
# 箱庭を用いた シミュレーション環境のつくりかた


細合 晋太郎
(東京大学)

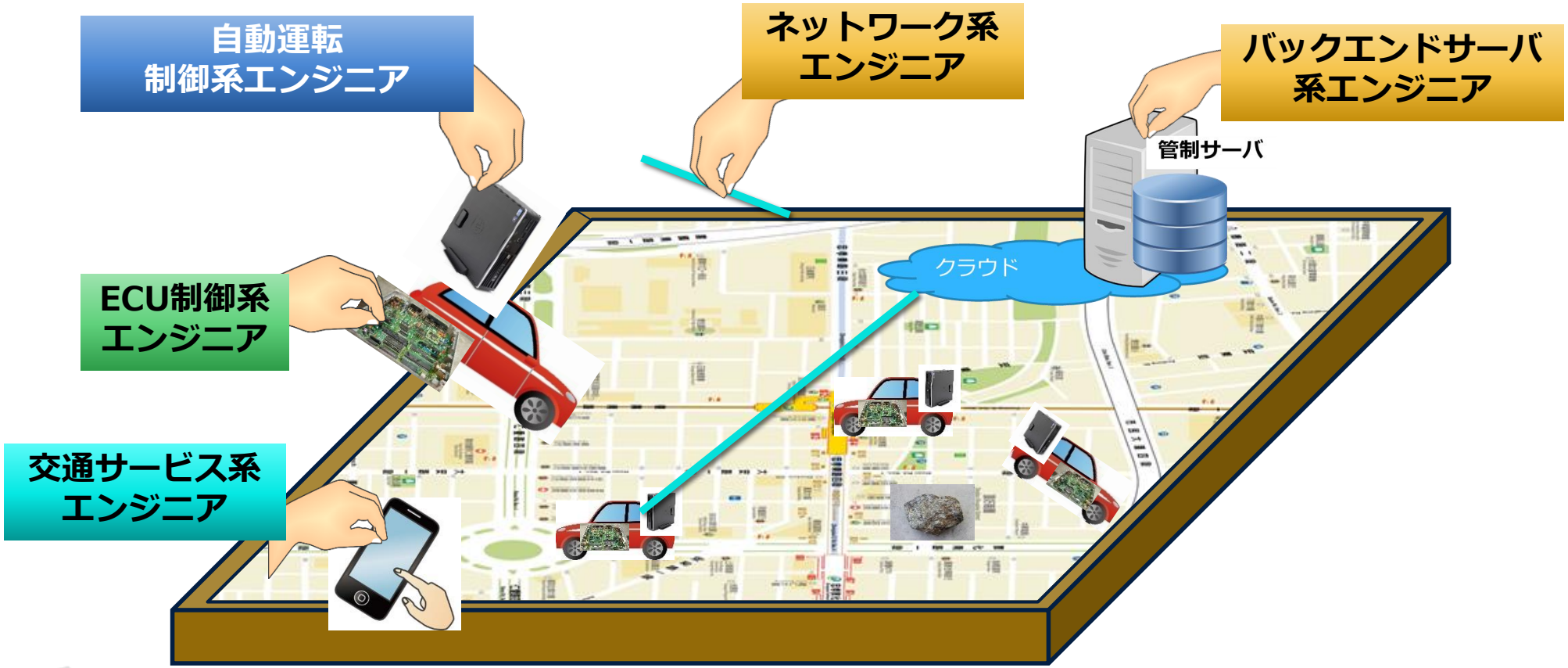
TOPPERS
Toyohashi OPen Platform for Embedded Real-time Systems

# アジェンダ

1. 箱庭の紹介
2. 箱庭のしくみ
3. 制御プログラムの変更方法
4. 仮想環境の変更方法
5. ロボットの変更方法

# 『箱庭』とは？ コンセプトと狙い

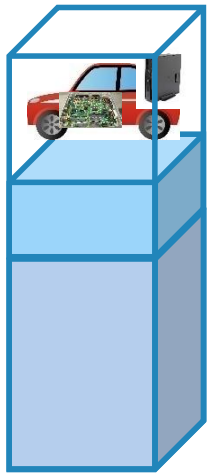
- 箱の中に，様々なモノをみんなの好みで配置して，いろいろ試せる！
  ⇒ 各技術者が開発対象と興味(＝アセット)を持ち寄って，机上で実証実験



# 『箱庭』のしくみ

- 仮想環境（Unity）のモデルをプログラムから動かす


箱庭Asset



仮想モデル（Unity Asset）

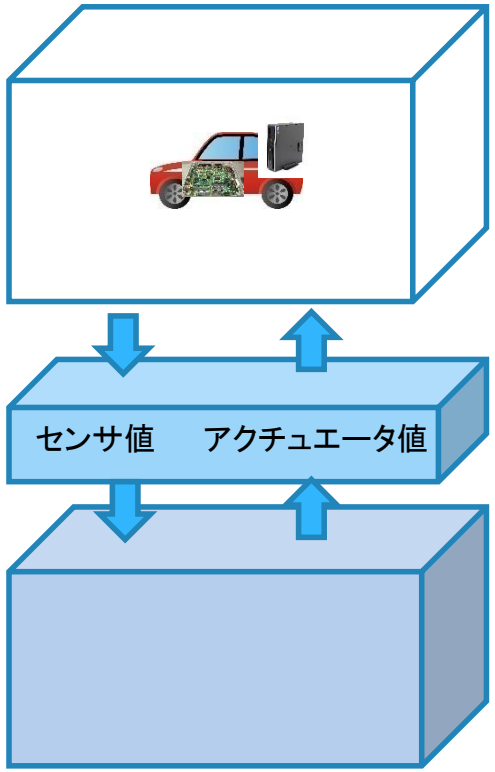
通信部（ROS or mmap)

