

Choreonoid

ROS 2 モバイルロボット チュートリアル

株式会社コレオノイド

中岡 慎一郎

2024/10/12

Part1

概要

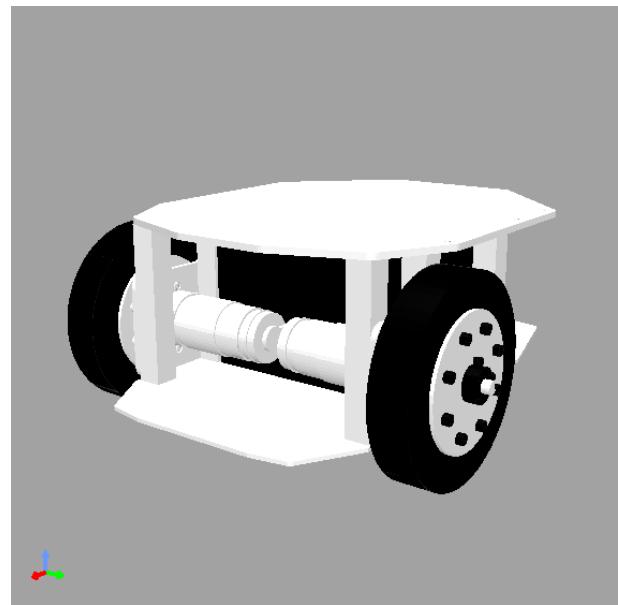
チュートリアルの目的

- ChoreonoidとROS 2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 2の基本が分かる
- 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。

内容

モバイルロボットのシミュレーション

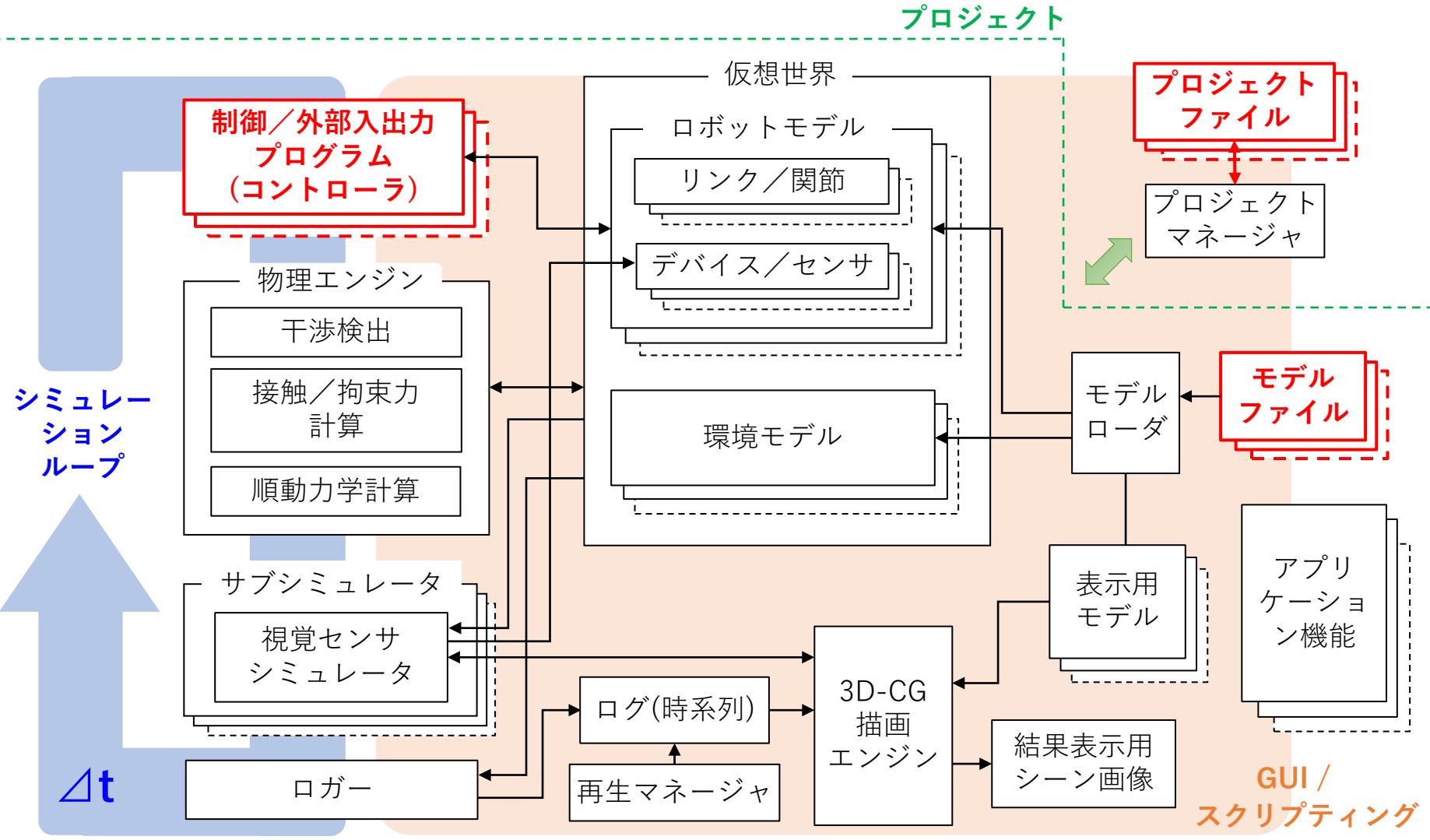
- 環境構築
- モデルの作成
- シミュレーション
- ロボットの制御
- ROS 2通信の利用
- 視覚センサの導入
- 状態の可視化
- 遠隔操作



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
 - 用意するためには必要な情報
 - 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



利用するROS 2要素

- ビルド・実行環境
 - ROSパッケージ
 - ビルドシステム “colcon”
 - launchファイル
- メッセージ通信
 - ノード
 - メッセージ型（入れ物）
 - トピック（用途）
 - 送受信
 - Publish（送信）
 - Sbscribe（受信）
- ツール
 - コマンド、可視化ツール、etc.

注：ROS 2の必要性

- シミュレーションするにあたって、必ずしも ROS 2を使う必要はありません
- 本チュートリアルは
 - ROS 2の基本
 - ChoreonoidとROS 2の連携の習得を目的としているため、あえてROS 2を使用しています
- 本チュートリアルの大部分はROS 2は無くても Choreonoidだけで同様のことが実現可能です

情報

- Choreonoid公式サイト
 - <https://choreonoid.org/>
- Choreonoid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_ros2_mobile_robot_tutorial
- Choreonoidフォーラム
 - <https://discourse.choreonoid.org/>
- ROS 2 ドキュメント
 - <https://docs.ros.org/en/jazzy/>
 - <https://docs.ros.org/en/humble/>

関連チュートリアル

- Choreonoid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(1)版Tankチュートリアル
 - <https://choreonoid.org/ja/documents/latest/ros/tank-tutorial/index.html>

本チュートリアルと同様の内容も含んでおり、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。

Part2

環境構築

対象環境

- Ubuntu 24/04, 22.04
- Choreonoid 2.2.0以上または開発版
- ROS 2 (Jazzy, Humble)
- GPU
 - 3D表示 (OpenGL) のハードウェアアクセラレーションが利用可能であること
 - Ubuntu 22.04であればデフォルトで利用可能となっていることが多い
- ゲームパッド
 - PS4またはPS5のゲームパッドを推奨

ROS 2環境の構築

- ROS 2のインストール
- ChoreonoidのROS 2へのインストール
 - choreonoid (本体)
 - choreonoid_ros (ROS 2プラグイン)
 - colcon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 2との連携」 - 「Choreonoid関連パッケージのビルド」 参照
<https://choreonoid.org/ja/documents/latest/ros2/build-choreonoid.html>

ROS 2のインストール (1/2)

```
sudo apt install software-properties-common  
sudo add-apt-repository universe  
sudo apt update && sudo apt install curl -y  
sudo curl -sSL  
https://raw.githubusercontent.com/ros/rosdistro/master/ros.key  
-o /usr/share/keyrings/ros-archive-keyring.gpg  
echo "deb [arch=$(dpkg --print-architecture) signed-by=  
/usr/share/keyrings/ros-archive-keyring.gpg]  
http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo  
$UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list >  
/dev/null  
sudo apt update  
sudo apt upgrade
```

詳細は以下を参照：

<https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.html>

ROS 2のインストール (2/2)

Ubuntu 24.04 (ROS2 Jazzy) の場合：

```
sudo apt install ros-jazzy-desktop
sudo apt install python3-colcon-common-extensions
echo "source /opt/ros/jazzy/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Ubuntu 22.04 (ROS2 Humble) の場合：

```
sudo apt install ros-humble-desktop
sudo apt install python3-colcon-common-extensions
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

rosdepのインストール

```
sudo apt install python3-rosdep  
sudo rosdep init  
rosdep update
```

パッケージの依存関係から必要なパッケージを
インストールしてくれるツール

チュートリアルで必要なパッケージのインストールに
使用します

Gitのインストール

```
sudo apt install git
```

バージョン管理システム

チュートリアルで必要なソースパッケージのインストールに使用します

ROS 2ワークスペースの作成

```
mkdir -p ~/ros2_ws/src  
cd ros2_ws
```

※ ~/ はホームディレクトリの簡略表現

Choreonoid関連パッケージの追加

```
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_ros2_mobile_robot_tutorial.git  
. ./choreonoid/misc/script/install-requisites-ubuntu-22.04.sh
```

Choreonoid関連パッケージのビルド

```
cd ~/ros2_ws  
colcon build --symlink-install
```

以下のようなメッセージが表示されたら成功

```
Starting >>> choreonoid  
Finished <<< choreonoid [120.0s]  
Starting >>> choreonoid_ros  
Finished <<< choreonoid_ros [10.0s]  
Starting >>> choreonoid_ros2_mobile_robot_tutorial  
Finished <<< choreonoid_ros2_mobile_robot_tutorial [5.0s]  
  
Summary: 3 packages finished [135.0s]
```

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/ros2_ws/install/setup.bash" >> ~/.bashrc
source $HOME/ros2_ws/install/setup.bash
```

※ 最初のビルド後に一度設定しておけばOK

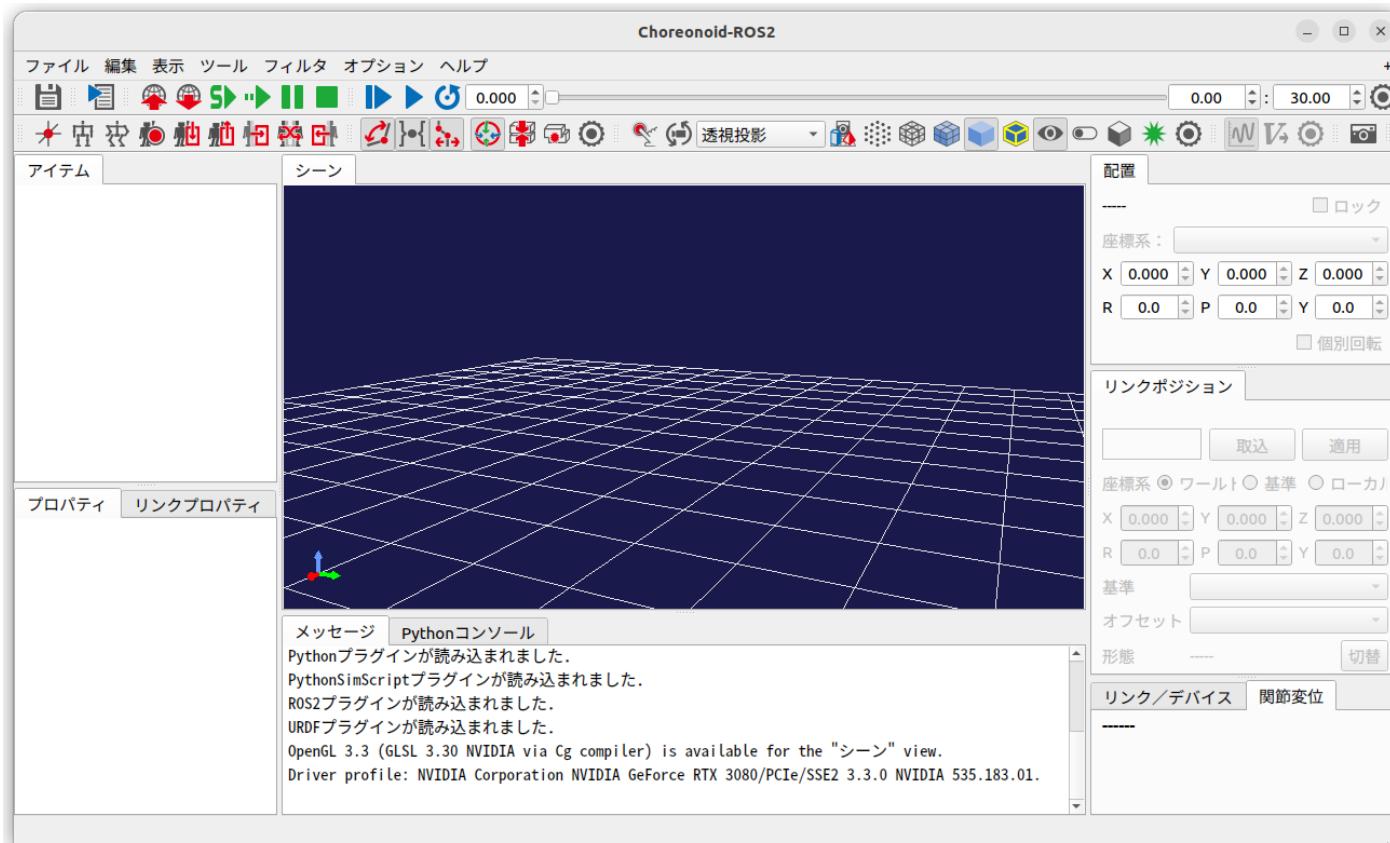
混信を防ぐ設定

```
echo "export ROS_LOCALHOST_ONLY=1" >> ~/.bashrc
```

- ネットワーク上の他のPCでもROS 2を使用している場合は、上記の設定で通信をローカルに限定しておきます
- 講習会等でも、他の参加者との混信を防ぐため、上記の設定をしておきます

Choreonoidの起動

```
ros2 run choreonoid_ros choreonoid
```



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.x” – “project” のディレクトリから、サンプルプロジェクト(.cnoidファイル)を開く
- 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_ros2_mobile_robot_tutorial
 - お手本となるパッケージ
 - 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してもOK
- ※ いずれも ~/ros2_ws/src 以下に格納

my_mobile_robotパッケージの作成

- パッケージ雛形の作成

```
cd ~/ros2_ws/src  
ros2 pkg create my_mobile_robot
```

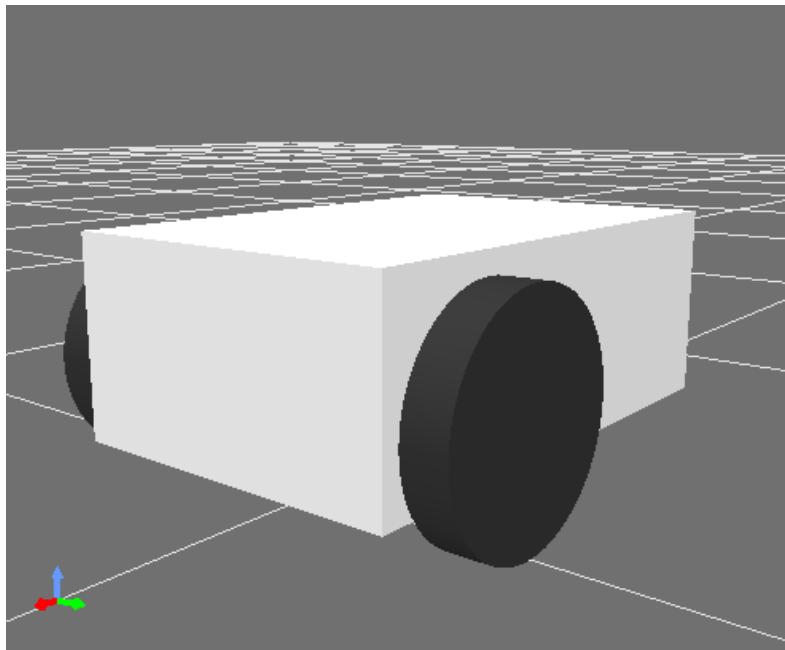
“~/ros2_ws/src” 以下に “my_mobile_robot” というディレクトリが作成されるので、その中に実習のファイルを作成していく

Part3

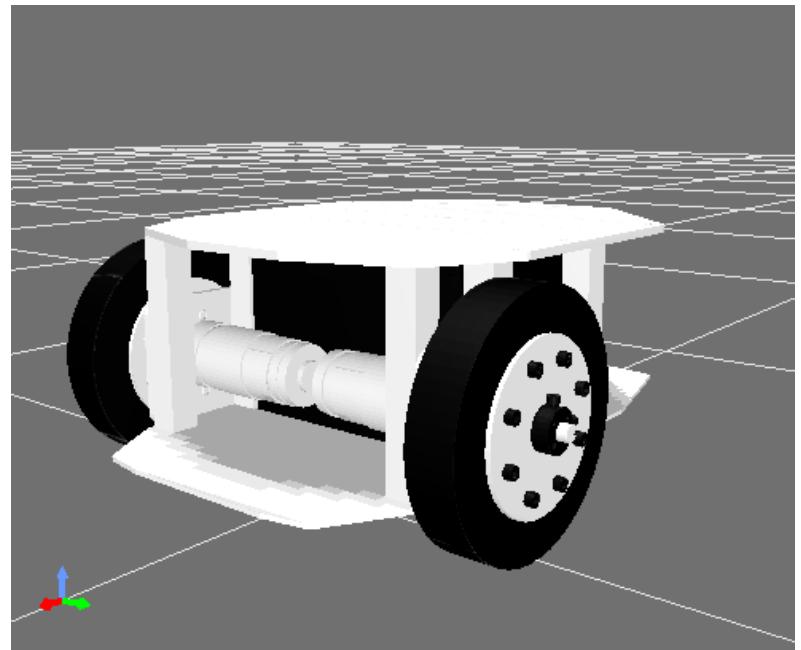
ロボットモデルの作成

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 (Part5)

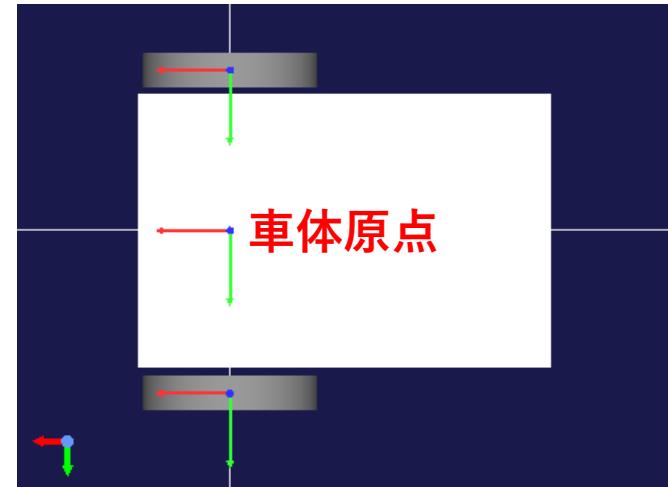
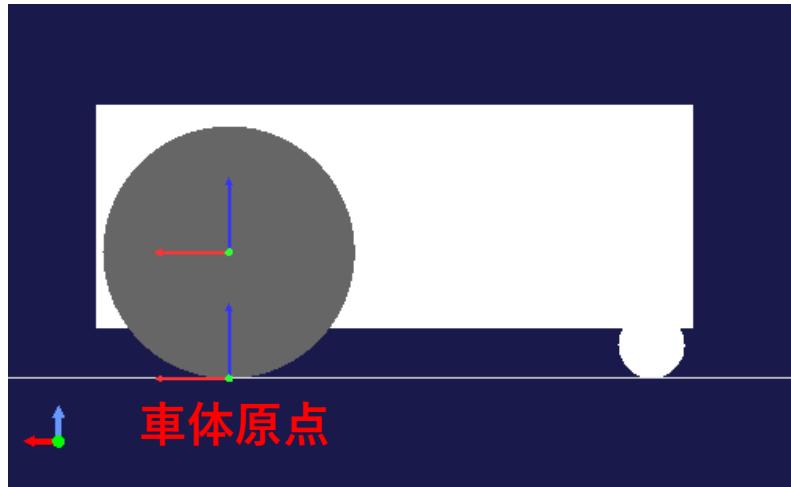
※ ヴィストン社 メガローバー2.1

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、分かりやすく扱いやすいところ**
 - X: 車軸中央、Y: 中央、Z: 床面



赤軸: X , 緑軸: Y , 青軸: Z (RGBの順)

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 ~/ros2_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)

```
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
```

※ インデントに注意！
(ここではスペース2文字を使用)

ヘッダ部

```
format: ChoreonoidBody      # 最初に書く  
format_version: 2.0          # 記述形式のバージョン  
angle_unit: degree           # 角度の単位を度にする(radianも可)  
name: MobileRobot            # モデルの名前  
root_link: Chassis           # ルートリンクの名前
```

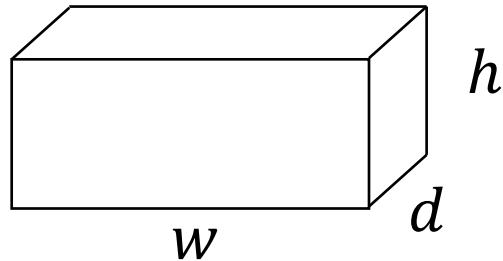
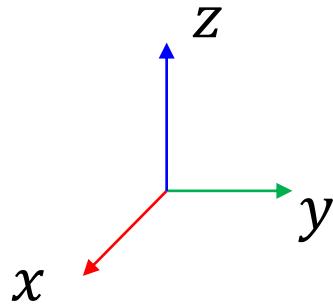
リンクの記述

