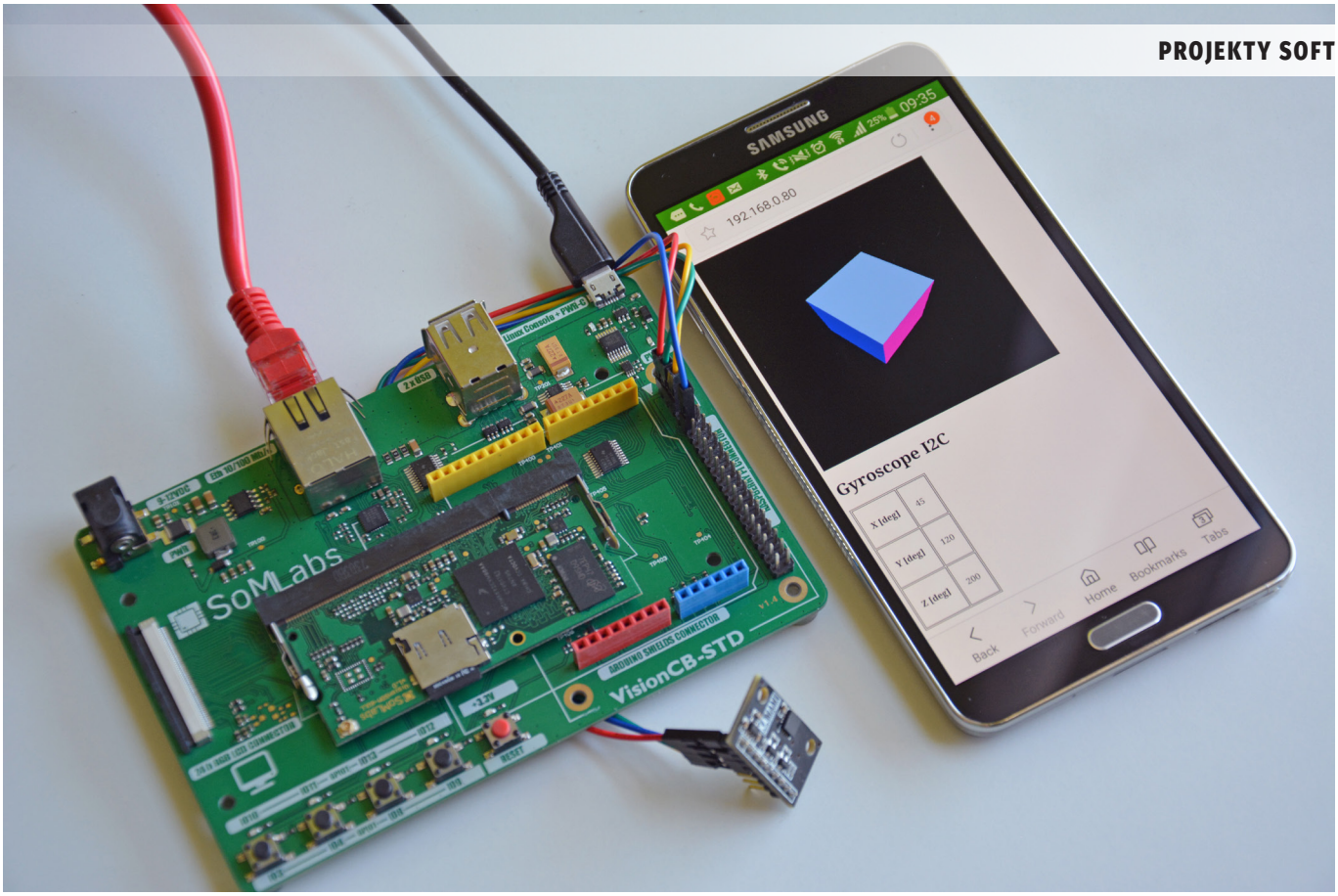


Serwer WWW z elementami grafiki 3D (2)

W drugiej części artykułu kontynuujemy tematykę praktycznego użycia pakietów Node.js oraz Three.js w urządzeniach wbudowanych pracujących pod kontrolą system operacyjnego Linux. Dotychczas zostały omówione zagadnienia instalacji frameworku Node.js, przygotowania prostego serwera WWW z podziałem na funkcje front-end/back-end, komunikacji z wykorzystaniem socket.io, a także tworzenia i odczytu danych z procesów potomnych. W artykule, na przykładzie modułu KAmodL3GD20, zaimplementujemy obsługę żyroskopu w przestrzeni użytkownika z wykorzystaniem interfejsu i2c-dev, a następnie rozbudujemy projekt graficzny strony WWW o elementy grafiki 3D, która w postaci sześciennej kostki będzie odwzorowywała ruch podłączonego modułu żyroskopowego.

W poprzedniej części artykułu, na potrzeby testów odczytu danych z procesu potomnego, przygotowano prosty kod *gryo-i2c.c*, którego zadaniem było generowanie losowych wartości dla mierzonego kąta obrotu w osiach X, Y oraz Z. W tym podrozdziale zaimplementujemy właściwą obsługę układu żyroskopowego *L3GD20*[1] firmy *STMicroelectronics*, zamontowanego na wygodnej do użycia płytce prototypowej *KAmodL3GD20* [2] (**fotografia 1**).

