FA

Feb 2022 Edition

INTRODUCTION TO DRONE SIMULATION USING COPPELIASIM AND PYTHON

QAZVIN ISLAMIC AZAD UNIVERSITY

Professor:

Dr.Babak Karasfi

Authors:

Arash Mehrzadi

Azam Tehrani

Hamidreza Hematyar

https://github.com/arash-mehrzadi/Autopilot-drone

فهرست

مقدمه:

با توجه به افزایش کاربرد ربات ها در زمینه های مختلف صنعتی،پزشکی،نظامی و حتی کاربرد گسترده آن در زندگی روزمره خود ما نیاز داریم،مطالعات گسترده ای بر روی این ابزارها داشته و آزمایشاتی را در جهت بهبود کارایی آن ها بر روی آن ها انجام دهیم.از این رو شبیه ساز هایی مانند coppeliasimمی توانند به راحتی این امکان را برای ما فراهم کند تا ما بتوانیم بدون صرف هزینه روبرو شدن با پیچیدگی های ساخت و تغییر به صورت سخت افزاری در یک ربات،ایده های خود را به صورت آزمایشی ارزیابی کنیم.

شبیه ساز ربات CoppeliaSim، با محیط توسعه یکپارچه، بر اساس یک معماری کنترل توزیع شده است: هر شی/مدل را می توان به صورت جداگانه از طریق یک اسکریپت تعبیه شده، یک پلاگین، یک گره ROS، یک کلاینت API از راه دور، یا یک راه حل سفارشی کنترل کرد. این باعث می‌شود که CoppeliaSim برای برنامه‌های چند روباتی بسیار متنوع و ایده‌آل باشد.

CoppeliaSim برای توسعه الگوریتم سریع، شبیه سازی اتوماسیون کارخانه، نمونه سازی سریع و تأیید، آموزش مرتبط با روباتیک، نظارت از راه دور، بررسی دوبار ایمنی، به عنوان دوقلو دیجیتال و موارد دیگر استفاده می شود.

هدف نهایی این سند،آشنایی با نرم افزار شبیه ساز coppeliasim و محیط توسعه ی آن،نحوه اتصال این شبیه ساز به زبان های برنامه نویسی،اتصال شبیه ساز به زبان برنامه نویسی پایتون ،ایجاد و شبیه سازی پرواز یک پهباد در محیط این نرم افزار و کنترل آن به وسیله کدهای زبان پایتون می باشد.

تاریخچه:

CoppeliaSim از V-REP تکامل یافته است که در اواخر نوامبر 2019 متوقف شد.و در صنعت، آموزش و پژوهش استفاده می شود. در ابتدا در داخل توشیبا تحقیق و توسعه توسعه داده شد و در حال حاضر به طور فعال توسط Coppelia Robotics AG، یک شرکت کوچک واقع در زوریخ، سوئیس، در حال توسعه و نگهداری است.

آشنایی با coppeliasim:

CoppeliaSim محاسباتی فشرده است. CoppeliaSim با صحنه های شبیه سازی V-REP سازگار است.

برنامه شبیه سازی CoppeliaSimisarobotics که کاربر گرافیکی پیچیده ای را برای Dworld3 شبیه سازی شده غیرفیزیکی ارائه می دهد. در این شبیه‌سازی، صحنه‌ها می‌توانند به سرعت با کشیدن، رها کردن و تنظیم اشکال ابتدایی و همچنین اشیاء و مناظر از پیش تعریف‌شده برای تشکیل سلسله مراتب صحنه جمع‌آوری شوند. با فشار دادن دکمه پخش، شبیه سازی فیزیک در زمان واقعی اجرا می شود و تکامل صحنه را می توان به صورت سه بعدی مشاهده کرد. انواع مختلفی از حسگرها و مفاصل موتور مانند را می توان با اشیاء ترکیب کرد تا عوامل ربات مانندی را تشکیل دهند که می توانند در سایر بخش های ساکن صحنه حرکت کنند و با آنها تعامل داشته باشند.

CoppeliaSim دارای یک رابط برنامه نویسی کاملاً یکپارچه (API)است که به این معنی است که یک اسکریپت می تواند به هر شیء متصل شود تا آن را در شبیه سازی کنترل کند. علاوه بر این، تقریباً هر جنبه ای از شبیه سازی را می توان با استفاده از اسکریپت ها در صورت نیاز سفارشی و خودکار کرد. این اسکریپت‌ها می‌توانند با استفاده از ویرایشگر ساخته‌شده نوشته شوند، بخشی از صحنه را ذخیره می‌کنند، و معمولاً به صورت «بازخوانی» اجرا می‌شوند، که در آن توابع موجود در اسکریپت‌ها توسط شبیه‌سازی اصلی یک بار در هر مرحله زمانی شبیه‌سازی فراخوانی می‌شوند و بنابراین می‌توانند برای اجرای اقدامات سنجش و کنترل ربات استفاده شوند.

زبان برنامه نویسی CoppeliaSim Lua است، که ممکن است برای بسیاری ناآشنا باشد، اما یادگیری آن برای هر کسی که با زبان هایی مانند پایتون آشناست، آسان است.

CoppeliaSima دارای ویژگی‌های داخلی دیگری است که بسیار مفید هستند، مانند ترسیم نموداری آسان از هر نوع کمیتی.

با استفاده از CoppeliaSimwellfollowverymuchthe«ساخت از ابتدا» و «یادگیری با انجام» فلسفه آموزشی در سالهای گذشته رباتیک با استفاده از ربات ربات سخت افزاری دنبال شده است. ربات‌هایی که می‌سازیم

و آزمایش می‌کنیم، برخلاف لگو، از قطعات اصلی تشکیل می‌شوند، و الگوریتم‌هایی که روی آن‌ها کار می‌کنیم نیز از اصول اولیه ساخته می‌شوند.

اگر به شکل کلی و دقیق به coppeliasim بپردازیم؛به شرح زیر است:

CoppeliaSim یک شبیه ساز قدرتمند ربات چند پلتفرمی است که دارای نسخه آموزشی رایگان است. قدرت CoppeliaSim از چندین ویژگی ناشی می شود:

1:coppelisim یک چارچوب یکپارچه ارائه می دهد که بسیاری از کتابخانه های قدرتمند داخلی و خارجی را که اغلب برای شبیه سازی های روباتیک مفید هستند، ترکیب می کند. این شامل موتورهای شبیه‌سازی پویا، ابزارهای سینماتیک رو به جلو/ معکوس، کتابخانه‌های تشخیص برخورد، شبیه‌سازی حسگر بینایی، برنامه‌ریزی مسیر، ابزار توسعه رابط کاربری گرافیکی و مدل‌های داخلی بسیاری از روبات‌های رایج است.

2:CoppeliaSim بسیار توسعه پذیر است. توسعه دهندگان CoppeliaSim یک API ارائه می دهند که به فرد امکان می دهد افزونه های سفارشی بنویسد که ویژگی های جدیدی را اضافه می کند. می‌توانید اسکریپت‌های Lua را مستقیماً در یک صحنه شبیه‌سازی جاسازی کنید، به‌عنوان مثال، داده‌های حسگر شبیه‌سازی شده را پردازش می‌کند، الگوریتم‌های کنترلی را اجرا می‌کند، رابط‌های کاربری را پیاده‌سازی می‌کند، یا حتی داده‌ها را به یک ربات فیزیکی ارسال می‌کند. آنها همچنین یک API راه دور ارائه می کنند که به فرد امکان می دهد برنامه های کاربردی مستقل را در بسیاری از زبان های برنامه نویسی توسعه دهد که قادر به انتقال داده ها به داخل و خارج از شبیه سازی CoppeliaSim در حال اجرا هستند.

3: coppeliasim بین پلتفرمی است، عمدتا منبع باز، و مجوز آموزشی رایگان ارائه می دهد.

از جمله ویژگی های دیگر این شبیه ساز میتوان به موارد زیر اشاره کرد :

استفاده از روشهای تخمین موقعیت بازوها و لمس کننده ها

تعیین برخورد با اشیای دیگر

تعیین فاصله با اشیای دیگر بر روی صحنه

طراحی مسیر حرکت

شبیه سازی سنسورهای مجاورت

سنسورهای شبیه به دوربین

استفاده از سنسورهایی با شبیه سازی بینایی انسان

علاوه بر قابلیت های فوق، v-rep ماشینهایی را طراحی کرده است تا کاربران با قرار دادن آنها بر روی صفحه به یک ربات آماده با سنسورهایی که بر روی آنان قرار گرفته اند دسترسی پیدا کنند.از جمله این ماشین ها میتوان به ربات خِپرا مجموعه ای آماده از انواع سنسورها را به طور هماهنگ با خود به همراه دارد.

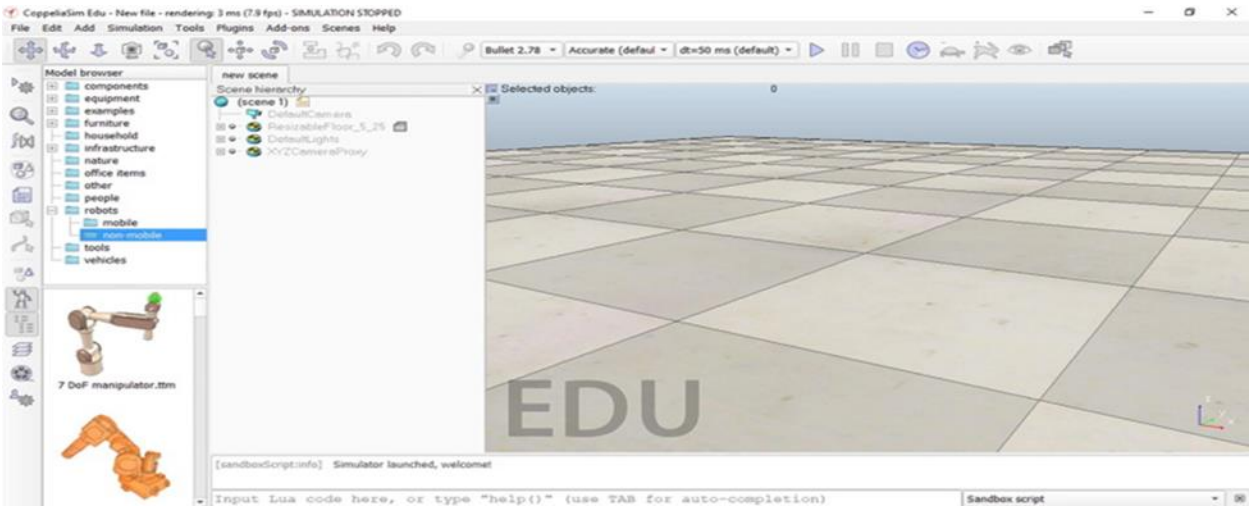
برای اجرای پروژه ها ، اولین قدم دانلود CoppeliaSim برای سیستم عامل شما خواهد بود. شما باید آخرین نسخه آموزشی نامحدود را دانلود کنید. در مرحله بعد باید CoppeliaSim را نصب کنید .

در ویندوز، شما به سادگی یک فایل EXE دارید که باید آن را اجرا کنید. در مک، ابتدا باید فایل دانلود شده را از حالت فشرده خارج کنید. دایرکتوری که از حالت فشرده تولید می شود، حاوی یک فایل coppeliaSim.app است که به شما امکان می دهد تا CoppeliaSim را از طریق مکانیسم های معمولی در مک اجرا کنید . در لینوکس، باید آرشیو فشرده tar. را از حالت فشرده خارج کنید) به عنوان مثال، با استفاده از دستوری مانند

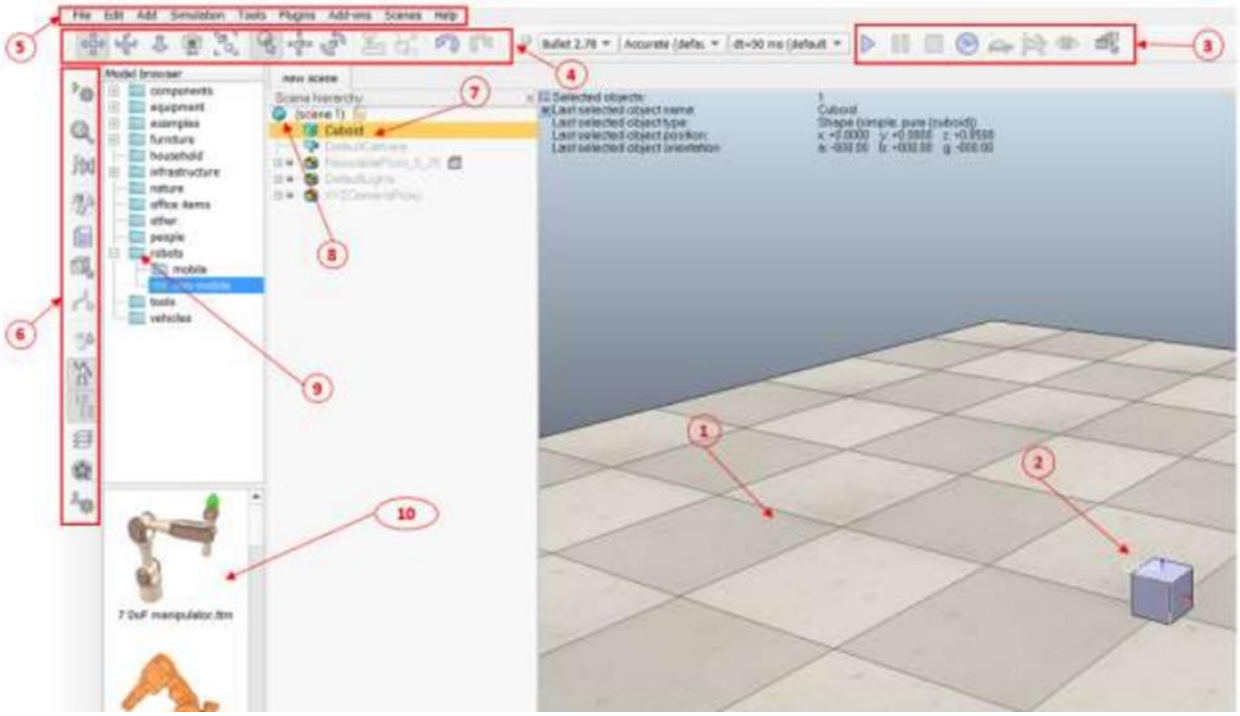
tar xvf CoppeliaSim_Edu_V4_0_0_Ubuntu18_04.tar.xz

سپس باید دایرکتوری ها را به دایرکتوری منبع CoppeliaSim تغییر دهید و اسکریپت پوسته coppeliaSim.sh را اجرا کنید.

پس از نصب این شبیه ساز شما در هنگام اجرای آن با صفحه زیر مواجه میشوید :



ما پهباد خود را در این شبیه ساز توسعه خواهیم داد و برای آشنایی شما با این محیط ما برخی از گزینه های مهم و پر کاربرد را که در تصویر 2 مشاهده میکنید ، شرح خواهیم داد.



ساختار coppeliasim:

COPPELIASIM حول یک معماری همه کاره طراحی شده است. هیچ عملکرد اصلی یا مرکزی در COPPELIASIM وجود ندارد. در عوض، COPPELIASIM دارای عملکردهای نسبتاً مستقل مختلفی است که می توانند در صورت لزوم فعال یا غیرفعال شوند. یک سناریوی شبیه سازی را تصور کنید که در آن یک ربات صنعتی باید جعبه ها را جمع آوری کند و آنها را به مکان دیگری منتقل کند. COPPELIASIM دینامیک را برای گرفتن و نگه داشتن جعبه ها محاسبه می کند و زمانی که اثرات دینامیکی ناچیز است، شبیه سازی سینماتیکی را برای سایر قسمت های چرخه انجام می دهد. این رویکرد محاسبه حرکت ربات صنعتی را به سرعت و دقیق ممکن می‌سازد، که اگر به طور کامل با استفاده از کتابخانه‌های دینامیک پیچیده شبیه‌سازی شده بود، اینطور نخواهد بود. این نوع شبیه‌سازی ترکیبی در این شرایط توجیه می‌شود، اگر ربات سفت و ثابت باشد و تحت تأثیر محیط خود قرار نگیرد. COPPELIASIM علاوه بر فعال سازی تطبیقی عملکردهای مختلف خود به شیوه ای انتخابی، می تواند از آنها به صورت همزیستی استفاده کند و یکی با دیگری همکاری کند. به عنوان مثال، در مورد یک ربات انسان نما، COPPELIASIM حرکات پا را با (الف) محاسبه اول سینماتیک معکوس برای هر پا انجام می دهد (یعنی از موقعیت و جهت دلخواه پا، تمام موقعیت های مفصل پا محاسبه می شود). و سپس (ب) موقعیت های مشترک محاسبه شده را به عنوان موقعیت های مشترک هدف توسط ماژول دینامیک اختصاص می دهد. این اجازه می دهد تا حرکت انسان نما را به روشی بسیار متنوع مشخص کنید، زیرا هر پا به سادگی باید یک مسیر 6 بعدی را دنبال کند: بقیه محاسبات به طور خودکار انجام می شود.

1:اشیاء صحنه

یک صحنه شبیه سازی COPPELIASIM شامل چندین شی صحنه یا اشیاء عنصری است که در یک سلسله مراتب درخت مانند مونتاژ شده اند. اشیاء صحنه زیر در COPPELIASIM پشتیبانی می شوند:

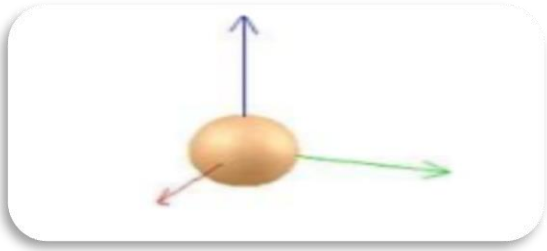
مفاصل: مفاصل جفت‌های پایین‌تری سینماتیکی هستند که دو یا چند شی صحنه را با یک تا سه درجه آزادی به هم پیوند می‌دهند (مانند منشوری، چرخشی، پیچ مانند و غیره).

مسیرها: مسیرها امکان تعاریف حرکتی پیچیده در فضا را فراهم می‌کنند (تسلسل ترجمه‌ها، چرخش‌ها و/یا مکث‌های آزادانه با هم ترکیب می‌شوند)، و برای هدایت مشعل ربات جوشکاری در طول یک مسیر از پیش تعریف‌شده، یا به عنوان مثال برای اجازه دادن به حرکات تسمه نقاله استفاده می‌شوند. فرزندان یک شی مسیر را می توان محدود کرد که در طول مسیر مسیر با سرعت معین حرکت کنند.

اشکال: شکل ها شبکه های مثلثی هستند که برای تجسم بدن سفت و سخت استفاده می شوند. سایر اشیاء صحنه یا ماژول‌های محاسباتی برای محاسبات خود به اشکال متکی هستند (تشخیص برخورد، محاسبه حداقل فاصله، و غیره).

دوربین‌ها و نورها: دوربین‌ها و نورها عمدتاً برای مصورسازی صحنه استفاده می‌شوند، اما می‌توانند روی سایر اشیاء صحنه نیز تأثیر بگذارند (مثلاً نورها مستقیماً بر حسگرهای رندر تأثیر می‌گذارند).

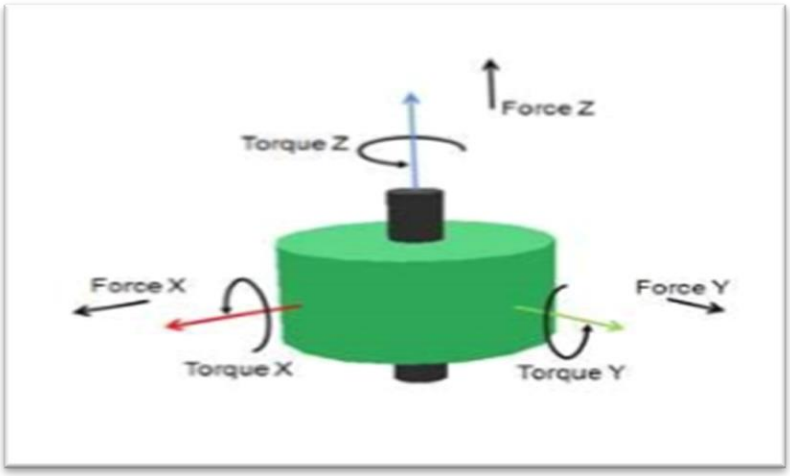
آدمک‌ها: آدمک‌ها «نقاط با جهت‌گیری» یا قاب‌های مرجع هستند که می‌توانند برای کارهای مختلف استفاده شوند و عمدتاً در ارتباط با سایر اشیاء صحنه استفاده می‌شوند و به این ترتیب می‌توان آنها را «کمک‌کننده» دید



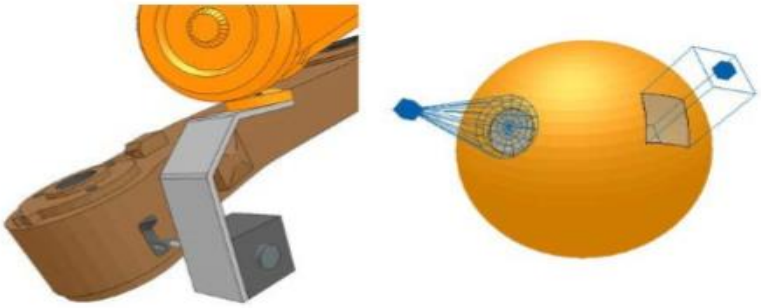
حسگرهای مجاورت: اشیاء حسگر مجاورتی یک محاسبه دقیق حداقل فاصله را در یک حجم تشخیص داده شده [4] انجام می دهند (شکل 1)برخلاف انجام تشخیص برخورد ساده بین برخی از پرتوهای حسگر انتخاب شده و محیط. از این رو امکان ایجاد اثرات بازتابی به دلیل زوایای حسگر/سطح را فراهم می کند.

حسگرهای رندر: حسگرهای رندر در COPPELIASIM حسگرهایی شبیه دوربین هستند که امکان استخراج اطلاعات تصویر پیچیده را از صحنه شبیه سازی (رنگ ها، اندازه اشیا، نقشه های عمق و غیره) فراهم می کنند (شکل 1). فیلتر داخلی و پردازش تصویر ترکیبی از عناصر فیلتر را به عنوان بلوک (با عناصر فیلتر اضافی از طریق پلاگین ها) امکان پذیر می کند. حسگرهای رندر از شتاب سخت افزاری برای گرفتن تصویر خام (OpenGL) استفاده می کنند.

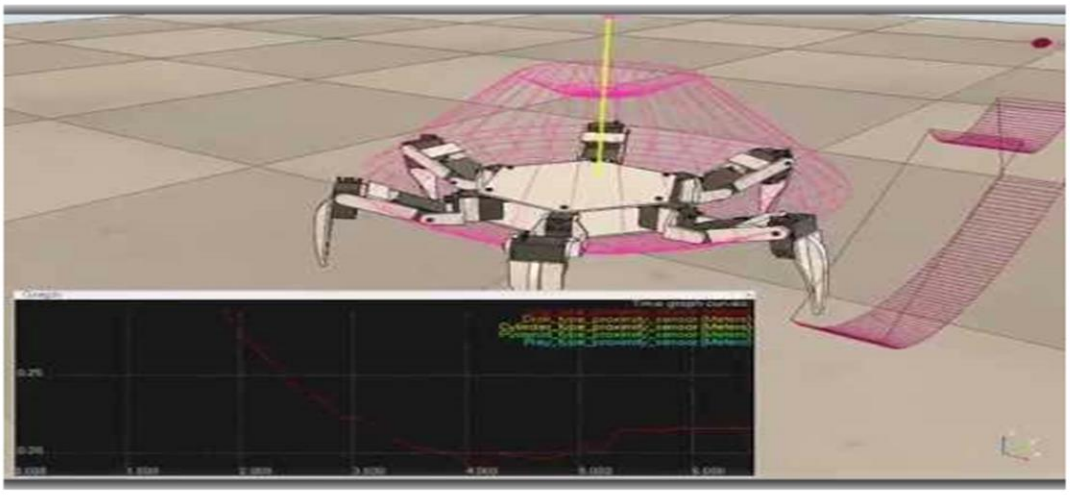
حسگرهای نیرو: حسگرهای نیرو پیوندهای صلب بین اشکال هستند که می توانند نیروها و گشتاورهای اعمال شده را ثبت کنند و به طور مشروط می توانند اشباع شوند.



