

# 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

2024/5/14

# Part1

概要

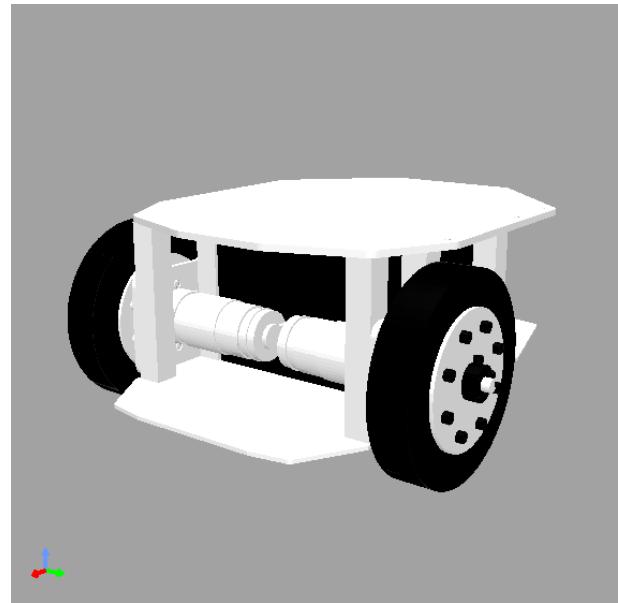
# チュートリアルの目的

- 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

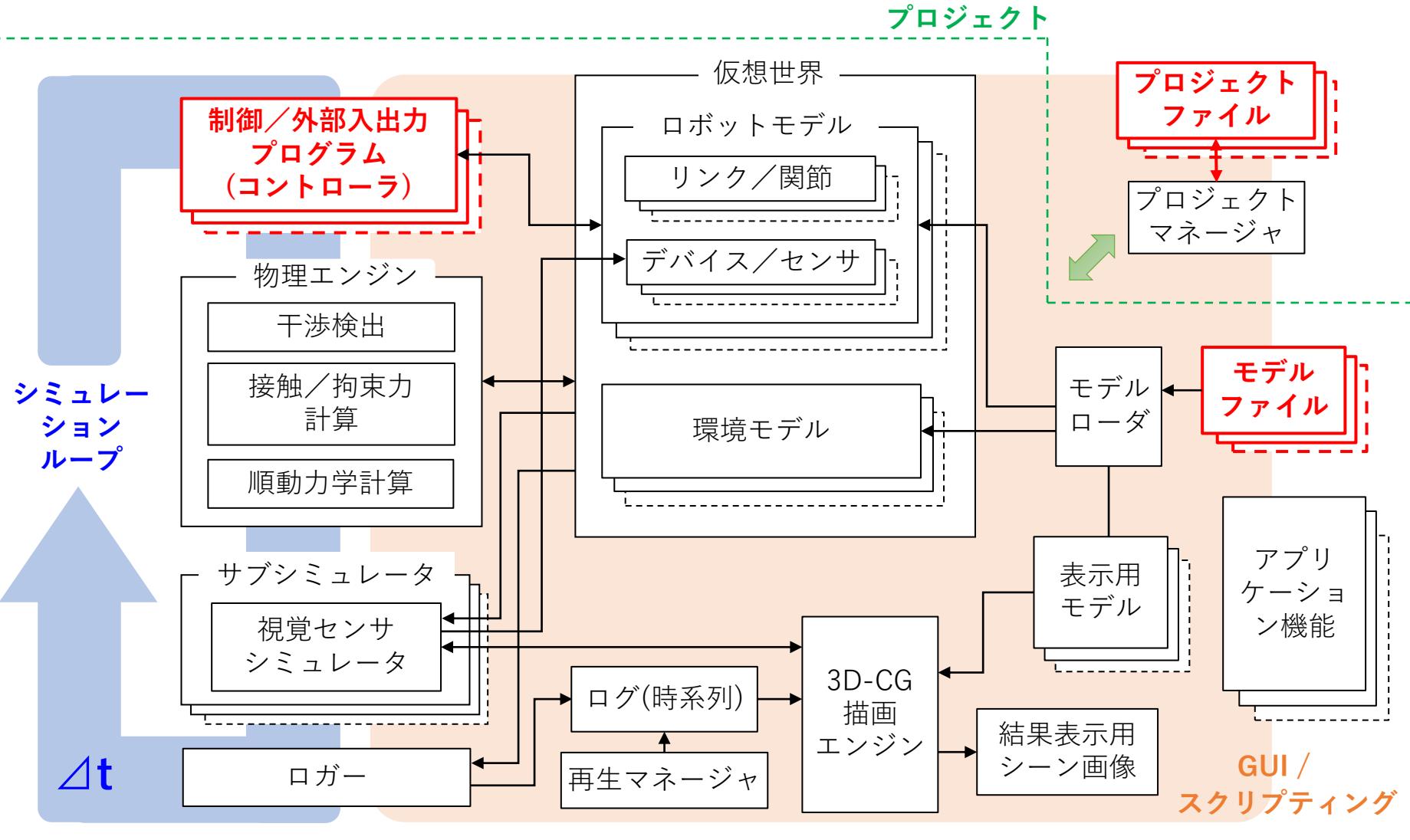
- 環境構築
- モデルの作成
- 制御
- 視覚センサの導入
- 状態の可視化
- 遠隔操作



# 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_ros)
  - [https://github.com/choreonoid/choreonoid\\_ros2\\_mobile\\_robot\\_tutorial](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 (Humble)
  - <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 22.04
- Choreonoid 2.1.1 もしくは開発版
- ROS 2 (Humble)
- 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

- CPU
  - インテルCoreプロセッサ第13世代以降ならまず問題なし
  - 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以上
  - 並列ビルドする場合は並列数に見合うメモリ容量
- GPU
  - CPU内蔵／外付けとも、最近のものなら問題なし
- ディスプレイ解像度
  - フルHD以上

# 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

```
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
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

# ROS 2ワークスペースの作成

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

# 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_ros_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

# 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

# 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
```

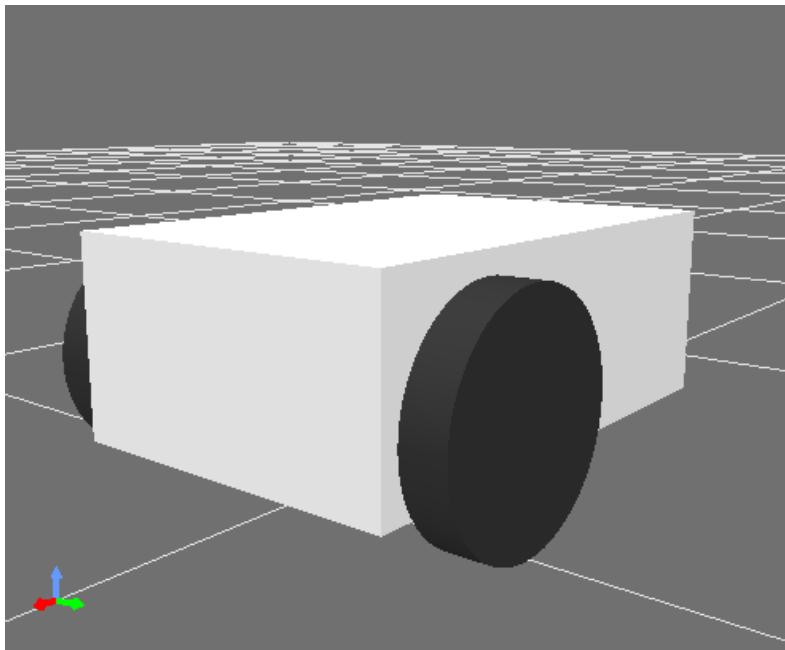
“~/ros2\_ws/src” 以下に “my\_mobile\_robot” というディレクトリが作成されるので、その中に実習のファイルを作成していく

