Pázmány Péter Katolikus Egyetem

Információs Technológiai és Bionikai Kar

SZAKDOLGOZAT

Orvosi képalkotó berendezések térbeli mozgását szimuláló tesztrendszer fejlesztése

Dósa Dániel

Molekuláris Bionika mérnöki BSc

2020

Témavezető: Hudoba Péter

Konzulens: Dr. Horváth András

Nyilatkozat

Alulírott Dósa Dániel, a Pázmány Péter Katolikus Egyetem Információs Technológiai és Bionikai Karának hallgatója kijelentem, hogy ezt a szakdolgozatot/diplomamunkát meg nem engedett segítség nélkül, saját magam készítettem, és a szakdolgozatban/diplomamunkában csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva / a dolgozat nyelvétől eltérő nyelvből fordítva, más forrásból átvettem, egyértelműen a forrás megadásával megjelöltem. Ezt a szakdolgozatot/diplomamunkát más képzésen nem nyújtottam be.

Dósa Dániel

Kivonat

A programfejlesztés során kiemelkedő fontosságú, hogy alapos tesztelést végezzünk, ami kitér minden funkcióra és a lehető legjobban átvizsgálja azokat. A mai modern eszközökben a legtöbb alkatrész mozgatását motorok végzik, ezeknek a koordinációját a motorvezérlő egység végzi. A motorvezérlő programozása közben esetlegesen vétett hibákat detektálni kell ahhoz, hogy ki lehessen javítani őket. Az orvosi képalkotó berendezések költségesek, nagyméretűek, és nehezek. A vezérlő egység tesztelését a leghatékonyabban a berendezéstől függetlenül, egy szimulált környezetben lehet megoldani. Így munka közben elkerülhető az esetleges ütközésekből adódó anyagi kár, kisebb eszközigényű a tesztelési folyamat, ezáltal nem követel akkora anyagi befektetést, illetve könnyebben szállítható berendezés is elégséges hozzá.

Ezek alapján, a kutató csoporttal, amihez csatlakoztam, megvizsgálva a szakirodalomban a hasonló szükségletre adott válaszokat, úgy döntöttünk hogy saját szimulációs szoftver fejlesztésébe kezdünk. Lehetőségem nyílt a szoftver fejlesztésének első fázisában részt venni. Ez a dolgozat az én munkámnak a leírását tartalmazza. Bemutatásra kerül benne az alkatrészek mozgásának leírására használt módszertan, az ütközésvizsgálatra használt egyenletrendszer és a megalkotott architektúra.

A program a JSON dokumentumként tárolt alkatrészeket Bounding Box modellekkel reprezentálva tárolja és jeleníti meg háromdimenziós vizualizációban. Kommunikál a virtuális motorokkal, és ezek alapján mozgatja a beállított alkatrészeket, illetve értelmezi ezek egyenes vonalú mozgását és a koordináta tengelyek menti forgását. A teljes futásidő alatt figyeli az esetleges ütközéseket, és jelzi, ha ezek megtörténnek. Ellenőrzi, hogy az alkatrészek a megadott határokon belül mozognak-e.

A motorok virtuális reprezentálásárában a Tango Controlls eszközkészletet használtuk. A teszt szoftver és a motorok között REST API kommunikáció valósul meg. Az ütközés vizsgáló szoftver teszteléséhez szükség volt a motorvezérlő modellezésére is. Ehhez

egy JSON dokumentumban állítható be, hogy a virtuális motorok mennyi ideig, milyen sebességgel mozogjanak.

A program elkészülése után lényegesen felgyorsulhat a vezérlőszoftver tesztelése és fejlesztése. A virtuális motorok bármekkora sebességre képesek, nem vonatkoznak rájuk teljesítménybeli, fogyasztási vagy biztonsági megkötések. Így a teszt akár többszörös sebességgel is futhat, ezzel rengeteg időt is megtakarítva a már fent említett biztonsági és anyagi megfontolásokon túl. Gyors és alapos tesztelés mellett az esetleges hibákra hamarabb derül fény, ezzel a fejlesztési folyamat is felgyorsul. Az orvosi berendezésekre vonatkozóan nagyon szigorúak a szabályozások, a gyorsabb tesztelésnek hála, a gépek hamarab juthatnak hozzá a szükséges tanusítványokhoz.

Köszönetnyilvánítás

Nagyon hálás vagyok a témavezetőmnek, Hudoba Péternek, aki nem csak szakmai területen mutatott irányt nekem, hanem állandó, megértő és segítőkész jelenlétével tette lehetővé a dolgozat elkészülését. Hálával tartozom Dr. Horváth Andrásnak, a belsős konzulensemnek, aki mindvégig nagyon bátorítóan fogalmazta meg a kritikáit. Végül köszönöm a feleségemnek, hogy erőt öntött belém, amikor én kifogytam a motivációból.

Tartalomjegyzék

1. fejezet

Bevezetés

1.1. A szakdolgozat felépítése

2. Fejezet Az Előzmények című fejezet arról szól, miért merült fel az igény az ütközésvizsgáló szoftver megalkotására. Ezen felül itt található az alkalmazott matematika módszerek leírása.

3. Fejezet A Tervezés című fejezetben találhatóak a használt eszközök, bemutatásra kerül a program architektúrája és a megavalósítás részletei.

4. Fejezet Az Eredmények című fejezetben látható az alkalmazás, úgy is mint működő ütközésvizsgáló szoftver, és mint a tesztelést megkönnyítő eszköz illetve a program további fejlesztésével kapcsolatos lehetőségek.

5. Fejezet Az Összefoglalás című fejezetben kritikai összegzésre kerül az elvégzett munka. Összegzem a dolgozat megírásából származó tapasztalataimat.

1.2. Az orvosi képalkotó berendezésekről

1.2.1. A képalkotók története

Bármilyen kultúrát is vizsgálunk láthatjuk, hogy a gyógyításért tett erőfeszítések egyidősek az emberiséggel. Az időszámításunk előtt 320-ban alapított alexandriai iskolában már a tudományosság igényével bíró munkáról beszélhetünk. Voltak, akik az agyvelőről és a környéki idegrendszerről, voltak, akik a szívről, szívbillentyűkről, keringésről és voltak, akik az emésztő szervrendszerről gyűjtöttek össze alapos információkat hála az intézetben rendszeresen tartott boncolásoknak. A Római Birodalom felbomlásával és a

klasszikus ókori világ elestével sajnos a rengeteg felhalmozott tudás jó része feledésbe merült, kis hányada az arab világra korlátozva élt tovább. Európába a XIII - XIV. században Olaszországban meginduló tudományos restauráció által kerültek vissza az addig felhalmozott és el nem felejtett anatómiai ismeretek. Andreas Vesalius (1514–1564) 1543-ban kiadott művében (De humani corporis fabrica libri septem) remek, pontos illusztrációk szerepeltek az emberi anatómiáról.[1]

Fontos áttörést az emberi test működésének a megértésében az orvosi képalkotó eljárások megalkotása, elterjedése hozott. A képalkotók nem csak a szervezet megismerésében, de a diagnosztikában, és betegségek előrejelzésében is fontos új lehetőségeket hoztak. Egy orvosi képalkotókkal foglalkozó dolgozatban nem lehet, hogy ne hangozzék el Wilheltm Conrad Röntgen neve. A német fizikus az akkoriban nagyon divatos témát, katód sugárzást vizsgálva lett figyelmes arra, hogy az általa X (mint ismeretlen) sugárzásnak nevezett sugárzás áthatol a nagyobb sűrűségű fémeket kivéve szinte mindenen. Pár hetes kísérletezés után előállt az eredményeivel, amiben már felhasználási javaslatokat is tett, és a felesége kezéről készült képet is mellékelte. Ez látható a 1.2 ábrán. [2]

1.2.2. A modern eszközök

Manapság rengeteg különböző méretben készül orvosi képalkotó berendezés. Ha megnézzük az 1.1 ábrán szereplő gépet láthatjuk, hogy milyen monumentális eszközökkel is találkozhatunk, amiket sok esetben nem is emberek mozgatnak. Egy mai orvosdiagnosztikai képalkotó berendezés több motort, sok mozgó alkatrészt tartalmaz. Vizsgáljunk meg egy komputertomográfot: ennek az eljárásnak az alapja, hogy a röntgensugár forrása és a detektor is mozog a tárgy körül, és a sok különböző irányból mért intenzitás adatokból készül el később a teljes kép. Ezek mellett ráadásul az asztalt is mozgatni kell, hogy a test megfelelő részéről készülhessen a kép. Vagy vegyünk egy mágnesesrezonancia vizsgálatot, amikor az elektromágnesekből és detektorokból álló, olykor akár a több tonnás gyűrűk percenként több százat forognak a vizsgálandó személy körül, hogy ezzel a folyamatosan változó mágneses mezővel a hidrogén atommagok mágneses pólusát elmozdítsák, majd detektálják a visszaállását. Vagy vegyük a következő példát: a SPECT-nek (egy foton emissziós komptertomográfnak) akár több fejegysége forog körbe a páciens körül, hogy a CT-hez hasonlóan minden szükséges szögből képet készíttessen róla. Fontos, hogy ezeknek az alkatrészeknek a helyzetét követni, tárolni, mozgásukat összhangba hozni tudjuk, hogy működő és biztonságos tesztelést tudjunk végezni. Alapvető célunk kell legyen, hogy a lehető legkisebb kellemetlenség árán tudjuk vizsgálni a betegeket.

1.1. ábra. Láthatjuk, mennyire nagy és összetett a berendezés, motorok által mozgtaható karral. Kép forrása a [3] hivatkozásban található

1.2. ábra. Az első gyógyászati röntgen kép, amit maga Wilhelm Röntgen készített a felesége, Anna Bertha Ludwig kezéről. A felirat németül annyit tesz: Kéz gyűrűkkel 1895 december 22-én Kép forrása a[4] hivatkozásban található

1.2.3. A képalkotó berendezések dolgozat szempontjából fontos tulajdonságai

A példa kedvéért vizsgáljunk meg egy SPECT-CT kombinált gépet. Ebben az esetben, ha mondjuk egy két fejegységes SPECT-ről beszélünk akkor ezeknek az egységeknek a betegtől való távolságának beállítását és a körülötte való elforgatását akár több motorral is elvégeztethetjük. Ezeken kívül már korábban említettem, hogy a tárgyasztalt is képesnek kell lennünk motorokkal mozgatni, és persze a CT sugárforrást és detektort is mozgatnunk kell a vizsgált alany körül, ezekhez is motorok szükségesek. Ezeket a motorokat vezérelni kell, és tisztában kell lennünk minden alkatrész aktuális pozíciójával és elfordulásával. A berendezésnek kell, hogy legyen egy központi vezérlő egysége, ami ezeket a motorokat összhangba hozza. Nyilván ez a vezérlő egység programozható kell, hogy legyen, mert egy-egy detektor vagy kamera állása akár mérésenként más kell, hogy legyen. Ezeknek a vezérlő programoknak a tesztelésére előnyös, ha van egy már elkészült számítógépes háromdimenziós modell, amiben a már korábban beállított méretű alkatrészek, a megadott és szabadsági fokok szerint mozoghatnak a beérkező motor vezérlő jelek alapján. Ennek a szimulátornak feladata az is, hogy észlelje, hogyha két alkatrész összeütközik, vagy ha egy motort még járatna a vezérlő, de a meghajtott alkatrész elérte a szélső állapotát és nem mozgatható már tovább.

1.3. A tesztelés fontosságáról

Miért tesztelünk? Elvárás, hogy egy fejlesztett szoftver vagy egy megalkotott berendezés helyesen működjön, legyen stabil, megbízható, könnyen kezelhető, logikus, átlátható és ízléses. Egy hibának sokféle következménye lehet. Lehet, hogy fel sem tűnik, észre sem vesszük, lehet azonban, hogy anyagi vesztességgel jár, ha például egy vezérlőszoftver hibásan van beállítva, és két robotkar összeütközik. Vagy időt veszítünk mert lassan működik egy szoftver, akadozik, hosszú idő mire betölt, vagy netán lefagy, és nem tudjuk elmenteni a munkánkat, így elveszítjük, amin korábban dolgoztunk. Bármilyen hiba is lép fel, azzal a fejlesztő vállalat jó híre is csorbul. A legveszélyesebb vagy legnagyobb probléma az lenne, ha egy mérnöki munka személyi sérüléssel vagy még rosszabb, akár halállal járna.

Hibák a szoftver fejlesztésben Egy hiba keletkezésének a legvalószínűbb útja, egy emberi tévedéssel kezdődik, adódhat ez átgondolatlanságból, az ismeret hiányából vagy, csak figyelmetlenégből is, például egy mellé ütésből. Angolul ezt mistake-nek vagy error-

nak hívjuk. Ebből, keletkezik egy tényleges hiba, a "testet öltött tévedés", ami belekerül a kódba vagy a dokumentációba. Ezt nevezi az angol bug-nak, defect-nek vagy fault-nak. Ebből nem minden esetben, de gyakran adódik meghibásodás, mikor nem jól, vagy nem úgy működik egy rendszer, ahogy szeretnénk. Ez nevezik failure-nek angolul.

Egy tesztelés során az a célunk, hogy minél több meghibásodást felfedezzünk, ezekből aztán debug-golással (a program hibakereső módban való futtatásával, amikor képes a fejlesztő a kívánt pontokon megszakítja a szoftver működését, és megvizsgálni egy-egy változó aktuális értékét) meghatározható a hiba, amit ki tudunk javítani. Ebből a hibából kiindulva akár egy úgynevezett root cause analízist is lehet és érdemes is végezni, vagyis megkeresni, hogy mi volt a kiváltó ok, mi volt az a tévedés, az a mistake ami a hibához vezetett.

A softvertesztelés 7 alapelve a következő:

1. A hibajelenlét kimutatása

   Ha nem találunk hibát, az nem jelenti azt, hogy nincs is hiba a rendszerben. Lehet, hogy csak nem sikerült kimutatni. A tesztelés csak a hiba meglétét tudja megmutatni, azt nem lehet kimutatni, hogy nem létezik hiba.

2. Nem lehetséges kimerítő tesztelés

   Minden programút bejárása túlságosan sok erőforrást igényel, annyira sok elágazás lehetséges, hogy nem jellemző, hogy mindent le tudjunk fedni. Általában kockázat alapú tesztelést végzünk, a legfontosabb, legkockázatosabb folyamatokat teszteljük a legalaposabban.

3. Korai tesztelés

   Minél hamarabb találunk meg egy hibát, annál olcsóbb egy hibát kijavítani. Például, ha már dokumentációs szakaszban kitűnik egy elírás, amit így nem fejlesztenek le, sokkal olcsóbban meg megúszható, mint ha fölösleges fejlesztési órák mennek rá.

4. Hibák csoportosulása

A hibák jellemzően nem egyenletesen oszlanak meg a kódban, a Paretto elv igaz a programfejlesztésre is. A hibák nyolcvan százaléka a kód húsz százalékában megtalálható. Ezek alapján a tesztelés során is érdemes a kritikus részekre összpontosítani.

5. Féregirtó paradoxon

   Ha mindig ugyan azokat a teszteket futtatjuk le, akkor azzal azon a területen eljuthatunk odáig, hogy új hibát nem fedezünk fel. Így könnyen arra a téves következtetésre juthatunk, hogy hibátlan a rendszerünk, pedig az 1. pont alapján, a valóság az, hogy csak azt tudjuk megmondani, hol nincs hiba.

6. A tesztelés függ a körülményektől

   A tesztmódszereket igazítani kell az adott projekthez, például a mi esetünkben , már a Szakdolgozati témabejelntő nyilatkozatban említett Hippokratészi esküre is hivatkozva a legfontosabb egy orvosi képalkotónál az, hogy ne ártsunk vele. Kötelességünk alaposan letesztelni, hogy ne fordulhasson elő semmilyen baleset.

