Seminarska naloga - 3D miška

Adam Prestor, Žiga Simončič

12. maj 2019

1 Uvod

Cilj naloge je implementacija 3D miške z uporabo ploščice, ki jo uporabljamo na vajah. Klasična miška deluje na dveh oseh (x in y), 3D miška pa naj bi pokrila še tretjo os (z), torej višino. Za implementacijo sva uporabila dva senzorja: pospeškometer in žiroskop. Teoretično bi zadostoval le pospeškometer, vendar je zaradi orientacije oz. lege miške potrebna tudi informacija o rotaciji, ki jo sporoča žiroskop.

Težav pri implementaciji je bilo kar nekaj, tako teoretičnih, kot tudi čisto praktičnih.

Največja težava, ki je še vedno nekoliko prisotna, je nenatančnost meritev. Pri obeh senzorji, posebej pa pri pospeškometru, vrednosti v mirovanju niso konstantne, ampak vseskozi škačejo".

Podobna težava je pri samem izračunu dejanskih vrednosti, ki nas zanimajo: pri obeh senzorjih je vrednosti potrebno vsaj enkrat integrirati. Integral je zvezna operacija, ki je v praksi ne moremo natančno izračunati, ampak računamo približke. Skupaj z zgoraj omenjeno težavo dobimo izračune, ki so približno natančni, vendar ne popolnoma. Sčasoma, ob daljšem delovanju, se napake še povečujejo oz. seštevajo.

Za implementacijo sva uporabila FreeRTOS (programski jezik C), za končni izračun vrednosti in vizualizacijo pa grafični pogon Unity (programski jezik C#).

Podrobnosti implementacije so opisane v poglavjih 2 (FreeRTOS) in 3 (Unity). Problemi so opisani v poglavju 4, rezultati v poglavju 5 in možne nadgradnje v poglavju 6.

2 Implementacija v FreeRTOS (C)

2.1 Žiroskop

Žiroskop je senzor, ki za vse tri smeri (x, y in z) sporoča kotno hitrost vrtenja (enota $\frac{^\circ}{s}$). Dejansko rotacijo dobimo tako, da zmnožimo hitrost s pretečenim časom (razliko v času). To ponavljamo in seštevamo.

2.1.1 Registri

Žiroskop lahko pred uporabo nastavimo. To storimo z registri *GYRO_FS_SEL* in *OFFS_USR*.

Z *GYRO_FS_SEL* izberemo eno izmed štirih stopenj občutljivosti in obsega vrednosti (stopnje od 0 do 3). Pri višji stopnji dobimo večji možen obseg vrednosti (enota $\frac{^\circ}{s}$), vendar s

tem tudi večjo občutljivost na spremembe in posledično večjo nestabilnost. V končni verziji sva uporabila stopnjo 0, saj je pri taki konfiguraciji senzor deloval bolj stabilno in dajal boljše rezultate.

OFFS_USR je skupina registrov za vsako od treh osi in predstavljajo odmik žiroskopa. Uporabljajo se za kalibracijo odmika v mirovanju. Te vrednosti se avtomatsko odštejejo od vrednosti, ki jih preberemo s senzorja. Kalibracijo lahko izvedemo v mirovanju tako, da pogledamo vrednosti, ki jih vrača senzor, in jih nato vpišemo v ustrezne registre za odmik. Problem lahko rešimo tudi programsko, da odmike hranimo v spremenljivkah in jih nato v programu odštejemo od prebreanih vrednosti. Ugotovili smo, da je žiroskop na ploščici relativno stabilen in natančen, zato posebna kalibracija ni bila potrebna.

2.1.2 Branje in izračun vrednosti

Vrednosti za vsako os so zapisane v dveh 8-bitnih registrih: *GYRO_OUT_H* in *GYRO_OUT_L*. H predstavlja register z bolj pomembnimi (višjimi) biti, L pa register z nižjimi. Prebrati je potrebno oba registra in ustrezno sestaviti 16-bitno predznačeno število, ki predstavlja meritev (enota $\frac{^\circ}{s}$).

Prebrane meritve je potrebno pretvoriti v stopinje (°).

