eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Gradu Amaierako Lana

Informatika Ingeniaritzako Gradua

Konputazioa

Zabot: Zabor sailkatzaile robotikoa

Aimar Zabala Vergara

Zuzendariak
Elena Lazkano Ortega
Igor Rodriguez Rodriguez

2023.eko ekainaren 20

Laburpena

[EU] Zabot proiektua beso robotiko baten eta beharrezko sentsoreen instalazioa eta kontrola gauzatzean datza; inguruko zaborra hautemateko, jasotzeko eta zakarrontzi egokian uzteko gai den sistema bat garatzeko helburuarekin. Beso robotikoaren kontrola ROS plataformaren eta Moveit tresnaren bitartez egikaritu da, eta proiektua mundu fisikora ekarri aurretik sistemaren portaera Unity softwarean simulatu da. Zehazki, zaborraren manipulaziorako baliatu den robota Niryo etxeko Niryo Ned izan da. Inguruko objektuak antzematearren, robota ikusmen artifizialarekin hornitu da: Intel enpresak garatutako RealSense kamera kokatu da, eta kameraren irudien prozesamendua ikusmen artifizialeko teknika ezberdinekin gauzatu da, besteak beste, Deep Learning modeloak. Sistema hau Donostiako Informatika Fakultatean instalatu da.

[ES] El proyecto Zabot consiste en llevar a cabo la instalación y control de un brazo robótico y de los sensores necesarios, con el objetivo de desarrollar un sistema capaz de detectar la basura de la zona, recogerla y depositarla en la papelera adecuada. El control del brazo robótico se ha realizado a través de la plataforma ROS y la herramienta Moveit, y antes de traer el proyecto al mundo físico se ha simulado el comportamiento del sistema en el software Unity. En concreto, el robot que se ha utilizado para la manipulación de la basura ha sido Niryo Ned, de la empresa Niryo. Para detectar objetos de la zona, el robot se ha equipado con visión artificial: se ha ubicado la cámara RealSense, desarrollada por la empresa Intel, y el procesamiento de las imágenes de la cámara se ha llevado a cabo con diferentes técnicas de visión artificial, entre ellas, los modelos Deep Learning. Este sistema se ha instalado en la Facultad de Informática de Donostia.

[EN] The Zabot project consists of carrying out the installation and control of a robotic arm and the necessary sensors, with the aim of developing a system capable of detecting the garbage in the area, collecting it and depositing it in the appropriate bin. The control of the robotic arm has been carried out through the ROS platform and the Moveit tool, and before bringing the project to the physical world, the behavior of the system has been simulated in the Unity software. Specifically, the robot that has been used to handle the garbage has been Niryo Ned, from the Niryo company. To detect objects in the area, the robot has been equipped with artificial vision: the RealSense camera, developed by the company Intel, has been located, and the processing of the camera images has been carried out with different artificial vision techniques, among them, the Deep Learning models. This system has been installed in the Faculty of Informatics of Donostia.

Gaien aurkibidea

Irudien aurkibidea

Taulen aurkibidea

Kode zatien aurkibidea

Atala I

Sarrera

KAPITULUA 1

Sarrera

Beso robotikoak gizakiaren besoaren pareko funtzionalitateak dituzten robot estatikoak dira. Egun, fabriketan *pick-and-place* moduko ataza errepikakorrak gauzatzeko erabili ohi dira, hau da, piezak hartzeko eta behar den tokian kokatzeko. Baina batzuetan guztiz errepikakorrak ez diren edota adimen puntu bat eskatzen duten zereginak egikaritzeko behar dira eta, kasu hauetan, ingurunearen araberako portaera bat izatea eskatzen zaie. Hau da, robotak bere ingurunearen jabe izan behar du. Honetarako, ingurua antzemateko sentsoreak behar dituzte, baita sentsoreen irakurketak prozesatzeko softwarea eta logika ere.

Hain zuzen ere, Zabot proiektuan ingurunearen egoeraren araberako portaera duen beso robotikoa behar da. Izan ere, robota gertuko zaborra antzemateko eta sailkatzeko gai izan behar da, jasotzeko eta egokia den zakarrontzian uzteko *pick-and-place* ataza aurrera eraman dezan. Horrelako beso robotiko batek gizakiek sortutako hondakinen prozesamendua eta birziklapena eraginkortzen du: askotariko zaborra duten zakarrontzietako edota hondakinontzi okerrera bota diren hondakinak modu automatikoan sailkatzeko aukera ematen du, gero leku egokira eraman daitezen eta, ahal izanez gero, birzikla daitezen.

Aipatutako sistema garatzeko, proiektu honetan Niryo Ned[1] baliatuko da beso robotiko gisa, erabilpen erraza eta intuitiboa ("user-friendly") bermatzen baitu. Robotaren kontrola eta softwarearen garapena ROS[2] plataformaren bitartez gauzatuko da. Robota bere ingurunearen jabe izan dadin, ikusmen artifizialez hornituko da. Hau da, ingurunearen argazkiak ateratzeko Intel-en RealSense kamera bat instalatuko da, eta irudi hauek prozesatzeko teknika ezberdinak inplementatuko dira, besteak beste, adimen artifiziala. Dena dela, mundu fisikora ekarri aurretik, sistema osoa Unity[3] plataformako simulazio ingurune batean prestatuko da.

[1] Niryo Ned robotaren espezifikazio mekanikoak: URL
[2] ROS plataformaren webgune ofiziala: URL
[3] Unity softwarearen webgune ofiziala: URL

KAPITULUA 2

Artearen egoera (*State-of-the-art*)

Objektuak automatikoki lokalizatzea eta sailkatzea interes handiko eremua da ikusmen artifizial eta robotikoan. Hasieran, ingurunearen irudiak prozesatzeko adimen artifizialeko teknika klasikoak (erabaki-zuhaitzak, erregresio logistikoa, *K-Nearest Neighbours*) baliatzen ziren, baina azken urteetan Deep Learning modeloak eta teknikak gailendu dira. Gehien bat CNN (*Convolutional Neural Network*) [1] motako sareak nagusitu dira, hauen azkartasuna eta eraginkortasuna direla eta. Gainera, *Transfer Learning* bidez, aurretik entrenatuak izan diren sare handiak berrerabil daitezke.

Hain zuzen ere, Benjamin Ioller-en lanean [2] YOLO-V3 [3] modelo aurre-entrenatua erabiltzen da botilak antzemateko, eta haien diametro, pisu eta gogortasunaren arabera sailkatzen dira. R Aarthi et al. -en lanean [4], aldiz, zaborra materialaren arabera sailkatzen da, Mask R-CNN [5] motako DL modelo baten bitartez. Hau galtzadan edo espaloian utzitako hondakinen irudiekin entrenatzen dute.

Baina robotaren manipulazioaren ideia Takuya Kiyowaka et al. -en lanean [6] barneratzen da, zinta higikor batetik igarotzen diren hondakinak zakarrontzi egokira bidaltzeko beso robotikoaren logika garatzea baitute helburu. Ingurunearen propietateak direla eta, askatasun-gradu bakarreko beso robotikoa erabiltzea nahikoa dela ondorioztatzen dute. DL modelo bat erabili ordez, entrenatzeko azkarragoak diren metodoak erabiltzea proposatzen dute, hala nola, *Background synthesis*, *Seamless cloning* eta *Histogram matching*. Datu-bilketa zintaren gainean kokatutako kamera baten bitartez egiten dute. Hain zuzen ere, datu-basea errealitateko irudiak bilduz eta hauek automatikoki anotatzeko sistema bat garatuz sortzen dute. Probak 3 zabor motarekin egiten dituzte: aluminiozko latak, beirazko botilak eta plastikozko botilak.

Zabot-en kasuan, ordea, ingurune fisikoa Unity bidez simulatzea proposatzen da, datu-bilketa masiboak egiteko aukera izateko. Halaber, irudiak prozesatzeko CNNen arkitekturan oinarritutako Deep Learning modelo bat garatzen da, VGG16 [7] modelo aurre-entrenatutik abiatuz. Hortaz, DL modeloa simulazioan ateratako irudiekin entrenatzen da eta, ondoren, errealitatera eramaten da. Halaber, zaborraren posizioa estatikoa izatea suposatzen den arren, robotaren inguruan edozein tokitan aurki daiteke. Ondorioz, 6 askatasun-graduko Niryo Ned beso robotikoaren manipulazioa lantzen da. Zaborra 3 klase orokorragoetan banatzea proposatzen da ere, errealitateko birzkiklapen arauak errespetatuz: organikoa,

papera/kartoia eta plastikoa.

Atala II

Kudeaketa

KAPITULUA 3

Proiektuaren Helburuen Dokumentua

Kapitulu honetan Zabot proiektuaren irismena eta helburuak sakonki azaltzen dira. Proiektua ahalik eta zehatzenen banatu eta planifikatu da, garapen prozesua eramangarriagoa izan dadin. Hain zuzen ere, hasieratik amaierara aurrera eramango diren ataza guztiak dokumentatu dira, eta haien kronologia nahiz denbora kosteak aurreikusi dira. Horrez gain, proiektuan zehar gerta litezken arriskuak eta haien ondorioak analizatu dira. Lan-metodologiarekin zerikusia duten faktoreak ikertu eta esleitu dira ere.

3.1 Irismena

Zabot proiektuaren xedea inguruko zaborra hautemateko, jasotzeko eta zakarrontzi egokian uzteko gai den beso robotiko baten instalazioa eta kontrola gauzatzea da.

Kontrolatuko den beso robotikoa Niryo etxeko Niryo Ned izango da. Honen kontrola ROS (Robot Operating System) ingurunean garatuko da, eskaintzen dituen paketeen laguntzarekin. Proiektu honetan pakete aipagarriena MoveIt izango da.

Bestetik, robota inguruneko zaborra hautemateko gai izan dadin, ikusmen artifiziala baliatuko du. Hortaz, ingurunearen irudiak aterako dituen kamera bat instalatuko da. Kamera hau RealSense bat izango da, RGB irudiaz gain sakonerari buruzko informazioa eskaintzen baitu. Horrez gain, robotaren logikaren barruan, irudi hauek prozesatzeko ikusmen artifizialeko teknika ezberdinak aplikatuko dira, esaterako, Deep Learning modeloak.

Baina proiektua mundu errealera eraman aurretik, honen simulazioa Unity plataforman eraiki eta ebaluatuko da, kalitate minimo bat eskuratu arte. Hortaz, lanak Unity-ko simulazio ingurunetik errealitaterako trantsizio bat jasango du.

Prozesu honetan zehar sortzen diren fitxategiak GitHub biltegi batean gorde eta kudeatuko dira. GitHub plataformak proiektuak denboran zehar izan duen eboluzioa dokumentatuta izatea ahalbidetuko du. Biltegiaren izena “zabot” da, eta esteka honen bitartez atzi daiteke: https://github.com/aimarz/zabot

3.2 Helburu zehatzen deskribapena

3.2.1 Kalitate helburuak

Atal honetan proiektuaren irismenak barne hartzen dituen kalitate minimo, ertain nahiz handiko ezaugarriak azaltzen dira. Proiektua amaitutzat eman ahal izateko kalitate minimoa bermatzen duten baldintza guztiak bete beharko ditu. Maila ertaineko nahiz handiko baldintzak ez dira derrigorrezkoak izango, baina proiektuaren kalitatea hobea izatea eragingo dute. Helburua lanari kalitate handiko ezaugarriak eskaintzea izango denez, plangintza horren arabera diseinatu da. Baldintza horietan garapen prozesua eutsiezina dela ikusten bada, maila baxuagoko inplementazioa egingo da. Zehazki, hauek dira maila bakoitzaren ezaugarriak:

1. Kalitate minimoa:
    - Simulazioan soilik garatu da.
    - RealSense kamera eta adimen artifiziala erabiliz, inguruan objektu sinple bakarra duelarik, artefaktuaren kokapen informazioaren hurbilpen on bat emateko gai da. Sinpleak kubo formako objektuak kontsideratuko dira.
    - Robota detektatutako objektua hartzeko egin beharreko ibilbidea kalkulatzeko eta exekutatzeko gai da, errore batekin.
    - Hartutako objektuaren kolorearen arabera, kolore berekoa den zakarrontzian uzteko gaitasuna dauka, %70-eko asmatze-tasarekin.
2. Kalitate ertaina:
    - Simulazioan nahiz mundu fisikoan garatu da.
    - Objektu sinple anitzak izan ditzakeen ingurune baterako prestatu da.
    - Ordena irizpide bati jarraituz, piezak banaka banaka detektatu, hartu eta dagokion tokian uzteko egin beharreko ibilbideak kalkulatzeko gai da.
    - Objektuen sailkapena %85ean zuzena da.
3. Kalitate handia:
    - Ingurunean forma konplexuko objektuak (zaborra) antzemateko gai da.
    - Errealitateko birziklapen arauei jarraituz, antzemandako objektua zakarrontzi berdean (organikoa), urdinean (papera eta kartoia) edo horian (plastikoa) uzteko gai da, %80-eko asmatze-tasarekin.

3.2.2 Irisgarritasun helburuak

Proiektua amaitutzat eman ostean, dagozkion fitxategiak GitHub-en atzigarri utziko dira modu publikoan. Garapenean zehar eman diren aurrerapen garrantzitsuenak dokumentatuta egongo dira. Halaber, fitxategien erabilera zuzena bermatzeko erabilpen-gida bat igoko da biltegi berdinera. Proiektuaren irisgarritasuna ahalik eta zabalena izateko helburuarekin, aipatutako dokumentazioak ingelesez egingo dira. Horrela, beste garatzaile batzuek proiektuaz baliatzeko nahiz hobekuntzak egiteko aukera izango lukete. Bestela, hurrengo urteetako ikasleren batek proiektu hau har lezake abiapuntu gisa, eta proposatzen diren etorkizuneko lanak erreferentzi moduan hartuz, bere Gradu Amaierako Lan propioa garatu.

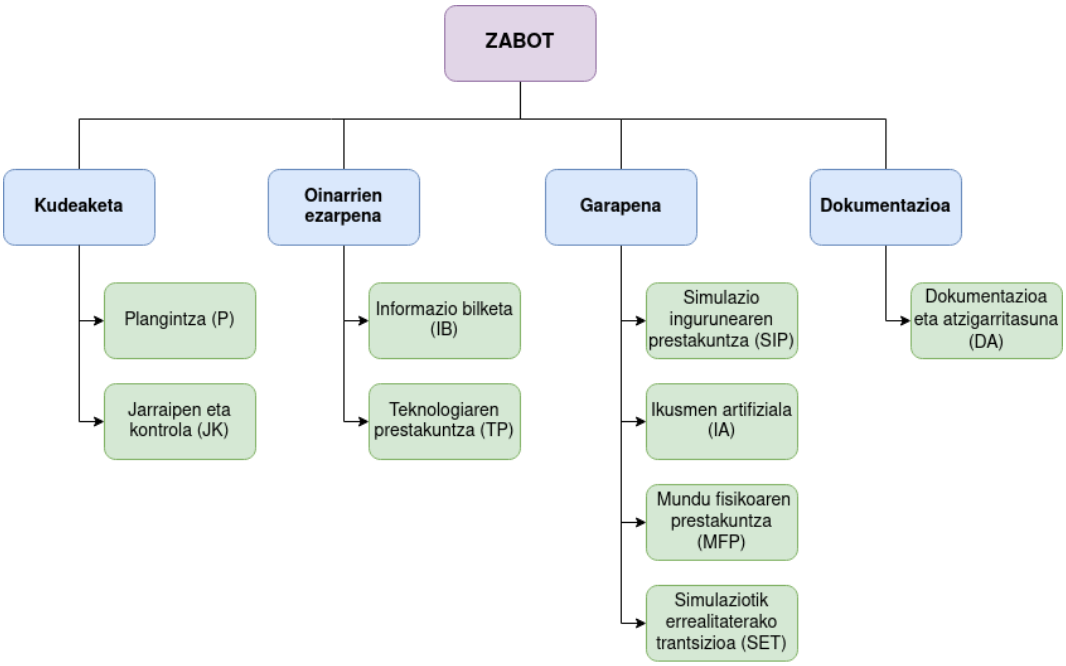
3.3 Interesatuak

Zabot proiektuarengan interesatuak honakoak dira:

- Proiektuaren egilea
- Proiektuaren zuzendariak
- RSAIT ikerketa taldea
- Garatzaileen komunitatea

3.4 Lanaren deskonposaketa egitura

Proiektuaren garapena hainbat adarretan eta lan-paketetan deskonposatu da (ikus 3.1 diagrama).



3.1 Irudia: LDE diagrama.

3.4.1 Kudeaketa adarra

Adar honek proiektuaren kudeaketa eta kontrolarekin lotutako lan-paketeak barneratzen ditu.

3.4.1.1 Plangintza (P)

Plangintza lan-paketeak proiektuaren garapena hasieratik amaierara definituta izateko beharrezko zereginak hartuko ditu barne. Hortaz, hasierako plangintza sortzeko atazak izango ditu, baita plangintza egunean mantentzekoak ere, behar izanez gero.

3.4.1.2 Jarraipena eta kontrola (JK)

Jarraipena eta kontrola lan-paketeak proiektuaren garapena kontrolpean mantentzeko beharrezko zereginak barneratuko ditu, baita plangintzaren betetzea bermatuko dutenak ere. Beraz, proiektuaren garapen egokia bermatuko duten atazak edukiko ditu, nagusienak dedikazioaren jarraipena, kalitate kudeaketa eta epeen kontrola izanik.

3.4.2 Oinarrien ezapenaren adarra

Hemen proiektuaren garapen prozesuaren muinera iritsi aurretik landu beharreko lan-paketeak biltzen dira.

3.4.2.1 Informazio bilketa (IB)

Informazio bilketa lan-paketeak garapenean zehar beharko diren informazioa eta jakintza eskuratzeko atazak barneratuko ditu. Bereziki, erabiliko diren teknologiak aurretik menperatzeko helburua izango du. Horrez gain, gurpila berriro asmatu behar izan gabe, Zabot-en antzeko irismena duen beste proiektu batetik hastea bermatuko du, denbora aurrezte handi bat suposatuz.

3.4.2.2 Teknologiaren prestakuntza (TP)

Lan-pakete honetan proiektuan zehar baliatuko diren teknologiak prestatzeko eta haien egoera egokia bermatzeko atazak egongo dira. Hortaz, beharrezko softwarearen instalazioaz eta konfigurazioaz zentratuko da, baita robotaren hardware atalaren egokitasuna aztertzeaz ere.

3.4.3 Garapenaren adarra

Adar honek proiektuaren garapen prozesuaren muina osatzen duten lan-paketeak barneratzen ditu.

3.4.3.1 Simulazio ingurunearen prestakuntza (SIP)

Simulazio ingurunearen prestakuntza lan-paketeak Zabot proiektuaren ingurune birtualaren eraikuntzarekin lotutako zereginak hartuko ditu barne. Hortaz, robotarentzat eszena birtuala prestatzeko eta bertan probak egikaritzeko atazak edukiko ditu.

3.4.3.2 Ikusmen artifiziala (AI)

Ikusmen artifiziala lan-paketeak simulazioko robota adimen artifizialaz hornitzeko eta eszenako argazkiak ateratzeko sistemaren kudeaketarako zereginak edukiko ditu. Beraz, beso robotikoari bere ingurunea aztertzeko eta horren arabera portaera desberdinak izateko gaitasuna emateko helburua izango du.

3.4.3.3 Mundu fisikoaren prestakuntza (MFP)

Mundu fisikoaren prestakuntza lan-paketekoak izango dira ingurune fisikoaren prestakuntzari lotutako atazak. Zehazki, ataza horiek robotaren, kamera fisikoaren eta eszenako zakarrontzien nahiz objektuen instalazioa bermatuko dute.

3.4.3.4 Simulaziotik errealitaterako trantsizioa (SET)

Simulaziotik errealitaterako trantsizioa lan-paketearen helburua jada eraikitako ingurune birtuala mundu fisikora modu egokian eramatea izango da, egin beharreko aldaketa guztiak egikarituz eta trantsizioaren errorea ahalik eta gehien minimizatuz. Horrez gain, errealitatean funtziona dezaketen ikusmen artifizialeko beste teknika batzuen ikerketa eta inplementazioari lotutako atazak barneratuko ditu.

3.4.4 Dokumentazioaren adarra

3.4.4.1 Dokumentazioa eta atzigarritasuna (DA)

Lan-pakete honek proiektuaren garapen prozesua dokumentatzeko eta Interneten bere atzigarritasuna bermatzeko beharrezko atazak barneratuko ditu. Halaber, erabilpen-gida egokia idazteko eta tribunalaren aurrean proiektuaren defentsa prestatzeko zereginak izango ditu.

3.5 Atazen eta denboraldien analisia

3.5.1 Egin beharreko atazen deskribapena

Proiektua lan-paketeetan banatu den modu berean, lan-pakete bakoitza gauzatzeko egin beharreko ataza konkretuak zehaztu dira.

**Oharra: Plangintzak 2 egunerapen jaso dituenez, ataza berriak definitzen joan dira. Ataza berri hauek izartxoen bitartez ezberdintzen dira, eta izartxoen kopuruak zein unetan gehitu diren adierazten du. Esate baterako, **SET.4 ataza plagintzaren bigarren egunerapenean definitu da.*

- **<u>Kudeaketa adarra:</u>**
  - **Plangintza (P):**
    - P1. Proiektuaren irismena zehaztu.
    - P2. LDE diagrama garatu.
    - P3. Lan-paketeak eta atazak zehaztu.
    - P4. Atazen arteko menpekotasunen analisia.
    - P5. Kronograma sortu.
    - P6. Ataza bakoitzerako dedikazioa estimatu.
    - P7. Proiektuarengan interesatuak hauteman.
    - P8. Komunikazio sistemak adostu.
    - P9. Arriskuen analisia egin.
    - P10. Plangintza eguneratu, behar izanez gero.
  - **Jarraipena eta kontrola (JK):**
    - JK1. Proiektuaren garapenari buruzko informazio garrantzitsua jaso.
    - JK2. Plangintzarekiko desbideratze esanguratsuen eta sortzen diren arriskuen identifikazioa.
    - JK3. Proiektuaren kalitate baldintzen bermatzea.

JK4. Kudeaketa bilerak.

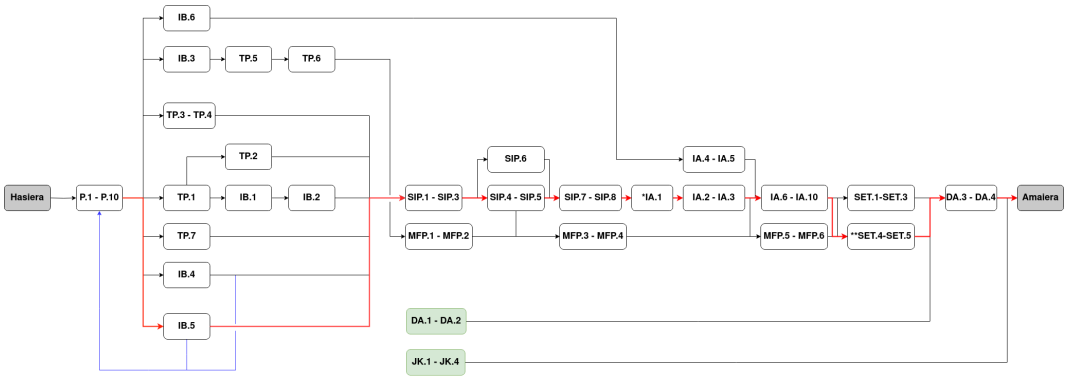
- **Oinarrien ezarpenaren adarra:**
  - **Informazio bilketa (IB):**
    IB1. ROSeko tutorialak aztertu.
    IB2. MoveIt paketeari buruzko informazioa jaso.
    IB3. Niryo robotaren zehaztapenak aztertu.
    IB4. Unity-ren web ofizialean erabilpen gidak begiratu.
    IB5. Antzeko irismena duten Unity proiektu libreak bilatu eta aztertu.
    IB6. Antzeko irismeneko Deep Learning modeloak aurkezten dituzten artikulu zientifikoak irakurri.
  - **Teknologiaren prestakuntza (TP):**
    TP1. ROS instalatu.
    TP2. Beharrezko ROS paketeak instalatu.
    TP3. Beharrezko Python paketeak instalatu.
    TP4. Unity instalatu.
    TP5. Niryo Studio instalatu.
    TP6. Niryo robot fisikoarekin proba sinpleak egin.
    TP7. GitHub biltegia prestatu eta konfiguratu.
- **Garapenaren adarra:**
  - **Simulazio ingurunearen prestakuntza (SIP):**
    SIP1. Antzeko Unity proiektu egokitik abiatu.
    SIP2. Proiektuaren tutoriala aurrera eraman.
    SIP3. Zakarrontzien eta objektuen 3D modeloak eskuratu.
    SIP4. Unity-ko eszena prestatu.
    SIP5. Zabot proiektuaren irismenera egokitzeko beharrezko moldaketak egin.
    SIP6. ROS eta Unity arteko komunikazio probak egin.
    SIP7. Ikusmen artifizialik gabe inplementazioa egin.
  - **Ikusmen artifiziala (IA):**
    *IA1. Posearen estimazioari lotutako Unity proiektutik ezaugarriak ekarri.
    IA2. RGB-D irudiak ateratzeko kamera kodetu eta eszenan kokatu.
    IA3. Eszena ausazkotzeko Randomizer-ak programatu.
    IA4. Ikusmen artifizialerako Deep Learning modeloa diseinatu.
    IA5. Diseinatutako modeloa inplementatu.
    IA6. Robotak hartu beharreko objektuak etiketatu.
    IA7. Modeloa entrenatzeko irudien eta etiketen datu-basea bildu.
    IA8. Entrenamendu prozesua diseinatu.
    IA9. Modeloaren entrenamendua aurrera eraman.
    IA10. Modeloaren eraginkortasuna ebaluatu.

- **Mundu fisikoaren prestakuntza (MFP):**
  - MFP1. Niryo robotaren kokalekua zehaztu.
  - MFP2. Robota instalatu.
  - MFP3. Zakarrontziak prestatu.
  - MFP4. Hartu beharreko objektuak prestatu.
  - MFP5. RGB-D kamera kokatu.
  - MFP6. Objektuak robotarentzat leku atzigarrian utzi.
- **Simulaziotik errealitaterako trantsizioa (SET):**
  - SET1. Garatutako ROS paketeak errealitaterako egokitu.
  - SET2. Robota kontrolatzeko ROS paketeak instalatu.
  - SET3. Simulazioan entrenatutako modeloarekin proba errealak gauzatu.
  - **SET4. Zaborra antzemateko ikusmen artifizialeko beste teknikak ikertu.
  - **SET5. Egokia den teknika bat inplementatu.

- **Dokumentazioaren adarra:**
  - **Dokumentazioa eta atzigarritasuna (DA):**
    - DA1. Memoria idatzi.
    - DA2. Proiektua GitHub-en atzigarri utzi, aldaketa dokumentatuekin.
    - DA3. Erabilpen-gida eta azalpen laburtua idatzi.
    - DA4. Proiektuaren defentsa prestatu.

3.5.2 Atazen arteko mendekotasunak

Proiektuaren zeregin eta jardueren menpekotasunak 3.2 diagraman jasotzen dira.



3.2 Irudia: Atazen arteko mendekotasunen diagrama.

Ezertan hasi aurretik, garbi dago proiektuaren plangintza garatzeari dagozkion jardueRak bete behar direla. Baina hori egin ostean, ataza multzo handi bat paraleloan garatzeko aukera sortzen da. Izan ere, informazio bilketa (IB) nahiz teknologiaren prestakuntza (TP) lan-paketeetan barne hartzen diren zeregin gehienak egiteko ez dago aurrebaldintzarik, eta askok ez dute menpekotasunik. Dena den, ezinbestekoa da argitzea IB.4 eta IB.5 atazen eta Plangintza (P) lan-paketeko jardueren arteko lotura (3.2 diagraman urdinez markatuta

dagoena). Izan ere, IB.4 eta IB.5 zereginak planifikatzeko P.1-P.9 atazak bete behar dira, baina, era berean, plangintza ezin da guztiz bukatu IB.4 eta IB.5 egin aurretik, hurrengo atazak definitzea eta haien denbora estimazioak ematea ezinezkoa izango litzatekeelako, informazio falta dela eta. Ondorioz, jarduera hauen artean menpekotasun ziklo bat dago, eta P.1-P.9 atazetara itzuli behar izatea eragingo du, plangintza eguneratu ahal izateko.

Behin IB eta TPko zereginak amaituta, proiektuaren garapena paraleloan mantentzeko aukera murrizten da, SIP lan-paketeko lehenengo atazek aurrebaldintza asko baitituzte. Ondoren, SIPko jardueren betekuntzak IA lan-paketean lanean hastea ahalbidetuko du. Hala ere, MFP.1-MFP.2 zereginei ekiteko nahikoa da TP.6 bukatzearekin, eta beraz, puntu bateraino paraleloan mantendu daitezke, MFPko beste ataza batzuekin batera. Behin aipatutako atazak amaituta, SET lan-paketeari ekin diezaioke. Azkenik, DA.3 eta DA.4 egin beharko dira, proiektua amaitutzat eman aurretik.

3.2 diagraman berdez koloreztatutako laukiak proiektuaren garapenaren prozesuan zehar egikarituko diren atazei dagozkie. Izan ere, proiektuaren jarraipena eta kontrola uneoro gauzatuko da, eta ez du aurrebaldintza garbirik izango. Era berean, memoriaren idazketa nahiz proiektuko aurrerapenen dokumentazioa pixkanaka beteko dira garapena aurrera doan ahala. Zeregin hauen gauzatzea ezinbestekoa izango da proiektuari amaiera ofiziala emateko.

3.5.3 Atazen garapen denboraldiak

Zabot proiektua 23 astetan zehar garatuko da. Hasiera urtarrilaren 16ean emango zaio, eta GrALaren entrega epea errespetatuz, ekainaren 25erako bukatuta eta dokumentatuta egongo da.

Atazen denboraldiei dagokienez, hasieran plangintza lan-paketeko zereginei emango zaie lehentasuna. Ondoren, garapenaren muinera iritsi aurretik, informazioaren bilketa eta teknologiaren prestakuntzari dagozkien atazak gailenduko dira. Gero, Unity plataforman simulazio ingurunea prestatzeari ipiniko zaio arreta. Honen ostean, simulazioko robota ikusmen artifizialarekin hornituko da. Azkenik, mundu fisikoan robotaren instalazioa eta ingurunearen prestakuntza gauzatuko dira, SET lan-paketea landu aurretik.

Jarraipena eta kontrola proiektuaren garapenaren prozesu osoan gauzatuko da. Memoria ere pixkanaka osatzen joatea da helburua, egindako aldaketa garrantzitsuenak momentuan dokumentatuz. Hala ere, erabiltzailearen gida eta GrALaren defentsaren prestakuntza amaieran egikarituko dira. Planifikatutako ataza guztien garapen denboraldi zehatza 3.1 irudiko Gantt kronograman bistaratzen da.

**Oharra: P.1 - P.4 motako idazkerak esplizituki azaltzen diren atazak eta haien artean daudenak barne hartzen direla adierazten du. Adibide horretan, P.1, P.2, P.3 eta P.4 atazei egiten zaie erreferentzia.*

3.5.4 Atazen dedikazio estimazioak

Hasieran, Zabot proiektua 300 ordutan aurrera eramatea espero zen, 12 kreditu adinako GrAL bat izatea espero baitzen. Baina garapenean aurrea joan ahala, plangintza 2 aldiz eguneratzeko beharra ikusi da. Izan ere, honakoa irismen oso zabaleko proiektu bat dela ikusi da eta, beraz, 300 orduak gainditzea espero da. Zehazki, bere osotasunean 329 orduko dedikazioa eskatuko duela planifikatu da.

	Urtarrila			Otsaila				Martxoa				Apirila				Maiatza					Ekaina		
Ataza	16	23	30	6	13	20	27	6	13	20	27	3	10	17	24	1	8	15	22	29	5	12	19
P.1 - P.6																							
P.7																							
P.8 - P.9																							
TP.1																							
IB.1 - IB.4																							
TP.2 - TP.4																							
TP.5 - TP.6																							
IB.5																							
TP.7																							
SIP.1																							
SIP.2																							
SIP.3																							
SIP.4 - SIP.6																							
P.10																							
SIP.7																							
*IA.1																							
IA.2																							
IA.3																							
IB.6																							
IA.4 - IA.5																							
IA.6 - IA.7																							
IA.8 - IA.10																							
MFP.1 - MFP.4																							
MFP.5 - MFP.6																							
SET.1 - SET.3																							
**SET.4																							
**SET.5																							
E450.1																							
E450.2																							
E450.3																							
DA.1																							
DA.2																							
DA.3																							
DA.4																							
JK.1																							
JK.2																							
JK.3																							
JK.4																							

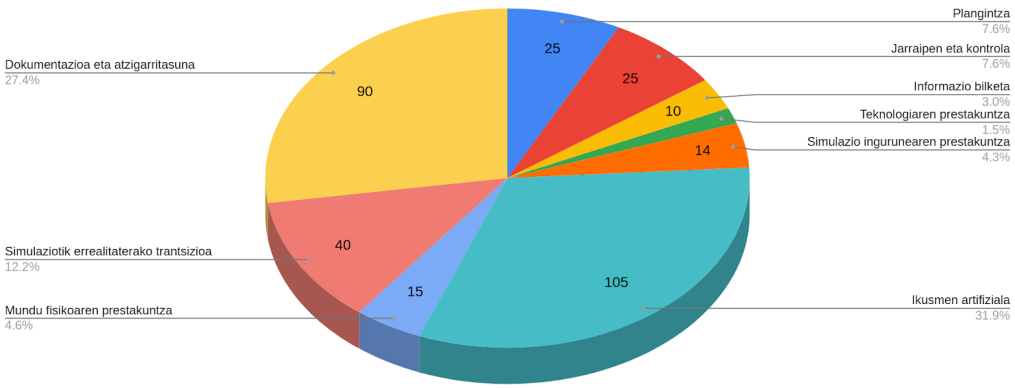
3.1 Taula: Proiektuaren garapenari dagokion planifikatutako Gantt kronograma.

Lan-pakete pisutsuena Ikusmen artifiziala izatea espero da, Deep Learning modeloen entrenamendu prozesuen iraupen luzea eta aurreikusgarritasun eza direla eta, batik bat. Era berean, dokumentazioa eta atzigarritasunari dagozkien atazei 90 orduko pisua eman zaie, erreferentzia gisa beste urteetako Gradu Amaierako Lanen plangintzak hartuz. Lan-pakete guztien dedikazio zehatzak 3.2 taulan ikus daitezke, eta proiektu osoarekiko duten proportzioa 3.3 diagraman bistaratzen da.

Lan-paketea	Orduak (h:mm)
Plangintza	25:00
Jarraipena eta kontrola	25:00
Informazio bilketa	10:00
Teknologiaren prestakuntza	5:00
Simulazio ingurunearen prestakuntza	14:00
Ikusmen artifiziala	105:00
Mundu fisikoaren prestakuntza	15:00
Simulaziotik errealitaterako trantsizioa	40:00
Dokumentazioa eta atzigarritasuna	90:00
PROIEKTUAREN DENBORA TOTALA	**329:00**

3.2 Taula: Lan-pakete bakoitzeko denbora estimazioak.

Ataza konkretuen denbora estimazioak 3.3 (Plangintza), 3.4 (Jarraipena eta kontrola), 3.5 (Informazio bilketa), 3.6 (Teknologiaren prestakuntza), 3.7 (Simulazio ingurunearen prestakuntza), 3.8 (Ikusmen artifiziala), 3.9 (Mundu fisikoaren prestakuntza), 3.10 (Simulaziotik errealitaterako trantsizioa) eta 3.11 (Dokumentazioa eta atzigarritasuna) tauletan ikus daitezke.



3.3 Irudia: Lan-paketeen denbora estimazioen diagrama.

PLANGINTZA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
P.1	2:00	**P.6**	3:00
P.2	3:00	**P.7**	1:00
P.3	4:00	**P.8**	1:00
P.4	4:00	**P.9**	3:00
P.5	3:00	**P.10**	1:00
GUZTIRA: 25h			

3.3 Taula: Plangintza lan-paketeko atazen denbora estimazioak.

JARRAIPENA ETA KONTROLA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
JK.1	5:00	**JK.3**	5:00
JK.2	5:00	**JK.4**	10:00
GUZTIRA: 25h			

3.4 Taula: Jarraipena eta kontrola lan-paketeko atazen denbora estimazioak.

INFORMAZIO BILKETA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
IB.1	1:30	**IB.4**	3:00
IB.2	1:00	**IB.5**	2:00
IB.3	0:30	**IB.6**	2:00
GUZTIRA: 10h			

3.5 Taula: Informazio bilketa lan-paketeko atazen denbora estimazioak.

TEKNOLOGIAREN PRESTAKUNTZA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
TP.1	0:30	**TP.5**	0:30
TP.2	0:30	**TP.6**	2:00
TP.3	0:30	**TP.7**	0:30
TP.4	0:30		
GUZTIRA: 5h			

3.6 Taula: Teknologiaren prestakuntza lan-paketeko atazen denbora estimazioak.

SIMULAZIO INGURUNEAREN PRESTAKUNTZA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
SIP.1	2:00	**SIP.5**	3:00
SIP.2	3:00	**SIP6**	1:00
SIP.3	2:00	**SIP.7**	2:00
SIP.4	1:00		
GUZTIRA: 14h			

3.7 Taula: Simulazio ingurunearen prestakuntza lan-paketeko atazen denbora estimazioak.

IKUSMEN ARTIFIZIALA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
***IA.1**	5:00	**IA.6**	6:00
IA.2	3:00	**IA.7**	5:00
IA.3	5:00	**IA.8**	2:00
IA.4	6:00	**IA.9**	62:00
IA.5	6:00	**IA.10**	5:00
GUZTIRA: 105h			

3.8 Taula: Ikusmen artifiziala lan-paketeko atazen denbora estimazioak.

MUNDU FISIKOAREN PRESTAKUNTZA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
MFP.1	2:00	**MFP.4**	3:00
MFP.2	3:30	**MFP.5**	3:00
MFP.3	3:00	**MFP.6**	0:30
GUZTIRA: 15h			

3.9 Taula: Mundu fisikoaren prestakuntza lan-paketeko atazen denbora estimazioak.

SIMULAZIOTIK ERREALITATERAKO TRANTSIZIOA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
SET.1	4:00	****SET.4**	5:00
SET.2	1:00	****SET.5**	27:00
SET.3	3:00		
GUZTIRA: 40h			

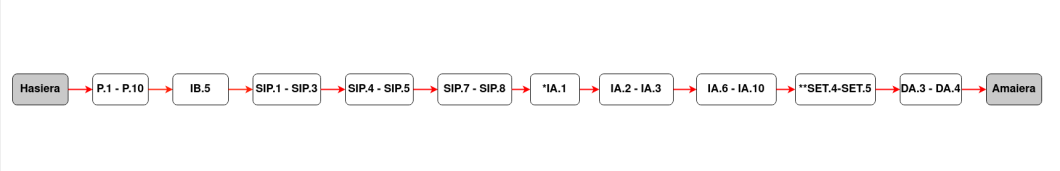
3.10 Taula: Simulaziotik errealitaterako trantsizioa lan-paketeko atazen denbora estimazioak.

