参加者の皆様にお知らせ

- お手本リポジトリ、スライド資料ともに昨日から更新しましたので、お手元のものも更新してください。

```
cd ~catkin_ws/src/choreonoid_ros_mobile_robot_tutorial
git pull
```

- 以下からダウンロード

https://choreonoid.org/ja/workshop/summar-training-2023.html

2023年8月24～26日

Choreonoid研修資料

株式会社コレオノイド

本研修の内容

- ChoreonoidとROSを用いてロボットのシミュレーションができるようになる
  - ロボットシミュレーションの基本が分かる
  - ROSの基本が分かる

    ※ 本講義はROS1を対象とします

- 必要なスキル
  - Linuxコマンドラインの操作
  - C++言語のプログラミング
    - 十分なスキルが無くても実習自体は進められます。必要に応じて学習していただければより理解が深まります。

講師紹介

中岡 慎一郎

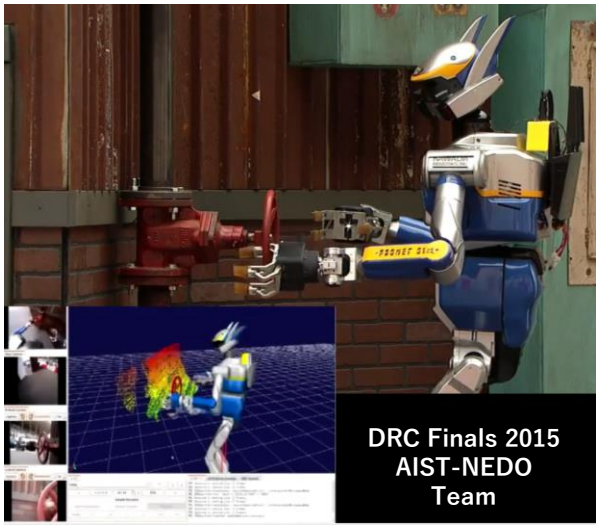
東京大学（2001年～）、産総研（2006～）にて18年間
ヒューマノイドロボットの研究に従事

2019年より株式会社コレオノイド代表取締役

研究内容：二足歩行ヒューマノイドロボットの全身動作の操り

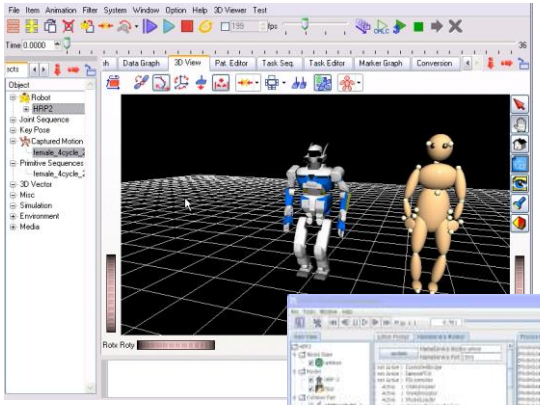
人全身動作模倣手法（2005）

全身動作振付手法（2010）



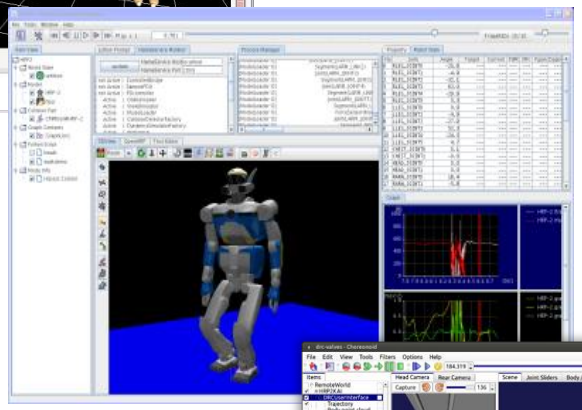
半自律遠隔操作タスク遂行手法（2015）

開発したロボットシミュレータ



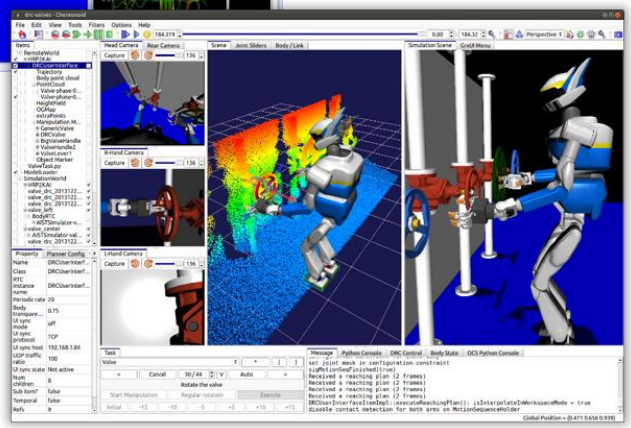
2003～（博士課程）

動作解析／生成用シミュレータ



2006～（産総研入所後）

分散コンポーネント型ロボットシミュレータ
OpenHRP3



2008～

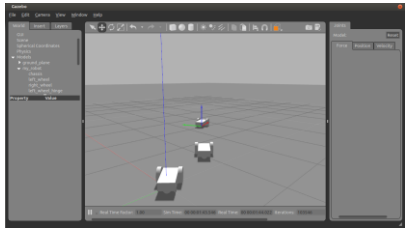
統合ロボットシミュレータ
Choreonoid

Part1

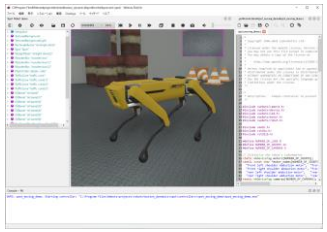
イントロダクション

ロボットシミュレータ色々

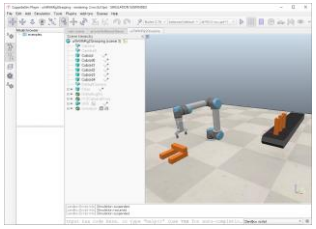
汎用ロボットシミュレータ



Gazebo

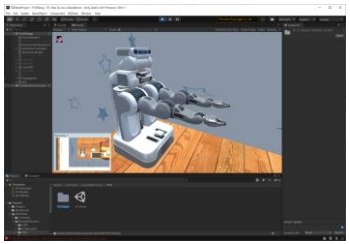


Webots

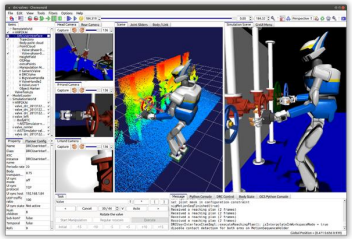


CoppeliaSim
(v-rep)

Issac Sim

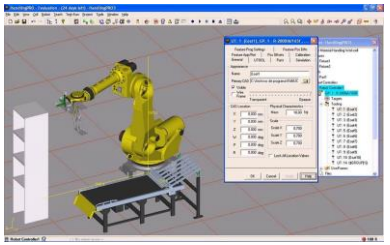


SigVerse



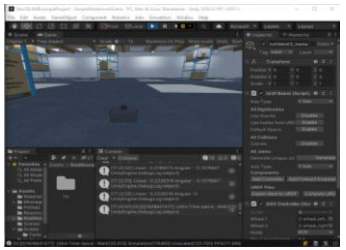
Choreonoid

産ロボ教示用シミュレータ



RoboGuide

ゲームエンジン



Unity

Choreonoid?

- 実は歴史と実績のある本格的ロボットシミュレータ

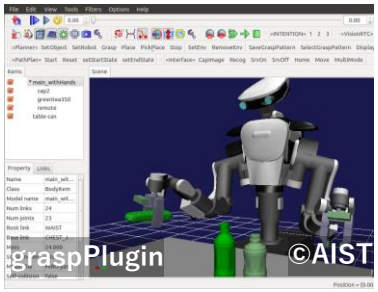
- 圧倒的な軽さ
- 「痒い所に手が届く」基本機能
  - タイムバー、個別リロード、etc.
- 尋常ではない柔軟性・拡張性

主な利用実績

2009年～

二足歩行ヒューマノイド
ロボットの研究開発

2009年～



マニピュレータの把持／
軌道／作業計画の生成

2010年～

ロボットダンス
パフォーマンスの実現

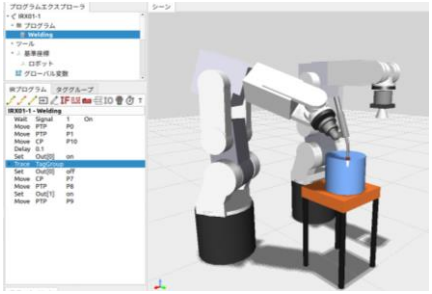
2014年～

遠隔作業ロボットの
半自律制御システム

2015年～

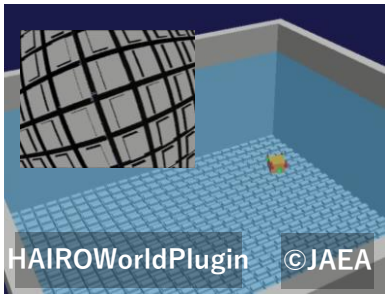
シミュレーションによる
災害対応ロボット競技会

2019年～



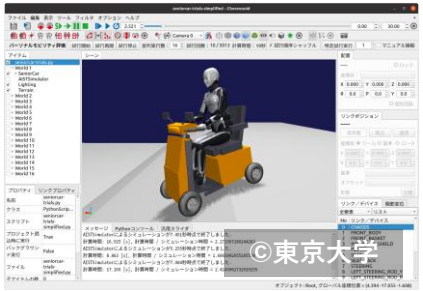
産業用ロボットの
動作教示

2020年～



廃炉のための
遠隔操作技術開発

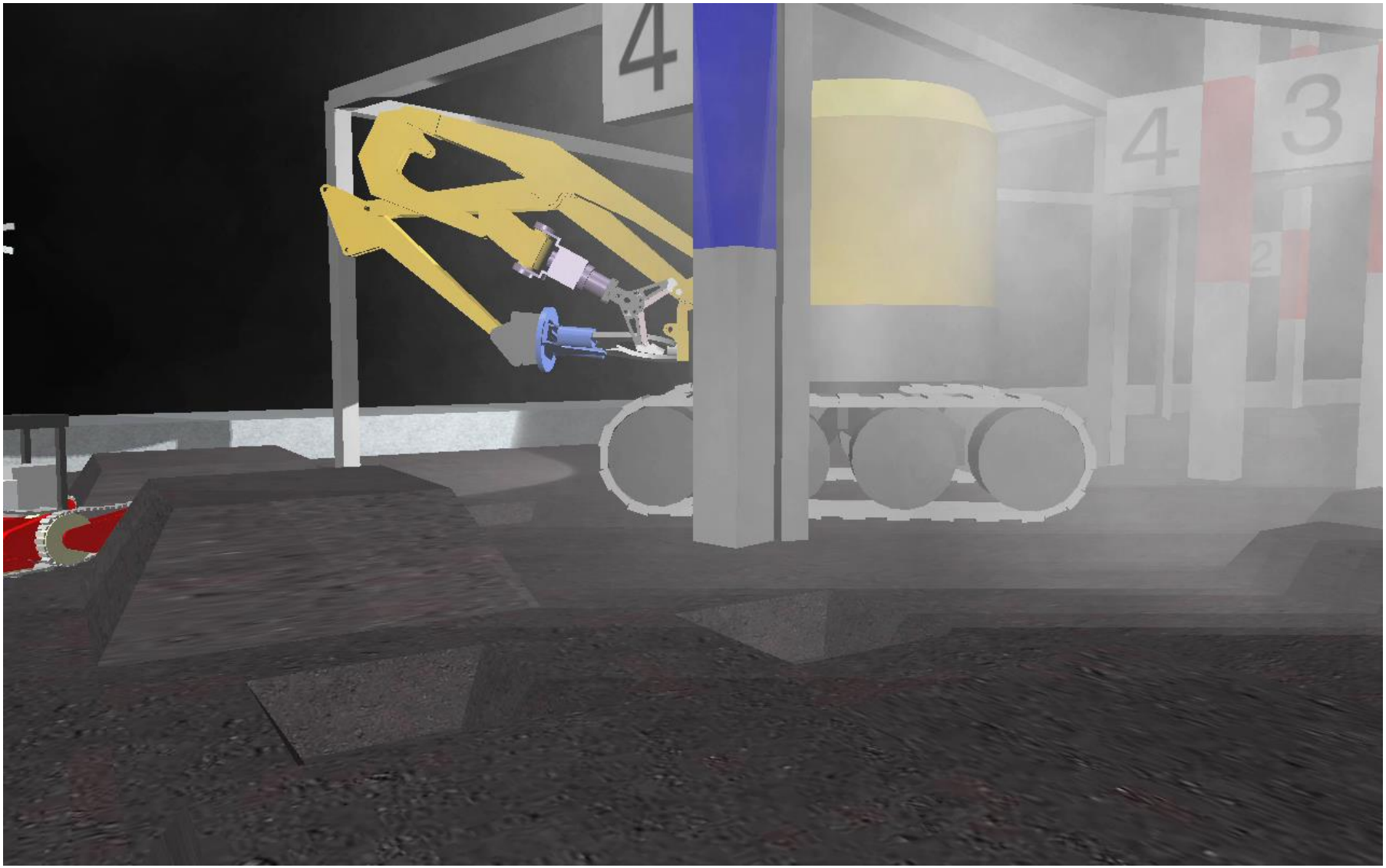
2021年～



福祉機器の
リスクアセスメント

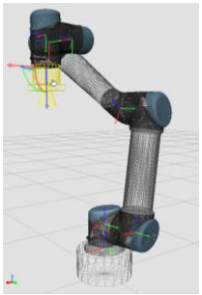
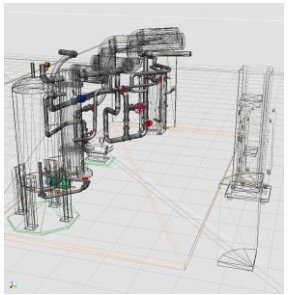
ロボットシミュレータとしての実用性は多くの実応用で実証済み！

World Robot Summit 災害対応競技



主な機能

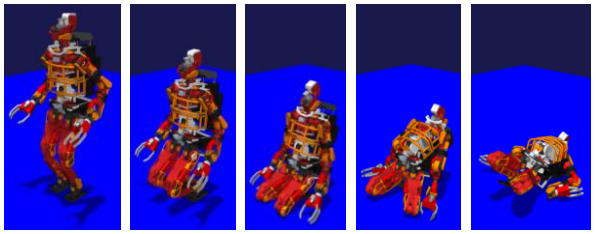
ロボット／環境モデルの読込／可視化／操作

・機構構造／状態の可視化

・配置／姿勢の変更(運動学／逆運動学)

運動学／動力学シミュレーション

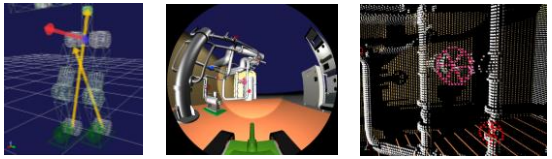


・多様な機構に対応
マニピュレータ、脚、車輪、クローラ、
マルチコプタ、etc.