```
links: # リンクのリスト（これ以下に複数のリンクを記述可能）
  -
    # リスト要素の開始
      name: Chassis          # リンク名
      joint_type: free        # 関節タイプ：free, revolute（回転）、prismatic（直動）
      center_of_mass: [ -0.08, 0, 0.08 ] # 重心（リンクローカル座標）
      mass: 14.0               # 質量
      inertia: [ 0.1, 0, 0,
                  0, 0.17, 0,
                  0, 0, 0.22 ]       # 慣性行列
      material: Slider         # 接触マテリアル：Slider（床との摩擦係数0）
      elements:
        -
          type: Shape          # 形状
          translation: [ -0.1, 0, 0.0975 ] # 並進位置（リンクローカル座標）
          geometry:
            type: Box           # プリミティブタイプ：立方体
            size: [ 0.36, 0.24, 0.135 ] # 立方体のサイズ
        -
          type: Shape          # 形状
          translation: [ -0.255, 0, 0.02 ] # 並進位置（リンクローカル座標）
          geometry:
            type: Cylinder       # プリミティブタイプ：円柱
            height: 0.01          # 高さ（Y軸方向）
            radius: 0.02          # 半径（Y軸まわり）
```

慣性行列の計算

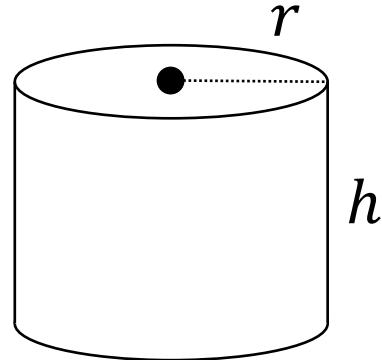
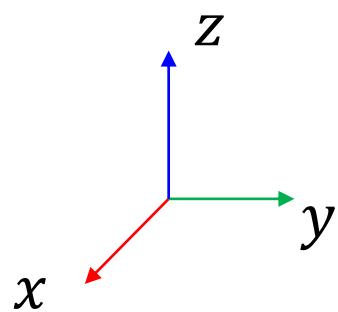
- CADツールを利用
 - プリミティブ形状に関する式から算出
- ※ 自動計算機能は未実装

直方体(Box)の慣性行列



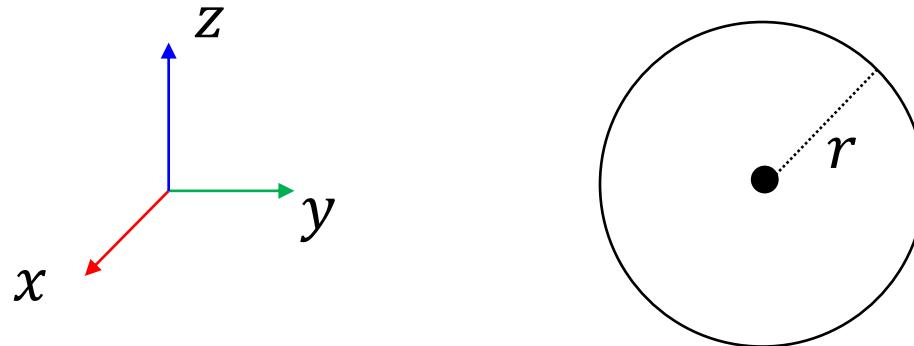
$$I = \begin{pmatrix} \frac{1}{12}m(w^2 + h^2) & 0 & 0 \\ 0 & \frac{1}{12}m(d^2 + h^2) & 0 \\ 0 & 0 & \frac{1}{12}m(w^2 + d^2) \end{pmatrix}$$

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



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

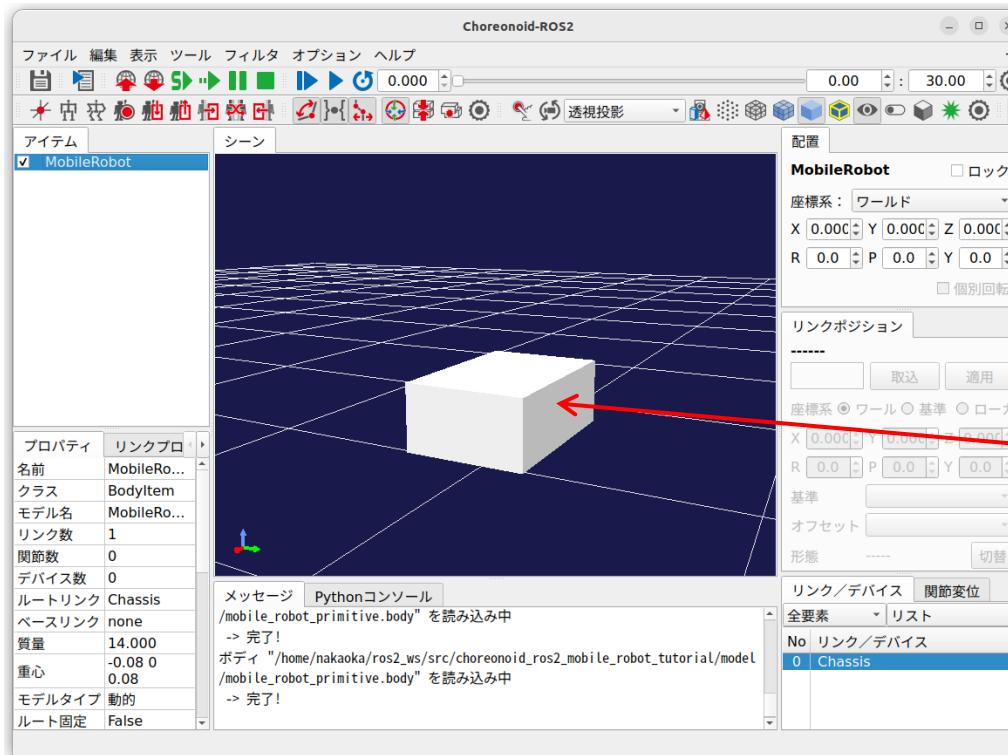
モデルの読み込み

- Choreonoidを起動

```
ros2 run choreonoid_ros choreonoid
```

- 「ファイル」 - 「読み込み」 - 「ボディ」

アイテムが
追加される →

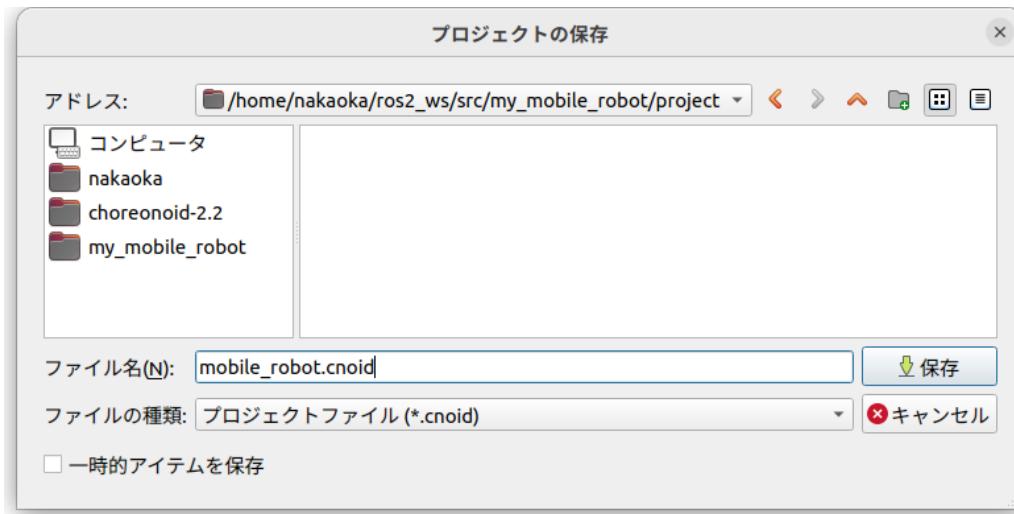


シーンビューの操作

- 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」 – 「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み込

- 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 ~/ros2_ws/src/my_mobile_robot/project  
ros2 run choreonoid_ros choreonoid mobile_robot.cnoid
```

左車輪の追加

“mobile_robot.body” に以下を追記する

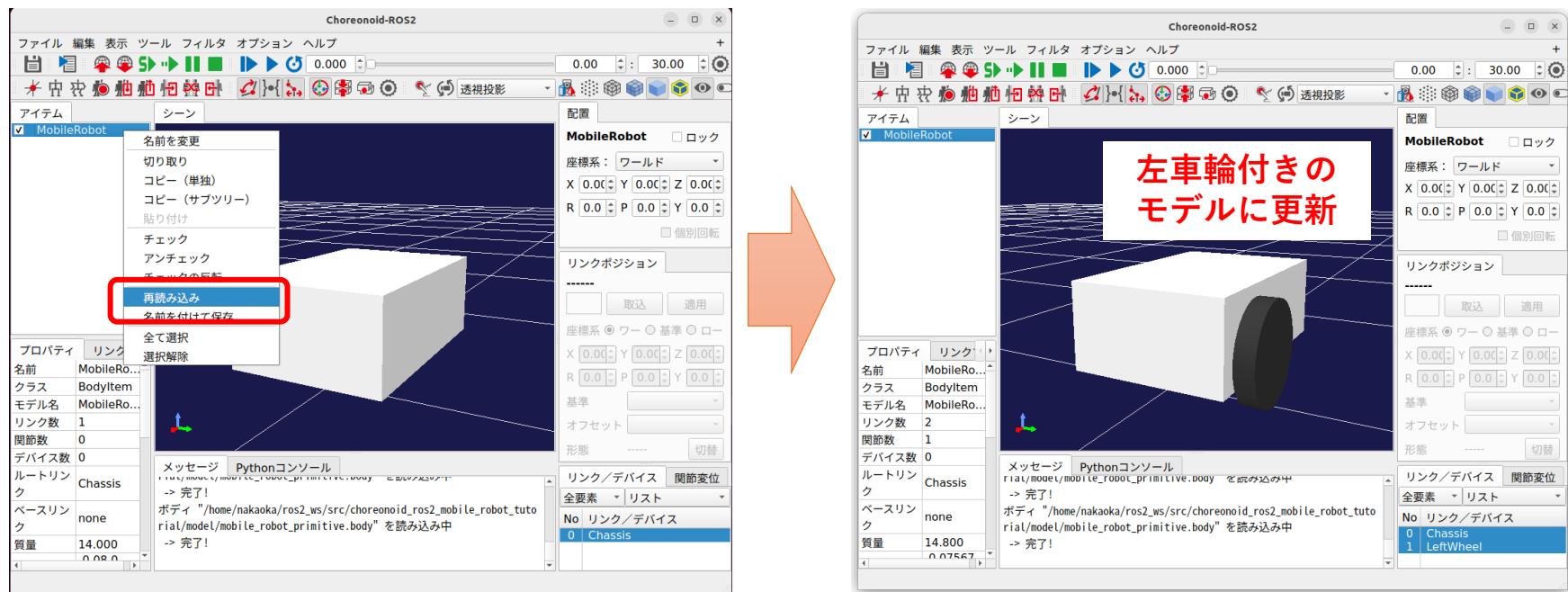
```
-  
  name: LeftWheel  
  parent: Chassis  
  translation: [ 0, 0.145, 0.076 ]  
  joint_type: revolute  
  joint_id: 0  
  joint_axis: [ 0, 1, 0 ]  
  center_of_mass: [ 0, 0, 0 ]  
  mass: 0.8  
  inertia: [ 0.0012, 0, 0,  
             0, 0.0023, 0,  
             0, 0, 0.0012 ]  
  material: Tire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

解説

```
-  
  name: LeftWheel  
  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 ]  
          # 親リンク名 (Chassisの子リンクとする)  
          # 親リンクからの相対位置 (親リンクローカル座標)  
          # 関節タイプ : 回転  
          # 関節ID (0から順番に降る)  
          # 関節軸 (リンクローカル座標)  
          # 接触マテリアル: Tire (床との摩擦係数が高くなる)  
          # 以下の記述内容にYAMLのアンカーをつけて右車輪で再利用する  
          # メッシュの分割数. 分割数を増やして滑らかにする  
          # アピアランス (表面の見た目を記述する)  
          # マテリアル (色等を記述する)  
          # 拡散反射光のR, G, B値 (各0.0~1.0)
```

再読み込み機能

- “MobileRobot” のアイテムを右クリックして「再読み込み」を実行
- もしくはアイテムを選択してCtrl + R



右車輪の追加

```
-  
  name: RightWheel  
  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  ※ 左車輪でつけたアンカーの内容をエイリアスとして取り込む
```

再読み込み (Ctrl + R) して更新

※ 完成版はお手本パッケージの”model/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
- アイテムのチェックで表示／非表示
- シーンビューの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

Part4

シミュレーションの実行

シミュレーションの準備

- 以下のプロジェクトアイテムを追加する
 - ワールド
 - AISTシミュレータ
- 以下のツリー構成（親子関係）にする

```
+ World
  MobileRobot
  AISTSimulator
```

手順1

- ・「アイテムビュー」上の選択を解除
- ・「ワールドアイテム」を生成
 - ・「ファイル」 - 「新規」 - 「ワールド」を実行
 - ・「生成」ボタンを押す

MobileRobot
World

- ・“MobileRobot”を“World”にドラッグする

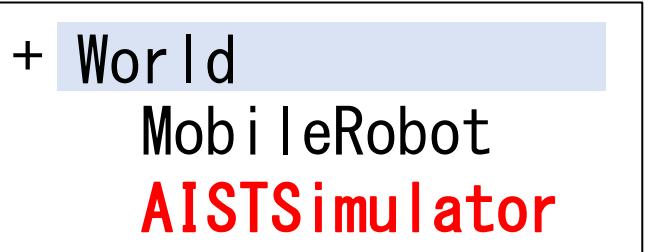
MobileRobot
World



+ World
MobileRobot

手順2

- “World” を選択
- 「AIStシミュレータアイテム」を生成
 - 「ファイル」 - 「新規」 - 「AIStシミュレータ」を実行
 - 「生成」 ボタンを押す



+ World
MobileRobot
AIStSimulator

記録モードの設定

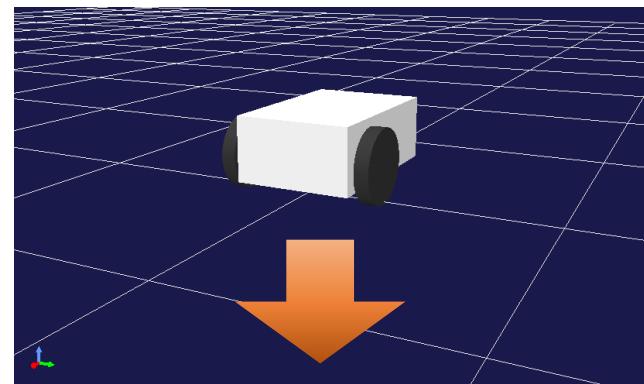
- AISSimulatorを選択
- 「プロパティ」ビューの「記録モード」を変更
 - 全て
 - シミュレーション開始時点からの全ての経過をログとして記録する
 - 末尾
 - 最新の状態から遡って「時間長」分の記録する
 - オフ
 - 記録を行わない
- 記録がある分はシミュレーション終了後もタイムバーで再生可能
- 「全て」にするとメモリを使い果たしてしまうリスクがある
- 本チュートリアルでは「末尾」を推奨

シミュレーションの実行

- ・シミュレーション開始ボタンを押す



- ・ロボットが落下する



- ・シミュレーション停止ボタンを押す



シミュレーション設定としての プロジェクトアイテム

- ワールドアイテム
 - ひとつの仮想世界に対応
 - 複数の仮想世界を持つ
 - 仮想世界ごとに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
- シミュレータアイテム
 - 物理計算の実装に対応
 - 物理計算（物理エンジン）の種類を切り替えられる
 - 物理計算に関する複数の設定を持つ

利用可能な物理エンジン

- AIST（産総研）エンジン（AISTシミュレータ）
 - 標準エンジン
- Open Dynamics Engine（ODEシミュレータ）
 - 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
- AGX Dynamics（AGXシミュレータ）
 - 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
- その他の利用可能な物理エンジン
 - NVIDIA PhysX, Bullet Physics, Springhead, Roki
 - ただし現状では対応が不十分

無重力シミュレーション

- “AISSimulator”を選択
- 「プロパティ」ビューで「重力加速度」に“0 0 0”を設定する
- シミュレーションを実行する
- シーンビューを編集モードにしてロボットをマウスでドラッグし、力をかけてみる

プロパティ	
デバイス状態の記録	True
干渉データの記録	False
コントローラスレッド	True
コントローラオプション	
シーンビュー編集モードをブロック	False
動力学モード	順動力学
積分モード	半陰オイラー
重力加速度	0 0 0
最小摩擦係数	0.0
最大摩擦係数	100.0
接触間引き距離	0.005
接触間引き深度	0.05
収束判定誤差基準	0.001
最大反復計算回数	25
接触補正深度	0.00025

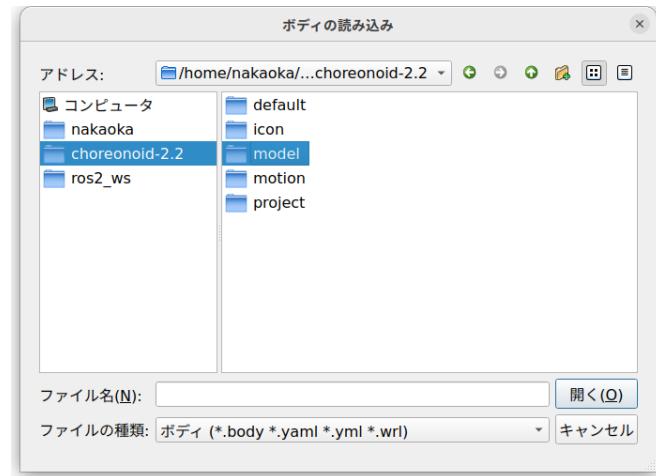
床の追加

- 重力加速度を元 (0 0 -9.8) に戻す
- 床モデルをWorldの子アイテムとして読み込む
 - choreonoid-2.x/model/misc/floor.body

ココ



※ 親子関係に注意！
(順番は前後してもOK)



- ロボットが落ちなくなる
- 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キーで指定可能
- 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

```
-  
  name: LeftWheel  
  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  
...
```

ここでは“Tire”
マテリアルを指定

タイムステップの設定

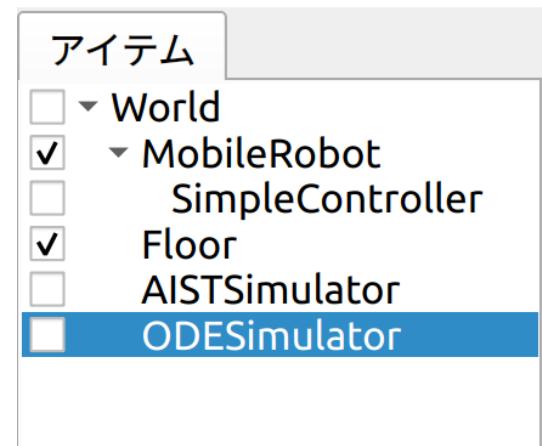
- ・シミュレーションの1コマあたりの時間
- ・デフォルトは0.001秒（1ミリ秒）
- ・シミュレーションの正確性・安定性とシミュレーション速度とのトレードオフ
- ・安定にシミュレーションできる範囲で必要に応じて増やしておく
- ・1ミリ秒の細かさであればほとんどのケースで安定

参考：ODEの利用

- ODEプラグインをビルドする

```
cd ~/ros2_ws  
colcon build --packages-select choreonoid  
--cmake-args -DBUILD_ODE_PLUGIN=ON --symlink-install
```

- 「ODEシミュレータ」アイテムを生成
- “ODESimulator”を選択して
シミュレーションを実行する



Part5

メッシュファイルの利用

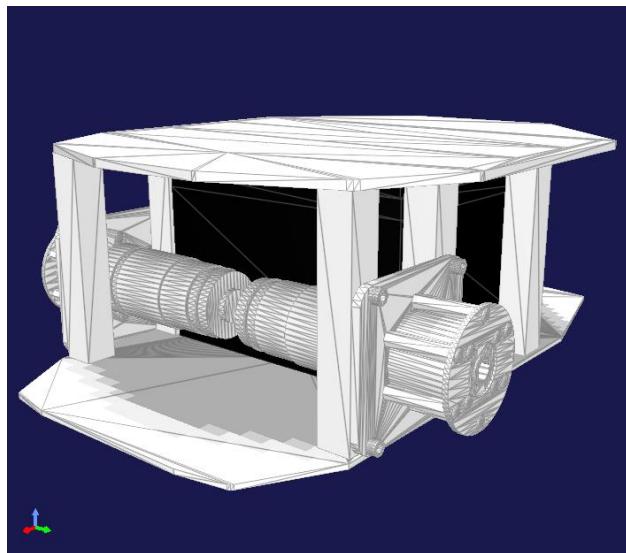
任意形状の実現

- ・モデリングツールやCADツールで形状を作成
- ・形状をメッシュファイルとしてエクスポート
- ・メッシュファイルをロボットモデルに取り込む

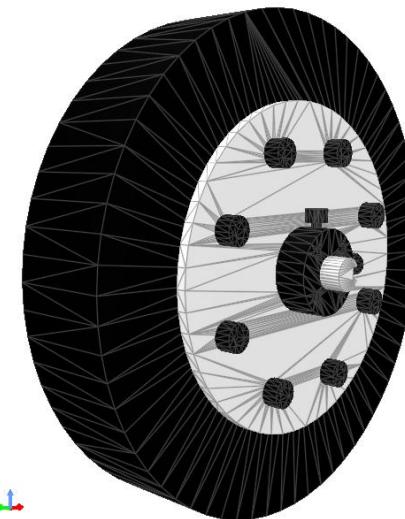
利用可能なメッシュファイル

- ネイティブでサポートしている形式
 - STL
 - OBJ
 - VRML97 (拡張子: wrl)
- Assimpライブラリでサポート
 - Collada (拡張子: dae)
 - その他多数

メガローバーのメッシュファイル



vmega_body.dae



vmega_wheel.dae

お手本パッケージの meshes 以下に格納

- meshes/vmega_body.dae
- meshes/vmega_wheel.dae
- meshes/r5.png (テクスチャ画像)

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

モデルファイルの修正（車体部分）

```
-  
  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"
```

(**uri: "package://my_mobile_robot/meshes/vmega_body.dae"** と書いてもOK)

メッシュファイル式をお手本パッケージから “my_mobile_robot” の “meshes” ディレクトリ（新たに作成）にコピーしておきます

モデルファイルの修正（ホイール部分）

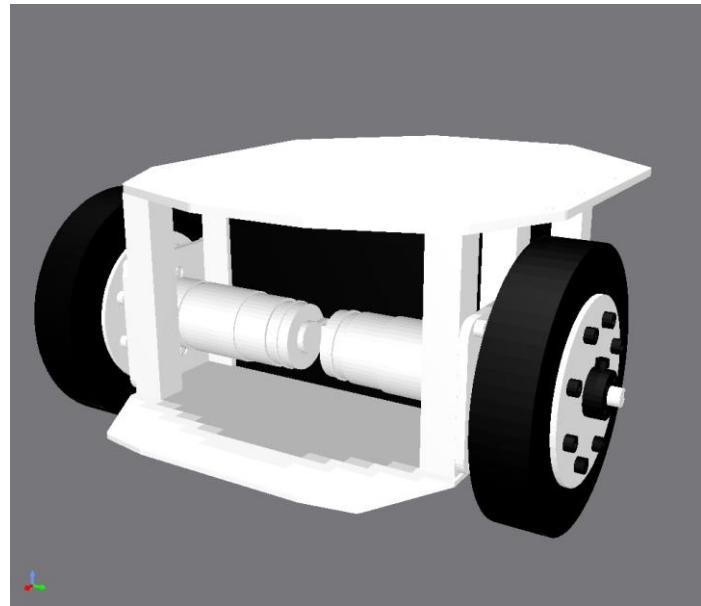
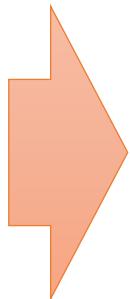
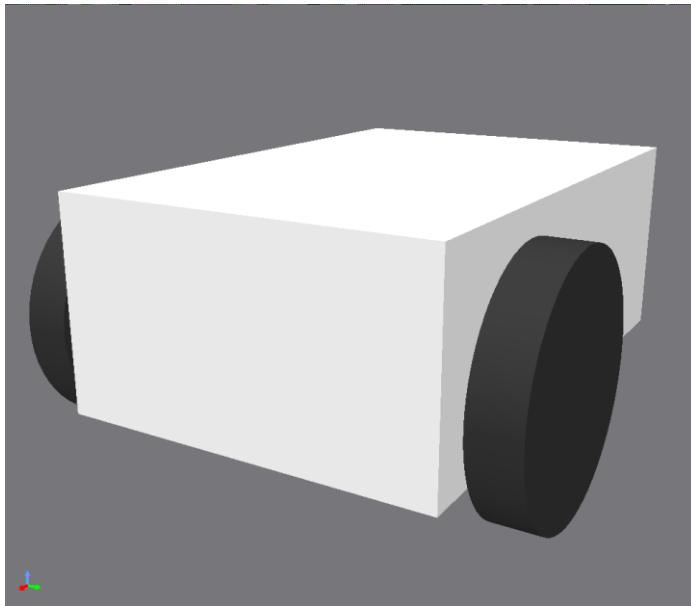
```
-  
  name: LeftWheel  
  parent: Chassis  
  translation: [ 0, 0.145, 0.076 ]  
  joint_type: revolute  
  joint_id: 0  
  joint_axis: [ 0, 1, 0 ]  
  center_of_mass: [ 0, 0, 0 ]  
  mass: 0.8  
  inertia: [ 0.0012, 0, 0,  
             0, 0.0023, 0,  
             0, 0, 0.0012 ]  
  material: Tire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

```
-  
  name: RightWheel  
  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"
```