Moduł KAmodL3GD20 – konfigurowanie jądra Linux i obsługa programowa

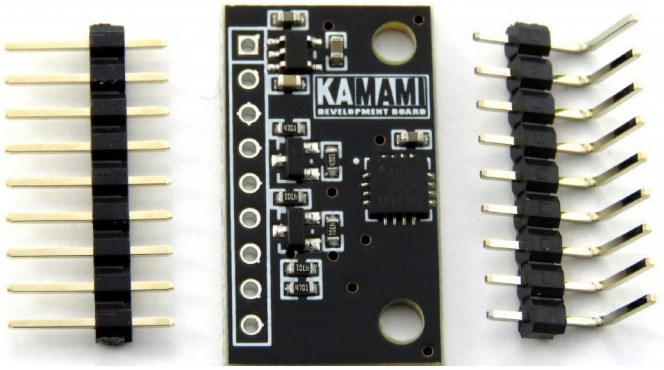
Wykorzystany do praktycznej realizacji projektu procesor *i.MX6 ULL* z rdzeniem *Cortex-A7*, został wyposażony w cztery sprzętowe kontrolery magistrali I2C. Konfigurację jądra systemu Linux rozpoczynamy więc od wywołania narzędzia menuconfig, a następnie włączenia sterowników dla sprzętowego kontrolera magistrali:

```
Device Drivers --->
    [*] I2C support --->
        [*] I2C Hardware Bus Support --->
            <*> IMX I2C interface
            < > GPIO-based bitbanging I2C
```

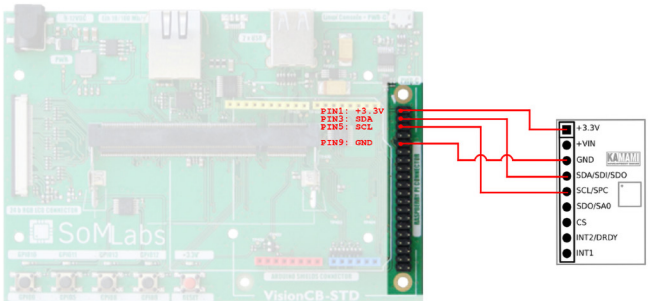
Do komunikacji z układem L3GD20 wykorzystany zostanie interfejs *i2c-dev*, który umożliwia uzyskanie dostępu do magistrali z poziomu przestrzeni użytkownika poprzez pliki specjalne urządzeń */dev/i2c-x* (gdzie wartość x oznacza kolejny numer porządkowy interfejsu I²C). Interfejs ten, poprzez wygodne API, umożliwia przygotowanie obsługi urządzenia peryferyjnego bezpośrednio w przestrzeni użytkownika – z pominięciem sterowników w jądrze systemu. Rozwiązanie to jest praktyczne na etapie wczesnego projektowania obsługi sprzętu (brak potrzeby rekompilacji jądra lub modułu) lub w sytuacji gdy sterownik wybranego urządzenia peryferyjnego nie został zaimplementowany w systemie. Włączenie interfejsu *i2c-dev* w jądrze systemu:

```
Device Drivers --->
    [*] I2C support --->
        <*> I2C device interface
```

Po zakończonym procesie konfiguracji jądra, niezbędna jest również edycja opisu *Device Tree* w którym to aktywujemy wybrany



Rysunek 1. Moduł żyroskopowy KAmodL3GD20 (źródło: kamami.pl)



Rysunek 2. Schemat połączeń modułu KAmodL3GD20 z płytą bazową VisionCB-STD

kontroler I²C – poprzez edycję pola status – oraz dokonamy konfiguracji funkcji alternatywnych dla wybranych wyprowadzeń procesora.

W prezentowanym projekcie wybrano wyprowadzenia numer 3 i 5 gniazda *J504* (umiejscowionego na płycie bazowej *VisionCB-STD* dla modułów *VisionSOM*), które mogą pełnić alternatywną funkcję linii *SDA* i *SCL* dla kontrolera *I2C2* (domyślnie linie te są sygnałami *TX* oraz *RX* kontrolera *UART5*). Pełny schemat połączeń przedstawiono na **rysunku 2**.

Bazując na schemacie połączeń z rysunku 2, dokonajmy edycji opisu *Device Tree*, w którym to aktywujemy kontroler magistrali *I2C2*:

```
&i2c2 {
    clock_frequency = <100000>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_i2c2>;
    status = "okay";
};
```

oraz wyprowadzeniom MX6UL_PAD_UART5_TX_DATA i MX6UL_PAD_UART5_RX_DATA, przypiszemy alternatywne funkcje linii SCL oraz SDA:

Name	Type	Register address		Default
		Hex	Binary	
Reserved	-	00-0E	-	-
WHO_AM_I	r	0F	000 1111	11010100
Reserved	-	10-1F	-	-
CTRL_REG1	rw	20	010 0000	00000111
CTRL_REG2	rw	21	010 0001	00000000
CTRL_REG3	rw	22	010 0010	00000000
CTRL_REG4	rw	23	010 0011	00000000
CTRL_REG5	rw	24	010 0100	00000000
REFERENCE	rw	25	010 0101	00000000
OUT_TEMP	r	26	010 0110	output
STATUS_REG	r	27	010 0111	output
OUT_X_L	r	28	010 1000	output
OUT_X_H	r	29	010 1001	output
OUT_Y_L	r	2A	010 1010	output
OUT_Y_H	r	2B	010 1011	output
OUT_Z_L	r	2C	010 1100	output
OUT_Z_H	r	2D	010 1101	output

