OpenRCF ver2.7 説明書

目次

1　OpenRCF の概要

Open Robot Control Framework (OpenRCF) は，ロボット制御用のソフトウェアを開発するためのフレームワークであり，オープンソースソフトウェアとして公開されている．**対応 OS は Windows** であり，**開発環境には Visual Studio，プログラミング言語には C#** を採用している．OpenRCF では，作成したプログラムが実行ファイル（～.exe）にまとめられ，その実行ファイルは一般的な Windows PC で起動することができる．

OpenRCF は以下の設計思想に基づいて開発されている．

- 開発環境のインストールや初期設定に係る手間を可能な限り簡略化し，ユーザーが迅速にロボットシミュレーションを始められるようにする．
- 実装されている全ての機能（public な関数と変数）に関する説明をマニュアル（本稿）に記載する．また，各機能の使用例となるサンプルコードも豊富に記載する．
- OpenRCF で作成したプログラムは，世の中で一般的に用いられている他のノートパソコン上でも，すぐに動作できるようにする．
- ロボットシミュレーションに関する基本機能（ベクトル・行列の計算など）から順に説明し，最終的にはユーザーが独自に考案したアルゴリズムもユーザー自身で実装できるようにする．

なお，OpenRCF を利用するためには，プログラミングに関する基礎知識（配列，if 文，for 文，while 文，構造体，class など）とロボット工学の基礎知識（線形代数，運動学など）が必要となる．

2　Visual Studio のインストールと OpenRCF の起動

下記のサイトから Visual Studio （2022 以降）をダウンロードする．

`https://visualstudio.microsoft.com/ja/downloads/`

Community（無償），Professional（有償），Enterprise（有償）のどれを選択しても OpenRCF を利用することができる．Visual Studio には複数のオプションがあるが，**「.Net デスクトップ開発」のみの選択で問題ない**（図 1）．なお，既に Visual Studio がインストールされている PC でも，「.Net デスクトップ開発」が含まれていないと OpenRCF を利用できない点に注意されたい．

図 1: Visual Studio をインストールする際に選択する項目（背景色は図と異なる場合がある）

Visual Studio のインストールが完了したら，OpenRCF の Zip ファイルを解凍する．Program というフォルダの中に **RobotController.sln（図 2）があるので，それを Visual Studio で開く．**もし「.Net Framework」のバージョンに関する警告が表示されたら，警告文で提示されたサイトから所定の「.Net Framework」をダウンロードする．

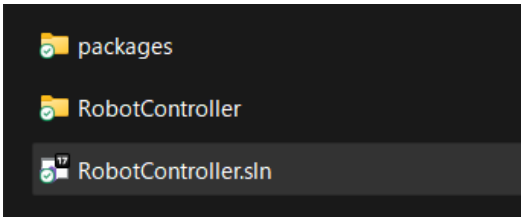


図 2: Visual Studio で開くファイル（RobotController.sln）

RobotController.sln を開くことができたら，開始ボタン（図 3）を押す．

図 3: Visual Studio の開始ボタン

コンパイルが正常に終了すると，図 4 のウィンドウが表示される．ウィンドウの下部にあるボタンやスライダーを操作することで，視点の回転とズームをすることができる．また，マウスの左クリックと右クリックでも，それぞれ視点の回転と水平移動ができる．マウスホイールの回転と押込みでは，それぞれ視点のズームと上下移動ができる．

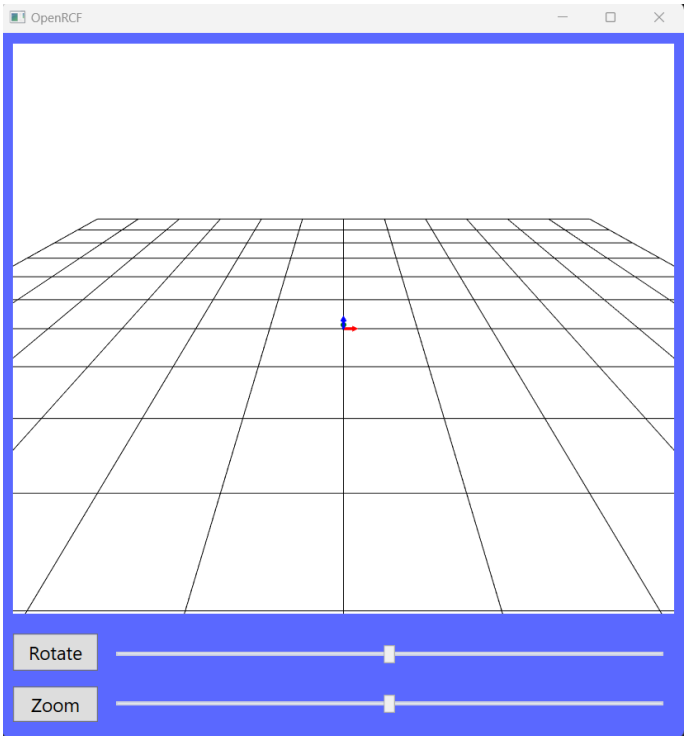


図 4: OpenRCF を起動すると表示されるウィンドウ

中央に表示されている 3 色の矢印は基準座標系を表しており，赤が x 軸，緑が y 軸，青が z 軸の正方向を指している．また，OpenRCF では特に指定のない限り，**長さの単位にはメートル，角度の単位にはラジアンを用いるものとし，軸に対する回転の正負は右ねじの法則に基づいて定められているものとする．**床に表示されている模様は，1 辺が 0.5 [m] の正方形で構成されており，長さを指

定する際の目安として利用することができる．

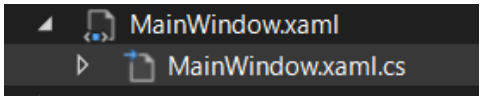


図 5: ソリューションエクスプローラー内の MainWindow.xaml.cs

ユーザーが主に編集するファイルは，ソリューションエクスプローラー（右側にある縦長のタブ）内にある「**MainWindow.xaml.cs**」である（図 5）．もしソリューションエクスプローラーが見当たらない，または誤って閉じてしまった場合は，左上にある「表示」の中から「ソリューションエクスプローラー」をクリックすると表示される．「MainWindow.xaml.cs」ファイルに記述されているコードを以下に示す．

MainWindow.xaml.cs ファイルに記述されているコード

```
void Setup()
{

}

void Draw()
{

}

void Button1_Click(object sender, RoutedEventArgs e)
{

}

(...)

void Button5_Click(object sender, RoutedEventArgs e)
{

}
```

関数 Setup() はソフトウェアが起動した直後に 1 度だけ実行される．関数 Draw() は物体を描画するための関数であり，一定の周期で繰り返し実行され続ける．周期は関数 Core.SetFPS(uint fps) を実行することでで変更できる．また，ソフトウェアに表示されているボタン X をクリックすると，関数 ButtonX_Click() が 1 度だけ実行される．原則として**これらの関数は消してはならない．**

Visual Studio の設定によっては，ウィンドウの上部に図 6 のような黒いボックスが表示されることがある．これを消すためには，「デバッグ」→「オプション」→「XAML ホットリロード」→「アプリ内ツールバーを有効にする」のチェックを外す．

図 6: ウィンドウの上部に表示されることがある黒いボックス

3 ベクトルの計算

まず，2 つのベクトル

$$\boldsymbol{a} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \qquad \boldsymbol{b} = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} \tag{1}$$

に対して，ボタンを押すと

$$\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b}, \qquad d = \boldsymbol{a}^{\mathrm{T}} \boldsymbol{b}, \qquad \boldsymbol{p} = \boldsymbol{a} + \boldsymbol{b}, \qquad n = \|\boldsymbol{a}\| \tag{2}$$

という計算をするサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Setup(), Button_Click() の周辺を変更

```
Vector a = new Vector(3);
Vector b = new Vector(3);

void Setup()
{
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;

    b[0] = 4;
    b[1] = 5;
    b[2] = 6;
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Vector c = a * b;
    c.ConsoleWrite();

    float d = a.Trans * b;
    Console.WriteLine(d);

    Vector p = a + b;
    p.ConsoleWrite();

    float n = a.Norm;
    Console.WriteLine(n);
}
```

上記のサンプルコードを入力したら開始ボタンを押す．ボタン 1 をクリックすると，コンソール（黒い画面）に計算結果が出力される．また，Vector 型の変数は演算の対象が float 配列であった場合，自動的にベクトルと見なして演算を実行できる機能がある．例えば，上記のサンプルコード 2 行目を「float[] b = new float[3];」に置き換えたとしても，同じ計算結果が得られる．なお，ベクトル $\boldsymbol{a}, \boldsymbol{b}$ の数値設定は，関数 SetValue() を用いて以下のように記述することもできる．

```
a.SetValue(1, 2, 3);
b.SetValue(4, 5, 6);
```

ベクトル $\boldsymbol{a}$, $\boldsymbol{b}$ に対して実行可能な演算を以下に表で示す．

演算の種類	数式	ソースコード
加算	$\boldsymbol{a} + \boldsymbol{b}$	a + b
減算	$\boldsymbol{a} - \boldsymbol{b}$	a − b
スカラー積（s 倍）	$s\,\boldsymbol{a}$	s * a
外積	$\boldsymbol{a} \times \boldsymbol{b}$	a * b
内積	$\boldsymbol{a} \cdot \boldsymbol{b} = \boldsymbol{a}^{\mathrm{T}}\,\boldsymbol{b}$	a.Trans * b
テンソル積	$\boldsymbol{a}\,\boldsymbol{b}^{\mathrm{T}}$	a * b.Trans
アダマール積	$\boldsymbol{a} \circ \boldsymbol{b}$	a ∧ b
連結	$\left[\boldsymbol{a}^{\mathrm{T}}\ \boldsymbol{b}^{\mathrm{T}}\right]^{\mathrm{T}}$	a & b

また，Vector クラスに含まれる変数と関数の一覧を以下に示す．

Vector クラスの変数と関数の一覧	
float[] Get	ベクトルの各成分を float 配列として取得する．
float[] Set	ベクトルの各成分に float 配列を代入（値渡し）する．
int Length	ベクトルの要素数を取得．
float Sum	各成分の総和を取得．
float AbsSum	各成分の絶対値の総和を取得．
float SquareSum	2 乗和を取得．
float Norm	ノルムを取得．
Vector Abs	各成分を絶対値としたベクトルを取得．
Vector Normalize	正規化したベクトルを取得．
SetValue(params float[] values)	引数の値を各成分に上書きする．
float Distance(float[] vector)	2 点間の距離を取得．
SetLimit(float min, float max)	各成分を下限と上限の範囲内に上書きする．
SetZeroVector()	零ベクトルに上書きする．
SetUnitVectorX()	x 成分を 1 とした単位ベクトルに上書きする．
SetUnitVectorY()	y 成分を 1 とした単位ベクトルに上書きする．
SetUnitVectorZ()	z 成分を 1 とした単位ベクトルに上書きする．
SetUnitVector(int n)	第 n 成分を 1 とした単位ベクトルに上書きする．

float FormedAngle(float[] vector)	ベクトルのなす角 θ を取得．単位はラジアン．
SetRandom(int min, int max)	ベクトルの各成分をランダムな整数に上書きする． 引数は設定する値の最小値と最大値を表す． 引数を省略した場合は min= -10, max= 10 となる．
ConsoleWrite(byte digit)	ベクトルの各成分をコンソールに表示する． 引数は小数点以下の桁数を表す（省略時は 2 となる）．
Follow(Vector v)	引数のベクトルに自動追従するようになる．
QuitFollow()	自動追従を解除．
MoveTo(Vector v, float distance)	引数 1 の位置に向かって，引数 2 の距離だけ移動する．

なお，ベクトルの自動追従機能（関数 Follow）については，本書で後述する他のサンプルコードで使い方を説明する．

4 行列の計算

まず，2 つの行列

$$\boldsymbol{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \qquad \boldsymbol{B} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \tag{3}$$

に対して，ボタンを押すと

$$\boldsymbol{C} = \boldsymbol{A}\,\boldsymbol{B}, \qquad \boldsymbol{D} = \boldsymbol{A}^{\mathrm{T}}\boldsymbol{B}, \qquad \boldsymbol{E} = \boldsymbol{A}^{-1}\,\boldsymbol{A}, \qquad \boldsymbol{F} = \boldsymbol{A} + \boldsymbol{B} \tag{4}$$

という計算をするサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Setup(), Button_Click() の周辺を変更

```
Matrix A = new Matrix(2, 2);
Matrix B = new Matrix(2, 2);

void Setup()
{
    A[0, 0] = 1;
    A[0, 1] = 2;
    A[1, 0] = 3;
    A[1, 1] = 4;

    B[0, 0] = 5;
    B[0, 1] = 6;
    B[1, 0] = 7;
    B[1, 1] = 8;
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Matrix C = A * B;
```

```
    C.ConsoleWrite();

    Matrix D = A.Trans * B;
    D.ConsoleWrite();

    Matrix E = A.Inv * A;
    E.ConsoleWrite();

    Matrix F = A + B;
    F.ConsoleWrite();
}
```

上記のサンプルを実行するとコンソール（黒い画面）に計算結果が出力される．n 行 m 列の行列 $\boldsymbol{A}$ を定義する際は「Matrix A = new Matrix(n, m);」と記述する（引数の左が行数，右が列数）．また，Matrix 型の変数は演算の対象が float 型の 2 次元配列であった場合，自動的に行列と見なして演算を実行できる機能がある．例えば上記のサンプルコード 2 行目を「float[,] B = new float[2, 2];」に置き換えたとしても，同じ計算結果が得られる．なお，行列 $\boldsymbol{A}, \boldsymbol{B}$ の数値設定は，関数 SetValue() を用いて以下のように記述することもできる．

```
A.SetValue(1, 2, 3, 4);
B.SetValue(5, 6, 7, 8);
```

行列 $\boldsymbol{A}, \boldsymbol{B}$ に対して実行可能な演算を以下に表で示す．

演算の種類	数式	ソースコード
加算	$\boldsymbol{A}+\boldsymbol{B}$	A + B
減算	$\boldsymbol{A}-\boldsymbol{B}$	A − B
スカラー積（s 倍）	$s\,\boldsymbol{A}$	s * A
積	$\boldsymbol{A}\,\boldsymbol{B}$	A * B
アダマール積	$\boldsymbol{A}\circ\boldsymbol{B}$	A ∧ B

また，Matrix クラスに含まれる変数と関数の一覧を以下に示す．

Matrix クラスの変数と関数の一覧	
float[,] Get	行列の各成分を float 配列として取得する．
float[,] Set	行列の各成分に float 配列を代入（値渡し）する．
int GetLength(int k)	k=0 ならば行数，k=1 ならば列数を取得．
SetValue(params float[] values)	引数の値を行から順に各成分に上書きする．
SetZeroMatrix()	零行列に上書きする．
SetIdentity()	単位行列に上書きする．
float Trace	行列の対角和を取得．
SetDiagonal(float diag)	引数を対角成分に配置した対角行列に上書きする．
SetDiagonal(params float[] vector)	引数を対角成分に配置した対角行列に上書きする．
float[] GetColumn(int column)	引数で指定した列を float 配列として取得．

SetColumn(int column, float[] vector)	引数で指定した列に float 配列を代入する.
Matrix Trans	転置した行列を取得.
Matrix Inv	逆行列を取得.
Matrix PseInv	擬似逆行列を取得.
SetRandom(int min, int max)	行列の各成分をランダムな整数に上書きする. 引数は設定する値の最小値と最大値を表す. 引数を省略した場合は min= −10, max= 10 となる.
ConsoleWrite(byte digit)	行列の各成分をコンソールに表示する. 引数は小数点以下の桁数を表す（省略時は 2 となる）.
Follow(Matrix m)	引数の行列に自動追従するようになる.
QuitFollow()	自動追従を解除.

なお，行列の自動追従機能（関数 Follow）については，本書で後述する他のサンプルコードで使い方を説明する.

次に，3 つの回転行列 $\boldsymbol{R}_x$, $\boldsymbol{R}_y$, $\boldsymbol{R} \in SO(3)$ に対して，ボタンを押すと

$$\boldsymbol{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\pi & -\sin\pi \\ 0 & \sin\pi & \cos\pi \end{bmatrix}, \qquad \boldsymbol{R}_y = \begin{bmatrix} \cos\pi & 0 & \sin\pi \\ 0 & 1 & 0 \\ -\sin\pi & 0 & \cos\pi \end{bmatrix}, \qquad \boldsymbol{R} = \boldsymbol{R}_x\,\boldsymbol{R}_y \tag{5}$$

という計算をするサンプルコードを以下に示す.

MainWindow.xaml.cs 内の関数 Button_Click() の周辺を変更

```
RotationMatrix Rx = new RotationMatrix();
RotationMatrix Ry = new RotationMatrix();

void Button1_Click(object sender, RoutedEventArgs e)
{
    Rx.SetRx(PI);
    Rx.ConsoleWrite();

    Ry.SetRy(PI);
    Ry.ConsoleWrite();

    RotationMatrix R = Rx * Ry;
    R.ConsoleWrite();
}
```

RotationMatrix 型の行列は，初期値（new 演算子でインスタンスが生成された直後の値）が単位行列となっている．また，Roll，Pitch，Yaw 角をそれぞれ θ_{R}，θ_{P}，θ_{Y} とすると，「RotationMatrix R = new RotationMatrix(θ_{R}, θ_{P}, θ_{Y});」と記述して初期値を設定することもできる．なお，RotationMatrix 型の変数に回転行列ではないもの（例えば零行列）を設定するとバグの原因となる点に注意されたい.