※ Z軸まわりに180度回転で左右反転

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

修正結果



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、影の描画をオフに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

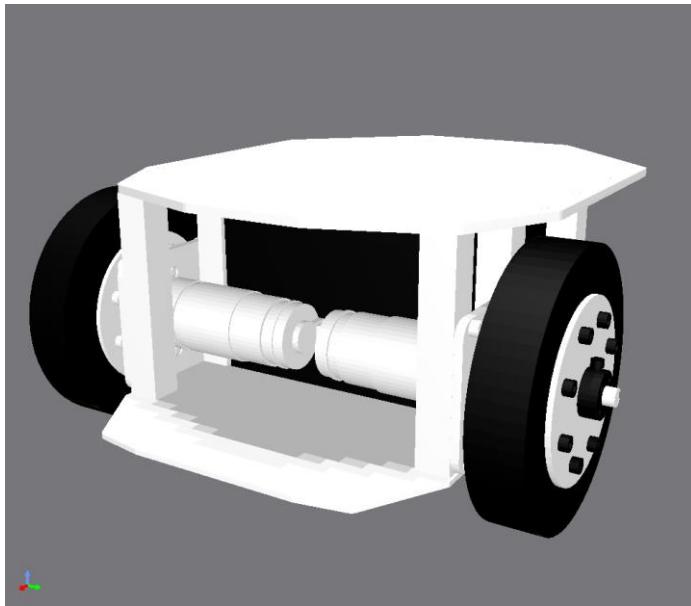
Part6

表示用モデルと干渉検出用モデルの切り分け

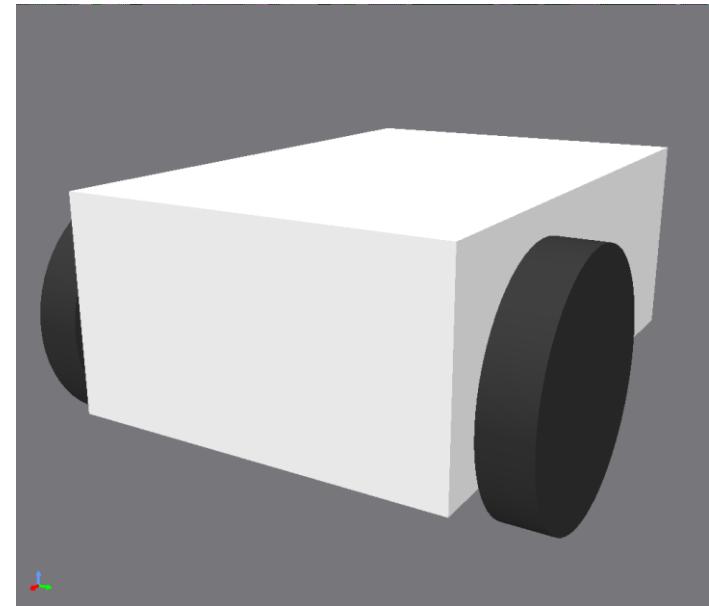
モデルの切り分け

- 表示用モデル
 - 3Dグラフィックとして描画
 - 視覚センサのシミュレーションでも使用される
 - 複雑なモデルにしても物理計算には影響しない
- 干渉検出用モデル
 - 干渉検出の計算で使用
 - 干渉計算の負荷や接触反力計算の安定性に影響
 - プリミティブの組み合わせやシンプルなメッシュとすることで、負荷の軽減や安定性の向上を実現

切り分けの例



表示用モデル



干渉検出用モデル

Bodyファイルでの切り分け

各リンクの記述

...

elements:

-

 type: Visual

 elements:

 表示用モデルの形状を記述

-

 type: Collision

 elements:

 干渉検出用モデルの形状を記述

車体の記述

```
-  
  name: Chassis  
  ...  
  elements:  
    -  
      type: Visual  
      elements:  
        -  
          type: Resource  
          uri: "../meshes/vmega_body.dae"  
        -  
          type: Collision  
          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
```

表示用モデル

干渉検出用モデル

左車輪の記述

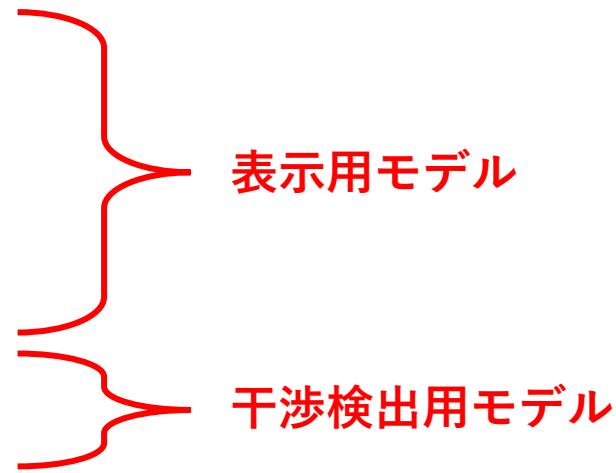
```
-  
  name: LeftWheel  
  ...  
  elements:  
    -  
      type: Visual  
      elements:  
        -  
          type: Resource  
          uri: "../meshes/vmega_wheel.dae"  
          rotation: [ 0, 0, 1, 180 ]  
        -  
          type: Collision  
          elements:  
            - &TireShape  
            type: Shape  
            geometry:  
              type: Cylinder  
              height: 0.03  
              radius: 0.076  
              division_number: 60
```



表示用モデル

右車輪の記述

```
-  
  name: RightWheel  
  ...  
  elements:  
    -  
      type: Visual  
      elements:  
        -  
          type: Resource  
          uri: "../meshes/vmega_wheel.dae"  
        -  
          type: Collision  
          elements:  
            - *TireShape
```

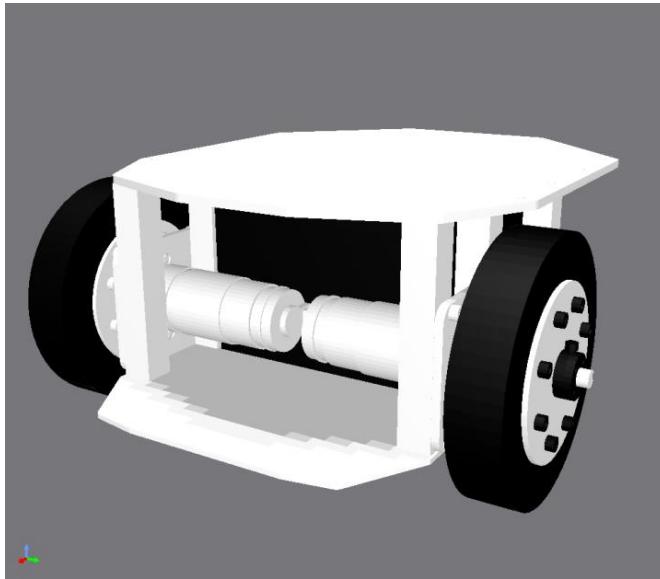


※ 完成版はお手本パッケージの “model/mobile_robot.body”

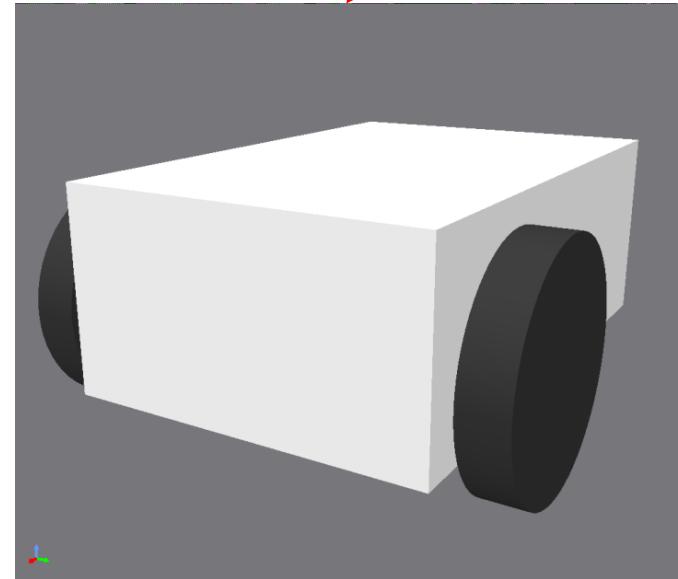
各モデルの確認

シーンバー

表示用モデル 干渉検出用モデル
を表示 を表示



表示用モデル



干渉検出用モデル

Part7

制御プログラム（コントローラ）の作成

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
 - ros2_controlのコントローラを利用する？
 - ROS 1用のros_controlについては対応していたが、ROS 2用のros2_controlについては未対応

コントローラのアイテム

- 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rclcppライブラリで実現可能
 - rclcpp = ROSクライアントライブラリのC++版

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」 - 「新規」 -
「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手順

- src/MobileRobotDriveTester.cppを作成
- package.xmlを編集
- ビルド用のCMakeLists.txtの修正・追加
- “colcon 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("LeftWheel");
    wheels[1] = body->joint("RightWheel");
    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;
}
```

※ 完成版はお手本プロジェクトの
“src/MobileRobotDriveTester.cpp”

package.xmlの編集

my_mobile_robot/package.xmlに以下を追記する

```
<buildtool_depend>ament_cmake</buildtool_depend>
```

```
<depend>choreonoid_ros</depend> ← 追加
```

```
...
```

これにより、(ROS対応版の) Choreonoidに依存するパッケージであることを示すことができる

トップのCMakeLists.txt

パッケージ初期化時に生成された雛形に赤字部分を追加する

```
...
# find dependencies
find_package(ament_cmake REQUIRED)
# uncomment the following section in order to fill in
# further dependencies manually.
# find_package(<dependency> REQUIRED)

find_package(choreonoid REQUIRED)

set(CMAKE_CXX_STANDARD ${CHOREONOID_CXX_STANDARD})
include_directories(${CHOREONOID_INCLUDE_DIRS})
link_directories(${CHOREONOID_LIBRARY_DIRS})

add_subdirectory(src)

...
```

これにより、Choreonoidのライブラリを使用してコントローラを実装できるようになる

srcのCMakeLists.txt

- src/CMakeLists.txtを以下の内容で新規作成

```
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

- colcon build コマンドでビルドする

```
cd ~/ros2_ws  
colcon build --symlink-install
```

※ ワークスペースのトップディレクトリである “~/ros2_ws” で実行する必要があります

- MyMobileRobotDriveTester.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

“~/ros2_ws/install/my_mobile_robot/lib/choreonoid-2.x/simplecontroller”

colcon build の処理

- ワークスペースのsrc以下のソースファイルをビルドしたバイナリファイルをinstallディレクトリに生成する
 - 実行時に実際に使われるのはinstall以下のファイル
- ビルドの必要のないファイルはコピーする
 - “--symlink-install” オプションを付けると、コピーの代わりにシンボリックリンクをはる
 - src以下のファイルを変更した場合、colcon buildを再度実行しなくても変更が有効となる
 - このオプションを付けていなかった場合は、再度colcon buildを実行してコピーしなおす必要がある

初回ビルド後の設定の反映

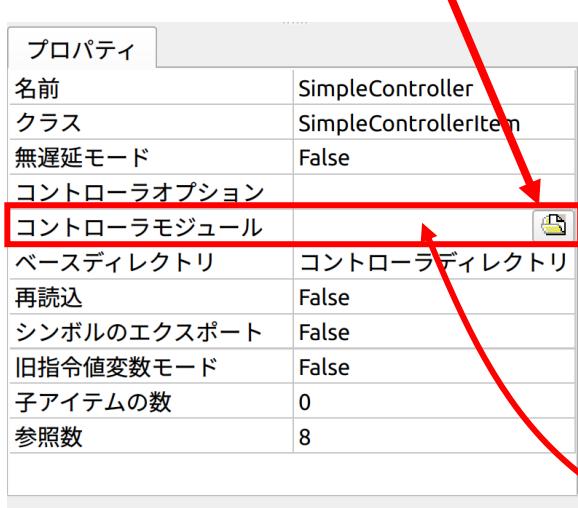
- 今回、my_mobile_robotパッケージのビルドを初めて実行
 - ワークスペースにmy_mobile_robotが追加されたことを認識させる必要がある
1. ワークスペースセットアップスクリプトを再度取り込む
 - 端末を起動しなおすか、以下を実行する

```
source ~/ros2_ws/install/setup.bash
```
 2. Choreonoidを起動しなおす

コントローラの本体の設定

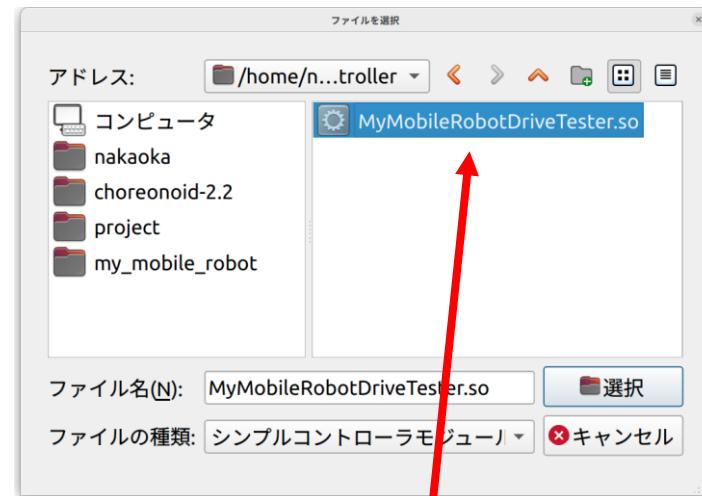
- “SimpleController”アイテムを選択し、「コントローラモジュール」プロパティで設定する

「コントローラモジュールの入力欄をダブルクリックして、このボタンを押す



“MyMobileRobotDriveTester”がここに設定される（直接入力してもよい）

ファイル選択ダイアログで “~/ros2_ws/install/my_mobile_robot/lib/choreonoid-2.x/simplecontroller” を開く

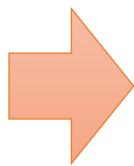


“MyMobileRobotDriveTester.so”を選択

補足：アイテム名の修正

- ・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
- ・アイテムの右クリックメニューで「名前を変更」

+ World
+ MobileRobot
SimpleController
Floor
AISTSimulator



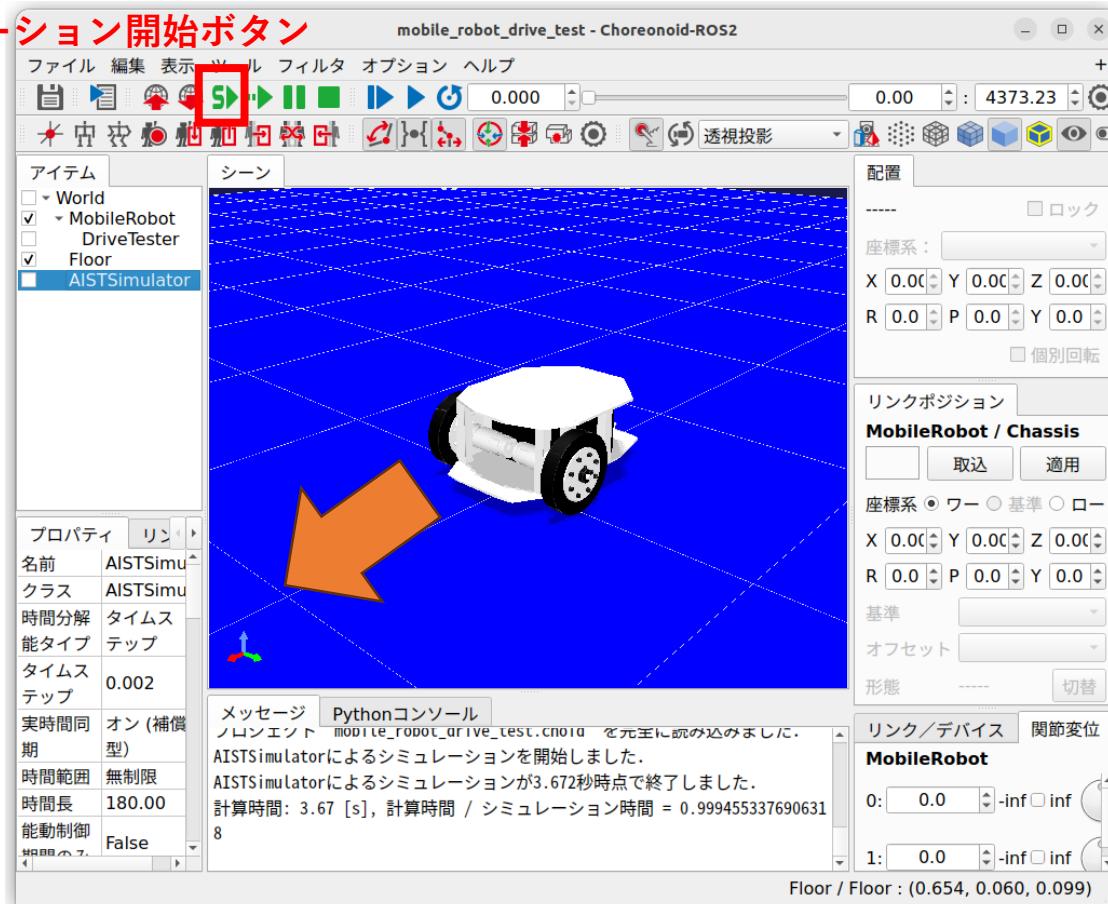
+ World
+ MobileRobot
DriveTester
Floor
AISTSimulator

プロジェクトファイルの保存

- 「ファイル」 - 「プロジェクトに名前を付けて保存」
- “project” ディレクトリに
“mobile_robot_drive_test.cnoid” として保存しておく

シミュレーションの実行

シミュレーション開始ボタン



シミュレーション開始ボタンを押すと、
ロボットが前進する（途中でスリップもする）

ソース解説 (1/2)

```
#include <cnoid/SimpleController> シンプルコントローラを利用するためのヘッダ
SimpleControllerクラスを継承して実装する
class MobileRobotDriveTester : public cnoid::SimpleController
{
public:    この関数をオーバーライドして初期化処理を実装する
    virtual bool initialize(cnoid::SimpleControllerIO* io) override;
    virtual bool control() override;
        この関数をオーバーライドして制御処理を実装する
private:
    cnoid::Link* wheels[2];    左右の車輪に対応するリンクオブジェクトを格納する変数
};                                         このオブジェクトを介して制御の入出力を行う

CNOID_IMPLEMENT_SIMPLE_CONTROLLER_FACTORY(MobileRobotDriveTester)
```

共有ライブラリとして生成されるコントローラモジュールから、コントローラのインスタンスを生成するエントリ関数を定義するマクロ。括弧の中にコントローラのクラス名を記述する

ソース解説 (2/2)

初期化処理の実装

引数ioを介して入出力に必要な設定やオブジェクトの取得を行う

```
bool MobileRobotDriveTester::initialize(cnoid::SimpleControllerIO* io)
{
    auto body = io->body(); 入出力用のBodyオブジェクトの取得。ロボット全体の情報に対応
    wheels[0] = body->joint("LeftWheel"); 左車輪のリンク（関節）の取得
    wheels[1] = body->joint("RightWheel"); 右車輪のリンク（関節）の取得
    for(int i=0; i < 2; ++i) {
        auto wheel = wheels[i];
        wheel->setActuationMode(JointTorque); 関節トルクを指令値にする
        io->enableOutput(wheel, JointTorque); 関節トルクを出力する
    }
    return true; 初期化成功時はtrueを返す
}

制御処理の実装。毎制御ループ（通常シミュレーションのタイムステップ）ごとに実行される
bool MobileRobotDriveTester::control()
{
    wheels[0]->u() = 1.0; 左車輪に 1.0 [N・m]のトルク指令を出力する。(uは関節トルクを表す変数)
    wheels[1]->u() = 1.0; 右車輪に 1.0 [N・m]のトルク指令を出力する
    return true; 制御を継続するときはtrueを返す。falseを返すとそれ以降制御をしなくなる
}
```

お手本パッケージのファイル

- Part6に対応するファイル
 - コントローラのソースファイル
 - src/MobileRobotDriveTester.cpp
 - プロジェクトファイル
 - project/mobile_robot_drive_test.cnoid
- 以下のコマンドでプロジェクトを起動

```
cd ~/ros2_ws/src/choreonoid_ros2_mobile_robot_tutorial  
ros2 run choreonoid_ros choreonoid project/mobile_robot_drive_test.cnoid
```

ビルドスクリプトの作成

- 以下のスクリプトを~/ros2_ws/build.shとして作成しておく

```
#!/bin/sh  
cd ~/ros2_ws  
colcon build --symlink-install --packages-select my_mobile_robot
```

my_mobile_robotだけビルドする

- 実行権限を付与

```
chmod +x build.sh
```

- 以下でビルドを実行

```
~/ros2_ws/build.sh
```

Part8

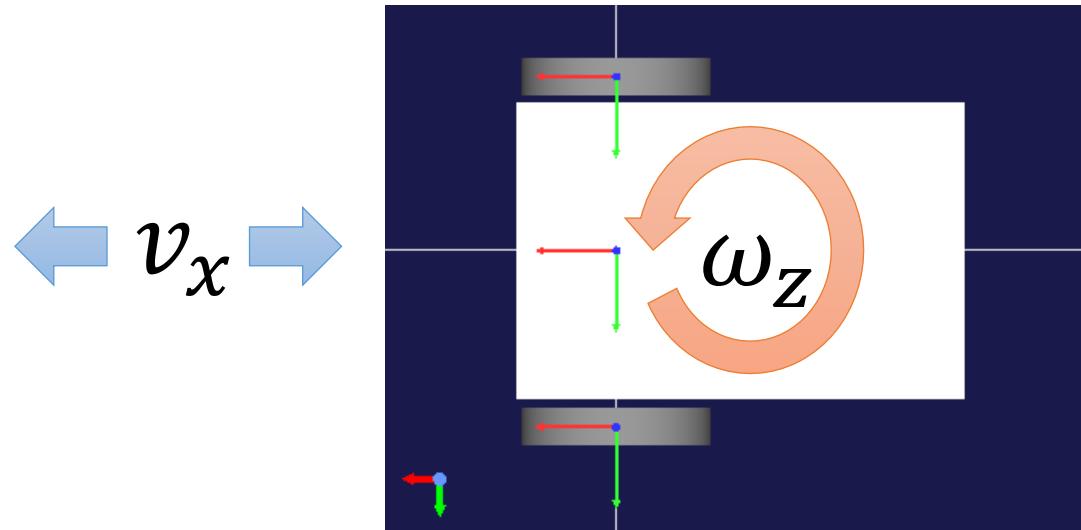
ROS通信を用いた制御

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

```
ros2 interface list
```

