Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko

Damir Štuhec

ZAJEMANJE PODATKOV GIBANJA NA NAPRAVAH iOS IN NJIHOVA UPORABA PRI RAZVOJU IGER

Diplomsko delo

Maribor, september 2012

ZAJEMANJE PODATKOV GIBANJA NA NAPRAVAH iOS IN NJIHOVA UPORABA PRI RAZVOJU IGER

Diplomsko delo

Študent: Damir Štuhec

Študijski program: Univerzitetni študijski program

Računalništvo in informacijske tehnologije

Mentor: doc. dr. David Podgorelec

Lektor: Aleksandra Pal, prof. slovenščine in angleščine

Maribor, september 2012

Univerza v Mariboru

Fakulteta za elektrotehniko, računalništvo in informatiko

Številka: E1031601
Datum in kraj: 04. 04. 2012, Maribor

Na osnovi 330. člena Statuta Univerze v Mariboru (Ur. l. RS, št. 01/2010)
izdajam

SKLEP O DIPLOMSKEM DELU

1. **Damirju Štuhecu,** študentu univerzitetnega študijskega programa RAČUNALNIŠTVO IN INFORMACIJSKE TEHNOLOGIJE, se dovoljuje izdelati diplomsko delo pri predmetu Multimedia.

2. **MENTOR: doc. dr. David Podgorelec**
   SOMENTOR: doc. dr. Gregor Klajnšek

3. **Naslov diplomskega dela:**
   ZAJEMANJE PODATKOV GIBANJA NA NAPRAVAH iOS IN NJIHOVA UPORABA PRI RAZVOJU IGER

4. **Naslov diplomskega dela v angleškem jeziku:**
   OBTAINING MOTION DATA ON iOS DEVICES AND THEIR USE FOR GAME DEVELOPMENT

5. Diplomsko delo je potrebno izdelati skladno z "Navodili za izdelavo diplomskega dela" in ga oddati v treh izvodih (dva trdo vezana izvoda in en v spiralo vezan izvod) ter en izvod elektronske verzije do 30. 09. 2012 v referatu za študentske zadeve.

Pravni pouk: Zoper ta sklep je možna pritožba na senat članice v roku 3 delovnih dni.

Dekan:

Obvestiti:

- kandidata,
- mentorja,
- somentorja,
- odložiti v arhiv.

ZAHVALA

Zahvaljujem se mentorju doc. dr. Davidu Podgorelcu za strokovno pomoč in nasvete ter somentorju doc. dr. Gregorju Klajnšku, za kreativne ideje in podporo.

Posebno zahvalo namenjam celotni družini, predvsem staršem, ki so mi omogočili študij in naredili vsak dan vreden truda, ter Maši, ki predstavlja največji navdih.

ZAJEMANJE PODATKOV GIBANJA NA NAPRAVAH IOS IN NJIHOVA UPORABA PRI RAZVOJU IGER

Ključne besede: podatki gibanja naprave, sistemsko programsko ogrodje Core Motion, mobilni operacijski sistem iOS, razvojno okolje Xcode, razvoj mobilnih iger.

UDK: 004.41:621.395.721.5(043.2)

Povzetek

Podatki gibanja prinašajo brezmejne možnosti v svet razvijanja mobilnih iger ter močno povečujejo faktor realnosti interakcije med uporabnikom in napravo oziroma aplikacijo.

Diplomska naloga se osredotoča na raziskovanje zajemanja podatkov gibanja na napravah iOS ter njihovo uporabnost pri razvoju iger.

Zanima nas, na kakšen način strojna oprema zajema podatke ter kakšne so možnosti in oblike uporabe zajetih podatkov pri razvoju aplikacij.

Na koncu naloge raziskovanje podpremo s konkretnimi primeri oziroma prototipi 2D iger, ki nakazujejo zmožnosti in področja uporabe zajetih podatkov gibanja naprave.

OBTAINING MOTION DATA ON IOS DEVICES AND THEIR USE FOR GAME DEVELOPMENT

Key words: device motion data, system framework Core Motion, mobile operating system iOS, development environment Xcode, mobile game development

UDK: 004.41:621.395.721.5(043.2)

Abstract

Motion data bring endless possibilities into the world of mobile game development and considerably enhance the reality factor of interaction between user and application.

The thesis focuses on obtaining motion data on the iOS devices and their applicability in game development.

We are focusing on how the hardware captures motion data and what are the possibilities and uses of captured data in mobile application development.

The thesis is concluded with application prototypes that demonstrate motion data possibilities and their usage.

VSEBINA

1 Uvod **1**
1.1 Opredelitev problema, ki je predmet raziskovanja 2
1.2 Namen, cilji in teze raziskovalnega dela 2
2 Razvojno orodje Xcode in ciljne naprave iOS **3**
2.1 Razvojno orodje Xcode 3
2.2 Ciljne naprave iOS 5
3 Struktura sistema iOS **7**
3.1 Core OS 8
3.2 Core Services 9
3.2.1 Core Motion 9
3.3 Media 9
3.4 Cocoa Touch 11
3.4.1 Razred UIAccelerometer 12
4 Gibalni mehanizem **14**
4.1 Merilnik pospeška 15
4.2 Giroskop 18
5 Sistemsko programsko ogrodje Core Motion **21**
5.1 Kontrolni razred CMMotionManager 22
5.2 Merilnik pospeška: razred CMAccelerometerData 23
5.3 Giroskop: razred CMGyroData 25
5.4 Magnetometer: razred CMMagnetometerData 26
5.5 DeviceMotion: razred CMDeviceMotion 27
5.5.1 Prostorska usmerjenost - **attitude** 28
5.5.2 Natančna stopnja rotacije naprave - **rotationRate** 31
5.5.3 Gravitacijska sila delujoča na napravo - **gravity** 32
5.5.4 Pospešek, ki ga uporabnik povzroča na napravo - **userAcceleration** 33
5.6 Povzetek 34
6 Uporaba podatkov gibanja pri razvoju iger **36**

6.1 Prototip 1 - upravljanje letala v 2D svetu z uporabo podatkov giroskopa 36
6.2 Prototip 2 - metanje žoge v 2D prostoru z uporabo podatkov merilnika pospeška 39
6.3 Prototip 3 - rotacija 3D kocke s pomočjo podatkov giroskopa 46
7 Sklep 49
8 Viri in literatura 50

KAZALO SLIK

Slika 2.1 - Integrirano razvojno okolje 3
Slika 2.2 - Storyboard 4
Slika 3.1 - Sloji mobilnega operacijskega sistema iOS 8
Slika 4.1 - Strojna oprema naprave iPhone 4S..................... 14
Slika 4.2 - Merilnik pospeška v iOS napravi iPhone 4S (označen z rdečo barvo) 16
Slika 4.3 - Shema strukture in principa delovanja merilnika pospeška 17
Slika 4.4 - Merilnik pospeška naprave iPhone 4S v mikro merilu 17
Slika 4.5 - Giroskop v napravi iPhone 4S (označen z rdečo barvo) 18
Slika 4.6 - Giroskop naprave iPhone 4S v mikro merilu 19
Slika 4.7 - Primerjava definiranih premikov naprave merilnika pospeška in giroskopa 20
Slika 5.1 - Struktura sistemskega ogrodja Core Motion sistema iOS 5 22
Slika 5.2 - Matrike posameznih rotacij okrog osi trirazsežnega evklidskega prostora...... 30
Slika 6.1 - Grafični vmesnik 2D igre (prototip 1)..................... 37
Slika 6.2 - Grafični vmesnik 2D igre (prototip 2)..................... 40
Slika 6.3 - Grafični vmesnik (prototip 3)..................... 46

1 UVOD

Zaznavanje gibanja in zajemanje podatkov gibanja je v tehnološkem svetu prisotno bolj kot kadar koli prej v zgodovini. Največji napredek v zaznavanju gibanja zagotovo čutimo v mobilni in igralniški industriji, kjer sta tako konkurenca kot število ciljnih uporabnikov najvišji.

Zaznavanje gibanja je v mobilnih napravah postalo nekaj vsakdanjega. Proizvajalci mobilnih naprav se iz dneva v dan trudijo izpopolniti strojno in programsko opremo, ki bo še bolje opisala realno gibanje mobilne naprave. Hitrost pridobivanja ter natančnost podatkov sta pri zaznavanju gibanja naprave postala zelo pomembna dejavnika. V mobilnih aplikacijah, predvsem igrah, je zahteva po dovršenosti podatkov gibanja namreč vedno večja. Razvijalci mobilnih aplikacij vedno bolj težijo k posnemanju naravne interakcije, ki jo srečujemo v realnem svetu.

Diplomska naloga govori o zajemanju podatkov gibanja na napravah iOS, zato se bomo v drugem poglavju posvetili predvsem ciljnim napravam iOS, na katerih je zaznavanje gibanja možno ter na katerih je potekalo samo raziskovanje.

Na vsaki mobilni napravi teče operacijski sistem, ki izkorišča in upravlja strojno opremo naprave ter omogoča izvajanje aplikacij. V tretjem poglavju bomo opisali mobilni operacijski sistem iOS ter njegovo strukturo. Namen je prikazati sistemsko umeščenost programskih ogrodij, ki so potrebna za zajemanje podatkov gibanja.

Za zaznavanje gibanja potrebujemo tudi ustrezno strojno opremo. Glede na trend spreminjanja velikosti modernih mobilnih naprav, se je strojna oprema preoblikovala v tehnično dovršene, mikroelektromehanske sisteme, ki z izjemno natančnostjo merijo določene podatke. V četrtem poglavju se bomo posvetili podrobni predstavitvi »gibalnega mehanizma«, ki zajema senzor merilnika pospeška in senzor giroskopa.

Če želimo zajete podatke senzorjev uporabiti v aplikacijah, potrebujemo tudi ustrezna programska ogrodja (angl. Software Framework). Programska ogrodja za zajemanje podatkov gibanja so specifična ogrodja, ki v nižjem programskem nivoju skrbijo za

komunikacijo s strojno opremo, v višjem programskem nivoju pa nudijo zajete podatke v različnih oblikah. Sistemsko programsko ogrodje, ki omogoča delo s podatki gibanja naprave, se v mobilnem operacijskem sistemu iOS imenuje Core Motion. Peto poglavje podrobno predstavlja strukturo in osnovno uporabo razredov tega sistemskega programskega ogrodja.

Naslednje poglavje, poglavje številka 6, je namenjeno prikazu praktične uporabe podatkov gibanja pri razvoju iger. V tem poglavju so predstavljeni trije razviti prototipi, ki nazorno prikazujejo uporabo podatkov gibanja naprave za interakcijo uporabnika s aplikacijo.

1.1 Opredelitev problema, ki je predmet raziskovanja

Predmet raziskovanja je uporaba podatkov gibanja v aplikacijah na mobilnih napravah z operacijskim sistemom iOS (iPod touch, iPhone, iPad). Pridobivanje teh podatkov temelji na uporabi sistemskega programske ogrodja Core Motion (angl. Core Motion framework), ki pridobiva podatke iz senzorjev, kot sta merilnik pospeška (angl. accelerometer) in giroskop (angl. gyroscope). Podatke, ki so popolnoma neodvisni od aplikacije, v kateri jih želimo uporabljati, lahko dobimo v neobdelani in obdelani obliki. V diplomski nalogi se bomo posebej posvetili pridobivanju teh podatkov, določanju razlik med oblikami prejetih podatkov in načinom uporabe teh podatkov v zabavnih aplikacijah, kot so igre.

1.2 Namen, cilji in teze raziskovalnega dela

Namen raziskovalnega dela, oziroma diplomske naloge, sta opis in predstavitev zajemanja podatkov gibanja na napravah, ki uporabljajo operacijski sistem iOS, ter priprave teh podatkov za uporabo v praktičnih aplikacijah. Predstaviti in demonstrirati želimo metode učinkovite uporabe podatkov gibanja, ki jih lahko uporabimo pri razvoju iger. Za prikaz metode v praksi bomo izdelali nekaj prototipov aplikacij, ki se bodo izvajale na ciljnih napravah in uporabljale zajete podatke gibanja.

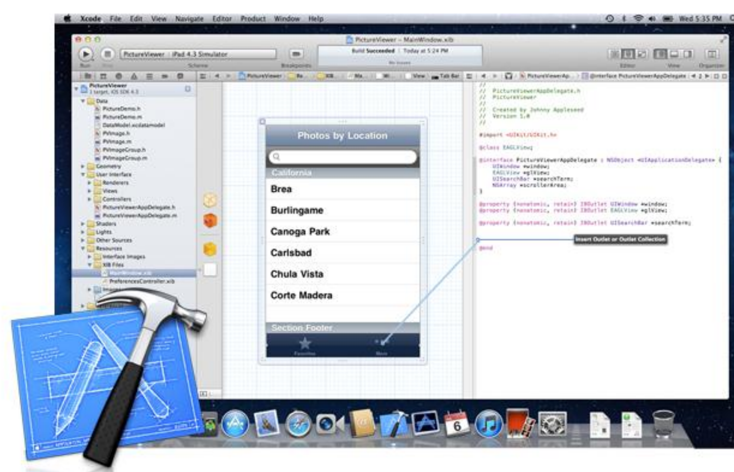
2 RAZVOJNO ORODJE Xcode IN CILJNE NAPRAVE iOS

V tem poglavju bomo predstavili razvojno orodje Xcode, ki smo ga uporabili kot osrednjo programsko opremo za raziskovanje ter razvoj prototipov aplikacij. Predstavili bomo tudi naprave z operacijskim sistemom iOS, ki omogočajo zajemanje podatkov gibanja.

2.1 Razvojno orodje Xcode

Xcode je produkt podjetja Apple Inc. Obsega več razvojnih orodij oziroma komponent, ki vključno z integriranim razvojnim okoljem omogočajo razvoj aplikacij za operacijska sistema Mac OS X in iOS, ki sta prav tako plod podjetja Apple Inc.