制御プログラム

一見簡単そうに見えるけど、各プログラム間とUnityの同期や、極力遅延を抑えた通信、差し替えやすいAssetの仕組みなど、複雑な仕掛けがたくさん含まれています。

TOPPERS
Toyohashi OPen Platform for Embedded Real-time Systems

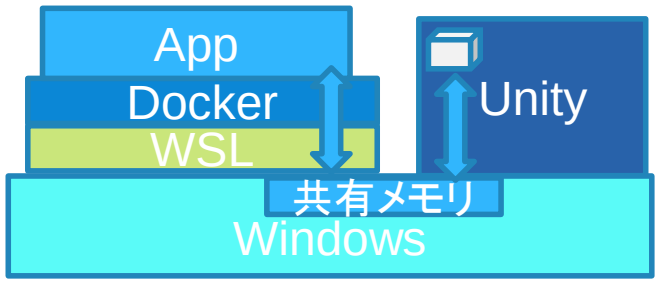
# 制御プログラムとUnityの通信（Athrill）




箱庭Asset

共有メモリ or UDP
（プログラムからは実機と同様に
Memory Mapped I/Oで扱える）

制御プログラム




- 制御プログラムとUnity上のモデルは、共有メモリ(またはUDP)を介して通信しています。
- このメモリ配置は、EV3とIOマップと同様の構成となっているため、実機と同じソースコードでシミュレーションを行うことができます。

# 今回の前提環境


- 単体ロボット向けシミュレータ
- マイコンシミュレータAthrill上のプログラムでUnity上のロボットを制御するプロトタイプ

- 導入までは、GithubのReadmeを参照ください。
- https://github.com/toppers/hakoniwa-single_robot

- また今回の資料ではUnityも必要となるため、UnityhubおよびUnityEditor 2021.3.5f1をインストールしてください。（2021.3.xxxのxxx部は最新のもので大丈夫です。Asset導入時にバージョンが違う旨のエラーがでますがLTS間は互換があるので大丈夫です。）

# Hakoniwa-single_robot project

- hakoniwa-single_robotのフォルダ構成
  - sdk
    - common (EV3モデル用のデバドラ等)
    - workspace（ユーザプログラム、ToppersASP形式）
  - unity
    - assets
      - single-robot (Unity Project)
  - (その他のフォルダは一旦無視)

- これらのうち、必要なフォルダがDocker コンテナ上にマウントされ、ビルドや実行可能な環境が構築されます。

# 制御プログラムのカスタマイズ(Athrill)

- hakoniwa-single-robot(ev3)のサンプルのソースコードは、
  hakoniwa-single-robot/sdk/workspace/base_practice_1/
  に配置されています。これはToppers/ASPのアプリケーションフォルダです。
- asp本体のコードは、Dockerコンテナ上に予め配置されており、上記のworkspaceフォルダはコンテナ起動時にマウントされます。

- ソースコードの変更方法は、基本的にEV3の変更と同様なため割愛します。
- 変更後は、hakoniwa-single-robotフォルダで、
  bash build-app.bash base_practice_1
  を実行しビルドします。この際、コンテナが起動している必要があるため、予め
  bash run-proxy.bash base_practice_1
  を起動しておく必要があります。

# コース（環境）のカスタマイズ


- コースはUnityのAssetとして作られており、凝った改変にはUnityの知識が必要となりますが、今回は最低限の変更方法だけお伝えします。

- コースにモノを追加する際の注意点は、接触するか否かと、物理演算の影響を受けるか否かだけ気をつければ、そこまでおかしな挙動にはなりません。

- hakoniwa-single-robotプロジェクトには、Unityの実行ファイル化したものしか同梱されていません。
- このため、新しくUnityプロジェクトを作成します。

# Unityプロジェクトの作成

- 箱庭Asset(single-robot-HackEV-hakoniwa-core.unitypackage)をDLしておきます。(参照元 Release:https://github.com/toppers/hakoniwa-Unity-Package/releases/tag/hackev-v1.0.0 )
- <hakoniwa-single-robotの展開フォルダ>/unity/assets/single-robotフォルダをリネームして保存しておきます。
- Unity Hubから「3D」プロジェクトを「single-robot」という名称で新規作成します。この際、プロジェクトのフォルダは、<hakoniwa-single-robotの展開フォルダ>/unity/assets/ 以下に作成するようにします。
- プロジェクトを開き、先ほどDLしたunitypackageをインポートします。
  (メニューから Assets / Import Packages / Custom Package)
- Assetsから、Scenes /「Toppers_Course.unity」を開きます。
- ターミナルに戻り、<hakoniwa-single-robotの展開フォルダ>で
  ./unity/config-proxy.bash を実行します。これで通信周りの設定が行われます。
- ▶を押し、一度起動してみます。画面が真っ黒になる場合は、カメラ設定に不備があるので修正します
  （ColorSensorカメラのTarget DisplayをDisplay2に変更）