DOKUMENTAZIOA ETA ATZIGARRITASUNA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
DA.1	70:00	**DA.3**	3:00
DA.2	5:00	**DA.4**	12:00
GUZTIRA: 90h			

3.11 Taula: Dokumentazioa eta atzigarritasuna lan-paketeko atazen denbora estimazioak.

3.5.5 Bide kritikoaren analisia

Atazen arteko mendekotasun diagrama eta zeregin bakoitza betetzeko beharko den denbora aztertu ondoren, proiektuaren bide kritikoa zein izango den iker daiteke. Bide hau proiektuaren amaitze-data ezarriko duen atazen sekuentzia izango da. Zabot proiektuaren kasuan, 3.4 irudian bistaratzen den sekuentziak osatzen du bide kritikoa.



3.4 Irudia: Proiektuaren bide kritikoaren diagrama.

3.4 diagraman ikusten den moduan, lan-pakete gehienetako ataza pisutsuenak hartzen ditu barne, hau da, aurrera eramateko denbora gehien eskatzen dutenak. Bide kritikoaren parte den atazaren baterako planifikatu baino denbora gehiago behar izanez gero, proiektuaren atzerapena gertatuko da. Baina, era berean, bideko zereginen bat azkarrago bukatu ezkero, proiektuaren amaitze-data aurreratuko litzateke.

Denbora dedikazioa estimatzeko atazarik zailena IA.9 izan da, modeloen entrenamenduen aurreikusgarritasun eza dela eta. Zeregin hau, denbora asko behar izateaz gain, bide kritikoan kokatzen da. Beraz, ezustekorik ez izatearren, ataza honi 62 ordu eskaintzea planifikatu da, behar baino denbora gehiago izango delakoan; nahiz eta proiektuak guztira 300 orduak gainditu.

Bestetik, bidearen hasierako zatian SIP.1 zeregina kokatzen da, hau da, Zabot-en antzeko irismena duen beste proiektu libre baten aukeraketa. Ataza hau proiektuaren garapenean oso garrantzitsua izango da, izan ere, aukeraketa ona egin ezkero hurrengo zeregin asko malguagoak bilakatuko dira, denbora gutxiago eskatuz. Aldiz, hasierako proiektu okerra hautatzen bada, lan gehiena zerotik egin beharko da eta garapen prozesua zailagoa izango da.

3.6 Lan-metodologia

Atal honetan proiektuaren garapen prozesua aurrera eramateko jarraituko den lan-metodologia azaltzen da.

Proiektuaren egileak urtarrila eta ekaina bitartean Zabot-i astean 15 ordu dedikatzeko konpromezua hartzen du. Dedikazio hori asteko 5 egunetan banatuko da, egun bakoitzean

3 ordu arituz. Orokorrean lanaldia astelehenetik ostiralerakoa izango da, baina oztoporik balego, gehienez 2 egun adinako lana asteburura mugitzeko aukera egongo litzateke. Horrez gain, proiektuan atzerapenik balego, hala beste edozein arrazoirengatik beharrezkoa ikusten bada, egilea prest egongo da egoera zuzendu arte egunero 4 ordu aritzera, 3 orduko lanaldiak egin ordez. Egileak ordu hauek goizeko 9:00etatik 12:00etara arte dedikatuko ditu, 13:00etara luzatzeko prest egonik.

Garapenean zehar egingo diren instalazioei dagokienez, EHU/UPVko Informatika Fakultatean gauzatuko dira. Robotaren behin betiko instalazioa Elena Lazkano zuzendariaren bulegoan egingo da. Beraz, mundu fisikoko exekuzioak bertan gauzatuko dira hein handi batean. Bestalde, nahiz eta proiektuaren zeregin asko egileak bere kasa landuko dituen, instalazioan zuzendarien laguntza nabarmena jasoko du. Beso robotikoa eta beharrezko beste dispositiboen horniketaz zuzendariak arduratuko dira.

3.7 Proiektuko informazio sistema eta komunikazio sistemaren berezitasunak

3.7.1 Informazio sistema

3.7.1.1 Egitura

Proiektu honetarako informazio sistema 2 azpiegituratan banatzen da:

- Hodeiekoa: Lehenengoa hodeian egongo da. Bertan biltzen den informazioa proiektuaren zuzendarientzat nahiz egilearentzat atzigarri egongo da uneoro. Azpiegitura honen barruan sartzen dira Google Drive plataforma nahiz Zabot proiektuaren GitHub biltegi publikoa. Google Drive-en kudeaketari lotutako fitxategiak gordeko dira, hala nola, proiektuaren plangintza. GitHub-en aldiz, simulazioa eraiki ahal izateko beharrezko edukiak bilduko dira, esate baterako, Unity-ko proiektuak eta ROS fitxategiak. Biltegia adar (*branch*) ezberdinetan banatuko da proiektuan fase aldaketa garrantzitsuak gertatzen diren ahala. Zehazki, 3 adar bananduko dira: *master* (simulazioa objektu sinpleekin), *real_garbage* (simulazioa errealitateko zaborrarekin) eta *real_world* (mundu fisikorako trantsizioa). GitHub plataformara igotzeko pisutsuegiak diren fitxategiak egongo balira, hauek Dropbox-era igoko lirateke.
- Lokala: Bigarren azpiegitura ordenagailu lokaletan kokatzen da. Hau proiektuaren egilearen eta zuzendarien ordenagailuek osatzen dute. Zuzendarien konputagailuan Niryo robot fisikoarekin komunikatzeko beharrezko fitxategiak eta paketeak egongo dira. Probak egin behar izanez gero, proiektuaren fitxategiak hodeieko GitHub biltegitik ekarriko dira. Egilearen ordenagailua, aldiz, proiektuan aldaketak eta aurrerapenak egiteko erabiliko da. Hori dela eta, une batzuetan egilearen konputagailu informazio lokala berriagoa izango da GitHub biltegian dagoena baino. Beraz, egilearen ardura izango da lanaldiaren ostean aurrerapen guztiak GitHub plataforman gordetzea.

Deep Learning modeloen entrenamenduak egilearen ordenagailu pertsonalean gauzatuko dira. Hauek dira makinaren oinarrizko espezifikazioak:

- **Prozesadorea:** Intel Core i5-7400 3.00GHz $\times$ 4

- **RAM memoria:** 16 GB
- **Txartel grafikoa:** NVIDIA GeForce GTX 1050 Ti 4 GB

3.7.1.2 Formatuak eta aldaketen dokumentazioa

Google Drive-eko dokumentu guztiek .pdf formatua izango dute, formatu honek atzipen azkarra eta irakurgarritasun ona bermatzen baitu. Irudiren bat partekatu behar izanez gero, *.jpg* edo *.png* formatuak onartuko dira soilik.

Aldiz, GitHub biltegian edozein formatuko fitxategiak onartuko dira, proiektuaren irismen zabala dela eta. Bertan egiten diren aldaketa guztiak modu egokian dokumentatuko dira. Kontuan izanik GitHub biltegia publikoa izango dela, ulergarritasuna ahalik eta gehien bermatzeko azalpen guztiak ingelesez emango dira. Amaieran, proiektua modu laburrean deskribatzen duen erabilpen-gida ere ingelesez idatziko da, eta "README.txt"izenarekin utziko da biltegian.

3.7.1.3 Segurtasun kopiak

Hodeieko informazioa galtzeko arriskua dela eta, uneoro segurtasun kopia lokalak mantenduko dira. Kopia hauek proiektuaren egilearen ordenagailu lokalean nahiz USB batean gordeko dira. Hortaz, uneoro aldaketa berrienekin eguneratu beharko dira, hondamendiren bat gertatzean galera minimoa izan dadin. USB bera bilera guztietara eraman beharko da, Interneteko azpiegituran egon litezken arazoak saihesteko.

3.7.2 Komunikazioa

3.7.2.1 Egilea eta zuzendarien arteko elkartrukea

Proiektuaren egilearen eta zuzendarien arteko informazio trukea posta-helbide elektroniko bitartez egingo da. Egileak komunikazioa azabala106@ikasle.ehu.eus helbidearen bitartez gauzatuko du. Zuzendarien kasuan, Elena Lazkanok e.lazkano@ehu.eus helbidea baliatuko du, eta Igorrek igor.rodriguez@ehu.eus. Komunikabide hau zalantzak galdetzeko, egindako aurrerapenei buruz jakinarazteko, arazoak komunikatzeko nahiz bilerak adosteko erabiliko da, besteak beste.

3.7.2.2 Bilerak

Bilerak presentzialki egingo dira, EHU/UPV-ko Informatika Fakultatean, Elena Lazkano zuzendariaren bulegoan. Presentzialki egiteko arazorik egonez gero, Webex plataforma bitartez egingo da. Bilera hauek orokorrean astero ospatuko dira, eta lehentasuna ostiral arratsaldeetan izango dute. Iraupena ordu 1 ingurukoa izango da. Salbuespen gisa, zuzendariek deitutako ezohiko bilera bat egin ahal izango da.

Proiektuaren egilearen ardura izango da astean zehar egindako aurrerapenak bilerara dokumentatuta eramatea eta Interneten atzigarri uztea. Bestetik, zuzendaria bilera aurretik bere ordenagailu lokalean proiektu eguneratua prestatzeaz arduratuko da, bilerako denborari probetxu handiena ateratzeko.

3.8 Arriskuen analisia

Atal honetan proiektuaren garapena desbideratu dezaketen arriskuak definitzen dira. Horrez gain, hauek gertatzeko probabilitateak zehazten dira eta, gertatu ezkero, izango luketen eraginaren arabera sailkatzen dira. Honako hauek izan dira Zabot proiekturako identifikatu diren arriskuak:

R1. **Hardwarearen matxura.** Zabot proiektuaren garapenaren amaieran simulaziotik mundu errealerako trantsizioa gauzatuko da. Trantsizio honek hardware erabilgarria eskura izatea eskatuko du. Zehazki, Niryo Ned beso robotikoa eta RealSense kamera beharko dira. Dispositibo hauek matxura izateko arriskua dutenez, erabiltzeko moduan dagoela ziurtatzeko robotarekin probak egingo dira lehenago, eta hainbat RealSense kamera izango dira eskura. Ahala eta guztiz ere, robotak hondaketa nabariren bat jasango balu, mundu errealerako trantsizioa bertan behera geratuko litzateke.

R2. **Hasierako proiektuaren aukeraketa.** Bide kritikoaren analisian azaldu den moduan, SIP lan-paketearen hasieran Zabot-en garapenaren oinarri esanguratsu bat definituko duen proiektu lagungarri bat hautatu beharko da. Aukeraketa honek arrisku bat suposatzen du, izan ere, proiektu okerra hautesteak garapen prozesua nabari zaildu lezake. Baina, aldi berean, aukera egokiak denbora dedikazioak murriztea ekarri lezake.

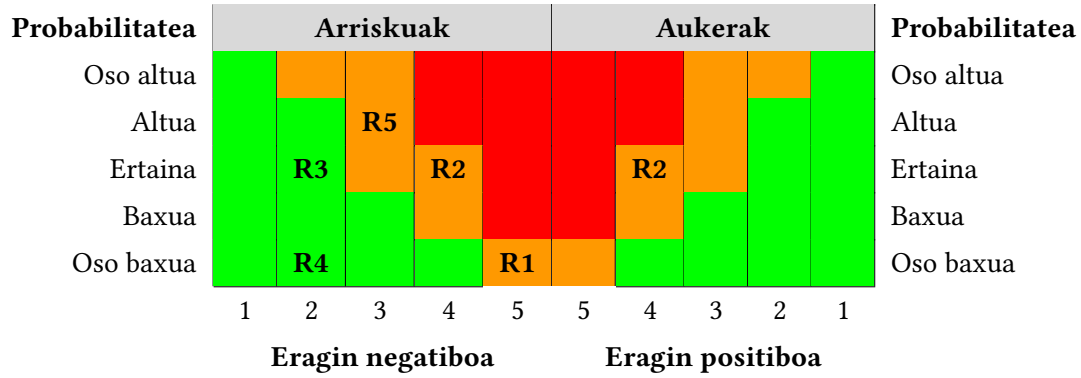
R3. **Bilera ospatzeko ezintasuna.** Bilerak ia astero ospatuko direla planifikatu da, normalean presentzialak eta ostiral arratsaldeetan izanik. Gerta liteke azken momentuan partaideren batek bilera batera ezin azaldu ahal izatea. Arrisku honen galerak murrizteko, bilera guztietan hurrengo 2 asteetan egin beharreko lana ahalik eta zehatzenen dokumentatuko da.

R4. **Fitxategien galera.** Proiektuaren fitxategi garrantzitsuenak disko gogorretan egongo direnez, hauek galtzeko arriskua dago. Hau gertatzeko probabilitatea murrizteko, informazio eguneratuaren segurtasun kopiak dispositibo ezberdinetan edukiko dira uneoro. Bestalde, proiektuaren egileak garapenaren aurrerapenak GitHub-eko biltegi batean gorde eta dokumentatuko ditu. Hori dela eta, arriskua dago aldaketa lokalak GitHub-era igo aurretik galtzeko, edozein arrazoi dela eta. Nahiz eta hau ezin den guztiz saihestu, ondorio posibleak minimizatzeko “commit”-ak aldaketa esanguratsuak dauden aldiro egikarituko dira.

R5. **Denbora estimazio okerrak.** Proiektuaren irismen eta iraupen zabala direla eta, zaila da ataza bakoitza bete ahal izateko beharrezko denbora zehatz mehatz estimatzea. Bereziki, software berriak baliatuko direnez, hala nola, Unity eta MoveIt, ez litzateke arraroa izango proiektuak planifikatu baino denbora gehiago eskatzea. Hori dela eta, estimazioak kalitate minimoa baino gehiago eskuratzeko helburua duen proiektu batentzat egin dira, eta entrega epemuga baino lehenago amaitzea planifikatu da. Halaber, beharrezkoa ikusten bada, plangintzaren eguneraketa aurrera eramango da.

Zehaztutako arriskuen probabilitatea eta izango luketen eragina 3.12 taulan biltzen dira. Bertan ikus daitekeen moduan, eremu gorria garbi dago, hau da, ez dago proiektuaren

garapena modu oso larrian oztopatuko duen arriskurik. Kasurik txarrenean eragin negatiboena izan dezakeena R1 da, baina honen gertatzeko probabilitate baxua dela eta, ez da hain galbidetsua proiektuarentzat. Halaber, aipatu beharra dago R2 arrisku bat izateaz gain, garapenari bultzada bat eman liezaiokeen aukera bat dela.



3.12 Taula: Arriskuen nahiz aukeren sailkapena probabilitatearen eta eraginaren arabera.

KAPITULUA 4

Jarraipena eta kontrola

Atal honetan proiektuaren garapenean zehar egindako jarraipenaren eta kontrolaren emaitzak azalduko dira, eta sortutako plangintzarekin alderatuko dira.

4.1 Proiektuaren kalitatea

Behin proiektua garatuta, hasieran zehaztutako kalitate irizpideak erreferentziatzat hartuz, bere kalitatea zehaztu daiteke. Hain zuzen ere, plangintzan helburua kalitate ertaineko produktu bat garatzea zela argitu zen.

Zabot ingurune birtualean nahiz fisikoan garatu da. Gainera, objektu anitzeko eszenetan aritzeko gai da. Simulazioan, pieza sinpleetarako prestatzeaz gain, errealitateko zaborrarekin funtzionarazteko saiakerak egin dira. Ikusi den moduan, zaborra pieza sinpleak baino hobeto sailkatzen du, baina ez da gai posizioa eta altuera modu onargarri batean inferitzeko. Aldiz, ingurune fisikoan pieza sinpleekin soilik egin dira exekuzioak, baina SAM bidezko inplementazioak inguruneko edozein formako objektuak antzemateko aukera ematen du. Hala ere, sailkapena koloreen bitartez gauzatzen du. Erabili diren piezekin %100eko asmatze-tasa lortu du sailkapenean. Hau guztia kontuan hartuta, eta plangintzako kalitate irizpideei erreparatuz, proiektuaren kalitatea ertaina eta handia artean kokatzen da.

4.2 Denboraldien desbiderapenak

Errealitatean egikaritutako garapen prozesuari dagokion kronograma 4.1 taulan ikus daiteke. Atazaren batean planifikatu baino beranduago edo denbora gehiago jardun bada, gorriz markatu da. Aldiz, zereginean espero baino lehenago jardun izan bada, taulan berdez ilustratu da.

4.1 taulan begies daitekeen moduan, hasieran planifikatutako kronogramak ez du desbiderapen handirik jasan. Honen arrazoi nagusietako bat plagintzaren eguneraketa (P.10 ataza) izan da. Zehazki, plangintza 2 alditan berritu da.

Lehenengo berrikuntza martxoaren 6ko astean egin zen, Zabot-en ingurune birtualaren oinarria finka zezakeen beste Unity proiektu libre bat topatu baitzen. Ondorioz, proiektu

berriaren tutorialaren jarraipena planifikatu behar izan zen, baita honek garapenean izan zitzakeen ondorioak ere. 4.1 kronograman ikusten den moduan, aipatutako bidea hartu izanak IA lan-paketeko zereginak arindu ditu, hauek bukatzeko espero baino dedikazio gutxiago eskaini behar izan zaio. Izan ere, aurkitutako Pose Estimation proiektu berriak ikusmen artifizialaren inplementazio bat ekartzen du bere baitan.

Plangintzaren bigarren egunerapena maiatzaren 15eko astean eman zen aurrera. Simulazioan entrenatutako Deep Learning modeloa mundu fisikoan probatu ostean, ikusi zen errealitateko objektuak hartzeko guztiz bideragarria dela. Hori dela eta, arazo hori konpontzen saiatu ordez, mundu fisikorako ikusmen artifizialeko beste teknika batzuk aplikatzea erabaki zen. Zehazki, SET.4 eta SET.5 ataza berriak definitu ziren. Hauek bukatzeko uste baino aste bat gutxiago behar izan denez, DA lan-paketeko zereginak amaitzeko malgutasun handiagoa egon da.

4.3 Dedikazioen desbiderapenak

Atazak garatu diren denboraldietan aldaketak egon diren moduan, hauetako batzuetarako behar izandako dedikazioetan desbiderapenak egon dira hasieran planifikatutakoarekin alderatu ezkero. Lan-pakete bakoitzari dedikatutako denbora eta izandako desbiderapena 4.2 taulan bistaratzen da. Espero baino dedikazio gehiago eskatu dutenak gorriz erakusten dira, eta denbora gutxiago behar izan dutenak urdinez. Era berean, lan-paketeei eskainitako benetazko orduak 4.1 irudian modu bisualean erakusten dira.

4.2 taulan ikus daitekeen moduan, proiektuak ez ditu desbiderapen gehiegi jasan, totalean 5 ordu gutxiagoko dedikazioa eskatu baitu. Dena den, kontuan izan beharra dago plangintzak 2 egunerapen jaso zituela. Hortaz, alderaketa lehenengo bertsioarekin egingo balitz, beste kuku batek joko luke, hasieran 300 orduko proiektua izatea planifikatu baitzen.

Desbiderapen nabarienak IA eta DA lan-paketeetan egon dira. Lehenengoa uste baino 20 ordu gutxiagotan egikaritu da. Izan ere, garapenaren erdian ingurune birtualaren oinarri gisa balio izan duen Unity-ko Pose Estimation proiektu librea topatu zen, eta honek jada ikusmen artifizialeko inplementazio bat eskaintzen zuen. DA lan-paketearen luzapena proiektu osoaren luzapenarekin bat dator. Izan ere, proiektua eta bere irismena zenbat eta zabalago, orduan eta gauza gehiago dokumentatu beharko dira. Hain zuzen ere, Zabot proiektua hasieran espero zena baino zabalagoa izatea suertatu da.

Lan-paketeek barneratzen dituzten ataza konkretuen dedikazioei dagokienez, 4.3 (Plangintza), 4.4 (Jarraipena eta kontrola), 4.5 (Informazio bilketa), 4.6 (Teknologiaren prestakuntza), 4.7 (Simulazio ingurunearen prestakuntza), 4.8 (Ikusmen artifiziala), 4.9 (Mundu fisikoaren prestakuntza), 4.10 (Simulaziotik errealitaterako trantsizioa) eta 4.11 (Dokumentazioa eta atzigarritasuna) tauletan ikus daitezke.

4.4 Arriskuak

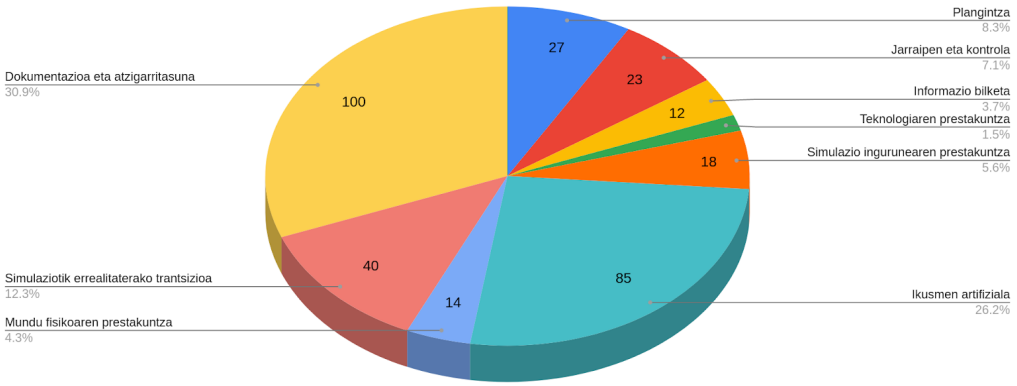
Plangintzan proiektuan eragina izan zezaketen hainbat arrisku identifikatu ziren. Horietako batzuk ez dira egikaritu, baina beste batzuk bai. Zehazki, honako hauek izan dira proiektuaren garapenean zehar gertatu diren arriskuak:

	Urtarrila			Otsaila				Martxoa				Apirila				Maiatza					Ekaina		
Ataza	16	23	30	6	13	20	27	6	13	20	27	3	10	17	24	1	8	15	22	29	5	12	19
P.1 - P.6																							
P.7																							
P.8 - P.9																							
TP.1																							
IB.1 - IB.4																							
TP.2 - TP.4																							
TP.5 - TP.6																							
IB.5																							
TP.7																							
SIP.1																							
SIP.2																							
SIP.3																							
SIP.4 - SIP.6																							
P.10																							
SIP.7																							
*IA.1																							
IA.2																							
IA.3																							
IB.7																							
IA.4 - IA.5																							
IA.6 - IA.7																							
IA.8 - IA.10																							
MFP.1 - MFP.4																							
MFP.5 - MFP.6																							
SET.1 - SET.3																							
**SET.4																							
**SET.5																							
E450.1																							
E450.2																							
E450.3																							
DA.1																							
DA.2																							
DA.3																							
DA.4																							
JK.1																							
JK.2																							
JK.3																							
JK.4																							

4.1 Taula: Egikaritutako garapen prozesuaren Gantt kronograma.

Lan-paketea	Orduak (h:mm)
Plangintza	27:00 (+2)
Jarraipena eta kontrola	23:00 (-2)
Informazio bilketa	12:00 (+2)
Teknologiaren prestakuntza	5:00
Simulazio ingurunearen prestakuntza	18:00 (+4)
Ikusmen artifiziala	85:00 (-20)
Mundu fisikoaren prestakuntza	14:00 (-1)
Simulaziotik errealitaterako trantsizioa	40:00
Dokumentazioa eta atzigarritasuna	100:00 (+10)
PROIEKTUAREN DENBORA TOTALA	**324:00 (-5)**

4.2 Taula: Lan-pakete bakoitza garatzeko behar izandako denbora eta plangintzarekiko desbiderapena.



4.1 Irudia: Lan-paketeei dedikatutako denboraren diagrama.

PLANGINTZA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
P.1	2:00	**P.6**	4:00 (+1)
P.2	2:00 (-1)	**P.7**	1:00
P.3	4:00	**P.8**	1:00
P.4	4:00	**P.9**	3:00
P.5	3:00	**P.10**	3:00 (+2)
GUZTIRA: 27h (+2)			

4.3 Taula: Plangintza lan-paketea garatzeko behar izandako denbora eta plangintzarekiko desbiderapena.

JARRAIPENA ETA KONTROLA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
JK.1	5:00	**JK.3**	5:00
JK.2	4:00 (-1)	**JK.4**	9:00 (-1)
GUZTIRA: 23h (-2)			

4.4 Taula: Jarraipena eta kontrola lan-paketea garatzeko behar izandako denbora eta plangintzarekiko desbiderapena.

INFORMAZIO BILKETA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
IB.1	1:30	**IB.4**	3:00
IB.2	1:00	**IB.5**	3:00 (+1)
IB.3	0:30	**IB.6**	3:00 (+1)
GUZTIRA: 12h (+2)			

4.5 Taula: Informazioaren bilketa lan-paketea garatzeko behar izandako denbora eta plangintzarekiko desbiderapena.

TEKNOLOGIAREN PRESTAKUNTZA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
TP.1	0:30	**TP.5**	0:30
TP.2	0:30	**TP.6**	2:00
TP.3	0:30	**TP.7**	0:30
TP.4	0:30		
GUZTIRA: 5h			

4.6 Taula: Teknologiaren prestakuntza lan-paketea garatzeko behar izandako denbora eta plangintzarekiko desbiderapena.

SIMULAZIO INGURUNEAREN PRESTAKUNTZA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
SIP.1	4:00 (+2)	**SIP.5**	3:00
SIP.2	6:00 (+3)	**SIP6**	1:00
SIP.3	1:00 (-1)	**SIP.7**	2:00
SIP.4	1:00		
GUZTIRA: 18h (+4)			

4.7 Taula: Simulazio ingurunearen prestakuntza lan-paketea garatzeko behar izandako denbora eta plangintzarekiko desbiderapena.

IKUSMEN ARTIFIZIALA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
***IA.1**	6:00 (+1)	**IA.6**	2:00 (-4)
IA.2	3:00	**IA.7**	3:00 (-2)
IA.3	2:00 (-3)	**IA.8**	2:00
IA.4	2:00 (-4)	**IA.9**	60:00 (-2)
IA.5	2:00 (-4)	**IA.10**	3:00 (-2)
GUZTIRA: 85h (-20)			

4.8 Taula: Ikusmen artifiziala lan-paketea garatzeko behar izandako denbora eta plangintzarekiko desbiderapena.

MUNDU FISIKOAREN PRESTAKUNTZA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
MFP.1	2:00	**MFP.4**	3:00
MFP.2	3:30	**MFP.5**	3:00
MFP.3	2:00 (-1)	**MFP.6**	0:30
GUZTIRA: 14h (-1)			

4.9 Taula: Mundu fisikoaren prestakuntza lan-paketea garatzeko behar izandako denbora eta plangintzarekiko desbiderapena.

SIMULAZIOTIK ERREALITATERAKO TRANTSIZIOA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
SET.1	3:00 (-1)	****SET.4**	6:00 (+1)
SET.2	1:00	****SET.5**	28:00 (+1)
SET.3	2:00 (-1)		
GUZTIRA: 40h			

4.10 Taula: Simulaziotik errealitaterako trantsizioa lan-paketea garatzeko behar izandako denbora eta plangintzarekiko desbiderapena.

DOKUMENTAZIOA ETA ATZIGARRITASUNA			
Ataza	**Orduak (h:mm)**	**Ataza**	**Orduak (h:mm)**
DA.1	80:00 (+10)	**DA.3**	3:00
DA.2	5:00	**DA.4**	12:00
GUZTIRA: 100h (+10)			

4.11 Taula: Dokumentazioa eta atzigarritasuna lan-paketea garatzeko behar izandako denbora eta plangintzarekiko desbiderapena.

R1. **Hardwarearen matxura.** Zoritxarrez, simulaziotik errealitaterako trantsizioko atazetan jarduten ari zen bitartean, Niryo Ned robot fisikoak matxura bat jasan zuen. Hain zuzen ere, robotaren eragingailua mugiarazten duen motorrak portaera okerra erakusten hasi zen. Dirudienez, beso robotikoak gainazal baten gainean indar asko aplikatzen badu, esaterako, mahaia baino beherago dagoen posizio batera joan nahi duenean, motorra matxuratzeko arriskua dago. Zorionez, robota egun oso bat deskantsatzen utzita arazoa konpondu egin zen. Hau ikusita, hurrengo probetan kontu gehiagoz ibiltzea erabaki zen.

R2. **Hasierako proiektuaren aukeraketa.** SIP lan-paketearen hasieran Zabot-en garapenaren oinarri esanguratsu bat definitu zuen proiektu lagungarri bat hautatu zen. Hasieran Pick-and-Place izeneko proiektutik abiatu zen, eta honek eskaintzen zuen tutoriala aurrera eraman zen. Baina garapenean aurrera joan zen ahala, Zabot-en irismenera gehiago hurbiltzen den beste proiektu bat topatu zen: Pose Estimation izenekoa. Azken honen tutoriala ere jarraitu behar izan zen. Hortaz, prozesu horretan denbora xahutze nabarmen bat egon zen, eta plangintza eguneratu behar izan zen. Dena dela, bigarren proiektuan murgiltzeak merezi izan zuen, IA lan-paketearen garapena asko arindu baitzuen. Hortaz, proiektuan R2-ren arrisku nahiz aukera aldeak egikaritu dira, aldi berean eragin negatiboa nahiz positiboa izanik.

R5. **Denbora estimazio okerrak.** Plangintzan aipatzen den moduan, zaila da ataza bakoitza bete ahal izateko beharrezko denbora zehatz mehatz estimatzea. Hain zuzen ere, hasieran egindako dedikazioen estimazio asko okerrak suertatu dira. Honek, plagintza 2 aldiz eguneratu ostean, proiektuaren dedikazio totala 300 ordutik igotzea eragin du. Hala ere, ataza batzuei uste baino denbora gutxiago eskaini zaie eta, beraz, eragindako desbiderapena ez da hain nabarmena izan.

Atala III

Garapena

KAPITULUA 5

Tresnak eta teknologiak

Kapitulu honetan Zabot proiektuan erabiliko diren tresnak nahiz teknologiak aurkeztu eta azalduko dira.

5.1 Niryo Ned beso robotikoa

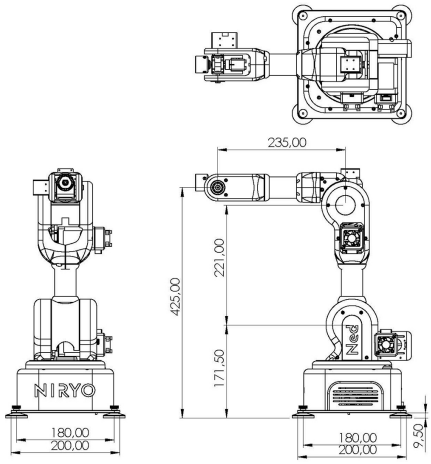
5.1.1 Zehaztapen mekanikoak

Zabot proiektua aurrera eramateko Niryo enpresak garatutako Niryo Ned beso robotikoa erabiliko da. Honen itxura 5.1 irudian ikus daiteke, eta bere dimentsioak 5.2 irudian.

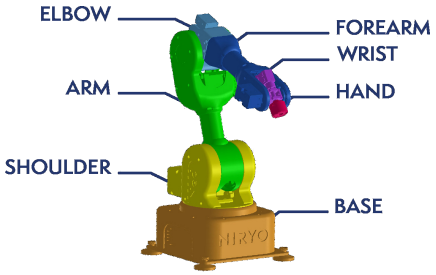
5.1 Irudia: Niryo Ned robota.

Robot honek 6 ardatz ditu eta, ondorioz, 6 askatasun-graduko mugimendua lor dezake. Ardatzen mugimendua 5.3 irudian ikus daiteke. Aipatu beharra dago “shoulder” ardatz nagusiaren biraketa ez dela 360 gradukoa. Horren ordez, [-175, 175] angeluen artean ibil daiteke eta, beraz, besoa ez da gai bere atzeko 10 biraketa graduak bisitatzeko. Halaber, robotaren gehienezko irismena 0,44 metrokoa da.

Objektuak hartu ahal izateko beso robotikoak defektuz matxarda edo pintza bat du instalatuta. Dena den, Niryo etxeak xurgagailu bat ere kaleratu du (5.4 irudia) , matxarda ordezkatu dezakeena.



5.2 Irudia: Niryo Ned robotaren dimentsioak.



5.3 Irudia: Niryo Ned robotaren ardatzak.

Xurgagailuak objektuen manipulazioa errazten du, izan ere, artefaktuak gainetik eta haien grabitate zentrutik heldu behar ditu. Aldiz, pintzarekin aritzean objektuen orientazioa ere kontuan hartu beharra dago. Arrazoi honengatik, Zabot proiektuko robot fisikoak xurgagailuarekin egingo du lan. Dena dela, kontuan izan beharra dago gehienez 350 gramoko piezak hartzeko gaitasuna daukala.

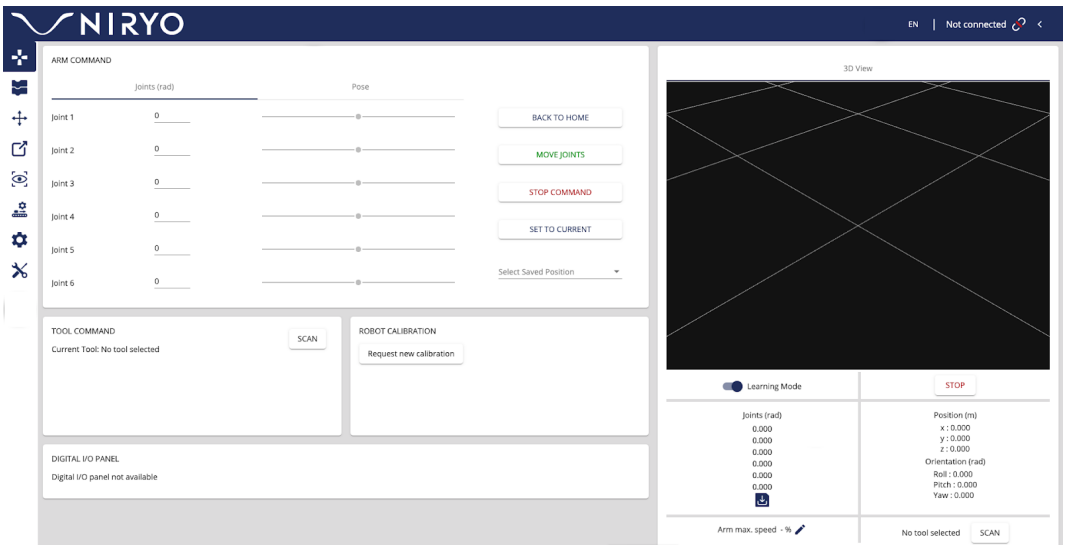
5.4 Irudia: Niryo etxeak diseinatutako xurgagailua.

Hori gutxi balitz, robotarentzat espresuki diseinatutako kamera bat txertatu liezaioke, 5.5 irudian begiesten den moduan. Hala ere, Zabot proiektuaren helburururako desegokia da, besoko kameraren ikuspegitik ezin baitira inguruko pieza guztiak ikusi. Beraz, kamera honen ordez, eszenan estatikoa den RGB-D motako RealSense bat instalatu eta erabiliko da.

5.5 Irudia: Niryo Ned robotarentzat egindako kamera.

5.1.2 Niryo Studio softwarea

Niryo etxeak haien robota kontrolatzeko software librea garatu du: Niryo Studio[1] Niryo Studio makinaren eta gizakien arteko interfazea da, ordenagailu baten bitartez Niryo Ned beso robotikoa kontrolatzea ahalbidetzen duena. Besteak beste, interfazeak robotaren parametroak aldatzea, artikulazioen mugimendua kontrolatzea eta portaera definitzen duten programak exekutatzeko aukera ematen du. Interfazearen egitura 5.6 irudian ikus daiteke.



5.6 Irudia: Niryo Studio interfazea.

Niryo Studio instalatuta duen ordenagailuaren eta beso robotikoaren arteko komunikazioa gauzatzeko 3 modu daude: Ethernet kable bidez, hotspot moduan, eta Wi-Fi sarearen bidez konexioa gauzatuz. Zabot proiektuaren kasuan, konexioa Ethernet kable bidez egikarituko da.

[1]Niryo Studio softwareari buruzko informazioa: URL

5.2 ROS: Robot Operating System

5.2.1 Zer da ROS?

ROS robotentzat aplikazioak garatzeko erabiltzen den framework-a da. Doakoa nahiz kode irekikoa da, eta gaur egun, ingenierari eta garatzaileen komunitate zabal batek osatzen du. Gehien bat UNIX sistemetarako eginda dago, baina beste sistema eragileetan agertzen hasi da.