以下に RotationMatrix クラスの関数一覧を示す．ただし，Matrix クラスに含まれる関数と機能が同じものは省略する．また，$\boldsymbol{R}_x(\theta)$, $\boldsymbol{R}_y(\theta)$, $\boldsymbol{R}_z(\theta)$ は，それぞれ x, y, z 軸まわりの角度 θ [rad] の回転

を表し，$\boldsymbol{R}(\theta, \boldsymbol{n})$ は軸 $\boldsymbol{n}$ まわりの角度 θ [rad] の回転を表す行列である．

RotationMatrix クラスの関数一覧（Matrix クラスの関数と同じものは省略）	
float[] RollPitchYaw	Roll，Pitch，Yaw 角に変換し，float 配列として取得．
SetRollPitchYaw(float r, float p, float y)	引数の Roll，Pitch，Yaw 角が表す姿勢を回転行列に変換し，各成分に上書きする．
SetRx(float theta)	各成分を $\boldsymbol{R}_x(\theta)$ に上書きする．
SetRy(float theta)	各成分を $\boldsymbol{R}_y(\theta)$ 上書きする．
SetRz(float theta)	各成分を $\boldsymbol{R}_z(\theta)$ に上書きする．
SetRn(float theta, float[] axis)	各成分を $\boldsymbol{R}(\theta, \boldsymbol{n})$ に上書きする． 軸の長さは 3，大きさ（ノルム）は 1 にする必要がある．
SetTimesRx(float theta)	右から $\boldsymbol{R}_x(\theta)$ を掛けた結果に上書きする．
SetTimesRy(float theta)	右から $\boldsymbol{R}_y(\theta)$ を掛けた結果に上書きする．
SetTimesRz(float theta)	右から $\boldsymbol{R}_z(\theta)$ を掛けた結果に上書きする．
SetTimesRn(float theta, float[] axis)	右から $\boldsymbol{R}(\theta, \boldsymbol{n})$ を掛けた結果に上書きする． 軸の長さは 3，大きさ（ノルム）は 1 にする必要がある．
float[] AngleAxisVector	角軸ベクトルに変換し，float 配列として取得．

5　タイムラベルの設定と経過時間の取得

Timer クラスを用いることで，時間軸にタイムラベル（TimeLabel）を設置し，現在時刻との経過時間（ElapsedTime）を取得することができる（図 7）．

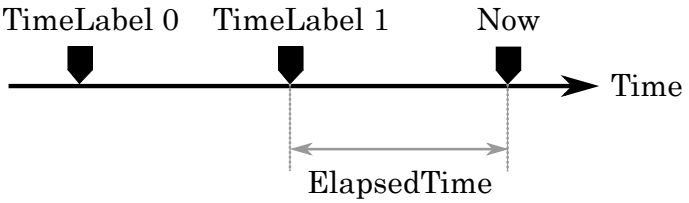


図 7: タイムラベルと経過時間の概念図

タイムラベルを設置し，経過時間を取得するサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Button_Click() を変更

```
void Button1_Click(object sender, RoutedEventArgs e)
{
    Timer.SetTimeLabel(0);
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    float elapsedTime = Timer.GetElapsedTimeSec(0);
```

```
    Console.WriteLine(elapsedTime);
}
```

上記のサンプルでは，ボタン 1 を押すとタイムラベル 0 が設置され，ボタン 2 を押すと，ボタン 1 を押してからの経過時間（単位は秒）がコンソールに出力される．関数 SetTimeLabel(l_n) の引数 l_n は，タイムラベルの番号を表す．同様に，関数 GetElapsedTimeSec(l_n) の引数 l_n も基準とするタイムラベルの番号を指定している．経過時間をミリ秒で取得したい場合は，関数 GetElapsedTimeMs(l_n) を用いる．

6 線，円，長方形，球の作成

線，円，長方形，球を作成するサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Draw() と Button_Click() の周辺を変更

```
Line Line = new Line();
Circle Circle = new Circle(0.5f);
Rectangle Rectangle = new Rectangle(1.0f, 0.8f);
Sphere Sphere = new Sphere(0.2f);

void Draw()
{
    Line.Draw();
    Circle.Draw();
    Rectangle.Draw();
    Sphere.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Line.Position1.SetValue(-1, 1, 0);
    Line.Position2.SetValue(1, 1, 1);
    Line.Width = 3;

    Circle.Position[0] = -1;
    Circle.Position[2] = 0.2f;
    Circle.Rotate.SetRx(0.1f * PI);

    Rectangle.Rotate.SetRx(0.1f * PI);

    Sphere.Position[0] = 1;
    Sphere.Position[2] = 0.2f;
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    Line.Color.SetRed();
    Circle.Color.SetOrange();
```

```
    Rectangle.Color.SetYellowGreen();
    Sphere.Color.SetPurple();
}
```

各初期化子の引数では，以下のように半径や辺の長さを指定する．

- new Circle(半径)
- new Rectangle(幅, 奥行)
- new Sphere(半径)

上記のサンプル実行すると各物体が原点に描画され，ボタン 1 を押すことで各物体の位置・姿勢が変化し，図 8 のようになる．さらに，ボタン 2 を押すことで各物体の色が変化する．

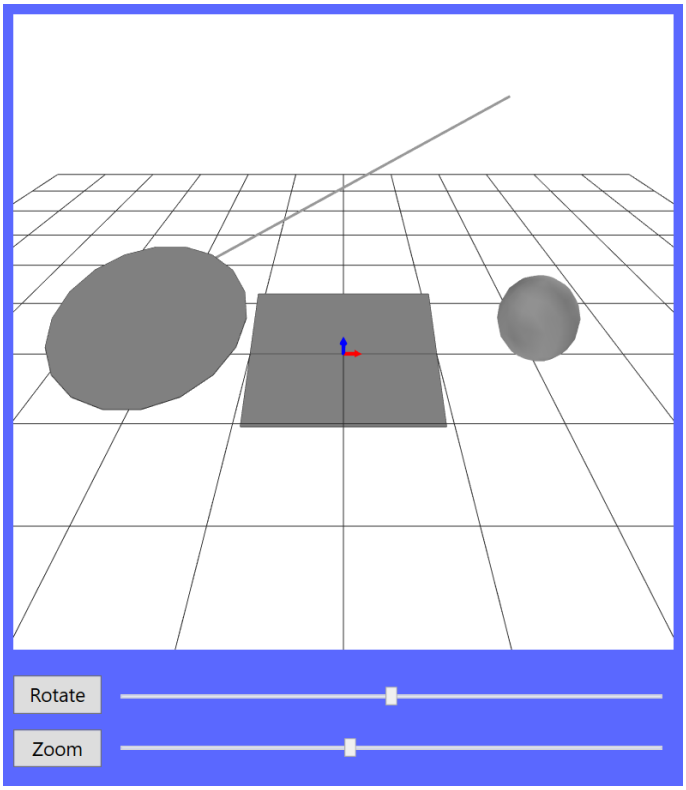


図 8: 線，円，長方形，球を描画するサンプルコードの実行結果

Line，Circle，Rectangle，Sphere クラスに含まれる変数と関数の一覧を以下に示す．

Line クラスの変数と関数の一覧	
Vector Position1	線の始点を表す 3 次元の位置ベクトル．
Vector Position2	線の終点を表す 3 次元の位置ベクトル．
Color Color	線の色．Color.Set〜で色を設定可能．
float Width	線の太さ（初期値は 1）．
float Length	線の長さ．

Draw()	線を描画する関数.
float[] Intersection(params IContact[] ic)	引数の物体と線の交点を返す.

Circle クラスの変数と関数の一覧	
Vector Position	位置を表す 3 次元のベクトル.
RotationMatrix Rotate	姿勢を表す回転行列.
float Radius	円の半径.
Color Color	円の色．Color.Set～で色を設定可能.
float LineWidth	線の太さ（初期値は 1）.
Draw()	円を描画する関数.
DrawLine()	外枠の線のみを描画する関数.
bool IsInside(float[] position)	引数の点が円の面上にあるか否かを返す. 面上にある場合は true，そうでない場合は false を返す.

Rectangle クラスの変数と関数の一覧	
Vector Position	位置を表す 3 次元のベクトル.
RotationMatrix Rotate	姿勢を表す回転行列.
float SizeX	幅.
float SizeY	奥行.
Color Color	長方形の色．Color.Set～で色を設定可能.
float LineWidth	線の太さ（初期値は 1）.
Draw()	長方形を描画する関数.
DrawLine()	外枠の線のみを描画する関数.
DrawLineNet(uint lineNum)	外枠と網目状の線を描画する関数.
bool IsInside(float[] position)	引数の点が長方形の面上にあるか否かを返す. 面上にある場合は true，そうでない場合は false を返す.

Sphere クラスの変数と関数の一覧	
Vector Position	位置を表す 3 次元のベクトル.
float Radius	球の半径.
Color Color	球の色．Color.Set～で色を設定可能.
float LineWidth	線の太さ（初期値は 1）.
Draw()	球を描画する関数.
DrawLine()	外枠の線（メッシュ）のみを描画する関数.
bool IsCollision(float[] position)	引数の点が球と干渉しているか否かを返す. 干渉している場合は true，そうでない場合は false を返す.

bool IsCollision(params ICollision[] ic)	引数の物体が球と干渉しているか否かを返す． 干渉している場合は true，そうでない場合は false を返す．

なお，大量の球を表示する必要がある場合は，Sphere クラスの代わりに RoughSphere クラスを用いると計算量を削減することができる（ポリゴン数の少ない球が生成される）．また，色の透明度は Color.Alpha（初期値は 255）で変更可能であり，これを 0 にすると物体は完全に透明となる．なお，透明な物体が他の物体と重なったときに，背後の物体が描画されないという問題が生じた場合は，Draw 関数の中で透明な物体を最後に描画すると解決する．

7　直方体，円柱，四角錐台，円錐台，矢印の作成

まず，直方体（Cuboid），円柱（Pillar），四角錐台（Square Frustum），円錐台（Cone Frustum）を作成するサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Draw() と Button_Click() の周辺を変更

```
Cuboid Cuboid = new Cuboid(0.5f, 0.5f, 0.4f);
Pillar Pillar = new Pillar(0.2f, 0.5f);
SquareFrustum SquareFrustum = new SquareFrustum(0.4f, 0.2f, 0.2f, 0.4f, 0.3f);
ConeFrustum ConeFrustum = new ConeFrustum(0.4f, 0.2f, 0.6f);

void Draw()
{
    Cuboid.Draw();
    Pillar.Draw();
    SquareFrustum.Draw();
    ConeFrustum.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Cuboid.Position[0] = -1;
    Cuboid.Rotate.SetRz(-0.25f * PI);

    Pillar.Position[0] = 1;
    Pillar.Rotate.SetRx(-0.1f * PI);

    SquareFrustum.Position[0] = -1;
    SquareFrustum.Position[2] = 1;
    SquareFrustum.Rotate.SetRy(0.25f * PI);

    ConeFrustum.Position[0] = 1;
    ConeFrustum.Position[2] = 1;
    ConeFrustum.Rotate.SetRx(0.5f * PI);
}

void Button2_Click(object sender, RoutedEventArgs e)
```

```
{
    Cuboid.Color.SetOrange();
    Pillar.Color.SetPurple();
    SquareFrustum.Color.SetBlue();
    ConeFrustum.Color.SetSkyBlue();
}
```

各初期化子の引数では，以下のように半径や辺の長さを指定する．

- new Cuboid(幅，奥行, 高さ)
- new Pillar(半径, 高さ)
- new SquareFrustum(底面の幅, 底面の奥行, 上面の幅, 上面の奥行, 高さ);
- new ConeFrustum(底面の半径, 上面の半径, 高さ);

上記のサンプル実行すると各物体が原点に描画され，ボタン 1 を押すことで各物体の位置・姿勢が変化し，図 9 のようになる．さらに，ボタン 2 を押すことで各物体の色が変化する．

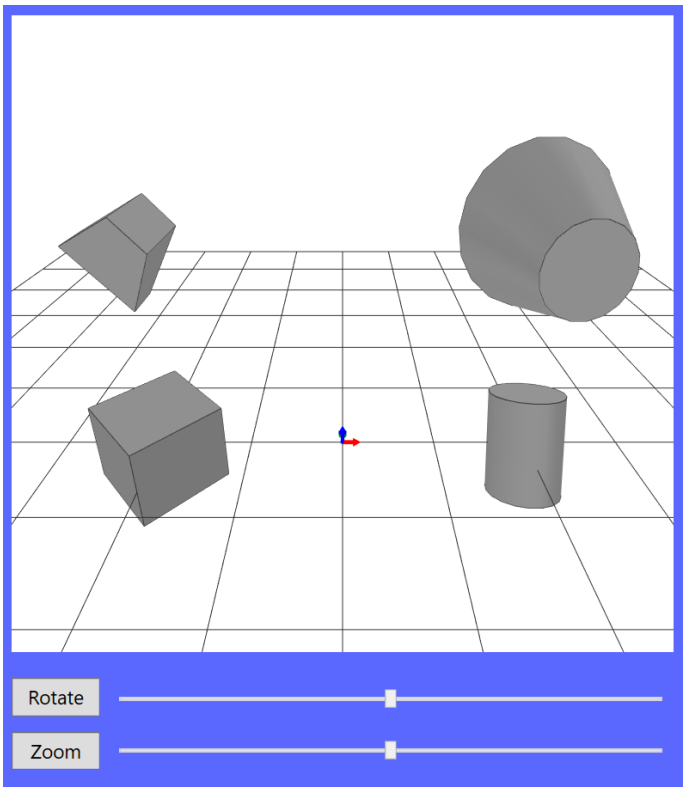


図 9: 直方体，円柱，四角錐台，円錐台を描画するサンプルコードの実行結果

Cuboid クラスと Pillar クラスに含まれる変数と関数の一覧を以下に示す．

Cuboid クラスの変数と関数の一覧	
Vector Position	位置を表す 3 次元のベクトル．

RotationMatrix Rotate	姿勢を表す回転行列.
float SizeX	幅.
float SizeY	奥行.
float SizeZ	高さ.
Color Color	直方体の色．Color.Set～で色を設定可能.
float DiagLength	対角線の長さを取得.
float LineWidth	線の太さ（初期値は 1）.
Draw()	直方体を描画する関数.
DrawLine()	外枠の線のみを描画する関数.
bool IsCollision(float[] position)	引数の点が直方体と干渉しているか否かを返す. 干渉している場合は true，そうでない場合は false を返す.
bool IsCollision(params ICollision[] ic)	引数の物体が直方体と干渉しているか否かを返す. 干渉している場合は true，そうでない場合は false を返す.

Pillar クラスの変数と関数の一覧	
Vector Position	位置を表す 3 次元のベクトル.
RotationMatrix Rotate	姿勢を表す回転行列.
float Length	円柱の高さ.
float Radius	円柱の半径.
Color Color	円柱の色．Color.Set～で色を設定可能.
float DiagLength	対角線の長さを取得.
float LineWidth	線の太さ（初期値は 1）.
Draw()	円柱を描画する関数.
DrawLine()	外枠の線のみを描画する関数.
DrawSide()	側面のみを描画する関数.
DrawTopBottom()	上面と底面のみを描画する関数.
DrawStripes()	縦縞付きの円柱を描画する関数.
bool IsCollision(float[] position)	引数の点が円柱と干渉しているか否かを返す. 干渉している場合は true，そうでない場合は false を返す.
bool IsCollision(params ICollision[] ic)	引数の物体が円柱と干渉しているか否かを返す. 干渉している場合は true，そうでない場合は false を返す.

SquareFrustum と ConeFrustum クラスにも，それぞれ Cuboid と Pillar クラスと同様な変数と関数が存在する．ただし，SquareFrustum クラスには SizeBottomX（底面の幅），SizeBottomY（底面の奥行），SizeTopX（上面の幅），SizeTopY（上面の奥行）があり，ConeFrustum クラスには RadiusBottom（底面の半径）と RadiusTop（上面の半径）がある．

その他にも矢印 (Arrow) を作成するクラスがあり，Cuboid や Pillar クラスと同様な手順で位置や姿勢，長さ，色などを変更できる．Arrow クラスに含まれる変数と関数の一覧を以下に示す．

Arrow クラスの変数と関数の一覧	
Vector Position	根本の位置を表す 3 次元のベクトル.
RotationMatrix Rotate	姿勢を表す回転行列.
float Length	矢印の長さ.
Color Color	矢印の色．Color.Set〜で色を設定可能.
Draw()	矢印を描画する関数.
FixRadius(float radius)	半径を引数で指定した値に固定する.
SetDirectionX()	既定の向きを x 軸方向に変更.
SetDirectionY()	既定の向きを y 軸方向に変更.
SetDirectionZ()	既定の向きを z 軸方向に変更.