Centralni del delovnega prostora Xcode je prav gotovo integrirano razvojno okolje (angl. Integrated Development Environment), katerega grafični uporabniški vmesnik prikazuje **Slika 2.1**. Glavni komponenti integriranega razvojnega okolja sta zraven običajnega urejevalnika izvorne kode še orodje za gradnjo uporabniškega vmesnika (angl. Interface Builder) in napredni prevajalnik LLVM.



Slika 2.1 - Integrirano razvojno okolje, Apple Inc.,
https://devimages.apple.com.edgekey.net/technologies/images/tools_overview_xcode_20110711.jpg

S pomočjo orodja Interface Builder lahko zelo hitro in kakovostno sestavimo uporabniški vmesnik, tako za operacijski sistem Mac OS X, kot za mobilni operacijski sistem iOS. Glede na to, za katero ciljno napravo razvijamo, se nam grafični vmesnik posameznega okna (UIView) aplikacije prikaže v ustrezni velikosti oziroma ločljivosti. Na prikazan uporabniški vmesnik lahko nato zelo intuitivno dodajamo grafične gradnike oziroma elemente. Prav tako je tudi povezovanje teh grafičnih gradnikov s podatkovnim in kontrolnim delom aplikacije zelo poenostavljeno, kar zagotavlja lažje razumevanje in sledenje načinu razvoja interaktivnih aplikacij s pristopom MVC (angl. Model-View-Controller). S peto verzijo mobilnega operacijskega sistema iOS je bila k integriranemu razvojnemu okolju dodana izjemna funkcionalnost poimenovana Storyboard. Storyboard ne omogoča le hkratnega pregleda vseh oken aplikacije, ampak tudi intuitivno določanje prehodov med njimi ter spreminjanje lastnosti prikazanih oken. Velika prednost te funkcionalnosti je torej prikaz celotnega poteka izvajanja aplikacije. Funkcionalnost Storyboard prikazuje **Slika 2.2**.

Slika 2.2 - Storyboard

Pri razvojnem orodju Xcode je potrebno posebej omeniti tudi prevajalnik LLVM, ki je odprtokodni projekt (LLVM.org, licenca: UIUC BSD). To je prevajalnik nove generacije, ki je nadomestil dolgoletni prevajalnik GNU GCC. Prevajalnik LLVM je zgrajen iz več neodvisnih komponent, ki imajo vsaka svojo nalogo oziroma več nalog. Ena od pomembnejših komponent pri prevajalniku LLVM je komponenta Clang Front-End, ki po besedah osebja pri LLVM.org dosega do 3-krat hitrejše čase prevajanja kot prevajalnik GCC. [13]

2.2 Ciljne naprave iOS

Pri našem projektu smo razvijali prototipe aplikacij in pri tem raziskovali zajemanje podatkov gibanja izključno za t.i. naprave iOS. Naprave iOS so produkti podjetja Apple Inc., na katerih je nameščen mobilni operacijski sistem iOS. Operacijski sistem iOS je leta 2007, s prihodom prve generacije naprav iPod touch, nadomestil predhodni mobilni sistem iPhone OS. Danes mobilni operacijski sistem iOS uporabljajo naslednje naprave:

- iPhone 1. generacija,
- iPhone 3G,
- iPhone 3GS,
- iPhone 4,
- iPhone 4S,
- iPod Touch 1. generacija,
- iPod Touch 2. generacija,
- iPod Touch 3. generacija,
- iPod Touch 4. generacija,
- iPad (1. generacija),
- iPad 2,
- iPad (3. generacija),
- Apple TV 2. generacija,
- Apple TV 3. generacija.

Pri izvedbi projekta oziroma raziskave smo bili omejeni na naprave, ki vsebujejo strojno opremo za zajemanje podatkov gibanja ter hkrati na naprave, ki so nam bile dostopne.

Naprave, s katerimi smo izvedli raziskavo in testiranje razvoja aplikacije, so:

- iPhone 4S,
- iPod Touch 3. generacija,
- iPad 2,
- iPad (3. generacija).

Ciljne naprave za poganjanje razvite aplikacije:

- iPhone 3GS, 4, 4S,
- iPod Touch 3., 4. generacija,
- iPad 2,
- iPad (3.generacija).

Prototipi aplikacij so razviti za operacijski sistem iOS 5, kar pomeni, da morajo imeti ciljne naprave za uspešno izvajanje aplikacije nameščen sistem iOS 5 ali novejšo verzijo le-tega.

3 STRUKTURA SISTEMA iOS

Za zajemanje podatkov gibanja na napravi ne zadostuje samo ustrezna strojna oprema, ampak potrebujemo tudi programska ogrodja, ki te podatke nudijo v ustrezni obliki za uporabo v aplikacijah.

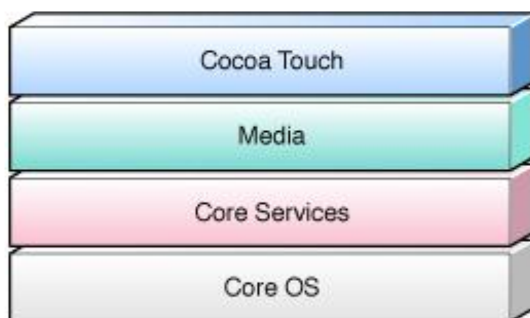
Za boljšo predstavo umeščenosti teh programskih ogrodij v mobilnem operacijskem sistem iOS bomo v naslednjem poglavju predstavili njegovo nivojsko strukturo.

Glavna naloga sistema iOS je, kot pri večini operacijskih sistemov, nadziranje strojne opreme ter zagotavljanje tehnologij, potrebnih za izvajanje in razvoj aplikacij. Sistem iOS je nadgradnja operacijskega sistema Mac OS X, ki poganja delovni postaji (angl. Workstation) Mac Pro in iMac, prenosne računalnike MacBook, MacBook Pro in MacBook Air ter strežnik Mac Mini. Strukturi omenjenih sistemov sta si zato zelo podobni.

Delovanje mobilnega operacijskega sistema iOS lahko opišemo zelo abstraktno, če povemo, da predstavlja neposredno povezavo med aplikacijo, ki se izvaja, in strojno opremo naprave. V realnosti je seveda povezava med aplikacijo in strojno opremo natančno definirana z množico sistemskih vmesnikov (angl. System Interface). Nastala abstrakcija med aplikacijo in strojno opremo naprave omogoča izvajanje istih aplikacij na različnih napravah iOS. Primer: aplikacijo razvito za iOS napravo iPhone, bi lahko brez težav izvajali na iOS napravi iPad, čeprav se razlikujeta v strojni opremi, kot je: procesor, sistemski pomnilnik, baterija, mrežna kartica, senzorji in ločljivost zaslona. Seveda bi morali za najboljšo uporabniško izkušnjo aplikacijo dodelati, toda v osnovi bi se izvajala brez težav.

Struktura sistema iOS je v praksi zgrajena iz večih programskih slojev. Na dnu strukture se nahaja sloj *Core OS*, ki zagotavlja osnovne nizkonivojske sistemske funkcionalnosti in tehnologije za uspešno komuniciranje višjih slojev s strojno opremo. Nad *Core OS* se nahaja sloj *Core Services*, ki zagotavlja večino ključnih sistemskih funkcionalnosti. Nad Core Services se nahaja sloj *Media*, ki zagotavlja vse tehnologije povezane z grafiko ter

avdio in video vsebinami. Najvišje v strukturi sistema iOS leži sloj *Cocoa Touch*. Je sloj, ki ga razvijalci aplikacij za sistem iOS izkoriščajo v največji meri. Zagotavlja namreč visokonivojske funkcionalnosti, ki omogočajo izkoriščanje strojne opreme naprav iOS. [5] V nadaljevanju so omenjeni sloji strukture sistema iOS opisani podrobneje, njihovo shemo pa prikazuje **Slika 3.1**.



Slika 3.1 - Sloji mobilnega operacijskega sistema iOS, Apple Inc., http://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Art/SystemLayers.jpg

3.1 Core OS

Core OS je nizkonivojski sistemski sloj, ki zagotavlja programska ogrodja, na katerih temelji večina funkcionalnosti v višje nivojskih sistemskih slojih. Običajno aplikacije ne dostopajo neposredno do storitev v tem sloju, skoraj zagotovo pa jih uporabljajo posredno, z uporabo storitev v visokonivojskih sistemskih slojih.

Core OS je edini sistemski sloj, ki je zmožen s pomočjo vključenih programskih ogrodij direktno komunicirati s strojno opremo naprave in zagotavljati zaščito na nizkem nivoju. Nekatera od teh programskih ogrodij so:

- *Core Bluetooth* (CoreBluetooth.framework),
- *Security* (Security.framework),
- *External Accessory* (ExternalAccessory.framework),
- *System* (System level). [6]

3.2 Core Services

Core Services je eden od najpomembnejših členov v strukturi sistema iOS. Vsebuje temeljna programska ogrodja, ki zagotavljajo sistemske storitve, brez katerih aplikacije ne bi delovale, oziroma ne bi omogočale večine danes popularnih funkcionalnosti. Core Services je sorazmeroma nizkonivojski sistemski sloj v sistemu iOS, zato veliko storitev oziroma tehnologij v visokonivojskih slojih temelji na le-teh v Core Services. Nekatere od glavnih programskih ogrodij v Core Services so:

- *Core Foundation* (CoreFoundation.framework),
- *Core Location* (CoreLocation.framework),
- *Core Data* (CoreData.framework),
- *Core Media* (CoreMedia.framework),
- *Core Telephony* (CoreTelephony.framework),
- *CFNetwork* (CFNetwork.framework),
- *Address Book* (AddressBook.framework),
- ***Core Motion*** (CoreMotion.framework). [7]

3.2.1 Core Motion

V sloju Core Services se nahaja tudi ogrodje, imenovano *Core Motion*, ki je za našo raziskavo najpomembnejše. *Core Motion* je namreč sistemsko programsko ogrodje, ki nam omogoča zajemanje in uporabo podatkov različnih senzorjev, ki se nahajajo v napravah iOS. *Core Motion* bomo podrobno predstavili v poglavju 5.

3.3 Media

Kot že samo ime pove, je ta sloj v strukturi sistema iOS zadolžen za zagotavljanje tehnologij, potrebnih za predstavitev multimedijskih vsebin, kot so grafika, avdio in video.

Tehnologije grafike

Pojem grafika lahko razumemo kot predstavitev grafičnih elementov na določenem zaslonu, ki jo opravljajo oziroma omogočajo tehnologije za grafično upodabljanje (angl. Rendering). Seveda se tehnologije upodabljanja za različne vrste predstavitev grafik razlikujejo, zato je na voljo več namenskih programskih ogrodij, ki to omogočajo. Ta so:

- *Core Graphics,*
- *Core Animation,*[3]
- *Core Image,*
- *OpenGL ES,*
- *Core Text.*

Avdio tehnologije

Omogočajo implementacijo funkcionalnosti predvajanja visokokakovostnega zvoka, zajemanja zvoka ter proženje vibracij, z namenom predstavitve bogate avdio vsebine na napravi iOS. Programska ogrodja, ki te funkcionalnosti omogočajo, so:

- *Media Player,*
- *AV Foundation,*
- *OpenAL,*
- *Core Audio.*

Podprti avdio formati so: AAC, Apple Lossless (alac), A-law, IMA/ADPCM (ima4), Linear PCM, µ-law, DVI/Intel IMA ADPCM, Microsoft GSM 6.10 in AES3-2003.

Video tehnologije

Danes si naprav brez zmožnosti predstavitve video vsebin skoraj ne znamo več predstavljati. Za predstavitev in zajemanje video vsebin so potrebne ustrezne tehnologije, ki omogočajo izkoriščanje vgrajene strojne opreme v napravi.

Najpogosteje uporabljene tehnologije oziroma programska ogrodja, ki jih razvijalec uporablja pri delu z video vsebinami na sistemu iOS so:

- *UIImagePickerController,*
- *Media Player,*
- *AV Foundation,*
- *Core Media.* [8]

3.4 Cocoa Touch

Cocoa Touch je »najvišji« sloj v strukturi sistema iOS. Funkcionalnosti tega sloja so visokonivojske, v večini grajene na osnovi vmesnikov iz nižje ležečih slojev, ki definirajo osnovno aplikacijsko infrastrukturo ter nudijo podporo za številne visokonivojske sistemske storitve. Razvijalec aplikacij se s tem slojem srečuje najpogosteje, saj zajema vse potrebne funkcionalnosti, ki jih potrebujemo v večini aplikacij.

Najpomembnejša programska ogrodja sloja Cocoa Touch so:

- *Message UI* (MessageUI.framework),
- *Event Kit UI* (EventKitUI.framework),
- *Map Kit* (MapKit.framework),
- *Game Kit* (GameKit.framework),
- *UIKit* (UIKit.framework): najpomembnejše ogrodje, ki zajema in omogoča implementacijo večine najpomembnejših funkcionalnosti modernih aplikacij. Te funkcionalnosti so:
    + upravljanje aplikacije,
    + upravljanje uporabniškega vmesnika,
    + animacija uporabniškega vmesnika,
    + delo z grafiko in »zasloni«,
    + večopravilnost,
    + prilagajanje kontrolnih elementov uporabniškega vmesnika,
    + zajemanje dogodkov zaslonskih dotikov oziroma potez,
    + delo s tekstovno in spletno vsebino,
    + integracija z drugimi aplikacijami na sistemu,

- sistemsko obveščanje.