# Part3

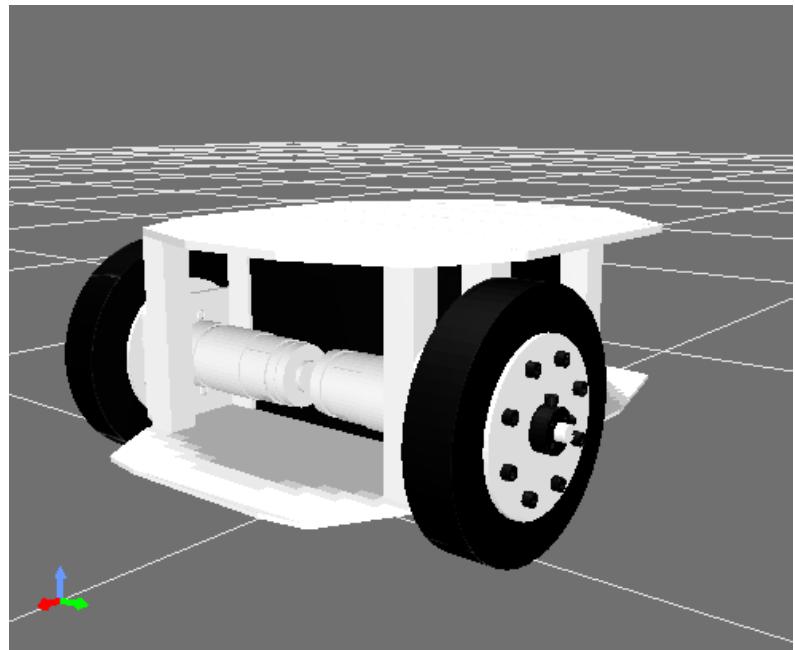
モデルの作成とシミュレーション

# 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

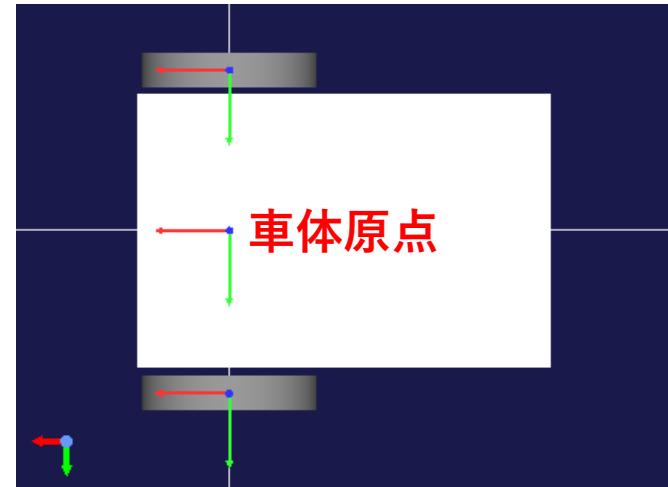
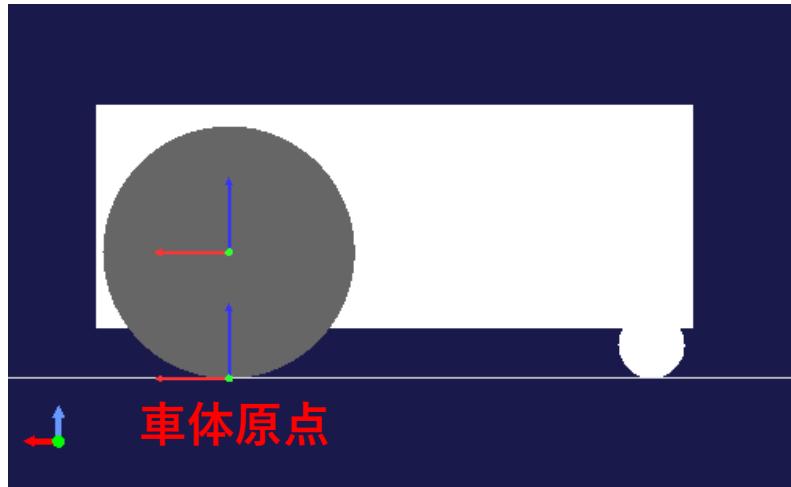
※ ヴィストン社 メガローバー2.1

# 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

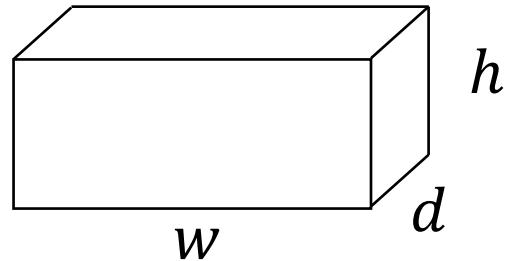
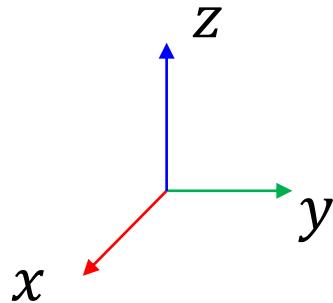
links:
-
  name: Chassis
  joint_type: free
  center_of_mass: [ -0.08, 0, 0.08 ]
  mass: 14.0
  inertia: [ 0.1,    0,      0,
             0,      0.17,   0,
             0,      0,      0.22 ]
  material: Slider
elements:
-
  type: Shape
  translation: [ -0.1, 0, 0.0975 ]
  geometry:
    type: Box
    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)

# 慣性行列の計算

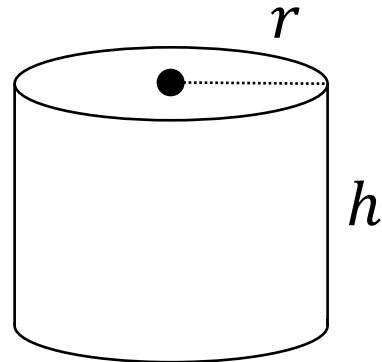
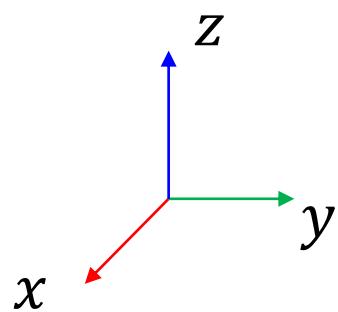
- CADツールを利用
  - プリミティブ形状に関する式から算出
- ※ 自動計算機能は未実装

# 直方体(Box)の慣性行列



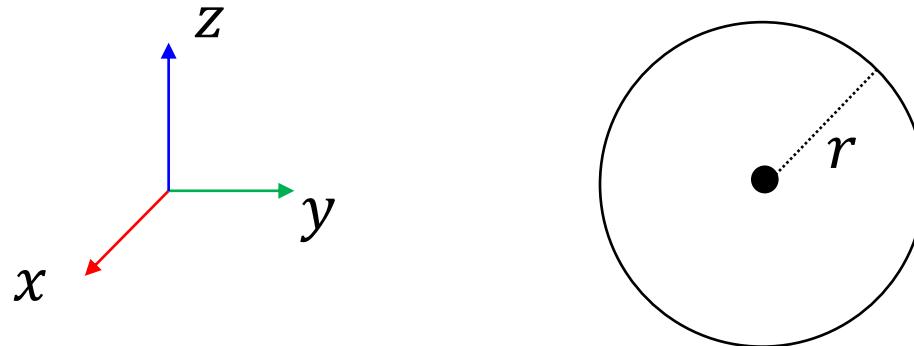
$$I = \begin{pmatrix} \frac{1}{12}m(w^2 + h^2) & 0 & 0 \\ 0 & \frac{1}{12}m(d^2 + h^2) & 0 \\ 0 & 0 & \frac{1}{12}m(w^2 + d^2) \end{pmatrix}$$

# 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



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

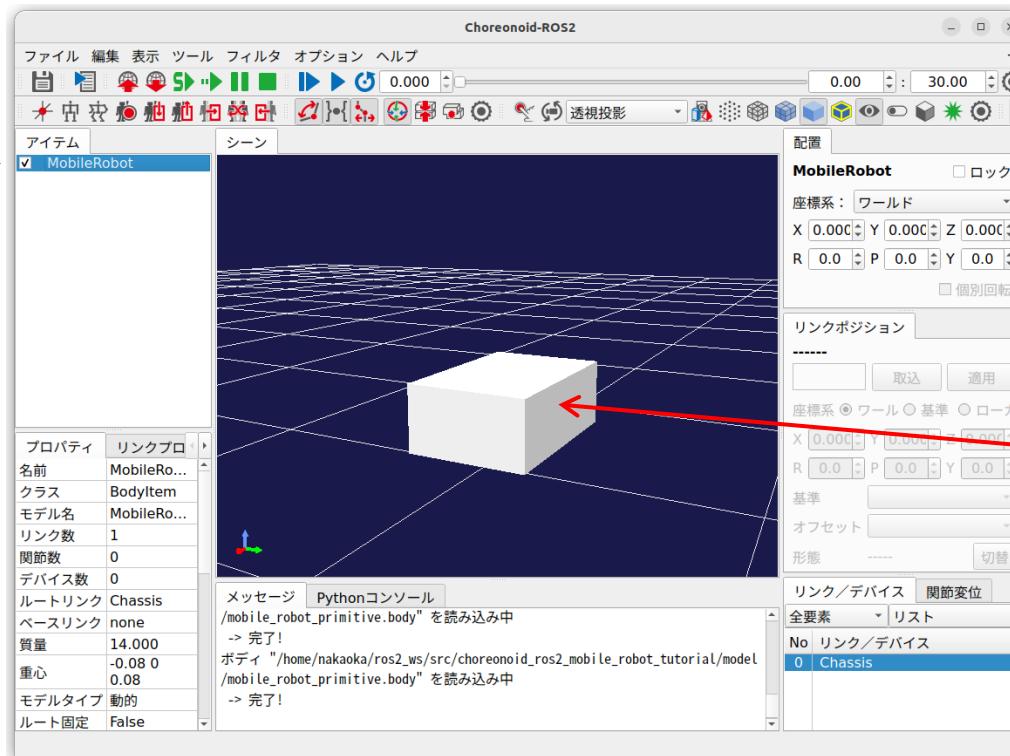
# モデルの読み込み

- 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

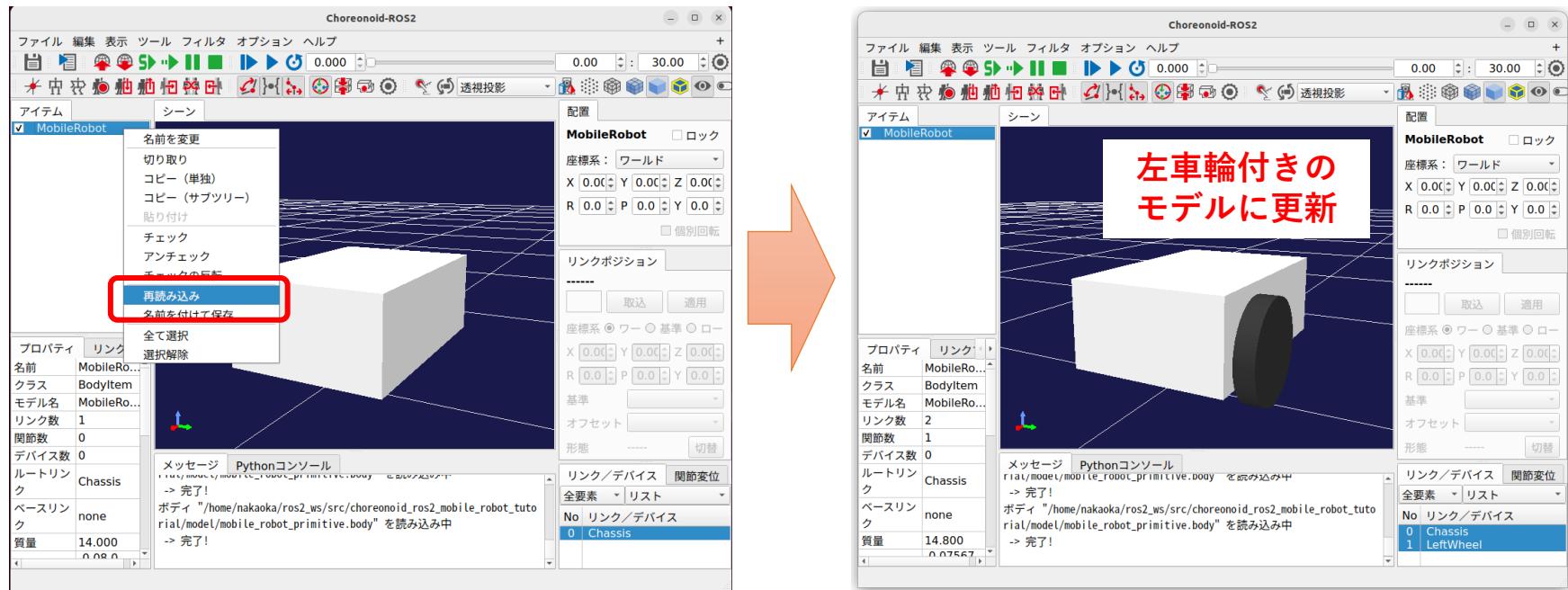
“mobile\_robot.body” に以下を追記する

```
-  
  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 ]
```

