

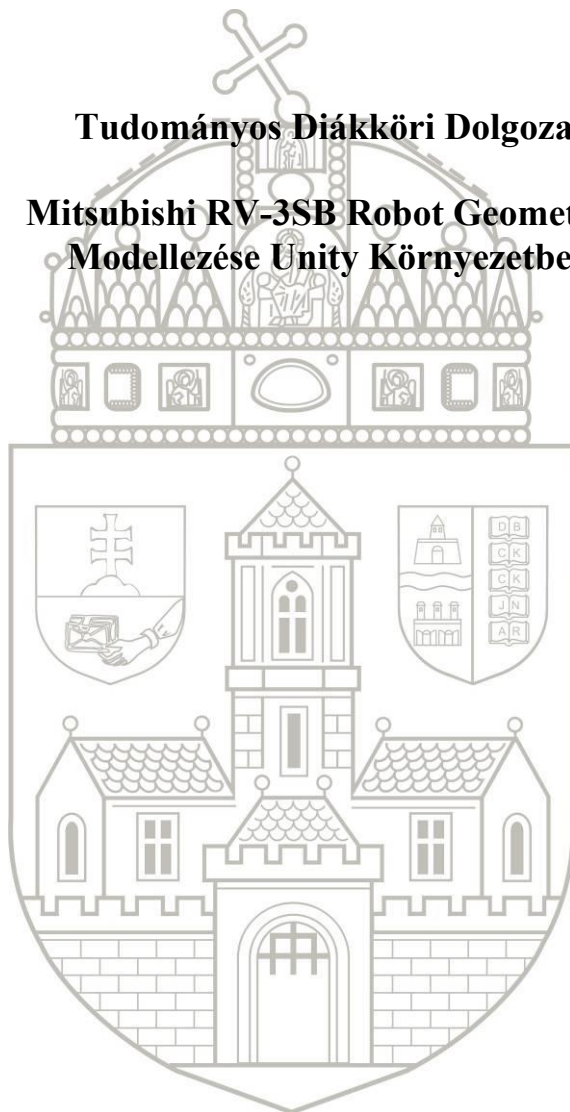


ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

BÁNKI DONÁT GÉPÉSZ ÉS
BIZTONSÁGTECHNIKAI MÉRNÖKI KAR

Tudományos Diákköri Dolgozat

Mitsubishi RV-3SB Robot Geometriai Modellezése Unity Környezetben



OE-BGK
2024.

Hallgató neve: Zsámbok Ákos
Konzulens neve: Varga Bence

ABSZTRAKT

A dolgozat célja egy Mitsubishi RV-3SB típusú ipari robot geometriai modellezésének bemutatása. A dolgozatban részletesen ismertetésre kerül a nyílt kinematikai láncsal rendelkező ipari robotok direkt geometriai modellezéséhez széleskörben használt Denavit-Hartenberg konvenció. Az inverz geometriai modell megalkotásához, a manipulátor speciális geometriai tulajdonságait kihasználva, az úgynevezett dekompozíciós eljárás kerül alkalmazásra. Ez a módszer lehetővé teszi, hogy az inverz geometria komplex, nemlinearitásokkal terhelt problémáját, könnyebben kezelhető részfeladatokra bontsuk. Mindkét módszer gyakorlati megvalósítása is ismertetésre kerül az RV-3SB robot példáján. A kiválasztott robot matematikai modellje Unity fejlesztő környezetben kerül implementálásra, amely lehetővé teszi a robotpóz 3D-ós szemléltetését, illetve különböző felhasználói felületek megvalósítását. Habár a Unity hagyományosan videójátékok fejlesztésére használt szoftver, teljes mértékben alkalmas ipari robotok digitális modellezésére is. A Unity játékmotor, a megfelelő matematikai modellek C# programozási nyelven történő implementálásával, lehetővé teszi egy olyan digitális iker létrehozását, amellyel a későbbiekben akár a fizikai robot működését valós időben szimuláljuk.

ABSTRACT

The aim of this paper is to present the geometric modeling of a Mitsubishi RV-3SB industrial robot. The paper provides a detailed overview of the Denavit-Hartenberg convention, which is widely used for the direct geometric modeling of industrial robots with open kinematic chains. To create the inverse geometric model, a decomposition method is applied, which leverages the specific geometric properties of the manipulator. This approach allows the complex, nonlinear challenges of inverse geometry to be broken down into more manageable sub-tasks. Both methods are demonstrated in practice using the RV-3SB robot as an example. The mathematical model of the selected robot is implemented in the Unity development environment, which enables 3D visualization of the robot's poses as well as the development of various user interfaces. Although Unity is traditionally used for video game development, it is fully capable of digital modeling for industrial robots. With the appropriate mathematical models implemented in C# within Unity's game engine, it is possible to create a digital twin that can later simulate the real-time operation of the physical robot.

1. Tartalomjegyzék

2. Bevezetés.....	3
3. Robotika alapjai.....	5
3.1. Robotikában használt koordináta rendszerek	5
3.2. Robot modellezés szintjei.....	6
3.3. Kinematikai lánc	7
4. Unity szoftver bemutatása	9
4.1. A Unity felhasználói felülete	9
4.2. Programozás a Unityben.....	10
5. A digitális iker elkészítése	12
5.1. Szimulációs program felhasználói felülete.....	13
5.2. Mozgás a szimulációs világban	13
5.3. A csuklók forgatása.....	15
6. Direkt geometriai feladat megoldása.....	17
6.1. Rotációs mátrix.....	17
6.2. Roll, Pitch, Yaw forgatás	19
6.3. Homogén transzformációk	22
6.4. Denavit-Hartenberg transzformáció	24
6.5. Denavit-Hartenberg transzformáció implementálása	26
7. Inverz geometriai feladat megoldása.....	29
7.1. Robot konfigurációk	29
7.2. Pozíció - orientáció leválasztás.....	30
7.3. Pozíció kiszámítása.....	32
7.4. Orientáció kiszámolása.....	35
7.5. Inverz geometria implementálása	36
8. Összegzés és továbbfejlesztési lehetőséget	40
9. Hivatkozások.....	41

10. Mellékletek.....	43
----------------------	----

2. Bevezetés

A robotikai szimuláció egyre növekvő jelentőségre tesz szert mind az oktatásban, mind az ipari alkalmazásokban, hiszen lehetőséget biztosít olyan komplex rendszerek és eszközök működésének részletes bemutatására és elemzésére, amelyeket a valóságban nehézkes vagy akár lehetetlen lenne bemutatni. Az ipari robotok tipikusan nagy méretűek, bonyolult felépítésűek és súlyosak, ami azt jelenti, hogy mozgatásuk, telepítésük és működtetésük jelentős logisztikai és biztonsági kihívásokkal jár. Ezen akadályok áthidalásában nyújtanak hatékony megoldást a virtuális szimulációk, amelyek lehetővé teszik a robotok digitális másolatainak, úgynevezett digitális ikreknek a használatát. Ezek a szimulációk hozzájárulnak ahhoz, hogy a robotokat és azok képességeit könnyedén, rugalmasan lehessen bemutatni bárhol és bármikor, csupán egy számítógép vagy laptop segítségével.

A digitális ikrek fejlesztése lehetőséget biztosít arra, hogy valós időben és valós környezetben modellezzük a robotok működését, interakcióit és az általuk végzett feladatokat. Ennek a technológiának köszönhetően különféle szcenáriók szimulálhatók, amelyekkel előre tesztelhető egy robot viselkedése vagy teljesítménye különféle körülmények között. A digitális ikrek nemcsak a karbantartási és fejlesztési folyamatokat teszik hatékonyabbá, hanem hozzájárulnak az oktatáshoz és az ismeretek szélesebb körű megosztásához is, hiszen a robotikus rendszerek működésének vizualizálása nagyban megkönnyíti a bonyolult folyamatok és mechanizmusok megértését.

A dolgozatom célja a Mitsubishi RV-3SB robotkar működésének modellezése a Unity játékmotor segítségével. Ez a robot, megtalálható az Óbudai Egyetem Bánki Donát Gépész és Biztonságtechnikai Mérnöki Karának laboratóriumában. A Unity platform választása nem véletlen, hiszen egy könnyen elérhető, felhasználóbarát és széles körben alkalmazott fejlesztési környezetről van szó, amely lehetőséget biztosít a valósághű fizikai szimulációk megvalósítására. A Unity segítségével létrehozott modell lehetőséget nyújt arra, hogy a robot virtuális környezetben működjön, így vizsgálhatók a különböző feladatok és műveletek, amelyek a robot fizikai használatától függetlenül is bemutathatók, elemezhetők és fejleszthetők. A szimuláció során a robot különböző kinematikai tulajdonságai kerülnek bemutatásra. A Unity játékmotor által biztosított felület és programozási lehetőségek lehetővé teszik a robotkar interaktív szimulációját, amely révén a felhasználók valós időben változtathatják a robot paramétereit, ezzel valósághű visszajelzést kapva annak viselkedéséről.

E dolgozat több lépésben mutatja be a szimuláció létrehozásának folyamatát: elsőként részletesen kitérek a robotika a dolgozat megértéséhez szükséges alapjaira. Ezt követően a

Unity szoftvert ismertetem, bemutatom a fejlesztői környezet alapjait, a felhasználói felületet, valamint a programozási lehetőségeket. A Unity lehetővé teszi a robot geometriai modelljének és mozgásának megvalósítását egy olyan virtuális környezetben, amelyben a valósághű fizikai hatások, például gravitáció, ütközések és egyéb dinamikai interakciók is szimulálhatók.

A dolgozat egyik fontos eleme a Denavit-Hartenberg transzformáció alkalmazása, amely lehetővé teszi a robotkar pozíciójának és orientációjának pontos meghatározását a szimulációs térben. Ez a transzformációs módszer lehetővé teszi a robot különböző mozgásainak és forgásainak matematikai modellezését, ami elengedhetetlen a robot pontos szimulációjához.

Összességében a dolgozat célja, hogy bemutassa, hogyan készítettem el egy ipari robot digitális iker változatát, valamint, hogy bemutassam a mögötte rejlő matematikai leírásokat.

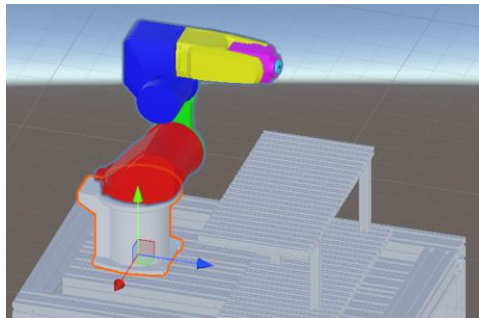
3. Robotika alapjai

3.1. Robotikában használt koordináta rendszerek

Robotikában többféle koordináta rendszert használunk attól függően, hogy a robot vagy a munkaterének éppen melyik részét vizsgáljuk. A robotikában 3 dimenziós Descart koordinátarendszereket használunk.

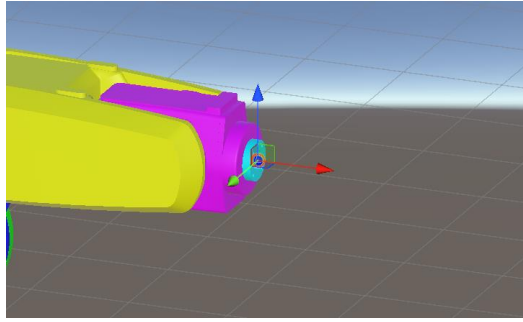
Egy koordinátarendszernek két tulajdonsága van, amiknek az ismerete szükséges ahhoz, hogy pontosan le tudjuk írni az elhelyezkedését a térben. Az egyik a pozíció, ez a koordinátarendszer origó pontját adja meg egy másik koordinátarendszerben. Itt fog elhelyezkedni a vizsgált koordinátarendszerünk $[0, 0, 0]$ pontja. A másik tulajdonság, az orientáció, ez azt mutatja meg, hogy az adott origó pont körül, hogyan van forgatva koordináta rendszer.

A világkoordináta rendszer a legalapvetőbb a robotikában használt koordinátarendszerek közül. Ez nem változtat sem pozíciót, sem orientációt. Ehhez képest tudjuk megadni a robot, illetve a környezetében lévő tárgyak helyzetét. (Lásd 1. ábra)



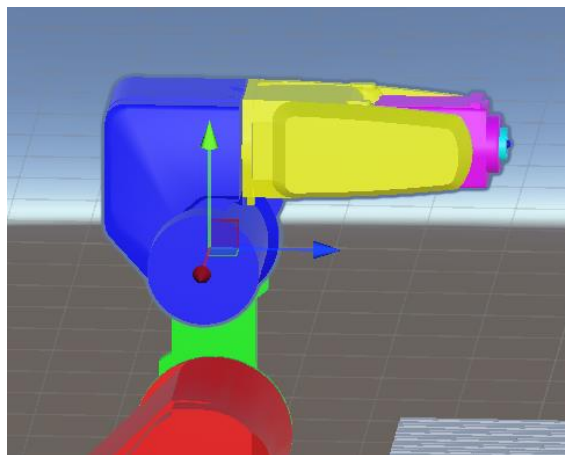
1. ábra: Világkoordináta rendszer

A szerszámkoordináta rendszer a szerszámhoz van rögzítve, ehhez képest nem mozdul el és nem változtat az orientációját. Ha a szerszám mozog a térben akkor a szerszám koordinátarendszer mozog vele együtt. Ennek a z tengelye mindig a szerszámból kifelé mutat. (Lásd 2. ábra)



2. ábra: Szerszámkoordináta rendszer

Minden szegmenshez rendelünk egy csuklókoordináta rendszert. Egy robot n számú csukló esetén $n + 1$ számú szegmessel rendelkezik. A szegmenseket 0-tól a csuklókat 1-től indexeljük. l_i szegmenshez $O_i x_i y_i z_i$ koordinátarendszer tartozik, ennek a jelölési rendszernek az az előnye, hogy l_1 szegmens $O_1 x_1 y_1 z_1$ koordinátarendszert fogja elfordítani. (Lásd 3. ábra)



3. ábra: Csuklókoordináta rendszer

3.2. Robot modellezés szintjei

A robot modellezésnek 3 szintje van, melyek különböző szempontok szerint írják le a robot mozgását matematikailag. Ezek a szintek a geometriai, kinematikai és a dinamikai. A magyar irodalmak a geometriait és a kinematikait gyakran összevonják, pedig pontosabban leírja őket, ha külön vannak említve.