色の透明度は Color.Alpha（初期値は 255）で変更可能であり，これを 0 にすると物体は完全に透明となる．また，本節で前述した変数・関数リストでは関数 SetPositionOffset と関数 SetRotateOffset を省略しており，これらの詳細については本書の「物体のグループ化」で後述する．

8　キーボードを用いた関数の実行と物体の操作

まず，新たに作成した関数 Test1()，Test2() をキーボードの「A」と「B」を押したときに実行するサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Setup() の周辺を変更

```
void Test1()
{
    Console.WriteLine("Test1");
}

void Test2()
{
    Console.WriteLine("Test2");
}

void Setup()
{
    Keyboard.KeyEventA = Test1;
    Keyboard.KeyEventB = Test2;
}
```

上記のサンプルではキーボードの「A」と「B」のみを用いたが，全てのアルファベットキー（26 個）と「Delete」,「BackSpace」キーを用いることができる．

次に，Keyboard クラス内にある Vector[] 型の変数 ShiftVector と RotationMatrix[] 型の変数 CtrlMatrix の説明をする．キーボードの数字キー「i」（$i = 0, 1, \cdots, 9$）を押した後に「Shift」＋「矢印」キーを押すことで，ShiftVector[i] の値を書き換えることができる．z 軸方向の移動はスラッシュと

バックスラッシュキーで行うことができる．同様に，数字キー「i」を押した後に「Ctrl」＋「矢印」キーを押すことで，CtrlMatrix[i] の値を書き換えることができる．z 軸まわりの回転はスラッシュとバックスラッシュキーで行うことができる．また，「Ctrl」＋「Delete」キーを押すと CtrlMatrix[i] を初期値（単位行列）に戻すことができる．以下に ShiftVector と CtrlMatrix を用いたサンプルコードを示す．

MainWindow.xaml.cs 内の関数 Setup() と Draw() の周辺を変更

```
PrickleBall PrickleBall = new PrickleBall(0.1f);

void Setup()
{
    PrickleBall.Position.Follow(Keyboard.ShiftVector[0]);
    PrickleBall.Rotate.Follow(Keyboard.CtrlMatrix[0]);
}

void Draw()
{
    PrickleBall.Draw();
}
```

上記のサンプルを実行すると，PrickleBall 型の物体が原点に表示される．物体の位置（Position）と姿勢（Rotate）は，それぞれ ShiftVector[0] と CtrlMatrix[0] の値に自動で追従（Follow）するため，キーボードから操作することができる．

「Shift」＋「矢印」キーを 1 回押すことによって生じる距離の変化量は，Keyboard クラス内の ShiftDistance（float 型）を書き換えることで変更することができる．また，「Ctrl」＋「矢印」キーを 1 回押すことによって生じる角度の変化量は，Keyboard クラス内の CtrlAngle（float 型）を書き換えることで変更することができる．

9　物体のグループ化

行列・ベクトルの自動追従機能（関数 Follow）と物体の位置・姿勢にオフセットを設定する機能を活用すると物体のグループ化ができる．以下に 5 つの直方体をグループ化し，机を作成するサンプルコードを示す．

MainWindow.xaml.cs 内の関数 Setup(), Draw()，Button_Click() の周辺を変更

```
Cuboid[] Cuboids = new Cuboid[5];

void Setup()
{
    Cuboids[0] = new Cuboid(1.1f, 1.1f, 0.1f);
    Cuboids[1] = new Cuboid(0.1f, 0.1f, 0.5f);
    Cuboids[2] = new Cuboid(0.1f, 0.1f, 0.5f);
    Cuboids[3] = new Cuboid(0.1f, 0.1f, 0.5f);
    Cuboids[4] = new Cuboid(0.1f, 0.1f, 0.5f);
```

```
    Cuboids[0].Position.Follow(Keyboard.ShiftVector[0]);
    Cuboids[0].Rotate.Follow(Keyboard.CtrlMatrix[0]);
}

void Draw()
{
    for(int i = 0; i < Cuboids.Length; i++)
    {
        Cuboids[i].Draw();
    }
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Cuboids[1].Position.Follow(Cuboids[0].Position);
    Cuboids[1].Rotate.Follow(Cuboids[0].Rotate);

    Cuboids[2].Position.Follow(Cuboids[0].Position);
    Cuboids[2].Rotate.Follow(Cuboids[0].Rotate);

    Cuboids[3].Position.Follow(Cuboids[0].Position);
    Cuboids[3].Rotate.Follow(Cuboids[0].Rotate);

    Cuboids[4].Position.Follow(Cuboids[0].Position);
    Cuboids[4].Rotate.Follow(Cuboids[0].Rotate);
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    Cuboids[1].SetPositionOffset(0.5f, 0.5f, -0.25f);
    Cuboids[2].SetPositionOffset(0.5f, -0.5f, -0.25f);
    Cuboids[3].SetPositionOffset(-0.5f, 0.5f, -0.25f);
    Cuboids[4].SetPositionOffset(-0.5f, -0.5f, -0.25f);
}
```

上記のサンプルを実行すると，キーボードから Cuboids[0] の位置･姿勢を操作することができるようになる．ボタン 1 を押すと Cuboids[1]～Cuboids[4] の位置･姿勢が Cuboids[0] に自動で追従（Follow）するようになる．これにより，Cuboids[0] の位置･姿勢が変更されると，自動的に Cuboids[1]～Cuboids[4] の位置・姿勢にも反映されるようになる．ボタン 2 を押すと Cuboids[1]～Cuboids[4] の位置にオフセットが設けられる．これにより，Position に代入された位置よりもオフセット分だけ移動した位置に物体が表示される．ここでは位置のみにオフセットを設けたが，姿勢にもオフセットを設定することができる．ボタン 1 とボタン 2 を押すと，Cuboids[0] の位置・姿勢を操作するだけで，机全体をまとめて動かせるようになる（図 10）．

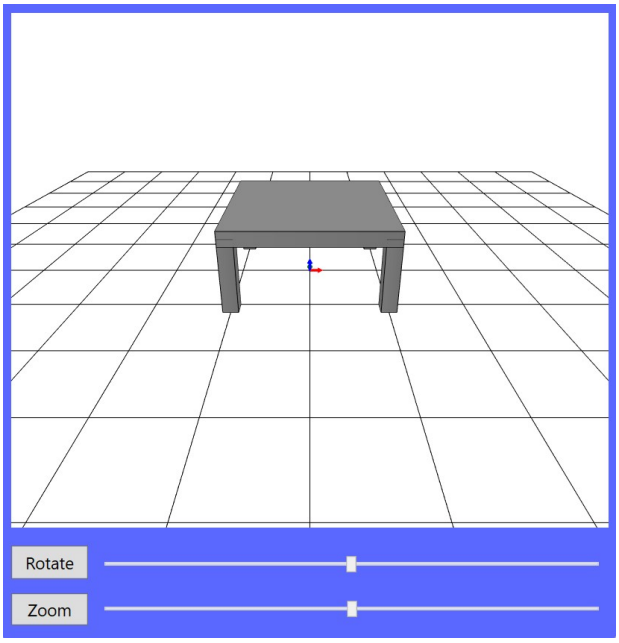


図 10: 机を作成するサンプルコードの実行結果

各物体の位置と姿勢にオフセットを設けるには，それぞれ以下の関数を用いる．

- SetPositionOffset(float x, float y, float z)
- SetRotateOffset(float roll, float pitch, float yaw)

これらの関数は Cuboid，Pillar，SquareFrustum，ConeFrustum クラスの全てに含まれている．Arrow クラスでは位置にのみオフセットを設定できる．

10 物体同士の干渉チェック

物体同士の干渉チェックを行うサンプルコードを示す．

MainWindow.xaml.cs 内の関数 Setup()，Draw()，Button_Click() の周辺を変更

```
Cuboid[] Cuboids = new Cuboid[2];
Pillar Pillar = new Pillar(0.2f, 0.8f);

void Setup()
{
    Cuboids[0] = new Cuboid(0.5f, 3, 0.8f);
    Cuboids[0].Position[0] = 3;
    Cuboids[0].Position[2] = 0.4f;

    Cuboids[1] = new Cuboid(0.5f, 3, 0.8f);
    Cuboids[1].Position[0] = -3;
    Cuboids[1].Position[2] = 0.4f;

    Pillar.Position[2] = 0.4f;
```

```
}

void Draw()
{
    Cuboids[0].Draw();
    Cuboids[1].Draw();
    Pillar.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Pillar.Position[0] += 0.2f;

    if (Pillar.IsCollision(Cuboids[0]))
    {
        Pillar.Position[0] -= 0.2f;
    }
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    Pillar.Position[0] -= 0.2f;

    if (Pillar.IsCollision(Cuboids[1]))
    {
        Pillar.Position[0] += 0.2f;
    }
}
```

上記のサンプルを実行すると図 11 のようになり，ボタン 1 またはボタン 2 を押すと円柱が動く．干渉チェックの機能により，円柱と両脇の壁が衝突することを防いでいる．Cuboids[0] と Cuboids[1] の両方と干渉チェックを行いたい場合は，IsCollision(Cuboids[0], Cuboids[1]) または IsCollision(Cuboids) と記述する．物体を表すクラスには基本的に関数 IsCollision() が含まれており，これを上記のサンプルと同様に用いることで干渉チェックを行うことができる．

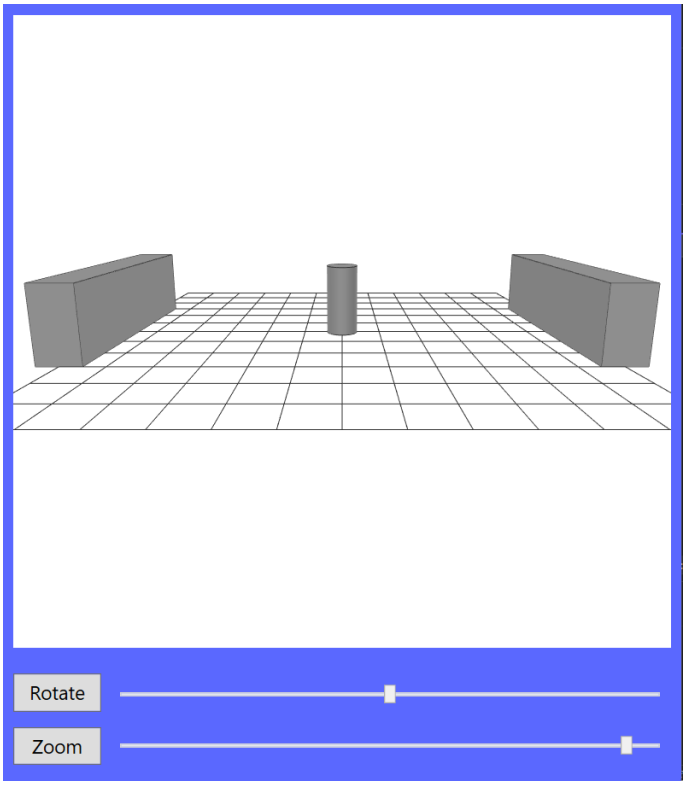


図 11: 干渉チェックを行うサンプルコードの実行結果

11 LiDAR の設置と点群データの取得

LiDAR を設置し，点群データを取得するサンプルコードを以下に示す.

MainWindow.xaml.cs 内の関数 Setup(), Draw()，Button_Click() の周辺を変更

```
LiDAR LiDAR = new LiDAR();
Cuboid[] Cuboids = new Cuboid[2];
Pillar Pillar = new Pillar(0.2f, 0.8f);

void Setup()
{
    Cuboids[0] = new Cuboid(0.5f, 3, 0.8f);
    Cuboids[0].Position[0] = 3;
    Cuboids[0].Position[2] = 0.4f;

    Cuboids[1] = new Cuboid(0.5f, 3, 0.8f);
    Cuboids[1].Position[0] = -3;
    Cuboids[1].Position[2] = 0.4f;

    Pillar.Position[2] = 0.4f;

    LiDAR.Position.Follow(Keyboard.ShiftVector[0]);
    LiDAR.Rotate.Follow(Keyboard.CtrlMatrix[0]);
```

```
    LiDAR.Position[0] = -1;
    LiDAR.Position[1] = -1;
    LiDAR.Position[2] = 0.4f;
}

void Draw()
{
    LiDAR.Draw();
    Cuboids[0].Draw();
    Cuboids[1].Draw();
    Pillar.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    float[][] pointCloud = LiDAR.PointCloud(Cuboids, Pillar);

    int pointNum = pointCloud.GetLength(0);
    float[] point0 = pointCloud[0];
    float[] point1 = pointCloud[1];
    float[] point2 = pointCloud[2];

    Console.WriteLine(point0[0]); // x
    Console.WriteLine(point0[1]); // y
    Console.WriteLine(point0[2]); // z
}
```

上記のサンプルを実行し，ボタン 1 を押すと図 12 のようになり，コンソールに 1 つ目の点 (x, y, z の座標) が表示される．LiDAR を上から見ると，レーザーは反時計回りに回転しており，x 軸の負の方向からスタートしている．このサンプルで取得した点群は LiDAR 座標系を基準としているため，各点の z 成分は常に 0 となる（実際には数値誤差があるため，〜E-08（〜$\times 10^{-8}$）などと表示されることがある）．絶対座標系を基準とした点群を取得する場合は，関数 PointCloud() を PointCloudAbs() に変更する．また，キーボードから LiDAR の位置・姿勢を操作することもできる（反応しない場合は，青いウィンドウをクリックしてから再試行する）．LiDAR の位置・姿勢が変更されても，点群データはボタン 1 を押さないと更新されない点に注意されたい．

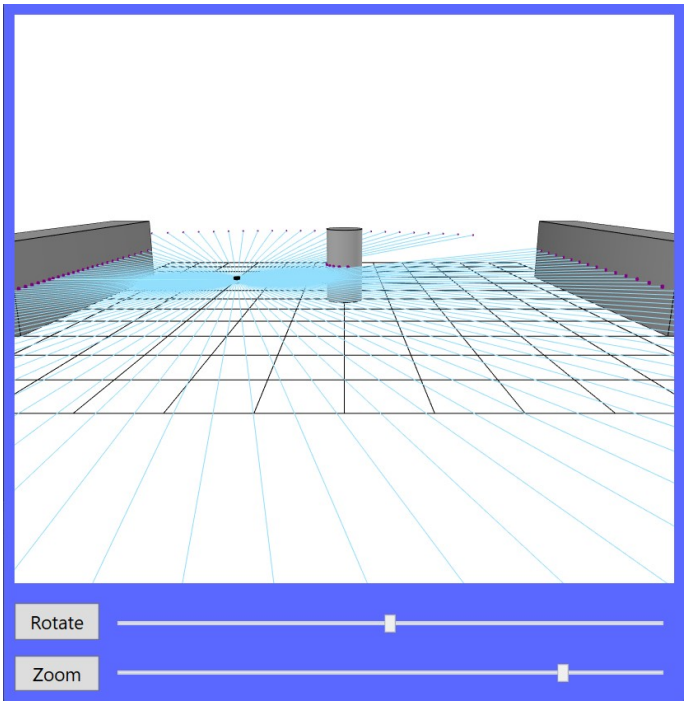


図 12: LiDAR の設置と点群データの取得を行うサンプルコードの実行結果

LiDAR クラスに含まれる変数と関数の一覧を以下に示す.

LiDAR クラスに含まれる変数と関数の一覧	
Vector Position	位置を表す 3 次元のベクトル.
RotationMatrix Rotate	姿勢を表す回転行列.
float StepAngle	ステップ角.
float LaserLength	レーザーの長さ.
float LaserWidth	レーザーの太さ（初期値は 1）.
Color LaserColor	レーザーの色．Color.Set〜で色を設定可能.
float LidarHeight	LiDAR の高さ.
float LidarRadius	LiDAR の半径.
Color LidarColor	LiDAR の色．Color.Set〜で色を設定可能.
float PointRadius	点の半径.
Color PointColor	点の色．Color.Set〜で色を設定可能.
Draw()	LiDAR を描画する関数.
float[][] PointCloud(params IContact[] ic)	LiDAR 座標系を基準に点群を取得.
float[][] PointCloudAbs(params IContact[] ic)	絶対座標系を基準に点群を取得.
SetPositionOffset(float x, float y, float z)	位置にオフセットを設定.