ROSek roboten softwarearen inplementazioa errazteko pakete, erreminta eta algoritmo multzo zabala eskaintzen du, eta aplikazio ezberdinetako kodearen berrerabilpena errazten du. Azken finean, robotentzat sistema eragile moduan funtzionatzen du, izan ere, hardware atalaz gehiegi arduratu behar izan gabe softwarea garatzea ahalbidetzen du. Gainera, garapenean fabrikatzailearen APIa ikasi eta erabili ordez (robot batetik bestera aldatzen dena), robot guztiek partekatu ditzaketen programak inplementatu daitezke ROS bidez.

Horrez gain, ROS lengoaiekiko independentea da, hau da, goi-mailako ia edozein programazio-lengoairekin idatzitako programak maneiatu ditzake. Esate baterako, Python, C++ eta Lisp-ekin funtziona dezake. Zabot proiektuaren kasuan, ROS aldeko kodea Python 3.8 lengoaiarekin idatziko da gehien bat.

Hori gutxi balitz, ROSek komando-lerroan exekutatu daitezkeen komandoen bilduma zabala eskaintzen du[2]. Hauei esker, fitxategi-sistemako nabigazioa eta pakete nahiz biltegien kudeaketa sinplifikatzen da.

5.2.2 ROS kontzeptuak

ROSek hiru kontzeptu-maila banatzen ditu: Fitxategi-sistemaren maila, Konputazio-grafoaren maila eta Komunitate-maila. Maila bakoitzean hainbat kontzeptu definitzen dira.

5.2.2.1 Fitxategi-sistemaren maila

Hemen disko gogorrean aurkitzen diren kontzeptuak definitzen dira. Nagusienak honakoak dira:

- Paketeak: ROSen softwarea antolatzeko unitate nagusia dira. Paketeak eraiki eta publikatu daitezkeen elementu sinpleenak dira. Hauek ROS exekutagarriak, datubaseak, konfigurazio fitxategiak eta abar eduki ditzakete.
- Pakete-manifestuak (Package Manifests): Manifestuek pakete bati buruzko informazioa eskaintzen dute, esate baterako, paketearen izena, bertsioa, lizentzia eta dauzkan dependentziak. Manifestuak *package.xml* fitxategietan idazten dira.
- Biltegiak (Repositories): Elkarrekin publikatu daitezken eta bertsio bera duten paketeen bildumak dira. Pakete bakarrekoak ere izan daitezke.
- Launch fitxategiak: Fitxategi hauetan hainbat ROS exekutagarri definitu daitezke, denak batera abiarazi ahal izateko. Bertan atzigarriak izango diren parametroak ere txertatu daitezke. Fitxategiek *.launch* luzera dute, eta roslaunch komandoarekin irekitzen dira. Adibide gisa 5.7 irudia dago.

[2] ROSek eskaintzen dituen komando-lerroko tresnak: URL

```
<launch>
  <node name="listener_node" pkg="hello_world" type="listener" output="screen"/>
  <node name="talker_node" pkg="hello_world" type="talker" output="screen"/>
</launch>
```

5.7 Irudia: Launch fitxategiaren adibidea.

5.2.2.2 Konputazio-grafoaren maila

ROSek prozesuak "Konputazio-grafo" izeneko sare baten bidez kudeatzen ditu. Sare hau P2P (*Peer to Peer*) motakoa da, hau da, lanak prozesuen artean banatzen dira eta informazio-trukeak edozein bikoteren artean eman daitezke. Konputazio-grafoan kontzeptu hauek topatzen dira:

- Nodoak: Konputazio lanak egiten dituzten prozesuak edo programak dira. Nodo bakoitzak ataza konkretu bat eramaten du, eta elkarren artean banatuak dira. Robotaren kontrola nodo askoren bitartez gauzatu ohi da, modularitatea bermatuz.

- Master: ROS *Master*-ak izenak erregistratzeko balio du, eta Konputazio-grafoaren bilaketa zerbitzuak eskaintzen dizkie nodoei. Beste hitz batzuetan esanda, nodo bakunei beste prozesuak kokatzea ahalbidetzen die, informazio-trukea eraman ahal izan dezaten.

- Mezuak: Nodoek elkarren artean informazioa trukatzeko erabiltzen dituzten datu-egiturak dira. Mezuak *.msg* luzerako fitxategietan definitzen dira. Hainbat eremu izan ditzakete, eremu bakoitza datu-mota primitiboa (integer, float, boolean…) ala konplexua (bektorea, klasea, beste mezua…) izan daitekelarik. 5.8 irudian *.msg* fitxategi baten adibidea begies daiteke.

```
float64[6] joints
geometry_msgs/Pose pick_pose
geometry_msgs/Pose place_pose
float32 scale
```

5.8 Irudia: Msg fitxategiaren adibidea.

5.2.2.3 Komunitate-maila

Maila honek komunitateen artean jakintza eta softwarea trukatzeko kontzeptuak barneratzen ditu. Zabot proiektuaren irismena kontuan izanda, aipatu beharreko kontzeptu bakarra ROSeko **distribuzioak** dira. Distribuzioak bertsiotutako ROS paketeen bildumak dira, Linux-en distribuzioen antzera. ROSen kasuan bi urtero distribuzio berri bat kaleratzen dute. Distribuzio berriek aurrerapenak eta konponketak ekartzen dituzte eta, beraz, azken bertsioarekin lan egitea komeni da. Hala ere, kontuan izan beharra dago instalatu daitezkeen distribuzioak Linux bertsioaren mendekoak direla.

Zabot proiektuaren garapenean, simulazioa garatzeko egun berriena den distribuzioa erabiliko da, **ROS Noetic** izenekoa. Hau Ubuntu 20.04 duen makina batean instalatu da. Aldiz, proba fisikoak egiteko paketeak **ROS Melodic**-en garatuko dira, Ubuntu 18.04 duen makina batean. Izan ere, Niryo Ned robotak bertsio hau dauka instalatuta.

5.2.3 Nodoen arteko komunikazioa

Nodoek elkarren artean mezuak trukatu ahal izateko, ROSek bi komunikazio eredu definitzen ditu: argitalpen/harpidetza eredua eta eskaera/erantzun eredua.

5.2.3.1 Argitalpen/harpidetza eredua

Komunikazio sistema honek *topic* izeneko buzoien bidez funtzionatzen du. Nodoak *topic* konkretuetara harpidetuta egon daitezke. Bitartean, beste nodo batzuk buzoian informazioa argitaratzeaz arduratuko dira. Horrela, harpidedunak nahi duten momentuan buzoiera iristen diren mezuak atzitzeko gai izango dira. Buzoi berean hainbat nodok argitara dezakete, eta nodo bat baino gehiago egon daitezke harpidetuta buzoi berera. Beraz, argitalpen/harpidetza ereduak komunikazio asinkronoa baliatzen du, *many-to-many* motakoa izateaz gain.

Ezberdindu ahal izateko, *topic* bakoitzak bere izena izango du, eta bertan argitaratuko diren mezuen mota ere definituta egongo da. Halaber, edukiera maximo bat izango dute, buzoietako mezuak irakurri ezean betetzen joango baitira. *Topic* beteak kudeatzeko hainbat politika existitzen dira. Esaterako, ilara moduan funtzionatu dezakete, irakurri gabeko mezu zaharrenak ezabatzen dituztelarik, berrienei tokia egiteko.

5.2.3.2 Eskaera/erantzun eredua

Komunikazio mota hau bezero/zerbitzari ereduaren parekoa da. ROSek aldez aurretik zerbitzariak definitzea ahalbidetzen du, bakoitzak bere izena izanik eta bere zerbitzua eskaintzen duelarik. ROS zerbitzuak *.srv* luzerako fitxategietan biltzen dira. Fitxategi hauen idazkera mezuen parekoa da, eskaera eta erantzunen eremuen izenak eta motak definitzen baitituzte. Adibide bat 5.9 irudian ikus daiteke. Noski, komunikazio mota honen bitartez *.msg* fitxategietan definitutako mezu konplexuak ere trukatu daitezke.

```
#Eskaera
int64 A
int64 B
string operation
---
#Erantzuna
int64 result
```

5.9 Irudia: Srv fitxategiaren adibidea.

Behin zerbitzariei dagozkien nodoak abian daudela, beste nodoek bezero moduan eskaerak egin ditzakete. Horretarako, *.srv* fitxategian definitutako eskaera formatua bete beharko dute. Zerbitzari batek eskaera jaso ostean, beharrezko konputazio eragiketak gauzatuko ditu eta, ondoren, eskaera igorri dion bezeroari erantzuna itzuliko dio. Bezeroa erantzuna jasotzeko prest egon beharko da. Beraz, eskaera/erantzun ereduak sinkronoa eta *one-to-one* motakoa den komunikazioa definitzen du.

5.2.4 Catkin sistema

Catkin ROSen eraikitze-sistema (*build system*) ofiziala da, CMake nahiz Python-eko funtzionalitateak bateratzen dituena. Catkin lan-eremu (*workspace*) batean kokatutako paketeak konpilatzeaz nahiz eraikitzeaz arduratzen da. Lan-eremu bat ROS programak garatzeko

karpeta bat besterik ez da. Hala ere, *workspace* bat catkin bidez kudeatu ahal izateko, hurrengo komandoaren bitartez hasieratu egin beharra dago:

```
catkin_init_workspace
```

Eta paketeak eraiki aurretik, lan-eremua aktibatu egin behar da ROS sistemak topa dezan. Hau karpeta nagusian hurrengo komandoa exekutatuz egiten da:

```
source devel/setup.bash
```

Catkin ingurune bateko paketeak agindu honen bitartez konpilatzen dira:

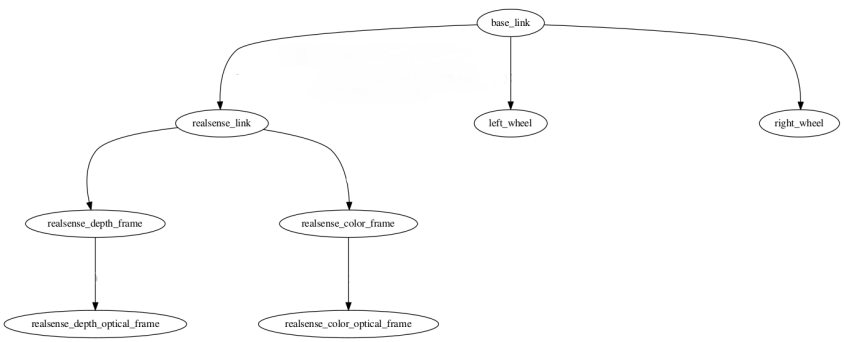
```
catkin_make
```

5.2.5 Transformatuen sistema eta *tf*

Beso robotikoari inguruneko piezak hartzeko aginduak eman ahal izateko, objektuen nahiz robotaren posizio eta orientazio zehatzak jakin behar dira. Proiektuaren helburua hondakinen posizioak RealSense kamera baten bitartez antzematea denez, kameraren parametroak ere jakin behar dira. Baina, honen ondorioz, detektatzen diren objektuen posizioak kameraren kokapenaren menpekoak izango dira. Beste hitz batzuetan esanda, kameraren koordenatu-sisteman definituko dira. Noski, *pick-and-place* ataza planifikatzeko posizio eta orientazio hauek guztiak munduaren erreferentzia-sisteman kalkulatu behar dira. Beraz, erreferentzia-sistema ezberdinak kudeatzeko eta haien arteko transformazioak egiteko beharra sortzen da.

Hain zuzen ere, ***tf*** hainbat koordenatu-sistema aldi berean kontrolatzeko eta haien arteko erlazioak definitzeko balio duen ROS paketea da. Sistema ezberdinak eta haien arteko erlazioak ***tf_tree*** izeneko zuhaitz motako egitura batean mantentzen ditu; eta espazioko puntuak, bektoreak eta abar sistema batetik bestera bihurtzeko funtzioak eskaintzen ditu. Aipatutako zuhaitza hurrengo komandoaren bitartez bistara daiteke (ikus 5.10 irudia ere):

```
rosrun tf view_frames
```



5.10 Irudia: *tf_tree* zuhaitzaren adibidea.

5.2.6 MoveIt

MoveIt[3] beso robotikoaren ibilbideak planifikatzeko funtzionalitateak eskaintzen dituen tresna da. Kalkuluak alderantzizko zinematika aplikatuz egiten ditu, hau da, definitutako espazioko puntu batera iristeko robotaren ardatz bakoitzak jasan behar duen mugimendua kalkulatzen du. Nahi izanez gero, planifikatutako ibilbidea exekutatzeko aukera eskaintzen du ere.

Zabot proiektuan Niryo Ned robota kontrolatzeko MoveIt softwarea erabiliko da, zeregin horrek dakartzan abstrakzioak eta konplexutasunak ebatzi behar izatea ekiditen baitu. Izan ere, alderantzizko zinematikaren aplikazioak kalkulu konplexuak egitea eskatzen du; bereziki, robota ardatz askokoa denean.

5.2.7 URDF (Unified Robotics Description Format) fitxategiak

URDF fitxategiak ez dira ROS plataforman soilik erabiltzen, baina oso ezagunak dira ROS komunitatean. URDF roboten modeloak zehazteko XML-en bereizmen bat da. Bertan, etiketen bidez, robotaren atalak eta artikulazio guztiak definitzen dira, eta haien arteko loturak zehazten dira. Elementu bakoitzaren erreferentzia-sistema lokala ere espezifikatzen da. Hortaz, URDF fitxategiak simulazio inguruneetan roboten definizioak kargatzeko interesgarriak dira. 5.11 irudian *.urdf* fitxategi baten zati baten adibidea dago.

```
<robot name="niryo_one">
  <!-- Properties -->
  <!-- Links -->
  <link name="world"/>
  <link name="base_link">
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://niryo_one_urdf/meshes/collada/base_link.dae"/>
      </geometry>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <mesh filename="package://niryo_one_urdf/meshes/stl/base_link.stl"/>
      </geometry>
    </collision>
  </link>

  <joint name="joint_world" type="fixed">
    <parent link="world"/>
    <child link="base_link"/>
    <origin rpy="0 0 0" xyz="0 0 0.63"/>
  </joint>
```

5.11 Irudia: URDF fitxategiko zati baten adibidea.

5.3 RealSense kamera

RealSense kamera Intel enpresak garatutako dispositiboa da, RGB irudi arruntak ateratzeaz gain, 3D espazioaren egitura antzemateko gaitasuna duena. Hau sakonerari buruzko informazioa eskuratzeko gai den sentsore bati esker lortzen du.

[3] MoveIt tresnaren webgune ofiziala: URL

Kamera hauek dronetan, robotetan, errealitate birtualeko dispositiboetan eta beste eremu askotan erabili ohi dira. Horrez gain, pisu nahiz kontsumo gutxikoak dira. Gainera, Intel-ek RealSense SDK 2.0 software librea[4] garatu du, haien kamerak kontrolatzeko, kalibratzeko eta atzitzeko balio duena. Software honek ROS plataformarekin bateragarritasuna eskaintzen du.

Kameraren eta beste ROS nodoen arteko komunikazioa *topic* bidez gauzatzen da. Hala, sentsoreak irudi eta informazio ezberdinak hainbat postontzitan argitaratzen ditu, aipagarrienak honakoak izanik:

- **/camera/color/image _raw** (RGB irudia)
- **/camera/depth/image_rect_raw** (sakoneraren informazioari dagokion irudia)
- **/camera/depth/color/points** (*Point Cloud*-aren informazioa)

Beraz, kamera hau Zabot proiektuan instalatzeko eta erabiltzeko egokia dela ondorioztatu da.

5.4 Unity

5.4.1 Zer da Unity?

Unity 2D nahiz 3D ingurune interaktiboak eraikitzeko balio duen garapen-plataforma da. Hasieran, bideojokoak garatzeko pentsatuta zegoen, baina gaur egun beste esparru askotara hedatu da. Esaterako, zinemagintzan erabiltzera iritsi da, baita robotikan ere; roboten portaerak simulatzeko gaitasuna eskaintzen baitu. Hain zuzen ere, azken erabilpen hau Zabot proiektuaren irismenarekin bat dator. Jakinik ikusmen artifiziala baliatuko duen robot bat prestatuko dela, Unity oso lagungarria izango da bai irudien datu-baseak biltzeko eta bai proba birtualak egiteko ere, proiektua mundu fisikora eraman aurretik.

5.4.2 Unity Hub eta editoreak

Unity Technologies taldeak urteetan zehar editore bertsio berriak garatzen ditu, hobekuntza asko eskainiz. Unity-ko proiektu bat hasi aurretik, editore bertsio bat aukeratzera derrigortzen da. Garapenean zehar editorearen bertsioa aldatzeko aukera dago, baina funtzionalitateak denboran zehar aldatzen dituztenez, ez dago ongi funtzionatuko duen bermerik. Beraz, proiektuak beti hasitako editorearekin irekitzea gomendatzen da. Honen ondorioz, kanpotik jaitsitako proiektuekin lan egiten denean, aldi berean editore bertsio asko instalatuta izateko beharra sortzen da. Hauek guztiak kudeatu ahal izateko, Unity Hub izeneko softwarea kaleratu zuten. Honen bidez, editore bertsio ezberdinak instalatzen dira eta Unity proiektuak kudeatzen dira. Beraz, Unity-rekin lanean hasi ahal izateko, instalatu beharreko lehenengo gauza Unity Hub izango da.

[4] Intel-en RealSense SDK 2.0 softwarearen webgune ofiziala: URL

5.4.3 Unity proiektuetako kontzeptu nagusiak

5.4.3.1 Assets

Proiektua eraikitzeko baliatzen diren piezak dira. Mota askotarikoak daude, adibide garbienak 3D objektuak, testurak, materialak eta soinu-efektuak izanik. Hauek *Assets/* karpetan gordetzen dira, proiektuaren direktorioaren barruan.

5.4.3.2 Eszenak

Eszenak ingurune birtuala definitzen duten instantziak dira. Irismen handiko bideojokoetan lurralde ezberdinak prestatzen dira, bakoitza eszena batean. Baina Zabot-en kasuan hau ez da beharrezkoa. Hortaz, proiektu finala eszena bakar batean prestatuko da, baina eszena ezberdinak erabiliko dira garapenean zehar izandako bertsio ezberdinak gordetzeko.

5.4.3.3 GameObjects

Eszena bateko edozein objekturi *GameObject* deritzo. Hortaz, asset-ak eszenan sartzean *GameObject* kontsideratzen dira. Hautetatik kanpo mota gehiago ere badaude, esaterako; partikulak, argiak eta botoiak. *GameObject*-ek hierarkia bat definitu dezakete: objektu batzuk besteen ume edo guraso izan daitezke. Halaber, objektuak desaktibatu egin daitezke, eszenan eraginik izan ez dezaten. *GameObject* hutsak ere sortu daitezke.

5.4.3.4 Osagaiak (*components*)

Osagaiak *GameObject*-ak berez dituzten edo manualki txertatzen zaizkien propietateak dira. Propietate hauei esker objektuaren portaera, itxura eta eszenan duen eragina definitzen dira. Osagai garrantzitsuena *Transform* da, *GameObject* guztiek defektuz izaten baitute. Honek objektuaren posizioa, biraketa eta eskala definitzen ditu, bere koordenatu-sistema lokalean. Objektu baten umeek gurasoen transformazioak eta koordenatu-sistema heredatuko dituzte. Beste osagai batzuk ere badaude: *Renderer* (objektua ikusi ahal izateko), *Rigidbody* (fisikak aplikatzeko), *Collider* (objektuaren mugak definitzeko, talkak antzeman ditzan) eta abar.

5.4.3.5 Programak (*scripts*)

Nahiz eta osagai gisa har daitezkeen, programak objektuei txertatzen zaizkien exekutagarriak dira, ingurunea martxan jartzean exekuta daitezen. Batzuetan botoi motako *GameObject*-ei ere lotzen zaizkie, botoia klikatzean exekuta daitezen. Programek eszena estatiko bat dinamiko bilakatzen dute, beharrezko logika eskainiz. Zabot proiektuan Unity aldeko kodea C# bidez idatziko da, gehien erabiltzen den programazio-lengoaia baita. Programazioa errazteko Unity-k *MonoBehaviour* izeneko klasea eskaintzen du. Unity-ko *script* guztiak honen herentziaz sortzen dira, eta eskaintzen dituen metodo asko baliatu ditzakete, berriak definitzeaz gain. Hauek dira heredatzen diren funtzio garrantzitsuenak:

- Start(): Funtzio honetan ezartzen diren agindu guztiak eszena martxan jarri bezain laster exekutatzen dira. Objektuak eta haien parametroak hasieratzeko erabili ohi da.
- Update(): Metodo hau simulazioko frame bakoitzeko exekutatzen da. Esaterako, Zabot-en interesgarria izan daiteke irudien datu-base bat bildu ahal izateko.

5.4.3.6 Botoiak

Eszenan txertatzen diren interfaze-motako objektuak dira. Botoi bat txertatzen denean, *Canvas* objektu bat sortzen da eta botoia bere ume gisa jartzen da. Botoiek osagai berezi bat daukate: programa bat etiketatu dezakete sakatzean exekuta dadin.

5.4.3.7 Kamerak

Kamerak eszena puntu konkretu batetik ikustea ahalbidetzen duten objektuak dira. Eszena berean hainbat kamera egon daitezke. Unity-k 8 *display* edo pantaila dauzka, eta kamera bakoitzari horietako bat esleitu beharko zaio. Horrela, eszena martxan jarri ostean pantaila konkretu bat aukeratu ahal izango da, bertara lotuta dagoen kamararen perspektiba ikusteko. Bestalde, kamera batek irudia pantailara bidali ordez, testura batean errenderizatu dezake.

5.4.3.8 Argiak

Eszenan argiztapena gehitzen duten *GameObject*-ak dira. Argien dinamikaz Unity-ren motor-grafikoa arduratzen da. Hauen intentsitatea, norabidea, kolorea eta beste hainbat propietate aldatzeko aukera dago.

5.4.3.9 Layer

Eszenako objektu bakoitzari *Layer* bat esleitu diezaioke. Hauek etiketa moduan funtzionatzen dute, bakoitzak bere izena duelarik. Objektu baten *Layer*-aren arabera, beste *GameObject*-ek berekiko duten portaera pertsonalizatu daiteke. Esate baterako, "ignoreCamera" etiketa sor daiteke, eta kamerari esan *Layer* horretako objektuak ez errenderizatzeko.

5.4.3.10 Prefab

Asset-en antzekoak dira, baina hauek Unity-ko osagaiak barneratuta dituzte. Beraz, proiektu askotan erabiliko diren objektuak beren propietateekin gordetzeko erabilgarriak dira. Eszenara sartzen diren momentuan, gorde ziren posizio, biraketa eta tamainarekin agertzen dira, baita beste osagai guztiekin ere.

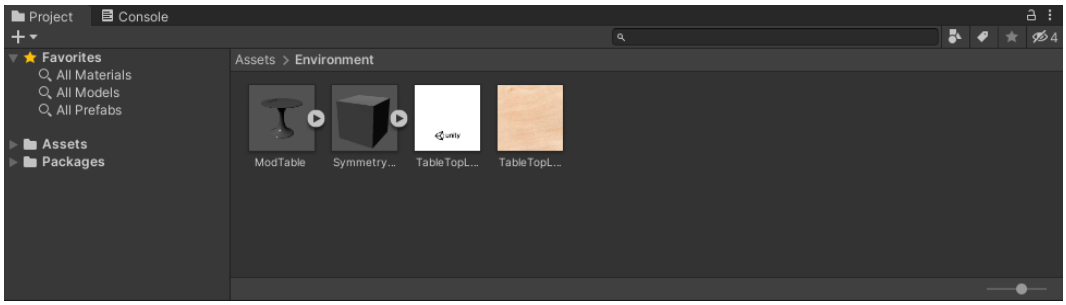
5.4.4 Unity editoreen leiho nagusiak

5.4.4.1 Project leihoa

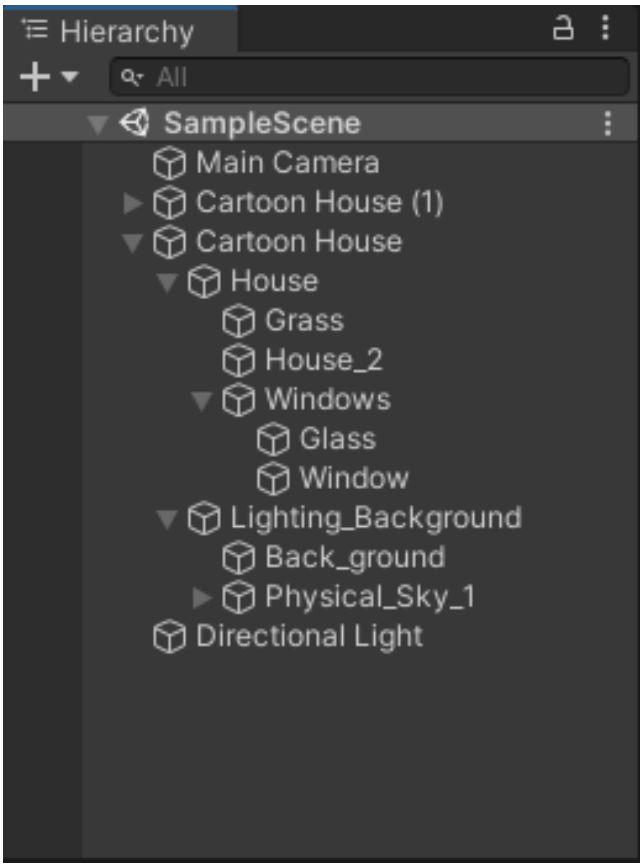
Editorearen azpian agertzen den leihoa da (5.12 irudia). Hemen proiektuaren barruko karpetak eta fitxategiak atzitu daitezke, baita erabiltzen diren paketeak kudeatu ere. Adibidez, *Assets/* direktorioko objektuak eszenan txertatzeko balio du, baita eszena ezberdinak kargatzeko ere.

5.4.4.2 Hierarchy leihoa

Leiho honetan eszenak eta bere baitako *GameObject*-ak kudeatzen dira (5.13 irudia) . Editorearen ezkerraldean kokatu ohi da. Izenak dioen bezala, objektuek osatzen duten hierarkia erakusten du, beste baten umeak diren objektuak gurasoaren barruan agertzen direlarik. Leiho honek eszenako objektuak aukeratzea eta ezabatzea ahalbidetzen du.



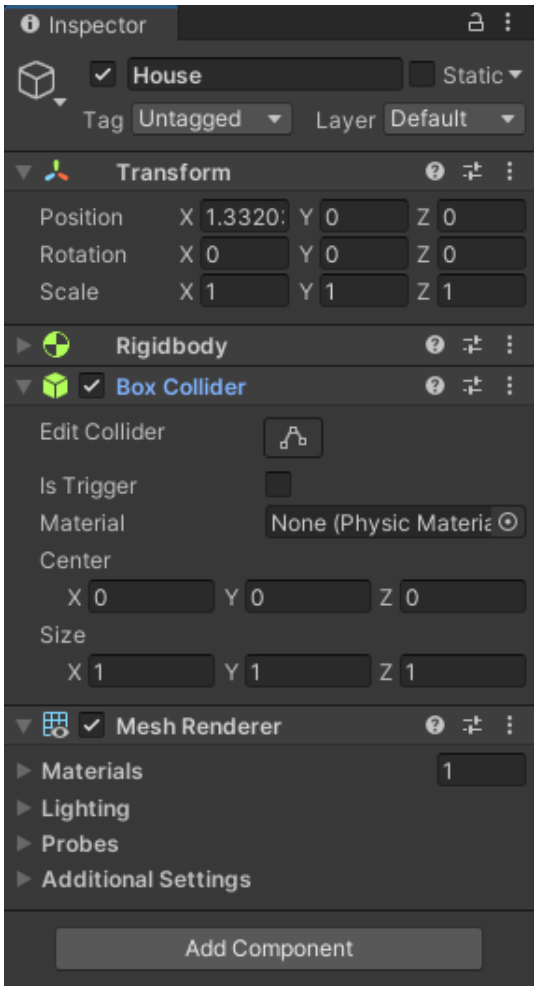
5.12 Irudia: Project leihoa.



5.13 Irudia: Hierarchy leihoa.

5.4.4.3 Inspect leihoa

Eskubikaldean egon ohi da, eta *GameObject*-en osagaiak kudeatzea du helburu. Hemen aukeratutako objektuak dituen propietate guztiak bistaratzen dira, eta horien balioak aldatzeko edo osagai berriak txertatzeko erabiltzen da (5.14 irudia). Objektuen *Layer*-ak ere hemen esleitzen dira.



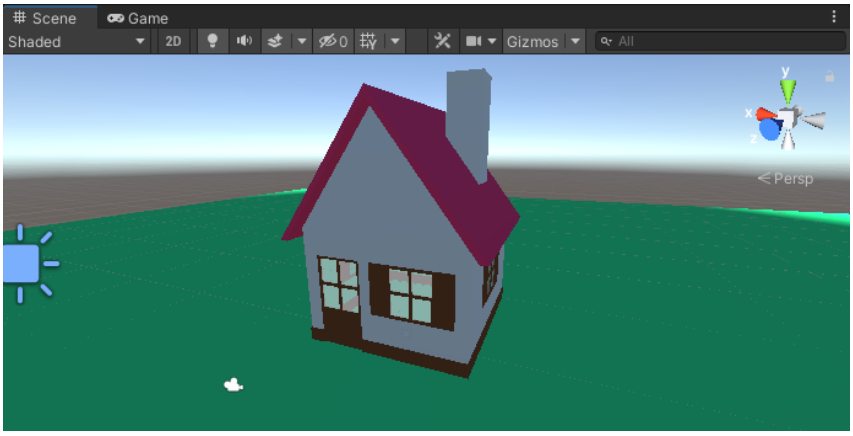
5.14 Irudia: Inspector leihoa.

5.4.4.4 Scene leihoa

Leiho hau editorearen erdian kokatzen da eta ingurune birtuala bistaratzen du (5.15 irudia). Oso erabilgarria da eszenako objektuak aukeratu eta beraien *Transform* atributuak aldatzeko *gizmo*-en bidez. 3D modeloen editoreen moduan funtzionatzen du.

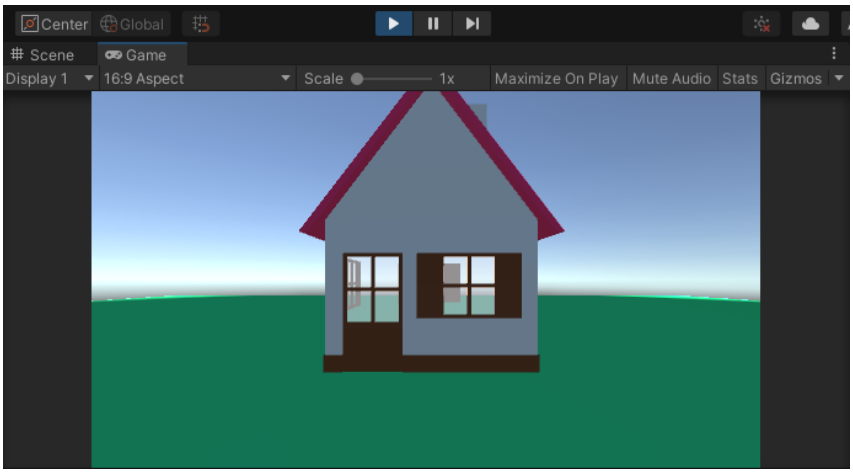
5.4.4.5 Game leihoa

Eszena martxan jartzean, hau da, *Play* botoia sakatzean, *Scene* leihoa ordezkatzen du. Ingurune birtualaren ikuspen libre bat izan ordez, kokatuta dauden kamerek ikusten dutena bistaratzen da (5.16 irudia). Leihoaren oiko ezkerraldeko botoi zabalgarriari esker *Display*



5.15 Irudia: Scene leihoa.

zenbakia aldatu daiteke. *Play* botoia sakatzen den momentuan eszenako programak abian jartzen dira.



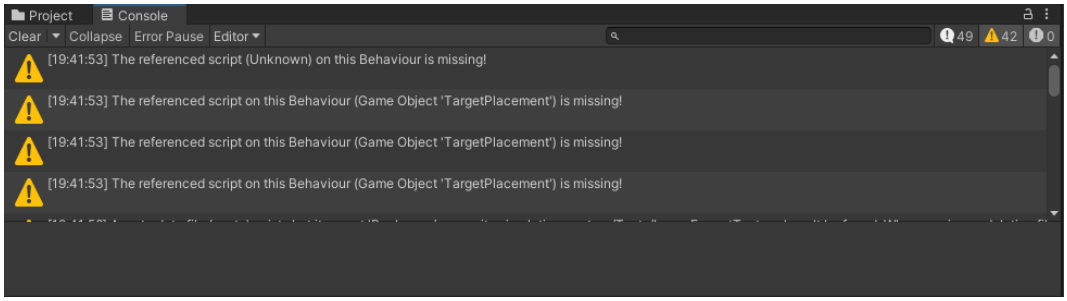
5.16 Irudia: Game leihoa.

5.4.4.6 Console leihoa

Project leihoaren leku berean topatzen da, beste zabalgarri baten moduan (5.17 irudia). Hemen proiektuan dauden erroreak bistaratzen dira, gorriz. Hauek dependentzia erroreak izan daitezke, baita eszenako *script*-en konpilazio edo exekuzio-erroreak. Abisuak eta jakinarazpenak ere pantailaratzen ditu, horiz eta zuriz, hurrenez hurren. Jakinarazpenen barruan programen barruko *print* moduko aginduen irteera bistaratzen da. Unity-k programa bidez mezuak pantailaratzeko agindu pertsonalizatua du: *Debug.Log()*. Leihoaren goiko eskuineko botoiei esker garrantzia-maila konkretu bateko mezuak ezkutatu daitezke.

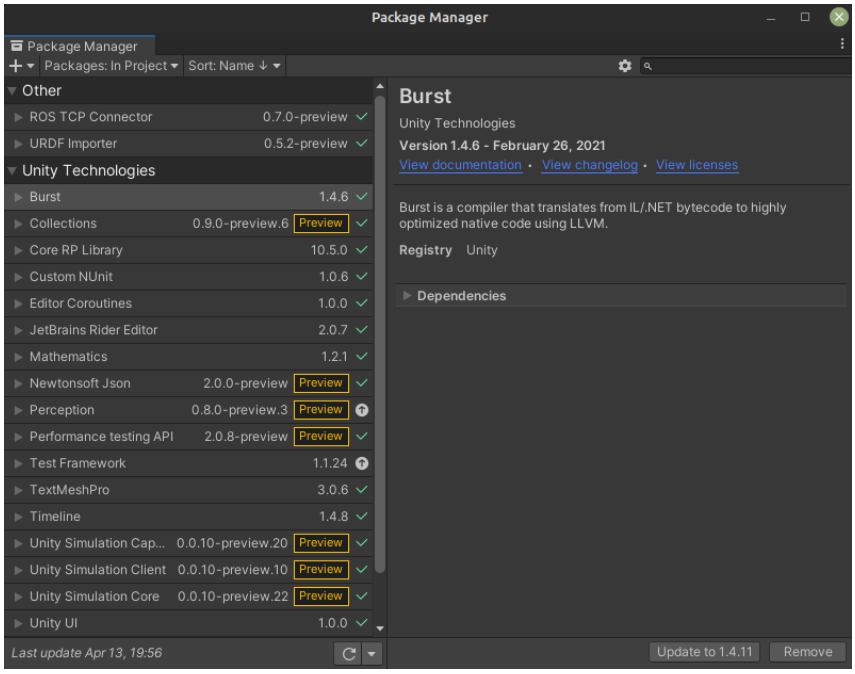
5.4.4.7 Paketeen kudeatzailea (*Package Manager*)

Leiho hau editoraren goiko "Window" zabalgarrian "Package Manager" klikatuta irekitzen da. Hemen proiektua erabiltzen ari den paketeak bistaratzen dira (5.18 irudia). Paketeak gehitzeko aukera ere ematen du, leihoaren goiko ezkerreko "+" botoia sakatu ezkero.



5.17 Irudia: Console leihoa.

Paketeak diskotik edo GitHub-eko URL bitartez gehitu daitezke. Jada sartuta daudenak eguneratzea edo kentzea ahalbidetzen du ere.



5.18 Irudia: Unity-ko paketeen kudeatzailea.

KAPITULUA 6

Niryo Ned simulazioan

6.1 Lehenengo abiapuntua: Pick-and-Place proiektua

Denbora eta baliabideak aurreztearren, Zabot proiektua zerotik hasi beharrean, jada egindako irismen antzeko proiektu publiko batetik abiatzea erabaki da. Noski, Zabot-en planifikazioaren bat egin ahal izateko, proiektua Unity-n eginda egon beharko da, eta robota kontrolatzeko ROS erabili beharko du. Bestalde, helburutzat *pick-and-place* motako zereginak gainditzea izan beharko du, hau da, beso robotikoarekin piezak hartzea eta leku egokian uztea.

Aipatutako hitz gako hauekin bilaketa bat egin ostean, GitHub-en "Pick-and-Place Tutorial" [8] izeneko proiektu bat aurkitu da, "Unity-Robotics-Hub" eremuko tutorialen barruan. Gainera, tutorial baten moduan antolatuta dago, hau da, proiektua garatu ahal izateko egin beharreko pausu guztiak azalduta ematen dira. Halaber, Niryo One[1] robotarekin dabil (Niryo Ned-en oso antzekoa dena), eta xede nagusia robotak inguruko kubo bat hartzea eta oinarri baten gainean uztea da. Unity eta ROS arteko komunikazioa ere gauzatzen du. Zabot proiektuarekin ezaugarri asko partekatzen dituenez, aipatutako lan publikotik abiatzea erabaki da, bertan azaltzen den tutoriala jarraituz.