7. Hibátlan rendszer téveszméje

   Ha a szoftverünk úgy látszik, hibátlanul működik, akkor is meg kell vizsgálnunk, hogy a megrendelői igényekkel egyezik-e. Itt két fogalom fontos:

   - Verifikálás: annak a vizsgálata, hogy a szoftver megfelel-e a dokumentációban leírtaknak. Ezt jellemzően a fejlesztő vállalat végzi.
   - Validálás: annak a vizsgálata, hogy a szoftver megfelel-e az elvárásoknak. Ezt jellemzően a megrendelő teszteli.

1.4. A célkitűzések

Az előzőek alapján szükség van egy olyan programra, ami a fizikai berendezés nélkül, képes csak a vezérlő egységet tesztelni. Szimulálja a motorokat és megjeleníti az alkatrészeket valós időben. Az én célom és feladatom az volt, hogy készítetsek egy programot, ami általánosan használható: képes beolvasni alkatrészek paramétereit, ezeket a lehető leglogikusabb formában tárolni, és megjeleníteni egy háromdimenziós szimulációban. A motorok mozgását a futás alatt folyamatosan figyeli, és az elmozdulások alapján a tárolt

adatokat frissíti, és a megjeleníti. Amennyiben az alkatrészek egymással ütköznének, vagy elérnék a végállapotukat, akkor azt jelezi. Emellett figyelembe kell vennünk, hogy bizonyos alkatrészek egymással való ütközésének vagy inkább összeérésének engedélyezésére szükség vam, gondolva például az egymáshoz érő vagy akár egymáshoz szerelt egységekre is.

2. fejezet

Előzmények

2.1. A projekt életrehívásának okai

Ahhoz hogy a 1.3 alfejezettben bemutatott tesztelés minél hatékonyabban működjön, sok nagy, dága berendezést kellene tartani, ezekkel teszteket végezni. Egy ilyen gép orvosi műszerekkel van felszerelve, hatalmas helyigényű, nagyon nehéz, emiatt nehezen mozgatható, ráadásul nagyon drága. Csak azért fenttartani, hogy a lehetséges ütközésekre teszteljék, és összetörjék egy-egy rosszul kivitelezett vezérlő szofter esetében, nem kifizetődő. Felmerült az igény arra, hogy a vezérlő egységet, a berendezés többi része nélkül, egy szimulátorra kapcsolva is tesztelni lehessen. Ezzel a megoldással nincs szükség akkora befektetésekre, a berendezés fenttartására. Helytakarékosabbá lehet tenni a tesztelést, és az esetleges fizikai, és anyagi károkat egy-egy hiba kimutatása közben el lehet kerülni.

2.2. Egy hasonló problémára adott megoldás

A legtöbb CAD vagyis Computer-aided Design (számítógép segítette tervezés) szoftver lehetőséget ad arra, hogy háromdimenzióban rajzoljuk meg az alkatrészeket, vagy a beolvasott tervek alapján gyakran el tudják készíteni a háromdimenziós képét egy-egy berendezéseknek. A legtöbb ilyen program csak rezolúciós alapon képes teszteket végezni, vagyis minden lehetséges állapotát megvizsgálja egy-egy alkatrésznek és ez alapján meg tudja határozni az ütközési zónákat. Léteznek olyan porgramok, amikben vannak olyan funkciók, amik képesek megvalósítani motor mozgásokat, sőt ütközéseket tesztelni. Egy ilyen program például az Altair Inspire [5] , aminek van mozgás analizáló funkciója. Miután a tesztelni kívánt szerkezetet virtuálisan felépítettük, a szoftver lehetőséget ad arra, hogy a motoroknak előre definiált mozgási görbét adjunk meg, és az ezekből adódó

esetleges ütközéseket képes jelezni. Ezt a görbét lehetőségünk van táblázatos formában importálni is.

Előnyei:

- háromdimenziós vizualizációra alkalmas
- képes motorkat reprezentálni
- jól optimalizált, emiatt az egész gépet, a legapróbb csavarig menő részletességgel képes vizsgálni
- importálható a motorok mozgási görbéje
- ekkora gép tervezése során felmerülő egyéb kérdésekre is választ adhat
- egy egész fejlesztői csapat áll a projekt mögött, jelentős infrastruktúrával és tapasztalattal

Hátrányai:

- nagyon hosszadalmas megalkotni a tesztelni kívánt gép háromdimenziós modelljét
- a motorok nem képesek folyamatos kommunikációra, így nem lehetne éles, futás idejű teszteket végezni, csak begyűjteni az adatokat a motorvezérlőtől, és arra futtatni egy tesztet.
- bár jól optimalizált, a rengeteg, számunkra fölösleges funkciója miatt igen nagy számolási kapacitás szükséges hozzá
- nem specifikusan a problémára ad választ
- egy ilyen szoftver igen sok pénzbe kerül
- a bonyolultsága miatt be kéne tanítani az emberekt, akik használni fogják.

2.2.1. A saját program fejlesztése melletti döntés indolkása

Ahhoz, hogy szimulált tesztelést tudjunk végezni, számos problémára kell megoldást találnunk. A rendszernek tudnia kell kezelni a vezérlő egységet, ugyanolyan üzenetváltásokat kell megvalósítania, mint egy létező, fizikai berendezésnek. Ennek a megoldására a munkacsoport, akikhez csatlakoztam a Tango protokollt találta legalkalmasabbnak.

Egy másik szükséglet, hogy a tesztrendszer minél általánosabban használható legyen. A fent bemutatott megoldás megköveteli, hogy a felhasználó elkészítse egy grafikus felületen a gép háromdimenziós modelljét, majd megadja a futtani kívánt programot. Ezzel nehezíti a program újrafelhasználhatóságát, és ellehetleníti a valós idejű, vezérlő egységhez kötött tesztelést. Olyan megoldást kellett hát készítnünk, ami képes a Tango eszközökkel komunikálni - mivel azok segítségével megfelelően tudunk a CAN-busz protokoll szerint kommunikáló vezérlővel kapcsolatot teremteni egy egységes felületen keresztül (CAN-busz, vagyis Controller Area Network, egy olyan busz szabvány, ami központi számítógép nélküli kommunikációt tesz lehetővé mikrokontrollerek között [6]). Ezen felül könnyen változtathatók vagy betölthetőek a modellezni kívánt gép paraméterei.

Egy-egy a fenti példában szereplő szoftver igen nagy befektetést igényel, nem csak az anyagiakat, de a megismerésére fordított időt is beleértve, . Ráadásul jellemzően rengeteg számunkra fölösleges funkciót tartalmaznak, amik esetlegesen megnehezíthetik, hogy arra használjuk őket, amire szeretnénk.

Egy nem túl bonyolult de annál specifikusabb problémára keresünk olyan megoldást, amit igény esetén könnyedén testreszabhatunk, új funkciókat implementálhatunk. Ezen okok miatt jellemző, hogy a nagyobb robotikával foglalkozó vállalatoknak saját tesztszoftvereik vannak. Mi is ezért döntöttünk úgy, hogy magunk keszítünk megoldást.

2.3. A szükséges matematikai háttér bemutatása

2.3.1. Bounding Box

A számítógépes háromdimenziós modellezés, a játékok, és a gépi tanuláson alapuló képelemző megoldások területén is elterjedt megoldás a Bounding Box-ok alkalmazása. E szerint egy alakzatot reprezentáljon egy olyan téglatest (doboz) amibe az alakzat éppen belefér. Egy alakzat Bounding Box modelljének egy lehetséges megközelítése, ha megszabjuk, hogy a generált doboz oldalai a használt koordináta rendszer tengelyeivel párhuzamosak legyenek (Axis-aligned bounding boxes). Ebben az esetben a téglatest oldalai az alakzat tengelyekre vett merőleges vetületeként meghatározhatóak. Minél teljesebben tölti ki a Bounding Box-ot az alakzat, annál optimálisabb a megalkotott modell. Tehát azt a koordináta rendszert keressük a térben minden alkatrészhez, aminek az alakzat tegelyekre vett merőleges vetületeinek szorzata (vagyis a Bounding Box térfogata) és az alakzat térfogata a lehető legkisebb mértékben tér el egymástól. Vagyis keressük azt

a koordináta rendszert, amikor $min(|V_{BoundingBox} - V_{alakzat}|)$ [7]

2.3.2. A Bounding Box-ok térbeli elhelyezkedését leíró adatok

A szimulált Bounding Boxokat úgy tekintettem, mint egyszerűsített merev testeket. A merev test modellben a testeket már nem csak tömegpontként értelmezzük, hanem figyelembe vesszük az alakjukat, kiterjedésüket, tömegmegoszlásukat. Annak megállapításához, hogy két alkatrész összeütközik-e vagy sem, nincs szükség a test tömegének, pláne a tömeg eloszlásnak ismeretére. A Bounding Box-ok mozgásának leírására tehát jól alkalmazható a modell egyszerűsített változata, amiben a tömegeloszlást nem vesszük figyelembe. Egy háromdimenziós test térbeli helyzetét egyértelműen meghatározza a test három, nem egy egyenes mentén elhelyezkedő pontjának helyzete. Ez egy egyszerű gondolatkísérlettel belátható. Ha egy testet egy adott pontján rögzítünk, az e körül a pont körül bármely irányba szabadon elmozdulhat. Ha a test még egy pontját fixáljuk, abban az esetben a két pont által meghatározott tengely mentén még mindig szabadon el tud forogni. Ha azonban egy harmadik, az előző kettővel nem egy tengelyen elhelyezkedő pontban is fixálva van a test, meggátoltuk azt a forgást is, ami a két pont esetén még lehetséges volt. Ezt a három pontot, ha a háromdimenziós derékszögű koordináta rendszerben akarjuk megadni, ahhoz pontonként három adatra van szükségünk. Ez a kilenc adat nem független egymástól merev, deformálódni nem képes testek esetén, hiszen a pontok egymáshoz viszonyított távolsága állandó. Így például, ha a három pont közötti három távolságot a koordinátáik segítségével szeretnénk kifejezni láthatjuk, hogy igazából 6 független változó van, egy merevtest szabadsági fokainak száma tehát 6. Abban az esetben, ha ez a test valamilyen felület, vagy görbe mentén képes csak mozogni, ez a szám csökkenhet.

Látható tehát hogy egy Bounding Box helyzetének leírásához 6 változó paraméterre van szükség. Amiatt, hogy egy merev test mozgása leírható elforgatások (rotáció) és egy eltolások (transzláció) egymásutánjaként érdemes volt ezt a hat változót a test középpontjába mutató helyvektorként és az akörül való forgásvektorként értelmezni. Annak a bizonyítására, hogy ezekkel a vektorokkal bármilyen mozgást leírhatunk vegyük egy tetszőleges merev test P pontjának helyvektortát egy K külső ponthoz rögzített C_K koordináta rendszer szerint P_K-nak és egy tetszőleges B belső ponthoz rögzített C_B koordinátarendszer szerint P_B-nek. Ekkor felírható a helyvektortok közötti összefüggés, miszerint

$$r_{PK} = r_{PB} + r_{KB} \tag{2.1}$$

ahol r_{PK} és r_{PB} rendre C_K, C_B koordinátarendszer szerinti helyvektor és r_{KB} a C_K koordinátarendszerben a B pont helyvektora. Ez alapján a P pont elemi elmozdulása felhírható a

$$dr_{PK} = dr_{PB} + dr_{KB} \tag{2.2}$$

alakban. Mivel a merev test definíciója szerint annak két pontja közötti távolság mindig állandó, így az r_{PB} vektor csak elfordulhat, vagyis felírható egy elmei szögelfordulás vektor segítségével, legyen ez $d\varphi$. Ekkor

$$dr_{PB} = d\varphi \times r_{PB} \tag{2.3}$$

vagyis behelyettesítve a korábbi egyenletbe:

$$dr_{PK} = dr_{KB} + d\varphi \times r_{PB} \tag{2.4}$$

Vagyis a merev test elemi elmozdulását felírhatjuk úgy is mint a B belső pont transzlációja és a B pont körüli rotáció. [8]

Az is belátható, hogy a B pont akármelyik belső pont lehet, tetszés szerint megválasztható. A könnyebb kezelhetőség érdekében a pozíció vektor az aktuális Bounding Box testátlóinak metszéspontjába mutat, és a forgatást e körül a pont körül értelmezzük. A transzláció elég egyszerűen leírható matematikailag, a képvektor előáll a kiindulási helyvektort és a transzlációs vektor összegeként. Bármilyen elforgatás a térben előállítható a három koordináta tengellyel párhuzamos egyenesek körüli tengelyes forgatások egymásutánjaként. Mivel a tengely körüli elforgatás egy homogén, lineáris leképezés, így létezik egy leképezési mátrix, amivel a fenti vektort megszorozva megkapjuk a képvektor. Mivel mátrixszorzásra visszavezethető, fontos figyelembe venni, hogy a leképezések sorrendje nem felcserélhető. A rotációs vektor a B belső ponthoz, (ami a mi esetünkben, mint már korábban említettem a testátlók metszéspontja) rögzített koordináta rendszer x, y, z tengelye körüli elforgatások mértékét tárolja az egyes koordinátáiban. [9]

2.3.3. A mozgás matematiki háttere

A dolgozatomban, mivel a projekt indulási szakaszába tudtam becsatlakozni, a célom az volt, hogy a két legegyszerűbben megvalósítható, leggyakrabban előforduló mozgás fajtát reprezentálni tudjam: ezek a lineáris és a forgó mozgás. A test transzlációja nem tér el egy pont transzlációjától, vagyis a középpontjának helyvektora és a transzlációs vektor összege a kép középpontjába mutat. Eközben a test nem fordul el a saját középpontja körül. Ezzel szemben, a test körmozgása közben a pozíciója és az elfordulása

is folyamatosan változik. Bár a körmozgás sokféleképpen reprezentálható lenne (csak a példa kedvéért, ha a test minden pontjára, vagy akár csak a csúcsokba mutató vektorokra értelmeznénk a fentebb kifejtett rotációt, akkor is eredményre jutnánk) mivel a programban a korábban leírtak szerint, mi ezzel a két vektorral, vagyis a pozíció és elfordulás vektorokkal szeretnénk leírni a test helyzetét, így ezeket kell tudnunk kiszámolni.