- run-proxy.bash base_practice_1　も起動しておき、ロボットが動くか確認します。

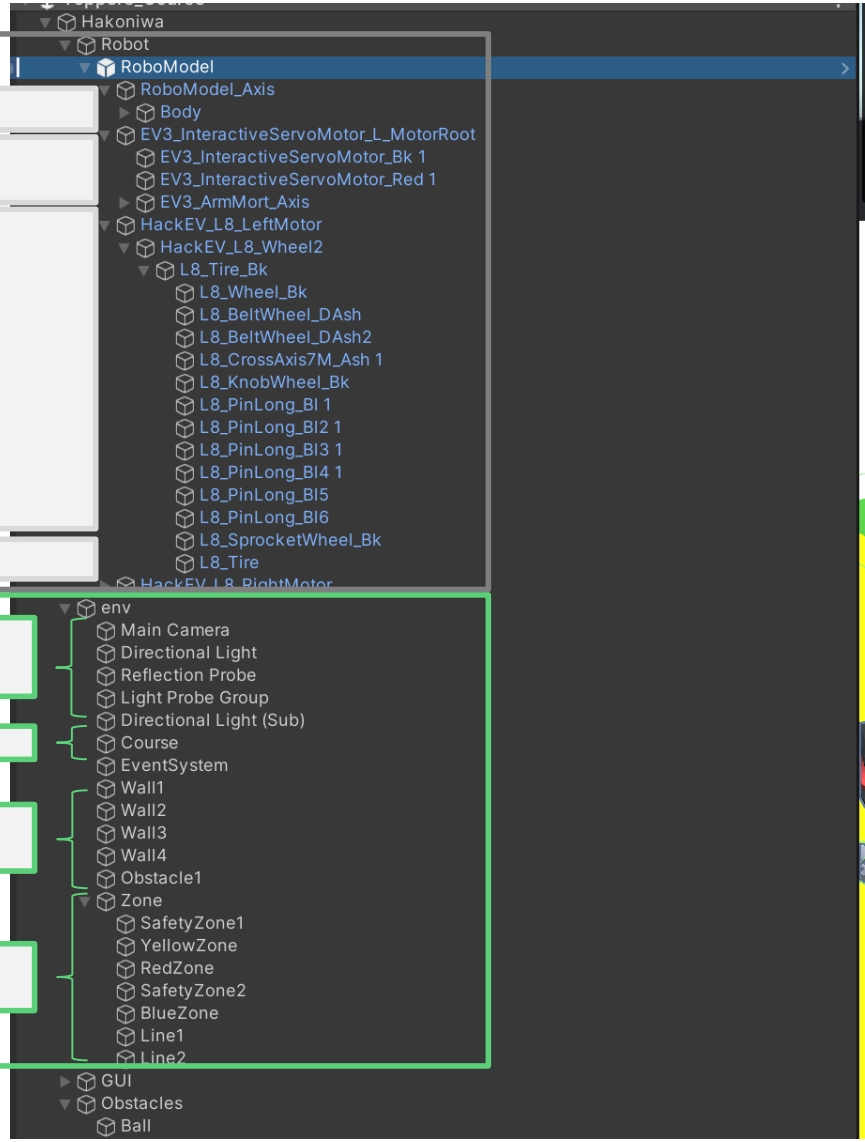
# 例題コースのオブジェクト構成

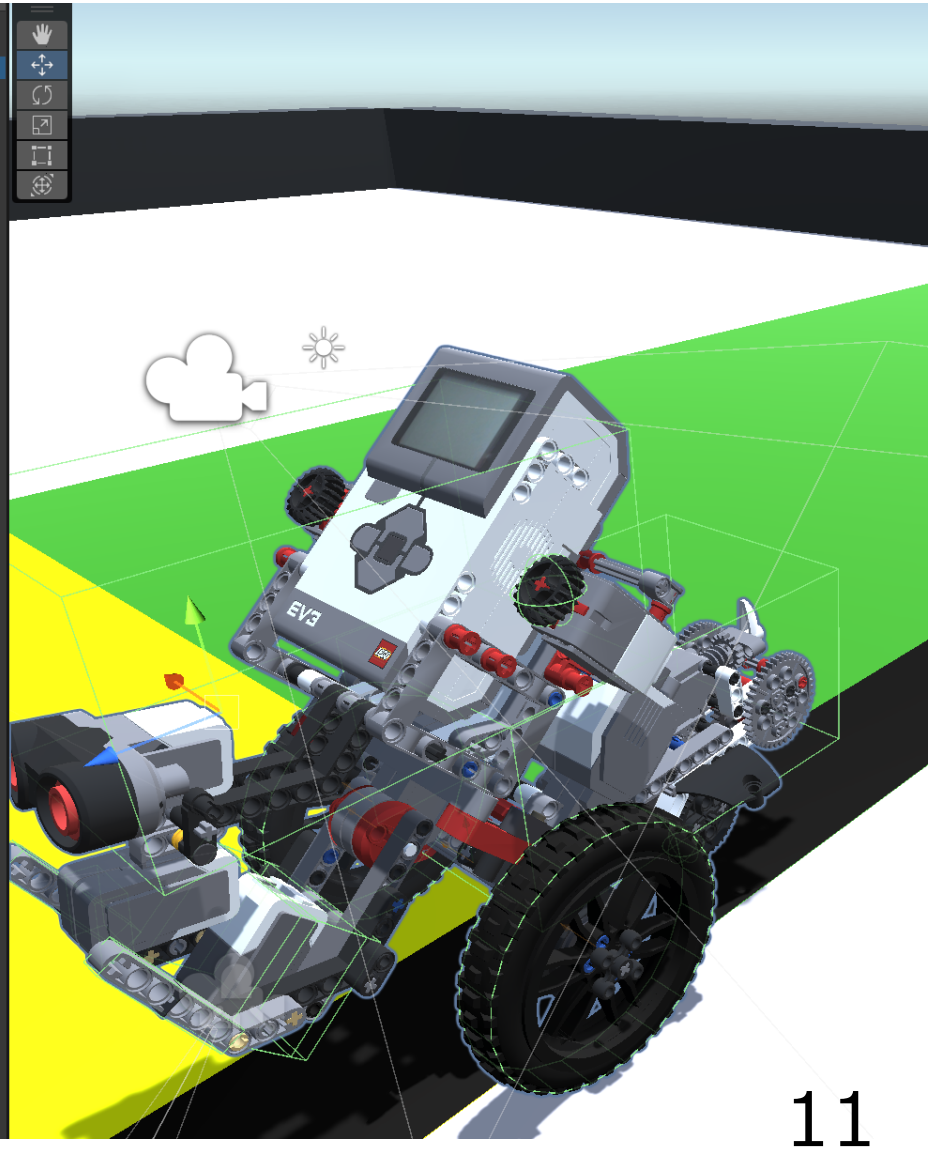
ロボット定義

- 本体パーツ（センサ類含む）
- アーム類
- 左タイヤ・モータ
- 右タイヤ・モータ

環境定義

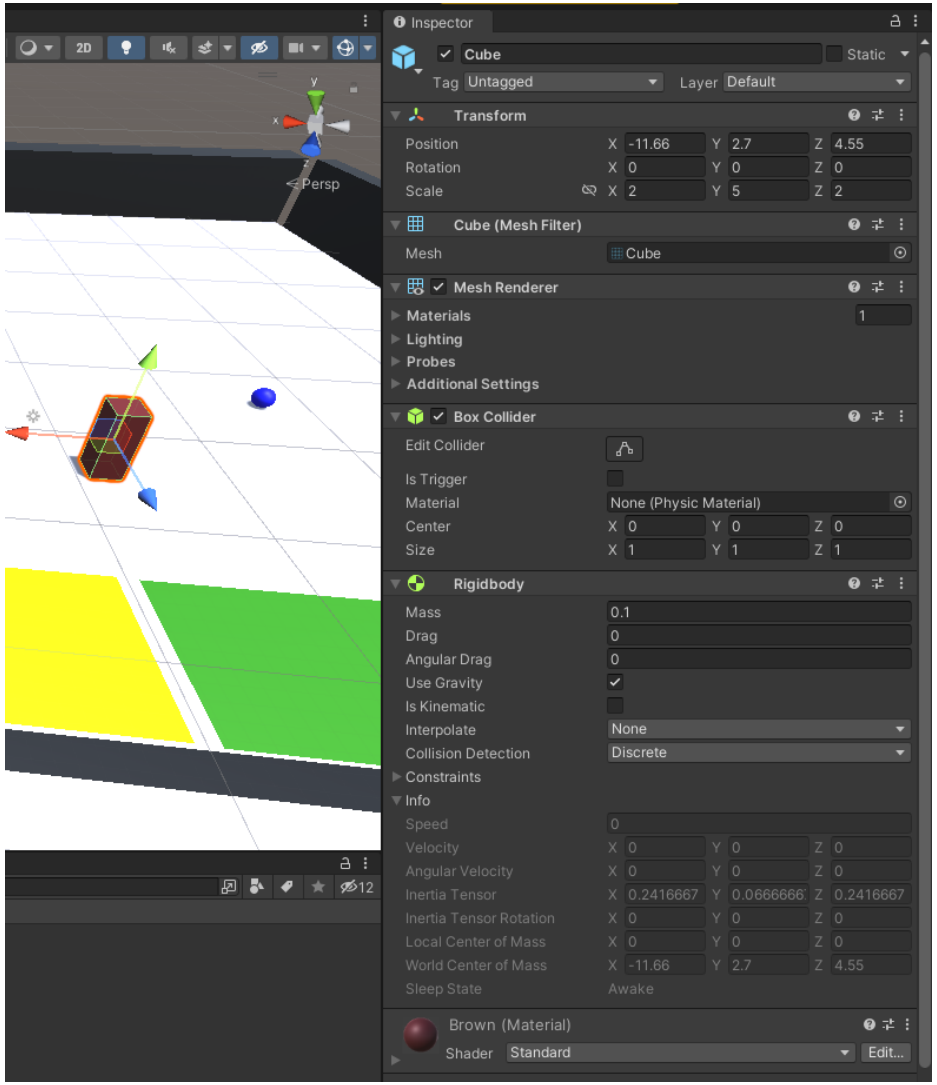
- カメラやライト類、基本的に変更不要
- 床材、必要に応じて拡張
- 四方の壁
- 床のコース模様



# オブジェクトのComponent

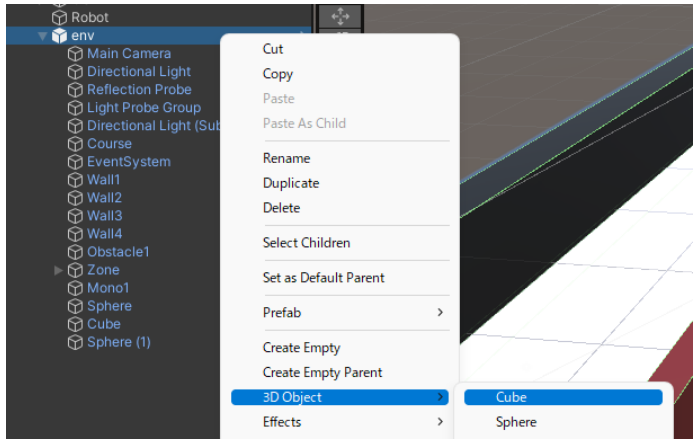
- Unityのオブジェクトは、様々なComponentを適用できます
- Transform
  - 位置・角度・スケールの定義
- Mesh Filter
  - 形状定義（挿入時に自動的に選ばれています）
- ○○ Renderer
  - レンダリングの定義。基本的に変更不要です。
- ○○ Collider
  - 接触判定の定義。このコンポーネントが付与されているオブジェクトにはロボットがぶつかり、無いと素通りします。床の模様などは見た目だけ欲しいので外しています。
- Rigidbody
  - 物理特性の定義。このコンポーネントが付与されていると物理演算の影響を受けます。一方、無いと不動になります。コースの壁などは動いて欲しくないので、外しています。
- Material
  - 色や材質を定義します。