6.1.1 Pick-and-Place tutoriala

Tutoriala 5 zatitan banatzen da:

- 0. zatia: ROS aldearen prestakuntza
- 1. zatia: Unity eszenaren sorkuntza
- 2. zatia: ROS-Unity integrazioa
- 3. zatia: Pick-and-Place Unity-n
- 4. zatia: Pick-and-Place robot fisikoan

[1] Niryo One robotaren espezifikazio mekanikoak: URL

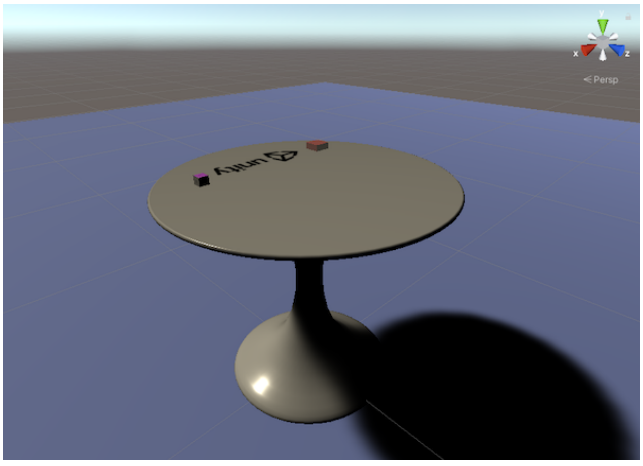
Proiektuaren biltegia GitHub-etik makina lokalera ekarri da, baina Zabot-en irismenera egokitu aurretik, eginda dagoena ulertu eta menperatzeko bere tutoriala pausuz pausu jarraitu da. 4.zatia izan ezik, beste guztiak eraman dira aurrera. Ondoren, jarraitutako prozesu hau sakonago azaltzen da:

6.1.1.1 0. zatia: ROS aldearen prestakuntza

Zati honetan 2 aukera proposatzen dira: Docker-en jada prestatutako irudia erabiltzea hala ROS makina lokalean prestatzea. Bigarren aukerarekin jarraitzea erabaki da. Hortaz, ROS direktorioa GitHub-etik deskargatutako biltegiaren barruan, *Unity-Robotics-Hub/tutorials/pick_and_place/ROS/* direktorioan egongo da. Horrez gain, beharrezko paketeak instalatu dira, ROS Noetic bertsioarekin bat datozenak. Pakete aipagarrienak MoveIt eta robotaren kontrolarekin lotutakoak dira. Azkenik, kode guztia eraikitzeko, ROS karpetaren barruan *source* eta *catkin_make* aginduak exekutatu dira.

6.1.1.2 1. zatia: Unity eszenaren sorkuntza

Lehenik eta behin, Unity Hub instalatu da, eta bertan 2020.3.11f1 (LTS) bertsioko editorea deskargatu da, Pick-and-Place proiektua editore horrekin eginda baitago. Gero, Unity Hub-en aipatutako proiektua ireki da, zehazki, *Unity-Robotics-Hub/tutorials/pick_and_place/PickAndPlaceProject/* direktorioan kokatzen dena. Behin editorearen barruan, eszena berri bat sortu da, eta bertara *Assets/Prefabs/* karpetan kokatzen diren *Table* (beso robotikoa egongo den mahaia), *Target* (hartu beharreko kuboa) eta *TargetPlacement* (pieza uzteko oinarria) 3D objektuak txertatu dira. Ondoren, kamera posizio eta orientazio egokiarekin kokatu da. Eszena 6.1 irudian ikus daiteke.



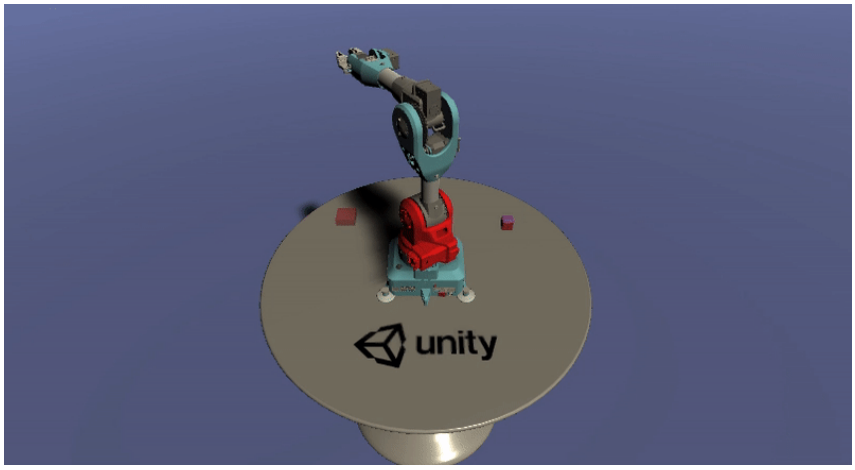
6.1 Irudia: Pick-and-Place tutorialeko eszena 1. atalean.

Robota inportatu aurretik, proiektuaren fisiken konfigurazioan *Solver Type* gisa *Temporal Gauss Seidel* ezarri da, tutorialean gomendatzen den moduan. Hasieran, Unity proiektua ireki den momentuan, Paketeen Kudeatzaileak automatikoki dependentziak aztertu eta instalatu ditu. Horietako bat URDF-Importer[2] da. Honi esker, *Assets/URDF/niryo_one/ niryo_one.urdf* robotaren definizioa inportatu da. URDF fitxategian Niryo Ned beso robotikoa

[2]URDF Importer Unity paketearen GitHub biltegia: URL

posizioa definitzen da, mahaiaren gainean egon dadin. Hala ere, kontuan hartu beharra dago Unity-ko (x,y,z) koordenatuak ROSeko (z,-x,y) koordenatuen parekoak direla.

Azkenik, Niryo Ned *GameObject*-ak dakarren *Controller.cs* programaren parametroak egokitu dira, eta robotaren *base_link* artikulazioa mugiezin moduan ezarri da, besoaren oinarria bere tokitik mugi ez dadin. Unity editoreko *Play* botoia sakatu ezkero, teklatuko gezien bidez beso robotikoaren artikulazioak banan bana aukeratu eta kontrolatu daitezke. Aukeratutako lotura gorriz agertuko da, 6.2 irudian ikusten den moduan.



6.2 Irudia: Robotaren kontrola Pick-and-Place tutorialeko 1. atalean.

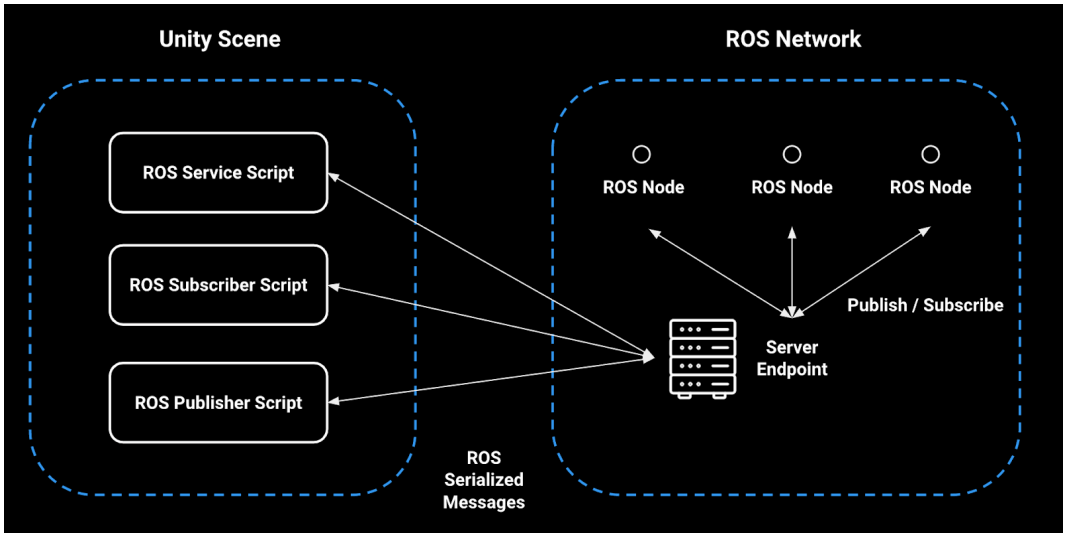
6.1.1.3 2. zatia: ROS-Unity integrazioa

ROS eta Unity arteko komunikazioa ROSeko ROS-TCP-Endpoint[3] eta Unity-ko ROS-TCP-Connector[4] paketeen bidez egikaritzen da.

6.3 irudian ikusten den moduan, ROSen aldean *Server Endpoint* izeneko nodo bat egongo da. Beste nodo guztiek honekin komunikatzeko gaitasuna izango dute. Aldi berean, Unity-ko paketeak eskaintzen duen *ROSConnection* klaseari esker, bertako programek *Server Endpoint* nodoarekin komunikatu ahal izango dute. Komunikazio hau ROSen definitzen den moduan egingo da, hau da, argitalpen/harpidetza edota eskaera/erantzun eredua jarraituz. Horrez gain, bi aldeen artean igarotzen diren mezuak ROSek definitzen duen moduan serializatu beharko dira. Honetarako ere ROS-TCP-Connector paketea baliatuko da, *MessageGeneration* plugin bat eskaintzen baitu ROSeko mezuak eta zerbitzuak C# kodera pasatzeko, baita hauek serializatzeko funtzioak ere.

Beraz, lehenik eta behin, beharrezko ROS mezuak C#-era eraikiko dira. Hau editorean *Robotics -> Generate ROS Messages...* atalean egin daiteke. Proiektuko ROS direktorioa zehaztu beharko da, eta bertan definitzen diren mezuak eraikitzeko aukera emango du. Kasu honetan, *ROS/src/moveit_msgs/msg/RobotTrajectory.msg* mezua beharko da, hemen beso robotikoak egin beharreko ibilbideak (poseen sekuentzia) bidaltzeko mezuak definitzen baitira. Horrez gain, *ROS/src/niryo_moveit/msg/* direktorioko *NiryoMoveitJoints.msg* eta *NiryoTrajectory.msg* mezuak eraiki dira. Lehenengoan robotaren artikulazio bakoitzerako

[3] ROS-TCP-Endpoint ROS paketearen GitHub biltegia: URL
[4] ROS-TCP-Connector Unity paketearen GitHub biltegia: URL



6.3 Irudia: Unity eta ROS arteko komunikazioaren eskema.

balioak nahiz objektua hartu eta uzteko poseak definitzen dira. Bigarrengo mezuan, berriz, *pick-and-place* zeregina egikaritzeko egin beharreko ibilbideen zerrenda bat definitzen da. Robotaren ibilbideak kalkulatzeko zerbitzuari deitzeko jarraitu beharreko formatua *Assets/ROSMessages/NiryoMoveit/srv/MoverService.srv* fitxategian deskribatzen denez, hau ere eraiki behar izan da.

ROSekin komunikazioa ondo dabilela ziurtatzeko, tutorialak eskaintzen duen *SourceDestinationPublisher* C# programa erabiliko da. Hau hartu beharreko objektuaren eta helburuko oinarri-objektuaren posizioak lortzeaz arduratuko da, ROSeko */niryo_joints* izeneko buzoiera bidaltzeko. Programako funtzio garrantzitsuena *Publish()* da. Kodea 6.1 kode zatian ikus daiteke. Aipatu beharra dago Unity-ko koordenatu sistematik ROSekora igarotzeko *.To<FLU>()* metodoa erabiltzen dela. Alderantzizkoa *.To<RUF>()* metodoarekin egingo litzateke.

Hortaz, azaldutako programa erabiltzeko eszenan *Publisher* izeneko *GameObject* huts bat gehitu da, eta honi *SourceDestinationPublisher script*-a txertatu zaio. Programako parametroei eszenako robota, *Target* eta *TargetPlacement* estekatu behar zaizkio, objektuak *Inspector* leihora eramanez. Ondoren, eszenan botoi bat gehitu da, eta honen *OnClick()* eremuan lehen sortutako *Publisher* objektuaren *Publish()* funtzioa ezarri da. Horrela, botoia sakatzean funtzio hori exekutatuko da.

Bestetik, bi aldetako komunikazioa aurrera eramateko Unity-ko paketeari makinaren IP helbidea zehaztu behar zaio. Helbidea *Robotics -> ROS Settings -> ROS IP Address* eremuan jarri behar da. ROSen aldeari dagokionez, nahikoa da terminal batean **roslaunch niryo_moveit part_2.launch** komandoa exekutatzearekin, beharrezko *server_endpoint* eta *trajectory_subscriber* nodoak martxan jartzen baititu. Hau egin ostean, Unity editorean *Play* sakatu eta sortutako botoi berriari eman ezkero, terminalean robotaren artikulazioen nahiz eszenako objektuen informazioa agertzen da. Beraz, ROS eta Unity arteko komunikazioa modu egokian egikaritu da.

```
1   public void Publish()
2   {
3       var sourceDestinationMessage = new NiryoMoveitJointsMsg();
4
5       for (var i = 0; i < k_NumRobotJoints; i++)
6       {
7           sourceDestinationMessage.joints[i] =
                m_JointArticulationBodies[i].GetPosition();
8       }
9
10      // Pick Pose
11      sourceDestinationMessage.pick_pose = new PoseMsg
12      {
13          position = m_Target.transform.position.To<FLU>(),
14          orientation = Quaternion.Euler(90,
                m_Target.transform.eulerAngles.y, 0).To<FLU>()
15      };
16
17      // Place Pose
18      sourceDestinationMessage.place_pose = new PoseMsg
19      {
20          position = m_TargetPlacement.transform.position.To<FLU>(),
21          orientation = m_PickOrientation.To<FLU>()
22      };
23
24      // Finally send the message to server_endpoint.py running in ROS
25      m_Ros.Publish(m_TopicName, sourceDestinationMessage);
26  }
```

6.1 Kode zatia: *SourceDestinationPublisher* klaseko *Publish* funtzioa.

6.1.1.4 3. zatia: Pick-and-Place Unity-n

Aurreko zatian erabilitako *SourceDestinationPublisher* programak 2 aldeetako komunikazioak ongi funtzionatzen duela ziurtatzeko soilik balio du. Atal honetan *script* konplexuago bat proposatzen da: *TrajectoryPlanner*. Programa honek robotaren ibilbideak planifikatzeko eta bere matxarda kontrolatzeko beharrezko logika guztia barneratzen du. ROS aldearekin komunikatzeko *PublishJoints()* (6.2 kode zatia) eta *TrajectoryResponse()* metodoak definitzen ditu. Lehenengoa ROS aldeari eskaera egiteaz arduratzen da. Zehazki, kuboaren eta helburuaren poseak jakiteko, dagozkien *GameObject*-en *Transform* atributuak atzituko ditu, eta informazio hau ROS aldeari bidaliko dio. Bigarren funtzioa, aldiz, jasotako erantzuneko ibilbideak beso robotikoan exekutatzeaz arduratuko da.

Beraz, programa hau erabiltzeko aurreko zatiko pausu berdinak jarraitu behar dira, baina *TrajectoryPlanner*-ekin eginez.

ROS aldean, jasotako *pick_pose* eta *place_pose* aldagaietatik robotak jarraitu beharreko ibilbidea kalkulatzeko logika egongo da. Hau *ROS/src/niryo_moveit/scripts/mover.py* Python kodeak barneratzen du, MoveIt paketeari esker. Esaterako, bertako *plan_trajectory()* funtzioak hasierako robotaren artikulazioen angeluak emanda helburuko posera iristeko egin beharreko ibilbidea kalkulatzen du.

Dena den, funtzio nagusia *plan_pick_and_place()* izango da, zerbitzuaren eskaeratik egin beharreko ibilbideen sekuentzia osoa prozesatzen baitu. Ideia *pick-and-place* sekuentzia

```
1   public void PublishJoints()
2   {
3       var request = new MoverServiceRequest();
4       request.joints_input = CurrentJointConfig();
5
6       // Pick Pose
7       request.pick_pose = new PoseMsg
8       {
9           position = (m_Target.transform.position +
                m_PickPoseOffset).To<FLU>(),
10
11          // The hardcoded x/z angles assure that the gripper is
                always positioned above the target cube before grasping.
12          orientation = Quaternion.Euler(90,
                m_Target.transform.eulerAngles.y, 0).To<FLU>()
13      };
14
15      // Place Pose
16      request.place_pose = new PoseMsg
17      {
18          position = (m_TargetPlacement.transform.position +
                m_PickPoseOffset).To<FLU>(),
19          orientation = m_PickOrientation.To<FLU>()
20      };
21
22      m_Ros.SendServiceMessage<MoverServiceResponse>(m_RosServiceName,
            request, TrajectoryResponse);
23  }
```

6.2 Kode zatia: *TrajectoryPlanner* klaseko *PublishJoints* metodoa.

osoa 4 pausutan egitea da:

- Pre Grasp Pose: Robotaren matxarda *Target* objektuaren gainean kokatu.
- Grasp Pose: Robotarren matxarda jaitsi, behatzak objektuaren 2 aldetan egon daitezen.
- Pick Up Pose: Robotaren matxarda berriz Pre Grasp posiziora igo.
- Place Pose: Besoa *TargetPlacement* objektuaren gainera eraman.

Beraz, pausu bakoitzeko posizioa kalkulatzeko *plan_trajectory()* funtzioari dei egingo dio. Unity aldeko "TrajectoryPlanner" programa arduratuko da 2. posea exekutatu ondoren matxarda ixteaz, eta 4.a bukatu ostean berriz irekitzeaz.

Oraingoan, ROS nodoak abiarazteko hurrengo komandoa exekutatu behar da terminalean: **roslaunch niryo_moveit part_3.launch**. Editorean *Play* sakatu eta pantailako botoiari eman ezkero, robotak kuboa hartzen du eta oinarriaren gainean uzten du, 3 pausutan. Beraz, tutorialaren 3.zatia modu egokian inplementatu da.

6.2 Tutorialeko Unity proiektua Zabot proiekturako egokitzen

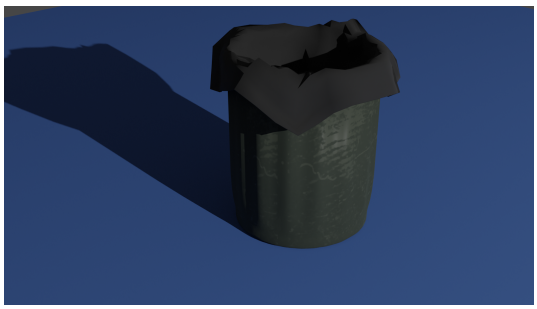
Zabot proiektuaren irismena kontuan izanez gero, tutorialean zehar garatutako proiektua ez da guztiz egokia. Hortaz, egokitzapen batzuk egin behar izan dira.

6.2.1 Eszena eta objektuak egokitzen

Eszenako objektuak eta haien distribuzioa ez dator bat Zabot-en irismenarekin. Hori dela eta, erabiltzen diren 3D objektuak aldatu dira.

Hasteko, inguruneko zaborra birziklatzeko hainbat zakarrontzi beharko dira, kolore ezberdinetakoak. Hasierako prototipo gisa, 3 zakarrontzi ipintzea erabaki da: gorria, berdea eta urdina (RGB). Etokizuneko ikusmen artifizialaren nahiz piezen manipulazioaren atalak sinplifikatzeko, zabor gisa zakarrontzien kolore bereko 3 kubo baliatzea xedatu da. Ideia kolore gorriko kuboa zakarrontzi gorrian uztea izango da, eta berdina beste objektuekin. Zakarrontziak estatikoak izango dira, hau da, aurredefinitutako leku finko batean egongo dira uneoro. Kuboak, berriz, dinamikoak izango dira.

Beraz, hurrengo pausua erabakitako 3D modeloak eskuratzea litzateke. Zakarrontzien kasuan, modeloa Free3D webgunetik deskargatu da[5]. 6.4 irudian ikus daiteke modeloa.



6.4 Irudia: Free3D webgunetik jaitsitako zakarrontziaren 3D modeloa.

Deskargatutako 3D objektuaren kolorea beltza denez, makina lokalean Blender programan kargatu da, eta ontziaren gaineko plastikoaren materialaren kolorea aldatu da. Lehen azaldu bezala, 3 kolore ezberdinetako zakarrontzi gorde dira, denak *.fbx* formatuan, Unity-ko eszenara inportatzeko erraztasunak ematen baititu.

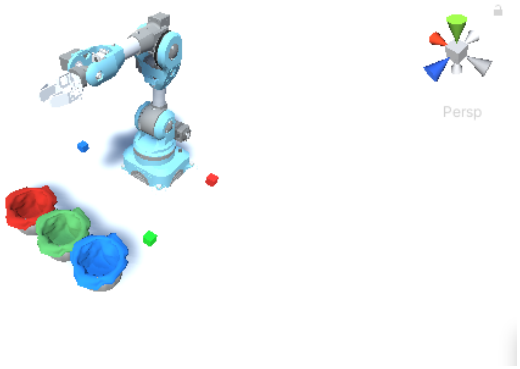
Kuboei dagokienez, forma sinpleko objektuak direla medio, Blender-en zerotik sortu dira. Zakarrontzien kolore bereko 3 bertsio egin dira eta *.fbx* formatuan esportatu dira.

Azkenik, Unity-ko editorean "Test" izeneko eszena berria sortu da, eta lortutako 3D objektuekin populatu da. Objektuen distribuzioa 6.5 irudian begiesten da.

6.2.2 Unity-ko programaren egokitzapena

Txertatutako 3D objektu berrientzat atributuak esleitu dira, eta hartu beharreko objektuak nahiz zakarrontziak gordeko dituzten bektoreak definitu dira, "pickObjects" eta "placeObjects" izenekoak, hurrenez hurren. Aldaketa hauek 6.3 kode zatian ikus daitezke.

[5]Free3D webgunean "lachlandauth" erabiltzaileak sortutako zakarrontziaren 3D modeloa: URL



6.5 Irudia: Pick-and-Place tutorialeko eszena Zabot proiektura egokituta.

```
    [SerializeField]
    GameObject m_RedCan;
    public GameObject RedCan {get=> m_RedCan;set=> m_RedCan=value;}
    [SerializeField]
    GameObject m_GreenCan;
    public GameObject GreenCan {get=> m_GreenCan;set=> m_GreenCan=value;}
    [SerializeField]
    GameObject m_BlueCan;
    public GameObject BlueCan {get=> m_BlueCan;set=> m_BlueCan=value;}

    [SerializeField]
    GameObject m_RedCube;
    public GameObject RedCube {get=> m_RedCube;set=> m_RedCube=value;}
    [SerializeField]
    GameObject m_GreenCube;
    public GameObject GreenCube {get=> m_GreenCube;set=>
        m_GreenCube=value;}
    [SerializeField]
    GameObject m_BlueCube;
    public GameObject BlueCube {get=> m_BlueCube;set=> m_BlueCube=value;}

    private GameObject[] pickObjects;
    private GameObject[] placeObjects;
```

6.3 Kode zatia: *TrajectoryPlanner* klasean gehitutako atributuak.

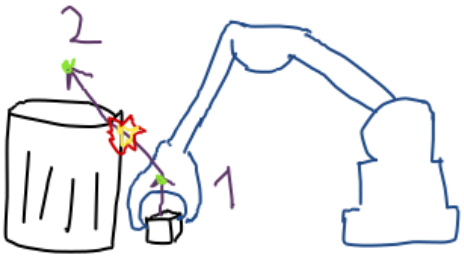
Definitutako bektore berriak hasieratzeko *Start()* funtzioan 2 agindu gehitu dira, 6.4 kode zatian erakusten den moduan. Bektore hauek objektuen *Transform* atributuak indizeen bitartez atzitzeko erabiliko dira. Esaterako, kubo berdea hartu nahi bada, bektoreak 1 indizearekin atzituko dira (2. posizioa), kubo berdearen eta zakarrontzi berdearen *GameObject*-ak eskuratzeko, eta hauetatik poseak lortzeko. Azaldutako atzipenak *PublishJoints()* funtzioan egingo dira, ROSera bidaltzeko eskaera sortzerakoan, E1 eranskinean erakusten den bezala.

```
1    pickObjects = new GameObject[] {m_RedCube, m_GreenCube, m_BlueCube};
2    placeObjects = new GameObject[]  {m_RedCan, m_GreenCan, m_BlueCan};
```

6.4 Kode zatia: *TrajectoryPlanner* klaseko atributu berrien hasieraketak, *Start()* funtzioan.

6.2.3 Bestelako hobekuntzak

Orain arteko inplementazioarekin proba batzuk egin ostean, ikusi da kuboren bat zakarrontzitik oso gertu baldin badago, hau jasotzean talka jasotzeko arriskua dagoela. Izan ere, pieza hartu ostean, besoa pixka bat altxatzen da (1) eta zuzenean zakarrontziaren gainerako ibilbidea egiten du (2), 6.6 irudian bistaratzen den moduan.



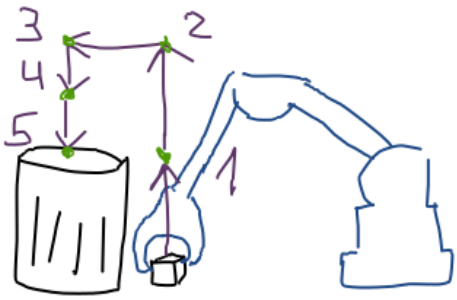
6.6 Irudia: Pick-and-Place tutorialeko mugimendu planifikatzailearen talka-arriskua.

Hau ekiditeko, kuboari eutsi ostean besoa altuera nabari batera altxaraztea erabaki da. Horrez gain, etorkizunean forma konplexuagoko eta altuera handiagoko objektuak izan daitezkeela kontsideratu da. Beraz, besoa zakarrontziaren gainean altuera handiago batean kokatzeko prestatu da.

Konponketa hauek egin ostean, beste hainbat proba gauzatu dira. Proba hauetan ikusi da beso robotikoaren abiadurarekin lotutako beste arazo batzuk sortu direla. Esaterako, piezari heldu ostean, robota altuera nabarmen batera higitzen da, baina hasierako denbora-tarte berean. Hortaz, mugimendu hori abiadura handiagoarekin exekutatzen du eta, ondorioz, batzuetan pieza matxardatik askatzen da.

Beraz, ibilbidea pausu gehiagotan egikaritzea erabaki da. Zehazki, robotak pieza 2 pausutan altxatuko du, eta zakarrontziaren gainera iritsi ostean, uzteko prozesua ere 2 pausutan egingo du. Ibilbidea 6.7 irudian bistaratzen da.

Horrez gain, lehen robotak *pick-and-place* ibilbidea pieza zakarrontzian askatu ostean bukatzen zuen. Hortaz, beste pieza bat hartu beharko balu, besoa zakarrontziaren barruan egongo litzateke. Hori dela eta, kuboa utzi ondoren robota berriz altxarazi egiten da.



6.7 Irudia: Beso robotikoaren ibilbidea egindako zuzenketen ondoren.

Aipatutako aldaketak ROS aldean inplementatu dira, bertan egiten baita ibilbidearen kalkulua. Zehazki, *ROS/src/niryo_moveit/scripts/mover.py* fitxategia editatu da. Lehen posizioen sekuentzia osoa 4 pausukoa zen, baina orain 7koa izango da. Definitutako pose berriak *pre_grasp_pose2*, *pick_up_pose2* eta *place_pose2* moduan izenpetu dira.

Egindako egokitzapenen zuzentasuna bermatzeko helburuarekin, eszenako kubo guztiak biltzeko hainbat exekuzio egin dira. Horietako baten bideoa ere grabatu da[6]. Proba hauei esker, ikusi da egindako inplementazioa zuzena dela.

6.3 Beste abiapuntu posiblea: Pose Estimation proiektua

Tutoriala aurrera eraman ostean, ikusi da Pick-and-Place proiektuan piezen posizioak hasieratik ezagunak direla, Unity-ko eszenatik atzitzen baitira. Baina hau ezingo litzateke mundu fisikoan egin eta, horregatik, helburua posizioak kamera baten bitartez estimatzea da. Bilaketa bat egin ostean, arazo honi konponbidea eman liezaiokeen Unity proiektu libre bat aurkitu da: "Object Pose Estimation" [9]. Proiektu hau GitHub-en aurkitzen da, eta Pick-and-Place bezala, "Unity-Robotics-Hub" eremuko tutorialen barruan kokatzen da. Beraz, proiektua pausuz pausu garatzeko eta ulertzeko tutoriala eskaintzen du.

Pose Estimation proiektuaren helburu nagusia inguruneko pieza bat ikusmen artifizialaren bitartez antzematzea eta beso robotikoarekin *pick-and-place* zeregina gauzatzea da. Beraz, Zabot garatzeko oso lagungarria izango litzateke. Hala ere, proiektu honek Niryo Ned robota erabili ordez UR3[7] baliatzen du. Gainera, Pick-and-Place tutoriala jada aurrera eraman da eta beharrezko aldaketak egin zaizkio. Beraz, ez dago garbi Pose Estimation proiektuarekin hasteak merezi duenik.

Haursnarketa baten ostean, Pose Estimation proiektuan murgiltzea oso lagungarria izango dela ikusi da, eta etorkizunean Zabot-en garapenean denbora aurrezte handi bat ekarriko duela ondorioztatu da. Hori dela eta, proiektu berri honen tutoriala aurrera eramatea erabaki da.

6.3.1 Pose Estimation tutoriala

Tutorial hau ere 5 zatitan banatzen da:

[6] Ikusmen artifizialik gabe simulazioan kuboak biltzeko egindako proba: bideoa
[7] UR3 robotaren espezifikazio mekanikoak: URL

- 0. zatia: ROS aldearen prestakuntza
- 1. zatia: Unity eszenaren sorkuntza
- 2. zatia: Eszena datuak biltzeko prestatu
- 3. zatia: Datuak bildu eta modeloa entrenatu
- 4. zatia: Pick-and-Place

Proiektuko kontzeptuak eta metodologiak ulertzeko, GitHub biltegia deskargatu ostean tutoriala pausuz pausu jarraitu da.

6.3.1.1 0. zatia: ROS aldearen prestakuntza

Lehenik eta behin, proiektuak eskatzen dituen ROS eta Python paketeak instalatu dira. Deep Learning modelo bat erabiltzen duenez, PyTorch eta antzeko liburutegiak behar ditu. Ondoren, ROS biltegia eraiki da *catkin_make* aginduaren bitartez, baina soilik pakete konkretu batzuk eraikiz. Azkenik, makina lokalaren IP helbidea *src/ur3_moveit/config/params.yaml* fitxategian ezarri da.

6.3.1.2 1. zatia: Unity eszenaren sorkuntza

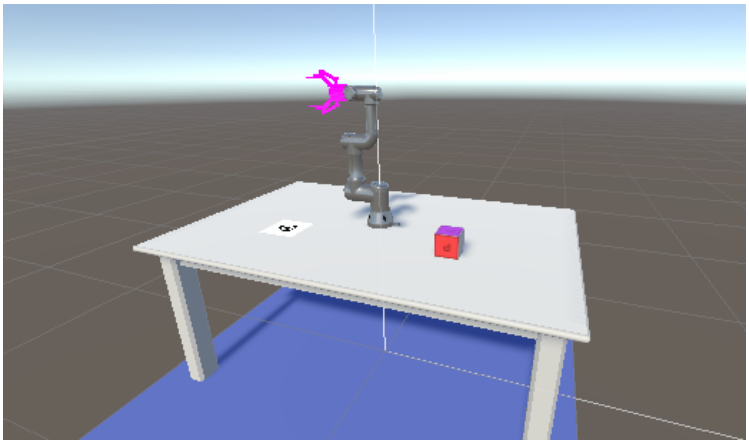
Kasu honetan, 2020.2.6f1 bertsioko Unity editorea instalatu da, eta honekin Universal Render Pipeline (URP) erabiltzen duen proiektu bat sortu da. Hitz gutxitan, Render Pipeline bat eszenaren errenderizazioan jarraitu beharreko pausuak deskribatzen dituen modelo kontzeptuala da. Kontzeptu hauek Zabot proiektuaren irismenetik kanpo geratzen direnez, nahikoa da jakitearekin Pose Estimation tutorialeko paketeren batek URP erabiltzea eskatzen duela.

Proiektua zerotik sortu denez, beharrezko Unity paketeak manualki instalatu behar izan dira. Zehazki, URDF Importer, TCP Connector eta Perception[8] paketeak instalatu dira, GitHub estekak erabiliz. Lehenengo biak jada ezagunak dira. Hirugarrenak eszena datuak biltzeko prestatzeko balio du, eszena ausazkotzeko tresnak nahiz kamerentzat datu-bilketa programak eskaintzen ditu eta. Perception paketeko funtzionalitate batzuk URP erabiltzen duten proiektuetan soilik dabiltza.

Perception paketeak, kameratik eskuratutako irudiak etiketatutako datu-base moduan gordetzeko, *Ground Truth Renderer Feature* osagaia ezarpenetan gehitzea eskatzen du. Hau egiteko, *Assets/* karpetan kokatutako *ForwardRenderer.asset* fitxategia aukeratu da, eta *Inspector* leihoan "Add Renderer Feature" botoiarekin aipatutako osagaia gehitu zaio. Hau egin ostean, eszena prestatzen hasi daiteke.

"TutorialPoseEstimation" izeneko eszena berri bat sortu da, eta bertako kamera eta argiaren propietateak tutorialean azaltzen den moduan egokitu dira. Ondoren, tutorialak eskaintzen dituen 3D objektuak deskargatu eta txertatu dira. Guztira 5 objektu gehitu dira: hartu beharreko kuboa, kuboa uzteko oinarria, mahaia, mahaiaren azpiko zorua eta beso robotikoa, mahaiaren gainean. UR3 robotaren definizioa URDF Importer paketearen bitartez kargatu da. Eszenaren egoera 6.8 irudian ikus daiteke. Amaitzeko, beso robotikoaren *Controller.cs script*-aren parametroak egokitu dira eta oinarria mugiezina izan dadin esan zaio.

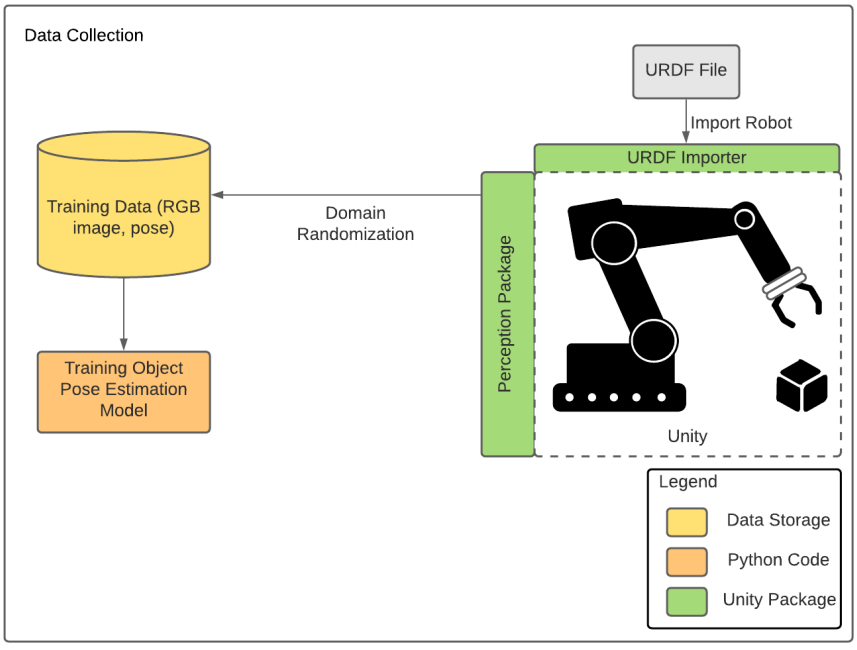
[8] Perception Unity paketearen GitHub biltegia: URL



6.8 Irudia: Pose Estimation tutorialeko eszena 1. atalaren ostean.

6.3.1.3 2. zatia: Eszena datuak biltzeko prestatu

6.9 irudian adierazten den bezala, helburua Unity-ko simulaziotik Deep Learning modelo bat entrenatzeko datu-basea biltzea izango da. Datu-base honetan eszenaren instantzia ezberdinen irudi mordoa pilatuko da, eta irudi bakoitzeko hartu beharreko kuboaren posea erregistratuta izango da, etiketa moduan. Horrela, modeloa eszenaren irudi batetik kuboaren posizioa eta orientazioa ateratzeko entrenatuko da.



6.9 Irudia: Datuak biltzeko eta modeloa entrenatzeko prozesuaren eskema.

Baina esandako datu-bilketa egin ahal izateko, eszena simulazioko frame bakoitzeko aldatu beharko da. Hau da, *Domain Randomization* teknika [10] aplikatuko da. Hau Perception paketeko *Scenario* eta *Randomizer*-ekin lor daiteke. Azken finean, *Randomizer*-ak kodetutako programak dira, objektu konkretu batzuen parametroak ausazko modu batean

aldatzea eragiten dutenak. *Scenario*-a simulazioaren fluxua kontrolatzeaz eta gehitutako *Randomizer*-en parametroak egikaritzeaz arduratzen da. Frame bakoitzeko exekuta daiteke.

Baina ezer egin aurretik, eszenako kamera datuak biltzeko prestatu behar da. Horretarako, editoreko *Game* leihoan kamerarentzat 650x400 bereizmena gehitzea proposatzen da, entrenamenduko nahiz inferentziako irudi guztiek tamaina bera izan dezaten. Ondoren, kamera *GameObject*-ari "Perception Camera.cs" izeneko *script*-a txertatu zaio. Programa hau irudiak biltzeaz eta etiketatzeaz arduratzen da. Baina ongi funtziona dezan, proiektuko konfigurazioan *Asynchronous Shader Compilation* desaktibatu behar izan da.

"Perception Camera" osagaiari *Labeler* izeneko kontzeptuak gehitu diezazkioke. *Labeler*-ak irudiak etiketa konkretu batzuekin anotatzeaz arduratzen dira. Esaterako, Pose Estimation proiektuaren kasuan kuboaren posizioa eta orientazioa behar direnez, objektuaren *Bounding Box*-ari dagokion anotazioa egiten duen *BoundingBoxLabeler3D* gehituko da. Labeler hau Perception paketeak inplementatuta ekartzen du jada.

Baina noski, *Labeler*-ari etiketatu beharreko objektuak zeintzuk diren adierazi behar zaio. Hau *Id Label Config* fitxategiei esker adierazi daiteke. Beraz, *Assets/* karpetan *IdLabelConfig* berria sortu da, eta *BoundingBoxLabeler3D* osagaiari estekatu zaio. Horrez gain, kuboaren *GameObject*-ari *Labeling.cs script*-a gehitu zaio, eta bertan "cube_position" izeneko etiketa sortu da eta *IdLabelConfig*-ekin lotu da. Horrela, kamerak irudietan agertzen diren kuboak fitxategi batean etiketatuko ditu, "cube_position" eremuaren barruan. Kontuan izan beharra dago *Bounding Box*-aren informazioa (posizioa, orientazioa eta tamaina) kameraren koordenatu-sistemaren arabera anotatzen dela. Beraz, Deep Learning modeloak kuboaren posizioa eta orientazioa kameraren erreferentzia-sisteman kalkulatuko ditu.