A geometriai szint a robot szerkezetét írja le, sebességtől és időtől függetlenül egy mozdulatlan robot esetében. Különböző koordinátarendszerek segítségével, adja meg a robot pontos pozícióját. A geometriai szintet tovább lehet bontani 2 részre.

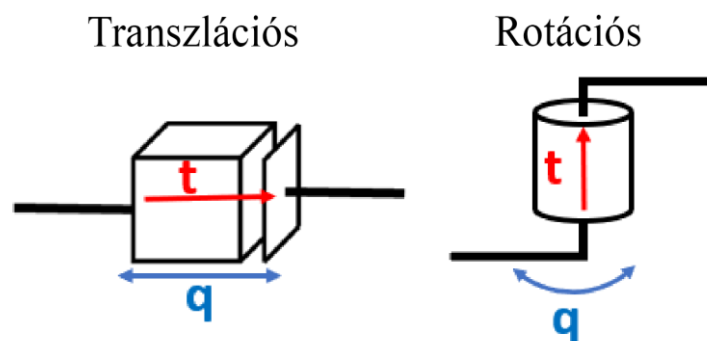
- Direkt geometria: Ennek a lényege, hogy a csuklószőgek ismeretéből, számoljuk ki a szerszám pozícióját és orientációját. Ez egy egyértelmű feladat. Az adott csuklószőgekhez csak és kizárólag egy szerszámpozíció tartozhat.
- Inverz geometria: Ez pont a direkt geometria fordítottja. Ez esetben a csuklók által bezárt szögeket nem ismerjük, csak a szerszám pozícióját és orientációját és ebből szükséges visszaszámolnunk, hogy milyen csuklószőgek szükségesek a kívánt szerszámpozíció és orientáció eléréséhez. Ez már nem egy egyértelmű feladat. Az adott szerszám pozícióhoz többféleképpen is eljuthatunk, így a feladatnak nem csak 1 megoldása van. (Ezzel részletesebben a Robot konfigurációk fejezetben foglalkozok)

Kinematikai szint esetében a csuklók szögsebessége és a szerszámsebesség közötti kapcsolatot vizsgáljuk. Ezt is fel lehet bontani direkt és inverz részekre, attól függően, hogy melyik irányból közelítjük meg a feladatot.

A dinamikai szint még tovább megy. Itt már a különböző gyorsulások és nyomatékok közötti kapcsolatot vizsgáljuk. Ennek a segítségével tudjuk kiszámolni, hogy a csuklóknak mekkora nyomatékot szükséges leadniuk a kívánt szerszámgyorsulás eléréséhez.

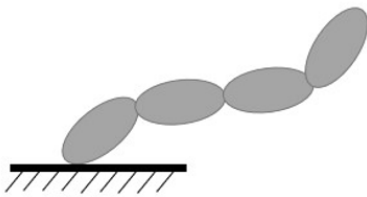
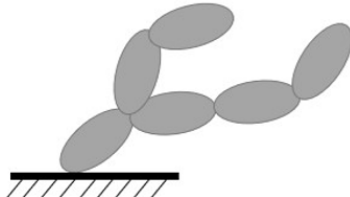
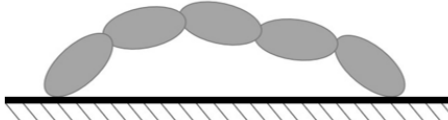
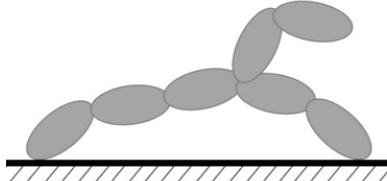
3.3. Kinematikai lánc

Modellezés során úgy tekintjük, hogy a robotok merev testekből (ezeket hívjuk szegmenseknek), illetve ezeket összekötő csuklókból állnak. A csuklókat jellemezhetjük a tengelyük irányát leíró t egységvektorral, ami a lineáris elmozdulás irányát adja meg transzlációs csukló esetén, míg rotációs csuklóknál a forgástengely irányát jelöli ki. Továbbá a J_1 csukló lineáris elmozdulását d_1 -gyel, szögelfordulását pedig θ_1 -gyel jelöljük. Általános esetben a q_1 , a csuklózváltozót jelöli, amely a csukló típusától függően jelenthet eltolást és elforgatást is. (Lásd 4. ábra)



4. ábra: Transzlációs és rotációs csukló [1]

A robotok mozgását és irányíthatóságát alapvetően a robot kinematikai láncja határozza meg. Egy robot kinematikai lánc szempontjából lehet nyitott vagy zárt, illetve egyszerű vagy összetett. A nyitott kinematikai láncsal rendelkező robot esetén csak egy szegmens van, ami csatlakozik a fix alaphoz, míg zárt kinematikai lánc esetén legalább 2 csukló. Egyszerű kinematika esetén minden szegmens legfeljebb 2 másik szegmenshez csatlakozik, míg összetett kinematikai lánc esetében legalább az egyik szegmens több mint 2 másik szegmenshez csatlakozik. (Lásd 5. ábra)

	
Egyszerű, nyílt kinematikai lánc	Összetett, nyílt kinematikai lánc
	
Egyszerű, zárt kinematikai lánc	Összetett, zárt kinematikai lánc

5. ábra: Kinematikai láncok [1]

A dolgozat során feldolgozott RV-3SB egy olyan robot, ami 6 rotációs csuklóval és nyitott, egyszerű kinematikai láncsal rendelkezik.

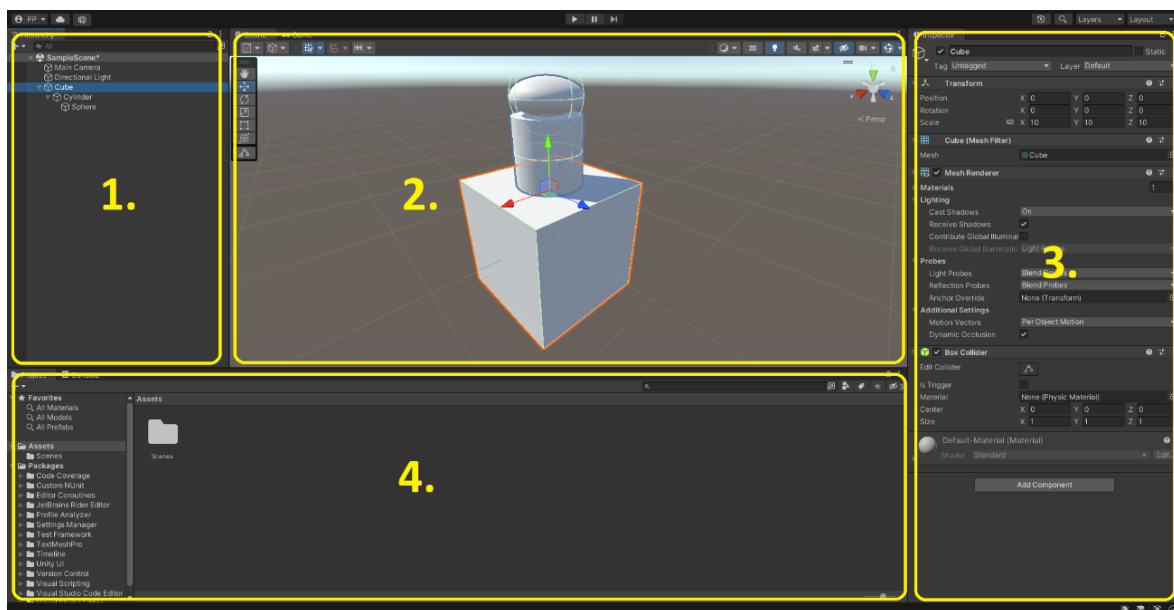
4. Unity szoftver bemutatása

A Unity az egyik legelterjedtebb, több platformra is használható játékmotor és fejlesztési környezet. Rengeteg mai fejlett játékot készítettek vele. Az egyik nagy előnye, hogy több platformra is lehet vele alkalmazásokat fejleszteni, többek között mobil eszközökre, PC-re, konzolokra, de alkalmas VR és AR eszközökre is. Ez nagy segítség a fejlesztőknek, hogy egyetlen környezetből több eszközre is ki tudják adni az alkalmazásukat. Emellett könnyen átlátható és felhasználóbarát felülettel rendelkezik, így alkalmas a kezdő fejlesztők számára is. A Unity az Nvidia által fejlesztett PhysX fizikai motort használja. Ez teszi lehetővé a valósághű fizikai szimulációkat, például a gravitációt, az ütközések, objektumok közötti interakciók kezelését.

A dolgozat során azért is esett a Unityre a választás, mert ennek a fejlesztőkörnyezetnek van ingyenesen elérhető verziója, ami szintén profi eszközöket biztosít. Ezt leginkább kisebb fejlesztők vagy indie stúdiók szokták használni, de sok otthoni amatőr videójáték készítő is ezt használja. [2]

4.1. A Unity felhasználói felülete

Egy új projekt létrehozásakor a Unity alap felhasználói felületével találkozunk. Mint a legtöbb fejlesztő vagy modellező alkalmazás esetében, úgy itt is lehetőségünk van ezeket bezárni, vagy új ablakokat megnyitni. Alapvetően 4 ablak van megnyitva. (Lásd: 6. ábra)



6. ábra: Unity felhasználói interfésze [3]

Az első rész a Hierarchy, itt látható az összes objektum, amit létrehoztunk. Léteznek szülő objektumok (parent) és gyerek objektumok (child). A Hierarchy ablakban pedig mapparendszer szerűen egymás alá, illetve fölé tudjuk rendelni az objektumokat, hogy egymás parent, illetve child objektumai legyenek. Egy objektum tud egyszerre több objektumnak is a szülőobjektuma lenni. Illetve egy objektum lehet egyszerre parent és child is, olyan formán, hogy alá van rendelve egy objektumnak, de ő alá is van rendelve egy másik objektum. Ez az alárendeltségi kapcsolat a gyakorlatban kényszerek formájában mutatkozik meg, így a szülőobjektum bizonyos változásai kihatnak a gyerek objektumaira. Például, ha elmozgatjuk a térben a szülő objektumot akkor a child objektum is pontosan annyit fog elmozdulni.

A második ablak a scene. Itt láthatjuk térben az aktuálisan létrehozott GameObjecteket és ezeket itt tudjuk szerkeszteni is. A GameObject a Unity alapvető objektumtípusa, tulajdonképpen a szimulációs tér felépítéséhez használt minden komponens, úgymint a 3D modellek, gombok vagy szövegmezők stb., GameObjectnek minősülnek. A scene ablakban tudjuk grafikusán módosítani az objektumok helyzetét és orientációját is. Az ablak bal felső részén van a Game fül. Erre kattintva átváltunk a játék nézetre. Ez az a nézet, amit akkor látunk, ha elindítjuk a programunkat.

A harmadik fül az inspector ablak. Itt az aktuálisan a hierarchy-ban kijelölt GameObject tulajdonságait láthatjuk és szerkeszthetjük. Itt tudjuk megadni, hogy milyen anyagból van az objektum, az x , y és z koordinátáit, az elfordulását, a méretét.

A negyedik rész a projekt ablak. Itt látható minden elem, amit felhasználtunk a projektünk során, legyen az C# script, 3D modell, vagy akár hang file. a projekt ablak bal felső sarkában át tudunk váltani a konzol földre. Itt tudnak megjelenni a különböző hibaüzenetek, de meg tudunk jeleníteni log-okat is, ami nagyban meg tudja könnyíteni a programozás közbeni hibakeresést. [4] [5]

4.2. Programozás a Unityben

Ahhoz, hogy ne csak egy statikus virtuális világunk legyen, amiben benne vannak a GameObject-ek, létre kell hoznunk különböző scripteket, amikkel tudjuk módosítani az objektumok tulajdonságait. A Unity több programozási nyelv használatára ad lehetőséget, ezek a UnityScript, a Boo illetve a C#. Én a dolgozatom során a C# magasszintű programozási nyelvet használtam, mert ezzel már többször találkoztam a tanulmányaim során.

Ha létrehozunk egy új C# scriptet, az a projekt ablakban fog megjelenni, ebbe tudjuk írni a kódot, amit szeretnénk. Viszont ez így még nem fog lefutni, ahhoz, hogy a program futása során

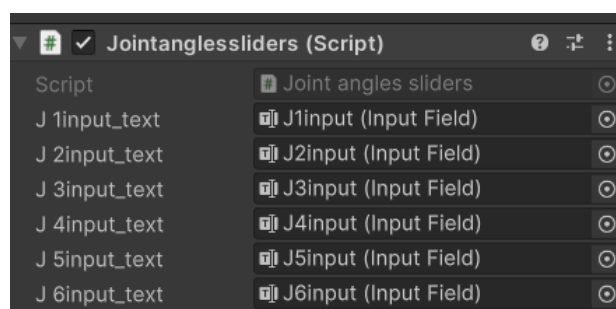
használva legyen a scriptünk, a projekt fülből rá kell húznunk egy GameObjectre a Hierarchy ablakban. Én minden Scripthez, amit használtam létrehoztam egy új üres objektumot, aminek nincs fizikai kiterjedése. Így később könnyen megtalálható, hogy hol tudok új Objektumokat átadni a scriptnek.