Zraven naštetih funkcionalnosti ogrodje UIKit podpira tudi dodatne funkcionalnosti, ki so specifične za posamezne naprave iOS. Te funkcionalnosti so:

- delo z vgrajeno digitalno kamero,
- podatki o napravi,
- stanje baterije,
- podatki senzorja bližine,
- **podatki senzorja merilnika pospeška** (angl. Accelerometer). [9]

3.4.1 Razred UIAccelerometer

Pred pojavitvijo sistemskega programskega ogrodja Core Motion, katerega namen je zajeti podatke različnih senzorjev gibanja v enotno programsko ogrodje, je operacijski sistem iOS nudil informacije o gibanju naprave preko enega samega senzorja. Ta senzor je merilnik pospeška, ki je bil do pojavitve operacijskega sistema iOS 4 in naprave iPhone 4 edini senzor gibanja v napravah iOS.

Podatke merilnika pospeška je lahko[1] razvijalec uporabljal preko razreda *UIAccelerometer*, ki se v sistemski strukturi nahaja v prej opisanem sloju Cocoa Touch.

Spodnji kodni blok prikazuje delegacijo protokola *UIAccelerometerDelegate* v poljubnem ustvarjenem razredu, s čimer prevajalniku povemo, da želimo prejemati podatke merilnika pospeška.

```
@interface MojRazred : UIViewController <UIAccelerometerDelegate>
...
@end
```

V razredu »MojRazred« lahko sedaj definiramo metodo *accelerometer:didAccelerate:*, ki je metoda, deklarirana v protokolu *UIAccelerometerDelegate*. V naslednjem kodnem bloku

[1] Možnost uporabe razreda *UIAccelerometer* opisujemo v pretekliku, zaradi priporočitve uporabe razreda CMAccelerometerData, ki se nahaja v ogrodju Core Motion, s katerim do podatkov senzorja dostopamo hitreje in lažje, hkrati pa dobimo dostop do podatkov drugih senzorjev.

je prikazana definicija omenjene metode ter osnovna uporaba prejetih podatkov merilnika pospeška.

```
- (void)accelerometer:(UIAccelerometer *)accelerometer
        didAccelerate:(UIAcceleration *)acceleration
{
        double pospesekVzdolzX = acceleration.x;
        double pospesekVzdolzY = acceleration.y;
        double pospesekVzdolzY = acceleration.z;

        igralec.position.x = pospesekVzdolzX * faktor;
        igralec.position.y = pospesekVzdolzY * faktor;

        ...

}
```

Razred *UIAccelerometer* je s prihodom mobilnega operacijskega sistema iOS 5 postal odsvetovana oziroma opuščena funkcionalnost (angl. Deprecated), zato ga v nadaljevanju diplomskega dela ne bomo posebej obravnavali ali uporabljali.

4 GIBALNI MEHANIZEM

V tem poglavju bomo podrobno predstavili strojno opremo, ki je sposobna zaznavanja in merjenja dogodkov oziroma stanj, kot so: pospešek, težnost, rotacija in usmerjenost. Osredotočili se bomo predvsem na merilnik pospeška (angl. Accelerometer) in giroskop (angl. Gyroscope) naprave iPhone 4S, ki je najnovejša od naprav v družini iPhone. **Slika 4.1** prikazuje razstavljeno napravo iPhone 4S z vso strojno opremo.

Slika 4.1 - Strojna oprema naprave iPhone 4S, iFixit,
http://guide-images.ifixit.net/igi/fCQZXu4THRiQtWdv.huge

Iz dneva v dan se mobilne naprave »tanjšajo«, njihova zmogljivost pa se veča. To zahteva vedno bolj zmogljive in hkrati vedno manjše komponente, ki čim učinkoviteje zapolnijo omejen prostor, ki ga dovoljuje naprava. To zahtevo industrija velikokrat izpolni z uporabo t.i. tehnologije MEMS. MEMS, oziroma mikroelektromehanski sistemi, so sistemi, ki združujejo mehanske elemente, senzorje, aktuatorje in elektroniko. Pri MEMS je najpomembnejša velikost, ki je v povprečju okrog 50 mikrometrov.

Tudi merilnik pospeška in giroskop, ki sta predstavljena v nadaljevanju, spadata v skupino naprav MEMS.

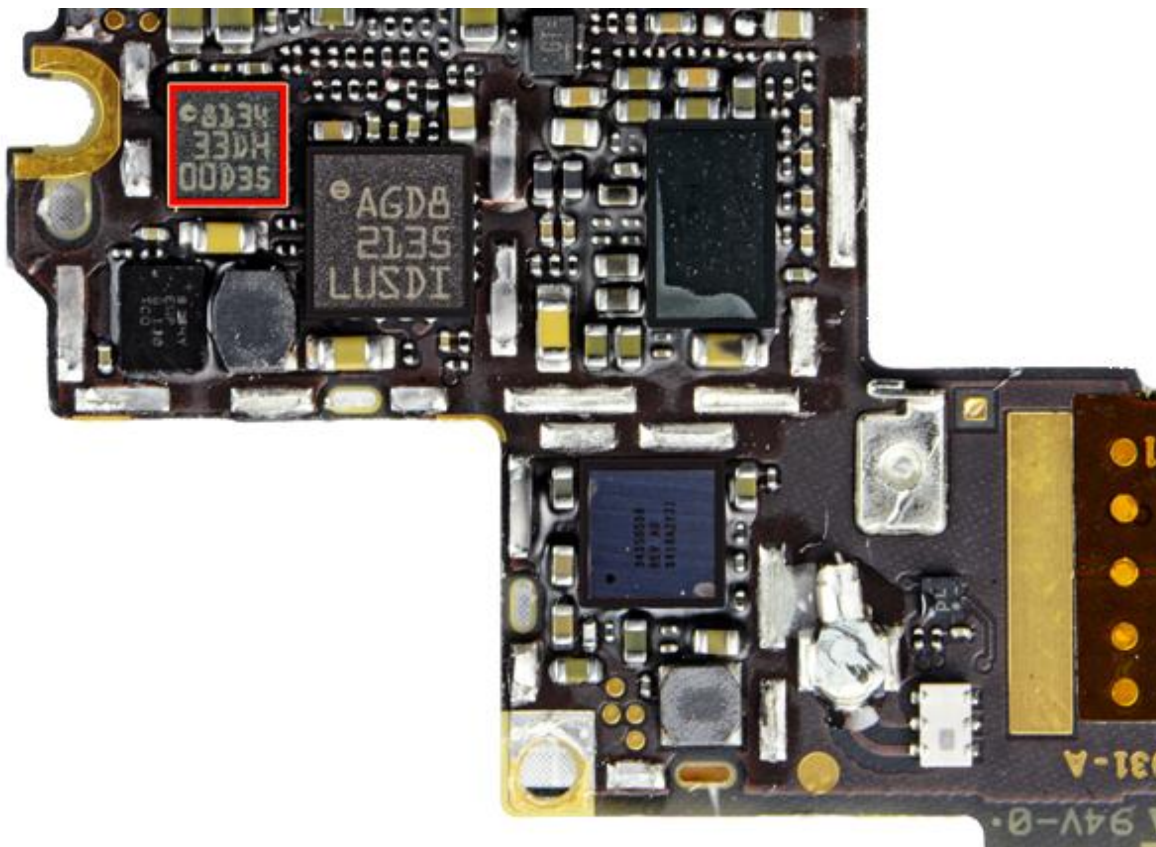
4.1 Merilnik pospeška

Iz stališča naprav iOS lahko merilnik pospeška razumemo kot napravo za zaznavanje jakosti in smeri sile oziroma t.i. dejanskega pospeška (angl. Proper acceleration), ki ga lahko predstavi v obliki vektorja v evklidskem prostoru. [16]

V relativni teoriji je dejanski pospešek fizična pospešitev danega objekta. Lahko ga zaznamo kot odvisnost med pospeškom nekega objekta in prostim padom, ali pa kot odvisnost med objektom v mirovanju in merjenim objektom v gibanju.

Merilci pospeška so pogosto uporabljeni v avtomobilski industriji, računalništvu, mobilnih napravah, navigacijskih sistemih, športu itd. Seveda se merilniki pospeška od področja do področja uporabe razlikujejo v sami strukturi in namenu, zato je tudi merilnik pospeška v napravah iOS nekaj posebnega.

Glavna naloga merilnika pospeška v napravah iOS je zaznavanje lege oziroma usmerjenosti naprave. Tako lahko merilnik pospeška zazna in natančno izmeri linearni premik naprave v vseh štirih smereh evklidskega prostora, dobljene podatke pa preko podatkovnega izhoda nudi drugim komponentam v napravi. Umeščenost merilnika pospeška v logično ploščo (angl. Logic board) naprave iPhone 4S prikazuje **Slika 4.2**. [10]

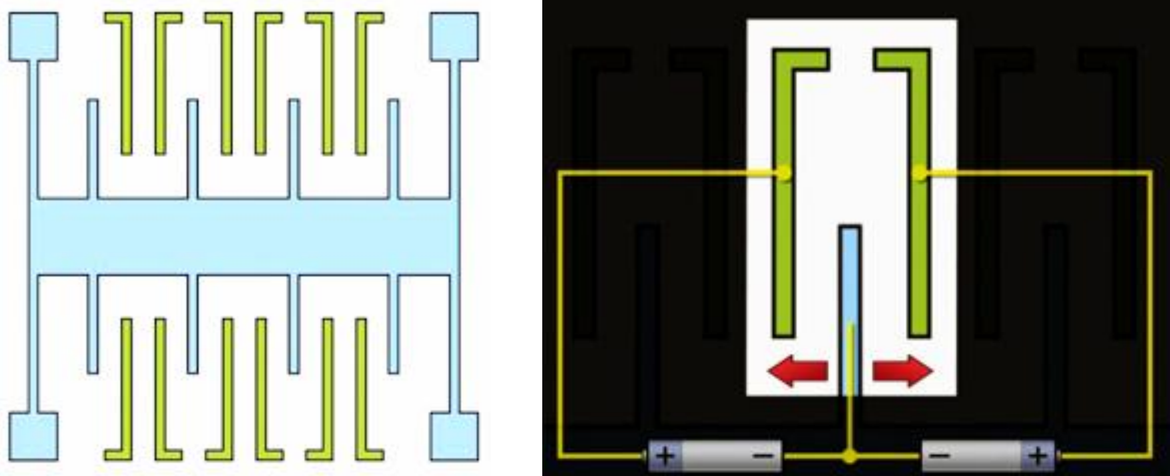


Slika 4.2 - Merilnik pospeška v iOS napravi iPhone 4S (označen z rdečo barvo), iFixit

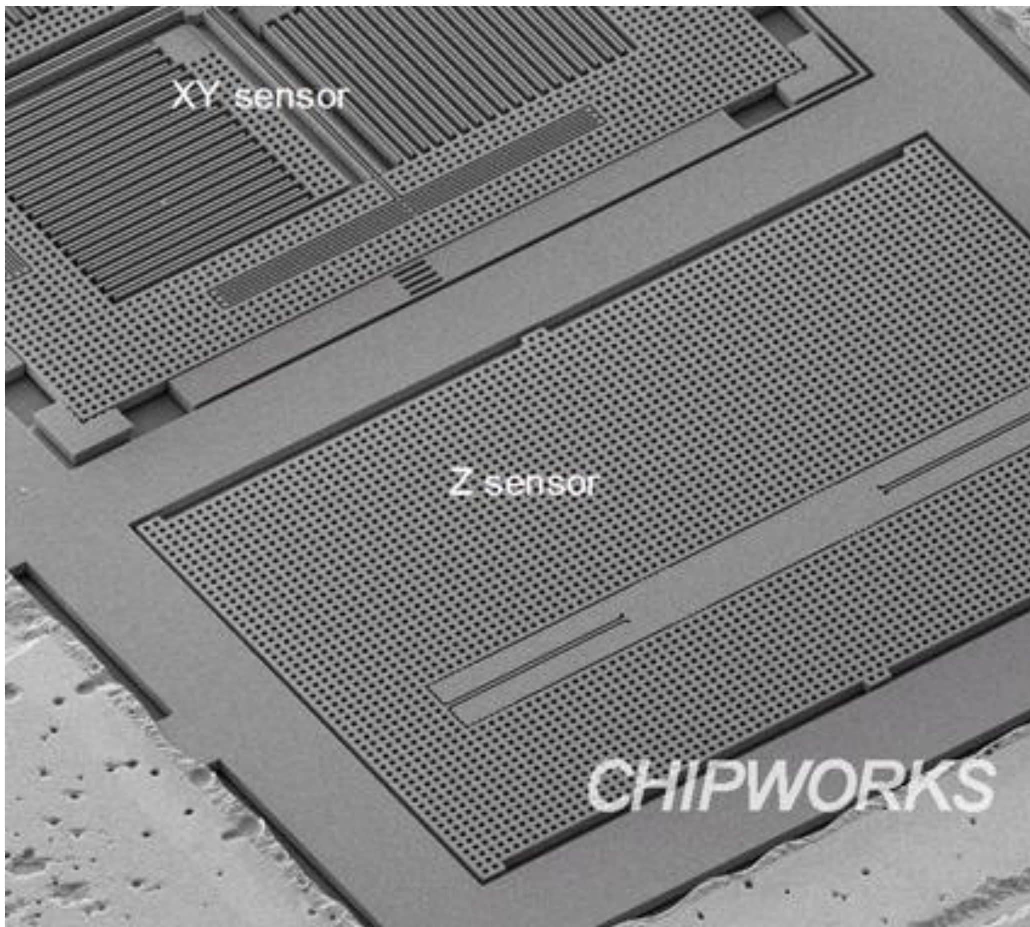
http://guide-images.ifixit.net/igi/vO4Wn4yDDvYiSWUE.huge

Temeljni material, iz katerega je izdelan merilnik pospeška v tehnologiji MEMS, je silicij. Silicij je material, ki ga je mogoče z uporabo določenih kemičnih snovi obdelovati z izjemno natančnostjo. To inženirjem omogoča izdelavo mikroelektromehanskih komponent, ki imajo neverjetno zapleteno strukturo.

