Figury w 3D sterowane ruchem

Michał Stręk, Damian Ćwik

Maj 2025

1 Wprowadzenie i Opis projektu

1.1 Opis projektu

Projekt "Figury w 3D sterowane ruchem" skupia się na wizualizacji oraz interaktywnym sterowaniu obiektami wyświetlanymi na ekranie LCD. Mamy możliwość prezentacji różnych kształtów, w tym sześcianu i piramidy, a także specjalnej poziomicy reagującej na orientację urządzenia.

1.2 Komponenty sprzętowe

1. FRDM-MCXN947
2. ICM-20948 9-DoF IMU
3. LCD display module st7735s

2 Implementacja programu

2.1 Struktura kodu

1. **Projekt.c** - zawiera główną funkcję **main()** odpowiedzialną za inicjalizację systemu i struktur **cube**, **pyramid**, **level**, a także realizację głównej pętli programu.
2. **figures.h, figures.c** - moduł odpowiedzialny za zarządzanie figurami 3D oraz ich przetwarzanie i rendering.
3. **moving_average_filter.h, moving_average_filter.c** - zawierają implementację filtru średniej ruchomej, który służy do wygładzania sygnałów z żyroskopu poprzez redukcję szumów i drgań.
4. **lcd.h, lcd.c, icm20948.h, icm20948.c** - zewnętrzne biblioteki do obługi ekranu, oraz żyroskopu 9 osiowego.

2.2 Implementacja kodu

2.2.1 figures.h

Plik nagłówkowy **figures.h** zawiera deklaracje struktur **Vec_2** oraz **Vec_3**, przeznaczonych do reprezentacji wektorów odpowiednio dwu- i trójelementowych.

1. **Vec_2**

```
typedef struct{
        uint16_t x, y;
}Vec_2;
```

2. **Vec_3**

```
typedef struct{
        float x, y, z;
}Vec_3;
```

Do przechowywania danych dla następujących figur zostały również zastosowane struktury:

1. **Cube**

```
typedef struct{
        Vec_3 vertices[CUBE_VERTICES_COUNT];
        Vec_2 edges[CUBE_EDGES_COUNT];
        uint16_t color;
        float angleX;
        float angleY;
        float angleZ;
}Cube;
```

2. **Pyramid**

```
typedef struct{
        Vec_3 vertices[PYRAMID_VERTICES_COUNT];
        Vec_2 edges[PYRAMID_EDGES_COUNT];
        uint16_t color;
        float angleX;
        float angleY;
        float angleZ;
}Pyramid;
```

3. **Level**

```
typedef struct {
        int x, y;
        float angleX;
        uint16_t color;
}Level;
```

2.2.2 figures.c

Plik **figures.c** zawiera kluczowe funkcje odpowiedzialne za transformacje geometryczne figur 3D oraz ich rzutowanie na płaszczyznę 2D. Implementacja bazuje na operacjach macierzowych oraz wektorowych, zapewniając możliwość manipulacji obiektami w przestrzeni trójwymiarowej przed ich wizualizacją na ekranie.

1. Macierz obrotu wokół osi x:

$$R_x(\theta) = \begin{vmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{vmatrix}$$

Zrealizowano w kodzie w następujący sposób:

```
void rotate_x(Vec_3 *v, const float angle) {
        float cosA = cos(angle);
        float sinA = sin(angle);
        float y = v->y * cosA - v->z * sinA;
        float z = v->y * sinA + v->z * cosA;
        v->y = y;
        v->z = z;

        return ;
}
```

2. Macierz obrotu wokół osi y:

$$R_y(\theta) = \begin{vmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{vmatrix}$$

Zrealizowano w kodzie w następujący sposób:

```
void rotate_y(Vec_3 *v, const float angle){
        float cosA = cos(angle);
        float sinA = sin(angle);
        float x = v->x * cosA - v->z * sinA;
        float z = v->x * sinA + v->z * cosA;
        v->x = x;
        v->z = z;

        return ;
}
```