Ezután meg kell hívnunk a C# kódban az objektumokat, amiket használni szeretnénk (Lásd 7. ábra). Itt deklaráltam minden csuklószőg bemeneti mezőhöz egy InputField osztályú változót, amelyet a virtuális térben kialakított kezelői felületen létrehozott szövegdobozokhoz (Input Field) rendeltem hozzá. Ez teszi lehetővé, hogy felhasználó interakcióba léphessen a programunkkal és a beadott csuklószőg értékeket programkódban felhasználjuk a megfelelő számítások elvégzéséhez.

```
//Csuklószőg bemeneti mezők meghívása az UI-ról
public InputField J1input_text;
public InputField J2input_text;
public InputField J3input_text;
public InputField J4input_text;
public InputField J5input_text;
public InputField J6input_text;
```

7. ábra: Objektumok meghívása C#-ban

Ezt követően, ha a hierarchy ablakban kijelöljük a GameObjectet, amihez hozzárendeltük a scriptet akkor az inspector ablakban megjelennek ezek a deklarált változók (Lásd 8. ábra). Viszont ahhoz, hogy a Unity tudja, hogy melyik változóval melyik objektum tulajdonságait kösse össze, meg kell fognunk a hierarchy ablakban az objektumot és át kell húznunk a C# script inspector ablakába, a megfelelő változónévhez. Így azt a változót összeköttöttük az Objektummal. Ez a kapcsolat két irányú, ha az objektumnak változik egy tulajdonsága akkor az a változóérték is egyből módosul míg, ha a változóban módosítunk egy értéket azzal együtt azonnal módosul az objektum tulajdonsága is. [6]



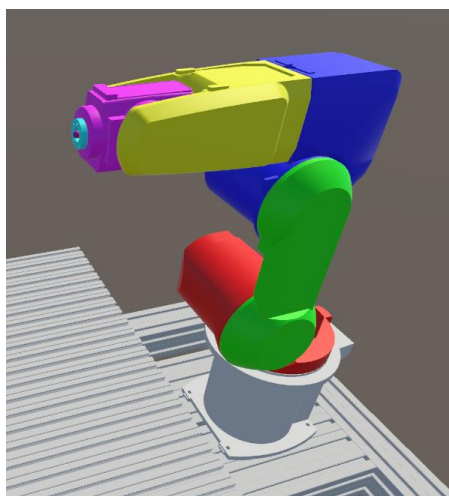
8. ábra: Objektumok tulajdonságainak átadása a C# scriptnek

5. A digitális iker elkészítése

A projekt során két 3D-s modellre volt szükségem. Az egyik maga a Mitsubishi RV-3SB robot modellje. A Mitsubishi szolgáltatásai közé tartozik, hogy az általa gyártott robotok 3D modelljeit ingyenesen elérhetővé teszi a weboldalán [7]. Ezek nem mindenre kiterjedő modellek, nem tartalmazzák a robot belső alkatrészeit, csak a külső geometriáját. Én is innen töltöttem le az RV-3S robot modelljét, amely típuszáma eltérő a dolgozatban szereplő robotétól, azonban a két manipulátor kinematikai szempontból megegyező felépítésű. A típus megjelölésben a különbség kisebb szerkezeti eltérést jelöl csak, mivel az RV-3SB robot esetében minden csuklóhoz tartozik fék.

A dolgozatomban vizsgált robot az Óbudai Egyetem Bánki Donát Gépész és Biztonságtechnikai Mérnöki Karán található. Ezért, hogy az általam készített alkalmazást később fel lehessen használni más TDK-khoz vagy szakdolgozatokhoz, az egyetemen ténylegesen megtalálható környezetében szerettem volna elkészíteni. A karon egy korábbi szakdolgozatot készítő hallgató, már elkészítette az asztal pontos 3D modelljét. Ezt az ő általa készített modellt használtam fel én is. [8]

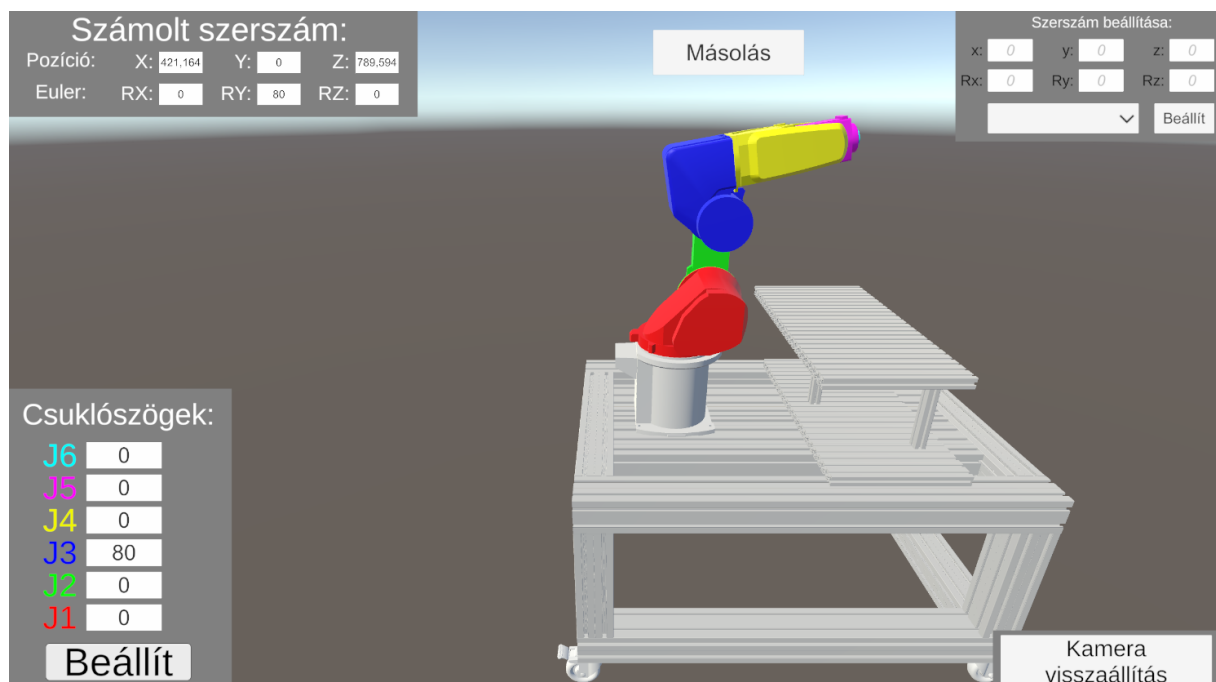
Azért, hogy vizuálisan jobban elkülönüljenek a szegmensek egymástól, mindegyiket különböző anyagúra állítottam be, így mindegyik szegmensnek más színt tudtam adni. Ennek egy másik előnye, hogy a csuklósögek értékét állító bemeneti mezők mellé is oda tudtam írni az általa módosított csukló számát a megfelelő színnel, így jól látható, hogy melyik bemeneti mező melyik szegmenst fogja elforgatni. (Lásd 9. ábra)



9. ábra: RV-3SB Digitális iker modellje

5.1. Szimulációs program felhasználói felülete

A szimulációs program felhasználói felületét (más néven UI-t) úgy készítettem el, hogy letisztult és könnyen kezelhető legyen. A bal alsó sarokban a felhasználónak lehetősége van megadni csuklózögeket, amikre a robot beáll a „Beállít” gomb megnyomásakor. A bal felső sarokban a szerszám kiszámolt pozíciója és orientációja látható. A jobb felső sarokba tettem az inverz geometriához szükséges bemeneteket. Itt adható meg, hogy a felhasználó milyen pozícióba, illetve orientációba szeretné beállítani a szerszámot. A gyors beállítás megkönnyítése végett fönt középen elhelyeztem egy másolás gombot. Ez a megnyomásakor átmásolja bal felülről az aktuális szerszám pozíciót és orientációt a jobb felső sarokba az inverz geometriához tartozó bemenetekhez. Ezeken kívül jobb alul elhelyeztem egy gombot, ami visszaállítja a kamerát a kiinduló állásba (Ezt a következő fejezetben tárgyalom részletesebben). (Lásd 10. ábra)

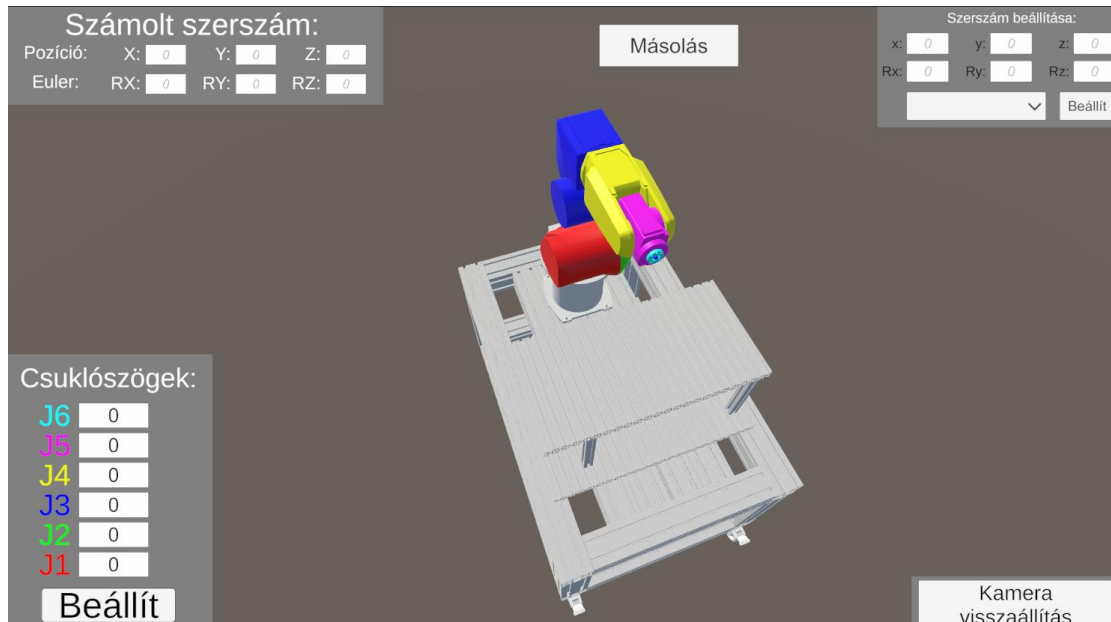


10. ábra: Szimulációs program felhasználói felülete

5.2. Mozgás a szimulációs világban

Ahhoz, hogy a szimulált robot pozíciója mindig, minden pozíciójában jól látható legyen először hozzáadtam több kameraállást a programhoz, de aztán ezt az ötletet elvettem, mert mindig volt olyan állása a robotnak, ami nem látszott egyértelműen, ha pedig újabb és újabb kameraállásokat készítettem akkor már az volt követhetetlen név alapján, hogy melyik honnan

fogja mutatni a robotot. Ehelyett inkább úgy döntöttem, hogy jobb irány, ha lefejlesztem a mozgását a kamerának. Így a WASD gombok segítségével lehet mozogni a szimulációs térben a robot körül, space gombbal lehet emelkedni shift gombbal pedig süllyedni lehet. Lehet forogni is a kamerával, ha lenyomva tartjuk a jobb egér gombot.



11. ábra: Mozgás a szimulációs világban

Ennek a leprogramozásához készítettem egy kockát, amihez hozzáadtam child Objektumként a kamerát így, ha egy C# scriptből változtatom a kocka helyzetét akkor a kamera is mozog vele. A mozgathoz tartozó programot a „Viewer Control” nevű scriptbe írtam meg. Ennek az elején adtam meg, hogy milyen mozgási sebességekkel rendelkezzen a kocka, illetve, hogy milyen kezdeti elfordulása legyen. (Lásd 12. ábra)

```
//Kamera mozgatható sebességei illetve az egérrel történő mozgás érzékenységeinek megadása
public float sebesseg = 750f;
public float egererzenyseg = 750f;
public float fel_le_sebesseg = 750f;

//Alap koordináták megadása
//A szimuláció elindításakor ez lesz a kamera kiinduló pozíciója illetve a "Kamera visszaállítása" gomb is ide állítja vissza a kamerát
public float Alap_x = 1500;
public float Alap_y = 1000;
public float Alap_z = -500;
public float Alap_pitch = 24;
public float Alap_yaw = -90;

//kezdeti nézőpont RPY szögeinek megadása
//A szimuláció elindításakor ez lesz a kamera kiinduló elfordulása illetve a "Kamera visszaállítása" gomb is ide állítja vissza a kamerát
float pitch = 24;
float yaw = -90;
```

12. ábra: Viewer Control script, mozgási sebességek beállítása

Az update metódusba írtam meg, magukat a mozgásokat. Először a billentyűzettel történő irányítást majd az egérrel történő irányítás látható. Az egérrel történő irányítás során, ha a felhasználó lenyomja a jobb egérgombot, akkor a program „elkapja” az egeret, ami azt jelenti, hogy az nem tud kimenni a szimulációs program ablakából, ez mellett láthatatlanná is válik. (Lásd 13. ábra)

```
//WASD-gombokkal transláció mozgások a térben
transform.Translate(Vector3.right * Input.GetAxis("Horizontal") * sebesseg * Time.deltaTime);
transform.Translate(Vector3.forward * Input.GetAxis("Vertical") * sebesseg * Time.deltaTime);

//space vagy shift nyomva van-e
float fel = 0;
if (Input.GetKey(KeyCode.Space))
    fel = 1f;
if (Input.GetKey(KeyCode.LeftShift))
    fel = -1f;
//fel-le mozgás amennyiben a space vagy a shift gombok nyomva vannak
transform.position += Vector3.up * fel * fel_le_sebesseg * Time.deltaTime;

//Ha a jobb egérgomb le van nyomva akkor forgás a térben
if (Input.GetMouseButton(1))
{
    //Ha az egér nincs az ablakba zárva akkor oda zárása illetve láthatatlanná tétele
    if (!egerzarva)
    {
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
        egerzarva = true;
    }
    //kamera yaw és pitch szögeinek beállítása az egér elmozdulásának megfelelően
    yaw += Input.GetAxis("Mouse X") * egererzekenyseg * Time.deltaTime;
    pitch -= Input.GetAxis("Mouse Y") * egererzekenyseg * Time.deltaTime;
}
else if(egerzarva) //Ha az egér az ablakhoz van zárva de már nincs erre szükség akkor feloldása és láthatóvá tétele
{
    Cursor.lockState = CursorLockMode.None;
    Cursor.visible = true;
    egerzarva = false;
}

//Új megváltozott yaw és pitch szögek átadása a kamerának
transform.eulerAngles = new Vector3(pitch, yaw, 0f);
```

13. ábra: Mozgásért felelős kódrészlet

5.3. A csuklók forgatása

A modell összeállítása után következett a kényszerek elhelyezése. Ehhez először a forgatási tengelyek helyzetét kellett meghatároznom. Itt a Robothoz tartozó dokumentációt használtam fel, ebben le vannak írva a tengelyek pozíciói. Ezután a kar minden elemének eltoltam a koordinátarendszerét ezekbe a pozíciókba, így megkaptam a csuklókoordináta rendszereket, amiknek az egyik tengelye mindig egybe esik a csuklók forgástengelyével.

