NaoMarkTracker

Ember-robot interfész házi feladat

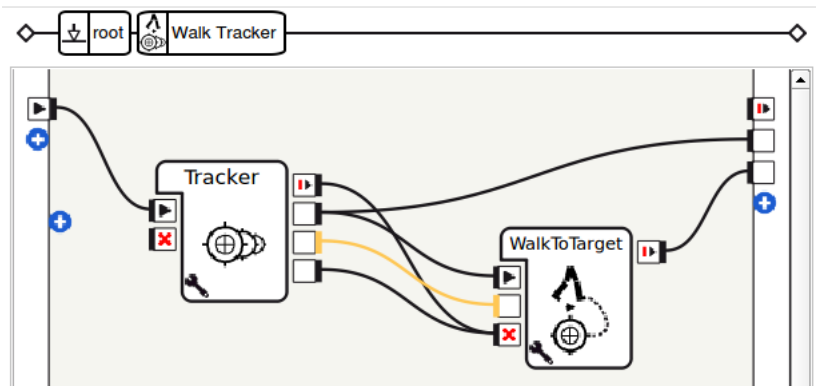
Antal János Benjamin
G9PTHG
antal.janos.benjamin@gmail.com

2016-12-09

Tartalomjegyzék

1. Motiváció

A félév során néhány esemény hatására különböző kis jeleneteket megvalósító programot terveztem készíteni házi feladatként. A feladatom készítése során az egyik jelenet megvalósításához szükség volt egy olyan modulra, amely a *FaceTracker*höz és a *RedBallTracker*hez hasonló funkcionalitással rendelkezik, csak arc és piros labda helyett NaoMarkokkal dolgozik, hogy a robot meg tudjon közelíteni és követni egyes NaoMarkokkal megjelelölt tárgyakat vagy személyeket. Sajnos a Choreography nem rendelkezett ilyen modullal, és az interneten sem sikerült megoldást találnom a problémára. Továbbá a szívesebben fejlesztek szöveges programozási környezetben, mint grafikusban, mivel közelebb áll a szakmai tudásomhoz Ezen okok miatt, amikor Zainkó Csaba felvetette, hogy akár ezt a modult meg is lehet írni C++-ban, akkor megváltoztattam a saját magam által kitúzött célt a házi feladat kapcsán.



1.1. ábra. *Walk Tracker* belső felépítése

Az így kialakult elképzelés alapján a fentebb említett két modul funkcionalitásának egy részét megvalósító modult terveztem készíteni NaoMarkok követésére. A hozzájuk tartozó dokumentáció áttanulmányozása után, véleményem szerint a teljes funkcionalitást megvalósító modul elkészítése túlmutat a tárgy keretein belül, így csak az adott feladathoz feltétlenül szükséges funkcionalitást valósítom meg. Természetesen a munkám során törekedtem az érthető és egyszerűen továbbfejleszthető modul készítésére, hogy a jövőben lehetőség legyen a kiegészítésre.

2. Specifikáció

2.1. Megvalósítandó funkcionalitás

Az 1.1.ábrán látható a *Walk Tracker* belső felépítése, mely két részre oszlik: *Tracker* és *WalkToTarget*. Előbbi felelős azért, hogy a kamera képe alapján meghatározza piros labda vagy az arc koordinátáit, majd a sárgán jelölt kapcsolat

mentén ezt az információt átadja a *WalkToTarget*nek, amely megközelíti a megadott paramétereknek megfelelően. A *Tracker* beállításainál kiválaszthatjuk a követendő objektumot (piros labda vagy egy arc), így az a választott értéknek megfelelő modult használja a belső működése során. A *Tracker* python nyelvű forráskódja megvizsgálása után az alábbi függvényekre szűkítettem a szükséges függvényeket:

- *void startTracker()*: utasítja a trackert, hogy iratkozzon fel a LandmarkDetected eseményre, ezzel lehet elindítani a követést.
- *void stopTracker()*: utasítja a a trackert, hogy iratkozzon le a LandmarkDetected eseményre, ezzel lehet leállítani a követést.
- *std::vector<float>getPosition()*: a követendő NaoMark koordinátáit adja vissza a robotközpontú koordináta-rendszerben.[1]
- *bool isActive()*: igazat ad vissza, ha éppen fut a követés.
- *bool isNewData()*: igazat ad vissza, a legutóbbi getPosition hívás óta frissültek a NaoMark helyét leíró koordináták.
- *void setLandmarkRadius(float radius)*: a követni kívánt NaoMark sugarának beállítása, a pozíció meghatározása a NaoMark méretén alapul, alapértelmezett 0.05 méter.
- *void setLandmarkId(int markId)*: a követni kívánt NaoMark ID-jának megadása. Ha 0-t állítunk be, akkor az észlelt NaoMarkok közül az elsőt fogja követni, egyéb esetben csak a kívánt ID-val rendelkezőt.

Feladatom volt tehát az említett függvényeket megvalósító modul készítése C++ nyelven, amely így lehetőséget teremt a NaoMarkok követésére az archoz és piros labdához hasonlóan.

2.2. Felhasznált funkciók, információk

A megvalósítás során felhasználtam a *ALLandMarkDetection* modult. Ez a modul generálja a *LandmarkDetected* eseményt, amely azt jelzi, hogy a robot talált egy NaoMarkot. Ezután a *ALMemoryProxy* használatával megkaphatóak a következő információk:

- a megtalálás időbélyege
- a megtalált NaoMarkok mindegyikéről a következő információ:
  - a NaoMark helyzete és mérete kameraszögekben
  - a NaoMark azonosító, ha van
- a kamera koordinátái a robotközpontú koordináta-rendszerben
- a kamerát tartalmazó fej elforgatását a három koordináta tengely körül

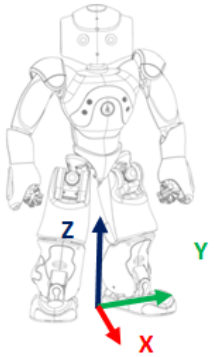
Ezen információk a NaoMark méretével kiegészítve elegendőek ahhoz, hogy meghatározzuk a NaoMark pozícióját a robot koordináta rendszerében.

[1] Pontosabban a FRAME_TORSO nevű térben, lásd bővebben

3. Működés

A modul funkciói két részre különíthetőek el:

- **LandmarkDetected esemény feldolgozása**: az ALLandMarkDetection modul által generált esemény adatainak tárolása
- **NaoMark pozíciójának meghatározása**: az eltárolt információk alapján a NaoMark pozíciójának meghatározása a 2.1.ábrán mutatott koordináta rendszerben.



2.1. ábra. A robot koordináta rendszere

3.1. LandmarkDetected esemény feldolgozása

A *LandmarkDetected* eseményt a *ALLandmarkDetection* modul generálja, ha van az eseményre feliratkozott modul. A feliratkozás az *ALMemoryProxy::subscribeToEvent(eventName, callbackModule, callbackMethod)* három string típusú paraméterrel rendelkező függvény meghívásával lehetséges. A feliratkozás automatikusan elindítja a NaoMark detektciót. Ha detektál egy NaoMarkot, akkor meghívja a *callbackModule* modul *callbackMethod* paraméter nélküli függvényét, ezzel jelezve, hogy a detekció adatai az *ALMemoryProxy "landmarkDetected"* kulcshoz tartozó adatblokkba került. Ezt az *ALMemoryPoxy::getData(key)* függvény segítségével tudjuk kiolvasni. A feldolgozásnak gyorsnak kell lennie (<300ms), így az értéket csak eltároljuk a *detectInfo* tagávltozóban.

Az *isNewData* működése során a *detectInfo*ban keresi a megfelelő azonosítójú NaoMarkot. Ha talál egyet, akkor azt a *lastDetectedMarkInfo*ban eltárolja. Ez azért szükséges, mert ha egy új NaoMark detekció során nem detektálja az adott azonosítójú NaoMarkot, akkor az *isNewData* igaz visszatérési értéke esetén a *getPosition* meghívása hibás működést eredményezne. A *lastDetectedMarkInfo* megváltozásakor a *lastDetectedCameraPos* változóban az adott észlelés pillanatában a kamera pozícióját leíró *Position6D* struktúra és eltárolásra kerül. Ez a későbbiekben a NaoMark pozíciójának pontosabb meghatározásához szükséges.