Formula iz dokumentacije[1] za pretvorbo surovih meritev v pravilne fizikalne vrednosti ($\frac{^\circ}{s}$) je

$$GYRO = Gyro\_Sensitivity * Angular\_rate, \tag{1}$$

kjer je *GYRO* prebrana meritev, *Gyro_Sensitivity* pa določa dokumentacija glede na izbrano stopnjo občutljivosti. Zanima nas torej *Angular_rate*, ki ga izrazimo. V našem primeru je stopnja občutljivosti *FS_SEL=0*, kar določa vrednost 131.

Končna formula za pretvorbo surove vrednosti meritve v vrednost z enoto $\frac{^\circ}{s}$ je

$$Angular\_rate = \frac{GYRO}{131}. \tag{2}$$

Iz zgornje formule dobimo še rotacijo (°) oz. spremembo rotacije v časovnem obdobju Δt

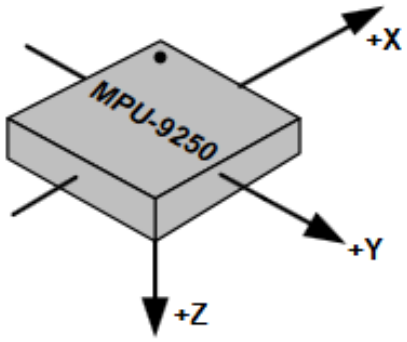
$$Angular\_rate = \frac{GYRO}{131} * \Delta t. \tag{3}$$

Vsako iteracijo zanke dobljene vrednosti za vsako os seštejemo. Seštevanje vrednosti ni implementirano na ploščici v FreeRTOS-u, ampak te vrednosti preko serijske povezave pošiljamo v grafični pogon Unity, kjer nadaljujemo z obdelavo in matričnimi operacijami.

2.2 Pospeškometer

Modul MPU-9250 ima vgrajen tri-smerni pospeškometer. Pospeške meri v smeri x, y in z kot je prikazano na sliki 1. Izmerjeni pospeški so zapisani s 16 bitnimi predznačenimi števili. Izmerjeni

[1] `https://www.invensense.com/wp-content/uploads/2015/02/MPU-9250-Register-Map.pdf`



Slika 1: Orientacija osi pospeškometra na čipu MPU-9250. Pika na sliki označuje pin 1.

pospeški so zapisani v šestih registrih, vsaka smer ima dva registra za hranjenje velikosti, visoki in nizki bit.

2.2.1 Registri

MPU-9250 ima tudi nekaj nastavitev za vgrajeni pospeškometer, kot so nastavitev natančnosti. Natančnost je nastavljiva v registru $ACCEL\_CONFIG$ z bitoma 3 in 4. Možne nastavitve natančnosti so 2g, 4g, 8g in 16g od najbolj do najmanj natančne. V našem primeru smo izbrali natančnost 2g, ker smo želeli imeti čim bolj natančne meritve. Testirali smo tudi ostale natančnosti, vendar nam manjša natančnost ni pomagala, da bi bile meritve manj šumne.

Obstajajo tudi druge možne nastavitve pospeškometra, v naši rešitvi jih nismo upoštevali.

2.2.2 Izračun vrednosti pospeška

Za pretvorbo prebrane vrednosti iz registrov v fizikalno enoto $\frac{m}{s^2}$ potrebujemo nastavljeno natančnost, v našem primeru 2g, označimo jo z $ACCEL_{FS}$ in konstanto, ki jo lahko razberemo iz dokumentacije, v tem primeru je to število 32768, označimo jo kot $CONST_G$. Glede na izbrano natančnost, moramo prebrano vrednost pospeškometra deliti s številom $\frac{ACCEL_{FS}}{CONST_G}$. Tako dobima vrednost pospeška v enoti gravitacijskega pospeška, da bi dobili $\frac{m}{s^2}$ moramo to vrednost pomnožiti še z vrednostjo gravitacijskega pospeška v $\frac{m}{s^2}$, ta pa je približno 9,80. Enačbo za pretvorbo lahko strnemo v enačbo

$$ACCEL = \frac{ACCEL_{FS} * GtoMS}{CONST_G}, \tag{4}$$

kjer je $GtoMS$ konstanta gravitacijskega pospeška v $\frac{m}{s^2}$, $ACCEL$ pa izračunan pospešek v $\frac{m}{s^2}$. V našem primeru bomo pozicijo izrazili v cm namesto v m, kar pomeni, da moramo dobljeno vrednost pomnožiti s 100.