Úgy írtam meg a scriptet, hogy minden frame frissítés során az csuklók elforgatásának mértékét kvaterniókkal adtam meg, a UNITY saját függvényét felhasználva:

Quaternion.AngleAxis(„elforgatási szög”, „elforgatás tengelye”).

A fenti metódus, a forgástengelyből és a forgatás szögéből számítja ki a forgatást leíró kvaternió értékét. Az elforgatás tengelyéhez egy Vektor3 típusú bemenetet vár, az elforgatási szöget pedig float típusúként kell megadnunk. Azoknak a tengelyeknek az esetében, ahol y tengely körül forgatunk, ott invertálni az elforgatási szög értékét, mert a Unity balsodrású koordinátarendszereket alkalmaz. (Lásd 14. ábra) A kiszámított kvaterniók ismeretében a kiválasztott GameObject (pl.: Joint2) „*transform.rotation*” tulajdonságát írva, tudom elforgatni a virtuális térben a csuklókat.

```
//1-es csukló:
Vector3 Joint1_tengely = new Vector3(0, 1, 0);
Quaternion Joint1_forgat = Quaternion.AngleAxis(-float.Parse(J1input_text.text), Joint1_tengely);
Joint1.transform.rotation = Joint1_forgat;

//2-es csukló:
Vector3 Joint2_tengely = new Vector3(1, 0, 0);
Quaternion Joint2_forgat = Quaternion.AngleAxis(float.Parse(J2input_text.text), Joint2_tengely);
Joint2.transform.rotation = Joint1_forgat * Joint2_forgat;
```

14. ábra: Csuklók elforgatása a tengelyük mentén

5.4. Math.NET Numerics

A program elkészítése során felhasználtam a Math.NET Numerics könyvtárat. Ez egy nyílt forráskódú, .NET környezetre fejlesztett matematikai könyvtár, amely különböző matematikai és tudományos számítások elvégzésére szolgál. A könyvtár rendkívül széleskörű funkciókat biztosít, beleértve a lineáris algebrát, statisztikát, numerikus optimalizálást, interpolációt és különböző matematikai modellezési technikákat.

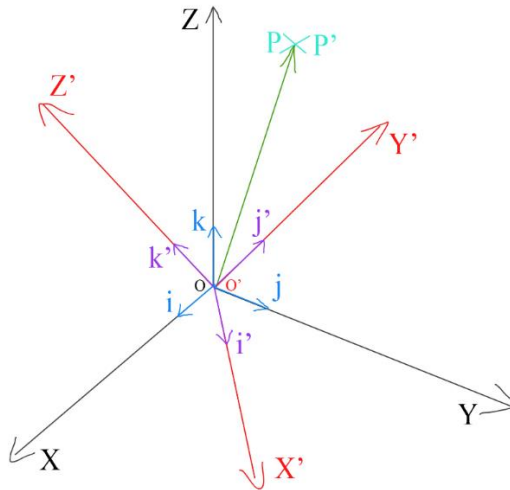
Azért választottam ennek a használatát, mert egyszerűen lehet benne különféle mátrixműveleteket végezni, amire a számítások során többször szükségem volt. [9]

6. Direkt geometriai feladat megoldása

Az ipari robotok alkalmazása szempontjából az egyik legfontosabb információ a robotról az a szerszámkoordináta. Ez adja meg, hogy a világkoordináta-rendszerben melyik pontba van a szerszám munkapontja. Ezt a feladatot hívjuk direkt geometriai feladatnak.

6.1. Rotációs mátrix

Ahhoz, hogy később a csuklók állásából ki tudjuk számolni a szerszámkoordinátát, először meg kell értenünk, hogy egy $O'x'y'z'$ elforgatott koordináta-rendszerben értelmezett P' pont hogyan adható meg $Oxyz$ koordináta-rendszerben. O és O' origó pontok, illetve P és P' pontok is egybe esnek, csak nem ugyan abban a koordináta-rendszerben vannak értelmezve. (Lásd 15. ábra)



15. ábra: P pont az elforgatott koordináta-rendszerben

P és P' pontok pozícióját a p , illetve p' helyvektorokkal írhatjuk, ahogy az (1)-es egyenletben látható.

$$p = \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \quad p' = \begin{bmatrix} \alpha' \\ \beta' \\ \gamma' \end{bmatrix} \quad (1)$$

$Oxyz$ koordináta rendszer x, y, z tengelyeinek irányát kijelölő egységvektorok i, j és k , míg $O'x'y'z'$ -hez tartozóak i', j' és k' . Ezeknek, mivel mindegyik egységvektor az abszolút értékük megegyezik. (Lásd (2) egyenletet)

$$|i| = |j| = |k| = |i'| = |j'| = |k'| = 1 \quad (2)$$

A p és p' helyvektorok az $Oxyz$, illetve $O'x'y'z'$ koordináta rendszerek egységvektoraival felírva. (Lásd (3) egyenlet)

$$p = \alpha \cdot i + \beta \cdot j + \gamma \cdot k \quad (3a)$$

$$p' = \alpha' \cdot i' + \beta' \cdot j' + \gamma' \cdot k' \quad (3b)$$

Ha ismerjük p' -t és szeretnénk megkapni p -t, akkor először be kell vezetnünk e_{ab} jelölést, ez jelöli az a egységvektor vetületét b irányba. Ezt felhasználva már fel tudjuk írni $O'x'y'z'$ egységvektorait $Oxyz$ koordináta-rendszerben. (Lásd (4) egyenlet)

$$\begin{aligned} i' &= e_{i'i} \cdot i + e_{i'j} \cdot j + e_{i'k} \cdot k \\ j' &= e_{j'i} \cdot i + e_{j'j} \cdot j + e_{j'k} \cdot k \\ k' &= e_{k'i} \cdot i + e_{k'j} \cdot j + e_{k'k} \cdot k \end{aligned} \quad (4)$$

Ezután be tudunk helyettesíteni a p' helyvektort leíró (3b) egyenletbe i' , j' és k' helyére. (Lásd (5) egyenlet)

$$\begin{aligned} p' &= \alpha' \cdot (e_{i'i} \cdot i + e_{i'j} \cdot j + e_{i'k} \cdot k) + \\ &+ \beta' \cdot (e_{j'i} \cdot i + e_{j'j} \cdot j + e_{j'k} \cdot k) + \\ &+ \gamma' \cdot (e_{k'i} \cdot i + e_{k'j} \cdot j + e_{k'k} \cdot k) \end{aligned} \quad (5)$$

Ebből az egyenletből ki tudjuk emelni i -t, j -t és k -t. (Lásd (6) egyenlet)

$$\begin{aligned} p' &= (\alpha' \cdot e_{i'i} + \beta' \cdot e_{j'i} + \gamma' \cdot e_{k'i}) \cdot i + \\ &+ (\alpha' \cdot e_{i'j} + \beta' \cdot e_{j'j} + \gamma' \cdot e_{k'j}) \cdot j + \\ &+ (\alpha' \cdot e_{i'k} + \beta' \cdot e_{j'k} + \gamma' \cdot e_{k'k}) \cdot k \end{aligned} \quad (6)$$

Ez után ki tudjuk emelni α -t β -t és γ -t, így megkapva a rotációs mátrixot. Ennek a mátrixnak az oszlopai sorban az i' , j' és k' egységvektorok i , j és k irányú vetületeit jelölik ki. (Lásd (7) egyenlet)

$$\begin{bmatrix} p \\ \alpha' \\ \beta' \\ \gamma' \end{bmatrix} = \begin{bmatrix} \alpha' \cdot e_{i'i} + \beta' \cdot e_{j'i} + \gamma' \cdot e_{k'i} \\ \alpha' \cdot e_{i'j} + \beta' \cdot e_{j'j} + \gamma' \cdot e_{k'j} \\ \alpha' \cdot e_{i'k} + \beta' \cdot e_{j'k} + \gamma' \cdot e_{k'k} \end{bmatrix} = \overbrace{\begin{bmatrix} e_{i'i} & e_{j'i} & e_{k'i} \\ e_{i'j} & e_{j'j} & e_{k'j} \\ e_{i'k} & e_{j'k} & e_{k'k} \end{bmatrix}}^{\text{Rotációs mátrix}} \cdot \begin{bmatrix} p' \\ \alpha' \\ \beta' \\ \gamma' \end{bmatrix} \quad (7)$$

A fenti egyenletből jól látható, hogy a rotációs mátrix egy ortogonális mátrix, hiszen az mátrix oszlopaiból képzett vektorok páronként merőlegesek egymásra és egységnyi hosszúságúak. Ez egyben azt is mutatja, hogy rotációs mátrixok az orientációt redundáns módon írják le. Hiszen a mátrix 9 elem közül csak 3 adható meg szabadon.

Az imént megkaptuk $R_{Oxyz, O'x'y'z'}$ rotációs mátrixot. Amennyiben egy $O'x'y'z'$ -ben értelmezett pontot, ez esetben p' -t megszorozzuk ezzel a mátrixal, úgy megkapjuk a helyzetét $Oxyz$ -ben, vagyis p -t.

Ez viszont egyelőre csak egy irányú művelet. p helyvektort ismerve nem tudjuk megkapni p' -t. Ahhoz, hogy visszafelé is képesek legyünk forgatni, szükségünk van R transzformációs mátrix inverzére. (Lásd (8) egyenlet)

$$p' = R^{-1} \cdot p \quad (8)$$

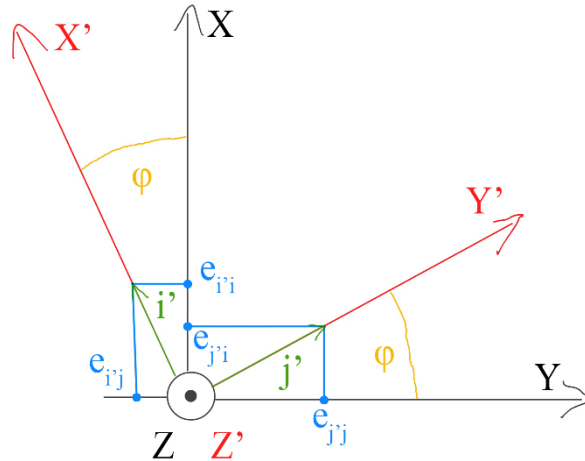
Itt fel tudjuk használni a rotációs mátrixoknak egy fontos tulajdonságát, azt, hogy ortogonális mátrixok. Egy mátrix transzponáltját pedig egyszerűen elő tudjuk állítani, ha a sorait és az oszlopait megcseréljük. (Lásd (9) egyenlet)

$$p' = R^T \cdot p \quad (9)$$

6.2. Roll, Pitch, Yaw forgatás

Vannak úgynevezett Elemi forgatások. Ide azokat a forgatásokat csoportosítjuk, amik a koordináta-rendszer egy-egy nevezetes tengelye körül forgatnak (x, y, z) . Ilyenkor a rotációs mátrix 9 eleme közül 4 irányba 0 a vetület, amelyik tengely körül forgatunk, annak az egység vektora a képével egybeeső így az 1, a másik 4 vetület pedig sinus és cosinus függvények segítségével számolható ki.

z tengely körüli forgatás: (Lásd 16. ábra, (10) egyenlet)

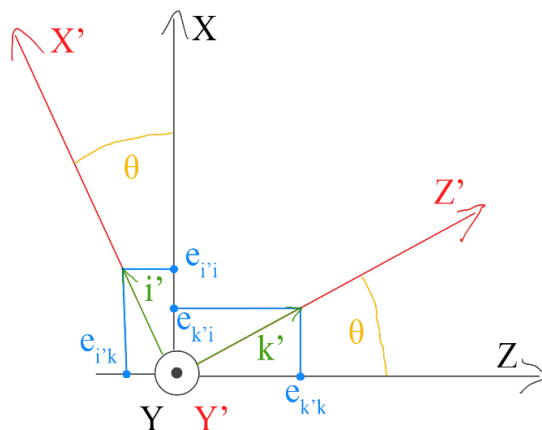


16. ábra: Elemi forgatás z tengely körül

$$i' = \begin{bmatrix} \cos \varphi \\ \sin \varphi \\ 0 \end{bmatrix} \quad j' = \begin{bmatrix} -\sin \varphi \\ \cos \varphi \\ 0 \end{bmatrix} \quad k' = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad (10)$$

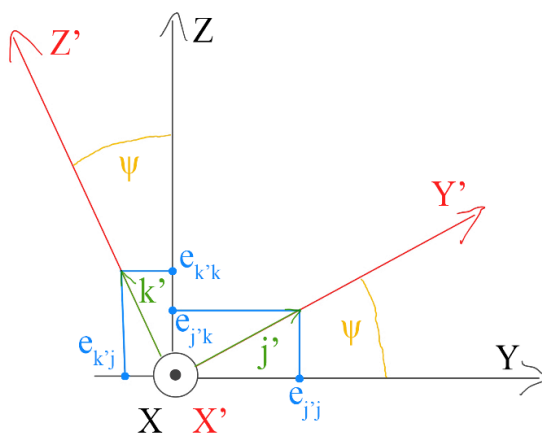
$$R_{z,\varphi} = \begin{bmatrix} \cos \varphi & -\sin \varphi & 0 \\ \sin \varphi & \cos \varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

y tengely körüli forgatás: (Lásd 17. ábra, (11) egyenlet)

17. ábra: Elemi forgatás y tengely körül

$$\begin{aligned}
 i' &= \begin{bmatrix} \cos \theta \\ 0 \\ -\sin \theta \end{bmatrix} & j' &= \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} & k' &= \begin{bmatrix} \sin \theta \\ 0 \\ \cos \theta \end{bmatrix} \\
 R_{y,\theta} &= \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix}
 \end{aligned} \tag{11}$$

x tengely körüli forgatás: (Lásd 18. ábra, (12) egyenlet)

18. ábra: Elemi forgatás x tengely körül

$$\begin{aligned}
 i' &= \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} & j' &= \begin{bmatrix} 0 \\ \cos \psi \\ \sin \psi \end{bmatrix} & k' &= \begin{bmatrix} 0 \\ -\sin \psi \\ \cos \psi \end{bmatrix} \\
 R_{x,\alpha} &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \psi & -\sin \psi \\ 0 & \sin \psi & \cos \psi \end{bmatrix}
 \end{aligned} \tag{12}$$

A térbeli orientáció learás egyik alapvető módszere az Euler szögek használata. Ennek az a lényege, hogy a térbeli forgatást 3 elemi forgatással írja le. Erre egy példa a R_{ZYZ} leírási mód. Ez esetben először a z tengely körül, majd az elforgatott y tengely körül végül pedig ismét az elforgatott z tengely körül forgatunk. Ilyenkor a térbeli orientációt 3 szögértékkel tudjuk megadni. Ennél a leírási módnál fontos, hogy a forgatások sorrendje nem kommutatív, ha más sorrendben hajtjuk végre őket akkor összességében egy másik orientációt fogunk leírni.