A körmozgás

A körmozgást egyértelműen meghatározza a térben a forgás középpontja, a forgástengely, valamint a sugár hossza. Mivel tudjuk, hogy a körmozgást végző alkatrészünk pályájának sugara megegyezik a pozíciójának a forgás középpontjától számított távolságával így a sugár megadására nincs szükség. Leonhard Euler a merevtestek mozgásának leírását úgy valósította meg, hogy a testhez hozzárendelt egy koordináta rendszert és kimondta, hogy ebbe a koordináta rendszerbe három egymás utáni elforgatással az eredeti koordináta rendszerből át lehet térni. Egy tetszőleges tengely körüli forgatás egyik megvalósítása lehet a következő: áttérünk a forgástengely, mint x tengely szerinti koordináta rendszerbe, ott az új x tengely szerint elforgatjuk a testet, majd az áttérési mártix transzponáltjával visszatérünk az eredeti koordináta rendszerünkbe. Vagyis a forgatási mátrix előáll a következő módon:

$$R = TR_aT^T \tag{2.5}$$

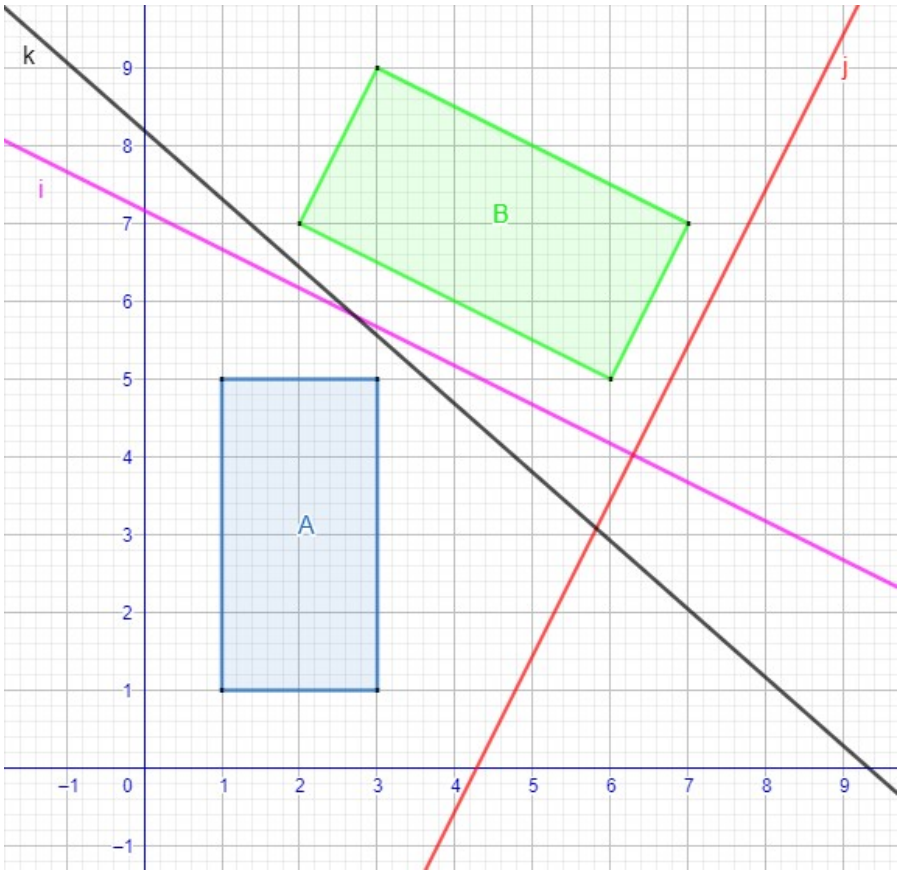
ahol R a forgatási mátrix, T az áttérési mátrix, R_a az x tengely körüli forgatás abban a koordináta rendszerben, amibe áttértünk. A forgásvektorunkat is meg kell alkotnunk. A test elforduálása, vagyis a forgatás előtti belső koordináta rendszerhez viszonytott elforduálásának Euler szögei három, nem egy egyenesre eső pontjának koordinátáiból, és a forgatás előtti koordinátákból kiszámolható. [10]

2.3.4. Ütközési egyenletrendszer

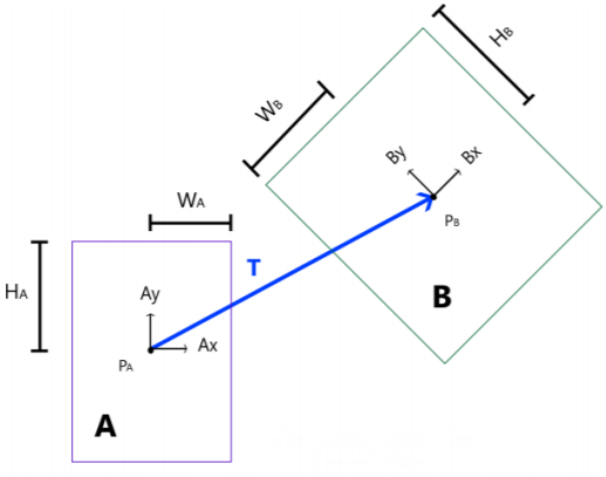
Ütközés vizsgáló egyenlőtlenség kétdimenzióban:

Az ütközési egyenletek az úgynevezett Separating Axis Theorem (szabad fordításban: elválasztó tengely elmélet) alapján írhatók fel. Először a könnyebb megértés érdekében két dimenzióban a tétel így szól:

"Két konvex alakzat akkor és csak akkor nem érinti és nem is metszi egymást, ha létezik egy olyan egyenes, amire ezeknek az alakzatoknak a merőleges vetületét véve szakadás van a két vetület között. Ilyenkor a tengelyt elválasztó tengelynek, a két vetület közötti szakaszt meg elválasztó szakasznak nevezzük." [11]



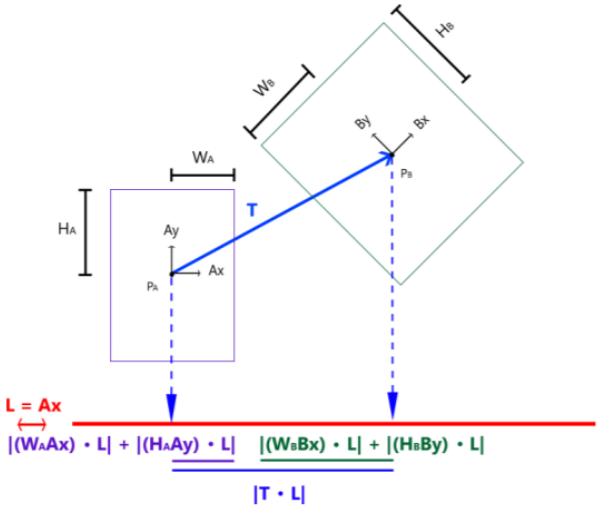
2.1. ábra. Az A és a B téglatestek között látható, hogy létezik a szürke, k jelű elválasztó egyenes, ekkor léteznie kell egy olyan elválasztó egyenesnek is, ami párhuzaoms az egyik téglatest valamelyik oldalával, mint a lila színű, i jelű egyenes.



2.2. ábra. Az A és a B téglatest, a középpontba az oldalakkal párhuzamos egységvektorok, a belső koordináták, és ezek távolsága. Jelölve vannak még az oldalak hosszának fele is. Kép forrása a [11] hivatkozásban található

Ez a definíció ekvivalens azzal, hogy létezik olyan egyenes, ami úgy osztja két félsíkra a teljes síkot, hogy nem érinti egyik alakzatot sem, és a két alakzat két külön félsíkon található. (Ha létezik ilyen egyenes, akkor az merőleges az elválasztó tengelyre) Téglalapok esetén, ha létezik ilyen elválasztó egyenes, akkor belátható, hogy léteznie kell olyan elválasztó egyenesnek is, ami párhuzamos valamelyik téglalap valamelyik oldalával. (látható a 2.1 ábrán) Vagy másképpen akkor és csak akkor nem ér össze két téglatest, ha létezik legalább egy olyan elválasztó egyenes, ami párhuzamos valamelyik téglatest valamelyik élével. Ez alapján elég megvizsgálnunk a téglalapok oldalaival párhuzamos egyeneseket. Ahhoz, hogy kimondhassuk két téglalapról, hogy nem érnek össze, léteznie kell legalább egy ilyen elválasztó egyenesnek. Nekünk legalább egyet találni kell. Mivel a téglalap szemközti oldalai párhuzamosak, téglalaponként két egyenest elég megvizsgálnunk. Két téglalap esetén tehát négyet.

Vegyünk két téglalapot, nevezzük őket A-nak és B-nek. Nevezzük a téglalapok átlóinak metszéspontját a középpontjuknak, és jelöljük ezeket a középpontokat P_A -val és P_B -vel. Minden egyenesnél értelmezzünk egy lokális koordináta rendszert, amik a középpontokhoz rögzítettek, és a koordináta rendszerek tengelyei a téglalapjuk oldalaival párhuzamosak. Ezek alapján a négy megvizsgálandó irány, amire merőleges elválasztó egyeneseket keresünk az A_x, A_y, B_x, B_y ahol ezek a koordináta rendszereink egységvektorai. Ebben az esetben, ha Proj() az A_x tengelyre való vetítést jelenti és T a két középpont távolsága (ahogyan a 2.2 ábrán látható) akkor felírható a vizsgálathoz szükséges egyenlőtlenség az



2.3. ábra. Pirossal jelölve a vizsgált tengelyt láthatjuk, lilával az A jelű téglalap két oldalának a felének a hosszvektorának a merőleges vetületét láthatjuk, míg zölddel a B jelű téglatestét erre a tengelyre. A kék szakasz a középpontjukat összekötő vektor merőleges vetületét jelöli. Láthatjuk, hogy a lila és zöld szakaszok együttes hozza nem éri el a kék szakasz hosszát, vagyis a létezik erre a tengelyre merőleges elválasztó egyenes. Kép forrása: Kép forrása a [11] hivatkozásban található

A_x vektorra:

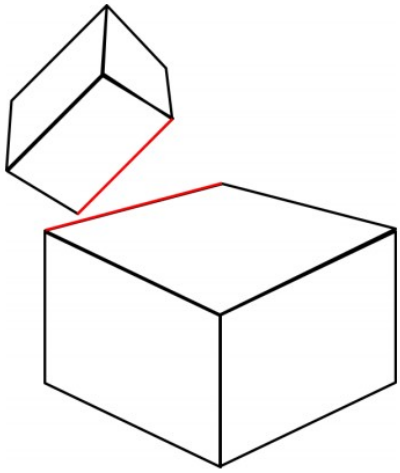
$$|Proj(T)| > 1/2 * |Proj(A)| + 1/2 * |Proj(B)| \tag{2.6}$$

mivel a téglalapok esetében bármilyen tengelyre történő merőleges vetítés esetén a téglalap középpontjának vetülete a kép szakasz felezőpontjában helyezkedik el. Nevezzük a téglalap oldalhosszainak felét W-nek mint szélesség és H-nak mint magasság, a vetítési tengely egységvektora legyen L. Ha tisztában vagyunk azzal, hogy egy v vektornak egy egyenesre vett merőleges vetületének hossza egyenlő a v vektor és az egyenes egységvektorának skaláris szorzatával, akkor ez az egyenlőtlenég a következőképpen írható át:

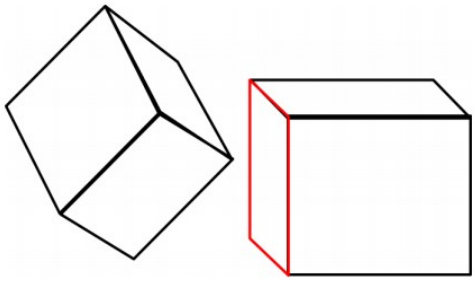
$$|T \cdot L| > |(W_A * A_x) \cdot L| + |(H_A * A_y) \cdot L| + |(W_B * B_x) \cdot L| + |(H_B * B_y) \cdot L| \tag{2.7}$$

minden ahol a "*" a skalárral való szorzást, a "·" a skalárszorzást jelenti.

Tehát, ha a fenti egyenlőtlenséget minden $L = \{A_x, A_y, B_x, B_y\}$ tengelyre megvizsgáljuk, és van olyan eset, amikor igaz, akkor létezik elválasztó szakasz. Ha létezik elválasztó szakasz, létezik elválasztó egyenes, és ha létezik elválasztó egyenes, akkor a két téglatest nem ér össze.



2.4. ábra. Bár nincs olyan elválasztó sík, amelyik bármely téglatest bármely lapjával párhuzamos, a képen pirossal jelzett élekkel párhuzamos síkok elválasztó egyenesek lesznek. Vagyis az elválasztó síkot meghatározó két él a két téglatest egy-egy éle. A kép forrása a [11] hivatkozásban található



2.5. ábra. Ebben az esetben létezik olyan elválasztó sík, ami az egyik téglatest egyik oldalával párhuzamos. Vagyis az elválasztó síkot meghatározó két él egy téglatest két éle. A kép forráKép forrása a [11] hivatkozásban található

Áttérés 3 Dimenzióba:

Ahogy a síkot két félsíkra bontja egy egyenes, úgy bontja a teret két féltérre egy sík. Három dimenzióban tehát nem elválasztó egyeneseket, hanem elválasztó síkokat keresünk, az elválasztó szakaszok ezekre a síkokra merőlegesek.

A 2.4 és 2.5 ábrákon láthatunk két olyan esetet, amit fontos megvizsgálnunk a háromdimenziós ütközésvizsgálatnál. A 2.4 ábrán látható, hogy a két téglatest az éleivel közeledik egymás felé, és egyértelmű, hogy nem ütköznek. Mégsem tudunk az egyik oldallal párhuzamos elválasztó síkot kifeszíteni a két téglatest közé. Azt viszont látjuk, hogy az a sík, ami mind a két, egymás felé forduló éllel merőleges, elválasztó síkot alkot. Ha a 2.5 ábra esetét vizsgáljuk, látjuk, hogy nincs olyan élpár, mint az első esetben, viszont a jelzett oldallal párhuzamos sík elválasztó sikként tud szolgálni. Élk helyett beszéljünk inkább arról a három irányról téglatestenként, amit az élek meghatároznak. Ahogy a példákból kiderült minden lehetséges iránypár által meghatározott síkot meg kell vizsgálnunk. Ezek alapján a vizsgálandó síkok száma: 6 (3 téglatestenként)+ 9 (3*3 tengely), azaz összesen 15 lehetséges sík valamelyikével párhuzamos elválasztó síkot keresünk.

Az egyenlőtlenségünk nagyon hasonlít a két dimenziósra, annyi különbséggel, hogy itt 3 dimenzióra vizsgálunk, így az egyes középpontokhoz rögzített koordinátarendszerek egységvektorait

- A_x (a kódban axis parallel uintvector)
- A_y (a kódban face parallel uintvector)
- A_z (a kódban up parallel uintvector) (ezekhez hasonlóan értendőek a B_x, B_y, B_z a B téglatestre) rendre az x, y, z tengeyekkel párhuzamos egységvektorok
- A_A (a kódban axis) a vízszintes kiterjedés fele
- U_A (a kódban up) a függőleges kiterjedés fele
- F_A (a kódban face) a kifelé mutató kiterjedés fele (ezekhez hasonlóan értendőek a A_B, U_B, F_B a B téglatestre)

Ekkor az egyenlőtlenség:

$$\begin{aligned} |T \cdot L| > |(A_A * A_x) \cdot L| + |(U_A * A_y) \cdot L| + |(F_A * A_z) \cdot L| + \\ + |(A_B * B_x) \cdot L| + |(U_B * B_y) \cdot L| + |(F_B * B_z) \cdot L| \end{aligned} \tag{2.8}$$

Akkor, ha bármely L

$$\begin{aligned} L = \{A_x, A_y, A_z, B_x, B_y, B_z, \\ (A_x \times B_x), (A_x \times B_y), (A_x \times B_z), (A_y \times B_x), (A_y \times B_y), (A_y \times B_z), \\ (A_z \times B_x), (A_z \times B_y), (A_z \times B_z)\} \end{aligned} \tag{2.9}$$

-re az egyenlőtlenség igaz, akkor létezik elválasztó szakasz, vagyis elválasztó sík, vagyis a téglatestek nem érnek össze. A "*" a skalárral való szorzást, a "$\cdot$" a skalárszorzát, a "$\times$" pedig a vektoriális szorzatát jelenti.