Messages: 以下に表示される

- メッセージ型の内容を表示

```
ros2 interface show 型名
```

ROSのTwistメッセージ型

- geometry_msgs/msg/Twist
- 速度 + 角速度

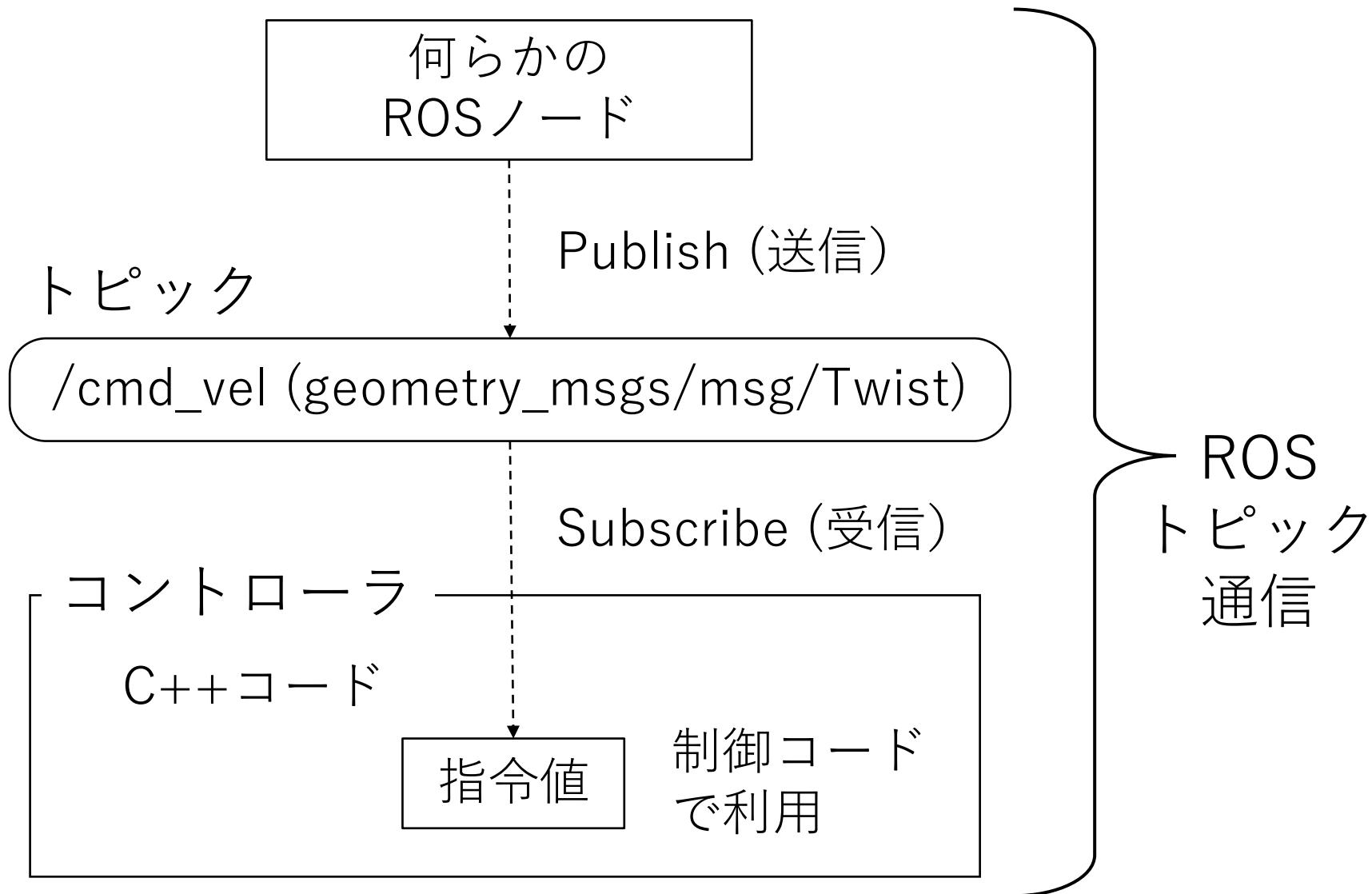
```
ros2 interface show geometry_msgs/msg/Twist
```

```
Vector3 linear
  float64 x
  float64 y
  float64 z
```

※ “Vector3” は
“geometry_msgs/msg/Vector3” 型

```
Vector3 angular
  float64 x
  float64 y
  float64 z
```

Twistメッセージの入力



ROS通信の種類

	概要	用途
トピック通信	一方向・非同期通信	センサーデータの送受信等
サービス通信	双方向・同期通信	機能の実行、切替等
アクション通信	複数のサービス、トピックを組み合わせて、まとめた動作を処理	タスクやナビゲーションの実行等
パラメータ通信	ノードのパラメータを取得・設定	各種設定

※ 本チュートリアルでは主にトピック通信を扱う

コントローラの作成と導入

- package.xmlを編集して、他に必要なパッケージの記述する
- MobileRobotDriveTester.cppを改良して、MobileRobotDriveController.cppを作成
- rclcpp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

package.xmlの編集

my_mobile_robot/package.xmlに以下を追記する

```
<buildtool_depend>ament_cmake</buildtool_depend>

<depend>choreonoid_ros</depend>
<depend>rclcpp</depend>
<depend>geometry_msgs</depend>

<exec_depend>rqt_robot_steering</exec_depend>
<exec_depend>joy</exec_depend>
<exec_depend>joy_teleop</exec_depend>
<exec_depend>image_transport_plugins</exec_depend>
<exec_depend>xacro</exec_depend>

...
```

依存パッケージのインストール

rosdepを使用して必要なパッケージをインストールする

```
rosdep install -y --from-paths ~/ros2_ws/src --ignore-src
```

package.xmlに記述した

- rqt_robot_steering
- joy
- joy_teleop
- image_transport_plugins
- xacro

といったパッケージが（まだインストールされていなければ）インストールされる

依存パッケージの概要

パッケージ	概要	本チュートリアルで必要となるPart
rclcpp	ROSクライアントライブラリのC++版	Part8 (本パート)
geometry_msgs	基本的なROSメッセージ型	
rqt_robot_steering	TwistメッセージをpublishするGUIツール	
joy	ゲームパッドデバイスの状態を読み込んでJoyメッセージとしてpublishするノード	Part10: ゲームパッドによる操作
joy_teleop	JoyメッセージをTwistメッセージに変換するノード	
image_transport_plugins	画像データをpublishするimage_transport機能のプラグイン（通信時の画像圧縮等に対応）	Part15: ROS通信を用いた状態の出力
xacro	ROSのモデルファイルのテキストを処理するコマンド	Part17: Rvizによる可視化

MobileRobotDriverController.cppの準備

- “my_mobile_robot” のsrcディレクトリでソースファイルをコピーして、エディタで開く

```
cd ~/ros2_ws/src/my_mobile_robot/src  
cp MobileRobotDriveTester.cpp MobileRobotDriverController.cpp  
gedit MobileRobotDriverController.cpp
```

rclcppライブラリ

- ROSクライアントライブラリのC++版
 - C++でROSの通信処理などを利用できる
 - ROS 1のroscppライブラリと同じ位置づけ
 - ただしAPIは異なる
 - コントローラの実装に利用する
-
- 公式ドキュメントのチュートリアル
<https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries.html>

MobileRobotDriveController.cpp (1/4)

```
#include <cnoid/SimpleController>
#include <rclcpp/rclcpp.hpp>
#include <geometry_msgs/msg/twist.hpp>
#include <memory>

class MobileRobotDriveController : public cnoid::SimpleController
{
public:
    virtual bool configure(cnoid::SimpleControllerConfig* config) override;
    virtual bool initialize(cnoid::SimpleControllerIO* io) override;
    virtual bool control() override;

private:
    cnoid::Link* wheels[2];
    rclcpp::Node::SharedPtr node;
    rclcpp::Subscription<geometry_msgs::msg::Twist>::SharedPtr subscription;
    geometry_msgs::msg::Twist command;
    rclcpp::executors::StaticSingleThreadedExecutor::UniquePtr executor;
};

CNOID_IMPLEMENT_SIMPLE_CONTROLLER_FACTORY(MobileRobotDriveController)
```

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

MobileRobotDriveController.cpp (2/4)

```
bool MobileRobotDriveController::configure(cnoid::SimpleControllerConfig* config)
{
    node = std::make_shared<rclcpp::Node>(config->controllerName());

    subscription = node->create_subscription<geometry_msgs::msg::Twist>(
        "/cmd_vel", 1,
        [this](const geometry_msgs::msg::Twist::SharedPtr msg) {
            command = *msg;
        });

    executor = std::make_unique<rclcpp::executors::StaticSingleThreadedExecutor>();
    executor->add_node(node);

    return true;
}
```

MobileRobotDriveController.cpp (3/4)

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

MobileRobotDriveController.cpp (4/4)

```
bool MobileRobotDriveController::control()
{
    constexpr double wheelRadius = 0.076;
    constexpr double halfAxeWidth = 0.145;
    constexpr double kd = 0.5;
    double dq_target[2];

    executor->spin_some();

    double dq_x = command.linear.x / wheelRadius;
    double dq_yaw = command.angular.z * halfAxeWidth / 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;
}
```

トップのCMakeLists.txt

赤字部分を追加する

```
...
# uncomment the following section in order to fill in
# further dependencies manually.
# find_package(<dependency> REQUIRED)

find_package(rclcpp REQUIRED)
find_package(geometry_msgs REQUIRED)
find_package(choreonoid REQUIRED)

set(CMAKE_CXX_STANDARD ${CHOREONOID_CXX_STANDARD})
include_directories(${CHOREONOID_INCLUDE_DIRS})
link_directories(${CHOREONOID_LIBRARY_DIRS})

add_subdirectory(src)

...
```

これにより、rclcppライブラリとgeometry_msgsのメッセージ型を使用できるようになる

srcのCMakeLists.txt

- src/CMakeLists.txtを以下のように編集

```
choreonoid_add_simple_controller(  
    MyMobileRobotDriveTester MobileRobotDriveTester.cpp)  
  
choreonoid_add_simple_controller(  
    MyMobileRobotDriveController MobileRobotDriveController.cpp)  
  
ament_target_dependencies(  
    MyMobileRobotDriveController rclcpp geometry_msgs)
```

※ 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

ビルト

- Part6で作成したbuild.shスクリプトでビルトを行う

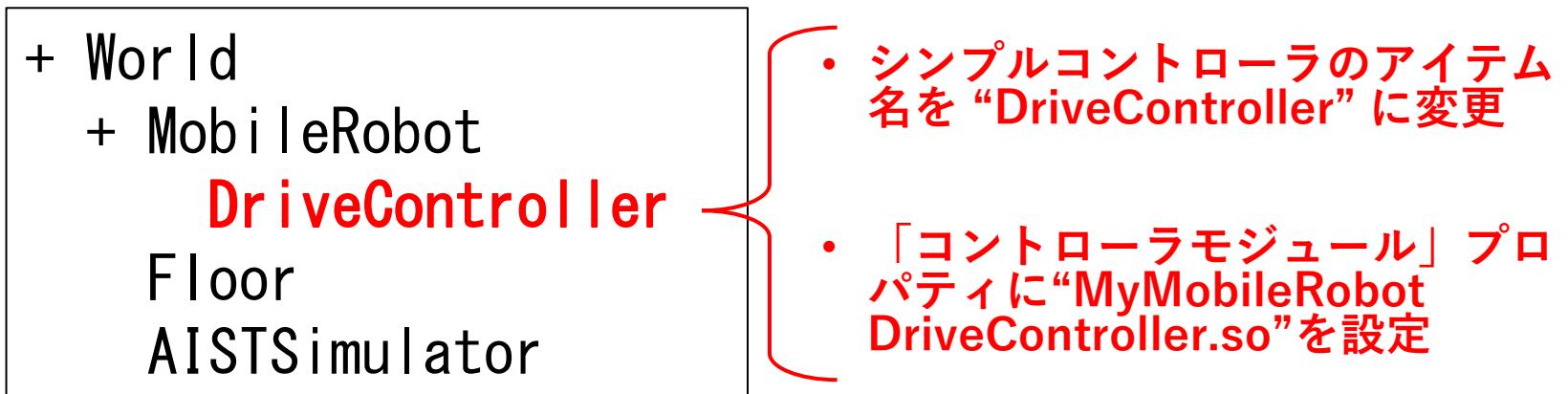
```
cd ~/ros2_ws  
./build.sh
```

“~/ros2_ws/install/my_mobile_robot/lib/choreonoid-2.x/simplecontroller” に
“MyMobileRobotDriveController.so” が生成される

プロジェクトの構成

Part6で作成したプロジェクト

“project/mobile_robot_drive_test.cnoid” を修正する



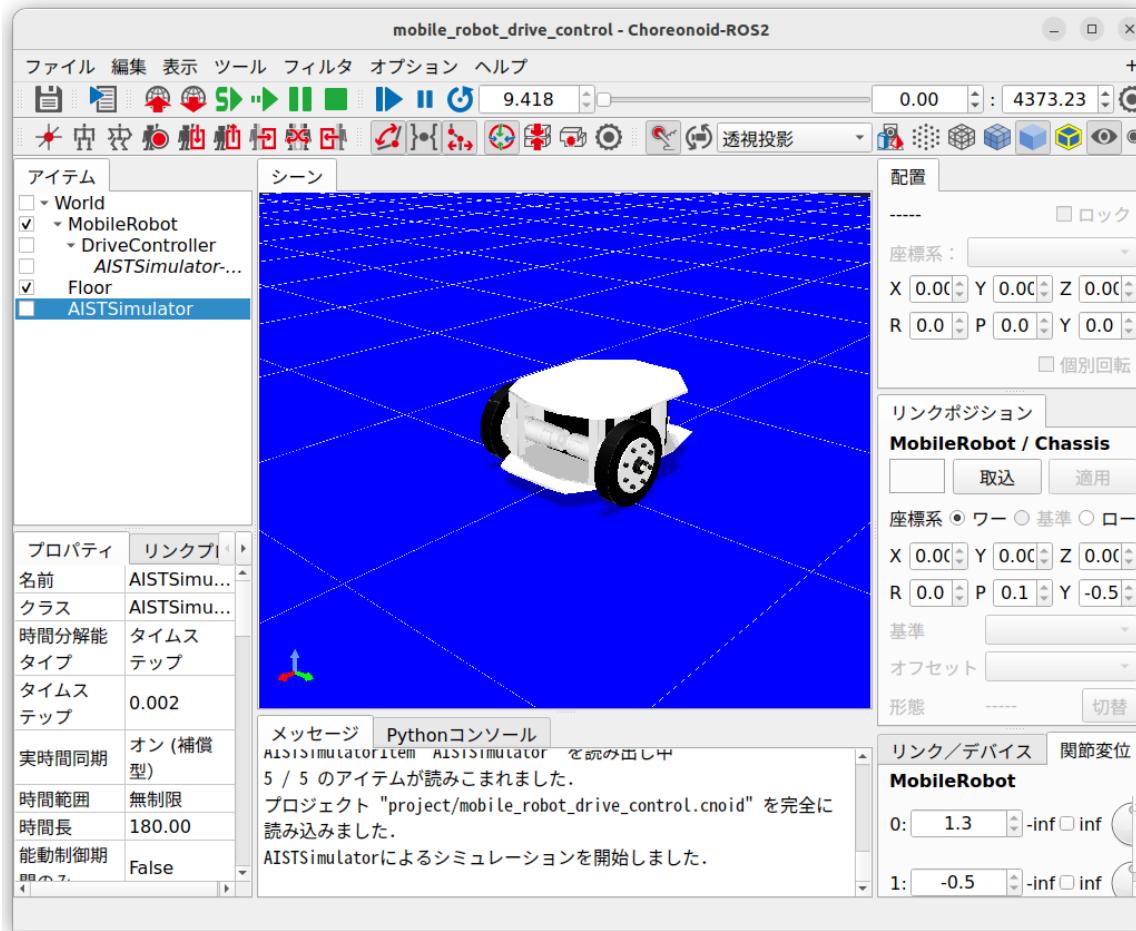
プロジェクトを

“project/mobile_robot_drive_control.cnoid”

として保存する

※ 同名のプロジェクトファイルがお手本パッケージにもあります

シミュレーションの実行



シミュレーション開始しても、
そのままではロボットは動かない

トピックの確認

利用可能なトピックを表示

```
ros2 topic list
```

/cmd_velが表示されているか？

内容の確認

```
ros2 topic info /cmd_vel
```

※ Choreonoidを起動したのとは別の端末上で実行します

※ Ubuntu標準の端末であれば“Shift + Ctrl + N”や“Shift + Ctrl + T”で
端末を追加できます

/cmd_vel トピックの Publish

- 様々な手段がある
 - “ros2 topic pub” コマンド
 - rqt_robot_steeringツール
 - ゲームパッド + joyノード

ros2 topic コマンドによる操作

```
ros2 topic pub /cmd_vel geometry_msgs/msg/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

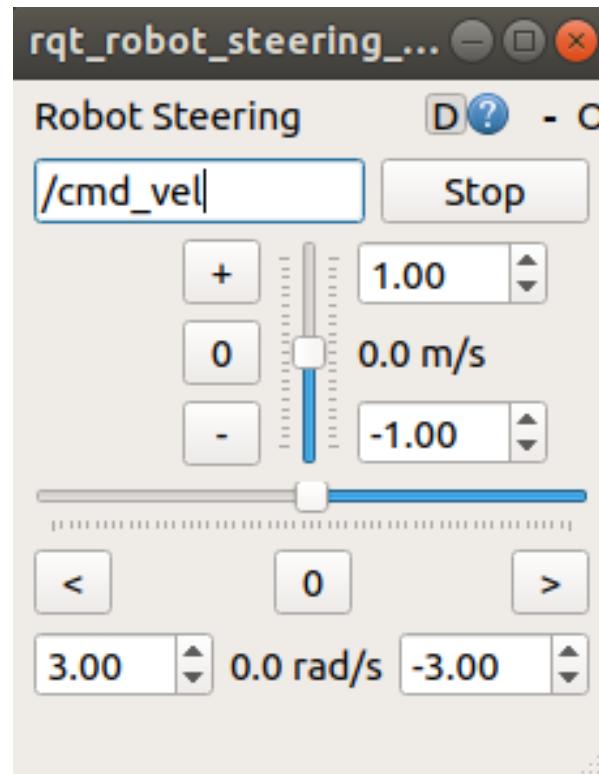
※ コロン(:)の後にスペースが必要！

publishされている内容を確認

```
ros2 topic echo /cmd_vel
```

rqt_robot_steering による操作

```
ros2 run rqt_robot_steering rqt_robot_steering
```

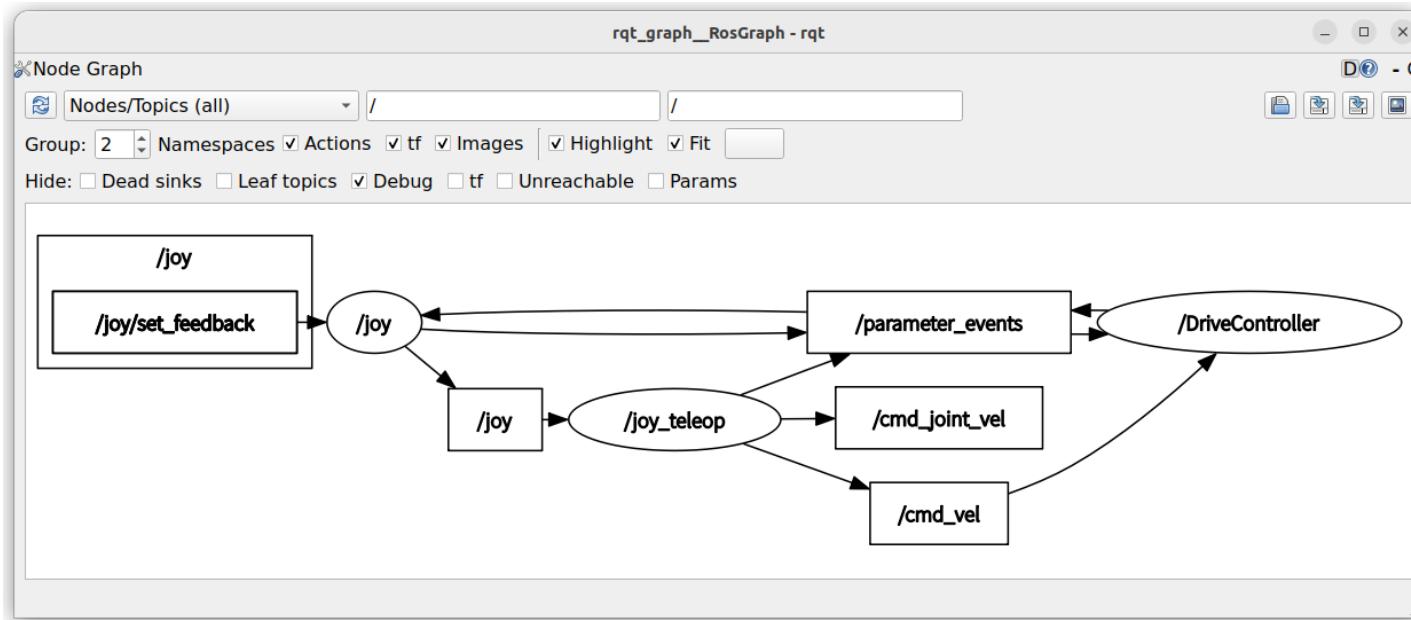


ノード接続グラフの表示

- rqt_graphを用いることで、ノード、トピック等の接続関係をグラフ表示できる

以下を実行

```
rqt_graph
```



Part9

ROS通信の実装について

ノード実装におけるROS1とROS2の違い

	ROS 1	ROS 2
プロセスあたり ノード数	単一	複数可
ノードのスピ ループ処理	単一	任意のExecutorで処理 可能
Choreonoid上で 直接動作するコン トローラの実装	單一ノード、スピループを 共有 (コントローラの数や通信回 数が増えると負荷分散ができ ず重くなる)	コントローラごとに独 自のノード、スピ ループを持たせること を推奨

ソース解説 (1/4)

```
#include <cnoid/SimpleController>
#include <rclcpp/rclcpp.hpp>          rclcppを利用するためのヘッダ
#include <geometry_msgs/msg/twist.hpp> twistメッセージ型を利用するためのヘッダ
#include <memory>                   スマートポインタを利用するためのヘッダ

class MobileRobotDriveController : public cnoid::SimpleController
{
public:    コントローラ生成時に実行される処理を実装する関数 (initializeは制御開始時)
    virtual bool configure(cnoid::SimpleControllerConfig* config) override;
    virtual bool initialize(cnoid::SimpleControllerIO* io) override;
    virtual bool control() override;

private:
    cnoid::Link* wheels[2];
    rclcpp::Node::SharedPtr node;    コントローラのROSノードを格納する変数
    rclcpp::Subscription<geometry_msgs::msg::Twist>::SharedPtr subscription; subscriberを格納
    geometry_msgs::msg::Twist command; Twist型の指令値を格納
    rclcpp::executors::StaticSingleThreadedExecutor::UniquePtr executor;
};                                     subscribe処理を回すためのexecutorを格納

CNOID_IMPLEMENT_SIMPLE_CONTROLLER_FACTORY(MobileRobotDriveController)
```

ソース解説 (2/4)

シンプルコントローラアイテム生成時に実行する処理を実装。ここに実装した処理はシミュレーション開始前でも有効になる。サブスクライバをここで生成しておくことで、シミュレーション開始前に/cmd_velトピックにアクセスできるようになる。