※ インデントに注意！  
(ここではスペース2文字を使用)

# 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
```

※ インデントに注意！

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

# 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」—「新規」—「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"0 0 0"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つ
  - 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つ
  - 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

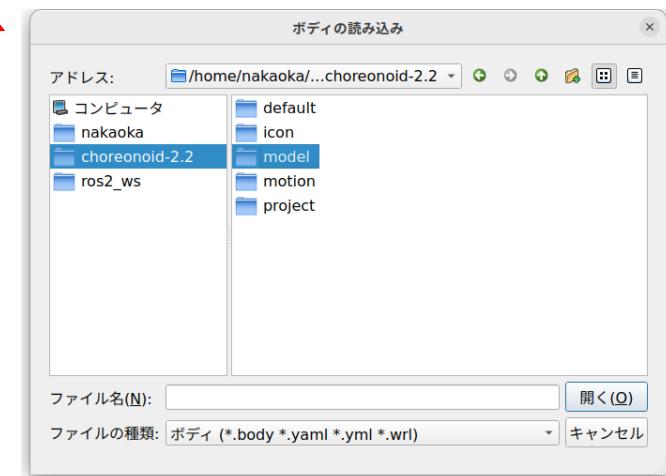
- 産総研物理エンジン
  - 標準エンジン
  - AISTシミュレータアイテム
- Open Dynamics Engine
  - オープンソースで広く利用されている物理エンジン
  - ODEプラグインで導入
  - ODEシミュレータアイテム
- AGX Dynamics
  - 商用物理エンジンで高機能・高性能
  - ライセンスの購入が必要
  - AGX Dynamicsプラグインで導入
  - AGXシミュレータアイテム
- 実験的対応のある物理エンジン
  - NVIDIA PhysX, Bullet Physics, Springhead, Roki

# 床の追加

- 床モデルを読み込む
  - choreonoid-2.x/model/misc/floor.body

+ World  
  MobileRobot  
  **Floor**  
  AISTSimulator

ココ



※ 親子関係に注意！

- 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

# 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"
```

( uri: "package://my\_mobile\_robot/meshes/vmega\_body.dae" と書いてもOK )

メッシュファイル"vmega\_body.dae"は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) にコピーします

※ [https://github.com/vstoneofficial/megarover\\_samples](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: 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

※ “vmega\_wheel.dae”と“r5.png”もお手本からコピーしておく

※ 完成版はお手本パッケージの“model/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

# Part4

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
  - ros2\_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++を用いて自分で実装
  - 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
- ROS2Controlアイテム
  - ros2\_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」 - 「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

- src/MobileRobotDriveTester.cppを作成
- package.xmlを編集
- ビルド用のCMakeLists.txtの修正・追加
- 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("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</depend> ← 追加
```

```
...
```

これにより、Choreonoid本体に依存するパッケージであることを示すことができる（今回のコントローラはROS2は使用しないので、Choreonoid本体のみに依存）

# トップのCMakeLists.txt

パッケージ初期化時に生成された雛形に以下を追加する

```
# 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”

# 補足：アイテム名の修正

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

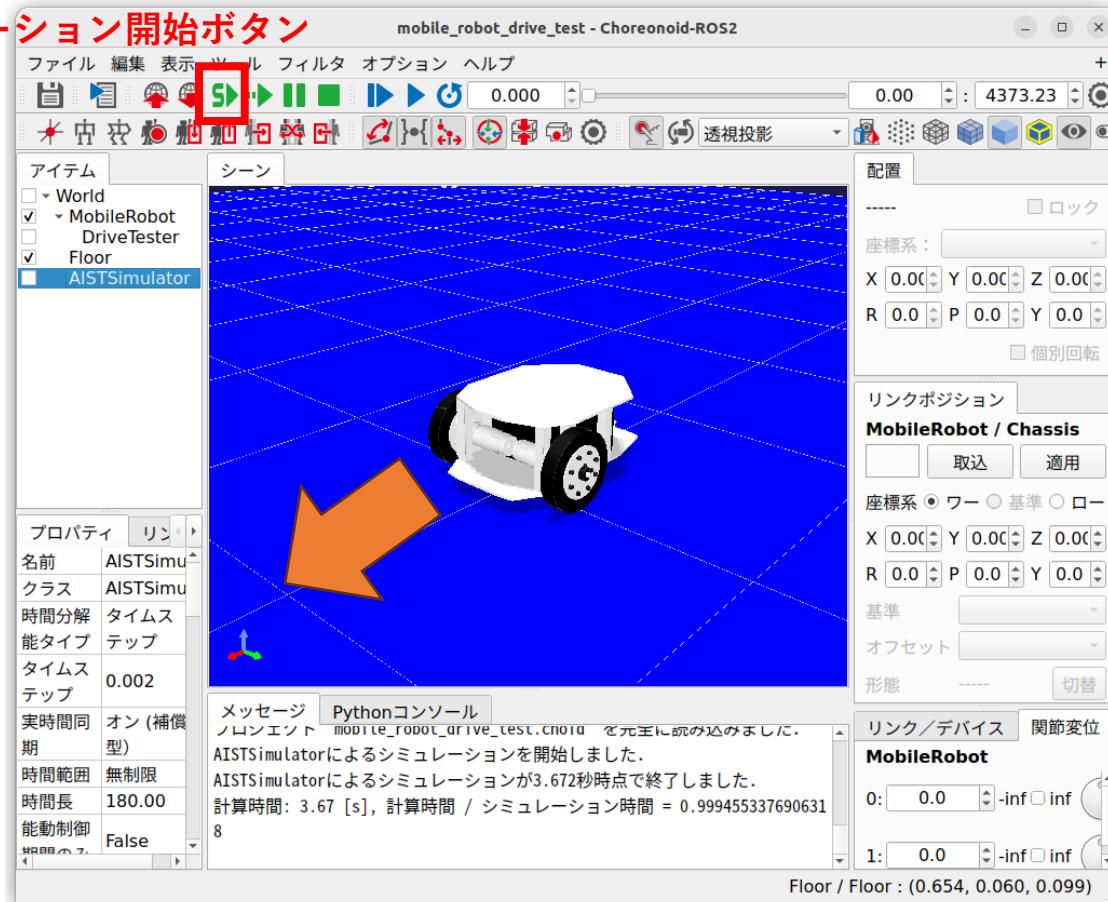
```
+ World
  + MobileRobot
    DriveTester
    Floor
    AISTSimulator
```

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”

# 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）

# Part5

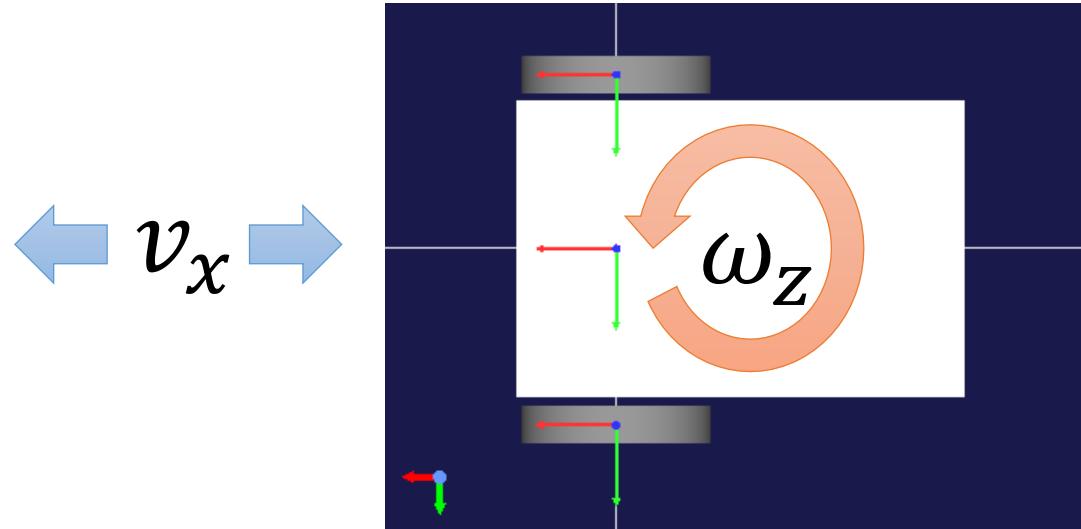
ROS通信を用いた制御

# 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( $\omega_z$ )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