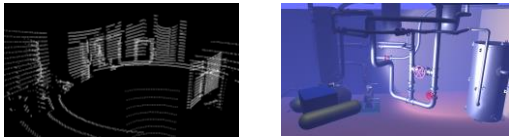
・用途に適した物理エンジンを選択可能

センサ／デバイスシミュレーション

力学センサ　カメラ　RGB-Dカメラ

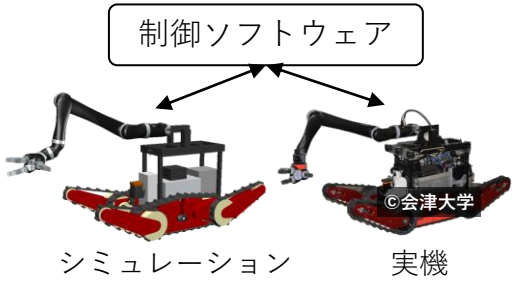


LiDAR　ライト



多様なセンサ／デバイスに対応

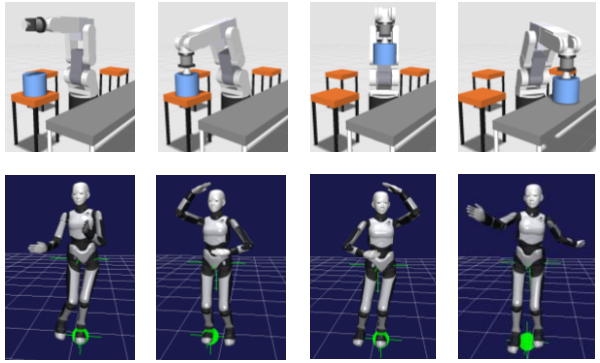
制御ソフトウェアの接続



・様々な規格(API/ミドルウェア)に対応

・実機と共通のソフトウェアを利用可能

動作教示／振付



ロボットの動作を作成・編集することが可能

スクリプティング

```
worldItem = WorldItem()
RootItem.instance.addChildItem(worldItem)

robotItem = BodyItem()
robotItem.load("${SHARE}/model/SR1/SR1.body")
robot = robotItem.body
robot.rootLink.setTranslation([0.0, 0.0, 0.7135])
robot.calcForwardKinematics()
robotItem.storeInitialState()

simulatorItem = AISTSimulatorItem()
```

各種操作／シミュレーションを自動化

仮想環境でロボットを扱う様々な機能を統合的に利用可能！

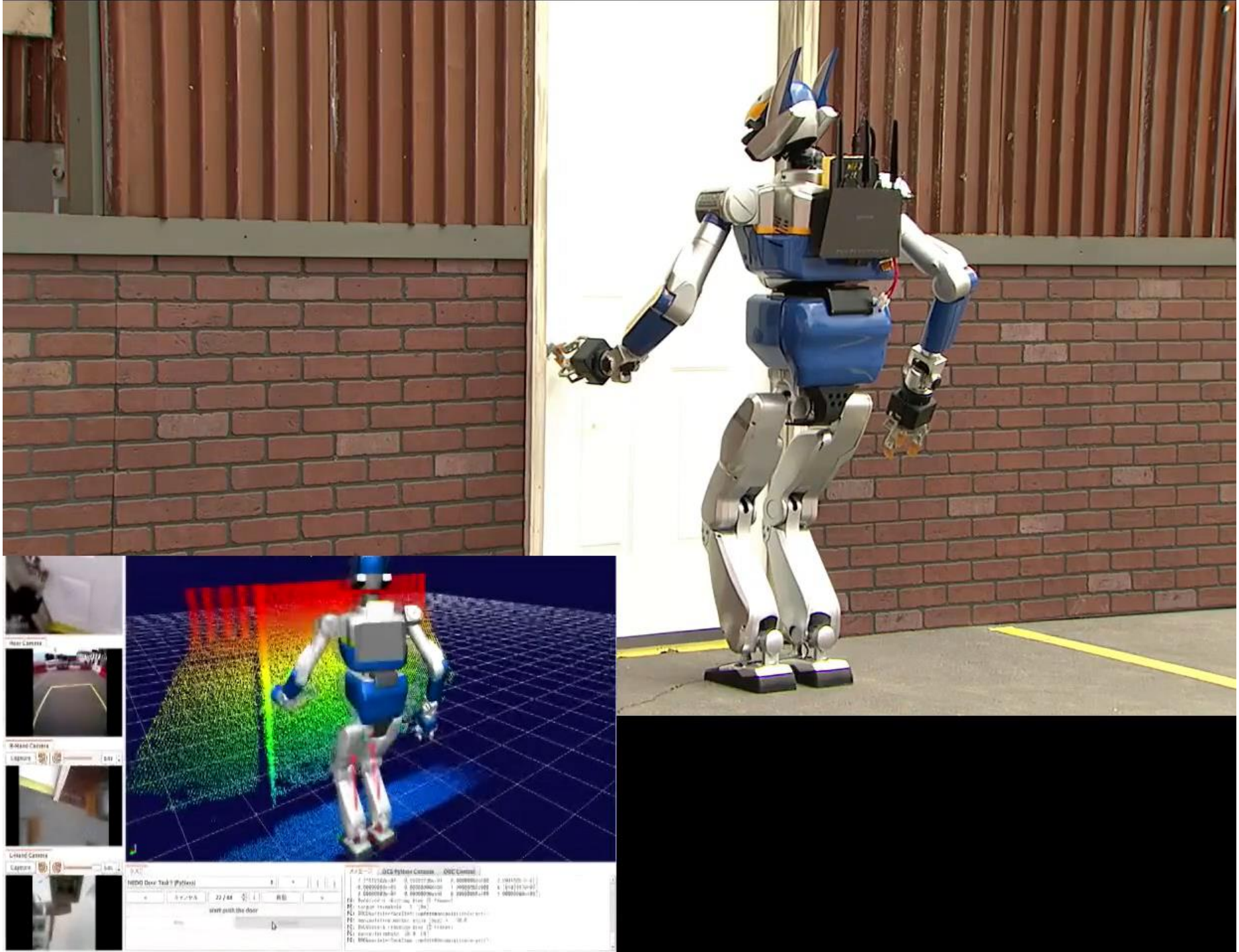
なぜシミュレータを使うのか

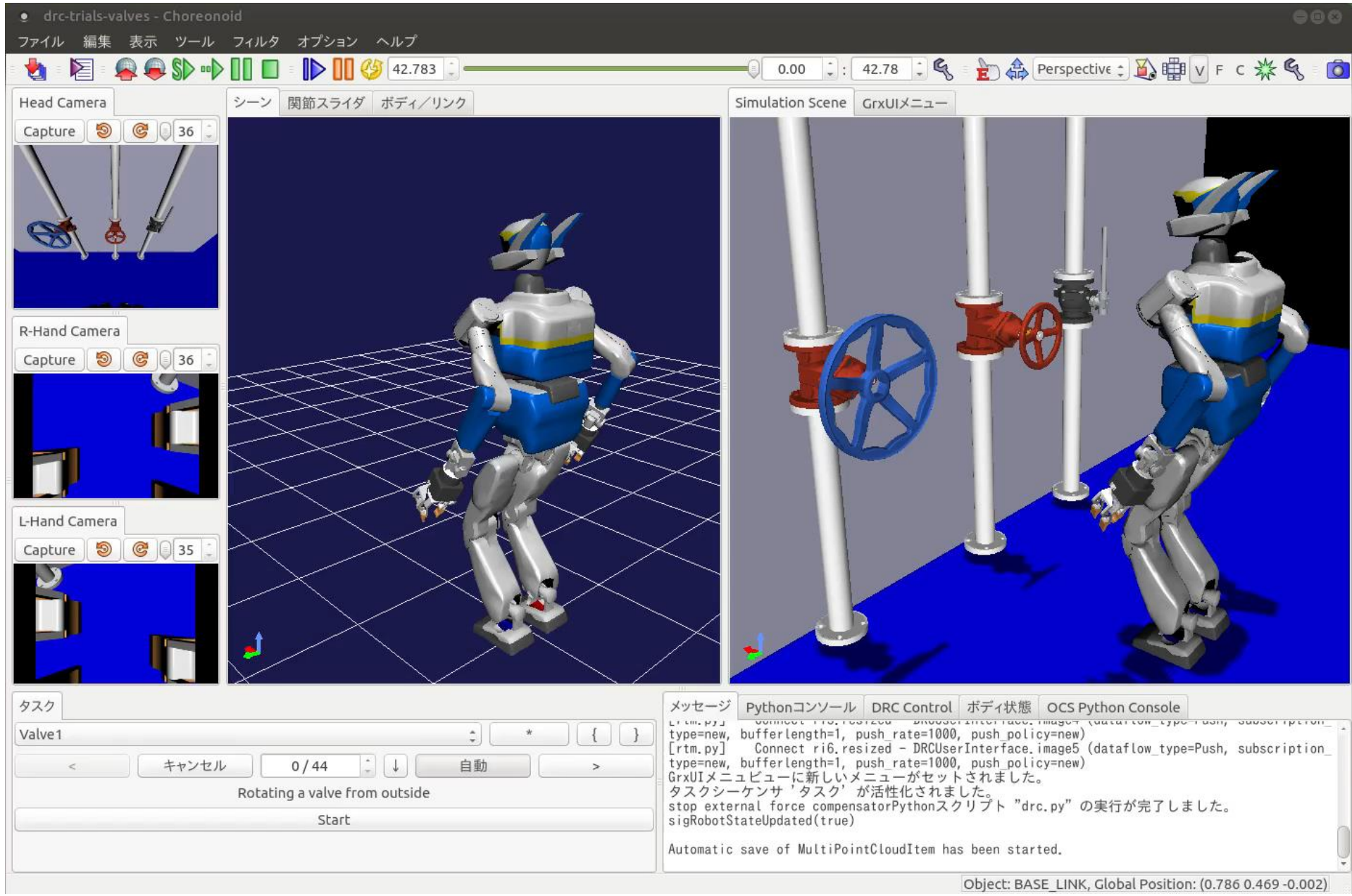
ロボットの研究開発・導入・運用に関わる

リスクの低減

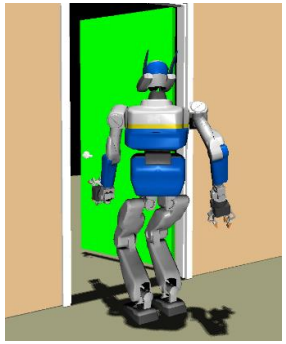
コストの削減

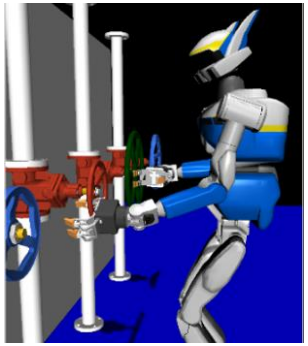
効率の向上

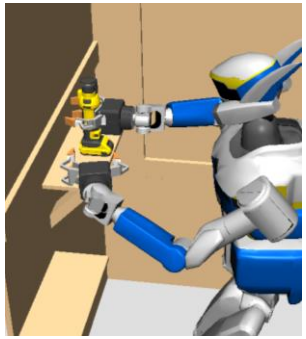

Head Camera
R-Hand Camera
Capture
L-Hand Camera
Capture
start push the door


drc-trials-valves - Choreonoid
ファイル 編集 表示 ツール フィルタ オプション ヘルプ
42.783
0.00
42.78
Perspective
Head Camera
Capture
36
R-Hand Camera
Capture
36
L-Hand Camera
Capture
35
シーン
関節スライダ
ボディ／リンク
Simulation Scene
GrxUIメニュー
タスク
Valve1
*
{
}
<
キャンセル
0 / 44
↓
自動
>
Rotating a valve from outside
Start
メッセージ
Pythonコンソール
DRC Control
ボディ状態
OCS Python Console
type=new, bufferlength=1, push_rate=1000, push_policy=new)
[rtm.py] Connect ri6.resized - DRCUserInterface.image5 (dataflow_type=Push, subscription_
type=new, bufferlength=1, push_rate=1000, push_policy=new)
GrxUIメニュビューに新しいメニューがセットされました。
タスクシーケンサ 'タスク' が活性化されました。
stop external force compensatorPythonスクリプト "drc.py" の実行が完了しました。
sigRobotStateUpdated(true)
Automatic save of MultiPointCloudItem has been started.
Object: BASE_LINK, Global Position: (0.786 0.469 -0.002)


NEDO
NEDO
NEDO







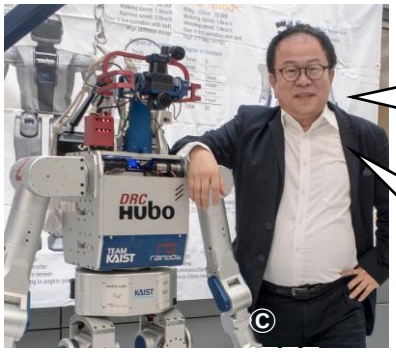
本物のロボットをもっと動かしてもらえませんか？
これじゃつまらなくて番組が作れません！

密着取材TV局ディレクター

この取り組みをもっと進めていきたい…

本当にシミュレーションが必要か？

DRC Finals 優勝
KAIST (DRC-Hubo) チーム
代表 Oh教授



シミュレータも用意したけどメンバーがあまり使いたがりませんでした

ロボット実機を直接使用してタスクの開発や練習を進めました

信頼性が高く壊れにくいロボットが対象であれば
シミュレータは必要ない！？

実機派の主張

- シミュレーション
  - モデル作成が面倒
  - シミュレーションが不完全
  - 結局検証やチューニングを実機で行うことになり二度手間
- 実機（実世界）
  - 手っ取り早い
  - 実世界こそが「完全な」物理シミュレーション
  - うちのロボットは壊れない or 壊さない