3. fejezet

Tervezés

3.1. A használt eszközök ismertetése

3.1.1. Tango

A Tango egy vezérlő/irányító rendszer, a CORBA (Common Object Request Broker Architecture, lehetővé teszi a kommunikációt különböző nyelveken írt, különböző számítógépeken futtatott programok között) egy keretrendszere, ami elrejti annak bonyolultságát, és új, specifikus irányítási rendszerhez szükséges funkciókat tartalmaz. A Tango egy egységes interfészt nyújt minden eszközhöz.[12]

3.1.2. Python

A Python-t egy holland származású informatikus Guido van Rossum kezdte el fejleszteni a nyolcvanas évek végén. Nevét a Monty Python-ról kapta. A Python általános célú, magas szintű programozási nyelv. Vagyis a programban használt kifejezések az emberi beszédhez állnak közelebb, nem pedig a gépi kódhoz. A programnyelv megalkotásánál gondoskodtak az olyan, a számítógépes architektúra miatt fontos folyamatokról, mint például a memóriakezelés. Emellett implementáltak bizonyos funkciókat mint például az objektumorientált nyelvi funkciók vagy a szövegeket, listákat, könyvtárakat kezelő rutinok. A Pythonhoz elterjedtsége miatt ráadásul rengeteg könyvtárat alkottak meg az évek során, ezzel még több funkciót téve elérhetővé. [13]

3.1.3. OpenGL

A Silicon Graphics API-a, vagyis alkalmazásprogramozási interfésze azzal a céllal készült, hogy a 3D gyorsítókkal való bonyolult kommunikációra alternatívát nyújtson, egy

minimális követelmény megléte után hardware függetlenül. Az OpenGL egy alacsony szintű API, amit a benne definiált függvények, és grafikai leképezések meghívásával vezérelhetünk. A dolgozatomban egységbe zárva csak a vizualizációért felelős fájlokban használtam. [14]

3.1.4. pygame

A pygame Python modulok halmazát foglalja magába, azzal a céllal, hogy videójáték készítését könnyítse meg. Előnye, hogy majdnem minden operációs rendszeren egyformán jól alkalmazható, és nagyon egyszerűen megtanulható. A modulra a háromdimenziós szimulátorhoz, és az ezekhez tazozó felhasználói interakciókhoz volt szűkség. Főleg a térben való mozgásra, és a billentyűparancsok implementálására szolgált. [15]

3.1.5. NumPy

A nunpy egy teljesen nyílt modul, kifejezetten a bonyolultabb számítási feladatok elvégzésére. A NumPy függvényeit a kódban főleg vektor műveletek elvégzésére használtam. Ez egy nagyon széles körben elterjedt modul, rengeteg elérhető példával és jó dokumentációval. A legkevesebb problémám a szükséges NumPy függvény megtalálásával volt. [16]

3.1.6. JSON

A JSON vagyis JavaScript Object Notation egy könnyen olvasható adattároló és adat cserélő protokoll, ami a JavaScript nyelvből alakult ki. Alkalmas egyszerű adatstruktúrák és asszociatív tömbök tárolására egyaránt. Mivel nagyon elterjedt, és általánosan használható, Python is natívan támogatja, így bármely JSON fájl egyszerűen beolvastatható, és jól kezelhető. [17]

3.1.7. REST API

A REST (Representational State Transfer) egy szoftver-architektúrális szabályrendszer. Egy ilyen típusú architektúra két részre bontható. Van egy szerver oldal és egy kliens oldal és közöttük kommunikáció. A kliens kérdéseket fogalmaz meg és indít a szerver felé. A szerver ezeket e kérdéseket feldolgozza, és megalkotja a megfelelő válaszokat, azokat küldi vissza a kliensnek. [18] Egy REST architektúrára hat állításnak kell igaznak lennie:

- Kliens-szerver architektúra: Léteznie kell nagyon jól külön választható kliens és szerver architektúrának, és a két réteget az interface-nek kell összekötnie. Mind-

addig, amíg az interface változatlan, a szerver és a kliens áthelyezhető és külön fejleszthető. Megvan tehát a jól elkülöníthető feladatköre a kliensnek, és meg van a szervernek is, ezek között nincs átjárhatóság.

- Állapotmentesség: A szerver nem tárolja a kliens állapotát. Bármikor beérkezik egy kérés a szerver felé, annak minden szükséges információt tartalmazni kell a megvalósításhoz.

- Gyorsítótárazhatóság: Egy kérésnek tartalmaznia kell az információt, hogy gyorsítótárazható-e az adott üzenet váltás, vagy sem. A gyorsítótár segítségével nem kell újra lekérnünk bizonyos kéréseket, hanem használhatjuk a már korábban kapott válaszokat. A meghatározás, hogy egy kérés gyorsítótárazható-e segít abban, hogy elkerüljük az esetleges elavult adatok kezelését.

- Réteges felépítés: Lehetőség van arra, hogy egy kliens ne egyenesen a végponti szerverhez kapcsolódjon, hanem közvetítő segítségét vegye igénybe. Erről, hogy kihez is csatlakozik éppen, a kliens általában nem tud.

- Igényelt kód: Ez az egyetlen opcionális megszorítás. Lehetővé teszi, hogy a szerver programrészleteket adjon át a kliensnek, amiket az képes futtatni.

- Egységes interface: egyszerűsíti, és szétválasztja a két oldalt, lehetővé téve a független fejlesztést. Ahhoz, hogy ez teljesüljön négy pontnak kell megfelelni:

  - Erőforrás azonosítása: a kérésekben megtörténik az erőforrások azonosítása, és a válaszban csak egy-egy kért adat reprezentációját küldi vissza a szerver a kliensnek. Esetünkben ez a válasz JSON formátumban érkezik, UTF-8-ban kódolva.

  - Erőforrás manipulációja ezeken a reprezentációkon keresztül: ha a kliens egy erőforrás reprezentációjával rendelkezik, és joga van hozzá, akkor ennek birtokában, képesnek kell lennie arra, hogy kitörölje, vagy módosítsa az erőforrást a szerver oldalon.

  - Önleíró üzenetek: Minden üzenetnek tartalmaznia kell a feldolgozásához szükséges információkat.

  - Hipermédia, mint az alkalmazásállapot motorja: A kliens aktuális állapotát a küldött és fogadott üzenetek határozzák meg.

Ha ezeknek a pontoknak megfelel egy program akkor RESTfull-nak nevezzük. [19]

3.2. A használt arhitektúra

A vezérlőegység független teszteléséhez elengedhetetlen, hogy az általa irányított motorok reprezentálva legyenek. A motorok és a vezérlőegység között CAN-busz kapcsolat, a megjelenítést, tesztelést végző szoftver és a virtuálus motorok közötti REST API kapcsolat megvalósítására a Tango Control eszközkészletet használjuk.

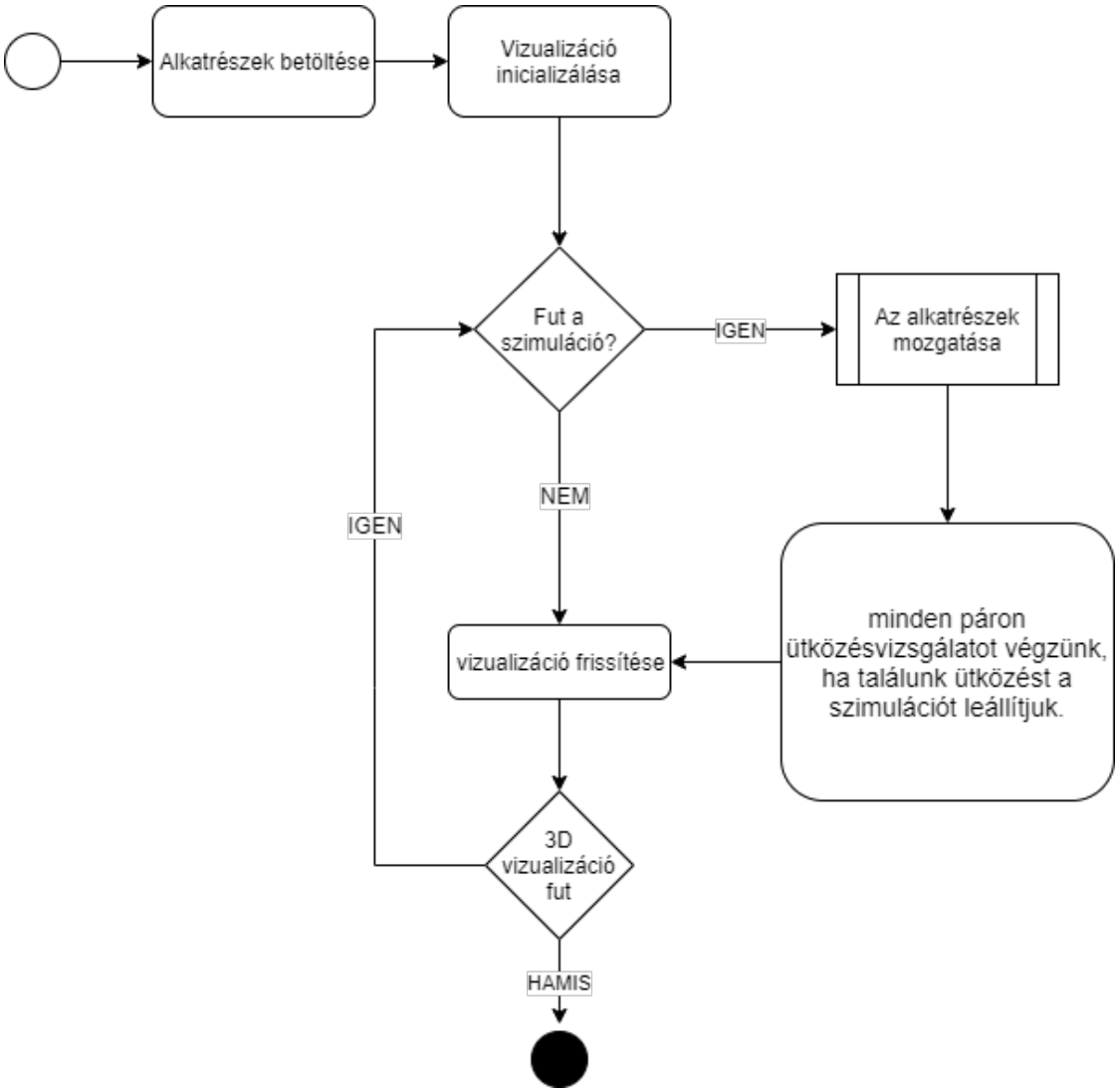
A megvalósítandó architektúrában egy központi egység tárolja és kezeli az alkatrészket reprezentáló adatokat, és elvégzi az ütközés vizsgálatot, vizualizálót is, ezt a továbbiakban adapternek nevezzük. Egy motor pozitív és negatív irányba képesek forogó ésvagy lineáris mozgatást végezni. A Tango eszköz csak ezt az adatot, vagyis a motor előjeles sebességét tárolja az aktuális időpillanatban, minden más információt az adapter tárol. A Tango eszköz, sebességét a motorvezérlő állítja be, és az adapter ezt a beállított sebességet kérdezi le.

3.2.1. Adapter

Az Adapter feladatai tehát a fenti, bekezdés alapján az alkatrészek adatainak tárolása, a motorokkal való kommunikáció, az ütközésvizsgálat és végül, de nem utolsó sorban a háromdimenziós vizualizáció. Amiatt, hogy az alkatrészeket reprezentáló adatokat az adapterben tároljuk, elég csak a motork aktuális sebességét rendszeresen lekérdezni. Figyelembe kell venni, hogy egy motor, akár több alkatrészt is megmozgathat, akár csatolt alkatrészekről is beszélhetünk és egy alkatrészre több motor is hatással lehet.

A 3.1 ábrán látható az adapter folyamatábrája. Az alkatrészek betöltése és a vizualizáció inicializálása után megkezdődik egy While ciklus. Ez a ciklus biztosítja, hogy időben folytonosan futni tudjon a program. A ciklusban történik az alkatrészek mozgatása és az ütközés vizsgálat. Ha ütközésre kerül sor a szimuláció megáll.

A mozgatásról a 3.2 ábrán láthatjuk a folyamatábrát. Végigiterálunk az alkatrészek listáján. Az aktuális alkatrészen elvégzünk egy végállapot vizsgálatot. Ha a vizsgálat eredménye pozitív, akkor a szimuláció futását jelző flag beállításával véget ér a szimuláció, és innentől csak a vizualizácó fut. Ha negatív a teszt eredménye, és motorról van szó, akkor megmozgatja a beállított alkatrészeket (ez akár önmaga, vagy másik motor is lehet). Ezután fovább iterál a következkező alkatrészre. Ha nem motor az aktuális alkatrész, a mozgatási lépés kimarad, csak a végállapot vizsgálat történik meg.

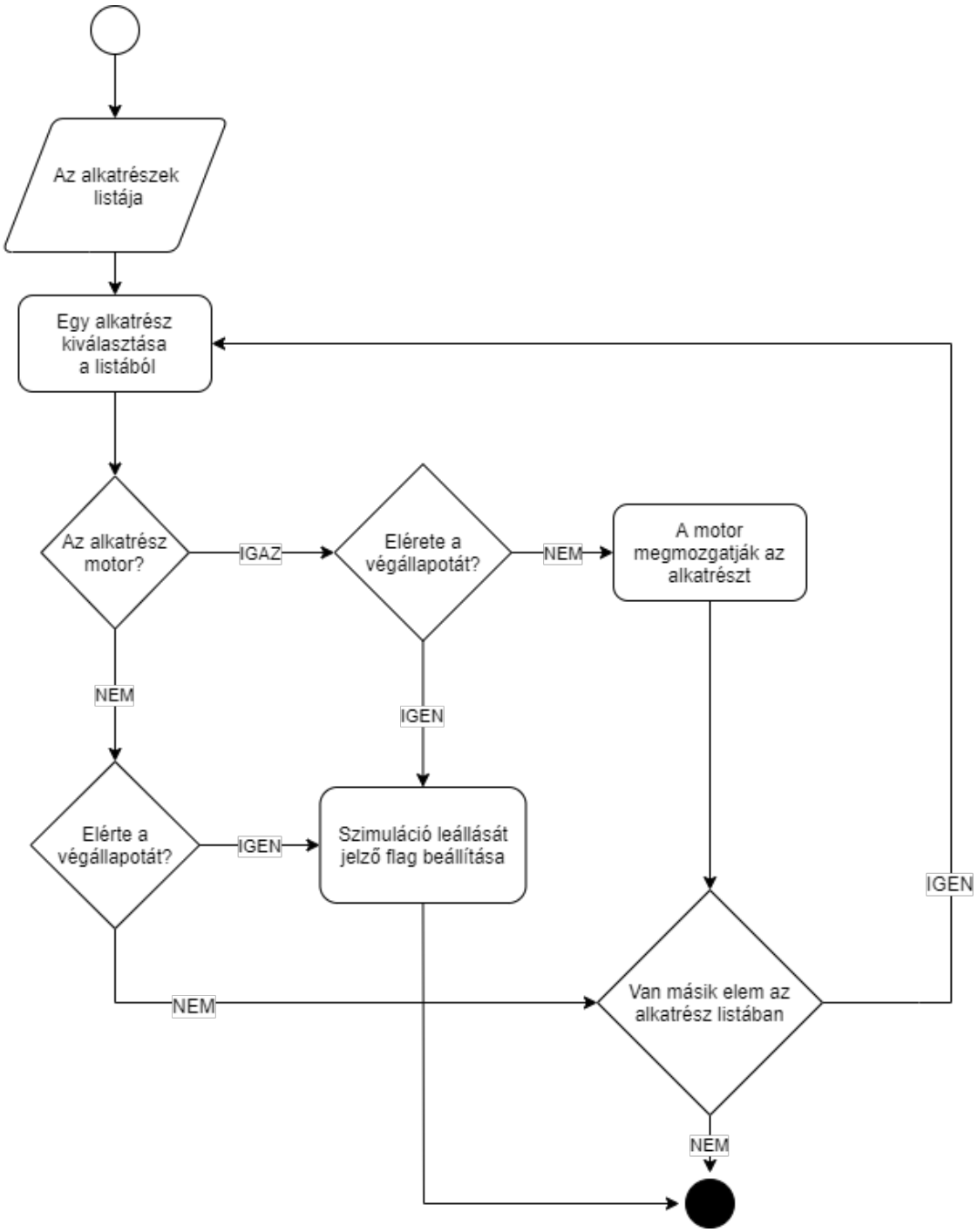


3.1. ábra. Az adapter folyamatábrája

Az ütközésvizsgálat során minden lehetséges alkatrészpárt, ami nincs benne a kihagyni való alkatrészek listájában, letesztelünk. Ha ütközést találunk, akkor a szimuláció futását jelző flag-et levesszük, a csak a vizualizáció renderelése folytatódik. Ha nem ütközik, akkor teszteljük a következp párt. A program akkor ér véget, ha a vizualizációs ablakot bezárjuk, vagy ha megnyomjuk az Escape billentyűt.