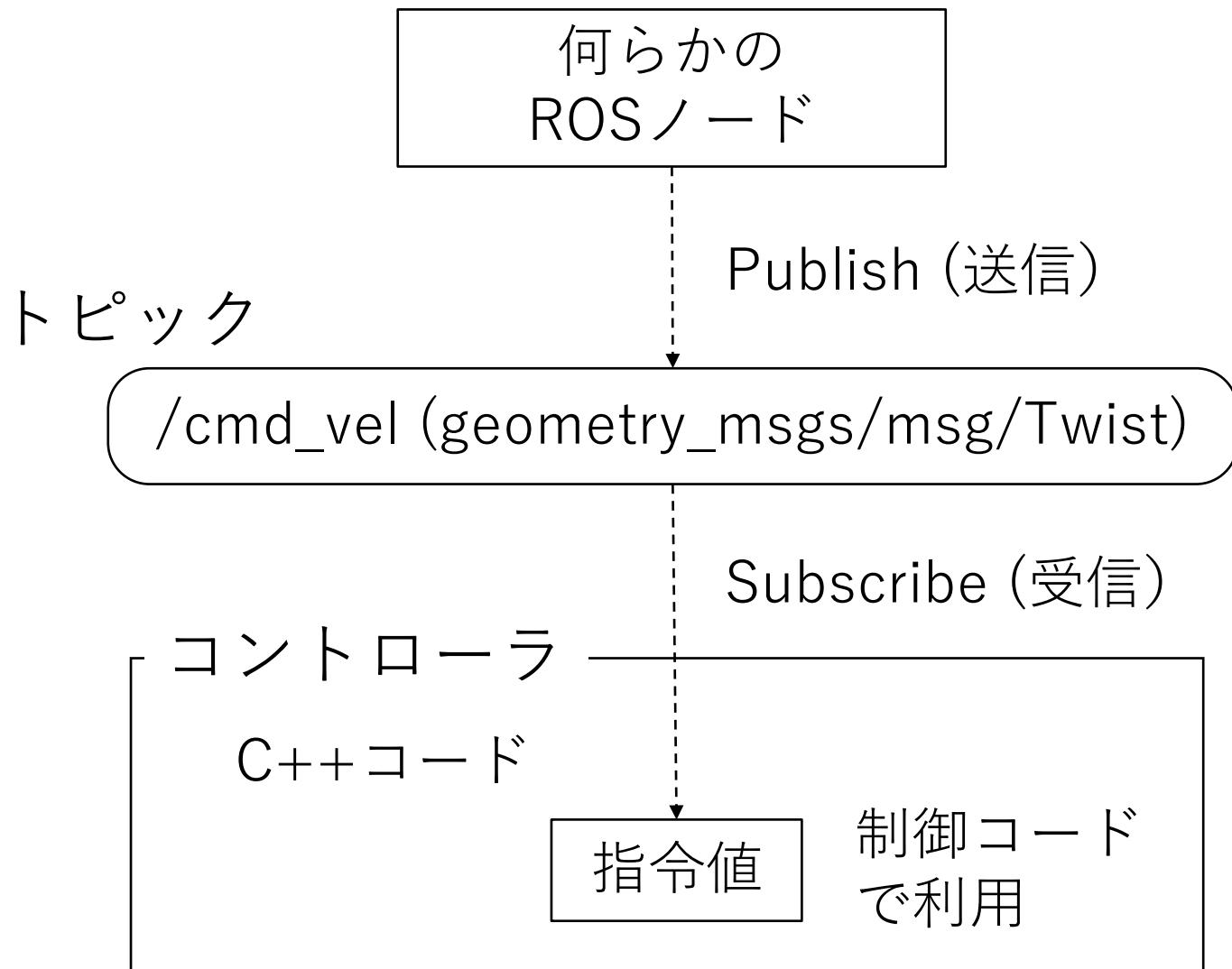
```
ros2 interface show geometry_msgs/msg/Twist
```

```
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



# コントローラの作成

- package.xmlを編集
- 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
  - お手本リポジトリの該当プロジェクトファイルは“mobile\_robot\_drive\_control.cnoid”になります

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

<exec_depend>joy</exec_depend>
<exec_depend>joy_teleop</exec_depend>
<exec_depend>rqt_robot_steering</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 --from-paths  
~/ros2_ws/src/choreonoid_ros2_mobile_robot_tutorial  
-y --ignore-src
```

package.xmlに記述した

- joy
- joy\_teleop
- rqt\_robot\_steering
- 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

# 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;
}
```

# 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

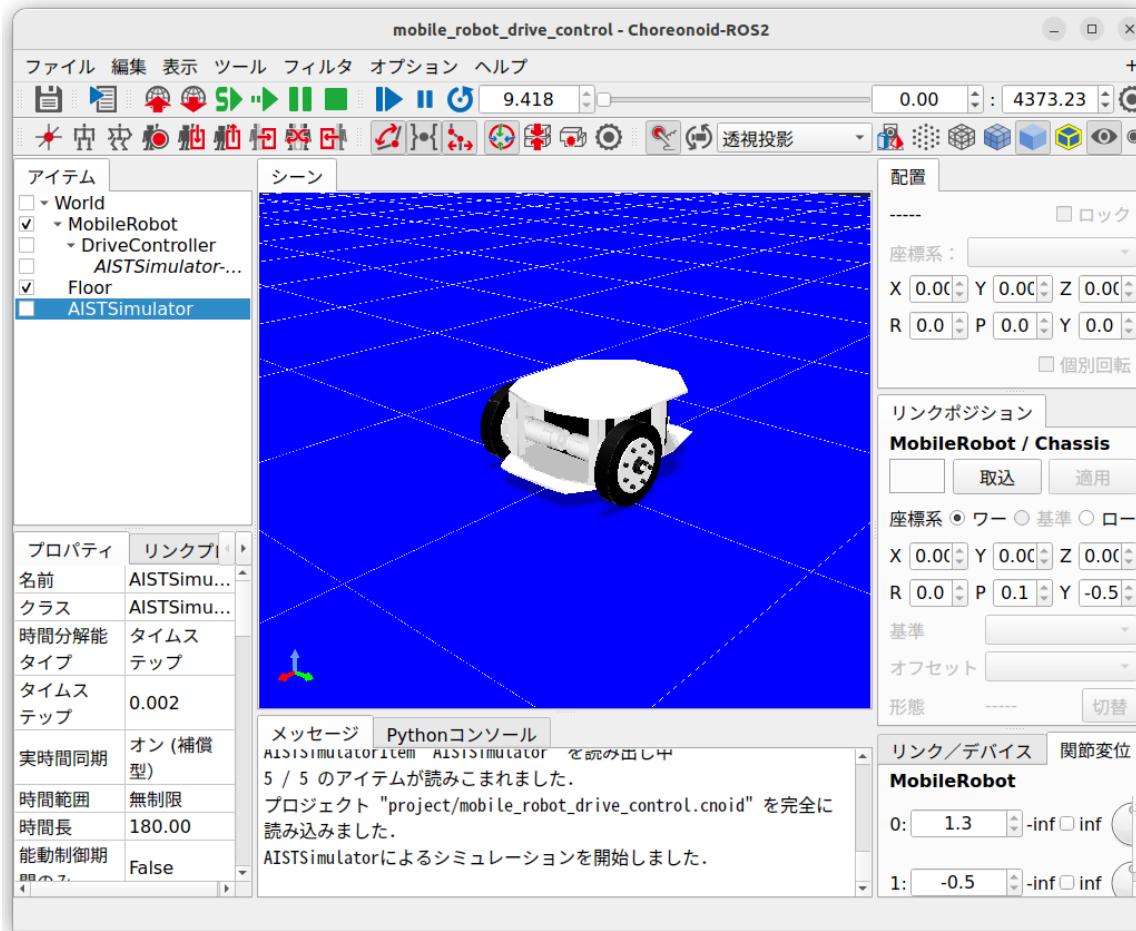
# プロジェクトの構成

- + 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”

# 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

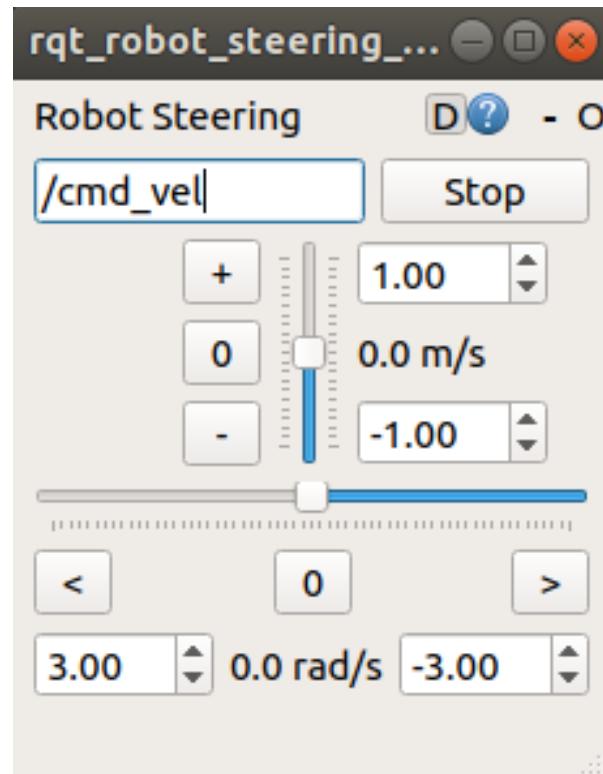
※ コロン(:)の後にスペースが必要！

publishされている内容を確認

```
ros2 topic echo /cmd_vel
```

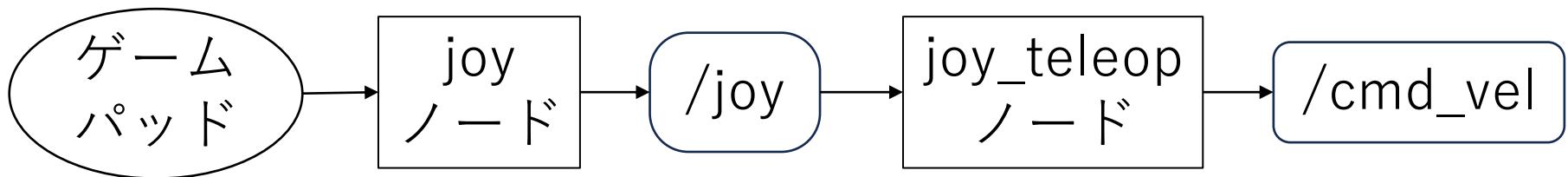
# rqt\_robot\_steering による操作

```
ros2 run rqt_robot_steering rqt_robot_steering
```



# 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



# 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
```