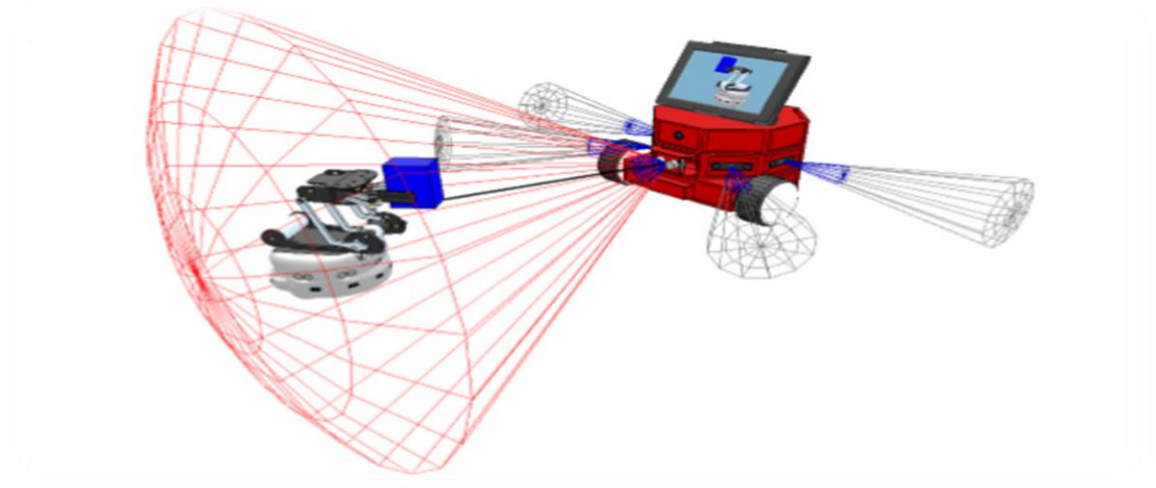
آسیاب ها: آسیاب ها حجم های محدب قابل تنظیمی هستند که می توانند برای شبیه سازی عملیات برش سطحی روی اشکال (مانند فرز، برش لیزری و غیره) استفاده شوند.



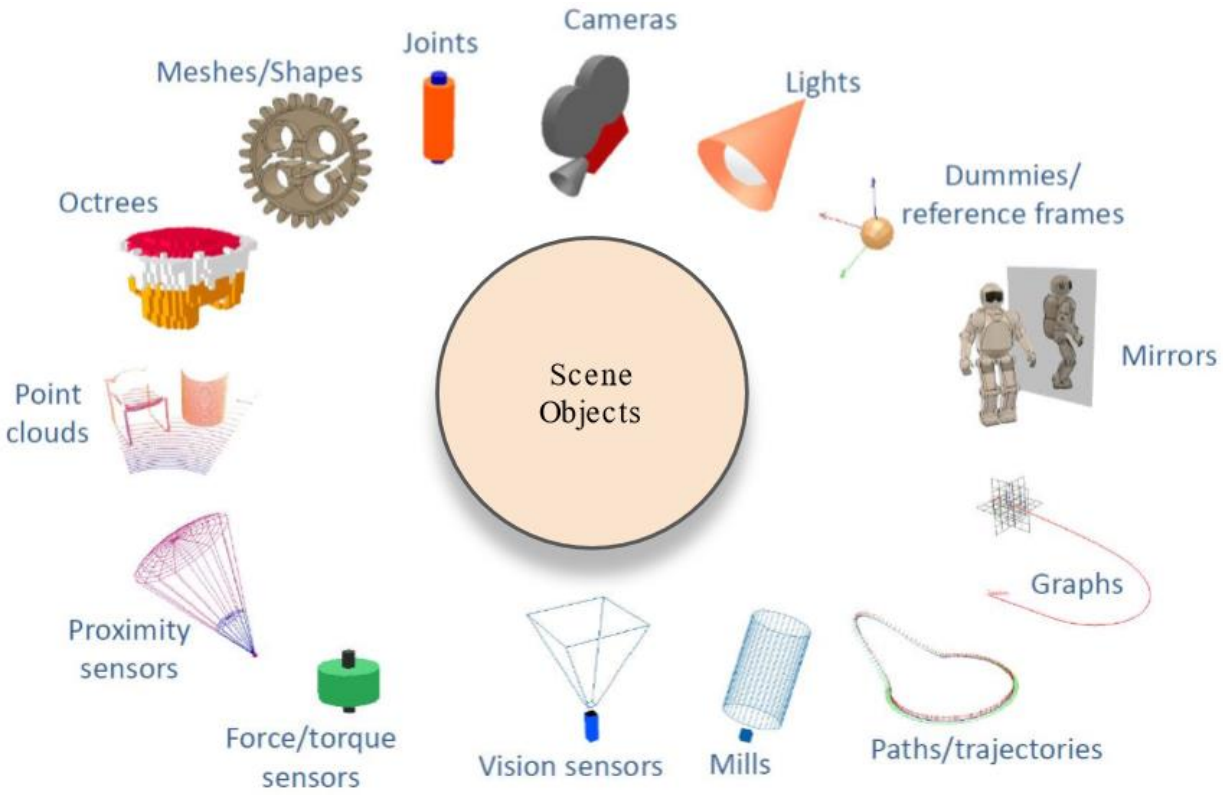
نمودارها: نمودارها اشیاء صحنه ای هستند که می توانند طیف وسیعی از جریان های داده یک بعدی را ضبط کنند. جریان‌های داده را می‌توان مستقیماً نمایش داد (نمودار زمانی یک نوع داده معین)، یا برای نمایش نمودارهای X/Y یا منحنی‌های سه بعدی با یکدیگر ترکیب کرد.



بستر آزمایشی ربات مجازی:



عکس1. ربات متحرک (سمت راست) مجهز به 5 حسگر مجاورت و یک سنسور رندر. سنسور رندر برای تشخیص در این مورد استفاده نمی شود، بلکه برای تولید بافت برای پنل LCD استفاده می شود. (مدل ربات سمت چپ توسط .K-Team Corp، مدل ربات سمت راست توسط .Cubictek Corpو NT Research Corp است).



نمای کلی از scene objects

2. ماژول های محاسبه

اشیاء صحنه به ندرت به تنهایی مورد استفاده قرار می گیرند، آنها بیشتر بر روی (یا همراه با) اشیاء صحنه دیگر کار می کنند (مثلاً یک حسگر مجاورت اشکال یا ساختگی هایی را که با حجم تشخیص آن تلاقی می کنند، تشخیص می دهد). علاوه بر این، COPPELIASIM دارای چندین ماژول محاسبه است که می توانند مستقیماً روی یک یا چند شی صحنه عمل کنند. ماژول های اصلی محاسبه COPPELIASIM در زیر آمده است:

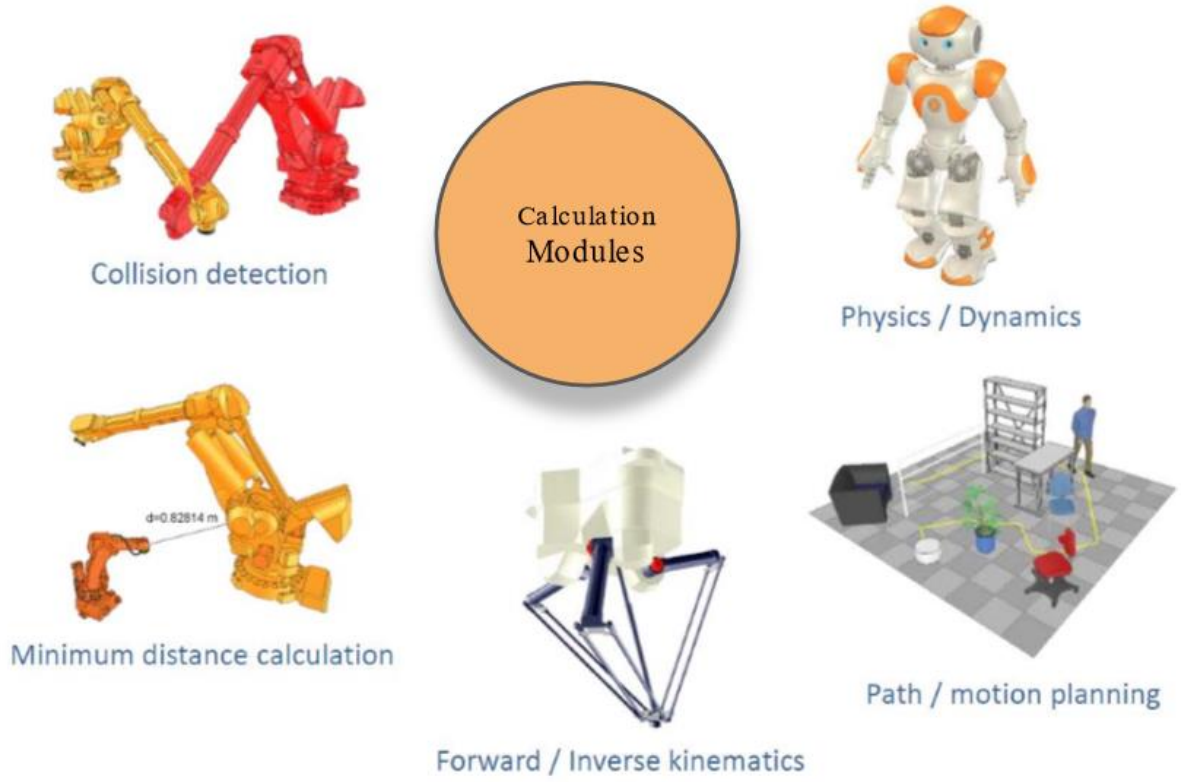
- **ماژول سینماتیک رو به جلو و معکوس**: محاسبات سینماتیکی را برای هر نوع مکانیزم (شاخه ای، بسته، اضافی، حاوی حلقه های تو در تو و غیره) مجاز می کند. ماژول بر اساس محاسبه کمترین مربعات میرایی شبه معکوس [7] است. از حل شرطی، وضوح میرا و وزن دار، و محدودیت های مبتنی بر اجتناب از موانع پشتیبانی می کند.

- **ماژول دینامیک یا فیزیک**: امکان مدیریت محاسبات و تعامل دینامیک بدن صلب (پاسخ برخورد، گرفتن، و غیره) از طریق کتابخانه Bullet Physics [1]را فراهم می کند.

– **ماژول برنامه‌ریزی مسیر**: به وظایف برنامه‌ریزی مسیر هولونومیک و کارهای برنامه‌ریزی مسیر غیرهولونومیک (برای وسایل نقلیه شبیه خودرو) از طریق رویکردی مشتق‌شده از الگوریتم درخت تصادفی با کاوش سریع (RRT) [6]اجازه می‌دهد.

-**ماژول تشخیص برخورد**: امکان بررسی تداخل سریع بین هر شکل یا مجموعه ای از اشکال را فراهم می کند. به صورت اختیاری، کانتور برخورد نیز قابل محاسبه است. این ماژول از ساختارهای داده مبتنی بر درخت باینری از جعبه های مرزی جهت دار [5] برای شتاب ها استفاده می کند. بهینه‌سازی اضافی با تکنیک ذخیره‌سازی انسجام زمانی حاصل می‌شود.

- **ماژول محاسبه حداقل فاصله**: امکان محاسبه سریع حداقل فاصله بین هر شکل (محدب، مقعر، باز، بسته و غیره) یا مجموعه ای از اشکال را فراهم می کند. ماژول از ساختارهای داده ای مشابه ماژول تشخیص برخورد استفاده می کند. بهینه‌سازی اضافی نیز با تکنیک ذخیره‌سازی انسجام زمانی حاصل می‌شود. به جز ماژول‌های دینامیک یا فیزیک که مستقیماً روی همه اشیاء صحنه فعال شده به صورت پویا عمل می‌کنند، سایر ماژول‌های محاسباتی به تعریف یک تکلیف محاسبه یا شی محاسباتی نیاز دارند، که مشخص می‌کند ماژول روی کدام اشیاء صحنه باید کار کند و چگونه کار کند. برای مثال، اگر کاربر بخواهد حداقل فاصله بین شکل A و شکل B را به طور خودکار محاسبه کند و شاید هم ثبت شود، باید یک شی حداقل فاصله تعریف شود که به عنوان پارامترهای شکل A و شکل B باشد. شکل 2 COPPELIASIM را نشان می دهد. حلقه شبیه سازی معمولی، از جمله اشیاء صحنه اصلی و ماژول های محاسبه.



تصویر مرتبط با calculation modules

3:مقیاس پذیری

تخریب یک یا چند شیء صحنه می تواند شامل تخریب خودکار یک شی محاسباتی مرتبط باشد. به روشی مشابه، کپی کردن یک یا چند شی صحنه می تواند شامل تکرار خودکار اشیاء محاسباتی مرتبط باشد. این همچنین شامل تکثیر خودکار اسکریپت های کنترل مرتبط می شود (به بخش بعدی مراجعه کنید). نتیجه این است که اشیاء منظره تکراری به طور خودکار کاملاً کاربردی خواهند بود، و اجازه می‌دهند یک رفتار انعطاف‌پذیر مانند. plug-and-play داشته باشند.

معرفی رابط کاربری:

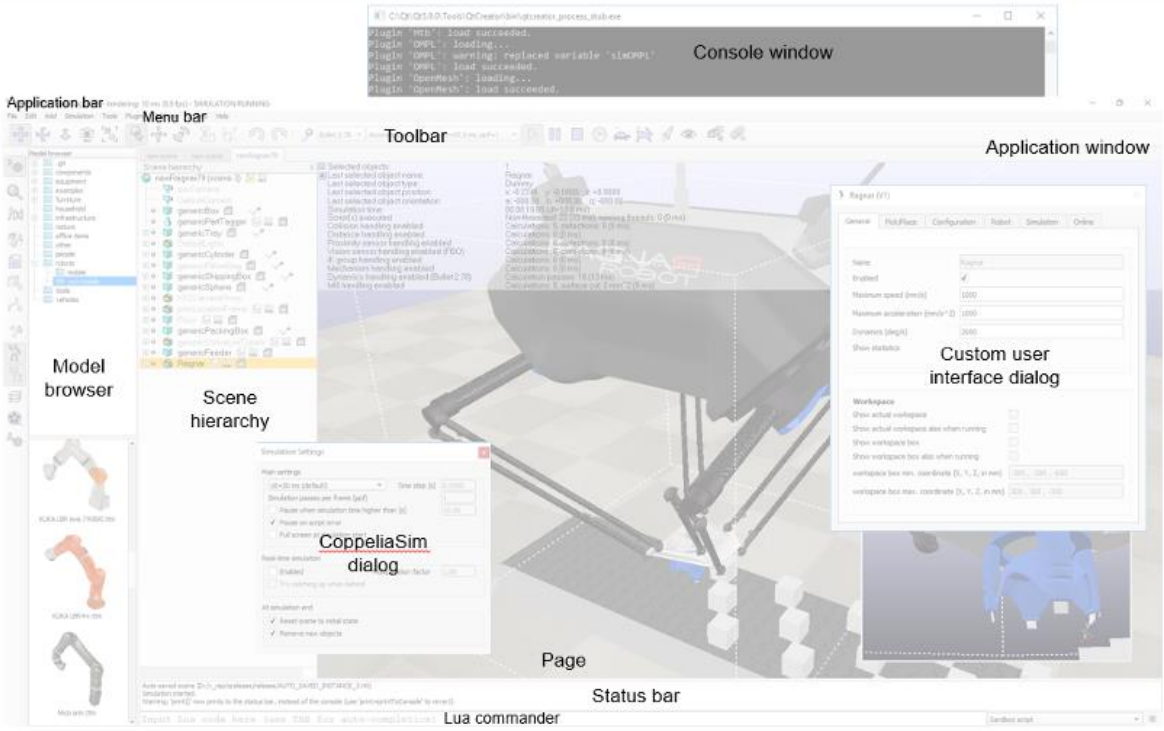
برنامه CoppeliaSim از چندین عنصر تشکیل شده است. عناصر اصلی آن عبارتند از:

یک پنجره کنسول: در ویندوز، زمانی که برنامه CoppeliaSim شروع به کار می کند، یک پنجره کنسول ایجاد می شود اما مستقیماً دوباره پنهان می شود. این رفتار پیش فرض پنهان کردن پنجره کنسول را می توان در گفتگوی تنظیمات کاربر تغییر داد. تحت لینوکس، CoppeliaSim باید از کنسول شروع شود، که در کل جلسه CoppeliaSim قابل مشاهده است. در MacOSX، بهترین کار این است که CoppeliaSim را از یک ترمینال راه اندازی کنید تا پیام ها قابل مشاهده باشند. کنسول یا پنجره ترمینال نشان می دهد که چه افزونه هایی بارگذاری شده اند و آیا روش اولیه سازی آنها موفقیت آمیز بوده است. پنجره کنسول تعاملی نیست و فقط برای خروجی اطلاعات استفاده می شود. کاربر می‌تواند مستقیماً اطلاعات را با دستور (printاز داخل یک اسکریپت)، یا با دستورات C printf یا std::cout از داخل یک افزونه به پنجره کنسول خروجی دهد. علاوه بر آن، کاربر می تواند به صورت برنامه نویسی، پنجره های کنسول کمکی را برای نمایش اطلاعات خاص یک شبیه سازی ایجاد کند.

یک پنجره برنامه: پنجره برنامه، پنجره اصلی برنامه است. برای نمایش، ویرایش، شبیه سازی و تعامل با یک صحنه استفاده می شود. دکمه های چپ و راست ماوس، چرخ ماوس و همچنین کیبورد وقتی در پنجره برنامه فعال می شوند، عملکردهای خاصی دارند. در پنجره برنامه، عملکرد دستگاه های ورودی (ماوس و صفحه کلید) ممکن است بسته به زمینه یا مکان فعال سازی متفاوت باشد.

چندین دیالوگ: در کنار پنجره برنامه، کاربر همچنین می تواند با تنظیم تنظیمات یا پارامترهای گفتگو، یک صحنه را ویرایش کرده و با آن تعامل داشته باشد. هر دیالوگ مجموعه ای از توابع یا توابع مرتبط را گروه بندی می کند که برای یک شی هدف مشابه اعمال می شود. محتوای دیالوگ ممکن است به زمینه حساس باشد (مثلاً وابسته به وضعیت انتخاب شی).

شکل زیر یک نمای معمولی از برنامه coppeliasim را نشان می دهد:



[User interface elements]

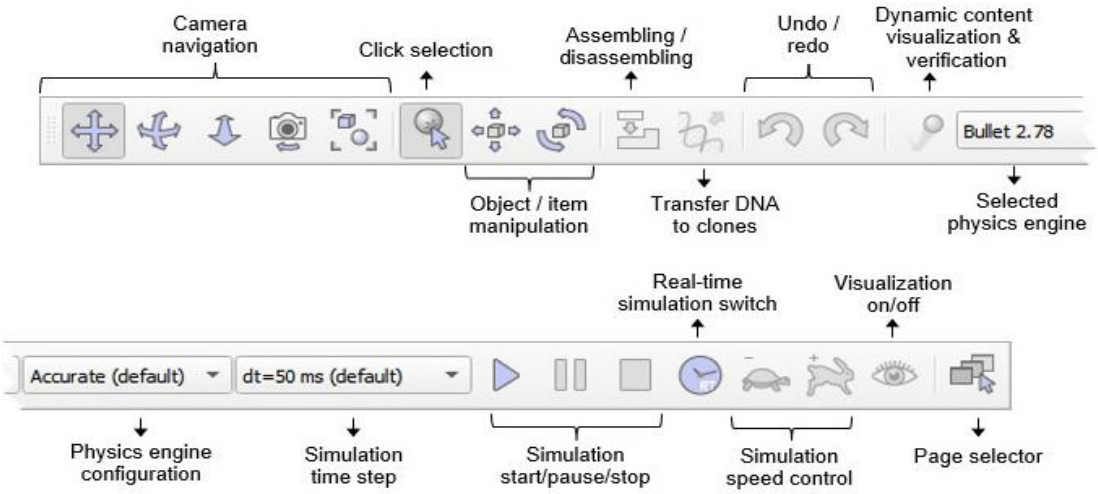
هنگامی که برنامه CoppeliaSim را راه اندازی می کنید، CoppeliaSim یک صحنه پیش فرض را مقداردهی اولیه می کند. کاربر آزاد است چندین صحنه را به صورت موازی باز کند. هر صحنه پنجره برنامه و دیالوگ ها را با صحنه های دیگر به اشتراک می گذارد، اما فقط محتوای صحنه فعال در پنجره برنامه یا دیالوگ ها قابل مشاهده خواهد بود (فقط یک صحنه در یک زمان معین قابل مشاهده است).

نوار برنامه: نوار برنامه نوع مجوز کپی CoppeliaSim شما، نام فایل صحنه ای که در حال حاضر نمایش داده می شود، زمان استفاده شده برای یک پاس رندر (یک پاس نمایش)، و وضعیت فعلی شبیه ساز (وضعیت یا نوع شبیه سازی) را نشان می دهد. از حالت ویرایش فعال). نوار برنامه، و همچنین هر سطحی در پنجره برنامه، همچنین می‌تواند برای کشیدن و رها کردن فایل‌های مربوط به CoppeliaSim به داخل صحنه استفاده شود. فایل‌های پشتیبانی‌شده شامل فایل‌های *.) ttt فایل‌های صحنه (CoppeliaSimو فایل‌های «*.) «ttmفایل‌های مدل CoppeliaSim)هستند.

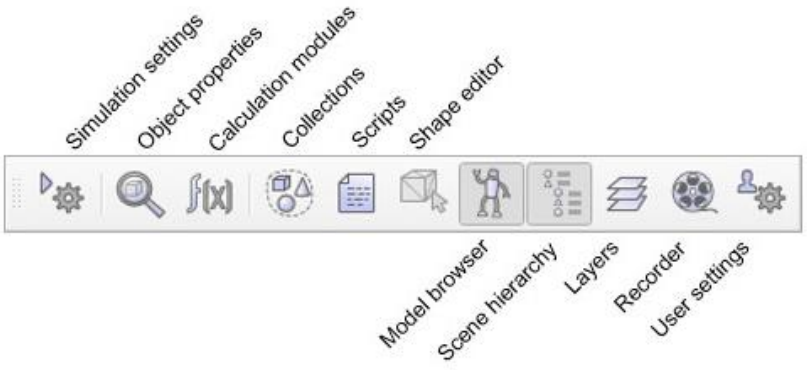
نوار منو: نوار منو امکان دسترسی به تقریباً تمام عملکردهای شبیه ساز را فراهم می کند. بیشتر اوقات، موارد موجود در نوار منو یک دیالوگ را فعال می کنند. محتوای نوار منو به زمینه حساس است (یعنی به وضعیت فعلی

شبیه ساز بستگی دارد). بیشتر توابع موجود در نوار منو می‌توانند از طریق یک منوی بازشو، دوبار کلیک کردن روی نماد در نمای سلسله مراتبی صحنه یا از طریق کلیک بر روی دکمه نوار ابزار قابل دسترسی باشند.