3.2.2. Tango szerver

A Tango arhitektúrálisan három nagy egységre bontható fel. Az egyik a klines applikáció, jelen esetben az adapterünk, a másik a tango adatbázis, és végül a harmadik a Tango szerver. Az eszközökhöz való hozzáférés a szerveren keresztül történik. Ez a szerver több eszközből áll, amik különböző eszköz osztályokhoz tartozhatnak. A szerver elindulásakor az egyes eszközök példányosodnak az eszközosztályok alapján. A kliens az adatbázison



3.2. ábra. Az alkatrész mozgatás folyamatábrája

keresztül importálja az eszközt, és onnan tud üzeneteket is küldeni.

3.2.3. A REST API kommunikáció

A Tango a CORBA és a REST API kommunikációra kínál saját megoldást. Ezek közül a második modernebb, felhasználóbarátabb és elterjedtebb. A Tango-hoz kapcsolódó legtöbb alkalmazás előre telepítetten elérhető a Tango Box nevű virtuális számítógép képfájlban. A dolgozat írása során ezt használtam minden Tango-val kapcsolatos feladat megoldására. A virtuális gépen már adott egy lehetséges REST API beállítás, amit már csak, mint docker fájlt, futtatni kell a kapcsolódás lehetőségéhez.

3.2.4. Tango device - A motor

A Tango eszközöknek beállíthatók parancsok, skalár, spektrum, kép és továbbított változók, csövek és állapotok. A motornak, amint reprezentálni szeretnénk van ki- és bekapcsolt állapota és sebessége, valamint az ezekhez tartozó bekapcsolás, kikapcsolás, inicializálás és sebesség beállítás parancsai. A sebesség beállítás egy *double* típusú értéket vár bemenetnek, ezt állítja be a motor sebességének. Ezek a parancsok, állapotok, és a változó elérhető REST API híváson keresztül, a tesztprogram és az adapter is így kommunikál a motor Tango eszközökkel.

3.2.5. Unit test, a vezérlőegység szimulálása

Az elkészült adapter és a motorok teszteléséhez fontos, hogy szimulálni tudjuk a vezérlő szoftvert is. Ennek a szükségletnek a betöltésére készült a *unit test*. A porgram arhitektúrája sokban merít az adapterben bemutatott megoldásokból. A program indításakor egy bekapcsolás parancsot és egy iniciálás parancsot hívunk meg minden motorra a REST API kommunikáción keresztül. Itt is szinte a teljes futásidő alatt fut egy while ciklus. Ennek segítségével folyamatosan figyelni tudjuk az egyes motoroknak beállított programot. Mikor a motornak új sebességet kell beállítani, egy PUT requesten keresztül, meghívjuk a sebesség beállításáért felelős parancsot, és megadjuk a megfelelő értéket. Ha az összes motor porgramja végzett, a motor Tango eszközöket kikapcsoljuk és a unit test végetér.

3.3. Az adapter megvalósítása

3.3.1. Az adatok betöltése

Az alapvető célkitűzés, (ahogy a 1.4 alfejezetben látható) hogy a program minél általánosabban felhasználható legyen csak akkor teljesülhet, ha képes a tesztelendő eszköz paramétereit beolvasni, és a kapott adatokkal dolgozni. Ahogy a használt eszközök között a 3.1.6 alfejezetben bemutattam, erre kiválóan alkalmas a JSON struktúra. Ennek a struktúrának a megalkotásakor szem előtt kell tartani, hogy a .json fájlt bárki nyitja is meg, később érthető legyen számára, és ki tudja tölteni a megfelelő adatokkal. A beolvasó programrész lehetőséget biztosít két tárolási módszer használatára. Vagy az összes alkatrészt egy nagy fájlban, JSON könyvtárak listájaként kezeli, Vagy egy "központi" .json fájlban az alkatrészeket egyesével tároló .json fájlok elérési útját tárolja listába rendezve. Ezt a paramétert a "json_type" kulcs alatt lehet állítani. Ha az érték "part_list" akkor csak akkor az alkatrészeket paramétereit tartalmazza a lista, ha "path_list" akkor az elérési utakat találjuk a listában. Tartalmaz az adatfájlunk egy listát arról is, hogy mely alkatrészek egymással való ütközését nem akarjuk vizsgálni. Itt a lista elemei az alkatrészek neveit tartalmazza párokba szedve. Ezek után már csak az alkatrészeket, vagy azok elérési útját tartalmazó lista következik.

Hogy tárolunk egy alkatrészt? Ennek megalkotásához vizsgáljuk meg először, hogy milyen paraméterei vannak egy alkatrésznek:

- Először is elengedhetetlen, hogy be tudjuk azonosítani az alkatrészt, erre szolgáljon a **name** változó
- A test térbeli pozíciójának meghatározására szolgáljon a középpontjába (a testátlók metszéspontjába) mutató helyvektor. Erre vezessük be a **position** változót, ami egy három elemű lista.
- Az alkatrész Bounding box-ának a kiterjedéseit is szűkséges tárolni. Erre használjuk az **extention**, három elemű listát, ami a téglatest három élének hosszát tartalmazza.
- A test helyzetének meghatározására szolgáló harmadik vekort, az elfordulását, vagyis az Euler szögeket tároló három elemű listát nevezzük **rotation** -nek.
- Tárolnunk kell azt az információt is, hogy az adott alkatrész milyen mozgásra képes. A szoftver két mozgás fajtára legyen felkészítve, az egyenes vonalú, és a körmoz-

gásra, rendre **linear_motion** és **rotating_motion** kulcs alatt tároljuk az ehhez szükséges információkat.

- Ha az aktuális alkatrész egy motor, akkor tárolnunk kell az ehhez szükséges adatokat is. Erre szolgáljon egy objektum, amit a **motor** változóban tároljunk.

A lineáris mozgást leíró objektum a következő elemekből épüljön fel:

- A **direction** nevű háromelemű lista a mozgás irányvektorát tartalmazza, ebből, és a test helyzetéből, amit a **position**-ben tárolunk egyértelműen meghatározható a pályája
- Két darab háromelemű listában tárolva a két végpont koordinátáit is tárolnunk kell. A **linear_end_state** nevű változó legyen erre hivatott.

A körmozgás adatainak tárlolására használt objektum szerkezete legyen a következő:

- A **rotating_center_point** a forgás középpontjának koordinátáit tartalmazza egy három elemű listában.
- A **rotation_axis** a forgástengely irányvektora legyen, így a forgás középpontjával meghatározható a forgástengely.
- Az **end_states** változó itt is a lehetséges szélső állapotok meghatározására szolgáljon.

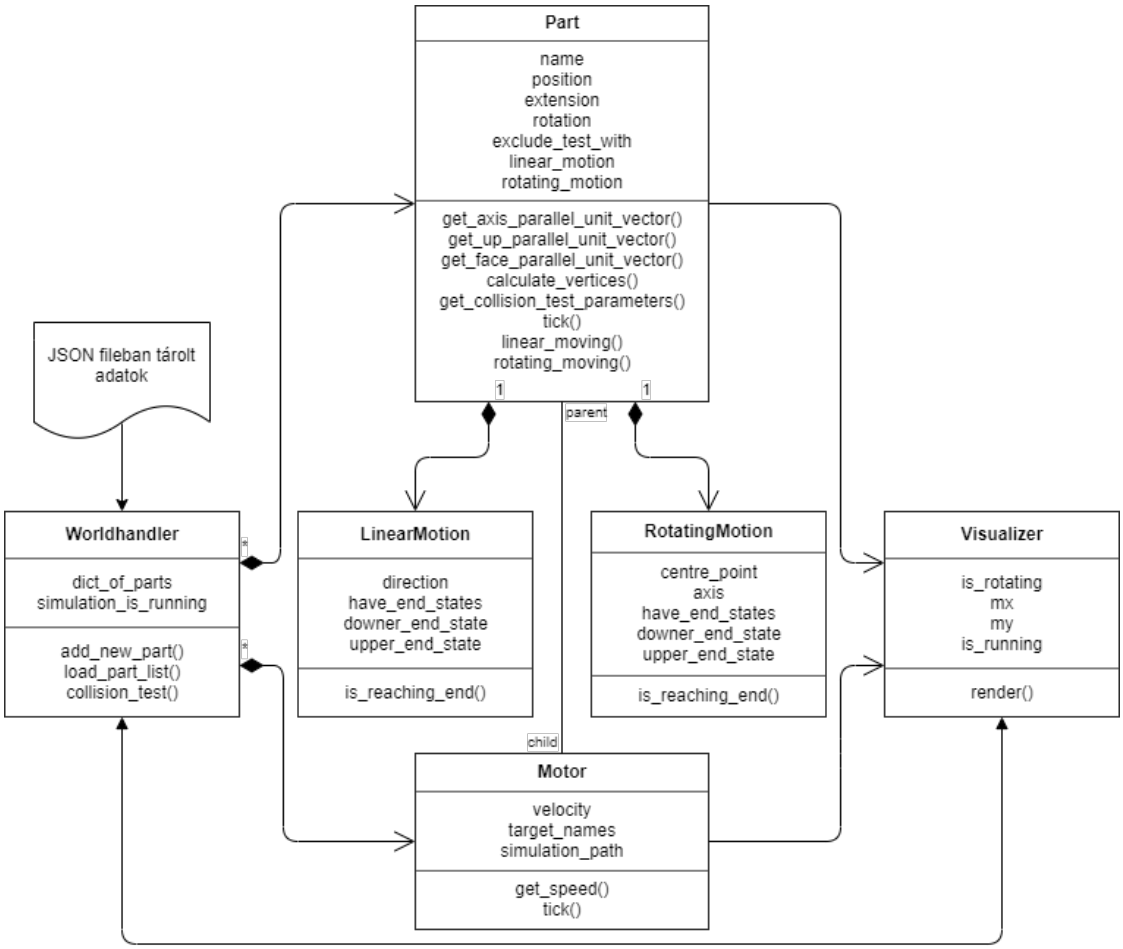
A motor tárolására pedig álljon a kovetkező elemekből:

- A **target_names** kulcshoz tartozó objektumban a kulcsok a mozgatott alkatrészek neveit, az értékek a mozgás fajtáját jelöljék.
- A **simulation_path** kulcs a Tango szerveren való elérési utat tartalmazza.

```
{
  "part": {
    "name": "bed_rotatory",
    "position": [75, 55, 150],
    "extension": [200, 10, 10],
    "rotation": [0, 0, 0],
    "motor": {
      "target_names": {
```

```
      "bed_rotatory": "rotationally",
      "bed": "rotationally"
    },
    "simulation_path": "/petspect/motors/bedrotatory"
  },
  "linear_motion": {
    "direction": [1, 0, 0],
    "linear_end_states": [
      [74, 55, 150],
      [376, 55, 150]
    ]
  },
  "rotating_motion": {
    "rotation_centre_point": [275, 60, 150],
    "rotation_axis": [1, 0, 0]
  }
 }
}
```

1. kódrészlet. Példa egy motor tárolására.



3.3. ábra. Az alkalmazott osztályrendszer ULM diagramja

3.3.2. Az alkalmazott osztályrendszer ismertetése

A 3.3 ábrán látható az osztályrendszer ULM diagrammja. A WorldHandler osztály hivatott a szimuláció kezelésére. Ez tárolja az alkatrészeket objektumba rendezve. Az alkatrészek tárolására a Part osztály hivatott. Ha az alkatrész motor, akkor a Motor osztály tárolja. A Part osztály függvényei számolják ki az ütközésvizsgálathoz szükséges paramétereket. Ha mozgó alkatrész tároluni, a mozgási adatokat a LinearMotion és a RotatingMotion osztályokban tároljuk. A Motor osztály a Part osztály funkcióin felül a mozgatásokért felelős. A Visualizer osztály felelős az adatok megjelenítésért.

Part

A Part osztály segítségével tudjuk eltárolni az egyes alkatrészeket. Függvényei összegyűjtik, kiszámolják és szükség esetén módosítják az alkatrész ütközésvizgsálatához, mozgatásához, megjelenítéséhez szükséges adatokat. Az osztály változóiról a 3.1 táblázatban található bővebb információ, a 3.2 táblázat pedig az osztály függvényeiről szól.

Változók neve	Típusa	Tárolt adat
name	string	Az alkatrész neve.
position	numpy array	Az alkatrész pozíciójának helyvektora.
extention	numpy array	Az alkatrészt tároló Boudary Box oldalhosszai.
rotation	numpy array	Az alaktrész orientációját határozza meg Euler szögekkel.
exclude_test_with	lista	Azon alkatrészek neve, amikkel nincs ütközésvizsgálat.
rotating_motion	RotationMotion	A forgó mozgást leíró adatok.
linear_motion	LinearMotion	Az egyenes vonalú mozgást leírjó adatok.

3.1. táblázat. Part osztály változói.

Függvények	Funkciója
get_axis_parallel_unit_vector()	Kiszámolja az axis-szal párhuzamos egységvektort.
get_up_parallel_unit_vector()	Kiszámolja az up-pal párhuzamos egységvektort.
get_face_ parallel_unit_vector()	Kiszámolja az face-szel párhuzamos egységvektort.
calculate_vertices()	Kiszámolja és listába rendezi a Bounding Box szögeinek aktuális koordinátáját.
get_collision_test_ parameters()	Kiszámolja, és objektumba rendezi az ütközésvizsgálazhoz szükséges paramétereket.
tick()	Meghívja a lefuttatandó végállapot elérés teszteket.
linear_moving()	A bementben megkapott sebességgel elmozdítja az alkatrész pozícióját a tárolt irányba.
rotating_moving()	A bemenetben megkapott sebességgel elforgatja az alkatrészt a tárolt adatok szerint.

3.2. táblázat. Part osztály függvényei.

Motor

A Part osztály leszármazottja a Motor, ebben tároljuk a motor aktuális sebességét, a motor által mozgatott alkatrészek nevét és a Tango device elérési útját. Az osztálynak a *get_speed()* függvénye kéri be a Tango device-tól a sebesség adatokat egy "connect.py" nevű Python fájlban található *request()* függvény segítségével. A működés szempontjából különösen fontos megoldás itt is a *tick()* függvényben van. Elősziris örökli az ős osztály *tick()* függvényében foglaltakat, meghívja a a *get_speed()* függvényt, és az összes mozgatott alkatrész aktuális megfelelő mozgásfüggvényét a motor aktuális sebességével. A változókról bővebben a 3.3 táblázatban, a függvényekről a 3.4 táblázatban olvashat.

Változók neve	Típusa	Tárolt adat
velocity	number	A motor aktuális sebességét tárolja.
target_names	object	A mozgás típusát és a mozgatott alkatrészek nevét tárolja kulcs - érték párokba rendezve.
simulation_path	string	A szimulált motor elérési útját tartalmazza.

3.3. táblázat. Morot osztály változói.

Függvények	Funkciója
get_speed()	Meghívja a REST API kommunikációért felelős függvényt, így bekéri vele az aktuális sebességet.
tick()	Örökli a Part *tick()* függvényét és meghívja az általa mozgatott alkatrészek megfelelő mozgásfüggvényeit.