Hau egin ondoren, *Domain Randomization* teknika inplementatu da. Lehenik, eszenan "Simulation Scenario" izeneko *GameObject* hutsa sortu da eta *Pose Estimation Scenario.cs script*-a txertatu zaio. Horrela, *Scenario* moduan funtzionatuko du. Programako *Automatic Iteration* eremua aktibatuta egonez gero, eszena frame bakoitzeko aldatuko du. Ondoren, kuboaren Y ardatzeko biraketa ausaz aldatzeko *YRotationRandomizer.cs Randomizer*-a estekatu zaio. Tutorialak jada inplementatuta eskaintzen du (6.5 kode zatia).

Programak Perception paketeak eskaintzen dituen *Parameters* klasea baliatzen du. Klase honek ausazko balioak lagintzeko aukera eskaintzen du. Lagindu nahi den balioaren motaren arabera, aldaera asko existitzen dira: *FloatParameter*, *BooleanParameter*, *IntegerParameter*... Horrez gain, programak *Randomizer* klasetik heredatzen du, *Scenario*-ra gehitu daitezkeen *Randomizer* guztien oinarria baita eta funtzio lagungarriak eskaintzen baititu.

Dena den, *Randomizer*-ek objektuengan eragiteko *RandomizerTag*-en laguntza behar dute. *RandomizerTag*-ak objektuei lotzen zaizkien programak dira, eta etiketa moduan funtzionatzen dute. Beraz, *Randomizer*-ek *RandomizerTag* konkretu bat bila dezakete, eta bertan definitzen diren funtzioei dei egin diezaiekete, edota *Tag* hori duten objektu guztiak eskuratu. Hortaz, *Randomizer*-ak funtziona dezan, kuboari tutorialeko *YRotationRandomizerTag.cs* txertatu zaio (6.6 kode zatia).

Unity simulazioa abian jarri ezkero, hartu beharreko kuboak frame bakoitzeko biraketa bat jasaten du Y ardatzean. Horrez gain, eszenak bere *Bounding Box*-a bistaratzen du uneoro. Beraz, orain arte ongi inplementatu da.

Y ardatzeko biraketaz gain, objektuaren posizioa ausazkotzea komeni da. Ataza hau garatzeko beharrezko programak ere inplementatuta ematen dira. *Randomizer*-a "Robo-

```
public class YRotationRandomizer : Randomizer
{
    public FloatParameter rotationRange = new FloatParameter { value
        = new UniformSampler(0f, 360f) }; // in range (0, 1)

    protected override void OnIterationStart()
    {
        IEnumerable<YRotationRandomizerTag> tags =
            tagManager.Query<YRotationRandomizerTag>();
        foreach (YRotationRandomizerTag tag in tags)
        {
            float yRotation = rotationRange.Sample();

            // sets rotation
            tag.SetYRotation(yRotation);
        }
    }
}
```

6.5 Kode zatia: Pose Estimation tutorialeko *YRotationRandomizer.cs* programa.

```
public class YRotationRandomizerTag : RandomizerTag
{
    private Vector3 originalRotation;

    private void Start()
    {
        originalRotation = transform.eulerAngles;
    }

    public void SetYRotation(float yRotation)
    {
        transform.eulerAngles = new Vector3(originalRotation.x,
            yRotation, originalRotation.z);
    }
}
```

6.6 Kode zatia: Pose Estimation tutorialeko *YRotationRandomizerTag.cs* programa.

tArmObjectPositionRandomizer" izango da, eta honek *SurfaceObjectPlanner* izeneko C# klasean definitutako funtzioak baliatuko ditu. Azken hau objektuaren kokalekuak baldintza konkretu batzuk betetzeaz arduratzen da: ez duela beste objektu batekin talka egiten, beso robotikoarentzat atzigarria dela eta mahaiaren gainean kokatzen dela.

Bestalde, *RandomizerTag*-a *RobotArmObjectPositionRandomizerTag* izango da, eta etiketa moduan funtzionatzeaz gain, "mustBeReachable" aldagai boolear bat definituko du, editorearen interfazetik bere balioa aldatu daitekelarik. Aldagai horren balioa true izanez gero, *Randomizer*-ak kuboa beso robotikoarentzat irisgarria den posizio batean utziko du.

Azkenik, simulaziotik errealitaterako trantsizioaren errorea minimizatzeko helburuarekin, eszenako argiaren propietate batzuk ausazkotu dira. Zehazki, argiaren intentsitatea, orientazioa eta kolorea parametrizatu dira, tutorialeko *LightRandomizer.cs* eta *LightRandomizerTag.cs* programei esker. Simulazioa martxan jarriz gero, eszenak desiragarria den

portaera erakusten du.

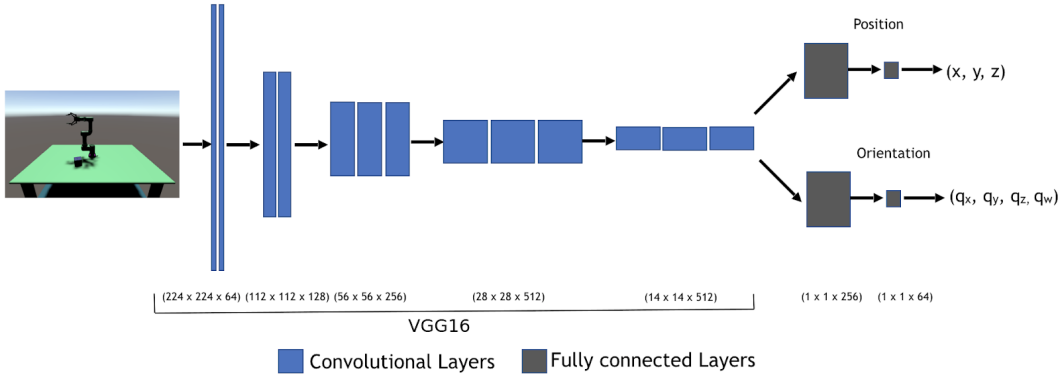
6.3.1.4 3. zatia: Datuak bildu eta modeloa entrenatu

Eszena datuak biltzeko prestatuta dagoenez, simulaziotik modeloa entrenatu ahal izateko datu-base bat eskura daiteke. *Train* multzorako 30000 irudi etiketatu bilduko dira eta *Validation*-erako 3000. Beraz, hasteko, *Simulation Scenario* objektuko "Total Frames" parametroari 30000 balioa esleitu zaio, eta *Play* botoia sakatu da. Ausazkotasun hazi moduan 539662031 ezarri da. Bukatzean, automatikoki datu-basea gordetzen du, kamera objektuko *Perception Camera* osagaiak erakusten duen direktorioan. *Train* multzoaren karpetari "UR3_single_cube_training" izena eman zaio. Barruko "RGB…" karpetak kamerak ateratako irudiak biltzen ditu, eta "Dataset…"-ean irudi guztien anotazioak gordetzen dira, *.json* formatuko fitxategietan. Anotazioaren formatuaren adibide gisa E2 eranskina dago.

Validation multzoa lortzeko prozesu berdina jarraitu da, baina kasu honetan, 3000 frame iteratu dira eta ausazkotasun haziari 539662032 balioa eman zaio, bi multzoak berdinak izan ez daitezen. "UR3_single_cube_validation" izenarekin gorde da.

Tutorialak Deep Learning modelo bat definitzeko eta entrenatzeko beharrezko fitxategiak eskaintzen ditu. Beraz, deskargatu egin dira. Entrenamendu prozesua GPU bidez exekutatzeko dependentziak *environment-gpu.yml* fitxategian biltzen dira. Hauek guztiak Anaconda bidez instalatu dira, "pose-estimation" izeneko ingurune birtual batean.

Inplementatutako modeloa sare neuronal konboluzionaletan (CNN) oinarritzen da. Eskeleto moduan ImageNet [11] datu-basearen gainean aurrez entrenatutako VGG16 sarea erabiltzen du. Metodo honi *Transfer Learning* deritzo. VGG16-ren azken FC (*Fully Connected*) geruzak kendu egin zaizkio, eta Pose Estimation ataza konkreturako behar direnak txertatu zaizkio. Zehazki, VGG16 eskeletoaren irteera 2 FC bloketan banatuko da. Lehenengoa kuboaren posizioa kalkulatzeaz arduratuko da eta bigarrenak orientazioa aterako du. Orientazioa kuaternoi bektore baten bidez adierazten denez, bektore hau normalizatzeko *LinearNormalized* izeneko geruza pertsonalizatu bat erabiltzen da. Modeloaren eskema grafikoa 6.10 irudian ikusten da, eta Pytorch-eko inplementazioa 6.7 kode zatian.



6.10 Irudia: Pose Estimation tutorialeko modeloaren eskema grafikoa

Modeloa entrenatzeko fitxategien artean *config.yaml* aurkitzen da. Erosotasuna bermatzearren, bertan parametro eta hiperparametro guztien esleipenak agertzen dira, 6.11 irudian ageri den moduan. *Train* nahiz *Validation* multzoak kokatzen diren makina lokaleko direktorioa zehaztu behar izan da, baita entrenamenduaren prozesuaren irteera non gorde.

```
        self.model_backbone = torchvision.models.vgg16(pretrained=True)
        # remove the original classifier
        self.model_backbone.classifier = torch.nn.Identity()

        self.translation_block = torch.nn.Sequential(
            torch.nn.Linear(25088, 256),
            torch.nn.ReLU(inplace=True),
            torch.nn.Linear(256, 64),
            torch.nn.ReLU(inplace=True),
            torch.nn.Linear(64, 3),
        )
        self.orientation_block = torch.nn.Sequential(
            torch.nn.Linear(25088, 256),
            torch.nn.ReLU(inplace=True),
            torch.nn.Linear(256, 64),
            torch.nn.ReLU(inplace=True),
            torch.nn.Linear(64, 4),
            LinearNormalized(),
        )

    def forward(self, x):
        x = self.model_backbone(x)
        output_translation = self.translation_block(x)

        if self.is_symetric == False:
            output_orientation = self.orientation_block(x)
            return output_translation, output_orientation

        return output_translation

```

6.7 Kode zatia: Pose Estimation tutorialeko modeloaren inplementazioa.

Entrenamendu prozesua hasteko nahikoa da *config.yaml* fitxategia dagoen direktorioan terminal bat irekitzearekin, eta bertan "pose-estimation" ingurunea aktibatu ondoren, hurrengo komandoa exekutatzearekin:

```
python -m pose_estimation.cli train
```

Aldiz, modeloa ebaluatu nahiz izanez gero, hurrengo komandoa exekutatu beharko da:

```
python -m pose_estimation.cli evaluate
```

Makina lokalean 2 aginduak probatu dira, eta ongi funtzionatzen dutela ikusi da. Dena den, entrenamenduak ordu asko irauten ditu. Hori dela eta, 4. zatirako tutorialak eskaintzen duen modelo entrenatua deskargatu eta erabiliko da.

6.3.1.5 4. zatia: Pick-and-Place

Atal honetan entrenatutako Deep Learning modeloa *pick-and-place* atazak aurrera eramateko erabiliko da.

```yaml
estimator: UR3_single_cube_model
train:
  dataset_zip_file_name_training: UR3_single_cube_training
  batch_training_size: 30
  accumulation_steps: 10
  epochs: 25
  beta_loss: 1
  sample_size_train: 0
val:
  dataset_zip_file_name_validation: UR3_single_cube_validation
  batch_validation_size: 30
  eval_freq: 4
  sample_size_val: 0
test:
  dataset_zip_file_name_test: UR3_single_cube_validation
  batch_test_size: 30
  sample_size_test: 0
dataset:
  image_scale: 224
  download_data_gcp: False
  gcs_bucket: None
  pose_estimation_gcs_path: None
  symmetric: False
adam_optimizer:
  lr: 0.0001
  beta_1: 0.9
  beta_2: 0.999
checkpoint:
  load_dir_checkpoint: None
  save_frequency: 1
system:
  log_dir_system: /home/aimar/Documentos/output_models
  data_root: /home/aimar/Documentos/Train30k_Val3k
```

6.11 Irudia: Entrenamendu prozesuko parametro eta hiperparametroen definizioa config.yaml fitxategian.

Lehenik eta behin, modeloaren parametroen fitxategia (UR3_single_cube_model.tar) proiektuaren *ROS/src/ur3_moveit* direktorioan kokatu da. Ondoren, Unity-ko eszenan ROSekin komunikatzeko beharrezko *Prefab*-a txertatu da, *Assets/TutorialAssets/Prefabs/Part4* karpetan kokatzen dena. *Prefab*-aren barruan eszenako 3 botoi daude. Lehenengoak beso robotikoaren posizioa hasieratzeko balioko du, izan ere, Deep Learning modeloa robota hasierako posizioan ageri diren irudiekin entrenatu da. Beraz, robota mugitu ostean hasierako posiziora itzularazi beharko da. Bigarren botoiak eszena ausazkotzeko balioko du, *Scenario*-ko *Randomizer*-ei dei eginez. Hirugarrena ROSekin komunikazio abiarazteko izango da, irudia publikatuz.

Halaber, editoreko *Robotics -> ROS Settings* leihoan makina lokalaren IP helbidea ezarri da. Pick-and-Place tutorialean bezala, ROSekin komunikatzeko *TrajectoryPlanner.cs script*-a baliatzen da. Baina oraingoan ROSeko zerbitzariari eszenaren irudi bat bidali beharko dio, bertan modeloak inferentzia prozesua aurrera eraman dezan. Hau *InvokePoseEstimationService()* funtzioaren bitartez egikarituko da. Hau bukatzean, erantzun gisa kuboaren posea jasoko du, hau da, bere posizio eta orientazioaren informazioa. Erantzuna tratatzeaz *PoseEstimationCallback()* metodoa arduratuko da. Kuboaren posizioarekin eta orientazioarekin *PublishJoints()* funtzioari dei egingo dio, eta hau ROSen MoveIt paketearekin komunikatuko da beso robotikoaren ibilbidea kalkulatzeko. Eszena martxan jarri aurretik, datu-bilketa prozesua amaitu denez gero, *Simulation Scenario* objektuaren "Automatic Iteration" eremua desgaitu da. Horrela, *Randomizer*-aren deiak automatikoki egin ordez, botoi baten

bidez egingo dira. Halaber, kameraren *Perception Camera.cs* programa desaktibatu da, irudi gehiago bildu ez ditzan. Hau guztia egin ostean, ROSeko beharrezko nodoak martxan jarri dira ondorengo komandoaren bitartez:

```
roslaunch ur3\_moveit pose\_est.launch
```

Unity-ko eszena abian jarriz gero, botoi guztiek ongi funtzionatzen dutela ikusi da. Irudia ROSera bidaltzeko botoiari eman eta denbora tarte batera, beso robotikoa kuboaren posiziora mugitzen da, hartu egiten du eta oinarriaren gainean uzten du. Hala ere, onartu beharra dago prozesua errore batekin egiten duela. Gainera, kasu batzuetan ez da gai kuboa hartzeko, ausaz robotaren atzean agertzen baita, hau da, irisgarria ez den posizio batean.

6.4 Ingurune fisikoaren konfigurazioa

Proiektua simulazioan garatzearen arrazoia adimen artifizialeko modelo bat entrenatzeko irudiak eta datuak biltzea da. Gero simulaziotik errealitaterako trantsizio bat gauzatuko denez, ingurune fisikoaren eta birtualaren artean ezberdintasunik ez izatea komeni da, simulazioan entrenatutako modeloak errealitatean ongi funtziona dezan. Hori dela eta, Unity-ko proiektuarekin jarraitu aurretik, ingurune fisikoaren konfigurazioa zehaztu eta prestatu da.

Niryo Ned beso robotikoa egurrezko mahai zirkular baten gainean instalatzea erabaki da. Mahaiaren erradioa 0.55 metrokoa da, beraz, robotaren irismena guztiz aprobetxatzea ahalbidetuko du. Dena dela, jakinik atzeko eremuko irismenik ez duela, robota erdi-erdian kokatu ordez, aurreko aldean azalera gehiago utzi zaio, 6.12 irudian ikusten den moduan.

6.12 Irudia: Robotaren posizioa mahai zirkularraren gainean.

Halaber, inguruko zaborra 3 klasetan sailkatzea erabaki da: organikoa, papera nahiz kartoia, eta plastikoa. Beraz, 3 ontzi prestatu dira klase bakoitzerako; gorria, urdina eta berdea, hurrenez hurren. Ondoren, mahaiaren izkin batean kokatu dira. Beraien kokaleku zehatza 6.13 irudian erakusten da.

Hasiera moduan, ingurunean errealitateko zaborra kokatu ordez, kolore nahiz tamaina ezberdinetako kuboak prestatu dira. Hauek 6.14 irudian ikusten dira. Kubo bakoitza kolore

6.13 Irudia: Eszena fisikoko ontziak eta haien posizioa mahaian.

bereko ontzian uztea izango da helburua (salbuespen bezala, kubo horiak ontzi berdean). Xurgagailua kuboei ondo eusteko gai izan dadin, piezak gomaz estali dira. Proba fisikoetan ez dira beti pieza guztiak kokatuko; batzuetan gutxiago ala zero ipiniko dira. Horrela, robota zabor kopuru aldakorreko egoeretan funtzionatzeko prest egon beharko luke.

6.14 Irudia: Eszena fisikoko kuboak.

Azkenik, eszenaren argazkiak aterako dituen RealSense kamera finkatu da. Kamera hau robotaren gainean egongo da, behera begira, euskarri bati esker. Horrela, goiko perspektiba batetik ingurune osoaren argazkiak ateratzeko gai izango da. Euskarriaren oinarria robotaren atzean kokatu da. Kameraren eta euskarriaren posizio zehatzak 6.15 irudian begies daitezke.

6.15 Irudia: Kamera eta bere euskarria eszena fisikoan.

6.5 Ingurune fisikoa simulaziora eramaten

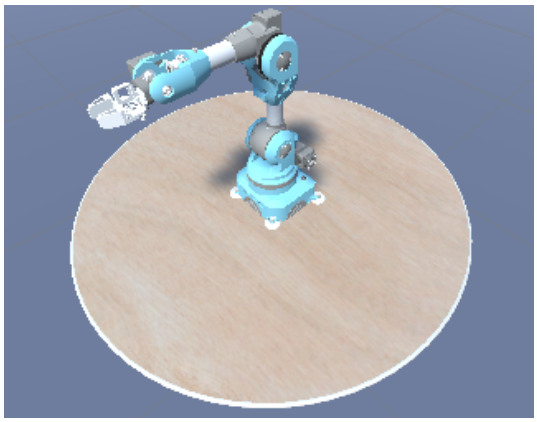
Ikusmen artifizialak ongi funtzionatu ahal izateko, simulazioko eszena ingurune fisikoaren ahalik eta antzekoena izan beharko luke. Beraz, Unity-ko eszenan zenbait aldaketa egin dira.

Aldaketa hauek Pick-and-Place tutorialeko eszena hobetuaren gainean egin dira. Hasiera batean Pose Estimation tutorialeko eszenatik abiatzea aukera hobea dirudi, baina saiakera batzuen ostean, shader-ekin bateraezintasun arazoak daudela ikusi da, proiektuak URP erabiltzen baitu. Ondorioz, Niryo Ned robota eta beste objektu batzuk testurarik gabe inportatzen dira, eta guztiz arrosa kolorekoak ikusten dira. Arazo hauei konponbidea eman ezin izan zaienez, Pick-and-Place tutorialeko eszena hartuko da oinarritzat.

Bestalde, kontuan izan beharra dago mundu fisikoan robotak xurgagailua baliatuko duela piezak manipulatzeko, tutorialetan agertu den pintza erabili ordez. Hori dela eta, xurgagailuarekin datorren Niryo Ned robotaren URDF definizioa aurkitzeko bilaketa bat eman da Interneten, baina ez da topatu. Beraz, simulazioan matxarda erabiliko da. Honek simulazioan piezak manipulatzeko informazio gehiago behar izatea dakar. Izan ere, xurgagailua erabiliz gero, hartu beharreko objektuaren zentrua eta altuera jakitearekin nahikoa da. Aldiz, matxarda baliatzean objektuaren orientazioa ere jakin behar da. Objektuak konplexuak baldin badira, prozesua asko zaildu daiteke [12]. Dena den, simulazioko matxardatik ingurune fisikoko xurgagailurako trantsizioa erraza izatea aurreikusten da.

Abiapuntuko eszena birtuala ingurune fisikoaren antzekoa izan dadin, objektu guztien oinarritzat errealitateko egurrezko mahai zirkularraren parekoa den *GameObject* bat ezarri da. Modeloa Pick-and-Place tutorialetik atera da, bertan maila zirkular bat erabiltzen baita. Baina honek kolore zuriko testura bat erabiltzen duenez, egurrezko testura batekin ordezkatu da. Halaber, eszenaren proportzioak kontutan hartuz, mahaiaren dimentsioak ingurune fisikoarekin parekatu dira, kontuan izanik Unity editoreko distantzia unitate

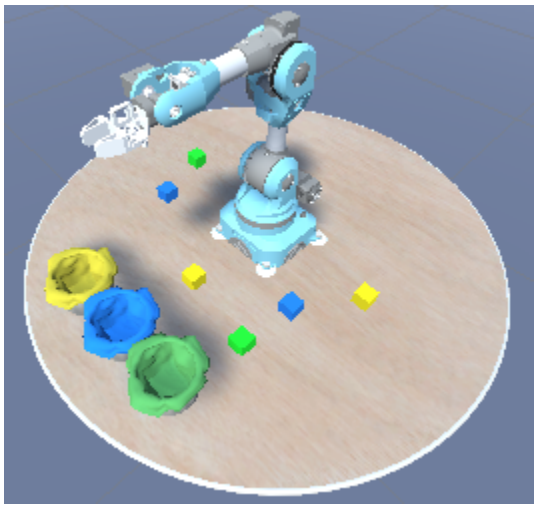
1 errealitateko metro baten parekoa dela. Ingurune birtualeko mahaia 6.16 irudian ikus daiteke.



6.16 Irudia: Ingurune fisikoko mahaia Unity simulazioan.

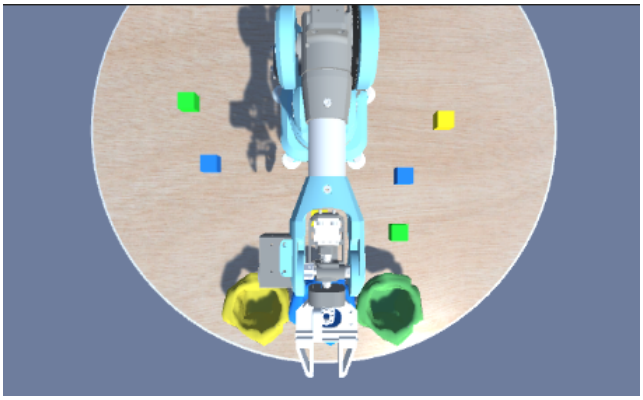
Horrez gain, zaborraren sailkapenerako zakarrontzien koloreak egokitu dira. Orain koloreak berdea (organikoa), urdina (papera eta kartoia) eta horia (plastikoa) izango dira. Hartu beharreko piezen koloreak ere egokitu dira: gorriaren ordez horia ezarri da.

Halaber, jakinik robotaren inguruan zabor kopuru asko egon daitekeela, eszenan 3 kubo jarri ordez 6 kokatu dira; 2 kubo kolore bakoitzeko. Aldaketa hauen guztien ostean lortutako eszena 6.17 irudian bistaratzen da.



6.17 Irudia: Kuboak eta zakarrontziak Unity simulazioan.

Hori gutxi balitz, eszenako kamera nagusia ingurune fisikoarekin bat datorren posizioan kokatu da. Orain robotaren gainean kokatzen da, behera begira; robota, zakarrontziak eta inguruko piezak ikusten dituelarik. Kameraren Z bektorea (nora begiratzen duen zehazten duena) mahaiaren gainazalarekiko perpendikularra izango da. Horrela, argazkien sakonerako informaziotik piezen altuera ateratzea tribialagoa izango da, baita beraien X nahiz Y koordenatuak inferitzea ere. Laburbilduz, azaldutako kameraren kokalekuak Deep Learning modeloaren entrenamendu prozesua eta eraginkortasuna hobetuko du. Kamerak simulazioan duen perspektiba 6.18 irudian begiesten da.



6.18 Irudia: Eszena birtualeko kameraren perspektiba.

Baina orain dagoen moduan, kamerak RGB irudiak errenderitzatzen ditu. Hau da, ez du sakonerako informaziorik eskaintzen.

6.6 RGB-D kamera birtuala

Kamerari RGB-D argazkiak ateratzeko gaitasuna eskaintzeko, Samarth Brahmbhatt garatzailearen gida [13] jarraitu da. Bertan azaltzen den moduan, proiektuan "RGBD Material" izeneko material berria sortu da, eta shader gisa gidan proposatzen den *Unlit/RGBDShader* ezarri zaio. Shader honek *_CameraDepthTexture* aldagaiaren bitartez kameraren sakonerako informazioa eskuratuko du, eta errenderizatzen den RGBA irudiaren alfa kanalean informazio hori idatziko du, 6.8 kode zatian azaltzen den moduan. Hortaz, kameratik urrutiago dauden pixelak gardenagoak izango dira.

```
1    fixed4 frag(output o) : COLOR
2    {
3     float depth = UNITY_SAMPLE_DEPTH(tex2D(_CameraDepthTexture, o.uv));
4     depth = pow(Linear01Depth(depth), _DepthLevel);
5     return depth;
6    }
```

6.8 Kode zatia: Sakonerako informazioa prozesatzeko shader-a.

Baina *_CameraDepthTexture* atzitu ahal izateko, kamerak sakonerako informazioa emateko prest egon behar du. Hau hurrengo aginduaren bitartez lor daiteke:

```
Camera.main.depthTextureMode = DepthTextureMode.Depth;
```

Agindu horrek eszenako kamera nagusia eskuratzen du eta bere *depthTextureMode* atributua aldatzen du. Hortaz, Unity-ko editorean kamerak "MainCamera" tag-a izatea ziurtatu behar da.

Azkenik, gidan azaltzen den moduan, kamerak *Game* leihoko *Display* batean errenderizatu ordez, *Assets/* karpetan kokatu den 256x256 tamainako testura baten gainean errenderizatzeko esan zaio. Horrela, 6.9 kode zatiari esker, testuraren informazioa eskuratu daiteke eta *.png* formatura kodetu.

```
Camera cam = Camera.main;
RenderTexture render_tex = cam.targetTexture;
int W = render_tex.width;
int H = render_tex.height;

// sortu testura bat informazioa gordetzeko
Texture2D rgbd_im = new Texture2D(W, H, TextureFormat.RGBAFloat,
    false);

// GPU -> CPU
RenderTexture prev = RenderTexture.active;
RenderTexture.active = render_tex;
// Irakurri pixelak GPUtik
rgbd_im.ReadPixels(new Rect(0, 0, W, H), 0, 0);
//Kodetu PNG formatura
byte[] pngBytes = rgbd_im.EncodeToPNG();
```

6.9 Kode zatia: Kameraren irteerako testuraren informazioa eskuratzeko eta gordetzeko kode-zatia.

6.7 Pose Estimation funtzionalitateak Zabot-era ekartzen

Pose Estimation tutoriala gauzatu ondoren, ikusmen artifizialaren inplementazioan behar diren funtzionalitate asko baliatzen direla ikusi da. Beraz, Pose Estimation proiektuko funtzionalitate batzuk Zabot-eko eszenara ekartzea erabaki da.

6.7.1 Pertzepzio kamera

Ekarritako lehenengo funtzionalitatea pertzepzio kamerarena izan da. Honetarako, proiektuan Perception paketea instalatu da, eta eskaintzen duen *Perception Camera.cs script*-a Zabot proiektuko kamera nagusiari erantsi zaio. Horrez gain, programaren osagaien artean *BoundingBox3DLabeler* bat sortu da, eta *Id Label Config* berri bat estekatu zaio. Aldi berean, eszenako 6 kuboetako bakoitzari *Labeling.cs script*-a gehitu zaio, eta sortutako *Id Label Config*-arekin lotzeko etiketa bana definitu zaie: “green_cube1”, “green_cube2”, “blue_cube1”, “blue_cube2”, “yellow_cube1” eta “yellow_cube2”. Laburbilduz, kamerak irudiak 6 kuboen *Bounding Box*-aren informazioarekin anotatuko ditu, *.json* formatuko fitxategietan, kubo bakoitzaren informazioa dagokion etiketaren barnean egongo delarik, E3 eranskinean erakusten den moduan.

6.7.2 Ausazkotasuna

Bigarrenik, datu-bilketarako, eta exekuzioak errealitatera ahalik eta gehien hurbildu daitezen, eszena ausazkotzeko beharrezko osagaiak ekarri dira.

Hasteko, *Scenario* gisa funtzionatuko duen *SimulationScenario* objektua sortu da, eta Perception paketeko *Pose Estimation Scenario* programa txertatu zaio. Pose Estimation tutorialeko *Randomizer* eta *RandomizerTag* guztiak ere ekarri dira, eta dagokien objektuetan gehitu dira.

Dena dela, Pose Estimation proiektuak kuboaren Y ardatzeko errotazioa soilik ausazkotzen du, eta hau ez dator bat Zabot-en irismenarekin. Izan ere, errealitatean zaborra edozein tamainatakoa izan daiteke. Hori dela eta, *YRotationRandomizerTag.cs* programan

objektuaren eskala aldatzeko balioko duen *SetScale()* funtzio berria gehitu da, 6.10 kode zatian begies daitekeena.

```
1    public void SetScale(float scalex, float scaley, float scalez)
2    {
3        transform.localScale = new
             Vector3(1.3f*scalex,1.3f*scaley,1.3f*scalez);
4    }
```

6.10 Kode zatia: *YRotationRandomizerTag* programan gehitutako *setScale* funtzioa.

Halaber, *YRotationRandomizer.cs script* nagusian *SetScale()* metodoari dei egingo zaio, ausaz lagindutako eta 3 ardatzetako bakoitzerako eskala biderkatzaile batekin. Gainera, erabiltzaileari Unity-ko editorearen bidez *uniformScale* aldagai boolearraren balioa aldatzen utziko zaio. Aldagai honen balioa true izanez gero, objektuaren 3 ardatzetan eragiketa berdina aplikatuko da, hau da, eskalaketa uniformea egingo da. Biderkatzaileen laginketa tartea erabiltzaileak defini dezake ere. Zabot proiektuaren kasuan, [0.6,1.5] arteko balioak ematea erabaki da, objektua simulazioko pintzarentzat handiegia ala txikiegia izan ez dadin. Aplikatuko diren eskalaketak ez-uniformeak izango dira.

Pose Estimation tutorialeko azken atalean kuboaren ausazko posizioak hobegarriak direla ikusi da, batzuetan robotaren atzean ipintzen baita, besoa irisgarria ez den toki batean. Halaber, UR3 beso robotikoaren irismen-mugak ez datoz bat Niryo Ned-enekin eta, gainera, eszenan zakarrontzi estatikoak kokatu dira. Hori guztia dela eta, *RobotArmObjectPosition-Randomizer.cs Randomizer*-ari, eta konkretuki honek baliatzen duen *SurfaceObjectPlanner.cs* programari hobekuntza batzuk egin zaizkio. Aipatutako azken programako *Passes()* funtzioak lagindutako posizio bat egokia den ala ez begiratzen du. Beraz, Niryo Ned robotaren irismenarekin bat datozen bete beharreko distantzia baldintza gehiago definitu dira. Zakarrontziengandik distantzia minimo bat bermatzeko ere Z ardatzeko posizioari murriztapen bat ezarri zaio. Murriztapen hauen balioak Unity editoreko *Scene* leihoan objektuak mugituz eta *Inspector* leihoan transformazioen balioak begiratuz eskuratu dira.

Bestalde, jakinik Zabot proiektuaren helburua zabor kopuru aldakorreko ingurune batean funtzionatzea dela, hau simulatzeko ere *SurfaceObjectPlanner* programa egokitu da. Ideia kuboei eszenan agertzeko %50eko probabilitatea ematea izango da. Beraz, *PlaceObject()* metodoan objektua kokatu behar den momentuan, [0,1] arteko zenbaki erreal bat lagintzen da, eta lagindutako zenbakia 0.5 baino handiagoa bada, Z ardatzeko posizio gisa 100 balioa ezartzen zaio. Ondorioz, kameraren ikus-eremutik kanpo geratuko da. Hala ere, hau egin aurretik ziurtatu da kamerak argazkian agertzen diren kuboak soilik anotatzen dituela. Hau da, kubo horien Z posizio gisa 100 ezartzen bada, kamerak ez ditu ikusiko, eta anotazioen *.json* fitxategian haien etiketak ez dira agertuko, nahiz eta *Id Label Config*-era estekatuta egon.

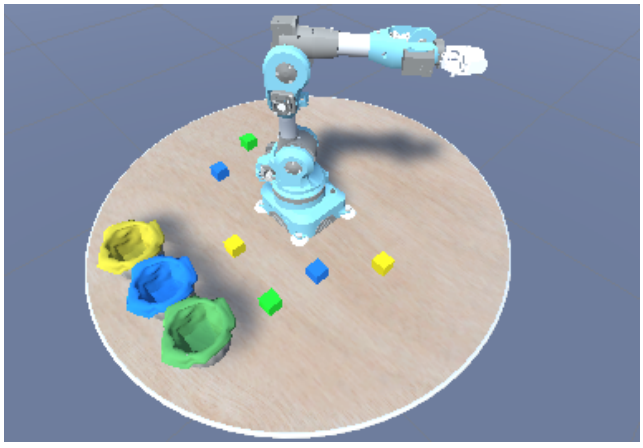
Azkenik, Pose Estimation proiektutik eszena ausazkotzeko botoia ekarri da. Honek datu-bilketa amaitu ostean exekuzioak egiteko balioko du, *Game* leihoan botoiari eman ezkero *Randomizer*-ak iterazio bakar batean soilik exekutatzen baitira.

6.7.3 Robota hasieratu

Pose Estimation tutorialean ikusi da ikusmen artifizialaren eraginkortasuna maximizatzeko exekuzioro robota hasierako posiziora eramatea komeni dela. Hau egin gabe ere Deep Learning modelo bat entrenatu liteke, eta agian sakoneraren informazioari esker beso robotikoari kasurik ez egiten ikasiko luke. Ondorioz, robotaren egoera eta posizioarekiko independenteki funtziona lezake. Baina honek beste arazo bat ekarriko luke: oklusioa. Izan ere, kameraren posizioa dela eta, robotaren besoak inguruko zabor batzuk ikustea eragotzi dezake, bereziki besoaren azpian kokatzen badira. Honen aurrean, modeloak zaborrik geratzen ez dela erabakitzen duenean, beso robotikoa beste leku batera eraman beharko litzateke, azpian izan zitzakeen piezak bistan gera litezen.

Beste aukera bat Unity-ko *Layer*-ak erabiltzea izango litzateke. Hauei esker, Niryo Ned objektuari *Layer* konkretu bat esleitu liezaioke, eta kamerari *Layer* horretako objektuak ez errenderizatzeko esan. Horrela, kamerak eszena beso robotikorik gabe ikusiko luke eta, beraz, oklusioaren arazoa desagertuko litzateke. Baina hau ingurune fisikora eramatean ez litzateke egokia izango, errealitatean ezin baita horrelako operaziorik egin: kamerak robota derrigorrez ikusiko du.

Azkenean, Zabot proiektuan aipatutako 2 aukeren tarteko bide bat hartzea erabaki da. Zehazki, exekuzioro robota hasierako posiziora eramango da, baina konfigurazio horrek oklusiorik eragingo ez duela bermatuko da. Hau da, hasierako posizio gisa robotaren artikulazio nagusia zentrutik -130 gradu biratuko da eta, ondorioz, besoa izkin batean geratuko da, azpian zaborrik izango ez duelarik. Robotaren hasierako konfigurazio berria 6.19 irudian erakusten da.



6.19 Irudia: Niryo Ned robotaren hasierako konfigurazio berria.

Azaldutako metodoa inplementatzeko, Pose Estimation proiektuko robota hasieratzeko botoia eszenan txertatu da, eta beharrezko funtzioak *TrajectoryPlanner.cs script*-era ekarri dira. Ondoren, *ResetRobotToDefaultPosition()* funtzioan lehenengo artikulazioaren helburuko errotazioa 130ra aldatu da.

6.7.4 ROS aldeko programak

Hartu beharreko piezen poseak estimatu ahal izateko beharrezko ROS fitxategiak ere Pose Estimation proiektutik kopiatu dira .

Lehenik eta behin, *ROS/src/ur3_moveit/src/ur3_moveit/setup_and_run_model.py* programa *ROS/src/niryo_moveit/src/niryo_moveit* karpetara ekarri da, entrenatutako modeloaren definizioa eta inferentzia prozesurako beharrezko funtzioak eskaintzen baititu. Ondoren, *ROS/src/ur3_moveit/scripts/* direktorioko *script* guztiak ("mover.py" izan ezik) R*OS/src/niryo_moveit/scripts/* karpetan kopiatu dira. Unity eta ROS arteko lehenengo komunikaziorako *ROS/src/ur3_moveit/srv/PoseEstimationService.srv* zerbitzuaren definizioa ere ekarri da.