نوار ابزار: نوار ابزار عملکردهایی را ارائه می دهد که اغلب به آنها دسترسی پیدا می شود (مانند تغییر حالت ناوبری، انتخاب صفحه دیگر و غیره). برخی از توابع در نوار ابزار 1 و همه عملکردهای نوار ابزار 2 نیز از طریق نوار منو یا منوی بازشو قابل دسترسی هستند. برای جزئیات بیشتر به پایین تر مراجعه کنید. هر دو نوار ابزار را می توان متصل و باز کرد، اما اتصال فقط با موقعیت های اولیه مربوطه کار می کند. شکل زیر عملکرد هر دکمه نوار ابزار را توضیح می دهد:

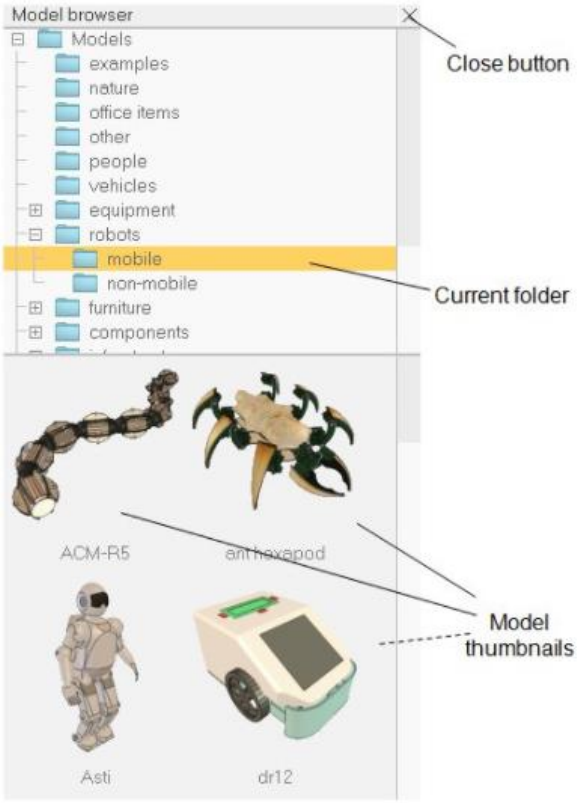


[Toolbar 1]



[Toolbar 2]

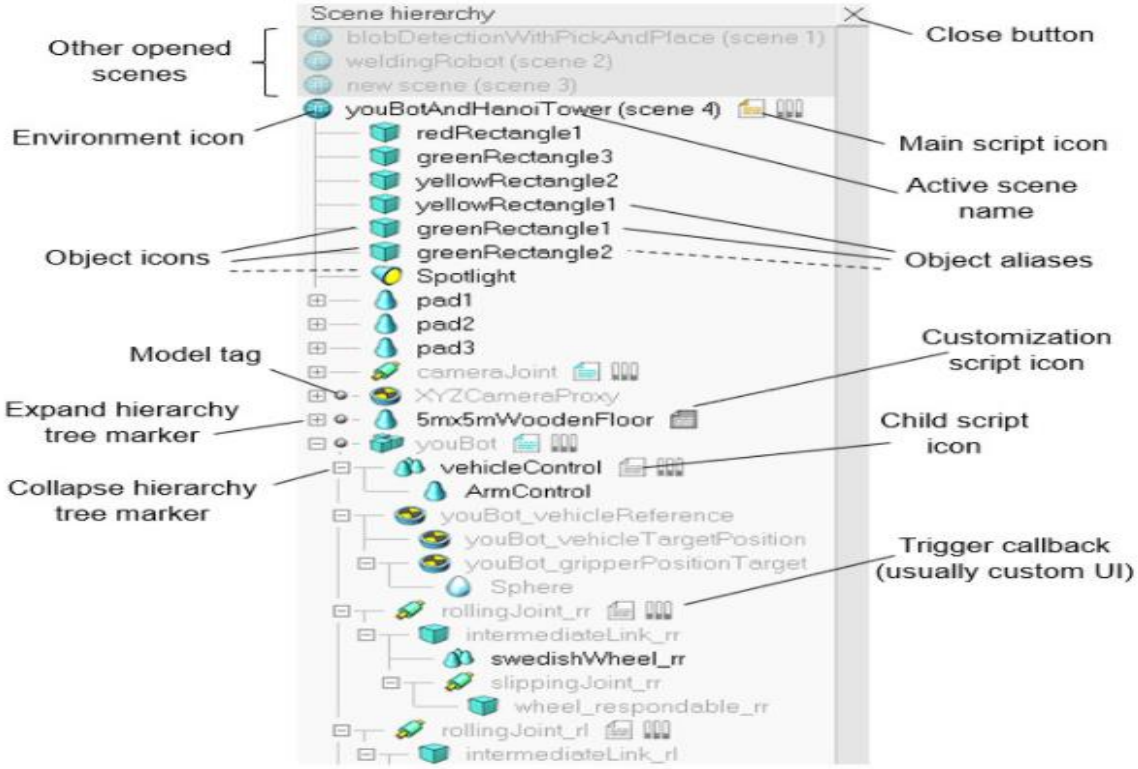
مرورگر مدل: مرورگر مدل به طور پیش فرض قابل مشاهده است، اما می توان با دکمه نوار ابزار مربوطه آن را تغییر داد. در قسمت بالای خود یک ساختار پوشه مدل CoppeliaSim و در قسمت پایین آن، تصاویر کوچک مدل های موجود در پوشه انتخاب شده را نمایش می دهد. برای بارگیری خودکار مدل مربوطه، می‌توان تصاویر کوچک را به صحنه کشیده و رها کرد. اگر قسمت ریزش پشتیبانی نشود یا مناسب نباشد، تصاویر کوچک گرفته شده تیره به نظر می‌رسند.



[Model browser]

سلسله مراتب صحنه: سلسله مراتب صحنه به طور پیش فرض قابل مشاهده است، اما می توان با دکمه نوار ابزار مربوطه آن را تغییر داد. محتوای یک صحنه را نمایش می دهد (یعنی همه اشیاء صحنه که یک صحنه را می سازند). از آنجایی که اشیاء صحنه در یک ساختار سلسله مراتبی ساخته می شوند، سلسله مراتب صحنه درختی از این سلسله مراتب را نشان می دهد و عناصر منفرد را می توان گسترش داد یا جمع کرد. با دوبار کلیک کردن روی یک نماد، گفتگوی ویژگی مربوط به نماد کلیک شده باز یا بسته می شود. دوبار کلیک کردن روی یک شیء مستعار امکان ویرایش آن را فراهم می کند. چرخ ماوس و همچنین کشیدن نوارهای پیمایش نمای سلسله مراتبی صحنه امکان جابجایی محتوا را به بالا/پایین یا چپ/راست می دهد. کنترل و انتخاب شیفت همیشه پشتیبانی می

شود. اشیاء در سلسله مراتب صحنه را می توان به منظور ایجاد یک رابطه والد-فرزند کشیده و روی شی دیگری رها کرد. اگر شبیه ساز در حالت ویرایش باشد، سلسله مراتب صحنه محتوای متفاوتی را نمایش می دهد. برای اطلاعات بیشتر به حالت های ویرایش شکل مراجعه کنید.



[Scene hierarchy]

صفحه: هر صحنه ممکن است تا 8 صفحه داشته باشد، هر یک از آنها ممکن است تعداد نامحدودی بازدید داشته باشد. یک صفحه را می توان به عنوان محفظه ای برای بازدیدها مشاهده کرد. برای جزئیات بیشتر به بخش صفحات و بازدیدها مراجعه کنید.

بازدیدها: می‌تواند تعداد نامحدودی بازدید در یک صفحه وجود داشته باشد. نما برای نمایش صحنه (که شامل یک محیط و اشیاء است) استفاده می شود که از طریق یک شی قابل مشاهده (مانند دوربین ها، نمودارها یا حسگرهای بینایی) دیده می شود.

متن اطلاعاتی: متن اطلاعات اطلاعات مربوط به انتخاب شی/مورد فعلی و وضعیت ها یا پارامترهای شبیه سازی در حال اجرا را نمایش می دهد. نمایش متن را می توان با یکی از دو دکمه کوچک در سمت چپ بالای صفحه

تغییر داد. از دکمه دیگر می توان برای تغییر پس زمینه سفید استفاده کرد و کنتراست بهتری را بسته به رنگ پس زمینه یک صحنه ایجاد کرد.

نوار وضعیت: نوار وضعیت اطلاعات مربوط به عملیات، دستورات انجام شده را نمایش می دهد و همچنین پیام های خطا را از مفسرهای اسکریپت نمایش می دهد. از داخل یک اسکریپت، کاربر همچنین می‌تواند رشته‌هایی را به نوار وضعیت یا کنسول با تابع sim.addLog خروجی دهد. نوار وضعیت به طور پیش‌فرض تنها دو خط را نمایش می‌دهد، اما می‌توان اندازه آن را با استفاده از دسته جداسازی افقی تغییر داد.

فرمان :Luaیک حلقه خواندن-ارزیابی-چاپ، که ورودی متنی را به نوار وضعیت CoppeliaSim اضافه می‌کند و امکان وارد کردن و اجرای کد Lua را در همان لحظه، مانند ترمینال، می‌دهد. کد را می توان در اسکریپت sandbox یا هر اسکریپت فعال دیگری در CoppeliaSim اجرا کرد.

رابط های کاربر سفارشی: رابط های کاربری سفارشی سطوح رابط کاربری تعریف شده توسط کاربر هستند که می توانند برای نمایش اطلاعات (متن، تصاویر و غیره) یا یک گفتگوی سفارشی استفاده شوند و به کاربر اجازه تعامل را به روشی سفارشی می دهند.

منوی بازشو: منوهای بازشو منوهایی هستند که پس از کلیک راست ماوس ظاهر می شوند. برای فعال کردن یک منوی بازشو، مطمئن شوید که ماوس در حین عملیات کلیک حرکت نمی کند، در غیر این صورت ممکن است حالت چرخش دوربین فعال شود (برای جزئیات بیشتر به بخش دوربین مراجعه کنید). هر سطح در پنجره برنامه (مثلاً نمای سلسله مراتبی صحنه، صفحه، نمای و غیره) ممکن است منوی بازشو متفاوتی را فعال کند (حساس به زمینه). محتوای منوهای بازشو نیز ممکن است بسته به وضعیت شبیه سازی فعلی یا حالت ویرایش تغییر کند. اکثر عملکردهای منوی بازشو نیز از طریق نوار منو قابل دسترسی هستند، به جز آیتم منوی نمایش که تنها زمانی ظاهر می شود که منوی بازشو در یک نما یا صفحه فعال شود.

آموزش ساخت یک مدل شبیه سازی به عنوان مثال:

این آموزش گام به گام شما را در ساخت یک مدل شبیه سازی ، یک ربات یا هر مورد دیگری راهنمایی می کند. این یک موضوع بسیار مهم است، شاید مهمترین جنبه، به منظور داشتن یک مدل شبیه سازی زیبا، نمایش سریع، شبیه سازی سریع و پایدار.

برای نشان دادن فرآیند ساخت مدل، ما دستکاری کننده زیر را می سازیم:

[Model of robotic manipulator]

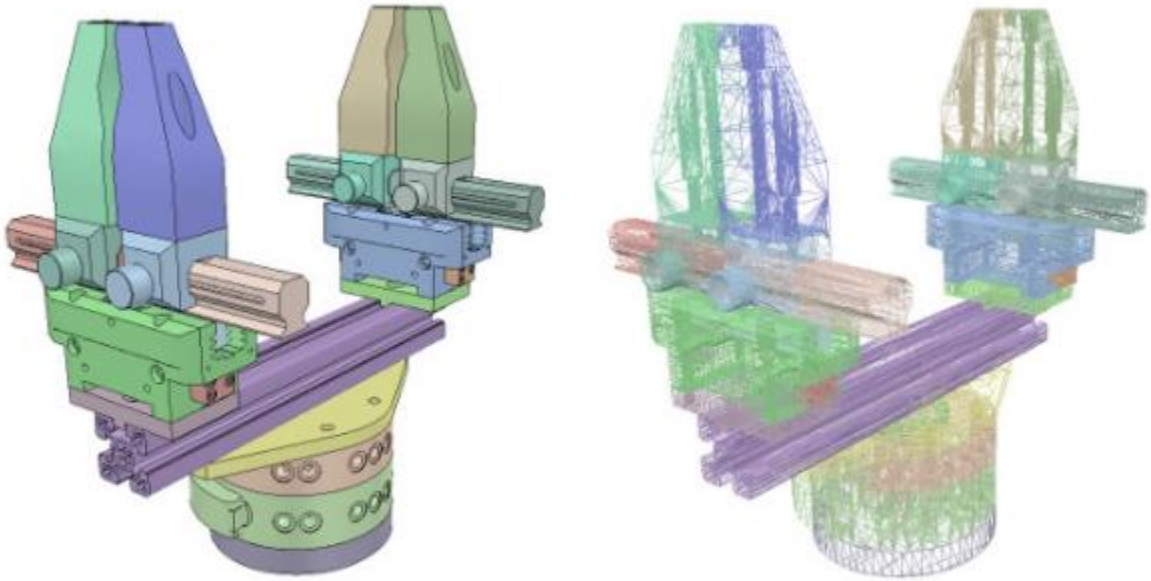
ساختن اشکال قابل مشاهده:

هنگام ساخت یک مدل جدید، ابتدا فقط جنبه بصری آن را بررسی می کنیم: جنبه دینامیکی (مدل ساده تر/بهینه تر آن)، اتصالات، حسگرها و غیره در مرحله بعدی بررسی خواهد شد.

اکنون می‌توانیم مستقیماً با [نوار منو --> افزودن --> شکل اولیه --> ...] شکل‌های اولیه را در CoppeliaSim ایجاد کنیم. هنگام انجام این کار، ما این امکان را داریم که اشکال خالص یا اشکال منظم ایجاد کنیم. شکل خالص برای تعامل پویا بهینه می‌شود، و همچنین مستقیماً به صورت پویا فعال می‌شود (به عنوان مثال سقوط، برخورد، اما این می‌تواند در مرحله بعد غیرفعال شود). اشکال ابتدایی مش های ساده ای خواهند بود که ممکن است حاوی جزئیات کافی یا دقت هندسی برای کاربرد ما نباشند. گزینه دیگر ما در این مورد، وارد کردن مش از یک برنامه خارجی است.

هنگام وارد کردن داده های CAD از یک برنامه خارجی، مهمترین چیز این است که مطمئن شوید که مدل CAD خیلی سنگین نیست، یعنی شامل مثلث های زیادی نباشد. این نیاز مهم است زیرا یک مدل سنگین در نمایش آهسته خواهد بود و همچنین ماژول های محاسباتی مختلفی را که ممکن است در مراحل بعدی مورد استفاده

قرار گیرند (مانند محاسبه حداقل فاصله یا دینامیک) کند می کند. مثال زیر معمولاً ممنوع است (حتی اگر همانطور که در ادامه خواهیم دید، ابزارهایی برای ساده کردن داده ها در CoppeliaSim وجود داشته باشد):

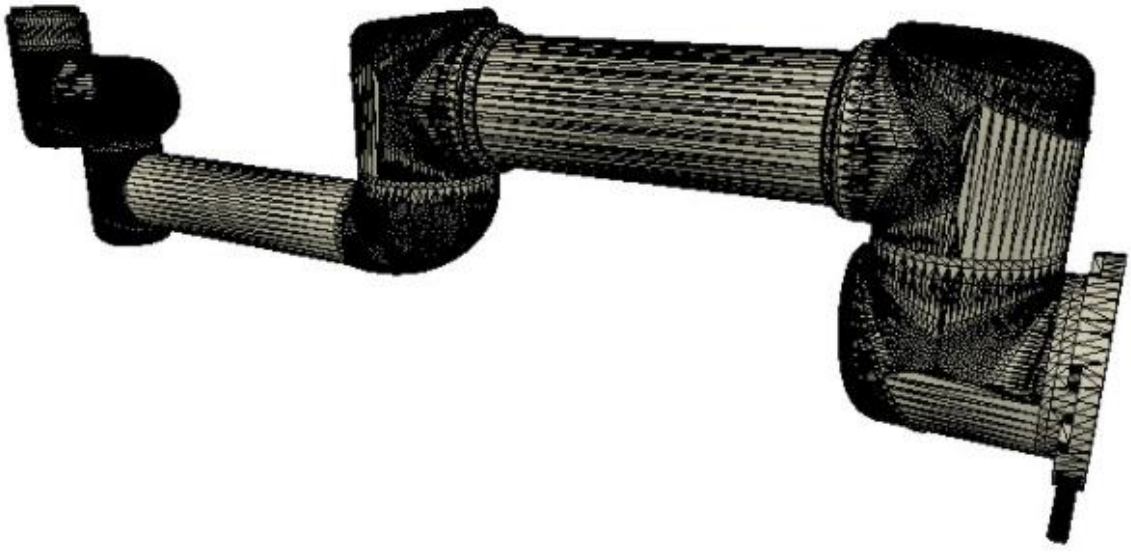


[Complex CAD data (in solid and wireframe)]

داده های بالای CAD بسیار سنگین هستند: شامل مثلث های زیادی است (بیش از 47000)، که اگر فقط از یک نمونه از آن در یک صحنه خالی استفاده کنیم، مشکلی نیست. اما اغلب اوقات شما می خواهید چندین نمونه از یک ربات مشابه را شبیه سازی کنید، انواع مختلفی از گیره ها را وصل کنید و شاید آن روبات ها با ربات ها، دستگاه ها یا محیط دیگر تعامل داشته باشند. در آن صورت، یک صحنه شبیه سازی می تواند به سرعت بسیار کند شود. به طور کلی، ما توصیه می کنیم رباتی را با مجموع 20000 مثلث مدل کنید، اما اغلب اوقات 5000-10000 مثلث نیز خوب عمل می کند. به یاد داشته باشید: تقریباً در هر جنبه ای کمتر بهتر است.

چه چیزی باعث می شود مدل فوق اینقدر سنگین باشد؟ اولاً، مدل‌هایی که حاوی سوراخ‌ها و جزئیات کوچک هستند، برای نمایش صحیح به چهره‌های مثلثی بسیار بیشتری نیاز دارند. بنابراین، در صورت امکان، سعی کنید تمام سوراخ ها، پیچ ها، داخل اجسام و غیره را از داده های مدل اصلی خود حذف کنید. اگر داده‌های مدل اصلی را دارید که به‌عنوان سطوح/اشیاء پارامتریک نشان داده می‌شوند، در اکثر مواقع انتخاب آیتم‌ها و حذف آنها (مثلاً در Solidworks) ساده است. دومین مرحله مهم این است که داده های اصلی را با دقت محدود صادر کنید: اکثر برنامه های کاربردی CAD به شما امکان می دهند سطح جزئیات مش های صادراتی را مشخص کنید. همچنین ممکن است مهم باشد که اشیاء را در چند مرحله صادر کنید، زمانی که طراحی از اشیاء بزرگ و کوچک تشکیل شده است. این برای جلوگیری از تعریف دقیق اجسام بزرگ (مثلث های زیاد) و اجسام کوچک بسیار دقیق (مثلث های بسیار کم) است: به سادگی اجسام بزرگ را ابتدا (با تنظیم تنظیمات دقیق مورد نظر) و سپس اشیاء کوچک (با تنظیم تنظیمات دقیق) صادر کنید.).

حال فرض کنید که همه ساده سازی های ممکن را همانطور که در بخش قبل توضیح داده شد اعمال کرده ایم. ممکن است پس از وارد کردن، همچنان با یک مش خیلی سنگین مواجه شویم:



[Imported CAD data]

می توانید متوجه شوید که کل ربات به صورت یک مش وارد شده است. بعداً نحوه تقسیم مناسب آن را خواهیم دید. همچنین به جهت گیری اشتباه مش وارد شده توجه کنید: بهترین کار این است که جهت گیری را همانطور که هست حفظ کنید تا زمانی که کل مدل ساخته شود، زیرا اگر در مرحله بعد بخواهیم موارد دیگری را که مربوط به همان ربات هستند وارد کنیم، آنها این کار را انجام خواهند داد. به طور خودکار موقعیت / جهت گیری صحیح را نسبت به مش اصلی داشته باشد.

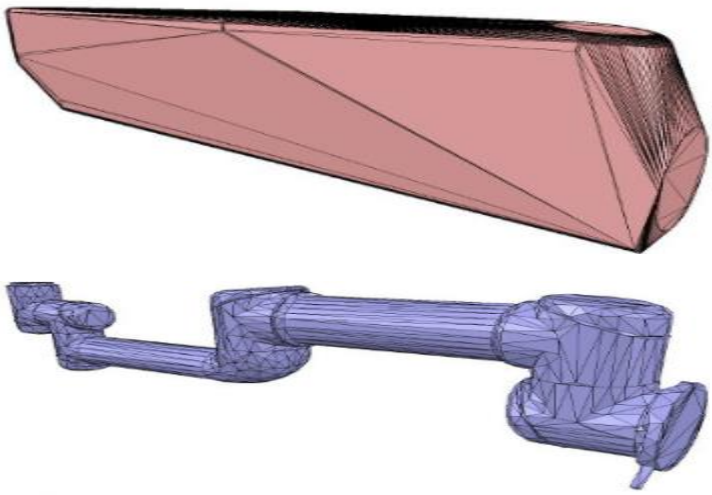
در این مرحله، ما چندین عملکرد در اختیار داریم تا مش را ساده کنیم:

تقسیم خودکار مش: امکان ایجاد یک شکل جدید برای تمام عناصری که از طریق یک لبه مشترک به هم مرتبط نیستند را می دهد. این همیشه برای مش انتخاب شده کار نمی کند، اما همیشه ارزش امتحان را دارد، زیرا کار بر روی عناصر مش کنترل بیشتری نسبت به زمانی که مجبور باشیم روی همه عناصر به طور همزمان کار کنیم، به ما می دهد. این تابع با [نوار منو --> ویرایش --> گروه بندی/ادغام --> تقسیم اشکال انتخاب شده] قابل دسترسی است. گاهی اوقات، یک مش بیش از حد انتظار تقسیم می شود. در این صورت، به سادگی عناصری را که به طور منطقی به یکدیگر تعلق دارند (یعنی ویژگی های بصری یکسانی دارند و بخشی از پیوند یکسان هستند) را در یک شکل واحد ادغام کنید ([نوار منو --> ویرایش -> گروه بندی/ادغام --> ادغام اشکال انتخاب شده]).

استخراج بدنه محدب: به شما امکان می دهد تا مش را با تبدیل آن به یک بدنه محدب ساده کنید. این تابع با [نوار منو --> ویرایش --> انتخاب شکل به اشکال محدب] قابل دسترسی است.

جدا کردن مش: امکان کاهش تعداد مثلث های موجود در مش را فراهم می کند. عملکرد را می توان با [نوار منو --> ویرایش --> حذف شکل انتخاب شده...] مشاهده کرد.

هیچ ترتیب از پیش تعریف شده ای وجود ندارد که توابع فوق را بتوان/باید اعمال کرد (به جز اولین مورد در لیست، که همیشه باید ابتدا امتحان شود)، این امر به شدت به هندسه مش که ما سعی در ساده کردن آن داریم بستگی دارد. تصویر زیر عملکردهای اعمال شده در مش وارد شده را نشان می دهد (فرض کنیم اولین مورد در لیست برای ما کار نمی کند):



[Convex hull, and decimated mesh]

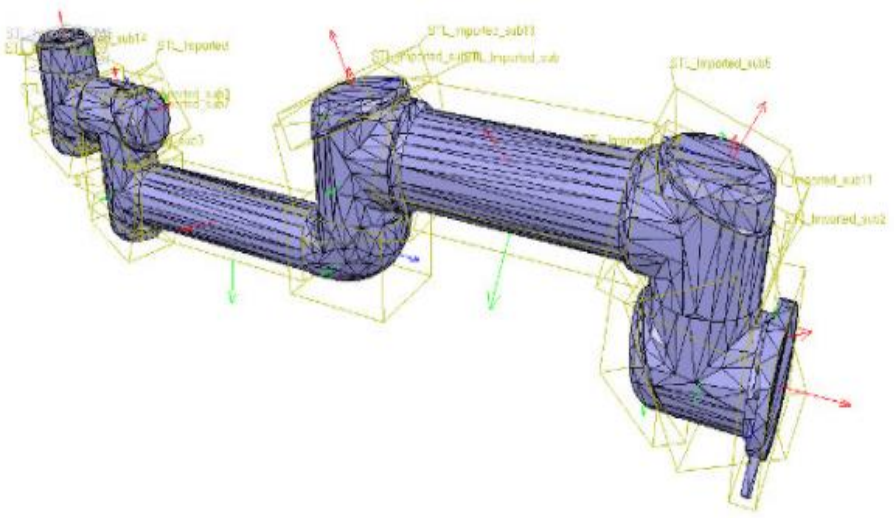
توجه کنید که چگونه بدنه محدب در این مرحله به ما کمک نمی کند. ما تصمیم می‌گیریم ابتدا از تابع decimation مش استفاده کنیم، و تابع را دو بار اجرا کنیم تا تعداد مثلث‌ها را بر 50 تقسیم کنیم. بیش از 136000 مثلث!). تعداد مثلث ها/راس های یک شکل را می توان در گفتگوی هندسه شکل مشاهده کرد. مثلث های 2660 برای یک مدل ربات کامل مثلث های بسیار کمی هستند و ظاهر بصری ممکن است کمی از آن آسیب ببیند.