3.4. táblázat. Motor osztály függvényei.

WorldHandler

Az eltárolt alkatrészeket és motorokat mind tudnia kell kezelni a programnak. Ez a WorldHandler osztály segítségével történik. Az osztály egy objektumként tárolja az alkatrészek adatait a saját nevükhöz rendelve. A WordHandler osztály a futás során egyszer példányosodik a *main()* függvényben. A **dict_of_parts** könyvtáron kívül a másik változója a **simulation_is_running**, ami igaz értékkel generálódik. Ez az a változó aminek az igazságtartalmát *main()* -ben futó "while" ciklus ellenőrzi, ha ezt hamisra állítjuk, akkor a szimuláció véget ér. Bővebb információért a változókról keresse a 3.5 táblázatot, a föggvényekről a 3.6 táblázatot.

Változók neve	Típusa	Tárolt adat
dict_of_parts	object	Tárolja az összes alkatrészt. A kulcs az alkatrész neve, az érték pedig maga az alkatrészt tároló objektum.
simulation_is_running	boolean	Tárolja, hogy éppen fut-e a szimuláció. A *main()* függvény csak akkor hívja meg a *tick()* függvényt, ha az értéke igaz.

3.5. táblázat. WorldHandler osztály változói.

Függvények	Funkciója
load_part_list()	Felelős az alkatrészeket tartalmazó JSON file betöltéséért. Ahogy már korábban a 3.3.1 -es részben említettem, képes közvetlenül, és az elérési utak listája alapján is betölteni az adatokat. Az alkatrészek beolvasását a könnyebb olvashatóság, és az átláthatóság érdekében kiszerveztem az *add_new_part()* függvénybe.
add_new_part()	Betölti a JSON -ből beimportált már objektumban tárolt adatokat a megfelelő (Motor vagy Part) osztályba. Tudnia kell kezelni, hogyha bizonyos, nem kötelező változók kulcsait nem tartalmazza a bemeneti objektum.
tick()	Rendre meghívott függvény, ami az alkatrészek *tick()* függvényeit és a *collision_test()* függvényt hívja meg. Ez a függvény felelős az időben folyamatos működésért.
collision_test()	A vizsgálandó párosításokra meghívja az ütközésvizsgálatot végző függvényt, és a kapott eredményeket kezeli. Ha ütközés történik, akkor leáll a szimuláció és a konzolon megjelenik egy üzenet arról, hogy mely alkatrészek ütköztek össze.

3.6. táblázat. WorldHandler osztály függvényei.

Visualizer

A vizualizációhoz használt függvényeket PyGame és OpenGL modulokat egységbezártan, csak a Visualizer osztályban és a csak innen meghívódó statikus függvényekben szerettem volna használni. Így a Visualizer osztáy kezel minden háromdimenzós megjelenítéshez kapcsolódó feladatot. Az osztály egyetlen függvénye a *render()* (megalkotásához nagy segítség volt a [20] hivatkozás alatt található tanfolyam), ez a WordlHandler osztály *tick()* függvénye mellett a másik olyan függvény, ami a *main()* "while" ciklusából hívódik, és így újra és újra frissülni tud a megjelenített kép és benne minden alkatrész helyzete, képe. A render függvényen belül a PyGame eszköztárának köszönhetően értelmezni tudja a szoftver az egérrel történő elforgatásokat, a görgővel történő nagyítást és kicsinyítést, a nyilakkal, vagy az A, W, S, D gombokkal való mozgatást, a P gombbal a szimuláció szüneteltetését. Itt fontos megjegyezni, hogy a vezérlő szoftver, vagy a későbbiekben bemutatásra kerülő tesztszoftver, amit csináltam, a szimulációtól függetlenül fut, így arra nincs hatással a szimuláció megállítása. Az Escape billentyűvel ki tudunk lépni a szimulációból. A *render()* függvény bemenetéül kapja a WorldHandler osztályú szimuláció változót. Az alkatrészekre meghívja a *calculate_ vertecies()* függvényt, ami az alkatrészek csúcsainak koordinátáival tér vissza. Ezeket adja át a *shapes_ to_ display.bounding_ box()* függvénynek, ami kirajzolja az alakzatot.

Változók neve	Típusa	Tárolt adat
mx	number	Az egér x tengely szerinti pozícióját tárolja.
my	number	Az egér y tengely szerinti pozícióját tárolja.
is_running	boolean	Tárolja, hogy éppen fut-e rogram, ha az értéke hamis, a *main()* kilép a while ciklusból, és a program végetér.

3.7. táblázat. Visualizer osztály változói.

3.3.3. LinearMotion

A lineáris mozgáshoz szükséges adatok tárolására szolgáló osztály. Bővebben a 3.9 táblázatban. Az egyetlen függvénye a végállapotok elérését hivatott vizsgálni, ez a függvény az alkatrészek *tick()* függvényéből hívódik.

Változók neve	Típusa	Tárolt adat
direction	numpy.array	Az egyenes vonalú mozgás irányvektorát tartalmazza.
have_end_states	boolean	Azt az információt tárolja, hogy van-e meghatározva végállapota az alkatrésznek.
downer_end_state	list	Az alsó végállapot koordinátáit tárolja.
upper_end_state	list	A felső végállapot koordinátáit tárolja.

3.8. táblázat. LinearMotion osztály változói.

3.3.4. RotatingMotion

A forgó mozgással kapcsolatos információk tárolására szolgáló osztály. Egyetlen függvénye az elfordulás végállapotainak elérését teszteli.

Változók neve	Típusa	Tárolt adat
centre_point	numpy.array	A forgómozgás középpontjának helyvektorát tartalmazza.
axis	numpy.array	A forgás tengely irányvaktorát tartalmazza.
have_end_states	boolean	Azt az információt tárolja, hogy van-e meghatározva végállapota az alkatrésznek.
downer_end_state	list	Az alsó végállapot Euler szögeit tartalmazza.
upper_end_state	list	A felső végállapot Euler szögeit tartalmazza.

3.9. táblázat. RotatingMotion osztály változói.

3.3.5. Egyéb, nem osztályhoz tartozó függvények

A 3.10 fáblázatban összeszedtem azokat a föggvényeket, amin nincsenek osztályba rendezve, mégis fontos szerepet játszanak a program működésében.

Függvények	Funkciója
collision_test.add_axes()	Az ütközésvizsgálathoz megkapott paraméterekből kiszámolja a lehetséges elválasztó tengelyek egységvektorait, és egy listába rendezi őket.
collision_test.are_collision()	Elvégszi az ütökzésvizsgálatot a 2.3.4 alfejezetben leírtak alapján.
vector_transformation.rotate()	A bementben kapott irányvektort a kapott Eluer szögek szerint elforgatja.
vector_transformation.shift()	A bementben kapott irányvektort a kapott eltolási vektorral összegzi, ezzel megkapva az eltolás képvektorát.
shapes_to_display.arrows()	A vizuaálizóció során használt koordináta jelölő vektorokat rajzolja ki.
shapes_to_display.Bounding_box()	A bemenetben megkapott csúcsok alapján a Bounding Box-ok kirajzolásáért felelős.
connect.request()	A REST API kapcsolatot valósítja meg a Tango device-okkal.
main.main()	Itt példányosodik a WorldHandler és a Visualizer, és egy, a program teljes futásideje alatt futó while ciklus folyamatosan hívja meg a WordHandler *tick()* függvényét, valamint a Visualizer *renred()* függvényét.

3.10. táblázat. Osztálykhoz nem köthető függvények.

3.4. A motorvezérlő modellezése - unit test

Ahhoz hogy tesztelni tudjam az elkészült szoftvert szükséges volt, hogy legyen egy tesztkörnyezet hozzá. Elkészítettem egy modelljét a motorvezérlőnek. Ugyebár a motorokat amiket a fizikai berendezésben megtalálhatnánk Tango device-ok hivatottak reprezentán. Ezért a deviceokat lehetőség van REST API protokollon keresztül irányítani. Így a teszt-

szoftver képes beolvasni egy JSON fájlt ami tartalmazza a használandó motor adatait, és a működési szekvenciát, amit végigjár, mondhatni a motor saját programját. Később, egy éles teszt során a motorvezérlő által a Tango device-ok valós időben vezérelhetők lehetnek, az ütközés vizsgáló rendszer ettől teljesen független, de a teszteléshez a jelenlegi fázisban elég ha csak a teszt futtatása előtt tudjuk megadni a motorok programját.

Ez a unit test program nagyon hasonló felépítésű, mint az ütközésvizsgáló. Itt is létezik Motor és WorldHandler osztály. Az előbbi a betöltött motorokat tárolja listába rendezve, az utóbbiban pedig egy-egy motor adatait tároljuk. A *main()* függvényben futó while ciklus ugyanúgy hívja a unit test-hez tartozó WorldHandler *tick()* függvényét, ami a Motor osztály *tick()* függvényét hívja meg. A *Motor.tick()* megkapja az aktuális időt, és az alapján figyeli, hogy hol tart az aktuális motor a programjában. Ha szükséges megváltoztatja a *Motor.speed* változót, ezzel a tárolt sebességét. A Motor speed változójának a setter-jében nem csak a változó beállítása történik, hanem egy PUS request is kimegy a motor felé. Így valósul hát meg a unit test, vagyis a motork programszerű működtetetése.

3.5. A Tango eszközök beállításának módja

Egy Tango eszköz beállítása során vannak olyan folyamatok, amik elvégzéséhez a Tango ad segítséget (például ilyen az eszköz osztály generálása), és vanak olyan folyamatok amik beállítását kód szinten tudjuk megtenni. Most lépésről lépésre bemutatom, hogyan állítottam be az általam használt Tango eszközöket.

3.5.1. TANGO Code Generator

A *Pogo* nevű ikon kiválasztásával elindul a *TANGO Code Generator* nevű program. Egy új osztály kódjának generálásához a *file/New* menüponton jutunk el. Valós email címet érdemes megadni, az osztáy családot *AcceleratorComponents*-re állítani, Bus *Not Applicabel* legyen. Mivel a projekt alatt Python nyelvet használunk, azt kellett még beállítani illetve megadni az osztály nevét. Erre példát a mellékletben, az A.1 abrán látni.

Ezek után megadhatjuk milyen parancsokat (függvényeket), milyen skalár, spektrum, kép, vagy tobábbított attribútumot szeretnék generálni az osztályunkba. A jobb egérgombbal a parancsokra vagy az attributumokra kattintva adhatunk meg új változókat és függvényeket. A motor reprezentálásához es *Speed* nevű változót állítottam be, valamint

az *On*, *Off* állapotokat. Hozzáadtam még az *On*, *Off*, *SetSpeed* függvényeket, amik rendre a kikapcsolásért, bekapcsolásért és a sebesség érték beállításáért felelősek. Az A.2 ábra mutatja be ezt. Így már generálható az osztály.

Ezek után a generált kódba implementálnunk kell a kívánt logikát. Esetünkben ez a *SetSpeed* parancs beállítását jelentette, vagyis hogy a kapott bemeneti paramétert állítsa be a Speed skalár változó értékeként.

3.5.2. A szerver beállítása

Ha elkészült a kód, a *Jive* alkalmazáson keresztül lehet a *Server Wizard* segítségével új szervert beállítani, a szever nevének és a példányának megadása által. Erről az A.3 ábrán látni felvételt. Ezután futtatnunk kell a generált és kiegészített eszközosztály tartalmazó kódot, megadva az *instance* nevét. Ha ez megtörtént a telepítő varázsló következő oldalán kiválaszthatjuk a beállított osztályt, majd deklarálhatjuk ez eszközöket. Az itt beállított eszköz névnek tartalmaznia kell két felsőbb szintet, ami meghatározza az eszköz elérési útját. Az A.4 ábrán látható ez az ablak. A beállítások után a Motor.py futása leáll, újra kell indítani. Ha ez megtörténik, és a tangobox-web docker-t elindítottuk, akkor már képesek leszünk elérni az eszközt REST API hívásokon keresztül.

4. fejezet

Eredmények

4.1. A porgram funkcióinak bemutatása a célok függvényében

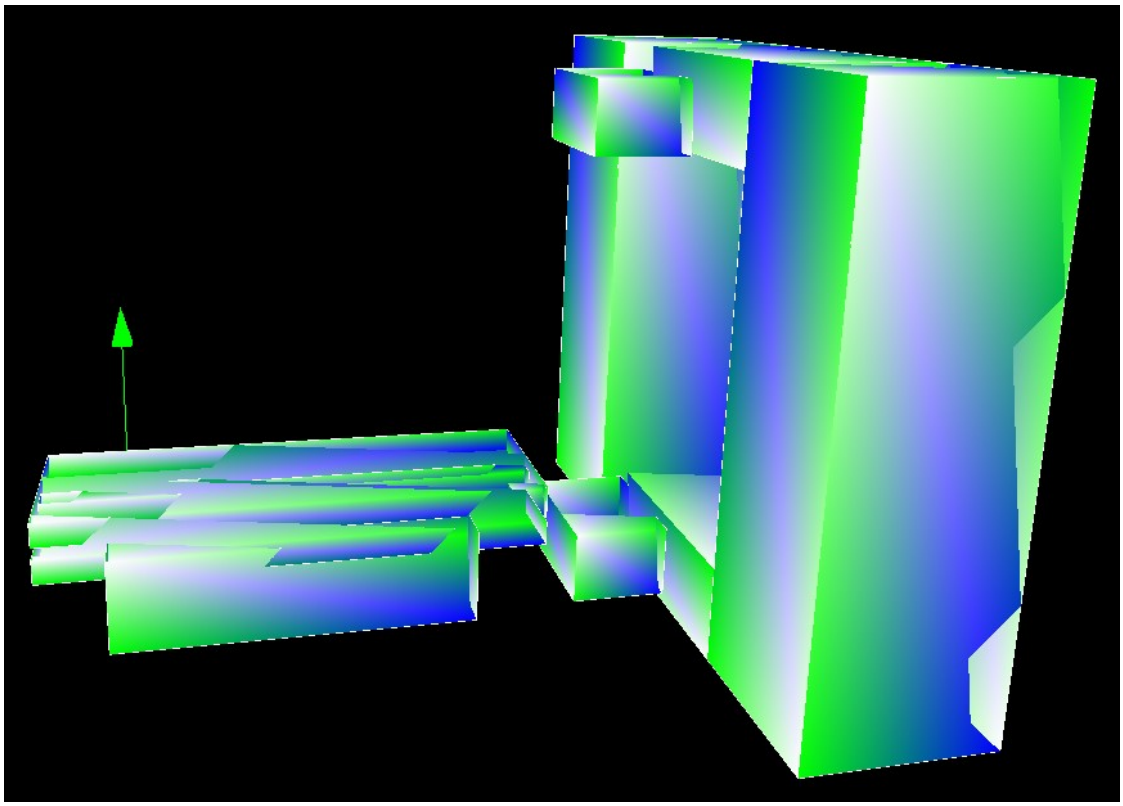
A következő fejezetben egy példával szemléltetve be fogom mutatni, hogyan tesz eleget az 1.4 alfejezetben felállított céloknak a szoftver.

4.1.1. Adatok tárolása és beolvasása a gyakorlatban

A 3.3.1 fejezetben bemutattam milyen JSON struktúrát alakítottam ki és ebben hogyan tároljuk az alkatrészek adatait. Az adatok sikeres beolvasását, helyes betöltődését, a megfelelő módon való tárolását jól bizonyítja, a várt háromdimenziós kép megjelenése. Összeállítottam egy teszt JSON fájlt amiben egy SPECT - CT képalkotó sematizált Bounding Box modelljét adtam meg. A dokumentum *part_list* típusú, vagyis maga a fájl tartalmazza az alkatrészek leírását is, és nem az elérési utak vanak megadva. Szerepelnek "kivétel a tesztelés alól" paraméterben az egymáshoz szerelt alkatrészek nevei, kapcsolódás szerint párba, majd listába rendezve. Ezután tartalmazza a fájl egyes alkatrészeket leíró paramétereket. Az alkatrészek beállításait az A.1, A.2, A.3 és az A.4 táblázatokban bővebben bemutatom, vagy megtekinthető a csatolt kód spect_ct.json nevű fájljában.