2.2.3 Kalibracija pospeška

Zaradi konstantnega gravitacijskega pospeška smo imeli kar nekaj problemov z izračunavanjem pozicije ploščice. Ta problemu, recimo mu kar problem konstantnega pospeška, smo rešili s sprotno kalibracijo.

Predno se lotimo kalibracije, moramo omeniti še upragovanje pospeška (angl. *thresholding*). Pri opazovanju pospeška v mirovni legi smo opazili, da pospešek tudi v mirovni legi nekoliko niha. Radi bi, da je v mirovanju pospešek enak nič, to zagotovimo s upragovanjem. Ostane nam samo še določanje praga, tega smo določili empirično. V našem primeru smo ga nastavili na vrednost $0,5\frac{m}{s^2}$.

Kalibracije smo se lotili tako, da smo opazovali zgodovino meritev. Če ustrezno število zaporednih meritev ni preseglo praga, smo predvidevali, da je ploščica v stanju mirovanja. Prebrane vrednosti teh meritev smo potem povprečili in tako določili nov pospešek v mirovnem stanju. Poleg tega, ko ugotovimo da smo v mirovnem stanju, moramo pospešek ponastaviti na 0 v vse smeri in ponastaviti hitrost na 0. Ponastavljanje hitrosti je narejeno v grafičnem pogonu Unity. Nove pospeške izračunamo tako, da od prebrane vrednosti odštejemo pospešek v mirovnem stanju.

3 Implementacija v grafičnem pogonu Unity

Preko serijske povezave v grafičnem pogonu Unity prebiramo vrednosti, ki jih pošiljamo iz FreeRTOS-a. Pošiljamo in beremo: spremembo rotacije, pospešek in spremembo časa (Δt).

V splošnem si operacije v Unity-ju sledijo tako:

- Branje vrednosti,
- popravljanje vrednosti,
- izračun in
- prikaz / vizualizacija.

Za implementacijo smo uporabili Unity 2019.1.0f2.

3.1 Branje

Branje je implementirano v posebni niti, ki konstanto bere vrednosti preko serijske povezave in posodablja globalne spremenljivke. Glavna zanka Unity-ja omenjene globalne spremenljivke uporablja za ustrezen prikaz miške na zaslonu.

Glavno razlog take implementacije je hitrost, saj Unity glavna zanka hkrati bere in posodablja prepočasi - zato sva implementirala posebno nit za branje. Drugim nitim pa Unity ne dovoljuje spreminjanje Unity objektov - zato se Unity objekti še vedno posodabljajo v glavni zanki.

3.2 Popravljanje vrednosti

Vrednosti za pospeškometer in žiroskop pred nadaljnjo obdelavo nekoliko popraviva. Za obe vrste meritev hraniva zgodovino. Privzeto je zgodovina nastavljena na 20 vrednosti (za vsako vrsto meritve). Velikost zgodovine je poljubno nastavljiva.

Vsako novo prebrano vrednost dodava v zgodovino nato izračunava povprečje brez ekstremov. To pomeni, da ignorirava največjo in najmanjšo vrednost v zgodovini. Izračunano vrednost uporabiva za nadaljnjo obdelavo.

Ker vrednost v mirujočem stanju lahko tudi škačejo"(predvsem pri pospeškometru), imamo implementirano tudi mejo, koliko visoka mora vrednost biti, da jo upoštevava (angl. *threshold*). Za pospeškometer je to že implementirano v FreeRTOS-u (zaradi kalibracije), vrednosti za rotacijo pa merimo v Unity-ju.