3. Macierz obrotu wokół osi z:

$$R_z(\theta) = \begin{vmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Zrealizowano w kodzie w następujący sposób:

```
void rotate_z(Vec_3 *v, const float angle) {
        float cosA = cos(angle);
        float sinA = sin(angle);
        float x = v->x * cosA - v->y * sinA;
        float y = v->x * sinA + v->y * cosA;
        v->x = x;
        v->y = y;

        return;
}
```

W celu wyświetlenia figur trójwymiarowych na ekranie LCD, w kodzie zastosowano proces rzutowania z przestrzeni 3D na przestrzeń 2D. W tej funkcji pomijana jest współrzędna na osi z, a wykorzystywane są osie x i y.

Funkcja **cubeDraw** odpowiada za wizualizację sześcianu na ekranie LCD. Jej działanie polega na skopiowaniu oryginalnych wierzchołków, a następnie zastosowaniu na nich transformacji rotacji wokół osi x, y i z. Po tych operacjach, wierzchołki są rzutowane ortograficznie (z pominięciem współrzędnej z), skalowane i translowane do współrzędnych pikselowych ekranu. Finalnie, funkcja rysuje krawędzie sześcianu, łącząc przetworzone wierzchołki.

```
void cubeDraw(const Cube cube){
        Vec_3 transformedVerticies[CUBE_VERTICES_COUNT];

        for (int i = 0; i < CUBE_VERTICES_COUNT; i++) {
                transformedVerticies[i] = cube.vertices[i];
        }

        for(int i = 0; i < CUBE_VERTICES_COUNT; i++){
                rotate_y(&transformedVerticies[i], cube.angleY);
                rotate_x(&transformedVerticies[i], cube.angleX);
                rotate_z(&transformedVerticies[i], cube.angleZ);
        }

        for (int i = 0; i < CUBE_EDGES_COUNT; i++){
                Vec_3 p1 = transformedVerticies[cube.edges[i].x];
                Vec_3 p2 = transformedVerticies[cube.edges[i].y];

                uint16_t x1 = p1.x * 40 + LCD_WIDTH / 2;
                uint16_t y1 = p1.y * 40 + LCD_HEIGHT / 2;

                uint16_t x2 = p2.x * 40 + LCD_WIDTH / 2;
                uint16_t y2 = p2.y * 40 + LCD_HEIGHT / 2;

                LCD_Draw_Line(x1, y1, x2, y2, cube.color);
        }
}
```

Funkcja **pyramidDraw** jest odpowiedzialna za transformację i wizualizację piramidy na ekranie LCD. Jej działanie rozpoczyna się od obliczenia środka geometrycznego piramidy. Następnie, wierzchołki są przesuwane do początku układu współrzędnych, aplikowane są na nich rotacje wokół osi x, y i z, a po obrocie wierzchołki są przesuwane z powrotem do pierwotnej pozycji, co zapewnia obrót piramidy wokół jej własnego środka. Po transformacjach 3D, wierzchołki są rzutowane ortograficznie, skalowane i translowane do współrzędnych pikselowych ekranu. Finalnie, funkcja rysuje krawędzie piramidy, łącząc przetworzone wierzchołki.

```
void pyramidDraw(const Pyramid pyramid) {
        Vec_3 transformedVertices[PYRAMID_VERTICES_COUNT];

        Vec_3 center = {0, 0, 0};
        for (int i = 0; i < PYRAMID_VERTICES_COUNT; i++) {
                center.x += pyramid.vertices[i].x;
                center.y += pyramid.vertices[i].y;
                center.z += pyramid.vertices[i].z;
        }
        center.x /= PYRAMID_VERTICES_COUNT;
        center.y /= PYRAMID_VERTICES_COUNT;
        center.z /= PYRAMID_VERTICES_COUNT;

        for (int i = 0; i < PYRAMID_VERTICES_COUNT; i++) {
                transformedVertices[i] = pyramid.vertices[i];
```

```
                transformedVertices[i].x -= center.x;
                transformedVertices[i].y -= center.y;
                transformedVertices[i].z -= center.z;

                rotate_y(&transformedVertices[i], pyramid.angleY);
                rotate_x(&transformedVertices[i], pyramid.angleX);
                rotate_z(&transformedVertices[i], pyramid.angleZ);

                transformedVertices[i].x += center.x;
                transformedVertices[i].y += center.y;
                transformedVertices[i].z += center.z;
        }

        for (int i = 0; i < PYRAMID_EDGES_COUNT; i++) {
                Vec_3 p1 = transformedVertices[pyramid.edges[i].x];
                Vec_3 p2 = transformedVertices[pyramid.edges[i].y];

                uint16_t x1 = p1.x * 40 + LCD_WIDTH / 2;
                uint16_t y1 = p1.y * 40 + LCD_HEIGHT / 2;

                uint16_t x2 = p2.x * 40 + LCD_WIDTH / 2;
                uint16_t y2 = p2.y * 40 + LCD_HEIGHT / 2;

                LCD_Draw_Line(x1, y1, x2, y2, pyramid.color);
        }
}
```