```
bool MobileRobotDriveController::configure(cnoid::SimpleControllerConfig* config)
{
    サブスクライバのためのROSノードを生成する。ROS 2では同一プロセスに複数のノードを生成可能。
    node = std::make_shared<rclcpp::Node>(config->controllerName());
    サブスクライバを生成する。メッセージ型はTwist。
    subscription = node->create_subscription<geometry_msgs::msg::Twist>(
        "/cmd_vel", 1, トピック名を"cmd_vel"とする
        [this](const geometry_msgs::msg::Twist::SharedPtr msg) {
            command = *msg; サブスクライブした時のコールバック関数。
        }); パブリッシュされたTwistの値をcommand変数にコピーする。
    サブスクライブ処理のためには、Executorが必要となるので生成する
    executor = std::make_unique<rclcpp::executors::StaticSingleThreadedExecutor>();
    executor->add_node(node); executorにnodeを追加すると、そのノードの処理を行うようになる

    return true; 処理に成功したらtrueを返す
}
```

ソース解説 (3/4)

```
bool MobileRobotDriveController::initialize(cnoid::SimpleControllerIO* io)
{
    auto body = io->body();
    wheels[0] = body->joint("LeftWheel");
    wheels[1] = body->joint("RightWheel");
    for(int i=0; i < 2; ++i){
        auto wheel = wheels[i];
        wheel->setActuationMode(JointTorque);
        io->enableInput(wheel, JointVelocity); 速度制御をするため、現在速度の入力も  
有効化する
        io->enableOutput(wheel, JointTorque);
    }
    return true;
}
```

ソース解説 (4/4)

```
bool MobileRobotDriveController::control()
{
    constexpr double wheelRadius = 0.076;          車輪の半径。速度の算出に利用
    constexpr double halfAxeWidth = 0.145;         車軸の長さの半分。角速度の算出に利用
    constexpr double kd = 0.5;                      速度差分に対するゲイン値
    double dq_target[2];                            速度目標値

    executor->spin_some();                         executorの処理を回す関数。とりあえずこの制御関数の中でまわしてみる。サブスクライブ時のコールバック関数はここで呼ばれることになる。
    double dq_x = command.linear.x / wheelRadius;   速度指令値を車軸角速度に変換
    double dq_yaw = command.angular.z * halfAxeWidth / wheelRadius;
    dq_target[0] = dq_x - dq_yaw;                   車軸角速度目標値
    dq_target[1] = dq_x + dq_yaw;                   角速度指令値を車軸角速度に変換
                                                の計算

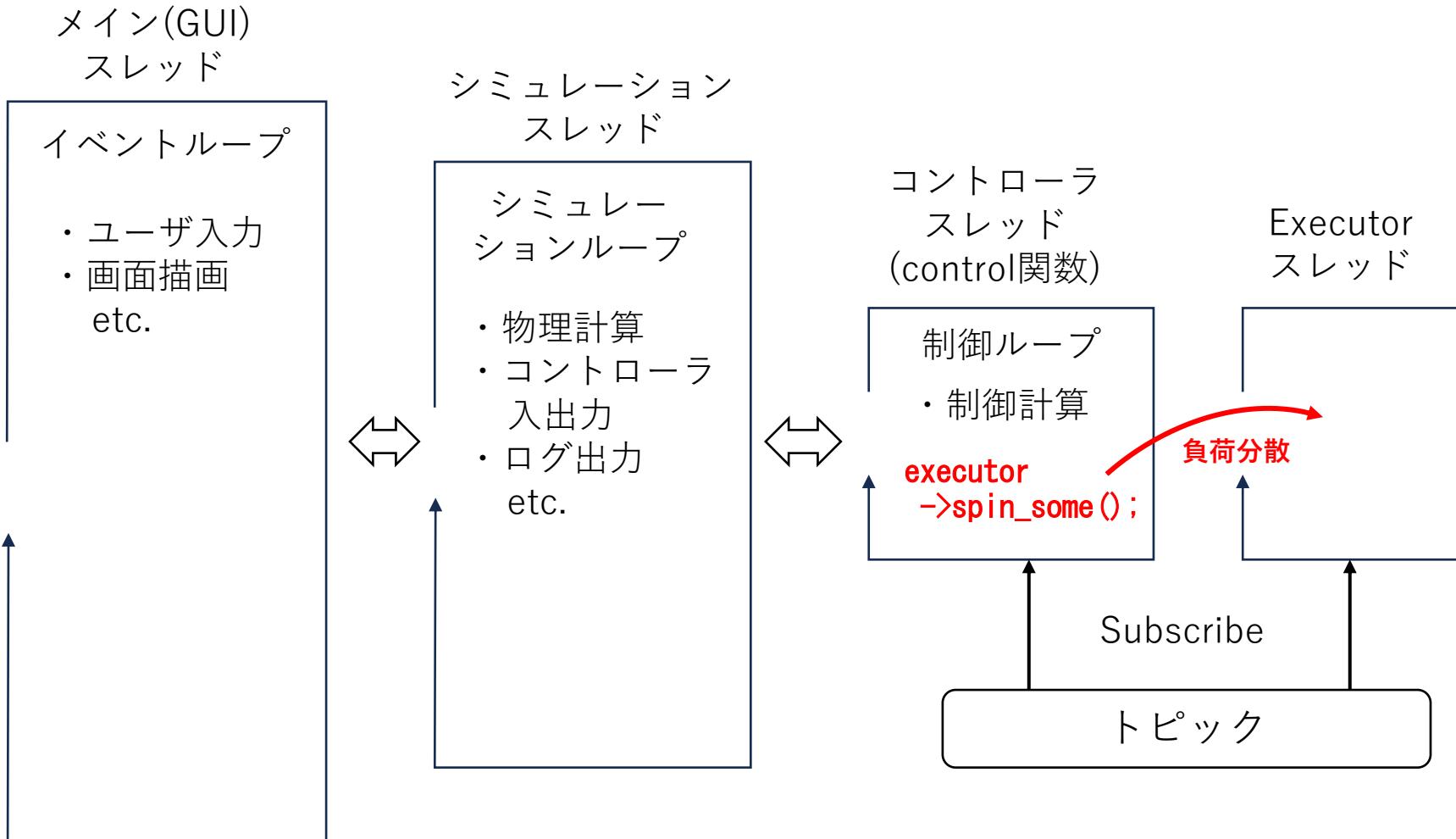
    for(int i=0; i < 2; ++i){
        auto wheel = wheels[i];
        wheel->u() = kd * (dq_target[i] - wheel->dq()); 車軸トルク指令値を設定
    }

    return true;
}
```

コントローラの実行効率向上

- Executorを専用のスレッドで動かす
 - 通信回数・通信量が増えたときのシミュレーションループ、制御ループへの負荷をおさえる
 - リアルタイムファクターを向上させる

スレッド構成



MobileRobotDriveControllerEx.cpp (1/5)

```
#include <cnoid/SimpleController>
#include <rclcpp/rclcpp.hpp>
#include <geometry_msgs/msg/twist.hpp>
#include <memory>
#include <thread>
#include <mutex>

class MobileRobotDriveController : public cnoid::SimpleController
{
public:
    virtual bool configure(cnoid::SimpleControllerConfig* config) override;
    virtual bool initialize(cnoid::SimpleControllerIO* io) override;
    virtual bool control() override;
    virtual void unconfigure() override;

private:
    cnoid::Link* wheels[2];
    rclcpp::Node::SharedPtr node;
    rclcpp::Subscription<geometry_msgs::msg::Twist>::SharedPtr subscription;
    geometry_msgs::msg::Twist command;
    std::unique_ptr<rclcpp::executors::StaticSingleThreadedExecutor> executor;
    std::thread executorThread;
    std::mutex commandMutex;
};

CNOID_IMPLEMENT_SIMPLE_CONTROLLER_FACTORY(MobileRobotDriveController)
```

MobileRobotDriveControllerEx.cpp (2/5)

```
bool MobileRobotDriveController::configure(cnoid::SimpleControllerConfig* config)
{
    node = std::make_shared<rclcpp::Node>(config->controllerName());
    subscription = node->create_subscription<geometry_msgs::msg::Twist>(
        "/cmd_vel", 1,
        [this](const geometry_msgs::msg::Twist::SharedPtr msg) {
            std::lock_guard<std::mutex> lock(commandMutex);
            command = *msg;
        });
    executor = std::make_unique<rclcpp::executors::StaticSingleThreadedExecutor>();
    executor->add_node(node);
executorThread = std::thread([this] () { executor->spin(); });
    return true;
}
```

※ command変数にロックをかけて排他制御

※ Executor用のスレッドを起動し、そこでspin関数を実行し、通信のためのスピループをまわす

MobileRobotDriveControllerEx.cpp (3/5)

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

MobileRobotDriveControllerEx.cpp (4/5)

```
bool MobileRobotDriveController::control()
{
    constexpr double wheelRadius = 0.076;
    constexpr double halfAxeWidth = 0.145;
    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 * halfAxeWidth / 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;
}
```

MobileRobotDriveControllerEx.cpp (5/5)

```
void MobileRobotDriveController::unconfigure()
{
    if(executor) {
        executor->cancel();
        executorThread.join();
        executor->remove_node(node);
        executor.reset();
    }
}
```

※ コントローラアイテム削除時に実行
Executorスレッドを停止する

※ お手本プロジェクトの完成版ファイルは
“src/MobileRobotDriveControllerEx.cpp”

srcのCMakeLists.txt

- src/CMakeLists.txtを以下のように編集

```
choreonoid_add_simple_controller(
    MyMobileRobotDriveTester MobileRobotDriveTester.cpp)

choreonoid_add_simple_controller(
    MyMobileRobotDriveController MobileRobotDriveControllerEx.cpp)

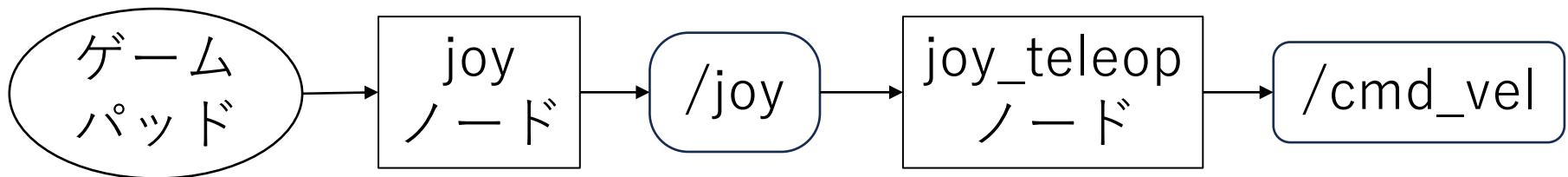
ament_target_dependencies(
    MyMobileRobotDriveController rclcpp geometry_msgs)
```

Part10

ゲームパッドによる操作

ゲームパッドによる操作

- ゲームパッドを接続する
 - PS3, 4, 5のゲームパッドを推奨
- 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



joy トピック

joy ノードを起動

```
ros2 run joy joy_node
```

利用可能なトピックを表示

```
ros2 topic list
```

joy トピックの内容を表示

```
ros2 topic info /joy
```

Joy メッセージ型の内容を表示

```
ros2 interface show sensor_msgs/msg/Joy
```

publishされている内容を確認（ゲームパッドを操作してみる）

```
ros2 topic 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
- ros2 param 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

```
/joy_teleop:  
  ros_parameters:  
    move:  
      type: topic  
      interface_type: geometry_msgs/msg/Twist  
      topic_name: /cmd_vel  
      deadman_buttons: [0]  
      axis_mappings:  
        linear-x:  
          axis: 1  
          scale: 1.0  
          offset: 0  
        angular-z:          ※ お手本パッケージの完成版は  
          axis: 0            config/joy_teleop_twist.yaml  
          scale: 2.0  
          offset: 0
```

joy_teleop ノードの起動

※ joy ノードは起動したままとすること！

```
ros2 run joy_teleop joy_teleop  
--ros-args --params-file  
~/ros2_ws/src/my_mobile_robot/config/joy_teleop.yaml
```

/cmd_vel トピックの内容を確認

```
ros2 topic echo /cmd_vel
```

※ 第一ボタン(“X” もしくは “A” 等)を押しながら左スティックを操作すると Twist の速度・角速度が変化する

“mobile_robot_driver_controller.cnoid” を起動して、ロボットをゲームパッドで操作する

Part11

launchファイルの利用

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

準備

launchディレクトリを作成する

```
cd ~/ros2_ws/src/my_mobile_robot  
mkdir launch
```

トップのCMakeLists.txtにパッケージのファイルを
インストールする記述を追加する

```
...  
ament_package()  
  
install(DIRECTORY project launch config model meshes DESTINATION share/${PROJECT_NAME})
```

joy_teleop用Launchファイル

joy, joy_teleopノードを起動するLaunchファイル

```
<launch>
  <node pkg="joy" exec="joy_node" name="joy" respawn="true"/>
  <node pkg="joy_teleop" exec="joy_teleop" name="joy_teleop">
    <param from=
      "&$(find-pkg-share my_mobile_robot)/config/joy_teleop.yaml"/>
  </node>
</launch>
```

“launch/joy_teleop_launch.xml”として保存する

ビルドしてファイルをインストールする

```
cd ~/ros2_ws
./build.sh
```

Launchファイルの起動

- launchファイルがインストールされていれば、以下のコマンドで起動できる

```
ros2 launch my_mobile_robot joy_teleop_launch.xml
```

- シミュレーションが起動していれば、ゲームパッドで操作できる

Launchの終了

終了するときは端末から”Ctrl + C” を入力すると、
Launchファイルで起動した全てのノードが終了する

全て起動するLaunchファイル

Choreonoidを起動してプロジェクトを読み込み、シミュレーションを開始してゲームパッドも使えるようにする

```
<launch>
  <include file=
    "$(find-pkg-share my_mobile_robot)/launch/joy_teleop_launch.xml"/>
  <node pkg="choreonoid_ros" exec="choreonoid"
        args="--start-simulation $(find-pkg-share
my_mobile_robot)/project/mobile_robot_drive_control.cnoid" />
</launch>
```

“launch/drive_control_launch.xml”として保存し、一度ビルドしてインストールしておく

以下のコマンドで起動

```
ros2 launch my_mobile_robot drive_control_launch.xml
```

Launchファイルの形式

- ROS 1ではXML形式のみ
- ROS 2ではXMLの他にPython、YAMLの形式にも対応
- PythonがROS 2のネイティブ形式
 - スクリプトで書けるので柔軟性が高い
 - 記述は複雑になる
- ROS 1のXML形式に慣れているならXML形式を使えばよい

Python版Launchファイル

“drive_control_launch.py”

```
import os
from ament_index_python import get_package_share_directory
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch_xml.launch_description_sources import XMLLaunchDescriptionSource
from launch_ros.actions import Node

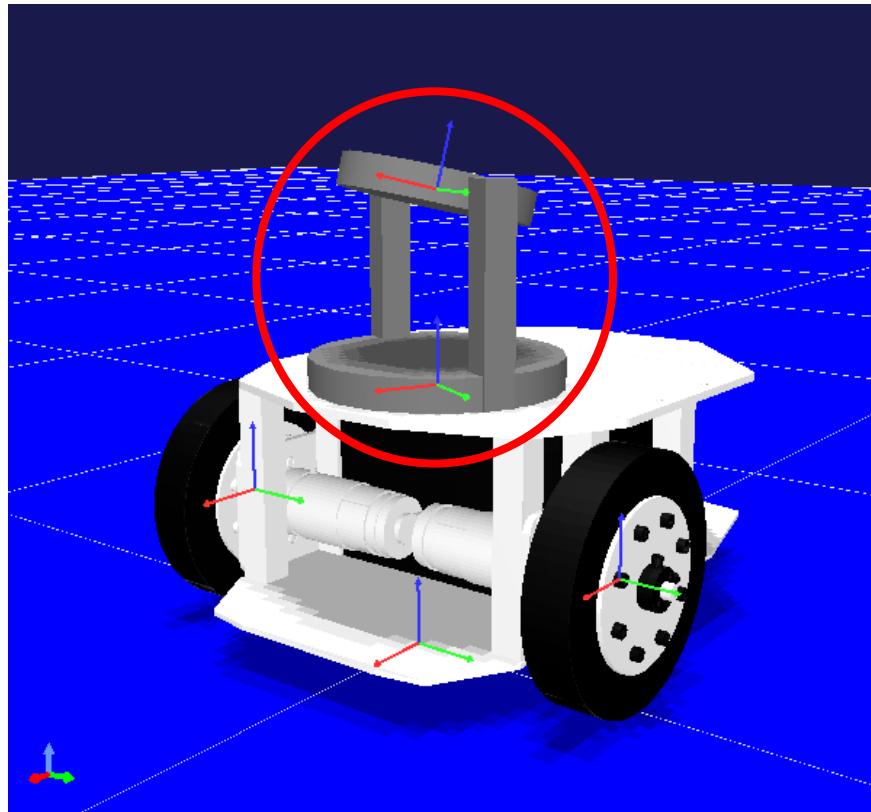
def generate_launch_description():
    launch_include = IncludeLaunchDescription(
        XMLLaunchDescriptionSource(
            os.path.join(
                get_package_share_directory('my_mobile_robot'),
                'launch/joy_teleop_launch.xml'))
    )
    choreonoid_node = Node(
        package = 'choreonoid_ros',
        executable = 'choreonoid',
        arguments = [
            '--start-simulation',
            os.path.join(
                get_package_share_directory('my_mobile_robot'),
                'project/mobile_robot_drive_control.cnoid')]]
    )
    return LaunchDescription([ launch_include, choreonoid_node ])
```

Part12

関節のモデリング

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 ]
```

パン関節の追加(2/2)

```
- 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
```

※ パン関節の形状の記述

チルト関節の追加

```
-  
  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
```

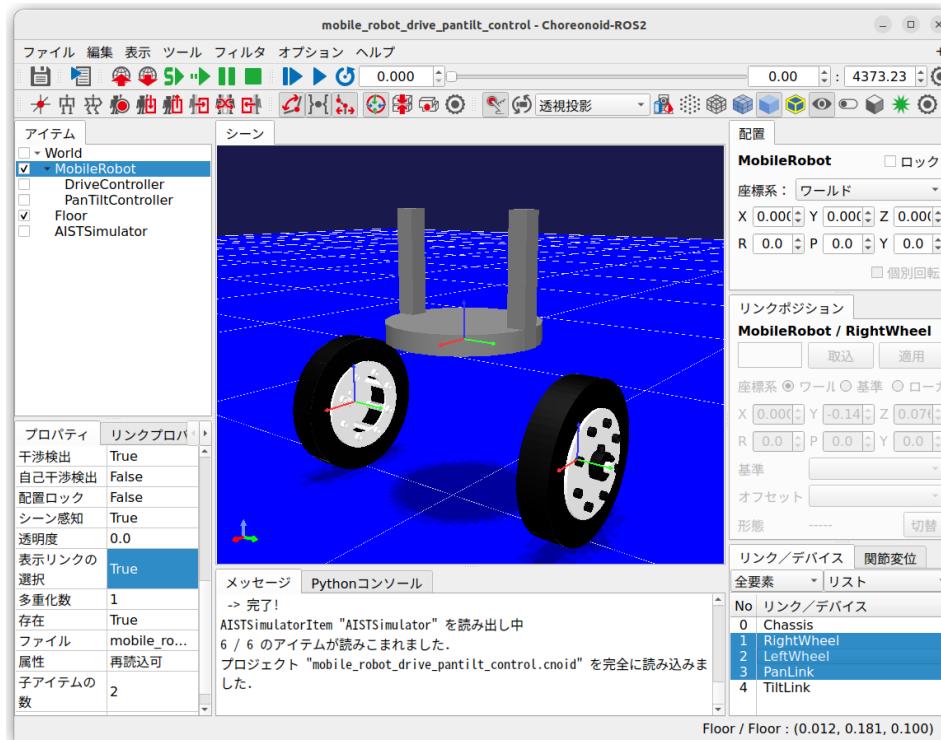
※ お手本パッケージの完成版は “model/mobile_robot_pantilt.body”

関節リミット値の設定

- ・今日は設定しない
- ・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



Part13

関節の制御

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
- ROSメッセージ型
 - geometry_msg/msg/Vector3
- ROSトピック名
 - /cmd_joint_vel

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をセットした場合、それらが同時に機能します

MobileRobotPanTiltController.cpp (1/5)

```
#include <cnoid/SimpleController>
#include <rclcpp/rclcpp.hpp>
#include <geometry_msgs/msg/vector3.hpp>
#include <memory>
#include <thread>
#include <mutex>

class MobileRobotPanTiltController : public cnoid::SimpleController
{
public:
    virtual bool configure(cnoid::SimpleControllerConfig* config) override;
    virtual bool initialize(cnoid::SimpleControllerIO* io) override;
    virtual bool control() override;
    virtual void unconfigure() override;

private:
    cnoid::Link* joints[2];
    rclcpp::Node::SharedPtr node;
    rclcpp::Subscription<geometry_msgs::msg::Vector3>::SharedPtr subscription;
    geometry_msgs::msg::Vector3 command;
    std::unique_ptr<rclcpp::executors::StaticSingleThreadedExecutor> executor;
    std::thread executorThread;
    std::mutex commandMutex;
};

CNOID_IMPLEMENT_SIMPLE_CONTROLLER_FACTORY(MobileRobotPanTiltController)
```

※ 赤字はMobileRobotDriveControllerEx.cpp と異なる部分

MobileRobotPanTiltController.cpp (2/5)

```
bool MobileRobotPanTiltController::configure(cnoid::SimpleControllerConfig* config)
{
    node = std::make_shared<rclcpp::Node>(config->controllerName());
    subscription = node->create_subscription<geometry_msgs::msg::Vector3>(
        "/cmd_joint_vel", 1,
        [this](const geometry_msgs::msg::Vector3::SharedPtr msg) {
            std::lock_guard<std::mutex> lock(commandMutex);
            command = *msg;
        });
    executor = std::make_unique<rclcpp::executors::StaticSingleThreadedExecutor>();
    executor->add_node(node);
    executorThread = std::thread([this] () { executor->spin(); });

    return true;
}
```

MobileRobotPanTiltController.cpp (3/5)

```
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) {
        auto joint = joints[i];
        joint->setActuationMode(JointTorque);
        io->enableInput(joint, JointVelocity);
        io->enableOutput(joint, JointTorque);
    }
    return true;
}
```

MobileRobotPanTiltController.cpp (4/5)

```
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;
}
```

MobileRobotPanTiltController.cpp (5/5)

```
void MobileRobotPanTiltController::unconfigure()
{
    if(executor) {
        executor->cancel();
        executorThread.join();
        executor->remove_node(node);
        executor.reset();
    }
}
```

※ お手本パッケージの完成版ファイルは
“src/MobileRobotPanTiltController.cpp”

srcのCMakeLists.txt

- src/CMakeLists.txtに以下を追記

```
choreonoid_add_simple_controller(  
    MyMobileRobotPanTiltController MobileRobotPanTiltController.cpp)  
  
ament_target_dependencies(  
    MyMobileRobotPanTiltController rclcpp geometry_msgs)
```

※ 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

ビルドしてMyMobileRobotPanTiltController.so
を生成する

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”を設定

プロジェクトを
“project/mobile_robot_drive_pantilt_control.cnoid”
として保存する

※ 同名のプロジェクトファイルがお手本パッケージにもあります

ros2 topicコマンドによる操作