Rysunek 3. Zestaw wybranych rejestrów układu L3GD20

Do realizacji opisywanego projektu użyto komputera jednopłytkowego VisionSOM pracującego pod kontrolą dystrybucji Debian. Autor celowo pominął opisy przygotowania komputera do pracy – przygotowanie karty SD z systemem Debian oraz konfiguracja i kompilacja jądra systemu i plików Device Tree została przedstawiona w ramach artykułu „Emulator konsoli NES w systemie Linux na komputerze VisionSOM" publikowanego na łamach „Elektroniki Praktycznej" 03/2018. Komplet informacji został również udostępniony na stronie Wiki producenta komputera [3].

```
&iomuxc {
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_hog_1>;
    imx6ul-evk {
        pinctrl_i2c2: i2c2grp {
            fsl,pins = <
                MX6UL_PAD_UART5_TX_DATA__I2C2_SCL
0x4001b8b0
                MX6UL_PAD_UART5_RX_DATA__I2C2_SDA
0x4001b8b0
            >;
        };
    };
};
```

Po zakończonej konfiguracji jądra oraz edycji plików *Device Tree*, można przystąpić do kompilacji i zaktualizowania obrazu systemu.

Po ponownym uruchomieniu komputera, poprawność konfiguracji i kompilacji jądra systemu możemy sprawdzić poprzez wyświetlenie listy dostępnych kontrolerów I²C w katalogu */dev*:

```
root@somlabs:~# ls -l /dev/ | grep i2c
crw-rw---- 1 root i2c     89,    1 May 25 18:36 i2c-1
```

Różnica w numeracji kontrolerów (w pliku *Device Tree* aktywowano kontroler *I2C2*, natomiast w katalogu */dev* odnajdujemy kontroler *i2c-1*) wynika z różnic indeksowania – jądro systemu stosuje numerację zaczynającą się od wartości 0. Poprawnie skonfigurowany system pozwala na przejście do kolejnego etapu – weryfikacji połączeń sprzętowych. Jedną z najszybszych metod na nawiązanie komunikacji z podłączonym do magistrali sprzętem jest wykorzystanie pakietu *i2c-tools* w skład którego wchodzą między innymi takie narzędzia jak *i2cdetect*, *i2cset* oraz *i2cget*. Instalacja pakietu *i2c-tools* w dystrybucji *Debian*, odbywa się w sposób standardowy dla narzędzia *apt-get*:

```
root@somlabs:~# apt-get install i2c-tools
Reading package lists... Done
Building dependency tree
Reading state information... Done
Suggested packages:
  libi2c-dev python-smbus
The following NEW packages will be installed:
  i2c-tools
0 upgraded, 1 newly installed, 0 to remove and 39
not upgraded.
Need to get 0 B/57.6 kB of archives.
After this operation, 196 kB of additional disk
space will be used.
```

Ponieważ zagadnienia związane z obsługą modułu żyroskopu są jedynie tematem uzupełniającym główną tematykę artykułu (pomimo tego, że w drugiej części artykułu stanowią jego obszerny element), po szczegółowe informacje związane z programową obsługą układu L3GD20, Autor odsyła do dokumentacji układu [1].

Listing 1. Otworzenie pliku urządzenia /dev/i2c-1 oraz ustawienie adresu I2C_SLAVE

```
#define GYRO_ADDR    0x6b

int main (void)
{
    int i2c_fd, ret;
    /* open i2c device */
    i2c_fd = open ("/dev/i2c-1", O_RDWR);
    if (i2c_fd < 0)
    {
        printf ("Failed to open the i2c bus\n");
        return EXIT_FAILURE;
    }
    /* set slave address */
    ret = ioctl (i2c_fd, I2C_SLAVE, GYRO_ADDR);
    if (ret < 0)
    {
        printf ("Failed to acquire bus access and/or talk
to slave\n");
        goto exit;
    }
}
```

Program *i2cdetect* umożliwia przeskanowanie wybranej magistrali I²C, wskazanej poprzez parametr -y. Wynikiem działania polecenia jest tablica adresów wraz z listą dostępnych na magistrali urządzeń:

```
root@somlabs:~# i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- UU -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- 6b -- -- --
70: -- -- -- -- -- -- -- --
```

Polecenie *i2cdetect* poprawnie wykryło podłączony do magistrali układ L3GD20 o przypisanym przez producenta adresie 0x6B (wykryte jednocześnie urządzenie o adresie 0x38 z oznaczeniem UU, jest obsługiwane przez sterownik w jądrze i nie jest dostępne w przestrzeni użytkownika). W ramach dodatkowego testu, korzystając z narzędzia *i2cget*, można wykonać odczyt zawartości rejestru *WHO_AM_I* układu L3GD20 – rejestr ten przechowuje ustalony przez producenta numer identyfikacyjny układu – zestawienie wybranych rejestrów żyroskopu L3GD20 przedstawiono na **rysunku 3**.