در این مرحله می‌توانیم ربات را به پیوندهای جداگانه تقسیم کنیم (به یاد داشته باشید، در حال حاضر فقط یک شکل برای کل ربات داریم). شما می توانید این کار را به دو روش مختلف انجام دهید:

تقسیم خودکار مش: این عملکرد که قبلاً در بخش قبلی توضیح داده شد، شکل را بررسی می کند و شکل جدیدی را برای همه عناصری که از طریق یک لبه مشترک به هم مرتبط نیستند ایجاد می کند. این همیشه کار نمی کند، اما همیشه ارزش امتحان کردن را دارد. این تابع با [نوار منو --> ویرایش --> گروه بندی/ادغام --> تقسیم اشکال انتخاب شده] قابل دسترسی است.

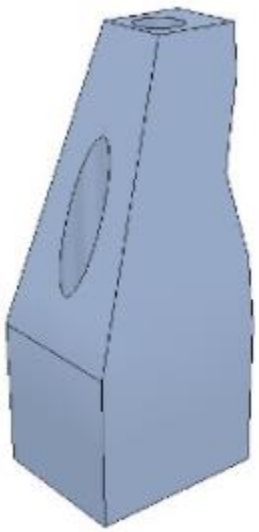
تقسیم مش دستی: از طریق حالت ویرایش مثلث، می‌توانید به‌طور دستی مثلث‌ها را انتخاب کنید که منطقاً به هم تعلق دارند، سپس بر روی Extract shape کلیک کنید. این یک شکل جدید در صحنه ایجاد می کند. پس از آن عملیات مثلث های انتخاب شده را حذف کنید.

در مورد مش ما، روش 1 به خوبی کار کرد:



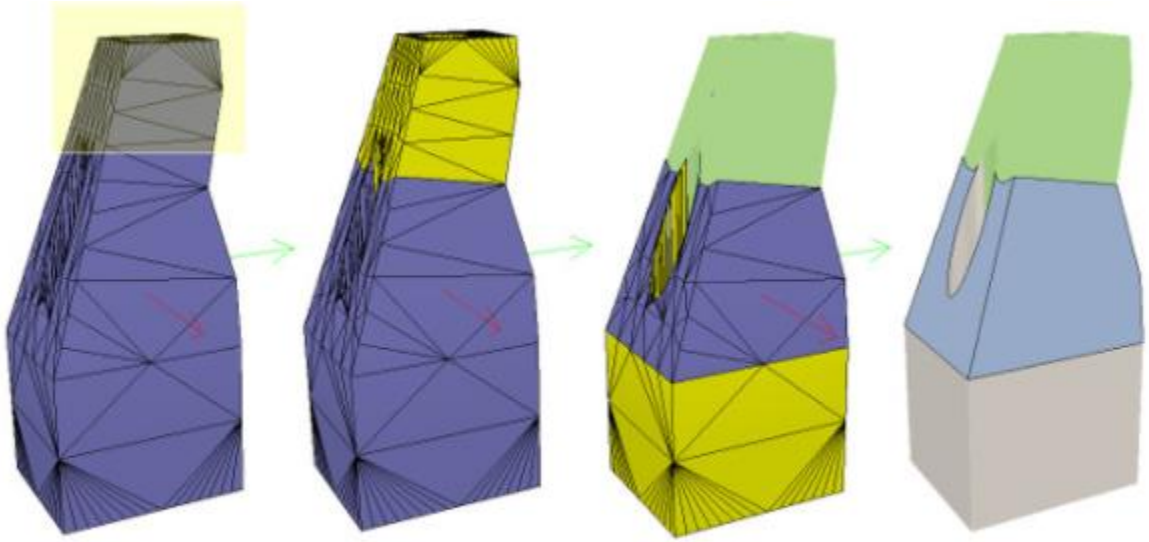
[Divided mesh]

اکنون، می‌توانیم شکل‌های فردی را بیشتر اصلاح و ساده کنیم. همچنین گاهی اوقات، اگر از بدنه محدب آن به جای آن استفاده شود، شکل ممکن است بهتر به نظر برسد. در غیر این صورت، برای به دست آوردن نتیجه دلخواه، مجبور خواهید بود از چندین تکنیک توضیح داده شده در بالا به طور مکرر استفاده کنید. به عنوان مثال مش زیر را در نظر بگیرید:



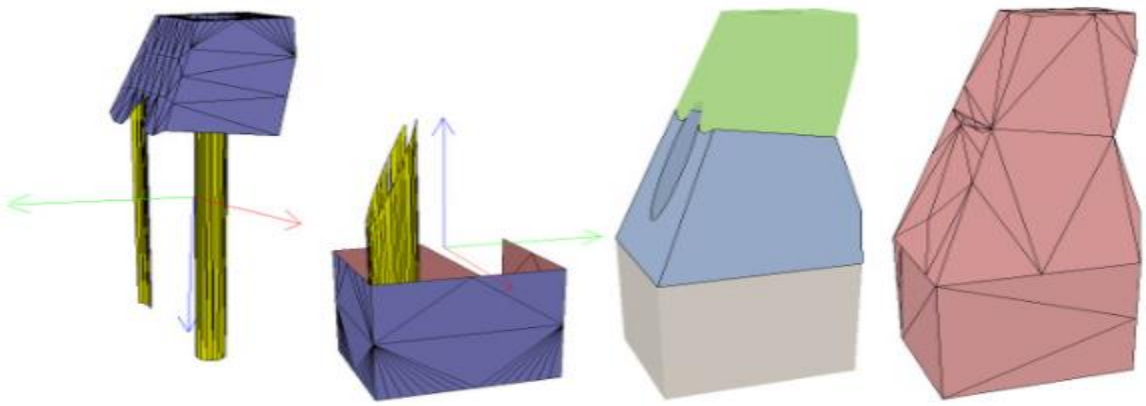
[Imported mesh]

مشکل شکل بالا این است که به دلیل حفره هایی که در آن وجود دارد، نمی توانیم آن را به خوبی ساده کنیم. بنابراین باید از طریق حالت ویرایش شکل، راه پیچیده‌تری را طی کنیم، جایی که می‌توانیم عناصر جداگانه‌ای را که منطقاً به یک موجود فرعی محدب تعلق دارند، استخراج کنیم. این فرآیند می تواند چندین بار تکرار شود: ابتدا 3 عنصر محدب تقریبی را استخراج می کنیم. در حال حاضر، مثلث هایی که بخشی از دو سوراخ هستند را نادیده می گیریم. هنگام ویرایش یک شکل در حالت ویرایش شکل، می‌توان لایه‌های دید را تغییر داد تا ببینیم چه چیزی توسط سایر موارد صحنه پوشش داده می‌شود.



[Step 1]

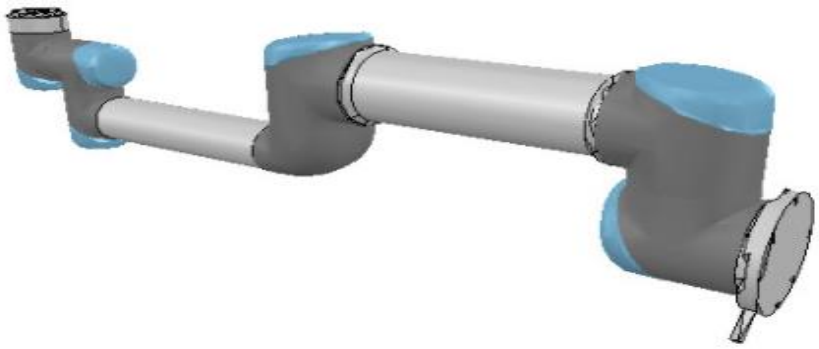
ما در نهایت با مجموعه ای از سه شکل مواجه می شویم، اما دو مورد از آنها نیاز به بهبود بیشتری دارند. حالا می توانیم مثلث هایی که بخشی از سوراخ ها هستند را پاک کنیم. در نهایت، بدنه محدب را به‌صورت جداگانه برای 3 شکل استخراج می‌کنیم، سپس آنها را با [نوار منو --> ویرایش --> گروه‌بندی/ادغام --> ادغام اشکال انتخاب شده] با هم ادغام می‌کنیم:



[Step 2]

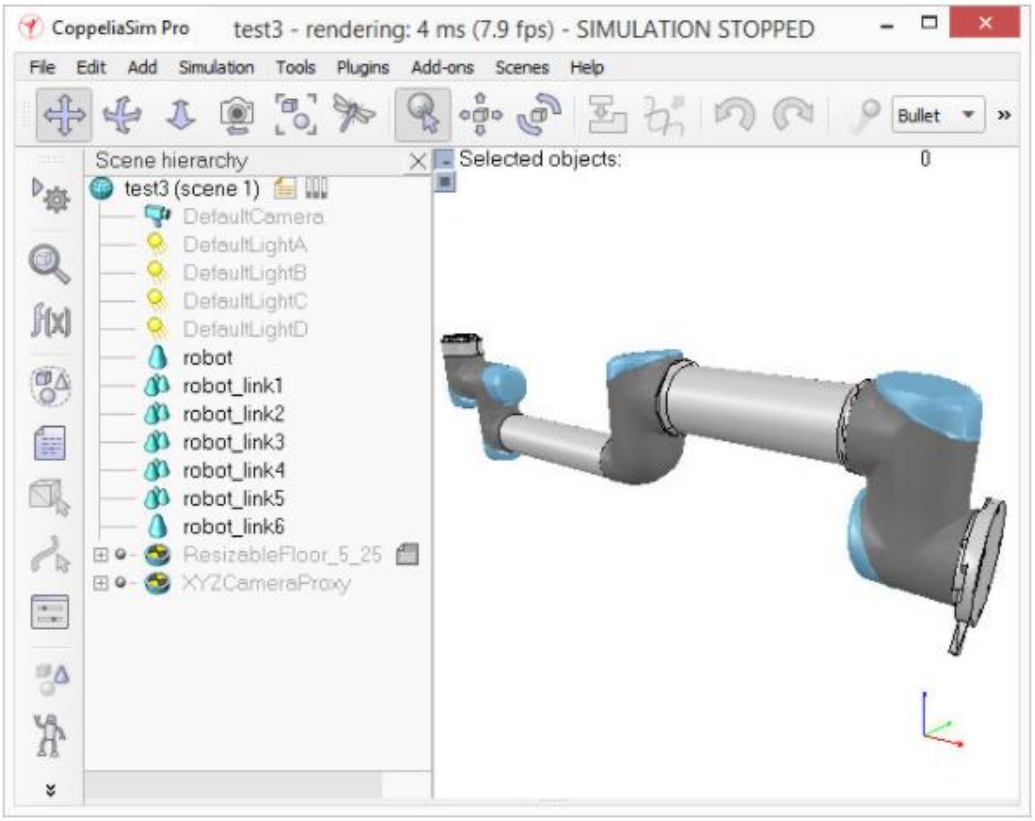
در CoppeliaSim، می‌توانیم یک زاویه سایه‌اندازی را تعیین کنیم که نشان می‌دهد شکل چگونه نمایش داده می‌شود. این پارامتر و چند پارامتر دیگر مانند رنگ شکل را می توان در ویژگی های شکل تنظیم کرد. به یاد داشته باشید که شکل ها طعم های مختلفی دارند. در این آموزش ما تا به حال فقط به اشکال ساده پرداخته ایم: یک شکل ساده دارای مجموعه ای از ویژگی های بصری است (یعنی یک رنگ، یک زاویه سایه و غیره). اگر دو شکل را ادغام کنید، نتیجه یک شکل ساده خواهد بود. شما همچنین می توانید اشکال را گروه بندی کنید، در این صورت، هر شکل ویژگی های بصری خود را حفظ می کند.

در مرحله بعد، می‌توانیم عناصری را که به‌طور منطقی به یکدیگر تعلق دارند، ادغام کنیم (اگر بخشی از یک عنصر صلب باشند، و اگر ویژگی‌های بصری یکسانی داشته باشند). سپس ویژگی های بصری عناصر مختلف را تغییر می دهیم. ساده‌ترین کار تنظیم چند شکل است که رنگ‌ها و ویژگی‌های بصری متفاوتی دارند، و اگر رنگ را با یک رشته خاص نامگذاری کنیم، بعداً می‌توانیم به راحتی آن رنگ را به صورت برنامه‌نویسی تغییر دهیم، همچنین اگر شکل بخشی از یک شکل ترکیبی باشد. سپس، همه شکل‌هایی را که ویژگی‌های بصری یکسانی دارند، انتخاب می‌کنیم، سپس شکلی را که قبلاً تنظیم شده بود را کنترل می‌کنیم، سپس روی Apply to selection کلیک می‌کنیم، یک‌بار برای Colors، یک‌بار برای ویژگی‌های دیگر، در خواص شکل: این همه را منتقل می‌کند. ویژگی های بصری به اشکال انتخاب شده (از جمله نام رنگ در صورت ارائه). در نهایت به 17 شکل مجزا می رسیم:



[Adjusted visual attributes]

اکنون می‌توانیم شکل‌هایی را که بخشی از همان پیوند هستند با [نوار منو --> ویرایش --> گروه‌بندی/ادغام -> شکل‌های انتخاب شده گروه‌بندی کنیم] گروه‌بندی کنیم. در نهایت به 7 شکل می رسیم: پایه ربات (یا پایه درخت سلسله مراتب ربات) و 6 لینک موبایل. همچنین مهم است که اشیاء خود را به درستی نامگذاری کنید: شما این کار را با دوبار کلیک بر روی نام مستعار شی در سلسله مراتب صحنه انجام می دهیم. به طور پیش فرض، اشکال به لایه دید 1 اختصاص داده می شود، اما می توان آنها را در ویژگی های مشترک شی تغییر داد. به طور پیش فرض، فقط لایه های دید 1-8 برای صحنه فعال می شوند. اکنون موارد زیر را داریم:



[Individual elements compositn the robot]

هنگامی که یک شکل ایجاد یا اصلاح می شود، CoppeliaSim به طور خودکار موقعیت و جهت قاب مرجع آن را تنظیم می کند. قاب مرجع یک شکل همیشه در مرکز هندسی شکل قرار می گیرد. جهت قاب به گونه ای انتخاب می شود که کادر مرزی شکل تا حد امکان کوچک باقی بماند. این همیشه خوب به نظر نمی رسد، اما ما همیشه می توانیم در هر زمانی چارچوب مرجع یک شکل را تغییر جهت دهیم. اکنون فریم های مرجع همه اشکال ایجاد شده خود را با [نوار منو --> ویرایش --> کادر محدودکننده جهت مجدد --> با قاب مرجع جهان] تغییر جهت می دهیم. گزینه های بیشتری برای تغییر جهت قاب مرجع در گفتگوی هندسه شکل دارید.

ساختن مفاصل:

اکنون ما از مفاصل/موتورها مراقبت خواهیم کرد. بیشتر اوقات، ما موقعیت و جهت دقیق هر یک از مفاصل را می دانیم. در این صورت، ما به سادگی اتصالات را با [-- Menu bar> -- Add> -- Joints> ...] اضافه می کنیم، سپس می توانیم موقعیت و جهت آنها را با گفتگوی موقعیت و گفتگوی جهت تغییر دهیم. در شرایط دیگر، ما فقط پارامترهای Denavit-Hartenberg یعنی (D-H)را داریم. در آن صورت، می‌توانیم مفاصل خود را از

طریق مدل ابزار واقع در Models/tools/Denavit-Hartenberg joint creator.ttm در مرورگر مدل بسازیم. در موارد دیگر، ما هیچ اطلاعاتی در مورد مکان ها و جهت گیری های مشترک نداریم. سپس، باید آنها را از مش وارد شده استخراج کنیم. فرض کنیم این مورد ماست. به جای کار بر روی مش تغییر یافته و تقریبی تر، یک صحنه جدید باز می کنیم و دوباره داده های CAD اصلی را وارد می کنیم. اغلب اوقات، ما می توانیم مش ها یا اشکال ابتدایی را از مش اصلی استخراج کنیم. اولین مرحله تقسیم مش اصلی است. اگر کار نکرد، آن را از طریق حالت ویرایش مثلث انجام می دهیم. بیایید فرض کنیم که بتوانیم مش اصلی را تقسیم کنیم. ما اکنون اشیاء کوچکتری داریم که می توانیم آنها را بررسی کنیم. ما به دنبال شکل‌های چرخشی هستیم که می‌توانند به عنوان مرجع برای ایجاد اتصالات در مکان‌هایشان، با جهت‌گیری یکسان استفاده شوند. ابتدا تمام اشیایی که مورد نیاز نیستند را بردارید. برای تجسم / دستکاری آسان تر، گاهی اوقات کار در چندین صحنه باز نیز مفید است. در مورد ما، ابتدا روی پایه ربات تمرکز می کنیم: این ربات حاوی یک استوانه است که موقعیت صحیح اولین اتصال را دارد. در حالت ویرایش مثلث داریم:

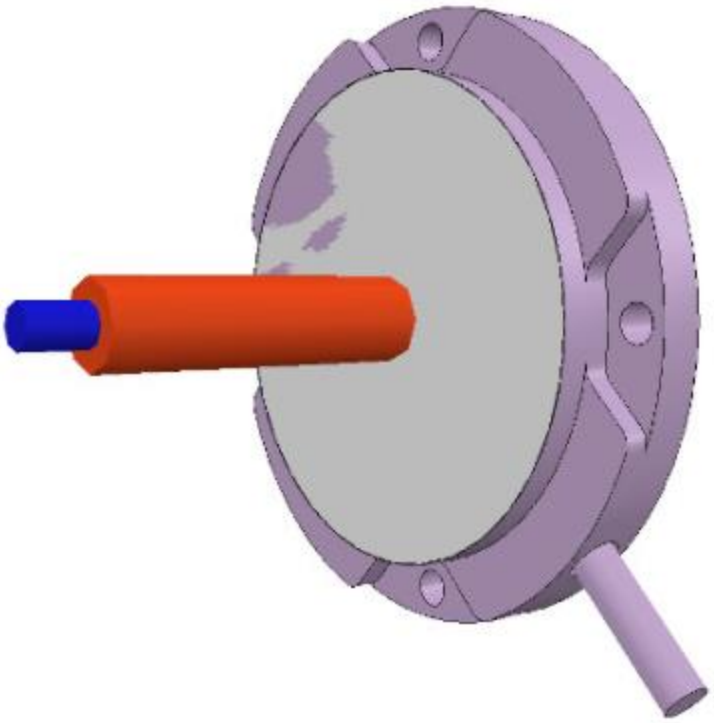
[Robot base: normal and triangle edit mode visualization]

نمای دوربین را از طریق دکمه نوار ابزار انتخاب صفحه تغییر می دهیم تا از کنار به شیء نگاه کنیم. دکمه نوار ابزار مناسب برای مشاهده می تواند برای کادربندی صحیح شی در نسخه مفید باشد. سپس به حالت ویرایش رأس

سوئیچ می کنیم و تمام رئوس متعلق به دیسک بالایی را انتخاب می کنیم. به یاد داشته باشید که با روشن/خاموش کردن برخی از لایه ها، می توانیم اشیاء دیگر را در صحنه پنهان کنیم. سپس به حالت ویرایش مثلث برمی گردیم:

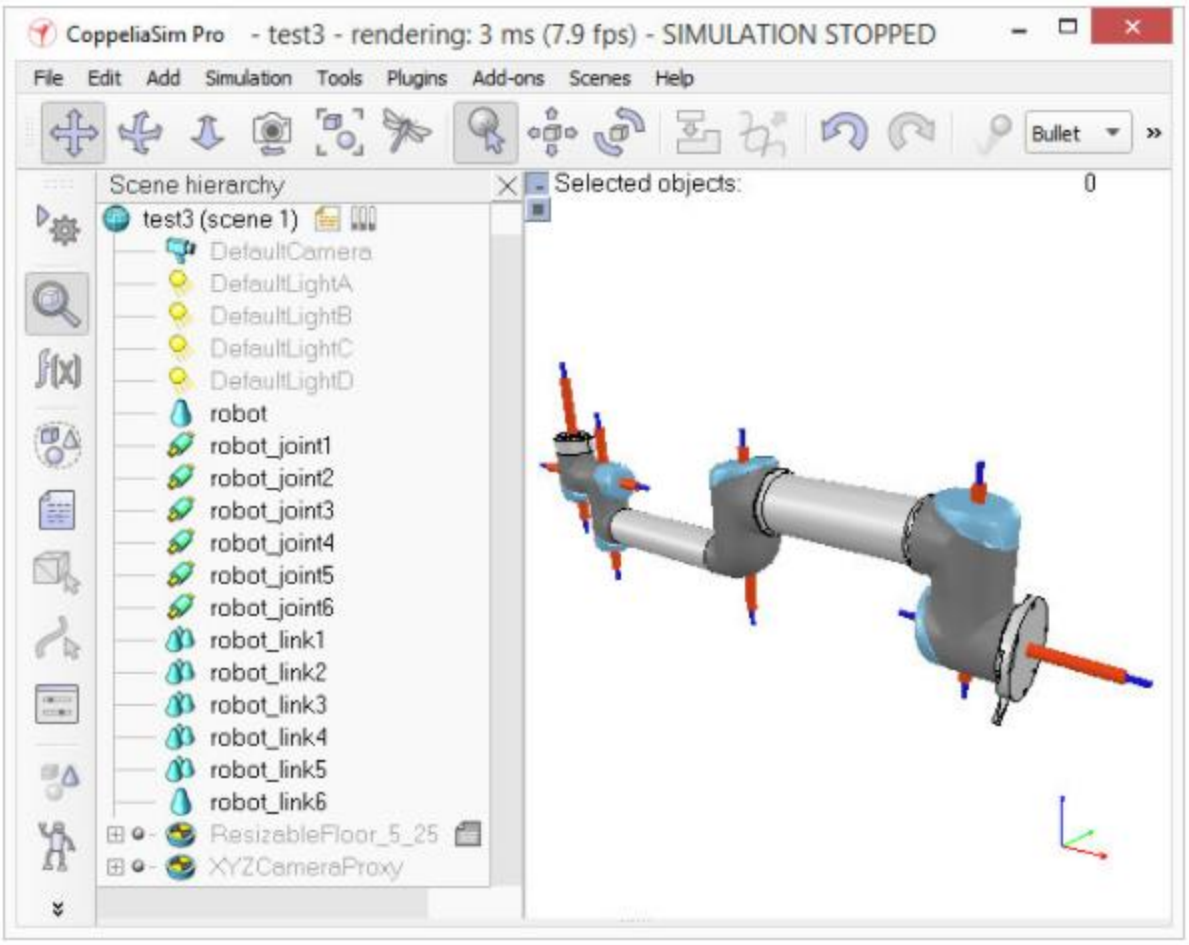
[Selected upper disc, vertex edit mode (1 & 2), triangle edit mode (3)]

اکنون بر روی Extract cylinder کلیک می کنیم (Extract shape نیز در آن صورت کار می کند)، این فقط یک شکل استوانه ای در صحنه، بر اساس مثلث های انتخاب شده ایجاد می کند. از حالت ویرایش خارج می شویم و تغییرات را کنار می گذاریم. اکنون یک اتصال چرخشی را با [-- <Menu bar> -- <Add> -- <Joint> Revolute]اضافه می کنیم، آن را انتخاب می کنیم، سپس شکل سیلندر استخراج شده را کنترل می کنیم. در محاوره موقعیت، در زبانه موقعیت، روی Apply to selection کلیک می کنیم: این اساساً موقعیت x/y/z سیلندر را روی اتصال کپی می کند. هر دو موقعیت اکنون یکسان هستند. در گفتگوی جهت‌گیری، در زبانه جهت‌یابی، روی Apply to selection نیز کلیک می‌کنیم: جهت اشیاء انتخاب‌شده ما اکنون نیز یکسان است. گاهی اوقات، برای به دست آوردن جهت یا جهت چرخش صحیح، لازم است مفصل را حدود 180/90 درجه در اطراف چارچوب مرجع خود بچرخانیم. در صورت نیاز می‌توانیم این کار را در برگه چرخش آن دیالوگ انجام دهیم (در این صورت، فراموش نکنید که روی دکمه Own frame کلیک کنید). به روشی مشابه می‌توانیم مفصل را در امتداد محور آن جابجا کنیم یا حتی عملیات پیچیده‌تری انجام دهیم. این چیزی است که ما داریم:



[Joint in correct location, with the correct orientation]

اکنون مفصل را در صحنه اصلی خود کپی می کنیم و آن را ذخیره می کنیم (فراموش نکنید کار خود را به طور منظم ذخیره کنید! عملکرد لغو/دوباره مفید است، اما از شما در برابر سایر حوادث محافظت نمی کند). ما روش فوق را برای تمام مفاصل ربات خود تکرار می کنیم، سپس نام آنها را تغییر می دهیم. ما همچنین تمام اتصالات را در خواص مفصل کمی طولانی تر می کنیم تا همه آنها را ببینیم. به طور پیش فرض، اتصالات به لایه دید 2 اختصاص داده می شوند، اما می توان آنها را در ویژگی های مشترک شی تغییر داد. اکنون همه اتصالات را به لایه دید 10 اختصاص می دهیم، سپس لایه دید 10 را به طور موقت فعال می کنیم تا صحنه نیز آن مفاصل را تجسم کند (به طور پیش فرض، فقط لایه های دید 1-8 برای صحنه فعال می شوند). این چیزی است که ما داریم:



[Joints in correct configuration]

در این مرحله، می‌توانیم شروع به ساخت سلسله مراتب مدل کنیم و تعریف مدل را تمام کنیم. اما اگر بخواهیم ربات opur به صورت پویا فعال شود، یک مرحله میانی اضافی وجود دارد:

ساخت اشکال پویا:

اگر می خواهیم ربات ما به صورت پویا فعال شود، یعنی به برخورد، سقوط و غیره واکنش نشان دهد، باید اشکال را به درستی ایجاد/پیکربندی کنیم: یک شکل می تواند:

پویا یا استاتیک: یک شکل پویا (یا غیر ایستا) می افتد و تحت تأثیر نیروها/گشتاورهای خارجی قرار می گیرد. از طرف دیگر یک شکل ایستا (یا غیر دینامیک) در جای خود باقی می ماند یا حرکت والد خود را در سلسله مراتب صحنه دنبال می کند.