Merilnik pospeška je v večini sodobnih »pametnih« mobilnih napravah izdelan po principu zaznavanja sprememb v kapacitivnosti. Gre za princip številnih kondenzatorskih mikro paličic ter gibljive kondenzatorske sredice, ki se ob premiku naprave premakne in dotakne določene paličice, kar povzroči pretok električnega toku. Na podlagi tega je nato izmerjena jakost in smer pospeška naprave. **Slika 4.3** ponazarja shemo strukture in delovanja merilnika pospeška, **Slika 4.4** pa prikazuje dejanski merilnik pospeška v mikro merilu.



Slika 4.3 - Shema strukture in principa delovanja merilnika pospeška, [11]



Slika 4.4 - Merilnik pospeška naprave iPhone 4S v mikro merilu, Chipworks

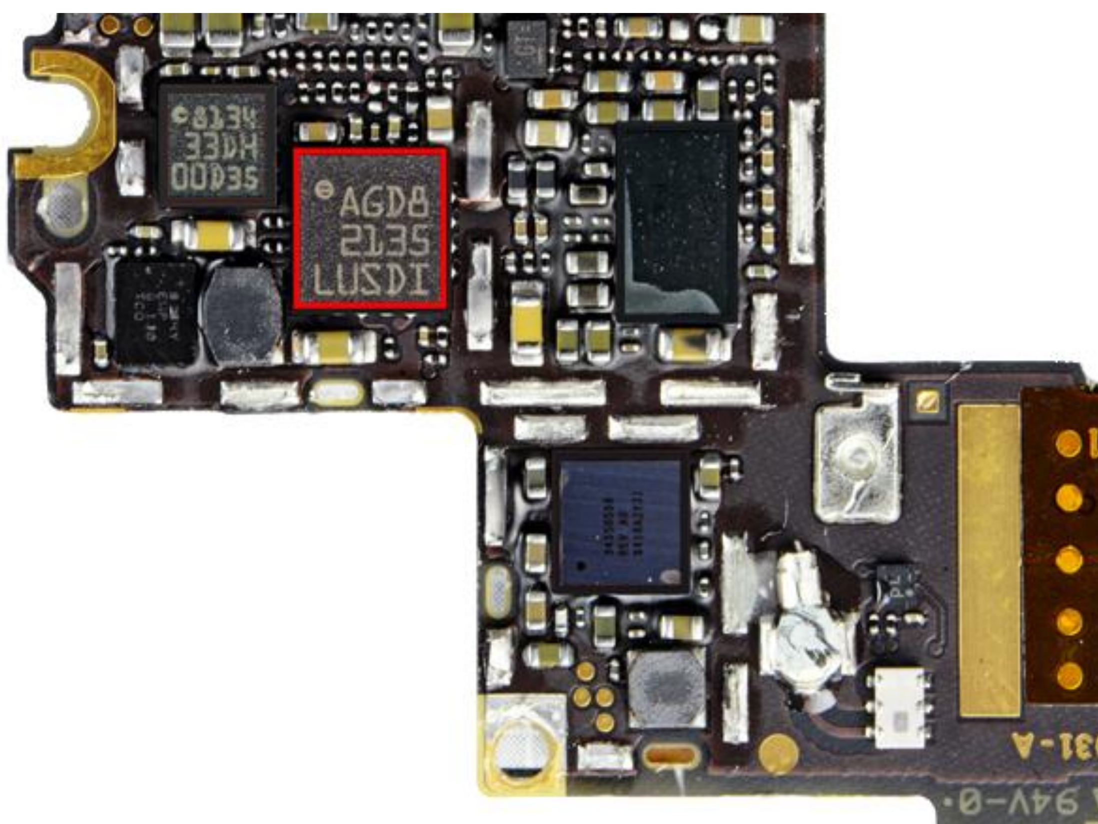
http://www.chipworks.com/uploadedimages/Technical_Competitive_Analysis/Teardowns/iPhone4/accelerometer-ST-b.jpg

4.2 Giroskop

»Giroskop je naprava, ki ponazarja in izrablja načelo ohranitve vrtilne količine v fiziki.« [17] Je torej naprava, ki zaznava jakost in smer spremembe orientacije objekta. V sodobne mobilne naprave je praktično nemogoče vgraditi mehanske giroskope, ki temeljijo na simetrični vrtavki, obešeni v kardanski sklop, zato so tudi giroskopi izdelani v tehnologiji MEMS.

Giroskop je pogosto uporabljen v avtomobilih, letalih, ladjah, kamerah, igralnih palicah in krmilnikih ter zadnje čase predvsem v mobilnih napravah.

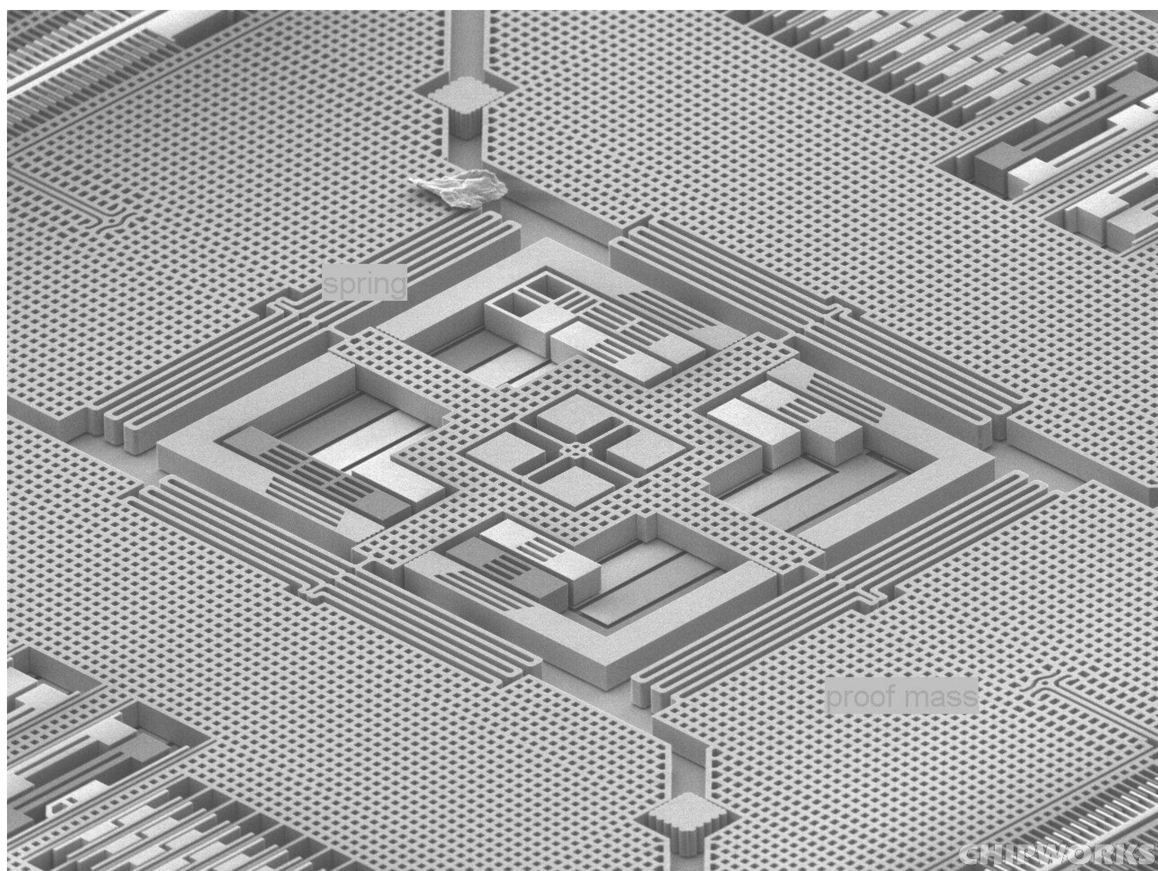
Naprave iOS so proizvajalci začeli opremljati z giroskopom leta 2010, ko je podjetje Apple Inc. predstavilo napravo iPhone 4. Umeščenost giroskopa v logično ploščo naprave iPhone 4S prikazuje **Slika 4.5**. [12]



Slika 4.5 - Giroskop v napravi iPhone 4S (označen z rdečo barvo), iFixit

http://guide-images.ifixit.net/igi/vO4Wn4yDDvYiSWUE.huge

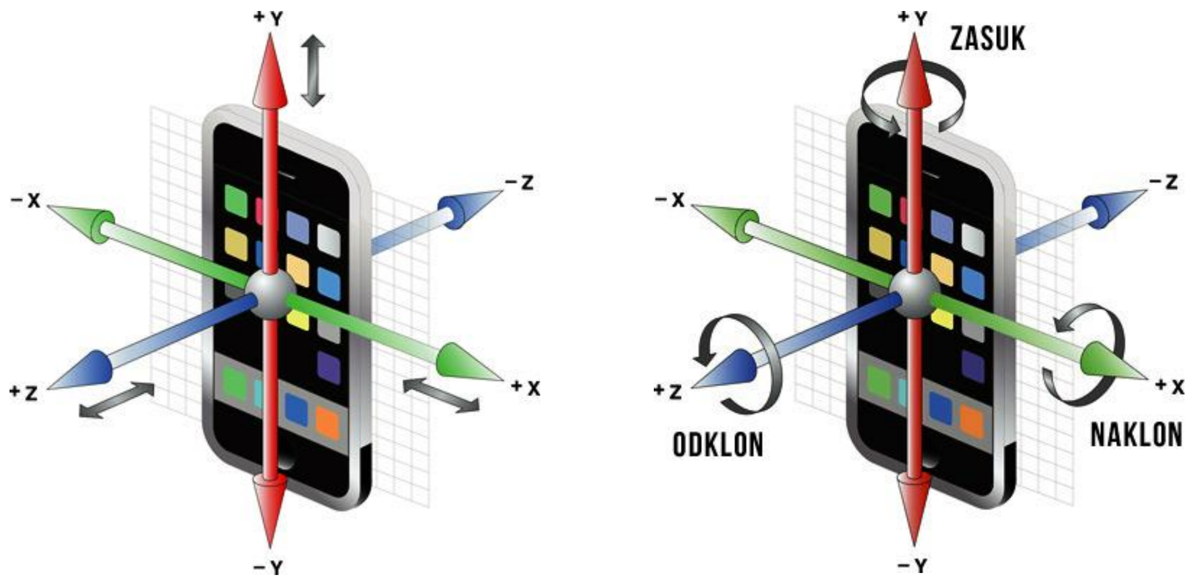
Tip giroskopa v napravah iOS je vibracijsko strukturiran giroskop (angl. Vibrating structure gyroscope), ki deluje na principu vibrirajoče silicijeve ploščice. Premikanje naprave povzroči vpliv nastalega Coriolis-ovega pospeška oziroma sile na vibrirajočo silicijevo ploščico, ki omogoča vztrajnost oziroma ohranjanje prvotne ravnine vibrirajoče ploščice. Zaradi nastale razlike v legah med napravo in vibrirajočo ploščico lahko giroskop izmeri kot med ravnino vibracijske ploščice in ravnino X, Y ali Z evklidskega prostora. **Slika 4.6** prikazuje dejanski giroskop naprave iPhone 4S v mikro merilu.



Slika 4.6 - Giroskop naprave iPhone 4S v mikro merilu, Chipworks

http://www.chipworks.com/media/wpmu/uploads/blogs.dir/4/files/2011/10/Untitled-4.jpg

Pojem »premikanje naprave« lahko pri tridimenzionalnem giroskopu, ki se nahaja v napravah iOS, razčlenimo na tri vrste rotacij. Te so: naklon, zasuk oziroma kotaljenje in odklon (angl. Pitch, roll, yaw). **Slika 4.7** ponazarja primerjavo med definiranimi premiki naprave pri merilniku pospeška ter pri giroskopu.



Slika 4.7 - Primerjava definiranih premikov naprave merilnika pospeška (leva polovica) in giroskopa (desna polovica), Apple Inc.

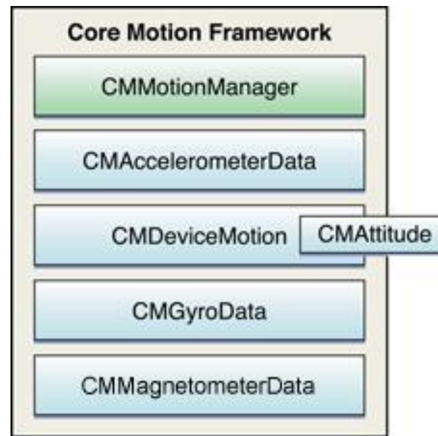
http://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/MotionEvents/MotionEvents.html

5 SISTEMSKO PROGRAMSKO OGRODJE CORE MOTION

V tem poglavju bomo podrobno predstavili sistemsko programsko ogrodje, ki se nahaja v sistemskem sloju, imenovanem Core Services. Core Services je relativno »nizek« sistemski sloj, ki nam omogoča komunikacijo s strojno opremo ter uporabo izmerjenih senzorskih podatkov v aplikacijah. Programsko ogrodje, v katerem sta zajeti uporaba in delo z izmerjenimi podatki senzorjev gibanja, se pri mobilnem operacijskem sistemu iOS imenuje Core Motion.

Core Motion je sistemsko programsko ogrodje, ki združuje oziroma centralizira procesiranje podatkov najpomembnejših senzorjev naprav iOS v enotnem ogrodju. Predstavljeno je bilo leta 2010, s prihodom operacijskega sistema iOS 4 ter naprave iPhone 4, ki je za razliko od predhodnih naprav iOS vsebovala tudi giroskop. Nastalo je z namenom poenostavitve predhodnih ločenih dostopov do podatkov posameznih senzorjev. Core Motion trenutno podpira procesiranje podatkov naslednjih senzorjev: merilnika pospeška, giroskopa in magnetometra. Ti senzorji posredujejo neobdelane oziroma izvorne podatke, ki jih Core Motion pred uporabo pretvori v definirano obliko. [4]

Kot vsako programsko ogrodje v računalniškem programiranju je tudi ogrodje Core Motion sestavljeno iz več blokov generične programske kode, ki predstavljajo dostop do funkcionalnosti ciljnega sistema. Pri objektno orientiranem okolju, kakršno je tudi okolje programskega jezika objektivni-C (angl. Objective-C), ki je primarni razvojni programski jezik za sistem iOS, so programska ogrodja v večini sestavljena iz abstraktnih programskih razredov. Uporaba takega programskega ogrodja poteka s kompozicijo in/ali dedovanjem teh razredov. **Slika 5.1** predstavlja poenostavljeno strukturo sistemskega ogrodja Core Motion v sistemu iOS 5.