そうは言っても…

- 実機を使えない(製造/導入前)
- 実環境を使えない(危険/社会的制約)
- 大規模システム
- 繰り返し試行（機械学習）

課題

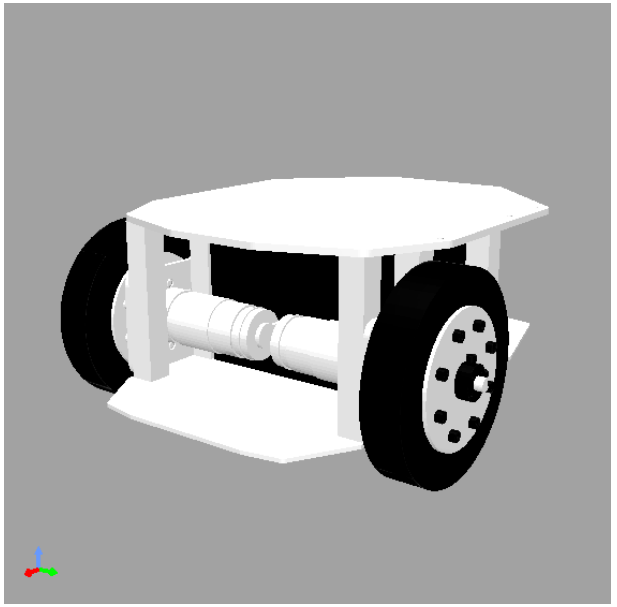
1. **シミュレーション性能の向上**
2. **モデル作成にかかる手間の削減**

Part2

実習の概要

実習内容

- モバイルロボットのシミュレーション
- 1日目
  - 環境構築
  - モデルの作成
  - 制御
- 2日目
  - 視覚センサの導入
  - 状態の可視化
  - 遠隔操作



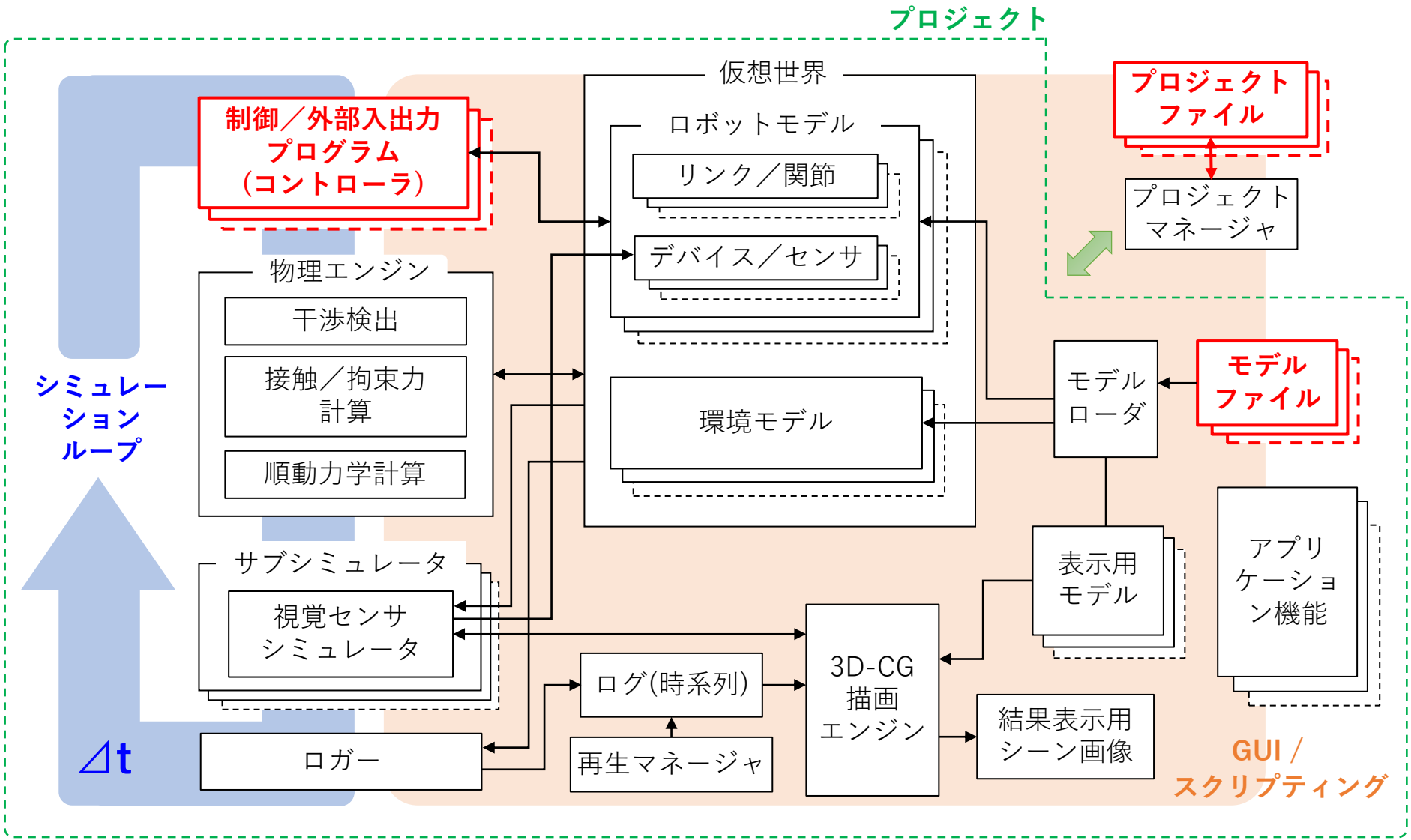
シミュレーションするにあたって

- 用意しなければならないもの
  - モデル（ロボット／環境）
  - 入出力／制御プログラム

- 用意するために必要な情報
  - モデルの作成方法（記述形式）
  - 入出力の仕様（API）

  ※ 基本的にはシミュレータごとに異なる

シミュレータの構成


プロジェクト
仮想世界
ロボットモデル
リンク／関節
デバイス／センサ
制御／外部入出力
プログラム
（コントローラ）
プロジェクト
ファイル
プロジェクト
マネージャ
物理エンジン
干渉検出
接触／拘束力
計算
順動力学計算
環境モデル
モデル
ローダ
モデル
ファイル
シミュレー
ション
ループ
サブシミュレータ
視覚センサ
シミュレータ
表示用
モデル
アプリ
ケーショ
ン機能
ログ(時系列)
3D-CG
描画
エンジン
結果表示用
シーン画像
ロガー
再生マネージャ
⊿t
GUI /
スクリプティング

利用するROS要素

- ビルド・実行環境
  - パッケージ
  - Catkin
  - launchファイル
- メッセージ通信
  - メッセージ型（入れ物）
  - トピック（用途）
  - ノード
    - Publish（送信）
    - Sbscribe（受信）
- ツール
  - シミュレータ、可視化、操作、編集、etc.

注：ROSの必要性

- 必ずしもROSを使う必要はありません
- 本実習は
  - ROSの基本
  - ChoreonoidとROSの連携

  の習得を目的としているため、あえてROSを使用しています
- 本実習の大部分は実はROSは無くてもChoreonoidだけで実現可能です

情報

- Choreonoid公式サイト
  - https://choreonoid.org/
- マニュアル
  - Choreonoid 最新版（開発版）マニュアル
    - https://choreonoid.org/ja/documents/latest/index.html
- ソースコード
  - https://github.com/choreonoid/choreonoid
  - https://github.com/choreonoid/choreonoid_ros
  - https://github.com/choreonoid/choreonoid_ros_mobile_robot_tutorial
- 掲示板
  - https://discourse.choreonoid.org/

関連チュートリアル

- Choreooidマニュアルの以下のページ
  - Tankチュートリアル
    - https://choreonoid.org/ja/documents/latest/simulation/tank-tutorial/index.html
  - ROS版Tankチュートリアル
    - https://choreonoid.org/ja/documents/latest/ros/tank-tutorial/index.html

本実習と同様の内容をカバーしていて、各項目について詳細な解説がありますので、参考にしてください。

Part3

環境構築

対象環境

- Ubuntu 20.04
- Choreonoid 2.0.0
- ROS1 (Noetic Ninjemys)
- GPUのセットアップ
  - 3D表示（OpenGL）のハードウェアアクセラレーションが効くように
  - nvidia製GPUの場合はプロプライエタリドライバをインストール
- ゲームパッド
  - PS4またはPS5のゲームパッドを推奨

推奨スペック（本実習対象）

- CPU
  - Intel第13世代（Raptor Lake）ならほぼ問題なし
  - Choreonoid本体をビルドする際はコア数が多いほどよい
- メモリ
  - 8GBあれば動くはず
  - 並列ビルドする場合は並列数に見合うメモリ容量
- GPU
  - NVIDIA製GPU推奨
- ディスプレイ解像度
  - フルHD以上

インストール（ビルド）

- Ubuntuで通常のビルドを行うのは比較的簡単

```
sudo apt install git
git clone https://github.com/choreonoid/choreonoid.git
cd choreonoid
./misc/script/install-requisites-ubuntu-20.04.sh
cmake -B build
cmake --build build --parallel 32
```

※ “—parallel”の後の数値は使用しているCPUの論理コア数を指定
※ make install しなくてもビルドディレクトリのバイナリをそのまま実行できます

ROS環境の構築

- ROSのインストール

- ChoreonoidのROSへのインストール
  - choreonoid (本体)
  - choreonoid_ros （ROSプラグイン）
  - Catkinワークスペース上にソースを展開してビルド

※ マニュアルの「ROSとの連携」-「Choreonoid関連パッケージのビルド」参照
https://choreonoid.org/ja/documents/latest/ros/build-choreonoid.html

ROSのインストール

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
sudo apt install curl
https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add -
sudo apt update
sudo apt install ros-noetic-desktop-full
echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc
source ~/.bashrc
sudo apt install python3-rosdep python3-rosinstall python3-rosinstall-generator python3-wstool build-essential python3-vcstool
sudo rosdep init
rosdep update
```

catkin workspaceの作成

```
mkdir catkin_ws
cd catkin_ws
mkdir src
catkin init
```

Choreonoidパッケージの追加とビルド

```
cd src
git clone https://github.com/choreonoid/choreonoid.git
git clone https://github.com/choreonoid/choreonoid_ros.git
git clone
 https://github.com/choreonoid/choreonoid_ros_mobile_robot_tutorial.git

./choreonoid/misc/script/install-requisites-ubuntu-20.04.sh

catkin config --cmake-args -DBUILD_CHOREONOID_EXECUTABLE=OFF
 -DCMAKE_BUILD_TYPE=Release

catkin build
```

ワークスペースセットアップスクリプトの取り込み

```
echo "source $HOME/catkin_ws/devel/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

※ 最初のビルド後に一度設定しておけばOK

Choreonoidノードの起動

- ROSノードとしてChoreonoidを起動

端末1: ROSマスターを起動

```
roscore
```

端末2: Choreonoid（ROSノード版）を起動

```
rosrun choreonoid_ros choreonoid
```

※ Ubuntuの標準端末であれば、Ctrl + Tで新しい端末のタブを追加できます

サンプルの実行

- メニューの「ファイル」-「プロジェクトを開く」を選択
- “choreonoid-2.0” – “project” のディレクトリから、サンプルプロジェクトを開ける
- シミュレーション開始ボタンを押す

実習用のROSパッケージ

- my_mobile_robot
  - 自分で作成
  - ここにファイルを作成していく
- choreonoid_ros_mobile_robot_tutorial
  - お手本となるパッケージ
  - 以下から取得
    - https://github.com/choreonoid/choreonoid_ros_mobile_robot_tutorial.git
  - 実習で作成するファイルが予め全て入っている
  - 必要に応じてここからmy_mobile_robotにファイルをコピーする

my_mobile_robotパッケージの作成

- パッケージ雛形の作成

```
cd ~/catkin_ws/src
catkin create pkg my_mobile_robot
```

- package.xmlの編集

<package format="2"> は？

```
…
<depend>choreonoid</depend>
<depend>choreonoid_ros</depend>
…
<export>
  <build_type>cmake</build_type>
</export>
…
```

※ デフォルトでは ”catkin”

Catkinのビルド設定

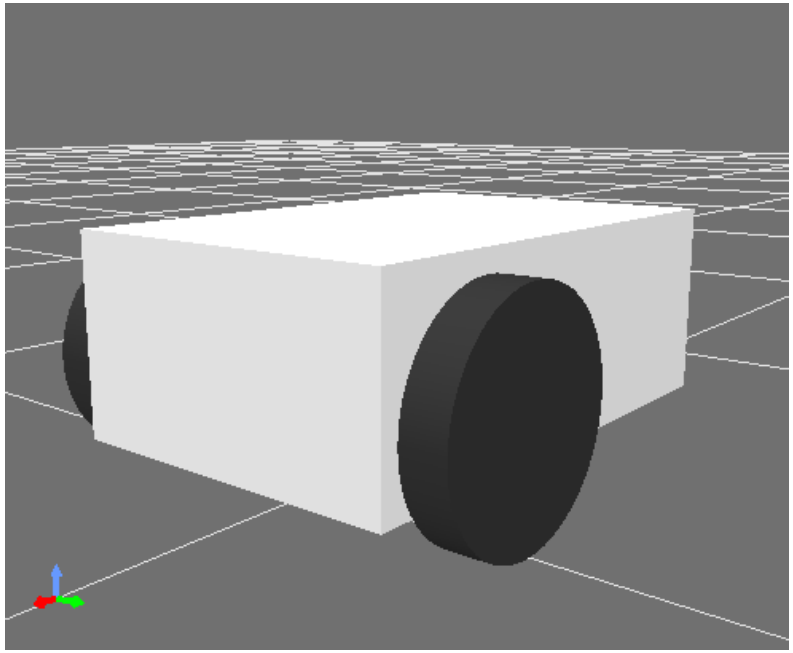
- チュートリアル用に、my_mobile_robotパッケージだけをビルドするように設定しておく