3.2. NaoMark pozíciójának meghatározása

A NaoMark pozíciójának meghatározása homogén transzformációs mátrixok segítségével történik. A három transzformációs mátrix az alábbi:

- **robotToCamera**: A kamera pozíciója alapján felépülő mátrix, mely azért felelős, hogy a NaoMark pozícióját kamera koordináta rendszeréből a robot a 2.1.ábrán ismertetett koordináta rendszerébe transzformálja.
- **cameraToLandmarkRotationTransform**: A NaoMark kameraképen lévő pozícióját a vízszintes és függőleges tengely körüli forgatási szögekkel

leírva kapjuk meg. Ez a mátrix azért felőlős, hogy a kamera koordináta rendszerében a megfelelő irányba állítsa a NaoMark pozícióját.

- **cameraToLandmarkTranslationTransform**: A NaoMark kamerától történő eltolásáért felelős mátrix. Feladata az, hogy a NaoMark pozíciója a valóságnak megfelelő távolságra legyen a kamerától, a kamera koordináta rendszerének közepén.A távolság meghatározása a NaoMark méretén alapszik, így ha az eltér az alapértelmezett 6,5 cm-től, akkor mindenképpen be kell állítani a megfelelő működés érdekében.Mivel a távolság meghatározása a NaoMark méretén alapszik, így nem nagy pontosságú. A megfigyeléseim alapján a közel (20-40 cm) lévő NaoMarkok esetén

A távolság meghatározása a NaoMark méretén alapszik, így ha az eltér az alapértelmezett 6,5 cm-től, akkor mindenképpen be kell állítani a megfelelő működés érdekében.Mivel a távolság meghatározása a NaoMark méretén alapszik, így nem nagy pontosságú. A megfigyeléseim alapján a közel (20-40 cm) lévő NaoMarkok esetén a távolsághoz mérten jelentős (5-10, akár 12 cm is lehet) a pontatlanság, míg nagyobb (16cm átmérőjű) és messzebb (4m) lévő NaoMarkok esetén a távolsághoz képest nem jelentős (10-15cm) a pontatlanság mértéke.

4. Megvalósítás

4.1. Környezet felállítása, buildelés

A modul készítése során a következő programokat használtam:

- qibuild 1.14.1
- Nao Geode cross toolchain (linux x64) 1.14.5

A továbbiakban feltételezem ezen eszközök meglétét, illetve hogy a qibuild-et a parancssorban "qibuild"-ként lehet futtatni, illetve a cross toolchain elérési útja "/path/to/geoderoot", valamint az általam készített trackermodule.zip elérési útja a "/path/to/trackermodule.zip". Ezután az alábbi lépések szükségesek a modul buildeléséhez:

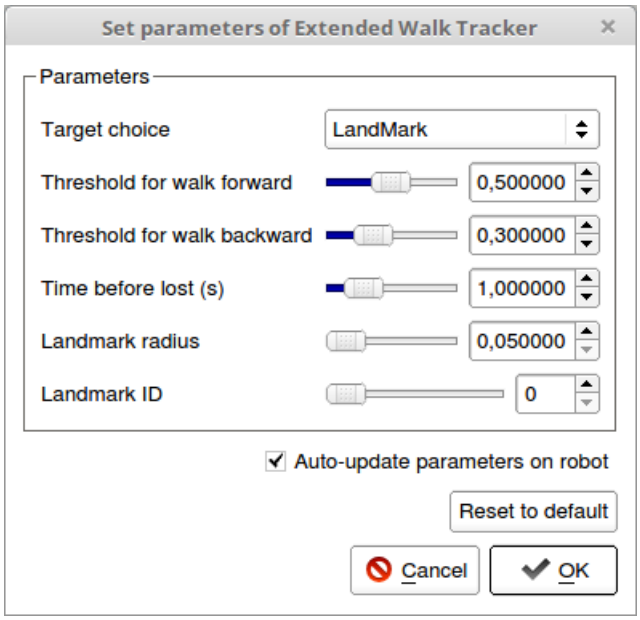
1. `mkdir worktree; cd worktree`
2. `qibuild init`
3. `qitoolchain create cross-atom /path/to/geoderoot/toolchain.xml`
4. `unzip /path/to/trackermodule.zip -d .; cd trackermodule`
5. `qibuild configure -c cross-atom`
6. `qibuild make -c cross-atom`