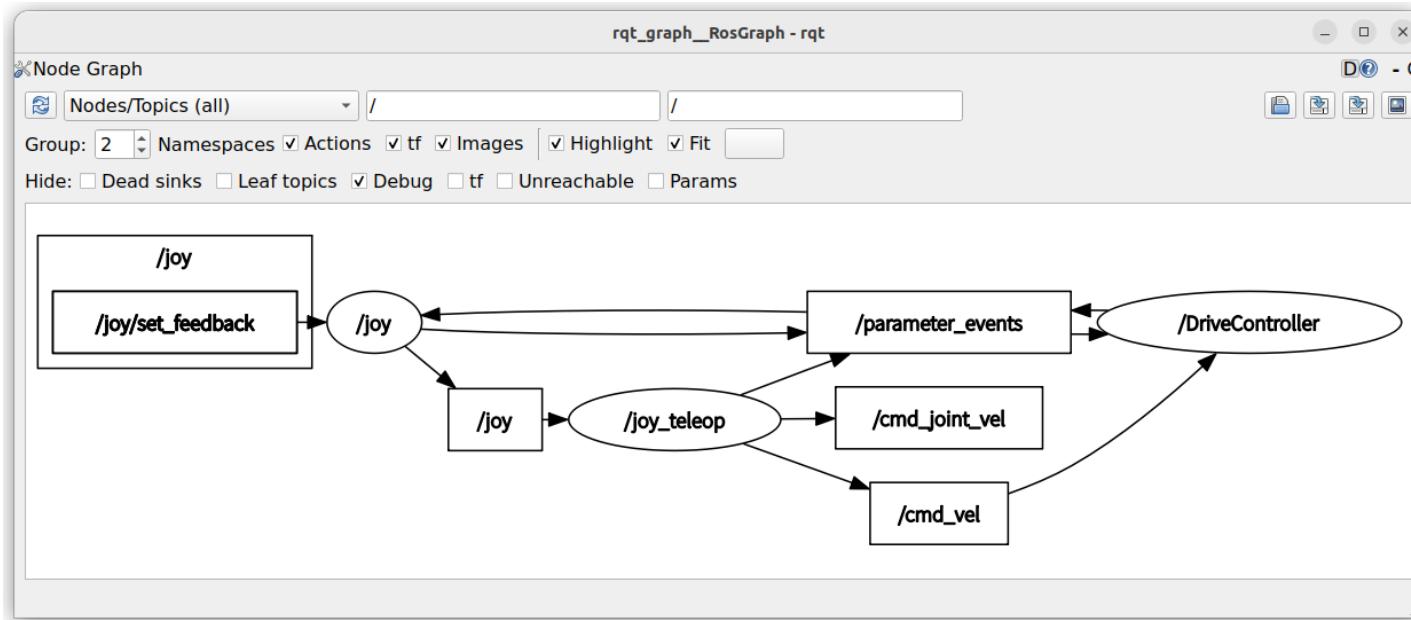
シミュレーション中のロボットを  
ゲームパッドで操作する

# 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
```



# 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 (1/2)

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_all.yaml"/>
  </node>
</launch>
```

“launch/joy\_teleop\_launch.xml”として保存する

以下のコマンドで起動

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

# Launchファイルの作成 (2/2)

“launch/drive\_control\_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_drive_control.cnoid” />
</launch>
```

※ 実際のプロジェクトファイルを指定

以下のコマンドで起動

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

※ joy\_teleop\_launch.xml も同時に起動

# 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

# 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
- スレッドの話を深く話す
- control関数がスレッドになる得ることについても
- 模式図を書くとよいかも
  - シミュレータのスレッドとかも含むような

# MobileRobotDriveController.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)
```

# MobileRobotDriveController.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;
}
```

# MobileRobotDriveController.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;
}
```

# MobileRobotDriveController.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;
}
```

# MobileRobotDriveController.cpp (5/5)

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

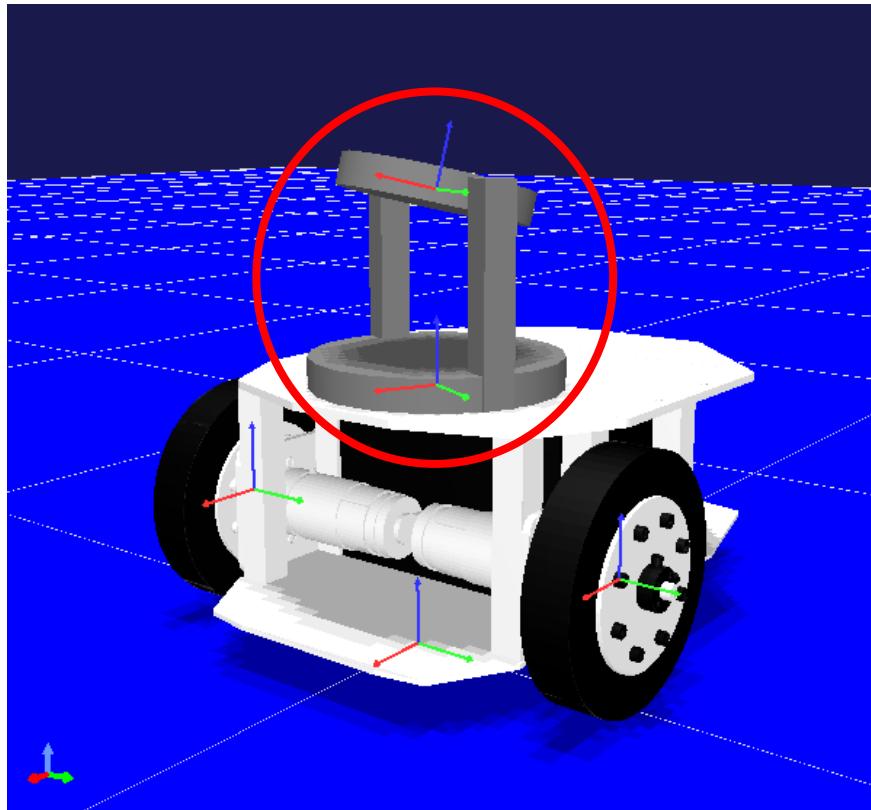
※ お手本プロジェクトの完成版ファイルは  
“src/MobileRobotDriveControllerEx.cpp”

# Part6

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 ]
```

※ インデントに注意！

# 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:
      -
        type: Shape
        geometry: { type: Box, size: [ 0.02, 0.02, 0.13 ] }
        appearance: *GRAY

  -
    type: Transform
    translation: [ 0, -0.07, 0.065 ]
    elements: *PanFrame
```

※ インデントに注意！

# 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
```

※ インデントに注意！

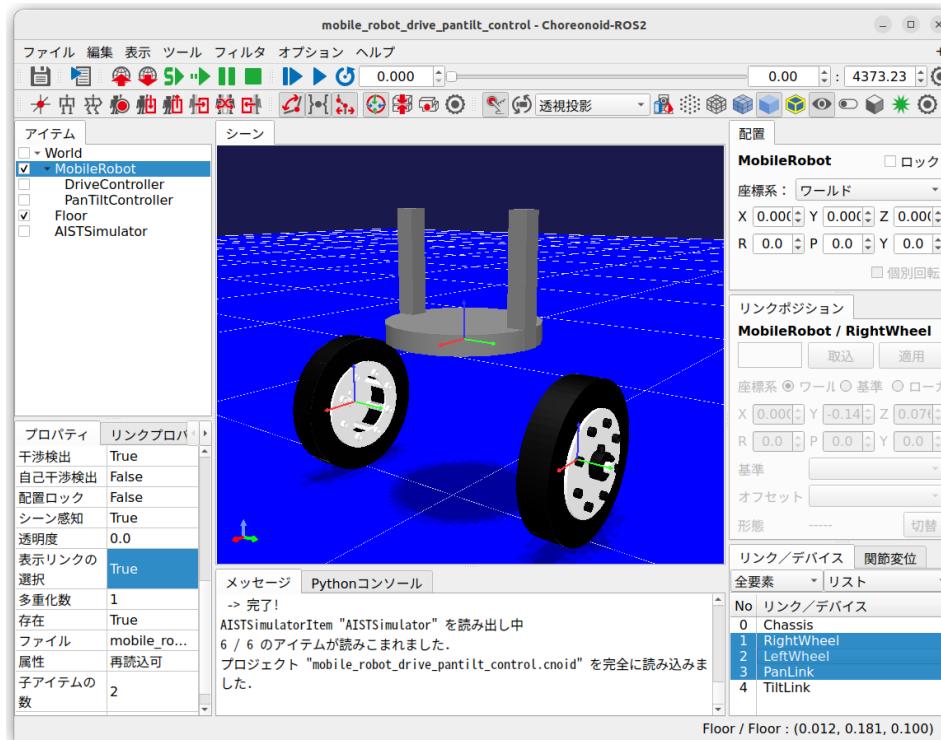
※ お手本パッケージの完成版は “model/mobile\_robot\_pantilt.body”

# 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
- ・AIIST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



# Part7

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
  - $\omega_z$  : パン軸の目標角速度
  - $\omega_y$  : 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MobileRobotDriveController.cppとは異なる部分

# 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) {
        cnoid::Link* 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();
    }
}
```

※ お手本プロジェクト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

# 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

※ 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”

# 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

config/joy\_teleop.yaml に追記する

```
joy_teleop:  
  ros_parameters:  
    ...  
  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
```

※ 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”

# joy\_teleop ノードの起動

※ joy ノードも起動しておく