Part4

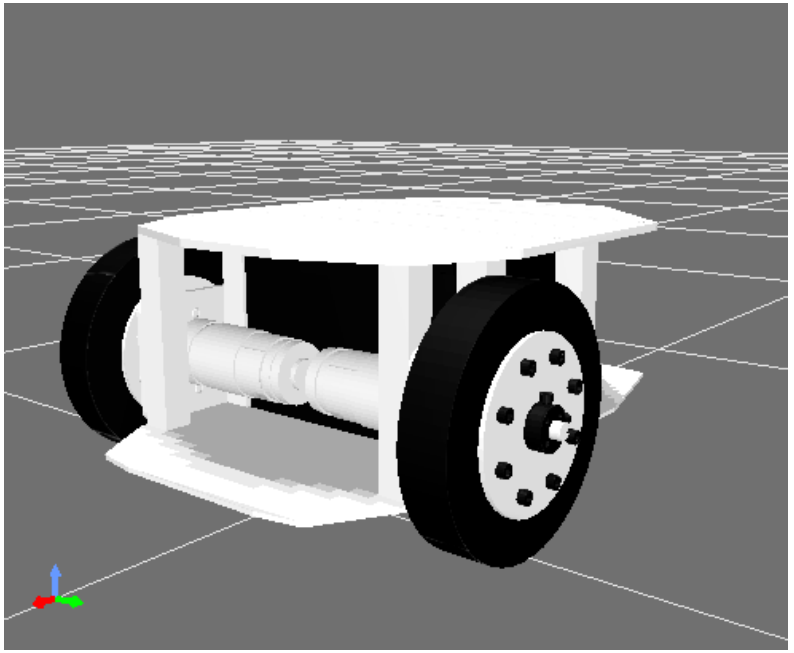
モデルの作成とシミュレーション

モデルの作成

- 車輪型モバイルロボットを作る



プリミティブ版



メッシュ版

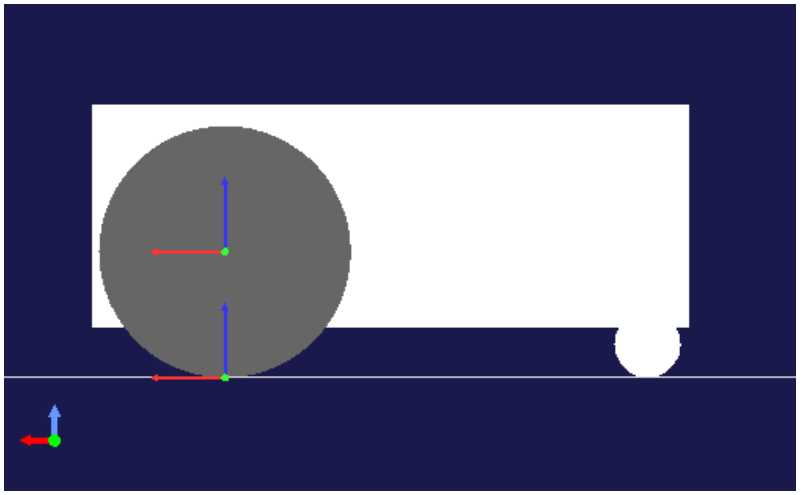
※ ヴィストン社 メガローバー2.1

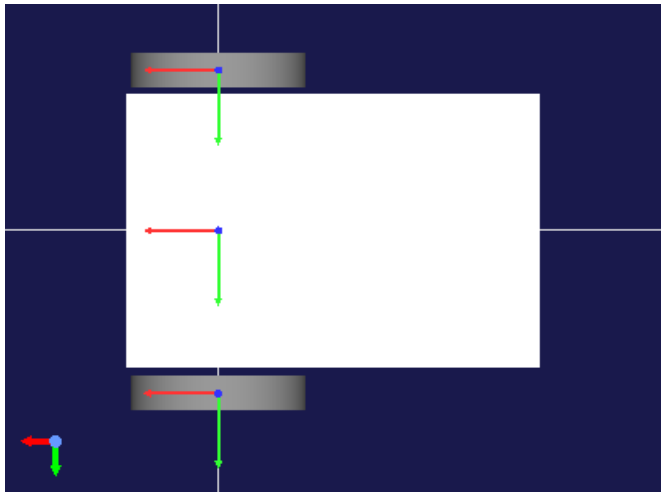
モデルファイル記述形式

- Body形式
  - Choreonoidネイティブ
  - YAMLで記述
- URDF
  - ROS標準
  - XMLで記述
- Xacro
  - XMLマクロ言語、ROS標準
  - URDFの記述を効率化

モデリング座標の方針

- 座標系　　**ロボットで一般的**
  - X: 前後方向、Y: 左右方向、Z: 上下方向
- 原点　　**どこでもよいが、分かりやく扱いやすいところ**
  - X: 車軸中央、Y: 中央、Z: 床面





モデルファイルの作成

- “my_mobile_robot” のディレクトリに “model” ディレクトリを作成

```
cd ~/catkin_ws/src/my_mobile_robot
mkdir model
```

※ ROSの慣習としては”urdf”や”robots”といったディレクトリ名が使われることが多い

- gedit等のテキストエディタを用いて ”mobile_robot.body” ファイルを作成する

```
gedit model/mobile_robot.body
```

車体の記述 (Body形式)

※ インデントに注意！

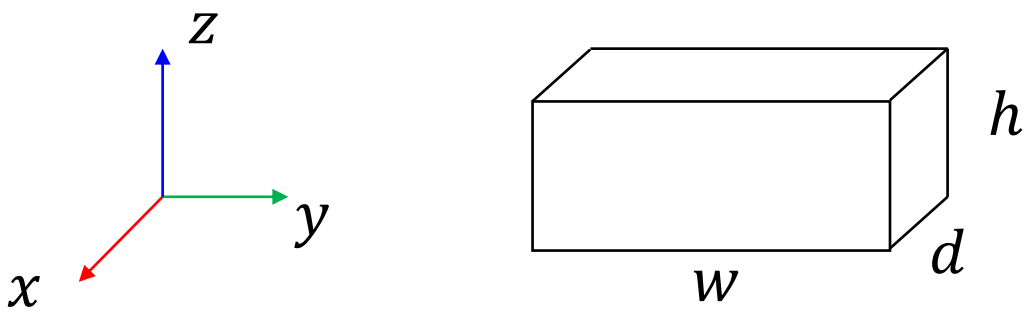
```
format: ChoreonoidBody
format_version: 2.0
angle_unit: degree
name: MobileRobot
root_link: Chassis

links:
  -
    name: Chassis
    joint_type: free
    center_of_mass: [ -0.08, 0, 0.08 ]
    mass: 14.0
    inertia: [ 0.1,   0,     0,
               0,     0.17,  0,
               0,     0,     0.22 ]
    material: Slider
    elements:
      -
        type: Shape
        translation: [ -0.1, 0, 0.0975 ]
        geometry:
          type: Box
          size: [ 0.36, 0.24, 0.135 ]
      -
        type: Shape
        translation: [ -0.255, 0, 0.02 ]
        geometry:
          type: Cylinder
          height: 0.01
          radius: 0.02
```

慣性行列の計算

- CADを利用
- プリミティブ形状に関する式から算出

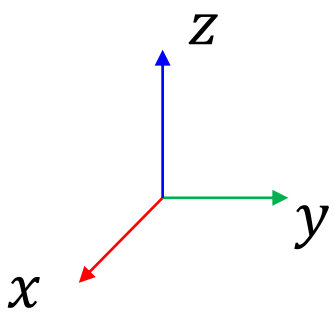
※ 自動計算機能は未実装

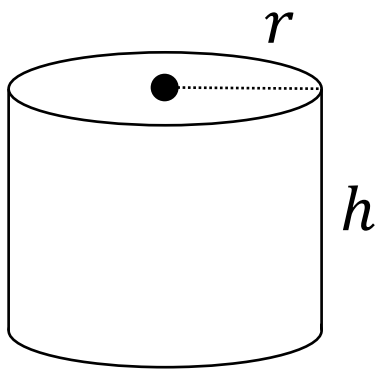
直方体(Box)の慣性行列



$$I = \begin{pmatrix} \frac{1}{12}m(w^2+h^2) & 0 & 0 \\ 0 & \frac{1}{12}m(d^2+h^2) & 0 \\ 0 & 0 & \frac{1}{12}m(w^2+d^2) \end{pmatrix}$$

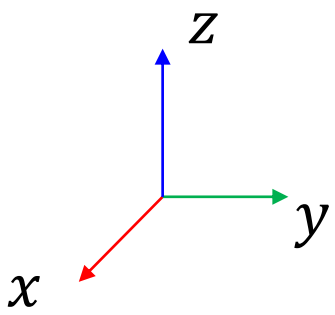
円柱(Cylinder)の慣性行列

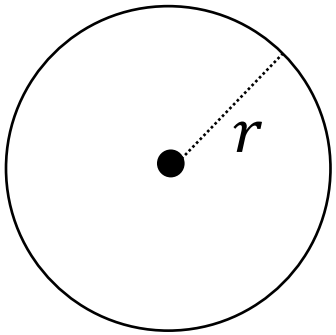




$$I = \begin{pmatrix} \frac{1}{12}m(3r^2 + h^2) & 0 & 0 \\ 0 & \frac{1}{12}m(3r^2 + h^2) & 0 \\ 0 & 0 & \frac{1}{2}mr^2 \end{pmatrix}$$

球(Sphere)の慣性行列





$$I = \begin{pmatrix} \frac{2mr^2}{5} & 0 & 0 \\ 0 & \frac{2mr^2}{5} & 0 \\ 0 & 0 & \frac{2mr^2}{5} \end{pmatrix}$$

モデルの読み込み

- 「ファイル」－「読み込み」－「ボディ」
- 左上の「アイテム」ビュー上にツリー表示
- 中央の「シーン」ビュー上に3D表示
- リロード機能
  - コンテキストメニューの「再読み込み」
  - 選択状態でCtrl + R

シーンビューの操作

- ビューモード／エディットモード
  - ダブルクリック／ESC／切り替えボタンで切り替え
  - エディットモードではロボットの移動／姿勢変更が可能
- 視点操作（ビューモード）
  - 左ドラッグ：視点回転
  - 中央ドラッグ
    - 視点平行移動
    - Ctrlを押しているとズーム
  - ホイール：ズーム
  - 右ボタン：コンテキストメニュー

プロジェクトファイルの保存

- 「ファイル」ー「プロジェクトに名前を付けて保存」を選択
- 「プロジェクトの保存」ダイアログで保存する
  - “my_mobile_robot”以下に”project”ディレクトリを作成
  - ファイル名：”mobile_robot.cnoid”

プロジェクトファイルの読込

- メニューの「ファイル」－「プロジェクトを開く」から読み込む

- または、Choreonoid起動時のコマンドラインで指定する

```
cd ~/catkin_ws/src/my_mobile_robot/project
rosrun choreonoid_ros choreonoid mobile_robot.cnoid
```

右車輪の追加

```
-
  name: RightWheel
  parent: Chassis
  translation: [ 0, -0.145, 0.076 ]
  joint_type: revolute
  joint_id: 0
  joint_axis: [ 0, 1, 0 ]
  center_of_mass: [ 0, 0, 0 ]
  mass: 0.8
  inertia: [ 0.0012, 0,      0,
             0,      0.0023, 0,
             0,      0,      0.0012 ]
  material: Tire
  elements:
    - &TireShape
      type: Shape
      geometry:
        type: Cylinder
        height: 0.03
        radius: 0.076
        division_number: 60
      appearance:
        material:
          diffuseColor: [ 0.2, 0.2, 0.2 ]
```

※ インデントに注意！

- リロード(Ctrl + R)して更新

左車輪の追加

```
-
   name: LeftWheel
   parent: Chassis
   translation: [ 0, 0.145, 0.076 ]
   joint_type: revolute
   joint_id: 1
   joint_axis: [ 0, 1, 0 ]
   center_of_mass: [ 0, 0, 0 ]
   mass: 0.8
   inertia: [ 0.0012, 0,      0,
              0,      0.0023, 0,
              0,      0,      0.0012 ]
   material: Tire
   elements:
     - *TireShape
```

※ インデントに注意！

- リロード(Ctrl + R)して更新

※ 完成版はお手本パッケージの”robot/mobile_robot_primitive.body”

モデルの操作

- シーンビューを編集モードに
  - 車体のドラッグで移動・回転
  - 車輪のドラッグで回転
- 配置ビューを用いて車体の移動
- 関節変位ビュー上のスライダ（ダイアル）操作
- チェックで表示／非表示
- コンテキストメニューから原点／重心表示
- 「表示リンクの選択」プロパティをTrue
  - 「リンク／デバイス」ビューで表示リンクの選択

シミュレーションの実行

- 以下のアイテムを追加
  - ワールド
  - AISTシミュレータ

```
+ World
    MobileRobot
    AISTSimulator
```

Worldアイテムを選択し、「ファイル」ー「新規」ー「AISTシミュレータ」で作成

※ 親子関係に注意！

- シミュレーション開始ボタンを押す
- モデルが落下する
  - cf.「重力加速度」プロパティを”０００”にしてみる

なぜWorldもSimulatorもアイテム？

- World
  - ひとつのChoreonoid上で複数の仮想世界を持てる
    - 仮想世界でシミュレーション設定を変える
    - 複数シミュレーションを同時に実行する
    - シミュレーション結果を重ねて表示・比較する
- Simulator
  - 物理計算に関する複数の設定を持てる
  - 物理計算（物理エンジン）の実装を切り替えられる
    - 用途によって使い分ける

利用可能な物理エンジン

- 産総研物理エンジン
  - 標準エンジン
  - AISTシミュレータアイテム
- Open Dynamics Engine
  - オープンソースで広く利用されている物理エンジン
  - ODEプラグインで導入
  - ODEシミュレータアイテム
- AGX Dynamics
  - 商用物理エンジンで高機能・高性能
  - ライセンスの購入が必要
  - AGX Dynamicsプラグインで導入
  - AGXシミュレータアイテム

- 実験的対応のある物理エンジン
  - NVIDIA PhysX, Bullet Physics, Springhead, Roki

床の追加

- 床モデルを読み込む
  - choreonoid-2.0/model/misc/floor.body

```
+ World
    MobileRobot
    Floor
    AISTSimulator
```

- 落ちなくなる
- 車体をドラッグして引っ張ってみる

接触マテリアル

- 各リンクに material: マテリアル名 で指定可能
- 利用可能なマテリアルはChoreonoid本体のshare/default/materials.yamlに記載
- マテリアルの組み合わせごとに摩擦係数や反発係数を設定