4.1.2. Kirajzolás

Most hogy láttuk, az adatok betöltése sikeresen megtörtént, vizsgáljuk meg, hogy a kirajzolás működik-e. Téglatesteket szeretnénk látni háromdimenziós szimulátorként, amiben a nyilakkal az A,W, S, D billentyűkkel és az egérrel is tudunk manipulálni. A szimulációt szüneteltetni tudjuk a P billentyű leütésével, és az Esc billentyűvel ki tudunk lépni. A teszt alkatrészeket úgy állítottam össze, hogy egy felettébb leegyszerűsített



4.1. ábra. A vizualizáció eredménye. Látható hogy megjelennek az alkatrészek.

SPECT - CT -t mintázzon, vagyis felismerhetőnek kell lennie a vizualizáció után. Az eredmény a 4.1 ábrán látható. Bár látható, hogy a téglatestek kirajzolásra kerülnek, az ábrán láthatóak renderelési hibák. Ez egy ismert probléma, amit z-fighting -nak neveznek. Ennek a kijavítása a későbbiekben nagyban növelhetné az esztétikai élményt, de a program alapvető célját - vagyis hogy ütközéseket vizsgáljon - nem hátráltatja.

4.1.3. A motorok mozgásának megadása

A 3.4 fejezetben olvasható, hogyan is működik a motorok irányítása. Beállítottam egy működési szekvenciát minden motornak, ami a következő mozgásokat tartalmazza:

1. A motorok bekapcsolnak, és mindegyik felé kimegy egy *init* hívás, így minden motor sebessége biztosan 0 a szofver indulásakor.

2. 40 másodperc után az ágyat mintázó alkatrész elindul a mérő terület irányába, 20 másodpercen keresztül halad.

3. a 20 másodperc után 100 másodpercen keresztül nem mozdul. Ha letelik ez az idő, de más motor programja még fut, akkor 0 sebesség értékkel "várja" meg hogy a többi motor is "végezzen" majd az össze motor leáll.

4. Közben indulástól számított 60 másodperc után elindul az ágy forgató motorja és a két SPECT fejegység forgató motorja is.

5. Az ágy forgató motorja 2 másodmpercig üzemel majd leáll, a forgató motorok szinkron pályán mozognak 50 másodpercig.

4.1.4. Végállapot vizsgálat

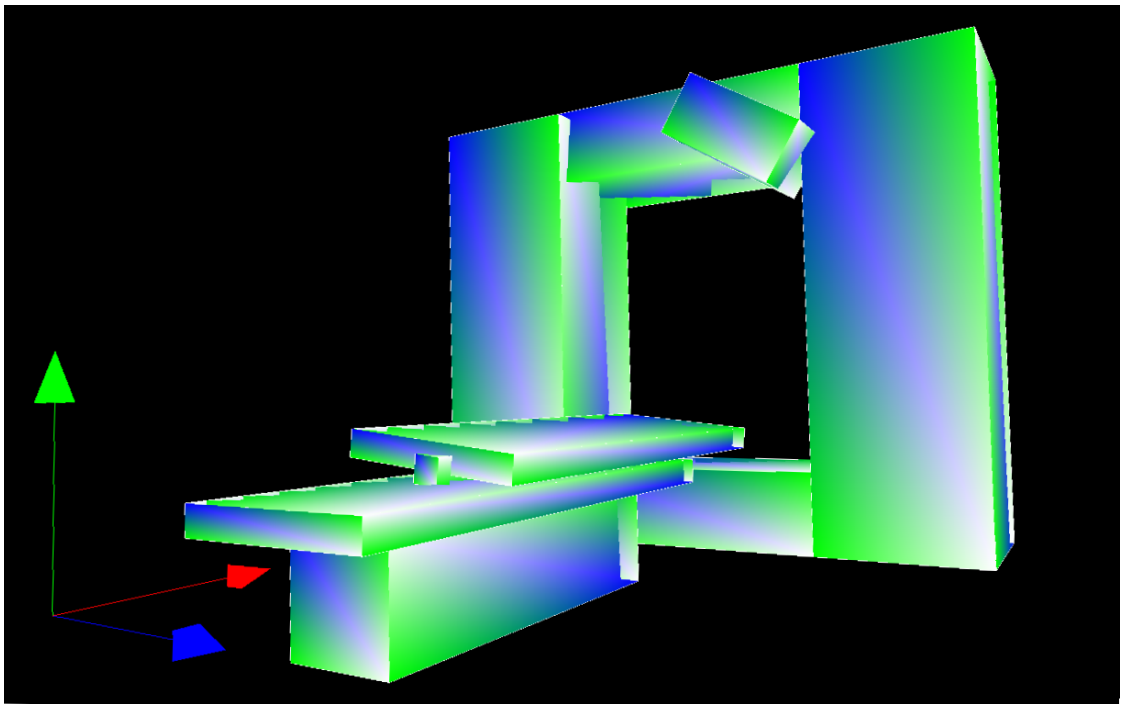
A A.2 és a A.3 táblázatokban látható, mely alkatrészekhez van végállapot megadva. Ha ezekkel a beállításokkal elindítjuk a tesztet, hamar elérkezünk addig a pontig, ahol az asztal elfordulása eléri a felső végállapotot. Ilyenkor az elvárt működés, hogy a *bed* alkatrész *tick()* függvényében található elfordulási végállapot vizsgáló kódrészlet alapján kiírdójon a konzolra a *"bed reach its rotating end state!!!"* felirat, a vizualizáción láthatjuk a 0,1 piradián mértékű elfordulást, erről a képmetszet a 4.2 ábrán látható. Ilyenkor a 4.1.4 üzenet jelenik meg a konzolon.

```
"bed reach its rotating end state!!!"
```

2. kódrészlet. Az ütközésre figyelmeztető üzenet a konzolon

4.1.5. Ütközés vizsgálat

Az előző fejezetben részletezett mozgásszekvenciát folytatva feltételezzük, hogy az egymással szemben forgó SPECT fejegységek forgásirányát szeretnék megváltoztatni valami miatt. Tegyük fel, hogy valamilyen oknál fogva bekerül egy hiba a rendszerbe, és csak az egyik motor sebeségét állítjuk a mínusz egyszeresére. A másik SPECT fejegységhez



4.2. ábra. A *bed* alkatrész szélső értékű elfordulása látható.

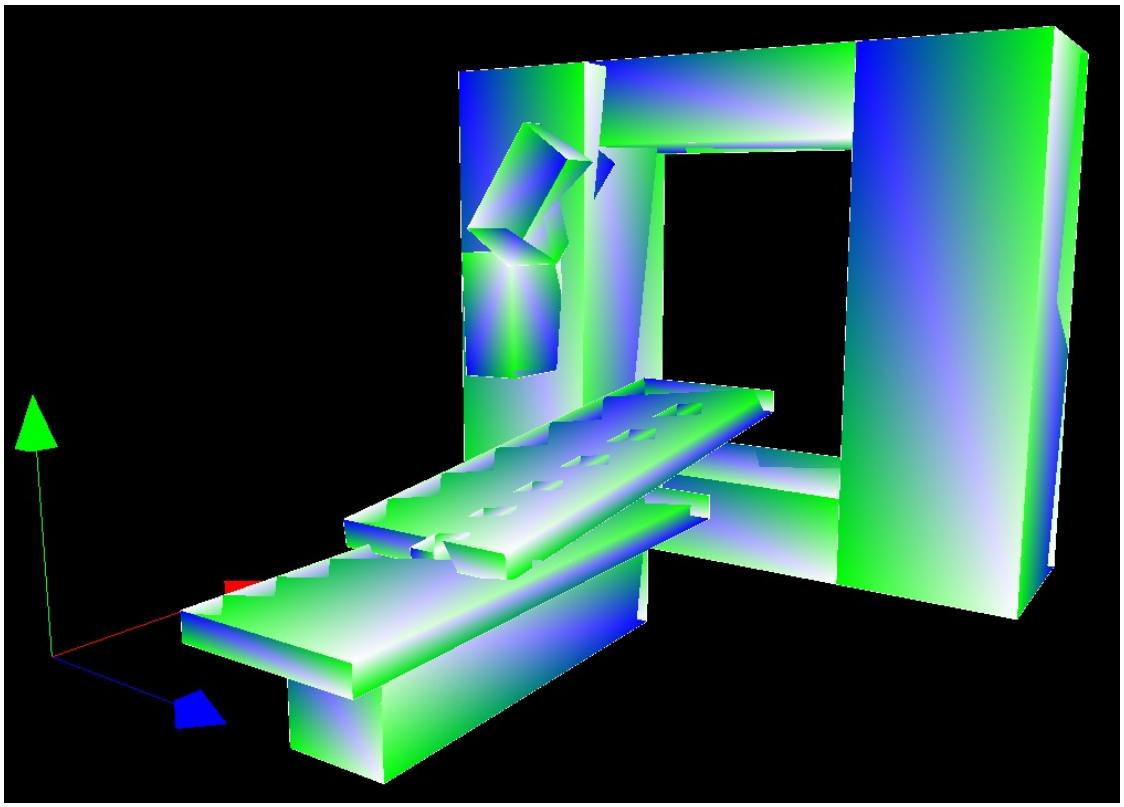
tartozó motor sebessége változatlan marad. Hogy eljusson a teszt eddig a pontig, ahhoz változtatni kell az ágy mozgási beállításain. Az így beállított program látható a 4.1 táblázatban.

bed_slider		bed_rotatory		spect_rotatory_1		spect_rotatory_2	
sebesség	időtartam	sebesség	időtartam	sebesség	időtartam	sebesség	időtartam
0	40	0	60	0	60	0	60
0.5	20	0.01	2	0.05	50	0.05	50
0	100	0	100	-0.05	50	0.05	50
-	-	-	-	0	100	0	100

4.1. táblázat. Az egyes motorok mozgásai: látható, hogy milyen sebességgel mozognak, és mennyi időn keresztül teszik ezt.

Ezekkel a beállításokkal a két fejegység ütközése várható. A program elindul, megérkeznek a sebesség adatok, és a motorok a vártnak megfelelően mozgatják az alkatrészeket. Az ütközésről készült felvétel a 4.3 ábrán látható. Ekkor az elvártaknak megfelelően megjelenik a 4.1.5 felirat a konzolon.

```
"The name = spect_1, and the name = spect_2 parts will collides
```



4.3. ábra. A SPECT fejegységek ütközése látható a képen.

```
each other!!!"
```

3. kódrészlet. Az ütközésre figyelmeztető üzenet a konzolon

4.2. A program megírásának igazi eredménye

Azt, hogy a valóságos, fizikai képalkotó berendezés milyen sebességgel mozoghat rengeteg minden befolyásolja. Egy valós alkatrész rendelkezik tömeggel, számolni kell azzal, hogy az alkatrészek minél gyorsabban mozognak annál nagyobb a mozgási energiájuk, és ezt az energiát kordában kell tartani, meg kell tudni fékezni. A mozgatás sebességét úgy kell megválasztani, hogy az biztonságos legyen mind az emberekre, mind a gép alkatrészeire nézve. Ez egy korlátozó tényező. De befolyásolja a mogás lehetséges sebességét a motorok teljesítménye is. Az alkatrészek emelési sebességének, bármilyen irányú gyorsításának lehetséges mértékét korlátozza a használt motorok teljesítménye. Körmozgás esetén a legapróbb kiegyensúlyozatlanság miatt is felléphet rázkódás a berendezésben. Minél nagyobb a sebesség és a forgatott test tömegközéppontjának a forgásközépponttól való távolsága a rezgés annál nagyobb lehet. Ez kellemetlenséget okozhat a vizsgált páciensnek, az eszköz szerkezetében pedig komoly károkat tehet.

Ezek a korlátozó tényezők egy szimulált környezetben, ahol a cél az esetleges ütközések kiszűrése, nem állnak fent. A mozgási sebességet és annak változásait nem befolyásolja az energiamegmaradás törvénye, nincsenek biztonsági előírások. A szimulált motorok akkora sebességgel, és pontban akkor mozgatják az alkatrészeket, ahogy beállításra kerültek.

Ezek alapján az ütközés vizsgáló szimuláció segítségével végzett teszt lényegesen felgyorsítható: egyszerűen a mozgási sebességek felszorzásával. Ahányszor gyorsabban mozognak az alkatrészek annyiad ideig tart a szimuláió. Ennek a gyosításnak csak a számítógép paraméterei szabnak határt. Vagyis az elkészült tesztszoftver eredménye, hogy a segítségével lényegesen alaposabb tesztelést lehet végezni adott idő alatt.

4.3. Továbbfejlesztési lehetőségek

4.3.1. A program általam felfedezett hibáinak javítása

Mint a 1.3 fejezetben bemutattam nem létezik hibátlan porgram, de a hibák ismerete segít, és lehetőséget ad a kijavításukra. Az általam készített szoftver betölti a funkcióját, de ebben is vannak olyan hiányosságok, amik kijavítása nagyban javítaná a felhasználói élményt, újabb funkciókat tenne elérhetővé.

A legfontosabb implementálandó részlet az adapter időfüggővé tétele. Jelenleg egy *tick* hosszát, amit a program futásánál úgy használunk mint órajelet, az ezalatt a program által elvégzett feladatok hossza határozza meg. Ez az érték eltérő lehet, az eltelt idő nem lesz egyforma a két tick között, mégis jelenleg ez az időmérés alapja a szoftverben. A *unit test* programot már úgy terveztem, hogy idő függő legyen, ennek a megoldásait lehetne implementálni az adapterbe is, elvégre nagyon hasonló architektúrájuk van.

Szintén előrelépés lenne, ha szabadon választott tengely körüli forgást is értelmezni tudnánk, nem csak a koordináta tengelyek körülit. A 2.3.3 alfejezetben bemutatott módszerrel lehetőség lenne erre.

Fontos lenne a háromdimenziós renderelés szebbé tétele. Jelenleg egy *z-fhiting* nevű porbléma áll fent. Ez egy elég jól ismert hibajelenség, így megoldási javaslatokat is lehet rá találni. A problémát az olyan felületek vagy élek okozzák, amelyek térbeli helyzete megegyezik. Ilyenkor az alakzatok egy képponttal való eltolással különválaszthatók. Igy

már a mélységi pufferelés meg tudná határozni, hogy melyik alakzat van előtérben, és azt jelenítené meg.

A kód átnézése során lehet találni olyan megoldásokat, amik sértik vagy nem használták ki az objektum orientált programozás alapelveit. Az osztályrendszerek bizonyos változóinak és függvényeinek priváttá tétele sokban hozzájárulna a kód fenttarthatóságához.

4.3.2. A fejlesztés lehetséges következő lépései

Jelenleg az alkatrészeket Bounding Boxok reprezentálják a programban. Mivel a legtöbb alkatrész alakja téglatesthez közelít, vagy jól reprezentálható több kisebb téglatest együttes értelmezésével a program megfelelő pontosságú becslést ad az ütközésekről. A tesztelés során kaphatunk hamis pozitív eredmény ezzel a módszerrel, vagyis jelezhet olyan ütközést a Bounding Box-ok között, ami a valóságban nem fordulna elő az eszközök alakja miatt. Viszont, és ez a fontos, hamis negatív eredményt - vagyis hogy egy ütközést nem sikerült detektálni, pedig a valóságban megtörtént volna - nem segít elő ez a módszer. Ha később szükség lenne egymáshoz nagyon közel mozgó, vagy a téglatesttől nagyon eltérő alakú alkatrészek reprezentálására is, akkor a Complex collision teszt jól implementálható. Ezzel a módszerrel a program az ütköző Bounding Box-ok esetén a valós formákat használva képes lenne megvizsgálni, hogy valóban összeére a két alkatrész.