12　関数の並列分散処理

Parallel クラスを用いることで，ユーザーが定義した関数の並列分散処理を行うことができる．まず，関数 Test1()，Test2() を指定した時間間隔（ミリ秒）で同時に実行するサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Button_Click() の周辺を変更

```
void Test1()
{
    Console.WriteLine("Test1");
}

void Test2()
{
    Console.WriteLine("Test2");
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Parallel.Run(Test1, 5000, 500);
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    Parallel.Run(Test2, 5000, 1000);
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    Parallel.RunEndless(Test2, 1000);
}
```

ボタン 1 を押すと関数 Test1() が 500 [ms] に 1 回の周期で実行され，5000 [ms] が経過すると並列処理が終了する．ボタン 2 を押すと関数 Test2() が 1000 [ms] に 1 回の周期で実行され，5000 [ms] が経過すると並列処理が終了する．ボタン 3 を押すと関数 Test2() が 1000 [ms] に 1 回の周期で永続的に実行され続ける（ソフトウェアを閉じると終了する）．

次に，関数 Test1() を一定の時間が経過した後に 1 回だけ実行し，関数 Test2() を特定の条件が満たされた後に 1 回だけ実行するサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Button_Click() の周辺を変更

```
void Test1()
{
    Console.WriteLine("Test1");
}

void Test2()
{
```

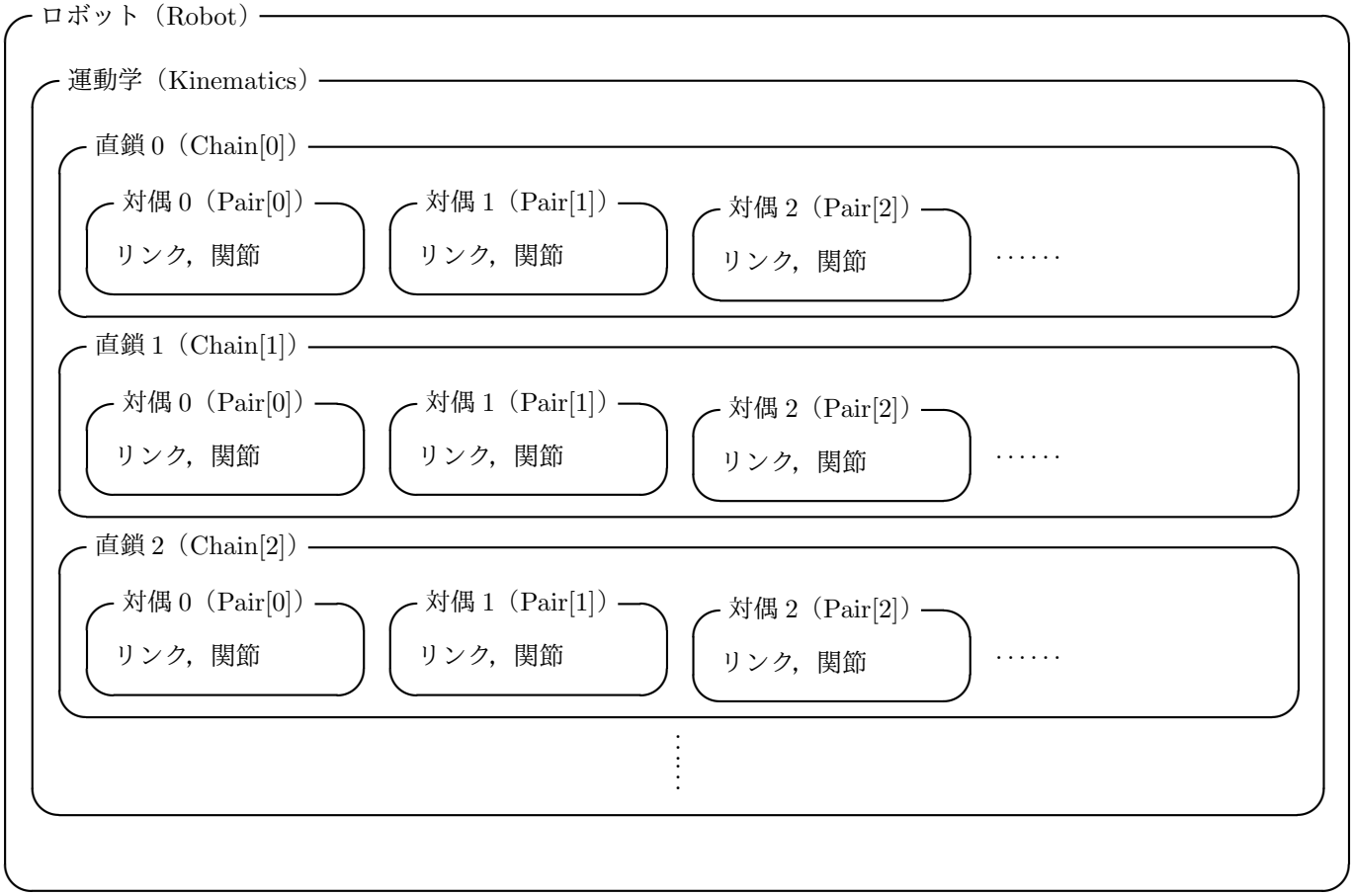
```
    Console.WriteLine("Test2");
}

int count = 0;

bool CanStart()
{
    if (3 < count) return true;
    else return false;
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Parallel.RunWait(Test1, 3000);
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    Parallel.RunWait(Test2, CanStart);
}

void Button3_Click(object sender, RoutedEventArgs e)
{
    count++;
}
```

ボタン 1 を押すと関数 Test1() が 3000 [ms] 後に 1 回だけ実行される．ボタン 2 を押すと関数 CanStart() の返り値が true になった後に関数 Test2() が 1 回だけ実行される．ボタン 3 を押すと count が増加し，4 以上になると関数 CanStart() の返り値が true となる．

Parallel クラスでは，UI（ボタンやスライダーなど）に関連する機能を並列実行することはできない（コンパイルエラーとなる）．一方で ParallelUI クラスでは，UI 関連の機能も含めて並列実行をすることができるが，実行周期の精度に数十ミリ秒の誤差が生じる場合がある．Parallel クラスと ParallelUI クラスに含まれる関数は，どちらも名称と使い方が同じである．

13　多関節ロボットに含まれるデータの階層構造

多関節ロボットを作成する際には Robot クラスを用いる．関節の出力軸とリンクの組を 1 つの対偶（Pair）としており，Robot クラスの内部には複数の Pair クラスがある．その階層構造を以下に示す．



6 自由度のロボットアームを用いて直鎖（Chain）と対偶（Pair）の概念を表すと，図 13 のようになる．

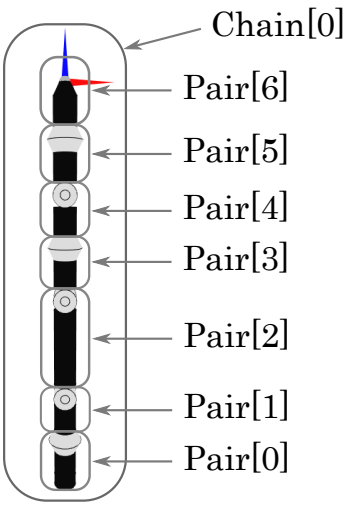


図 13: 直鎖（Chain）と対偶（Pair）の概念図

Pair クラスには，関節とリンクに関する情報が格納されており，以下のような変数と関数が含まれている．

Pair クラスに含まれる変数と関数の一覧	
float q	関節の変位．

float qHome	ホームポジション時の関節変位.
float qTar	関節変位の目標値．qTars[0] と等価．軌道生成時に使用.
float[] qTars	関節変位の目標値を格納する配列．軌道生成時に使用.
float[] JointRange	関節変位の可動域を表す 2 次元の配列. JointRange[0] が下限，JointRange[1] が上限となる.
SetJointRangeInfinite()	関節変位の可動域制限を無制限にする.
Vector p	関節の位置を表す 3 次元のベクトル.
RotationMatrix R	リンクの姿勢を表す回転行列.
Vector Link	リンクの長さを表す 3 次元のベクトル.
Vector LinkInit	基準姿勢時におけるリンクの長さを表す 3 次元のベクトル.
Vector Axis	関節の軸を表す 3 次元のベクトル. 大きさ（ノルム）は 1 にする必要がある.
Vector AxisInit	基準姿勢時における関節の軸を表す 3 次元のベクトル. 大きさ（ノルム）は 1 にする必要がある.
Vector pt	リンクを 3:1 に内分する点を表す 3 次元の位置ベクトル.
Vector pc	リンクを 2:2 に内分する点を表す 3 次元の位置ベクトル.
Vector pb	リンクを 1:3 に内分する点を表す 3 次元の位置ベクトル.
SetRevolute()	回転対偶に設定する.
SetPrismatic()	直進対偶に設定する.
SetParallelA()	平行 4 節リンクを表す回転対偶（根本側）に設定する.
SetParallelB()	平行 4 節リンクを表す回転対偶（手先側）に設定する.
bool IsLocked	関節変位が固定されているか否かを表す.
bool IsLimited	関節変位が可動域の境界にあるか否かを返す.
float Mass	関節の質量 [kg]．自重トルクの計算に用いられる.
float TauSelf	計算された自重トルクが格納される変数．単位は Nm.
bool IsCollision(float[] position)	引数の点が対偶と干渉しているか否かを返す.
Color JointObj.Color	関節の色．Color.Set〜で色を設定可能.
float JointObj.Radius	関節の半径.
Color LinkObj.Color	リンクの色．Color.Set〜で色を設定可能.
float LinkObj.Radius	リンクの半径.

Chain[i] 内の **Pair**[j] にアクセスするためには，「**Robot.Kinematics.Chain**[i]**.Pair**[j].〜」と記述するが，この表記を省略して「**Robot**[i, j].〜」とすることもできる．多関節ロボットを作成する際には，主に基準姿勢時におけるリンクベクトル（LinkInit）と関節軸ベクトル（AxisInit）を設定することになる.

14　ロボットアームの作成と運動学の計算

6 自由度のロボットアームを作成し，運動学の計算をするサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Setup()，Draw()，Button_Click() の周辺を変更

```
Robot Robot = new Robot(6);

void Setup()
{
    Robot[0, 0].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 0].AxisInit.SetUnitVectorZ();
    Robot[0, 1].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 1].AxisInit.SetUnitVectorY();
    Robot[0, 2].LinkInit.SetValue(0, 0, 0.30f);
    Robot[0, 2].AxisInit.SetUnitVectorY();
    Robot[0, 3].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 3].AxisInit.SetUnitVectorZ();
    Robot[0, 4].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 4].AxisInit.SetUnitVectorY();
    Robot[0, 5].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 5].AxisInit.SetUnitVectorZ();
    Robot[0, 6].LinkInit.SetValue(0, 0, 0.15f);

    Robot.Kinematics.ForwardKinematics();
    Robot.Kinematics.SetJointLinkRadiusAuto();
    Robot.Kinematics.SetTargetsToEffector();
}

void Draw()
{
    Robot.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Robot[0, 1].q += 0.2f;
    Robot.Kinematics.ForwardKinematics();
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    Robot.Kinematics.ReturnHomePosition();
    Robot.Kinematics.ForwardKinematics();
}

void Button3_Click(object sender, RoutedEventArgs e)
{
    Robot.Kinematics.Target[0].Position[0] = 0.5f;
    Robot.Kinematics.Target[0].Position[1] = 0;
    Robot.Kinematics.Target[0].Position[2] = 0.6f;
```

```
    Robot.Kinematics.Target[0].Rotate.SetRy(0.5f * PI);

    Robot.Kinematics.InverseKinematics();
}
```

上記のサンプルを実行すると，まずロボットアームが表示される（図 14）．ボタン 1 を押すとロボットの関節変位が更新され，順運動学が計算される．変位が可動域の限界値に達した関節は赤色で描画される．ボタン 2 を押すと各関節変位がホームポジションに戻る．ボタン 3 を押すと目標の位置・姿勢が上書きされ，逆運動学が計算される．**初期化子 new Robot(n_0, n_1, $\cdots$) では，直鎖 i の自由度 n_i を指定している．**

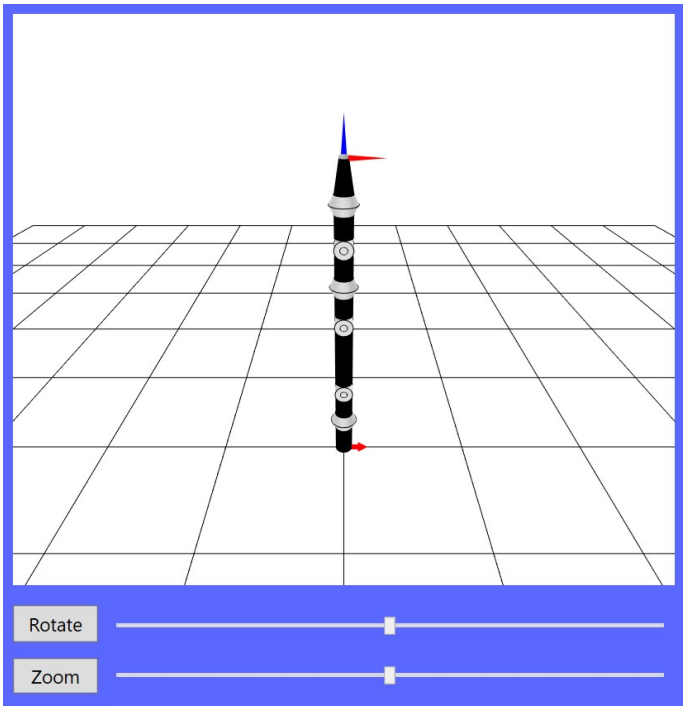


図 14: 6 自由度のロボットアームを作成するサンプルコードの実行結果

Kinematics クラスと Target クラスに含まれる変数と関数の一覧を以下に示す．

Kinematics クラスに含まれる変数と関数の一覧	
Chain[] Chain	直鎖を表す配列．
Vector BasePosition	ベースの位置を表す 3 次元のベクトル． このベクトルを書き換えることでロボットを移動できる．
RotationMatrix BaseRotate	ベースの姿勢を表す回転行列．
ReturnHomePosition()	各関節変位をホームポジションに戻す．
SetJointLinkRadiusAuto()	関節とリンクの太さ（半径）を自動で設定する．
ForwardKinematics()	順運動学を計算する．
InverseKinematics()	逆運動学を計算する．
Target[] Target	逆運動学で指定する目標の位置・姿勢．

SetTargetsToEffector()	目標の位置・姿勢を効果器の位置・姿勢に一致させる.
BlockVector CalcTauSelf()	自重トルクを計算する．Pair クラスの TauSelf も上書きする.
bool IsCollisionSelf()	効果器がロボット自身と干渉しているか否かを返す.
bool IsCollisionEffc(ICollision ic)	引数の物体が効果器と干渉しているか否かを返す.
bool IsCollisionBody(ICollision ic)	引数の物体がロボット（効果器を除く）と干渉しているか否かを返す.
RecordCurrentJointAngles()	現在の各関節変位を記録する.
ReadRecordJointAngles()	記録した各関節変位を読み取る.

Target クラスに含まれる変数と関数の一覧	
Vector Position	位置を表す 3 次元のベクトル.
RotationMatrix Rotate	姿勢を表す回転行列.
bool Priority	目標の優先度（true で高優先，false で低優先）. 優先度付き逆運動学を計算する際に用いる.
float Diameter	球の直径.
Color Color	球の色．Color.Set〜で色を指定可能.
int DOF	拘束自由度を取得.
SetDOF6()	拘束自由度を 6 に設定（位置 3 ＋姿勢 3）.
SetDOF5()	拘束自由度を 5 に設定（位置 3 ＋方向 2）.
SetDOF3()	拘束自由度を 3 に設定（位置 3 ＋姿勢 0）.
SetDOF0()	拘束自由度を 0 に設定.

15　移動マニピュレータの作成と運動学の計算

移動マニピュレータを作成するサンプルコードを以下に示す.

MainWindow.xaml.cs 内の関数 Setup()，Draw()，Button_Click() の周辺を変更

```
Robot Robot = new Robot(3, 6);

void Setup()
{
    Robot.SetPlanarJoint3DOF(0.6f, 0.6f, 0.4f);

    Robot[1, 0].LinkInit.SetValue(0, 0, 0.15f);
    Robot[1, 0].AxisInit.SetUnitVectorZ();
    Robot[1, 1].LinkInit.SetValue(0, 0, 0.15f);
    Robot[1, 1].AxisInit.SetUnitVectorY();
    Robot[1, 2].LinkInit.SetValue(0, 0, 0.30f);
    Robot[1, 2].AxisInit.SetUnitVectorY();
    Robot[1, 3].LinkInit.SetValue(0, 0, 0.15f);
    Robot[1, 3].AxisInit.SetUnitVectorZ();
```

```
    Robot[1, 4].LinkInit.SetValue(0, 0, 0.15f);
    Robot[1, 4].AxisInit.SetUnitVectorY();
    Robot[1, 5].LinkInit.SetValue(0, 0, 0.15f);
    Robot[1, 5].AxisInit.SetUnitVectorZ();
    Robot[1, 6].LinkInit.SetValue(0, 0, 0.15f);

    Robot.Kinematics.Target[1].Priority = false;

    Robot.Kinematics.ForwardKinematics();
    Robot.Kinematics.SetJointLinkRadiusAuto();
    Robot.Kinematics.SetTargetsToEffector();
}

void Draw()
{
    Robot.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Robot[0, 0].q += 0.1f;
    Robot[0, 1].q += 0.1f;
    Robot[0, 2].q += 0.1f * PI;
    Robot.Kinematics.ForwardKinematics();
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    Robot.Kinematics.ReturnHomePosition();
    Robot.Kinematics.ForwardKinematics();
}

void Button3_Click(object sender, RoutedEventArgs e)
{
    Robot.Kinematics.Target[0].Position.SetValue(-0.5f, 0, 0.4f);
    Robot.Kinematics.Target[0].Rotate.SetRz(0.25f * PI);

    Robot.Kinematics.Target[1].Position.SetValue(0.5f, 0, 0.8f);
    Robot.Kinematics.Target[1].Rotate.SetRy(0.5f * PI);

    Robot.Kinematics.InverseKinematics();
}
```

上記のサンプルを実行すると，まず原点に移動マニピュレータが出現する．このロボットでは，**移動台車を表す直鎖機構が Chain[0]，ロボットアームを表す直鎖機構が Chain[1] となっている．移動台車の位置・姿勢は，仮想的な直進対偶 2 つと回転対偶 1 つで表されている．**関数 SetPlanarJoint3DOF(x, y, z) を実行することで自動的に Chain[0] 内の各関節が直進対偶 2 つと回転対偶 1 つに設定され，引数 x, y, z は順に移動台車の幅，奥行き，高さとなる．ボタン 1 を押すことで移動台車の位置・姿勢

が変化し，変位が可動域の限界値に達した関節は赤色で描画される．ボタン 2 を押すと各関節変位がホームポジションに戻る．ボタン 3 を押すと目標の位置・姿勢が上書きされ，逆運動学が計算される．Chain[i] の効果器に与える目標の位置・姿勢が Target[i] となる（図 15）．このサンプルでは Target[1] の Priority（優先度）を false（低優先）としているので，移動台車と Target[0] との誤差が優先的に最小化される．このように各目標に優先度が設定されている逆運動学は，優先度付き逆運動学と呼ばれている．