Zerbitzu berri hau martxan jarri ahal izateko *ROS/src/niryo_moveit/launch/part_3.launch* fitxategian zenbait aldaketa egin dira:

- Lehen, Unity-rekin komunikatzeko, ROSeko *endpoint* zerbitzari moduan ROS-TCP-Endpoint paketeak eskaintzen duen *default_server_endpoint.py* programa erabiltzen zen, baina orain Pose Estimation proiektuko *server_endpoint.py* baliatuko da. Hortaz, nodo hau exekutatzeko beharrezko lerroak gehitu dira launch fitxategian, eta aurreko zerbitzariarenak kendu dira.
- Posearen estimaziorako zerbitzariari (*pose_estimation_script.py*) dagokion nodoa abiarazteko lerroa idatzi da, 6.11 kode zatian begiesten dena, alegia.

```
1    <node name="pose_estimation" pkg="niryo_moveit" type= "
         pose_estimation_script.py" args="--wait" output="screen"/>
```

6.11 Kode zatia: Posearen estimaziorako ROS aldeko *part_3.launch* fitxategian gehitutako nodoa.

Hortaz, hurrengo komandoa exekutatu ezkero, ROS aldea prest egongo da Unity-rekin komunikatzeko eta jasotzen dituen irudietatik kuboen poseak estimatzeko:

```
roslaunch niryo_moveit part_3.launch
```

6.8 Ikusmen artifiziala simulazioan

6.8.1 Datu-bilketa

Datu-bilketarako Perception paketeko *Perception Camera.cs script*-a baliatzen da. Baina Pose Estimation tutorialeko 1. atalean aipatzen den bezala, Perception paketeko funtzionalitate batzuk ongi funtziona dezaten, esaterako, irudien anotazioa, proiektuak URP erabili behar du. Eta Zabot-en Unity proiektuak ez du URP erabiltzen, shader-ekin eta testurekin arazoak ematen baititu. Laburbilduz, Zabot-en proiektuan ezin dira kameraren irudiak anotatu URP erabiltzen ez duelako, baina URP erabili ezkero testura arazoak agertzen dira eta, beraz, kamerak ateratzen dituen argazkiak ez dira zuzenak.

Dilema honen aurrean, eszena bera URP erabiltzen duen proiektu batean garatzea erabaki da. Horrela, URP proiektua soilik Perception paketearekin irudien anotazioak biltzeko erabiliko da, baina ateratzen diren argazkiak ez dira gordeko. RGB-D argazkiak biltzeko URP erabiltzen ez duen proiektua baliatuko da, bertan testura arazorik ez baitago.

Ikerketa baten ostean jakin da Unity-ko proiektuko edukiak *.unitypackage* fitxategi batera esportatzeko aukera dagoela, gero beste proiektu batean ireki ahal izateko. Hortaz,

URPko eszena garatzeko prozesua zerotik errepikatu ordez, proiektua esportatuz egin da. Zehazki, Unity editoreko goiko menuan *Assets/Export Package...* aukerari sakatu zaio, eta fitxategi guztiak aukeratu ostean, *Export...* botoiari eman zaio. Ondoren, esportatu den proiektuaren editore bertsio berarekin (2020.3.11f1) URP erabiltzen duen proiektu bat sortu da, eta *Assets/Import Package/Custom Package...* aukerarekin inportatu egin da. Aldaketa txiki batzuk ere egin behar izan dira, esate baterako, *Ground Truth Renderer* Feature osagaia ezarpenetan gehitu.

Bilduko den datu-basea 2 zatitan banatzen da: *Train* eta *Validation*. Train multzorako 50000 irudi anotatu biltzea erabaki da, *Scenario*-ren ausazko hazi gisa 539662031 erabiltzen delarik. *Validation* multzoak, aldiz, 5000 irudi izango ditu, 539662032 haziarekin egindako exekuziotik lortutakoak.

Beraz, parametro horiek kontuan harturik, multzo bakoitzerako irudien anotazioak biltzeko exekuzio bana egin da URP erabiltzen duen proiektuan. Hauek bildu ostean, URP erabiltzen ez duen proiektura jo da anotatutako frame bakoitzaren argazkiak eskuratzeko. Baina Perception paketeak ongi funtzionatzen ez duenez, manualki argazkiak ateratzeko *Get Frame RGBD.cs script*-a garatu da. Programa honek frame bakoitzeko exekutatzen den *Update()* funtzioa inplementazen du. Zehazki, iterazioro kamerak errenderizatu duen testura irakurtzen du eta *Assets/Captures /* direktorioan gordetzen du, Perception paketeak baliatzen duen izendapen berdin-berdina jarraituz. Programa hau eszenan "FrameCapturer" izeneko *GameObject*-ari txertatu zaio, eta eszena martxan utzi da irudiak biltzeko.

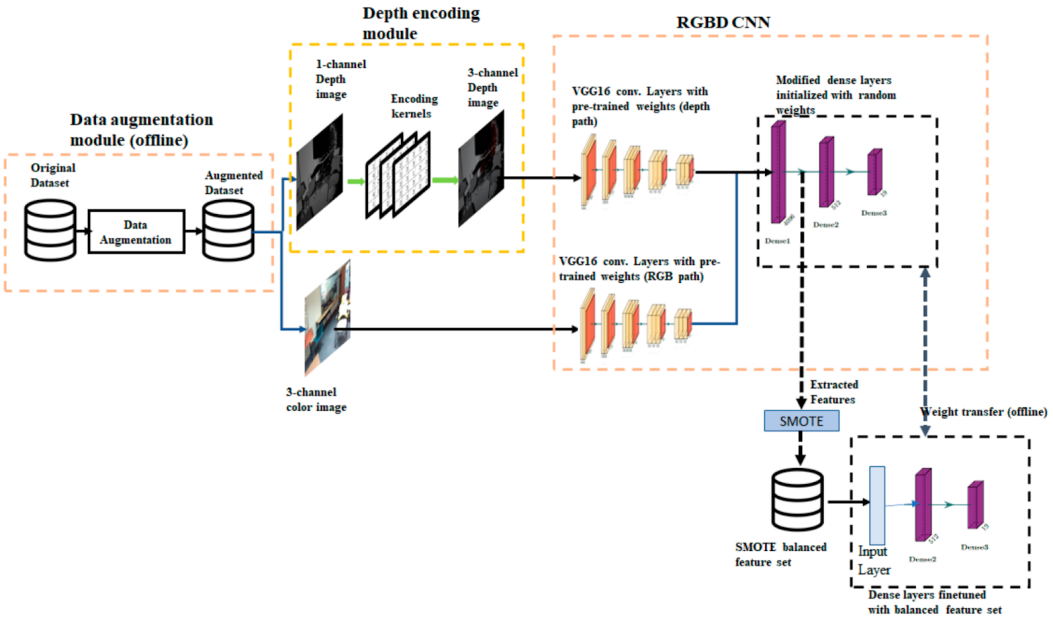
Arraroa badirudi ere, ikusi da "FrameCapturer"-ak gutxi gorabehera 7000 argazki gordeta daramatzanean Unity aplikazioa eta ordenagailu guztia izozteko joera duela, inolako erremediorik gabe. Hau saihesteko, argazki bilketa 5000ko multzoka egitea erabaki da. Esaterako, *Train* multzoaren kasuan, hasieran eszena 5000 iterazioz utzi da martxan eta dagozkien irudiak bildu dira. Ondoren, 10000 iterazioz utzi da martxan, baina soilik 5000-10000 frame-ei dagozkien argazkiak bildu dira. Hau egiteko nahikoa da *Get Frame RGBD.cs* programan kontagailu bat eta baldintza egitura bat gehitzearekin. Prozesua 50000 irudiko *Train* multzoa eskuratu arte errepikatu da. Noski, *Validation* multzoko argazkiak exekuzio bakarrean bildu dira.

6.8.2 Lehenengo modeloaren arkitektura

Deep Learning modeloaren artitekturaren oinarria Pose Estimation tutorialetik hartu da. Tutorialeko modeloak sarrera moduan RGB irudi bat jasotzen du soilik, baina Zabot-en kasuan, RGB kanalez gain Depth kanala ere jasoko du. Gainera, tutorialean irteera gisa kuboaren posizioaren eta orientazioaren informazioa soilik itzultzen du. Baina Zabot proiektuko objektuak altuera eta klase ezberdinetakoak izan daitezkeenez, objektuaren Y ardatzeko eskala (altuera) eta bere mota (organikoa, papera/kartoia, plastikoa) ere inferitu beharko ditu. Hori guztia dela eta, arkitekturan zenbait aldaketa egin behar izan dira.

Oinarrizko arkitekturak eskeleto moduan ImageNet datu-basearen gainean entrenatutako VGG16 modeloa baliatzen du. ImageNet-eko irudiak RGB motakoak dira, eta beraz, VGG16 eskeletoa ez dago prestatuta RGB-D kanaletako irudiak prozesatzeko. Ondorioz, simulazioko eszenaren irudien RGB kanalak jatorrizko moduan prozesatuko dira, eta D kanala beste alde batetik tratatzea erabaki da. Hala ere, D kanalaren prozesamenduari buruzko zalantzak daudenez, antzeko arazoak ebatzi dituzten artikulu zientifikoak ikertu dira. Horien artean, Radhakrishnan et al. -en lanean [14] sakonerari dagokion kanala

VGG16 modeloarekin prozesatzea proposatzen da, baina aurretik konboluzio eragiketa batzuk aplikatzen zaizkio, 3 kanaleko irudi batera bihurtzearren eta VGG16-rentzat sarrera onargarria izateko. Jatorrizko RGB kanalak beste VGG16 modeloaren instantzia batekin tratatzen dira, eta hau egin ostean, kanal guztietatik prozesatutako informazioa bildu egiten da eta FC geruzetara bidaltzen da. Artikuluan proposatzen den arkitekturaren xehetasunak 6.20 irudian bistaratzen dira.



6.20 Irudia: Radhakrishnan et al. -en artikuluan proposatzen den arkitekturaren eskema.

Zabot proiekturako metodo bera aplikatzea erabaki da. Hortaz, datu-basea egituratzeko balio duen *single_cube_dataset.py* programako *pre_processing()* funtzioan irudiko sakonera-ren informazioa bananduta irakurtzeko aginduak gehitu dira, Python-eko PIL paketeko *Image* klasea baliatuz, 6.12 kode zatian ikusten den bezala.

```
    image_RGB = Image.open(image_name).convert("RGB")
    transform = self.get_transform()
    imageRGB = transform(image_RGB).unsqueeze(0)

    image_Depth = Image.open(image_name).convert("RGBA").getchannel("A")
    imageDepth = transform(image_Depth).unsqueeze(0)

```

6.12 Kode zatia: Datu-basea egituratzean irudien sakonerako informazioa bananduta irakurtzeko aginduak.

Ondoren, *model.py* fitxategian arkitekturaren definizioa aldatu da. Zehazki, sakonera-ren informazioaren prozesaketaz arduratuko den bigarren VGG16 eskeletoa definitu da. Sakonerako kanala aurreprozesatzeko konboluzio geruza definitu da ere. Artikulu zientifikoan proposatzen den moduan, geruzaren parametroak filtro gaussiar konstanteei dagozkienak izango dira. Hortaz, parametro hauek ez dira optimizatu beharko, eta Pytorch-i gradienteak ez kalkulatzeko esan beharko zaio. Honen guztiaren inplementazioa 6.13 kode

zatian ikus daiteke.

```
    self.depth_preprocess = torch.nn.Conv2d(1, 3, kernel_size=5, stride
    =1, padding=2)
    self.depth_preprocess.weight = torch.nn.Parameter(torch.FloatTensor
    ([[[
          [1, 4, 7,  4, 1],
          [4, 16, 26, 16, 4],
          [7, 26, 41, 26, 4],
          [4, 16, 26, 16, 4],
          [1, 4, 7,  4, 1]]],


        [[[0, 0, 0, 0, 0],
          [0, 16, 26, 16, 0],
          [0, 26, 41, 26, 0],
          [0, 16, 26, 16, 0],
          [0, 0, 0, 0, 0]]],


        [[[0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0],
          [0, 0, 1, 0, 0],
          [0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0]]]]))
    self.depth_preprocess.weight.requires_grad = False

```

6.13 Kode zatia: Irudien sakonerako informazioa prozesatzeko modeloaren konboluzio geruza estatikoa.

Modeloaren irteera ere egokitu beharra dago, baina ez dago garbi zein izango den output-a. Izan ere, eszenako kubo kopurua ez dago definituta; 0 eta 6 arteko edozein zenbaki adina pieza egon daitezke. Eta ezaguna den moduan, Deep Learning modeloen arkitekturak estatikoak dira: geruza guztien sarrera nahiz irteera kopuruak hasieratik definituta daude eta aldaezinak dira. Hortaz, irteera kopurua ezin daiteke sarrerako irudiaren arabera aldatu, pentsa zitekeen bezala.

Irtenbide moduan, modeloak beti objektu bakarrarentzako informazioa kalkulatzea erabaki da. Objektu hori aukeratzeko irizpide asko existitzen dira, baina kasu honetan, pieza altuena kontutan hartzea erabaki da, sarrera moduan sakoneraren informazioa erabiltzen dela aprobetxatuz. Esaterako, modeloari 4 kubo dituen irudi bat bidali ezkero, kubo altuenari dagozkion posizioa, orientazioa, altuera eta klasea inferitu beharko ditu. Eszenan zaborrik ez dagoen kasu partikularrerako laugarren klase berri bat definitzea erabaki da. Beraz, berriz ere *single_cube_dataset.py* programako *pre_processing()* metodoa egokitu da, E4 eranskinean begies daitekeen bezala.

Modeloaren kodean klasea inferitzeko FC geruzen bloke berri bat gehitu da. Modeloa pisutsuegia izan ez dadin, piezaren altueraren kalkulua posizioaren bloke berean egitea erabaki da. FC bloke guztien sarrerak eta irteerak ere egokitu dira, eta entrenamendu prozesuan lagunduko dutelakoan, *BatchNorm1d* geruzak gehitu dira. Erregularizazio teknika moduan *Dropout* geruzak ere txertatu dira. Bloke hauen egitura 6.14 kode zatian erakusten da. Noski, *forward()* funtzioa ere egokitu da arkitektura berria errespetatuz.

```
    self.num_cubes = 1
    self.num_classes = 4
    self.translation_block = torch.nn.Sequential(
        torch.nn.Dropout(0.5),
        torch.nn.Linear(25088*2, 256*2),
        torch.nn.BatchNorm1d(256*2),
        torch.nn.ReLU(inplace=True),
        torch.nn.Dropout(0.5),
        torch.nn.Linear(256*2, 64),
        torch.nn.BatchNorm1d(64),
        torch.nn.ReLU(inplace=True),
        torch.nn.Linear(64, 3*self.num_cubes+self.num_cubes),
    )

    self.class_block = torch.nn.Sequential(
        torch.nn.Dropout(0.5),
        torch.nn.Linear(25088*2, 256),
        torch.nn.BatchNorm1d(256),
        torch.nn.ReLU(inplace=True),
        torch.nn.Dropout(0.5),
        torch.nn.Linear(256, 64),
        torch.nn.BatchNorm1d(64),
        torch.nn.ReLU(inplace=True),
        torch.nn.Linear(64, self.num_classes),
    )

    self.orientation_block = torch.nn.Sequential(
        torch.nn.Linear(25088*2, 256),
        torch.nn.BatchNorm1d(256),
        torch.nn.ReLU(inplace=True),
        torch.nn.Linear(256, 64),
        torch.nn.BatchNorm1d(64),
        torch.nn.ReLU(inplace=True),
        torch.nn.Linear(64, 4*self.num_cubes),
        LinearNormalized(self.num_cubes),
    )

```

6.14 Kode zatia: Ezaugarri guztiak inferitzeko modeloaren FC geruzak.

Azkenik, modeloa entrenatzeko fitxategietan irteera berriak kontutan izateko beharrezko aldaketak egin dira, eta *overfitting*-a saihesteko *train.py* fitxategian definitzen den Adam optimizatzaileari *weigth_decay = 5e-4* parametroa gehitu zaio.

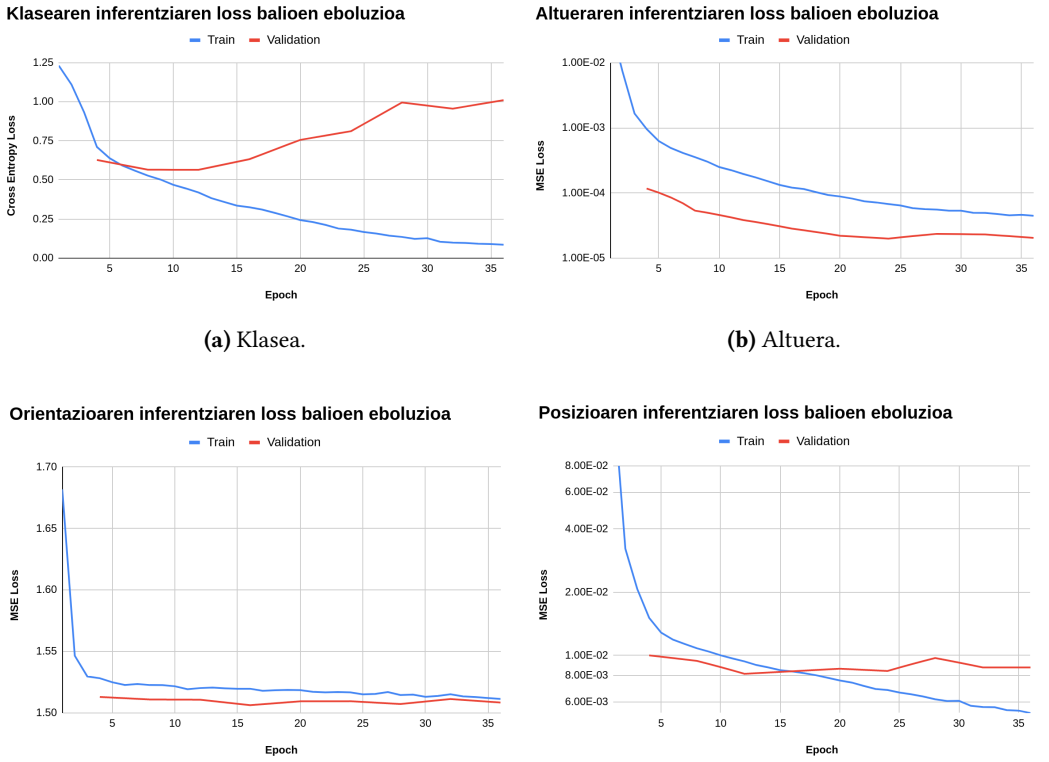
6.8.3 Lehenengo modeloaren entrenamendua

Entrenamendua honako hiperparametroekin egin da:

- Learning rate = 2e-4
- Batch size = 12 (4 GB memoriako GPUak onartzen duen gehiena)
- Epoch kopurua = 36
- Dropout = 0.5

- Accumulation steps = 20
- Weight decay = 5e-4

Prozesuak guztira 46 ordu iraun ditu. 6.21 irudiko grafikoetan *Train* nahiz *Validation* multzoen irteera bakoitzerako lortutako *loss* kurbak bistaratzen dira.



(a) Klasea. **(b)** Altuera.

(c) Orientazioa. **(d)** Posizioa.

6.21 Irudia: Ezaugarri guztiak inferitzeko modeloak entrenamenduan izandako loss kurbak.

Kurbak aztertu ezkero, ikus daiteke kasu batzuetan *overfitting*-a gertatu dela. Nabariena klasearen iragarpenean ematen da: prozesu osoan *Train* multzoan *loss* balioa txikitzen doa, baina *Validation* multzoan 12. *epoch*-etik aurrera nabari handitzen da. Beraz, inferentziarako baliagarriak izan daitezkeen patroi orokorrak ikasi ordez, *Train* multzoko irudi partikularren tratamendua buruz ikasten ari da. Baina hau harritzekoa da, jakinik *Dropout* eta *weight decay* bezalako erregularizazio teknikak baliatu direla. Emaitza hauen beste arrazoi bat modeloaren arkitektura izan daiteke, ataza askotarako VGG16 eskeleto bera erabiltzen baitu. Beste era batera esanda, 4 aldagaien kalkulurako eskeleto bera baliatzen da, eta honen parametroak lauak aldi berean optimizatzeko helburuarekin aldatzen dira.

Kuboaren posizioarekin ere antzeko gauza bat gertatzen da, nahiz eta hain nabaria ez izan: 24. *epoch* arte *Validation* galera minimizatzen doa, baina 28. *epoch*-ean goradaka aipagarria jasaten du, *Train* multzoan hobetzen jarraitzen duen arren.

Entrenamendu prozesuan hobekien ikasi den aldagaia Y ardatzeko eskala, hau da, kuboaren altuera izan da. Izan ere, prozesu osoan bi multzoetako *loss* balioak uniformeki

optimizatu dira. Hala ere, onartu beharra dago Y ardatzeko eskala lortzea ataza sinplea dela, informazio gehiena irudiaren sakonerako kanalak eskaintzen baitu. Baina, harrigarria badirudi ere, zati handi batean *Validation* multzoko galera txikiagoa izan da *Train* multzokoa baino. Halaber, dirudienez, modeloak altuera inferitzen trebatzeko gaitasuna dauka oraindik. *Epoch* gehiago entrenatzen utziko balitz, seguruenik *loss* balio txikiagoetara iritsiko litzateke.

Orientazioari dagokionez, garbi ikusten da modeloa ez dela gai aldagai hau inferitzen ikasteko. Hasiera batean arraroa dirudi, Pose Estimation proiektuko tutorialean emaitza onargarriak eskuratzen baitziren. Hausnarketa baten ostean, ondorioztatu da eszenaren konfigurazioa dela eta kuboen biraketak kalkulatzea oso zaila dela. Izan ere, eszenan kokatu diren kuboak X nahiz Z ardatzekiko simetrikoak dira. Hau da, Y ardatzarekiko 90º, 180º, 270º ala 360º biratuta ere, kanpotik orientazio berdin-berdinarekin ikusten dira. Aldiz, Pose Estimation tutorialean kubo berezi bat baliatzen da, gaineko aurpegian "R" hizkia marraztuta baitu eta, beraz, goitik ikusi ezkero bere orientazioa jakin daiteke.

Arazo horren konponbide bat piezen 3D modeloa edo testura aldatzea izan liteke, simetrikoa ez den ezaugarriren bat izan dezan ("R" hizkia marraztuta, adibidez). Beste soluzio bat Y ardatzeko biraketa egikaritzen duen *Randomizer*-aren balio tartea [0,90) gradutan ezartzea izango litzateke, [0,360] ordez. Horrela, kuboaren egoera konkretu bati biraketa posible bakarra legokioke, 4 aukera izan ordez. Hala ere, mundu fisikoan xurgagailua erabiliko denez eta, ondorioz, orientazioa behar ez denez, orientazioaren inferentzia alde batera uztea erabaki da. Simulazioko probetan piezei ongi heldu ahal izateko, Unity-ko eszenatik objektuaren biraketa erreala irakurriko da.

Laburbilduz, kurben azterketan ikusi da entrenamendu prozesua ez dela oso ona izan. Dena den, soilik *loss* balioak ikusita zaila da esatea modeloa eraginkorra den ala ez, nahiz eta ondorio batzuk atera daitezkeen. Esate baterako, translation aldagaiaren *Validation* galera txikiena 0.008 ingurukoa izan da. Galera hau MSE (*Mean Squared Error*) funtzioaren bitartez kalkulatzen da. Adibide gisa, benetazko balioa 0.1 bada eta 0.2 balioa inferitzen bada, MSE errorea 0.01ekoa da. Jakinik Unity-n defektuz distantziak metrotan neurtzen direla, modeloak 0.1 metro inguruko errorearekin asmatzen du kuboaren posizioa. Hasiera batean, errore nahiko handia dela dirudi. Baina beste aldagai batzuen kasuan, besteak beste, klasearen inferentzian, *loss* balioa konkretu bat izateak ez du informazio gehiegi aportatzen. Hori dela eta, Unity-ko ingurune birtualean proba batzuk egitea erabaki da.

6.8.4 Lehenengo modeloaren eraginkortasuna ebaluatzen

Probak egitean aldagai bakoitzarentzat emaitza onenak eman dituen parametroen instantziarekin geratzea erabaki da. Beste hitz batzuetan esanda, Deep Learning modeloaren FC blokeen parametroak banan-bana kargatuko dira, dagokion aldagaiaren inferentzian galera txikiena lortu den *epoch*-eko instantziarekin. Horrela, posizioari eta altuerari dagokion FC blokea 24. *epoch*-eko parametroekin hasieratuko da, eta klaseari dagokion blokea 12. *epoch*-eko pisu nahiz bias-ekin. Orientazioa alde batera utziko da, eta eskeletoen parametrotzat 12. *epoch*-akoak hartuko dira. Hau dena E5 eranskineko kode zatiari esker egin da. Irteerako modeloaren parametroak *Niryo_1by1_model_mix_ep12-24.tar* fitxategian gorde dira, *ROS/src/niryo_moveit/models/* karpetaren barruan.

Noski, Unity eszenan probak egiteko Scenario *GameObject*-ari ausazkotasun hazi berri bat esleitu zaio: 539662033. Bestela, entrenamendu prozesuan ikusi dituen irudiekin arituko litzateke eta, ondorioz, baldintza errealekin bat ez datozen exekuzioak izango lirateke.

Halaber, ROS aldeko fitxategietan aldaketa batzuk egin dira, batez ere, entrenatutako modelo berria kargatu eta erabiltzeko beharrezkoak direnak.

Azkenik, ROS nodoak martxan jarri dira eta Unity-ko eszena abiatu da. 30 exekuzio-proba egin dira, eta robota ez da gai izan pieza bakar bat hartzeko ere. Errore handiena piezen posizioen kalkuluan nabaritzen da. Izan ere, kuboen dimentsioak 0.04 metrokoak dira gutxi gorabehera, eta erroreen analisiak aztertu den moduan, modeloaren batez besteko errorea 0.1 metro ingurukoa da, ardatz guztian kontuan hartuta. Hortaz, matxarda hartu nahi duen kuboarengandik nahiko urrun kokatzen du. Pieza askoko inguruneetan zaila egiten da ikustea robotak kubo egokia (altuena) hartzeko intentzioa daukan ala ez. Ondorioz, piezen sailkapena ongi egiten duen edo ez esatea zaila suertatzen da. Pieza gutxiko inguruneetan, aldiz, kasu askotan kubo altuena ongi sailkatzen du, nahiz eta hartzeko gai ez izan (dagokion koloreko zakarrontzirako bidea egiten du). Sailkapenean %70-80 inguruko asmatze-tasa duela ikusi da. Piezen altuera ongi inferitzen ikasi duela ere antzeman da.

Dena den, objektuen posizioaren inferentzian errore-balio onartezinak dituenez, modeloaren arkitekturan aldaketa batzuk egitea eta entrenamendu prozesua errepikatzea erabaki da.

6.8.5 Modelo berriaren arkitektura

Aurreko entrenamenduko kurbak aztertzean, hipotesi moduan hurrengoa planteatu da: modeloaren eraginkortasun mugatua VGG16 eskeletoen parametroak zeregin asko aldi berean gauzatzeko eguneratzearen ondorio da. Arkitektura berria diseinatzean hipotesi hau kontuan hartu da.

Hortaz, inferitu beharreko aldagaiak 2 modelo ezberdinetan banatzea erabaki da. Lehenengo modeloa kubo altuenaren posizioa eta altuera kalkulatzeaz arduratuko da. Bigarrena, aldiz, pieza altuenaren klasea inferitzeko baliatuko da. Aurretik aipatu den bezala, kuboaren orientazioaren kalkulua alde batera utziko da. Banaketa horri esker, modelo bakoitzaren eskeletoak ataza konkretuak aurrera eramateko espezializatuko dira. Halaber, entrenamendu prozesua kontrolatzea errazagoa bilakatuko da.

Bi modeloek aurreko ataleko antzeko arkitektura izango dute. Zehazki, aurretik entrenatutako bi VGG16 eskeleto izango dituzte, bata RGB kanalak prozesatzeko eta bestea irudiaren sakonerako informaziorako. Lehenengo modeloak 2 FC bloke izango ditu; bata posizioa kalkulatzeko eta bestea Y ardatzeko eskalarako. Posizioaren blokeak sarrera moduan RGB-D kanal guztien prozesatutako irteera jasoko du, baina altuerarenak soilik sakonerako informazio prozesatua hartuko du sarrera moduan. Inplementazioa 6.15 kode zatian ikus daiteke. Bigarrenak kuboaren sailkapenerako balioko duen FC bloke bakarra edukiko du, 6.16 kode zatian erakusten dena.

6.8.6 Modelo berrien entrenamendua

6.8.6.1 Posizioa eta altueraren modeloa

Posizioa eta altuera inferitzeko modeloaren entrenamendua honako hiperparametroekin egin da:

- Learning rate = 2e-4

```
    self.translation_block = torch.nn.Sequential(
        torch.nn.Linear(25088*2, 256*2),
        torch.nn.BatchNorm1d(256*2),
        torch.nn.ReLU(inplace=True),
        torch.nn.Linear(256*2, 64*2),
        torch.nn.BatchNorm1d(64*2),
        torch.nn.ReLU(inplace=True),
        torch.nn.Linear(64*2, 3*self.num_cubes),
    )

    self.scaleY_block = torch.nn.Sequential(
        torch.nn.Linear(25088, 128),
        torch.nn.BatchNorm1d(128),
        torch.nn.ReLU(inplace=True),
        torch.nn.Linear(128, 32),
        torch.nn.BatchNorm1d(32),
        torch.nn.ReLU(inplace=True),
        torch.nn.Linear(32, self.num_cubes),
    )

```

6.15 Kode zatia: Objektuen posizioa eta altuera inferitzeko modeloaren inplementazioa.

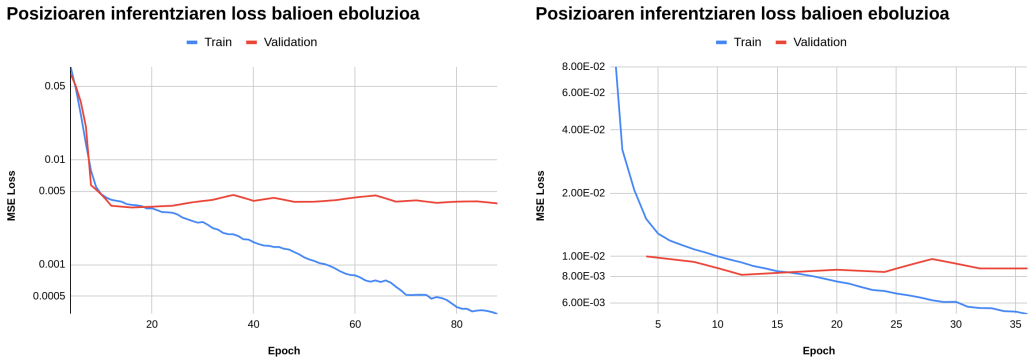
```
    self.class_block = torch.nn.Sequential(
    torch.nn.Linear(25088*2, 256*2),
    torch.nn.BatchNorm1d(256*2),
    torch.nn.ReLU(inplace=True),
    torch.nn.Linear(256*2, 64),
    torch.nn.BatchNorm1d(64),
    torch.nn.ReLU(inplace=True),
    torch.nn.Linear(64, self.num_classes),
    )

```

6.16 Kode zatia: Objektuen klasea inferitzeko modeloaren inplementazioa.

- Batch size = 14
- Epoch kopurua = 88
- Accumulation steps = 10
- Weight decay = 5e-4

Prozesuak guztira 80 ordu iraun ditu. 6.22 irudian entrenamenduan *Train* eta *Validation* multzoetan posizioaren eta altueraren inferentzian izandako *loss* funtzioaren eboluzioa bistaratzen da.

Translazioaren kasuan, dirudienez modeloari ez zaio asko kostatzen balio txikietara heltzea. Oso azkar ikasteko gai da, 10.*epoch*-ean jada 0.004 inguruko MSE balioak erakusten baititu. Hala ere, hortik aurrera *overfitting*-a nagusitzen da, *Validation* multzoko *loss*-a berdintsu mantentzen baita *Train*-ekoa uniformeki hobetzen doan bitartean. Jada ezagutzen diren erregularizazio teknika guztiekin probatu denez (*dropout*, *weight-decay* eta *batch*



(a) Posizioa. **(b)** Altuera.

6.22 Irudia: Kubo altuenaren posizioa eta altuera inferitzeko modeloak entrenamenduan izandako loss kurbak.

normalization), suposatu da modeloak posizioa inferitzeko daukan gaitasun maximoa erakutsi duela (*Train* multzoa buruz ikasten hasi aurretik). Hau da, hiperparametroen bilaketa sakonagoa eginez gero, errorea asko jaitsiko ez dela suposatu da. Beraz, posizioari dagokion entrenamendu prozesua ontzat hartu da. Dena den, denbora gehiago izanez gero, egindako hipotesia ziurtatzea komenigarria litzateke, hiperparametroen beste konbinazio batzuk probatuz.

Altueraren entrenamendua, aldiz, oso eraginkorra izan da. Ez da *overfitting* kasu aipagarririk nabaritzen. Hala ere, ez da harritzekoa, sakonerako informazioa emanik kubo altuenaren altuera kalkulatzea prozesu erraza baita. Posizioaren kurben antzera, 10.*epoch*-ean jada balio oso txikietara iristen dira, 3e-6 ingurura, zehazki. Hortik aurrera *loss*-ak jaisten jarraitzen du, baina era motelago batean. *Validation* multzoko minimoa 2.1e-6 da, eta 44.*epoch*-ean kokatzen da. Geroztik *Validation* multzoari dagokion kurba gorakorra bihurtzen da, *Train*-en oraindik hobetzen ari bada ere.

Eskuratutako MSE erroreak perspektiban jartzearren, bakoitzak erakutsi duen *loss* minimoa distantzia unitateetara bihurtuko da. Hala, altueraren inferentzian MSE errore minimoa 2.1e-6 izan da, eta honek 1.5mm inguruko errorea suposatzen du. Jakinik hartu beharreko kuboak 4cm inguruko tamaina dutela, eskuratutako errorea hutsala da. Posizioaren inferentzian, ordea, 0.004eko MSEak 6cm inguruko batazbesteko errorea suposatzen du errealitatean. Nahiz eta errore nabaria izan, eskuratutako parametro horiekin aurrera jarraitzea erabaki da.

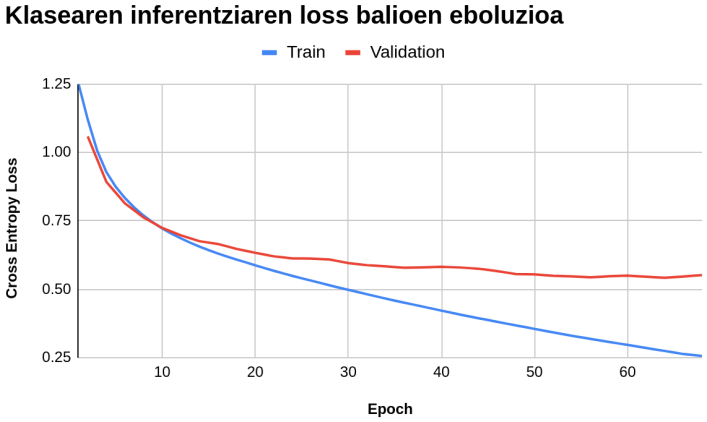
6.8.6.2 Klasearen modeloa

Klasea inferitzeko modeloaren entrenamendua honako hiperparametroekin gauzatu da:

- Learning rate = 1e-6
- Batch size = 16
- Epoch kopurua = 68
- Accumulation steps = 10

- Weight decay = 1e-4

Prozesuak guztira 50 ordu iraun ditu. 6.23 irudian bistaratzen da entrenamenduan *Train* eta *Validation* multzoetan izandako *loss* funtzioaren eboluzioa.



6.23 Irudia: Kubo altuenaren klasea inferitzeko modeloak entrenamenduan izandako loss kurbak.

6.23 irudian ikusten den moduan, *loss* balioen eboluzioa leunagoa izan da. Nahiz eta aurrekoa bezain nabaria ez izan, *overfitting* pixka bat gertatu dela begies daiteke ere. Izan ere, 30. *epoch*-etik aurrera *Train* multzoko *loss* balioak modu aipagarrian txikitzen jarraitzen du, baina modeloa ikasten ari den gehiena ez da baliagarria *Validation* multzorako. Ondorioz, *Validation*-i dagokion *loss*-a era motelago batean optimizatzen du. Minimoa 64.*epoch*-ean topatzen du, *Validation* multzoko galera funtzioaren balioa 0.54 ingurukoa delarik. Une horretan entrenamendu prozesua desegonkortu egiten dela dirudi, kurbak gora egiten baitu.

Kontuan izanik normala dela *Train* multzoaren gainean hobeto funtzionatzea, entrenamendu prozesua ontzat ematea erabaki da. Seguruenik erabilitako hiperparametroak ez dira egokienak, baina hiperparametroen bilaketa sakon bat gauzatzeak denbora asko eskatuko luke. Hori dela eta, etorkizunerako lan moduan utzi da.

6.8.7 Modelo berrien eraginkortasuna ebaluatzen

Deep Learning modelo berrien eraginkortasuna ingurune birtualean ebaluatu nahi izan da. Exekuzioak egiteko, entrenamenduan *Validation* multzoan *loss* balio minimoa erakutsi duen parametroen instantziekin hasieratu dira. Zehazki, klasearen modeloa 64. *epoch*-eko parametroekin hasieratu da, altueraren burua 44.*epoch*-ekoekin eta posizioarena 16.*epoch*-ekoekin.

Guztira 8 proba egin dira. Proba hauek egitean argiaren ausazkotasuna kendu da, baina eszenako beste ausazko propietateak mantendu dira. Exekuzioak bideo batean bildu dira[9].

Egindako probetan ikusi da kubo altuenaren bila joateko irizpidea barneratu duela, eta hau pieza guztien artean identifikatzeko gai da. Halaber, kuboak kolorearen arabera sailkatzea ongi ematen zaio. 8 exekuziotan 7 sailkapen zuzen egin ditu, hau da, %88ko asmatze-tasa erakutsi du.

[9]Kuboekin entrenatutako modeloarekin simulazioan egindako probak: bideoa

Kubo altuenaren posizioaren inferentzia, aldiz, ez da klasearena bezain egonkorra. Exekuzio guztietan beso robotikoa pieza altuenaren bila joan da, baina soilik 4 kasutan lortu du kuboa hartzea. Izan ere, modeloaren irteerako posizioa ez da guztiz zehatza, eta batzuetan robotaren matxarda kuboaren inguruko toki batera bideratzen da. Dena den, egiten duen errorea onargarria kontsideratu da eta, beraz, kuboekin lan egin ordez errealitateko zaborrarekin aritzeari ekin zaio.