Polecenie *i2cget* pozwala na odczyt danych z wybranego układu podrzędnego i określonego rejestru. Pierwszym parametrem polecenia jest numer porządkowy magistrali, następnie adres układu podrzędnego, numer rejestru spod którego będziemy wykonywać odczyt oraz rozmiar odczytywanej danej (domyślnie jeden bajt). Odczyt zawartości rejestru 0x0F (*WHO_AM_I*) z układu o adresie 0x6B podłączonego do magistrali *i2c_1*, może zostać realizowany następująco:

```
root@somlabs:~# i2cget -y 1 0x6b 0x0F
0xd4
```

Odczytana wartość 0xD4 jest zgodna z wartością ustaloną przez producenta, tak więc komunikacja z układem L3GD20 przebiega prawidłowo – możemy przystąpić do implementacji obsługi żyroskopu na potrzeby programu *i2c-gyro*.

Funkcję main() z pliku *gyro-i2c.c* rozpoczynamy od otworzenia pliku urządzenia */dev/i2c-1*, a następnie za pomocą wywołania ioctl (fd, I2C_SLAVE, adres), ustawienia adresu urządzenia peryferyjnego z którym będzie realizowana dalsza wymiana komunikatów

Listing 2. Inicjalizacja układu L3GD20 poprzez zapis rejestrów CTRL_REG[X]

```
#define AUTO_INCREMENT  0x80
static int gyro_init (int i2c_fd)
{
    unsigned char init_seq[6];
    init_seq[0] = (CTRL_REG1 | AUTO_INCREMENT);
    init_seq[1] = 0xCF; /* CTRL_REG1: normal mode, xyz enable */
    init_seq[2] = 0x01; /* CTRL_REG2: <default value> */
    init_seq[3] = 0x00; /* CTRL_REG3: <default value> */
    init_seq[4] = 0x80; /* CTRL_REG4: 250dps, Block Data Update
*/
    init_seq[5] = 0x02; /* CTRL_REG5: <default value> */
    if (write (i2c_fd, init_seq, 6) != 6) return -1;
    return 0;
}
```

Listing 3. Funkcja odczytu prędkości kątowych w osiach X, Y i Z

```
static int gyro_get_xyz (int i2c_fd, float *x, float *y, float *z)
{
    unsigned char reg_addr = OUT_X_L | AUTO_INCREMENT;
    unsigned char reg_data[6];
    int ret;

    struct i2c_msg messages[] =
    {
        {
            GYRO_ADDR,
            0,
            sizeof(reg_addr),
            &reg_addr
        },
        {
            GYRO_ADDR,
            I2C_M_RD,
            sizeof(reg_data),
            reg_data
        }
    };

    struct i2c_rdwr_ioctl_data packets =
    {
        messages,
        sizeof(messages) / sizeof(struct i2c_msg)
    };

    ret = ioctl (i2c_fd, I2C_RDWR, &packets);
    if (ret < 0) return ret;
    *x = (short) (reg_data[0] + ((short)reg_data[1] << 8));
    *y = (short) (reg_data[2] + ((short)reg_data[3] << 8));
    *z = (short) (reg_data[4] + ((short)reg_data[5] << 8));
    return 0;
}
```

– **listing 1**. W następnej kolejności, poprzez zapis rejestrów sterujących *CTRL_REG[x]*, wykonamy inicjalizację układu L3GD20. Do tego celu przygotowano funkcję gyro_init(), której zadaniem jest wysłanie na magistralę 6. bajtów danych – adresu rejestru *CTRL_REG1* oraz kolejnych wartości wpisywanych odpowiednio do rejestrów *CTRL_REG1 – CTRL_REG5*. W celu optymalizacji funkcji gyro_init() do postaci pojedynczego wywołania write(), wykorzystano wbudowany w układ mechanizm automatycznego zwiększania wartości adresu po każdym zapisie bajtu danych – włączenie tej funkcji wymaga ustawienia najstarszego bitu w adresie rejestru. W ramach funkcji inicjalizacji włączono pomiary w osiach X, Y oraz Z, ustawiono układ do pracy w trybie odpytywania (wymagający sprawdzenia bitu gotowości w rejestrze *STATUS_REG*), ustawiono zakres pomiarowy na wartość 250 dps oraz blokową aktualizację danych (wartości pomiarowe dla osi X, Y i Z przechowywane są w postaci 16-bitowej – włączenie funkcji *Block Data Update* zapewnia, że dane przechowywane w starszej i młodszej części rejestru pochodzą z jednej próbki pomiaru). Kod funkcji gyro_init() został przedstawiony na **listingu 2**.

Przedstawiona na *Listingu 2* funkcja gyro_init() realizuje wyłącznie prostą operację zapisu za pomocą wywołania write(). W przypadku funkcji realizujących odczyt danych z wybranych rejestrów (np. rejestru statusu lub rejestrów przechowujących dane pomiarowe) nie możemy wykorzystać najprostszych wywołań systemowych read()/write(), ponieważ każde z tych wywołań generuje bit stopu. Układy peryferyjne I²C o „organizacji rejestrowej" wymagają złożonych sekwencji zapis/odczyt bez generowania bitu stopu pomiędzy tymi operacjami. Do tego celu należy wykorzystać wywołanie systemowe ioctl (fd, I2C_RDWR, struct i2c_rdwr_ioctl_data *msgset), które umożliwia wykonanie dowolnej transakcji na magistrali I²C w formie pojedynczej sekwencji. Ostatni argument powyższego wywołania stanowi wskaźnik do struktury i2c_rdwr_ioctl_data:

```
struct i2c_rdwr_ioctl_data
{
    struct i2c_msg *msgs;
    __u32 nmsgs;
};
```