Funkcja **levelDrawRotated** wizualizuje obiekt "Poziomica" (**Level**), reprezentujący linię obracającą się w płaszczyźnie ekranu. Ten 2D-obiekt, zdefiniowany jako linia, operuje w płaszczyźnie xy. W funkcji wykonywana jest rotacja linii o kąt **level.angleX** wokół środka ekranu, po której obliczane są współrzędne pikseli do wyświetlenia.

```
 float centerX = LCD_WIDTH / 2.0f;
float centerY = LCD_HEIGHT / 2.0f;

float lineLength = sqrtf(LCD_WIDTH * LCD_WIDTH + LCD_HEIGHT *
   LCD_HEIGHT);

float halfLength = lineLength / 2.0f;

float x1 = -halfLength;
float y1 = 0;
float x2 = halfLength;
float y2 = 0;

float angle = level.angleX;
float cosAngle = cosf(angle);
float sinAngle = sinf(angle);

float rotX1 = x1 * cosAngle - y1 * sinAngle;
float rotY1 = x1 * sinAngle + y1 * cosAngle;

float rotX2 = x2 * cosAngle - y2 * sinAngle;
float rotY2 = x2 * sinAngle + y2 * cosAngle;
```

```
uint16_t screenX1 = (uint16_t)(rotX1 + centerX);
uint16_t screenY1 = (uint16_t)(rotY1 + centerY);
uint16_t screenX2 = (uint16_t)(rotX2 + centerX);
uint16_t screenY2 = (uint16_t)(rotY2 + centerY);

if (screenX1 >= LCD_WIDTH) screenX1 = LCD_WIDTH - 1;
if (screenY1 >= LCD_HEIGHT) screenY1 = LCD_HEIGHT - 1;
if (screenX2 >= LCD_WIDTH) screenX2 = LCD_WIDTH - 1;
if (screenY2 >= LCD_HEIGHT) screenY2 = LCD_HEIGHT - 1;

LCD_Draw_Line(screenX1, screenY1, screenX2, screenY2, level.color);
```

2.3 moving_average_filter.h, moving_average_filter.c

Plik nagłówkowy **moving_average_filter.h** definiuje strukturę **MovingAverageBuffer**, która służy jako bufor danych filtru średniej ruchomej. Określa również stałą **SAMPLE_COUNT** (domyślnie 10), która odpowiada za liczbę próbek używanych do uśredniania. Znajdują się tu też deklaracje funkcji **initMovingAverage** oraz **updateMovingAverage**.

```
#define SAMPLE_COUNT 10

typedef struct {
        float buffer[SAMPLE_COUNT];
        uint8_t index;
} MovingAverageBuffer;
```