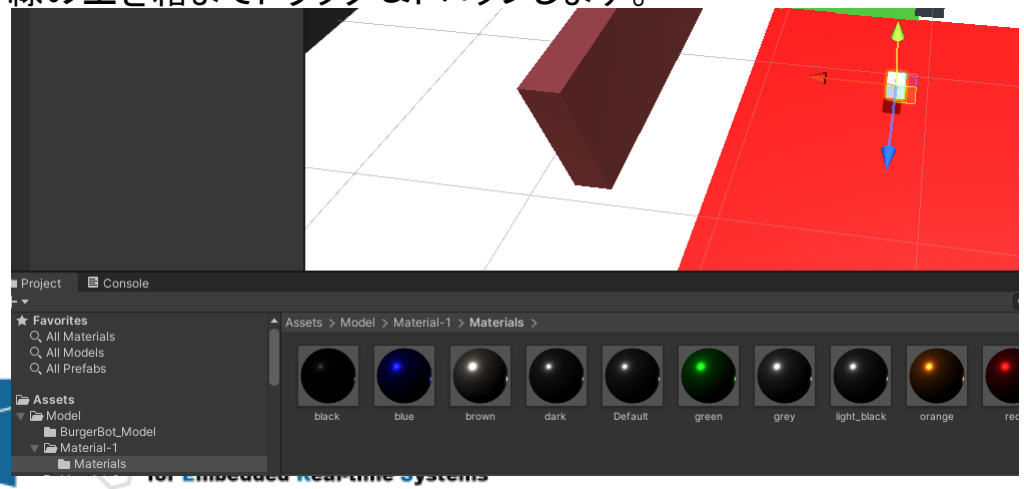
# 障害物の追加例

- コース上に動かせる箱型の障害物を配置します。

① 左のツリーのenvで右クリック、3D Object＞Cubeと選択し、箱を追加します。



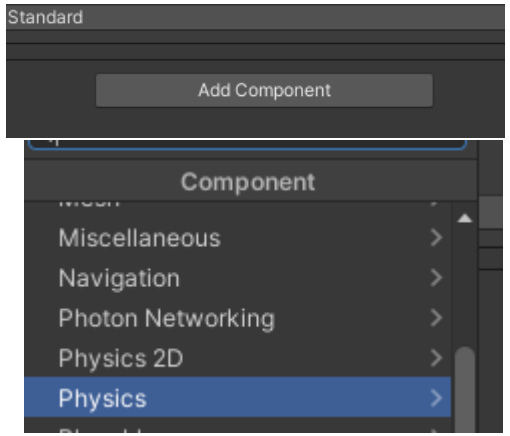

②左下部のAssetsから、Model/Material-1/Materialsと選択し、緑の玉を箱までドラッグ &ドロップします。




③右のリストのTransformで、下記の値を入力し、位置と大きさを変更します。

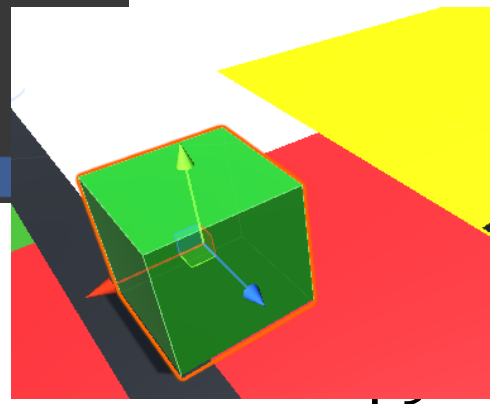

環境によって初期値が異なる場合があります。適宜数値を変更して調整してください。

④ロボットが押せるようにしたいため、物理特性を付与します。
右リストの最下部のAdd Componentを選択し、Physics / Rigidbodyを選択します。





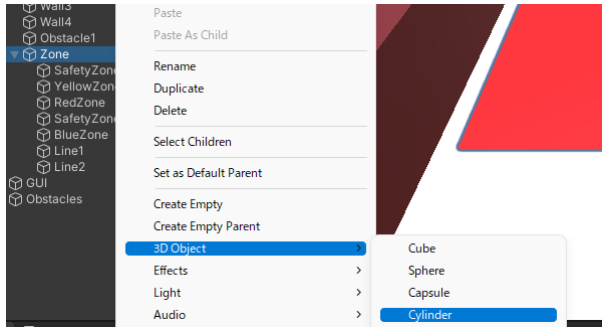
これで障害物が追加されました。



# コースの変更例

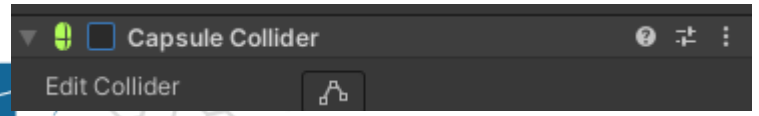
- コースの模様に、紫色の円を追加します。

① 左のツリーのenvで右クリック、3D Object＞Cylinderと選択し、円柱を追加します。
（Unity上では自由形状は作れず、基本形状を組合せるか、テクスチャを貼って表現します）

