TANSZÉKVEZETŐ

SZAKDOLGOZAT FELADAT

Czurkó Dániel
szigorló mérnökinformatikus hallgató részére

Objektumok felismerése és követése felhőből vezérelt drónokkal

A drónok és a hozzá kapcsolódó szoftverek rohamos fejlődésével ezen légijárművek felhasználása egyre inkább elterjedté vált az élet számos területén. Egy még kevéssé elterjedt alkalmazása a drónoknak, mikor beltéren repülnek és ott végeznek el ellenőrző illetve szállító feladatokat. Egy épületen belül pontos navigálásra és nagyon gyors reagálásra van szükség amire a drón fedélzeti számítógépe nem feltétlenül elegendő. E probléma megoldására, a hálózatok alacsony késleltetését kihasználva, a számításokat kiszervezik egy nagy teljesítményű számítógépbe, a felhőbe. Így gyorsan és pontosan lehet komplex feladatokat elvégezni a drónok segítségével.

A hallgató feladata, egy olyan szoftver komponens elkészítése, mely egy drón kamerájának a képén objektumokat ismer fel, majd egy kitüntetettet objektumot követni képes. Az objektum követése a drón mozgatásával történik, a megfelelő vezérlő üzenetek elküldésével. A követés során fontos a gyors reakció, ellenkező esetben az objektum eltűnhet a drón kamerájáról.

A hallgató feladatának a következőkre kell kiterjednie:

- Elemezzen képfeldolgozási módszereket, amelyek objektumokat ismernek fel videostreamen és vizsgálja a szükséges számítási teljesítményeket! Vizsgálja meg, milyen módon valósítható meg mozgó objektumok folyamatos követése, amennyiben a megfigyelő pont mozoghat!
- Tervezze meg az objektum felismerő és követő szoftver komponenst a tanszéken rendelkezésre álló szimulált drón környezethez! A komponens feladata, hogy a képi információk alapján kiadja a drón mozgatásához szükséges parancsokat!
- Implementálja a megtervezett komponenseket! A funkciókat a tanszék által biztosított felhő környezetben kell futtatni!
- Szimuláció segítségével végezzen méréseket a megoldás pontosságára és gyorsaságára való tekintettel! Értékelje a megvalósított funkciót és tegyen javaslatot továbbfejlesztési lehetőségekre!
- Eredményeit részletesen dokumentálja!

Tanszéki konzulens: Dr. Fehér Gábor, egyetemi docens

Budapest, 2020. október 9.

/ Dr. Magyar Gábor /
tanszékvezető

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

1117 Budapest Magyar tudósok krt. 2.
Telefon: +36 1 463 2448 Fax: +36 1 463 3107
E-mail: magyar@tmit.bme.hu http://www.tmit.bme.hu

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Czurkó Dániel

Objektumok felismerése és követése felhőből vezérelt drónokkal

KONZULENS
Dr. Fehér Gábor

BUDAPEST, 2021

Abstract

A drónok alkalmazásánál egy gyakran előkerülő megoldandó probléma, hogy a drónoknak követnie kell valamit vagy valakit. Légi felvételeknél gyakran egy mozgó embert, például egy biciklist, windsurf-öst vagy pedig egy síelőt kell követnie. A csomagszállításnál is számos olyan szituáció adódhat elő amikor valamilyen objektumot kell követnie például mikor egy előre meghatározott objektumra kell leszállnia.

Robotok és drónok építése során egy fontos kérdés, hogy mekkora számítási kapacitást tervezzünk a robotunkra. Természetesen azt szeretnénk ha minél több mindenre képes lenne a robotunk ehhez azonban nagy számítási kapacitást kellene a robotnak magával cipelnie, ami sokszor nem megoldható. Ugyanakkor ha rendelkezésünkre áll egy nagyon alacsony késleltetésű hálózat, például egy 5G-s mobilhálózat és a hálózat szélén egy nagy számítási kapacitású számítógép, a felhő akkor meg lehet oldani, hogy szinte végtelen számítási kapacitás álljon a robotunk rendelkezésére. Ennek megoldása során a robot irányításához szükséges komplex és nagy számítási kapacitású vezérlési logikát kiszervezzük a hálózat szélén elhelyezkedő nagy számítási kapacitású számítógépbe.

A szakdolgozat során a feladatom egy felhőben futó objektum követő rendszer elkészítése, mely képes egy szimulált drón kamera kép alapján történő irányítására oly módon, hogy a drón kövessen egy előre meghatározott mozgó objektumot.

A feladat megoldása során mozgó objektumnak egy járkáló embert választottam. A rendszer tervezése és megvalósítása oly módon történt, hogy a szimulált drónt egyszerűen ki lehet cserélni egy valósságos drónra, valamint nem csak mozgó ember felismerésére és követésére alkalmas, hanem több mint ötven másik mozgó objektumot is képes követni.

Tartalomjegyzék

1 Bevezetés

1.1 Téma bemutatása

1.1.1 Drónok objektumkövetésének jelentősége

A drónok, vagyis pilóta nélküli repülőgépek (angolul *Unmanned Aerial Vehicle*, UAV) eredetileg a hadiiparban terjedtek el, ahol olyan feladatokra is be lehetett vetni őket ami emberek számára túl veszélyes. Napjainkra viszont az élet számos területén elterjedtek mint például: a légi fotózás, a csomagszállítás vagy a mezőgazdaság. Nehezen megközelíthető helyszínek feltérképezésében, vészhelyzetek kezelésében, valamint katasztrófasújtott területeken is be lehet vetni gyorsaságuk és kompaktságuk következtében. Fotózásnál olyan felvételeket készíthetünk, amelyek eddig csak helikopterről vagy esetleg még onnan sem voltak lehetségesek, ezzel számos területet, látnivalót teljesen új szemszögből tekinthetünk meg. Másrészről egy veszélyes területre sokkal egyszerűbb egy szenzorokkal felszerelt drónt beküldeni, hogy adatokat gyűjtsön és feltérképezze a helyszínt, mint emberekkel elvégezni ugyanezt, akik esetleg az életüket teszik kockára ezért. Gyáron belül is lehetne használni drónokat, akik megfigyelik, hogy nincs-e valahol valami probléma, minden megfelelően működik-e valamint segíthetnek a különböző csomagok, alkatrészek pakolásában is.

A drónok alkalmazásánál egy gyakran előkerülő megoldandó probléma, hogy a drónoknak követnie kell valamit vagy valakit. Légi felvételeknél gyakran egy mozgó embert, például egy biciklist, windsurf-öst vagy pedig egy síelőt kell követnie. A csomagszállításnál is számos olyan szituáció adódhat elő amikor valamilyen objektumot kell követnie például mikor egy előre meghatározott objektumra kell leszállnia. Olyan szituáció is előfordulhat, hogy amikor a drónok rajban repülnek egymást kell követniük. Így az objektumok követése egy fontos részét képzi a drónok működésének.

Fejlesztés során számos probléma adódhat egy igazi drónnal. Ha nem vagyunk elég körültekintőek összetörhetjük akár a drónt, akár a környezetét, sőt saját magunkban is kárt tudunk okozni egy drónnal. Ennek a problémának egy egyszerű megoldása, ha a fejlesztés elején egy megfelelős szimulációt használunk és nem az igazi drónt. Így semmi probléma nem történik, ha esetleg nem úgy történnek az események ahogy elképzeltük. Egyszerűen újraindítjuk a szimulációt és már próbálhatjuk is újra. A szimulációban számos szituációt kipróbálhatunk és olyan bonyolult és összetett világot építhetünk, amilyet szeretnénk, ami adott esetben a valóságban csak nagyon nehezen lehetne létrehozni.

Az élvezhető és gyors szimulációhoz, szükség van egy olyan erős számítógépre ami a szimulációhoz szükséges számításokat megfelelően gyorsan el tudja végezni. Általában egy egyszerű laptop vagy asztali számítógép nem rendelkezik elég számítási kapacitással ehhez. Napjainkban egyre elterjedté vált a felhő használata vagyis számos feladatot és számítást kiszervezünk egy mindenki által elérhető adatközpontba. Ennek mintájára a szimulációhoz szükséges számításokat kiszervezhetjük egy távoli nagy számítási kapacitású számítógépbe és a saját számítógépünkön csak a szimuláció grafikus végeredménye jelenik meg.

Ennek megfelelően a drón objektum követő alkalmazásához egy távoli, az én esetemben az egyetemen futó szimulációt fogok használni.

1.1.2 Robotépítés nehézségei

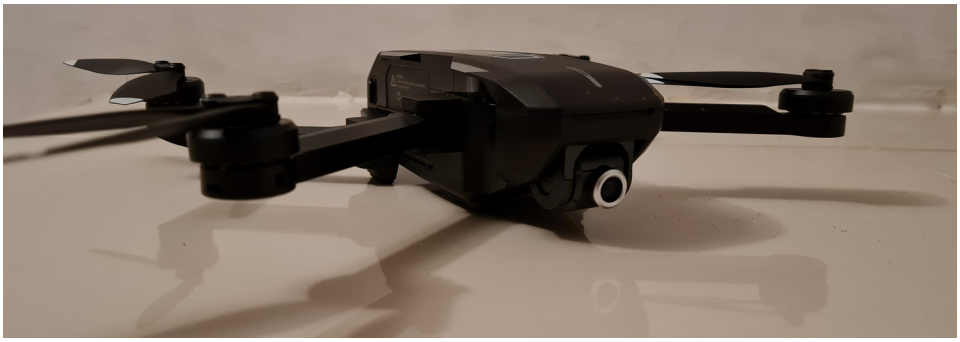
Robotok építése során egy fontos kérdés, hogy mekkora számítási kapacitást tervezzünk a robotunkra. Természetesen azt szeretnénk ha minél több mindenre képes lenne a robotunk ehhez azonban nagy számítási kapacitást kellene a robotnak magával cipelnie, ami sokszor nem megoldható. Ugyanakkor ha rendelkezésünkre áll egy nagyon alacsony késleltetésű hálózat, akkor meg lehet oldani, hogy szinte végtelen számítási kapacitás álljon a robotunk rendelkezésére. Ehhez szükség van, hogy a nagy számítási kapacitást a hálózat széléhez tudjuk telepíteni, fizikailag közel a robotunkhoz. Így, a robot irányításához szükséges komplex és nagy számítási kapacitású vezérlési logikát ki tudjuk szervezni a hálózat szélén elhelyezkedő szerverbe. Ennek hatására a robotunk leegyszerűsödik, mivel tulajdonképpen két feladata lesz. Az első, hogy a különböző szenzoraiból és kameráiból érkező adatokat el kell küldenie a hálózat szélén elhelyezkedő szervernek, a második pedig, hogy a szervertől érkező utasításokat végrehajtsa. Így a robotunkba kevesebb bonyolult számításokat végző alkatrészt kell beépíteni és egyszerűbb és kompaktabb lesz. Miután kevesebb számítást kell végeznie, tovább képes működni az akkumulátoráról. Ez a robot akár egy drón is lehet, aki elküldi a szenzorainak és kameráinak az adatát és várja az utasításokat, hogy melyik irányba repüljön. Ez az elrendezés látható az alábbi 1.1. ábran is. Itt az alacsony késleltetésű hálózatot egy 5G-s hálózat reprezentálja.

1.1. ábra: A drón az 5G hálózaton keresztül elküldi az adatokat és várja az irányító parancsokat

Ez az architektúra lehetővé teszi, hogy az alacsony szintű, mint a rotorok vezérlése és a magas szintű, mint, hogy kövessen egy előre meghatározott objektumot, vezérlési logikát, egymástól szétválasszuk. Az így létrehozott architektúra sokkal kevésbé kötött és könnyebben továbbfejleszthető és egyszerűen kiegészíthető új funkciókkal. Ha egy egységes interfészt tudunk biztosítani a drón és a hálózat szélén elhelyezkedő számítógép között akkor lehetővé válik, a drónt és a hozzá kapcsolódó, a távoli számítógépen futó alkalmazások egymástól teljesen külön történő fejlesztése.

A fentebb leírt architektúrát követő robotokat, drónokat, felhőből vezérelt robotoknak, drónoknak szokás hívni röviden.

Az egyetemen az alábbi 1.2 ábrán látható Yuneec Mantis Q nevű drónt használjuk a különböző felhőből vezérelt drón projektekhez:



1.2. ábra: Yuneec Mantis Q drón

A szakdolgozatom során használt szimuláció is a felhőből vezérelt drón architektúrát követi, csak egy a felhőben futó szimulált drón segítségével és a bonyolult objektum követő algoritmus is ugyanabban a felhőben fog futni.

1.1.3 Felhasználási területek

A fentebb 1.1.2 alfejezetnél bemutatott koncepció számos új és továbbfejlesztési lehetőséget nyit meg a drónok felhasználása terén. A borászatban például könnyen meg tudjuk figyelni drónok segítségével a szőlő-fürtök növekedését és bonyolult algoritmusokkal ellenőrizni tudjuk, hogy megfelelően fejlődnek-e és nem kaptak-e el valamilyen betegséget. Így, nem kell a borászoknak hetente többször az egész termést végignézni. Adott esetben a drón mintát is tudna venni, ha úgy ítéli meg, hogy erre szükség van, továbbá a permetezésben is részt tudna venni.

A mezőgazdaság más területein is a segítségünkre lehetnek a felhőből vezérelt drónok. Hatalmas szántóföldeket egyszerűbb egy drónnal végigrepülni, mint azt nekünk végignézni A drón részletesebb megfigyeléseket képes tenni és adott esetben mintát is tud venni, ha úgy ítéli meg az algoritmusa, hogy valami rendellenességet talált.

Ezeken kívül a felhőből vezérelt drón sokkal precízebb és bonyolultabb mozgásokra és hosszabb repülési időre képes, mint egy átlagos drón, így csomagok, segélyek és vakcinák szállításában is aktívabban részt tudnak majd venni.

Látható, hogy az élet számos területén segítségünkre lehetnek a ezek a felhőből vezérelt drónok, de az csak később fog kiderülni, hogy melyik területen fognak ténylegesen elterjedni.

1.2 Dolgozat felépítése

A szakdolgozatomban először az elméleti hátteret és a használt szoftver komponenseket fogom ismertetni, majd a tervezés szakaszban bemutatom a használt szimulációs környezetet, valamint hogyan tervezem kiegészíteni új komponensekkel, hogy képes legyen a szimulált drón objektumokat követni. Ezt követően a megvalósítás szakaszban részletesen ismertetem az elkészült komponenseket és ezek kommunikációját, majd az utolsó szakaszban bemutatom, hogy az elkészített komponensek futtatása során milyen tapasztalatokat szereztem, illetve mely paraméterek állításával tudtam az objektum követés hatékonyságát növelni. Végül az összefoglaló szakaszban összegzem az elkészült munkát.

2 Elméleti háttér és szoftverek bemutatása

2.1 Képfeldolgozási algoritmusok objektumok felismerésére

2.1.1 Gépi tanuló algoritmusok

A szakdolgozat során a feladatom egy előre meghatározott mozgó objektum követése volt. Ennek során a kamera által készített videófolyamot használtam, melyen objektumokat ismertem fel.

Az objektumok felismerése egy komplex feladat. Fel kell ismerni egy képen, hogy melyik pixelek tartoznak egy objektumhoz, majd ha felismertünk egy objektumot el kell dönteni, hogy milyen objektumot ismertünk fel. Nehéz felfogni a problémának a komplexitását, mivel ez a feladat nekünk, embereknek triviális. Szinte észre sem vesszük, hogy ezt csináljuk nap mint nap. Ez persze nem tökéletes hasonlat gondolhatnánk, mivel nekünk két szemünk van, így sokkal jobban látunk térben és távolság alapján is meg tudjuk határozni mi az ami egy objektumnak számít és mi az ami nem. Míg egy objektum felismerő algoritmusnak csak „egy szeme van”, így csak a különböző színek elhelyezkedése alapján van lehetősége meghatározni, hogy a képnek melyik része számít egy objektumnak.

A probléma bonyolultsága és sokszínűsége miatt érdemes gépi tanuláson alapuló algoritmusokat használni objektumok felismerésére. A gépi tanuláson alapuló algoritmusok közül a mély tanuláson[1] alapuló algoritmusokat választottam. A mély tanuló algoritmusok során egy neurális hálót használunk, hogy az algoritmus a bemenethez meghatározza a legvalószínűbb kimenetet. Különböző struktúrájú neurális hálók léteznek. A különböző struktúrákat, rétegeket különböző feladatoknál alkalmazhatjuk. A rekurrens rétegeket, idősor alapú problémákra érdemes alkalmazni ilyen a a természetes nyelvfeldolgozás Az előrecsatolt rétegeket regressziós és osztályozási feladatokra szokás használni, míg a konvolúciós rétegeket valamilyen jellegzetesség felismerésére érdemes alkalmazni, mint például amikor egy képen szeretnénk valamilyen jellegzetességet, objektumok felismerni. A különböző konvolúciós rétegeket tartalmazó neurális hálók segítségével lehet egészen pontosan felismerni, hogy milyen objektum van egy képen. Így az objektumfelismerés megvalósításához konvolúciós rétegeket tartalmazó mély tanuláson alapuló gépi tanuló algoritmusokat fogok használni.

Az objektumok felismerésére és meghatározására alkalmas gépi tanuló algoritmusokat alapvetően két csoportba sorolhatjuk. Vannak a kétlépcsős és az egylépcsős objektum felismerők. A kétlépcsősök első lépésnek csak azt próbálják meghatározni, hogy hol vannak egyáltalán objektumok a képen. Majd második lépésként ezeket az objektum javaslatokat odaadják egy konvolúciós neurális hálónak, hogy osztályozza milyen objektum van a képen. A másik oldalon az egylépcsős objektum felismerők, mindezt egy lépésben végzik el, megkapják a képet és egy lépésben felismerik az objektumokat és osztályozzák is őket.

A kétlépcsős algoritmusok általában pontosabbak, de lassabbak, míg az egylépcsős objektum felismerők gyorsabbak, de kevésbé pontosak. Így, alapvetően választanunk kell, hogy melyik a fontosabb számunkra az adott helyzetben: a pontosság vagy a sebesség. A szakdolgozatom során az objektum felismerést arra használom, hogy videó folyamon ismerjek fel objektumokat és azokat kövessem. Itt fontos a sebesség, mivel a képkockák gyorsan jönnek egymás után és végeznem kell egy képkocka feldolgozásával míg megérkezik a következő. Így, ebben az esetben a felismerés sebességének fontosabb szerepe van mint a pontosságnak. Ha esetleg egy képkockán nem ismert fel valamit az objektum felismerő, a következő képkockán már kicsit másképpen helyezkednek el az objektumok és így már képes felismerni azt az objektumot, amit az előző képkockán nem sikerült. A kétlépcsős objektum felismerőknek egy másik előnye a pontosságon kívül, hogy külön lehet fejleszteni az objektum meghatározó részét, illetve az objektum osztályozó részét melynek feladata a kép objektumnak jelölt részeinek az osztályozása.

A gépi tanuláson alapuló objektum felismerő algoritmusok körében három elterjedt megoldás ismert:

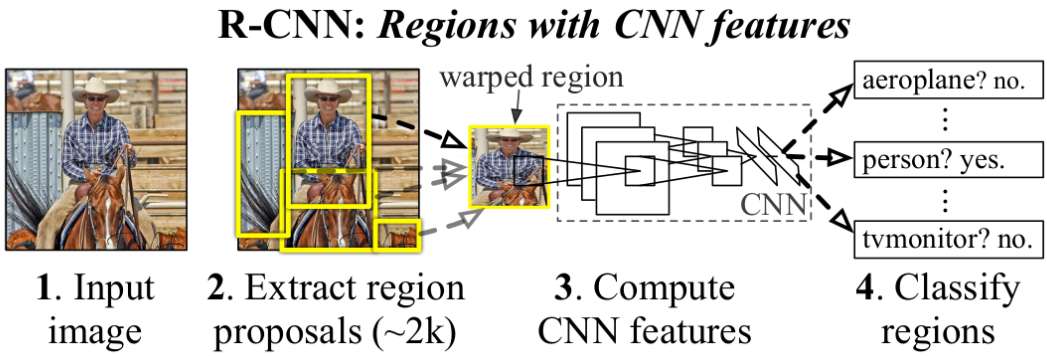
- R-CNN (Region proposal Convolutional Neural Network)
- YOLO (You Only Look Once)
- SSD (Single Shot multiBox Detector)

A következő fejezetekben ezeket fogom bemutatni.