Funkcja **initMovingAverage** odpowiada za wyzerowanie bufora filtru, przygotowując go do pracy.

```
void initMovingAverage(MovingAverageBuffer *ma) {
        for (int i = 0; i < SAMPLE_COUNT; i++) {
                ma->buffer[i] = 0.0f;
        }
        ma->index = 0;
}
```

Funkcja **updateMovingAverage** odpowiada za aktualizację bufora filtru nową wartością oraz za obliczanie bieżącej średniej ruchomej ze zgromadzonych próbek.

```
float updateMovingAverage(MovingAverageBuffer *ma, float
    newValue) {
        ma->buffer[ma->index] = newValue;
        ma->index = (ma->index + 1) % SAMPLE_COUNT;

        float sum = 0.0f;
        for (int i = 0; i < SAMPLE_COUNT; i++) {
                sum += ma->buffer[i];
        }
        return sum / SAMPLE_COUNT;
}
```

2.4 Główny plik

Plik **main.c** odpowiada za inicjalizację całego systemu, oraz stworzenie i uruchomienie zadań w systemie RTOS.

2.4.1 Inicjalizacja figur

W pliku **main.c** inicjowane są struktury (**Cube**, **Pyramid**, **Level**), które definiują figury geometryczne. Na tym etapie przypisywane są im startowe wierzchołki, krawędzie, domyślne kąty obrotu (równe zeru) oraz kolory.

```
Cube cube = {
        .vertices = {
                {-1, -1, -1}, {1, -1, -1}, {1, 1, -1}, {-1, 1, -1},
                {-1, -1, 1}, {1, -1, 1}, {1, 1, 1}, {-1, 1, 1}
        },
        .edges = {
                {0, 1}, {1, 2}, {2, 3}, {3, 0},
                {4, 5}, {5, 6}, {6, 7}, {7, 4},
                {0, 4}, {1, 5}, {2, 6}, {3, 7}
        },
        .color = 0xFFFF,
};

Pyramid pyramid = {
        .vertices = {
                {-1, -0.577, -0.816}, {1, -0.577, -0.816}, {0, 1.155,
                    -0.816},
                {0, 0, 0.816}
        },
        .edges = {
                {0, 1}, {1, 2}, {2, 0},
                {0, 3}, {1, 3}, {2, 3}
        },
        .color = 0xFFFF,
};

Level level = {
        .x = 0,
        .y = LCD_HEIGHT / 2,
        .angleX = 0.2f,
        .color = 0xFFFF
};
```

2.4.2 Funkcja main

Na początku funkcji **main()** następuje inicjalizacja kluczowych peryferiów mikrokontrolera, obejmująca konfigurację portów GPIO, zegarów systemowych oraz interfejsów komunikacyjnych dla żyroskopu 9 osiowego (**IMU_ICM20948**) i ekranu LCD (**ST7735S**).

```
BOARD_InitBootPins();
BOARD_InitBootClocks();
BOARD_InitBootPeripherals();

#ifndef BOARD_INIT_DEBUG_CONSOLE_PERIPHERAL
BOARD_InitDebugConsole();
PRINTF("Run\r\n");
LCD_Init(LP_FLEXCOMM0_PERIPHERAL);
#endif
LCD_Init(LP_FLEXCOMM0_PERIPHERAL);
```

```
imuInit(LP_FLEXCOMM2_PERIPHERAL, &enMotionSensorType);


if(IMU_EN_SENSOR_TYPE_ICM20948 == enMotionSensorType)
{
        PRINTF("Motion sersor is ICM-20948\r\n" );
}
else
{
        PRINTF("Motion sersor NULL\r\n");
}
```