Slika 5.1 - Struktura sistemskega ogrodja Core Motion sistema iOS 5, Apple Inc.
http://developer.apple.com/library/ios/documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/Art/core_motion_objects.jpg

Core Motion je ogrodje, specifično namenjeno za delo s podatki senzorjev gibanja, zato ni samodejno »dodano« k novo ustvarjenim projektom, kot so nekatera programska ogrodja, ki zagotavljajo osnovne komponente vsake aplikacije. Če želimo uporabljati storitve ogrodja Core Motion v aplikaciji, moramo ogrodje vključiti s pomočjo eksplicitne reference.

5.1 Kontrolni razred CMMotionManager

Kot lahko vidimo na zgornji sliki strukture Core Motion, je razred CMMotionManager označen z drugačno, zeleno barvo. CMMotionManager je namreč »kontrolni« programski razred, ki predstavlja izhodišče pridobivanja podatkov senzorjev. S sklicevanjem na lastnosti razreda CMMotionManager lahko pridobimo podatke določenega senzorja ali kontroliramo nastavitve zajemanja podatkov pripadajočega senzorja. [2]

Za uspešno uporabo razreda CMMotionManager je najprej potrebno deklarirati primerek oziroma instanco razreda:

```
CMMotionManager *motionManager = [[CMMotionManager alloc] init];
```

Deklarirana instanca razreda sedaj omogoča dostop do štirih vrst podatkov.

Ti so:

- **neobdelani podatki merilnika pospeška** (angl. Raw accelerometer data),

```
... = motionManager.accelerometerData;
```

- **neobdelani podatki giroskopa** (angl. Raw gyroscope data),

```
... = motionManager.gyroData;
```

- **neobdelani podatki magnetometra** (angl. Raw magnetometer data),

```
... = motionManager.magnetometerData;
```

- **procesirani podatki gibanja/lege naprave** (angl. Procesed device-motion data).

```
... = motionManager.deviceMotion;
```

5.2 Merilnik pospeška: razred CMAccelerometerData

CMAccelerometerData je eden od modro označenih razredov v strukturi ogrodja Core Motion, ki jo vidimo na sliki zgoraj (**Slika 5.1**). Modro označeni razredi so namenjeni za delo s podatki specifičnega senzorja naprave. Razredu CMAccelerometerData pripada senzor merilnika pospeška (angl. Accelerometer), kar pomeni, da so lastnosti razreda

ustrezno prirejene obliki izmerjenih podatkov senzorja. Instanca razreda CMAccelerometerData prejme s klicem lastnosti *accelerometerData,* ki se nahaja v razredu CMMotionManager, neobdelane podatke merilnika pospeška.

Do prejetih podatkov dostopamo preko razredne spremenljivke, poimenovane *acceleration*, ki je definirana na naslednji način:

```
@property(readonly, nonatomic) CMAcceleration acceleration
```

Razredna spremenljivka *acceleration* je določena s podatkovnim tipom razreda CMAcceleration, ki je definiran v razredu CMAccelerometer. Definicija lastnosti *acceleration* je sledeča:

```
typedef struct {
        double x;        // relativni pospešek vzdolž osi X
        double y;        // relativni pospešek vzdolž osi Y
        double z;        // relativni pospešek vzdolž osi Z
} CMAcceleration;
```

Podatki so rezultat meritve pospeška vzdolž treh prostorskih osi ter gravitacijske sile v danem trenutku poizvedbe. Izmerjeni podatki so izraženi v enoti *g* (gravitacijska sila), ki določa pospešek relativen s prostim padom, ki ni enak splošnemu pospešku, katerega merska enota je m/s^2.

Osnovno pridobivanje podatkov merilnika pospeška je prikazan v spodnjem kodnem bloku.

```
CMMotionManager *motionManager = [[CMMotionManager alloc] init];
[motionManager startAccelerometerUpdates];

...

CMAccelerometerData *podatkiMerPospeska = motionManager.accelerometerData;

double pospesekX = podatkiMerPospeska.acceleration.x;
double pospesekY = podatkiMerPospeska.acceleration.y;
double pospesekZ = podatkiMerPospeska.acceleration.z;

...

[motionManager stopAccelerometerUpdates];
```

5.3 Giroskop: razred CMGyroData

Razred CMGyroData je prav tako eden od modro označenih razredov v strukturi Core Motion sistema iOS 5, ki mu pripada senzor giroskopa. Instanca razreda CMGyroData prejme preko lastnosti *gyroData*, ki se nahaja v razredu CMMotionManager, neobdelane podatke giroskopa.

Do podatkov dostopamo preko razredne spremenljivke, poimenovane *rotationRate*, razreda CMGyroData, ki je definirana na naslednji način:

```
@property(readonly, nonatomic) CMRotationRate rotationRate
```

Razredna spremenljivka *rotationRate* je določena s podatkovnim tipom CMRotationRate, razreda CMGyro, ki je definirana na naslednji način:

```
typedef struct {
        double x;        // stopnja rotacije vzdolž osi X
        double y;        // stopnja rotacije vzdolž osi Y
        double z;        // stopnja rotacije vzdolž osi Z
} CMRotationRate;
```

Prejeti podatki predstavljajo rotacijo naprave vzdolž treh prostorskih osi v danem trenutku meritve oziroma poizvedbe. Podatki meritve so izraženi s stopnjo rotacije vzdolž določene osi, in sicer v radianih na sekundo (*rad/s*).

Osnovno pridobivanje podatkov giroskopa je prikazano v spodnjem kodnem bloku:

```
CMMotionManager *motionManager = [[CMMotionManager alloc] init];
[motionManager startGyroUpdates];

...

CMGyroData *podatkiGiroskopa = motionManager.gyroData;

double rotacijaX = podatkiGiroskopa.rotationRate.x;
double rotacijaY = podatkiGiroskopa.rotationRate.y;
double rotacijaZ = podatkiGiroskopa.rotationRate.z;

...

[motionManager stopGyroUpdates];
```

5.4 Magnetometer: razred CMMagnetometerData

Razred CMMagnetometerData je prav tako eden od modro označenih razredov v strukturi Core Motion sistema iOS 5, ki omogoča delo s podatki magnetometra v napravah iOS. Instanca razreda CMMagnetometerData prejme izvorne podatke magnetometra preko lastnosti *magnetometerData*, ki se nahaja v kontrolnem razredu CMMotionManager.

Do podatkov dostopamo v aplikaciji preko razredne lastnosti, poimenovane *magneticField*, razreda CMMagnetometerData, ki je definirana na naslednji način:

```
@property(readonly, nonatomic) CMMagneticField magneticField
```

Razredna spremenljivka *magneticField* je določena s podatkovnim tipom CMMagneticField, razreda CMMagnetometer, ki je definirana na naslednji način:

```
typedef struct {
        double x;        // gostota magnetnega polja vzdolž osi X
        double y;        // gostota magnetnega polja vzdolž osi Y
        double z;        // gostota magnetnega polja vzdolž osi Z
} CMMagneticField;
```

Podatki, ki jih pridobimo s klicem lastnosti *magnetometerData* ter nato razredne lastnosti *magneticField* predstavljajo gostoto magnetnega polja v območju naprave. Podatki meritev so izraženi z enoto mikro Tesla (μT).

Osnovno pridobivanje podatkov giroskopa je prikazan v spodnjem kodnem bloku:

```
CMMotionManager *motionManager = [[CMMotionManager alloc] init];
[motionManager startMagnetometerUpdates];

...

CMGyroData *podatkiMagnetometra = motionManager.magnetometerData;

double magnetnoPoljeX = podatkiMagnetometra.magneticField.x;
double magnetnoPoljeY = podatkiMagnetometra.magneticField.y;
double magnetnoPoljeZ = podatkiMagnetometra.magneticField.z;

...

[motionManager stopMagnetometerUpdates];
```

5.5 DeviceMotion: razred CMDeviceMotion

Prav tako kot prej opisani razredi CMAccelerometerData, CMGyroData in CMMagnetometerData, je tudi razred CMDeviceMotion namenjen za delo s podatki senzorjev.

Za razliko od prej omenjenih razredov, ki služijo za zajemanje in delo s trenutnimi ter neobdelanimi podatki specifičnega senzorja, nam razred CMDeviceMotion nudi podatke senzorjev merilnika pospeška ter giroskopa, ki so obdelani s pomočjo senzorskih združitvenih algoritmov (angl. Sensor fusion algorithms). Da bomo lažje razumeli bistvo združevanja podatkov senzorjev, poglejmo najprej omejitve dveh omenjenih senzorjev.

Merilnik pospeška je senzor, katerega naloga je merjenje pospeška naprave, ki ga povzroča uporabnik, z upoštevanjem gravitacijske sile. Če bi ga pri razvoju aplikacij želeli uporabiti kot senzor nagiba naprave, bi bili podatki zelo nenatančni in polni podatkovnega šuma. Razlog je seveda njegova arhitektura, ki je grajena za zajemanje linearnega pospeška naprave in ne rotacije naprave. Prav tako sta pri neobdelanih podatkih merilnika pospeška sili gravitacije in pospeška, povzročenega na napravo s strani uporabnika, združeni.

Giroskop je naslednji senzor za merjenje podatkov gibanja v napravah iOS, ki meri stopnjo rotacije v lastnem referenčnem okvirju. Težava se pri giroskopu pojavi, kadar želimo v aplikaciji zaznavati spremembe gravitacijske sile ali linearne translacije naprave. Prav tako je slabost neobdelanih podatkov giroskopa padec natančnosti sorazmerno s časom (angl. Data drift) ter neničelna rotacija pri mirovanju naprave.

Tukaj nastopi razred CMDeviceMotion ter tako imenovano združevanje senzorskih podatkov (angl. Sensor fusion). Splošno gledano, je cilj združevanja podatkov senzorjev kompenzacija omenjenih slabosti posameznih senzorjev in možnost izračunavanja drugih oblik gibanja, ki jih s samostojno uporabo senzorjev ni mogoče zaznavati.

Z uporabo razreda CMDeviceMotion, lahko dostopamo do podatkov naslednjih oblik gibanja:

- **prostorska usmerjenost oziroma orientacija naprave** (angl. Attitude),
- **natančna stopnja rotacije naprave (**angl. Unbiased rotation rate),
- **gravitacijska sila, delujoča na napravo** (angl. Force of gravity) in
- **pospešek, ki ga uporabnik povzroča na napravo** (angl. Acceleration).

5.5.1 Prostorska usmerjenost - ***attitude***

Prostorska usmerjenost je lastnost, ki pove trenutno orientacijo naprave v lastnem referenčnem okvirju. Do podatkov o usmerjenosti naprave dostopamo v aplikaciji preko lastnosti poimenovane *attitude*, razreda CMDeviceMotion, ki je definirana na naslednji način:

```
@property(readonly, nonatomic) CMAttitude *attitude
```

Kot lahko vidimo v zgornjem kodnem bloku, je lastnost *attitude* določena z objektnim tipom CMAttitude. CMAttitude je razred, ki nudi podatke o prostorski usmerjenosti, izražene v treh matematičnih oblikah, ki so opisane v nadaljevanju. [1]

- <u>prostorska usmerjenost izražena s Eulerjevimi koti</u>

»Eulerjevi koti so pripomoček za opis usmerjenosti togega telesa v trirazsežnem evklidskem prostoru. Telesu lahko določimo usmerjenost po nizu treh vrtenj, ki jih opisujejo Eulerjevi koti.« [14] Pri uporabi lastnosti *attitude*, izražene v obliki Eulerjevih kotov, pridobimo podatke vrtenj oziroma rotacij, ki jih omenja citat. Te rotacije so naklon, zasuk in odklon (angl. Pitch, roll, yaw), ki smo jih omenili že pri opisu delovanja giroskopa (glej podpoglavje **4.2**) ter ponazorili s **Slika 4.7**.

Naklon oziroma »pitch« je rotacija okrog prečne oziroma lateralne osi (angl. Lateral axis), ki poteka od levega do desnega roba naprave, gledajoč v zaslon napravo. Lastnost *pitch* je definirana na naslednji način:

```
@property(readonly, nonatomic) double pitch
```

Zasuk ali »roll« je rotacija okrog vzdolžne oziroma longitudinalne osi (angl. Longitudinal axis), ki poteka od vrhnjega roba do spodnjega roba naprave, gledajoč v zaslon naprave. Lastnost *roll* je definirana na naslednji način:

```
@property(readonly, nonatomic) double roll
```

Odklon oziroma »yaw« je rotacija okrog navpične oziroma vertikalne osi (angl. Vertical axis), ki poteka skozi napravo in je na njo pravokotna. Lastnost *yaw* je definirana na naslednji način:

```
@property(readonly, nonatomic) double yaw
```

Vse tri rotacije so izražene v radianih.

- prostorska usmerjenost, izražena z matriko rotacije

Matrika rotacije oziroma matrika vrtenja je matrika, s katero opišemo rotacijo v Evklidskem prostoru. V našem primeru gre za rotacije v trirazsežnem evklidskem prostoru, zato je tudi matrika rotacije velikosti 3x3. **Slika 5.2** ponazarja rotacije okrog posamezne osi v desno orientiranem prostoru.