پاسخ‌پذیر یا غیر پاسخ‌پذیر: یک شکل پاسخ‌پذیر باعث واکنش برخورد با سایر اشکال پاسخ‌پذیر می‌شود. آنها (و/یا) برخورد دهنده آنها در صورت پویا بودن در حرکت خود تحت تأثیر قرار می گیرند. از سوی دیگر، اشکال غیر قابل پاسخ در صورت برخورد با اشکال دیگر، پاسخ برخورد را محاسبه نمی کنند.

دو نکته بالا در اینجا نشان داده شده است. اشکال پاسخگو باید تا حد امکان ساده باشند تا امکان شبیه سازی سریع و پایدار را فراهم کنند. یک موتور فیزیک می تواند 5 نوع شکل زیر را با درجات مختلف سرعت و ثبات شبیه سازی کند:

اشکال خالص: یک شکل خالص پایدار خواهد بود و توسط موتور فیزیک بسیار کارآمد اداره می شود. اشکال این است که اشکال خالص در هندسه محدود هستند: بیشتر مکعب ها، استوانه ها و کره ها. در صورت امکان، از آن‌ها برای مواردی استفاده کنید که برای مدت طولانی‌تری با اقلام دیگر در تماس هستند (مثلاً پاهای یک ربات انسان‌نما، پایه دستکاری‌کننده سریال، انگشتان دستگیر و غیره). اشکال خالص را می توان با [-- Menu bar> -- Add> [Primitive shapeایجاد کرد.

اشکال مرکب خالص: شکل مرکب خالص مجموعه ای از چندین شکل خالص است. عملکرد تقریباً به همان شکل خالص دارد و خواص مشابهی دارد. اشکال ترکیبی خالص را می توان با گروه بندی چندین شکل خالص [نوار منو --> ویرایش --> گروه بندی/ادغام --> گروه بندی اشکال انتخاب شده] تولید کرد.

اشکال محدب: یک شکل محدب کمی پایدارتر خواهد بود و زمانی که موتور فیزیک آن را مدیریت می کند کمی زمان محاسبات بیشتری را می گیرد. هندسه کلی تری (فقط شرط: باید محدب باشد) نسبت به اشکال خالص اجازه می دهد. در صورت امکان، از اشکال محدب برای مواردی استفاده کنید که به طور پراکنده با موارد دیگر در تماس هستند (مثلاً پیوندهای مختلف یک روبات). اشکال محدب را می توان با [نوار منو --> افزودن --> انتخاب بدنه محدب] یا با [نوار منو --> ویرایش --> انتخاب شکل به شکل های محدب] ایجاد کرد.

اشکال محدب مرکب یا اشکال محدب تجزیه شده: شکل متلاشی شده محدب مجموعه ای از چندین شکل محدب است. تقریباً به خوبی اشکال محدب عمل می کند و ویژگی های مشابهی دارد. اشکال تجزیه شده محدب را می توان با گروه بندی چندین شکل محدب [نوار منو --> ویرایش --> گروه بندی/ادغام --> اشکال انتخابی گروه]،

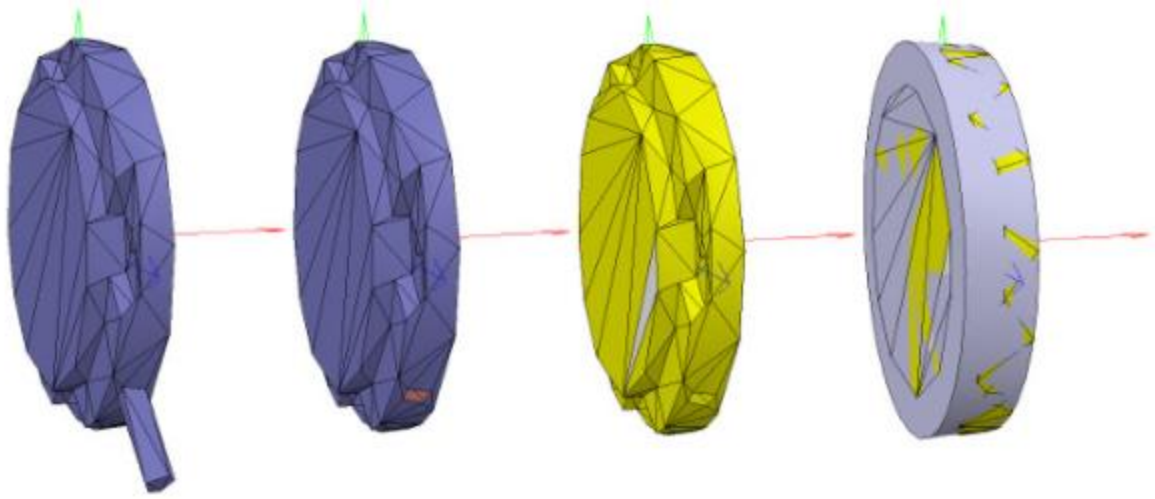
با [نوار منو --> افزودن --> تجزیه محدب انتخاب...]، یا با [نوار منو --> ویرایش --> انتخاب شکل به تجزیه محدب آن...].

اشکال تصادفی: شکل تصادفی شکلی است که محدب و خالص نباشد. به طور کلی عملکرد ضعیفی دارد (سرعت محاسبات و پایداری). تا حد امکان از استفاده از اشکال تصادفی خودداری کنید.

بنابراین ترتیب اولویت به این صورت خواهد بود: اشکال خالص، اشکال ترکیبی خالص، اشکال محدب، اشکال محدب مرکب و در نهایت اشکال تصادفی. حتما این صفحه را نیز بخوانید. در مورد رباتی که می خواهیم بسازیم، پایه ربات را به صورت یک استوانه خالص و سایر پیوندها را به صورت محدب یا محدب تجزیه شده می سازیم.

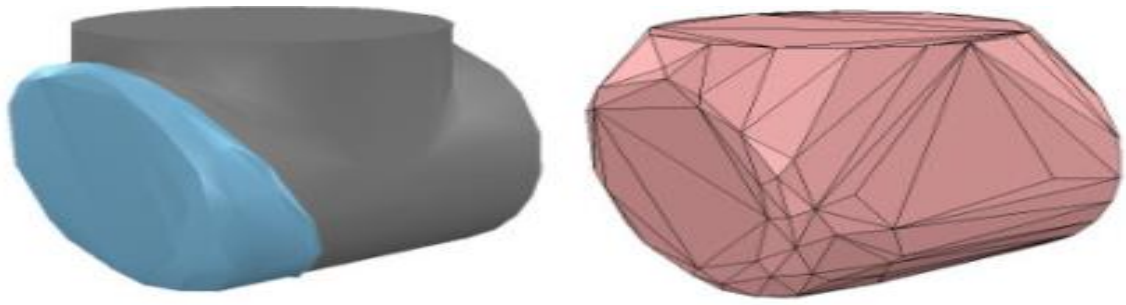
ما می‌توانیم از اشکال فعال پویا به عنوان بخش‌های قابل مشاهده ربات نیز استفاده کنیم، اما احتمالاً به اندازه کافی خوب به نظر نمی‌رسد. بنابراین، در عوض، برای هر شکل قابل مشاهده ای که در قسمت اول آموزش ایجاد کرده ایم، یک همتای فعال پویا می سازیم، که آن را مخفی نگه می داریم: قسمت مخفی مدل پویا را نشان می دهد و منحصراً توسط موتور فیزیک استفاده می شود، در حالی که بخش قابل مشاهده برای تجسم، بلکه برای محاسبات حداقل فاصله، تشخیص سنسور مجاورت و غیره استفاده خواهد شد.

ربات شی را انتخاب می کنیم، آن را در یک صحنه جدید کپی و پیست می کنیم (برای اینکه مدل اصلی دست نخورده بماند) و حالت ویرایش مثلث را شروع می کنیم. اگر ربات شی یک شکل مرکب بود، ابتدا باید آن را از حالت گروه بندی خارج می کردیم ([نوار منو --> ویرایش --> گروه بندی/ادغام --> گروه بندی کردن اشکال انتخاب شده]) سپس تک تک اشکال را ادغام می کردیم ([نوار منو --> ویرایش --> گروه بندی/ادغام --> ادغام اشکال انتخاب شده]) قبل از اینکه بتوانید حالت ویرایش مثلث را شروع کنید. حالا چند مثلثی که نشان دهنده کابل برق هستند را انتخاب کرده و پاک می کنیم. سپس تمام مثلث های آن شکل را انتخاب کرده و Extract cylinder را کلیک می کنیم. اکنون می توانیم حالت ویرایش را ترک کنیم و شی پایه خود را به صورت یک استوانه خالص نشان می دهیم:

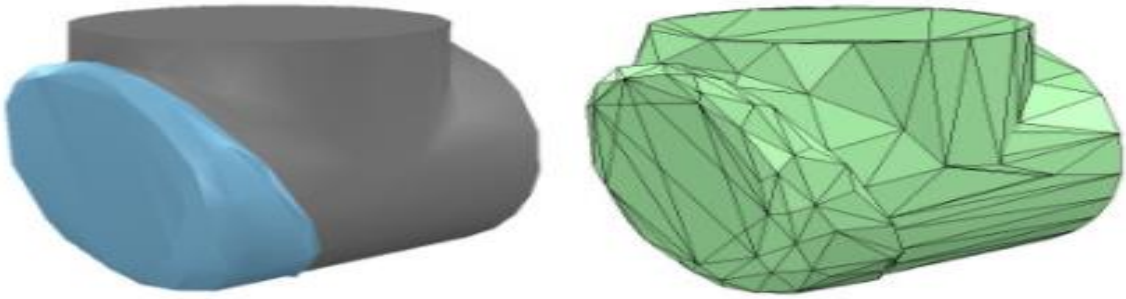


[Pure cylinder generation procedure, in the triangle edit mode]

شکل جدید را (با دوبار کلیک بر روی نام مستعار آن در سلسله مراتب صحنه) به robot_dyn تغییر نام می دهیم، آن را به لایه visibility 9 اختصاص می دهیم، سپس آن را در صحنه اصلی کپی می کنیم. بقیه پیوندها به صورت اشکال محدب یا اشکال محدب مرکب مدل خواهند شد. اکنون اولین پیوند همراه (یعنی شی robot_link1)را انتخاب می کنیم و با [-- <Menu bar <Add -- [Convex hull of selectionاز آن یک شکل محدب ایجاد می کنیم. ما آن را به robot_link_dyn1 تغییر نام دادیم و آن را به لایه دید 9 اختصاص دادیم. زمانی که استخراج بدنه محدب جزئیات کافی از شکل اصلی را حفظ نمی کند، همچنان می توانید چندین بدنه محدب را به صورت دستی از عناصر سازنده آن استخراج کنید، سپس تمام بدنه های محدب را با آنها گروه بندی کنید. [نوار منو --> ویرایش --> گروه بندی/ادغام --> گروه بندی اشکال انتخاب شده]. اگر به نظر مشکل ساز یا زمان بر است، می توانید به طور خودکار یک شکل تجزیه شده محدب را با [نوار منو --> افزودن --> تجزیه محدب انتخاب...] استخراج کنید:

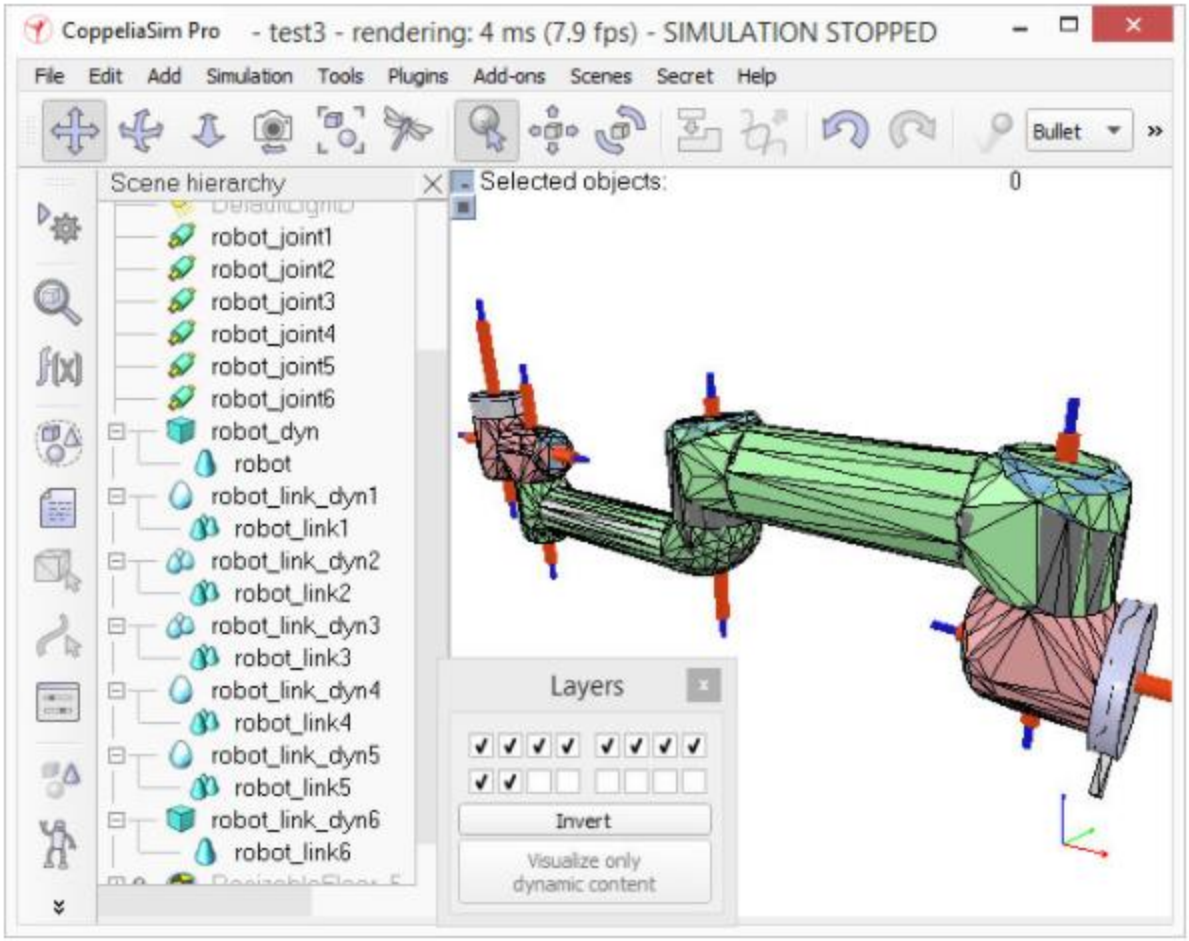


[Original shape, and convex shape pendant]



[Original shape, and convex decomposed shape pendant]

اکنون همین روش را برای تمام پیوندهای ربات باقی مانده تکرار می کنیم. پس از انجام این کار، هر شکل قابل مشاهده را به آویز دینامیکی نامرئی مربوطه وصل می کنیم. ما این کار را با انتخاب ابتدا شکل قابل مشاهده انجام می دهیم، سپس از طریق کنترل کلیک کنید و آویز پویا آن را انتخاب کنید و سپس [نوار منو --> ویرایش --> آخرین شی انتخاب شده را بسازید. همین نتیجه را می توان با کشیدن شکل مرئی بر روی آویز پویا در سلسله مراتب صحنه به دست آورد:



[Visible shapes attached to their dynamic pendants]

ما هنوز باید به چند چیز توجه داشته باشیم: اول، از آنجایی که می‌خواهیم اشکال پویا فقط برای موتور فیزیک قابل مشاهده باشد، اما نه برای سایر ماژول‌های محاسباتی، تیک تمام ویژگی‌های ویژه شی برای اشکال پویا را در ویژگی‌های مشترک شیء برداریم. .

سپس، ما هنوز باید اشکال پویا را به عنوان پویا و پاسخ پذیر پیکربندی کنیم. ما این کار را در خواص دینامیک شکل انجام می دهیم. ابتدا شکل دینامیک پایه (یعنی robot_dyn)را انتخاب کنید، سپس مورد Body is Responable را علامت بزنید. اولین 4 پرچم ماسک پاسخ‌پذیر محلی را فعال کنید و 4 پرچم ماسک پاسخ‌پذیر محلی آخر را غیرفعال کنید: مهم است که پیوندهای پاسخ‌پذیر متوالی با یکدیگر برخورد نکنند. برای اولین پیوند پویا همراه در ربات ما (یعنی robot_link_dyn1)، مورد Body is Responable را نیز فعال می‌کنیم، اما

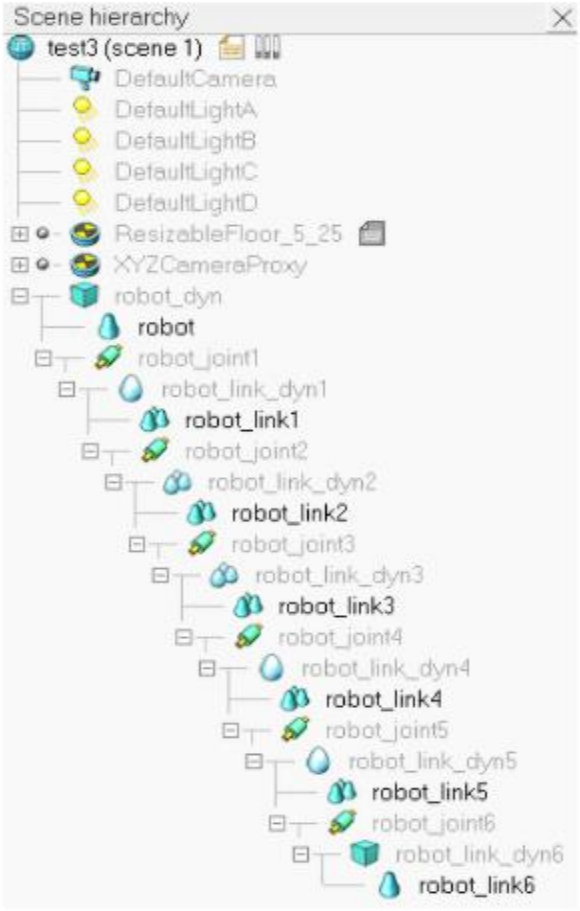
این بار 4 پرچم ماسک پاسخ‌پذیر محلی را غیرفعال می‌کنیم و 4 پرچم ماسک پاسخ‌پذیر محلی را فعال می‌کنیم. ما روش فوق را با سایر پیوندهای پویا تکرار می کنیم، در حالی که همیشه پرچم های ماسک پاسخ پذیر محلی را متناوب می کنیم: هنگامی که مدل تعریف می شود، اشکال پویا متوالی ربات هنگام تعامل با یکدیگر، هیچ پاسخ برخوردی ایجاد نمی کند. سعی کنید همیشه ساختاری داشته باشید که در آن پایه پویا ربات و آخرین پیوند پویا ربات تنها 4 پرچم اول ماسک پاسخ‌پذیر محلی را فعال کرده باشد، تا بتوانیم ربات را به یک پلت فرم موبایل متصل کنیم یا گیره به آخرین پیوند پویا ربات بدون تداخل برخورد دینامیکی.

در نهایت، ما هنوز باید اشکال پویا خود را به عنوان Body پویا تگ کنیم. ما این کار را در خواص دینامیک شکل نیز انجام می دهیم. سپس می‌توانیم ویژگی‌های تانسور جرم و اینرسی را به‌صورت دستی وارد کنیم، یا با کلیک روی محاسبه خواص جرم و اینرسی برای اشکال محدب انتخاب‌شده، آن مقادیر را به‌طور خودکار محاسبه کنیم (توصیه می‌شود). این و آن ملاحظات طراحی پویا را نیز به خاطر بسپارید. این پایه پویا ربات یک مورد خاص است: بیشتر اوقات ما می خواهیم پایه ربات (یعنی robot_dyn) غیر دینامیک (یعنی استاتیک) باشد، در غیر این صورت، اگر به تنهایی استفاده شود، ربات ممکن است در حین حرکت سقوط کند. اما به محض اینکه پایه ربات را به یک پلت فرم متحرک وصل می کنیم، می خواهیم پایه پویا (یعنی غیر ایستا) شود. ما این کار را با فعال کردن Set to dynamic if gets prind item انجام می دهیم، سپس غیرفعال کردن Body is dynamic آیتم است. اکنون شبیه سازی را اجرا کنید: همه اشکال پویا، به جز پایه ربات، باید سقوط کنند. که اشکال بصری متصل آویزهای پویا خود را دنبال خواهند کرد.