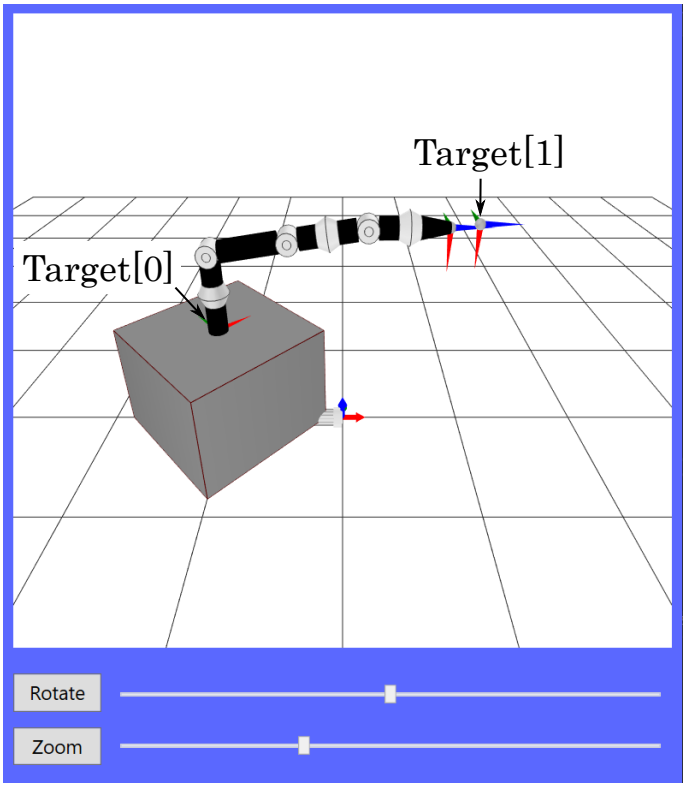


図 15: 移動マニピュレータを作成するサンプルコードの実行結果

移動台車を表す直方体は，Robot クラス内にある Cuboid 型の変数 Torso である．移動台車の骨格を表す Chain[0] 内の変数に，直方体 Torso が貼り付けられている．Torso に関する設定は，関数 SetPlanarJoint3DOF() の内部で行われている．Robot クラスは Robot.cs ファイルの中に記述されており，ユーザーがソースコードを閲覧・修正することができる．また，関数 Robot.Torso.IsCollision() の引数に障害物となる物体を代入することで，移動台車と障害物の干渉チェックを行うことができる．

16　双腕マニピュレータの作成と運動学の計算

双腕マニピュレータを作成するサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Setup()，Draw()，Button_Click() の周辺を変更

```
Robot Robot = new Robot(1, 6, 6);

void Setup()
```

```
{
    Robot[0, 0].LinkInit.SetValue(0, 0, 0.75f);
    Robot[0, 0].AxisInit.SetUnitVectorZ();
    Robot[0, 1].LinkInit.SetValue(0, 0, 0.5f);
    Robot.Kinematics.Target[0].SetDOF0();

    Robot[1, 0].LinkInit.SetValue(0, 0.25f, 0);
    Robot[1, 0].AxisInit.SetUnitVectorZ();
    Robot[1, 1].LinkInit.SetValue(0, 0.15f, 0);
    Robot[1, 1].AxisInit.SetUnitVectorY();
    Robot[1, 2].LinkInit.SetValue(0, 0, -0.3f);
    Robot[1, 2].AxisInit.SetUnitVectorY();
    Robot[1, 3].LinkInit.SetValue(0, 0, -0.3f);
    Robot[1, 3].AxisInit.SetUnitVectorZ();
    Robot[1, 4].LinkInit.SetValue(0, 0, -0.1f);
    Robot[1, 4].AxisInit.SetUnitVectorY();
    Robot[1, 5].LinkInit.SetValue(0, 0, -0.1f);
    Robot[1, 5].AxisInit.SetUnitVectorZ();
    Robot[1, 6].LinkInit.SetValue(0, 0, -0.1f);

    Robot[2, 0].LinkInit.SetValue(0, -0.25f, 0);
    Robot[2, 0].AxisInit.SetUnitVectorZ();
    Robot[2, 1].LinkInit.SetValue(0, -0.15f, 0);
    Robot[2, 1].AxisInit.SetUnitVectorY();
    Robot[2, 2].LinkInit.SetValue(0, 0, -0.3f);
    Robot[2, 2].AxisInit.SetUnitVectorY();
    Robot[2, 3].LinkInit.SetValue(0, 0, -0.3f);
    Robot[2, 3].AxisInit.SetUnitVectorZ();
    Robot[2, 4].LinkInit.SetValue(0, 0, -0.1f);
    Robot[2, 4].AxisInit.SetUnitVectorY();
    Robot[2, 5].LinkInit.SetValue(0, 0, -0.1f);
    Robot[2, 5].AxisInit.SetUnitVectorZ();
    Robot[2, 6].LinkInit.SetValue(0, 0, -0.1f);

    Robot.Kinematics.ForwardKinematics();
    Robot.Kinematics.SetJointLinkRadiusAuto();
    Robot.Kinematics.SetTargetsToEffector();
}

void Draw()
{
    Robot.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Robot[0, 0].q += 0.1f;
    Robot[1, 1].q -= 0.1f;
    Robot[2, 1].q -= 0.1f;
    Robot.Kinematics.ForwardKinematics();
```

```
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    Robot.Kinematics.ReturnHomePosition();
    Robot.Kinematics.ForwardKinematics();
}

void Button3_Click(object sender, RoutedEventArgs e)
{
    Robot.Kinematics.Target[1].Position.SetValue(0.5f, 0.5f, 0.8f);
    Robot.Kinematics.Target[1].Rotate.SetRy(0.5f * PI);

    Robot.Kinematics.Target[2].Position.SetValue(0.5f, -0.5f, 0.8f);
    Robot.Kinematics.Target[2].Rotate.SetRy(0.5f * PI);

    Robot.Kinematics.InverseKinematics();
}
```

上記のサンプルを実行すると，双腕マニピュレータが表示される（図 16）．ボタン 1 を押すと腰と肩の関節が動き，ボタン 2 を押すとホームポジションに戻る．ボタン 3 を押すと目標の位置・姿勢が更新され，逆運動学が計算される．ここでは Target[0] の拘束自由度を 0 としている（Target[0].SetDOF0）ため，逆運動学を計算する際に Target[0] は無視される．このロボットでは，**胴体を表す直鎖機構が Chain[0]，両腕を表す直鎖機構が Chain[1]，Chain[2] となっている．**

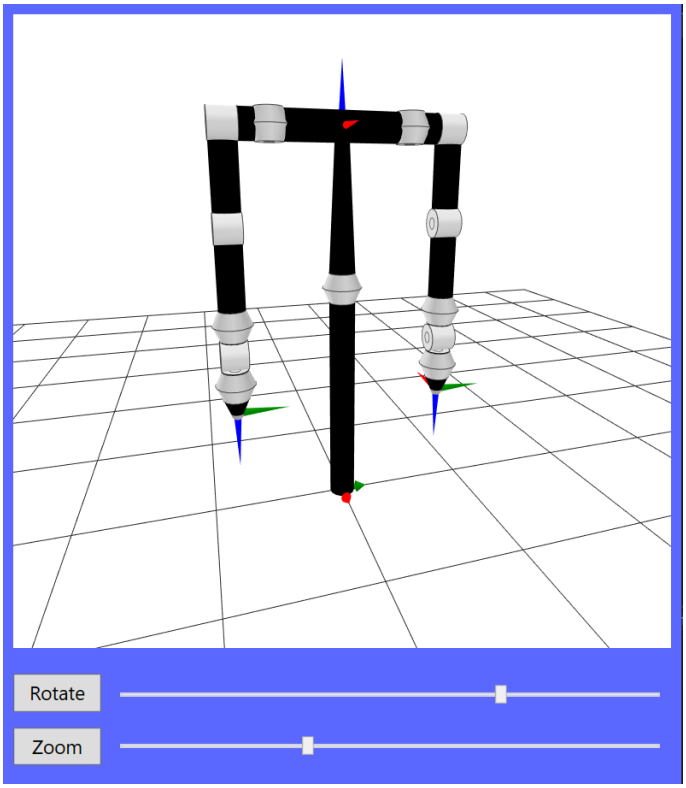


図 16: 双腕マニピュレータを作成するサンプルコードの実行結果

17　4 足歩行ロボットの作成と運動学の計算

4 足歩行ロボットを作成するサンプルコードを以下に示す.

MainWindow.xaml.cs 内の関数 Setup()，Draw()，Button_Click() の周辺を変更

```
Robot Robot = new Robot(6, 3, 3, 3, 3);

void Setup()
{
    Robot.SetFloatingJoint6DOF(0.6f, 0.4f, 0.12f);
    Robot[0, 2].q = 0.41f;
    Robot[0, 2].qHome = 0.41f;

    Robot[1, 0].LinkInit.SetValue(0.3f, 0.2f, 0);
    Robot[1, 0].AxisInit.SetUnitVectorX();
    Robot[1, 1].LinkInit.SetZeroVector();
    Robot[1, 1].AxisInit.SetUnitVectorY();
    Robot[1, 2].LinkInit.SetValue(-0.06f, 0, -0.2f);
    Robot[1, 2].AxisInit.SetUnitVectorY();
    Robot[1, 3].LinkInit.SetValue(0.06f, 0, -0.2f);
    Robot.Kinematics.Target[1].SetDOF3();

    Robot[2, 0].LinkInit.SetValue(0.3f, -0.2f, 0);
    Robot[2, 0].AxisInit.SetUnitVectorX();
    Robot[2, 1].LinkInit.SetZeroVector();
    Robot[2, 1].AxisInit.SetUnitVectorY();
    Robot[2, 2].LinkInit.SetValue(-0.06f, 0, -0.2f);
    Robot[2, 2].AxisInit.SetUnitVectorY();
    Robot[2, 3].LinkInit.SetValue(0.06f, 0, -0.2f);
    Robot.Kinematics.Target[2].SetDOF3();

    Robot[3, 0].LinkInit.SetValue(-0.3f, 0.2f, 0);
    Robot[3, 0].AxisInit.SetUnitVectorX();
    Robot[3, 1].LinkInit.SetZeroVector();
    Robot[3, 1].AxisInit.SetUnitVectorY();
    Robot[3, 2].LinkInit.SetValue(-0.06f, 0, -0.2f);
    Robot[3, 2].AxisInit.SetUnitVectorY();
    Robot[3, 3].LinkInit.SetValue(0.06f, 0, -0.2f);
    Robot.Kinematics.Target[3].SetDOF3();

    Robot[4, 0].LinkInit.SetValue(-0.3f, -0.2f, 0);
    Robot[4, 0].AxisInit.SetUnitVectorX();
    Robot[4, 1].LinkInit.SetZeroVector();
    Robot[4, 1].AxisInit.SetUnitVectorY();
    Robot[4, 2].LinkInit.SetValue(-0.06f, 0, -0.2f);
    Robot[4, 2].AxisInit.SetUnitVectorY();
    Robot[4, 3].LinkInit.SetValue(0.06f, 0, -0.2f);
    Robot.Kinematics.Target[4].SetDOF3();

    Robot.Kinematics.ForwardKinematics();
```

```
    Robot.Kinematics.SetJointLinkRadiusAuto();
    Robot.Kinematics.SetTargetsToEffector();

    Robot[0, 0].JointObj.Radius = 0;
    Robot[0, 1].JointObj.Radius = 0;
    Robot[0, 2].JointObj.Radius = 0;
}

void Draw()
{
    Robot.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Robot[0, 0].q += 0.1f;
    Robot[0, 1].q += 0.1f;
    Robot[0, 5].q += 0.1f * PI;
    Robot.Kinematics.ForwardKinematics();
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    Robot.Kinematics.ReturnHomePosition();
    Robot.Kinematics.ForwardKinematics();
    Robot.Kinematics.SetTargetsToEffector();
}

void Button3_Click(object sender, RoutedEventArgs e)
{
    Robot.Kinematics.Target[1].Position[0] += 0.1f;
    Robot.Kinematics.Target[1].Position[1] += 0.1f;
    Robot.Kinematics.Target[1].Position[2] += 0.1f;
    Robot.Kinematics.InverseKinematics();
}
```

上記のサンプルを実行すると図 17 のようになる．このロボットでは，胴体の位置・姿勢を表す直鎖機構が **Chain[0]**，四肢を表す直鎖機構が **Chain[1]**〜**Chain[4]** となっている．

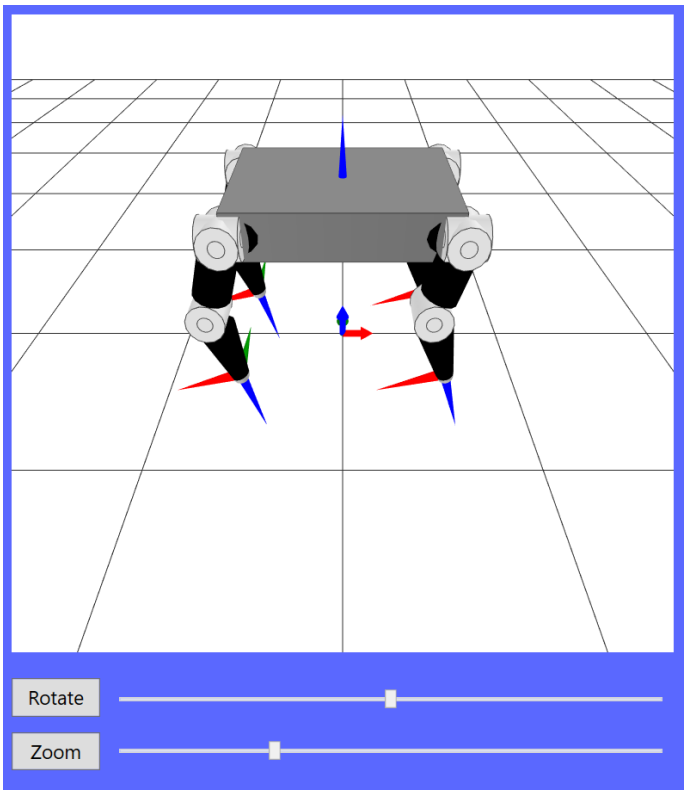


図 17: 4 足歩行ロボットを作成するサンプルコードの実行結果

胴体の位置・姿勢は，仮想的な直進対偶 3 つと回転対偶 3 つで表されている．関数 SetFloatingJoint 6DOF(x, y, z) を実行することで自動的に Chain[0] 内の各関節が直進対偶 3 つと回転対偶 3 つに設定され，引数 x, y, z は順に胴体の幅，奥行き，高さとなる．また，胴体の位置を表す 3 つの直進対偶は太さ（Joint.Radius）を 0 にすることで非表示にしている．胴体の位置を表す変位は Robot[0, 0].q, Robot[0, 1].q, Robot[0, 2].q の 3 つとなり，姿勢を表す変位は Robot[0, 3].q, Robot[0, 4].q, Robot[0, 5].q の 3 つとなる．このロボットでは逆運動学を計算する際に与える目標が 5 つ（Target[0]〜Target[4]）となり，Target[0] は胴体に与える目標の位置・姿勢となる．例えば，胴体の高さ（Target[0].Position[2]）を低くして逆運動学を計算すると，ロボットを伏せさせることができる．

18　ロボットと障害物の干渉チェックと自重トルクの計算

まず，3 軸のロボットと障害物の干渉チェックをするサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Setup()，Draw()，Button_Click() の周辺を変更

```
Robot Robot = new Robot(3);
Cuboid Obstacle = new Cuboid(0.5f, 0.5f, 0.4f);

void CollisionCheck()
{
    if(Robot.IsCollision(Obstacle))
    {
```

```
        Obstacle.Color.SetRed();
    }
    else
    {
        Obstacle.Color.SetGray();
    }
}

void Setup()
{
    Robot[0, 0].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 0].AxisInit.SetUnitVectorZ();
    Robot[0, 1].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 1].AxisInit.SetUnitVectorY();
    Robot[0, 2].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 2].AxisInit.SetUnitVectorY();
    Robot[0, 3].LinkInit.SetValue(0, 0, 0.15f);
    Robot.Kinematics.ForwardKinematics();
    Robot.Kinematics.SetJointLinkRadiusAuto();

    Obstacle.Position.Follow(Keyboard.ShiftVector[0]);
    Obstacle.Rotate.Follow(Keyboard.CtrlMatrix[0]);

    Parallel.RunEndless(CollisionCheck, 100);
}

void Draw()
{
    Robot.Draw();
    Obstacle.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Obstacle.SetPositionOffset(0, 0, 0.2f);
    Obstacle.Position.Follow(Robot.Kinematics.Chain[0].pe);
    Obstacle.Rotate.Follow(Robot.Kinematics.Chain[0].Re);
}
```

上記のサンプルコードでは，関数 CollisionCheck() を新たに作成し，その関数を 100ms に 1 回の周期で並列実行（Parallel.Run）させている．障害物はキーボードの「Shift ＋矢印」または「Ctrl ＋矢印」キーで動かすことができる．ロボットと障害物が干渉すると，障害物が赤くなる．ボタン 1 を押すと，障害物がロボットの効果器（手先）に貼り付く．このように自動追従（Follow）の機能は，ロボットの部位に対しても適用できる．ロボットに貼り付けられた障害物を他の物体（Cuboid や Pillar 型などの変数）と干渉チェックをすることもできる．