2.1.2 R-CNN

Az R-CNN [2], melyben az „R” a Region Proposal vagyis régió javaslat szavakra utal egy kétlépcsős objektumfelismerő algoritmus, melyet 2014-ben mutatták be Ross Girshick, Jeff Donahue, Trevor Darrell és Jitendra Malik. Az R-CNN első lépésnek egy selective search [3] nevű algoritmus segítségével jelöli ki azokat a régiókat egy képen, amelyek feltehetően tartalmaznak valamilyen objektumot, majd ezeket a régió javaslatokat odaadja egy konvolúciós neurális hálónak, hogy meghatározza, osztályozza milyen objektum van az adott régióban. Később 2015 áprilisában megjelent az R-CNN egy gyorsított verziója a Fast R-CNN [4] majd 2015 júniusában megjelent a Faster R-CNN [5] mely egy Region Proposal Network (RPN), konvolúciós neurális hálót használ a selective search algoritmus helyett, a kép azon régióinak a meghatározására ami egy objektumot tartalmazhat és így gyorsabb lesz egy kép feldolgozása mint a Fast R-CNN esetén.

Az alábbi 2.1 áran látható az R-CNN felépítése. Látható, hogy első lépésnek felismeri az objektumokat, majd második lépésnek konvolúciós és előre csatolt rétegeket tartalmazó neurális háló segítségével meghatározza, hogy mi lehet az a felismert objektum.

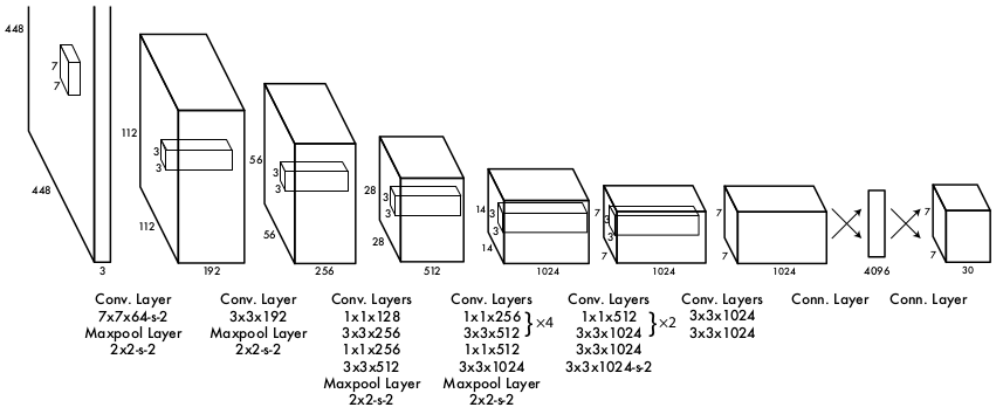


2.1. ábra: R-CNN felépítése. [2]

2.1.3 YOLO

A YOLO a „You Only Look Once”, mint „csak egyszer nézed meg”, szavak kezdőbetűiből áll össze. Ez az objektumfelismerő algoritmus egy egylépcsős objektumfelismerő. A YOLO első verzióját 2015-ben mutatták be Joseph Redmon, Santosh Divvala, Ross Girshick és Ali Farhadi [6]. A cikkben leírtak alapján GPU segítségével képes 45 képkockát feldolgozni másodpercenként. Ez nagyon gyorsnak számít. Később megjelentek újabb verziói a YOLO-nak. A YOLO9000 [7] (~YOLOv2), mely képes 9000 objektum kategóriában felismerni objektumokat bár nem igazán pontos, a 2018-ban bemutatott YOLOv3 [8] mely az előzőekhez képest egy nagyobb modellt használ, de pontosabb. A YOLOv3-at követően Joseph Redmon kilépett a YOLO fejlesztéséből, mivel a munkáját etikátlan célokra használták. A fejlesztést Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao vették át és 2020 áprilisában megjelent a YOLOv4 [9]

Az alábbi 2.2 ábrán láthatóak a YOLO neurális hálót felépítő különböző konvolúciós rétegek.

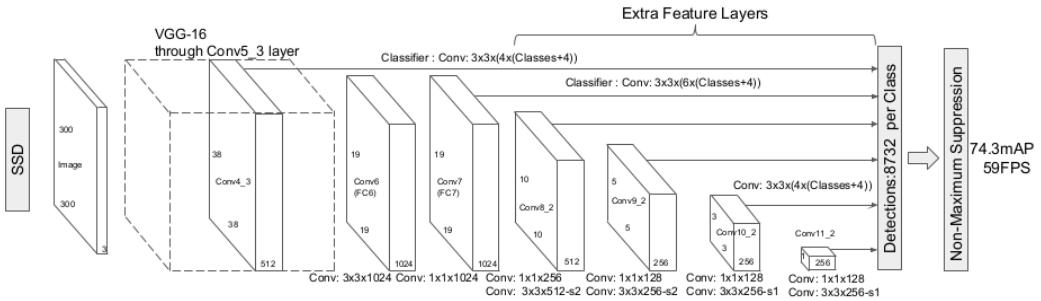


2.2. ábra: YOLO felépítése. [6]

2.1.4 SSD

Az SSD [10] a Single Shot MultiBox Detector szavakból áll össze, Single Shot mint, hogy az objektum helyzetét és osztályozását egy lépésben végzi el, tehát egy egylépcsős objektum felismerő, Multibox [11], ez a technikának a neve amelyet az objektumok helyzetének meghatározására használ, Detector, mivel ez egy olyan neurális hálózat mely objektumokat ismer fel és osztályoz. Az SSD-t 2015-ben mutatta be Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu és Alexander C. Berg.

Az alábbi 2.3 ábrán láthatóak az SSD neurális hálót felépítő különböző méretű konvolúciós rétegek.



2.3. ábra: Az SSD felépítése. [10]

2.1.5 Objektum felismerő algoritmusok összehasonlítása

Alább az 1. táblázat-ban láthatóak a Faster R-CNN, YOLOv4, valamint SSD objektum felismerő algoritmusokat gyorsaság és pontosság szerinti összehasonlító táblázat. A gyorsaságot mértékegysége az [fps], vagyis hány képkockát képes feldolgozni az objektum felismerő másodpercenként. A pontosságot az AP-nek „average precision” vagyis átlagos pontosság segítségével adom meg százalékban. Az AP, a precision-recall (precizitás-felidézés) görbe alatti területként határozható meg. A táblázatban a gyorsaság értékek alapján a Faster R-CNN igencsak lassú, míg a YOLO illetve az SSD hasonlóan gyorsak. Ha a pontosság értékeket vesszük figyelembe akkor viszont az SSD igencsak pontatlan míg a YOLO és a Faster R-CNN hasonlóan pontosak. Ezek alapján a YOLO egy ideális választás lehet.

1. táblázat: A különböző objektum felismerő algoritmusokat összehasonlító táblázat az MS COCO adathalmazon futtatva [9]

	Faster R-CNN	YOLOv4 (416x416)	SSD300
gyorsaság[fps]	9.4	38	43
pontosság	39.8%	41.2%	25.1%

2.2 Mozgó objektumok folyamatos követése

A robotoknál egy gyakran előkerülő probléma, hogy valamilyen mozgó objektumot kell követniük, például szeretnénk, hogy egy drón levideózza ahogy lesíelünk a hegyről. Felmerül a kérdés, hogy milyen lehetőségei vannak egy robotnak ehhez. Használhat a robotunk egy egyszerű kamerát [12], kamerát és még további szenzorokat[13] vagy 360 fokos kamerát az objektum követésére[14]. A 360 fokos kamera nagy előnye, hogy a robot körül elhelyezkedő tárgyak egy képkockán lesznek rajta. Beltéri környezetben a robotunk és a mozgó objektum közötti lévő távolság meghatározására akusztikus jelek is használhatóak [15] Emellett felszerelhetünk infraledeket a mozgó objektumunkra és így az infravörös fény mintázata alapján tudja a robotunk követni azt. [16]. Végül, lidar segítségével is követhet a robotunk objektumokat [17].

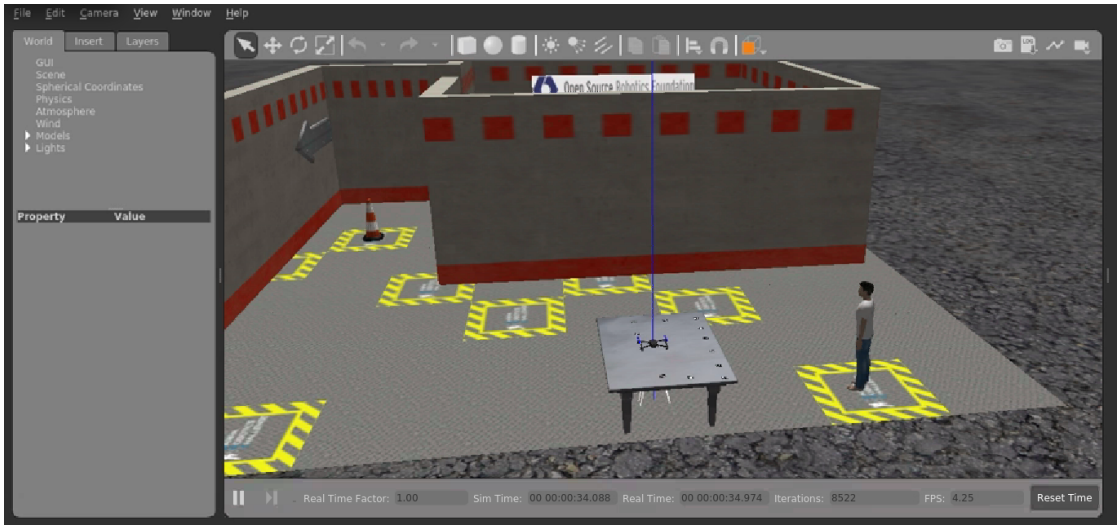
A szakdolgozatom során egyszerű kamera által készített videófolyam alapján fogok mozgó objektumokat követni.

2.3 Használt szoftverek bemutatása

2.3.1 Gazebo

A Gazebo [18] egy nyílt forráskódú projekt, melynek segítségével háromdimenziós szimulációs környezeteket tudunk készíteni, ahol különböző robotokat lehet mind kültéri, mind beltéri környezetben szimulálni. Számos már előre elkészített modell és világ áll rendelkezésünkre, amit használhatunk és kombinálhatunk a szimulált környezetünk felépítése során, de akár mi is elkészíthetünk mindent a legapróbb részletekig. A világok és a modellek leírására egy XML alapú SDF, (Simulation Description Format, mint szimuláció leíró formátum), formátumot használhatunk.

Alább a 2.4. ábra egy szimuláció kinézetét mutatja Gazebo-ban. A szimulációban egy ember, egy asztal, az asztalon pedig egy drón található. A háttérben falak láthatóak. Az asztal szélén elhelyezkedő kis fekete négyzetek jelzők, markerek, fiducial markerek pontosan, hasonlóak, mint a QR kódok csak nem valami adatot tárolnak rajta, hanem a lokalizációban van szerepe. Ezek segítenek, hogy a drón kamera kép alapján tudjon navigálni, el tudja helyezni magát egy térképen.



2.4. ábra: Gazebo szimuláció

2.3.2 ROS

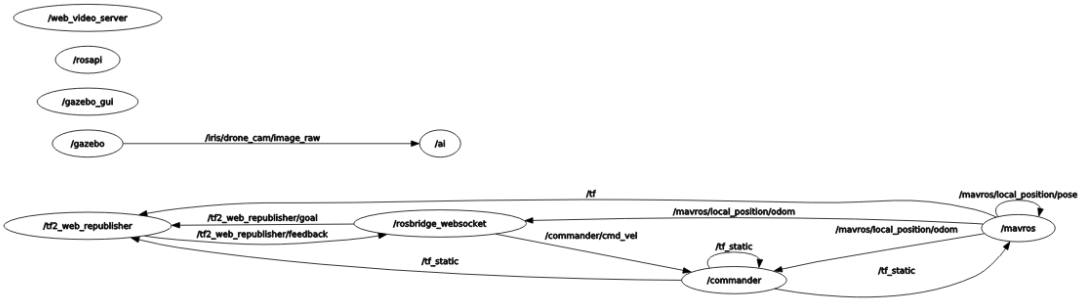
A ROS [18], neve a “Robot Operating System” kezdőbetűiből áll össze és robotok operációs rendszere lenne a magyar lefordítása, de általában csak ROS-nak szokták nevezni. A ROS egy nyílt forráskódú projekt és máig aktív fejlesztés alatt áll. A ROS-nak elérhető egy másik, újabb változata, melyet ROS2-nek hívnak. Ebben sok architekturális elemet újragondoltak. A szakdolgozat során a ROS-t használtam. A ROS legfrissebb verziója ROS Noetic néven érhető el.

A ROS tulajdonképpen egy keretrendszer, komplex programok és könyvtáraknak a gyűjteménye, melynek célja, hogy minél inkább megkönnyítse a robotok fejlesztését. Egy környezetet ad a fejlesztők kezébe, melynek segítségével egyszerűen modularizálható a robotok különböző komponenseinek a fejlesztése. Könnyen kiegészíthetjük egy már működő ROS-t használó robotunkat egy új kamera modullal, ami kapcsolódik a többi ROS komponenshez és a kamerájának a képét elérhetővé teszi más ROS komponensek számára. Egy ilyen modult, ha elkészült közzétehetünk és így más fejlesztők is tudják majd használni, ha a saját robotjukhoz egy kamerát is szeretnének szerelni. Ezek alapján nagyon sok esetben nekünk már nem kell komponenseket fejleszteni a robotunkhoz, hanem használhatjuk a mások által már korábban elkészítetteket.

A ROS alapvetően node-ból épül fel. Ezek a node-ok egy kisebb feladatot ellátó szoftverkomponensek, és különféleképpen szinkron és aszinkron módon képesek egymással kommunikálni. Egy node például irányíthatja a drónnak a kameráját és a kameraképet elérhetővé teheti más node-ok számára. Egy másik node kapcsolódhat az első node-hoz és így eléri az általa közzétett kamera képeket és ezeken valamilyen felismeréseket például objektumfelismerést végezhet és a felismert objektumok koordinátáját közzéteheti a többi node számára, Végül egy harmadik node, a felismerések alapján vezérelheti a drónnak a mozgását.

Az alábbi 2.5. ábra ROS node-ok és a köztük lévő kapcsolatokat mutatja. Látható, hogy vannak olyan node-ok, akik nem kapcsolódnak semelyik másik node-hoz,

mint a „/web_video_server” vagy a „/gazebo_gui” nevű és vannak olyan node-ok is, akik több másik node-hoz is kapcsolódnak például a „/commander” vagy a „mavros” nevű.



2.5. ábra: ROS node-ok és a köztük lévő kapcsolatok

A ROS-nak egy fontos része a roslaunch program. Ennek segítségével egyszerre több node-ot is el tudunk indítani a megfelelően paraméterekkel. A roslaunch programnak egy “launch” kiterjesztésű fájlban, XML segítségével tudjuk megadni, hogy pontosan milyen node-ok induljanak el és milyen paraméterekkel. Az 2.3.1-es alfejezetben ismertetett Gazebo-t is roslaunch segítségével szokás elindítani, így könnyen meg tudjuk adni, hogy milyen paraméterekkel induljon el, milyen világot és milyen modellt töltsön be, valamint ezen kívül specifikálhatjuk, hogy melyik ROS node-ok induljanak még el a Gazebo mellett

A szakdolgozatom során én is a roslaunch programot használtam ROS node-nak az elindításához.

2.3.3 Mavlink

A Mavlink [19] egy egyszerű, megbízható és hatékony nyílt forráskódú kommunikációs protokoll, limitált erőforrással és sávszélességgel rendelkező rendszerekhez. Segítségével drónokkal és más légi járművekkel tudunk egy egységes protokollon keresztül kommunikálni. Napjainkban a Mavlinknek a kettes verziója, a Mavlink 2, az elterjedt, ez a verzió visszafele kompatibilis az egyes verzióval. A Mavlink 2 a biztonságos kommunikációt valamint a nagyobb rugalmasságot biztosító kiegészítéseket tartalmaz.

A szakdolgozatom során egy mavros nevű ROS node-on keresztül én is Mavlink-et fogok használni a szimulált drónnal való kommunikációhoz. Így egyszerűen át tudok majd váltani, hogy nem a szimulált drónnal kommunikálok Mavlink segítségével, hanem egy olyan igazi drónnal, aki ismeri a Mavlink protokollt.

2.3.4 PX4

A PX4 [20] egy nyílt forráskódú, moduláris és könnyen konfigurálható repülést irányító szoftver. Számos eszközt biztosít a drónok vagy más repülő járművek alapvető vezérléséhez, irányításához. Segítséget ad, hogy magas szintű parancsok alapján, például induljon előre a drón, milyen alacsony szintű parancsokat kell kiadni például hogyan kell vezérelni a rotorokat, hogy elinduljon előre a drónunk, ezenkívül segítséget ad az alapvető szenzor adatokat értelmezésében is.

Lehetőség van a korábban említett Gazebo, ROS és a PX4 szoftvereket együtt használni, így tudunk olyan drónt szimulálni, amin PX4 fut és Mavlink segítségével kommunikál a már korábban említett mavros-al. A mavros-on keresztül pedig a ROS-hoz kapcsolódik a szimulált drónunk. Így ROS node-ok segítségével új funkciókkal tudjuk a drónunkat kiegészíteni például, hogy mozgó objektumokat kövessen vagy akár a már meglevő, mások által készített ROS node-okat is fel tudjuk használni.

Másik nagy előnye a PX4-nek, hogy ugyanazt a PX4 szoftvert tudjuk használni a szimulált drón esetén, mint ami egy valódi drónon fut és annak repülését segíti. Ezt a módszert, melynek során ugyanazt a PX4 szoftvert futtatjuk a valóságban és a szimulációban angolul „software in the loop (SITL)”-nak, vagyis szoftver a hurokban-nak nevezik . Így, ideális esetben, ha ugyanazokkal a paraméterekkel rendelkező drónunk van a szimulációban, mint a valóságban és természetesen mind a kettőn PX4 fut, akkor a szimulátorban futó drónunkat egyszerűen ki tudjuk cserélni egy valóságos drónra úgy, hogy a valóságban ugyanazt fogja csinálni a drónunk, mint amit hasonló helyzetben a szimulációban csinált volna. Ennek segítségével a drónunk fejlesztését igencsak meg tudjuk könnyíteni. Mivel a fejlesztés első fázisában szimulációban dolgozunk és ott könnyen sok mindent ki tudunk próbálni és semmi probléma nem lesz, ha esetleg nem úgy történnek a dolgok, mint ahogy terveztük volna. Majd mikor már úgy látjuk, hogy minden megfelelően működik a szimulációban áttérhetünk az igazi drónra és kipróbálhatjuk azzal is a programunkat és ekkor az igazi drón ugyanazt fogja csinálni, mint a szimulált. Persze lehetséges, hogy egy-két paraméteren állítani kell, hogy a valóságos környezetben is megfelelően működjön a programunk. Ezzel a módszerrel, hogy először szimulációban dolgozunk és utána a valóságban, sok drón és környezetünk összetörésétől meg tudjuk óvni magunkat.

A mozgó objektum követésének kifejlesztése során én is ezt az elvet követem és a szakdolgozat ennek az első lépése, amikor a szimulált drónon futó szoftvert fejlesztem ki. A szakdolgozatot követően kezdek majd el foglalkozni, hogy az objektumkövetést egy valóságos drónon is kipróbáljam.

A bevezetőben (1.2. ábra) említett Yuneec Mantis Q drónon is egy PX4-hez nagyon hasonló szoftver fut és így lehet vele Mavlink segítségével kommunikálni. Az objektumkövető programot majd ezen a drónon fogom kipróbálni.

2.3.5 Docker

A docker [21] egy nyílt forráskódú projekt a Linux konténerek kezeléséhez. Segítséget biztosít különböző paraméterekkel rendelkező konténerek készítéséhez, indításához, leállításához, törléséhez és hálózatba szervezéséhez. A docker alapja a docker image. Ez tulajdonképpen egy sablon, egy minta, egy váz, leírja milyen komponensekből, könyvtárakból és hogyan épüljön majd föl a konténerünk. Használhatunk egy már meglévő mások által készített docker image-t is, de készíthetünk sajátot is. Ha saját docker image-t készítünk egy Dockerfile nevű szöveges fájlba írjuk le hogyan nézzen majd ki a konténerünk és a docker build program fogja majd e fájl alapján elkészíteni nekünk a docker image-t. Egy ilyen docker image futtatott példánya a konténer.