Ezután a "worktree/trackermodule/build-cross-atom/sdk/lib/naoqi" könyvtárban megjelenik a *liblandmarktracker.so* fájl, mely az elkészült modult tartalmazza. A fájlt felmásolva a robotra és a az elérési útját beírva a /home/nao/naoqi/preferences/autoload.ini fájlba, a modul elérhetővé válik a következő indítás után.

4.2. Használat Choreography-ban

Ha a modul elérhető a roboton, akkor lehetőség nyílik Choreographyn keresztül elérni és használni. A házi feladatom során az 1.1.ábrán bemutatott *WalkTracker* funkcionalitását egészítettem ki az elkészített modullal.

Első lépésként a *Walk Tracker* konfigurációs paramétereit bővítettem a 4.1.ábrán látható *Landmark radius* és *Landmark ID* paraméterekkel. Ezek a fentebb már említett paraméterek, azaz a NaoMark mérete és a NaoMark azonosítója.



4.1. ábra. A kiegészített *Extended Walk Tracker* konfigurációs ablaka

Következő lépésként a *Walk Tracker*-ben található *Tracker* doboz pyhton kódját módosítottam úgy, hogy a *Face Tracker* és *Red Ball Tracker* mellett a *LandmarkTracker*-t is használja, a megfelelő beállításokkal. A megváltoztatott *if-else* szerkezet az alábbiakban látható:

```
if (self.TargetName == "Red Ball"):
    self.trackerProxy = ALProxy("ALRedBallTracker")
elif (self.TargetName == "LandMark"):
    self.trackerProxy = ALProxy("LandMarkTracker")
    self.trackerProxy.setLandmarkId(self.getParameter("Landmark ID"))
    self.trackerProxy.setLandmarkRadius(self.getParameter("Landmark radius"))
else:
    self.trackerProxy = ALProxy("ALFaceTracker")
```

5. Összefoglalás

5.1. Továbbfejlesztési lehetőségek

Az elkészített modul használható, azonban nem tökéletes. A fejlesztés késői fázisában, valamint a tesztelés során felmerült továbbfejlesztési lehetőségek:

- **setWholeBodyOn**: a függvény megvalósítása jelenleg semmit nem csinál. Igény szerint pontosítani lehetne a függvény funkcionalitását és ez alapján elkészíteni egy implementációját a modul egyéb függvényeinek kiegészítésével együtt.
- **A NaoMark és a kamera tengelye által bezárt szög**: az aktuális implementáció nem veszi figyelembe azt, hogy a NaoMark nem mindig merőleges a kamera előremutató tengelyére. Ez akkor okozhat hibát, ha a NaoMark nem teljesen szemben van a kamerával, hanem a saját függőleges tengelye körül kissé elforgatva, mivel a távolság meghatározása a NaoMark szélességén alapszik.

5.2. Értékelés

A munkám során megismerkedtem a NaoQI C++ API-jával, azon belül komolyabb ismereteket szereztem a *LandmarkDetection*, *ALMotionProxy*, *ALMemoryProxy* modulokról és a robot által használt homogén koordinátás transzformációkról, valamint a robot belső eseménykezeléséről. Ezen információk alapján készítettem el a C++ nyelvű modult, amely a hiányosságai ellenére a gyakorlatban jól használható. A pontossága egy nagyságrendben van a két beépített hasonló modul pontatlanságával. A hiányosságok abból erednek, hogy a tárgy erőforrásai végesek, valamint egyéb egyetemi tantárgyak miatt nem tudtam a hiánytalan modulhoz szükséges időt (ami az elvárt ráfordítás többszöröse véleményem szerint) fordítani a házi feladatomra. A tesztelés és kipróbálás során a robot meg tudta becsülni a NaoMark pozícióját, és ez alapján a szükséges távolságot tudta tartani a NaoMarkhoz képest. Ezen eredményekre alapozva véleményem szerint teljesítettem a házi feladatban kitűzött célokat.