タイムステップの設定

- シミュレーションの１コマあたりの時間
- デフォルトは0.001秒（1ミリ秒）
- シミュレーションの正確性・安定性とシミュレーション速度とのトレードオフ
- 安定にシミュレーションできる範囲で必要に応じて増やしておく
- 1ミリ秒の細かさであればほとんどのケースで安定

メッシュファイルの利用(車体)

```
-
   name: Chassis
   joint_type: free
   center_of_mass: [ -0.08, 0, 0.08 ]
   mass: 14.0
   inertia: [ 0.1,   0,     0,
              0,     0.17,  0,
              0,     0,     0.22 ]
   material: Slider
   elements:
     -
       type: Resource
       uri: "../meshes/vmega_body.dae"
```

- メッシュファイル”vmega_body.dae”はお手本パッケージからコピーします
- uri: “package://パッケージ名/meshes/vmega_body.dae” と書いてもOK

※ https://github.com/vstoneofficial/megarover_samples の
メッシュファイルを利用（一部修正）

メッシュファイルの利用(ホイール)

```
-
   name: RightWheel
   parent: Chassis
   translation: [ 0, -0.145, 0.076 ]
   joint_type: revolute
   joint_id: 0
   joint_axis: [ 0, 1, 0 ]
   center_of_mass: [ 0, 0, 0 ]
   mass: 0.8
   inertia: [ 0.0012, 0,      0,
              0,      0.0023, 0,
              0,      0,      0.0012 ]
   material: Tire
   elements:
     -
       type: Resource
       uri: "../meshes/vmega_wheel.dae"
```

```
-
   name: LeftWheel
   parent: Chassis
   translation: [ 0, 0.145, 0.076 ]
   joint_type: revolute
   joint_id: 1
   joint_axis: [ 0, 1, 0 ]
   center_of_mass: [ 0, 0, 0 ]
   mass: 0.8
   inertia: [ 0.0012, 0,      0,
              0,      0.0023, 0,
              0,      0,      0.0012 ]
   material: Tire
   elements:
     -
       type: Resource
       uri: "../meshes/vmega_wheel.dae"
       rotation: [ 0, 0, 1, 180 ]
```

※ 完成版はお手本パッケージの “robot/mobile_robot.body”

利用可能なメッシュファイル

- ネイティブでサポートしている形式
  - STL
  - OBJ
  - VRML97 (拡張子: wrl)
  - Choreonoid標準シーンファイル（独自形式）
- Assimpライブラリでサポート
  - Collada (拡張子: dae)
  - その他多数

補足：影の設定について

- デフォルトでは影の描画が有効
- 描画が重い場合は、影の描画を無効化すると多少改善されます
- シーンバー右端の設定ボタンを押して設定ダイアログを開く
- 「ライティング」の「ワールドライド」の「影」のチェックを外す

2日目

Part5

ロボット制御の基本

ロボット車体の制御

- いかにして左右の車輪の回転を制御するか？

制御プログラムの導入

- 制御プログラム＝コントローラ
- 自前でコントローラを実装する
  - Choreonoidのシンプルコントローラ形式で実装する
- 既存のコントローラを利用する
  - ros_controlのコントローラも利用可能

制御のためのアイテム

- シンプルコントローラアイテム
  - C++を用いて自前で実装
  - ROSとの連携もroscppライブラリで実現可能
- ROSControlアイテム
  - ros_controlのコントローラ（モジュール）を利用

シンプルコントローラの導入

- 該当アイテムを追加

```
+ World
  + MobileRobot
      SimpleController
    Floor
    AISTSimulator
```

MobileRobotアイテムを選択し、
「ファイル」ー「新規」ー
「シンプルコントローラ」で作成

※ 親子関係に注意！

- 「コントローラモジュール」プロパティでコントローラ本体を指定

まずとにかく動かしてみる

- 左右ホイール（のモーター）に一定のトルクを発生させてロボットを動かす
- コントローラの名前は“MobileRobotDriveTester”とする

コントローラ本体の作成

- src/MobileRobotDriveTester.cppを作成
- ビルド用のCMakeLists.txtも作成
- catkin buildでビルド
- MobileRobotDriveTester.so が生成される

「コントローラモジュール」プロパティに設定

マニュアルの関連ページ

- コントローラの実装（とそれに続くページ）
  - https://choreonoid.org/ja/documents/latest/simulation/howto-implement-controller.html
- Tankチュートリアル
  - https://choreonoid.org/ja/documents/latest/simulation/tank-tutorial/index.html
- ROS版Tankチュートリアル
  - https://choreonoid.org/ja/documents/latest/ros/tank-tutorial/index.html

MobileRobotDriveTester.cpp (1/2)

コントローラクラスの定義

```
#include <cnoid/SimpleController>

class MobileRobotDriveTester : public cnoid::SimpleController
{
public:
    virtual bool initialize(cnoid::SimpleControllerIO* io) override;
    virtual bool control() override;

private:
    cnoid::Link* wheels[2];
};

CNOID_IMPLEMENT_SIMPLE_CONTROLLER_FACTORY(MobileRobotDriveTester)
```

MobileRobotDriveTester.cpp (2/2)

初期化関数、制御関数

```
bool MobileRobotDriveTester::initialize(cnoid::SimpleControllerIO* io)
{
    auto body = io->body();
    wheels[0] = body->joint("RightWheel");
    wheels[1] = body->joint("LeftWheel");
    for(int i=0; i < 2; ++i){
        auto wheel = wheels[i];
        wheel->setActuationMode(JointTorque);
        io->enableOutput(wheel, JointTorque);
    }
    return true;
}

bool MobileRobotDriveTester::control()
{
    wheels[0]->u() = 1.0;
    wheels[1]->u() = 1.0;
    return true;
}
```

トップのCMakeLists.txt

パッケージ初期化時に生成された雛形を修正する

```
cmake_minimum_required(VERSION 3.10.0)
project(my_mobile_robot)

find_package(catkin REQUIRED
  COMPONENTS roscpp std_msgs sensor_msgs trajectory_msgs choreonoid)

set(CMAKE_CXX_STANDARD ${CHOREONOID_CXX_STANDARD})
set(CMAKE_CXX_EXTENSIONS OFF)

include_directories(${catkin_INCLUDE_DIRS} ${CHOREONOID_INCLUDE_DIRS})

link_directories(${CHOREONOID_LIBRARY_DIRS})

add_subdirectory(src)
```

※ お手本パッケージのファイルをコピーする場合、projectコマンドで指定するプロジェクト名は実際のパッケージ名と一致するようにしてください。

srcのCMakeLists.txt

- src/CMakeLists.txtを以下の内容で作成

```
choreonoid_add_simple_controller(
  MyMobileRobotDriveTester MobileRobotDriveTester.cpp)
```

※ お手本パッケージとの競合を避けるため、”My” をつけておきます

ビルド

- catkin build コマンドでビルドする

```
catkin_build
```

※ catkinワークスペース内のディレクトリであればどこで実行してもOK

- **My**MobileRobotDriveTester.so が生成される

「コントローラモジュール」プロパティに設定

生成されるディレクトリは
“~/catkin_ws/devel/lib/choreonoid-x.x/simplecontroller”

補足：アイテム名の修正

```
+ World
  + MobileRobot
      SimpleController
    Floor
    AISTSimulator
```

```
+ World
  + MobileRobot
      DriveTester
    Floor
    AISTSimulator
```

シンプルコントローラアイテムの名前も変更しておくと分かりやすくなる

※ お手本パッケージの該当プロジェクトファイルは
“project/mobile_robot_drive_test.cnoid”

Part6

ROS通信を用いた制御

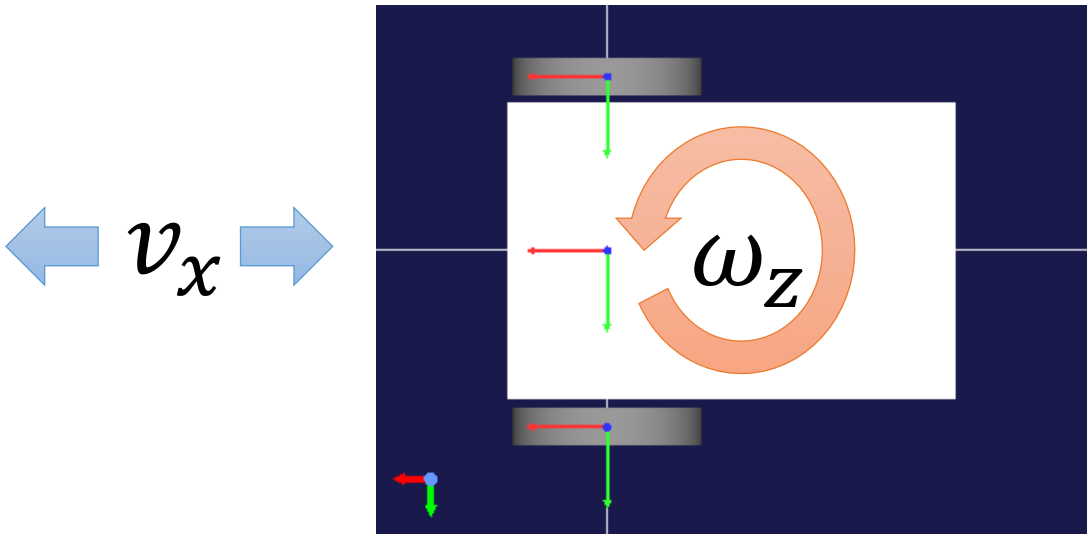
外部入力に基づく制御

- 入力する値（指令値）の種類を決める

- どうやって入力を得るか？
  - ゲームパッド等のデバイスから直接入力
  - 通信を用いる
    - ROS、OpenRTM、ソケット通信、etc.

速度指令値

- 前後方向の速度(v_x)とロボット全体のYaw軸まわりの角速度(ω_z)で制御



- 速度指令値に追従するよう両ホイールのトルクを設定する（目標速度制御）

ROSメッセージ型

- ROS通信で送信／受信する値の型
- デフォルトで多数定義されている
- 定義を追加することも可能
- メッセージ型の一覧を表示

```
rosmsg list
```

- メッセージ型の内容を表示

```
rosmsg show 型名
```

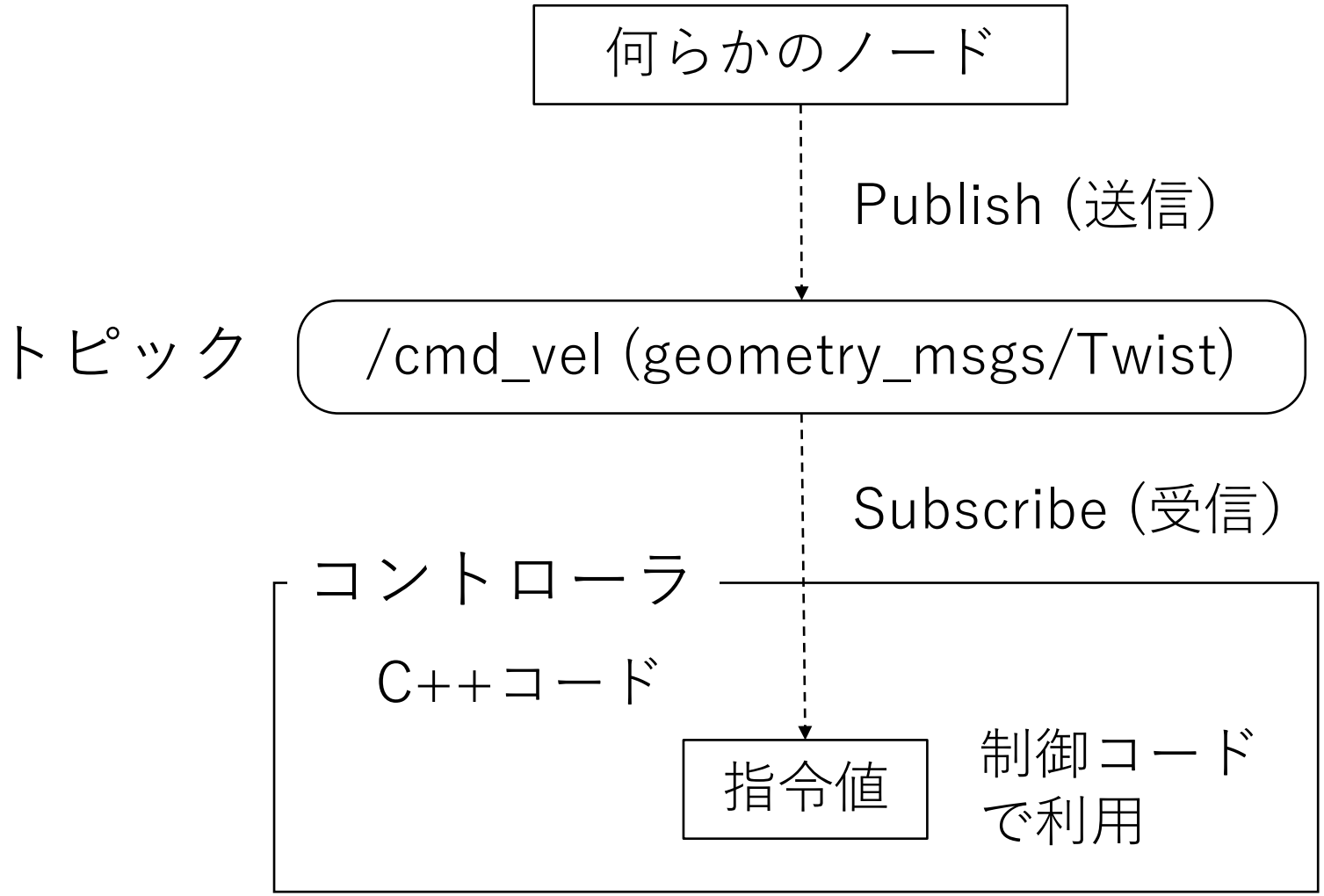
ROSのTwistメッセージ型

- geometry_msgs/Twist
- 速度 + 角速度

```
rosmsg show geometry_msgs/Twist
```

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Twistメッセージの入力



コントローラの作成

- MobileRobotDriveTester.cppを改良して、MobileRobotDriveController.cppを作成
- roscppライブラリを利用
  - ROSトピックのsubscribeを実装
- src/CMakeLists.txtに追記してビルド
- SimpleControllerアイテムのモジュールを入れ替える
  - お手本リポジトリの該当プロジェクトファイルは“mobile_robot_twist.cnoid” になります

