm_parameters);
  count_menu = 1;
}

// Menu -> Playing (count_menu==2, OK rising edge, is_Calibrated)
if (fsm1.state == fsm_menu && count_menu == 2 && SOK.re && is_Calibrated) {
  set_state(fsm1, fsm_play);
  count_menu = 1;
}
```

Para guiar o utilizador do estado, conforme o número de count_menu, o display deverá indicar qual ó estado que se alcança ao premir o butão SOK. Para isso, na função print_display, adicionamos uma seta que muda de posição conforme o valor de count_menu, como se pode ver nas figuras x –

Figuras 5-8 – Fotos do display ilustrando o comportamento do menu face à leitura do botão SNEXT.

Através de uma condição de tempo de estado, essa seta está ligada 750 milissegundos e desligada durante 750 milissegundos. Os seguinte excertos de código ilustram a implementação deste aspeto.

```
//MENU DISPLAY
if (fsm.state == fsm_menu && fsm.tis % 1500 <= 750){
  display.clearDisplay();
  display.setCursor(52,0);
  display.printf("Menu");
  display.setCursor(0,16);
  display.printf("Calibrate ");
  display.setCursor(0,24);
  display.printf("Play ");
  display.setCursor(0,32);
  display.printf("Set parameters");
  display.setCursor(0,40);
  display.printf("Calibrated values");
  display.display();
}
```

```
if (fsm.state == fsm_menu && countmenu == 1 && fsm.tis % 1500 >= 750){
display.clearDisplay();
display.setCursor(52,0);
display.printf("Menu");
display.setCursor(0,16);
display.printf("Calibrate <-");
display.setCursor(0,24);
display.printf("Play ");
display.setCursor(0,32);
display.printf("Set parameters");
display.setCursor(0,40);
display.printf("Calibrated values");
display.display();
}
```

3.2 Estados Calibração

Tendo em conta que os valores transmitidos pela MPU6500 não apresentam grande fiabilidade, é recomendada a calibração do acelerómetro. Tendo isso em conta, o estado Calibrating serve para guardar uma amostra de valores, durante 2 segundos, de cada leitura da MPU (ax, ay, az – acelaração e wx, wy e wz – velocidade).

Durante estes 2 segundos, o display deve apresentar uma animação que durante o tempo da amostragem dos valores como se pode ver nas figuras x - x+2.

Figuras 9-11 – Fotos do display durante o estado “Calibrating”

Através destas amostras, fazemos uma média dos valores, para obter um valor de calibração para cada elemento. Este valor será um valor de offset, agora através podemos obter uma leitura correta, para isso basta subtrair o valor calibrado à leitura do valor atual da MPU.

Após os 2 segundos, vamos para o estado “Calibrated” durante 3 segundos. Este estado serve para colocar no display os valores calibrados, de maneira a que o utilizador consiga ter noção se a calibração foi eficaz ou não. Os valores devem oscilar perto de 0 exceto para az, que deve oscilar perto de 1, como se pode ver na figura x. No fim desta rotina, a variável booleana “isCalibrated” é atríbuida com o valor TRUE. O papel desta variável passa por impossibilitar o utilizador de ir para o estado de jogo (“Playing”), enquanto os valores não estiverem calibrados.

Figura 12 – Display do estado “Calibrated”

No caso do utilizador tentar começar a jogar sem recorrer à calibração, a máquina de estados assume o estado “Not Calibrated” durante 3 segundos, no qual o display indica para o utilizador que ainda não efetuou a calibração dos valores, como pode ser visto na figura x.

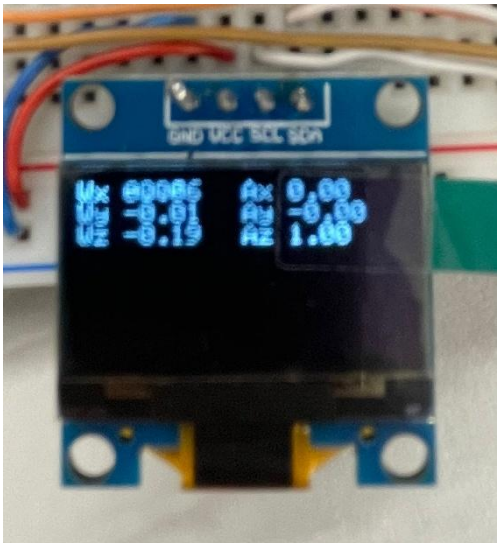
Após o intervalo de tempo definido, o utilizador volta automaticamente ao menu.

Figura 13 – Display do estado “Not Calibrated”

Para além destes estados, no menu, o utilizador pode selecionar a opção “Calibrated Values”, transitando para o estado “Show values”

Neste estado, à semelhança do estado “Calibrated”, o utilizador consegue visualizar os valores da calibração, no entanto, o regresso para o menu não é automático (ao contrário do momento da calibração), e deve regressar ao menu apenas quando clicar no botão SESC.

A figura x demonstra o display do estado “Show values” enquanto que o excerto do código demonstra a função print_display.



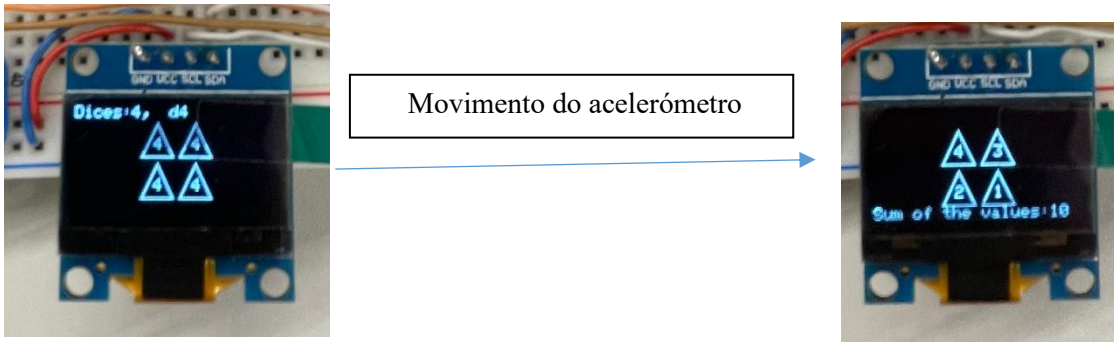