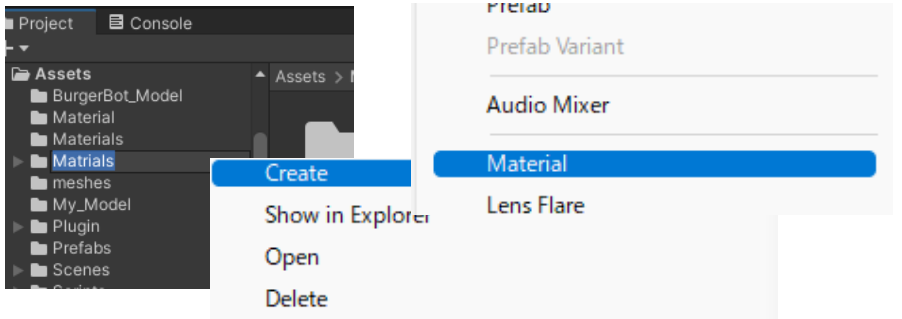


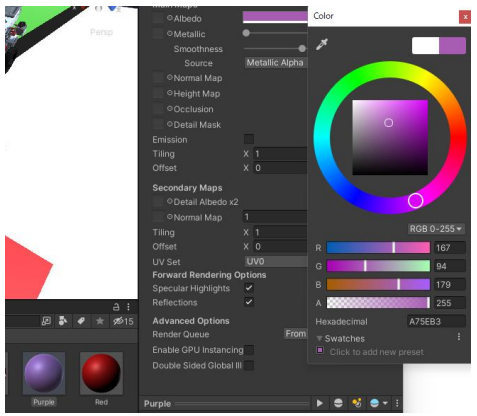

②Scaleで薄い円盤状に変形し、Positionで位置合わせをします。
全く同じ位置にオブジェクトを重ねると、混じって表示されるため、
Y値のように基準面0からわずかに浮かせています。

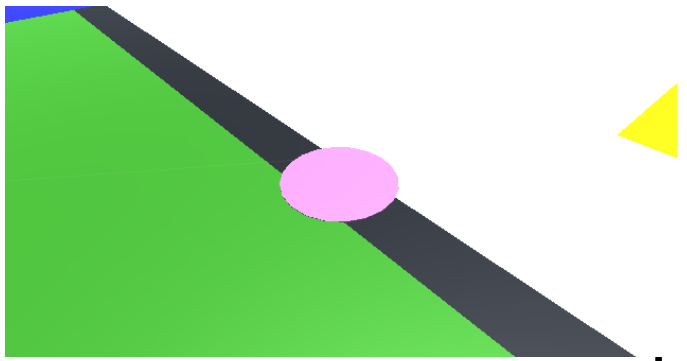

③接触判定は不要なのでColliderのチェックを外します。




④新しい色を追加するには、Materialを新規作成します。
Albedoをクリックしカラーサークルで任意の色にします。

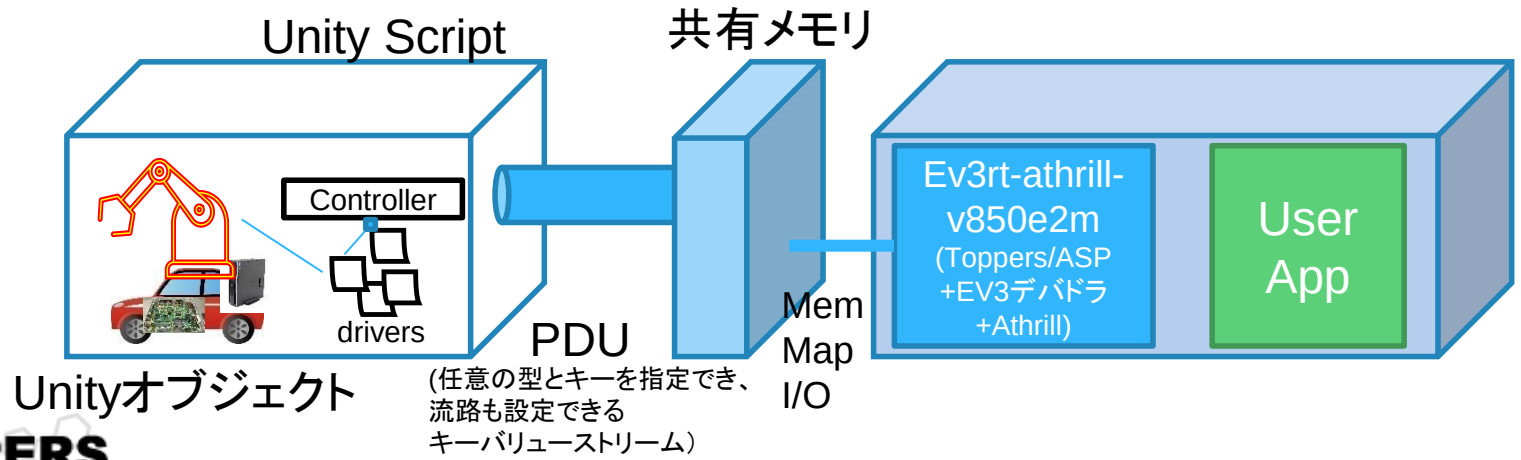


# ロボットのカスタマイズ

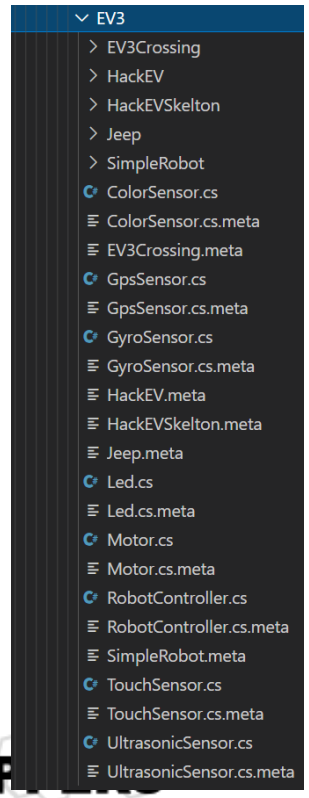
- 新しいパーツを追加するために必要な手順
  - Unityに新たなパーツのオブジェクトを追加
  - そのオブジェクトを動かすためのUnityScript（C#コード）を追加
  - 通信に用いるデータタイプの定義（PDU, grpc/protobuf）
  - メモリ配置の定義
  - ev3rt-athrill-v850e2m内のデバドラ