Pole nmsgs określa liczbę transakcji w pojedynczej sekwencji, natomiast pole msgs zawiera wskaźnik do tablicy struktur opisujących poszczególne transakcje w postaci:

```
struct i2c_msg
{
  __u16 addr; /* slave address */
```

```
  __u16 flags;
  #define I2C_M_TEN 0x0010
  #define I2C_M_RD 0x0001
  /* ... */
  #define I2C_M_RECV_LEN 0x0400
  __u16 len; /* msg length */
  __u8 *buf; /* pointer to msg data */
};
```

Pole addr struktury i2c_msg opisuje sprzętowy adres układu peryferyjnego którego dotyczy dana transakcja. Pole flags umożliwia sterowanie daną transakcją poprzez zestaw dodatkowych flag, np. flaga I2C_M_RD informuje, że dana transakcja jest transakcją odczytu. Pole len określa długość bufora z danymi do wysłania/odebrania, natomiast pole *buf zawiera wskaźnik do bufora danych. Na bazie wywołania ioctl (fd, I2C_RDWR, struct i2c_rdwr_ioctl_data *msgset) utworzono funkcję gyro_get_xyz() odczytującą prędkości kątowe w osiach X, Y i Z (**listing 3**).

Analogicznie do funkcji gyro_get_xyz() utworzono również funkcję gyro_get_status(), której zadaniem jest odczyt rejestru statusu *STATUS_REG* i sprawdzanie bitu informującego o gotowości danych do odczytu. Ponieważ projekt wykorzystuje wyłącznie pomiary żyroskopowe (bez korelacji danych np. z pomiarami z akcelerometru) niezbędna jest również najprostsza kalibracja układu - funkcja gyro_calib() na podstawie 200. pomiarów wartości spoczynkowej określa przedziały wyznaczające brak ruchu żyroskopu. W głównej pętli programu, prędkość kątowa przeliczana jest na wartość kąta obrotu (drogę) na podstawie prostego całkowania – funkcja get_timestamp() określa wartości przedziałów czasowych dt pomiędzy kolejnymi iteracjami pętli. Choć tak zrealizowana metoda pomiarowa jest obarczona dość dużym błędem, jest ona wystarczająca do odwzorowania ruchu obiektu w postaci animowanej kostki 3D wyświetlanej w graficznym interfejsie użytkownika. Kompletny kod źródłowy programu *gyro-i2c* został przedstawiony na **listingu 4**.

Node.js – rozbudowa interfejsu o proste elementy grafiki 3D (Three.js)

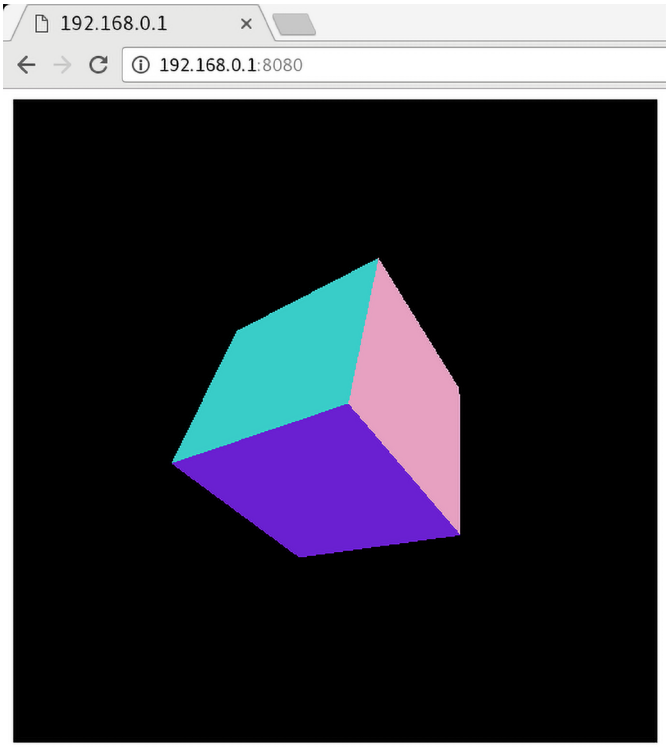
Na obecnym etapie realizacji projektu przygotowane zostały już wszystkie elementy składowe. W poprzedniej części artykułu omówiono *back-end* w postaci serwera WWW ze skryptu *main.js*, którego zadaniem był odczyt danych z procesu potomnego *gyro-i2c* oraz przesłanie danych za pomocą *socket.io* do warstwy *front-end* – interfejsu graficznego w postaci strony *index.html*.

W ostatnim podrozdziale artykułu rozbudujemy interfejs graficzny aplikacji o prostą grafikę 3D w postaci sześciennej kostki, odwzorowującej ruch podłączonego modułu żyroskopu. Do realizacji operacji graficznych wykorzystamy bibliotekę *Three.js*, która to natomiast korzysta z API *WebGL* – oficjalnego rozszerzenia możliwości języka JavaScript o interfejs grafiki 3D.

Bezpośrednie wykorzystanie interfejsu *WebGL* jest dość uciążliwe, choćby ze względu na dużą liczbę operacji niskiego poziomu, jakie spoczywają na programiście – definicja wierzchołków, buforów, macierzy transformacji, operacje związane z wyświetlaniem sceny, obsługa *shaderów*, oświetlenia, modeli, kamer i wiele innych.