A docker konténereket virtualizációs tulajdonságai miatt előszeretettel használják különböző applikációk futtatására egy adatközpontban, mivel így nem zavarják egymást a különböző applikációk. Másik kedvelt tulajdonsága a docker

konténereknek, hogy ha egyszer összeépítettünk egy konténert, akkor azt ugyanúgy tudjuk futtatni saját gépünkön egy adatközpontban vagy bármilyen más docker által támogatott számítógépen és ugyanúgy fog működni. Más szóval ha egyszer elkészítünk egy docker konténert azt szinte bárhol tudjuk majd használni.

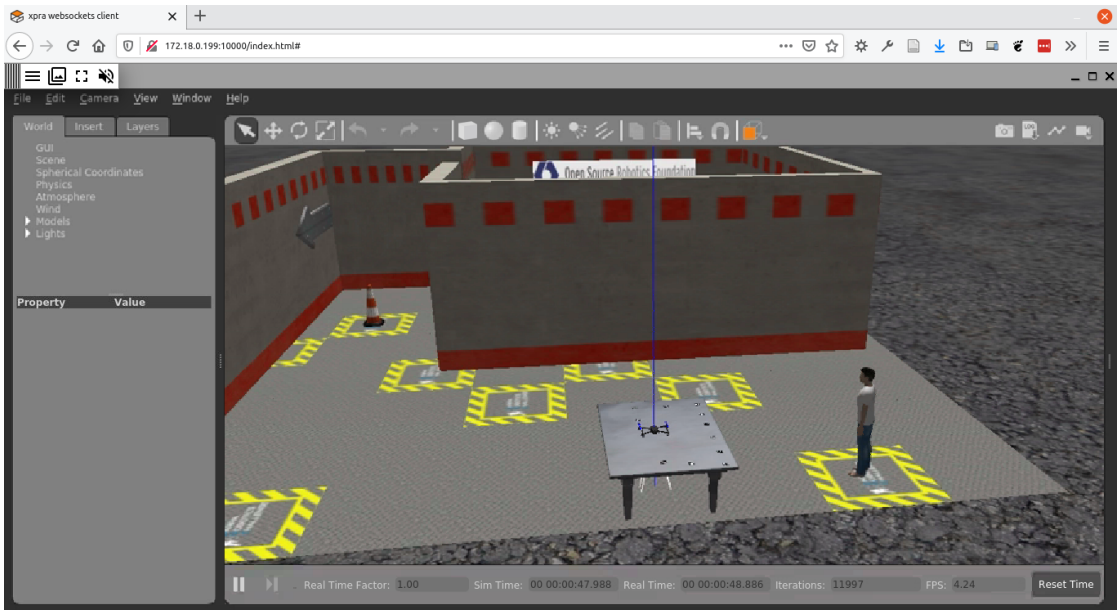
A szakdolgozatom során használt drón szimuláció docker konténerek segítségével épül fel. Így én is a szakdolgozatom során docker konténerek segítségével fogom az objektumkövetést megvalósítani.

2.3.6 Xpra

Az Xpra[22] egy nyílt forráskódú távoli display server[23], melynek segítségével a távoli számítógépen futó grafikus programokat és akár az egész asztalt képesek vagyunk a lokális számítógépünkre továbbítani.

A szakdolgozatom során ezt használtam, hogy megtekintsem a távoli számítógépen futó szimulációt. Az Xpra-t úgy állítottuk be, hogy egy böngészőbe továbbítsa a grafikus programokat így a saját gépünkre semmilyen extra programot nem kellett telepíteni a használatához.

Alább a 2.6. ábra egy szimulációt mutat, amely egy böngésző ablakban jelenik meg az Xpra segítségével.



2.6. ábra: Gazbo szimuláció egy böngészőben Xpra segítségével.

3 Szimulált drón környezet bemutatása

A drón objektum követő funkciójáért felelős komponenst egy már kész szimulációs környezethez készítettem el. Ez a szimulációs környezet felelős:

1. A drón és a környezet szimulációjáért
2. A szimulált drónnal történő kommunikációért
3. A szimulációs környezet és egy lokális számítógép közötti titkosított kapcsolat kialakításáért
4. Az egész szimuláció kapcsolatáért a ROS-al
5. a drón egyszerű irányításának lehetővé tételéről
6. A szimuláció megjelenítéséről a böngészőben
7. A valós dróntól érkező videó átalakításáért, hogy a rendszer többi tagja számára egységes legyen, és független attól, hogy a videó szimulált dróntól érkezik vagy pedig egy valóságos dróntól

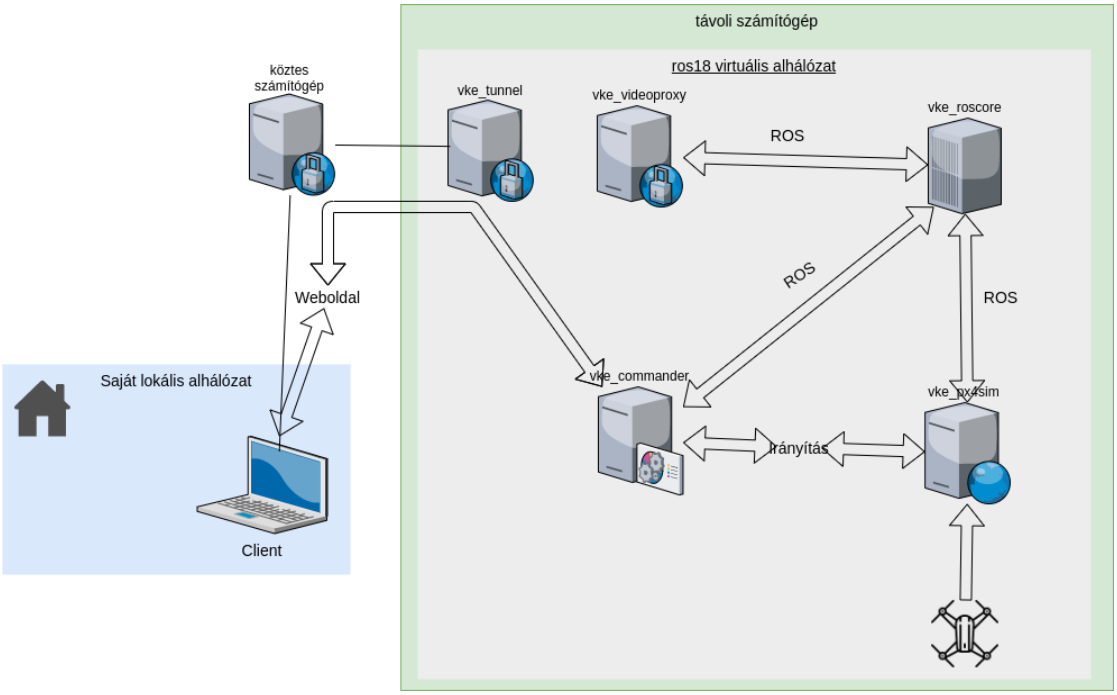
Ezen funkciók közül a harmadik és a hetedik pontot emelném ki. A harmadik pont, vagyis a szimulációs környezet és egy lokális gép közötti kapcsolat létrehozásának, a távoli számítógépen futó szolgáltatások elérésében van szerepe. Ahhoz, hogy a böngészőn keresztül meg tudjuk tekinteni a szimulációt, valahogyan hozzá kell kapcsolódnunk a távoli számítógéphez. Ehhez a távoli gépnek vagy egy publikus ip címre van szüksége vagy pedig valamilyen VPN (mint Virtual Private Network, virtuális privát hálózat) megoldással csatolhatjuk a lokális számítógépünket a távoli számítógép alhálózatába. Így egy alhálózatban lesznek és a lokális számítógépünkről egyszerűen elérhetjük a távoli számítógép szolgáltatásait. A szakdolgozat során használt számítógépnek nincs publikus ip címe csak egy másik számítógépen keresztül lehet hozzáférni az internetről. Így csak a VPN megoldás maradt és a publikus címmel rendelkező számítógépen keresztül kellett létrehozni a VPN-t a távoli számítógéppel. A VPN-es megoldás biztonságosabbnak is számít, mivel nem biztonságos közvetlenül elérhetővé tenni egy távoli számítógépet az interneten egy publikus ip cím segítségével , kitéve különböző lehetséges támadásoknak.

A hetedik pont, vagyis a valós dróntól érkező videó átalakításának, a valós drón egyszerű használatában van szerepe. Így, ha nem a szimulált drónt, hanem egy valós drónt szeretnénk használni, csak ezt a funkciót kell bekapcsolni és a rendszer többi tagja számára teljesen közömbös lesz, hogy most a valós drónnal vagy pedig a szimulálttal kommunikálnak.

A fentebb felsorolt funkciók mind docker konténerekbe becsomagolva végzik a feladatukat, így nem zavarják egymás futását és minden funkciónak így létre lehet hozni a futásához legoptimálisabb szoftver környezetet, valamint egyszerűen át tudjuk mozgatni az egész szimulációt egy másik számítógépre, ha máshol lenne szükség a futására.

Az alábbi 3.1. ábra a konténerek elhelyezkedését mutatja. Látható, hogy a szimulációban részt vevő konténerek a távoli számítógépen egy külön ros18 nevű

virtuális alhálózatban helyezkednek el. A saját gépemmel a vke_tunnel nevű konténer segítségével a köztes számítógépen keresztül tudok a ros18 virtuális alhálózatba becsatlakozni. Az általam létrehozott konténereket is a ros18 nevű virtuális alhálózatba csatoltam be.



3.1. ábra: A szimulációs környezethez tartozó konténerek elhelyezkedése

Alább látható az öt konténer, amelyek felelősek a szimulált környezet létrehozásáért:

1. vke_tunnel
2. vke_roscore
3. vke_commander
4. vke_px4sim
5. vke_videoproxy

Ezek a konténerek mind egy egységes séma alapján kerültek felépítésre. Először egy Dockerfile-ba kell leírni a konténer futásához szükséges környezet létrehozásának parancsait, valamint a működéshez szükséges könyvtárakat és fájlokat is itt kell megadni. Egy docker-entrypoint.sh fájlban kell megadni, hogy milyen parancsoknak kell lefutnia a konténer indulásakor. A compile.sh fájlba kell megadni a docker image elkészítéséhez szükséges parancsot a megfelelő paraméterekkel, például, hogy mi legyen a neve az elkészített docker image-nek. Végül a <konténer_neve>.sh nevű fájlba kell megadni a konténer elindításához, illetve leállításához szükséges parancsokat megfelelően felparaméterezve, például milyen docker image-et használjon, mi legyen az

elindított konténernek a neve, milyen hálózatokhoz csatlakozzon, fel legyen-e csatolva valamelyik mappa a konténerbe. Az általam felépített konténerek is ezt a sémát fogják követni.

A következőkben a vke_tunnel, vke_roscore, vke_commander, vke_videoproxy és vke_px4sim konténereket fogom bemutatni.

3.1 vke_tunnel

A vke_tunnel konténer feladata, hogy egy külső számítógépet becsatoljon a drón szimuláció virtuális alhálózatába VPN segítségével. A saját lokális számítógépünkön csak az openvpn programra és egy konfigurációs fájlra van szükségünk a becsatlakozáshoz. Tulajdonképpen még van egy köztes számítógép is, ami publikus ip címmel rendelkezik, de a vke_tunnel ezt elrejti előlünk. Nagy előnye, hogy a vke_tunnel-en keresztül csatlakoztatjuk a külső számítógépeket a virtuális alhálózatba, hogy ha másik számítógépre költöztetjük az egész szimulációt és ott más a hálózati architektúra, akkor csak a vke_tunnel-t kell átírni és a többi konténert nem kell megváltoztatni. Az OpenVPN-ről további információk itt olvashatóak: [24]

A VPN használatának számos előnye van. A szimulációt elérhetővé tévő számítógép és ezen kívül minden más számítógép szolgáltatásait egyszerűen elérjük a lokális számítógépünkön keresztül, mivel ip szinten egy alhálózatban vagyunk, valamint a számítógépünkön indított konténereket is be tudjuk csatolni ebbe az alhálózatba.

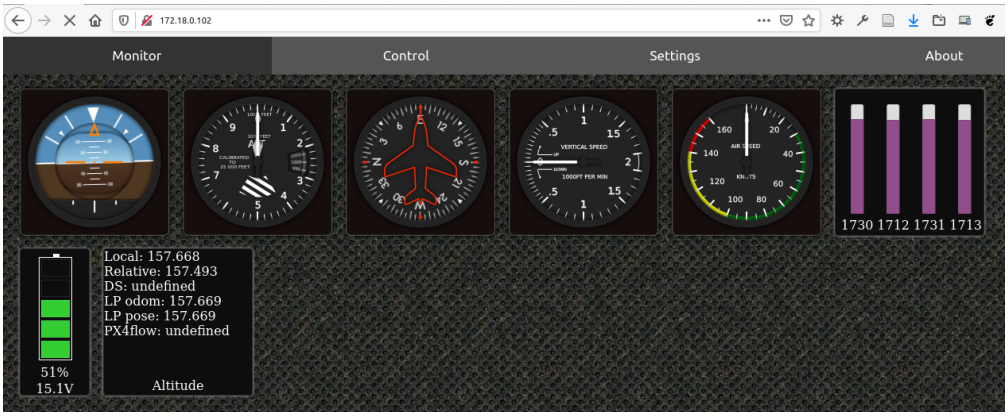
3.2 vke_roscore

A vke_roscore a ROS futásához szükséges. A ROS node-ok kommunikációjához elengedhetetlen, hogy először egy roscore nevű programot elindítsunk. A roscore fogja irányítani és segíteni a ROS node-ok kommunikációját. Az egész drón szimuláció működéséhez elég egy roscore-t elindítani. Fontos, ha nem a vke_roscore konténerben indítunk egy ROS Node-ot, hanem egy másik konténerben, akkor ott meg kell adnunk, hogy milyen ip címen fut a roscore. Ezt követően a konténerekben futó ROS node-ok maguktól megtalálják majd a roscore-t, nem kell más programot futtatnunk.

3.3 vke_commander

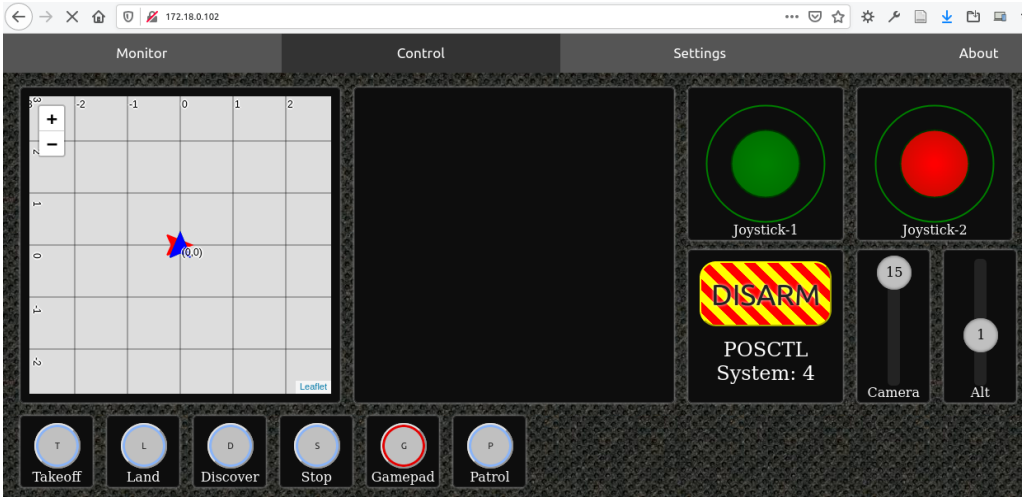
A vke_commander nevű konténer feladata, hogy kommunikáljon a drónnal. Ez a konténer különböző web alapú eszközöket kínál a drón irányítására, megjeleníti egy böngészőben a drónból érkező adatokat, valamint más konténerek számára is egyszerűvé teszi a drón irányítását.

A vke_commander, mavros (2.3.3) segítségével kommunikál a drónnal. Az alábbi 3.2. ábra mutat egy a vke_commander által biztosított weboldalt, ahol a drónból érkező különböző adatokat láthatjuk számszerűen és vizuális ábrákkal megjelenítve. Ennek a weboldalnak nagy előnye volt, hogy ha változnak a weboldalon a számok, akkor tudom, hogy rendben hozzákapcsolódtam a szimulált drónhoz.



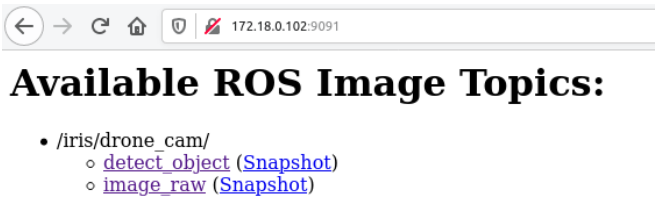
3.2. ábra: A drónból érkező adatok megjelenítése egy weboldalon.

A vke_commander egy másik szolgáltatása, hogy lehetővé teszi egy weboldalon keresztül, a drón irányítását. A 3.3. ábra mutatja a weboldalt, amin keresztül irányítani tudjuk a drónt. Meg tudjuk adni a drónnak, hogy szálljon fel, valamint a jobb oldalt látható körökkel tudjuk irányítani a mozgását, a csúszkával pedig a kamera álláson tudunk módosítani. Az objektum követés fejlesztése során ez nagy segítségemre volt, amikor egy bizonyos pozícióból szerettem volna elindítani az objektum követést vagy szerettem volna az esetlegesen eltévelyedett drónt visszanavigálni a kiindulási pozícióra. A jobb oldalt látható térképnek és a középen látható sötét ablaknak egy másik projektben van szerepe.



3.3. ábra: A drón weboldalon keresztüli irányításnak a felülete.

Végül, a vke_commander képes a különböző ROS Node-ok által közzétett videóknak a megjelenítésére egy weboldalon keresztül. Az alábbi 3.4. ábra azt a felületet mutatja, ahol lehetőségünk van kiválasztani a megnézni kívánt videófolyamot majd az 3.5. ábra szerint megjeleníti nekünk.



3.4. ábra: Lehetőség van kiválasztani, hogy melyik videó folyamot jelenítse meg.

3.5. ábra: A szimulált drón kamerájának képe egy böngésző ablakon keresztül

A vke_commander alapvetően mindent weboldalakon keresztül valósít meg, így semmilyen külön szoftver telepítése nem szüksége a drón irányításához vagy megfigyeléséhez. Ennek következtében, a drón irányítása és megfigyelése is platformfüggetlen és bármilyen operációs rendszeren egyszerűen tudjuk használni, ahol elérhető egy böngésző. Természetesen a vke_commander nem csak a szimulált drónnal hanem egy valódi drónnal is működik.

3.4 vke_videoproxy

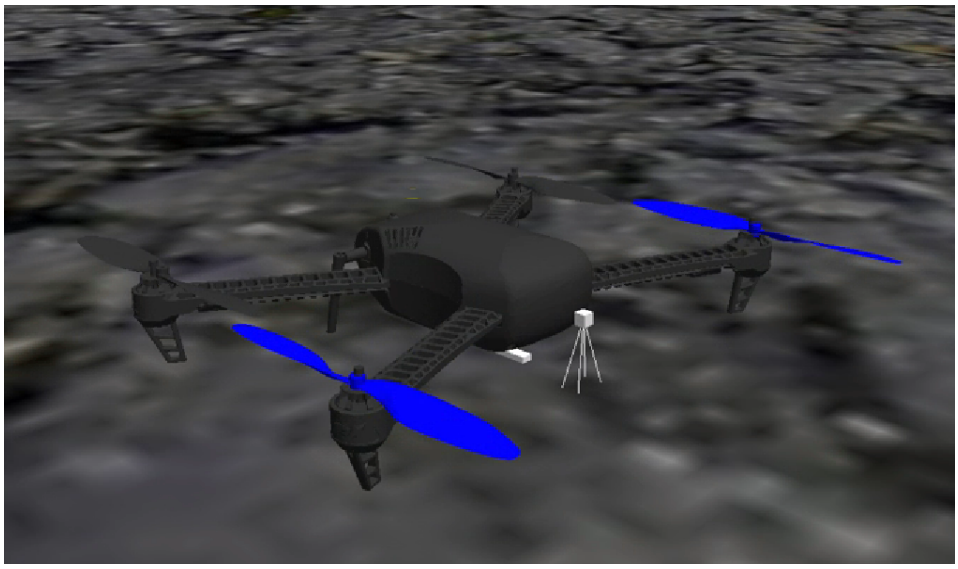
A vke_videoproxy konténernek akkor van jelentősége, ha valódi drónt használunk. Ekkor ez a konténer rácsatlakozik a drónból érkező videofolyamra, azt átkódolja, és elérhetővé teszi az alhálózatban a többi ROS node számára. Alapvetően a drónból érkező videofolyamra szoktuk használni, de igazából bárhonnan, akár az okostelefonunkról érkező videófolyamot is tudja kezelni és elérhetővé tenni a többi ROS node számára.