```
ros2 topic pub -1 /cmd_joint_vel geometry_msgs/msg/Vector3  
“{ x: 0, y: 0.5, 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

右スティックの状態を/cmd_joint_velにpublish

```
joy_teleop:  
  ros_parameters:  
    move:  
      ...  
    pan_tilt:  
      type: topic  
      interface_type: geometry_msgs/msg/Vector3  
      topic_name: /cmd_joint_vel  
      deadman_buttons: [0]  
      axis_mappings:  
        y:  
          axis: 4  
          scale: 2.0  
          offset: 0  
        z:  
          axis: 3  
          scale: 2.0  
          offset: 0
```

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

Launchファイルの作成

```
<launch>
  <include file=
    "$(find-pkg-share my_mobile_robot)/launch/joy_teleop_launch.xml"/>
  <node pkg="choreonoid_ros" exec="choreonoid"
        args="--start-simulation $(find-pkg-share
my_mobile_robot)/project/mobile_robot_drive_pantilt_control.cnoid" />
</launch>
```

“launch/drive_pantilt_control_launch.xml” として保存し、一度ビルドしてインストールしておく

以下のコマンドで起動

```
ros2 launch my_mobile_robot drive_pantilt_control_launch.xml
```

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

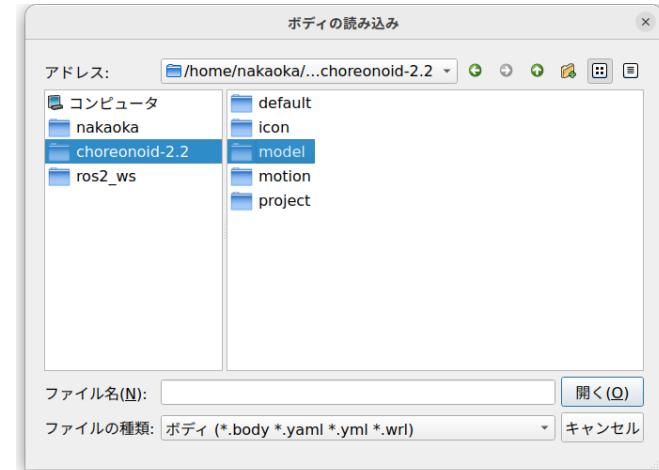
Part14

視覚センサの追加

環境モデルの導入

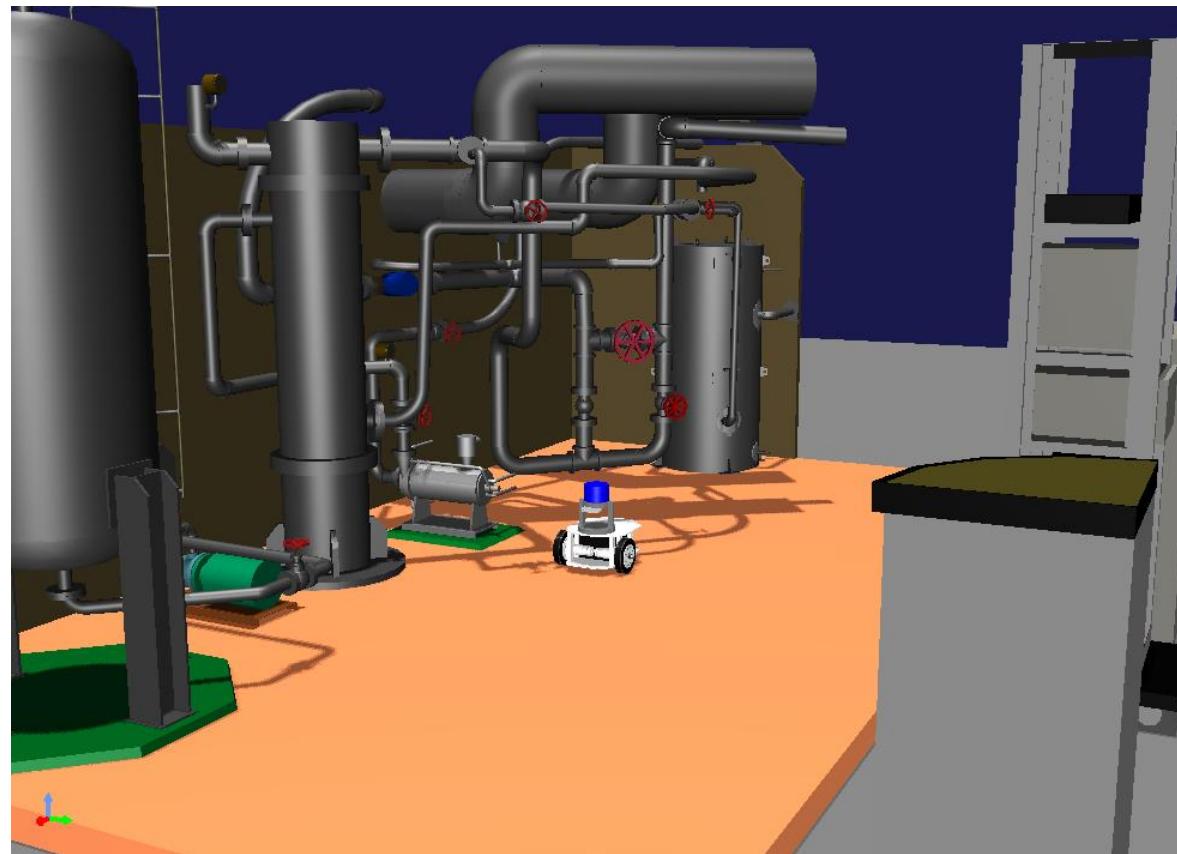
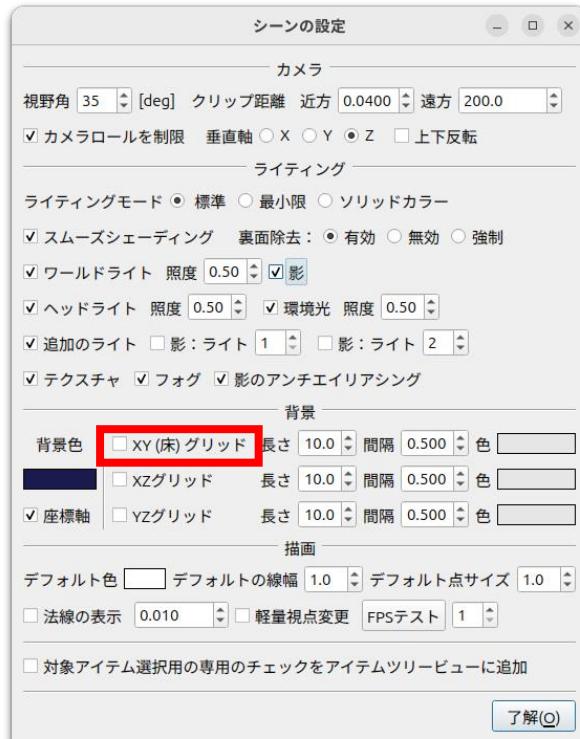
- 床モデル (floor) を削除
- 研究プラントのモデル (Labo1) を読み込む
 - “choreonoid-2.x/model/Labo1/Labo1v2.body”

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



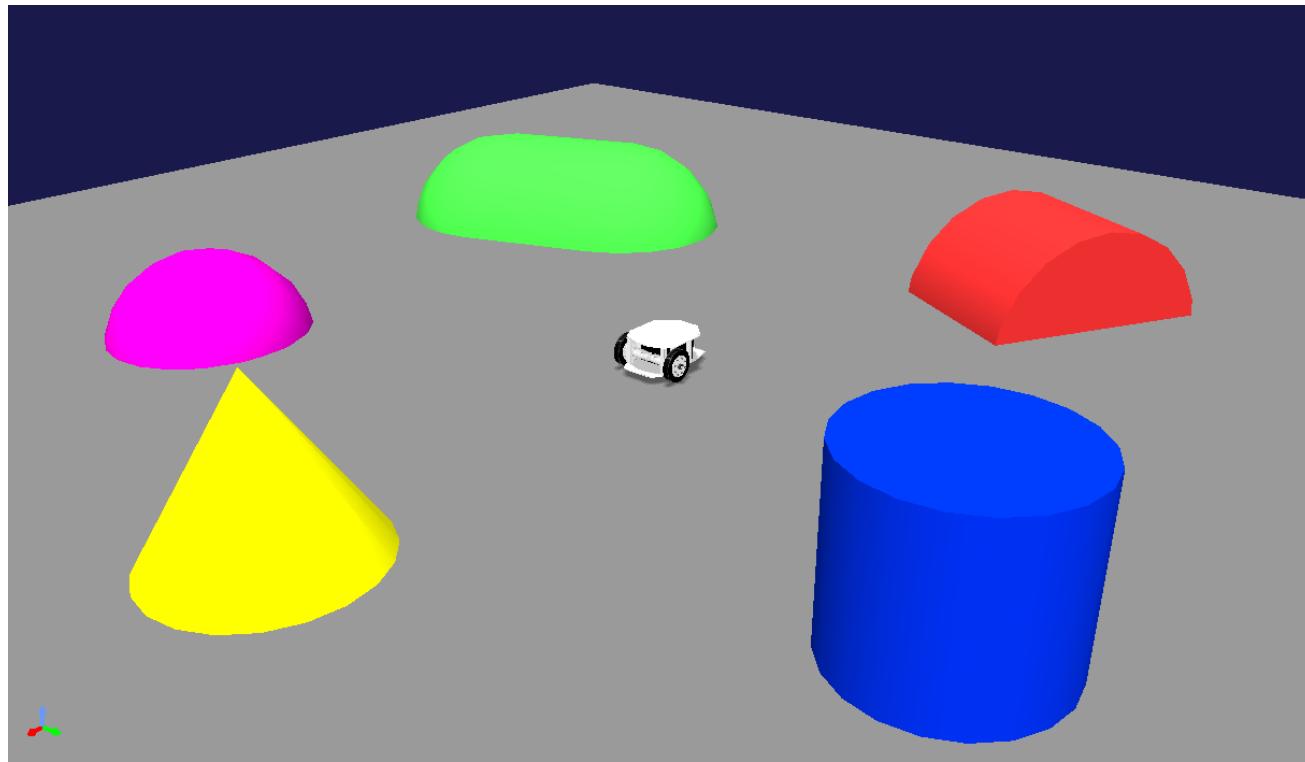
シーン描画設定

- XY(床)グリッドのチェックを外す



簡易地形モデル

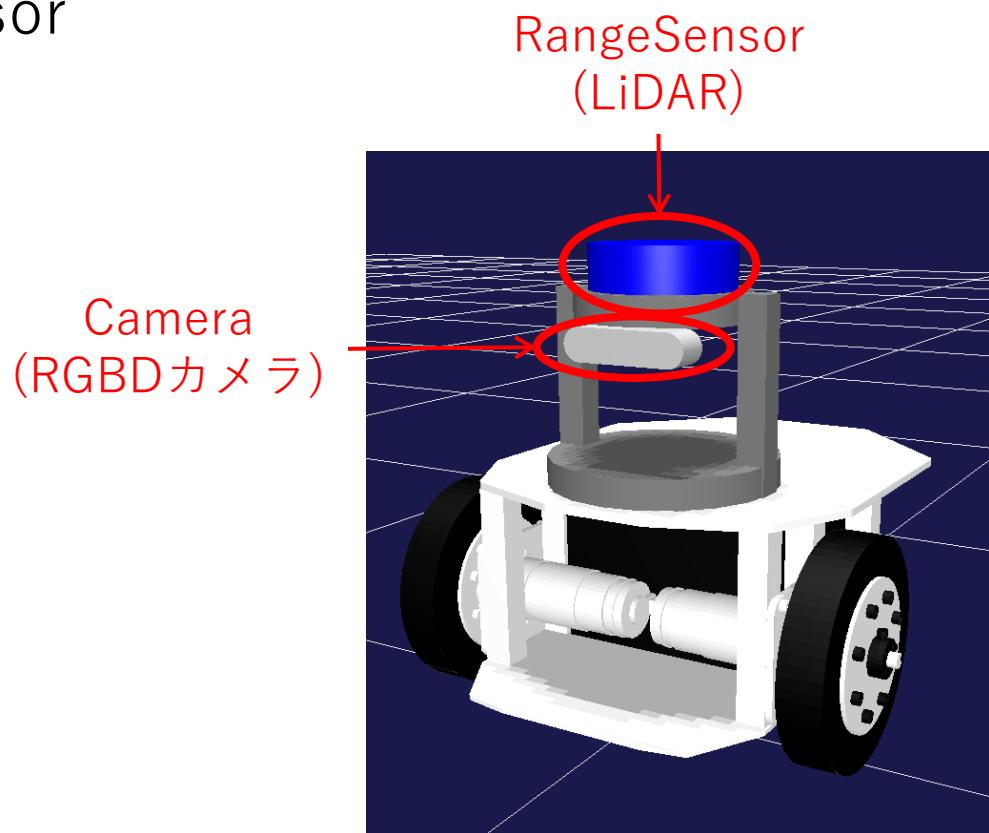
お手本パッケージの “model/terrain.body”



Labo1モデルを使用すると描画やシミュレーション
が重くなる場合はこちらを使用してください

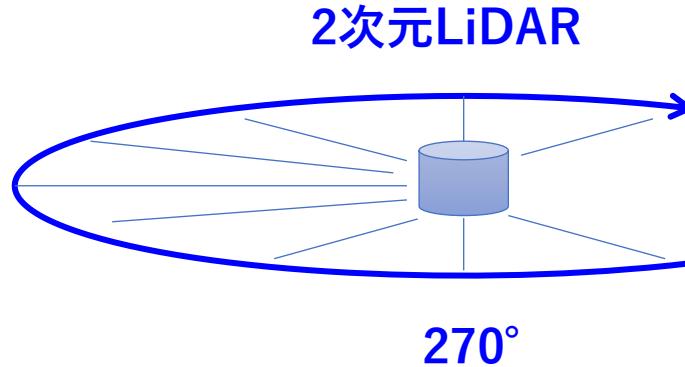
センサの追加

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



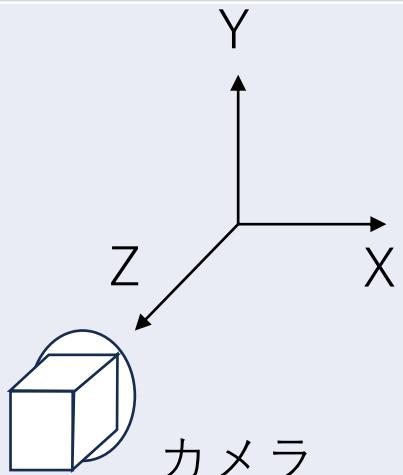
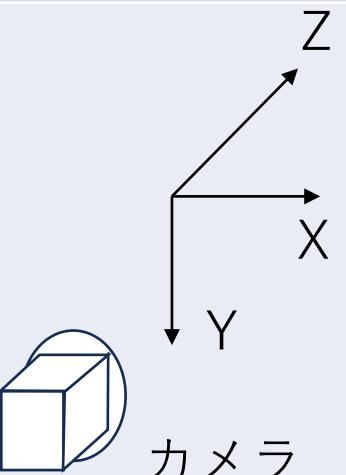
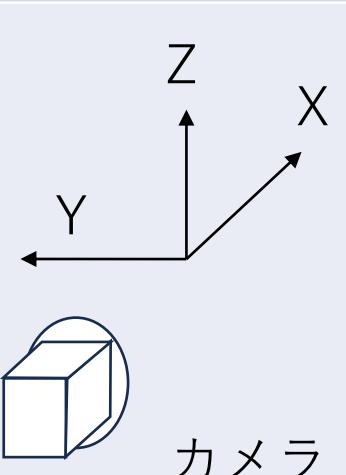
レンジセンサ(LiDAR)の追加

```
- name: TiltLink
...
elements:
  -
    type: Shape
    ...
  -
    type: RangeSensor
    name: LiDAR
    translation: [ 0, 0, 0.025 ]
    optical_frame: robotics
    yaw_range: 270.0
    yaw_step: 1.0
    scan_rate: 10
    max_distance: 10.0
    detection_rate: 0.9
    error_deviation: 0.005
    elements:
      -
        type: Shape
        rotation: [ 1, 0, 0, 90 ]
        geometry: { type: Cylinder, radius: 0.05, height: 0.03 }
        appearance:
          material: { diffuse: [ 0, 0, 1 ], specular: [ 1, 1, 1 ] }
```



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 (カメラ)	ロボット工学 距離センサ
前方	-Z	Z	X
鉛直上向	Y	-Y	Z
図示	 カメラ	 カメラ	 カメラ

LiDARの記述内容

```
-  
  type: RangeSensor          センサの型を指定  
  name: LiDAR                センサの名前を指定 (ROSトピック名などにも反映される)  
  translation: [ 0, 0, 0.025 ] チルトリンクにおけるセンサのローカル座標  
  optical_frame: robotics    光学フレームを指定  
  yaw_range: 270.0            センサの水平計測範囲を270°に設定  
  yaw_step: 1.0               計測の解像度を1°に設定  
  scan_rate: 10                計測レート (1秒間の計測回数) を10に設定  
  max_distance: 10.0          最大計測距離を10mに設定  
  detection_rate: 0.9          検出率を0.9 (90%)に設定  
  error_deviation: 0.005      誤差を0.005m (5mm)に設定  
  elements:  
    -  
      type: Shape              センサの形状を記述  
      rotation: [ 1, 0, 0, 90 ]  
      geometry: { type: Cylinder, radius: 0.05, height: 0.03 }  
      appearance:  
        material: { diffuse: [ 0, 0, 1 ], specular: [ 1, 1, 1 ] }
```

RGB-Dカメラの追加 (1/2)

```
- name: TiltLink
...
elements:
...
-
  type: Camera
  name: RealSense
  translation: [ 0.06, 0, -0.02 ]          光学フレームCVのカメラを
  rotation: [ [ 0, 1, 0, 90 ], [ 0, 0, 1, -90 ] ]    前方に向かせるための回転
  optical_frame: cv                         光学フレームを指定
  format: COLOR_DEPTH                      この記述によりカメラがRGB-Dカメラになる
  field_of_view: 62                          視野角
  width: 320                                カメラ画像解像度を320 x 240に設定
  height: 240
  frame_rate: 30                            フレームレートを30に設定
  detection_rate: 0.9
  error_deviation: 0.005
  elements:
    形状の記述...
```

インテルRealSense風の
RGB-Dカメラを追加する

RGB-Dカメラの追加 (2/2)

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

RealSense風の形状を
プリミティブで記述

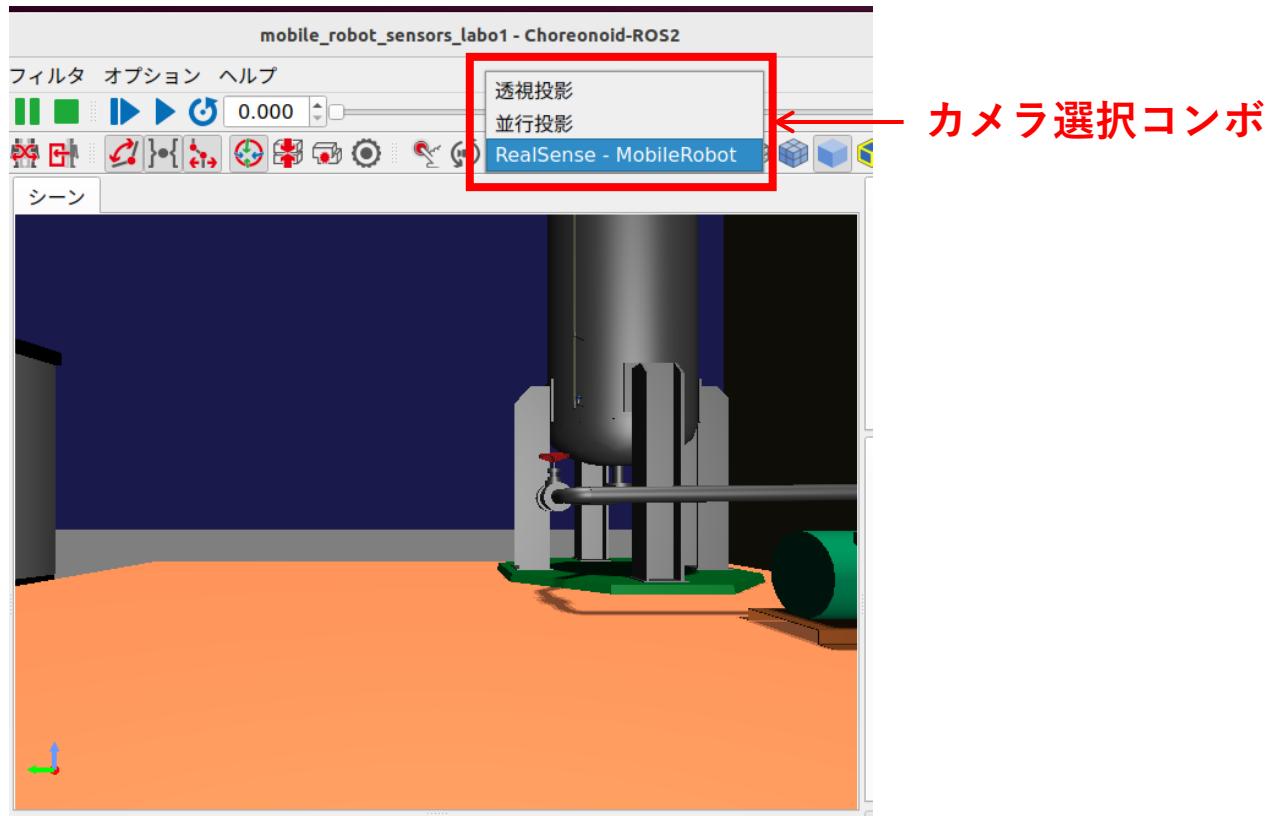
※ お手本パッケージの完成版は “model/mobile_robot_sensors.body”

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

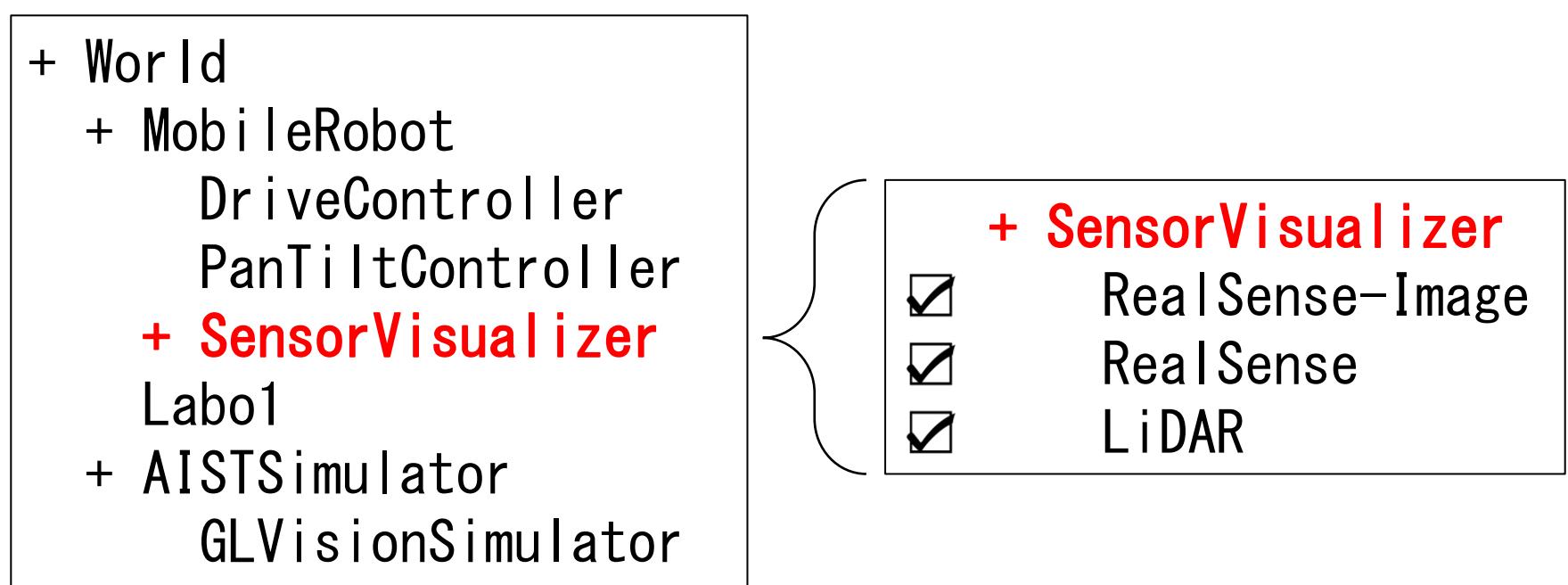
- Choreonoid上で表示
- Choreonoid外部で表示
 - 後ほどROS通信、ROSツールを用いて実現する