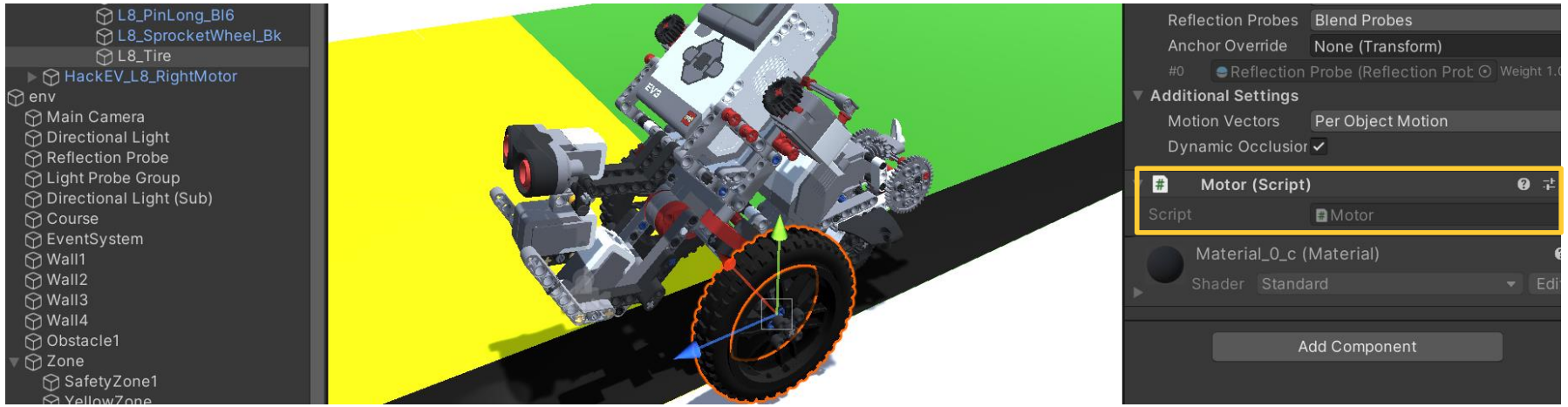
# ロボットのカスタマイズ：ソース読解 Unity(1)

- Assets/Script/PluggableAsset/Assets/Robot/EV3/*
  - RobotController.cs：ロボット制御の主体
  - (Sensor/Actuator) .cs：各オブジェクトのドライバ

ソースツリー




オブジェクトとスクリプトの関連付け



# ロボットのカスタマイズ：ソース読解 Unity(2)

RobotController.cs

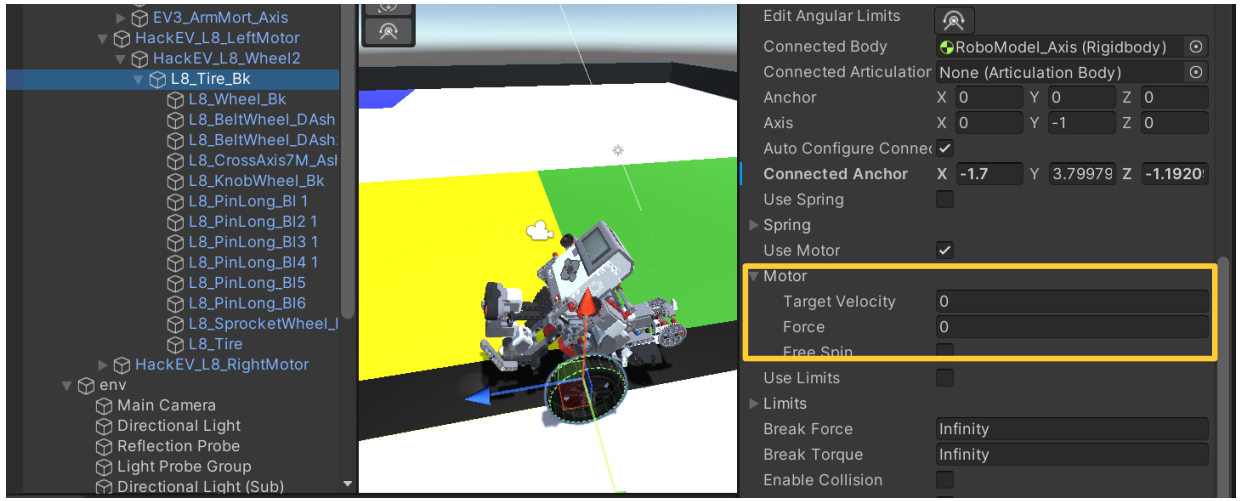
```
public void DoActuation()
{
    int power_a = 0;
    int power_b = 0;
    int power_c = 0;
    int led_color = 0;
    led_color = this.pdu_reader.GetReadOps().GetDataUInt8Array("leds")[0];
    power_a = this.pdu_reader.GetReadOps().Refs("motors")[0].GetDataInt32("power");
    power_b = this.pdu_reader.GetReadOps().Refs("motors")[1].GetDataInt32("power");
    power_c = this.pdu_reader.GetReadOps().Refs("motors")[2].GetDataInt32("power");


    if (this.led != null)
    {
        this.led.SetLedColor((LedColor)(((led_color) & 0x3)));
    }
    if (this.motor_a != null)
    {
        this.motor_a.SetTargetVelicty(power_a * powerConst);
    }
    if (this.motor_b != null)
```