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

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

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

Slika 5.2 - Matrike posameznih rotacij okrog osi trirazsežnega evklidskega prostora, Wikipedia,
http://upload.wikimedia.org/wikipedia/sl/math/5/1/4/5148f88bf9e6811e35615c08d2839793.png

Za pridobivanje podatkov o prostorski usmerjenosti, izraženi z matriko rotacije, se v razredu CMAttitude sklicujemo na lastnost *rotationMatrix*, ki je definirana na naslednji način:

```
@property(readonly, nonatomic) CMRotationMatrix rotationMatrix
```

Kot je razvidno iz zgornjega kodnega bloka, je lastnost *rotationMatrix* določena s tipom CMRotationMatrix, katerega definicijo prikazuje naslednji kodni blok:

```
typedef struct {
        double m11, m12, m13;     // elementi prve vrstice v matriki
        double m21, m22, m23;     // elementi druge vrstice v matriki
        double m31, m32, m33;     // elementi tretje vrstice v matriki
} CMRotationMatrix;
```

Vsak element v strukturi podatkovnega tipa CMRotationMatrix, predstavlja element v matriki rotacije. Vsak element je predstavljen z realnim številom.

- <u>prostorska usmerjenost, izražena s kvaternionom</u>

»Kvaternion je nekomulativna razširitev kompleksnega števila. Sestavljen je iz treh imaginarnih enot *i*, *j*, in *k*, za katere veljajo zveze: $i^2 = j^2 = k^2 = ijk = -1$.« [15] Definicija kvaterniona je q = **w** + **x**i + **y**j + **z**k. Prednost kvaterniona je v tem, da ga lahko uporabimo za opisovanje kakršne koli rotacije. Vsako rotacijo v prostoru lahko namreč opišemo s kotom okrog določene osi. Omenjen kot določa parameter w, os, okrog katere rotiramo, pa določajo parametri x, y in z.

Za pridobivanje podatkov o prostorski usmerjenosti, izraženi s kvaternionom, se v razredu CMAttitude sklicujemo na lastnost *quaternion*, ki je definirana na naslednji način:

```
@property(readonly, nonatomic) CMQuaternion quaternion
```

Lastnost *quaternion* je določena s tipom CMQuaternion, ki je definiran s strukturo:

```
typedef struct {
        double x;        // vrednost za os x
        double y;        // vrednost za os y
        double z;        // vrednost za os z
        double w;        // vrednost kota rotacije
} CMQuaternion;
```

*5.5.2 Natančna stopnja rotacije naprave - **rotationRate***

Natančna stopnja rotacija naprave ali »unbiased device rotation rate« je lastnost razreda CMDeviceMotion, ki pove, kakšna je trenutna stopnja rotacije naprave okrog posamezne

osi trirazsežnega prostora. Do tega podatka dostopamo preko lastnosti *rotationRate*, ki je definirana na naslednji način:

```
@property(readonly, nonatomic) CMRotationRate rotationRate
```

Definicija lastnosti *rotationRate* je popolnoma enaka kot pri istoimenski lastnosti razreda CMGyroData. Pri obeh lastnostih namreč dostopamo do podatkov giroskopa. Razlika je le v procesiranju podatkov. Medtem ko nam lastnost rotationRate, razreda CMGyroData, nudi podatke v neobdelani (angl. Raw) obliki, nam omenjena lastnost, v razredu CMDeviceMotion, nudi izboljšano oziroma natančnejšo obliko teh podatkov. To dosega s pomočjo združitvenih algoritmov ogrodja Core Motion.

Pri uporabi neobdelanih podatkov stopnje rotacije naprave prihaja namreč do majhne napake pri mirovanju naprave. To napako imenujemo »bias« oziroma odmik od realnega stanja.

Osnovno uporabo natančne stopnje rotacije naprave prikazuje naslednji kodni blok:

```
...

double stopnjaRotacijeOkrogX = deviceMotion.rotationRate.x;
double stopnjaRotacijeOkrogY = deviceMotion.rotationRate.y;
double stopnjaRotacijeOkrogZ = deviceMotion.rotationRate.z;

...
```

*5.5.3 Gravitacijska sila delujoča na napravo - **gravity***

Gravitacijska sila je lastnost, ki nam opiše velikost gravitacijske sile, ki deluje na napravo. Podatek je izmerjen v času poizvedbe lastnosti *gravity*, razreda CMDeviceMotion, ki je vektorska predstavitev gravitacijske sile, delujoče v referenčnem okvirju naprave.

Lastnost *gravity* je razdeljena na osi X, Y in Z. Posamezen parameter odraža vrednost težnostne sile vzdolž pripadajoče osi. Spodnji kodni blok prikazuje definicijo lastnosti *gravity* ter definicijo njene strukture.

```
@property(readonly, nonatomic) CMAcceleration gravity


typedef struct {
        double x;        // gravitacijska sila vzdolž x osi
        double y;        // gravitacijska sila vzdolž y osi
        double z;        // gravitacijska sila vzdolž z osi
} CMAcceleration;
```

Osnovno uporabo lastnosti *gravity* oziroma podatka o gravitacijski sili delujoči na napravo prikazuje naslednji kodni blok:

```
...

double gravitacijskaSilaX = deviceMotion.gravity.x;
double gravitacijskaSilaY = deviceMotion.gravity.y;
double gravitacijskaSilaZ = deviceMotion.gravity.z;

...
```

Izmerjeni podatki so izraženi v enoti *g* (gravitacijska sila oziroma težni pospešek).

5.5.4 Pospešek, ki ga uporabnik povzroča na napravo - ***userAcceleration***

Uporabnik s premikanjem naprave povzroči delovanje sil na napravo, ki, po drugem Newtonovem zakonu, povzročajo njen pospešek.

Podatki so zajeti v lastnost *userAcceleration*, ki je vektorska predstavitev sil pospeška, povzročenega s strani uporabnika. Spodnji kodni blok prikazuje definicijo lastnosti *userAcceleration* ter definicijo njene strukture.

```
@property(readonly, nonatomic) CMAcceleration userAcceleration


typedef struct {
        double x;        // sila pospeška vzdolž x osi
        double y;        // sila pospeška vzdolž y osi
        double z;        // sila pospeška vzdolž z osi
} CMAcceleration;
```

Osnovno uporabo lastnosti *userAcceleration* oziroma podatka o pospešku, povzročenem na napravo, prikazuje naslednji kodni blok:

```
...

double silaPospeskaX = deviceMotion.userAcceleration.x;
double silaPospeskaY = deviceMotion.userAcceleration.y;
double silaPospeskaZ = deviceMotion.userAcceleration.z;

...
```

Kot vidimo, lahko pri razredu CMDeviceMotion, ločeno dostopamo do težnega pospeška (gravitacije) in pospeška, ki ga na napravo povzroča uporabnik. Zasluge za to imajo ravno združitveni algoritmi razreda CMDeviceMotion, s katerimi lahko iz določenih podatkov filtriramo posamezne komponente gibanja.

Skupen pospešek naprave dobimo s seštevanjem vrednosti iz lastnosti *gravity* in lastnosti *userAcceleration*. Seštevek teh vrednosti je enak neobdelanim podatkom, ki jih nudi lastnost *acceleration* razreda CMAccelerometerData (glej podpoglavje 5.2).

5.6 Povzetek

Zaradi obsežnosti programskega ogrodja Core Motion bomo v tem podpoglavju predstavili strukturiran povzetek ogrodja s seznamom funkcionalnosti.

Neobdelani podatki merilnika pospeška:

- CM**AccelerometerData**
    - *.acceleration* (CM**Acceleration**).

Neobdelani podatki giroskopa:

- CM**GyroData**

        + *.rotationRate* (CM**RotationRate**).

Neobdelani podatki magnetometra:

- CM**MagnetometerData**
    + *.magneticField* (CM**MagneticField**).

Podatki merilnika pospeška in giroskopa, obdelani s pomočjo združitvenih algoritmov:

- CM**DeviceMotion**
    + *.attitude* (CM**Attitude**),
        - *.pitch, .roll, .yaw* (double),
        - *.rotationMatrix* (CM**RotationMatrix**),
        - *.quaternion* (CM**Quaternion**),
    + *.rotationRate* (CM**RotationRate**),
    + *.gravity* (CM**Acceleration**),
    + *.userAcceleration* (CM**Acceleration**).

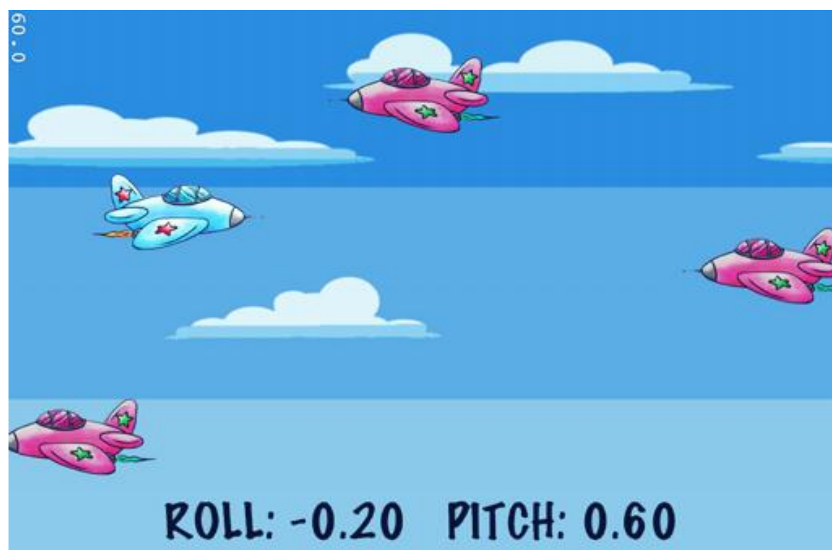
6 UPORABA PODATKOV GIBANJA PRI RAZVOJU IGER

Kot praktični del raziskovanja problema diplomske naloge smo razvili tri prototipne aplikacije, ki izkoriščajo zajete podatke gibanja na napravah iOS. Vsak izdelan prototip povzema oziroma prikazuje drugačno metodo zajemanja in uporabe podatkov gibanja. S prototipi smo skušali prikazati različne pristope k uporabi funkcionalnosti zaznavanja gibanja na mobilnih napravah.

Za razvoj prvih dveh prototipov smo uporabili zunanjo knjižnico Cocos2D, pri tretjem pa smo uporabili vključene sistemske knjižnice oziroma programska ogrodja sistema iOS. Knjižnico Cocos2D smo uporabili z namenom hitrejše gradnje posodobitvenega cikla igre ter lažjega prikaza grafičnih elementov in uporabniškega vmesnika. Knjižnice ne bomo posebej opisovali, saj za predstavitev uporabe podatkov gibanja ni ključnega pomena.

6.1 Prototip 1 - upravljanje letala v 2D svetu z uporabo podatkov giroskopa

V prvem prototipu bomo predstavili preprosto metodo uporabe podatkov giroskopa v programskem ogrodju Core Motion. Cilj je bil izdelati enostavno 2D igro, v kateri igralec z nagibanjem naprave usmerja letalo, z namenom izogibanja sovražnim letalom. Grafični vmesnik prikazuje **Slika 6.1**.



Slika 6.1 - Grafični vmesnik 2D igre (prototip 1)

Namen je bil izdelati čim bolj naravno interakcijo med nagibom naprave ter premikom osrednjega objekta v igri, ki je v našem primeru letalo. Hitro lahko ugotovimo, da so možni premiki letala gor-dol in levo-desno. Če skušamo omenjene premike letala povezati z lastnostmi enega od razredov ogrodja Core Motion, vidimo, da je razred CMDeviceMotion z njegovo lastnostjo *attitude* najbolj ustrezen.

Lastnost *attitude* določa usmerjenost naprave v lastnem referenčnem okvirju. To lahko zelo enostavno preslikamo v premike letala, in sicer s komponentami lastnosti *attitude*, ki so naklon, zasuk in odklon (angl. Pitch, roll, yaw). Pri tem moramo upoštevati pokrajinsko (angl. Landscape) postavitev igralne scene. Za premik letala gor-dol bomo uporabili komponento *roll*, za premik letala levo-desno pa komponento *pitch* (glej **Slika 4.7**). Ker gre v našem primeru za 2D igro, ne bomo potrebovali tretje dimenzije oziroma *yaw* komponente lastnosti *attitude*, ki določa usmerjenost glede na os Z v prostoru.

Večja kot je usmerjenost oziroma nagib naprave v določeni smeri, večja je vrednost pripadajoče komponente lastnosti *attitude*. Komponente lahko imajo tudi negativne vrednosti, kar je odvisno od strani nagiba. Ti trditvi nam pojasnita, da lahko vrednosti komponent *roll* in *pitch* uporabimo kot določanje hitrosti letala: komponento *roll* za hitrost v smeri osi Y, ter komponento *pitch* v smeri osi X. Hitrost letala v vsakem posodobitvenem ciklu igre dodamo k poziciji letala.