3.5 vke_px4sim

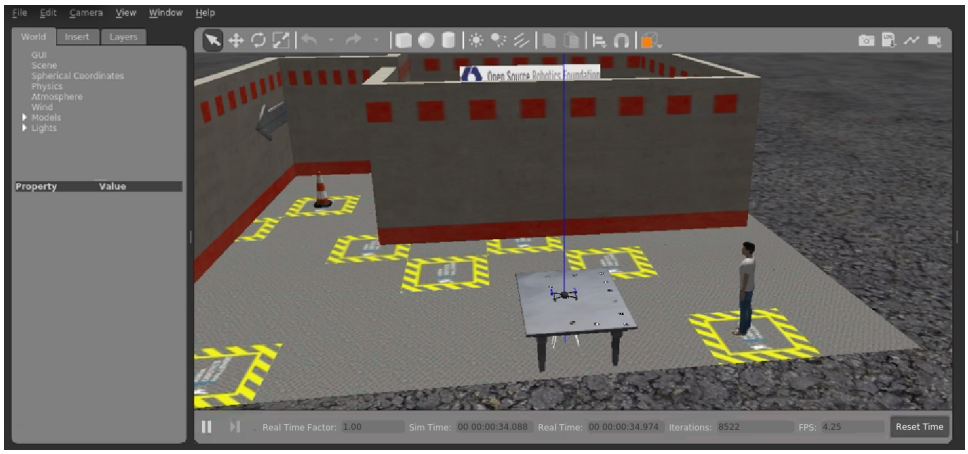
A vke_sim konténer feladata a szimuláció kezelése, elindítása, a megadott modell-ek és a szimulációs világ betöltése. A docker-entrypoint.sh fájlban tudjuk megadni környezeti változók segítségével, hogy milyen drón modellt és milyen szimulációs világot töltsön be. Ezek mellett a drónon futó PX4 szoftver elindítása is ennek a konténernek a feladata. Az alábbi 3.6. ábra mutatja a szimuláció során használt drónt. Sajnos ez nem egyezik meg teljesen a bevezető szakaszban látható Yuneec Mantis Q-van (1.2. ábra) de ez volt az a drón, ami a paramétereiben a leginkább hasonlított rá,

így ezt használtam a szimulációk során. Remélhetőleg ha majd átváltok a szimulált drónról a valódi drónra, nem okoznak problémát a kis eltérések.

Az alábbi 3.7. ábra egy összetett szimulációs világra mutat példát, amit a vke_px4sim konténer segítségével hoztam létre. Egy asztal látható, rajta a nemrégiben bemutatott drónnal, egy álló ember, valamint a háttérben falak láthatóak különböző elrendezésben.



3.6. ábra: A szimulációk során használt drón.



3.7. ábra: A drón és egy összetett szimulációs környezet.

4 Tervezés

4.1 Objektum felismerő és követő szoftver komponens terve

4.1.1 Komponensek terve

Az objektumkövetés megvalósításához alapvetően két nagyobb feladatot kellett megoldani. Az első az objektumok felismerése és a képen a koordinátáinak a meghatározása volt. A második pedig ezen koordináták alapján a drón irányítása oly módon, hogy kövesse a mozgó embert. Felmerül a kérdés, hogy hány komponens, ROS node segítségével érdemes megoldani ezt a két feladatot. Alapvetően két variáció lehetséges, ha mindent egy komponensbe teszek, illetve, ha felismerés alkot egy komponenst és az irányítás pedig a másikat. Mind a két variációnak megvannak az előnyei és a hátrányai.

Abban az esetben, ha egy komponens segítségével oldanám meg, akkor ez a komponens megkapja bemenetnek a videófolyamot és kimenetnek pedig a drón irányításához szükséges parancsokat adja. Mindent lokális változók és függvény hívások segítségével oldok meg. Legfeljebb majd a programnak lesz egy része, amelyik az objektumok felismerésével foglalkozik és egy másik amelyik pedig a drón irányításával. Ez a megoldás egyszerű viszont nagyon merev, nehéz továbbfejleszteni és kis változtatás miatt lehetséges, hogy az egész kódot át kell írni.

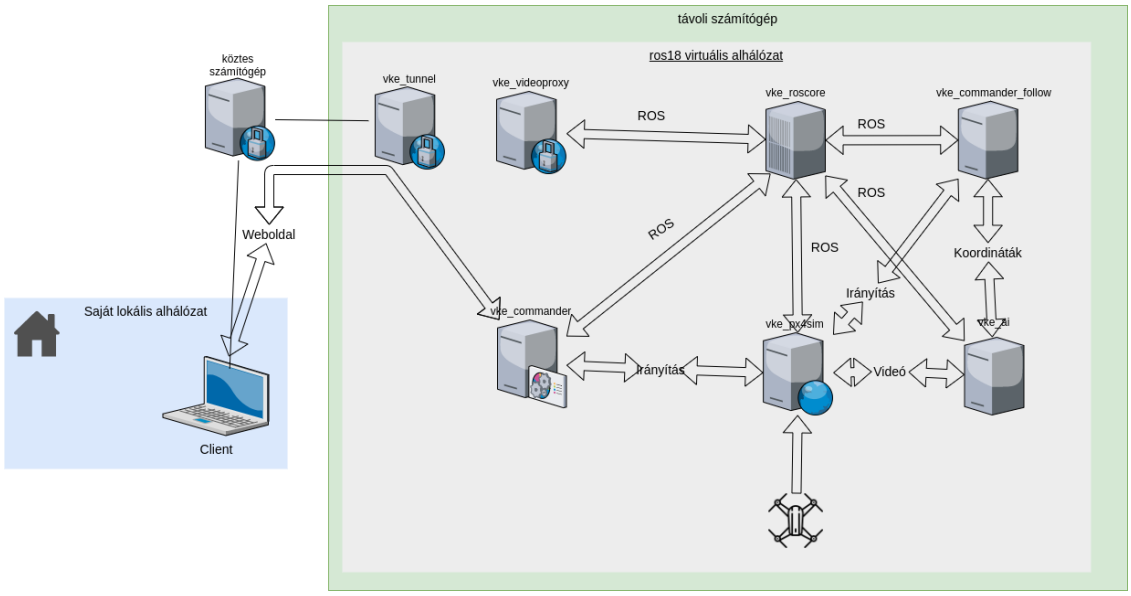
A másik esetben, mikor két komponens segítségével oldom meg a feladatot, akkor az egyik komponens megkapja a videofolyamot, elvégzi rajta a felismerést, majd a felismert objektum koordinátáit valahogyan eljuttatja a másik komponensnek, amely a koordináták alapján összeállítja az irányításhoz szükséges parancsokat és azokat elküldi a drónnak. Ennek a megoldásnak egy hátránya van, hogy a két komponens közötti kommunikációt nekem kell valahogyan megvalósítani. A másik oldalon viszont, ha meg tudom oldani a két komponens kommunikációját, akkor a két feladatot megoldó két komponenst egymástól teljesen függetlenül lehet fejleszteni. Ha úgy tűnik optimálisnak, akkor akár az egész objektumfelismerő logikát is egyszerűen ki tudom cserélni, az nem fogja zavarni a másik komponenst (ez csak akkor lehetséges, ha a két komponens közötti kommunikációt ehhez nem kell megváltoztatni). Akár két külön konténerbe is lehet szervezni a két komponenst. Ebben az esetben a két komponens futásához szükséges szoftverkörnyezetet egyszerűen létre tudom hozni és nem lesznek kompatibilitási problémák a két komponens által használt könyvtárak között. Ez a két komponens két ROS node-ként fog futni a két konténeren belül, mivel a ROS node-ok közötti kommunikáció igencsak egyszerű. Tulajdonképpen csak az üzenet felépítését kell megadnom, a többit, hogy hogyan jut el az üzenet az egyik konténertől a másikig, a ROS megoldja nekem. Ezek alapján a második variációt választottam, vagyis a problémát két komponens segítségével oldottam meg.

Felmerülhet a kérdés, hogy miért nem próbálkozok kettőnél több komponenssel. Ez természetesen lehetséges, de véleményem szerint már túlzottan elbonyolítaná a fejlesztést és nem is egyértelmű, hogy mi lenne a három komponens feladata.

A következő fontos kérdés, hogy melyik objektum felismerő algoritmust válasszam. A 1. táblázat alapján látható, hogy a YOLOv4 jó tulajdonságokkal rendelkezik, valamint az interneten számos leírást találtam, hogyan lehet YOLOv4 segítségével Python-ban egy objektumfelismerő programot elkészíteni, amely videofolyamon ismer fel objektumokat. Ezek alapján a YOLOv4-et használtam az objektumok felismerésére.

4.1.2 Komponensek elhelyezkedése a szimulált drón környezetben

A szimulált drón környezetben konténerek segítségével valósulnak meg a különböző funkciók. Ennek alapján én is konténerek segítségével valósítottam meg az objektumok felismerését, valamint a drón irányítását. Tulajdonképpen két konténerrel egészítettem ki az eddigi konténereket. Az objektumok felmeréséért felelős konténert **vke_ai**-nak valamit a drón irányításáért felelős konténert **vke_commander_follow**-nak neveztem el. Ezek alapján az összes konténer elhelyezkedését az alábbi 4.1. ábra mutatja. Látható, hogy a két új konténert is a ros18 virtuális alhálózatba helyeztem el.



4.1. ábra: A vke_ai és vke_commander_follow konténerek elhelyezkedése.

4.1.3 Kommunikáció a komponensek között

A vke_ai és a vke_commander_follow komponenseken belül futó ROS node-ok közötti kommunikáció megvalósításában segítségemre van a ROS és nekem csak az üzenet formátumát kell meghatározni, de így is több lehetőség, kérdés merül fel, a kommunikáció pontos megszervezésével kapcsolatban.

1. Érdemes-e mindig valamilyen üzenetet küldeni, akkor is ha nincs felismert objektum vagy csak akkor, ha van felismert objektum ekkor az üzenet hiánya fogja jelezni a másik fél számára, hogy nincs új felismerés?

2. Érdemes-e az egy képkockán található összes felismert objektumot egy üzenetben elküldeni, vagy inkább egyesével?
3. Érdemes-e minden objektumot elküldeni, vagy csak azt az objektumot, akit követni szeretnék és ha nincs a képen akkor nem küldök semmit?

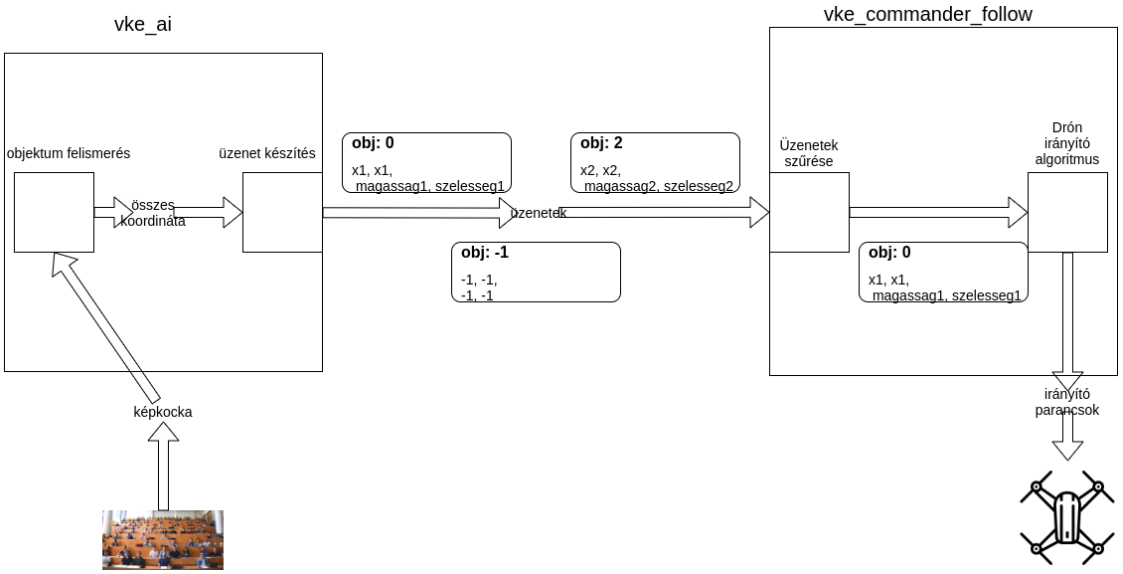
Az első kérdéshez kapcsolódóan az a tapasztalatom korábbi ROS-os projektjeimből, hogy érdemesebb és fejlesztési szempontból kényelmesebb, ha mindig küldünk valamilyen üzenetet a másik fél számára. Az objektumkövetés esetén például a felismert objektumok sorszáma mindig egy pozitív szám, így egy negatív számot küldhetek a másik félnek abban az esetben, ha nincs objektum. A másik fél, aki a drón irányítását végzi tudja, ha negatív számot kap akkor nincs felismert objektum. Ebben az esetben tud valamit reagálni erre, például ekkor arra utasítja a drónt, hogy ne csináljon semmit, vagy kezdjen el egy helyben forogni, hátha így a kamera látószögébe kerül az objektum. A vke_commander_follow egész bonyolult algoritmusokat is futtathat az alapján, hogy hol látta utoljára az objektumot és ez alapján adja ki az utasításokat a drónnak.

Összefoglalva, a szakdolgozatom során mindig küldök üzentet, akkor is ha nincs objektum, valamint a másik fél, ha nincs objektum arra utasítja a drónt, hogy ne csináljon semmit.

A második kérdés arról szól, hogy szeretném-e, hogy ha a másik fél, aki megkapja a felismert objektumok koordinátáját, tudja, hogy melyik felismert objektumok voltak egy képen. Fontos szempont, ha több objektum koordinátáját szeretném egyszerre átküldeni, akkor az ehhez szükséges ROS üzenetnek az összeállítása bonyolultabb, mint ha csak egy darab felismert objektum adatait szeretném átküldeni. Szakdolgozatom során csak egy objektumot kell követnem egy egyszerű szimulációs környezetben, így nincs jelentősége annak, hogy még milyen más felismert objektumok vannak az adott képkockán. Ennek alapján az egyszerűbb megvalósítással rendelkező megoldást választottam, vagyis amikor csak egymás után, egyesével elküldöm az egyes felismert objektumok koordinátáit.

A hármas számú kérdés azzal foglalkozik, hogy melyik komponensnek a feladata a kitüntetett objektumnak a kiválasztása: a vke_ai-nak vagy a vke_commander_follow-nak. Véleményem szerint az a logikusabb, ha a vke_commander_follow komponensnek a feladata a követni kívánt objektumnak a kiválasztása. Mivel a vke_ai feladata csak az, hogy felismerjen egy képen objektumokat és a koordinátákat elküldje, míg a vke_commander_follow komponens feladata, hogy ezeket a koordinátákat értelmezze. Ezek alapján vke_ai komponens minden felismert objektum koordinátáját elküldi a vke_commander_follow komponensnek és a vke_commander_follow a sok objektum közül kiválasztja azt amelyiket követnie kell.

Az alábbi 4.2 ábrán látható a vke_ai és a vke_commander_follow komponens közötti kommunikáció folyamata. Egy képkocka érkezik, ezt feldolgozza a vke_ai, a felismert objektumok koordinátáit egyesével elküldi a vke_commander_follow komponensnek. Ha nincs objektum, akkor -1-et tartalmazó üzenetet küld. A vke_commander-follow komponens kiválasztja a kapott üzenetekből a 0-as számú objektumot tartalmazó üzeneteket. Végül a 0-as számú objektumot tartalmazó üzenetek alapján lefuttatja a drón irányító algoritmust és elküldi az irányító parancsokat a drónnak.



4.2. ábra: Kommunikáció folyamata

Összefoglalva a vke_ai mindig küld üzenetet, akkor is ha nincs felismert objektum; egyesével küldi az objektumok koordinátáit; és minden felismert objektum koordinátáját elküldi.

4.1.4 Mozgó objektum folyamatos követése

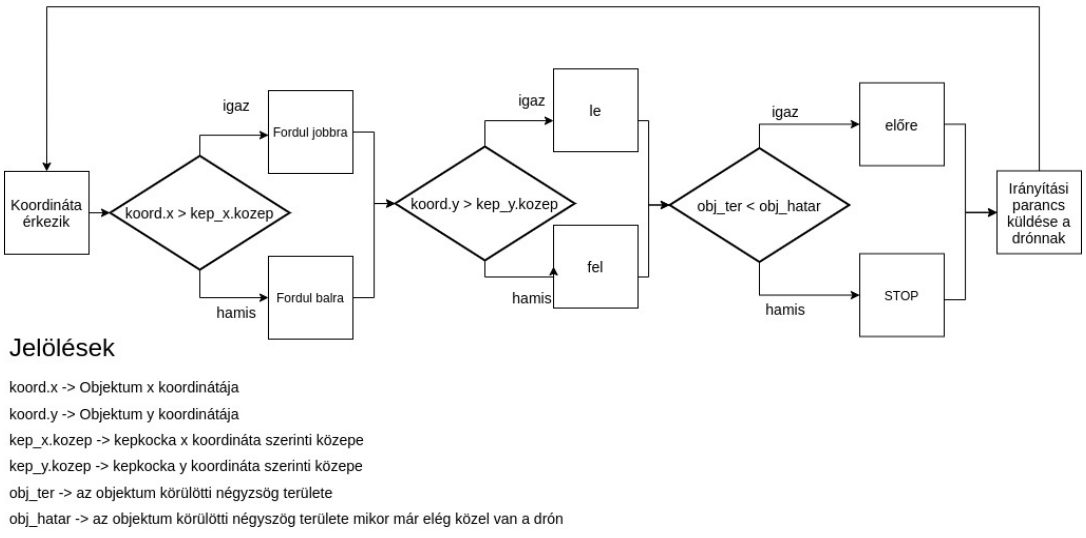
A drónt, a felismert objektum, vagyis a mozgó ember koordinátái alapján kell irányítani. Fontos, hogy ezek a koordináták az adott képkockán határozzák meg az objektum helyzetét. Az lenne a cél, hogy mikor a képen az ember jobbra mozog, akkor a drón is kövesse és forduljon utána jobbra. Mikor az ember balra mozog, akkor az előzőhöz hasonlóan a drón is forduljon utána balra. A kiindulási pozícióban, amit a korábbi 4.2 alfejezet 4.4 ábrája is mutat, a drón távol van még az mozgó embertől, ezért közelebb kell repülnie, de azt szeretném ha csak egy bizonyos távolságig közelítené meg a mozgó embert. Ha a drón valamilyen okból túl közel lenne az emberhez, akkor lebegjen egy helyben és ne csináljon semmit. Fontos lenne még, hogy a drón tartani tudjon egy bizonyos magasságot a mozgó emberhez képest. Alapvetően ezt a három problémát kell megoldani a koordináták alapján, vagyis mikor mozogjon a drón jobbra-balra; mikor közelítsen-lebegjen egy helyben; mikor mozogjon fel-le. A vke_ai a mozgó objektum koordinátái mellett a felismert objektum köré rajzolt négyszög méreteit is elküldi., Ez fontos adat lesz a drón és az ember közötti távolság meghatározásában.