6.9 Zaborrarekin lanean

Egin beharreko atazak sinplifikatzearren, orain arte egitura sinpleko objektuekin aritu da lanean. Ikusi da puntu bateraino robota inguruko kuboak antzemateko, hartzeko, kolorearen bitartez sailkatzeko eta dagokion ontzian uzteko gai dela. Beraz, Zabot proiektuaren irismena berreskuratuz, inplementazioa mundu errealeko hondakinetara orokortzeko ordua da. Atal honetan azalduko diren aldaketa guztiak Zabot-en GitHub biltegiko beste adar batean egikaritu dira, *real_garbage*[10] izenekoa.

6.9.1 Hondakinen 3D modeloak

Lehenik eta behin, hondakinen 3D modeloak eskuratu behar dira, Unity-ko eszena birtualean txertatu ahal izateko. Jakinik robotak zaborra 3 klaseren artean sailkatu behar duela, zehazki; organikoa, papera/kartoia eta plastikoaren artean, sailkapen honek barne hartzen dituen hondakinak bilatu behar dira. Horrez gain, Deep Learning modeloak orokortzeko gaitasuna izan dezan, aniztasun zabaleko zaborra biltzea komeni da. Irizipide hauek kontuan hartuz, berriz ere Sketchfab webgunera jo da 3D modelo libreen bila. 6.1 taulan agertzen dira bildu diren hondakinen modeloak, haien klasea eta egilea.

Hondakina	**Klasea**	**Egilea**
Brokolia	Organikoa	pooang
Oilasko hegal frijitua	Organikoa	Andrei Alexandrescu
Latak	Plastikoa	Sunbox Games
Plastikozko botila	Plastikoa	RoutineStudio
Tarta zatia	Organikoa	SPM3D
Arrautzen kaxa	Papera eta kartoia	Antoine Dresen
Paper zimurtua	Papera eta kartoia	Incg5764
Laranja zukuaren kaxa	Plastikoa	matousekfoto
Kartoiezko kaxa	Papera eta kartoia	3dcube
Sagar muina	Organikoa	Andrei Alexandrescu
Ogia	Organikoa	SmallpolyArt
Plastikozko poltsa	Plastikoa	Animandyation
Esne kaxa	Plastikoa	Chris
Plastikozko edalontziak	Plastikoa	Sousinho
Plastikozko sardexka	Plastikoa	abdillaamy

6.1 Taula: Sketchfab webgunean bildutako hondakinen 3D modeloak.

[10] Zabot GitHub biltegiko *real_garbage* adarra: URL

6.9.2 Ingurune birtualaren egokitzapena

Behin 3D modeloak eskuratuta, Unity-ko simulazioan txertatu dira, "GarbageClassifier" izeneko eszena berri batean. Eszena populatua 6.24 irudian ikus daiteke. Horrez gain, gehitutako hondakin bakoitzaren *GameObject*-ari kameraren anotaziorako beharrezko ezaugarriak gehitu zaizkie. *TrajectoryPlanner.cs* programa ere egokitu da, 6 kuborekin lan egin beharrean txertatutako zaborrarekin ibil dezan.

6.24 Irudia: Simulazioko eszena errealitateko zaborrarekin populatuta.

Azkenik, eszenan objektu gehiago daudenez, *SurfaceObjectPlanner.cs* programan piezak kokatzeko probabilitatea %10era jaitsi da, %50 izan ordez.

6.9.3 Datu-bilketa

Datuak biltzeko 6.8.1 ataleko teknika bera jarraitu da. Hau da, Unity-ko eszena URP erabiltzen duen beste proiektu batera eraman da, bertan kameraren anotazioak jasotzeko. RGB-D irudiak, aldiz, jatorrizko proiektuan jaso. Berriz ere, 50000 irudiko *Train* datu-basea bildu da, eta 5000 anotatutako irudiko *Validation* multzoa.

6.9.4 Modeloen entrenamendua

Definitu diren modeloak 6.8.5 atalean deskribatzen den arkitektura bera jarraitzen dute. Hortaz, 2 modelo entrenatu behar izan dira, bat hondakin altuenaren posizioa eta altuera kalkulatzeko, eta bestea haren klasea inferitzeko.

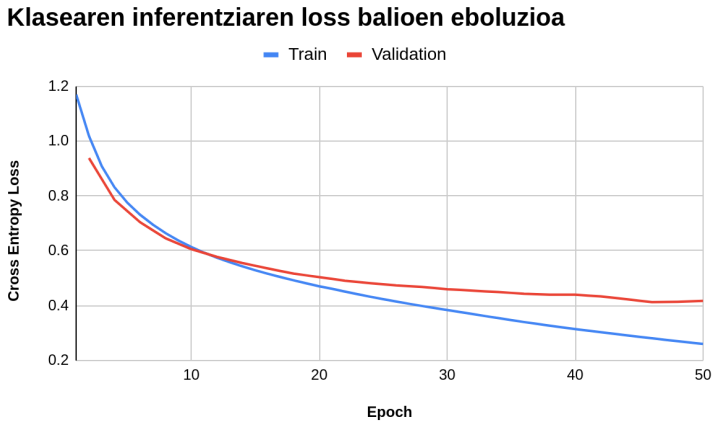
6.9.4.1 Klasearen modeloa

Entrenamendu proba batzuk egin ostean, ikusi da ez dela komeni 6.8.6 atalean baliatutako hiperparametro berberak erabiltzea. Izan ere, kuboak kolorearen bitartez sailkatzea ataza askoz errazagoa da hondakinak sailkatzea baino. Hori dela eta, *learning-rate* txikiagoa zehaztu da, besteak beste. Hauek izan dira behin betiko entrenamenduko hiperparametroak:

- Learning rate = 1e-6
- Batch size = 16
- Epoch kopurua = 50

- Accumulation steps = 10
- Weight decay = 5e-3

Entrenamendu prozesuak 35 ordu iraun ditu. *Train* eta *Validation* multzoetan izandako *loss* balioen eboluzioa 6.25 irudian bistaratzen da.



6.25 Irudia: Hondakin altuenaren klasea inferitzeko modeloak entrenamenduan izandako loss kurbak.

Kurbak aztertu ezkero, ikus daiteke klasea inferitzeko modeloaren entrenamendua eraginkorra izan dela. *Validation* multzoari dagokion *loss*-a amaiera arte optimizatzen jarraitu du, *Train* multzoarengandik diferentzia gehiegirik izan gabe. Hortaz, ez da *overfitting* kasu nabaririk egon. Halaber, dirudienez, ikasteko gaitasun maximora iritsi da, *Validation*-en kurbak 50.*epoch*-ean gorakada txiki bat jasan baitu.

Gainera, harritzekoa badirudi ere, kuboen sailkatzailea baino *loss* balio txikiagoa lortu du, minimoa 0.41ekoa izanik. Kuboen sailkatzailearen kasuan, minimoa 0.54 izan da. Hortaz, hondakinak hobeto sailkatzeko gai da kuboak baino. Hasiera batean harrigarria dirudi, objektu sinpleak kolorearen arabera sailkatzea ataza errazagoa baita benetazko zaborrarekin egitea baino. Baina, hausnarketa baten ostean, gertaera hau justifikatu dezakeen hipotesi bat garatu da.

Zehazki, modeloaren konplexutasuna kontuan hartuta, uste da hondakinak sailkatzeko gaitasun nahikoa daukala. Izan ere, hauetako bakoitzak forma, tamaina eta kolore ezberdina dauzka. Hortaz, hondakinak sailkatzeko ezaugarri asko errepara ditzake modeloak. Aldiz, kuboak sailkatzeko modu bakarra koloreari erreparatzea da. Hori gutxi balitz, eszenan argiaren intentsitatea, orientazioa eta kolorea ausazkotzen dituen *Randomizer*-a gehitu da eta, beraz, piezen koloreak identifikatzea zailtzen da. Esate baterako, eszenan kubo hori bat badago baina argiaren kolorea urdina bada, berdearen antza gehiago izan dezake.

Laburbilduz, hondakinak sailkatzeko modeloaren entrenamenduak arrakasta izan du.

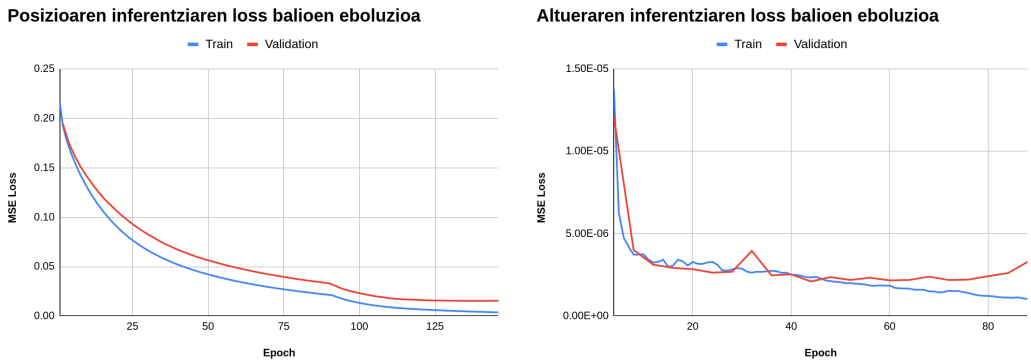
6.9.4.2 Posizioa eta altueraren modeloa

Translazioari dagokion modeloa entrenatzean ere hiperparametro ezberdinak erabili dira:

- Learning rate = 4e-7

- Batch size = 14
- Epoch kopurua = 146
- Accumulation steps = 10
- Weight decay = 5e-3

Aldaketa aipagarriena *learning-rate* hiperparametroan gertatu da, kuboekin entrenatzean 2e-4 balioko



(a) Posizioa. **(b)** Altuera.

6.26 Irudia: Hondakin altuenaren posizioa eta altuera inferitzeko modeloak entrenamenduan izandako *loss* kurbak.

Kasu honetan, ikus daiteke altueraren eta posizioaren grafikoak oso antzekoak izan direla. Kurbak soilik aztertu ezkero, entrenamendu prozesua oso eraginkorra izan dela ondorioztatu daiteke. Izan ere, prozesu osoan *Validation* nahiz *Train* multzoko *loss* balioak txikitzen doaz, elkarren artean diferentzia handirik izan gabe. Modeloak orokortzeko gaitasuna erakusten du, *Train* multzotik ikasten dituen patroiak *Validation*-erako erabilgarriak baitira. Gainera, modeloa ikasteko gaitasun maximora iritsi edo gertu geratu dela dirudi, azken 20 *epoch*-etan lortzen den hobekuntza oso txikia baita.

Lortu diren kurbak desiragarriak izan diren arren, MSE *loss* balioak txarrak izan dira. Esaterako, posizioaren inferentzian *Validation* multzoko *loss* minimoa 0.015 ingurukoa izan da. Honek 0.12 metro baino gehiagoko batazbesteko errorea suposatzen du (x,y,z) ardatz guztiak kontuan hartu ezkero. Altueraren kasuan, minimoa 0.003 ingurukoa izan da, eta honek 0.05 metroko errorea suposatzen du. Jakinik eszenako hondakinek batazbeste 0.05 metroko dimentsioak dituztela, modeloaren erroreak onartezinak direla ondorioztatu da. Baina, aurretik azaldu den bezala, entrenamendu prozesua zuzena izan da eta lortu diren kurbak itxura ona dute. Hori dela eta, diseinatutako modeloarentzat hondakinen posizioa eta altuera kalkulatzea ataza konplexuegia dela ondorioztatu da eta, beraz, emaitzak hobetzeko arkitektura aldatu beharko litzatekeela. Hala eta guztiz ere, entrenatutako modeloekin proba batzuk egin dira.

6.9.5 Modeloen eraginkortasuna ebaluatzen

Entrenatutako Deep Learning modeloen eraginkortasuna ebaluatzeko 5 exekuzio egin dira simulazioan. Exekuzio hauek mahaian objektu bat bakarrik egonik egin dira, hortaz, aurrera eraman beharreko ataza sinplifikatu da. 6.2 taulan bistaratzen dira egindako probak.

Proba ID	Objektuaren posizioa (x,y,z)	Inferitutako posizioa (x,y,z)	Objektuaren altuera	Inferitutako altuera	Objektuaren klasea	Inferitutako klasea
1	(-0.03,0.64,0.26)	(-0.09, 1.3, 0.27)	0.02	-0.03	2	2
2	(-0.18,0.62,0.16)	(0.14, 0.99, -0.13)	0.01	-0.04	1	3
3	(-0.08,0.62,0.23)	(0.26,1.18,0.25)	0.03	0.17	3	3
4	(0.15,0.65,0.18)	(0.46,1.15,0.36)	0.04	0.05	3	3
5	(-0.31,0.65,0.06)	(0.31, 0.96, 0.09)	0.03	0.006	1	2

6.2 Taula: Zabor errealarekin entrenatutako modeloekin simulazioan egindako esperimentazioaren emaitzak.

6.2 taulan ikusten den moduan, modeloaren eraginkortasuna oso kaskarra da. Klasea sailkatzeko gaitasun minimo bat badauka, baina irteerako posizioak errore magnitude onartezina erakusten dute. Hain zuzen ere, entrenamenduko MSE balioak aztertuz ondorioztatu zitekeen moduan, posizioa inferitzean batazbeste 15cm-ko errorea dauka. Hala ere, objektu bakarreko eszenan errendimendu hobea erakustea espero zen. Dirudienez, hondakinaren altuera kalkulatzea ere kostatzen zaio, batzuetan balio negatiboak itzultzera ere iristen da eta.

Emaitzak ikusita, buelta gehiegirik eman gabe, proiektu honetarako zabor errealarekin lan egitea bertan behera utzi da. Hondakinekin lanean aritzen onak diren modeloak garatzeak eta entrenatzeak denbora asko eskatuko duela ikusi da. Gainera, objektu hauek forma konplexua izan dezaketenez, beso robotikoarekin hauei heltzea ere asko zaildu daiteke. Beraz, etorkizunerako lan moduan planteatuko da, eta proiektu honetan errealitaterako trantsizioa pieza sinpleekin gauzatuko da.

KAPITULUA 7

Niryo Ned mundu fisikoan

7.1 Simulaziotik errealitaterako trantsizioa

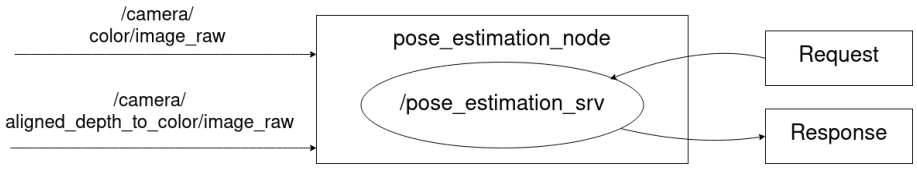
Behin ingurune birtualean beso robotikoa pieza sinpleak antzemateko, sailkatzeko eta *pick-and-place* motako zereginak egikaritzeko gaitasunaz hornituta, mundu fisikoan funtzionatu ahal izateko simulaziotik errealitaterako trantsizioa gauzatu daiteke.

Ezer baino lehen, Niryo Ned robotak baliatzen dituen paketeak GitHub-eko *ned_ros* biltegitik [15] deskargatu dira. Hauek ROS Melodic distribuzioarentzat prestatuta daudenez, Ubuntu 18.04, ROS Melodic eta Python2 dituen ordenagailu batean instalatu dira. Pakete horiei esker, robot fisikoarekin komunikatu eta aginduak bidali liezazkioke. Gazebo eta RViz softwareen bitartez robotaren portaera simulatzeko aukera ere ematen dute, nahiz eta Zabot proiektuan egingo ez den. Ordenagailu berean orain arte garatutako Unity proiektua eta ROS aldeko fitxategiak ere prestatu dira, "zabot" izeneko karpeta baten barruan.

7.1.1 Modeloaren eraginkortasuna errealitatean

Ezustekoak ekiditeko, robotari *pick-and-place* zereginak egitea agindu aurretik, simulazioan kuboekin entrenatutako modeloaren eraginkortasuna mundu fisikoan neurtzea erabaki da. Honetarako, modeloaren exekuzioarekin lotutako programak Python2-rako egokitu dira, *zabot/ROS/niryo_moveit/* direktorioan kokatutakoak, hain zuzen ere. Horrez gain, *pose_estimation_script.py* zerbitzariak Unity aldetik irudi bat jaso ordez, eszena fisikoan instalatutako RealSense kameraren RGB-D argazkiarekin funtzionatuko du. Beraz, zerbitzariaren kodea egokitu da, kamerak publikatzen dituen ROS *topic*-etara harpidetuta egon dadin. Zehazki, RGB irudia */camera/color/image_raw topic*-ean egongo da, eta RGB irudiarekin bateratutako sakonerako informazioa */camera/aligned_depth_to_color/image_raw* postontzian. Harpidetza hauek 7.1 irudiko konputazio-grafoan bistaratzen dira. *Topic*-etatik informazioa jasotzen den bakoitzean, aldagai globaletan eguneratuko dira. Dena dela, kamerak aipatutako postontzietan informazioa publika dezan dagozkion nodoak abiarazi beharko dira, hurrengo komandoaren bitartez:

```
roslaunch realsense2_camera rs_rgbd.launch
```



7.1 Irudia: *pose_estimation_script* zerbitzariaren harpidetzak.

Zerbitzariak eskaeraren bat jasotzen duen momentuan, aldagai globaletako ROS *Image* formatuko irudiak *CV2* paketeko formatura bihurtuko ditu, *CvBridge*[1] paketeari esker. Ondoren, RGB irudiari alfa kanala gehitzen zaio, eta bertan sakonerako informazioa txertatzen da. Azkenik, irteerako RGB-D irudia *zabot/ROS/niryo_moveit/images* direktorioan gordetzen da. Eragiketa hauen kodea 7.1 kode zatian erakusten da.

```
1    bridge = CVBridge()
2    cv_image_rgb=bridge.imgmsg_to_cv2(rgb_image, desired_encoding='brg8')
3    cv_image_d = bridge.imgmsg_to_cv2(rgb_image, desired_encoding='
     passthrough')
4    rgba = cv2.cvtColor(cv_image_rgb, cv2.COLOR_RGB2RGBA)
5
6    # bete alfa kanala sakonerako informazioarekin
7    rgba[:, :, 3] = cv_image_d
8
9    image_name = "Input" + str(count) + ".png"
10   image_path = PACKAGE_LOCATION + "/images/" + image_name
11
12   cv2.imwrite(image_path, rgba)
13
```

7.1 Kode zatia: Kameraren RGB eta D irudiak biltzeko kodea.

Irteerako irudiaren helbidearekin *run_model()* funtzioari deitzen zaio zerbitzarian bertan, modeloaren exekuzioa aurrera eraman dezan.

Modeloaren eraginkortasuna neurtzeko, robota Unity-n definitutako hasierako posiziora eraman da, eta 5 exekuzio egin dira, bakoitzean eszenan kubo bakarra kokatu delarik. Kokatutako kuboak urdinak nahiz horiak izan dira. Bezero programa bat erabili ordez, exekuzioak **rosservice call /pose_estimation_srv** komandoaren bitartez egikaritu dira. Lortutako emaitzak 7.1 taulan bistaratzen dira. Posizioak kameraren erreferentzia-sistemarekiko ematen dira.

7.1 taulan ikusten den moduan, modeloaren eraginkortasuna oso kaskarra da. Egia da kuboak sailkatzeko gaitasuna duela, baina haien posizio nahiz altueraren inferentziak errealitatetik oso urrun daude. Gainera, kontuan izan beharra dago proba sinpleak egin direla, eszenan soilik kubo bat kokatu baita. Emaitzak aztertu ondoren, modeloaren inferentziak robotari agintzeko erabiltzea arriskutsua eta alperrikakoa izango litzatekela ondorioztatu da. Hori dela eta, simulazioan entrenatutako Deep Learning modeloa baztertzea erabaki da. Horren ordez, Ikusmen Artifizialaren eremuko beste teknika batzuk aplikatzea erabaki da.

[1] CvBridge ROS paketearen informazioa: URL

Proba ID	Kuboaren posizioa (x,y,z) kamerarekiko	Inferitutako posizioa (x,y,z) kamerarekiko	Kuboaren altuera	Altuera inferitua	Kuboaren kolorea	Kolore inferitua
1	(0.1,-0.2,0.96)	(-1.23,0.43,0.05)	0.04	0.334	Urdina	Urdina
2	(0,0.1,0.96)	(0.15,0.2,0.23)	0.04	0.218	Urdina	Urdina
3	(0,0,0.96)	(0.17,-0.14,0.54)	0.04	0.161	Urdina	Urdina
4	(0.1,0.2,0.98)	(0.4,0.22,0.18)	0.02	0.128	Horia	Horia
5	(-0.1,0,0.98)	(-0.21,0.34,0.09)	0.02	0.074	Horia	Horia

7.1 Taula: Simulazioan kuboekin entrenatutako modeloarekin errealitatean egindako proben emaitzak.

7.2 Kolore bidezko segmentazioa

Zabot proiekturako prestatu diren objektuak 3 koloretara mugatzen direnez, eszenaren RGB irudi batetik abiatuta mahaiaren gainean dauden kuboak kolorearen arabera segmenta daitezke, haien (u,v) posizioak pixeletan eskuratzeko. Ondoren, kubo bakoitzak irudian duen (u,v) posizioa jakinik, RealSense kamerak sortzen duen *Point Cloud*-a atzitu daiteke kameraren erreferentzia-sistemarekiko puntuaren (x,y,z) koordenatuak eskuratzeko.

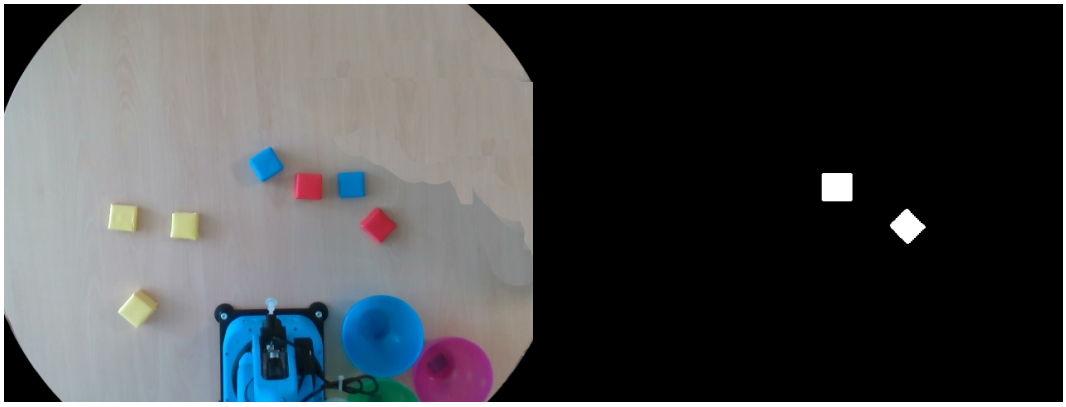
Teknika hau *segment_blob.py* izeneko programan inplementatu da, Python-eko *CV2* paketea baliatuz. Lehenik eta behin, eszenan kubo gorriak, urdinak eta horiak kokatzea erabaki denez, kolore horietako bakoitza banaka segmentatzeko RGB muga minimoak eta maximoak topatu dira. Muga horiek zorrotzak izan behar dira, irudian RGB balio tarte horietan dauden pixelak kuboenak soilik izatea baita helburua. Dena den, ez da komeni RGB tartea txikiegia izatea, inguruneko argiak edo beste faktore batzuk irudian eragin dezakete eta. Definitu diren mugak 7.2 taulan ikus daitezke.

Kolorea	Gorriaren tartea	Berdearen tartea	Urdinaren tartea
Gorria	[130,235]	[20,120]	[10,115]
Urdina	[0,49]	[0,194]	[60,220]
Horia	[0,221]	[175,246]	[70,140]

7.2 Taula: Eszenako kuboen koloreen RGB tarteak.

Horrela, adibidez, kameratik eskuratutako irudiko pixelak kolore gorriarentzat definitutako mugekin segmentatu ezkero, soilik pieza gorriak detektatzen dira. 7.2 irudian ikus daiteke adibide bat, mugak betetzen dituzten pixelak zuriz bistaratzen direlarik eta gainerakoa beltzez.

Ondoren, *segment_blob.py* zerbitzari bezala konfiguratu da, *BlobSegmentService.srv* fitxategian definitutako formatuak errespetatuko dituelarik. Zehazki, sarrera moduan ROS irudi bat jasoko du, eta irteera moduan eszenako kubo handienaren zentroideak irudian dituen (u,v) koordenatuak nahiz bere klasea itzuliko ditu. Piezarik aurkitzen ez badu, klase moduan 0 itzuliko du eta, bestela, 1 kuboa gorria baldin bada, 2 urdina bada eta 3 horia



(a) Segmentatu gabe. **(b)** Segmentatuta.

7.2 Irudia: Eszenako pieza gorrien segmentazioa.

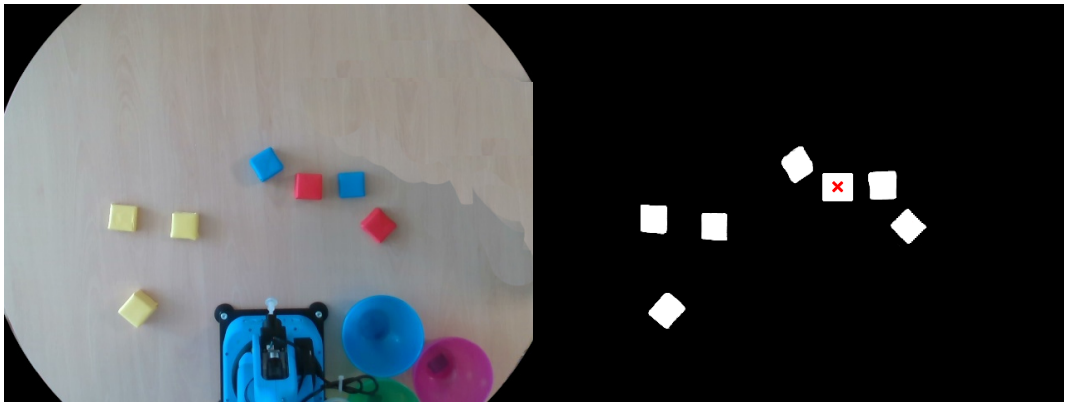
bada.

Kuboak segmentatzeko zerbitzariak eskaera bakoitzeko kubo bakarra prozesatzen duenez, objektu anitzeko eszenak garbitu ahal izateko hainbat eskaera egin beharko dira. Egia da zerbitzaria eskaera batetik mahaiaren gainean dauden kubo guztien zentroideak lortzeko gai izan litekeela, baina hau ez inplementatzea erabaki da. Izan ere, ingurunea dinamikoa izan daiteke, bertan dauden objektuen posizioa aldakorra izan daitekelarik. Horrez gain, robotak *pick-and-place* sekuentzia exekutatzean eszenako beste piezaren bat lekuz aldatu dezake, eta beraz, egindako kalkuluak ez lirateke zuzenak izango.

Eskaera moduan irudi bat jasotzen duenean, *CvBridge* baliatuz *CV2*-ko irudi batera bihurtuko du. Horrez gain, hartu beharreko piezak ez diren objektuak antzeman ez ditzan, esaterako, robota eta ontziak, irudian erreparatu nahi ez diren eremuetako pixelei kolore beltzaren balioak ezartzen zaizkie. Gero, *findCentroid()* funtzioari deitzen zaio. Hau irudia segmentatzeaz eta kubo handienaren zentroidea topatzeaz arduratuko da. Baina aldi berean 3 koloretako piezak antzeman behar dituenez, ez da nahikoa kolore baten parametroak ezartzearekin. Hortaz, maskara osatzeko 3 iterazio egiten dira, eta iterazio bakoitzean *img_segmented* aldagaiarekin **OR** eragiketa aplikatzen da. Ondorioz, segmentazioa pixkanaka osatzen da, iterazio bakoitzean kolore zehatz bateko piezak gehitzen direlarik. Emaitzaren adibide gisa 7.3 irudia dago.

Maskaran objektuak detektatzeko *CV2* paketeak eskaintzen duen *findContours()* metodoa baliatzen da eta, ondoren, aurkitutako pieza guztiak iteratzen dira handienarekin geratzeko. Objekturik detektatu ez badu, kuboaren klasea adierazten duen *label* izeneko aldagaiari 0 balioa emango zaio. Bestela, kuboaren kolorea jakiteko jatorrizko irudia kalkulatutako zentroidearen koordenatuekin atzituko da, eta pixelaren RGB balioak aztertuz kuboaren kolorea jakingo da, 7.2 kode zatian ikusten den moduan.

Hori gutxi balitz, *pose_estimation_script.py* programa ere egokitu da. Lehen bezala zerbitzari bat izaten jarraituko du, baina, aldi berean, irudia segmentatzeko zerbitzuari dei egiteko bezero moduan funtzionatuko du. Gainera, sakonerako informazioari dagokion *topic*-a irakurri ordez, kamerak eskaintzen duen *Point Cloud*-aren *topic*-era harpidetuta egongo da, */camera/depth_registered/points*, hain zuzen ere. *Point Cloud*-a objektuaren zentroidea (u,v) koordenatuetatik kameraren espazioko (x,y,z) puntura bihurtzeko baliatuko



(a) Segmentatu gabe. **(b)** Segmentatuta eta handienaren zentroidea topatuta.

7.3 Irudia: Eszenako pieza guztien segmentazioa.

```
    bgr = self.orig_img[resy,resx]

    if bgr[0]>150 and bgr[2]<40:
        label = 2 #Blue
    elif bgr[1]>130:
        label = 3 #Yellow
    else
        label = 1 #Red

```

7.2 Kode zatia: Zentroideari dagokion kuboaren kolorea eskuratzeko kodea.

da. Bihurketa honen kodea 7.3 kode zatian bistaratzen da.

```
    pcd = convertCloudFromRosToOpen3d(point_cloud)

    points = np.asarray(pcd.points)
    numcols = point_cloud.width
    xyz_cam = points[numcols*resy+resx]

```

7.3 Kode zatia: Zentroidea (u,v) koordenatuetatik (x,y,z)-ra bihurtzeko kodea.

Noski, robotari aginduak eman ahal izateko zentroidearen posizioa kameraren espaziotik munduaren erreferentzia-sistemara igaro behar da. Hau *tf* paketearen bitartez lortu daiteke. Zehazki, ataza honetarako *transform_pose()* funtzioa inplementatu da, 7.4 kode zatian ikus daitekeena.

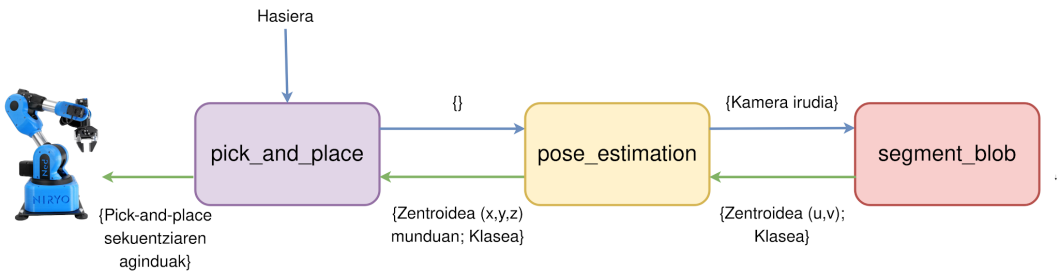
Baina funtzio horrek ongi funtzionatzeko, kamerak erreferentzia-sistema globalean duen posizioa eta orientazioa zehaztu behar dira. Hau *tf* paketea erabiltzen duen beste nodo baten bitartez egin daiteke. Nodoari *camera_transform* izena eman zaio. Dagokion kodea E6 eranskinean bistaratzen da. Halaber, munduarekiko kameraren posizioa eta orientazioa launch fitxategian definitu dira, E7 eranskinean ikusten den moduan. Nodoak bertatik irakurtzen ditu parametro gisa.

```
1    def transform_pose(input_pose, from_frame, to_frame):
2
3    tf_buffer = tf2_ros.Buffer()
4    listener = tf2_ros.TransformListener(tf_buffer)
5
6    pose_stamped = tf2_geometry_msgs.PoseStamped()
7    pose_stamped.pose = input_pose
8    pose_stamped.header.frame_id = from_frame
9    pose_stamped.header.stamp = rospy.Time.now()
10
11   try:
12       output_pose_stamped = tf_buffer.transform(pose_stamped, to_frame,
     rospy.Duration(1))
13       return output_pose_stamped.pose
14
15   except (tf2_ros.LookupException, tf2_ros.ConnectivityException,
     tf2_ros.ExtrapolationException):
16       raise
17
```

7.4 Kode zatia: Poseak erreferentzia-sistemaz aldatzeko funtzioa.

Azkenik, *pose_estimation* nodoari eskaerak egingo dizkion bezeroa programatu da *pick_and_place.py* fitxategian. Erantzun gisa eszenako pieza baten munduko posizioa eta sailkapena jasoko ditu, eta informazio horrekin *pick-and-place* ataza planifikatu eta exekutatzeaz arduratuko da. Robotari aginduak *ned_ros* biltegiko *niryo_robot_python_ros_wrapper* paketeari esker emango dizkio. Sekuentziaren exekuzioari dagokion kodea E8 eranskinean bistaratzen da.

Laburbilduz, *pick_and_place* nodoak *pose_estimation* zerbitzariari eskaera egingo dio. Honek *segment_blob*-en zerbitzuari deituko dio, kameraren irudi bat bidaliz. Erantzun moduan pieza baten zentroidea eta klasea eskuratuko ditu. Azkenik, prozesamendu baten ostean, *pick_and_place* nodoari zentroidearen posizioa munduko koordenatuetan eta piezaren klasea bidaltzen zaizkio, hau robotari aginduak emateaz ardura dadin. Azaldutako komunikazioaren eskema 7.4 irudian ikusten da. Erosotasunagatik, bezero-zerbitzari ereduan funtzionatzen duten nodo guztiak launch fitxategi bakar batean bateratu dira, *pick_and_place.launch* deiturikoa.



7.4 Irudia: Kolore bidezko segmentaziorako nodoak eta haien arteko komunikazioa.

Exekuzio batzuen ostean, ikusi da RealSense kameraren nodoa abiatu eta publikatzen dituen lehenengo 50 irudiak ez direla zuzenak, normala dena baino berdeago ikusten baitira.

Hortaz, nodoa abiarazi ostean, beste edozen egin baino lehen tarte bat itxarotea komeni da.

7.2.1 Esperimentazioa

Inplementatutako kolore bidezko segmentazioarekin mundu fisikoan hainbat proba egin dira. 2 exekuziorekin bideoak ere grabatu dira. Batean kuboak robotaren irismenaren mugatik gertu ipini dira[2], eta bigarrenean ausazko posizioetan kokatu dira[3].

Orokorrean errendimendu ona erakutsi du. Hasteko, kameraren irudia oso azkar segmentatzen da eta, ondorioz, robotak mahaiaren gainean dauden piezak biltzeko denbora gutxi behar du. Gainera, kuboen saikapenean arrakasta erakutsi du, ia denak zakarrontzi egokira eraman baititu, %90 baino gehiagoko asmatze-tasa eskuratuz.

Dena dela, kasu batzuetan, piezen segmentazioa eta zentroidearen (u,v) koordenatuen kalkulua zehatzak izan arren, robotari kuboa hartzea kostatzen zaio. Xurgagailua kuboaren erdigunean ipini ordez, izkin batean kokatzen du edo, bestela, robotak eragingailua piezaren gainean kokatzen du, baina ukitzera iritsi gabe. Ondorioz, kasu horietan ez da piezari heltzeko gai. Baina hau ez da kolore bidezko segmentazioaren arazoa, beste faktore batzuena baizik. Hain zuzen ere, MoveIt planifikatzailearen, kameraren nodoak publikatzen duen *PointCloud*-aren, eta launch fitxategian eskuz zehaztutako kameraren posearen erroreek eragina izan dezakete.

Halaber, ikusi da inguruneko argiztapenak kolore bidezko segmentazioa baldintzatzen duela. Izan ere, kuboak segmentatzeko aurretik eskuz definitutako RGB mugak baliatzen dira. Hauek egoera konkretu batean definituak izan dira eta, beraz, argiztapena aldatzean inplementatutako teknikaren eraginkortasuna kolokan jartzen da.

Hori gutxi balitz, kolore bidezko segmentazioa metodo sinplea da eta orokortzeko gaitasun eza duka. Izan ere, probak 3 koloretako piezekin gauzatu dira, baina beste kolore batzuekin lan egin nahi izanez gero, programa egokitu egin beharra dago. Ez hori bakarrik; teknikak suposatzen du hartu behar diren piezen koloreak bereizgarriak direla inguruneakiko. Hortaz, mahaiaren antzeko kolorea duten piezak sartuko balira, seguruenik ez luke ongi funtzionatuko. Eta kolore ezberdinetako objektu konplexuekin lan egitea bideraezina izango litzateke. Beraz, teknika konplexuago bat inplementatzea erabaki da.