3.3 Izračuni

3.3.1 Pozicija

Da bi iz pospeška dobili pozicijo, moramo uporabiti dvojni integral pospeška, namreč velja zveza

$$a(t) = v'(t) = x''(t). \tag{5}$$

Računanje določenega integrala s pomočjo numeričnih metod ni velik problem. V našem primeru bomo zaradi hitrosti operirali le z najnovejšimi vrednostimi, tako da bomo s prvim integralom dobili hitrost po enačbi

$$v(t + \Delta t) = v(t) + a(t + \Delta t) * \Delta t, \tag{6}$$

ter potem pozicijo s ponovnim integriranjem hitrosti,

$$x(t + \Delta t) = x(t) + v(t + \Delta t) * \Delta t, \tag{7}$$

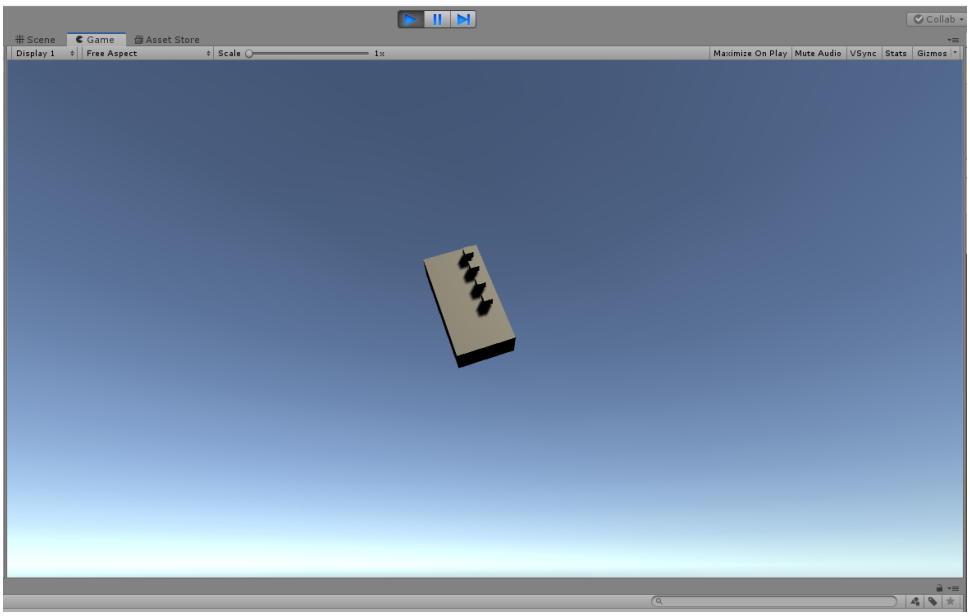
kjer je Δt čas med zaporednima meritvama.

Slednja metoda je dokaj nenatančna, če ni čas med dvema merjenjema prevelik. V našem primeru je povprečni čas med zaporednima meritvama približno 5-6 ms, kar je relativno malo, zato predpostavljamo, da napake v integraciji ne bi smele biti prevelike.

3.3.2 Rotacija

Izračun rotacije objekta je načeloma preprost - dobljene popravljene vrednosti samo seštevamo in dobimo končno rotacijo objekta. Do neke mere to drži, npr. če bi imeli samo 1 os. Ker pa imamo več osi, moramo upoštevati, da žiroskop zaznava kotno hitrost v globalnem smislu. Če žiroskop po neki osi obrnemo za 90°, potem se lokalno gledano rotacijske osi na žiroskopu spremenijo. Torej je potrebno upoštevati tudi lokalno orientacijo.

Unity nam za rešitev problema ponuja relativno enostavno rešitev - t.i. kvaternioni. Uporabljajo se za ponazoritev rotacije objekta in so sestavljeni iz štirih števil: x, y, z ter w. V



Slika 2: Prikaz vizualizacije miške v Unity-ju.

primerjavi z rotacijskimi matrikami so bolj stabilni in učinkoviti, izognejo pa se tudi problema *gimbal lock*[2], ki nastane pri uporabi Eulerjevih ("navadnih") kotov. V ozadju za izračun uporabljajo imaginarna števila.