W bibliotece *Three.js* scena budowana jest z obiektów (sama scena jest również obiektem, w którym umieszczamy inne obiekty). Do podstawowych obiektów możemy zaliczyć: figury geometryczne (biblioteka posiada zdefiniowane kilka gotowych do użycia obiektów takich jak sfera czy sześcian), materiały przypisywane do figur geometrycznym (określające m.in. ich kolor i fizykę odbijania światła), źródła światła oraz obserwatora sceny (czyli „kamerę", która obserwuje scenę w określonym położeniu).

Rozbudowę aplikacji rozpoczynamy od pobrania kodu biblioteki *Three.js* (plik *three.min.js*) do katalogu w którym umieszczono



Rysunek 4. Prezentacja danych odczytanych z żyroskopu w postaci animacji 3D

Ponieważ kod odpowiedzialny za animację 3D jest wykonywany przez przeglądarkę po stronie komputera użytkownika, należy upewnić się, że wybrana przez nas przeglądarka internetowa wspiera API WebGL v1.

skrypt *main.js* oraz stronę *index.html* (pełny kod źródłowy skryptu *main.js* oraz strony *index.html* został przedstawiony na listingu 6 i listingu 9 w pierwszej części artykułu):

```
wget http://threejs.org/build/three.min.js
```

Edycję pliku *index.html* rozpoczynamy od zdefiniowania w sekcji <head> „płótna" canvas (o wymiarach 500x500px oraz identyfikatorze mycanvas) w którym będzie renderowana docelowa animacja:

```
<canvas id="mycanvas" width="500" height="500">
</canvas>
```

Następnie w sekcji <head> dołączamy bibliotekę *Three.js*:

```
<script src='three.min.js'></script>
```

W dalszej części skryptu definiujemy zmienne w których będziemy przechowywać wyniki pomiarów w osi x, y, z oraz informacje o tworzonej scenie i dołączonych do niej obiektach:

```
var camera, scene, renderer;
var geometry, material, mesh;
var x, y, z;
```

Listing 4. Kompletny kod źródłowy aplikacji gyro-i2c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/i2c-dev.h>
#include <linux/i2c.h>
#include <sys/time.h>

#define WHO_AM_I  0x0F
#define CTRL_REG1  0x20
#define CTRL_REG2  0x21
#define CTRL_REG3  0x22
#define CTRL_REG4  0x23
#define CTRL_REG5  0x24
#define REFERENCE  0x25
#define OUT_TEMP  0x26
#define STATUS_REG  0x27
#define OUT_X_L  0x28
#define OUT_X_H  0x29
#define OUT_Y_L  0x2A
#define OUT_Y_H  0x2B
#define OUT_Z_L  0x2C
#define OUT_Z_H  0x2D
#define FIFO_CTRL_REG  0x2E
#define FIFO_SRC_REG  0x2F
#define INT1_CFG  0x30
#define INT1_SRC  0x31
#define INT1_TSH_XH  0x32
#define INT1_TSH_XL  0x33
#define INT1_TSH_YH  0x34
#define INT1_TSH_YL  0x35
#define INT1_TSH_ZH  0x36
#define INT1_TSH_ZL  0x37
#define INT1_DURATION  0x38
#define GYRO_ADDR  0x6b
#define AUTO_INCREMENT  0x80

int x_low = 0, y_low = 0, z_low = 0;
int x_high = 0, y_high = 0, z_high = 0;

static unsigned long get_timestamp ()
{
    struct timeval tv;
    gettimeofday (&tv,NULL);
    return tv.tv_sec * 1000000UL + tv.tv_usec;
}

static int gyro_init (int i2c_fd)
{
    unsigned char init_seq[6];

    init_seq[0] = (CTRL_REG1 | AUTO_INCREMENT);
    init_seq[1] = 0xCF; /* CTRL_REG1: normal mode, xyz enable */
    init_seq[2] = 0x01; /* CTRL_REG2: <default value> */
    init_seq[3] = 0x00; /* CTRL_REG3: <default value> */
    init_seq[4] = 0x80; /* CTRL_REG4: 250dps, Block Data Update */
    init_seq[5] = 0x02; /* CTRL_REG5: <default value> */
    if (write (i2c_fd, init_seq, 6) != 6) return -1;
    return 0;
}

static int gyro_get_status (int i2c_fd)
{
    unsigned char reg_addr = STATUS_REG;
    unsigned char reg_data[1];
    int ret;

    struct i2c_msg messages[] =
    {
        {
            GYRO_ADDR,
            0,
            sizeof(reg_addr),
            &reg_addr
        },
        {
            GYRO_ADDR,
            I2C_M_RD,
            sizeof(reg_data),
            reg_data
        }
    };

    struct i2c_rdwr_ioctl_data packets =
    {
        messages,
        sizeof(messages) / sizeof(struct i2c_msg)
    };

    ret = ioctl (i2c_fd, I2C_RDWR, &packets);
    if (ret < 0) return ret;
    return (reg_data[0] & (1 << 3));
}

static int gyro_get_xyz (int i2c_fd, float *x, float *y, float *z)
{
    unsigned char reg_addr = OUT_X_L | AUTO_INCREMENT;
    unsigned char reg_data[6];
    int ret;

    struct i2c_msg messages[] =
    {
        {
            GYRO_ADDR,
            0,
            sizeof(reg_addr),
            &reg_addr
        },
        {
            GYRO_ADDR,
            I2C_M_RD,
            sizeof(reg_data),
            reg_data
        }
    };