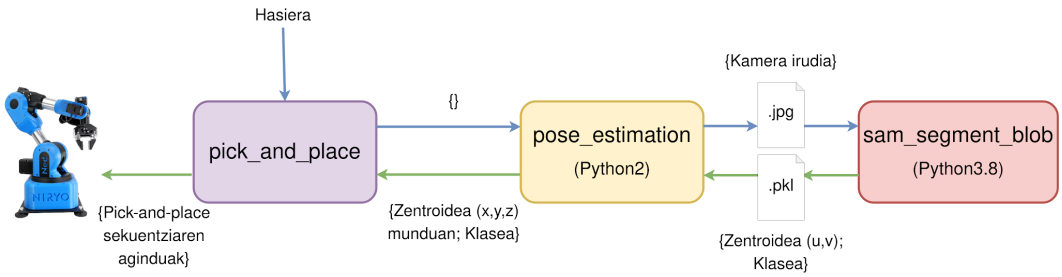
7.3 SAM bidezko segmentazioa

Kolore bidezko segmentazioaren orokortzeko gaitasun eza jakinda, irudia segmentatzeko teknika konplexuago eta eraginkorrago bat baliatzea erabaki da. Zehazki, Meta enpresak garatutako **Segment Anything Model (SAM)** [16] tresna erabiliko da. Modelo hau mila milioi maskara baino gehiagoko 11 milioi irudiren gainean entrenatua izan da, haietan agertzen diren objektu ezberdin guztiak segmentatzeko gaitasuna gara dezan. Kode irekikoa da, eta GitHub-en topa daiteke[4]. Halaber, modelo beraren 3 bertsio existitzen dira, bakoitzak bere parametroak dituelarik. Proiektu honetan defektuzko bertsioa erabiliko da, parametro gehiena dituena baita. *ViT-H* SAM deitzen da, eta ikasitako parametroak GitHub-etik deskargatu daitezke.

[2]Kuboak mugan egonik kolore bidezko segmentazioarekin egindako proba: bideoa
[3]Kuboak ausaz kokatuta kolore bidezko segmentazioarekin egindako proba: bideoa
[4]SAM tresnaren GitHub biltegia: URL

Hala ere, SAMek Python3.8 edo berriagoa den bertsio bat eskatzen du. Hori dela eta, lehendik Python2-n garatu diren ROS zerbitzariak ezin izango dira SAMekin komunikatu ROS bitartez. Beraz, *pose_estimation* nodoaren eta SAM exekutatuko duen programaren arteko elkatrukea fitxategien bidez egitea erabaki da. ROS nodoak RealSense sentsoretik jasotzen duen irudia direktorio konkretu batean idatziko du *CV2* paketeari esker, eta Python3.8 lengoaian idatzitako programak fitxategia existitzen dela ikusten duenean, irudia irakurri eta direktoriotik ezabatu egingo du. Ondoren, *pose_estimation* nodoa erantzuna beste fitxategi baten bidez jasotzeko itxaroten geratuko da. Fitxategiaren sorkuntzaz SAM bidezko segmentazioaren programa arduratuko da, baita bertan irudiaren prozesamendua egin ondoren lortutako informazioa idazteaz ere. Zehazki, irudian segmentatu den azalera handieneko objektuaren zentroidearen (u,v) koordenatuak eta klasea idatziko ditu.

Azaldutako komunikazioaren eskema 7.5 irudian ikus daiteke. Hortaz, kolore bidezko segmentazioko *segment_blob* nodoa kendu egin da, eta horren ordez, manualki SAMi dagokion Python3.8 programa exekutatuko da.



7.5 Irudia: SAM bidezko segmentaziorako nodoak eta haien arteko komunikazioa.

Baina ezer egin aurretik, SAMen eraginkortasuna neurtzeko programa bat garatu da. Programa honek *segment_anything* nahiz *supervision* paketeak inportatzen ditu, prozesatzeko irudi bat irakurtzen du eta SAM *ViT-H* modeloa dagozkion parametroekin kargatzen du. Ondoren, irudia segmentatzeko deiak egiten dira, eta aurkitutako objektu bakoitzeko maskara bat gordetzen da *.pgm* formatuan. Kodea 7.5 irudian bistaratzen da.

Programa hori kameratik ateratako irudi batekin probatu da. SAMek segmentatzeko gaitasun handia daukala ikusi da, baina, kasu batzuetan, piezak eta haien itzalak irudi ezberdinetan gordetzen ditu. Beraz, mahaiaren gainean dauden pieza kopurua adina baino objektu gehiago detektatzen ditu, eta emaitza hauek filtratzea ataza zaila da. Hori dela eta, pieza bakoitzaren segmentazioari erreparatu ordez, mahaiaren detekzioa abiapuntutzat hartzea erabaki da. Izan ere, mahaia ongi segmentatzeko gai da, eta dagokion maskarak mahaiaren gainean dauden objektuen posizioak jakiteko informazio nahikoa eskaintzen du. Hainbat probaren ostean, guztietan ikusi da SAMek mahaiaren maskara lehenengo gordetzen duela eta, ondorioz, hau segmentazio guztien artean aurkitzeko ez du inolako prozesamendurik eskatzen.

Baina mahaiaren maskaratik abiatzeak aurreprozesaketa bat egikaritzea eskatzen du. Lehenik eta behin, maskara alderantzizkatu behar da, piezak zuriz ager daitezen. Baina honek maskarak mahaiaren kanpoko eremua barneratzea dakar, baita robota eta ontziak ere. Hau ikusita, emaitzari beste maskara bat aplikatzea erabaki da, interesatzen ez diren eremuak kanpoan uzteko asmoarekin. Zehazki, 2 irudien arteko **AND** eragiketa gauzatzen da. Aplikatzen den maskara eta lortzen den emaitza 7.6 irudian erakusten dira. Ondoren,

```
import supervision as sv
from segment_anything import sam_model_registry,
        SamAutomaticMaskGenerator

image_bgr = cv2.imread(file_path)
image_rgb = cv2.cvtColor(image_bgr, cv2.COLOR_BGR2RGB)

DEVICE=torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
MODEL_TYPE = "vit_h"

sam=sam_model_registry[MODEL_TYPE](checkpoint="sam_vit_h_4b8939.pth")
sam.to(device=DEVICE)

mask_generator = SamAutomaticMaskGenerator(sam)
result = mask_generator.generate(image_rgb)

detections = sv.Detections.from_scam(result)
for i in range(len(detections)):
    mask = detections.mask[i]
img = mask.astype(np.uint8)
img *= 255
cv2.imwrite(str(i)+".pgm",img)

```

7.5 Kode zatia: SAMen eraginkortasuna probatzeko programa.

zarata moduan gelditu litezkeen partikula txikiak desagerrarazteko 5 pixeleko tamainako *erode* eta *dilate* eragiketa morfologikoak aplikatzen dira. Azaldutako prozesua 7.6 irudian modu bisualean erakusten da.

Behin eszenan dauden piezak ongi filtratuta, azalera handienekoaren zentroidea nahiz klasea topatzeko *segment_blob.py* programako funtzioak berrerabili dira. Programa berriari *sam_segment_blob.py* izena eman zaio. Sarrerako irudia *sensor_image.jpg* fitxategitik irakurriko du, eta irteera *centroid.pkl* fitxategian gordeko du, Python-eko *pickle* paketearen bitartez. Irteerakoa Python2-n idatzitako nodo batek irakurriko duenez, fitxategia idazterakoan *pickle*-eko 2. protokoloa baliatuko du.

Azkenik, 7.2.1 atalean aztertu den moduan, kameraren nodoak publikatzen duen *Point Cloud*-ak errorea izan dezake. Honen eragina murriztearren, zentroidearen (u,v) koordenatuak kameraren espazioko (x,y,z)-ra bihurtzeko irakurritako azken **K** *Point Cloud*-ak atzituko dira. Emaitzako (x,y,z) koordenatuak atzipen guztien batazbestekoa izango dira, baina proba gehienetan xurgagailua altueran motz geratzen denez, Z koordenatuaren kasu partikularrean maximoa hartuko da. **K** parametroa handia izateak zehaztasun bermea emango du, baina programaren exekuzioa asko motelduko du ere. Hortaz, **K=3** izatea erabaki da.

7.3.1 Esperimentazioa

Kolore bidezko segmentazioarekin konparatzeko helburuarekin, SAMen bidezko inplementazioa antzeko esperimentazioan ebaluatu da. Hemen ere hainbat proba egin dira, eta horien artetik 2ren bideoak grabatu dira; bata kuboak robotaren irismenaren mugan jarriz[5]

[5]Kuboak mugan egonik SAM bidezko segmentazioarekin egindako proba: bideoa

(a) SAMen mahaiaren segmentazioa. **(b)** Mahaiaren segmentazioa alderantzizkatuta.

(c) Maskara. **(d)** Maskara aplikatu osteko emaitza.

7.6 Irudia: SAMen irteerari egiten zaion aurreprozesaketa eta lortzen den emaitza.

eta bestea piezak ausazko posizioetan ipiniz[6].

Bistakoa izan den lehenengo gauza teknika berriaren moteltasuna izan da. Ikusi da SAM bidezko segmentazioa gutxi gorabehera kolore bidezkoa baino 2 aldiz motelagoa dela, baldintza berdinetan. Halaber, aurreko kasuan bezala, zentroidea ongi topatu arren batzuetan robotak xurgagailua ez du leku egokira eramaten eta, ondorioz, ez da gai pieza hartzeko. Beraz, *Point Cloud* bat baino gehiago erabiltzeak ez du konpondu arazoa. Gainera, SAMekin exekuzioak motelagoak direnez, kuboa lehenengo saiakeran ez hartzea larriagoa suertatzen da. Gainera, kasu batzuetan SAMen segmentazioak zarata sortzen duela aztertu da.

Hortik kanpo, exekuzioen emaitzak kolore bidezko segmentazioaren antzekoak izan dira. Piezak sailkatzeko haien kolorea aztertzen denez, sailkapenak emaitza berdintsuak erakutsi ditu ere. Baina kontuan izan beharra dago egindako probek ez dutela SAMen potentziala guztiz erakusten. Izan ere, SAMek edozein koloretako piezak antzemateko gaitasuna dauka. Hortaz, SAMen bidezko segmentazioa kolore bidezkoa baino sendoagoa da eta ataza orokortzeko gaitasun hobea dauka.

[6]Kuboak ausaz kokatuta SAM bidezko segmentazioarekin egindako proba: bideoa

Atala IV

Ondorioak eta etorkizunerako lana

KAPITULUA 8

Ondorioak

Proiektu honetan inguruko objektuak antzemateko, jasotzeko eta sailkatzeko beharrezkoak diren beso robotikoaren kontrola eta logika inplementatu dira. Errealitateko hondakinak dituen ingurune batean funtzionatzea lortu ez den arren, objektu sinpleekin emaitza onak erakusten ditu.

Honen ildotik, orain dela gutxi Facebook enpresak kaleratutako SAM modeloak objektuak detektatzeko duen eraginkortasuna ziurtatu da. Beraz, etorkizunera begira, robotika esparruan ikusmen artifizialeko aplikazioetan berrikuntzak ekar ditzake.

Zabot proiektuaren garapenari esker, beste hainbat ondorio atera ahal izan dira:

8.1 Deep Learning modeloak

6 kapituluan zehar hainbat DL modelo diseinatu eta entrenatu dira. Beraz, hauen arkitekturan egiten diren aldaketak dituzten ondorioak aztertu dira, baita entrenamendu prozesuko hiperparametroen doiketan duen eragina ere.

8.1.1 Arkitektura

Zabot proiektuan robotaren inguruko zaborraren ezaugarri ezberdinak irudi batetik inferitzea interesatzen da. Hasiera batean, 6.8.2 atalean, ezaugarri guztiak (posizioa, altuera, orientazioa eta klasea) inferitzeko DL modelo bakarra diseinatu da. Geroago ikusi den moduan, horrelako modelo baten entrenamendua egikaritzea ataza oso konplexua da, hiperparametro berdinak erabiliz *loss* guztiak batera optimizatzea zaila baita. Gainera, horrelako modelo baten arkitekturak GPU memoria asko eskatzen du.

Hau ikusita, 6.8.5 atalean modeloa 2 zatitan banatzea erabaki da: bata posizioa nahiz altuera inferitzeko eta bestea sailkapenaz arduratzeko. Banaketa honek ezaugarri bakoitza inferitzeko erabiltzen diren FC blokeetan parametro gehiago jartzea ahalbidetzen du, GPU memoria gutxiago erabiltzen baita. Gainera, entrenamendu prozesua eta hiperparametroak fintzea asko errazten da, *loss* kurba gutxiago hartu behar baitira kontuan. Azkenean modelo banatuen entrenamenduak denbora gehiago eskatu du, baina emaitza askoz hobeak

eskuratu dira. Hortaz, adimen artifizialarekin ebatzi beharreko ataza ezberdinak bananduta mantentzea komeni dela ikusi da.

Horrez gain, *BatchNorm1d* motako geruzak entrenamendua egonkortu eta azkartzen dutela ikusi da. Erregularizazio teknika moduan ere funtzionatzen du, *overfitting*-a ekiditeko. Bestalde, *Dropout*-en kasuan, ikusi da emaitzak okertu egiten dituela. Honen arrazoia, Xiang Li et al. -en lanean [17] aztertzen den moduan, *BatchNorm* eta *Dropout* teknikek askotan elkarrekin ongi funtzionatzen ez dutelako izan daiteke.

8.1.2 Entrenamendu prozesua

Entrenamenduetan ez da hiperparametroen bilaketa sakonik egin, baina nahikoa izan da oinarrizko ondorio batzuk ateratzeko.

Lehenik eta behin, *learning rate* parametroa ongi fintzea oso garrantzitsua dela ondorioztatu da. Balio handiegiak eman ezkero, *loss* kurbek ez dute joera uniforme bat mantentzen, eta gorakadak nahiz beherakadak izaten dituzte. Aldiz, *learning rate* txikiek prozesua egonkortzen dute, baina entrenamendua asko moteldu dezakete. Horrez gain, ikusi da Zabot-en diseinatutako modelo partikularrak parametro horrekiko oso sentikorrak direla, aldaketa txiki batek *loss* kurben portaera guztiz aldatu dezake eta.

Beste hiperparametroen kasuan, ez da eragin zuzenik antzeman. Esaterako, *weight decay* parametroa *overfitting*-a murrizteko erabili da, baina bere balioa aldatuta ere, ez da ezberdintasun handirik aurkitu. Beraz, hiperparametro honen balioa ez da asko ukitu. Eta *batch size*-en kasuan, beti GPUaren memorian onartzen duen handiena zehaztu da, *batch size* handiek entrenamendu prozesua azkartu eta egonkortu egiten baitute.

Baina, orokorrean, begietsi da modeloak 50000 irudiko datu-base baten gainean entrenatzeak denbora asko eskatzen duela. Beraz, hiperparametroen bilaketa sakon bat egitea, gomendagarria izan arren, konputazionalki oso garestia izango litzateke. Hori dela eta, etorkizunerako lan moduan planteatu da.

8.2 Zaborrarekin aritzeko zailtasunak

6.9 atalean, simulazioan hondakinekin aritzeko saiakera bat egin da. Hainbat motatako eta egitura konplexuko zaborraren 3D modeloak bildu dira, eta kuboekin jarraitutako prozesu berdina egikaritu da. 6.9.5 atalean hondakinen irudien gainean entrenatutako DL modeloaren eraginkortasuna ebaluatu da, baldintza sinpleetan. Aztertu den moduan, modeloa ez da gai hondakinen posizioa errore txiki batekin estimatzeko.

Aurretik modelo bera kuboen irudien gainean entrenatu da, eta 10 aldiz errore txikiagoa erakutsi du posizioaren inferentzian. Hortaz, objektu sinpleetatik konplexuetara igarotzeak posizioaren estimazioa modu nabarian zailtzen duela ikusi da. Hori dela eta, Zabot proiektua hondakinetara hedatzeko erabili direnak baino DL modelo konplexuagoak behar direla ondorioztatu da. Era berean, hauen entrenamendua gauzatzeak baliabide gehiago eskatuko ditu.

Hori gutxi balitz, zaborra errealitatean maneiatzeak proiektua beste konplexutasun maila batera eraman dezake. Izan ere, hondakin batzuk ezin izango dira xurgagailuaren bitartez hartu. Horren ordez, matxarda erabiltzeko beharra sortzen da, eta honen manipulazioa xurgagailuarena baino askoz konplexuagoa suertatzen da [12].

8.3 Errealitaterako trantsizioaren zailtasunak

7.1.1 atalean ikusi da simulazioan entrenatutako modeloa errealitatera eramateak porrota izan duela. Jakinik simulazioko eszena ingurune fisikoaren antzekoa dela, zaila egiten da asmatzea zein den porrotaren arrazoia.

Gainera, ingurune birtuala ausazkotzeko *Randomizer*-ak inplementatu dira. Hauei esker, eszenako argiztapena eta objektuen posizioa, tamaina nahiz errotazioa ausazkotu dira eta, beraz, bildutako irudiak eta anotazioak mota askotarikoak izan dira. Hau kontuan izanda, teorian, ingurune birtualean *pick-and-place* ataza egikaritzea zailagoa izan beharko litzateke errealitatean baino. Helburua entrenatutako modeloak orokortzeko gaitasuna garatzea izan da, errealitaterako trantsizioa errazagoa izan dadin. Hala eta guztiz ere, modeloak emaitza oso kaskarrak erakutsi ditu errealitatean.

Laburbilduz, simulaziotik errealitaterako trantsizioa modu eraginkorrean egitea zeregin oso zaila dela ondorioztatu da. Izan ere, eszena birtualak ingurune fisikoa zehatz-mehatz errepresentatu behar du, eta hau lortzeko zehaztasun maila ikaragarria behar da. Esaterako, kameraren posizioa edo biraketa pixka bat aldatzen bada, neurrigabeko ondorioak izan ditzake.

8.4 Argiztapenaren eragina

Probak robot fisikoaren gainean egin direnean, programa berdinarekin egun batetik bestera portaera guztiz ezberdinak ikusi dira. Objektuen detekzioa kolorearen bitartez egiten denean argiztapenaren eragina oraindik gehiago nabarmendu da. Izan ere, kuboak segmentatzeko RGB tarte batzuk aurredefinitzen dira, baina argiaren egoera aldatu ezkero, definitutako tarteak baliogabetu egiten dira. Hortaz, tarteak handiak izatea komeni da, baina, aldi berean, honek zehaztasuna kentzen dio metodoari. SAMen inplementazioaren kasuan ere, begietsi da objektuek mahaiaren gainean sortzen dituzten itzalak segmentatzen dituela. Ondorioz, honek posizioaren estimazioari errore bat gehitzen dio.

Beraz, argiaren egoerak ikusmen artifizialeko metodoetan eragin handia duela ondorioztatu da. Eta argiztapena egun batetik bestera aldatzen den faktore bat denez, kontrolpean mantentzeko teknikak inplementatzea oso garrantzitsua da.

KAPITULUA 9

Etorkizunerako lana

Atal honetan Zabot proiektuari egin liezazkioken hobekuntzak eta aldaketak proposatzen dira.

9.1 Hobekuntzak

Zabot proiektuaren eraginkortasuna nahiz egonkortasuna hobetzeko edota funtzionalitate gehiago eskaintzeko garatu daitezkeen alderdiak azaltzen dira hemen.

9.1.1 Zakarrontzi eta hondakin mota gehiago txertatu

Zabot-en 3 motako hondakinak soilik landu dira: organikoak, plastikoa, eta papera nahiz kartoia. Baina errealitatean mota gehiago existitzen dira, besteak beste, beira eta errefusa. Beraz, proiektuaren hobekuntza bat zabor mota gehiago tratatzea izango litzateke. Honek eszenan hondakin berriak gehitzea, zakarrontzi gehiago txertatzea Deep Learning modeloa berriz entrenatzea eskatuko luke.

Halaber, probetxuzkoa izango litzateke Zabot-en jada tratatzen diren hondakin moten artean adibide gehiago txertatzea. Hau da, zabor gehiagoren 3D modeloak eskuratzea eta modeloa hauek ezagutzeko entrenatzea.

9.1.2 Hondakinak ez diren objektuak txertatu

Proiektu honetan garatu den sistemak robotaren inguruko objektu guztiak hondakinak direla suposatzen du. Baina ingurune batzuetan hau ez da derrigorrez horrela izan behar; baliteke zabor gisa kontsideratu behar ez diren objektuak egotea.Hori dela eta, Zabot proiektuaren hobekuntza posible bat Deep Learning modeloak hondakinen eta ez-hondakinen artean banatzen ikastea izango litzateke.

Baina, aurretik, objektu bat zaborra den ala ez kontsideratzeko irizpideak aztertu eta eztabaidatu beharko lirateke. Ondoren, irizpide hauen arabera, eszenan hondakinak ez diren objektuak txertatu beharko lirateke. Azkenik, mota orotariko objektuak dituzten inguruneen tratamendua pentsatu beharko litzateke. Izan ere, Zabot proiektuan uneoro

eszenako objektu altuenari erreparatzen zaio, baina hau hondakina kontsideratuko ez balitz, bestelako tratamendu bat jaso beharko luke.

9.1.3 Posizio aldakorreko zakarrontziak

Hondakinak ez bezala, Zabot-eko eszenan zakarrontzian posizio finko batean mantendu dira. Honek *pick-and-place* zeregina errazten du. Seguruenik, robotak mundu errealean izan litzakeen aplikazio gehienetan premisa bera betetzen da.

Hala ere, sistema posizio aldakorreko zakarrontziak dituzten inguruneetan modu egokian lan egiteko gai balitz, sistema sendoagoa eta egonkorragoa izango litzateke. Ondorioz, etorkizunerako hobekuntza moduan proposatzen da.

9.1.4 Deep Learning modelo gehiago aztertu

Ikusi den moduan, Zabot irismen zabaleko proiektu bat da. Ondorioz, esparru eta eremu askotan murgiltzea eskatu du, baina sakontasun nabaririk gabe. Esparru horietako bat Machine Learning izan da.

Zabot-en kasuan, Deep Learning modelo baten arkitektura diseinatu da eta entrenamendua gauzatu da. Hala ere, arkitektura gutxi aztertu dira eta ez zaio nahi bezain besteko denbora eskaini, entrenamendu prozesuak baliabide asko eskatzen baititu. Hau jakinik, Deep Learning modelo berriak diseinatzea eta aztertzea probetxuzkoa izango dela uste da, Zabot-en lortutako emaitzak hobetzearren. Gainera, entrenamendu prozesua zehaztasun maila handiagoarekin garatzea gomendatzen da, baita hiperparametroen bilaketa sakonago bat gauzatzea ere.

9.1.5 Robotaren hasieraketa kendu

6.7.3 atalean erabaki den moduan, proiektu honetako simulazioan *pick-and-place* zeregin bat gauzatu aurretik, robota hasierako posiziora eraman behar da. Hasierako posizio hau inguruko hondakinen oklusioa saihesteko moduan definitu da. Eta hasieratze honek robotaren operazioak motelagoak izatea dakar.

Proiektuan hausnartu den moduan, ordezko aukera bat *pick-and-place* ataza robota hasieratu gabe egitea da. Baina honek beste arazo batzuk dakartza, esate baterako, piezen oklusioa eta entrenamendu prozesuaren zailtzea. Hala eta guztiz ere, ongi inplementatzea lortu ezkero lortuko lirateken onurek merezi dutela uste da. Bestela, ingurune fisikoan egin den moduan, ontziak izkin batean koka litezke, robotaren eta zakarrontzien artean zaborrik egongo ez dela suposatuz.

9.1.6 Robot fisikoaren ibilbidea zuzendu

7.2.1 eta 7.3.1 ataletako esperimentazioetan ikusi den bezala, segmentazio bidezko metodoak objektuen zentroideak topatzeko ongi funtzionatzen dute, baina, kasu batzuetan, robotak xurgagailua ez du leku egokian kokatzen.

Hobekuntza moduan robot fisikoaren ibilbidea zuzentzea proposatzen da. Hau Niryo Ned robotak dakarren kameraren bitartez lor daiteke. Lehenik, robota segmentazioaren bitartez kalkulatutako posiziora higituko da. Behin helburuko piezaren gainean kokatuta, beso robotikoan instalatutako kameratik ateratako irudi bat azter daiteke. Irudi honen

prozesamenduari esker, robota bere traiektoria zuzentzeko eta xurgagailua objektuaren erdi-erdian kokatzeko gai izango da.

9.1.7 SAM bidezko segmentazioaren sailkapena orokortu

Segmentazioa SAM bidez gauzatzeak edozein forma eta egiturako objektuak antzematea ahalbidetzen du. Hala ere, proiektu honetan piezen sailkapena kolorearen bitartez egin da eta, beraz, ezin izan da SAMen gaitasuna ongi ebaluatu. Beraz, errealitateko hondakinak sailkatzeko metodo bat garatzea proposatzen da, eta SAM bidezko segmentazioarekin bateratzea.

Gainera, egitura konplexuko objektuekin lan egiteak beste ondorio batzuk ekar ditzake. Esaterako, xurgagailua hondakin batzuk hartzeko gai ez izatea gerta daiteke, edo objektuaren zentroidearen kalkulua findu behar izatea. Hau kontuan izanik, etorkizuerako lan moduan planteatzen da.

9.2 Aldaketak

Hemen garapenean zehar hartutako erabaki edo bide konkretuen aldakuntzak proposatzen dira, sistemari beste irismen bat ematearren.

9.2.1 Simulaziotik errealitaterako trantsizioa berdiseinatu

7.1.1 atalean simulazioan entrenatutako modeloak errealitatean emaitza onartezinak erakutsi ditu. Hortaz, proiektu honetan beste ikusmen artifizialeko teknika batzuk inplementatzea erabaki da. Hala eta guztiz ere, komenigarria da gauzatutako trantsizioaren akatsak topatzea eta zuzentzea, simulazioan datu-bilketa masiboak egin baitaitezke.

Hasiera batean, eszena birtuala mundu fisikoaren antzekoago egitea planteatzen da, DL modeloak simulazioan ikasi duena baliagarria izan dadin. Honek simulazioko objektuen eta tresnen kokapena egiaztatzea eta gehiago fintzea eskatuko luke. Baina beste faktore batzuk aztertzea komeni da ere, hala nola, inguruneren argiztapena eta 2 inguruneetako kamerek ateratzen dituzten argazkien konparaketa.

9.2.2 Matxardarekin lan egin

Proiektu honetan, *pick-and-place* zeregina errazteko helburuarekin, piezak xurgagailuaren bidez manipulatzen dira. Baina Niryo Ned robotak matxarda instalatzeko aukera ere badu, eta beraz, interesgarria izan liteke proiektua honekin lantzea. Izan ere, zailtasun batzuk ekarri ditzakeen arren, forma konplexuagoko edota pisutsuagoak diren zaborrak manipulatzeko gaitasuna eman liezaioke beso robotikoari.

9.2.3 Ingurunea aldatu

Proiektuan egin liteeken beste aldaketa bat ingurunea aldatzea litzateke. Zabot-en, robota eta objektu guztiak mahai zirkular baten gainean kokatzen dira, baina ez du zertan horrela izan behar.

Adibide gisa, hondakinak zinta higikor batean ipintzea proposatzen da. Hortaz, robota linealki mugitzen ari diren piezak hartzeko gaitasunarekin hornitu beharko da. Zintaren

abiadura exekuzio batetik bestera aldakorra baldin bada zeregina zailtzen da, denbora tarte batean irudi bat baino gehiago atera beharko baitira eta horiekin abiadura kalkulatu.

Eranskinak

E1 eranskina: *PublishJoints* metodoan objektu anitzekin aritzeko aldatutako kode zatia

```
1    // Pick Pose
2    request.pick_pose = new PoseMsg
3    {
4        position = (pickObjects[ind].transform.position +
             m_PickOffset).To<FLU>(),
5
6        // The hardcoded x/z angles assure that the gripper is always
             positioned above the target cube before grasping.
7        orientation = Quaternion.Euler(90,
             pickObjects[ind].transform.eulerAngles.y, 0).To<FLU>()
8    };
9
10   // Place Pose
11   request.place_pose = new PoseMsg
12   {
13       position = (placeObjects[ind].transform.position +
             m_PlaceOffset).To<FLU>(),
14       orientation = m_PickOrientation.To<FLU>()
15   };
```

E2 eranskina: Datu-bilketako kubo bakarreko eszenako irudien anotazioen formatua.


```
  "filename": "RGB023b6bf7-a655-4661-b4a8-b118dc9c9fdc/rgb_2.png",
    "format": "PNG",
    "annotations": [
      {
        "id": "10196fae-7579-46e9-9df2-a32d92bc3afb",
        "annotation_definition": "0bfbe00d-00fa-4555-88d1-471b58449f5c"
          ,
        "values": [
          {
            "label_id": 1,
            "label_name": "cube_position",
            "instance_id": 1,
            "translation": {
              "x": 0.21462567150592804,
              "y": 0.24230802059173584,
              "z": 0.729233980178833
            },
            "size": {
              "x": 0.027140958234667778,
              "y": 0.023532239720225334,
              "z": 0.031195538118481636
            },
            "rotation": {
              "x": -0.37931567430496216,
              "y": -0.59675765037536621,
              "z": 0.59675765037536621,
              "w": 0.37931567430496216
            },
```

E3 eranskina: Datu-bilketako kubo anitzeko eszenako irudien anotazioaren adibidea.


```
    "label_id": 6,
    "label_name": "yellow_cube2",
    "translation": {
      "x": 0.21462567150592804,
      "y": 0.24230802059173584,
      "z": 0.729233980178833
    },
    "size": {
      "x": 0.027140958234667778,
      "y": 0.023532239720225334,
      "z": 0.031195538118481636
    },
    "rotation": {
      "x": -0.37931567430496216,
      "y": -0.59675765037536621,
      "z": 0.59675765037536621,
      "w": 0.37931567430496216
    }
  }, {
    "label_id": 1,
    "label_name": "green_cube1",
    "translation": {
      "x": 0.22674001753330231,
      "y": 0.18295066058635712,
      "z": 0.7278938889503479
    },
    "size": {
      "x": 0.017394907772541046,
      "y": 0.026212491095066071,
      "z": 0.016277266666293144
    },
    "rotation": {
      "x": -0.070282094180583954,
      "y": 0.70360535383224487,
      "z": -0.70360535383224487,
      "w": 0.070282094180583954
    }
```

E4 eranskina: Datu-basea aurreprozesatzean kubo altuenaren informazioa erauzteko *single_cube_dataset.py* programan gehitutako aginduak.

```
    translation = []
    orientation = []
    scaleY = []
    label = []
    name2label = { 'nothing': 0, 'green': 1, 'blue': 2, 'yellow': 3  }
    max_height_ind = -1
    max_height = 0
    for i in range(len(position_list)):
        obj_height = position_list[i]["size"]["y"]
        if obj_height>max_height:
            max_height = obj_height
            max_height_ind = i
    if max_height_ind ==-1:
        translation = [0]*3
        orientation = [0]*4
        scaleY = [0]*1
        label = [0]
    else:
        translation = list(position_list[max_height_ind]["translation"].
values())
        orientation = list(position_list[max_height_ind]["rotation"].
values())
        scaleY = [position_list[max_height_ind]["size"]["y"]]
        label = [name2label[position_list[max_height_ind]["label_name"].
split("_")[0]]]

```

E5 eranskina: Modeloaren bloke bakoitzean *epoch* ezberdinetako parametroak ezartzeko kodea.

```
    folder = os.path.dirname(__file__) + "/"
    filename1 = "Niryo_1by1_model_ep12.tar"
    filename2 = "Niryo_1by1_model_ep24.tar"
    path1 = folder+ filename1
    path2 = folder+ filename2

    #Ireki 24.epoch-eko modeloa
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu
    ")
    model = PoseEstimationNetwork(is_symetric=False)
    checkpoint = torch.load(path2, map_location=device)
    model.load_state_dict(checkpoint["model"])
    loaded_config = copy.deepcopy(checkpoint["config"])

    #Gorde 24.epoch-eko parametroak
    epoch24_params = copy.deepcopy(model.state_dict())

    #Kargatu 12.epoch-eko parametroak
    checkpoint = torch.load(path1, map_location=device)
    model.load_state_dict(checkpoint["model"])
    loaded_config = copy.deepcopy(checkpoint["config"])
    model_dict = model.state_dict()

    # 24.epoch-eko parametroak filtratu, soilik posizioaren blokekoekin
    geratzeko
    epoch24_params_filtered = {k: v for k, v in epoch24_params.items() if
     ("translation_block" in k)}
    # Ordezkatu parametroak
    model_dict.update(epoch24_params_filtered)

```

E6 eranskina: *camera_transform* nodoaren kodea.

```
    def run(self):
        broadcaster = tf2_ros.StaticTransformBroadcaster()
        static_transformStamped = geometry_msgs.msg.TransformStamped()

        static_transformStamped.header.stamp = rospy.Time.now()
        static_transformStamped.header.frame_id = self.frame_id
        static_transformStamped.child_frame_id = self.child_frame_id

        static_transformStamped.transform.translation.x = self.trans_x
        static_transformStamped.transform.translation.y = self.trans_y
        static_transformStamped.transform.translation.z = self.trans_z

        quat = tf.transformations.quaternion_from_euler(self.rot_roll,
                        self.rot_pitch, self.rot_yaw)
        static_transformStamped.transform.rotation.x = quat[0]
        static_transformStamped.transform.rotation.y = quat[1]
        static_transformStamped.transform.rotation.z = quat[2]
        static_transformStamped.transform.rotation.w = quat[3]

        broadcaster.sendTransform(static_transformStamped)

        rospy.spin()

```

E7 eranskina: Kamera fisikoaren posizioa eta orientazioaren definizioak launch fitxategian.

```
1   <node name="camera_transform" pkg="niryo_pick_and_place" type="
        camera_transform.py" output="screen">
2     <param name="frame_id" type="string" value="world" />
3     <param name="child_frame_id" type="string" value="camera_link" />
4     <param name="trans_x" type="double" value="0.325" />
5     <param name="trans_y" type="double" value="0.035" />
6     <param name="trans_z" type="double" value="1.024" />
7     <param name="rot_roll" type="double" value="0.0" />
8     <param name="rot_pitch" type="double" value="-3.141592" />
9     <param name="rot_yaw" type="double" value="1.57075" />
10  </node>
```

E8 eranskina: Robot fisikoaren *pick-and-place* ataza planifikatzeko aginduak.

```
    # Opening Gripper/Pushing Air
    niryo_robot.release_with_tool()
    #Going to pick pose
    niryo_robot.move_pose(cube_pos.x, cube_pos.y, cube_pos.z+0.05, 0.0,
    1.57, 0)
    niryo_robot.move_pose(cube_pos.x, cube_pos.y, cube_pos.z, 0.0, 1.57,
    0)
    # Picking
    niryo_robot.grasp_with_tool()
    #Go up again
    niryo_robot.move_pose(cube_pos.x, cube_pos.y, cube_pos.z+0.13, 0.0,
    1.57, 0)
    # Moving to place pose
    niryo_robot.move_pose(can_positions[label-1][0],can_positions[label
    -1][1],can_positions[label-1][2], 0.0, 1.57, 0)
    # Release
    niryo_robot.release_with_tool()

```

Bibliografia

[1] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

[2] Benjamin Ioller. Bottle sorting by CNN and physical features extraction via robotic arm. working paper or preprint, November 2019.

[3] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement, 2018.

[4] R Aarthi. and G Rishma. A vision based approach to localize waste objects and geometric features exaction for robotic manipulation. *Procedia Computer Science*, 218:1342–1352, 2023.

[5] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn, 2018.

[6] Takuya Kiyokawa, Hiroki Katayama, Yuya Tatsuta, Jun Takamatsu, and Tsukasa Ogasawara. Robotic waste sorter with agile manipulation and quickly trainable detector. *IEEE Access*, 9:124616–124631, 2021.

[7] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. September 2014.

[8] Unity pick-and-place tutoriala. https://github.com/Unity-Technologies/Unity-Robotics-Hub/tree/main/tutorials/pick_and_place. Atzitua: 2023-01-30.

[9] Unity object pose estimation tutoriala. https://github.com/Unity-Technologies/Robotics-Object-Pose-Estimation. Atzitua: 2023-03-07.

[10] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. March 2017.

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[12] Ander Iriondo, Elena Lazkano, and Ander Ansuategi. Affordance-based grasping point detection using graph convolutional networks for industrial bin-picking applications. *Sensors*, 21(3), 2021.

[13] Samarth Brahmbhatt. Rgbd errenderizazioa unity-n. https://samarth-robo.github.io/blog/2021/12/28/unity_rgbd_rendering.html. Atzitua: 2023-03-25.

[14] Radhakrishnan Gopalapillai, Deepa Gupta, Mohammed Zakariah, and Yousef Ajami Alotaibi. Convolution-based encoding of depth images for transfer learning in rgb-d scene classification. *Sensors*, 21(23):7950, Nov 2021.

[15] Niryo ned robotak baliatzen dituen ros paketeak. https://github.com/NiryoRobotics/ned_ros. Atzitua: 2023-05-10.

[16] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross Girshick. Segment anything, 2023.

[17] Xiang Li, Shuo Chen, Xiaolin Hu, and Jian Yang. Understanding the disharmony between dropout and batch normalization by variance shift, 2018.

[18] Intel realsense sdk 2.0 softwarea. https://www.intelrealsense.com/sdk-2/. Atzitua: 2023-01-28.

[19] Moveit tresna. https://moveit.ros.org/. Atzitua: 2023-02-03.

[20] Cv bridge paketearen informazioa. http://wiki.ros.org/cv_bridge. Atzitua: 2023-05-10.

[21] Segment anything proiektuaren github biltegia. https://github.com/facebookresearch/segment-anything. Atzitua: 2023-05-20.

[22] Ros komandoak. http://wiki.ros.org/ROS/CommandLineTools. Atzitua: 2023-01-24.

[23] Ros kontzeptuak. http://wiki.ros.org/ROS/Concepts. Atzitua: 2023-01-23.

[24] Urdf importer. https://github.com/Unity-Technologies/URDF-Importer. Atzitua: 2023-02-16.

[25] Ros tcp endpoint. https://github.com/Unity-Technologies/ROS-TCP-Endpoint. Atzitua: 2023-02-16.

[26] Ros tcp connector. https://github.com/Unity-Technologies/ROS-TCP-Connector. Atzitua: 2023-02-16.

[27] Unity perception paketea. https://github.com/Unity-Technologies/com.unity.perception. Atzitua: 2023-03-10.

[28] Niryo studio. https://docs.niryo.com/product/ned/v3.1.1/en/source/software/niryo_studio.html. Atzitua: 2023-02-05.

[29] lachlandauth. Zakarrontziaren modeloa free3d-n. https://free3d.com/3d-model/trash-can-840994.html. Atzitua: 2023-02-14.

[30] Ur3 robota. https://www.universal-robots.com/media/240787/ur3_us.pdf. Atzitua: 2023-03-7.