Następnie tworzone jest zadanie **drawTask** (task), które realizuje główną pętlę programu. Zadanie to odpowiada za kompleksowe zarządzanie wizualizacją figur, odczytem danych z żyroskopu oraz obsługę interakcji z przyciskiem.

```
if (xTaskCreate(draw_task, "DrawTask", 2048, NULL, 1, NULL) != pdPASS) {
        PRINTF("DrawTask creation failed!\r\n");
}

vTaskStartScheduler();
```

2.4.3 Funkcja drawTask

Na początku funkcji inicjowane są zmienne kątów obrotu (**angleX**, **angleY**, **angleZ**), ustawiane na zero. Tworzone są też filtry średniej ruchomej (**maX**, **maY**, **maZ**), które pomogą wygładzać odczyty z żyroskopu. Inicjowana jest również zmienna śledząca stan przycisku.

```
float angleX = 0.0f, angleY = 0.0f, angleZ = 0.0f;
float dt;
MovingAverageBuffer maX, maY, maZ;
initMovingAverage(&maX);
initMovingAverage(&maY);
initMovingAverage(&maZ);
bool sw2_0 = 1, sw2_1 = 1;
```

Następnie w głownej pętli programu sprawdzane jest czy przycisk został naciśnięty, jeśli tak to figura zostaje zmieniona na następną w kolejce.

```
sw2_1 = sw2_0;
sw2_0 = GPIO_PinRead(BOARD_INITBUTTONSPINS_SW2_GPIO,
    BOARD_INITBUTTONSPINS_SW2_GPIO_PIN);

if (sw2_0 == 0 && sw2_1 == 1) {
        currentFigure = (currentFigure + 1) % FIGURES_COUNT;
}
```

W każdej pętli programu pobierane są surowe dane z żyroskopu. Te odczyty są następnie konwertowane na przyrosty kątów (w radianach) za pomocą wzoru $\Delta\theta = \omega * \Delta t$ i wygładzane za pomocą filtrów średniej ruchomej. Na koniec, przefiltrowane przyrosty kątów są dodawane do ogólnych kątów obrotu (**angleX**, **angleY**, **angleZ**), aktualizując orientację figury.

```
imuDataGet2(&stAngles, &stGyroRawData, &stAccelRawData, &stMagnRawData);

float gyroX = (stGyroRawData.s16X / 131.0) * 3.1415926f  * dt;
float gyroY = (stGyroRawData.s16Y / 131.0) * 3.1415926f  * dt;
```

```
float gyroZ = (stGyroRawData.s16Z / 131.0) * 3.1415926f  * dt;

float filteredGyroX = updateMovingAverage(&maX, gyroX);
float filteredGyroY = updateMovingAverage(&maY, gyroY);
float filteredGyroZ = updateMovingAverage(&maZ, gyroZ);

angleX += filteredGyroX * dt;
angleY += filteredGyroY * dt;
angleZ += filteredGyroZ * dt;
```

Po przetworzeniu danych z żyroskopu, ekran LCD jest najpierw czyszczony. Następnie, w zależności od wybranej figury (**CUBE**, **PYRAMID** lub **LEVEL**), jej kąty obrotu są aktualizowane na podstawie odczytów z żyroskopu, a figura jest rysowana na ekranie. Po narysowaniu, bufor graficzny LCD jest odświeżany, co powoduje wyświetlenie obrazu. Na koniec, zadanie jest usypiane na krótki czas, aby kontrolować częstotliwość odświeżania i oszczędzać zasoby procesora.

```
LCD_Clear(0x00);

if (currentFigure == CUBE) {
        cube.angleX = angleX;
        cube.angleY = angleY;
        cube.angleZ = angleZ;
        cubeDraw(cube);
} else if (currentFigure == PYRAMID) {
        pyramid.angleX = angleX;
        pyramid.angleY = angleY;
        pyramid.angleZ = angleZ;
        pyramidDraw(pyramid);
}else if (currentFigure == LEVEL) {
        level.angleX = angleX;
        levelDrawRotated(level);
}

LCD_GramRefresh();
vTaskDelay(MSEC_TO_TICK(DELAY));
```