Az alábbi 4.3. ábra mutatja a drón irányításának a lépéseit. Ismerem a videófolyam képkockáinak a méretét és ehhez viszonyítom majd az egyes koordinátákat. Első lépésként érkezik egy koordináta, ami a mozgó emberhez tartozik. Ezt követően megnézem, hogy ez a koordináta a kép jobb vagy a bal felén helyezkedik-e el. Ha jobb felén akkor jobbra, míg ha a bal felén, akkor balra mozgató parancsot regisztrálok. Ezután az ember koordinátájának a magasságát vizsgálom meg. Azt figyelem, hogy a képnek az alsó vagy pedig a felső részén helyezkedik-e el. Ha az alsó felén akkor felfelé, míg ha a kép felső felén, akkor lefelé mozgató parancsot regisztrálok. Végül a koordinátákkal érkező szélességi és magassági parancsokat is

megvizsgálom, ezek az ember köré rajzolható négyszög magasságának és szélességének felelnek meg. Ezeknek az adatoknak a segítségével ki tudom számolni az ember köré rajzolható négyszög területét. Ez a terület, ha közelebb van a drón akkor nagy, míg ha távol van a drón, akkor pedig kicsi és a kettő között pedig onnantól, hogy távol vannak egymástól odáig, hogy közel lesznek, a terület folyamatosan nő. Így, a négyszögnek a területével meg tudom becsülni a drón és az ember közötti távolságot. Ennek alapján megvizsgálom ezt a területet és ha egy határértéknél kisebb, akkor egy előre parancsot regisztrálok, míg ha ennél a határértéknél nagyobb, akkor pedig megállok. Szerencsétlen esetben megtörténhet, hogy megáll a drón, viszont az ember a szimulációban pont a drón felé kezd el mozogni és így nagyon közel lesznek egymáshoz. Ebben az esetben a drón csak egy helyben fog állni és megvárja amíg az ember elkezd távolodni és ha elég távol lesz akkor elindul utána. Valóságban ilyen probléma nem lesz, mert ehhez hasonló esetekben csak egyszerűen egy megfelelő távolságból kikerüljük majd a drónt. Egy másik probléma akkor adódhat mikor túl közel van a drón az emberhez, az embernek csak egy részét látja és azt a részt ismeri fel embernek. Ekkor a felismert rész köré rajzolt négyszög területe kicsi lesz és ennek alapján azt hihetném, hogy nagyon távol van a drón az embertől, pedig ellenkezőleg nagyon közel van. Ennek megoldására a terület vizsgálatánál a határértéket úgy állítom be, hogy megfelelően nagy távolság legyen a mozgó ember és a drón között.

Utolsó lépésként pedig a regisztrált parancsokat elküldöm a drónnak.

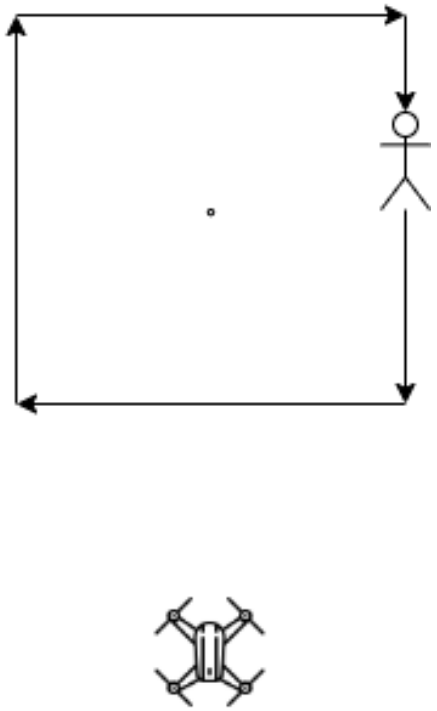
A koordináták kisebb pontatlanságai miatt érdemesebb lehet majd egy kicsit bonyolultabbra készíteni a drón követő algoritmust. Amikor a koordináta függőleges és vízszintes helyzetét figyelem, nem egy ponthoz viszonyítok, hogy annál kisebb-e vagy nagyon, hanem egy tartományhoz viszonyítom, hogy ennél a tartománynál kisebb-e, nagyobb-e vagy éppen a tartományon belül helyezkedik-e el a koordináta. Meg fogom vizsgálni a vke_ai komponensből érkező koordinátákat, hogy mennyire stabil az értékük és mennyire ugrál ide-oda és ez alapján fogom eldönteni, hogy érdemes-e a fentebb leírt ötletet beleépíteni az algoritmusba.



4.3. ábra: Drón irányításának lépései

4.2 Szimulációs világ megtervezése

A vke_ai és vke_commander_follow komponenseket egy szimuláció segítségével futtattam. Ez a szimuláció alapvetően egy mozgó objektumból, amit majd a drón felismer és magából a drónból áll. A mozgó objektumnak egy járkáló embert választottam. Az interneten elérhető útmutatók alapján ez tűnt a legkönnyebben megvalósíthatónak. A mozgó ember az origó körül egy négyzet alakú pályát jár be. Ezt mutatja alább a 4.4. ábra. Az ábrán látható ahogy a mozgó ember egy négyzet alakban mozog az origó körül. Az ábra alján látható a drónt és a mozgó emberhez képest ez lesz a kiindulási pozíciója.



4.4. ábra: A szimulációban az ember és a drón elrendezése

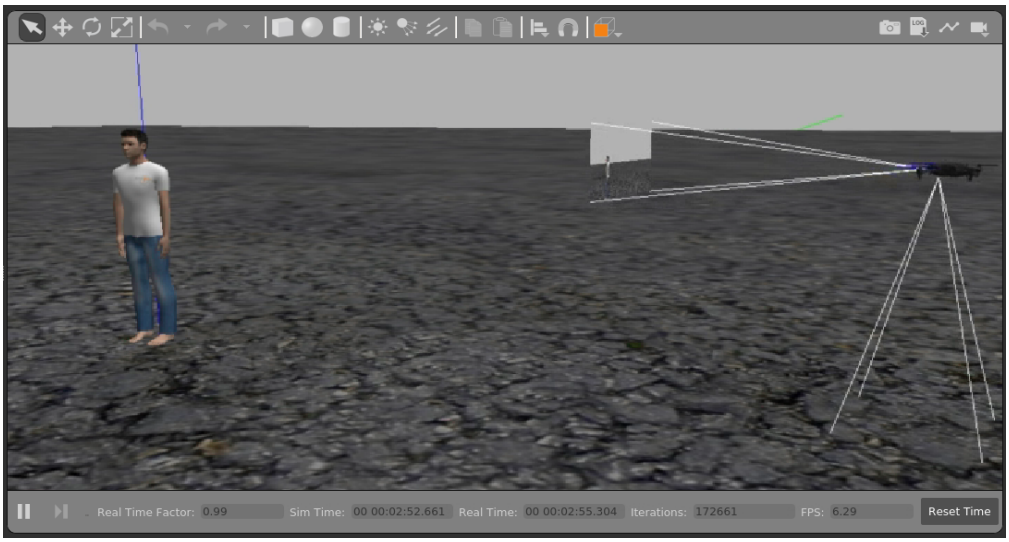
Az objektum követést két lépcsőben fejlesztettem, így könnyebb volt a különböző repüléshez és vezérléshez szükséges paraméterek beállítása. Első lépésnek csak egy álló emberrel próbáltam el az objektumkövetést, ennek során a drón csak odarepült és eközben meg tudtam határozni, hogy milyen tartományban kell finomhangolnom a paramétereket. Következő lépés során mozgó embert kellett követnie a drónnak és ekkor végeztem el a paraméterek finomhangolását.

5 Komponensek megvalósítása

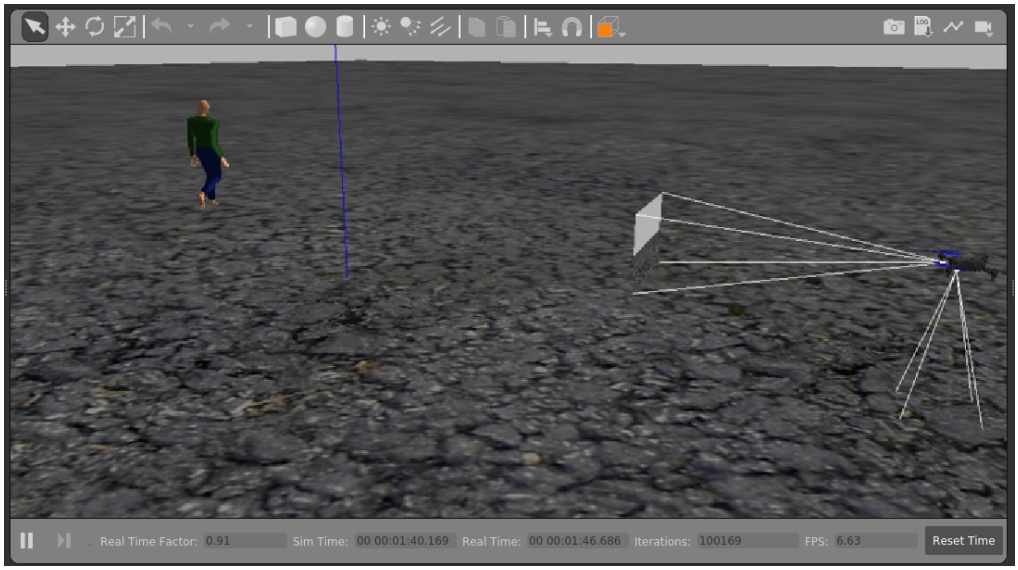
5.1 Megvalósított szimuláció bemutatása

A terveknek megfelelően készítettem el a szimulációs környezetet. Két szimulációs világot is összeépítettem, az egyikben egy mozgó, míg a másikban egy álló emberrel. Az álló és a mozgó embernek két különböző kinézetű embert választottam, hogy könnyen meg tudjam különböztetni a két szimulációt. A szimulációknál szükséges beállítani, hogy legyen valamilyen mintázatú föld, ellenkező esetben a drón „leesik” a szimulációból és egyszerűen addig zuhan, amíg le nem állítom a szimulációt. Egy aszfalt mintázatú földet választottam, ezt találtam a legegyszerűbbnek beállítani. A Gazebo-ban lehetőség van egyébként háromdimenziós domborzatot is beállítani. Az alábbi 5.1. ábra az álló emberrel készült szimulációt mutatja.

A képen bal oldalon, a szimulációs világ origójában az egy helyben álló ember látható. A képen jobb oldalon pedig a repülő drón figyelhető meg. A drónból fehér vonalak indulnak ki lefele és előre. Ezek a kamerát jelképezik a Gazebo-ban. Tehát a drónnak két kamerája van, egy, ami előrefelé néz és egyébként állítható és egy másik, amelyik lefelé néz. A kettő közül csak az előre felé nézőhöz férünk hozzá és ezt fogom majd használni az objektumfelismeréshez is. A képen még látható az aszfalt mintázatú föld is.



5.1. ábra: Szimuláció az álló emberrel és a repülő drónnal



5.2. ábra: Szimuláció a mozgó emberre és a repülő drónnal

A fenti 5.2. ábra a mozgó emberrel készült szimulációt mutatja. A képen bal oldalt látható a mozgó ember. A képen jobb oldalt pedig a repülő drón figyelhető meg a két kamerával. A képen még látható az aszfalt mintázatú föld az előző álló emberes szimulációhoz hasonlóan

A mozgó emberrel elkészített szimulációnál fontos szempont volt, hogy milyen gyorsan mozog az ember és mekkora négyszög alakú pályán sétál. Ha túl gyorsan mozog az ember, akkor nagyon gyorsan kikerül az ember a drón kamerájának a látószögéből és így a drón nem fogja tudni követni. Ha pedig túlságosan lassan mozog az jobb ugyan, mert egyszerűbben tudja követni a drón, viszont nem nagyon fog hasonlítani a valóságra. Így, megkerestem azt a sebességet, ami a lehető leggyorsabb azok közül, amellyel mozgó embert még biztonságosan tud követni a drón. A másik kérdés, hogy mekkora legyen a mozgó ember által bejárt négyszögnek az oldala. Ha túl kicsi, akkor az ember alig mozog valamit és a drón miután odarepült, gyakorlatilag csak egy helyben fog lebegni, mert a drón és a mozgó ember távolsága alig változik. Ezek alapján az lenne az ideális, ha minél nagyobb lenne ez a négyzet. Így választottam egy viszonylag nagy négyzetet, amin majd az ember mozogni fog.

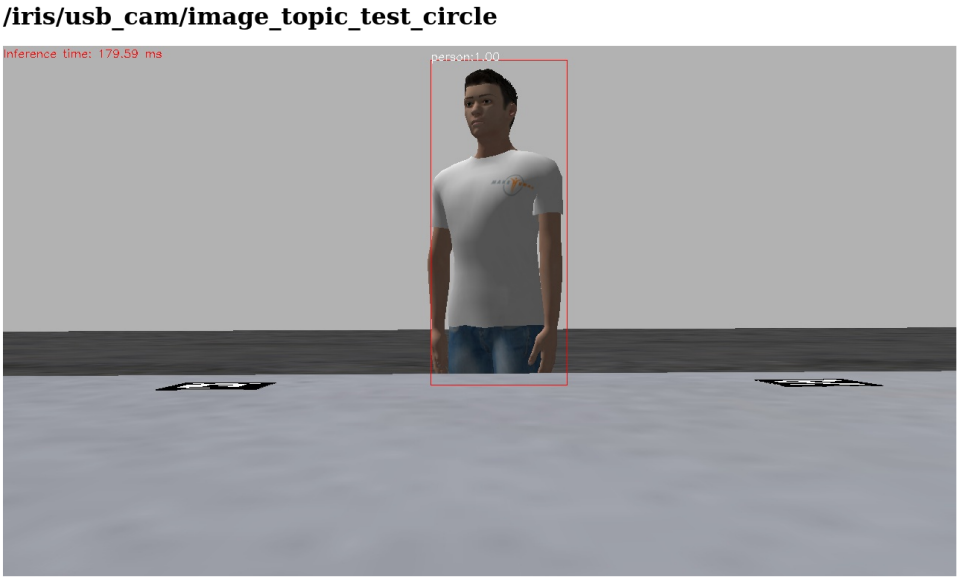
A négyszög méretét nyolc egységnek választottam. A mozgás sebességét közvetlenül nem tudom beállítani. Arra van lehetőség, hogy megadjam mennyi idő teljen el, míg a négyszög egyik sarkából a másikba elsétál a mozgó ember. Ezt az időt 15 másodpercnek választottam.

5.2 Objektum felismerés hardver gyorsítással

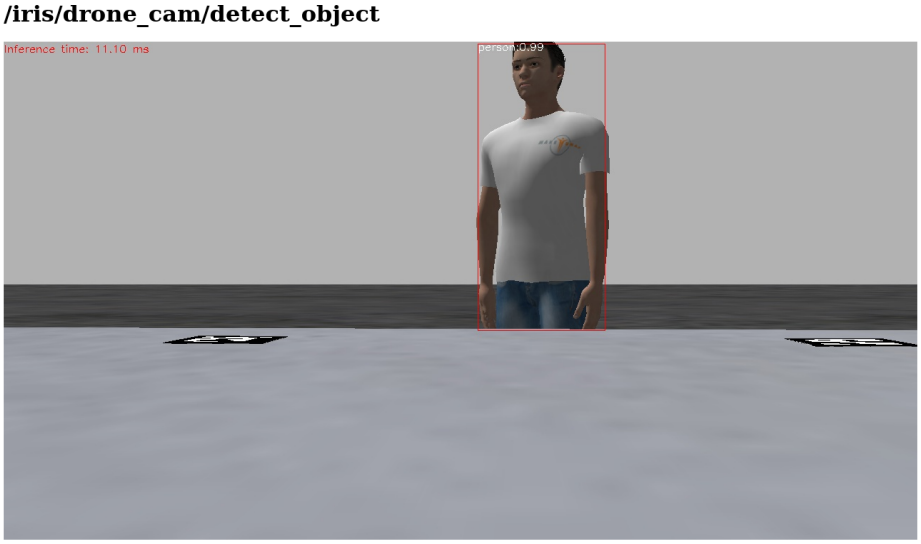
A gépi tanulás alapú algoritmusoknál fontos szempont, hogy az algoritmust csak a processzor segítségével futtatjuk vagy besegít a futtatásba a grafikus kártya, a GPU is.

A grafikus kártya segítségével futtatott gépi tanulás alapú algoritmusok általában gyorsabban lefutnak mintha csak simán a processzort használtuk volna. Az egyetemen lévő számítógép, amit a szimulációk futtatására használok rendelkezik egy erős Nvidia grafikus kártyával. Így, minden adott lenne, hogy az objektum felismerés során használt YOLOv4-et a grafikus kártya segítségével futtassam. A másik oldalon viszont rengeteg kompatibilitási probléma adódhat míg egy Nvida grafikus kártyát beüzemelünk. Lehet, hogy szerencsénk lesz és minden rögtön működik, de később is figyelni kell, mert a frissítéseknél újabb kompatibilitási problémák léphetnek fel.

Elvégeztem egy gyors mérést, hogy milyen gyorsan fut a YOLOv4 az egyetemi számítógépen grafikus kártya nélkül és grafikus kártya segítségével, mekkora gyorsulás érhető el a grafikus kártya használatával.



5.3. ábra: YOLOv4 futásának ideje GPU nélkül



5.4. ábra: YOLOv4 futásának ideje GPU segítségével

A fenti 5.3. ábra és 5.4. ábra mutatja a mérések eredményét. Látható, hogy GPU nélkül a YOLOv4 lefutása 179,59 ms-ot vett igénybe, míg GPU segítségével ehhez csak kicsivel több, mint 11 ms volt szükséges. Látható, hogy GPU felhasználásával hatalmas ugrás érhető el a futási sebesség terén. Az objektumfelismerés hatékony működéséhez elengedhetetlen, hogy az adott képkocán lévő objektumot gyorsan felismerjük, így mindenképpen használom majd a grafikus processzor által nyújtott gyorsítási lehetőséget az esetleges kompatibilitási problémák ellenére is.

5.3 Kiegészítés valós kamerához

Az objektumfelismerő algoritmus alapvetően csak a szimulátorból érkező videófolyamon ismer fel objektumokat. A szimulátor a világnak egy nagyon leegyszerűsített mása, ezért itt nem tudom igazából kipróbálni, hogy mennyire működik jól az objektumfelismerő algoritmus. Így, érdemes lenne valahogyan egy bonyolultabb környezetről készített videófolyamot eljuttatni ehhez a felismerő komponenshez. Ezt legegyszerűbben az okostelefonom segítségével készített videofolyammal tudtam megoldani. Ehhez először is szükséges, hogy a telefonom bekerüljön a ros18 virtuális alhálózatba. Ezt a következőképpen oldottam meg.: A telefonomon készíttettem egy wifi hotspot-ot és ehhez kapcsolódik hozzá a laptopom egy második wifi interfésszel, az első wifi interfésszel az internethez kapcsolódik. Azért a telefon csinálja a hotspot-ot, mert az igazi drón, amit majd a jövőben szeretnék használni így működik. A drón csinálja a wifi hotspot-ot és ahhoz hozzákapcsolódva tudom irányítani. Ezt követően még a laptopomon NAT szabályok segítségével be kellett állítanom, hogy a laptop az internetről az okostelefonnak érkező csomagokat a második interfészén az okostelefon hálózatába továbbítsa. A telefon a videofolyam elküldése során létre akar hozni egy új kapcsolatot a ros18 virtuális alhálózatban lévő konténerekkel ezért egy második NAT szabály is szükséges, ami a telefontól érkező csomagokat továbbítja a vke_tunnel segítségével a ros18 virtuális alhálózatba. Ezen beállítások után már képes vagyok videofolyamot továbbítani a telefontól a ros18-ban lévő konténerekhez. Végül utolsó lépésként a vke_videoproxy konténert is be kellett állítanom, hogy fogadja a telefontól érkező videofolyamot és azt a többi ROS node számára elérhetővé tegye. Ezen beállítások után már képes vagyok felhasználni a telefontól érkező videófolyamot, a vke_videoproxy konténerben futó ROS node-hoz hozzákapcsolódva.

5.4 vke_ai komponens bemutatása

5.4.1 Komponens bemenete és kimenete

A vke_ai komponens a drón kamerájából érkező videófolyamot elemzi és az objektumokat ismeri fel rajta. A felismerés elvégzéséhez szüksége van egy videófolyamra, ezért rákapcsolódik egy ROS node-ra, ami képes továbbítani neki a drón kamerájából érkező videófolyamot. Ez lesz a vke_ai komponens bemenete. Ezen a videófolyamon felismeri az objektumokat, majd a koordinátáikat közzé teszi a többi ROS node számára. Ezen kívül a videofolyam egyes képkockáin felismert objektumok köré rajzol egy négyszöget, ezeket a megváltoztatott képkockákat mint egy új videófolyam elérhető teszi a többi ROS node számára. Így a vke_ai komponensnek egy

bemenete lesz a drónból érkező videófolyam és két kimenete lesz a felismert objektumok koordinátái és a megváltoztatott videófolyam.

5.4.2 Kiinduló docker image

Az objektumfelismerést végző konténer a 5.2 fejezetben bemutatott rövid mérés alapján GPU segítségével fog működni. Ezért létrehoztam egy *czdani/opencv4_4_cudnn11_0_cuda8_ubuntu20_04* nevű docker image-et, amiben összegyűjtöttem azokat a könyvtárakat és csomagokat, amelyek a grafikus processzor által gyorsított YOLOv4 lefuttatásához és egy ROS node-ba becsomagolásához szükségesek.