次に，各関節に生じる自重トルク（ロボット自身の重さによって生じるトルク）を計算するサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Setup(), Draw(), Button_Click() の周辺を変更

```
Robot Robot = new Robot(3);

void Setup()
{
    Robot[0, 0].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 0].AxisInit.SetUnitVectorZ();
    Robot[0, 1].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 1].AxisInit.SetUnitVectorY();
    Robot[0, 2].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 2].AxisInit.SetUnitVectorY();
    Robot[0, 3].LinkInit.SetValue(0, 0, 0.15f);
    Robot.Kinematics.ForwardKinematics();
    Robot.Kinematics.SetJointLinkRadiusAuto();
}

void Draw()
{
    Robot.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Robot[0, 1].q += 0.1f;
    Robot.Kinematics.ForwardKinematics();

    Robot.Kinematics.CalcTauSelf();
    Console.WriteLine(Robot[0, 0].TauSelf);
    Console.WriteLine(Robot[0, 1].TauSelf);
    Console.WriteLine(Robot[0, 2].TauSelf);
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    Robot[0, 0].Mass = 0.4f;
    Robot[0, 1].Mass = 0.4f;
    Robot[0, 2].Mass = 0.4f;
    Robot[0, 3].Mass = 0.4f;
}
```

ボタン 1 を押すと各関節の自重トルク（単位は Nm）が計算され，その結果が Pair クラス内の変数 TauSelf に格納される．ボタン 2 を押すと，各関節の質量が 0.4 [kg] に上書きされる（初期値は 0.2 [kg]）．実際のロボットではリンクと関節の両方に質量があるが，ここでの計算では両者の質量をまとめて関節側に設定している．また，Robot[0, 3].Mass は手先の質量となる．直鎖 i の自由度を n_i とすると，直鎖 i の効果器の質量は Robot[i, n_i].Mass である．例えば 6 自由度のロボットアームであれば，効果器（手先）の質量は Robot[0, 6].Mass となる．また，関節の軸ベクトルが重力の方向（z 軸）と平行であった場合，その関節の自重トルク（TauSelf）は常に 0 となる．

19　多関節ロボットの軌道生成

まず，それぞれの関節に目標角度を 1 つだけ設定し，その目標角度に到達するような軌道を生成するサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Setup()，Draw()，Button_Click() の周辺を変更

```
Robot Robot = new Robot(3);

void Setup()
{
    Robot[0, 0].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 0].AxisInit.SetUnitVectorZ();
    Robot[0, 1].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 1].AxisInit.SetUnitVectorY();
    Robot[0, 2].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 2].AxisInit.SetUnitVectorY();
    Robot[0, 3].LinkInit.SetValue(0, 0, 0.15f);
    Robot.Kinematics.ForwardKinematics();
    Robot.Kinematics.SetJointLinkRadiusAuto();
}

void Draw()
{
    Robot.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Robot[0, 0].qTar = -1;
    Robot[0, 1].qTar = -1;
    Robot[0, 2].qTar = -1;
    Robot.Trajectory.MoveToJointTargets(1000, 1);
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    Robot[0, 0].qTar = 1;
    Robot[0, 1].qTar = 1;
    Robot[0, 2].qTar = 1;
    Robot.Trajectory.MoveToJointTargets(3000, 1);
}
```

各関節の目標角度が qTar であり，関数 MoveToJointTargets(t, n) を実行すると，t [ms] が経過した時点で qTar に到達するような軌道が生成され，ロボットが動き出す．ここでは 1 つの関節につき目標角度を 1 つのみ設定しているため $n = 1$ とする．なお，上記のサンプルでは 3 自由度のロボットを用いているが，同様な手順で全ての関節に目標角度を設定することが可能であり，軌道の生成もできる．

次に，1 つの関節に複数（n 個）の目標角度を設定し，それらを順に辿るような軌道を生成するサ

ンプルコードを以下に示す.

MainWindow.xaml.cs 内の関数 Setup()，Draw()，Button_Click() の周辺を変更

```
Robot Robot = new Robot(3);

void Setup()
{
    Robot[0, 0].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 0].AxisInit.SetUnitVectorZ();
    Robot[0, 1].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 1].AxisInit.SetUnitVectorY();
    Robot[0, 2].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 2].AxisInit.SetUnitVectorY();
    Robot[0, 3].LinkInit.SetValue(0, 0, 0.15f);
    Robot.Kinematics.ForwardKinematics();
    Robot.Kinematics.SetJointLinkRadiusAuto();
}

void Draw()
{
    Robot.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Robot[0, 0].qTars[0] = -1;
    Robot[0, 0].qTars[1] = 1;
    Robot[0, 0].qTars[2] = -1;

    Robot[0, 1].qTars[0] = 1;
    Robot[0, 1].qTars[1] = -1;
    Robot[0, 1].qTars[2] = 1;

    Robot[0, 2].qTars[0] = -1;
    Robot[0, 2].qTars[1] = 0;
    Robot[0, 2].qTars[2] = 1;

    Robot.Trajectory.MoveToJointTargets(6000, 3);
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    Robot.Kinematics.Target[0].SetDOF3();

    Robot.Kinematics.Target[0].Position.SetValue(0.2f, 0.1f, 0.3f);
    Robot.Kinematics.InverseKinematics();
    Robot.Trajectory.SetCurrentJointToTarget(0);

    Robot.Kinematics.Target[0].Position.SetValue(0.2f, 0, 0.3f);
    Robot.Kinematics.InverseKinematics();
```

```
    Robot.Trajectory.SetCurrentJointToTarget(1);

    Robot.Kinematics.Target[0].Position.SetValue(0.2f, -0.1f, 0.3f);
    Robot.Kinematics.InverseKinematics();
    Robot.Trajectory.SetCurrentJointToTarget(2);

    Robot.Kinematics.ReturnHomePosition();
    Robot.Trajectory.MoveToJointTargets(5000, 3);
}
```

関節の目標角度を記録する変数は配列 qTars[]（最大で 100 個）である．ボタン 1 を押すと合計で 6000 [ms] かけて 3 つの目標角度を辿るような軌道が生成され，ロボットが動き出す．ボタン 2 を押すと逆運動学の解が各関節の目標角度として設定される．関数 SetCurrentJointToTarget(i) は，現在の各関節角度 q を目標角度 qTars[i] に記録する．

Trajectory クラス内の変数と関数の一覧を以下に示す．

Trajectory クラスに含まれる変数と関数の一覧	
MoveToJointTargets(int timeMs, int targetNum)	各関節の目標値を辿るような軌道を生成する． 引数 1 は遷移時間 [ms]，引数 2 は目標値の数．
MoveToJointTargets(int[] timeMs)	各関節の目標値を辿るような軌道を生成する． 引数の配列は，それぞれの遷移時間 [ms] を表す．
SetCurrentJointToTarget(int qTarNum)	現在の各関節変位を qTars に記録する． qTars[i] の番号 i は引数で指定する．
bool IsMoveCompleted	移動が完了しているか否かを返す関数．

20　動作確認用の半透明なロボットの生成

前述した方法でメインのロボットを作成した後に，動作確認用の半透明なロボットを生成するサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Setup()，Draw()，Button_Click() の周辺を変更

```
Robot Robot = new Robot(6);
Robot SubRobot = new Robot(6);

void Setup()
{
    Robot[0, 0].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 0].AxisInit.SetUnitVectorZ();
    Robot[0, 1].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 1].AxisInit.SetUnitVectorY();
    Robot[0, 2].LinkInit.SetValue(0, 0, 0.30f);
    Robot[0, 2].AxisInit.SetUnitVectorY();
    Robot[0, 3].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 3].AxisInit.SetUnitVectorZ();
    Robot[0, 4].LinkInit.SetValue(0, 0, 0.15f);
```

```
    Robot[0, 4].AxisInit.SetUnitVectorY();
    Robot[0, 5].LinkInit.SetValue(0, 0, 0.15f);
    Robot[0, 5].AxisInit.SetUnitVectorZ();
    Robot[0, 6].LinkInit.SetValue(0, 0, 0.15f);

    Robot.Kinematics.ForwardKinematics();
    Robot.Kinematics.SetJointLinkRadiusAuto();
    Robot.Kinematics.SetTargetsToEffector();

    SubRobot.CopyJointLinkStructure(Robot);
    SubRobot.Kinematics.ForwardKinematics();
    SubRobot.Kinematics.SetJointLinkRadiusAuto();
    SubRobot.Transparency = 60;
}

void Draw()
{
    Robot.Draw();
    SubRobot.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    SubRobot[0, 1].q = 0.5f * PI;
    SubRobot.Kinematics.ForwardKinematics();
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    Robot.CopyJointAngles(SubRobot);
    Robot.Kinematics.ForwardKinematics();
}

void Button3_Click(object sender, RoutedEventArgs e)
{
    Robot.CopyJointTrajectory(SubRobot);
}
```

Robot クラス内の Transparency（初期値は 255）は透明度を表しており，これが 0 に近くなるほどロボットは透明になる．ボタン 1 を押すと図 18 のようになり，ボタン 2 を押すと，Robot が SubRobot の各関節変位をコピーする．また，ボタン 3 を押すと Robot が SubRobot の関節軌道（qTars[]）をコピーする．

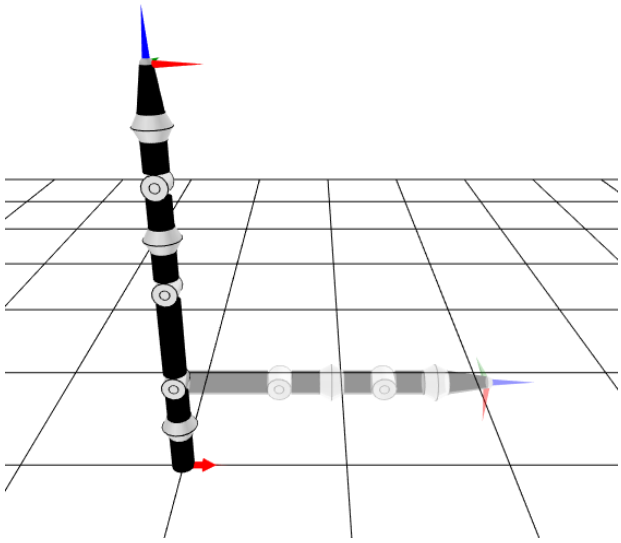


図 18: 半透明なロボット（Transparency = 60）

21　対向２輪ロボットの作成と LiDAR との結合

まず，対向２輪ロボットの骨格を生成し，目標速度を設定して微分逆運動学を計算するサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Setup()，Draw()，Button_Click() の周辺を変更

```
MobileRobot.TwoWheel TwoWheel = new MobileRobot.TwoWheel();

void Setup()
{
    TwoWheel.WheelRadius = 0.1f;
    TwoWheel.TreadWidth = 0.4f;
    TwoWheel.Distance = 0.3f;
}

void Draw()
{
    TwoWheel.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    TwoWheel.StartInverseKinematics();
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    TwoWheel.Velocity[0] = 0.1f;
    TwoWheel.Velocity[1] = 0;
```

```
}

void Button3_Click(object sender, RoutedEventArgs e)
{
    TwoWheel.Velocity[0] = 0;
    TwoWheel.Velocity[1] = 0.1f;
}
```

上記のサンプルコードを実行すると，図 19 のようになる．この段階では車体の基準点と 2 つの車輪のみが表示される．

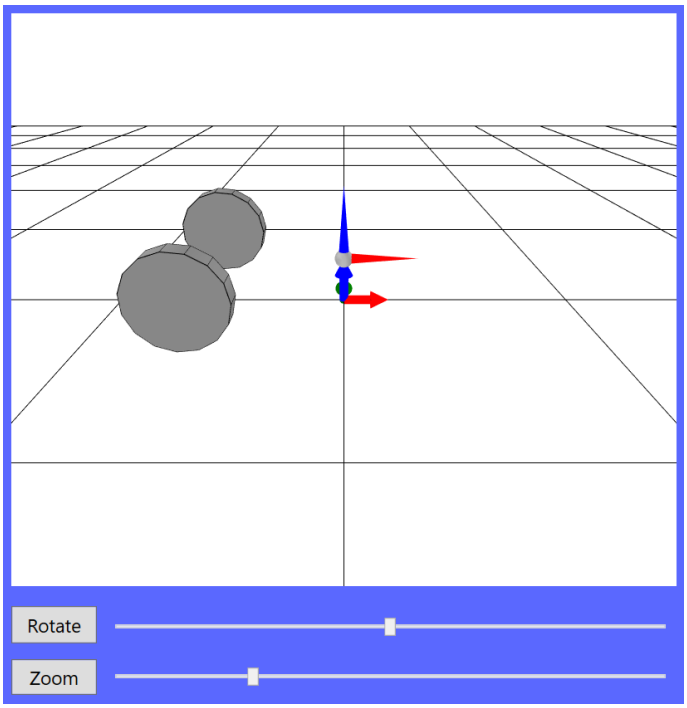


図 19: 対向 2 輪ロボットの骨格

対向 2 輪ロボットの骨格は，車輪の半径（WheelRadius），トレッド幅（TreadWidth），車体の基準点と車軸との距離（Distance）を表す 3 つのパラメータから形成されている．ボタン 1 を押すと一定の周期で微分逆運動学が計算され続けるようになる．初期設定では車体の速度が 0 となっているため，まだロボットは動かない．ボタン 2 を押すと車体の速度が設定されるため，ロボットが動き出す．ボタン 3 を押すと車体の速度が別の値に設定される．

TwoWheel クラスに含まれる変数と関数の一覧を以下に示す．

TwoWheel クラスの変数と関数の一覧	
Vector Position	車体の位置を表す 3 次元のベクトル．
RotationMatrix Rotate	車体の姿勢を表す回転行列．
Vector Velocity	車体の速度を表す 2 次元のベクトル．
Vector WheelVelocity	車輪の角速度を表す 2 次元のベクトル．

Vector WheelAngle	車輪の角度を表す 2 次元のベクトル.
Matrix J	微分順運動学の係数行列.
float WheelRadius	車輪の半径.
float TreadWidth	トレッド幅.
float Distance	車体の基準点と車軸との距離.
byte Transparency	ロボット全体の透明度（初期値は 255）.
StartInverseKinematics()	微分逆運動学を一定の周期で計算し続ける.
StartForwardKinematics()	微分順運動学を一定の周期で計算し続ける.
uint SamplingTimeMs	微分運動学を計算する周期. 単位はミリ秒.
InverseKinematics()	微分逆運動学を 1 周期分だけ計算する.
ForwardKinematics()	微分順運動学を 1 周期分だけ計算する.
Draw()	ロボットを描画する関数.

次に，対向 2 輪ロボットの骨格にボディや LiDAR を取り付けるサンプルコードを示す.

MainWindow.xaml.cs 内の関数 Setup()，Draw()，Button_Click() の周辺を変更

```
MobileRobot.TwoWheel TwoWheel = new MobileRobot.TwoWheel();
Cuboid Body = new Cuboid(0.5f, 0.36f, 0.1f);
LiDAR LiDAR = new LiDAR();
Cuboid Obstacle = new Cuboid();

void Setup()
{
    Body.Position.Follow(TwoWheel.Position);
    Body.Rotate.Follow(TwoWheel.Rotate);
    Body.SetPositionOffset(-0.12f, 0, 0);

    LiDAR.Position.Follow(TwoWheel.Position);
    LiDAR.Rotate.Follow(TwoWheel.Rotate);
    LiDAR.SetPositionOffset(0, 0, 0.08f);

    Obstacle.Position[0] = 1;
    Obstacle.Position[1] = 1;
    Obstacle.Position[2] = 0.2f;
}

void Draw()
{
    TwoWheel.Draw();
    Body.Draw();
    LiDAR.Draw();
    Obstacle.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
```

```
    float[][] pointCloud = LiDAR.PointCloud(Obstacle);
}
```

上記のサンプルコードを実行し，ボタン 1 を押すと図 20 のようになる．ボディと LiDAR の位置・姿勢は，関数 Follow() により車体の位置・姿勢に自動で追従するようになる．すなわち，車体が移動すれば自動的にボディと LiDAR も移動する．ただし，点群データはボタン 1 を押さないと更新されない．

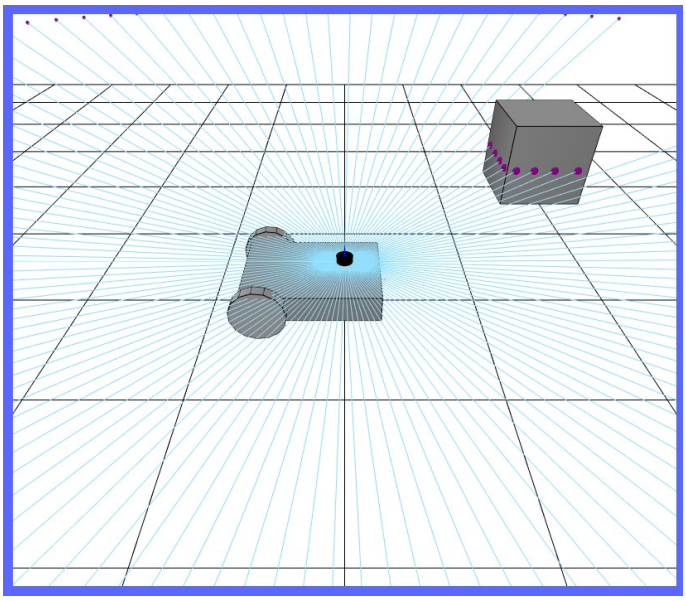


図 20: ボディと LiDAR が取り付けられた対向 2 輪ロボット

微分逆運動学の繰り返し計算と点群データの取得を同時に実行したい場合は，両者を適当な関数にまとめて，その関数を並列分散処理する．例えば，関数 StartInverseKinematics() の代わりに以下のコードを追加する．

```
void StepProcces()
{
    TwoWheel.InverseKinematics();
    float[][] pointCloud = LiDAR.PointCloud(Obstacle);
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Parallel.RunEndless(StepProcces, TwoWheel.SamplingTimeMs);
}
```