Gyakorlatban legtöbbször használt Euler szöges elforgatás a Roll, Pitch, Yaw (Későbbiekben RPY forgatás). Ez esetben először x tengely körül forgatunk ψ -vel ez a roll, majd y körül θ -vel, ez a pitch, végül pedig z körül φ -vel, ez pedig a yaw. (Lásd (13) egyenlet)

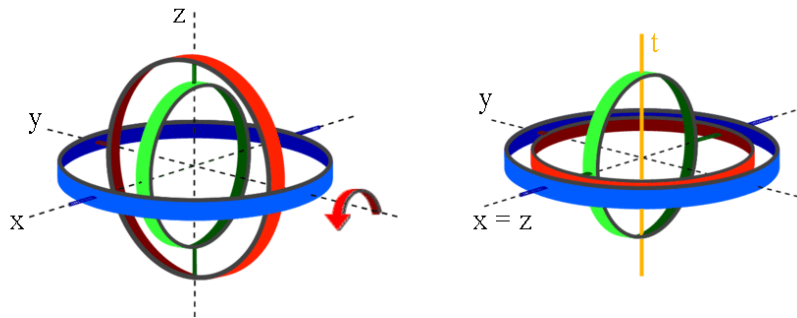
A továbbiakban bevezetem az $s\theta = \sin \theta$ és a $c\theta = \cos \theta$ jelöléseket. Erre a nagyobb egyenletek esetében van szükség.

$$R_{xyz} = R_{z,\varphi} \cdot R_{y,\theta} \cdot R_{x,\psi}$$

$$R_{xyz} = \begin{bmatrix} c\varphi & -s\varphi & 0 \\ s\varphi & c\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} c\theta & 0 & s\theta \\ 0 & 1 & 0 \\ -s\theta & 0 & c\theta \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & c\psi & -s\psi \\ 0 & s\psi & c\psi \end{bmatrix} = \quad (13)$$

$$\begin{bmatrix} c\varphi \cdot c\theta & -s\varphi \cdot c\psi + c\varphi \cdot s\theta \cdot s\psi & s\varphi \cdot s\psi + c\varphi \cdot s\theta \cdot c\psi \\ s\varphi \cdot c\theta & c\varphi \cdot c\psi + s\varphi \cdot s\theta \cdot s\psi & -c\varphi \cdot s\psi + s\varphi \cdot s\theta \cdot c\psi \\ -s\theta & c\theta \cdot s\psi & c\theta \cdot c\psi \end{bmatrix}$$

Az RPY forgatások egyszerűen és könnyen elképzelhetően írják le a térbeli elforgatást, de nem minden esetben működnek jól. Például abban az esetben, ha az egyik tengely körül 90° -al elforgatunk (19. ábra), előfordulhat, hogy két tengely egy vonalba kerül és így elveszítünk egy szabadsági fokot. Ilyen helyzetben numerikus problémák miatt a rotációs mátrixból nem számíthatók ki az RPY szögek egyértelműen. Ezt a jelenséget gimbal lock-nak nevezzük. A 19. ábra jól szemlélteti. y tengely körül elforgatunk 90° -kal, így jobb oldalon látható, hogy x engely egy vonalba esik z tengellyel. Ezzel elveszítettünk egy szabadságfokot, mert az ábrán t -vel jelölt tengely körül nem vagyunk képesek forgatást végezni.



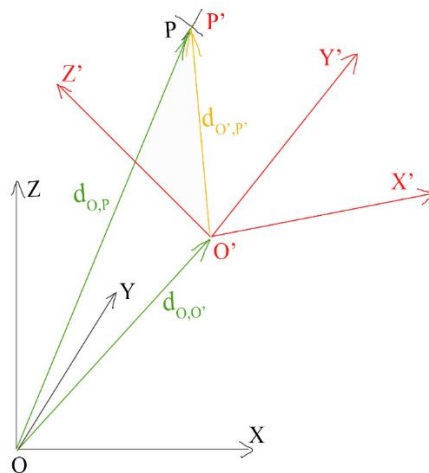
19. ábra: Gimbal Lock [10]

A gimbal lock matematikájának megértéséhez szükségünk van rá, hogy megismerjük az atan2 függvényt. Ez egy a programozásban és a matematikában is használt függvény. Az arctan függvény átalakított verziója. Két egyenes által bezárt szöget tudunk vele meghatározni, radiánban, akár csak az arctan függvénnyel. Az az előnye, az arctan-al szemben, hogy míg az arctan csak $-\frac{\pi}{2}$ és $+\frac{\pi}{2}$ közötti értéket tud visszaadni, addig az atan2 $-\pi$ és π közötti értéket ad vissza, így le tudjuk fedni a teljes 360° -os tartományt, ami a robotikában a 3D-s térben elég fontos. A másik előnye, hogy míg arctan függvény $x = 0$ esetén nincs definiálva, addig az atan2 ezeket az eseteket képes automatikusan lekezelni. A (14) egyenletben látható, hogy x és y különböző értékei esetén, hogyan számítható ki a függvény visszatérési értéke. Gimbal lock esetén az x és az y értéke is 0, ami azért probléma mert erre az esetre nincs definiálva az atan2 függvény.

$$\text{atan2}(y, x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & \text{ha } x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & \text{ha } x < 0 \text{ és } y \geq 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & \text{ha } x < 0 \text{ és } y < 0 \\ +\frac{\pi}{2} & \text{ha } x = 0 \text{ és } y > 0 \\ -\frac{\pi}{2} & \text{ha } x = 0 \text{ és } y < 0 \\ \text{Nem definiált} & \text{ha } x = 0 \text{ és } y = 0 \end{cases} \quad (14)$$

6.3. Homogén transzformációk

A robotmodellezés során nem csak elforgatni kell a koordinátarendszereket, miközben az origó pontjuk változatlan marad, hanem eltolásokat is kell végezni. (Lásd 20. ábra)



20. ábra: P pont helyzete az eltolt és elforgatott koordinátarendszerben

P pont nem írható le $d_{O,O'}$ és $d_{O',P'}$ vektorok összegeként, mert nem ugyan abban a koordinátarendszerben vannak értelmezve. Ehelyett két részre kell bontanunk a transzformációt. Először egy forgatást hajtunk végre O körül, így kapunk egy O' orientációjú koordinátarendszert O origóval, majd ezt toljuk el $d_{O,O'}$ -vel. Ezt, amikor a transzformációt egy eltolással és egy elforgatással írjuk le, rendezett párnak nevezzük.

A 20. ábra esetében a P pont helyzetét az alább látható egyenletten adhatjuk meg. (Lásd (15) egyenlet)

$$p = R_{O',O} \cdot p' + d_{O,O'} \quad (15)$$

Ezt könnyíti meg a Homogén transzformáció. Ez a művelet az eltolást és elforgatást egyetlen mátrix szorzással egyszerűsíti. Ehhez viszont be kell vezetnünk a homogén koordináták fogalmát. $p(x, y, z)$ homogén koordinátára alakítva $p(x, y, z, w)$. A w komponens a robotikában mindig 1 értéket vesz fel, csak a homogén koordináták egyéb felhasználása esetén szoktak ettől eltérő értéket használni, például képfeldolgozás során.

Az előbb bemutatott egyenlet (15) homogén transzformációval leírva a (16) egyenletben látható. Ebben az esetben p és p' homogén koordináták míg $H_{O,O'}$ egy homogén transzformációs mátrix.

$$P = H_{O,O'} \cdot P' \quad (16)$$

A homogén transzformációs mátrix általános alakja a (17) egyenletnél látható, ahol $R_{O,O'}$ az elforgatás, $d_{O,O'}$ pedig az eltolási rész.

$$H_{O,O'} = \begin{bmatrix} R_{O,O'} & d_{O,O'} \\ 0 & 1 \end{bmatrix} \quad (17)$$

Néhány példa a homogén transzformációkra. x tengely menti α -val való eltolást (18) transzformációs mátrixal végezhetjük el.

$$Trans_{x,\alpha} = \begin{bmatrix} 1 & 0 & 0 & \alpha \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (18)$$

y tengely körüli β -val való elforgatást a (19) egyenletben látható transzformációs mátrixal végezhetjük el.

$$Rot_{y,\beta} = \begin{bmatrix} \cos \beta & 0 & \sin \beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \beta & 0 & \cos \beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (19)$$

H_{R_z,T_y} transzformációs mátrix egyszerre forgat z tengely körül és eltol y tengely irányába. (Lásd (20) egyenlet)

$$H_{R_z, T_y} = \begin{bmatrix} \cos \psi & -\sin \psi & 0 & 0 \\ \sin \psi & \cos \psi & 0 & d \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (20)$$

6.4. Denavit-Hartenberg transzformáció

A Denavit-Hartenberg (D-H) konvenció az egyik legegyszerűbb és leggyakrabban használt módszer a szerszámkoordináták meghatározására, ennek lényege, hogy 2 szegmens között a Homogén transzformációkat 2 tengely menti eltolásra és 2 tengely menti elforgatásra lehet egyszerűsíteni, ha a koordinátarendszereket a megfelelő módon vesszük fel. A koordinátarendszerek felvételének 3 szabálya:

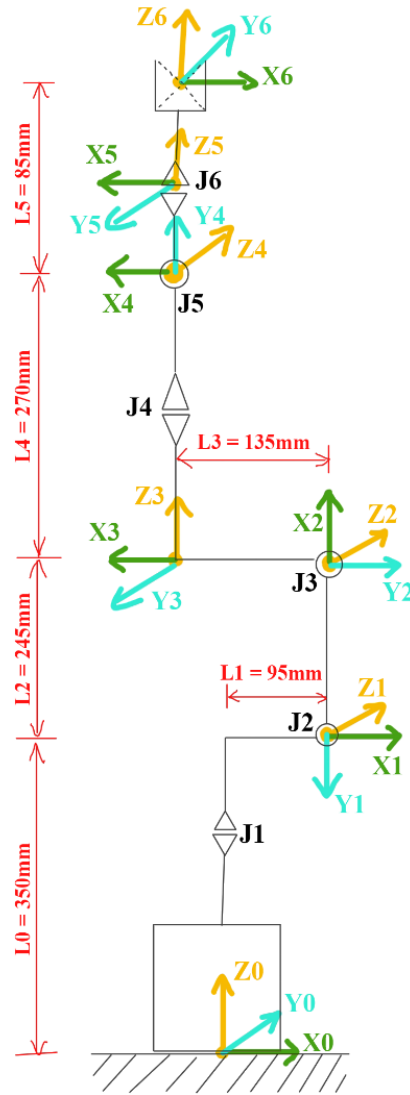
- z_i tengelyt úgy vesszük fel, hogy egybeessen J_{i+1} csukló forgástengelyével
- x_i tengelyt úgy vesszük fel, hogy z_{i-1} és z_i közös normálisába essen (mind a kettőre merőleges legyen). Amennyiben z_{i-1} és z_i egy irányba néznek úgy a J_{i+1} csukló irányába vesszük fel
- y_i tengelyt úgy vesszük fel, hogy jobbsodrású koordinátarendszert alkosson z_i -vel és x_i -vel

Ez alapján a 3 szabály alapján felvettem az RV-3SB csuklókoordináta rendszereit. (Lásd 21. ábra) A robot az itt látható pózban van, ha minden csuklószögének értéke 0.

Fontos kiemelni azt is, hogy a mi esetünkben J_1 , J_4 , és J_6 csukló pontos elhelyezkedését nem szükséges ismernünk, mert bárhol helyezkednek el a tengelyükön, az általuk végzett forgatás nem fog változni. Emiatt ezeknek a csuklóknak a koordinátarendszerét szabadon eltolhatjuk a z tengelyük mentén, ettől függetlenül ugyan úgy teljesíteni fogják a koordinátarendszerek felvételének 3 szabályát. Ezt kihasználva, praktikussági okokból el is toltam ezeket a koordinátarendszereket.

- J_1 -es csukló koordinátarendszerét eltoltam negatív irányba, hogy egybeessen a világkoordinátarendszerrel
- J_4 -es csukló koordinátarendszerét eltoltam a J_3 -as és J_4 -es csuklók közötti szegmens 90° -os töréséhez
- J_6 -os csukló koordinátarendszerét pedig eltoltam a J_5 -ös csuklóhoz

(Ezek közül az eltolások közül J_6 -os csuklóé, nem látszik az ábrán. Erre az ábra átláthatósága szempontjából volt szükség, de a D-H paraméterek felírásakor az a koordinátarendszer is eltolva lett figyelembe véve.)