A *czdani/opencv4_4_cudnn11_0_cuda8_ubuntu20_04* nevű docker image elkészítése során az nvidia/cuda:11.0-devel-ubuntu20.04 docker image-ből indultam ki, így tudtam használni a grafikus processzort a konténeren belül.

A *czdani/opencv4_4_cudnn11_0_cuda8_ubuntu20_04* nevű docker image az alábbi könyvtárakat tartalmazza:

- cuda
- cudnn
- opencv és a hozzá kapcsolódó csomagok
- ROS

A cuda-t nem kellett külön telepítenem a kiindulási nvidia/cuda:11.0-devel-ubuntu20.04 docker image már tartalmazta ezt. A cuda segítségével tudom használni a számítógépen elérhető grafikus processzort a konténeren belül is.

A cudnn-t amely az nvidia CUda Deep Neural Network library, cuda mély neurális háló könyvtár, szavakból áll össze, külön kellett telepítenem, mivel nem volt elérhető olyan docker image, ami Ubuntu20.04-re épül és a cudnn is telepítve van hozzá. Az Ubuntu20.04-re azért volt szükség, mivel a legújabb ROS noetic-et kellett használnom az egyik ROS csomag miatt és a ROS noetic csak Ubuntu20.04-el működik együtt. A cudnn telepítéséhez le kellett töltenem az nvidia honlapjáról több deb fájlt és azokat ez egyetemen lévő gépre, ahol dolgoztam, kellett másolnom. Végül megadtam a *czdani/opencv4_4_cudnn11_0_cuda8_ubuntu20_04* docker image-hez tartozó Dockerfile-ba, hogy telepítse az előző lépésben átmásolt deb fájlokat . A cudnn segítségével lehet neurális hálókat gyorsan a grafikus kártyán futtatni.

Az OpenCV egy nyílt forráskódú könyvtár különböző gépi látás alapú algoritmusok gyors futtatására. Az OpenCV a legfontosabb könyvtár a *czdani/opencv4_4_cudnn11_0_cuda8_ubuntu20_04 docker* image-ben. Segítségével fogom futtatni a YOLOv4 objektumfelismerő algoritmust és elemezni az eredményét, hogy például azokat a felismert objektumokat, amiknek nagyon kicsi a valószínűsége eldobjam vagy kiszűrjem a redundáns felismeréseket, vagyis amikor ugyanazt az objektumot ismertem fel többször. Ezen kívül az OpenCV-t fogom használni a képkockák átméretezésere, illetve a felismert objektumok köré a négyszög rajzolására is. Az OpenCV-t külön le kellett fordítanom, mivel grafikus kártyával is szerettem volna használni. Ehhez első lépésnek letöltöttem az OpenCV működéséhez szükséges különböző csomagokat. Letöltöttem az OpenCV forráskódját, majd még a fordítás előtt beállítottam különböző paramétereket, hogy hogyan forduljon le az OpenCV, például

legyen cuda és cudnn támogatás. Ezt követően lefordítottam, ez igen hosszú időt vett igénybe. A fordítás után minden készen állt, hogy grafikus kártyával gyorsított objektumfelismerést tudjak futtatni. Az OpenCV-ről további információk itt olvashatóak: [25]

A ROS-t is telepítettem ebbe a docker image-be, hogy lehetőségem legyen majd ROS node-okat futtatni. A telepítés egyszerű volt, egyszerűen megadtam a ROS csomagokat, amikre szükségem van és az Ubuntu csomagkezelője majd a docker build parancs futása során ezeket letölti és telepíti nekem.

Az *czdani/opencv4_4_cudnn11_0_cuda8_ubuntu20_04* docker image elkészítésének nagy előnye, hogy ezt a docker image-et csak egyszer kellett elkészítenem és utána akárhány helyen használhatom kiindulási docker image-nek. Ez azért fontos, mert mikor grafikus kártya segítségével gyorsított OpenCV algoritmusokat szeretnék futtatni docker konténerek segítségével és ehhez nekem kell elkészíteni a docker image-t, nem kell majd elkészítenem az ehhez szükséges Dockerfile-t majd pedig kivárni, amíg letöltődik az összes csomag, lefordul az OpenCV és elkészül a docker image, hanem egyszerűen használhatom a *czdani/opencv4_4_cudnn11_0_cuda8_ubuntu20_04* docker image-et kiindulási docker image-nek. Ez a kiindulási docker image tartalmazni fogja az összes szükséges könyvtárat a grafikus kártya segítségével gyorsított OpenCV algoritmusok futtatásához és nekem csak a feladatra specifikus könyvtárakkal kell majd kiegészítenem. Ha több olyan konténert is szeretnék, ami OpenCV és grafikus kártya segítségével futtat valamilyen gépi tanulás alapú algoritmust, akkor az ehhez szükséges specifikus docker image elkészítéséhez nem kell minden könyvtárat újra letöltögetnem, hanem egyszerűen itt is használhatom kiindulási docker image-nek a már emlegetett *czdani/opencv4_4_cudnn11_0_cuda8_ubuntu20_04* docker image-t.

A *czdani/opencv4_4_cudnn11_0_cuda8_ubuntu20_04* docker image-et a dockerhub-ra is feltöltöttem. Így, ha egy másik számítógépen is szeretném ezt használni akkor egyszerűen csak letöltöm a dockerhub-ról és nem kell azzal foglalkozni, hogy újra lefordítsam vagy pedig valahogyan átmásoljam a másik számítógépre a docker image-et.

A vke_ai docker image fejlesztése során kiindulási docker image-nek a fent leírt czdani/opencv4_4_cudnn11_0_cuda8_ubuntu20_04 docker image-et használtam és ezt egészítettem ki a YOLOv4 használatához szükséges fájlokkal, valamint a cv_bridge nevű ROS csomaggal. A cv_bridge ROS csomag segítségével a ROS-ban használt videófolyam képkockáit tudom alakítani az OpenCV által használt kép formátumra és vissza.

Ezek után rendelkezésemre állt az a környezet, aminek a segítségével objektum felismerő algoritmust vagyis a YOLOv4-et tudom majd futtatni. A következő lépés magának a python programnak az elkészítése, ami a tényleges felismerést a koordináták kinyerését és elérhetővé tételét elvégzi.

5.4.3 Komponens részeinek a bemutatása

A következő alfejezetben a vke_ai különböző részeit fogom bemutatni: mi történik a Dockerfile-ban, a docker-entrypont.sh-ban, a compile.sh-ban illetve a vke_ai.sh-ban, valamint milyen nagyobb egységekből osztályokból épül fel a felismerést elvégző python fájl.

A Dockerfile az előző részben (5.4.2) leírtaknak megfelelően alapvetően a cv_bridge letöltését és fordítás elvégző parancsokból, valamint egy-két a ROS és python együtt működéséhez szükséges fontos csomagok telepítését elvégző parancsból áll. A YOLOv4 futásához szükséges fájlokat nem itt töltöm le hanem majd egy mappába összegyűjtve felcsatolom a konténerhez annak indulásánál.

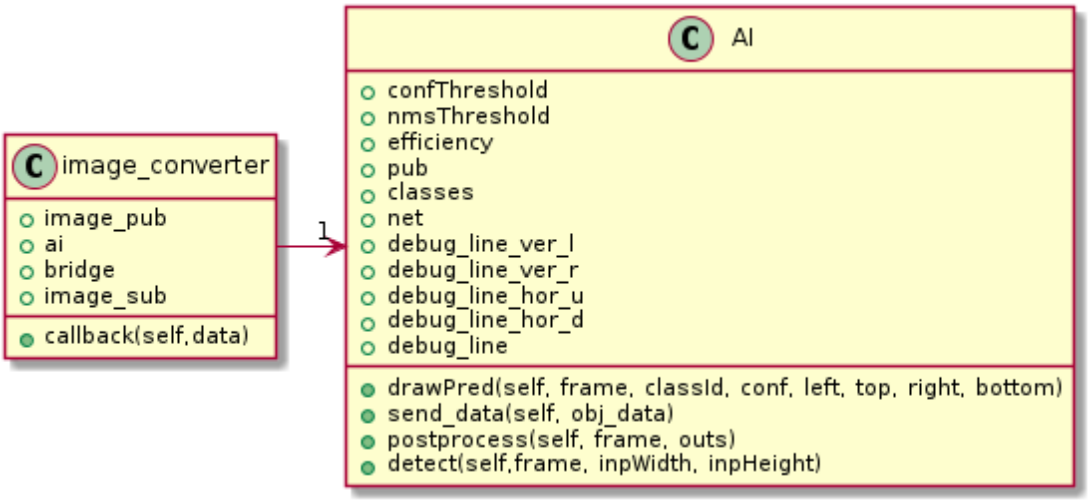
A docker-entrypoint.sh fájlban először lefuttatom a „catkin_make” parancsot, aminek a vke_ai és a vke_commander_follow komponensek kommunikációjában van szerepe, (erről bővebben a 5.5 alfejezetben olvashatunk) és elindítom a felismerést végző python fájlt. Így amikor elindítom a konténert akkor automatikusan lefut a docker-entrypoint.sh, ami elindítja a felismerésért felelős python fájlt és így maga a felismerés is automatikusan elindul.

A compile.sh fájlban megadom a docker image elkészítéséhez szükséges parancsot és az elkészült docker image-et is vke_ai-nak neveztem el.

A vke_ai.sh fájlban adom meg, hogy pontosan, hogyan induljon el a konténer. Beállítom a konténer ip címét, hogy a ros18 virtuális hálózathoz csatlakozzon, milyen könyvtárakat csatoljon a konténerhez, milyen ip címen találja roscore-t, illetve ami legfontosabb, hogy használja a konténer a gazdagép grafikus kártyáját. Ennek során csatolom a konténerhez azt a mappát amit a YOLOv4 futásához szükséges fájlokat tartalmazza.

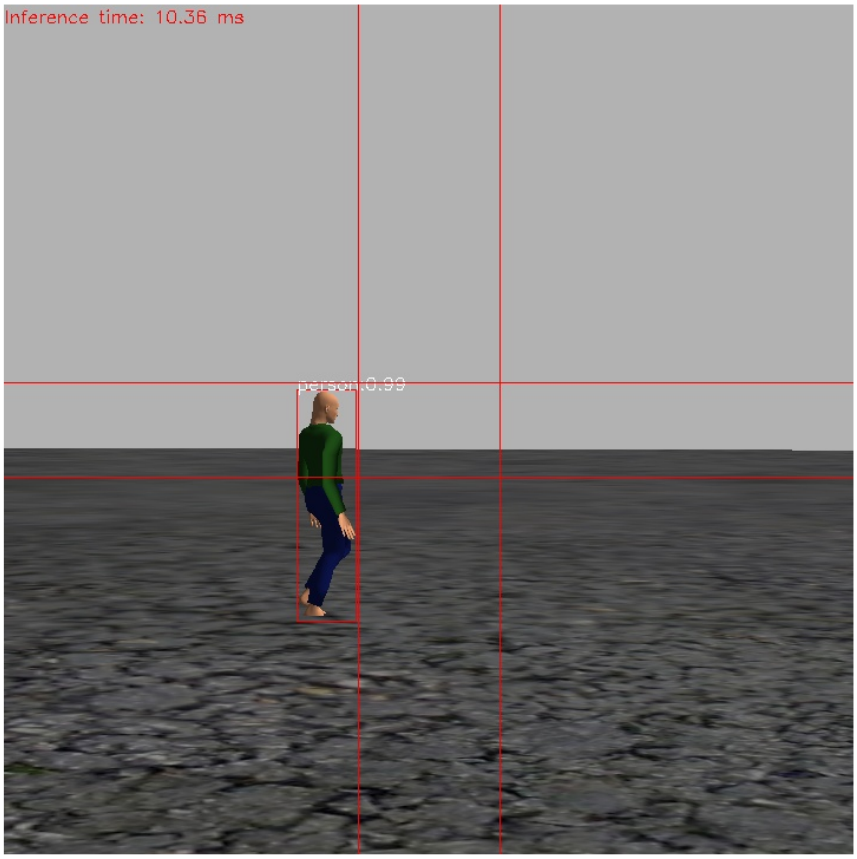
Az objektumok felismeréséért, a YOLOv4 futtatásáért és a koordináták elérhetővé tételéről gondoskodó python programot detect_video.py-nak neveztem el. Alább a 5.5. ábra mutatja a detect_video.py python programot alkotó AI és image_converter osztályt, valamint az őket alkotó változókat és függvényeket. Látható, hogy az AI osztály és a image_converter osztály között egy asszociációs kapcsolat van, mivel az image_converter osztály tartalmaz egy referenciát az AI osztályra.

Az AI osztály feladata, hogy elvégezze az objektumok felismerését és elérhetővé tételét a többi ROS node számára. Az image_converter osztály feladata , hogy rácsatlakozzon a videofolyamot közzé tevő ROS node-ra, fogadja a képkockákat, elindítsa az objektum felismerést, végül pedig a módosított képet a bekeretezett objektumokkal, elérhetővé tegye más ROS node-ok számára. A detect_video.py pontos működését a következő fejezetben ismertetem.



5.5. ábra: A detect_video.py python programot alkotót két osztály.

Az alábbi 5.6. ábra mutatja a detect_viedo.py python programot működése közben, ahogy a szimulált drón által készített videófolyam egy képkockáját feldolgozta. Látható, hogy az embert jól felismerte, 99%-ban embernek gondolja. A bal felső sarokban látható szám a neurális háló sebességét mutatja, vagyis egy képkocka végigfuttatása a neurális hálón 10,36 ms-ot vesz igénybe. Láthatóak még a képen függőleges és vízszintes vonalak. Ezeknek a vke_commander_follow fejlesztésben van szerepe. A drón attól függően fog különböző mozgásokat végezni, hogy a felismert embert körülvevő négyzet középpontja melyik piros egyenes által határolt tartományban található.



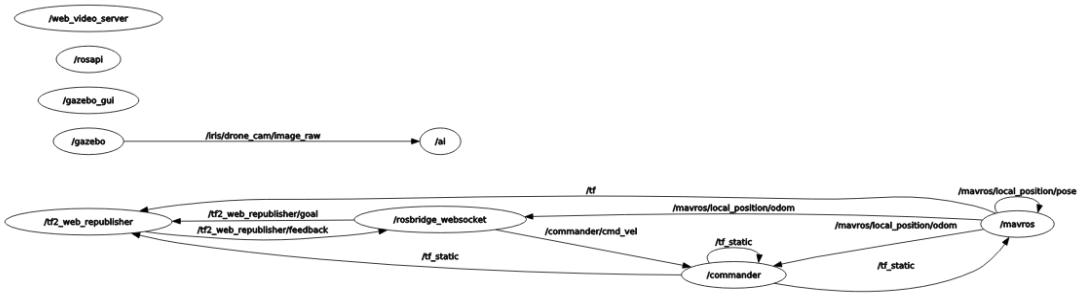
5.6. ábra: A detect_video.py python program működés közben.

5.4.4 Objektum felismerésének a folyamata

A vke_ai feladata az objektumok felismerése a kép koordinátáinak a közzététele más ROS node-ok számára. Ezt a detect_video.py python program segítségével valósítja meg. Ennek a folyamata egyszerű, mégis több függvényhívás szükséges hozzá.

ROS node-ok aszinkron kommunikációja callback függvények segítségével valósul meg. Így, mikor érkezik egy képkocka a videófolyamból a vke_ai komponensnek akkor automatikusan meghívódik egy előre megadott függvény. Ezt a függvényt a detect_video.py python programban akkor kell megadnom, amikor definiálom, hogy melyik másik ROS node-hoz szeretnék hozzákapcsolódni.

Az alábbi 5.7. ábra mutatja, hogy mely ROS node-ok és kapcsolataik szükségesek a detect_video.py python program futásához. A körök jelképezik ROS node-okat. A körök között lévő vonalak pedig a ROS node-ok között létrejött kapcsolatokat jelképezik. Látható, hogy a detect_video.py által futtatott „ai” ROS node, hozzákapcsolódik a „gazebo” nevű ROS node-hoz egy „/iris/drone_cam/Image_raw” nevű kapcsolaton keresztül. Ezen a kapcsolaton keresztül fogja megkapni az „ai” ROS node és ezzel együtt a detect_video.py python program is, a drón kamerájából érkező videófolyam képkockáit.



5.7. ábra: A vke_ai komponensen belül a detect_video.py python program futásához szükséges ROS node-ok és kapcsolataik

A callback függvényt, ami akkor hívódik meg automatikusan mikor egy új képkocka érkezik, a detect_video.py python programban *callback*-nek neveztem el és az image_converter osztálynak a tagfüggvénye. Ez a *callback* nevű függvény, függvény paraméterként megkapja magát az adatot, ebben az esetben képkockát, amelyik a másik ROS node-tól érkezett. A *callback* nevű függvény ezt az adatot átalakítja a ROS által használt kép formátumból az OpenCV által használt kép formátummá és az AI osztályon keresztül elindítja az objektumfelismerés folyamtatát a *detect* nevű függvény meghívásával. Mikor lefutott az objektumfelmerés, visszakapom a megváltoztatott képkockát, ahol a felismert objektumok be vannak keretezve egy piros négyszöggel.

Az alábbi 5.8. ábra ábra mutatja, hogyan néz ki egy képkocka a felismerés után, amikor nem szimulátorból, hanem a valóságból kapom a videófolyamot, amit az okostelefonom kamerájával készítettem a 5.3 alfejezet beállításait követően. Látható ahogy a teniszlabdát „sports_ball”-nak ismerte fel 99%-os bizonyossággal és egy piros négyszöggel be is keretezte. A bal felső sarokban látható, hogy gyorsan futott ebben az esetben a neurális háló. Egy képkocka feldolgozására kevesebb mint 10 ms volt szükséges, ráadásul valós és nem szimulációs képkockákon dolgozott. Utolsó lépésként a *callback* nevű függvény közzé teszi ezt a megváltoztatott képkockát más ROS node-ok számára. Ez a folyamat minden olyan esetben lejátszódik, amikor új képkocka érkezik.

5.8. ábra: A callback nevű függvény által közzétett képkocka a felismerés után

Az objektum felismerését egy képkockáról az image_converter osztály callback nevű függvénye kezdeményezi az AI osztály *detect* nevű függvényének a meghívásával. Ennek során az image_converter osztály átadja az AI osztálynak az OpenCV formátumú képkockát feldolgozásra. Az AI osztály *detect* függvénye magát az objektum felismerést és az utómunkákat végezi el. Először a megkapott képkockát átalakítja egy olyan speciális kép formátummá, amit a YOLOv4 ismer. Ezt követően végigfuttatja a YOLOv4 neurális hálón és elmenti ennek a kimenetét. A kimenetek olyan felismeréseket is tartalmaznak, amiknek kicsi a valószínűsége, hogy helyesek, valamint lehet, hogy egy objektumot többször is felismert a neurális háló. Ezeknek a kiszűrését az AI osztály *detect* függvénye a *postprocess* függvény meghívásával végzi el. Ez a függvény végignézi az összes elemet a kimenetben és ha annak a valószínűsége kisebb mint egy előre megadott paraméter ezt a *confThreshold* változón keresztül tudjuk állítani, akkor azt törli a felismerések közül. Ezek után törli a kimenet elemei közül a redundánsakat, vagyis azokat a felmeréseket, amik esetleg többször is szerepeltek. Végül utolsó lépéskét az AI osztály *detect* függvénye végigmegy egyenként a már szűrt kimenet elemein, vagyis a felismert objektumok listáján és két műveletet végez el. Elsőnek az eredeti képkockára kirajzolja az objektum köré a piros négyszöget a *drawPred* függvénynek meghívásával. Másodiknak pedig közzé teszi az adott felismerést a többi ROS node számára a *send_data* függvénnyel. Ezt követően befejezi a futását az AI osztály *detect* függvénye és visszaadja a már módosított képkockát a körberajzolt objektumokkal az image_converter osztálynak, aki majd ezt közzé teszi a többi ROS node számára.

5.5 Kommunikáció bemutatása

A vke_ai és a vke_commander_follow komponenseknek szükséges valamilyen módon kommunikálniuk egymással. A vke_ai által felismert objektumok koordinátáit el kell juttatni a vke_commander_follow-hoz, hogy ennek alapján tudja irányítani a drónt. A ROS segít nekünk, hogy a ROS node-ok tudjanak kommunikálni egymással, viszont nekünk kell megadni az üzenet formátumát ehhez. Mind vke_ai, mind a vke_commander_follow komponensben futó python program ROS node-ként fut, tehát a ROS segíteni fog a kommunikáció lebonyolításában, de nekünk kell a kommunikáció során használt üzenet formátumot definiálnunk. A kommunikáció során az alábbi öt adatot kell mindenképpen elküldeni.