```
// CALIBRATED state VALUES DISPLAY
if (fsm.state == fsm_show_values){
  display.clearDisplay();
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);
  display.setCursor(0, 0);
  display.printf("Wx %.2f\n", imu.w.x - wx_calibrated);
  display.printf("Wy %.2f\n", imu.w.y - wy_calibrated);
  display.printf("Wz %.2f\n", imu.w.z - wz_calibrated);
  display.setCursor(64, 0);
  display.printf("Ax %.2f", imu.a.x - ax_calibrated);
  display.setCursor(64, 8);
  display.printf("Ay %.2f", imu.a.y - ay_calibrated);
  display.setCursor(64, 16);
  display.printf("Az %.2f", imu.a.z - az_calibrated + 1);
  display.display();
}
```

Figura 14 – Display do estado “Show values”

3.3 Estados de Jogo e definição de parâmetros

No caso do utilizador optar pela opção “Play” no menu e assumindo que tenha feito a calibração, a máquina de estados assume o estado “Play”. Conforme um número de dados definido e o tipo de dado, no momento que o utilizador abanar o acelerómetro, o display irá respeitar os parâmetros e exibir um número aleatório (dentro dos valores permitidos do tipo do dado) para cada dado. Cada valor é somado, e essa soma deve ser apresentada também no display durante 5 segundos.

Nas figuras 15 e 16, conseguimos observar este comportamento.



Figuras 15 e 16 – Display durante jogadas para o caso de 4 dados d4

Para detetarmos o momento que o utilizador abana os dados na horizontal, criamos a variável booleana “moving”. Esta variável retorna TRUE, quando o valor calibrado da acelaração dos eixos x ou y

passa de 1. Para além disso, devemos detetar movimento durante 3 ciclos do microcontrolador (equivalente a 60ms). O excerto de código demonstra uma parte da lógica aqui descrita.

```
if(fsm1.state==fsm_play)
{
  drawex(number_of_dices,dice_numbers);
  bool moving = (fabs(imu.a.x - ax_calibrated) > 1 || fabs(imu.a.y - ay_calibrated) > 1);

  switch (fsmd.state) {

    case fsm_start:
      if (moving) {
        Serial.println("Movement detected -> fsm_detect");
        set_state(fsmd, fsm_detect);
      }
      break;

