第4章

鉄道模型の自動制御

「鉄道模型シミュレーター NX」では、「ギミック」の制御に汎用スクリプト言語の「Python」を新たに採用しています。
これにより、複雑な制御を、今まで以上の柔軟さで記述することができます。

コンセプト車両「夢空間」　オシ25 901

4-1 Pythonスクリプトをはじめよう

「VRM-NX」の自動制御と、実装されているスクリプト言語「Python」を学んでみましょう。

■「VRM-NX」と「Python」

従来の「VRMシリーズ」(「VRM4」以降)では、各種ギミック制御に独自の「スクリプト・コマンド」を採用していました。

しかし、最新バージョンである「鉄道模型シミュレーターNX」では、「ギミック」の制御に汎用スクリプト言語の「Python」を新たに採用しました。

これにより、複雑な制御を今まで以上の柔軟さで記述することができます。

また、「Python」の「実行環境」は「ゲーム」に内包されているため、「ゲーム」を「インストール」するだけで「プログラミング環境」を整えることができます。

■ 無料のスターターキット

「VRM-NX」は店頭で販売されている「**パッケージ版**」と、オンライン・サービスとして提供されている「**ONLINE版**」があります。

それらの「お試し版」として、無料で遊べる「**スターターキット**」が「公式サイト」からダウンロードできます。

扱えるパーツは最低限のものになりますが、「Python言語」での制御は製品と同等の実装になるので、仕様を理解できれば、これだけで「列車の自動運転」なども可能です。

*

「仮想空間で動く鉄道」を見ながら「Pythonプログラムを楽しく学習してみましょう。

■ レイアウトの作成

「VRM-NX」が起動したら、まずは「新規レイアウト」を作ります。

「ファイル」→「新規レイアウトの作成」を選択すると、黒一色だった画面にグリッドが表示されて、パーツが置けるようになります。

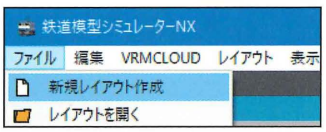

図4-1-1　新規レイアウト作成

■ Pythonで「Hello, World!」

さっそく、「Python」に触れてみましょう。

レイアウター・ツールを選択して「レイアウト」→「スクリプト」→「レイアウト・スクリプト・エディタ」を押すと、スクリプト編集画面が開きます。

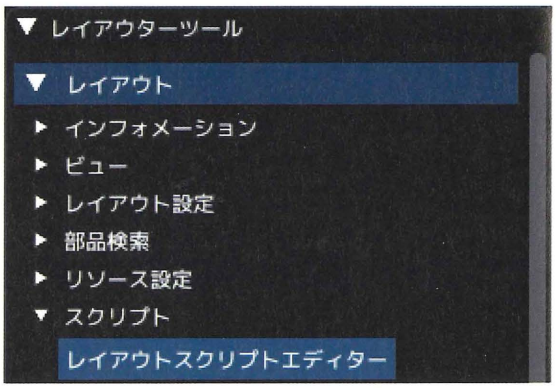

図4-1-2　レイアウト・スクリプト・エディタ

「VRM-NX」ではスクリプト操作可能なオブジェクトには、パーツ配置時に、「デフォルト・スクリプト」がセットされています。

*

リスト4-1-1は、レイアウトの「デフォルト・スクリプト」です。

リスト4-1-1　デフォルト・スクリプト

```
#LAYOUT
import vrmapi
def vrmevent(obj,ev,param):
    if ev == 'init':
        dummy = 1
    elif ev == 'broadcast':
        dummy = 1
    elif ev == 'timer':
        dummy = 1
    elif ev == 'time':
        dummy = 1
    elif ev == 'after':
        dummy = 1
    elif ev == 'frame':
        dummy = 1
    elif ev == 'keydown':
        dummy = 1
```

*

「Python」で「VRM-NX」を操作するときは「vrmapi」で定義されているの命令を使います。

たとえば、ビュワーの「スクリプトLOG」に「文字」を表示する命令は、「vrmapi.LOG」(string)になります。

「string」の部分に「ダブル・クオーテーション」で囲んだ「文字列」を入力することで、文字が表示されます。

*

今回は「ビュワーの起動時」に「スクリプトLOG」へ「Hello, World!」という文字を出力してみましょう。

[手順]