また，本書で前述した多関節ロボットを対向 2 輪ロボットの上に載せる場合には，以下のコードを関数 Setup() に追加する．

```
Robot.Kinematics.BasePosition.Follow(TwoWheel.Position);
Robot.Kinematics.BaseRotate.Follow(TwoWheel.Rotate);
```

上記のコードにより，多関節ロボットの順運動学に対向 2 輪ロボットの位置・姿勢が反映されるようになる．

22　メカナム台車の作成と LiDAR との結合

まず，メカナム台車の骨格を生成し，目標の速度と角速度を設定して微分逆運動学を計算するサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Setup()，Draw()，Button_Click() の周辺を変更

```
MobileRobot.Mecanum Mecanum = new MobileRobot.Mecanum();

void Setup()
{
    Mecanum.WheelRadius = 0.08f;
    Mecanum.TreadWidth = 0.5f;
    Mecanum.Distance = 0.5f;
}

void Draw()
{
    Mecanum.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Mecanum.StartInverseKinematics();
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    Mecanum.Velocity[0] = 0.1f;
    Mecanum.Velocity[1] = 0;
    Mecanum.Velocity[2] = 0.1f;
}

void Button3_Click(object sender, RoutedEventArgs e)
{
    Mecanum.Velocity[0] = 0;
    Mecanum.Velocity[1] = 0.1f;
    Mecanum.Velocity[2] = -0.1f;
}
```

上記のサンプルコードを実行すると，図 21 のようになる．この段階では車体の基準点と 4 つの車輪のみが表示される．

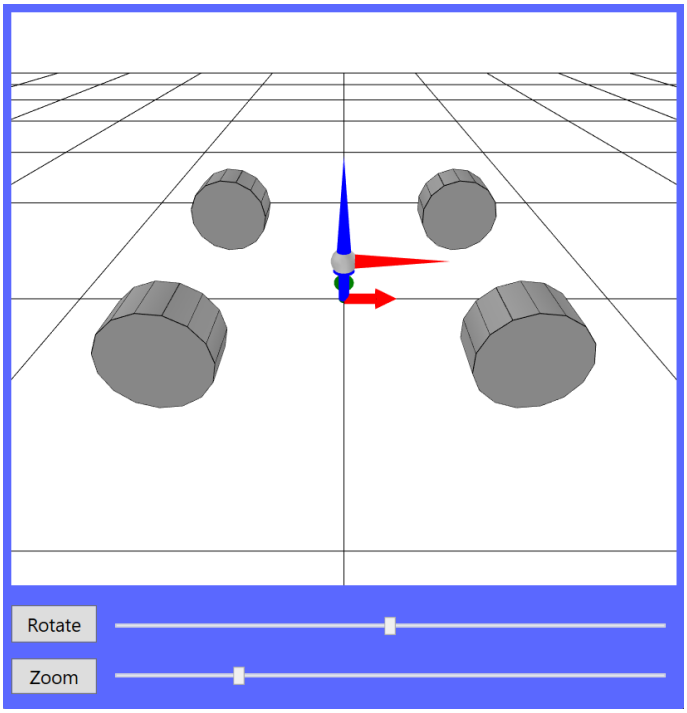


図 21: メカナム台車の骨格

メカナム台車の骨格は，車輪の半径（WheelRadius），トレッド幅（TreadWidth），前輪と後輪の軸間距離（Distance）を表す 3 つのパラメータから形成されている．ボタン 1 を押すと一定の周期で微分逆運動学が計算され続けるようになる．初期設定では車体の速度が 0 となっているため，まだロボットは動かない．ボタン 2 を押すと車体の速度と角速度が設定されるため，ロボットが動き出す．ボタン 3 を押すと車体の速度と角速度が別の値に設定される．ベクトル Mecanum.Velocity の各成分は，順に x, y 軸方向の並進速度と z 軸まわりの角速度を表す．また，各車輪の角速度と角度は，それぞれ Mecanum.WheelVelocity と Mecanum.WheelAngle という 4 次元のベクトルに格納されている．車輪の番号（インデックス）は，図 21 の右下にある車輪を 0 とし，そこから反時計回りに 1, 2, 3 と割り振られている．

Mecanum クラスに含まれる変数と関数の一覧を以下に示す．

Mecanum クラスの変数と関数の一覧	
Vector Position	車体の位置を表す 3 次元のベクトル．
RotationMatrix Rotate	車体の姿勢を表す回転行列．
Vector Velocity	車体の速度と角速度を表す 3 次元のベクトル．
Vector WheelVelocity	車輪の角速度を表す 4 次元のベクトル．
Vector WheelAngle	車輪の角度を表す 4 次元のベクトル．
Matrix Jinv	微分逆運動学の係数行列．
float WheelRadius	車輪の半径．
float TreadWidth	トレッド幅．
float Distance	前輪と後輪の軸間距離．

byte Transparency	ロボット全体の透明度（初期値は 255）.
StartInverseKinematics()	微分逆運動学を一定の周期で計算し続ける.
uint SamplingTimeMs	微分運動学を計算する周期．単位はミリ秒.
InverseKinematics()	微分逆運動学を 1 周期分だけ計算する.
Draw()	ロボットを描画する関数.

次に，メカナム台車の骨格にボディや LiDAR を取り付けるサンプルコードを示す.

MainWindow.xaml.cs 内の関数 Setup()，Draw()，Button_Click() の周辺を変更

```
MobileRobot.Mecanum Mecanum = new MobileRobot.Mecanum();
Cuboid Body = new Cuboid(0.6f, 0.4f, 0.2f);
LiDAR LiDAR = new LiDAR();
Cuboid Obstacle = new Cuboid();

void Setup()
{
    Body.Position.Follow(Mecanum.Position);
    Body.Rotate.Follow(Mecanum.Rotate);
    Body.SetPositionOffset(0, 0, 0.1f);
    Body.Color.SetYellow();

    LiDAR.Position.Follow(Mecanum.Position);
    LiDAR.Rotate.Follow(Mecanum.Rotate);
    LiDAR.SetPositionOffset(0, 0, 0.22f);

    Obstacle.Position[0] = 1;
    Obstacle.Position[1] = 1;
    Obstacle.Position[2] = 0.2f;
}

void Draw()
{
    Mecanum.Draw();
    Body.Draw();
    LiDAR.Draw();
    Obstacle.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    float[][] pointCloud = LiDAR.PointCloud(Obstacle);
}
```

上記のサンプルコードを実行し，ボタン 1 を押すと図 22 のようになる．ボディと LiDAR の位置・姿勢は，関数 Follow() により車体の位置・姿勢に自動で追従するようになる．すなわち，車体が移動すれば自動的にボディと LiDAR も移動する．ただし，点群データはボタン 1 を押さないと更新されない.

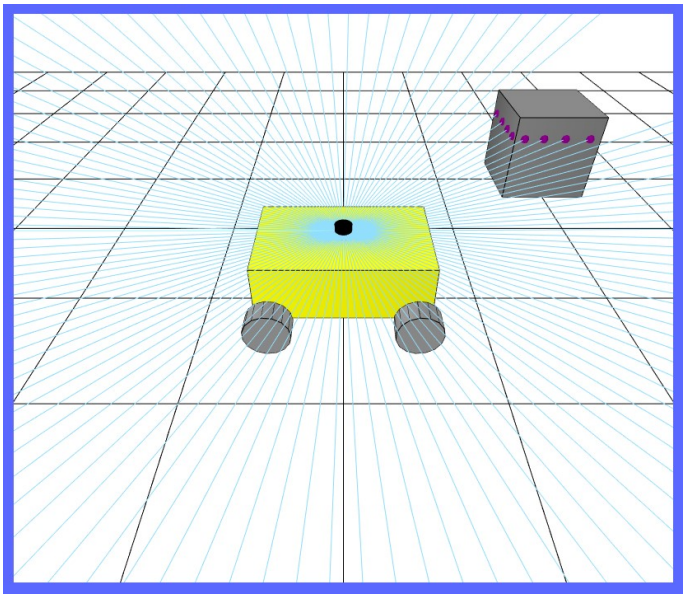


図 22: ボディと LiDAR が取り付けられたメカナム台車

微分逆運動学の繰り返し計算と点群データの取得を同時に実行したい場合は，両者を適当な関数にまとめて，その関数を並列分散処理する．例えば，関数 StartInverseKinematics() の代わりに以下のコードを追加する．

```
void StepProcces()
{
    Mecanum.InverseKinematics();
    float[][] pointCloud = LiDAR.PointCloud(Obstacle);
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Parallel.RunEndless(StepProcces, Mecanum.SamplingTimeMs);
}
```

また，本書で前述した多関節ロボットをメカナム台車の上に載せる場合には，以下のコードを関数 Setup() に追加する．

```
Robot.Kinematics.BasePosition.Follow(Mecanum.Position);
Robot.Kinematics.BaseRotate.Follow(Mecanum.Rotate);
```

上記のコードにより，多関節ロボットの順運動学にメカナム台車の位置・姿勢が反映されるようになる．

23　ゲームパッドとの通信

任天堂 Switch，PlayStation，X Box などのゲームパッドから，ボタンやジョイスティックの入力状態を取得するサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Button_Click() の周辺を変更

```
GamePad GamePad = new GamePad();

void Button1_Click(object sender, RoutedEventArgs e)
{
    GamePad.GetState();
    ConsoleWriteState();

    if (GamePad.A)
    {
        // Write Button A Process Here.
    }
    else if (GamePad.B)
    {
        // Write Button B Process Here.
    }
    else if (GamePad.X)
    {
        // Write Button X Process Here.
    }
    else if (GamePad.Y)
    {
        // Write Button Y Process Here.
    }
}
```

ゲームパッドを PC に接続した状態でボタン 1 を押すと，ゲームパッドの状態（どのキーが押されているか）を取得できる．

なお，ゲームパッドには X Input 方式と Direct Input 方式があり，この機能では X Input 方式を前提としたキー割り当てになっている．よって，Direct Input 方式のゲームパッドであった場合，キー割り当てが表記通りにならない場合がある点に注意されたい．例えば，ボタン B を押すと GamePad.A が true になるなどの現象が起こり得る．また，複数のゲームパッドが同時に接続されている場合は，関数 GetState() の引数でゲームパッドの ID（0〜3 の整数）を指定できる．

GamePad クラスに含まれる変数と関数の一覧を以下に示す．

GamePad クラスに含まれる変数と関数の一覧	
GetState()	ゲームパッドの状態を取得する．
bool A	ボタン A の状態．
bool B	ボタン B の状態．
bool X	ボタン X の状態．
bool Y	ボタン Y の状態．
bool R	ボタン R の状態．
bool L	ボタン L の状態．
float RT	ボタン RT の状態（押込の度合）．

float LT	ボタン LT の状態（押込の度合）.
bool Up	左十字キー上の状態.
bool Down	左十字キー下の状態.
bool Right	左十字キー右の状態.
bool Left	左十字キー左の状態.
float LeftStickX	左スティック（横方向）の状態.
float LeftStickY	左スティック（縦方向）の状態.
bool LeftStickDown	左スティック（真下方向への押込）の状態.
float RightStickX	右スティック（横方向）の状態.
float RightStickY	右スティック（縦方向）の状態.
bool RightStickDown	右スティック（真下方向への押込）の状態.
ConsoleWriteState()	取得した状態を全てコンソールに表示する.

24　Arduino・M5Stack との通信

Arduino と M5Stack は，どちらも同じ手順で通信ができるため，後述する「Arduino」を「M5Stack」と読み替えることで M5Stack との通信をすることもできる．なお，M5Stack に Arduino IDE でプログラムを書き込む方法については，M5Stack の公式ドキュメントを参照されたい．

Arduino との通信をするためには，まず Arduino 側に専用のプログラムを書き込む必要がある．Arduino 側のソースコードは本稿と同封されているファイル「Arduino.ino」（または M5Stack.ino）に記載されており，Arduino IDE というソフトウェアを用いることで閲覧や編集，書き込みができる．Arduino のコードに含まれる重要な変数と関数は以下の 3 つである（M5Stack のコードも同じ変数と関数を含む）.

- **int receiveData[128]**
  PC 側から送られてきたデータが格納される.
- **void Receive()**
  Arduino 側のシリアル受信バッファからデータを読み取り，受信データを receiveData に上書きする.
- **void Send(int intArray[], int arrayLength)**
  引数 1 に渡した int 型の配列を PC 側に送信する．送信する配列の要素数を引数 2 で指定する．例えば，Send(receiveData, 6) とすると receiveData[0]〜receiveData[5] の 6 つを PC 側に送信する.

初期設定では，200 ミリ秒に 1 回受信データ（receiveData）をそのまま PC に返信するプログラムとなっている.

次に，PC 側のサンプルコードを示す（M5Stack の場合は，コード内の Arduino を全て M5Stack に置き換える）.

MainWindow.xaml.cs 内の関数 Button_Click() の周辺を変更

```
SerialDevice.Arduino Arduino = new SerialDevice.Arduino(9600);

void Button1_Click(object sender, RoutedEventArgs e)
{
    Arduino.PortOpen("COM5");
}

short count = 0;

void Button2_Click(object sender, RoutedEventArgs e)
{
    Arduino.Send(count, count, count);
    count++;
}

void Button3_Click(object sender, RoutedEventArgs e)
{
    Console.WriteLine(String.Join(",", Arduino.ReceiveShortArray));
}
```

上記のサンプルでは COM ポートを COM5 としているが，COM の番号は適宜書き換える必要がある．COM ポートは Windows のデバイスマネージャーから確認できる．ボタン 1 を押すと Arduino と接続し，ボタン 2 を押すと Arduino に short 型の変数 count を送信する．Arduino における int は，C#では short となることに注意されたい．ボタン 3 を押すと Arduino から受信したデータがコンソールに出力される．PC から送信したデータ（count）を Arduino から受信していれば通信に成功している．

関数 PortOpen(string deviceName) の引数に渡す文字列には大きく 2 つの選択肢がある．1 つ目はデバイス名に含まれている単語を指定する方式である．例えば Arduino Uno のデバイス名は「Arduino Uno」となるので，文字列「Arduino」や「Uno」,「Arduino Uno」を引数に渡す．2 つ目は COM を指定する方式であり，例えば Arduino に COM5 が割り振られている場合は文字列「COM5」または「5」を引数に渡す．また，初期化子 new Arduino(int baudRate) の引数には設定したいボーレートを渡す．PC 側のボーレートを変更した際は，Arduino 側のボーレートも変更しなければならないことに注意されたい．

ソースコードを編集する際は，Arduino 側のシリアル受信バッファは 64 byte しかない点に注意しなければならない．例えば Arduino 側のソースコードに関数 delay(1000); を追加した場合，Arduino が停止している間に PC が送信したデータの総量が 64 byte を超えると通信システムが破綻する．なお，Arduino における int 型（C#では short 型）の変数を 1 つ表すために必要な容量は 2 byte である．また，小数を送受信する方法の 1 つとして，送信側でデータを 1000 倍してから整数に変換し，受信側で 1000 分の 1 を掛けるという方法が挙げられる．この手法を用いる際には，int 型には上限と下限があることや有効桁数に注意しなければならない．

Arduino クラスと M5Stack クラスに含まれる変数と関数は，どちらも名称や使い方が同じであり，

それらの一覧を以下に示す．

Arduino クラスと M5Stack クラスに含まれる変数と関数の一覧	
PortOpen(string deviceName)	引数で指定したポートを開く．
PortClose()	ポートを閉じる．
short[] ReceiveShortArray	受信データが格納されている配列．
ConsoleWriteReceiveData()	受信データをコンソールに表示する．
Send(params short[] sendData)	引数に渡した配列を Arduino に送信する．
SetBaudRate(int baudRate)	引数で指定したボーレートに設定する．

25 Dynamixel との通信

まず，U2D2（Power Hub 付き）などのシリアル変換機を用いて図 23 のように PC と Dynamixel を接続する．なお，シリアル変換機ではなく OpenRB（または OpenCM）というマイコンを経由して Dynamixel を動かす場合は，マイコンにシリアル変換用のコードを書き込むことで後述するサンプルコードを用いることができるようになる．また，型番の古い Dynamixel（DX/RX/EX/AX/MX シリーズ）は後述するサンプルコードでは動かすことができない点に注意されたい．

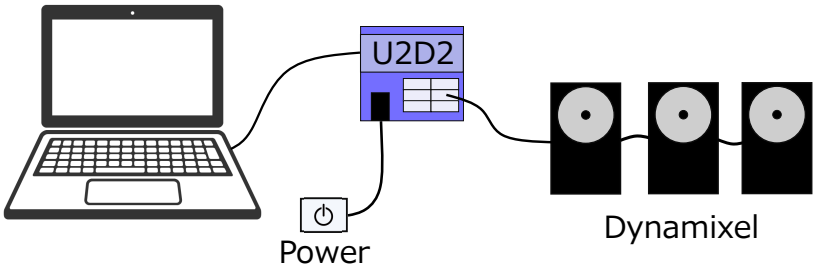


図 23: シリアル変換機（U2D2）の接続方法

Dynamixel との通信を行うサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Button_Click() の周辺を変更

```
SerialDevice.Dynamixel Dynamixel = new SerialDevice.Dynamixel(57600);
byte[] id = new byte[3] { 1, 2, 3 };

void Button1_Click(object sender, RoutedEventArgs e)
{
    Dynamixel.PortOpen("COM5");
}

void Button2_Click(object sender, RoutedEventArgs e)
{
    Dynamixel.TorqueEnable(id);
}
```

```
void Button3_Click(object sender, RoutedEventArgs e)
{
    Dynamixel.RequestPositionReply(id);
    Dynamixel.ConsoleWriteReceiveData(id);
}
```