MobileRobotDriveController.cpp (1/3)

```
#include <cnoid/SimpleController>
#include <ros/node_handle.h>
#include <geometry_msgs/Twist.h>
#include <mutex>

class MobileRobotDriveController : public cnoid::SimpleController
{
public:
    virtual bool initialize(cnoid::SimpleControllerIO* io) override;
    void commandCallback(const geometry_msgs::Twist& twist);
    virtual bool control() override;

private:
    cnoid::Link* wheels[2];
    std::unique_ptr<ros::NodeHandle> node;
    ros::Subscriber subscriber;
    geometry_msgs::Twist command;
    std::mutex commandMutex;
};

CNOID_IMPLEMENT_SIMPLE_CONTROLLER_FACTORY(MobileRobotDriveController)
```

※ 赤字はMobileRobotDriveTester.cppから修正／追加する部分

MobileRobotDriveController.cpp (2/3)

```
bool MobileRobotDriveController::initialize(cnoid::SimpleControllerIO* io)
{
    auto body = io->body();
    wheels[0] = body->joint("RightWheel");
    wheels[1] = body->joint("LeftWheel");
    for(int i=0; i < 2; ++i){
        auto wheel = wheels[i];
        wheel->setActuationMode(JointTorque);
        io->enableInput(wheel, JointVelocity);
        io->enableOutput(wheel, JointTorque);
    }

    node = std::make_unique<ros::NodeHandle>();
    subscriber = node->subscribe(
        "/cmd_vel", 1, &MobileRobotDriveController::commandCallback, this);

    return true;
}

void MobileRobotDriveController::commandCallback(const geometry_msgs::Twist& twist)
{
    std::lock_guard<std::mutex> lock(commandMutex);
    command = twist;
}
```

MobileRobotDriveController.cpp (3/3)

```
bool MobileRobotDriveController::control()
{
    constexpr double wheelRadius = 0.076;
    constexpr double halfAxleWidth = 0.142;
    constexpr double kd = 0.5;
    double dq_target[2];

    {
        std::lock_guard<std::mutex> lock(commandMutex);
        double dq_x = command.linear.x / wheelRadius;
        double dq_yaw = command.angular.z * halfAxleWidth / wheelRadius;
        dq_target[0] = dq_x + dq_yaw;
        dq_target[1] = dq_x - dq_yaw;
    }

    for(int i=0; i < 2; ++i){
        auto wheel = wheels[i];
        wheel->u() = kd * (dq_target[i] - wheel->dq());
    }

    return true;
}
```

srcのCMakeLists.txt

- src/CMakeLists.txtを以下の内容で作成

```
choreonoid_add_simple_controller(
  MyMobileRobotDriveTester MobileRobotDriveTester.cpp)

choreonoid_add_simple_controller(
  MyMobileRobotDriveController MobileRobotDriveController.cpp)

target_link_libraries(
  MyMobileRobotDriveController ${roscpp_LIBRARIES})
```

※ お手本パッケージとの競合を避けるため、”My” をつけておきます

プロジェクトの構成

```
+ World
  + MobileRobot
      DriveController
    Floor
    AISTSimulator
```

※「コントローラモジュール」プロパティに “MyMobileRobot DriveController.so”を設定

※ お手本パッケージの該当プロジェクトファイルは “project/mobile_robot_drive_control.cnoid”

トピックの確認

シミュレーションを開始する

利用可能なトピックを表示

```
rostopic list
```

/cmd_velが表示されているか？

内容の確認

```
rostopic info /cmd_vel
```

/cmd_velトピックのPublish

- 様々な手段がある
  - “rostopic pub” コマンド
  - rqt_robot_steeringツール
  - ゲームパッド + joyノード

rostopicコマンドによる操作

```
rostopic pub -1 /cmd_vel geometry_msgs/Twist
 '{ linear: { x: 0.5, y: 0, z: 0 }, angular: { x: 0, y: 0, z: 0 } }'
```

前後方向速度　　**旋回角速度**

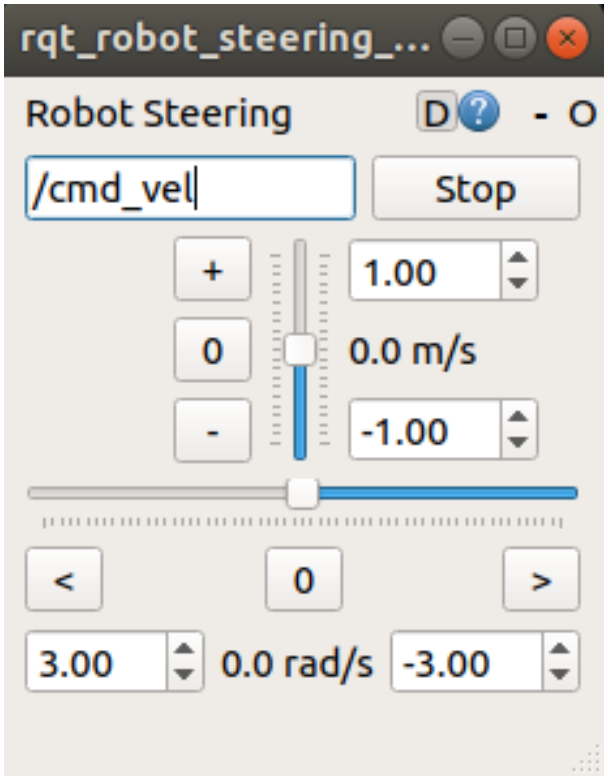
※ コロン(:)の後にスペースが必要！

publishされている内容を確認

```
rostopic echo /cmd_vel
```

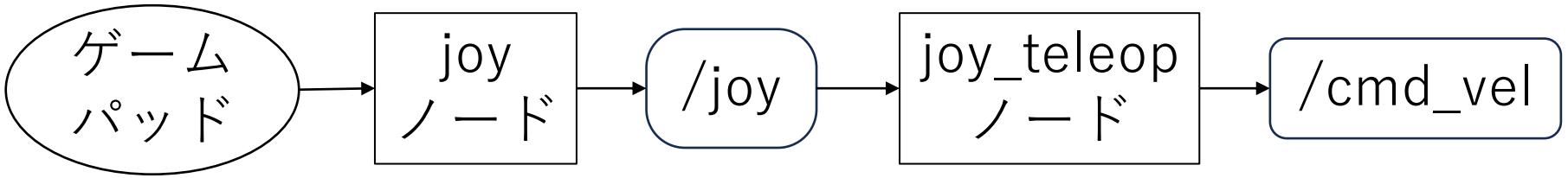
rqt_robot_steering による操作

```
rosrun rqt_robot_steering rqt_robot_steering
```



ゲームパッドによる操作

- ゲームパッドを接続する
- joyノードを用いてゲームパッドからの入力をjoyトピックとしてpublishする
- joy_teleopノードを用いてjoyトピックをtwistトピックに変換する



必要パッケージのインストール

```
sudo apt install ros-noetic-joy ros-noetic-joy-teleop
```

または

- Package.xmlの<exec_depend>タグを記述しておく
- <depend>joy</depend>
- <depend>joy_teleop0
- </depend>
- ハイフン、アンダースコアの違いに注意

```
rosdep install –I –from-paths
  ~/catin_ws/src/choreonoid_ros_mobile_robot_tutorial
```

joyトピック

joyノードを起動

```
rosrun joy joy_node
```

利用可能なトピックを表示

```
rostopic list
```

joyトピックの内容を表示

```
rostopic info /joy
```

Joyメッセージ型の内容を表示

```
rosmsg show sensor_msgs/Joy
```

publishされている内容を確認

```
rostopic echo /joy
```

トピックの変換

- /joyトピックから/cmd_velトピック(Twist型)に変換する
- joy_teleopノードによる変換が可能

手順

- YAML形式の設定ファイルを作成する
- rosparamコマンドで読み込む
- joy_teleopノードを起動する

設定ファイル

config/joy_teleop.yaml として作成する

```
teleop:
  move:
    type: topic
    message_type: geometry_msgs/Twist
    topic_name: /cmd_vel
    axis_mappings:
      -
        axis: 1
        target: linear.x
        scale: 1.0
      -
        axis: 0
        target: angular.z
        scale: 2.0
```

※ お手本パッケージの完成版はconfig/joy_teleop_twist.yaml

設定の読み込み＆起動

```
rosparam load joy_teleop.yaml
rosrun joy_teleop joy_teleop.py
```

“rostopic echo”コマンドで確認

```
rostopic echo /cmd_vel
```

シミュレーションを開始してロボットをゲームパッドで操作してみる

Launchファイルによる一括起動

- Launchファイルとは
  - ノードの起動の仕方を記述したもの
  - 複数のノードの起動も可能
  - 起動コマンドが複雑になる場合はLaunchファイルにまとめておく
  - 各パッケージのlaunchディレクトリに入れておく
- これまでのシミュレーションを一括起動するLaunchファイルを作成する

Launchファイルの作成

“launch/drive_control.launch”

```
<launch>
  <node pkg="joy" type="joy_node" name="joy" respawn="true" />
  <rosparam command="load" file="$(find
   my_mobile_robot)/config/joy_teleop.yaml"/>
  <node pkg="joy_teleop" type="joy_teleop.py" name="joy_teleop" />
  <node pkg="choreonoid_ros" type="choreonoid" name="chorenoid “
   args="--start-simulation $(find
   my_mobile_robot)/project/mobile_robot.cnoid" />
</launch>
```

※ 実際のプロジェクトファイルを指定

Launchファイルの実行

```
roslaunch my_mobile_robot drive_control.launch
```

終了するときは端末から”Ctrl + C” を入力する

Part7

関節のモデリングと制御

関節の追加

- 機器搭載用のパン・チルトの 2 軸を追加する

パン関節の追加(1/2)

```
links:
  ...
  -
    name: PanLink
    parent: Chassis
    translation: [ -0.02, 0, 0.165 ]
    joint_name: PanJoint
    joint_type: revolute
    joint_id: 2
    joint_axis: [ 0, 0, 1 ]
    center_of_mass: [ 0, 0, 0.03 ]
    mass: 1.0
    inertia: [ 0.002, 0,     0,
               0,     0.002, 0,
               0,     0,     0.003 ]
```

※ インデントに注意！

パン関節の追加(2/2)

※ インデントに注意！

```
-
  name: PanLink
  ...
  elements:
    -
      type: Shape
      translation: [ 0, 0, 0.01 ]
      rotation: [ 1, 0, 0, 90 ]
      geometry: { type: Cylinder, radius: 0.08, height: 0.02 }
      appearance: &GRAY
        material: { diffuse: [ 0.5, 0.5, 0.5 ] }
    -
      type: Transform
      translation: [ 0, 0.07, 0.065 ]
      elements:
        - &PanFrame
          type: Shape
          geometry: { type: Box, size: [ 0.02, 0.02, 0.13 ] }
          appearance: *GRAY
    -
      type: Transform
      translation: [ 0, -0.07, 0.065 ]
      elements: *PanFrame
```

チルト関節の追加

※ インデントに注意！

```
-
   name: TiltLink
   parent: PanLink
   translation: [ 0, 0, 0.12 ]
   joint_name: TiltJoint
   joint_type: revolute
   joint_id: 3
   joint_axis: [ 0, 1, 0 ]
   mass: 1.0
   inertia: [ 0.001, 0,     0,
              0,     0.001, 0,
              0,     0,     0.002 ]
   elements:
     -
       type: Shape
       rotation: [ 1, 0, 0, 90 ]
       geometry: { type: Cylinder, radius: 0.06, height: 0.02 }
       appearance: *GRAY
```

※ お手本パッケージの完成版は “model/mobile_robot_pantilt.body”

関節リミット値の設定

- 今回は設定しない
- 関節角度範囲の指定
  - joint_range: [下限値, 上限値]
- 関節角速度範囲の指定
  - joint_velocity_range: [下限値, 上限値]
- シミュレーション時にリミット値が有効となる（動作が制限される）かどうかは物理エンジンによる
- AISTシミュレータは動作の制限はなされない

表示するリンクの選択

- 対象ボディのプロパティで「表示リンクの選択」をtrueにする
- 「リンク／デバイス」ビューでリンクを選択する

パン・チルト軸の目標角速度制御

- 制御指令として以下を用いる
  - ω_z：パン軸の目標角速度
  - ω_y：チルト軸の目標角速度

- 目標角速度と現在速度の差分からトルクを計算する

コントローラの作成と導入

- MobileRobotPanTiltController.cppを作成
  - MobileRobotDriveController.cppと同様に
- src/CMakeLists.txtに追記してビルド
- SimpleControllerアイテムを追加する
  - ビルドしたコントローラモジュールをセット
  - 複数のコントローラが存在する場合、どちらも機能します

MobileRobotPanTiltController.cpp (1/3)

```
#include <cnoid/SimpleController>
#include <ros/node_handle.h>
#include <geometry_msgs/Vector3.h>
#include <mutex>

class MobileRobotPanTiltController : public cnoid::SimpleController
{
public:
    virtual bool initialize(cnoid::SimpleControllerIO* io) override;
    void commandCallback(const geometry_msgs::Vector3& omega);
    virtual bool control() override;

private:
    cnoid::Link* joints[2];
    std::unique_ptr<ros::NodeHandle> node;
    ros::Subscriber subscriber;
    geometry_msgs::Vector3 command;
    std::mutex commandMutex;
};

CNOID_IMPLEMENT_SIMPLE_CONTROLLER_FACTORY(MobileRobotPanTiltController)
```

※ 赤字はMobileRobotDriveController.cppとは異なる部分

MobileRobotPanTiltController.cpp (2/3)

```
bool MobileRobotPanTiltController::initialize(cnoid::SimpleControllerIO* io)
{
    auto body = io->body();
    joints[0] = body->joint("PanJoint");
    joints[1] = body->joint("TiltJoint");
    for(int i = 0; i < 2; ++i){
        cnoid::Link* joint = joints[i];
        joint->setActuationMode(JointTorque);
        io->enableInput(joint, JointVelocity);
        io->enableOutput(joint, JointTorque);
    }

    node = std::make_unique<ros::NodeHandle>();
    subscriber = node->subscribe(
        "/cmd_joint_vel", 1, &MobileRobotPanTiltController::commandCallback, this);

    return true;
}

void MobileRobotPanTiltController::commandCallback(const geometry_msgs::Vector3& omega)
{
    std::lock_guard<std::mutex> lock(commandMutex);
    command = omega;
}
```