تعریف مدل:

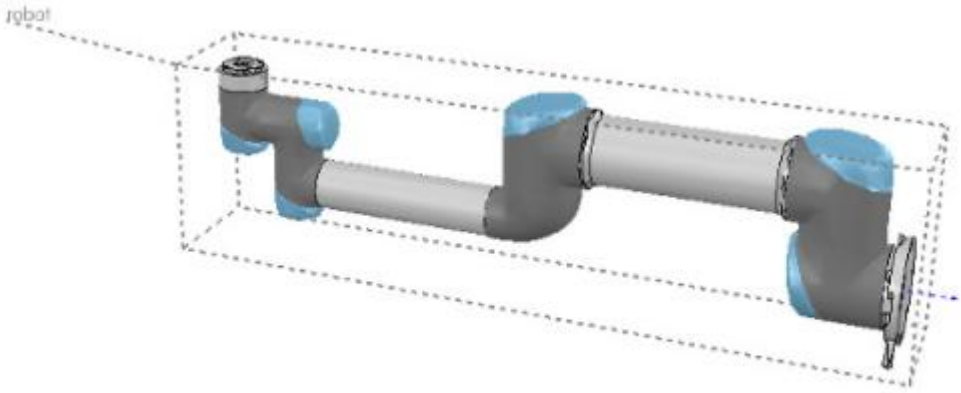
حالا ما آماده ایم تا مدل خود را تعریف کنیم. ما با ساختن هرارشی مدل شروع می کنیم: آخرین پیوند ربات پویا (robot_link_dyn6)را با انتخاب robot_link_dyn6 به مفصل مربوطه آن (robot_joint6)وصل می کنیم، سپس robot_joint6 را کنترل می کنیم، سپس [نوار منو --> ویرایش --> ساخت آخرین شی انتخاب شده والدین]. همچنین می‌توانستیم این مرحله را با کشیدن شی robot_link_dyn6 روی robot_link6 در سلسله مراتب صحنه انجام دهیم. ما اکنون robot_joint6 را به robot_link_dyn5 وصل می کنیم، و به همین ترتیب، تا زمانی که به پایه ربات رسیدیم. اکنون سلسله مراتب صحنه زیر را داریم:

تصویر مرتبط درصفحه بعد.



[Robot model hierarchy]

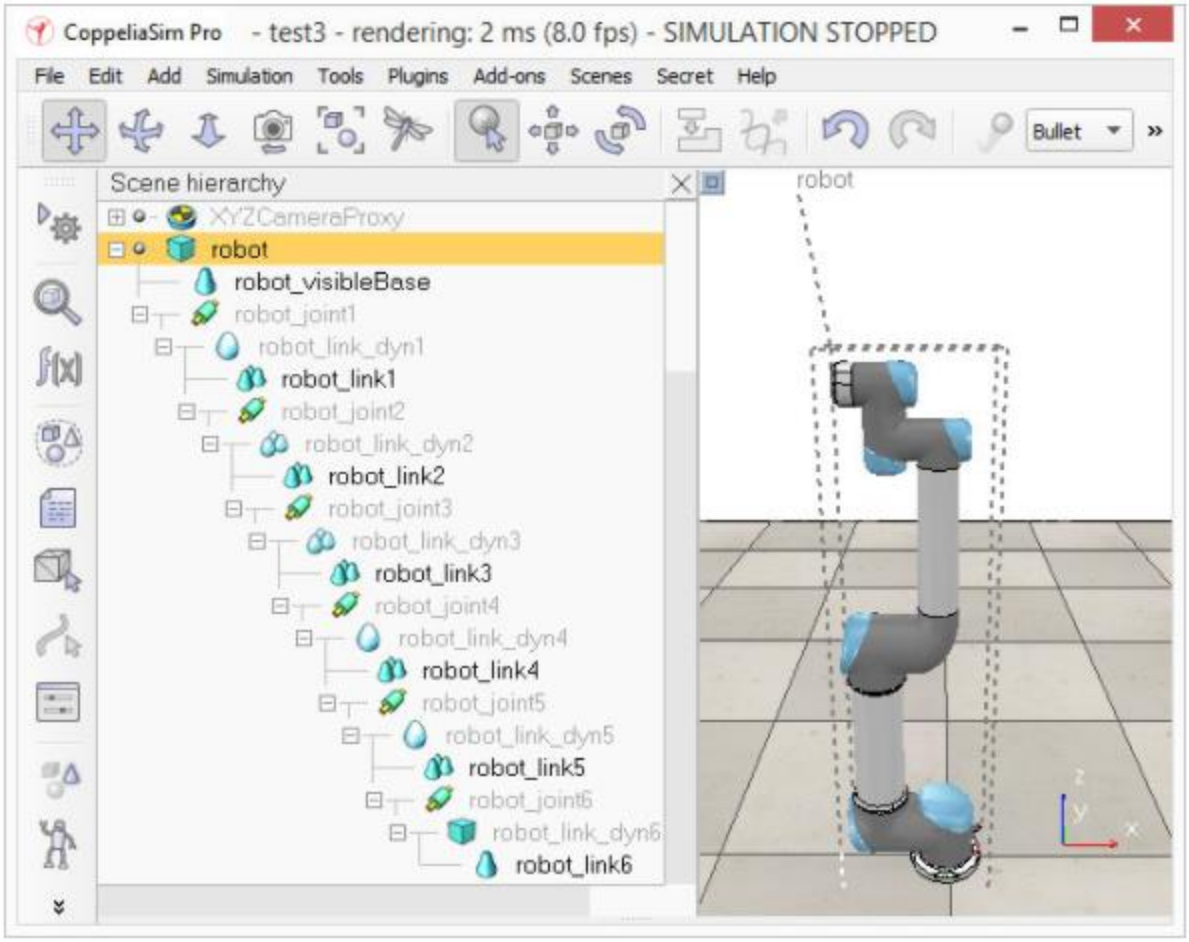
داشتن یک نام مستعار ساده برای پایه مدل خوب و منطقی تر است، زیرا پایه مدل نیز خود مدل را نشان می دهد. بنابراین نام robot_dyn را به robot تغییر می دهیم. اکنون پایه درخت سلسله مراتب (یعنی ربات شی) را انتخاب می کنیم و در ویژگی های مشترک شی، Object is model base را فعال می کنیم. یک جعبه محدود کننده مدل ظاهر شد که کل ربات را در بر می گرفت. با این حال به نظر می رسد که جعبه محدود کننده بیش از حد بزرگ است: این به این دلیل است که جعبه مرزی موارد نامرئی مانند اتصالات را نیز در بر می گیرد. اکنون با فعال کردن گزینه Don't show as inside selection model برای همه مفاصل، اتصالات را از کادر محدود کننده مدل حذف می کنیم. ما می‌توانیم همین رویه را برای همه موارد نامرئی در مدل خود انجام دهیم. این همچنین یک گزینه مفید برای حذف سنسورهای بزرگ یا موارد دیگر از جعبه محدود کننده مدل است. اکنون وضعیت زیر را داریم:



[Robot model bounding box]

ما اکنون از مدل خود در برابر تغییرات تصادفی محافظت می کنیم. همه اشیاء قابل مشاهده در ربات را انتخاب می کنیم، سپس به جای آن Select base of model را فعال می کنیم: اگر اکنون روی یک پیوند قابل مشاهده در صحنه کلیک کنیم، پایه ربات به جای آن انتخاب می شود. این به ما امکان می دهد مدل را طوری دستکاری کنیم که گویی یک شی واحد است. ما همچنان می‌توانیم اشیاء قابل مشاهده در ربات را از طریق کنترل-شفت-کلیک کردن در صحنه یا با انتخاب شیء در سلسله مراتب صحنه انتخاب کنیم. اکنون ربات را در موقعیت/جهت پیش فرض صحیح قرار می دهیم. ابتدا، صحنه فعلی را به عنوان مرجع ذخیره می کنیم (به عنوان مثال اگر در مرحله بعدی نیاز به وارد کردن داده های CAD با جهت گیری یکسان در ربات فعلی داشته باشیم). سپس مدل را انتخاب کرده و موقعیت/جهت آن را به طور مناسب اصلاح می کنیم. قرار دادن مدل (یعنی شی پایه آن) در X=0 و Y=0 عمل خوبی در نظر گرفته می شود.

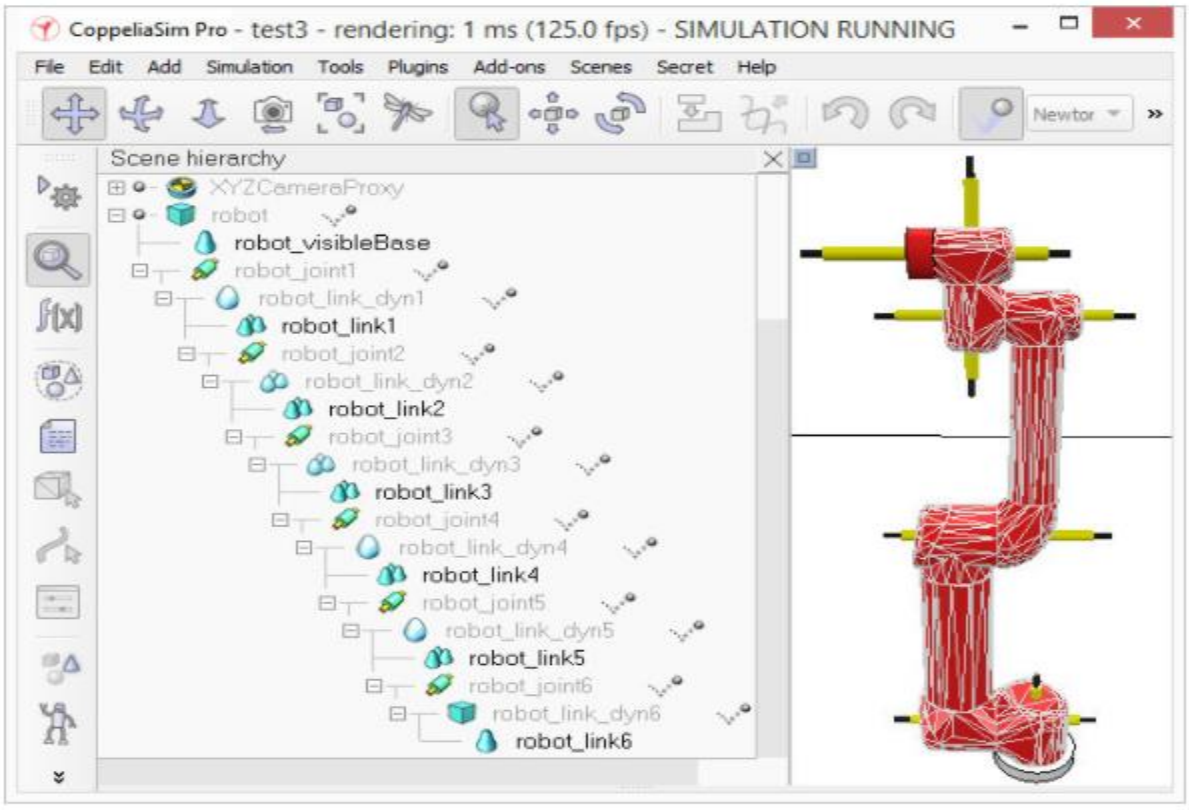
تصویر مرتبط در صفحه بعد.



[Robot model in default configuration]

اکنون شبیه سازی را اجرا می کنیم: ربات فرو می ریزد، زیرا مفاصل به طور پیش فرض کنترل نمی شوند. وقتی اتصالات را در مرحله قبل اضافه کردیم، اتصالات را در حالت نیرو/گشتاور ایجاد کردیم، اما موتور یا کنترل کننده آنها (به طور پیش فرض) غیرفعال بود. اکنون می توانیم مفاصل خود را مطابق با نیازهای خود تنظیم کنیم. در مورد ما، ما یک کنترلر PID ساده برای هر یک از آنها می خواهیم. در خصوصیات دینامیک مفصل، روی Propellers enabled کلیک می کنیم و حداکثر گشتاور را تنظیم می کنیم. سپس روی Control loop enabled کلیک کرده و Position control (PID) را انتخاب می کنیم. اکنون دوباره شبیه سازی را اجرا می کنیم: ربات باید موقعیت خود را حفظ کند. سعی کنید موتور فیزیک فعلی را تغییر دهید تا ببینید آیا این رفتار در همه موتورهای فیزیک پشتیبانی شده سازگار است یا خیر. شما می توانید این کار را از طریق دکمه نوار ابزار مناسب یا در ویژگی های دینامیک عمومی انجام دهید.

در طول شبیه سازی، ما اکنون محتوای پویا صحنه را از طریق دکمه تجسم محتوای پویا و نوار ابزار تأیید تأیید می کنیم. اکنون فقط مواردی که توسط موتور فیزیک در نظر گرفته شده اند نمایش داده می شوند و نمایشگر دارای کد رنگی است. بسیار مهم است که همیشه این کار را انجام دهید، مخصوصاً زمانی که مدل پویا شما مطابق انتظار رفتار نمی کند تا به سرعت مدل را اشکال زدایی کنید. به طور مشابه، همیشه در طول شبیه سازی به سلسله مراتب صحنه نگاه کنید: اشیاء فعال پویا باید یک نماد توپی را در سمت راست نام مستعار خود نشان دهند.



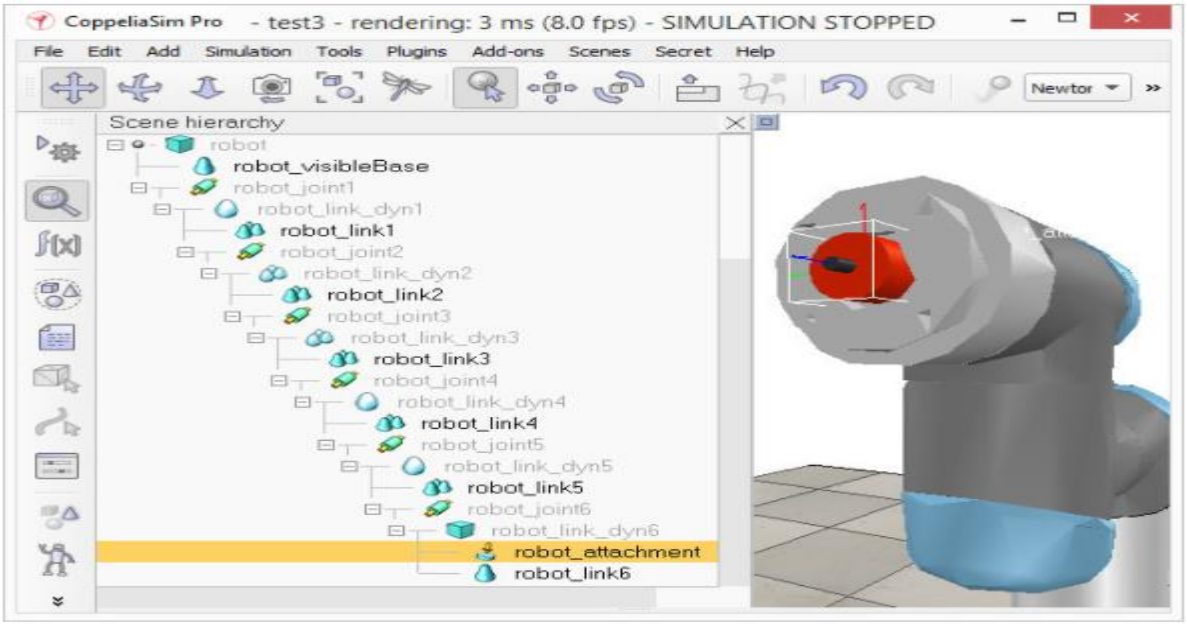
[Dynamic content visualization & verification]

در نهایت، باید ربات را طوری آماده کنیم که بتوانیم به راحتی یک گیره را به آن وصل کنیم یا به راحتی ربات را به یک پلت فرم متحرک وصل کنیم (مثلا). دو شکل فعال پویا را می‌توان به دو روش مختلف به همدیگر متصل کرد:

با گروه بندی آنها: شکل ها را انتخاب کنید، سپس [نوار منو --> ویرایش --> گروه بندی/ادغام --> گروه بندی اشکال انتخاب شده].

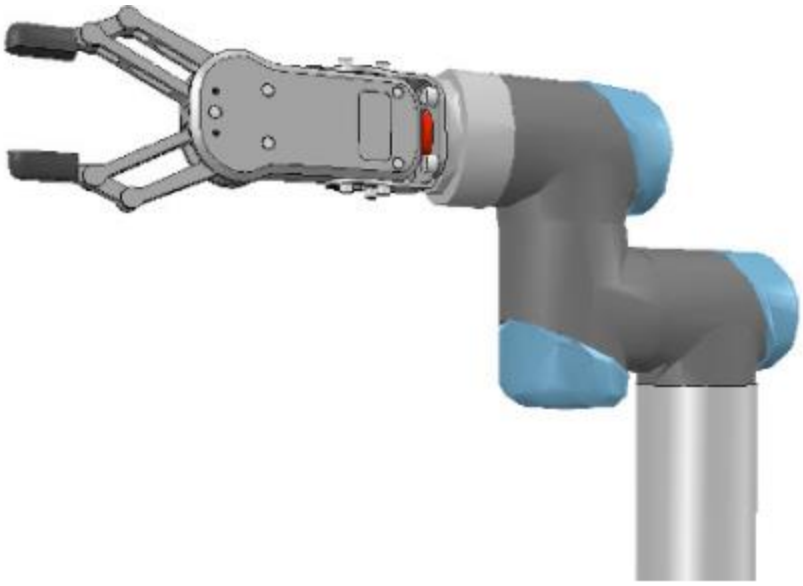
با اتصال آنها از طریق یک سنسور نیرو/گشتاور: یک حسگر گشتاور نیرو همچنین می تواند به عنوان یک پیوند صلب بین دو شکل جداگانه فعال به صورت پویا عمل کند.

در مورد ما، تنها گزینه 2 مورد علاقه است. یک حسگر نیرو/گشتاور با [-- <Menu bar -- <Add Force sensor]ایجاد می کنیم، سپس آن را به نوک ربات منتقل می کنیم، سپس آن را به شی robot_link_dyn6 متصل می کنیم. اندازه و ظاهر بصری آن را به طور مناسب تغییر می دهیم (سنسور نیرو/گشتاور قرمز اغلب به عنوان یک نقطه اتصال اختیاری درک می شود، مدل های مختلف ربات موجود را بررسی کنید). همچنین نام مستعار آن را به robot_attachment تغییر می دهیم:



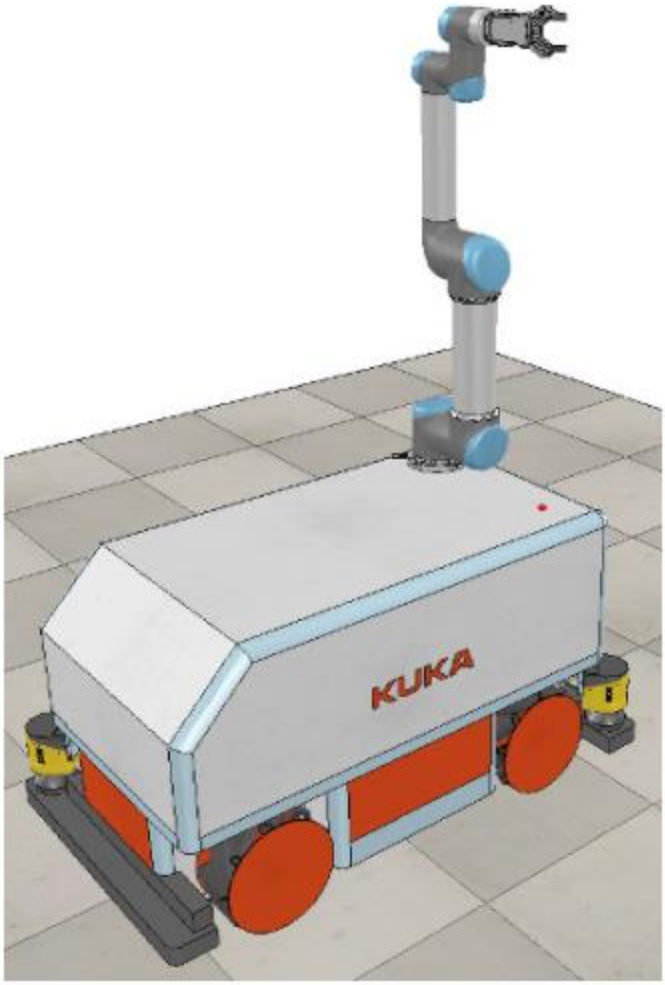
[Attachment force/torque sensor]

اکنون یک مدل گیره را به داخل صحنه می کشیم، آن را انتخاب می کنیم، سپس روی سنسور نیروی پیوست کنترل کلیک می کنیم، سپس روی دکمه اسمبلینگ/جداسازی نوار ابزار کلیک می کنیم. گیره در جای خود قرار می گیرد:



[Attached gripper]

گیره می‌دانست که چگونه خودش را بچسباند، زیرا در طول تعریف مدل خود به درستی پیکربندی شده بود. اکنون همچنین باید مدل ربات را به درستی پیکربندی کنیم تا بداند که چگونه خود را به یک پایه موبایل متصل کند. ما مدل ربات را انتخاب می کنیم، سپس روی اسمبلینگ در ویژگی های مشترک شیء کلیک می کنیم. یک رشته خالی برای مقادیر مطابقت «والد» تنظیم کنید، سپس روی تنظیم ماتریس کلیک کنید. این ماتریس تبدیل محلی شی پایه فعلی را به خاطر می‌سپارد و از آن برای موقعیت/جهت‌یابی خود نسبت به نقطه اتصال ربات متحرک استفاده می‌کند. برای تأیید اینکه کارها را درست انجام داده‌ایم، مدل Models/robots/mobile/KUKA Omnirob.ttm را به داخل صحنه می‌کشیم. سپس مدل ربات خود را انتخاب می کنیم، سپس روی یکی از نقاط پیوست روی پلت فرم موبایل کنترل کلیک می کنیم، سپس روی دکمه Assembling/disassembling toolbar کلیک می کنیم. ربات ما باید به درستی خود را در بالای ربات متحرک قرار دهد:



[Attached robot]

اکنون می‌توانیم موارد دیگری مانند حسگرها را به ربات خود اضافه کنیم. در برخی مواقع ممکن است بخواهیم اسکریپت های تعبیه شده را به مدل خود وصل کنیم تا رفتار آن را کنترل کنیم یا آن را برای اهداف مختلف پیکربندی کنیم. در این صورت، مطمئن شوید که نحوه دسترسی به دسته‌های شیء از اسکریپت‌های تعبیه‌شده را می‌دانید. ما همچنین می توانیم مدل خود را از طریق یک پلاگین، از یک کلاینت API راه دور، از یک گره ROS یا از یک افزونه کنترل/دسترسی/رابط کنیم.

اکنون مطمئن می شویم که تغییرات انجام شده در حین اتصال ربات و گریپر را برگردانده ایم، درخت سلسله مراتب مدل ربات خود را جمع می کنیم، پایه مدل خود را انتخاب می کنیم، سپس آن را با [-- Menu bar> -- Save model as <File ذخیره می کنیم. ...]. اگر آن را در پوشه مدل ذخیره کنیم، مدل در مرورگر مدل موجود خواهد بود.

نوشتن کد در CoppeliaSim:

CoppeliaSim یک شبیه ساز بسیار قابل شخصی سازی است: هر جنبه ای از شبیه سازی را می توان سفارشی کرد. علاوه بر این، خود شبیه‌ساز را می‌توان سفارشی و تنظیم کرد تا دقیقاً مطابق دلخواه رفتار کند. این از طریق یک رابط برنامه نویسی کاربردی ((APIپیچیده مجاز است. بیش از 6 رویکرد برنامه نویسی یا کدنویسی مختلف پشتیبانی می شود که هر کدام دارای مزایا (و بدیهی است که معایب) خاصی نسبت به سایرین دارند، اما هر شش مورد با یکدیگر سازگار هستند (یعنی می توان همزمان یا حتی دست به دست هم از آنها استفاده کرد). موجودیت کنترل یک مدل، صحنه یا خود شبیه ساز می تواند در داخل قرار گیرد:

یک اسکریپت تعبیه شده (یعنی سفارشی کردن یک شبیه سازی (به عنوان مثال یک صحنه یا مدل ها) از طریق اسکریپت): این روش که شامل نوشتن اسکریپت های Lua یا Python است، بسیار آسان و انعطاف پذیر است، با سازگاری تضمین شده با هر نصب پیش فرض) CoppeliaSimتا زمانی که توابع API سفارشی استفاده نمی شوند، یا با پلاگین های توزیع شده استفاده می شوند). این روش امکان شخصی سازی یک شبیه سازی خاص، یک صحنه شبیه سازی و تا حدودی خود شبیه ساز را فراهم می کند. این ساده ترین و پرکاربردترین روش برنامه نویسی است.