上記のサンプルでは，ボーレートが 57600，COM ポートが COM5，サーボの ID が 1，2，3 に設定されていることを前提としているが，これらは適宜書き換える必要がある．工場出荷時に設定されているサーボのボーレートと ID については，Dynamixel のマニュアルを参照されたい．COM ポートは Windows のデバイスマネージャーから確認できる．ボタン 1 を押すと COM ポートが開き，ボタン 2 を押すとサーボのトルクがオンになる．トルクがオンとなったサーボは，軸の角度を手で動かすことができなくなる．また，ボタン 3 では各サーボに位置の返信をリクエストしているが，リクエストを送信した時点では受信データは更新されない点に注意されたい．位置の返信をリクエストしてから数ミリ秒が経過するとサーボから位置データの返信があり，受信データ（ReceiveData）が自動で更新される．なお，送受信するデータの容量が大きい場合（例えば，位置と電流の読み取りをしながら，位置指令値の書き込みをする場合）は，ボーレートを 1000000（1 メガ）bps 以上にすることを推奨する．サーボ側のボーレートは，Dynamixel Wizard というソフトウェアでも変更できる．

以下の関数を用いることで各サーボに位置，速度，電流の指令値を送信することもできる．これらを実行するとサーボが動くため，非常停止ボタンを手元に配備しておくことを推奨する．

- WritePosition(byte[] id, int[] angle)
  引数 1（配列）で指定した ID のサーボに対し，引数 2（配列）の角度指令値を送信する．
- WriteVelocity(byte[] id, int[] velocity)
  引数 1（配列）で指定した ID のサーボに対し，引数 2（配列）の速度指令値を送信する．
- WriteCurrent(byte[] id, int[] current)
  引数 1（配列）で指定した ID のサーボに対し，引数 2（配列）の電流指令値を送信する．

なお，位置指令値は位置制御モード，速度指令値は速度制御モード，電流指令値は電流制御モード時にのみ有効となる．工場出荷時の Dynamixel は位置制御モードとなっており，制御モードを変更する関数も Dynamixel クラスに含まれている（後述する関数リストに記載している）．

複数のサーボと通信をする際は，ID が記述された byte 配列を直接引数に渡した方が処理が速くなる点に注意されたい．以下に「良い例」と「悪い例」を示す．

良い例

```
byte[] id = new byte[3] { 1, 2, 3 };
int[] angle = new int[3] { 1024, 512, 2048 };

void Button1_Click(object sender, RoutedEventArgs e)
{
    Dynamixel.RequestCurrentReply(id);
    WritePosition(id, angle);
}
```

上記の「良い例」では，複数のサーボに対して同時に信号が送信されるため，送受信に要する時間が短い．

悪い例

```
byte[] id = new byte[3] { 1, 2, 3 };
int[] angle = new int[3] { 1024, 512, 2048 };

void Button1_Click(object sender, RoutedEventArgs e)
{
    for(int i = 0; i < id.Length; i++)
    {
        Dynamixel.RequestCurrentReply(id[i]);
        WritePosition(id[i], angle[i]);
    }
}
```

上記の「悪い例」では，サーボ 1 つ 1 つに対して信号が順番に送信される（送受信に要する時間が長い）ため，待機中の信号が送信バッファに蓄積され，不具合（送信バッファの容量不足）の原因となる．また，コードの行数も「良い例」と比べて多くなる．

Dynamixel クラス内の変数や関数の一覧を以下に示す．

Dynamixel クラスに含まれる変数と関数の一覧	
PortOpen(string deviceName)	引数で指定したポートを開く．
PortClose()	ポートを閉じる．
ConsoleWriteReceiveData(params byte[] id)	受信データをコンソールに表示する．
SetBaudRate(int baudRate)	PC 側のボーレートを設定する．
TorqueEnable(params byte[] id)	サーボのトルクをオンにする．
TorqueDisable(params byte[] id)	サーボのトルクをオフにする．
ChangeID(byte id, byte newID)	サーボの id を変更する．
ChangeBaudRate57600(params byte[] id)	サーボのボーレートを 57600 に変更する．
ChangeBaudRate115200(params byte[] id)	サーボのボーレートを 115200 に変更する．
ChangeBaudRate1000000(params byte[] id)	サーボのボーレートを 1000000 に変更する．
ChangeBaudRate2000000(params byte[] id)	サーボのボーレートを 2000000 に変更する．
ChangeBaudRate3000000(params byte[] id)	サーボのボーレートを 3000000 に変更する．
Reboot(byte id)	サーボを再起動する．
FactoryReset(byte id)	サーボを工場出荷状態に初期化．
FactoryResetExceptID(byte id)	サーボを工場出荷状態に初期化（ID 以外）．
ChangePositionControlMode(params byte[] id)	位置制御モードに変更．
ChangeVelocityControlMode(params byte[] id)	速度制御モードに変更．
ChangeCurrentControlMode(params byte[] id)	電流制御モードに変更．
ChangeExtendedPositionControlMode(params byte[] id)	拡張位置制御モードに変更．

RequestPositionReply(params byte[] id)	サーボに位置の返信を要求する.
RequestVelocityReply(params byte[] id)	サーボに速度の返信を要求する.
RequestCurrentReply(params byte[] id)	サーボに電流の返信を要求する.
RequestTemperatureReply(params byte[] id)	サーボに温度の返信を要求する.
int Position(byte id)	サーボから受信した角度.
int[] Position(params byte[] id)	サーボから受信した角度（配列）.
int Velocity(byte id)	サーボから受信した速度.
int[] Velocity(params byte[] id)	サーボから受信した速度（配列）.
int Current(byte id)	サーボから受信した電流.
int[] Current(params byte[] id)	サーボから受信した電流（配列）.
int Temperature(byte id)	サーボから受信した温度.
int[] Temperature(params byte[] id)	サーボから受信した温度（配列）.
WritePosition(byte id, int angle)	サーボに位置指令値を送信する.
WritePosition(byte[] id, int[] angle)	複数のサーボに位置指令値を送信する.
WriteVelocity(byte id, int velocity)	サーボに速度指令値を送信する.
WriteVelocity(byte[] id, int[] velocity)	複数のサーボに速度指令値を送信する.
WriteCurrent(byte id, int current)	サーボに電流指令値を送信する.
WriteCurrent(byte[] id, int[] current)	複数のサーボに電流指令値を送信する.
long TimeSpanMs	連続して通信をする際の時間間隔（ミリ秒）.

連続してサーボとシリアル通信をする場合（例えば，電流の返信をリクエストした直後に角度指令値を送信する場合）は，自動的に TimeSpanMs で設定された時間（初期値は 4[ms]）だけ間隔が空けられるようになっている．この間隔が足りないと通信が不安定になる場合があり，必要な時間間隔はボーレートなどによって異なる（ボーレートが高いほど必要な時間間隔は短くなる）．コンソールにヘッダーやチェックサム（CRC）に関するエラーが頻出する場合は，TimeSpanMs を大きくすることで解決する可能性が高い．

26　力覚センサ（Leptrino 製）との通信

Leptrino 製の力覚センサから値を読み取るサンプルコードを以下に示す．

MainWindow.xaml.cs 内の関数 Button_Click() の周辺を変更

```
SerialDevice.Leptrino Leptrino = new SerialDevice.Leptrino(460800);

void Button1_Click(object sender, RoutedEventArgs e)
{
    Leptrino.PortOpen("COM5");
}

void Button2_Click(object sender, RoutedEventArgs e)
```

```
{
    Leptrino.RequestForceReply();
    Leptrino.ConsoleWriteForce();
}
```

このサンプルでは，力覚センサのボーレートが 460800 であり，COM ポートが COM5 であることを前提としているが，これらは適宜書き換える必要がある．工場出荷時に設定されている力覚センサのボーレートについては，力覚センサのマニュアルを参照されたい．COM ポートは Windows のデバイスマネージャーから確認できる．ボタン 2 では力覚データの返信をリクエストしているが，リクエストを送信した時点では力覚データは更新されない点に注意されたい．また，受信した力覚データは Leptrino.Force に short 配列として格納されている．力覚データはオフセットも含めた整数（short 型）として出力されるため，これをニュートン単位に変換するためには，別途キャリブレーションが必要となる．

Leptrino クラス内の変数や関数の一覧を以下に示す．

Dynamixel クラスに含まれる変数と関数の一覧	
PortOpen(string deviceName)	引数で指定したポートを開く．
PortClose()	ポートを閉じる．
SetBaudRate(int baudRate)	引数で指定したボーレートに設定する．
ContinuousMode()	連続出力モードに変更する．
StopContinuousMode()	連続出力モードを終了させる．
RequestForceReply()	力覚データの返信を要求する．
ConsoleWriteForce()	受信した力覚データをコンソールに表示する．
short[] Force	力覚データが格納された配列．
short Fx	x 軸方向の力．
short Fy	y 軸方向の力．
short Fz	z 軸方向の力．
short Mx	x 軸まわりのモーメント．
short My	y 軸まわりのモーメント．
short Mz	z 軸まわりのモーメント．
RequestForceLimitReply()	力覚データの上限の返信を要求する．
ConsoleWriteForceLimit()	受信した力覚データの上限をコンソールに表示する．
float[] ForceLimit	力覚データの上限が格納された配列．
float FxLimit	x 軸方向の力の上限．
float FyLimit	y 軸方向の力の上限．
float FzLimit	z 軸方向の力の上限．
float MxLimit	x 軸まわりのモーメントの上限．
float MyLimit	y 軸まわりのモーメントの上限．
float MzLimit	z 軸まわりのモーメントの上限．

RequestProductInfoReply()	製品情報の返信を要求する.
ConsoleWriteProductInfo()	受信した製品情報をコンソールに表示する.
string ModelName	力覚センサのモデル名.
string SerialNum	力覚センサのシリアルナンバー.
string FirmVersion	ファームのバージョン.
string OutputRate	出力レート.

27　新しい機能を自作し，公開する方法

ここでは例として，木の生成ができる機能（クラス）を自作し，公開する方法について説明する．まず,「OpenRCF」ファイルを右クリックし,「追加」→「新しい項目」を選択する（図 24）.

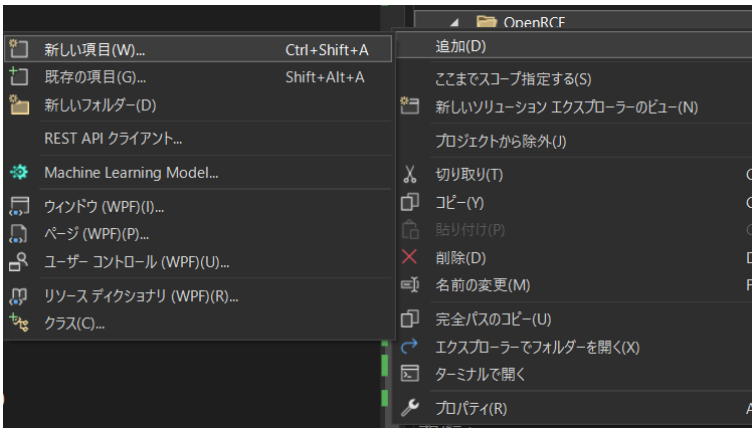


図 24:「新しい項目」の場所

次に「クラス（C#）」を選択し，新機能の内容を表すファイル名（ここでは Tree.cs とする）を付け,「追加」を押す．新しいクラスファイルには，以下のようなソースコードを記述する．

Tree.cs ファイルに記述する基本的なソースコード

```
using System;

namespace OpenRCF
{
    class Tree
    {

    }
}
```

このファイルを生成することで，メインのファイル（MainWindow.xaml.cs）から Tree クラスを利用できるようになる．実際に木を生成できるようにするためには，例えば以下のようなコードを Tree クラスに記述する．

Tree クラスに記述するソースコードの例

```
class Tree
{
    private Pillar Trunk = new Pillar(0.2f, 0.5f);
    private ConeFrustum Leaves = new ConeFrustum(0.4f, 0, 0.75f);

    public Tree()
    {
        Trunk.Color.SetBrown();
        Trunk.Position[2] = 0.5f * Trunk.Height;
        Leaves.Color.SetGreen();
        Leaves.Position[2] = Trunk.Height + 0.5f * Leaves.Height;
    }

    public void Draw()
    {
        Trunk.Draw();
        Leaves.Draw();
    }

    public void SetPosition(float x, float y)
    {
        Trunk.Position[0] = x;
        Trunk.Position[1] = y;
        Leaves.Position[0] = x;
        Leaves.Position[1] = y;
    }
}
```

上記のコードに含まれる public Tree() はコンストラクタ（最初に 1 度だけ実行される関数）である．また，ここでは描画と位置の設定をする機能をカプセル化した例を示している（詳細はオブジェクト指向のカプセル化を参照されたい）．この Tree クラスをメインファイルから利用した例は以下のようになる．

MainWindow.xaml.cs 内の関数 Draw(), Button_Click() の周辺を変更

```
Tree Tree = new Tree();

void Draw()
{
    Tree.Draw();
}

void Button1_Click(object sender, RoutedEventArgs e)
{
    Tree.SetPosition(1, 1);
}
```

上記のサンプルコードを実行し，ボタン 1 を押すと図 25 のようになる．

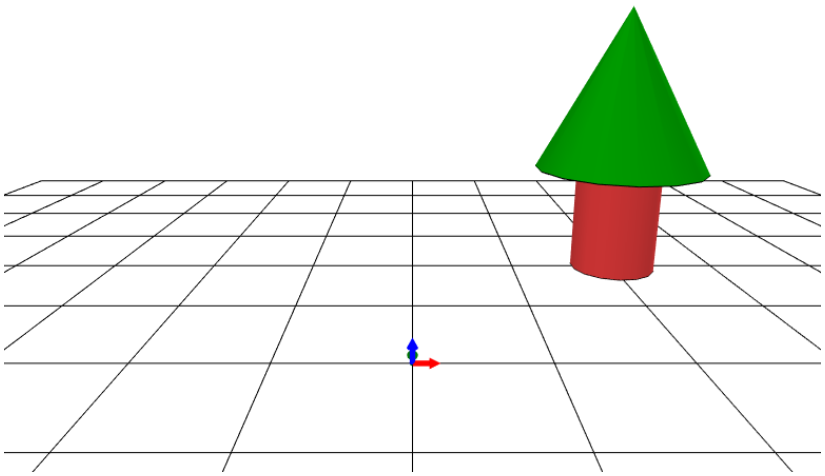


図 25: Tree クラスのサンプルコードを実行したときの結果

この木を生成できるようにする新機能を公開する場合は，Tree.cs ファイルまたは Tree クラスのソースコードをブログや GitHub などに投稿する．理想的には，自作した機能の説明書も同封されていることが望ましい．

28　ボタン等の変更・追加

ユーザーインターフェース（ボタンなど）の設置には，WPF という Visual Studio に元から備わっている機能を用いているため，インターネットで「WPF ボタン」などと検索すると多くの情報が得られる．

新たな UI を追加するための基本的な手順を以下に示す．

1. ソリューションエクスプローラー（右側にある縦長のバー）から「MainWindow.xaml」を開く（もしメインウィンドウ（背景が白のウィンドウ）が表示されない場合は下部にある「デザイン」をクリックする）．
2. 左上にある「表示」の中から「ツールボックス」をクリックする．
3. ツールボックスの中から追加したい UI を選び，メインウィンドウにドラッグ＆ドロップする．
4. Xaml というコードの中に新たな行が自動で追加されているので，その行を目的に応じて変更する．もし Xaml が表示されていない場合は，下部にある「XAML」をクリックする．

上記の手順でボタンを追加した場合は，そのボタンをダブルクリックすることで「MainWindow.xaml.cs」内に自動でボタンクリック関数が追加される．UI を削除する際には，Xaml に追加された行と「MainWindow.xaml.cs」内に自動で追加された関数を削除する．ボタン以外にも，チェックボックスやテキストボックス，スライダーなどの様々な UI を設置することができる．それらの詳細な仕様については，インターネット上に多くの情報が掲載されている．

29　その他の情報

- **コンソールを非表示にする方法**
  「プロジェクト」→「プロパティ」→「出力の種類」→（Windows アプリケーションに変更）

- **実行ファイル（〜.exe）の場所について**
  「bin」フォルダの中に「Debug」または「Release」というフォルダがあり，その中に実行ファイルが生成されている．実行ファイルと同じ階層には，いくつかの dll ファイルも生成されている．初期設定では，これらの dll ファイルもまとめて配布しないと，配布先のコンピュータで実行ファイルを起動することができない．

- **実行ファイルの中に dll ファイルをまとめる方法**
  実行ファイル単体で動作できるようにするためには ILMerge などを用いる．使い方についてはインターネット上に多くの情報が掲載されている．

- **フレームレート（FPS）を変更する方法**
  関数 Core.SetFPS() の引数に設定したい FPS を渡す（最大値は 50）．

30　ライセンス

OpenRCF は MIT ライセンスのもとで公開されています．再配布をする際は，下記の著作権表示と MIT ライセンスの全文を記載する必要があります．

Copyright（c）2022-2024 Sekiguchi Masanori
OpenRCF is released under the MIT license

31　著作権表示

OpenRCF に含まれるプログラムの一部は OpenTK を使用しています．

Copyright（c）2006-2022 Stefanos Apostolopoulos
OpenTK is released under the MIT license
`https://opensource.org/licenses/mit-license.php`