21. ábra: RV-3SB felvett csukló koordinátarendszerei

Ha így vettük fel a koordináta rendszereket akkor az $i - 1$ csuklóból az i csuklóba úgy tudjuk átforgatni a koordinátarendszert, hogy az alábbi 4 műveletet végezzük el:

- θ -vel forgatunk z tengely körül;
- d -vel eltolást végzünk z tengely irányába;
- a -val eltolást végzünk x tengely irányába;
- x körül forgatunk α -val.

Ez képlettel leírva az alábbi egyenletben látszik. (21)

$$H_{i-1,i} = Rot_{z,\theta} \cdot Trans_{z,d} \cdot Trans_{x,a} \cdot Rot_{x,\alpha} \quad (21)$$

$H_{i-1,i}$ azt a Homogén transzformációs mátrixot jelöli, ami az $i - 1$ koordináta rendszerből az i koordinátarendszerbe transzformál át. A Denavit-Hartenberg Homogén transzformációs mátrix általános alakja, (22) egyenletben látható.

$$H_{i-1,i} = \begin{bmatrix} \cos \theta_i & -\sin \theta_i \cdot \cos \alpha_i & \sin \theta_i \cdot \sin \alpha_i & \alpha_i \cdot \cos \theta_i \\ \sin \theta_i & \cos \theta_i \cdot \cos \alpha_i & -\cos \theta_i \cdot \sin \alpha_i & \alpha_i \cdot \sin \theta_i \\ 0 & \sin \alpha_i & \cos \alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (22)$$

Ez alapján ezek a mátrixok előállíthatók mindössze 4 változó alapján (θ, d, a, α) . Ezt a négy értéket együtt Denavit-Hartenberg paramétereknek hívjuk. A felvett koordinátarendszerek alapján végig mentem a robot csuklókoordináta rendszerein és meghatároztam a D-H paramétereket. (Lásd 1. táblázat) (θ_1 a J_1 -es csukló 0° -tól való elfordulását jelöli)

Transzformáció	θ	d	a	α
$x_0y_0z_0 \rightarrow x_1y_1z_1$	θ_1	L_0	L_1	-90°
$x_1y_1z_1 \rightarrow x_2y_2z_2$	$-90^\circ + \theta_2$	0	L_2	0°
$x_2y_2z_2 \rightarrow x_3y_3z_3$	$-90^\circ + \theta_3$	0	L_3	-90°
$x_3y_3z_3 \rightarrow x_4y_4z_4$	θ_4	L_4	0	90°
$x_4y_4z_4 \rightarrow x_5y_5z_5$	θ_5	0	0	-90°
$x_5y_5z_5 \rightarrow x_6y_6z_6$	$180^\circ + \theta_6$	L_5	0	0°

1. táblázat: RV-3SB D-H paraméterei

Amennyiben több ilyen transzformációt szeretnénk elvégezni egymás után akkor a homogén transzformációs mátrixokat össze kell szoroznunk a (23) -es egyenletben látható módon.

$$H_{i,i+2} = H_{i,i+1} \cdot H_{i+1,i+2} \quad (23)$$

Ahhoz, hogy a robotunk esetében megkapjuk a 6. csukló koordinátarendszeréből az 1. csukló koordinátarendszerébe forgató transzformációs mátrixot, fel kell írunk minden csuklóhoz a D-H transzformációs mátrixot és össze kell szoroznunk ezeket. (Lásd (24) egyenlet)

$$H_{0,6} = H_{0,1} \cdot H_{1,2} \cdot H_{2,3} \cdot H_{3,4} \cdot H_{4,5} \cdot H_{5,6} \quad (24)$$

Ha a 6 csukló koordinátarendszerében értelmezett P pontot megszorozzuk $H_{0,6}$ transzformációs mátrixal akkor megkapjuk P pont koordinátáit az első csukló koordinátarendszerében. [11] [12] [13] [14]

6.5. Denavit-Hartenberg transzformáció implementálása

Készítettem egy metódust, ami a D-H paramétereket megkapja egy 4 elemű float tömbben és abból előállítja az azokhoz a paraméterekhez tartozó transzformációs mátrixot, ezt a mátrixot pedig visszaadja visszatérési értéként. (Lásd 22. ábra)

```
//DH transzformációs mátrix előállítás a DH paraméterekből
public Matrix<float> Matrix_eloallitas(float[] h)
{
    //DH paraméterek kiszedése egyéni változókba
    float theta1 = h[0];
    float d1 = h[1];
    float a1 = h[2];
    float alfa1 = h[3];

    //Szögek szinuszának és koszinuszának kiszámolása
    float cos_theta1 = Mathf.Cos(theta1 * Mathf.PI / 180);
    float sin_theta1 = Mathf.Sin(theta1 * Mathf.PI / 180);
    float cos_alfa1 = Mathf.Cos(alfa1 * Mathf.PI / 180);
    float sin_alfa1 = Mathf.Sin(alfa1 * Mathf.PI / 180);

    //Denavit-Hartenberg transzformációs mátrix létrehozása az általános képlettel a D-H paraméterekből
    Matrix<float> A = Matrix<float>.Build.DenseOfArray(new float[,]
    {
        {cos_theta1, -sin_theta1*cos_alfa1, sin_theta1*sin_alfa1, a1*cos_theta1},
        {sin_theta1, cos_theta1*cos_alfa1, -cos_theta1*sin_alfa1, a1*sin_theta1},
        {0, sin_alfa1, cos_alfa1, d1},
        {0, 0, 0, 1}
    });

    return A;
}
```

22. ábra: Transzformációs mátrix előállító metódus

Létrehoztam egy „RobotMozog” metódust ami a „Beáll” gombra kattintás esetén fut le. Ebben megadtam változókkal a robot különböző paramétereit. (Lásd 23. ábra) Ezt azért így csináltam, mert így a kód könnyen hozzáigazítható egy másik robothoz is és könnyen elkészíthető a teljes szimulációs program egy más paraméterekkel rendelkező robottal is.

```
//Szegmensek méreteinek megadása változókba
float L0 = 350;
float L1 = 95;
float L2 = 245;
float L3 = 135;
float L4 = 270;
float L5 = 85;

//A csuklókoordináták beolvasása a UI-ról meghívott InputFieldekből változókba
float theta1 = float.Parse(jointText_q1.text);
float theta2 = float.Parse(jointText_q2.text);
float theta3 = float.Parse(jointText_q3.text);
float theta4 = float.Parse(jointText_q4.text);
float theta5 = float.Parse(jointText_q5.text);
float theta6 = float.Parse(jointText_q6.text);

//Itt kerülnek megadásra a robot DH paraméterei, ezeket akkor kell változtatni ha másik robot kerül a szimulációba
//Ezekből kerül kiszámolásra a D-H transzformációs mátrix
float[] h0_1 = new float[4] { theta1, L0, L1, -90 };
float[] h1_2 = new float[4] { -90 + theta2, 0, L2, 0 };
float[] h2_3 = new float[4] { -90 + theta3, 0, L3, -90 };
float[] h3_4 = new float[4] { theta4, L4, 0, 90 };
float[] h4_5 = new float[4] { theta5, 0, 0, -90 };
float[] h5_6 = new float[4] { 180+theta6, L5, 0, 0 };
```

23. ábra: Robot alap paramétereinek megadása, csuklósőzögek beolvasása

A paraméterek megadása után az általam készített mátrix előállító metódusnak átadtam ezeket az értékeket, így elkészítettem az összes csuklóból a következő csuklóba transzformáló mátrixokat, majd ezeket a megfelelő sorrendben összeszoroztam, hogy megkapjam a $H_{0,6}$ mátrixot. (Lásd 24. ábra)

```
//Minden szegmensből a következőbe transzformáló mátrix létrehozása
Matrix<float> H0_1 = Matrix_eloallitas(h0_1);
Matrix<float> H1_2 = Matrix_eloallitas(h1_2);
Matrix<float> H2_3 = Matrix_eloallitas(h2_3);
Matrix<float> H3_4 = Matrix_eloallitas(h3_4);
Matrix<float> H4_5 = Matrix_eloallitas(h4_5);
Matrix<float> H5_6 = Matrix_eloallitas(h5_6);

//A szegmensek közötti transzformációs mátrixok összeszorozása megfelelő sorrendben, hogy megkapjuk a H0_6 mátrixot
Matrix<float> H0_2 = H0_1 * H1_2;
Matrix<float> H0_3 = H0_2 * H2_3;
Matrix<float> H0_4 = H0_3 * H3_4;
Matrix<float> H0_5 = H0_4 * H4_5;
Matrix<float> H0_6 = H0_5 * H5_6;
```

24. ábra: Metódusok meghívása, $H_{0,6}$ mátrix előállításához

Ahogy a Homogén transzformációk fejezetben leírtam a Homogén transzformációs mátrixnak van egy része, ami az eltolásért felelős. Így a szerszám koordináta-rendszer világkoordináta-rendszerben elfoglalt pozíciója kiolvasható a $H_{0,6}$ mátrixból. Ezt kiolvasom, majd átadom a szerszámkoordinátát jelképező GameObject-nek (ezt később részletesebben tárgyalom), így az ide kerül mozgatásra. Emellett a kiolvasott koordinátákat kiíratom a UI-ra is. (Lásd 25. ábra)

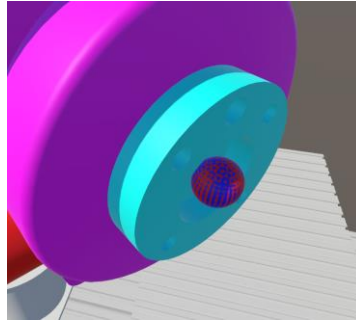
```
//Roll-Pitch-Yaw szögek kiszámolása (Euler szögek)
//Ez a D-H transzformációs mátrix megfelelő elemeiből történik
float rot_z = Mathf.Rad2Deg * Mathf.Atan2(H0_6[1, 0], H0_6[0, 0]);
float rot_y = Mathf.Rad2Deg * Mathf.Asin(-H0_6[2, 0]);
float rot_x = Mathf.Rad2Deg * Mathf.Atan2(H0_6[2, 1], H0_6[2, 2]);

//A kiszámolt szerszám koordináták átadása a GameObject-nek, hogy abba a pozícióba legyen mozgatva
Vector3 szerszam_krsz = new Vector3((float)Math.Round(H0_6[0,3], 3), (float)Math.Round(-H0_6[1,3], 3), (float)Math.Round(H0_6[2,3], 3));
szamoltszerszam.transform.localPosition = szerszam_krsz;

//A kiszámolt szerszám koordináták illetve orientáció 3 tizedes jegyre történő kerekítése
//Majd átadása a UI-ra, hogy ott megjelenítésre kerüljenek
szamoltszerszam_x.text = Math.Round(H0_6[0, 3], 3).ToString();
szamoltszerszam_y.text = Math.Round(-H0_6[1, 3], 3).ToString();
szamoltszerszam_z.text = Math.Round(H0_6[2, 3], 3).ToString();
szamoltszerszam_rz.text = Math.Round(rot_z, 3).ToString();
szamoltszerszam_ry.text = Math.Round(rot_y, 3).ToString();
szamoltszerszam_rx.text = Math.Round(rot_x, 3).ToString();
```

25. ábra: Szerszámkoordináták kiolvasása, jelölő gömb pozícióba mozgatása és koordináták kiírása

Ahogy már írtam, ahhoz, hogy vizuálisan is reprezentáljam a kiszámolt szerszámkoordinátákat, létrehoztam egy kis méretű gömb objektumot, aminek a középpontját a RobotMozog metódus minden lefutása esetén a szerszámkoordináták pozíciójába helyezek. (Lásd 26. ábra)



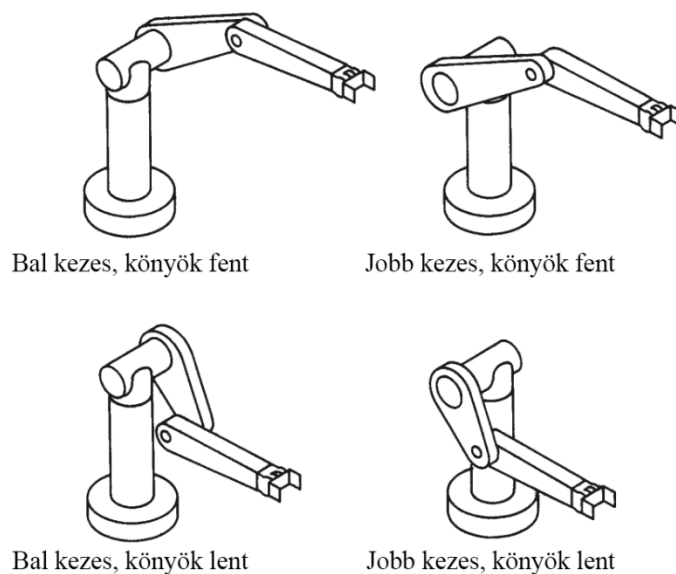
26. ábra: Szerszámkoordinátákat ábrázoló gömb

7. Inverz geometriai feladat megoldása

A dolgozatom másik része, hogy ne csak a csuklózókat tudja megadni a felhasználó és abból számolja ki a program, hogy hogy áll a robot és hol van a szerszámkoordinátája, hanem lehetőség legyen megadni a kívánt szerszám koordinátákat és ahhoz álljon pozícióba a robot.