یک افزونه یا اسکریپت :sandboxاین روش، که شامل نوشتن اسکریپت های Lua یا Python است، امکان شخصی سازی سریع خود شبیه ساز را فراهم می کند. افزونه‌ها (یا اسکریپت جعبه شنی) می‌توانند به‌طور خودکار شروع شوند و در پس‌زمینه اجرا شوند، یا می‌توان آن‌ها را به‌عنوان توابع فراخوانی کرد (مثلاً هنگام نوشتن واردکننده/صادرکننده راحت باشد). افزونه‌ها نباید مختص یک شبیه‌سازی یا مدل خاص باشند، بلکه باید عملکردی عمومی‌تر و محدود به شبیه‌ساز ارائه دهند.

یک پلاگین (یعنی سفارشی کردن شبیه ساز و/یا شبیه سازی از طریق یک افزونه): این روش اساسا شامل نوشتن یک افزونه برای CoppeliaSim است. اغلب، پلاگین ها فقط برای ارائه یک شبیه سازی با دستورات API سفارشی شده استفاده می شوند، و بنابراین در ارتباط با روش اول استفاده می شوند. مواقع دیگر، از پلاگین‌ها برای ارائه عملکرد ویژه CoppeliaSim استفاده می‌شود که به قابلیت محاسبه سریع (اسکریپت‌ها در اکثر مواقع کندتر از زبان‌های کامپایل‌شده)، یک رابط خاص برای یک دستگاه سخت‌افزاری (مثلاً یک ربات واقعی)، یا یک رابط ارتباطی خاص نیاز دارد. با دنیای بیرون

یک کلاینت API از راه دور (یعنی سفارشی کردن شبیه ساز و/یا شبیه سازی از طریق یک برنامه کلاینت API راه دور): این روش به یک برنامه خارجی (مثلاً واقع در یک ربات، ماشین دیگر و غیره) اجازه می دهد تا به روشی بسیار آسان به CoppeliaSim متصل شود. ، با استفاده از دستورات API از راه دور.

یک گره (ROSیعنی سفارشی کردن شبیه‌ساز و/یا شبیه‌سازی از طریق یک گره) :(ROSاین روش به یک برنامه خارجی (مثلاً روی یک ربات، ماشین دیگر و غیره) اجازه می‌دهد تا از طریق ROS، سیستم عامل ربات، به CoppeliaSim متصل شود.

یک گره در حال صحبت با TCP/IP، ZeroMQ، و غیره: این روش به یک برنامه خارجی (به عنوان مثال واقع در یک ربات، ماشین دیگر و غیره) اجازه می دهد تا از طریق وسایل ارتباطی مختلف به CoppeliaSim متصل شود.

6 روش بالا نیز در آموزش کنترلر خارجی مورد بحث قرار گرفته است. در جدول زیر به تفصیل مزایا و معایب هر روش توضیح داده شده است:

	Embedded script	Add-on / sandbox script	Plugin	Remote API client	ROS / ROS2 node	ZeroMQ node
Control entity is external (i.e. can be located on a robot, different machine, etc.)	No	No	No	Yes	Yes	Yes
Difficulty to implement	Easiest	Easiest	Relatively easy	Easy	Relatively easy	Easy
Supported programming language	Lua, Python	Lua, Python	C/C++	C/C++, Python, Java, JavaScript, Matlab, Octave	Any [1]	Any
Code execution speed	Relativ. fast [2]	Relativ. fast [2]	Fast	Depends on programming language	Depends on programming language	Depends on programming language
Communication lag	None [3]	None [3]	None	Yes	Yes	Yes
Control entity can be fully contained in a scene or model, and is highly portable	Yes	No	No	No	No	No
Control entity relies on	CoppeliaSim	CoppeliaSim	CoppeliaSim	Sockets, ZeroMQ or WebSockets	ROS / ROS2 framework	ZeroMQ
Stepped operation [4]	Yes, inherent	Yes, inherent	Yes, inherent	Yes	Yes	Yes
Non-stepped operation [4]	Yes, via threads	Yes, via threads	No (threads available, but API access forbidden)	Yes	Yes	Yes

[1] Depends on ROS / ROS2 bindings
[2] Depends on the programming language, but the execution of API functions is very fast
[3] Lua scripts are executed in CoppeliaSim's main thread, Python scripts are executed in separate processes
[4] *Stepped* as in *synchronized* with each simulation step

در واقع به شکل کلی ویژگی API در سه نسخه ارائه می شود:

- ZeroMQ-Based: نسخه ای سبک و ساده در استفاده و همراه با تمامی توابع API و پشتیبانی از اسکریپت های پایتون.
- Legacy: نسخه ی ساده، نسبتا سبک و بدون وابستگی و پشتیبانی از زبان های C/C++, Java, Python, Matlab, Octave و Lua
- B0-Based: این نسخه بر پایه ی BlueZero Middleware بنا شده و مانند نسخه ی Legacy قابلیت پشتیبانی از زبان های بسیاری را دارد.

	Easy to use	Directly available functions	Languages
ZeroMQ-based remote API	++	All	Python
Legacy remote API	+	Subset	C/C++, Python, Java, Matlab, Octave, Lua
B0-based remote API	+	Subset	C/C++, Python, Java, Matlab, Lua

اتصال به زبان برنامه نویسی پایتون:

همانطور که گفتیم اتصال به زبان های برنامه نویسی با استفاده از 3 نسخه فوق از REMOTE API برقرار میشود.
در این قسمت بعنوان مثال از زبان پایتون برای دریافت داده های یک Vision Sensor استفاده میکنیم و تصویر مشاهده شده توسط سنسور را دریافت میکنیم.

اما پیش از آن باید محیط توسعه را آماده کنیم و برای اینکار نیاز به نصب بودن پایتون و کتابخانه هایی مانند Numpy, Scipy و Matplotlib و IDE دلخواه میباشیم.

- ابتدا به پوشه ی محل نصب COPPELIASIM بروید.
- به مسیر programming/remoteApiBindings/python/python بروید.
- تمامی فایل ها با پسوند Py. را به دایرکتوری پروژه ی خود کپی کنید.
- فایل remoteApi را برحسب نوع سیستم عامل خود به داخل پروژه کپی کنید.برای مثال در ویندوز فایل remoteApi.dll

در ادامه مثال هایی از اتصال کد به شبیه ساز را مشاهده میکنید که راهکار بالا برای مثال شماره 2 و اتصال Legacy ارائه شده است.

مثال اول با استفاده از ZeroMQ-Based remote API کلاینت میباشد.

```
from time import sleep
from zmqRemoteApi import RemoteAPIClient
client = RemoteAPIClient('localhost',23000)
sim = client.getobject('sim')
sensor1Handle=sim.getObjectHandle('/VisionSensor')
sensor2Handle=sim.getObjectHandle('/PassiveVisionSensor')

sim.startSimulation()
while True:
    image,resX,resY=sim.getVisionSensorCharImage(sensor1Handle)
    sim.setVisionSensorCharImage(sensor2Handle,image)
    sleep(0.01)
sim.stopSimulation()
```

مثال دوم نیز مشابه با مثال اول میباشد با این تفاوت که برای انجام آن از Legacy remote API استفاده شده است.

```
import sim
from time import sleep
clientID=sim.simxStart('127.0.0.1',19997,True,True,5000,5)

if clientID!=-1:

res,sensor1Handle=sim.simxGetObjectHandle(clientID,'VisionSensor1',sim.simx_o
pmode_oneshot_wait)

res,sensor2Handle=sim.simxGetObjectHandle(clientID,'VisionSensor2',sim.simx_o
pmode_oneshot_wait)


res,resolution,image=sim.simxGetVisionSensorImage(clientID,sensor1Handle,0,si
m.simx_opmode_streaming)
    sim.simxStartSimulation(clientID,sim.simx_opmode_oneshot)
    while (sim.simxGetConnectionId(clientID)!=-1):

res,resolution,image=sim.simxGetVisionSensorImage(clientID,sensor1Handle,0,si
m.simx_opmode_buffer)
        if res==sim.simx_return_ok:

res=sim.simxSetVisionSensorImage(clientID,sensor2Handle,image,0,sim.simx_opmo
de_oneshot)
        sleep(0.01)
    sim.simxFinish(clientID)
```

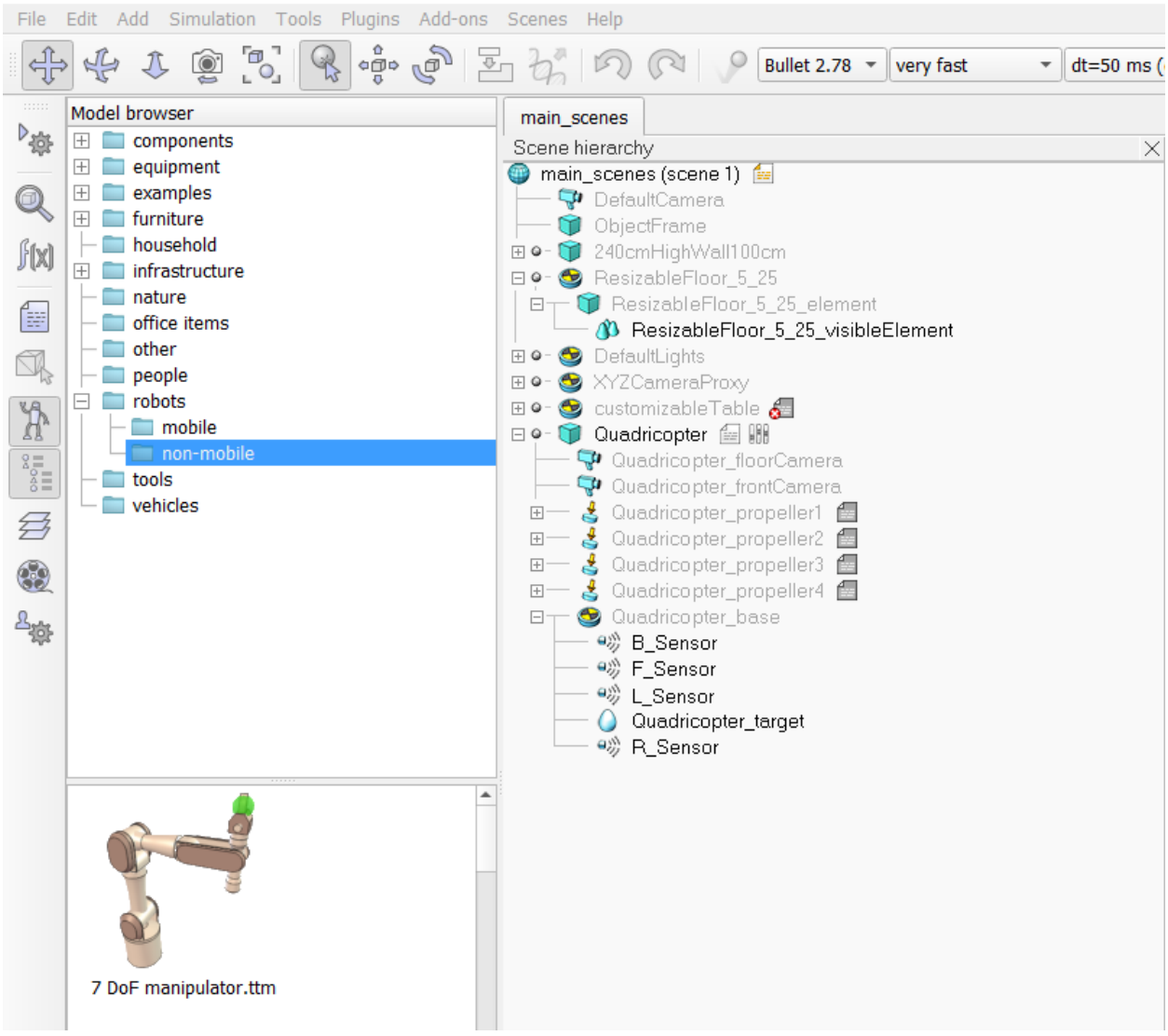
در مثال سوم نیز از B0-Based remote API استفاده شده است.

```
import b0RemoteApi
from time import sleep

with b0RemoteApi.RemoteApiClient('b0RemoteApi_pythonClient', 'b0RemoteApi')
as client:
    def imageCallback(msg):
        client.simxSetVisionSensorImage(sensor2Handle[1], False, msg[2],
client.simxDefaultPublisher())


    sensor1Handle = client.simxGetObjectHandle('VisionSensor1',
client.simxServiceCall())
    sensor2Handle = client.simxGetObjectHandle('VisionSensor2',
client.simxServiceCall())
    client.simxGetVisionSensorImage(sensor1Handle[1], False,
client.simxDefaultSubscriber(imageCallback))
    client.simxStartSimulation(client.simxDefaultPublisher())
    while True:
        client.simxSpinOnce()
        sleep(0.01)
    client.simxStopSimulation(client.simxDefaultPublisher())
```

با توجه به سه مثال مشاهده شده، واضح و مبرهن میباشد که استفاده از ZeroMQ-based به شدت ساده تر بوده و میزان خط های نوشته شده کمتر و ورودی توابع دارای پارامتر های کمتر و توابع به صورت کامل با توجه به ابجکت های مختلف نام های متفاوت به خود گرفته اند و دیگر خبری از فانکشن های جامع در آن نیست و این مورد برای توسعه دهنده های تازه کار به شدت کار را ساده کرده است.

ایجاد پروژه:

در ابتدا با استفاده از شبیه ساز VRep و منوی فایل یک سکانس جدید ایجاد میکنیم و با استفاده از پنل model browser موارد زیر را به سکانس خود اضافه میکنیم.

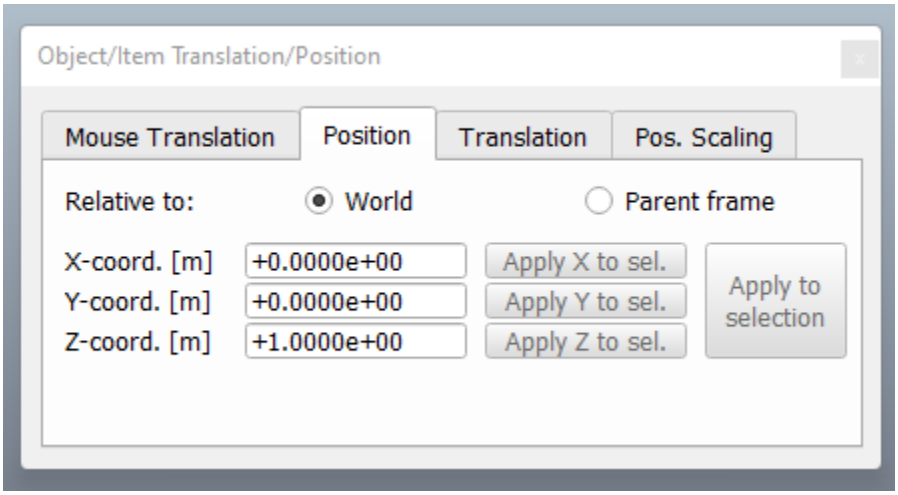


1- Resizable floor
2- Customizable table
3- Quadricopter
4- Cone type sensor

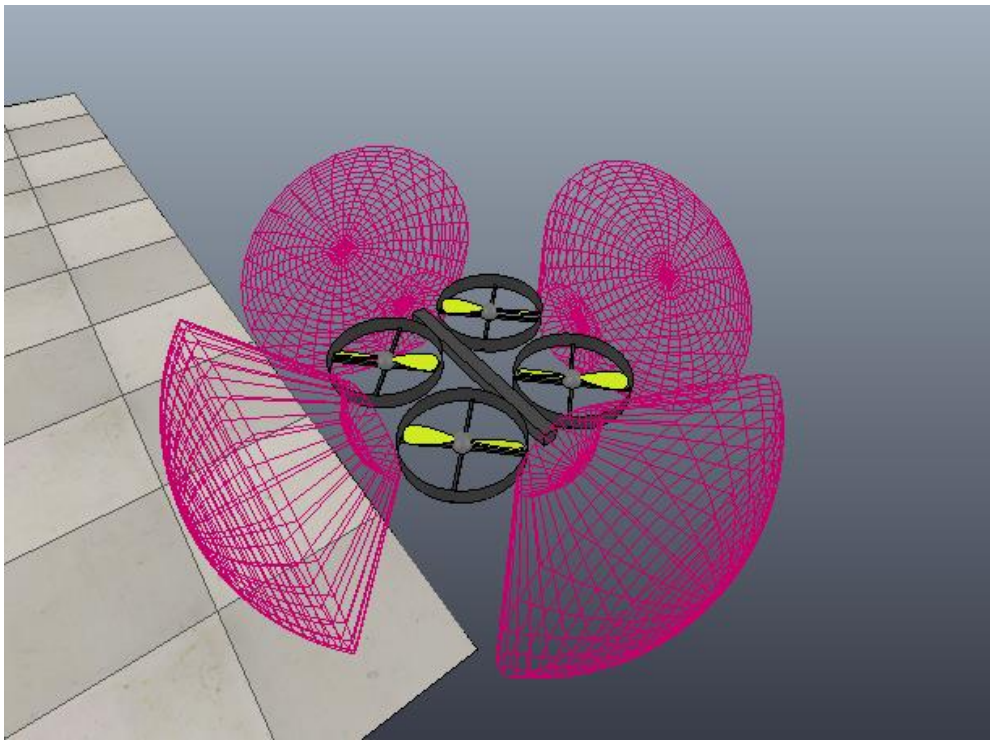
در موارد بالا quadcopter پرنده ما و المان floor سطح مورد ازمایش ما میباشد و با استفاده از چهار عدد proximity sensor از نوع Cone type میتوانیم نزدیک شدن پرنده به اشیاء مجاور و مقابل را متوجه شده و با استفاده از این داده ها تصمیم درست را در راستای راهبری صحیح این پرنده تا رسیدن به مقصد مورد نظر بگیریم.

موارد 1 تا 3 را با استفاده از Drag&Drop از پنل ModelBrowser به داخل سکانس منتقل میکنیم و با استفاده از پنل Scene Hierarchy آن ها را مرتب کرده با دبل کلیک کردن روی ایکون هر المان امکان ایجاد ویژگی های منحصر به فرد برای آن المان فراهم میشود.

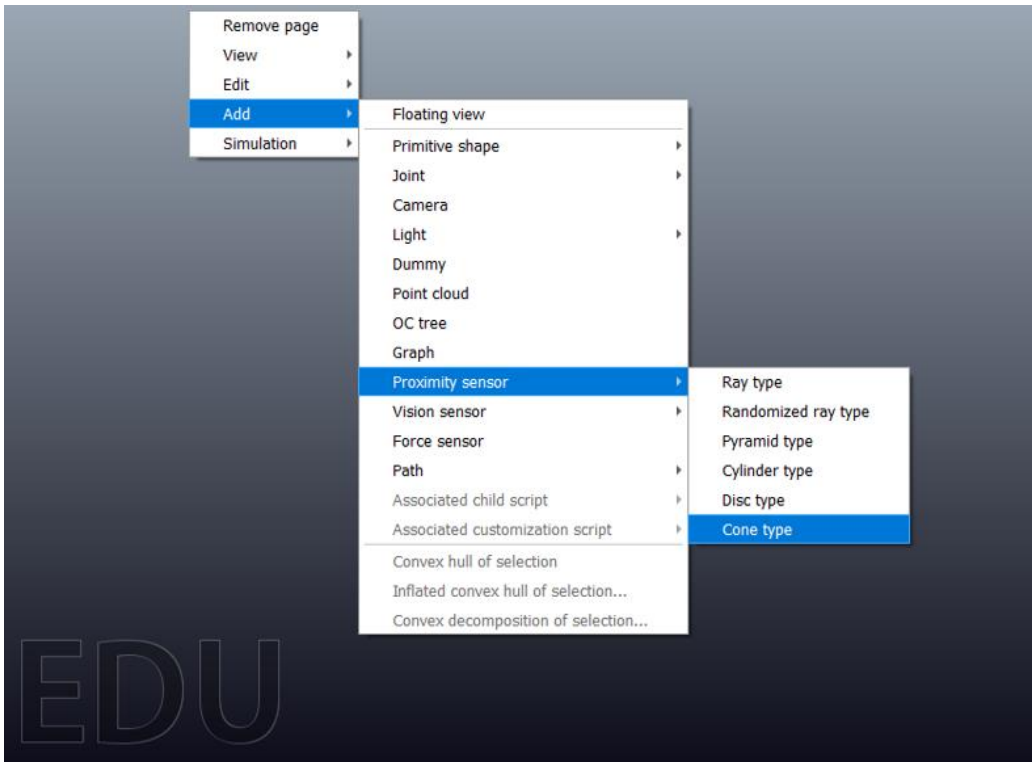
همچنین برای جابجای المان ها و تنضیم دقیقتر آن ها میتوان از منوی Object/Item Translation/Position به طور دقیق تر موقعیت هر المان را نسبت به مبدا سکانس یا مقادیر نسبی نسبت به المان والد تنظیم نمود.



سنسور های Proximity باید در هر 4 طرف پرنده به شکل زیر نصب گردند:

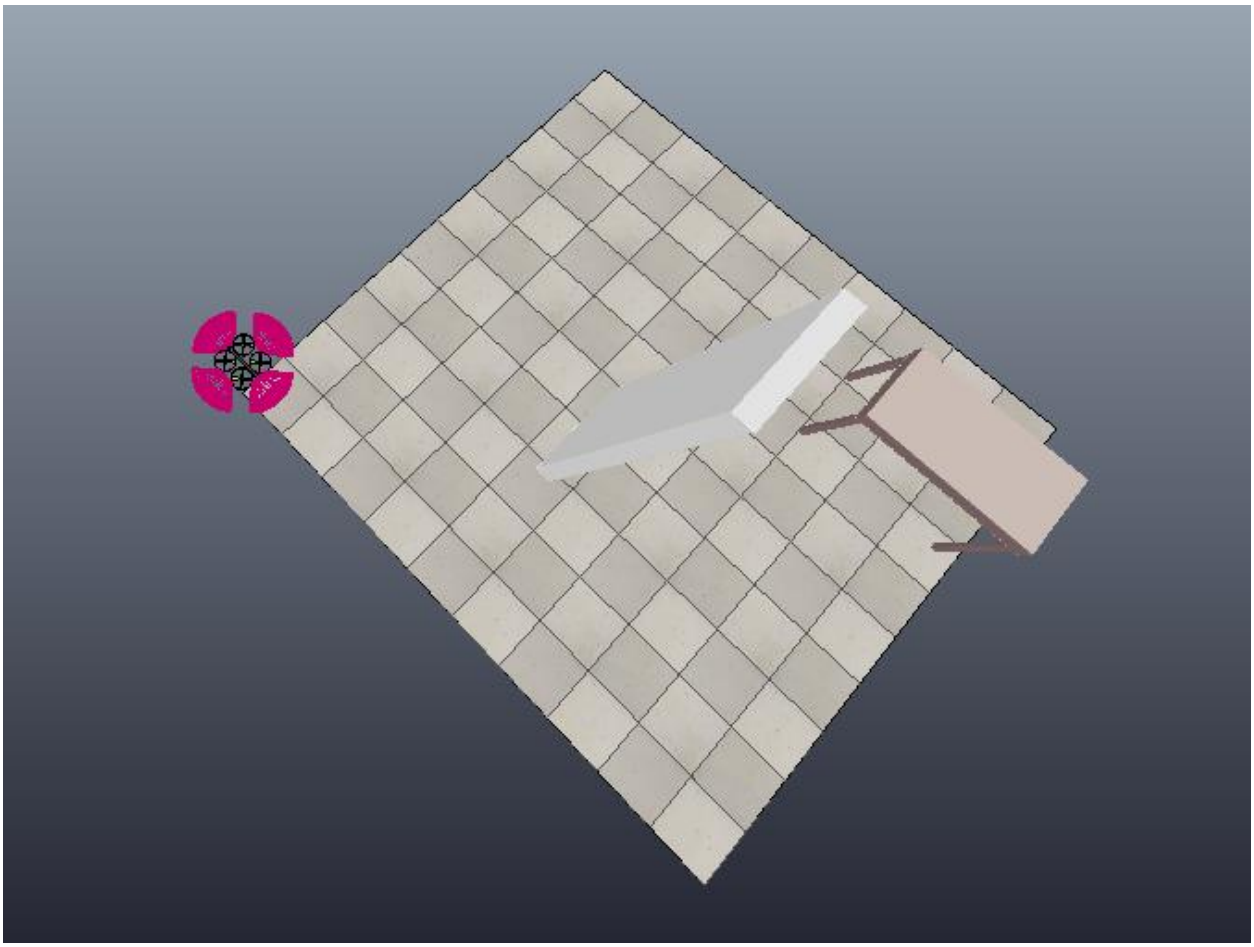


این سنسور ها بر خلاف موارد 1 تا 3 از منوی ModelBrowser در دسترس نیستن و با کلیک راست کردن داخل سکانس و منوهای زیر قابل دسترسی میشوند:



لازم به ذکر است در تمامی المان های بالا نام هر المان و ویژگی ها از جمله رنگ، زوایا، جهت قرارگیری، ابعاد و موقعیت قابل تغییر میباشد و همچنین نام هر المان ویزگی unique هر المان میباشد که قابلیت remoteApi با استفاده از همین نام قابلیت اتصال و کنترل آن المان را بدست می آورد.