Trenutno rotacijo imamo predstavljeno s kvarternionom, za izračun nove rotacije pa definirava delta kvaternion na način:

$$x = 0.5 * toRadians(vrednostX) \tag{8}$$

$$y = 0.5 * toRadians(vrednostY) \tag{9}$$

$$z = 0.5 * toRadians(vrednostZ) \tag{10}$$

$$w = 1 \tag{11}$$

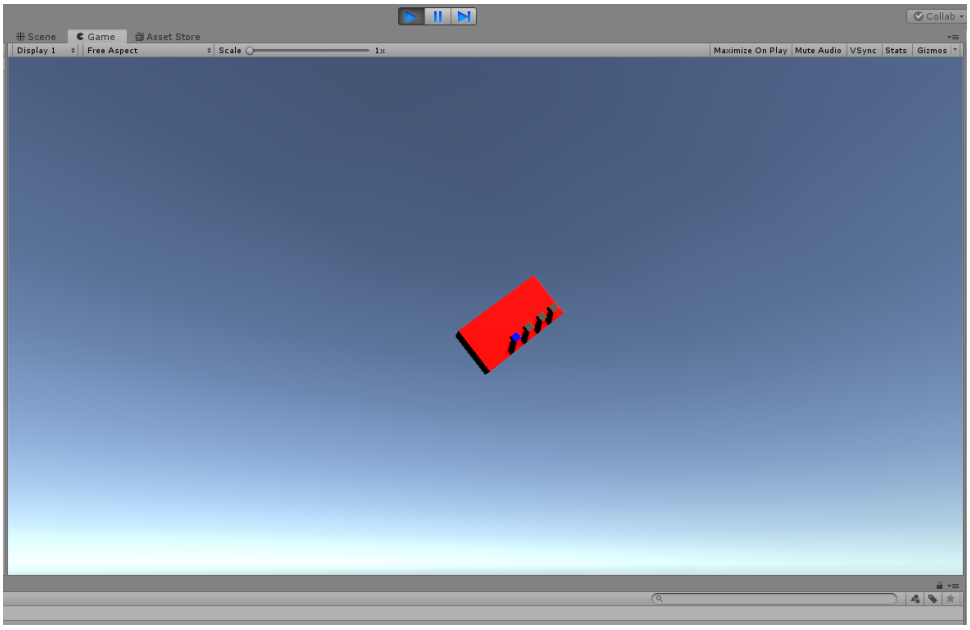
Da dobimo končno rotacijo, kvaternion začetne rotacija pomnožimo z zgoraj definiranim delta kvaternionom. Ta postopek upošteva lokalno orientacijo (ob rotaciji okoli različnih osi se vrednosti pravilno računajo) in se izogne problemu *gimbal lock*.

3.4 Prikaz

Izračunane vrednosti smo prikazali in vizualizirali z objektom, ki predstavlja miško (našo ploščico) v 3D prostoru. Vizualizacijo prikazuje slika 2.

Implementiran je tudi prikaz pritiska gumbov. Ob pritisku se gumb za del sekunde obarva modro. To je vidno na sliki 3, kjer je pritisnjen četrti gumb. V FreeRTOS-u drug task preverja stanje gumbov. Če je kateri pritisnjen, se na standardni izhod pošlje številko pritisnjenega

[2] `https://en.wikipedia.org/wiki/Gimbal_lock`



Slika 3: Pritisk gumba in zaklep miške.

gumba. Unity to prebere, ustrezno obarva gumb in naredi izpis. Dodatno gumb 1 (najbližji robu) povzroči ponastavitev pozicije, gumb 2 pa ponastavitev rotacije. Pri zaznavanju pritiskov lahko pride do zamika, ker uporabljava povpraševalni način (angl. *pooling*). Alternativa in možna nadgradnja je zaznavanje pritiskov s prekinitvami (angl. *interrupts*).

4 Problemi

4.1 Enakomerno gibanje

Kalibracija pospeška, omenjena v poglavju 2.2.3 nam omogoča, da zaznamo stanje, kjer se pospešek ne spreminja veliko, je le šum senzorja. Za tako stanje pravimo, da je ploščica v mirovni legi, kar pa ni nujno. Iz fizike vemo, da je pospešek enak nič takrat, ko telo miruje ali pa se giblje enakomerno, torej imamo pri naši kalibraciji problem z enakomernim gibanjem, saj ga smatra kot mirovanje.