- Az objektum x koordinátája
- Az objektum y koordinátája
- Az objektum köré rajzolható négyszög szélessége
- Az objektum köré rajzolható négyszög magassága
- A felismert objektum azonosítója

Mind az öt adat leírható egy egész számként, így minimum egy öt elemű egész számokból álló tömböt kell használni egy felismert objektum adatainak az elküldésére. A koordináták elküldését a 4.1.3 fejezetben bemutatottaknak megfelelően készítettem el.

A nemrég bemutatott öt elemű egész számokból álló tömböt használtam és minden képkocka feldolgozása után küldök egy ilyen tömböt. Ha nincs felismert objektum, akkor öt darab mínusz egyet küldök. Ez azért jó megoldás, mivel az öt szám közöl mindegyik, minden esetben pozitív egész szám. A koordináták a kép koordinátái és ezek a kép bal felső sarkától nőnek a kép jobb alsó sarkáig. A felismert objektum köré rajzolt négyszög méretei is mindig pozitív egész számok lesznek. Az objektumok azonosítói, amiket a YOLOv4 fel tud ismerni is nullától kezdődő pozitív egész számok, a nullás azonosítójú az ember objektum. Így, használhatok mínusz egyet annak jelölésére, hogy nincs felismert objektum.

A felismert objektumok koordinátáit egyesével küldöm el, vagyis minden képkockán minden felismert objektum esetén küldök egy öt elemű egész számokból álló tömböt.

Minden felismert objektum koordinátáját elküldöm. Majd a másik fél aki feldolgozza ezeket, kiszűri a számára fontos objektumok adatait.

Egy fontos kritérium, hogy az üzenet típusát mind a két félnek ismernie kell. Ebben az esetben a vke_ai-nak és a vke_commander_follow-nak is. Ha a két komponens egy számítógépen lenne, ezzel nem kellene külön foglalkozni. A mi esetünkben a vke_ai és a vke_commander_follow két külön konténerben helyezkednek el, ezért valahogyan biztosítani kell, hogy az előbbiekben bemutatott két üzenet típust mind a két fél ismerje.

Ahhoz, hogy egy üzenetet ismerjen a ROS az adott számítógépen több lépés is szükséges. Elsőnek létre kell hozni egy „msg” kiterjesztésű fájlt, amely tartalmazza az üzenet felépítését, a mi esetünkben az öt elemű tömböt. Be kell állítani két konfigurációs fájlt, a CmakeLists.txt és a package.xml nevű fájlokat, hogy elmondjuk a ROS-nak, hol találja az üzenetünk felépítését tartalmazó „msg” kiterjesztésű fájlt,

valamint jelezzük a ROS-nak, hogy saját típusú üzenetet szeretnénk használni és generálja le nekünk az ehhez szükséges fájlokat. Végül egy „catkin_make” parancs kiadását követően legenerálódnak a kívánt fájlok és már használhatjuk az általunk készített új üzenet típust.

Látható, hogy három fájl is szükséges, hogy tudjuk használni az új üzenet típusunkat. Így, mind a vke_ai, mind a vke_commander_follow konténereknek ismernie kellene ezt a három fájlt ahhoz, hogy tudjanak ilyen típusú üzenetekkel kommunikálni. Megtehetjük, hogy egyszerűen átmásoljuk ezeket a fájlokat és így a vke_ai-hoz és a vke_commander_follow-hoz tartozó fájloknál is van egy másolat ebből ebből a három fájlból. Ez nem egy elegáns megoldás és sok probléma lehet ha két fájl nem ugyanaz, például a vke_ai konténerben lévőn változtatunk valamit, de a vke_commander_follow konténerben elfelejtjük ezt átírni és a két konténer nem fog tudni egymással kommunikálni. Egy elegáns megoldás, ha csak egy másolat van ennek a három fájlnak, a vke_ai konténerhez tartozó fájloknál és a vke_commander_follow-nál pedig csak szimbólikus linkek vannak a vke_ai-nál lévő három fájlra. Így pont olyan lesz mintha a vke_commander_follow-hoz tartozó fájloknál is ott lenne a három fájl, de igazából csak szimbolikus linkek vannak. Ennek segítségével meg tudom oldani, hogy mindkét konténer ismerje ezt a kommunikációhoz szükséges három fájlt.

Ahhoz, hogy tényleges tudjanak kommunikálni, a konténerek indításánál még fel kell csatolnunk ezt a három fájlt mind két konténerhez és majd az indításnál mind a kettőben le kell futtatni a „catkin_make” parancsot. Ezek után már minden készen áll, hogy a két konténer a vke_ai és a vke_commander_follow kommunikáljanak egymással.

5.6 vke_commander_follow komponens bemutatása

5.6.1 Komponens bemenete és kimenete

A vke_commander_follow komponens a vke_ai komponensből érkező koordinátákat dolgozza fel. Kikeresi belőle az emberre vonatkozó koordinátákat és elemzi azokat. Ezt követően döntést hoz, hogy merre kell irányítani a drónt, majd a drónt irányító parancsokat közzé teszi a többi ROS node számára. A vke_commander_follow ROS node-ra fog majd rákapcsolódni egy másik ROS node és ő fogja ezeket az irányító parancsokat Mavlink üzenetként elküldeni a drónnak. Így a vke_commander_follow komponensnek egy bemenete van a vke_ai komponenstől érkező koordináták és egy kimenete a drónt irányító parancsok.

5.6.2 Komponens részeinek a bemutatása

A következő alfejezetben a vke_commander_follow különböző részeit mutatom be: mi történik a Dockerfile-ban, a docker-entrypont.sh-ban, a compile.sh-ban, illetve a vke_commander_follow.sh-ban, és milyen nagyobb egységekből osztályokból épül fel a drón vezérlését végző python fájl.

A Dockerfile nevű fájlban írtam le, hogyan épüljön fel a vke_commander_follow komponenshez szükséges docker image. A

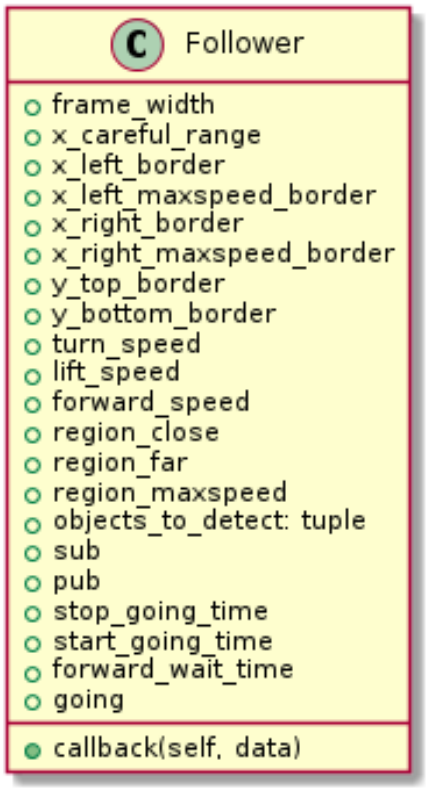
vke_commander_follow docker image-e egy ROS noetic-et tartalmazó docker image-ből indul ki és tartalmazza a ROS és python együtt működéséhez szükséges csomagot.

A docker-entrypoint.sh fájlban először lefuttatom a „catkin_make” parancsot, aminek a vke_ai és a vke_commander_follow komponensek kommunikációjában van szerepe, (erről bővebben a 5.5 alfejezetben olvashatunk) és elindítom a drón irányítását végző python programot. Így amikor elindítom a konténert akkor automatikusan lefut a docker-entrypoint.sh, ami pedig automatikusan elindítja drón irányításáért felelős python fájlt. Így összességében a konténer elindításával automatikusan a drón irányítása is elindul.

A compile.sh fájlban megadom a docker image elkészítéséhez szükséges parancsot és az elkészült docker image-et is vke_commander_follow-nak neveztem el.

vke_commander_follow.sh fájlban adom meg, hogy pontosan, hogyan induljon el a konténer. Beállítom a konténer ip címét, hogy a ros18 virtuális hálózathoz csatlakozzon, milyen könyvtárakat csatoljon a konténerhez, milyen ip címen találja roscore-t.

A follow_objects.py python program felelős, a felismert objektumok koordinátáinak a fogadásáért, szűréséért, valamint a drón irányításáért, vagyis milyen koordináta esetén, hogyan mozogjon a drón. Az alábbi 5.9. ábra mutatja a follow_objects.py python programot alkotó egy darab osztályt. Látható, hogy ez az osztály egy darab callback nevű függvényből és nagyon sok tagváltozóból áll. Ezeken a tagváltozókon keresztül tudjuk a drón mozgását és objektumkövetését beállítani. Ennek a Follower nevű osztálynak a feladata, hogy ha érkezik egy koordináta azt elemezze és ennek megfelelően irányítsa a drónt. A Follower osztály pontos működését a következő fejezetben ismertetem.



5.9. ábra: A follow_objects.py python programot alkotót osztály

5.6.3 Vezérlés folyamatának a bemutatása

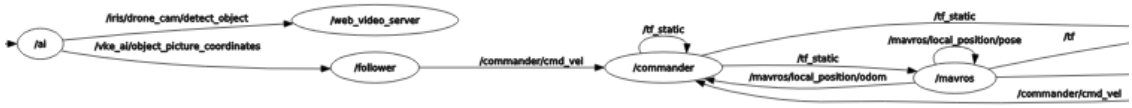
A vke_commander_follow feladata a felismert objektumok koordinátáinak fogadása, ezek közül azok kiválasztása, amelyek egy ember objektumhoz tartoznak, majd ezen szűrt koordináták alapján a drón irányítása. Ezen feladatokat a Follower osztály callback nevű függvénye valósítja meg.

A vke_ai és a vke_commander_follow komponensek közötti kommunikáció egy aszinkron kommunikáció és callback függvényeken keresztül valósul meg. Így, a follow_objects.py python programban amikor megadom, hogy jöjjön létre egy kapcsolat a vke_ai komponensben lévő ROS node-al, ahonnan a felismert objektumok koordinátáit megkapom, meg kell adni, hogy melyik függvény hívódjon meg, ha egy új koordinátát tartalmazó üzenet érkezik. Ennek a függvénynek a korábban már említett callback nevű függvényt adtam meg.

A drón irányításának a folyamata a következőképpen működik. Először érkezik egy üzenet a vke_ai komponensben futó ROS node-tól. Ennek hatására automatikusan meghívódik a callback nevű függvény. A callback nevű függvény először megvizsgálja, hogy az üzenet milyen objektumnak a koordinátáit tartalmazza és csak akkor foglalkozik tovább az üzenettel, ha az egy ember objektum koordinátáit tartalmazza. Következő lépés során, ha már biztos, hogy egy felismert emberre vonatkoznak a koordináták, elkezdi vizsgálni a koordináták értékeit. Először megvizsgálja az x

koordinátát, hogy szükséges-e a drónnak valamelyik irányba fordulnia, ha igen, akkor az ehhez szükséges parancsot elmenti. Utána megnézi az y koordinátát, hogy szükséges-e a drónnak emelkednie vagy pedig süllyednie, ha igen akkor az ehhez szükséges parancsot elmenti. Végül a magasság és szélesség adatok alapján, kiszámolja a felismert objektum köré rajzolható négyszög területét és megvizsgálja, hogy szükséges-e előre mennie a drónnak, ha igen akkor az ehhez szükséges parancsot elmenti. Utolsó lépésként a callback nevű függvény közzé teszi az elmentett irányító parancsokat a többi ROS node számára, így ezeket egy másik ROS node el tudja majd küldeni a drónnak Mavlink üzenetekként.

Az alábbi 5.10. ábra ábrán látható az objektum követés során futó fontos ROS node-ok és közöttük létrejött kapcsolatok. A körök jelképezik ROS node-okat. A körök között lévő vonalak pedig a ROS node-ok között létrejött kapcsolatokat jelképezik Látható, hogy az „/ai” ROS node, ami az objektumok felismerését végzi, megkapja a videofolyam képkockáit, ezt az ábrán az „/ai” csomópontba befutó nyílnak a vége jelzi. Ezt a videofolyamot feldolgozza. A megváltoztatott képkockákat a bekeretezett objektumokkal elküldi a „/web_video_server” nevű ROS node-nak a "/iris/drone_cam/detect_object" nevű kapcsolaton keresztül és ezen a ROS node-on keresztül tudjuk majd egy böngésző ablakon keresztül megnézni ennek a megváltoztatott videófolyamnak a tartalmát. A felismert objektumok koordinátáit az általunk definiált üzenet segítségével elküldi a „/follower” ROS node-nak egy "/vke_ai/object_picture_coordinates" nevű kapcsolaton keresztül. A „/follower” ROS node feldolgozza az üzeneteket és a mozgó ember koordinátái alapján eldönti, hogyan kell irányítani a drónt, majd az irányításhoz szükséges parancsokat elküldi a „/commander” ROS node-nak egy "/commander/cmd_vel" nevű kapcsolaton keresztül. A „/commander” ROS node, pedig a korábbi fejezetekben már említett „/mavros” nevű ROS node segítségével fogja ezeket az üzeneteket átalakítani Mavlink üzenetekké és elküldeni a drónnak.



5.10. ábra: Objektum követés során futó fontos ROS node-ok és a köztük lévő kapcsolatok

A callback függvény működésének bemutatása során a koordináták megvizsgálására nem tértem ki részletesebben. Ezt a következő részben ismertetem. A 4.1.4-es alfejezetben ismertettem, hogy a koordináták alapján, mikor hogyan kell irányítani a drónt, hogy követni tudja az objektumot, ebben az esetben a mozgó embert. Elkezdtem vizsgálni a vke_ai komponensből érkező üzenetekben lévő koordinátákat, hogy egy bizonyos objektum esetén, hogyan változnak az értékek, ha drón egy helyben áll. Ezek az értékek szinte folyamatosan változtak, egy érték körül ide-oda mozogtak. Így, mindenképpen érdemes a 4.1.4-es alfejezet végén lévő ötletet használni.

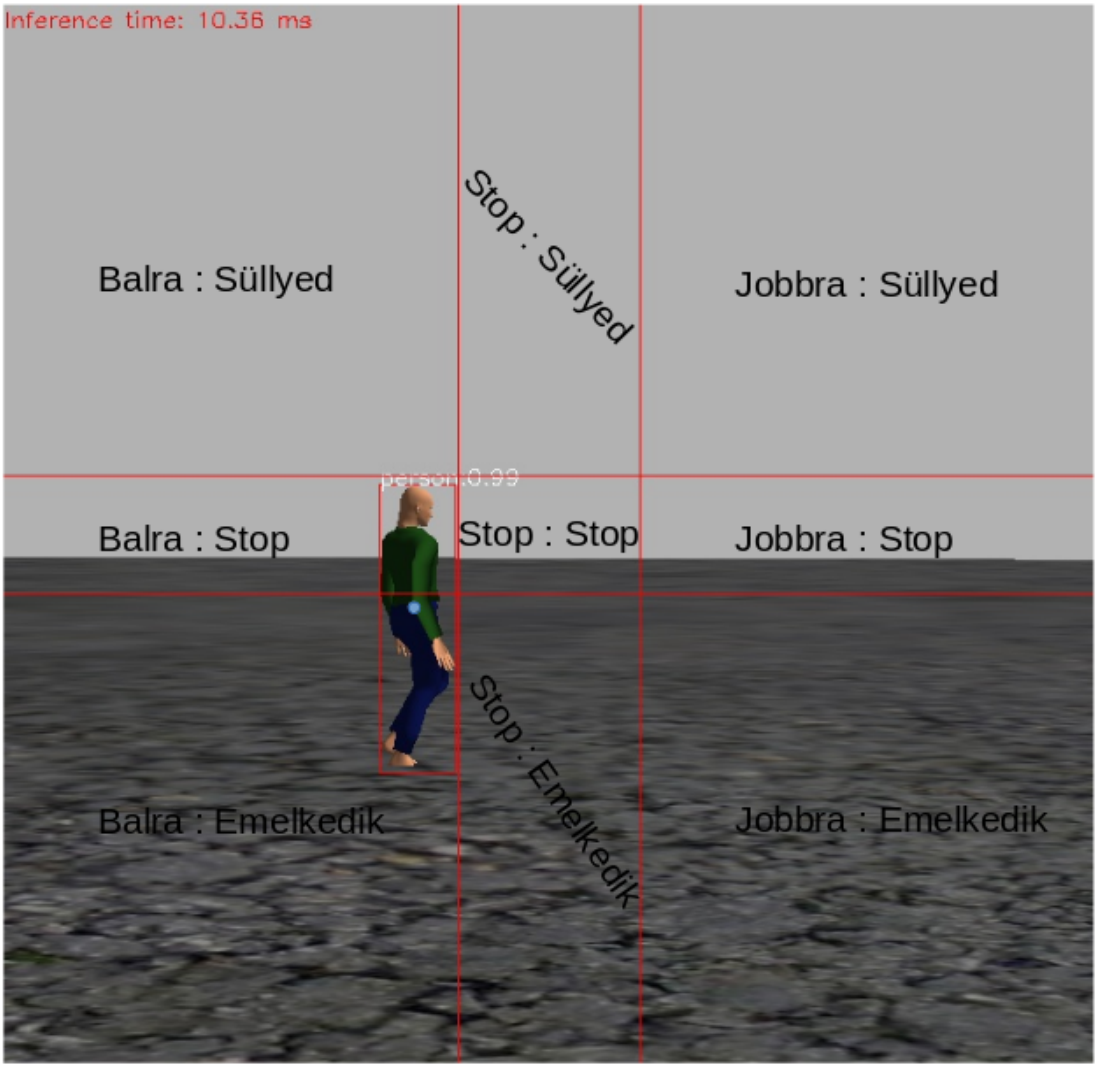
A koordináták értékeit nem egy konkrét értékhez hasonlítom, hogy ahhoz képes kisebb vagy nagyobb, hanem egy tartományhoz és ha egy tartományon belül van

valamely koordináta, akkor az ahhoz tartozó mozgást nem csinálja. Ha nem tartományhoz viszonyítanánk, akkor nem lenne olyan állapot amikor a drón nem csinál semmit. A drón akkor is ide-oda vagy fel-le mozogna, amikor már megfelelően beállította magát az objektumhoz képest és nem kellene csinálnia semmit. Ez az eset az álló emberrel végzet kísérletek során derült ki.

Az alábbi 5.11. ábra mutatja, hogy attól függően, hogy hol található a felismert mozgó ember középpontja különböző manővereket kell elvégezni a drónnal. Minden manőver jelölés az ábrán két részből áll és kettősponttal van elválasztva. Az első rész a drón vízszintes irányú mozgását jelöli, vagyis merre kell forognia vagy kell-e egyáltalán forogni. A második rész a drón függőleges irányú mozgását jelöli, vagyis kell-e emelkedni, süllyedni, vagy nem kell csinálni semmit. Az ábrán lévő ember középpontja, melyet a piros négyszög közepén elhelyezkedő kék pont jelöl, a bal alsó sarokban van, így a drónnak emelkednie kell és jobbra fordulnia. Az alábbi 5.11. ábra vke_ai komponens kimenete és arra írtam, rá, hogy az egyes tartományokban mit kell csinálnia a drónnak. Az ábrán látható függőleges és vízszintes vonalakat is a vke_ai komponensben rajzoltam rá, hogy segítse a drónt irányító algoritmusnak a fejlesztését. Ezek a függőleges és vízszintes vonalak megegyeznek a vke_commander_follow komponensben vizsgált tartományokkal.

A 5.11. ábra, azt szemléltette, hogy a képet különböző tartományokra osztottam és attól függően, hogy hol helyezkedett el a képen az ember, a drónnak különböző mozgásokat kellett végeznie. Ezek a tartomány vizsgálatok a drón előre és hátra mozgására nem vonatkoztak. Ezek a mozgások továbbra is a korábbi 4.1.4 fejezetben leírtaknak megfelelően működnek vagyis előre megy míg az ember köré rajzolt négyszög területe egy bizonyos nagyságot el nem ér és azt követően megáll a drón. A drón előre mozgása során csak két állapot van, mozog előre és megáll, míg a vízszintes mozgás esetén például három, jobbra mozog, balra mozog és megáll. Ezért jó megoldás, hogy csak egy konkrét értékhez képest nézem a területet és nem egy tartományhoz képest.

Összegezve, a drón mozgását irányító logikát ismertettem, már csak az a kérdés, hogy az ehhez szükséges határértékeket, amikor valami más mozgást kell csinálnia a drónnak az előzőhöz képest, hogyan tudom meghatározni. Ezt fogom ismertetni a következő fejezetben.