```
ros2 run joy joy_node
```

```
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
```

シミュレーション中のロボットの車体もパンチルトも  
ゲームパッドで操作できる

# 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

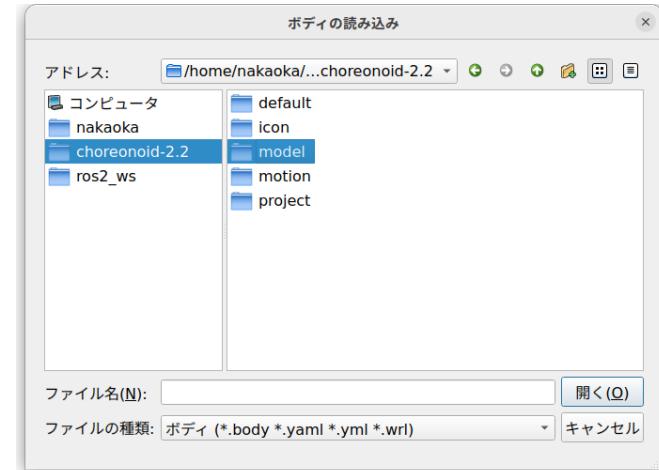
# Part8

視覚センサの追加

# 環境モデルの導入

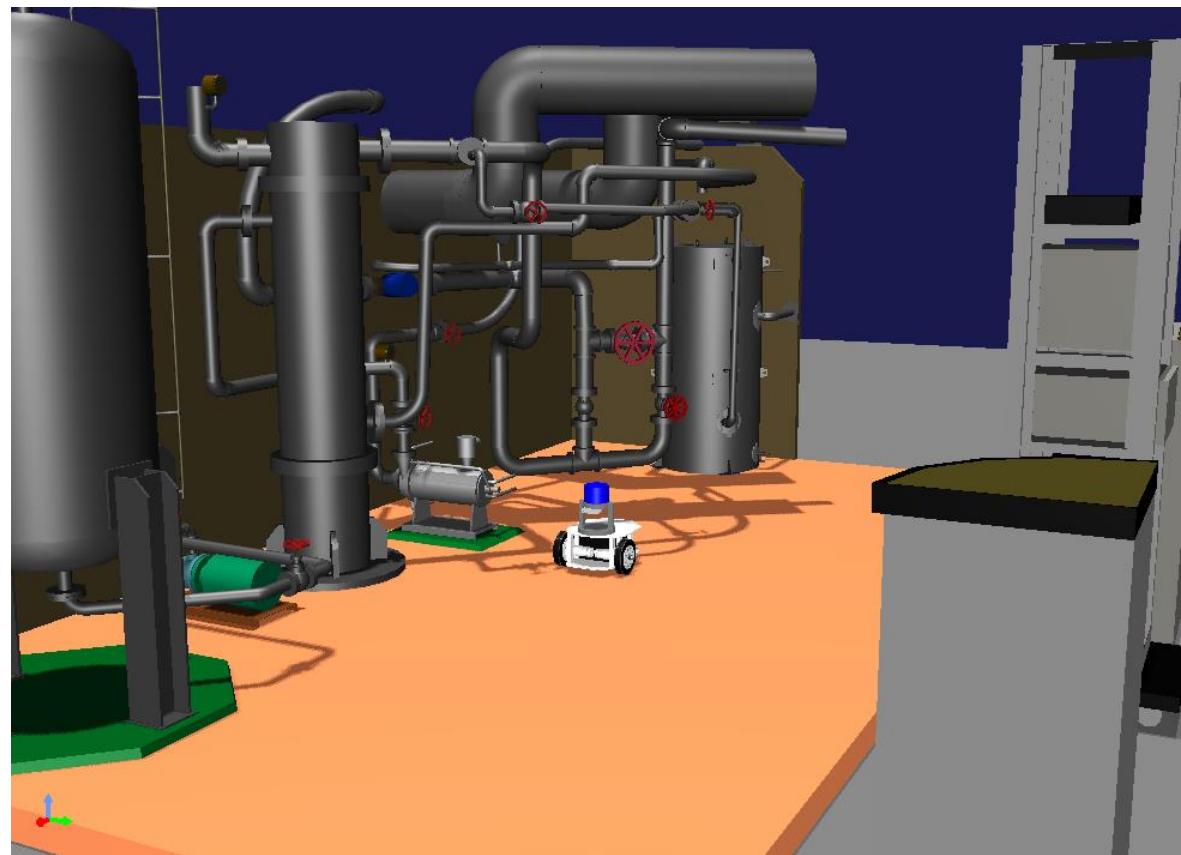
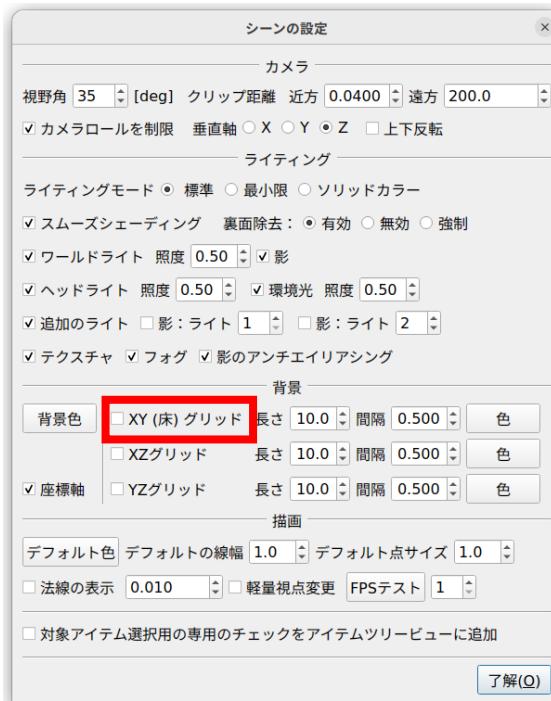
- 床モデル (floor) を削除
- 研究プラントのモデル (Labo1) を読み込む
  - “choreonoid-2.x/model/Labo1/Labo1v2.body”

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



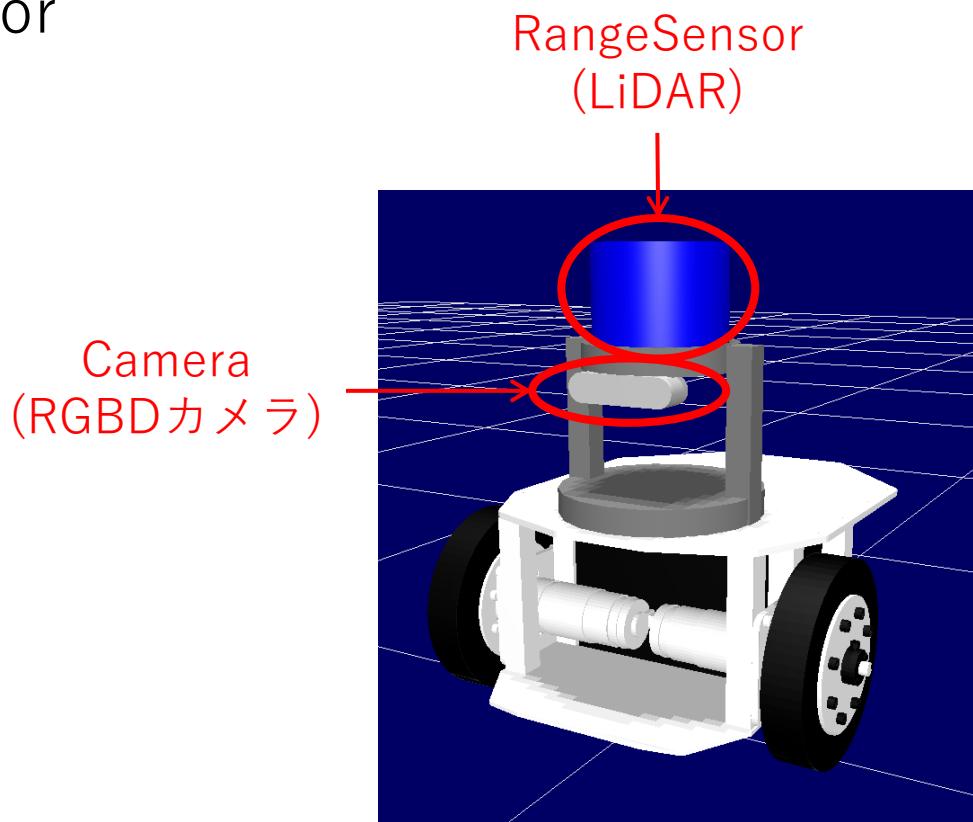
# シーン描画設定

- XY(床)グリッドのチェックを外す



# センサの追加

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



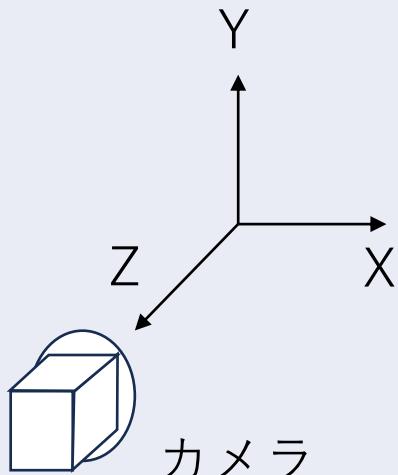
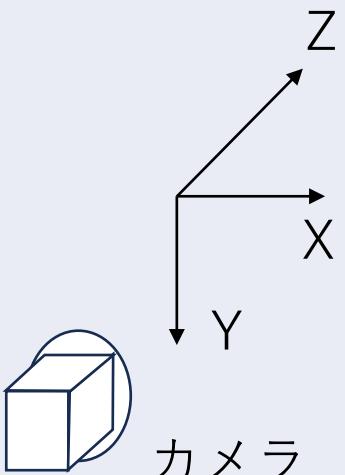
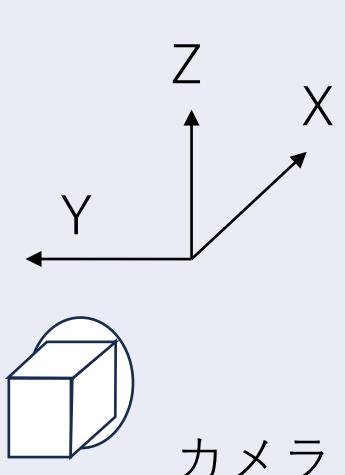
# レンジセンサ(LiDAR)の追加

```
-  
  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	 カメラ	 カメラ	 カメラ

# RGB-Dカメラの追加 (1/2)

```
- name: TiltLink
...
elements:
-
  type: Shape
...
-
  type: Transform
...
-
  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
```