7.1. Robot konfigurációk

Ahogy már írtam az inverz kinematika nem egy egyértelmű feladat, ezért egy szerszám pozíció nem definiálja egyértelműen a robotunk helyzetét, ugyan ahhoz a szerszám pozícióhoz többféleképpen is el tudunk jutni, így különböző konfigurációkat különböztethetünk meg azonos szerszám póz esetén. (Lásd 27. ábra)

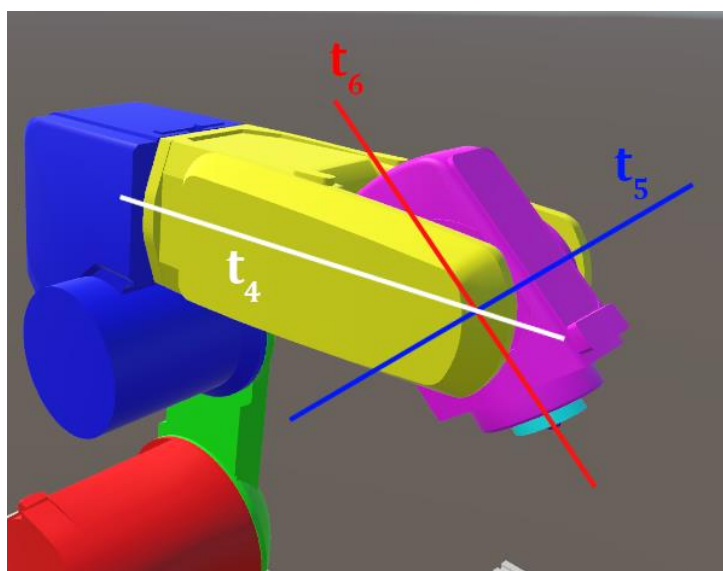


27. ábra: Robot konfigurációk

A robot programozása során, illetve a szimulációs programokban lehetőség van a kívánt megközelítendő pont felvitele után kiválasztani, hogy azt milyen konfigurációból közelítse meg a robot. Nem minden pont közelíthető meg az összes konfigurációval.

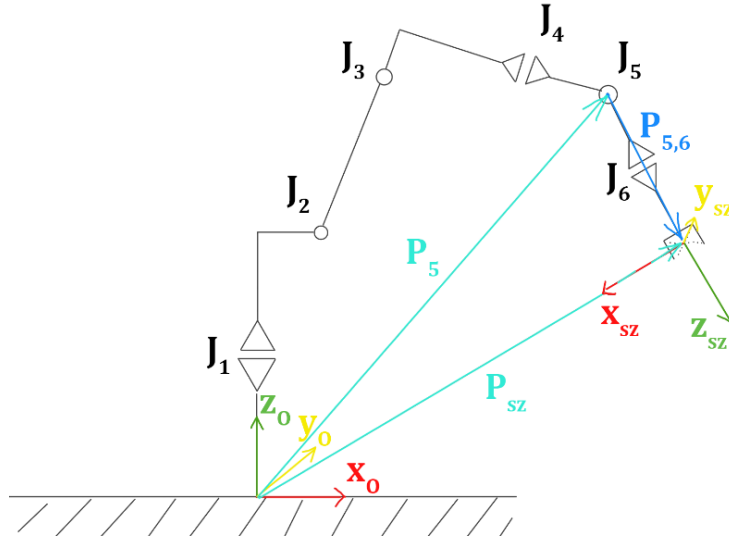
7.2. Pozíció - orientáció leválasztás

Gömbcsuklós szerkezettel rendelkező, illetve humanoid robotok esetén a pozícióért és az orientációért felelős csuklók szétválaszthatók. Ez az RV-3SB robot esetében azért lehetséges mert a 4-es, 5-ös és 6-os csuklók tengelyei egy pontban metszik egymást (Lásd 28. ábra), így ezzel a 3 csuklóval elő lehet állítani az orientációt.



28. ábra. ábra: Euler Kézcsukló Szerkezet

Az inverz geometriában ez a tulajdonság nagyon sokat segít nekünk. Így az inverz geometriai feladat szétválasztható egy inverz pozíció és egy inverz orientáció részfeladatra. (Lásd 29. ábra)



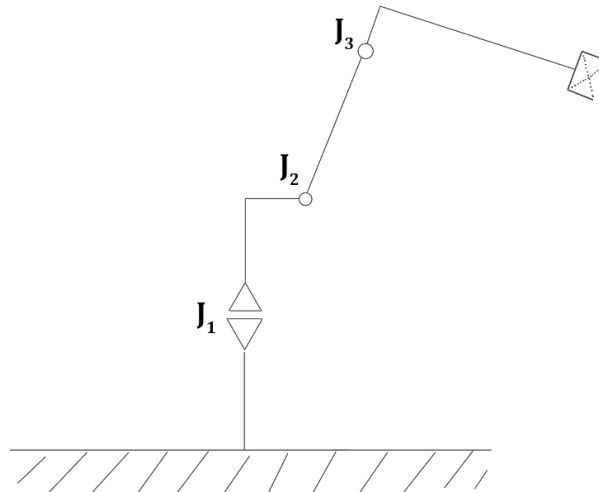
29. ábra: Pozíció leválasztás

P_5 -öt, úgy tudjuk kiszámolni, hogy a kívánt szerszámkoordinátákból kivonjuk a J_5 csukló és a szerszám középpont közti távolságot a szerszám z tengelyének irányába. Ehhez P_{sz} -t ismerjük, $P_{5,6}$ abszolút értékét a D-H paraméterekből tudjuk, $R_{0,6}$ rotációs mátrixot pedig az RPY szögekből ki tudjuk számolni a „Roll, Pitch, Yaw forgatás” című alfejezetben leírt (13) egyenletből. (Lásd a (25) egyenletet)

$$P_5 = P_{sz} - P_{5,6}$$

$$P_5 = P_{sz} - |P_{5,6}| \cdot R_{0,6} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad (25)$$

A leválasztás után egy humanoid alapkonfigurációjú robotot kapunk. (Lásd 30. ábra) Ezt a P_5 pontot kéztőpontnak hívjuk, ez lesz az a pont, ahova a J_1 , J_2 és J_3 pozícióért felelő csuklókkal el kell jutnunk. J_4 csukló csak a saját tengelye körül forog, ami nem változtat a pozíción, így ezt a csuklót most elhagyhatjuk.

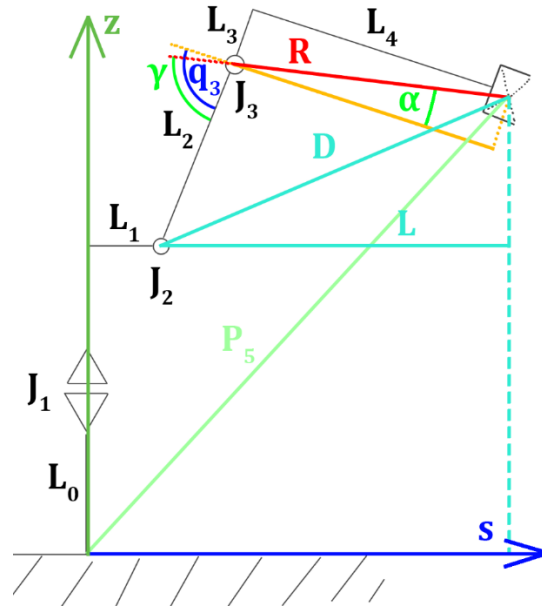


30. ábra: Leválasztás utáni humanoid alapkonzfiguráció

A csuklókoordináták kiszámolása során több esetben is elő fog fordulni, hogy az egyenleteknek több eredménye van. Ebből jól látható, hogy míg a direkt geometriai feladat mindig rendelkezik egyértelmű megoldással, az inverz geometria esetében már legtöbbször több megoldásunk is lehet, ezzel növelve a probléma komplexitását.

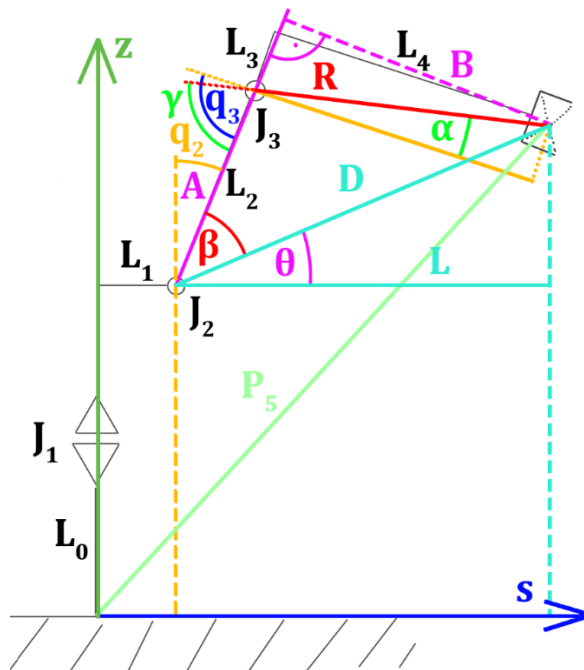
7.3. Pozíció kiszámítása

A leválasztás után már pusztán geometriai számításokkal képesek vagyunk meghatározni, hogy milyen csuklószögekkel tudunk eljutni P_5 pontba. Először a q_1 -et határozzuk meg. Mivel felülnézetből (Lásd 31. ábra) a robot és az x tengely által bezárt szöget csak q_1 értéke határozza meg, így ezt egy egyszerű szögfüggvénnyel kiszámíthatjuk. (Lásd (26) egyenlet)



32. ábra: q_3 értékének meghatározása geometriailag

q_2 -t szintén trigonometriai függvényekkel tudjuk meghatározni, de ennek az értéke már függ q_3 -tól is. (Lásd 33. ábra, (28) egyenlet)



33. ábra: q_2 értékének meghatározása geometriailag

$$\begin{aligned}
A &= L_2 + R \cdot \cos \gamma \\
B &= R \cdot \sin \gamma \\
L &= \sqrt{P_{5x}^2 + P_{5y}^2} - L_1 \\
\beta &= \text{atan2}(A, B) \\
\theta &= \text{atan2}(P_{5z} - L_0, L) \\
q_2 &= 90^\circ - \beta - \theta
\end{aligned} \tag{28}$$

7.4. Orientáció kiszámolása

Miután kiszámoltuk a q_1 , q_2 és q_3 értékét, már meg tudjuk határozni az orientációért felelős csuklók csuklósögét is. A teljes $R_{0,6}$ elforgatást a felhasználó által megadott RPY szögekből ki lehet számolni. Az $R_{0,3}$ -at, vagyis a leválasztott részen történő elforgatást is ki lehet számolni D-H módszerrel. Az $R_{3,6}$ elforgatást ebből a kettőből ki tudjuk számolni. (Lásd (29) egyenlet)

$$\begin{aligned}
R_{0,6} &= R_{0,3} \cdot R_{3,6} \\
R_{3,6} &= (R_{0,3})^{-1} \cdot R_{0,6} = (R_{0,3})^T \cdot R_{0,6}
\end{aligned} \tag{29}$$

Az $R_{3,6}$ mátrix parametrikusan felírva ugyan az, mint a z, x, z Euler féle elforgatás (Lásd (30)). Ebből tudjuk előállítani q_4 , q_5 és q_6 értékeket.

$$\begin{aligned}
R_{zxz} &= R_{z,\varphi} \cdot R_{x,\psi} \cdot R_{z,\theta} = \\
&\begin{bmatrix} c\varphi \cdot c\theta - c\psi \cdot s\varphi \cdot s\theta & -c\varphi \cdot c\theta - c\psi \cdot c\theta \cdot s\varphi & s\varphi \cdot s\psi \\ c\theta \cdot s\varphi - c\varphi \cdot c\psi \cdot s\theta & c\varphi \cdot c\psi \cdot c\theta - s\varphi \cdot s\theta & -c\varphi \cdot s\psi \\ s\psi \cdot s\theta & c\theta \cdot s\psi & c\psi \end{bmatrix}
\end{aligned} \tag{30}$$

Új jelölésként bevezetem, hogy $R_{3,6}[2,3]$ jelöli az $R_{3,6}$ mátrix 2. sorának 3. elemét. (Lásd (31) egyenlet)

$$\begin{aligned}
G &= \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \\
G[1,3] &= 3 \\
G[2,1] &= 4
\end{aligned} \tag{31}$$

q_5 értékét tudjuk a legegyszerűbben előállítani az $R_{3,6}[3,3]$ -ból, mivel az $\cos \psi$ vagyis a mátrixban lévő 3 elforgatás közül a másodiknak a koszinusz értéke. Ezek alapján ki tudjuk számolni ψ -t, ahol $\pm$ előjel két különböző csukló konfigurációt jelöl. (Lásd (32) egyenlet)

$$\tan x = \frac{\sin x}{\cos x} \tag{32}$$

$$\cos^2 x + \sin^2 x \rightarrow \sin x = \pm \sqrt{1 - \cos^2 x}$$

$$q_5 = \text{atan2} \left(\frac{\pm \sqrt{1 - \cos^2(R_{3,6}[3,3])}}{R_{3,6}[3,3]} \right)$$

q_4 értéke az R_{zzz} mátrixban φ . Ezt úgy tudjuk megkapni, ha $\frac{R_{3,6}[2,3]}{-R_{3,6}[1,3]}$ osztásnak vesszük a tangensét. Ugyanis így a $\sin \psi$ -k kiejtik magukat az egyenletből, és csak $\frac{\sin \varphi}{\cos \varphi}$ marad, aminek az atan2 értéke meg fogja adni φ -t. (Lásd (33) egyenlet)

$$q_4 = \text{atan2}(R_{3,6}[2,3], -R_{3,6}[1,3]) \quad (33)$$

q_6 értéke az R_{zzz} mátrixban θ . Ezt hasonló módszerrel lehet kiszámolni mint q_4 értékét. Ez esetben $\frac{R_{3,6}[3,2]}{R_{3,6}[3,1]}$ osztást fogjuk elvégezni és ennek vesszük az atan2 értékét. (Lásd (34) egyenlet)

$$q_6 = \text{atan2}(R_{3,6}[3,2], R_{3,6}[3,1]) \quad (34)$$

7.5. Inverz geometria implementálása

A dolgozatom során most nem fordítottam figyelmet a több konfiguráció kezelésére, a szimulációban nem lehet kiválasztani, hogy a robot melyik konfigurációból közelítse meg a szerszámkoordinátákat, de előkészítettem a szimulációt, hogy később képes legyen ezeket kezelni.