5.11. ábra: A drónnak milyen manővert kell csinálnia attól függően, hogy hol látja a mozgó embert.

5.6.4 Vezérléshez a küszöb értékek meghatározása

A drón vezérlését végző algoritmust az előző alfejezetben ismertettem, de szükség van még a határértékek meghatározására, amikor a drónnak valamit változtatnia kell a mozgásán, például eddig jobbra fordult de most már középen van a mozgó ember, így nem kell tovább forogni. A drón különböző mozgásainak a sebességét is meg kell határozni .

Elsőnek a sebesség meghatározását ismertetem. A drón ideális sebességének a meghatározásában a teleop_twist_keyboard[26] nevű ROS node volt a segítségemre. A teleop_twist_keyboard ROS node-al egy parancssori felületen keresztül billentyű leütések segítségével drón irányító parancsokat tudok kiadni, ugyanazokat, amiket a vke_commander_follow komponens is kiad. Így, a laptopom billentyűzetéről tudtam irányítani a szimulációban a drón mozgását. Miközben a drónt irányítottam és beállítottam egy viszonylag lassú sebességet, figyeltem a teleop_twist_keyboard ROS

node által kiadott drón irányító parancsokat és le tudtam olvasni, hogy az általam beállított sebesség mekkora értéknek felel meg. Ezt a sebességbeállítást és leolvasást elvégeztem mind a forgás, mind az előre mozgás és mind az emelkedés-süllyedés ideális sebességének a meghatározására. Az alábbi 2. táblázatban láthatóak a mért értékek. Ezek az értékek mind pozitívak. Ennek az értéknek a megadásával a drón az egyik irányba fog elindulni, ha ennek a negatív megfelelőjét vagyis a mínusz egyszeresét adom meg a drónnak, akkor a másik irányba indul el. Így, ha forgásnak a 2. táblázat szerint „0.492567”-t adok meg akkor az egyik irányba kezd el forogni, míg ha „-0.492567” akkor pedig a másik irányba forog ugyanakkora sebességgel. A teleop_twist_keyboard-ról további információk itt olvashatóak: [27]

2. táblázat: A drón különböző sebebesség értékei

Forgás	**Előre mozgás**	**Fel-le mozgás**
0.492567	0.540855	0.360855

A következő részben a különböző határértékek meghatározásának a folyamatát ismertetem. Először az előre mozgás befejezésének a határértékét határoztam meg, ez egy területérték lesz. Ennek során beállítottam a drónt az álló emberhez képest egy nem túl közeli távolságra és leolvastam az ember köré rajzolt négyszög szélességét és magasságát. Az ebből számított terület lesz a az a határérték, aminél ha közelebb ér a drón az emberhez, akkor megáll. Az alábbi 3. táblázatban láthatóak a mért értékek.

3. táblázat: A drón előre mozgása során figyelt határérték meghatározása

Négyszög szélesség [pixel]	80
Négyszög magasság [pixel]	450
Számított terület	36000
Szimulációba drón és ember távolsága [méter]	4,97

Ezt követően a forgás és az emelkedés-süllyedés során megfigyelt tartományokat határoztam meg. Ennek során ezeket beállítottam egy kezdeti kis értékre és a mozgó emberrel végeztem kísérleteket. A kísérletek során ha drón akár vízszintes akár függőleges irányba nem tudta beállítani magát a tartományon belülre, hanem mindig túlfutott rajta és csak egy ide-oda vagy fel-le mozgást végzett, akkor szélesítettem ezeken a tartományokon. Ezt a szélesítést addig végeztem, amíg a drón biztonságosan ezen tartományokon belülre tudta pozicionálni magát. Ezek a

tartományok láthatóak grafikusan ábrázolva a videofolyam egy képkockáján az előző alfejezet 5.11 ábráján. Az alábbi 4. táblázatban láthatóak ezek az értékek számszerűen is kifejezve.

4. táblázat: Függőleges és vízszintes tartományok határai

	Tartomány alsó határa	Tartomány felső határa
Vízszintes [pixel]	300	420
Függőleges [pixel]	320	400

A fenti mérések segítségével meghatároztam a drón mozgásának sebességét a különböző irányokba, valamint a határértékeket, melyek átlépése során a drónnak változtatni kell a mozgásán.

6 Tesztelés, értékelés

6.1 Tesztelés során szerzett tapasztalatok

Az objektum követés, vagyis a vke_ai és vke_commander_follow komponensek futtatása során kiderült egy meglepő probléma, amire nem számítottam. Az objektum követés nem volt megfelelően pontos, a szimulált drón sokszor elvesztette a mozgó embert. Ennek alapján a vke_commander_follow algoritmusába beépítettem egy-két kiegészítést. A következő alfejezetekben ezeket fogom ismertetni.

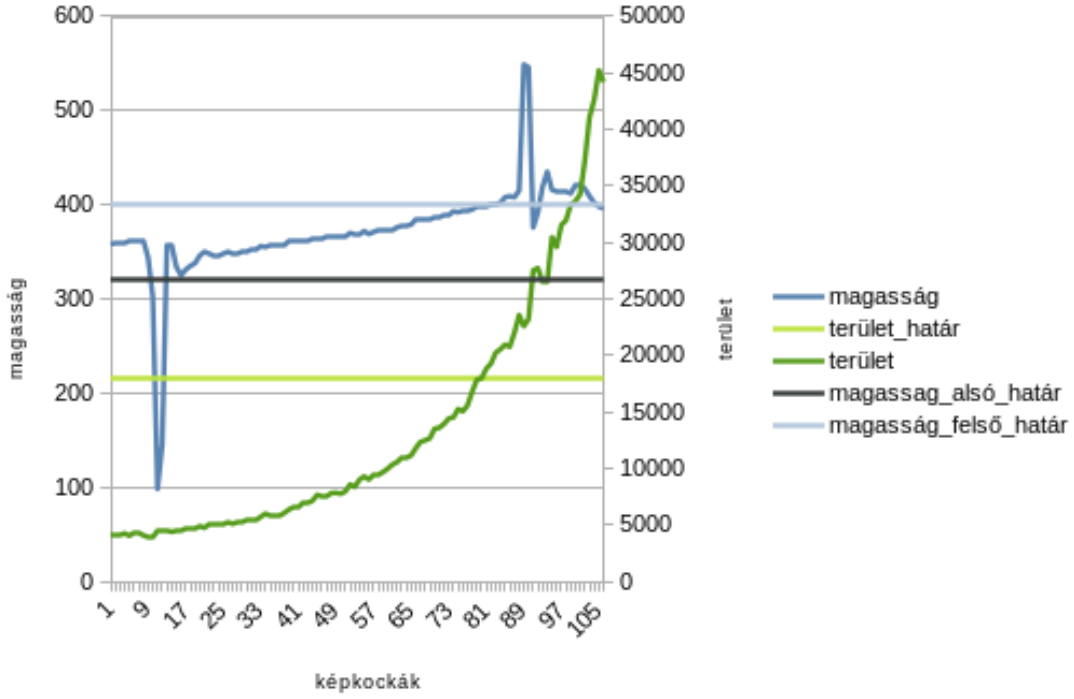
6.1.1 Drón indulás során megdől előre

Az objektum követés futtatása során egy meglepő problémába ütköztem amire az elején nem számítottam. Mikor egy drón előre repül, kissé megdőlés ez a szimulált drónra is igaz. Nagyobb probléma, hogy mikor a drón elindul előre, az egész drón kamerával együtt nagyon bedől lefelé és utána jön vissza a kissé bedőlt állapotban, majd indul el ténylegesen előre. Ez a bedőlés a kamerán is meglátszik és egy bedőlés során az egész kamerakép hirtelen felugrik aztán vissza. Ez a kamera kép alapján történő objektum követést nagyon megzavarja és erre az időre a vke_commander_follow pontatlan irányító parancsokat küld a drónnak. A drón megállása során is előkerül ez a probléma, csak akkor a bedőlés a másik irányba történik. Ugyanez a probléma a repülés sebességének a változása során is jelentkezik, és attól függően, hogy lassít vagy gyorsít a drón felfelé vagy lefelé dől be.

Az alábbi 6.1. ábra mutatja a drón objektum követése során rögzített adatokat. A kékes színű görbék a felismert ember középpontjának a magasságra vonatkoznak. Ezekre a magasság értékekre a bal oldali y tengely vonatkozik. A világoskék egyenes vonal a korábbi fejezetekben említett függőleges tartomány felső határát, míg a sötétkék egyenes vonal, ennek az alsó határát mutatja (lsd: 5.6.3 alfejezet). A drón igyekszik, hogy a magasságát ebben a tartományba tartsa, miközben közelít a mozgó emberhez . A kék vonal magának a magasságnak a változását mutatja az egyes képkockák között. Látszik, hogy alapvetően a megadott tartományban helyezkedik el két kiugrás kivételével és a végén, amikor kikerült a megadott tartományból elkezdi visszairányítani magát.

A zöldes színű görbék a felismert objektum köré rajzolt négyszög területére vonatkoznak. Ezekre a terület értékekre a jobb oldali y tengely vonatkozik. A világos zöld egyenes egy terület határ értéket mutat. A drón mikor elkezdi közelíteni az embert ezt egy nagy sebességgel teszi (ennek értéke a 2. táblázatban látható) és mikor átlépi a zöld vonallal jelzett határ értéket visszaveszi a sebességet a felére és óvatosabban közelíti az embert, hogy időben meg tudjon állni, mikor már elérte a megfelelő közelséget. A zöld színű vonal pedig a terület változását mutatja az egyes képkockák függvényében. A terület négyzetesen változik a távolság függvényében és mivel a drón minden képkockával közelebb kerül az objektumhoz így a képkockák függvényében is négyzetesen változik. Ez a négyzetes növekedés látható a zöld görbe esetén.

Ezen az ábrán (6.1. ábra) a kék színű görbe két kiugrást is mutat. Az első kiugrás lefelé, míg a második kisebb kiugrás felfelé történik. Ezek a kiugrások a drón bedőlése miatt történnek. Az első kiugrás során indul el a drón előre. Ekkor kezd el növekedni a zöld színű görbe is, ez a két esemény nem pont egyszerre történik a különböző késleltetések miatt. A második kiugrás során kezd el lassítani a drón. Ekkor keresztezi a zöld görbe a világos zöld görbét, sajnos a késleltetés miatt ezek sem pont egyszerre történnek, a terület érték először átlép egy határt, majd kis késleltetéssel reagál rá a drón és ekkor látható a kék görbén a második kiugrás. A zöld görbén nem láthatóak kiugrások sem az elindulás sem a sebességváltózást követően. Tehát a drón bedőlése a terület mérését nem zavarja. Megvizsgáltam és a felismert objektum középpontjának vízszintes elhelyezkedésének a mérését sem zavarja a bedőlés.



6.1. ábra: A drón repülése során a mozgó ember magassága és a köré rajzolt négyszög területének a változása

A drón bedőléséből adódó pontatlan értékek kiküszöbölésére a vke_commander_follow drónt irányító algoritmusának a fel-le mozgását szabályozó részét, a drón sebesség változását követően, egy időre letiltom, amíg a drón befejezi a bedőlő mozgást. Miután a vke_commander_follow drón irányító algoritmusát ezzel kiegészítettem a drón pontosabban követte a mozgó embert és kevesebbszer veszítette el.

6.1.2 Drón forgatásának finomítása

Az objektum követésénél egy gyakran előkerülő probléma volt, hogy amikor a drón már közelebb van a mozgó emberhez a forgással nem tudja megfelelően követni. Ha túl kicsire állítottam a drón forgási sebességét akkor nem fordult elég gyorsan az ember után és az ember egyszerűen kisétált a drón kamerájának a látószögéből. Ha

viszont túl nagyra állítottam a forgás sebességét akkor soha nem tudta a mozgó ember a középső tartományba pozicionálni, ahol abbahagyná a drón a forgó mozgást, hanem állandóan túlforog a nagy forgási sebesség miatt. Ennek a problémának a megoldására bevezettem egy második szélesebb tartományt a vízszintes irányba. Ha a drón ezen kívül van akkor nagy sebességgel forog (ennek értéke a 2. táblázatban látható), ha viszont a szélesebb tartományba kerül akkor lineárisan elkezdi csökkenteni a sebességét a szűkebb tartományig. Végül a drón befejezi a forgási mozgását amikor a szűkebb tartományba kerül . Ezzel kiegészítve a vke_commander_follow drón irányító algoritmusát a drón sokkal kevesebb alkalommal vesztette el a mozgó embert a kamerája látószögéből amikor közel volt a mozgó emberhez.

6.2 Kiértékelés

6.2.1 Az objektum követés eredményének bemutatása

Készítettem egy mérést, melynek során lefuttattam az objektum követő algoritmust és közben elmentettem a drón és az ember közötti a távolságot. Az alábbi 6.2. ábra mutatja, ahogy az elmentett távolság adatokat ábrázoltam a képkockák függvényében. Látható, hogy ez a távolság szépen egyenletes csökken, ahogy a drón közeledik a mozgó emberhez, míg a végén kissé 6 méter alatt a görbe egy vízszintes egyenessé változik, ahogy a drón elég közel ér hozzá. A 3. táblázat szerint azt állítottam be, hogy a drón akkor álljon meg, ha a közte és az ember között a távolság nagyjából 5 méter, így a 6 méter körüli távolság, melyet a mérés során kaptam pontos eredménynek számít.

Összegezve a drón követte a mozgó embert a kamerája segítségével készített videófolyam alapján.

6.2. ábra: A drón és az ember közötti távolság változása az egyes képkockák szerint

6.3 Továbbfejlesztési lehetőségek

Számos további fejlesztési lehetőség van a szakdolgozatom során ismertetett felhőből vezérelt objektumkövetéshez kapcsolódóan. Ilyen ahogy a szakdolgozatban is említettem, szimulált drón kicserélése egy valóságos drónra a drón által felismert álló objektumok elhelyezés egy térképen,így a drón azokhoz is oda tud navigálni. A mozgó objektum eltűnése esetén a drón elkezdheti keresni azt azok alapján, hogy hol látta utoljára. További lehetőségként lehet megemlíteni a szimuláció során fellépő késleltetés csökkentését megosztott memória használatával.

7 Összefoglalás

A félév során egy olyan komplex, korszerű, a kor modern informatikai eszközeit ötvöző objektum követő rendszert készítettem, amely képes egy felhő rendszerben futó szimulált drón mozgását oly módon irányítani, hogy a drón kövesse a szimulációban megjelenő mozgó embert. A rendszer tervezése és megvalósítása oly módon történt, hogy a szimulált drónt egyszerűen ki lehet cserélni egy valósságos drónra, valamint nem csak mozgó ember felismerésére és követésére alkalmas, hanem több mint ötven másik objektumot is képes követni.

Így, az élet számos területén felhasználhatjuk, ahol valamilyen objektum pontos követése a feladat akár beltéri akár kültéri környezetben.

8 Köszönetnyilvánítás

Köszönöm konzulensemnek és mentoromnak az évek során nyújtott rengeteg segítséget és útmutatást.

Köszönöm családomnak a szakdolgozatom kritikus átolvasását.

9 Irodalomjegyzék

[1] I. Goodfellow, Y. Bengio, és A. Courville, *Deep Learning*, Illustrated edition. Cambridge, Massachusetts: The MIT Press, 2016.

[2] R. Girshick, J. Donahue, T. Darrell, és J. Malik, „Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation”, in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, jún. 2014, o. 580–587, doi: 10.1109/CVPR.2014.81.

[3] J. Uijlings, K. V. D. Sande, T. Gevers, és A. Smeulders, „Selective Search for Object Recognition”, *Int. J. Comput. Vis.*, 2013, doi: 10.1007/s11263-013-0620-5.

[4] R. Girshick, „Fast R-CNN”, in *2015 IEEE International Conference on Computer Vision (ICCV)*, dec. 2015, o. 1440–1448, doi: 10.1109/ICCV.2015.169.

[5] S. Ren, K. He, R. Girshick, és J. Sun, „Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”, *IEEE Trans. Pattern Anal. Mach. Intell.*, köt. 39, sz. 6, o. 1137–1149, jún. 2017, doi: 10.1109/TPAMI.2016.2577031.

[6] J. Redmon, S. Divvala, R. Girshick, és A. Farhadi, „You Only Look Once: Unified, Real-Time Object Detection”, in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, jún. 2016, o. 779–788, doi: 10.1109/CVPR.2016.91.

[7] J. Redmon és A. Farhadi, „YOLO9000: Better, Faster, Stronger”, in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, júl. 2017, o. 6517–6525, doi: 10.1109/CVPR.2017.690.

[8] J. Redmon és A. Farhadi, „YOLOv3: An Incremental Improvement”, *ArXiv*, 2018.

[9] A. Bochkovskiy, C.-Y. Wang, és H. Liao, „YOLOv4: Optimal Speed and Accuracy of Object Detection”, *ArXiv*, 2020.

[10] W. Liu *és mtsai.*, „SSD: Single Shot MultiBox Detector”, 2016, doi: 10.1007/978-3-319-46448-0_2.

[11] C. Szegedy, S. Reed, D. Erhan, és D. Anguelov, „Scalable, High-Quality Object Detection”, *ArXiv*, 2014.

[12] S.-K. Weng, C.-M. Kuo, és S.-K. Tu, „Video object tracking using adaptive Kalman filter”, *J. Vis. Commun. Image Represent.*, köt. 17, sz. 6, o. 1190–1208, dec. 2006, doi: 10.1016/j.jvcir.2006.03.004.

[13] P. Chen, Y. Dang, R. Liang, W. Zhu, és X. He, „Real-Time Object Tracking on a Drone With Multi-Inertial Sensing Data”, *IEEE Trans. Intell. Transp. Syst.*, köt. 19, sz. 1, o. 131–139, jan. 2018, doi: 10.1109/TITS.2017.2750091.

[14] I. Marković, F. Chaumette, és I. Petrović, „Moving object detection, tracking and following using an omnidirectional camera on a mobile robot”, in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, máj. 2014, o. 5630–5635, doi: 10.1109/ICRA.2014.6907687.

[15] W. Mao, Z. Zhang, L. Qiu, J. He, Y. Cui, és S. Yun, „Indoor Follow Me Drone”, in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, New York, NY, USA, jún. 2017, o. 345–358, doi: 10.1145/3081333.3081362.

[16] K. E. Wenzel, A. Masselli, és A. Zell, „Visual tracking and following of a quadrocopter by another quadrocopter”, in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, okt. 2012, o. 4993–4998, doi: 10.1109/IROS.2012.6385635.

[17] T. Miyasaka, Y. Ohama, és Y. Ninomiya, „Ego-motion estimation and moving object tracking using multi-layer LIDAR”, in *2009 IEEE Intelligent Vehicles Symposium*, jún. 2009, o. 151–156, doi: 10.1109/IVS.2009.5164269.
[18] L. Joseph, *Learning Robotics using Python: Design, simulate, program, and prototype an autonomous mobile robot using ROS, OpenCV, PCL, and Python, 2nd Edition*, 2nd Revised edition. Packt Publishing, 2018.
[19] A. Koubâa, A. Allouch, M. Alajlan, Y. Javed, A. Belghith, és M. Khalgui, „Micro Air Vehicle Link (MAVlink) in a Nutshell: A Survey”, *IEEE Access*, köt. 7, o. 87658–87680, 2019, doi: 10.1109/ACCESS.2019.2924410.
[20] L. Meier, D. Honegger, és M. Pollefeys, „PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms”, in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, Seattle, WA, USA, máj. 2015, o. 6235–6240, doi: 10.1109/ICRA.2015.7140074.
[21] S. P. Kane és K. Matthias, *Docker: Up & Running: Shipping Reliable Containers in Production*, 2nd edition. Sebastopol, CA: O’Reilly Media, 2018.
[22] „xpra home page”. http://xpra.org/ (elérés dec. 01, 2020).
[23] „Display server”, *Wikipedia*. júl. 02, 2020, Elérés: dec. 01, 2020. [Online]. Elérhető: https://en.wikipedia.org/w/index.php?title=Display_server&oldid=965564877.
[24] M. Feilner, *OpenVPN: Building and Integrating Virtual Private Networks: Learn how to build secure VPNs using this powerful Open Source application*. Birmingham, 2006.
[25] A. Kaehler és G. Bradski, *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*, 1st edition. Sebastopol, CA: O’Reilly Media, 2017.
[26] „teleop_twist_keyboard - ROS Wiki”. http://wiki.ros.org/teleop_twist_keyboard (elérés dec. 07, 2020).
[27] *Mastering ROS for Robotics Programming - Second Edition Free eBook*. Packt Publishing, 2018.