MobileRobotPanTiltController.cpp (3/3)

```
bool MobileRobotPanTiltController::control()
{
    constexpr double kd = 0.1;
    double dq_target[2];

    {
        std::lock_guard<std::mutex> lock(commandMutex);
        dq_target[0] = command.z;
        dq_target[1] = command.y;
    }

    for(int i=0; i < 2; ++i){
        cnoid::Link* joint = joints[i];
        joint->u() = kd * (dq_target[i] - joint->dq());
    }

    return true;
}
```

srcのCMakeLists.txt

- src/CMakeLists.txtに以下を追記

```
choreonoid_add_simple_controller(
  MyMobileRobotPanTiltController MobileRobotPanTiltController.cpp)

target_link_libraries(
  MyMobileRobotPanTiltController ${roscpp_LIBRARIES})
```

※ お手本パッケージとの競合を避けるため、”My” をつけておきます

プロジェクトの構成

```
+ World
  + MobileRobot
      DriveController
      PanTiltController
    Floor
    AISTSimulator
```

※「コントローラモジュール」プロパティに “MyMobileRobot PanTiltController.so”を設定

※ お手本パッケージの該当プロジェクトファイルは
“project/mobile_robot_drive_pantilt_control.cnoid”

rostopicコマンドによる操作

```
rostopic pub -1 /cmd_joint_vel geometry_msgs/Vector3
 '{ x: 0, y: 0, z: 0.5 }'
```

チルト角速度　　パン角速度

※ コロン(:)の後にスペースが必要！

joy_teleop追加設定

config/joy_teleop.yaml に追記する

```
pan_tilt:
   type: topic
   message_type: geometry_msgs/Vector3
   topic_name: /cmd_joint_vel
   axis_mappings:
     -
       axis: 4
       target: y
       scale: 2.0
     -
       axis: 3
       target: z
       scale: 2.0
```

※ インデントに注意！

※ お手本パッケージの完成版はconfig/joy_teleop_all.yaml

設定の読み込み＆起動

※ joyノードも起動していること！

```
rosparam load joy_teleop.yaml
rosrun joy_teleop joy_teleop.py
```

“rostopic echo”コマンドで確認

```
rostopic echo /cmd_joint_vel
```

シミュレーションを開始してロボットをゲームパッドで操作してみる

補足

- 車輪の制御（MobileRobotDriveController）とパンチルト軸の制御（MobileRobotPanTiltController）でコントローラを分けましたが、これは段階的に実習を進めるためで、必ずしも分けて実装する必要はありません

Part8

視覚センサの追加

環境モデルの導入

- 床モデル（floor）を削除
- 研究プラントのモデル（Labo1）を読み込む
  - shareディレクトリの “Labo1/Labo1v2.body”

```
+ World
  + MobileRobot
      DriveController
      PanTiltController
    + SensorVisualizer
    Labo1
  + AISTSimulator
      GLVisionSimulator
```

シーン描画設定

- 床グリッド
- 光源
- 影

センサの追加

- センサの種類
  - AccelerationSensor
  - RateGyroSensor
  - IMU
  - ForceSensor
  - Camera
  - RangeSensor

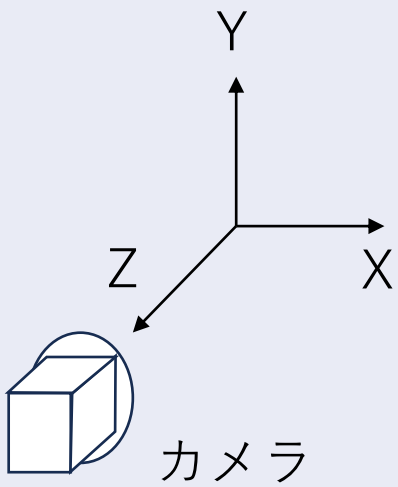
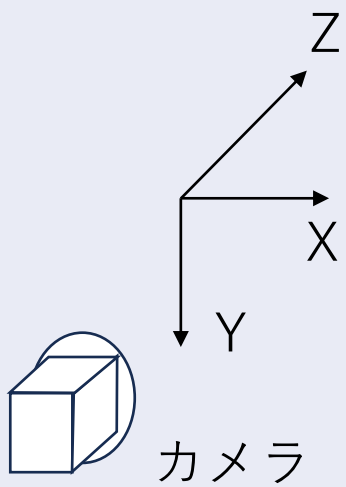
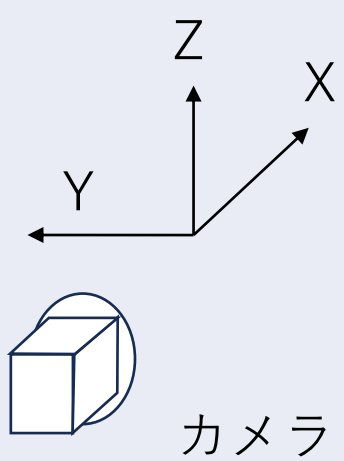
レンジセンサ(LiDAR)の追加

```
-
  name: TiltLink
  ...
  elements:
    -
      type: Shape
      ...
    -
      type: Transform
      translation: [ 0, 0, 0.04585 ]
      elements:
        -
          type: RangeSensor
          name: VLP_16
          optical_frame: robotics
          yaw_range: 360.0
          yaw_step:  0.4
          pitch_range: 30.0
          pitch_step: 2.0
          scan_rate:  20
          max_distance: 100.0
          detection_rate: 0.9
          error_deviation: 0.01
        -
          type: Shape
          rotation: [ 1, 0, 0, 90 ]
          geometry: { type: Cylinder, radius: 0.05165, height: 0.0717 }
          appearance:
            material: { diffuse: [ 0, 0, 1 ], specular: [ 1, 1, 1 ] }
```

※ センサの質量等は無視します

光学フレーム座標系

“optical_frame” フィールドで指定

シンボル	“gl”	“cv”	“robotics”
システム	OpenGL	OpenCV	ROS
前方	-Z	Z	X
鉛直上向	Y	-Y	Z
図示			

RGB-Dカメラの追加（1/2）

```
-
  type: Transform
  translation: [ 0.06, 0, -0.02 ]
  elements:
    -
      type: Camera
      name: RealSense
      optical_frame: robotics
      format: COLOR_DEPTH
      field_of_view: 62
      width: 320
      height: 240
      frame_rate: 30
      detection_rate: 0.9
      error_deviation: 0.005
    -
      形状の記述…
```

※ カメラの質量等は無視します

RGB-Dカメラの追加（2/2）

```
      -
        type: Transform
        translation: [ -0.012, 0, 0 ]
        elements:
          -
            type: Shape
            geometry: { type: Box, size: [ 0.024, 0.064, 0.022 ] }
            appearance: &SILVER
              material: { diffuse: [ 0.8, 0.8, 0.8 ], specular: [ 1, 1, 1 ] }
          -
            type: Transform
            translation: [ 0, 0.032, 0 ]
            elements:
              - &REAL_SENSE_SIDE
                type: Shape
                rotation: [ 0, 0, 1, 90 ]
                geometry: { type: Cylinder, radius: 0.011, height: 0.024 }
                appearance: *SILVER
          -
            type: Transform
            translation: [ 0, -0.032, 0 ]
            elements: *REAL_SENSE_SIDE
```

視覚センサ情報の可視化

- カメラ
  - シーンビューのカメラを切り替える
  - センサ可視化アイテムとImageビューを使用
- デプスセンサ／LiDAR
  - センサ可視化アイテムとシーンビュー

カメラの切り替え

- シーンバーのカメラ選択コンボで切り替える
- あくまでGUI上で表示する視点を切り替えるもの

シーンビューの追加

- メインメニューの「表示」－「ビューの生成」－「シーン」で追加できる
- 追加したらビューのタブをドラッグして好きな位置に配置する
- 各シーンビューをマウスでクリックしてフォーカスを入れて、その後シーンバーのカメラ選択コンボで視点を選択する
- 同時に複数視点の画像を確認できる

視覚センサのシミュレーション

- GLビジョンシミュレータアイテムを追加

```
+ World
  + MobileRobot
      DriveController
      PanTiltController
    Floor
  + AISTSimulator
      GLVisionSimulator
```

センサデータの出力と表示

- Choreonoid上で表示

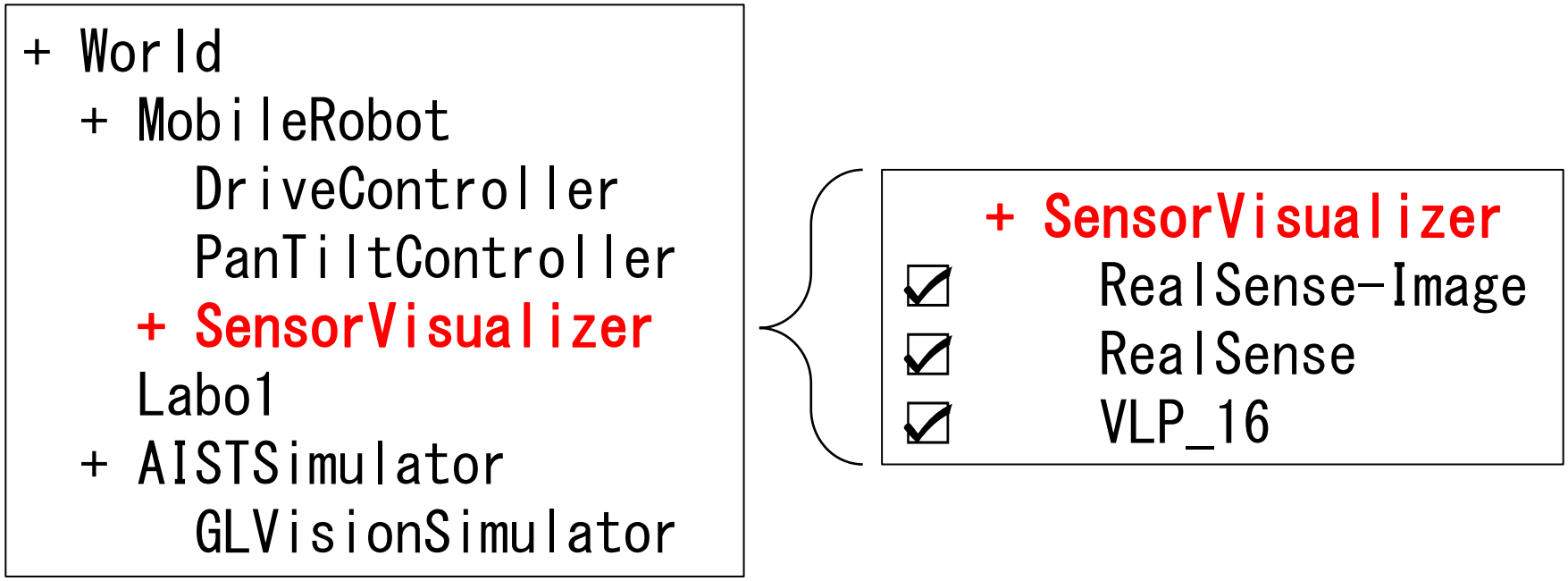
- Choreonoid外部で表示
  - 後ほどROS通信、ROSツールを用いて実現する

Choreonoid上で表示（1/3）

- GLVisionSimulatorの「ビジョンデータの記録」プロパティをtrueにする
- AISTSimulatorの「記録モード」を「オフ」にする
  - 実習用PCのメモリが少ないため
- 以上の設定により、センサデータがChoreooid上の表示用モデルに出力される

Choreonoid上で表示（2/3）

- SensorVisualizerアイテムを導入する
- 該当するセンサのチェックを入れる
- 環境モデルを非表示にするとセンサデータを確認しやすくなる



Choreonoid上で表示（3/3）

- カメラ画像については「画像ビュー」で表示可能
- 「画像ビューバー」の選択コンボで対象のセンサを選択する

お手本パッケージの該当ファイル

- プロジェクトファイル
  - “project/mobile_robot_sensors_labo1.cnoid”
- launchファイル
  - “launch/sensors_labo1.launch”

Part9

URDFファイル

URDFによるモデル記述

- ROS標準形式
- 多くのロボットモデルがURDFで配布されている
- ROSの機能を使用する際に必要となることが多い

Bodyファイル vs URDF

- Body
  - Choreonoid上の要素を全て記述できる
    - URDFの標準仕様ではセンサを記述できない
  - URDFよりも簡潔な記述になる
- URDF
  - 既存資産の活用
  - ROSで必要となることが多い

モバイルロボットのURDF

```
<?xml version="1.0"?>

<robot name="MobileRobot">
  <link name="Chassis">
    <inertial>
      <origin rpy="0 0 0" xyz="-0.08 0 0.08"/>
      <mass value="14.0"/>
      <inertia ixx="0.1" ixy="0" ixz="0" iyy="0.17" iyz="0" izz="0.22"/>
    </inertial>
    <visual>
      <geometry>
        <mesh filename="package://choreonoid_ros_mobile_robot_tutorial/meshes/vmega_body.dae"/>
      </geometry>
    </visual>
    <collision>
      <geometry>
        <mesh filename="package://choreonoid_ros_mobile_robot_tutorial/meshes/vmega_body.dae"/>
      </geometry>
    </collision>
  </link>
 <link name="RightWheel">
    <inertial>
    <mass value="0.8"/>
    ...
```

※ お手本パッケージの
“model/mobile_robot_sensors.urdf”

URDFファイルの読み込み

- 「ファイル」―「読み込み」―「ボディ」
- ダイアログ下部の「ファイルの種類」コンボボックスで「URDF」を選択
- URDF（xacro)ファイルを選択

Body <-> URDF 変換

- 手作業で変換する

- Body -> URDF
  - URDFBodyWriterを使用する
  - “feature/urdf-writer” ブランチで開発中

- URDF -> Body
  - URDFを読み込んでBody形式で保存する

Part10

ROS通信を用いた状態の出力と可視化

状態外部出力の方法

- シンプルコントローラで実装
  - C++で実装できるものならROSに限らずどのような通信も可能
  - roscppを用いることでROS通信も可能

- BodyROSアイテムを使用
  - ロボット／センサの状態をROS出力

BodyROSアイテムの導入

```
+ World
  + MobileRobot
      DriveController
      PanTiltController
      BodyROS
    + SensorVisualizer
    Labo1
  + AISTSimulator
      GLVisionSimulator
```