Ta problem bi lahko rešili tudi tako, da bi upoštevali, da je mirovno stanje lahko tudi enakomerno gibanje. Pri tem bi upoštevali izračunano hitrost preden pridemo v mirovno stanje. Če je hitrost večja od nekega praga, namreč tudi v mirovanju hitrost ni točno 0, potem imamo enakomerno gibanje.

4.2 Gravitacijski pospešek

Pri zaznavanju pospeška velik problem predstavlja gravitacijski pospešek. V mirujočem stanju na ravni podlagi je pospešek na oseh x in y enak 0, na osi z pa je enak gravitacijskemu pospešku.

Ko ploščico rotiramo, se gravitacijski pospešek ustrezno porazdeli med ostale osi (po Pitagorovem izreku). To pomeni, da bodo vrednosti na pospeškometru ob naklonu "pokvarjene".

Problem smo poskušali rešiti tako, da izračunamo kolikšen delež gravitacijskega pospeška vpliva na posamezno os in poskusili vrednosti ustrezno kompenzirati. Pri računanju deleža gravitacijskega pospeška na vsaki osi sva iz enačb odstranila kvadrat, saj v nasprotnem primeru dobimo relativno dolgo kvadratno enačbo z več rešitvami. Dobljene deleže smo odšteli od prebranih vrednosti pospeškov.

Rezultati niso bili preveč dobri, predvidoma zaradi dveh stvari:

- Poenostavitve kvadratne enačbe, zaradi česar izračun ni bil povsem natančen in
- napake pri rotaciji. Ta napaka se nato prenese na izračun vpliva gravitacijske sile.

Zaradi teh dveh razlogov je bil ta poskus rešitve problema neuspešen.

Vseeno pa je ta problem bilo nujno rešiti, saj zaradi vpliva gravitacijske sile že ob malenkostnem odklonu vrednosti "odplavajo"in objekt v vizualizacije zdrsne nekam izven zaslona.

Na koncu smo ga rešili na bolj osnoven način: če je ploščica nagnjena v smer x ali y za več kot 5°, potem se izračun pozicije oseh x in y zaklene. Če je miška v zaklenjenem stanju, se v vizualizaciji obarva rdečo (glej sliko 3). Premik po osi z (višini) se zaklene pri naklonu 10°.

To pa privede do novega problema, ki je opisan v naslednjem poglavju.

4.3 Napaka pri rotaciji in zaklepanje premikanja

Omenili smo, da je izračun rotacije dokaj natančen (v primerjavi s pospeškometrom), vendar tudi pri rotaciji hitro pride do napak. Zaradi uvedbe zaklepanja, ki je opisan v prejšnjem poglavju, se hitro znajdemo v situaciji, ko pride do napake 5°ali več in se nam miška zaklene, čeprav jo imamo v resnici na ravni podlagi.

Problem smo rešila s popravkom rotacije, ko zaznamo, da je ploščica v mirujoči ravnovesni legi. Ko se nekaj ciklov zanke pospešek po osi z ne spreminja in je ta enak gravitacijski sili, smo lahko prepričani, da smo v ravnovesni mirujoči legi (po osi z). Takrat lahko rotacijo po oseh x in y nastavimo na 0°. S tem smo uvedli popravljanje rotacije glede na os z pospeškometra.

4.4 Vračanje miške

Vračanje miške je problem, kjer se miška ob koncu gibanja premakne nazaj v nasprotno smer četudi ploščica stoji pri miru. Najverjetnejša razlaga za ta pojav je upragovanju meritev. Upragovanje povzroči, da meritve niso najbolj točne, zvezne, to se še posebej pozna pri manjših pospeških. Zaradi slednjega namesto, da bi se hitrost postopoma znižala do 0, se hitrost preceni in nato prevesi v negativno, kar povzroči efekt vračanja miške.

Žal se temu pojavu ne moremo izogniti, saj nas upragovanje reši šuma v mirovanju, kar je bolj pomemben problem v našem pogledu.