سکانس طراحی شده در نهایت باید شبیه به ساختار زیر باشد:



پس از ایجاد این سکانس، نیاز است پروژه ای با زبان پایتون را ایجاد کنیم و سپس کتابخانه های مورد نیاز و همچنین فایل های کتابخانه ای vrep که به منظور اتصال پایتون به شبیه ساز طراحی شده اند را به پروژه اضافه و در فایل main.py خود ایمپورت کنیم.

با استفاده از تکه کد زیر

```
client_id = vrep.simxStart(self._hostname, self._port, True, True, 5000, 5)
```

میتوانیم کلاینت ای جهت اتصال به remoteApi بسازیم.

لازم به ذکر است مقادیر hostname و port اشاره به localhost و 19999 به عنوان مقادیر پیش فرض در یک سیستم local با فرض نصب بودن شبیه ساز و اجرا کننده کد پایتون در یک ماشین دارد و در غیر اینصورت نیاز به اصلاح این پارامتر ها به مقادیر صحیح وجود دارد.

خوانش موقعیت ها و مقادیر سنسور ها و به صورت کلی دسترسی به هر المانی در remoteApi با استفاده از نام آن المان و صدا زدن فانکشن زیر:

```
vrep.simxGetObjectHandle(self._client_id, 'name',
vrep.simx_opmode_oneshot_wait)
```

و دریافت handle آن المان و سپس فراخوانی فانکشن های مرتبط صورت میگیرد.

بعنوان مثال پس از فراخوانی فانکشن GetObjectHandle و دریافت Handle مربوطه میتوانیم با استفاده از کد زیر :

```
vrep.simxGetObjectPosition(self._client_id, self._robot_handle, -1,
vrep.simx_opmode_buffer)
```

موقعیت المان مورد نظر، در کد بالا ربات، را بدست آوریم.

لازم به ذکر است موقعیت به دست آمده عددی اعشاری است و نسبت به 0و0 سکانس محاسبه میشود

شبیه ساز vrep توابع بسیاری برای خوانش دیتای المان های مختلف دارد، فانکشن بعدی که در راه اندازی این سیستم تاثیر به سزایی داشته است را در زیر مشاهده میکنید.

```
error_code, detection_state, detected_point, detected_object_handle,
detected_surface_normal_vector = vrep.simxReadProximitySensor(self.client_id,
self.sensor_handles[i], vrep.simx_opmode_streaming)
```

فانکشن ReadProximitySensor دیتای سنسور های پرنده را برای ما داخل محیط پایتون فراهم می آورد و میتوانیم بر اساس آن ها تصمیم گیری انجام دهیم. و همانطور که مشاهده میکنید، در اینجا نیز ما از هندل المان

سنسور به جای نام آن استفاده کردیم زیرا ابتدا با استفاده از فانکشن GetObjectHandle نام آن المان را نبدیل به هندل کرده بودیم.

در ادامه پس از خوانش موقعیت و اطلاعات سنسور و تصمیم گیری راجب سوی حرکت و مقدار حرکت نیاز است این اطلاعات را به پرنده خود انتقال دهیم که این فرایند با استفاده از فانکشن زیر میسر است:

```
def simxSetObjectPosition(clientID, objectHandle, relativeToObjectHandle,
position, operationMode):
```

این تابع با استفاده از دریافت هندل المان بعنوان ورودی و موقعیت مقصد، پرنده را به موقعیت تنظیم شده هدایت میکند.

اخرین تابع از سری توابع مهم remoteApi در پیاده سازی این پرژه، فانکشن SetObjectOrientation که وظیفه تنظیم جهت قرار گیری پرنده را برعهده دارد:

```
def simxSetObjectOrientation(clientID, objectHandle, relativeToObjectHandle,
eulerAngles, operationMode):
```

این فانکشن نیز با استفاده از هندل ربات اقدام به چرخش جهت ربات با استفاده از eulerAngles میکند.

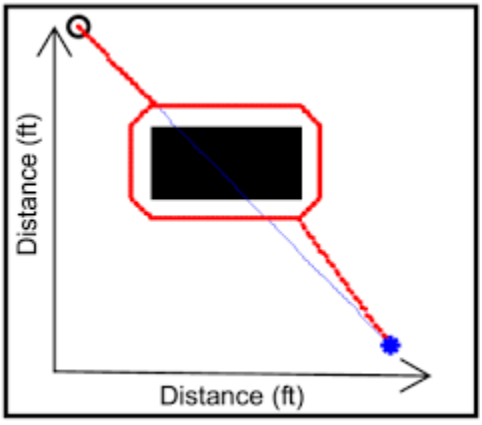
فانکشن های فوق اصلی ترین فانکشن های مورد نیار از remoteApi میباشد که وظایف زیر را برای ما به انجام میرسانند:

1- تبدیل نام به هندل
2- دریافت موقعیت
3- دریافت orientation
4- دریافت مقادیر Proximity Sensors
5- تنظیم موقعیت برای جابجای
6- تنظیم جهت و سوی پرنده

در ادامه نیاز داریم تا الگوریتم های باگ را به منظور جهت یابی به کار ببریم:

الگوریتم جهت یابی باگ 1

به تصویر زیر توجه کنید:



الگوریتم باگ یک با در نظر داشتن موقعیت تارگت شروع به حرکت میکند و پس از برخورد به مانع اقدام به دور زدن مانع تا رسیدن به موقعیت قبلی و ترسیم یک نقشه و سپس محاسبه بهترین مسیر و شروع به حرکت طبق آن میکند.

قطعه کد زیر مربوط به عملکرد حرکت پرنده در قالب باگ 1 میباشد:

```
def action_moving(self):

    if self.detect[4] < 0.6:

        self.state = States.ROTATING
        tmp = Quaternion()
        tmp.set_from_vector(self.PI / 2.0, Vector3(0.0, 0.0, 1.0))
        self.target_dir = tmp.rotate(self.bot_dir)

        return

    angle = Utils.angle_between_vectors(self.bot_dir,
self.target_pos.minus(self.bot_pos))

    if math.fabs(angle) > 1.0 * self.PI / 180.0:
        vrep.simxSetJointTargetVelocity(self.client_id,
self.left_propellers_handle,  self.PROPELLERS_SPEED + angle,
vrep.simx_opmode_streaming)
        vrep.simxSetJointTargetVelocity(self.client_id,
self.right_propellers_handle, self.PROPELLERS_SPEED - angle,
vrep.simx_opmode_streaming)
    else:
        vrep.simxSetJointTargetVelocity(self.client_id,
self.left_propellers_handle,  self.PROPELLERS_SPEED,
vrep.simx_opmode_streaming)
        vrep.simxSetJointTargetVelocity(self.client_id,
```

```
self.right_propellers_handle, self.PROPELLERS_SPEED,
vrep.simx_opmode_streaming)
```

توابع زیر نیز به ترتیب عملیات چرخش و دور زدن را انجام میدهند:

```
def action_rotating(self):

    angle = Utils.angle_between_vectors(self.bot_dir, self.target_dir)

    if math.fabs(angle) > 5.0 * self.PI / 180.0:
        vrep.simxSetJointTargetVelocity(self.client_id,
self.left_propellers_handle,   angle, vrep.simx_opmode_streaming)
        vrep.simxSetJointTargetVelocity(self.client_id,
self.right_propellers_handle, -angle, vrep.simx_opmode_streaming)
    else:
        self.state = States.ROUNDING

        if self.bot_rounding_state is None:
            self.bot_rounding_state = RobotRoundingState.START
            self.rounding_start_pos = self.bot_pos

def action_rounding(self):

    if self.bot_rounding_state is RobotRoundingState.START:
        if self.is_cur_pos_near_start_rounding_pos(0.5) is False:
            self.bot_rounding_state = RobotRoundingState.PROCESS

    elif self.bot_rounding_state is RobotRoundingState.PROCESS:
        if self.min_dist_to_target >
Utils.distance_between_points(self.bot_pos, self.target_pos):
            self.min_dist_to_target =
Utils.distance_between_points(self.bot_pos, self.target_pos)

        if self.is_cur_pos_near_start_rounding_pos(0.2) is True:
            self.bot_rounding_state = RobotRoundingState.END

    elif self.bot_rounding_state is RobotRoundingState.END:

        dist_to_target = Utils.distance_between_points(self.bot_pos,
self.target_pos)

        if dist_to_target * 0.95 < self.min_dist_to_target < dist_to_target *
1.05:
            self.state = States.MOVING
            self.bot_rounding_state = None
            self.rounding_start_pos = None
            return

    delta = self.detect[7] - self.detect[8]

    if delta < 0.0:
        obstacle_dist = self.detect[7] - self.INDENT_DIST
    else:
```

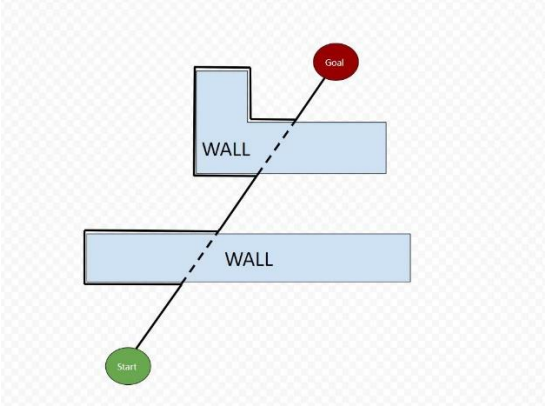
```
        obstacle_dist = self.detect[8] - self.INDENT_DIST

    u_obstacle_dist_stab = self.obstacle_dist_stab_PID.output(obstacle_dist)
    u_obstacle_follower = self.obstacle_follower_PID.output(delta)

    vrep.simxSetJointTargetVelocity(self.client_id,
self.left_propellers_handle,  self.PROPELLERS_SPEED + u_obstacle_follower +
u_obstacle_dist_stab - (1 - self.detect[4]), vrep.simx_opmode_streaming)
    vrep.simxSetJointTargetVelocity(self.client_id,
self.right_propellers_handle, self.PROPELLERS_SPEED - u_obstacle_follower -
u_obstacle_dist_stab + (1 - self.detect[4]), vrep.simx_opmode_streaming)
```

با داشتن این توابع تنها نیاز است حلقه ای بی نهایت با شرط پایان رسیدن به مقصد را ایجاد کنیم و داخل آن فانکشن های حرکت، چرخش و دور زدن موانع را اجرا کنیم:

```
def loop(self):

    self._init_values()

    while True:

        self.tick()

        self.stop_move()
        self.read_values()

        # self.calc_lenght_of_robot_track()

        self.read_from_sensors()

        self.target_pos.z = self.bot_pos.z = 0.0

        q_rot = Quaternion()
        q_rot.set_from_vector(self.bot_euler_angles.z, Vector3(0.0, 0.0,
1.0))
        self.bot_dir = q_rot.rotate(Vector3(1.0, 0.0, 0.0))

        if self.state == States.MOVING:
            self.action_moving()
        elif self.state == States.ROTATING:
            self.action_rotating()
        elif self.state == States.ROUNDING:
            self.action_rounding()
```

الگوریتم باگ 2

به تصویر زیر توجه کنید:



الگوریتم باگ 2 پس از هر برخورد به مانع با در نظر گرفتن تارگت، سعی میکند تا در نزدیک ترین مسیر به تارگت حرکت کند و دیگر نیازی به تهیه نقشه وجود ندارد.

قطعه کد زیر مربوط به عملکرد حرکت پرنده در قالب باگ 2 میباشد:

```
def action_moving(self):

    if self.detect[4] < 0.6:

        self.state = States.ROTATING
        tmp = Quaternion()
        tmp.set_from_vector(self.PI / 2.0, Vector3(0.0, 0.0, 1.0))
        self.target_dir = tmp.rotate(self.bot_dir)

        return

    angle = Utils.angle_between_vectors(self.bot_dir,
self.target_pos.minus(self.bot_pos))

    if math.fabs(angle) > 1.0 * self.PI / 180.0:
        vrep.simxSetJointTargetVelocity(self.client_id,
self.left_propellers_handle,  self.PROPELLERS_SPEED + angle,
vrep.simx_opmode_streaming)
        vrep.simxSetJointTargetVelocity(self.client_id,
self.right_propellers_handle, self.PROPELLERS_SPEED - angle,
vrep.simx_opmode_streaming)
    else:
        vrep.simxSetJointTargetVelocity(self.client_id,
self.left_propellers_handle,  self.PROPELLERS_SPEED,
vrep.simx_opmode_streaming)
        vrep.simxSetJointTargetVelocity(self.client_id,
self.right_propellers_handle, self.PROPELLERS_SPEED,
vrep.simx_opmode_streaming)
```

توابع زیر نیز به ترتیب عملیات چرخش و دور زدن را انجام میدهند:

```
def action_rotating(self):

    angle = Utils.angle_between_vectors(self.bot_dir, self.target_dir)

    if math.fabs(angle) > 5.0 * self.PI / 180.0:
        vrep.simxSetJointTargetVelocity(self.client_id,
self.left_propellers_handle,   angle, vrep.simx_opmode_streaming)
        vrep.simxSetJointTargetVelocity(self.client_id,
self.right_propellers_handle, -angle, vrep.simx_opmode_streaming)
    else:
        self.state = States.ROUNDING

def action_rounding(self):

    if self.is_bot_on_the_constant_direction():
        self.state = States.MOVING
        return

    delta = self.detect[7] - self.detect[8]

    if delta < 0.0:
        obstacle_dist = self.detect[7] - self.INDENT_DIST
    else:
        obstacle_dist = self.detect[8] - self.INDENT_DIST

    u_obstacle_dist_stab = self.obstacle_dist_stab_PID.output(obstacle_dist)
    u_obstacle_follower = self.obstacle_follower_PID.output(delta)

    vrep.simxSetJointTargetVelocity(self.client_id,
self.left_propellers_handle,  self.PROPELLERS_SPEED + u_obstacle_follower +
u_obstacle_dist_stab - (1 - self.detect[4]), vrep.simx_opmode_streaming)
    vrep.simxSetJointTargetVelocity(self.client_id,
self.right_propellers_handle, self.PROPELLERS_SPEED - u_obstacle_follower -
u_obstacle_dist_stab + (1 - self.detect[4]), vrep.simx_opmode_streaming)
```

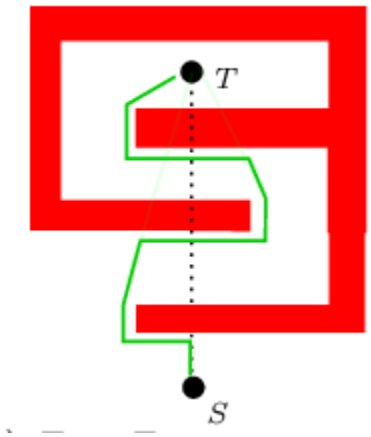
با داشتن این توابع تنها نیاز است حلقه ای بی نهایت با شرط پایان رسیدن به مقصد را ایجاد کنیم و داخل آن فانکشن های حرکت، چرخش و دور زدن موانع را اجرا کنیم:

```
def loop(self):

    self._init_values()

    while True:
```

```
        self.tick()

        self.stop_move()
        self.read_values()

        # self.calc_lenght_of_robot_track()

        if self.start_bot_pos is None:
            self.start_bot_pos = self.bot_pos
        if self.start_target_pos is None:
            self.start_target_pos = self.target_pos

        self.read_from_sensors()

        self.target_pos.z = self.bot_pos.z = 0.0

        q_rot = Quaternion()
        q_rot.set_from_vector(self.bot_euler_angles.z, Vector3(0.0, 0.0,
1.0))
        self.bot_dir = q_rot.rotate(Vector3(1.0, 0.0, 0.0))

        if self.state == States.MOVING:
            self.action_moving()
        elif self.state == States.ROTATING:
            self.action_rotating()
        elif self.state == States.ROUNDING:
            self.action_rounding()
```

الگوریتم باگ 3

به تصویر زیر توجه کنید:



الگوریتم باگ3 پس از شروع به حرکت با هر برخورد به مانع چرخشی در جهت نزدیک شدن به تارگت انجام میدهد و همواره با در نظر گرفتن فاصله ی میان نقطه ی برخورد و تارگت تصمیم به جرخش به جهت راست یا چپ میکند و بدین تربیت بهینه ترین الگوریتم بین سه الگوریتم مورد بحث میباشد.

قطعه کد زیر مربوط به عملکرد حرکت پرنده در قالب باگ 3 میباشد:

```
def action_moving(self):

    if self.detect[4] < 0.6:

        self.state = States.ROTATING
        tmp = Quaternion()
        tmp.set_from_vector(self.PI / 2.0, Vector3(0.0, 0.0, 1.0))
        self.target_dir = tmp.rotate(self.bot_dir)

        return

    angle = Utils.angle_between_vectors(self.bot_dir,
self.target_pos.minus(self.bot_pos))

    if math.fabs(angle) > 1.0 * self.PI / 180.0:
        vrep.simxSetJointTargetVelocity(self.client_id,
self.left_propellers_handle,  self.PROPELLERS_SPEED + angle,
vrep.simx_opmode_streaming)
        vrep.simxSetJointTargetVelocity(self.client_id,
self.right_propellers_handle, self.PROPELLERS_SPEED - angle,
vrep.simx_opmode_streaming)
    else:
        vrep.simxSetJointTargetVelocity(self.client_id,
self.left_propellers_handle,  self.PROPELLERS_SPEED,
vrep.simx_opmode_streaming)
        vrep.simxSetJointTargetVelocity(self.client_id,
self.right_propellers_handle, self.PROPELLERS_SPEED,
vrep.simx_opmode_streaming)
```

توابع زیر نیز به ترتیب عملیات چرخش و دور زدن را انجام میدهند:

```
def action_rotating(self):

    angle = Utils.angle_between_vectors(self.bot_dir, self.target_dir)

    if math.fabs(angle) > 5.0 * self.PI / 180.0:
        vrep.simxSetJointTargetVelocity(self.client_id,
self.left_propellers_handle,   angle, vrep.simx_opmode_streaming)
        vrep.simxSetJointTargetVelocity(self.client_id,
self.right_propellers_handle, -angle, vrep.simx_opmode_streaming)
    else:
        self.state = States.ROUNDING
```

```
def action_rounding(self):

    tmp_dir = Quaternion()
    tmp_dir.set_from_vector(self.PI / 2.0, Vector3(0.0, 0.0, 1.0))
    perp_bot_dir = tmp_dir.rotate(self.bot_dir)

    angle = Utils.angle_between_vectors(perp_bot_dir,
self.target_pos.minus(self.bot_pos))

    if self.min_dist_to_target is None or self.min_dist_to_target <=
Utils.distance_between_points(self.bot_pos, self.target_pos):
        self.min_dist_to_target = Utils.distance_between_points(self.bot_pos,
self.target_pos)
    elif math.fabs(angle) < 5.0 / 180.0 * self.PI:
        self.state = States.MOVING
        return

    delta = self.detect[7] - self.detect[8]

    if delta < 0.0:
        obstacle_dist = self.detect[7] - self.INDENT_DIST
    else:
        obstacle_dist = self.detect[8] - self.INDENT_DIST

    u_obstacle_dist_stab = self.obstacle_dist_stab_PID.output(obstacle_dist)
    u_obstacle_follower = self.obstacle_follower_PID.output(delta)

    vrep.simxSetJointTargetVelocity(self.client_id,
self.left_propellers_handle,  self.PROPELLERS_SPEED + u_obstacle_follower +
u_obstacle_dist_stab - (1 - self.detect[4]), vrep.simx_opmode_streaming)
    vrep.simxSetJointTargetVelocity(self.client_id,
self.right_propellers_handle, self.PROPELLERS_SPEED - u_obstacle_follower -
u_obstacle_dist_stab + (1 - self.detect[4]), vrep.simx_opmode_streaming)
```

با داشتن این توابع تنها نیاز است حلقه ای بی نهایت با شرط پایان رسیدن به مقصد را ایجاد کنیم و داخل آن فانکشن های حرکت، چرخش و دور زدن موانع را اجرا کنیم:

```
def loop(self):

    self._init_values()

    while True:

        self.tick()

        self.stop_move()
        self.read_values()

        # self.calc_lenght_of_robot_track()

        self.read_from_sensors()
```

```
        self.target_pos.z = self.bot_pos.z = 0.0

        q_rot = Quaternion()
        q_rot.set_from_vector(self.bot_euler_angles.z, Vector3(0.0, 0.0,
1.0))
        self.bot_dir = q_rot.rotate(Vector3(1.0, 0.0, 0.0))

        if self.state == States.MOVING:
            self.action_moving()
        elif self.state == States.ROTATING:
            self.action_rotating()
        elif self.state == States.ROUNDING:
            self.action_rounding()
```

پایان

سورس کد برنامه :

https://github.com/arash-mehrzadi/Autopilot-drone

منابع:

https://www.coppeliarobotics.com/helpFiles/index.html [1]

K. M. Lynch and F. C. Park, Modern Robotics: Mechanics, Planning, and Control. [2] Cambridge University Press, 2017

Freese M., Singh S., Ozaki F., Matsuhira N. (2010) Virtual Robot Experimentation Platform [3] V-REP: A Versatile 3D Robot Simulator. In: Ando N., Balakirsky S., Hemker T., Reggiani M., von Stryk O. (eds) Simulation, Modeling, and Programming for Autonomous Robots. SIMPAR 2010. Lecture Notes in Computer Science, vol 6472. Springer, Berlin, Heidelberg.

S. Chakraborty and P. Aithal, "A Custom Robotic ARM in CoppeliaSim," Chakraborty, [4] CoppeliaSim. International Journal of Sudip, & Aithal, PS,(2021). A Custom Robotic ARM in Applied Engineering and Management Letters (IJAEML), vol. 5, no. 1, pp. 38-50, 2021

https://www.coppeliarobotics.com/helpFiles/en/remoteApiOverview.htm [5]

https://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsPython.htm [6]