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
 - 「オン」にすると全フレームの画像データ、距離データがログとして記録される
 - 現状ではメモリが尽きるとクラッシュする
- 以上の設定により、センサデータがChoreonoid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)

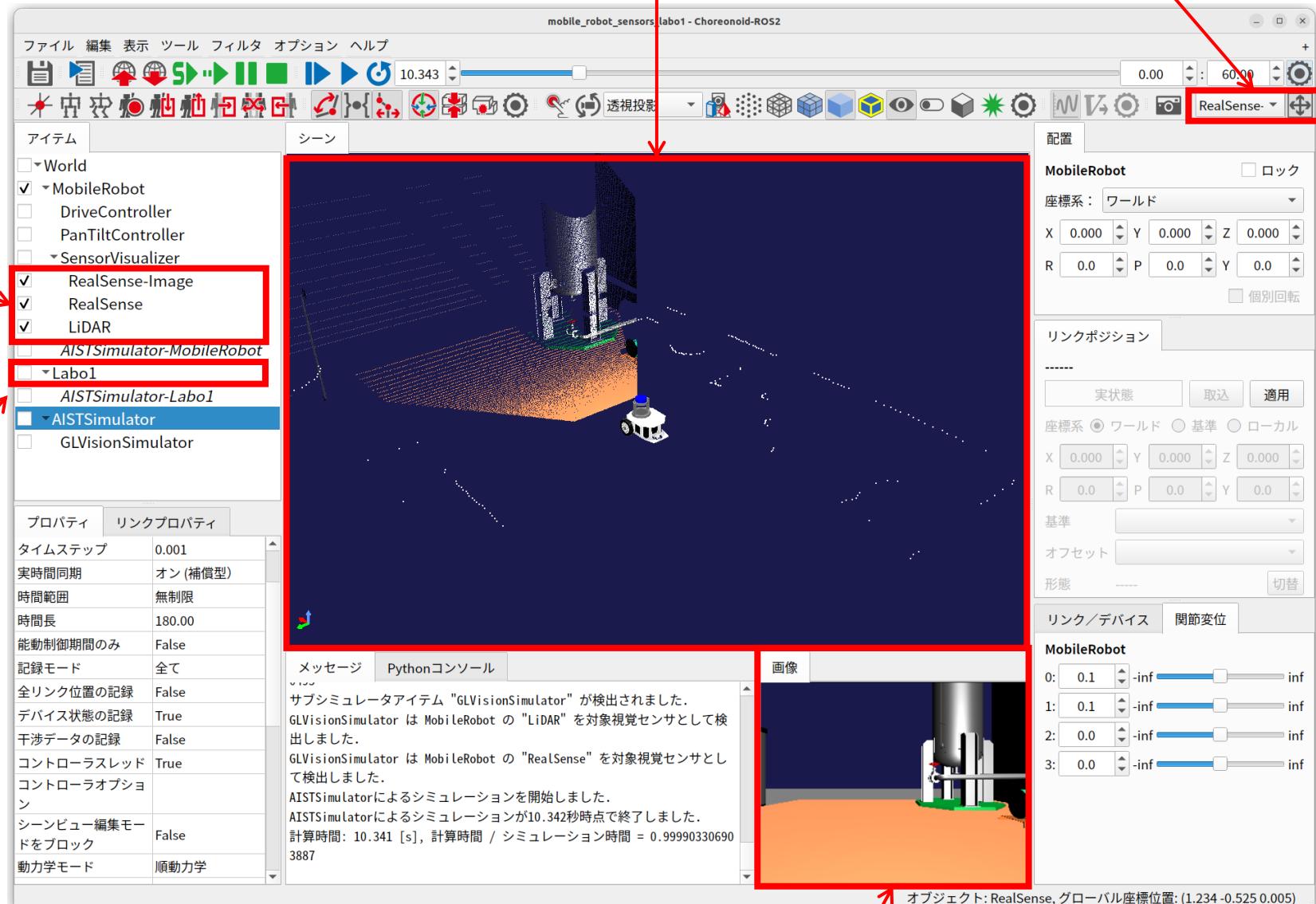
- カメラ画像については「画像ビュー」で表示可能
- 「画像ビューバー」の選択コンボで対象のセンサを選択する

センサデータの可視化

画像選択コンボ

センサ
可視化
アイテム

Labo1
のチェック
は外す
とよい



画像ビュー

プロジェクト&Launchファイル

- プロジェクトファイルを保存
 - “project/mobile_robot_sensors.cnoid”
- launchファイルを作成
 - “launch/sensors_launch.xml”

```
<launch>
  <include file=
    "$(find-pkg-share my_mobile_robot)/launch/joy_teleop.launch.xml"/>
  <node pkg="choreonoid_ros" exec="choreonoid"
        args="--start-simulation $(find-pkg-share
my_mobile_robot)/project/mobile_robot_sensors.cnoid" />
</launch>
```

※ 同名の完成版ファイルがお手本パッケージにあります

Part15

ROS通信を用いた状態の出力

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
 - rclcppライブラリを用いることで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

BodyROS2アイテムの導入

- + World
 - + MobileRobot
 - DriveController
 - PanTiltController
 - + SensorVisualizer
 - BodyROS2**
- Labo1
- + AISTSimulator
- GLVisionSimulator

「ファイル」 - 「新規」 -
「BodyROS2」で生成する

効率化のための補足

- BodyROS2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
 - GLVisionSimulatorの「ビジョンデータの記録」プロパティをfalseにする
 - SensorVisualizerを削除する
 - もしくは、各項目のチェックを外す

BodyROS2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
 - /MobileRobot/RealSense
- デプスカメラ画像（ポイントクラウド）
 - /MobileRobot/RealSense/point_cloud
- レンジセンサ距離データ
 - /MobileRobot/LiDAR/scan

トピックの確認

```
ros2 topic list
```

“-t” オプションを付けるとメッセージ型も確認できる

```
ros2 topic list -t
```

以下が表示されていればOK

```
/MobileRobot/LiDAR/scan [sensor_msgs/msg/LaserScan]  
/MobileRobot/RealSense [sensor_msgs/msg/Image]  
/MobileRobot/RealSense/point_cloud [sensor_msgs/msg/PointCloud2]  
/MobileRobot/joint_states [sensor_msgs/msg/JointState]
```

補足：画像データの追加トピック

- image_transport_pluginsをインストールしていると、以下のようなトピックも生成される

```
/MobileRobot/RealSense/compressed [sensor_msgs/msg/CompressedImage]  
/MobileRobot/RealSense/theora [theora_image_transport/msg/Packet]  
/MobileRobot/RealSense/zstd [sensor_msgs/msg/CompressedImage]
```

- これらは画像データを圧縮して送信するトピックで、これらを用いることで通信量を削減することができる

トピック	形式
compressed	JPEG
theora	Theora動画圧縮(ストリーミング)
zstd	zstd圧縮

トピックのデータの確認

- ・シミュレーションを開始
- ・“ros2 topic echo” コマンドで確認

```
ros2 topic echo /MobileRobot/joint_states
```

```
ros2 topic echo /MobileRobot/LiDAR/scan
```

仮想世界全体情報のPublish

- WorldROS2アイテムを追加する

```
+ World
  + MobileRobot
    DriveController
    PanTiltController
    BodyROS
  + SensorVisualizer
  Labo1
  + AISTSimulator
    GLVisionSimulator
WorldROS2
```

「ファイル」 - 「新規」 -
「WorldROS2」で生成する

Clock トピック

- “/clock” トピック
- 他のROSノードと時刻を共有（同期）
- WorldROS2アイテムがPublish
- “rosgraph_msgs/msg/Clock”型
- 利用方法
 - 時刻を必要とするノードのROSパラメータ “/use_sim_time” をtrueにセットする

プロジェクト&Launchファイル

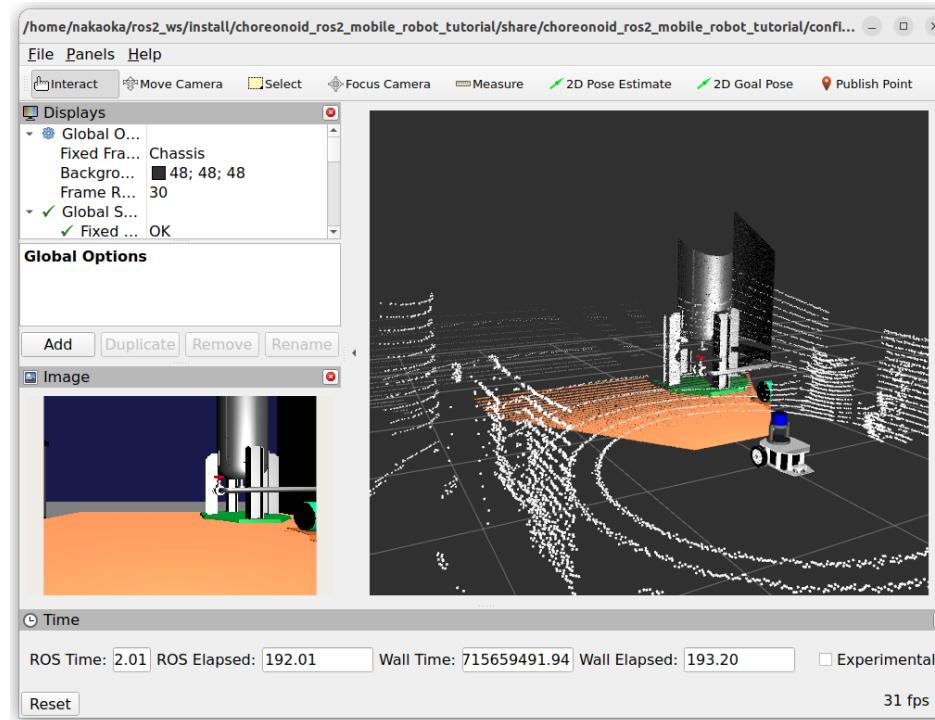
- プロジェクトファイルを保存
 - “project/mobile_robot_sensors_publish.cnoid”
- launchファイルを作成
 - “launch/sensors_publish_launch.xml”

```
<launch>
<include file=
    "$(find-pkg-share my_mobile_robot)/launch/joy_teleop.launch.xml"/>
<node pkg="choreonoid_ros" exec="choreonoid"
      args="--start-simulation $(find-pkg-share
my_mobile_robot)/project/mobile_robot_sensors_publish.cnoid" />
</launch>
```

※ 同名の完成版ファイルがお手本パッケージにあります

RVizによる状態表示

- ROSの可視化・操作ツール
- センサデータのトピックも可視化できる



Rviz利用の手順

- 可視化したい情報がROSトピックとして publishされるようにする
- ロボットモデルの可視化を行う場合
 - 対象のロボットモデルの**URDF**を用意する
 - robot_state_publisherノードを起動する
- 可視化の項目をrvizのGUIで追加する
- rvizの状態をconfigファイルとして保存する
- パラメータとconfigファイルを指定して起動するlaunchファイルを作成する

Part16

URDF

URDFとは

- ROS標準のモデルファイル記述形式
- 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
 - YAMLで記述
- URDF
 - 既存資産の活用
 - ROSで必要となることが多い
 - XMLで記述

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://my_mobile_robot/meshes/vmega_body.dae"/>
      </geometry>
    </visual>
    <collision>
      <geometry>
        <mesh filename="package://my_mobile_robot/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

- URDF → Body
 - URDFを読み込んでBody形式で保存する
- Body → URDF
 - body2urdfコマンドで変換する
 - “feature/urdf-writer” ブランチで開発中
 - 現状ではメッシュファイルへの参照を書き出す部分が不完全

Part17

Rvizによる可視化

rvizの起動

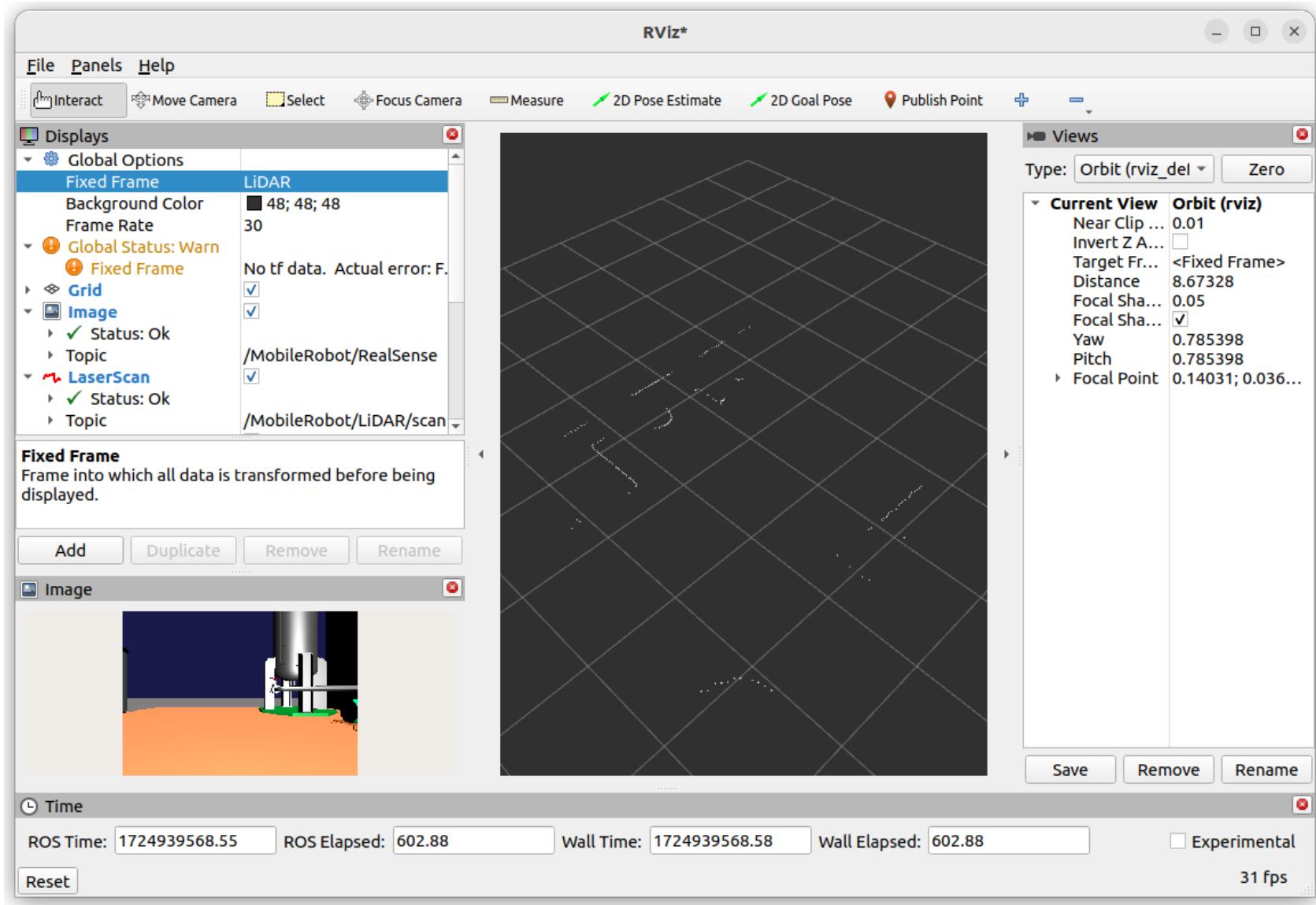
以下のコマンドで起動 (ROS 2ではrviz2になる)

```
ros2 run rviz2 rviz2
```

シミュレーションとセンサデータのpublishを開始

```
ros2 launch my_mobile_robot sensors_publish.launch.xml
```

何か可視化してみる



カメラ画像の可視化

1. 左下の”Add”ボタンを押す
2. “Image” を選択して “OK”
 - 左側パネルに “Image” のツリーが追加される
3. Imageの”Topic”に対象トピックを指定
 - /MobileRobot/RealSense
4. 左下の領域に画像が表示される

※ "/MobileRobot/RealSense/**compressed**" もしくは
"/MobileRobot/RealSense/**theora**"
を指定することで通信量を削減できます

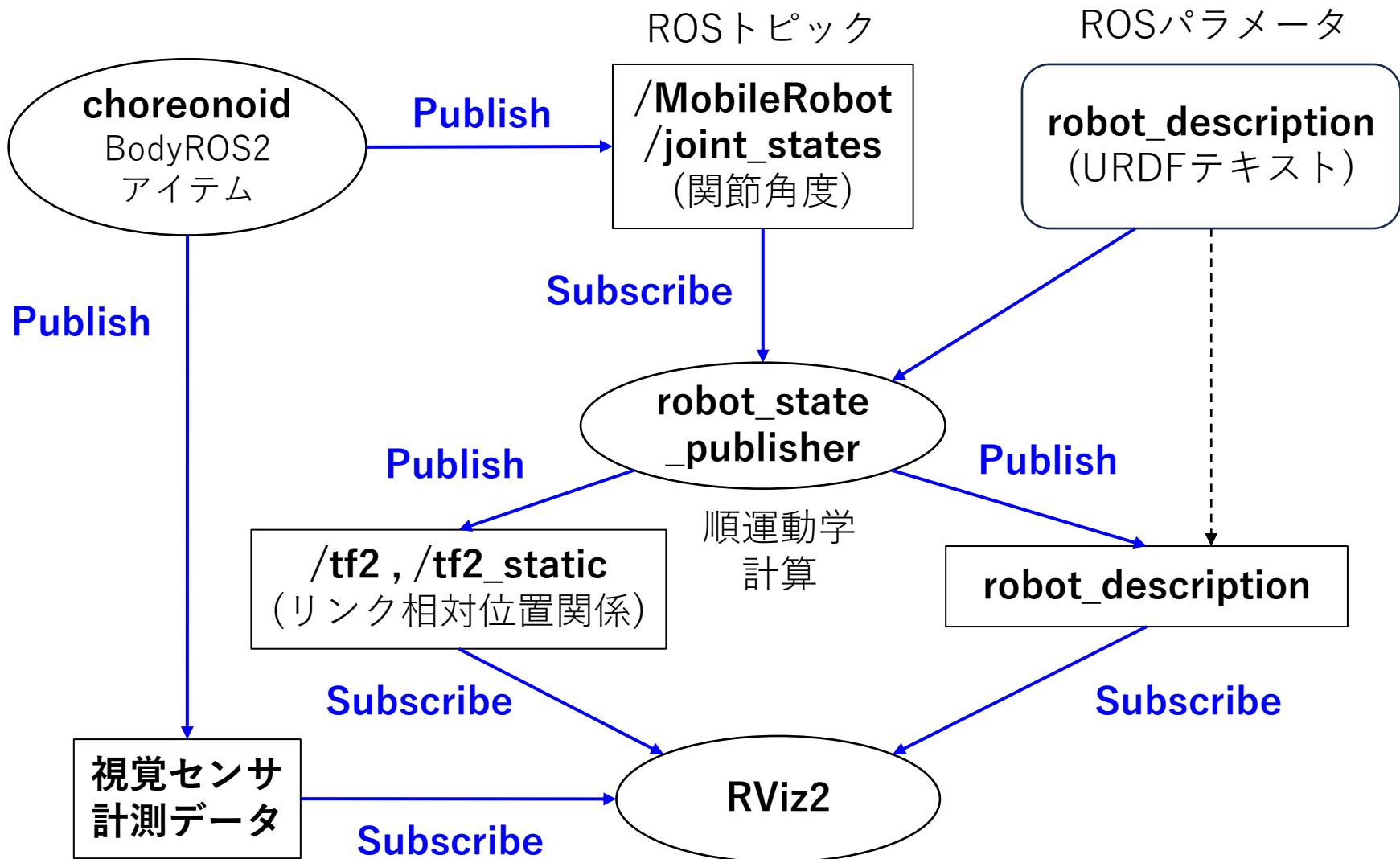
LiDARデータの可視化

1. 左下の”Add”ボタンを押す
2. “LaserScan”を選択して“OK”
 - 左側パネルに“LaserScan”的ツリーが追加される
3. LaserScanの”Topic”に対象トピックを指定
 - /MobileRobot/LiDAR/scan
4. 上にある”Global Options”の“Fixed Frame”に“LiDAR”を指定
5. 中央の領域にLiDARデータが表示される

ロボットモデルの導入

- ロボットモデルの表示
- センサデータの位置合わせ
- 必要な情報
 - ロボットのURDF
 - 各リンクの位置姿勢
 - センサの位置姿勢を含む
- robot_state_publisherを使用する

ロボットモデル表示の流れ



RVizの設定続き

- Image、 LaserScanに加えて…
- Addボタンで “RobotModel”を追加
 - “Description Topic” に “/robot_description” を指定
- Addボタンで “PointCloud2” を追加
 - “Topic” に “/MobileRobot/RealSense/point_cloud” を指定
- “Fixed Frame” に “Chassis” を指定
 - ロボットのベースリンクに対応
- 設定をファイルに保存
 - “File” – “Save Config As”
 - “my_mobile_robot/config/mobile_robot.rviz” に保存