    struct i2c_rdwr_ioctl_data packets =
    {
        messages,
        sizeof(messages) / sizeof(struct i2c_msg)
    };

    ret = ioctl (i2c_fd, I2C_RDWR, &packets);
    if (ret < 0) return ret;
    *x = (short) (reg_data[0] + ((short)reg_data[1] << 8));
    *y = (short) (reg_data[2] + ((short)reg_data[3] << 8));
    *z = (short) (reg_data[4] + ((short)reg_data[5] << 8));
    return 0;
}

static int gyro_calib (int i2c_fd)
{
    float x_raw, y_raw, z_raw;
    int ret;

    for (int i =0 ; i < 200 ; i++)
    {
        while (!gyro_get_status (i2c_fd));
        ret = gyro_get_xyz (i2c_fd, &x_raw, &y_raw, &z_raw);
        if (ret < 0) break;
        if (x_raw > x_high) x_high = x_raw;
        else if (x_raw < x_low) x_low = x_raw;
            if (y_raw > y_high) y_high = y_raw;
            else if (y_raw < y_low) y_low = y_raw;
                if (z_raw > z_high) z_high = z_raw;
                    else if (z_raw < z_low) z_low = z_raw;
    }
  return ret;
}

int main (void)
{
    int i2c_fd, ret;
    float x_raw, y_raw, z_raw;
    unsigned long pt = 0;
    /* actual angles */
    float angX = 0;
    float angY = 0;
    float angZ = 0;
    /* previous angles for calculation */
    float p_angX = 0;
    float p_angY = 0;
    float p_angZ = 0;
    /* open i2c device */
    i2c_fd = open ("/dev/i2c-1", O_RDWR);
    if (i2c_fd < 0)
    {
        printf ("Failed to open the i2c bus\n");
        return EXIT_FAILURE;
    }
    /* set slave address */
    ret = ioctl (i2c_fd, I2C_SLAVE, GYRO_ADDR);
    if (ret < 0)
    {
        printf ("Failed to acquire bus access and/or talk
to slave\n");
        goto exit;
    }
    /* gyro init */
    ret = gyro_init (i2c_fd);
    if (ret < 0)
    {
        printf ("gyro_init error!\n");
        goto exit;
    }
  /* gyro calib */
    puts ("Calibration...");
    ret = gyro_calib (i2c_fd);
    if (ret < 0)
    {
        printf ("gyro_calib error!\n");
        goto exit;
    }

    while (1)
    {
        while (!gyro_get_status (i2c_fd));
        /* read xyz raw values */
        gyro_get_xyz (i2c_fd, &x_raw, &y_raw, &z_raw);
        /* get timestamp */
        unsigned long ct = get_timestamp();
        if (pt == 0)
        {
            pt = get_timestamp();
            continue;
        }
        float dt = (float) (ct - pt) / 1000000.0;
        pt = get_timestamp();
        /* x-axis */
        if (x_raw >= x_high || x_raw <= x_low)
        {
            angX += ((p_angX + (x_raw * 0.00875))/2) * dt;
            p_angX = x_raw * 0.00875;
        }
        else p_angX = 0;
        /* y-axis */
        if (y_raw >= y_high || y_raw <= y_low)
        {
            angY += ((p_angY + (y_raw * 0.00875))/2) * dt;
            p_angY = y_raw * 0.00875;
        }
        else p_angY = 0;
        /* z-axis */
        if (z_raw >= z_high || z_raw <= z_low)
        {
            angZ += ((p_angZ + (z_raw * 0.00875))/2) * dt;
            p_angZ = z_raw * 0.00875;
        }
        else p_angZ = 0;
        printf ("%.1f %.1f %.1f\n", angX, angY, angZ);
        fflush (stdout);
    }
    exit:
    return EXIT_FAILURE;
}
```

Listing 5. Budowanie sceny z wykorzystaniem biblioteki Three.js

```
function init()
{
    scene = new THREE.Scene();
    camera = new THREE.PerspectiveCamera (70, 500/500, 0.01, 10);
    camera.position.z = 0.5;
    geometry = new THREE.BoxGeometry (0.2, 0.2, 0.2);
    material = new THREE.MeshNormalMaterial();
    mesh = new THREE.Mesh (geometry, material);
    scene.add (mesh);
    renderer = new THREE.WebGLRenderer ({ canvas: mycanvas});
    renderer.setSize (500, 500);
    document.body.appendChild (renderer.domElement);
}
```

Następnie implementujemy funkcję init(), której zadaniem jest zbudowanie sceny z określonych obiektów – **listing 5**. W pierwszej linii kodu funkcji init() tworzymy scenę, do której będziemy dołączali kolejno definiowane obiekty (kamerę, figurę geometryczną oraz materiał dla tej figury):

```
scene = new THREE.Scene();
```

W następnym kroku tworzymy obiekt kamery określając kąt jej widzenia (70 stopni), proporcje kadru, zakresy widzenia: bliski i daleki, a także jej umiejscowienie:

```
camera = new THREE.PerspectiveCamera (70, 500/500,
0.01, 10);
camera.position.z = 0.5;
```

Korzystając ze zdefiniowanych w bibliotece *Three.js* kształtów, tworzymy obiekt reprezentujący sześcian (BoxGeometry):