PDUから各デバイスの値を取得し、
各ドライバのSet＊系で設定する。

Motor.cs 一部抜粋

```
    this.rigid_body.drag = 0F;
    this.rigid_body.angularDrag = 0.05F;
    this.motor.force = this.force;
    this.motor.targetVelocity = this.targetVelocity;
    this.joint.motor = this.motor;
    this.isStop = false;
    Debug.Log("released stop");
```




各ドライバでは、
設定された値を
Unityのオブジェクト
のプロパティに反映する。
（センサ類は、この逆の流れでPDUに値を流す）

# ロボットのカスタマイズ：ソース読解 motor

- base_practice_1/app.c
  - ev3_motor_set_power(*motor*, *power*) :
- ../ev3rt-athrill-v850e2m (以下[ev3rt])
/sdk/common/ev3api/src/ev3api_motor.c
  - motor_command(*buf, size*) :
- [ev3rt]/target/v850_gcc/pil/include/driver_interface.h
  - extsvc_motor_command(*buf, size*)
- [ev3rt]/target/v850_gcc/drivers/motor/src/motor_dri.c

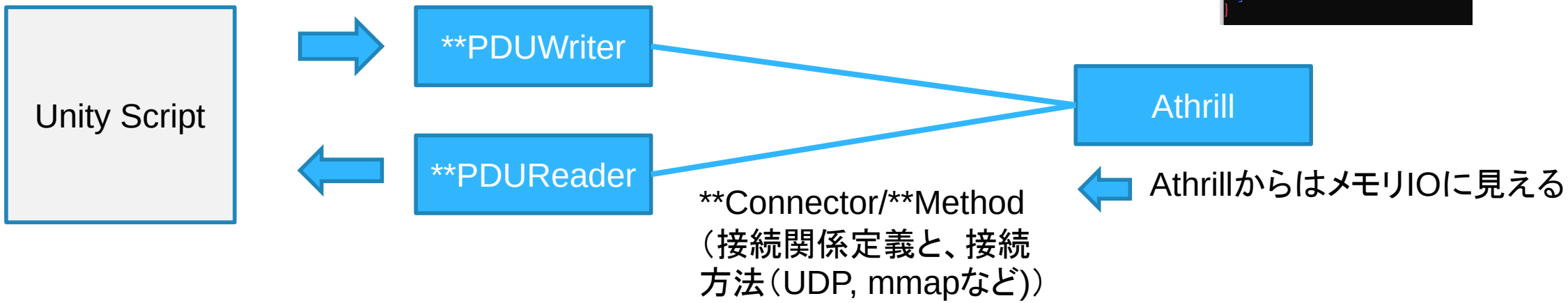
```
ER_UINT extsvc_motor_command(intptr_t cmd, intptr_t size, intptr_t par3, intptr_t par4, intptr_t par5, ID cdmid)
{
   …
}
static void motor_start(int port)
{
        int index = port + EV3_MOTOR_INX_POWER_TOP;
        sil_wrw_mem((uint32_t*)EV3_MOTOR_ADDR_INX(index), motor_power[index]);
        return;
}
```

ここで、ようやくI/Oのメモリにアクセスしている

# ロボットのカスタマイズ：ソース読解 PDU

- /unity/assets/single-robot/core_config.json
  - 各PDUおよびAthrillなどの関係定義と、やり取りするデータタイプの定義、その他設定
- /unity/assets/single-robot/pdu/**.json
  - 各データタイプ定義　（≒　構造体、protobuf）

Ev3PduMotor.json

# ロボットのカスタマイズ：ソース読解 その他


- /sdk/workspace/base_practice_1/memory.txt
  - メモリマップ定義

```
ROM, 0x00000000, 2048
RAM, 0x00200000, 2048
RAM, 0x03FF7000, 1024
RAM, 0x05FF7000, 10240
RAM, 0x07FF7000, 10240
DEV,  0x090F0000,  /root/athrill-device/device/ev3com/build/libev3com.so
```

PDU ⇔ メモリ間の変換
これを介することで、Athrillからはメモリに見える

- /sdk/workspace/base_practice_1/device_config.txt
  - #defineのような設定ファイル。PDUで指定した接続関係なども反映される。

# ロボットのカスタマイズ・・は、ちょっと難しそうです


- ハードウェアデバイスとデバイスドライバの両方の開発に近い実装が必要となるため、なかなかにハードルが高いです。（午前の高田先生の講義にあった、プラットフォームの層に手を入れることになります）
- とはいえ、実CPUと近いレベルでシミュレータが動いているのはやはり面白いです。ぜひ一度コードを追ってみてください。

- なお、ROS2版(hakoniwa-ros2sim)であれば、ROS トピックのレイヤで制御しているため、Athill版よりはだいぶ楽にパーツの追加が行えます。
  https://github.com/toppers/hakoniwa-ros2sim

# 『箱庭』WGへのご案内

- でっかく語って，少しずつ育てております！
  - **だんだんとカタチになってきました！**
  - https://toppers.github.io/hakoniwa/
- 箱庭の狙い・趣旨にご賛同いただける方のWGへの参画をお待ちしております！！
  - まずはSlackでの議論，活動内容へのご要望，コア技術やアセットの開発，などに参加したい方
  - 箱庭WGの技術成果を活用したい方
  - 製品開発に展開してみたい方


**Star && Watch お願いします！**

**公式Webサイトにて最新の技術情報や発表資料を公開中！**


**よろしくお願いいたします！！**

TOPPERS
Toyohashi OPen Platform for Embedded Real-time Systems