効率化のための補足

- BodyROSアイテムを使用して外部のROSツールで可視化を行うなら、Choreonoid上での可視化は必ずしも必要ない
- 可視化の設定を解除しておくことで動作が軽くなる

BodyROSアイテムの出力対象

- 関節角度（変位）をROSトピックとしてPublish
  - /MobileRobot/joint_states
- カメラ画像
- デプスカメラ画像（ポイントクラウド）
- レンジセンサ距離データ（ポイントクラウド）

シミュレーションを開始して、追加されたトピックを確認する

```
rostopic list
```

トピックの中身を確認する

```
rostopic echo /MobileRobot/joint_states
```

仮想世界全体情報のPublish

- WorldROSアイテムを追加する

```
+ World
  + MobileRobot
      DriveController
      PanTiltController
      BodyROS
    + SensorVisualizer
    Labo1
  + AISTSimulator
      GLVisionSimulator
    WorldROS
```

※ お手本パッケージの該当プロジェクトファイルは“project/mobile_robot_publish.cnoid”

Clockトピック

- 他のROSノードと時刻を共有（同期）
- WorldROSアイテムがPublish
- rosgraph_msgs/Clock型の/clockトピック
- ROSパラメータ /use_sim_time を trueに

```
rosparam set /use_sim_time true
```

または

```
<launch>
  <param name="/use_sim_time" value="true" />
  ...
</launch>
```

Rvizによる状態表示

- ROSの可視化・操作ツール
- モデルを読み込むにはURDF形式のモデルファイルが必要

Rviz用launchファイル

“launch/display.launch”

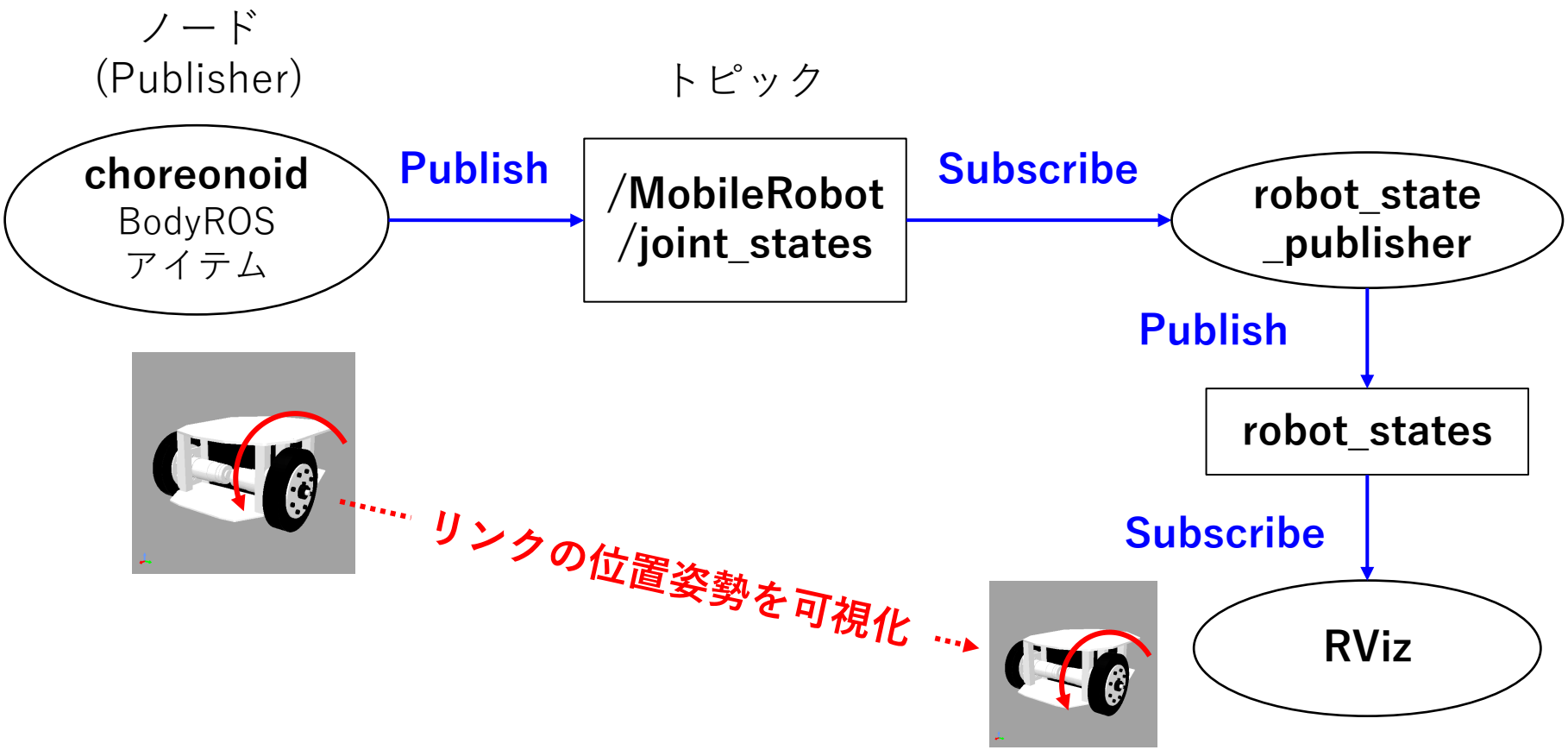
```
<launch>
  <arg name="model" default="$(find my_mobile_robot)/model/mobile_robot.urdf"/>
  <arg name="rvizconfig" default="$(find my_mobile_robot)/config/mobile_robot.rviz" />
  <param name="robot_description" command="$(find xacro)/xacro $(arg model)" />
  <remap from="/joint_states" to="/MobileRobot/joint_states" />
  <node name="robot_state_publisher" pkg="robot_state_publisher “
        type="robot_state_publisher" />
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)" required="true"
/>
</launch>
```

以下を実行

```
roslaunch my_mobile_robot display.launch
```

“config/mobile_robot.rviz” はrvizを起動後にGUIで設定して
“File” – “Save Config As” で保存しておく

joint_statesからrobot_statesへの変換



画像トピックの可視化

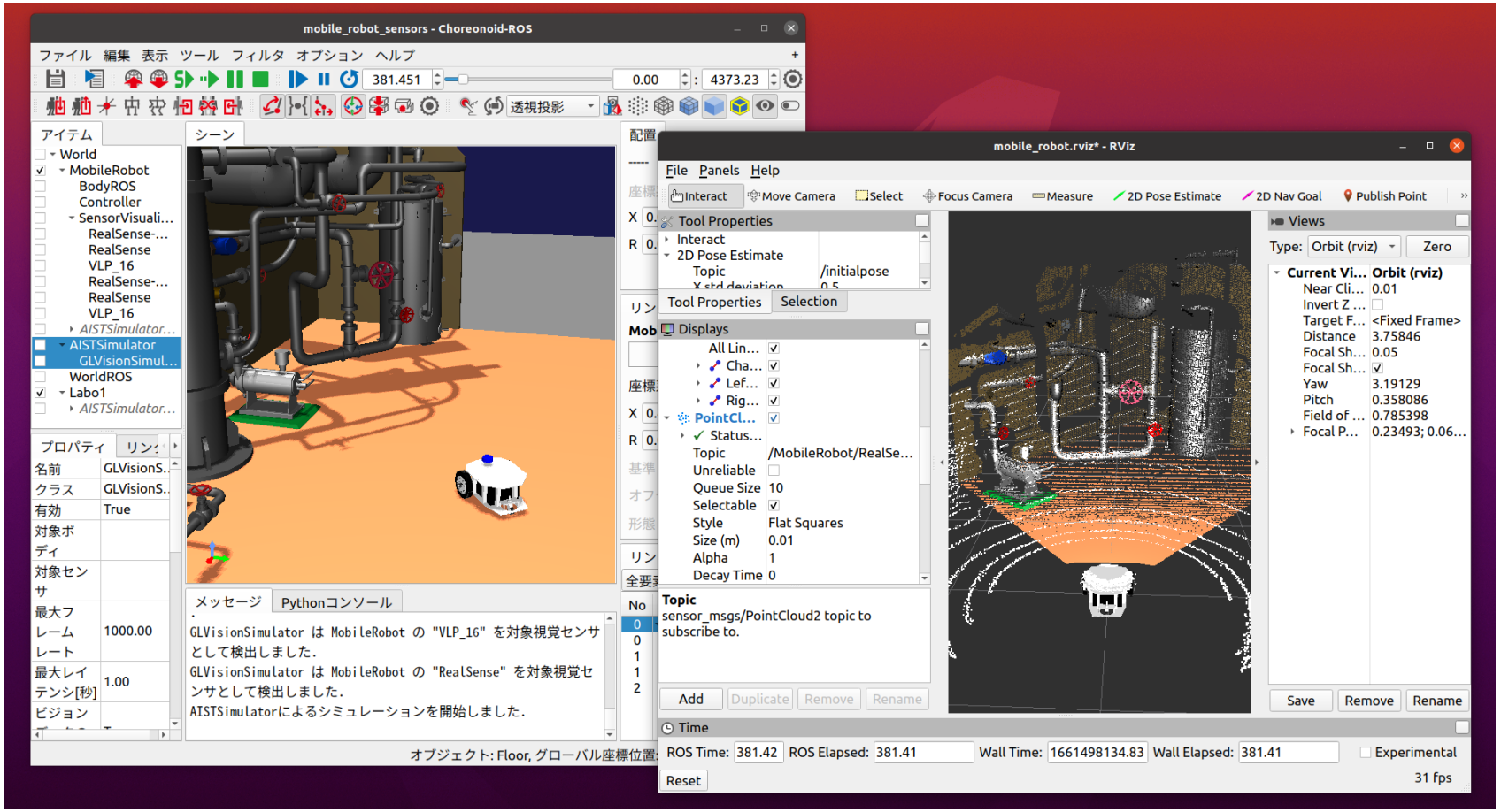
- rqt_image_viewを使う

起動

```
rosrun rqt_image_view rqt_image_view
```

左上のコンボボックスで表示する画像の
トピックを選択する

遠隔操作のシミュレーション



Part11

補足事項

センサ質量／慣性モーメント

関節軸（ロータ―）慣性

慣性センサ、力覚センサ

ライト（光源）の追加

- ライトデバイスをモデルに追加
- シーンの設定で「追加のライト」を有効化
- 必要に応じてライトの状態を制御

コントローラからのpublish

- Tankチュートリアル ROS版参照

干渉検出：メッシュ vs プリミティブ

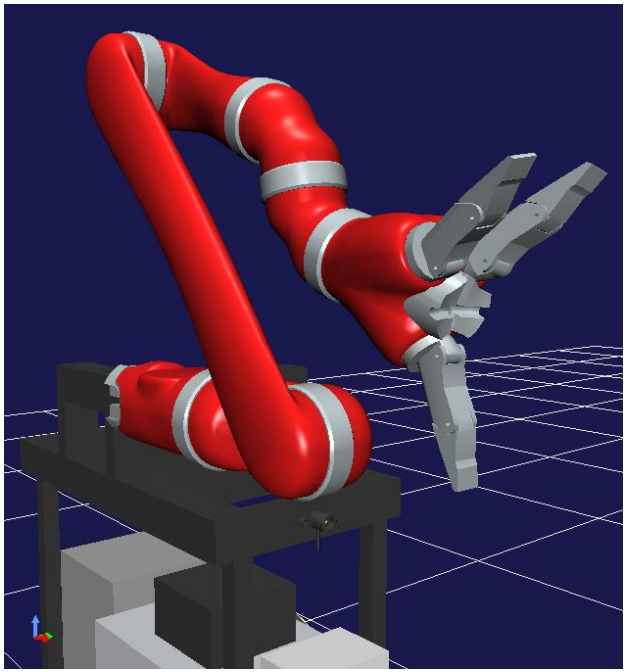
- 案外やられていない
- Gazebo（URDF）では干渉用モデルを設定できるが、「荒い形状のSTLメッシュ」であることがほとんど

- 凸形状、非凸形状

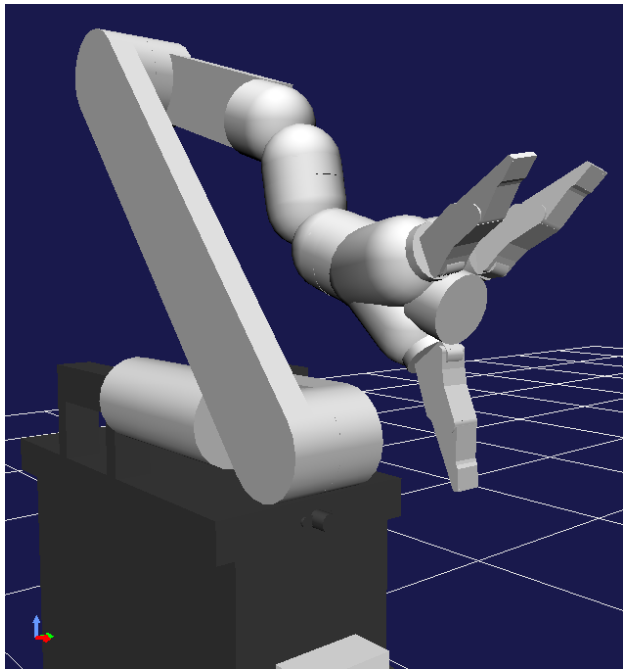
- 干渉点、法線、深さの情報が必要

表示用モデルと干渉用モデル

形状をプリミティブの組み合わせで近似し、計算を高速・安定化



表示用モデル



干渉用モデル

自己干渉検出の設定

アームの追加

- SubBodyノードを利用
- 独立したアームモデルを追加可能
- サンプル
  - PA10
  - UR3、UR5、UR10
  - AizuWheel
  - AizuSpider

Pythonスクリプト機能

参考: 物理エンジンODEの利用

- CMakeで “BUILD_ODE_PLUGIN” をONにして再ビルド

```
catkin config --cmake-args -DBUILD_ODE_PLUGIN=ON …
catkin build
```

- 「ファイル」-「新規」-「ODEシミュレータ」
  - World以下に配置
- “ODESimulator”アイテムを選択してシミュレーションを実行

シミュレーションの並列化

BodyファイルのRigidBody

ros_controlの利用

便利機能

- ボディ同期カメラ
- 距離計測機能
- ワールドログ
- 動作振付機能
- 動画作成機能
- メディア再生機能