Spodnji kodni blok prikazuje deklaracijo kontrolnega razreda CMMotionManager ogrodja Core Motion, ki nam omogoča dostop do podatkov gibanja naprave. Podrobnejši opis kontrolnega razreda smo podali v podpoglavju 5.1.

```
// datoteka: GameScene.h


        @interface GameScene : CCLayer
        {
                 CCSprite *player;
                 CGPoint playerVelocity;

                 ...

                 // Deklaracija »kontrolnega« razreda ogrodja CoreMotion
                 CMMotionManager *motionManager
        }
```

Naslednji kodni blok prikazuje konfiguracijo kontrolnega razreda CMMotionManager, ki vključuje nastavitev frekvence posodobitve podatkov, preverjanje razpoložljivosti podatkov ter zaganjanje postopka pridobivanja želenih podatkov.

```
// datoteka: GameScene.m


- (id)init
{
        if (self = [super init])
        {
                 // Alokacija in inicializacija kontrolnega razreda
                 motionManager = [[CMMotionManager alloc] init];

                 // Nastavitev frekvence posodobitve podatkov
                 motionManager.deviceMotionUpdateInterval = 1.0/60.0;

                 // Zagon pridobivanja podatkov razreda CMDeviceMotion
                 [motionManager startDeviceMotionUpdates];


                 // Preverjanje ce so podatki razreda CMDeviceMotion na voljo
                 if (!motionManager.isDeviceMotionAvailable)
                 {
                          // Ustavljanje pridobivanja podatkov
                          [motionManager stopDeviceMotionUpdates];

                          // Sproscanje pomnilnika
                          [motionManager release];
                 }

                 ...
        }
}
```

Naslednji in zadnji blok prikazuje dogajanje v metodi posodobitvenega cikla igre, kjer v vsakem ciklu najprej pridobimo trenutne podatke iz senzorjev, ki jih nato razdelimo na potrebne komponente ter uporabimo pri posodobitvi potrebnih podatkov. V našem primeru posodabljamo hitrost letala, ki jo nato prištejemo k poziciji iz prejšnjega posodobitvenega cikla, na koncu pa posodobimo še trenutno pozicijo letala.

```
// datoteka: GameScene.m

// Metoda posodobitvenega cikla igre
- (void)update:(ccTime)delta
{

        // Pridobivanje trenutnih deviceMotion podatkov iz kontrolnega razreda
        CMDeviceMotion *motionData = motionManager.deviceMotion;

        // Pridobivanje lastnosti attitude razreda CMDeviceMotion
        CMAttitude *attitude = motionData.attitude;

        // Razdelitev lastnosti attitude na komponenti pitch in roll
        double naklon = attitude.pitch;
        double zasuk = attitude.roll;

        // Posodabljanje hitrosti letala s trenutnimi vrednostmi naklona in zasuka
        playerVelocity.x = playerVelocity.x * zracniUpor + naklon * faktor;
        playerVelocity.y = playerVelocity.y * zracniUpor + zasuk * faktor;

        // Dodajanje vrednosti vektorja hitrosti k stari poziciji letala
        CGPoint novaPozicija = player.position;
        novaPozicija.x += playerVelocity.x;
        novaPozicija.y += playerVelocity.y;

        ...

        // Posodobitev pozicije letala
        player.position = novaPozicija;
}
```

6.2 Prototip 2 - metanje žoge v 2D prostoru z uporabo podatkov merilnika pospeška

Namen drugega prototipa je predstaviti zajemanje in uporabo podatkov merilnika pospeška. Pred prihodom programskega ogrodja Core Motion je bila uporaba podatkov merilnika precej zahtevnejša. Razred UIAccelerometer je namreč zagotavljal neobdelane podatke, kar pomeni, da sta bila sila gravitacije ter pospešek, ki ga je na napravo povzročal uporabnik, združena. S prihodom programskega ogrodja Core Motion ter njenih

združitvenih algoritmov sta omenjena pospeška postala na voljo ločeno. Kot je opisano v podpoglavju 5.5, nam razred CMDeviceMotion sedaj ponuja lastnosti *gravity* ter *userAcceleration*.

Drug prototip je enostavna 2D igra, v kateri skuša igralec s košarkaško žogo zadeti čim več košev. Grafični izgled prototipa prikazuje **Slika 6.2**.

Slika 6.2 - Grafični vmesnik 2D igre (prototip 2)[2]

Košarkaška žoga, osrednji objekt v igri, ima realne lastnosti, kot so odbojnost, gostota in trenje. Prav tako sta v igri upoštevana odboj ter sila gravitacije. »Metanje žoge« igralec doseže s povzročanjem pospeška na napravo. Če napravo sunkovito premaknemo v smeri navzgor, bomo tako povzročili silo na žogo v smeri navzgor. Enako velja za vse ostale

[2] Ozadje grafičnega vmesnika izvzeto iz vira: http://photos.travelblog.org/Photos/64811/413145/f/3983128-Central-Park-at-dusk-1.jpg, TravelBlog.org & Bloggers

smeri. Seveda moramo upoštevati, da se sila gravitacije in sila, povzročena s strani igralca, seštevata, zato bo rezultanta teh sil različna glede na smer potovanja žoge.

Spodnji kodni blok prikazuje deklaracijo že omenjenega kontrolnega razreda CMMotionManager ter nabor potrebnih instančnih spremenljivk.

```
// datoteka: GameScene.h


@interface GameScene : CCLayer
{
        // Deklaracija spremenljivke, ki doloca togo telo
        b2Body *ball;

        // Deklaracija spremenljivke sveta, ki zajema delo s simulacijo,
           upravljanje s telesi, posodobitveni cikel itd
        b2World *world;

        // Deklaracija oznake za izpisovanje povzrocenega pospeska na zogo
        CCLabelTTF *accelerationLabel;

        // Deklaracija oznake za izpisovanje števila točk
        CCLabelTTF *scoreLabel;

        // Deklaracija spremenljivk, ki dolocata zakljucno stanje pospeska
        float xAcceleration;
        float yAcceleration;

        ...

        // Deklaracija »kontrolnega« razreda ogrodja CoreMotion
        CMMotionManager *motionManager
}
```

Kot smo omenili, je v igri upoštevana tudi gravitacija. Spodnji kodni blok prikazuje določanje gravitacijskega vektorja, ki ga dodelimo ustvarjenemu igralnemu svetu. Določena gravitacija je dvakrat večja oziroma močnejša od zemljine, ki znaša približno 9.81 m/s^2.

```
// datoteka: GameScene.m


- (void)initPhysics
{
        // Dolocanje gravitacije
        b2Vec2 gravity;
        gravity.Set(0.0f, -20.0f);

        // Inicializacija sveta z doloceno silo gravitacije
        world = new b2World(gravity);
}
```

V nadaljevanju je prikazana osrednja inicializacijska metoda, v kateri inicializiramo potrebne komponente, definiramo statične grafične elemente, oznake ter druge vire in spremenljivke, ki jih potrebujemo v igri. Najpomembnejše stvari v tem bloku so alokacija in inicializacija razreda CMMotionManager, nastavitev posodobitvenega intervala pridobivanja podatkov ter zagon zajemanja podatkov gibanja. Pomembno je, da pred začetkom zajemanja podatkov preverimo, ali je naprava sposobna zaznavanja gibanja in ali so podatki na voljo.

```
// datoteka: GameScene.m


- (id)init
{
        if (self = [super init])
        {
                 // Alokacija in inicializacija kontrolnega razreda
                 motionManager = [[CMMotionManager alloc] init];

                 // Nastavitev frekvence posodobitve podatkov
                 motionManager.deviceMotionUpdateInterval = 1.0/60.0;

                 // Zagon pridobivanja podatkov razreda CMDeviceMotion
                 [motionManager startDeviceMotionUpdates];


                 // Preverjanje ce so podatki razreda CMDeviceMotion na voljo
                 if (!motionManager.isDeviceMotionAvailable)
                 {
                         // Ustavljanje pridobivanja podatkov
                         [motionManager stopDeviceMotionUpdates];

                         // Sproscanje pomnilnika
                         [motionManager release];
                 }
                 ...

                 // Dolocanje grafike za kosarkasko zogo
                 CCSpriteBatchNode *ball_sprite =
                 [CCSpriteBatchNode batchNodeWithFile:@"ball.png" capacity:100];

                 // Dodajanje kosarkaske zoge v sloj igre (angl. Game layer)
                 [self addChild:ball_sprite z:0 tah:kTagBallNode];
                 [self addNewSpriteAtPosition:ccp(screen.widt/2, screen.height/2];

                 // Dolocanje grafike za kosarkaski kos
                 CCSprite *basket =
                 [CCSprite spriteWithFile:@"bas.png" rect:CGRectMake(0,0,128,256)];
                 basket.position = CGPointMake(screen.width, screen.height/2);
                 [self addChild:basket];

                 // Dolocanje napisa za stevilo tock
                 scoreLabel = [CCLabelTTF labelWithString:@"Score: 0" fontSize:32];
                 scoreLabel.color = ccc3(255,255,255);
                 scoreLabel.position = ccp(screen.width-70, screen.height-30);
                 [self addChild:scoreLabel z:0];

                 // Dolocanje napisa za povzrocen pospesek na zogo
                 accelerationLabel =
                 [CCLabelTTF labelWithString:@"UserAcceleration   x:0.0  y:0.0"
                 fontSize:24];
                 accelerationLabel.color = ccc3(255,255,255);
                 accelerationLabel.position = ccp(screen.width/2, 20);
                 [self addChild: accelerationLabel z:0];

                 ...
        }
}
```

V naslednjem bloku je zajeta glavnina dela s podatki gibanja. Najprej pridobimo instanco razreda CMDeviceMotion, nato pa podatke iz njene lastnosti *userAcceleration*. Kot je že znano, lastnost *userAcceleration* zajema podatke pospeška, ki ga uporabnik povzroča na napravo vzdolž osi evklidskega prostora.

```
// datoteka: GameScene.m


- (void)update:(ccTime)delta
{
        // Dolocanje minimalnega praga zaznavanja pospeska
        float minPosAcc = 1.0f;

        // Dolocanje horizontalnega in vertikalnega mnozilnega faktorja
        float horMult = 80.0f;
        float verMult = 60.0f;

        ...

        // Pridobivanje trenutnih deviceMotion podatkov iz kontrolnega razreda
        CMDeviceMotion *motionData = motionManager.deviceMotion;

        // Pridobivanje lastnosti userAcceleration razreda CMDeviceMotion
        CMAcceleration userAcceleration = motionData.userAcceleration;

        // Preverjanje ce je trenutni pospesek vzdolz osi X vecji od praga
        if (userAcceleration.x > minPosAcc)
        {
             // Ce je pospesek vecji od praga, ga shranimo v lokalno spremenljivko
             xAcceleration = userAcceleration.x;
        }

        // Preverjanje ce je trenutni pospesek vzdolz osi Y vecji od praga
        if (userAcceleration.y > minPosAcc)
        {
             // Ce je pospesek vecji od praga, ga shranimo v lokalno spremenljivko
             yAcceleration = userAcceleration.y;
        }

        // Preverjanje ce je povzrocanje pospeska zakljuceno (zmanjsanje pospeska)
        if ((userAcceleration.x < minPosAcc || userAcceleration.y < minPosAcc)
         && (xAcceleration > minPosAcc || yAcceleration > minPosAcc))
        {

          // Dodelitev sile koncnega stanja pospeska na center zoge
          ball->ApplyForceToCenter(b2Vec2(horMult*xAcceleration*ball->GetMass(),
                                          (verMult*yAcceleration*ball->GetMass()));

          // Posodobitev oznake za koncno stanje pospeska vzdolz osi X in Y
          accelerationLabel.string =
                 [NSString stringWithFormat:@"UserAcceleration   x: %.2f  y: %.2f",
                  xAcceleration, yAcceleration];

          // Nastavitev vrednosti spremenljivke koncnega pospeska na vrednost 0
          xAcceleration = 0.0f;
          yAcceleration = 0.0f;
        }
}
```

Če se za trenutek še osredotočimo na prejšnji kodni blok, lahko vidimo, da preverjamo, ali so trenutni podatki pospeška večji od določenega praga. Ker že srednje hitro premikanje naprave nanjo povzroči minimalen pospešek, smo določili minimalno vrednost pospeška oziroma prag. Da povzročen pospešek upoštevamo, mora biti večji od določenega praga.

Naslednja ovira je bila zaznavanje maksimalne oziroma končne vrednosti povzročenega pospeška. Telo metode, ki je prikazana v zgornjem kodnem bloku, se izvaja v vsakem posodobitvenem ciklu, zato ne moremo uporabiti vsake trenutne vrednosti povzročenega pospeška, pa čeprav je večji od praga. Poglejmo primer iz realnosti. Sila, s katero košarkar vrže žogo, je sila, ki deluje tik pred izgubo kontakta z žogo. Tudi pri zajemanju pospeška moramo na to paziti ter žogi dodeliti le zadnjo oziroma končno silo v poteku pospeška. To storimo tako, da preverimo, ali je vrednost trenutnega pospeška manjša od praga ter, ali smo v katerem od posodobitvenih ciklov pred tem dosegli vrednost, večjo od določenega praga. Če poenostavimo, preverjamo, ali je v preteklosti prišlo do povzročitve pospeška, ki je sedaj že zaključen. Žogi nato dodelimo ustrezno silo vzdolž osi X in osi Y.

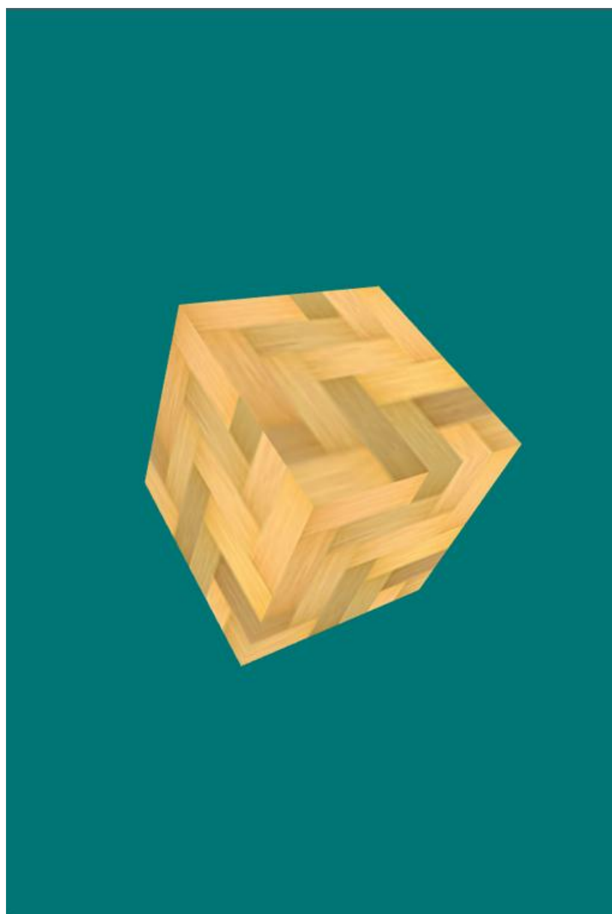
Spodnji blok prikazuje metodo s katero prenesemo silo povzročenega pospeška na žogo. Kot smo omenili, se sile povzročene na telo oziroma žogo seštevajo. Te sile, so sile povzročenega pospeška ter sila gravitacije.

```
- (void)ApplyForceToCenter:(const b2Vec2& force)
{
        // Preverjanje ce je telo, ki mu dodeljujemo silo, dinamicno
        if (m_type != b2_dynamicBody)
        {
                return;
        }

        // Pristevanje vrednosti nove sile k trenutni
        m_force += force;
}
```

6.3 Prototip 3 - rotacija 3D kocke s pomočjo podatkov giroskopa

Tretji prototip zajema poenostavljeno rotacijo kocke v 3D prostoru, s pomočjo podatkov giroskopa. Cilj prototipa je, prikazati prostorsko rotacijo glede na podatke gibanja naprave. Grafični vmesnik prikazuje slika **Slika 6.3**.