    case fsm_detect:
      if (moving) {
        fsmd.movement_count++;
        Serial.printf("Movement cycle %d\n", fsmd.movement_count);
        if (fsmd.movement_count >= 3) {
          Serial.println("3 cycles of movement -> fsm_roll");
          set_state(fsmd, fsm_roll);
        }
      } else {
        Serial.println("Movement stopped -> fsm_start");
        set_state(fsmd, fsm_start);
      }
      break;
```

Se detetarmos movimento durante 3 ciclos, então a nossa máquina de estados transita para o estado "Roll", no qual, a partir do tipo de dado, é gerado um número aleatório pela função "draw".

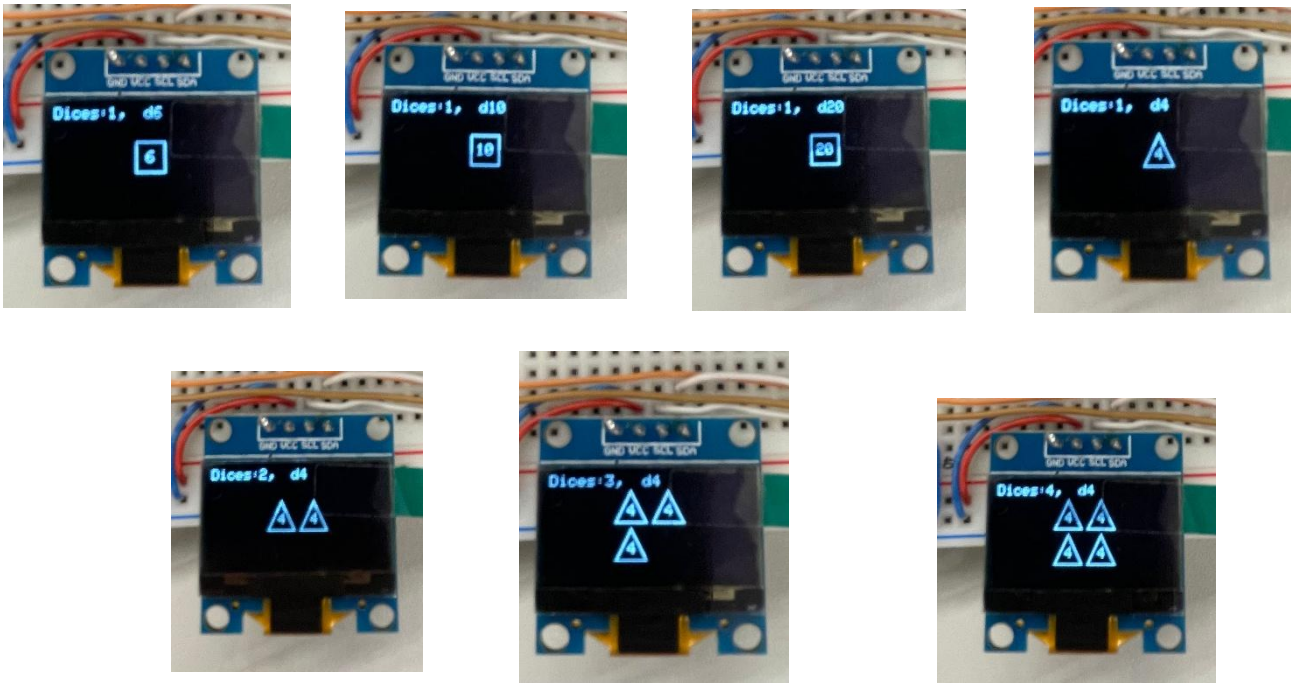
```
case fsm_roll:
  {
    uint8_t diceCount = random(1, 5);
    Serial.printf("Rolling %d dice...\n", diceCount);
    draw(number_of_dices,dice_numbers);

    Serial.println("Returning to start state");
    set_state(fsmd, fsm_start);
  }
  break;
```

O utilizador tem a possiblidade de definir a quantidade de dados que rola, bem como o tipo de dados (dado com 4,6,10 ou 20 faces).

Para isto o utilizador deve selecionar a opção "Set parameters" no menu. Ao pressionar SOK, a máquina de estados assume o estado "Parameters". Neste estado, ao pressionar o botão SOK, o utilizador altera o número de faces do dado, enquanto que o botão SNEXT é responsável por alterar a quantidade

de dados rolados. Nas figuras 17 – 23 é possível observar o que acontece se o utilizador premir o botão SOK três vezes e o botão SNEXT duas vezes.



Figuras 17-23 – Comportamento do display no estado “Parameters” quando os botões são premidos

Para controlar o tipo de dados e a quantidade de dados rolados, semelhante ao menu, temos uma variável inteira de contagem para cada um dos parâmetros – param_dice_numbers.

```
if (fsm1.state == fsm_parameters && SOK.re) {
  param_dice_numbers = (param_dice_numbers + 1) % 4;
  switch (param_dice_numbers) {
    case 0: dice_numbers = 4;  break;
    case 1: dice_numbers = 6;  break;
    case 2: dice_numbers = 10; break;
    case 3: dice_numbers = 20; break;
  }
}
```

Também temos a variável number_of_dices.

```
// Parameters: incrementa "number_of_dices" (NEXT rising edge)
if (fsm1.state == fsm_parameters && SNEXT.re) {
  number_of_dices = (number_of_dices % 4) + 1;
}
```

4 Conclusão e aspetos a melhorar

Consideramos que o trabalho foi bem conseguido, mas é possível realçar alguns aspetos a melhorar.

No que toca à calibração, não existe qualquer tipo de controlo no caso do utilizador abanar o acelerómetro durante o momento da calibração. Isto poderia ser implementado, se houvesse algum valor da amostragem que não fosse coerente com os outros valores. Nesse caso, esse valor deveria ser descartado.

Para além disso, o display do estado de jogo não é claro, isto é, não indica ao utilizador o que deve ser feito. Acrescentando a essa falha, também não temos nenhum tipo de animação durante o jogo, ao contrário do estado de calibração.

Referências

[1] https://github.com/adafruit/Adafruit_SSD1306

[2] https://github.com/adafruit/Adafruit-GFX-Library