RViz用launchファイル

“launch/display.launch.xml”

```
<launch>
  <arg name="model" value="$(find-pkg-share my_mobile_robot)/model/mobile_robot_sensors.urdf"/> URDFファイル
  <arg name="rvizconfig" value="$(find-pkg-share my_mobile_robot)/config/mobile_robot.rviz"/> rvizの設定ファイル

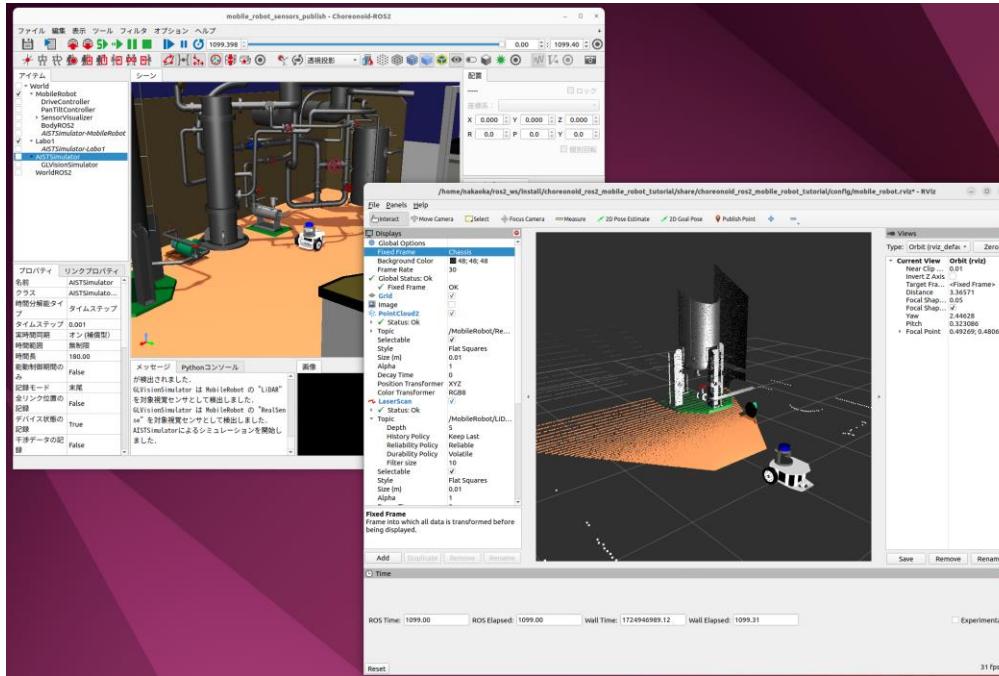
  <node pkg="robot_state_publisher" exec="robot_state_publisher">
    <param name="robot_description" value="$(command 'xacro $(var model)')"/>
    <param name="use_sim_time" value="true" />
    <remap from="/joint_states" to="/MobileRobot/joint_states"/> トピック名
  </node>

  <node pkg="rviz2" exec="rviz2" args="-d $(var rvizconfig)">
    <param name="use_sim_time" value="true" />
  </node>
  <node pkg="rqt_graph" exec="rqt_graph"/> rqt_graphも表示する
</launch>
```

全起動launchファイル

“launch/sensors_display.launch.xml”

```
<launch>
  <include file="$(find-pkg-share my_mobile_robot)/launch/sensors_publish.launch.xml"/>
  <include file="$(find-pkg-share my_mobile_robot)/launch/display.launch.xml"/>
</launch>
```



遠隔操作

- PCを2台用意する
- ROS_LOCALHOST_ONLY=0に戻す
- 必要に応じてROS_DOMAIN_ID同じ値に設定する

PC1（シミュレーション用、ロボット実機に相当）

```
cd ~/ros2_ws/src/my_mobile_robot/  
ros2 run choreonoid_ros choreonoid project/mobile_robot_sensors_publish.cnoid
```

PC2（遠隔操作端末）

```
ros2 launch my_mobile_robot joy_teleop_launch.xml
```

```
ros2 launch my_mobile_robot display_launch.xml
```

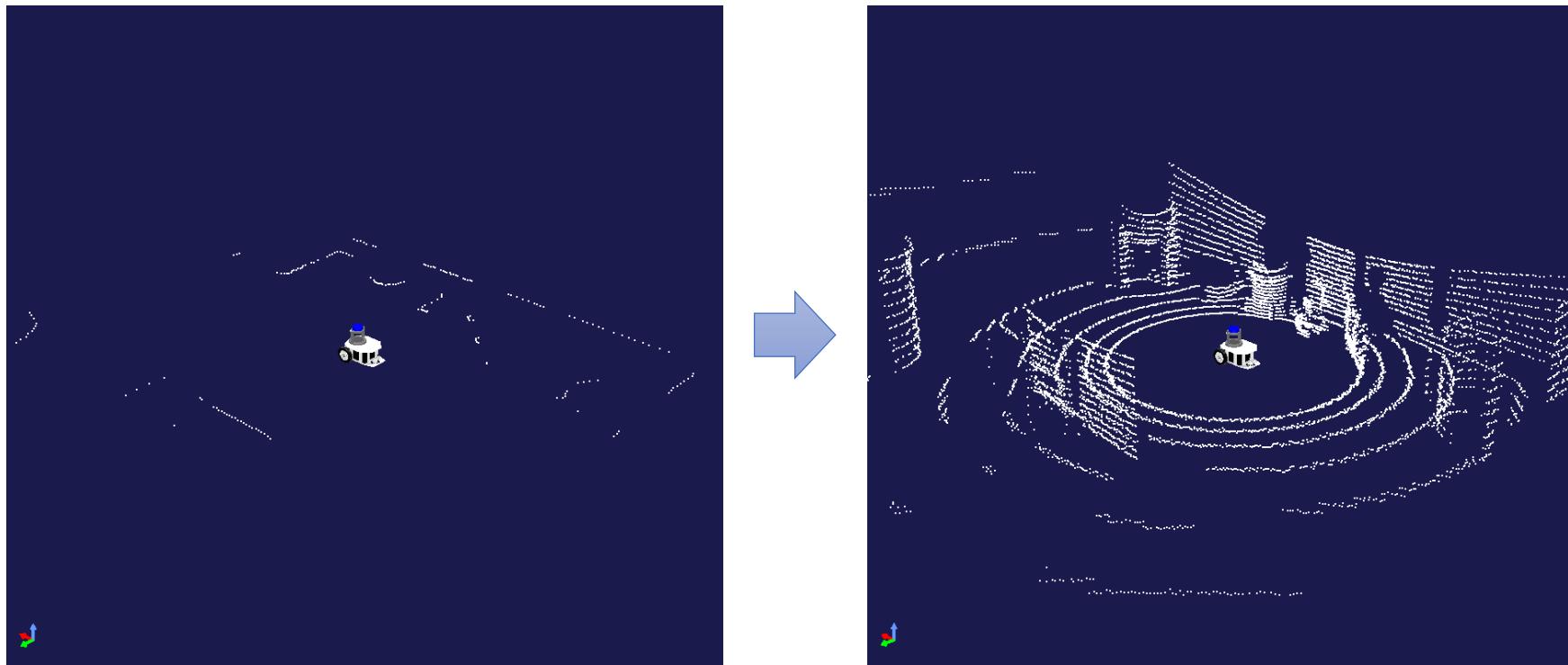
- rvizを見ながらゲームパッドを操作してChoreonoid上のロボットを遠隔操作する

Part18

センサ／デバイスの拡充

3次元LiDAR

- 垂直方向にも計測範囲がある3次元LiDARにも対応可能



3次元LiDARの記述

- VLP-16風のセンサを記述する

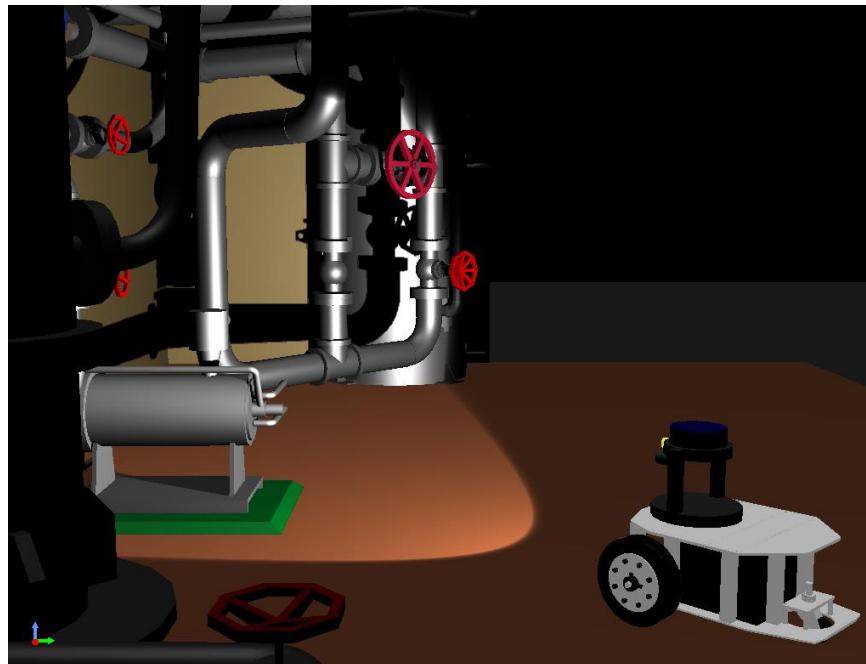
```
type: RangeSensor
name: LiDAR
translation: [ 0, 0, 0.025 ]
optical_frame: robotics
yaw_range: 360.0
yaw_step: 0.4
pitch_range: 30.0
pitch_step: 2.0
scan_rate: 20
max_distance: 100.0
detection_rate: 0.9
error_deviation: 0.005
```

ピッチ軸まわりの計測範囲と分解能を
記述することで3次元LiDARとなる

ROSトピックは“/MobileRobot/LiDAR/point_cloud”(PointCloud2型)
となる

ライト（光源）の追加

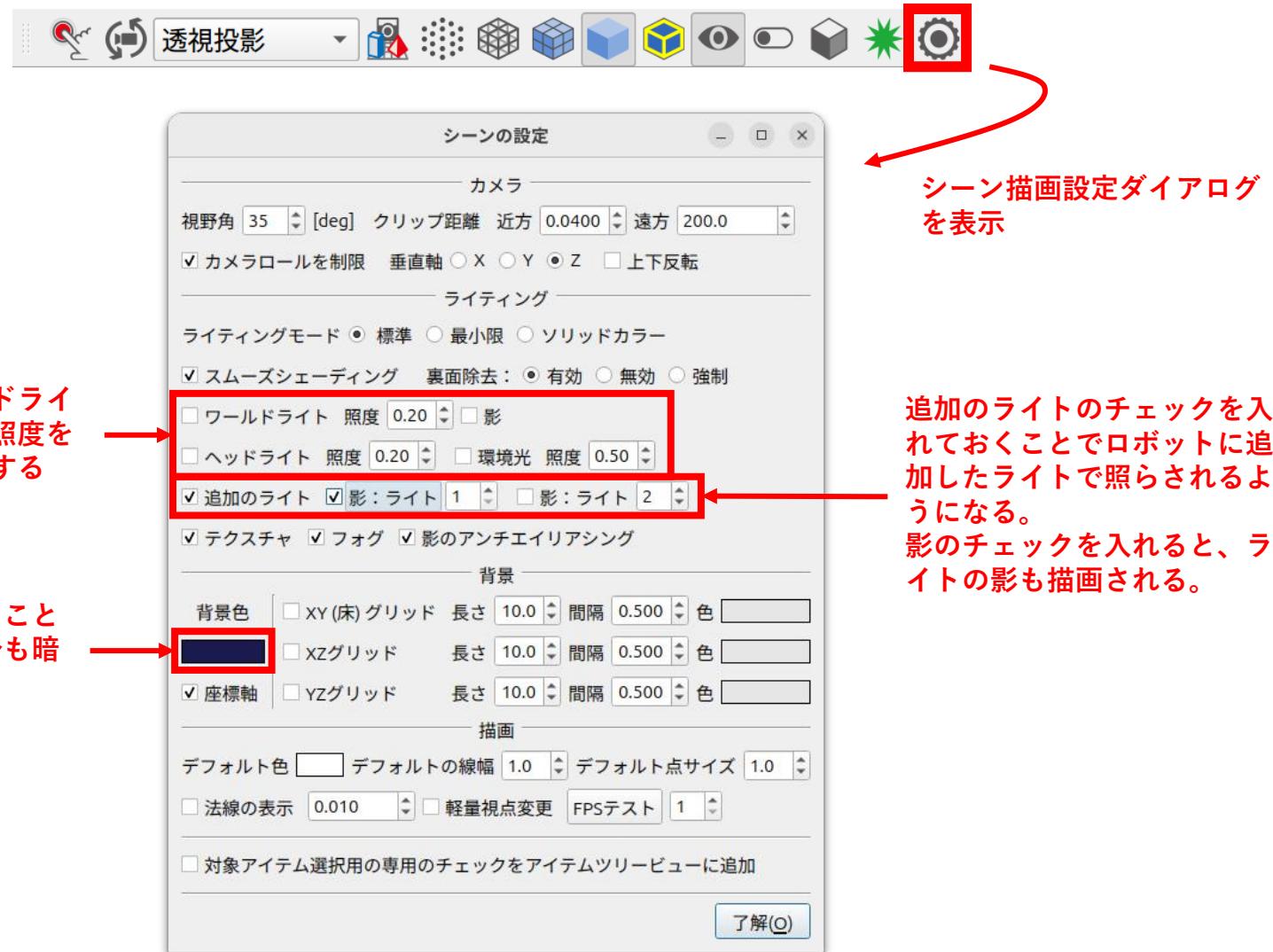
- ライトデバイスをモデルに追加
- シーンの設定で「追加のライト」を有効化
- 必要に応じてライトの状態を制御



ライトデバイスの記述例

```
- name: TiltLink
...
elements:
-
...
-
  type: SpotLight          SpotLight型を指定
  name: Light
  translation: [ 0.07, 0, 0 ]
  direction: [ 1, 0, 0 ]      照射方向
  beam_width: 36            最大輝度で光が照らされる角度
  cut_off_angle: 40         光が照らされる角度
  cut_off_exponent: 6       beam_widthからcut_off_angleの間の輝度の変化を決める値
  attenuation: [ 1, 0, 0.01 ]
  on: true
  elements:
  -
    type: Shape
    rotation: [ 0, 0, 1, 90 ]
    translation: [ -0.005, 0, 0 ]
    geometry: { type: Cylinder, height: 0.01, radius: 0.01 }
    appearance:
      material: { emissive: [ 0.8, 0.8, 0.3 ] }
```

光源の描画設定



視覚シミュレーションの設定

- ・シミュレーションで生成されるカメラ画像の描画設定はGLVisionSimulatorアイテムのプロパティで行う
 - ・背景色："0 0 0"（黒）
 - ・ヘッドライト: False（オフ）
 - ・ワールドライト: False（オフ）
 - ・追加のライト: True（オン）

お手本パッケージのファイル

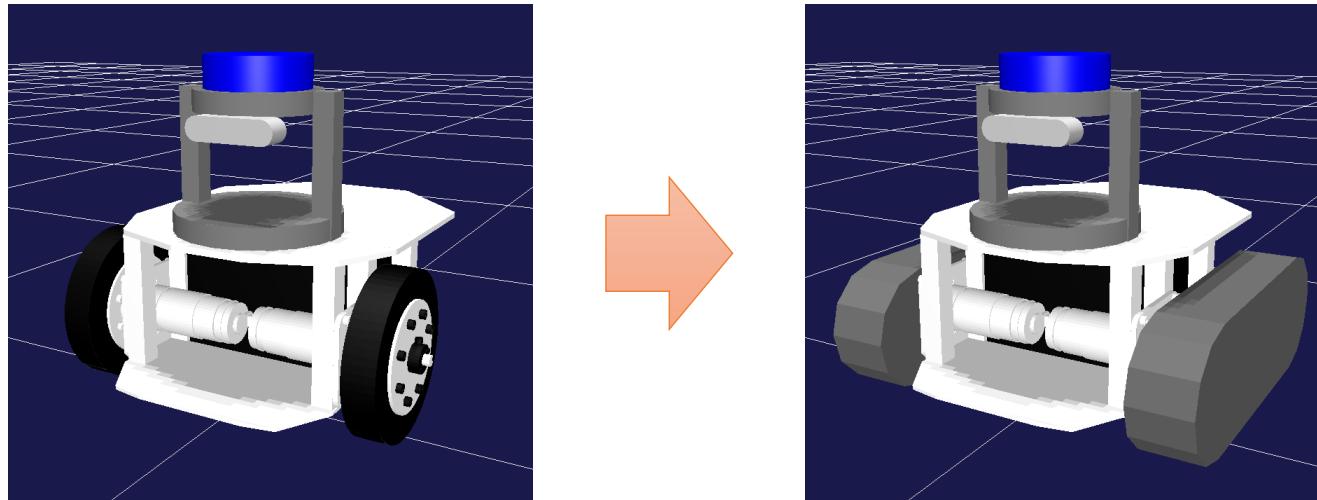
- ロボットのモデルファイル
 - model/mobile_robot_sensors_ex.body
- プロジェクトファイル
 - project/mobile_robot_sensors_ex_publish.cnoid
- Launchファイル
 - launch/sensors_ex_publish_launch.xml
 - launch/display_ex_launch.xml
 - config/mobile_robot_ex.rvizを使用
 - launch/sensors_ex_display_launch.xml

Part19

クローラの利用

クローラの利用

- ・災害対応ロボット等では、移動機構としてクローラ（無限軌道、continuous tracks）を用することも多い
- ・モバイルロボットの車輪をクローラに入れ替えて動かす



クローラの種類

- 簡易クローラ
 - 只の剛体
 - 環境との接触部分に推力を発生させる
- AGXクローラ
 - AGX Dynamicsプラグインで実現
 - 実物と同様に履帯をモデリング
 - 実物と同様の走破性を再現
- 本チュートリアルでは簡易クローラを使用

簡易クローラの記述 (1/3)

- LeftWheelについて、赤字の部分を修正して、左側のクローラを記述する

```
-  
  name: LeftTracks  
  parent: Chassis  
  translation: [ -0.08, 0.155, 0.06 ]  
  joint_type: pseudo_continuous_track      ※ この関節タイプとすることで、  
  joint_id: 0                                リンクが簡易クローラとなる  
  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: TankTracks  
  elements:  
    ...
```

簡易クローラの記述 (2/3)

- 左クローラの形状を記述する

```
elements:  
  - &TrackShape  
    type: Shape  
    geometry:  
      type: Extrusion  
      crossSection: [  
        -0.11, -0.06, 0.11, -0.06,  
        0.14, -0.052, 0.162, -0.03,  
        0.17, 0.0, 0.162, 0.03,  
        0.14, 0.052, 0.11, 0.06,  
        -0.11, 0.06, -0.14, 0.052,  
        -0.162, 0.03, -0.17, 0.0,  
        -0.162, -0.03, -0.14, -0.052,  
        -0.11, -0.06 ]  
      spine: [ 0, -0.03, 0, 0, 0.03, 0 ]  
      appearance:  
        material:  
          diffuseColor: [ 0.4, 0.4, 0.4 ]  
    
```

※ 形状はどのように記述してもよいが、ここでは Extrusion(押し出し形状)を用いる

※ 断面の2次元形状を記述
(x座標, z座標の並び)

※ 断面との交差点となる座標を
記述 (x, y, zの並び)

簡易クローラの記述 (3/3)

- RightWheelを修正して右クローラを記述する

```
-  
  name: RightTracks  
  parent: Chassis  
  translation: [ -0.08, -0.155, 0.06 ]  
  joint_type: pseudo_continuous_track  
  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: TankTracks  
  elements:  
    - *TrackShape
```

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

簡易クローラの制御

- Link::dq_target() に指令値をセットする
- 値は環境との接触点における相対速度
 - リンクローカル座標
- 指令値の速度を実現する力が接触点に発生する

※ AGXクローラの場合は、履帯を動かすホイールの回転軸を制御する

MobileRobotTrackController.cpp (1/4)

```
#include <cnoid/SimpleController>
#include <rclcpp/rclcpp.hpp>
#include <geometry_msgs/msg/twist.hpp>
#include <memory>
#include <thread>
#include <mutex>

class MobileRobotTrackController : public cnoid::SimpleController
{
public:
    virtual bool configure(cnoid::SimpleControllerConfig* config) override;
    virtual bool initialize(cnoid::SimpleControllerIO* io) override;
    virtual bool control() override;
    virtual void unconfigure() override;

private:
    cnoid::Link* trackUnits[2];
    rclcpp::Node::SharedPtr node;
    rclcpp::Subscription<geometry_msgs::msg::Twist>::SharedPtr subscription;
    geometry_msgs::msg::Twist command;
    std::unique_ptr<rclcpp::executors::StaticSingleThreadedExecutor> executor;
    std::thread executorThread;
    std::mutex commandMutex;
};

CNOID_IMPLEMENT_SIMPLE_CONTROLLER_FACTORY(MobileRobotTrackController)
```

※ 赤字はMobileRobotDriveControllerEx.cpp と異なる部分

MobileRobotTrackController.cpp (2/4)

```
bool MobileRobotTrackController::configure(cnoid::SimpleControllerConfig* config)
{
    node = std::make_shared<rclcpp::Node>(config->controllerName());

    subscription = node->create_subscription<geometry_msgs::msg::Twist>(
        "/cmd_vel", 1,
        [this](const geometry_msgs::msg::Twist::SharedPtr msg) {
            std::lock_guard<std::mutex> lock(commandMutex);
            command = *msg;
        });
}

executor = std::make_unique<rclcpp::executors::StaticSingleThreadedExecutor>();
executor->add_node(node);
executorThread = std::thread([this]() { executor->spin(); });

return true;
}
```

※ 車輪の場合と同様にTwist型のcmd_velトピックを制御指令の入力に使用します

MobileRobotTrackController.cpp (3/4)

```
bool MobileRobotTrackController::initialize(cnoid::SimpleControllerIO* io)
{
    auto body = io->body();
    trackUnits[0] = body->joint("LeftTracks");
    trackUnits[1] = body->joint("RightTracks");
    for(int i=0; i < 2; ++i) {
        auto trackUnit = trackUnits[i];
        trackUnit->setActuationMode(JointVelocity);
        io->enableOutput(trackUnit);
    }
    return true;
}

bool MobileRobotTrackController::control()
{
{
    std::lock_guard<std::mutex> lock(commandMutex);
    double v_angular = command.angular.z * trackUnits[0]->offsetTranslation().y();
    trackUnits[0]->dq_target() = command.linear.x - v_angular;
    trackUnits[1]->dq_target() = command.linear.x + v_angular;
}
return true;
}
```

※ 簡易クローラの場合はActuationModeを
"JointVelocity" としておきます

※ dq_target()に速度指令値をセットします。これは
通常は関節角速度を指令するためのものですが、簡易
クローラの場合は環境との接触点に発生する相対速度
になります。

MobileRobotPanTiltController.cpp (4/4)

```
void MobileRobotTrackController::unconfigure()
{
    if(executor) {
        executor->cancel();
        executorThread.join();
        executor->remove_node(node);
        executor.reset();
    }
}
```

※ お手本パッケージの完成版ファイルは
“src/MobileRobotTrackController.cpp”

srcのCMakeLists.txt

- src/CMakeLists.txtに以下を追記

```
choreonoid_add_simple_controller(  
    MyMobileRobotTrackController MobileRobotTrackController.cpp)  
  
ament_target_dependencies(  
    MyMobileRobotTrackController rclcpp geometry_msgs)
```

※ 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

ビルドしてMyMobileRobotTrackController.so
を生成する

クローラモデルの利用

- これまでのプロジェクトについて、ロボットモデルと移動用のコントローラをここで作成したものと入れ替える
- お手本パッケージの関連ファイル
 - モデル
 - model/mobile_robot_tracks.body
 - プロジェクト
 - project/mobile_robot_tracks.cnoid
 - Launchファイル
 - launch/tracks_launch.xml
 - launch/tracks_display_launch.xml

関連資料

- 公式マニュアル
 - Bodyファイルチュートリアル
 - <https://choreonoid.org/ja/documents/latest/handling-models/modelfile/modelfile-newformat.html>
 - 無限軌道の簡易シミュレーション
 - <https://choreonoid.org/ja/documents/latest/simulation/pseudo-continuous-track.html>
 - AGX Dynamics プラグイン
 - <https://choreonoid.org/ja/documents/latest/agxdynamics/index.html>
 - AGXVehicleContinuousTrackの特徴
 - <https://choreonoid.org/ja/documents/latest/agxdynamics/agx-continuous-track.html>