4.5 Branje vrednosti senzorjev

Branje izvajamo v neskončni zanki. Vsako iteracijo preberemo vrednosti senzorjev, izvedemo ustrezne izračune in pošljemo po serijski povezavi. Preden preberemo vrednosti preverimo,

če so v registrih že zapisane nove vrednosti, z uporabo prekinitev modula MPU. Branje in izračun pretečenega časa mora biti atomarna operacija. To smo dosegli z ukazoma *taskENTER_CRITICAL()* in *taskEXIT_CRITICAL()*, ki poskrbita, da se blok kode izvede atomarno, torej da vmes na pride do menjave aktivnosti.

Za izračun pretečenega časa smo uporabili funkcijo *xTaskGetTickCount()*.

Zanka se je prvotno izvedla vsakih 6-7 milisekund. To smo želeli optimizirati oz. pohitriti, saj več meritev pomeni večja natančnost. Poskušala sva optimizirati branje s senzorjev, saj ostali deli kode niso imeli vidnega vpliva na hitrost zanke. Prvotno smo vsak register posebej klicali funkcijo *i2c_slave_read()*, kar pomeni, da je ob vsakem klicu zanke prišlo do 3 osi x 2 registra + 3 osi x 2 registra = 12 branj oz. klicev te funkcije.

V zadnji verziji pa imamo namesto 12 različnih branj samo 2 branji (vsak po 6 bajtov). S tem smo pohitrili izvajanje zanke iz 6-7 milisekund na 5-6 milisekund.

5 Rezultati

Končni rezultat miške lahko vidimo že na slikah 2 in 3.

Kot že omenjeno se rotacija računa kar natančno, po osi z je zelo natančna, po oseh x in y pa nekoliko manj. S ponastavitvijo rotacije v ravnovesni legi morebitne napake ne predstavljajo večjega problema.

Večji problem predstavlja premikanje. V smeri x in y deluje relativno v redu, občasno je problem vračanja miške, ki je opisan v podpoglavju 4.4. Redko se tudi zgodi, da se miška prestavi po osi z, čeprav jo premikamo samo po oseh x in y.

Premikanje po osi z (višini) je dokaj problematično. Miško je zelo težko premikati po višini, brez da bi jo nekam nagnili. Vsak manjši nagib pa v trenutni implementaciji že povzroči zaklepanje miške (alternativa je, da nekam "odplava").

Da bi ta izdelek postal uporabna 3D miška, bi bilo potrebno še veliko dodatnega dela in nastavljanje mejnih parametrov, velikost zgodovine itd.

Opisane pomanjkljivosti bi morda lahko odpravili z možnimi nadgradnjami, ki so opisane v naslednjem poglavju 6.

6 Možne nadgradnje

6.1 Uporaba vgrajenega FIFO registra

MPU-9250 ima vgrajen FIFO register, ki si shranjuje zadnje prebrane meritve senzorjev. FIFO register je prilagodljiv, lahko mu podamo naprave, ki nas zanimajo. V tem primeru bi posamezno vrednost dobili z obdelavo nekaj zaporednih meritev. Najhitrejša hitrost polnjenja FIFO registra je 1 vzorec na milisekundo, tako da bi za primerljivo hitrost lahko brali približno 5 vzorcev in jih potem obdelali v kosu, npr. dobili povprečje.

6.2 Način nizke porabe energije

MPU omogoča tudi način nizke porabe energije. V tem načinu se pospeškometer zbudi le ob dovolj veliki spremembi pospeška (angl. *wake on motion*), prag lahko določimo z vpisom v register. V tem primeru bi spremembo lahko pozicijo določali digitalno glede na smer premika v vse 3 smeri, se pravi ob vsakem ponovni vzbuditvi se pozicija premakne za 1 glede na izmerjene vrednosti.

6.3 Računanje z zamikom

Računanje vrednosti bi lahko delali tudi z zamikom. Namesto da upoštevamo le trenutno vrednost, si shranimo zadnjih nekaj, precej, prebranih vrednosti. Izračun nove vrednosti bi tako pretvorili na obdelavo signala. Signal je v tem primeru zadnjih nekaj prebranih vrednosti. Pozitivna stran tega pristopa je, da bi lahko brez težav izločili šum, slaba pa to, da bi bile meritve z zamikom, kar pripomore k slabši uporabniški izkušnji.