```
geometry = new THREE.BoxGeometry (0.2, 0.2, 0.2);
```

oraz obiekt stanowiący „materiał" z jakiego wykony jest nasz sześcian (decyduje on m.in. o kolorze obiektu i sposobie rozpraszania światła) – wykorzystamy tutaj predefiniowany materiał MeshNormalMaterial:

```
material = new THREE.MeshNormalMaterial();
```

Z połączenia figury z materiałem możemy utworzyć obiekt klasy Mesh, który dodajemy do tworzonej sceny:

```
mesh = new THREE.Mesh (geometry, material);
scene.add (mesh);
```

W ostatnich liniach funkcji init(), określamy rozmiar i identyfikator powierzchni (mycanvas), na której będzie renderowana animacja:

```
renderer = new THREE.WebGLRenderer ({ canvas:
mycanvas});
renderer.setSize (500, 500);
document.body.appendChild (renderer.domElement);
```

Na **listingu 6** przedstawiono kod funkcji animate(), której zadaniem jest wykonanie obrotu obiektu, zgodnie z kątem obrotu zapisanym w zmiennych x, y, z. Pełna zawartość pliku *index.html* (wraz z wyróżnieniem zmian wprowadzonych w stosunku do poprzedniej części artykułu) została przedstawiona na **listingu 7**. Niewielkiej modyfikacji wymaga również sam kod serwera *main.js*. Dotychczas serwer na żądnie klienta udostępniał wyłącznie plik *index.html*. W aktualnie formie, przy ładowaniu strony głównej, klient zażąda również pliku *three.min.js* – serwer powinien to żądanie obsłużyć i dostarczyć klientowi wymaganą bibliotekę (**listing 8**). Po skopiowaniu do katalogu */tmp* skompilowanej wersji kodu *gyro-i2c* z listingu 4 oraz ponownym skryptu *main.js*, uruchomiony na komputerze jednopłytkowym serwer WWW prezentuje dane pomiarowe w postaci animowanej kostki 3D, jak przedstawiono to na **rysunku 4**.

Łukasz Skalski
contact@lukasz-skalski.com

1. ***http://bit.ly/2LPYzUS***
2. ***http://bit.ly/2mZilz3***
3. ***http://bit.ly/2v5RYvU***

Listing 6. Kod funkcji wykonującej obrót obiektu

```
function animate()
{
    requestAnimationFrame (animate);
    mesh.rotation.x = THREE.Math.degToRad(x);
    mesh.rotation.y = THREE.Math.degToRad(y);
    mesh.rotation.z = THREE.Math.degToRad(z);
    renderer.render (scene, camera);
}
```

Listing 7. Pełny kod źródłowy strony index.html po dodaniu elementów grafiki 3D

```
<!DOCTYPE html>
<html>
  <head>
    <canvas id="mycanvas" width="500" height="500"></canvas>
    <style>
      table, th, td {
        border: 1px solid black;
      }
      th, td {
        border: 1px solid black;
        padding: 15px;
      }
    </style>
    <script src='/socket.io/socket.io.js'></script>
    <script src='three.min.js'></script>
    <script>
      var camera, scene, renderer;
      var geometry, material, mesh;
      var x, y, z;
      function init() {
        scene = new THREE.Scene();
        camera = new THREE.PerspectiveCamera (70, 500/500, 0.01,
10);
        camera.position.z = 0.5;
        geometry = new THREE.BoxGeometry (0.2, 0.2, 0.2);
        material = new THREE.MeshNormalMaterial();
        mesh = new THREE.Mesh (geometry, material);
        scene.add (mesh);

        renderer = new THREE.WebGLRenderer ({ canvas: mycanvas});
        renderer.setSize (500, 500);
        document.body.appendChild (renderer.domElement);
      }

      function animate() {
        requestAnimationFrame (animate);
        mesh.rotation.x = THREE.Math.degToRad(x);
        mesh.rotation.y = THREE.Math.degToRad(y);
        mesh.rotation.z = THREE.Math.degToRad(z);
        renderer.render (scene, camera);
      }

      init();
      animate();
      var socket = io();
      socket.on ('xyz', function (data) {
        var arr = data.message.split(" ");
        x = arr[0];
        y = arr[1];
        z = arr[2];
        document.getElementById("x_val").innerHTML = x;
        document.getElementById("y_val").innerHTML = y;
        document.getElementById("z_val").innerHTML = z;
      });
    </script>
  </head>
  <body>
    <h1>Gyroscope I2C</h1>
    <table>
      <tr>
        <th>X [deg]</th>
        <td><p id="x_val">---</p></td>
      </tr>
      <tr>
        <th>Y [deg]</th>
        <td><p id="y_val">---</p></td>
      </tr>
      <tr>
        <th>Z [deg]</th>
        <td><p id="z_val">---</p></td>
      </tr>
    </table>
  </body>
</html>
```

Listing 8. Zmodyfikowany kod obsługi zapytania – skrypt main.js

```
var url = require('url');
var server = http.createServer (function handler (request,
response)
{
    var pathname = url.parse(request.url).pathname;
    console.log("Request for " + pathname + " received.");
    response.writeHead (200, {'Content-Type': 'text/html'});
    if(pathname == "/")
    {
        var index = fs.readFileSync (__dirname + '/index.html');
        response.write (index);
    } else if (pathname == "/three.min.js")
    {
        var script = fs.readFileSync (__dirname + '/three.min.
js');
        response.write (script);
    }
    response.end();
});
```