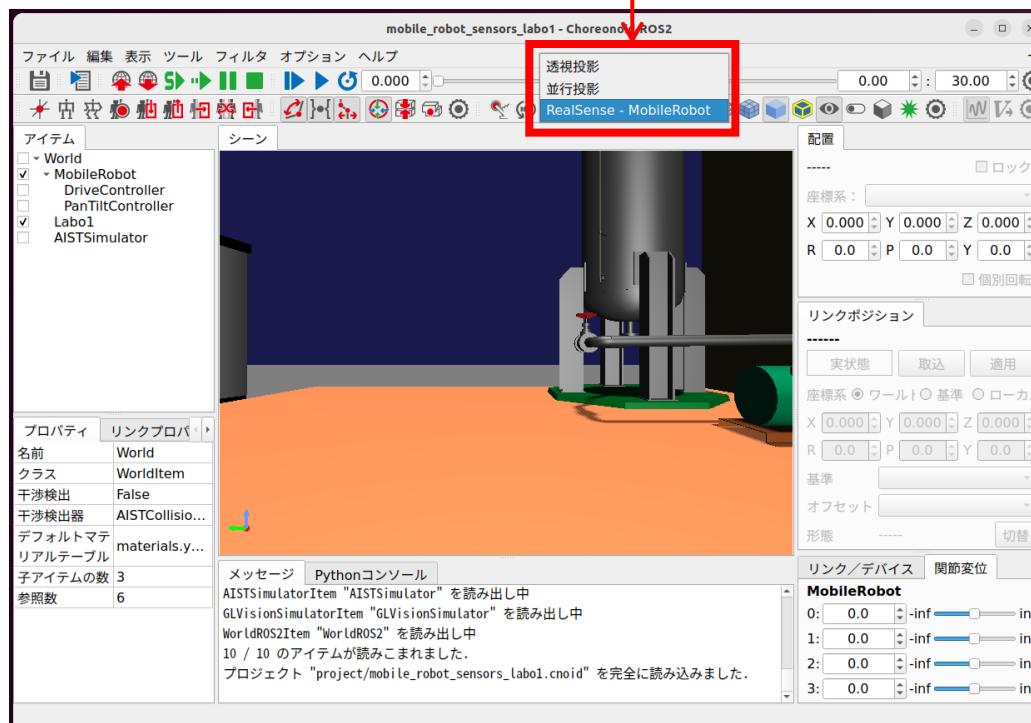
# 視覚センサ情報の可視化

- 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

カメラ選択コンボ



# 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

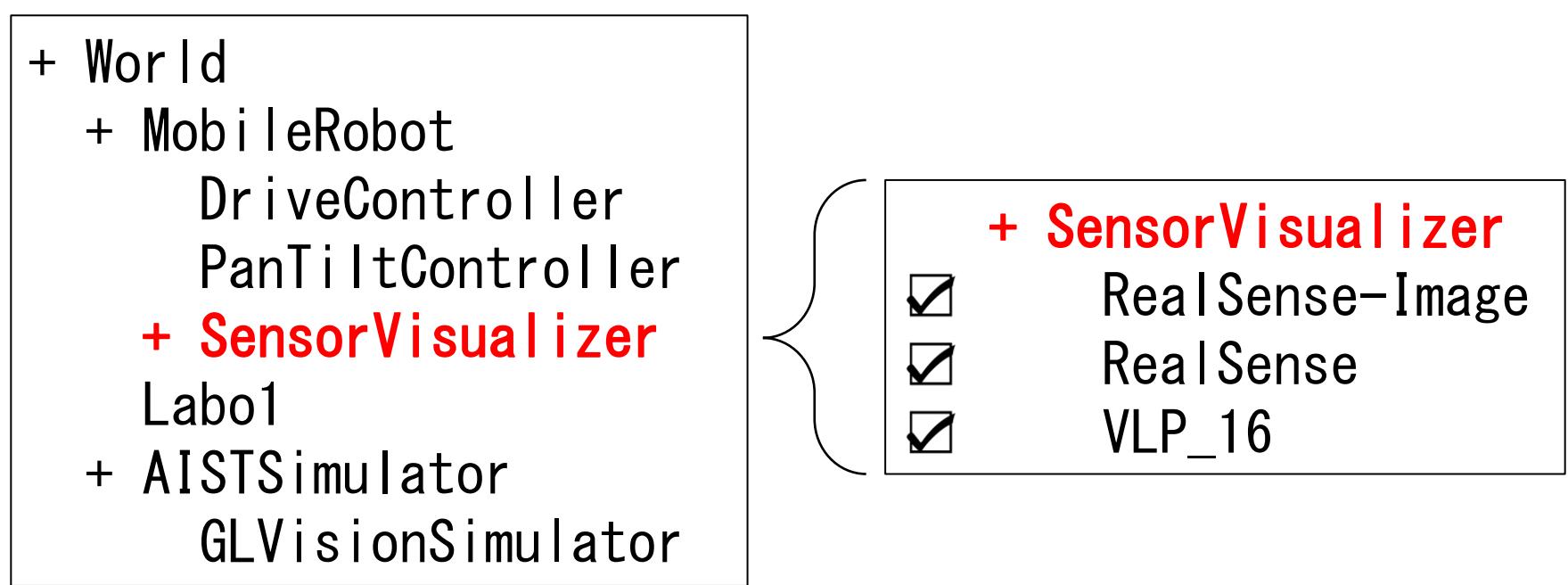
- Chreonoid上で表示
- Chreonoid外部で表示
  - 後ほどROS通信、ROSツールを用いて実現する

# 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)

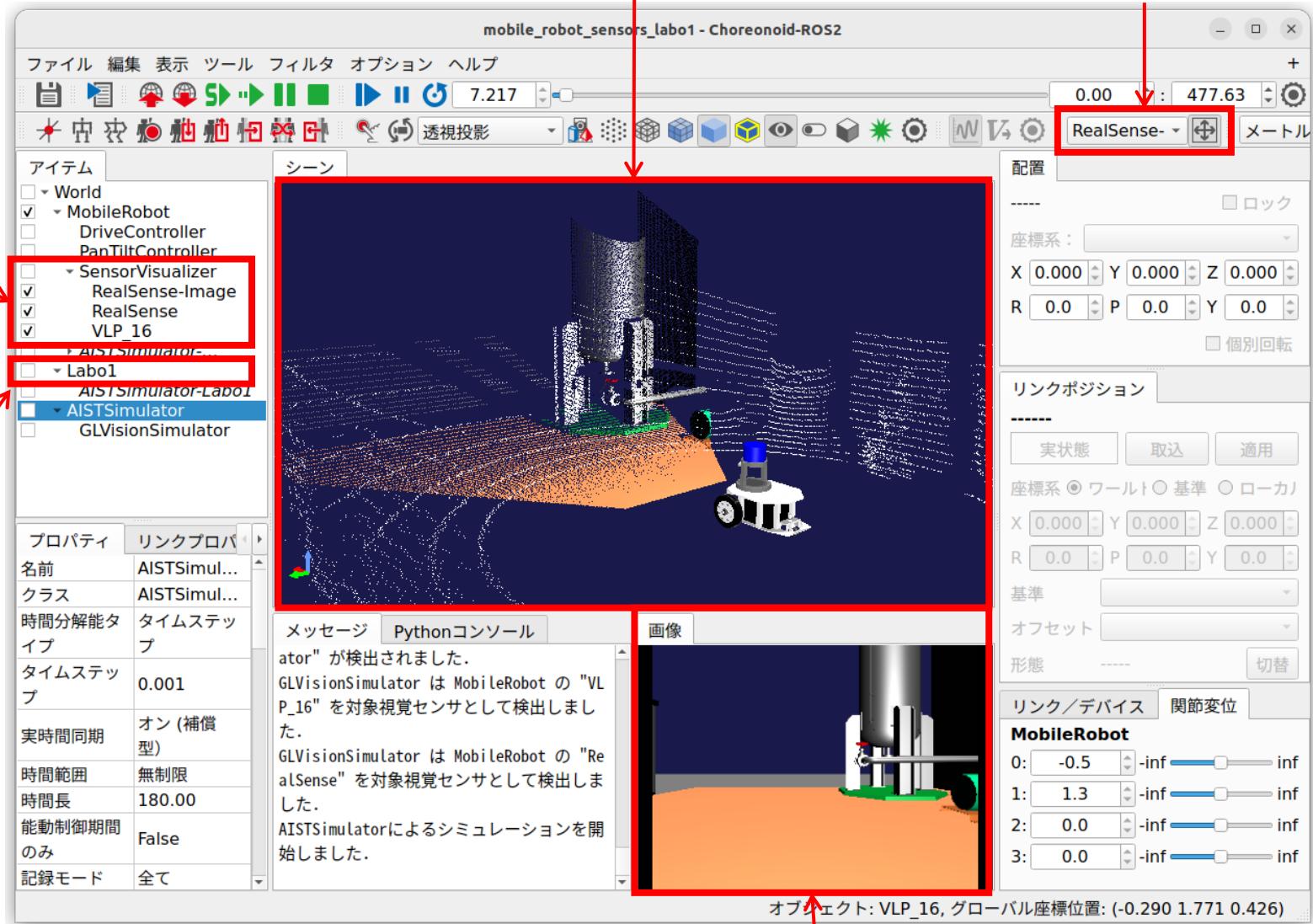
- カメラ画像については「画像ビュー」で表示可能
- 「画像ビューバー」の選択コンボで対象のセンサを選択する

## センサデータの可視化

## 画像選択コンポ

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

# 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\_lab01.cnoid”
- launchファイル
  - “launch/sensors\_lab01\_launch.xml”

# 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://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"/>
    ...>
    </inertial>
  </link>
</robot>
```

※ 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
  - 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
    - BodyROS2**
  - + SensorVisualizer
  - Labo1
  - + AISTSimulator
  - GLVisionSimulator

# 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

# 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（ポイントクラウド）
  - /MobileRobot/VLP\_16/point\_cloud

# 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/RealSense [sensor_msgs/msg/Image]  
/MobileRobot/RealSense/point_cloud [sensor_msgs/msg/PointCloud2]  
/MobileRobot/VLP_16/point_cloud [sensor_msgs/msg/PointCloud2]  
/MobileRobot/joint_states [sensor_msgs/msg/JointState]
```

シミュレーションを開始して、joint\_statesの中身を確認する

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

# センサOn/OffのROSサービス

- 各センサごとに “set\_enabled” というサービスを提供
- ture、falseの引数でコールするとオン／オフを切り替えられる

```
ros2 service /MobileRobot/RealSense  
set_enabled std_srvs/srv/SetBool  "{data: false}"
```