Az inverz geometriáért felelős C# scriptet is úgy kezdtem, hogy meghívtam a szükséges komponenseket a UI-ról (Lásd 34. ábra). Illetve itt is felhasználtam a Denavit-Hartenberg transzformációkhoz írt mátrix előállító és mátrix szorzó metódusokat.

```
//Az InputFieldek meghívása a UI-ról
//Így bolvasható a kívánt szerszámkoordináta és orientáció illetve kiírásra kerülhet a szükséges csuklókoordináta
public InputField input_x;
public InputField input_y;
public InputField input_z;
public InputField input_rx;
public InputField input_ry;
public InputField input_rz;

public InputField input_q1;
public InputField input_q2;
public InputField input_q3;
public InputField input_q4;
public InputField input_q5;
public InputField input_q6;

public InputField output_x;
public InputField output_y;
public InputField output_z;
public InputField output_rx;
public InputField output_ry;
public InputField output_rz;

//Meghívásra kerül a legördülő menü a későbbi konfiguráció választáshoz
public Dropdown ConfSelect;
```

34. ábra: Komponensek meghívása az inverz geometria C# scriptbe

Készítettem egy „JointAngles” nevű struktúrát, ami 6db float tulajdonsággal rendelkezik. Ez vizsgálja, hogy az egyes csuklókoordináták, amiket bemenetként kap, megfellelnek-e a robot csuklóinak a korlátozásainak. Amennyiben nem, úgy az érték lekérésekor, nem azt a számon, hanem NaN értéket ad vissza, ebből látható, hogy az nem egy a robot által fizikailag megvalósítható pozíció lenne. (Lásd 35. ábra)

```
//Létrehozásra kerül egy "JointAngles" nevű struktúra
//Ez 6db float értéket képes tárolni amik az értékük beállításakor csak akkor őrzik meg a kívánt értéket
//ha az az adott csukló tartományán belüli szögérték, amennyiben nem az, úgy NaN értéket vesznek fel
struct JointAngles
{
    //Itt kerül létrehozásra a 6 float változó
    float q1;
    float q2;
    float q3;
    float q4;
    float q5;
    float q6;

    //Mindegyik q-s változóhoz egy azonos számú Q változó létrehozása
    //Ezek a Q változók felelnek azért, hogy get utasítás esetén visszaadják a q értékét
    //set esetén pedig ha a beállítani kívánt szögérték a csukló fizikai határain belül van,
    //akkor azt a szögértéket adja a q-nak, ha pedig a határokon kívül van akkor NaN értéket tárol el
    public float Q1
    {
        get { return q1; }
        set
        {
            if (value > -170 * Mathf.Deg2Rad && value <= 170 * Mathf.Deg2Rad)
            {
                q1 = value;
            }
            else
            {
                q1 = float.NaN;
            }
        }
    }
}
```

35. ábra: Inverz geometria csuklókoordinátáit tároló struktúra

Ezután készítettem egy listát, ez a program elindulásakor nem tartalmaz elemet és a program bármikor tud hozzá új elemet adni vagy elemet törölni belőle. Minden eleme ennek a listának egy JointAngles struktúrát tárol. Ebbe tudnak eltárolásra kerülni később az elérhető konfigurációk, és az UI-n lévő legördülő menü alapján ebből kerül kiválasztásra, hogy melyikbe álljon be a robot. (Lásd 36. ábra és 37. ábra)

```
//Configurations nevű lista típusú változó létrehozása a különböző konfigurációk tárolására
List<JointAngles> Configurations = new List<JointAngles>();
```

36. ábra: Konfigurációs lista létrehozása

```
//Konfiguráció választás a legördülő lista alapján majd a konfigurációhoz tartozó csuklókoordináta kiírása a UI-ra
//Ez a metódus akkor kerül futtatásra amikor változik a legördülő menüben a kiválasztott elem
public void DropDownChanged()
{
    input_q1.text = (Math.Round(Configurations[ConfSelect.value].Q1 * Mathf.Rad2Deg, 2)).ToString();
    input_q2.text = (Math.Round(Configurations[ConfSelect.value].Q2 * Mathf.Rad2Deg, 2)).ToString();
    input_q3.text = (Math.Round(Configurations[ConfSelect.value].Q3 * Mathf.Rad2Deg, 2)).ToString();
    input_q4.text = (Math.Round(Configurations[ConfSelect.value].Q4 * Mathf.Rad2Deg, 2)).ToString();
    input_q5.text = (Math.Round(Configurations[ConfSelect.value].Q5 * Mathf.Rad2Deg, 2)).ToString();
    input_q6.text = (Math.Round(Configurations[ConfSelect.value].Q6 * Mathf.Rad2Deg, 2)).ToString();
}
}
```

37. ábra: Konfiguráció kiválasztása

Ezután a program elkezd végrehajtani az inverz kinematika matematikai műveleteit. Először kiszámoljuk az RPY szögekből a rotációs mátrixot és leválasztjuk az orientációt a kéztőpontnál. (Lásd 38. ábra)

```
//Rotációs mátrix előállítása a kívánt Roll Pitch Yaw szögekből
Matrix<float> R_rpy = Matrix<float>.Build.DenseOfArray(new float[,]
{
    {c(rz)*c(ry), (-s(rz)*c(rx))+c(rz)*s(ry)*s(rx), s(rz)*s(rx)+c(rz)*s(ry)*c(rx) },
    {s(rz)*c(ry), c(rz)*c(rx)+s(rz)*s(ry)*s(rx), (-c(rz)*s(rx))+s(rz)*s(ry)*c(rx) },
    {-s(ry), c(ry)*s(rx), c(ry)*c(rx) }
});

//Hatodik szegmens hosszának eltárolása egy változóban (a robot változása esetén módosítani kell)
float d6 = 85f;

//Kéztőpont pozíció számítás (Kinematic Decoupling)
float[] P_c = new float[3] { x - d6 * R_rpy[0, 2], y - d6 * R_rpy[1, 2], z - d6 * R_rpy[2, 2] };
```

38. ábra: RPY rotációs mátrix létrehozása és kéztőpont kiszámolása

Miután meghatároztuk a kéztőpontot, a program a már tárgyalt egyenletek felhasználásával meghatározza a pozícióért felelős csuklók csuklóváltozóinak értékét. (Lásd 39. ábra, 40. ábra és 41. ábra)

```
//Első csuklókoordináta számítása egy egyszerű szögfüggvénnyel
jointAngles.Q1 = Mathf.Atan2(i * P_c[1], i * P_c[0]);
if (jointAngles.Q1 == float.NaN) continue;
```

39. ábra: q_1 meghatározása

```
//Harmadik csuklókoordináta számítása
float R = Mathf.Sqrt(Mathf.Pow(270, 2) + Mathf.Pow(135, 2));
float k = Mathf.Sqrt(Mathf.Pow(P_c[0], 2) + Mathf.Pow(P_c[1], 2));
float alfa = Mathf.Atan2(135, 270);
if (k >= 95)
{
    float D = (Mathf.Pow(k - 95, 2) + Mathf.Pow(P_c[2] - 350, 2) - Mathf.Pow(245, 2) - Mathf.Pow(R, 2)) /
        (2 * 245 * R);
    jointAngles.Q3 = Mathf.Atan2(w * Mathf.Sqrt(1 - Mathf.Pow(D, 2)), D) + alfa;
}
else
{
    float D = (Mathf.Pow(-k + 95, 2) + Mathf.Pow(P_c[2] - 350, 2) - Mathf.Pow(245, 2) - Mathf.Pow(R, 2)) /
        (2 * 245 * R);
    jointAngles.Q3 = -(Mathf.Atan2(w * Mathf.Sqrt(1 - Mathf.Pow(D, 2)), D) - alfa);
}
if (jointAngles.Q3 == float.NaN) continue;
```

40. ábra: q_3 meghatározása

```
//Második csuklókoordináta számítása
jointAngles.Q2 = Mathf.PI / 2 - Mathf.Atan2(R * Mathf.Sin(jointAngles.Q3 - alfa),
    R * Mathf.Cos(jointAngles.Q3 - alfa) + 245) - Mathf.Atan2(P_c[2] - 350, k - 95);
if (jointAngles.Q2 == float.NaN) continue;
```

41. ábra: q_2 meghatározása

Végző lépésként kiszámolásra kerül az $R_{0,3}$ mátrix inverze, az $R_{3,6}$ mátrix, majd ez alapján a q_4 , q_5 és q_6 csuklózváltozók. (Lásd 42. ábra)

```
//Transzponálás elvégzése majd R_rpy mátrix szorzása az inverz H0_3 mátrixal
Matrix<float> R0_3 = H0_3.SubMatrix(0, 3, 0, 3);
Matrix<float> R0_3_T = R0_3.Transpose();
Matrix<float> R3_6 = R0_3_T * R_rpy;

//4, 5 é 6 csukló csuklózváltozóinak kiszámolása
for (int o = 1; o >= -1; o -= 2)
{
    jointAngles.Q5 = Mathf.Atan2(o * Mathf.Sqrt(1 - Mathf.Pow(R3_6[2, 2], 2)), R3_6[2, 2]);
    if (jointAngles.Q5 == float.NaN) continue;
    jointAngles.Q4 = (Mathf.Atan2(-o * R3_6[1, 2], o * -R3_6[0, 2])); // + Mathf.PI;
    if (jointAngles.Q4 == float.NaN) continue;
    jointAngles.Q6 = Mathf.Atan2(o * R3_6[2, 1], o * (-R3_6[2, 0]));
    if (jointAngles.Q6 == float.NaN) continue;
```

42. ábra: Orientációért felelős csuklók, csuklózváltozóinak kiszámolása

8. Összegzés és továbbfejlesztési lehetőséget

Ahogy én látom nagy jövő áll a szimulációs programok előtt, számtalan dolgot lehet velük kivitelezni, amit nélkülük csak magas költségek árán lehetne. Ahogy az ipari robotok egyre szélesebb körben terjednek el, úgy egyre több lehetőség nyílik a velük kapcsolatos kutatásokra is. Jelen dolgozatomban a Mitsubishi RV-3SB geometriai modellezését végeztem el, amelyet Unity környezetben implementáltam is. Az elkészített szimulációs szoftver azalábbi fontos funkciókkal rendelkezik:

- A program az elkészített felhasználói felületről képes adatokat fogadni a felhasználotól;
- A felhasználó megadhat csukló szögeket a kezelő felületen és a robot a kijelölt pózba áll, egyidejűleg az D-H transzformációval kiszámításra kerülnek a szerszám koordináták, amely szintén megjelenik a felhasználó számára;
- A kezelő felületen megadhatók szerszám koordináták is közvetlenül. Ezesetben a bemutatott szétcsatolós módszerrel a program meghatározza a szükséges csukló koordinátákat, a megadott szerszámpóz megvalósításához, és a kijelölt pózba állítja a robot.

Ezt a dolgot számos irányba lehet tovább fejleszteni, valamint a szimulációs programot is többféle módon lehet bővíteni és finomítani.

Az egyik fejlesztési irány lehet az inverz geometriai számítások pontosítása és bővítése a több konfiguráció támogatása érdekében. Ezáltal a program képes lenne különböző robotkonfigurációk kezelésére, ami nagyban hozzájárulhat a sokoldalúbb és valóságosabb szimulációhoz. Emellett lehetőség van a kinematikai és dinamikai modellek implementálására is, amelyek révén nemcsak egy statikus, helyhez kötött robotot látnánk, hanem valós mozgásokat és mozgáspályákat is tudnánk szimulálni.

Egy további lépés lehet a szimuláció összekapcsolása a robotvezérlővel, aminek segítségével a robot aktuális pozíciója valós időben követhető lenne a szimulációban. Ezzel egyidejűleg lehetőség nyílna a robot irányítására a szimulációs környezetben, ami kiváló eszköz lehetne a gyakorlati alkalmazások teszteléséhez és fejlesztéséhez. Az ilyen irányú fejlesztések nemcsak az ipari robotok működésének jobb megértését segítik elő, hanem hozzájárulnak ahhoz is, hogy a robotok egyre inkább integrálhatók legyenek a különféle ipari és kutatási alkalmazásokba. Továbbá lehetőséget nyújt komplexebb VR vagy AR alkalmazások fejlesztésére is.

9. Hivatkozások

- [1] V. Bence, *Fixpont iterációs módszerek alkalmazása a robotikában*, Budapest: OE-KVK, 2021.
- [2] „Unity Blog,” Unity Technologies, 14 június 2021. [Online]. Available: <https://blog.unity.com/industry/making-robots-more-accessible-with-forgeos-and-unity>.
- [3] P. Péter, *Virtuális Környezet Fejlesztése Ipari Robotkar Ütközésének*, Budapest: OE-BGK, 2023.
- [4] „Unity User Manual 2021.3 (LTS),” Unity Technologies, 2023. [Online]. Available: <https://docs.unity3d.com/Manual/index.html>.
- [5] J. K. Haas, „A History of the Unity Game Engine,” Worcester Polytechnic Institute, 2014.
- [6] „C#-dokumentáció,” Microsoft Corporation, 2024. [Online]. Available: <https://learn.microsoft.com/hu-hu/dotnet/csharp/>.
- [7] „Robots - Industrial/Collaborative Robots-MELFA Dimensions/CAD,” Mitsubishi, [Online]. Available: <https://www.mitsubishielectric.com/fa/download/cad/rbt/robot/index.html>.
- [8] N. Buuveibaatar, *Simulation software design for RV-3SB robotarm*, Budapest: OE-BGK, 2023.
- [9] „Math.NET Numerics,” [Online]. Available: <https://numerics.mathdotnet.com/>. [Hozzáférés dátuma: 25 12 2024].
- [10] J. Zeithöfler, „ResearchGate,” ResearchGate GmbH, [Online]. Available: https://www.researchgate.net/figure/illustrates-the-principle-of-gimbal-lock-The-outer-blue-frame-represents-the-x-axis-the_fig4_338835648. [Hozzáférés dátuma: 03 11 2024].
- [11] H. R. Denavit J., „A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices,” *Journal of Applied Mechanics*, %1. kötet22, pp. 215- 221, 1955.
- [12] W. K. a. J. Kleinfinger, „A new geometric notation for open and closed-loop,” 1986.
- [13] S. H. M. V. Mark W. Spong, *Robot Modeling and Control*, John Wiley & Sons, 2020.

[14] M. Gyula, Robotika, Typotex, 2011.