Slika 6.3 - Grafični vmesnik (prototip 3)

Posebnost tega prototipa so podatki v obliki kvaternionov. Prednost kvaternionov je v tem, da lahko z njimi opišemo katero koli rotacijo. Podroben opis kvaternionov v ogrodju Core Motion se nahaja v podpoglavju 5.5.1. Podatke smo, kot vedno poprej, zajemali iz programskega ogrodja Core Motion, in sicer iz razreda CMAttitude. CMAttitude nudi podatke v treh oblikah. Ena izmed podatkovnih oblik je kvaternionska. Kvaternionsko obliko pridobimo iz lastnosti *quaternion*, razreda CMAttitude.

Ker je deklaracija in inicializacija kontrolnega razreda CMMotionManager popolnoma enaka kot pri prejšnjih primerih (prototip 1, prototip 2), je ne bomo posebej opisovali.

Spodnji kodni blok združeno prikazuje deklaracijo in inicializacijo kontrolnega razreda CMMotionManager ter deklaracijo in inicializacijo spremenljivke referenceAttitude, tipa CMAttitude, ki hrani referenčno prostorsko usmerjenost.

```
// datoteka: MainViewController.h

@interface MainViewController : GLKViewController

        @property (nonatomic, retain) CMMotionManager *motionManager;
        @property (nonatomic, retain) CMAttitude *referenceAttitude;

@end



// datoteka: MainViewController.m

- (void)viewDidLoad
{
        [super viewDidLoad];

        self.motionManager = [[CMMotionManager alloc] init];
        self.motionManager.deviceMotionUpdateInterval = 1.0/60.0;

        [self.motionManager startDeviceMotionUpdates];

        if (self.motionManager.isDeviceMotionAvailable)
        {
                self.referenceAttitude = self.motionManager.deviceMotion.attitude;

        } else {
                [self.motionManager stopDeviceMotionUpdates];
        }
}
```

Za trenutek se ustavimo pri referenčni prostorski usmerjenosti, ki jo hrani instančna spremenljivka *referenceAttitude*. Referenčna prostorska usmerjenost je podatek o prostorski usmerjenosti, ki jo pridobimo na začetku aplikacije ter jo hranimo skozi izvajanje aplikacije. Hranimo jo zato, ker vsako rotacijo izvajamo na podlagi produkta trenutne prostorske usmerjenosti in inverzne vrednosti referenčne oziroma začetne prostorske usmerjenosti.

Naslednji kodni blok prikazuje zajemanje podatkov trenutne prostorske usmerjenosti ter omenjen produkt obeh prostorskih usmerjenosti. Podatki, ki jih dobimo, so predstavljeni v

obliki kvaternionov s katerimi lahko izračunamo oziroma določimo rotacijsko matriko. Parameter w, lastnosti *quaternion*, predstavlja kot rotacije, parametri x, y, z pa os okrog katere rotiramo osrednji objekt. Na koncu preostane le še množenje translacijske in izračunane rotacijske matrike ter določanje rotacije osrednjemu objektu.

```
// datoteka: MainViewController.m

- (void)update
{
        // Zajemanje podatkov razreda CMDeviceMotion
        CMDeviceMotion *deviceMotion = self.motionManager.deviceMotion;

        // Pridobivanje podatkov o trenutni prostorski usmerjenosti
        CMAttitude *attitude = deviceMotion.attitude;

        // Preverjanje ce referencna prostorska usmerjenost ni enaka nil
        if (self.referenceAttitude != nil)
        {
                // Mnozenje trenutne prostorske usmerjenosti z inverzom referencne
                // prostorske usmerjenosti
                [attitude multiplyByInverseOfAttitude:self.referenceAttitude];
        }

        // Pridobivanje produkta prostorskih usmerjenosti v obliki kvaterniona
        CMQuaternion quaternion = attitude.quaternion;

        // Deklaracija translacijske matrike - pomik telesa nazaj vzdolz Z osi
        GLKMatrix4 viewMatrix = GLKMatrix4MakeTranslation(0.0f, 0.0f, -6.0f);

        // Deklaracija rotacijske matrike dolocene s parametri kvaterniona
        GLKMatrix4 rotationByQuaternion =
          GLKMatrix4MakeRotation((float)(speed * acos(quaternion.w) * 180.0/M_PI),
                                 quaternion.x, quaternion.y, quaternion.z);

        // Mnozenje obeh transformacijskih matrik
        viewMatrix = GLKMatrix4Multiply(modelViewMatrix, rotationByQuaternion);

        // Dolocanje transformacijske matrike osrednjemu objektu
        self.effect.transform.modelviewMatrix = viewMatrix;
}
```

7 SKLEP

V diplomski nalogi smo podrobno predstavili arhitekturo operacijskega sistema iOS, s poudarkom na umeščenosti pomembnih programskih ogrodij v slojih sistema. Opisali in predstavili smo princip delovanja strojne opreme za zaznavanje in zajemanje podatkov gibanja naprave ter v celoti predstavili sistemsko programsko ogrodje Core Motion, ki omogoča zajemanje in delo s podatki gibanja.

Skozi raziskavo programskega ogrodja Core Motion, smo izdelali nekaj aplikacijskih prototipov, s katerimi smo prikazali več vrst uporabe podatkov gibanja ter opozorili na področja uporabe pri posameznih vrstah podatkov.

Pri raziskovanju področja zajemanja in uporabe podatkov gibanja na napravah iOS, smo si nabrali izkušnje, s katerimi bi lahko izdelali zahtevno 2D igro oziroma aplikacijo, ki bi koristila podatke gibanja v naprednih oblikah kot so filtriranje določenih podatkovnih območij, združevanje podatkov z namenom definiranja novih premikov in podobno.

Raziskavo diplomske naloge, bi lahko nadgradili, z dodatnimi aplikacijskimi prototipi, s katerimi bi ponazorili še preostale metode in načine podatkov gibanja.

8 VIRI IN LITERATURA

[1] Apple Inc., iOS Developer Library, CMAttitude Class Reference, http://developer.apple.com/library/ios/#documentation/CoreMotion/Reference/CMAttitude_Class/Reference/Reference.html#//apple_ref/doc/c_ref/CMAttitude [10. 8. 2012]

[2] Apple Inc., iOS Developer Library, CMMotionManager Class Reference, http://developer.apple.com/library/ios/#documentation/CoreMotion/Reference/CMMotionManager_Class/Reference/Reference.html#//apple_ref/occ/cl/CMMotionManager [30. 6. 2012]

[3] Apple Inc., iOS Developer Library, Core Animation Programming Guide, http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/CoreAnimation_guide/Articles/WhatisCoreAnimation.html#//apple_ref/doc/uid/TP40004689-SW1 [7. 6. 2012]

[4] Apple Inc., iOS Developer Library, Event Handling Guide for iOS, http://developer.apple.com/library/ios/#documentation/EventHandling/Conceptual/EventHandlingiPhoneOS/MotionEvents/MotionEvents.html#//apple_ref/doc/uid/TP40009541-CH4-SW1 [29. 6. 2012]

[5] Apple Inc., iOS Developer Library, iOS Technology Overview, http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html#//apple_ref/doc/uid/TP40007898 [5. 6. 2012]

[6] Apple Inc., iOS Developer Library, iOS Technology Overview, Core OS Layer, http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreOSLayer/CoreOSLayer.html#//apple_ref/doc/uid/TP40007898-CH11-SW1 [5. 6. 2012]

[7] Apple Inc., iOS Developer Library, iOS Technology Overview, Core Services Layer, http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreServicesLayer/CoreServicesLayer.html#//apple_ref/doc/uid/TP40007898-CH10-SW5 [5. 6. 2012]

[8] Apple Inc., iOS Developer Library, iOS Technology Overview, Media Layer, http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/MediaLayer/MediaLayer.html#//apple_ref/doc/uid/TP40007898-CH9-SW4 [5. 6. 2012]

[9] Apple Inc., iOS Developer Library, iOS Technology Overview, Cocoa Touch Layer, http://developer.apple.com/library/ios/#documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSTechnologies/iPhoneOSTechnologies.html#//apple_ref/doc/uid/TP40007898-CH3-SW1 [5. 6. 2012]

[10] Chipworks, iPhone 4S Teardown: A closer look at the chips inside, http://www.chipworks.com/en/technical-competitive-analysis/resources/recent-teardowns/category/phone/page/2/ [22. 6. 2012]

[11] Engineerguy, Watch how iPhone accelerometers work and are made, http://www.gottabemobile.com/2012/05/24/watch-how-iphone-accelerometers-work-and-are-made/ [22. 6. 2012]

[12] iFixit, Teardown, iPhone 4S Teardown, http://www.ifixit.com/Teardown/iPhone-4S-Teardown/6610/2 [20. 6. 2012]

[13] Chris Lattner , LLVM.org, LLVM Overview, http://llvm.org/ [4. 6. 2012]

[14] Wikipedia, Eulerjevi koti, http://sl.wikipedia.org/wiki/Eulerjevi_koti [3. 8. 2012]

[15] Wikipedia, Kvaternion, http://sl.wikipedia.org/wiki/Kvaternion [5. 8. 2012]

[16] Wikipedia, Proper acceleration, http://en.wikipedia.org/wiki/Proper_acceleration [20. 6. 2012]

[17] Wikipedia, Žiroskop, http://sl.wikipedia.org/wiki/%C5%BDiroskop [22. 6. 2012]

Univerza v Mariboru

Fakulteta za elektrotehniko, računalništvo in informatiko

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani **DAMIR ŠTUHEC**,

z vpisno številko **E1031601**,

sem avtor diplomskega dela z naslovom:

ZAJEMANJE PODATKOV GIBANJA NA NAPRAVAH iOS IN NJIHOVA UPORABA PRI RAZVOJU IGER

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom (naziv, ime in priimek)

  doc. dr. DAVID PODGORELEC

  in somentorstvom (naziv, ime in priimek)

  doc. dr. GREGOR KLAJNŠEK

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v DKUM.

V Mariboru, dne 14. 9. 2012 Podpis avtorja Damir Štuhec

Univerza v Mariboru

Fakulteta za elektrotehniko, racunalnistvo in informatiko

IZJAVA O USTREZNOSTI DIPLOMSKEGA DELA

Podpisani mentor **doc. dr. DAVID PODGORELEC** izjavljam, da je

(ime in priimek mentorja)

študent **DAMIR ŠTUHEC** izdelal diplomsko

(ime in priimek študenta)

delo z naslovom: **ZAJEMANJE PODATKOV GIBANJA NA NAPRAVAH iOS IN NJIHOVA UPORABA PRI RAZVOJU IGER**

(naslov diplomskega dela)

v skladu z odobreno temo diplomskega dela, Navodili o pripravi diplomskega dela in mojimi navodili.

Datum in kraj: 14. 9. 2012, MARIBOR

Podpis mentorja:

Univerza v Mariboru

Fakulteta za elektrotehniko, računalništvo in informatiko

IZJAVA O ISTOVETNOSTI TISKANE IN ELEKTRONSKE VERZIJE ZAKLJUČNEGA DELA IN OBJAVI OSEBNIH PODATKOV DIPLOMANTOV

Ime in priimek diplomanta: **DAMIR ŠTUHEC**

Vpisna številka: **E1031601**

Študijski program: **RAČUNALNIŠTVO IN INFORMACIJSKE TEHNOLOGIJE**

Naslov diplomskega dela:

ZAJEMANJE PODATKOV GIBANJA NA NAPRAVAH iOS IN NJIHOVA UPORABA PRI RAZVOJU IGER

Mentor: **doc. dr. DAVID PODGORELEC**

Somentor: **doc. dr. GREGOR KLAJNŠEK**

Podpisani-a **DAMIR ŠTUHEC** izjavljam, da sem za potrebe arhiviranja oddal elektronsko verzijo zaključnega dela v Digitalno knjižnico Univerze v Mariboru. Diplomsko delo sem izdelal sam-a ob pomoči mentorja. V skladu s 1. odstavkom 21. člena Zakona o avtorskih in sorodnih pravicah dovoljujem, da se zgoraj navedeno zaključno delo objavi na portalu Digitalne knjižnice Univerze v Mariboru.

Tiskana verzija diplomskega dela je istovetna elektronski verziji, ki sem jo oddal za objavo v Digitalno knjižnico Univerze v Mariboru.

Podpisani izjavljam, da dovoljujem objavo osebnih podatkov vezanih na zaključek študija (ime, priimek, leto in kraj rojstva, datum diplomiranja, naslov diplomskega dela) na spletnih straneh in v publikacijah UM.

Datum in kraj: 14. 9. 2012, Maribor

Podpis diplomanta: Damir Štuhec