# 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**

# Clock トピック

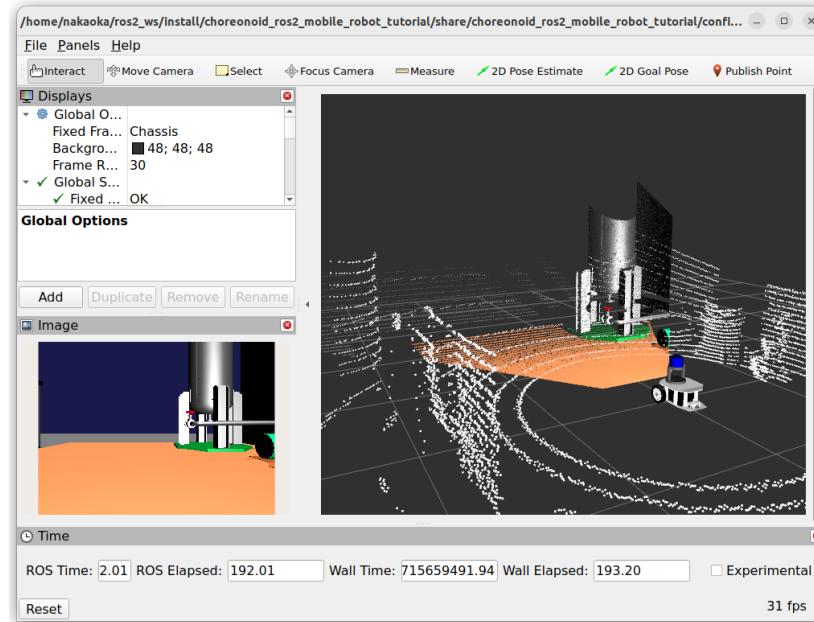
- “/clock” トピック
- 他のROSノードと時刻を共有（同期）
- WorldROS2アイテムがPublish
- “rosgraph\_msgs/msg/Clock”型
- 利用方法
  - 時刻を必要とするノードのROSパラメータ “/use\_sim\_time” をtrueにセットする

# 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\_lab01.cnoid”
- launchファイル
  - “launch/sensors\_lab01\_launch.xml”

# 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.xml”

```
<launch>
  <arg name="model"
    default="$(find-pkg-share my_mobile_robot)/model/mobile_robot_sensors.urdf"/>
  <arg name="rvizconfig"
    default="$(find-pkg-share my_mobile_robot)/config/mobile_robot.rviz"/>

  <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"/>
  </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"/>
</launch>
```

rvizの設定ファイル  
も必要  
(お手本パッケージ  
に入っています)

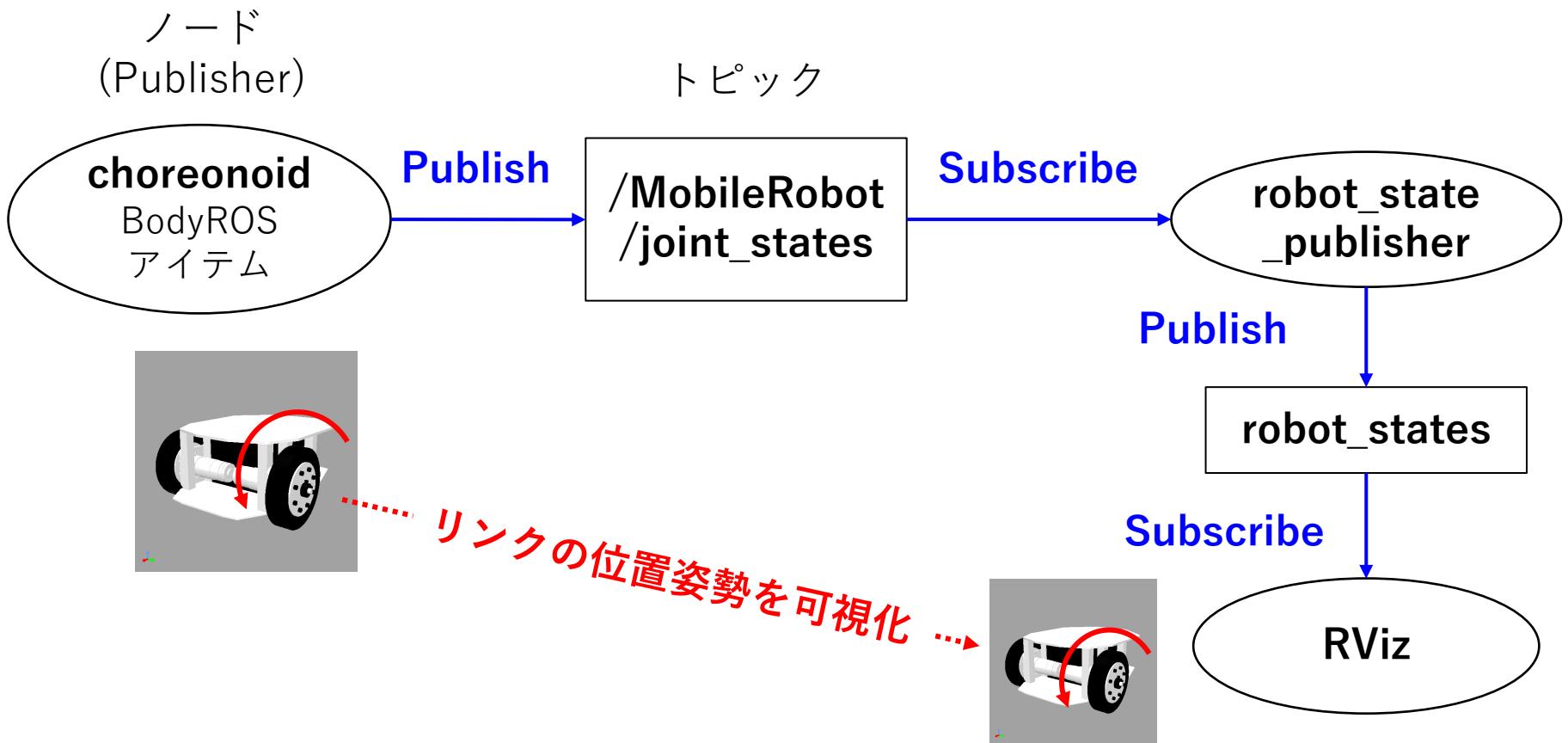
rqt\_graphも表示する

# Rvizの起動と設定

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

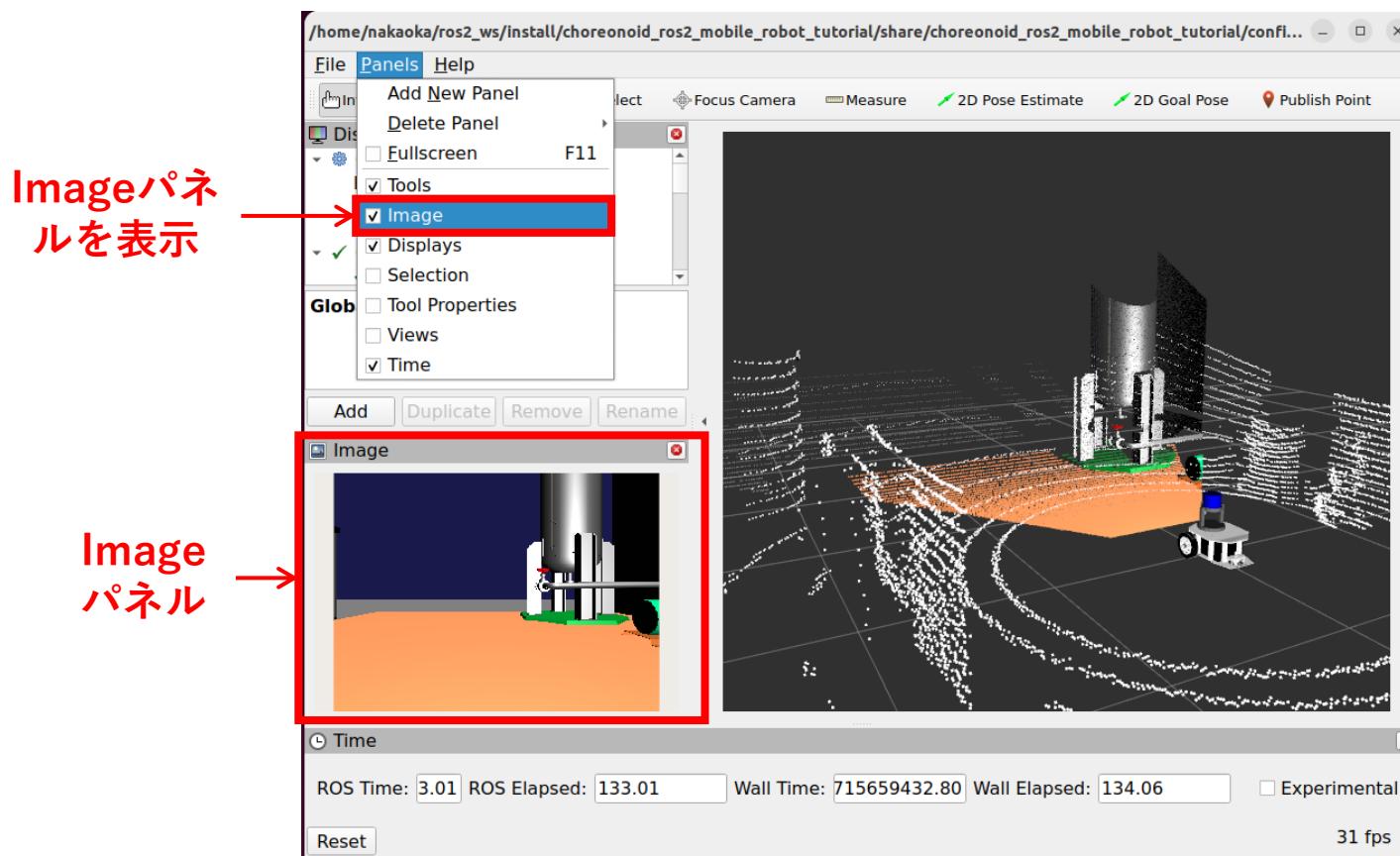
“config/mobile\_robot.rviz” はrvizを起動後にGUIで設定して  
“File” – “Save Config As” で保存しておく

# joint\_statesからrobot\_statesへの 変換



# 画像トピックの可視化

RVizの”image”パネルを使う



# 全起動launchファイル

“launch/sensors\_lab01\_display\_launch.xml”

```
<launch>
  <include file="$(find_pkg-share my_mobile_robot)/launch/sensors_lab01_launch.xml"/>
  <include file="$(find_pkg-share my_mobile_robot)/launch/display_launch.xml"/>
</launch>
```