A valódi, fizikai orvosi képalkotó berendezésekben a vezérlő egység munkáját érzékelők segítik. Vannak érzékelők a mozgó alkatrészek kezdeti és végállapotában, vannak szerelő ajtó nyitást jelző szenzorok, vannak amik a vizsgálati alany pozícióját figyelik. Ahhoz, hogy a program a jövőben egy valós vezérlőegységre kötve, valós időben legyen képes teszteket végezni, ezeket a szenzorokat is szimulálni kell, meg kell határozni egy olyan felületet, amin ezeknek a szenzoroknak a jelét manipulálni lehet.

5. fejezet

Összefoglalás

5.1. Feladat

Munkám során lehetőségem nyílt csatlakozni egy olyan kutató csoport munkájába, aminek célja, hogy az orvosi képalkotó berendezések fejlesztését minnél hatákonyabbá, fenttarthatóbbá tegye. Próbálják feltárni ezeknek, a modern orvostudomány számára nélkülözhetetlen gépeknek a megvalósítása közben adódó, nem túl hatékony, vagy rosszul kezelt részfolyamatait, és ezekre jobb, gyorsabb, egyszerűbb megoldást találni. Ezzel segítve közelebb a tudományt a jövő orvosi biotechnológiájához.

A feladat, amivel kapcsolatban becsatlakozhattam ebbe a munkába, hogy rövidítsük a motorok által mozgatott berendezések tesztelési idejét, csökkentsük a teszteléshez szükséges eszközök számát, és növeljük a folyamat hatékonyságát. Ahogy bemutattam, egy valós vezérlő egységhez kapcsolt szimulált orvosi képalkotó berendezés jeletős előnyökkel jár a tesztelési idő, a befektetett erőforrás, a biztonság, a mobilitás, és a változtathatóság területén is.

Feladatom az volt, hogy egy ilyen szimulációs teszt szoftver megalkotásának kezdeti fázisába kapcsolódjak be. Az első ütem, vagyis az architektúra megtervezése, a használt eszközök kiválasztása, megismerése rendkívül érdekes és tanulságos volt számomra.

A 4 fejezetbem bemutattam, hogy a porgrammal kapcsolatos kitűzött célokat hogyan sikerült elérnünk. Sikeresen megalkottunk egy szoftvert, ami valós időben jeleníti meg egy háromdimenziós vizualizációban a Bounding Box-ok által reprezentált alkatrészek helyzetét. A szimulált motorok működését figyeli, és az általuk előidézett mozgásokat megjeleníti a vizualizációban. Folyamatosan teszteli az alkatrészek egymáshoz viszonyí-

tott helyzetét, ha tiltott állapotra kerül sor - vagyis ha két alkatrész ütközik, vagy kilép abból a mozgási zónából, amit meghatároztunk neki - ez esetben leáll a szimuláció és jelzi a problémát.

5.2. Eredmények

Ennek a programnak köszönhetően jelentősen rövidül majd a tesztelésre fordítandó idő, az orovosi képalkotók alaposabb átvizsgálására lesz lehetőség, és ez reményeink szerint gyorsabb fejlődést eredményezhet majd .

5.3. Jövőbeli tervek

Ha lehetőségem nyílik rá örömmel folytatnám a munkát ennek a programnak a fejlesztésén. Akár hogyan is alakul a jövőm, elmondhatatlanul jó élmény volt egy valódi orvostechnológiához kötődő projektben részt vennem az elmúlt évek Bionikai tanulmnyai után. Kifejezetten örülök neki, hogy a dolgozat kapcsán elkezdtem ismerkedni a Python programozási nyelvvel és az objektumorientált programozással. Nekem ez egy abszolút új élmény volt, és az ezirányú tanulmányaimat mindenképpen folytatni szeretném, azt gondolom, ez elengedhetetlen a szakmai fejlődésemhez.

Irodalom

[1] J. Szentágothai és M. Réthelyi, *Funkcionális anatómia I.* Medicina, 2006, 1. fejezet –Az ember saját testérol alkotott képzetei és történeti fejlodésük.

[2] O. Glasser, *Dr. W. C. Roentgen*, 2. kiad. Thomas, 1958, 36. old.

[3] 2020. dec. cím: https://api.allhospital.info/storage/images/5cda50b2da775.jpeg.

[4] 2020. dec. cím: https://www.atlasobscura.com/articles/roentgen-xrays-discovery-radiographs.

[5] 2020. dec. cím: https://www.altair.com/inspire/.

[6] 2020. dec. cím: https://hu.wikipedia.org/wiki/CAN-busz.

[7] 2020. dec. cím: https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection.

[8] P. Vankó, *Kísérleti fizika 1.* TÁMOP-4.1.2 A1 és a TÁMOP-4.1.2 A2 könyvei, 2014, 6 Merev testek mozgása.

[9] 2020. dec. cím: https://gyires.inf.unideb.hu/KMITT/d03/ch05s06.html.

[10] 2020. dec. cím: https://gyires.inf.unideb.hu/KMITT/d03/ch05s07.html.

[11] 2020. cím: https://www.jkh.me/files/tutorials/Separating%20Axis%20Theorem%20for%20Oriented%20Bounding%20Boxes.pdf.

[12] 2020. dec. cím: https://tango-controls.readthedocs.io/en/latest/.

[13] 2020. cím: https://pythonprogramming.net/python-fundamental-tutorials/.

[14] 2020. dec. cím: https://opengl.en.softonic.com/?ex=BB-1549.0.

[15] 2020. dec. cím: https://www.pygame.org/docs/.

[16] 2020. dec. cím: https://numpy.org/doc/stable/.

[17] 2020. dec. cím: https://www.json.org/json-en.html.

[18] 2020. dec. cím: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.

[19] 2020. dec. cím: https://hu.wikipedia.org/wiki/REST.

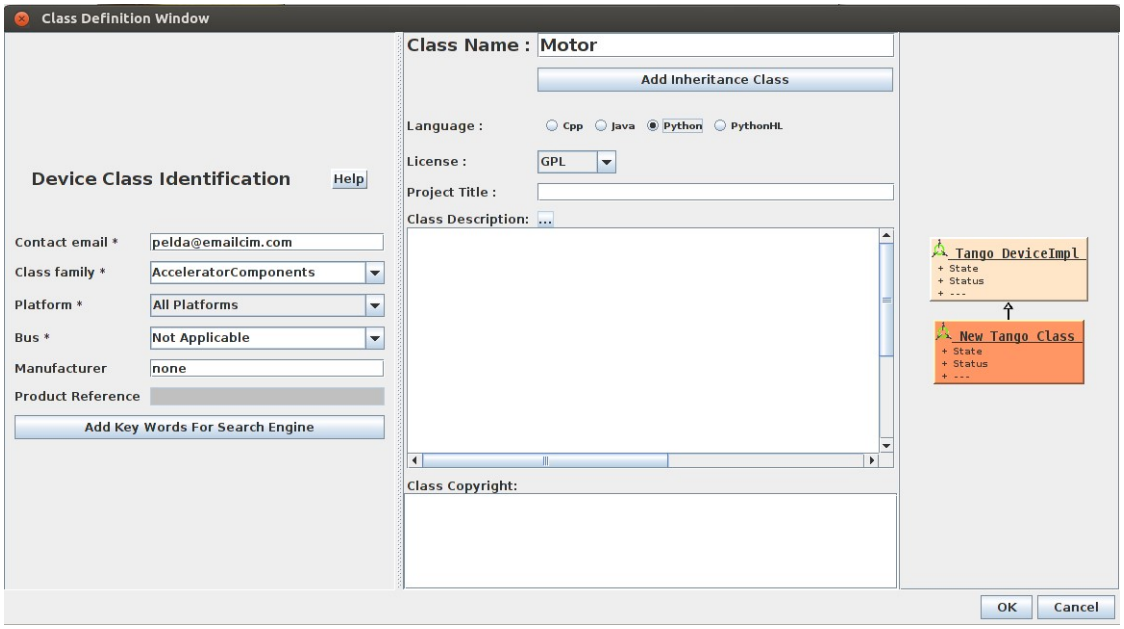
[20] 2020. dec. cím: https://pythonprogramming.net/opengl-rotating-cube-example-pyopengl-tutorial/.

A. függelék

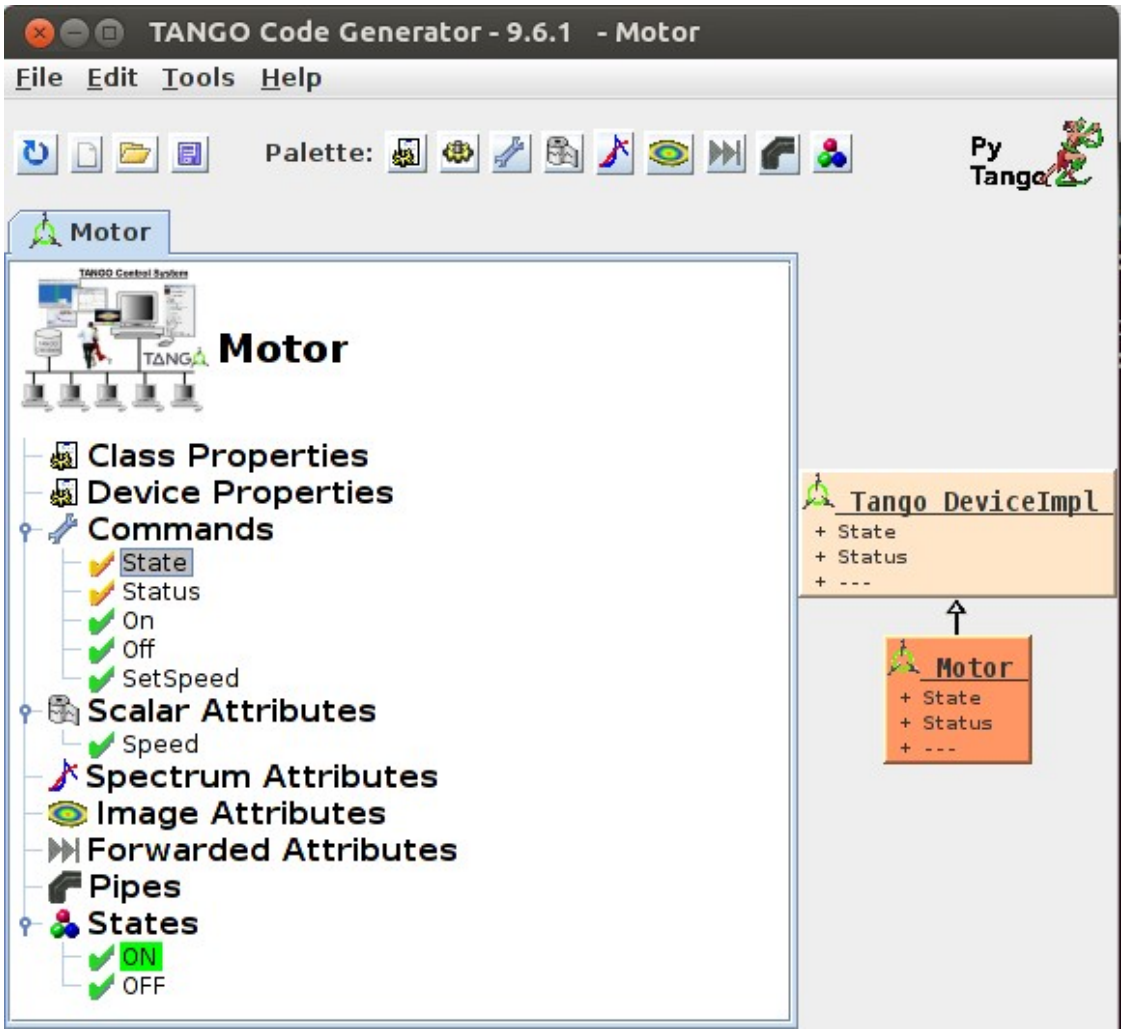
Melléklet

A.1.

A Tango eszköz beállításárólt készült képernyképek



A.1. ábra. Az új eszöz beállítása, a képen látható paraméterekkel generálható a motor eszköz osztálya.



A.2. ábra. A generált Motor eszköz osztály beállításai

A.3. ábra. A szerver beállításának módja.

A.4. ábra. Egy eszköz deklarálása. Erre minden eszköznál szűkséeg van, akkor is, ha ugyan annak az osztálynak egy másik példányáról van szó.

Név	Pozíció	Kiterjedés	Elfordulás
bed_ holder_1	(75, 20, 150)	(150, 40, 40)	(0, 0, 0)
bed_ holder_2	(75, 45, 150)	(200, 10, 70)	(0, 0, 0)
bed_ slider	(175, 45, 150)	(15, 15, 15)	(0, 0, 0)
bed_ rotatory	(75, 55, 150)	(200, 10, 10)	(0, 0, 0)
bed	(75, 60, 150)	(200, 10, 70)	(0, 0, 0)
ct_1	(275, 120, 40)	(80, 240, 80)	(0, 0, 0)
ct_2	(275, 120, 260)	(80, 240, 80)	(0, 0, 0)
spect_ rotatory_ 1	(275, 20, 150)	(80, 40, 140)	(0, 0, 0)
spect_ rotatory_ 2	(275, 220, 150)	(80, 40, 140)	(0, 0, 0)
spect_1	(210, 25, 150)	(40, 30, 60)	(0, 0, 0)
spect_2	(210, 215, 150)	(40, 30, 60)	(0, 0, 0)

A.1. táblázat. A spect_ct.json fájlban tárolt alkatrész adatok. A teljes JSON a csatolt kódban ezen a néven megtalálható.

Név	Lineáris mozgás		
	irányvektor	alsó végállapot	felső végállapt
bed_ rotatory	(1, 0, 0)	(74, 55, 150)	(376, 55, 150)
bed	(1, 0, 0)	(74, 60, 150)	(376, 60, 150)

A.2. táblázat. A lineárisan mozgó allkatrészek beállításai.

A.2. A szoftver tesztelésére használt adatok

Az alábbi táblázatokban megtalálhatók azok a tesztadatok, amik a 4.1 fejezetbe található teszt bemenetelei voltak. A teszt ezekkel az adatokkal megismételhető. A A.1 táblázat tartalmazza az alkatrészek kezdeti pozícióját, kiterjedését és elfordulását.

Név	Körmozgás mozgás			
	forgáskö-zéppont	forgás-tengely	alsó végállapot	felső végállaot
bed_ rotatory	(275, 60, 150)	(1, 0, 0)	-	-
bed	(275, 60, 150)	(1, 0, 0)	(-0.1, 0, 0)	(0.1, 0, 0)
spect_1	(210, 120, 150)	(1, 0, 0)	-	-
spect_2	(210, 120, 150)	(1, 0, 0)	-	-

A.3. táblázat. A forgó mozgást végző allkatrészek beállításai.

Név	Mozgatott alkatré-szek	Mozgatás	Az én általam hasz-nált elérési út
bed_slider	bed_rotatory, bed	lineárisan	"/spectct/motors /bedslider"
bed_ rotatory	bed_ rotatory, bed	körmozgással	"/spectct/motors /bedrotatory"
spect_rotatory_1	spect_1	körmozgással	"/spectct/motors /spectrotatory1"
spect_rotatory_2	spect_2	körmozgással	"/spectct/motors /spectrotatory2"

A.4. táblázat. A motork beállításai.