[1]「デフォルト・スクリプト」で「ビュワー起動時」に実行される部分は、「if ev=='init':」内の範囲です。

つまり、ビュワーの起動時に「Hello, World!」と表示させる場合は該当部分を、

```
if ev== 'init':
    vrmapi.LOG("Hello, World!")
```

と書き換えます。

[2]元からある「dummy=1」は構文を保持するためのダミーコマンドなので、「if-else」内に1行以上の命令を書いた場合は「dummy=1」を消すか、頭に「#」を付けてコメントアウトします。

図4-1-3　デフォルトコード編集後

[3]編集が完了したら、左側のツールボックスにある「運転」を押してビュワーで確認してみます。

右上の青い四角マークを押して「ビュワーウィンドウ」を表示させ、「ビュワー」→「情報」→

「ScriptLOG」→「スクリプトLOGを開く」を押します。

「スクリプトLOGウインドウ」が表示され、3行目に「string : Hello, World!」を表示させることができました。

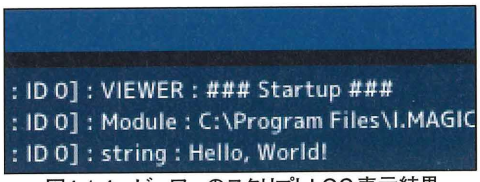

図4-1-4　ビュワーのスクリプトLOG表示結果

[4]ビュワーを終了するときは右上の赤い三角をクリックするか、「Esc」キーを押します。

[5]動作を確認したら「ファイル」→「名前をつけて保存」を使ってレイアウトファイルを保存します。

※「VRM-NX」では、「Python」を使って柔軟な制御ができる代わりに、構文エラーや例外発生時にソフトウェアが異常終了する場合があります。
　そのため、レイアウト・ファイルはこまめに保存することをお勧めします。*

このように「VRM-NX」を使えば、専用の開発環境がなくても「Python」を使ってプログラミングしたり、遊んだりできます。

*

「VRM-NX」では、(a)「Python」を使って列車の速度を変更したり、(b)センサで列車の位置を検出したり、(c)ポイントや信号を切り替えたりできます。

これらを複合的に構成することで、列車の自動運転や信号機とポイントを組み合わせたATCなどの高度な自動制御も可能となります。

4-2 イベント・ハンドラー

前節で、デフォルトの「レイアウト・スクリプト」に手を入れて、ビュワー起動時の「スクリプトLOG」に、「Hello, World!」を表示させました。
　この節では、これらの動作について、「イベント・ハンドラー」の視点から、もう少し詳しく紹介します。

図4-2-1　Pythonで自動運転

リスト4-2-1　「Hello, World!」表示スクリプト

```
import vrmapi
def vrmevent(obj,ev,param):
    if ev== 'init':
        vrmapi.LOG("Hello, World!")
    elif ev == 'broadcast':
        dummy = 1
    elif ev == 'timer':
        dummy = 1
    elif ev == 'time':
        dummy = 1
    elif ev == 'after':
        dummy = 1
    elif ev == 'frame':
        dummy = 1
    elif ev == 'keydown':
        dummy = 1
```

■ 引数の「obj」には何が入っている？

デフォルトで生成される関数「vrmevent」の「obj引数」には、「VRM-NX」の部品に対応する「オブジェクト・データ」が入っています。

これには、他のイベントから呼び出される場合を除いて、基本的には「自分自身のオブジェクト」が入ります。

以下は「データ型」の一例です。

表4-1　データ型

自動センサ	VRMATS
音源	VRMBell
地上カメラ	VRMCamera
車輌	VRMCar
踏切、ホームドア	VRMCrossing
エミッター	VRMEmitter
レイアウト	VRMLayout
モーションパス	VRMMotionPath
ポイント	VRMPoint
信号	VRMSignal
スカイドーム、天候	VRMSky
スプライト	VRMSprite
システム	VRMSystem
編成	VRMTrain
ターンテーブル	VRMTurntable

「VRM-NX」では、これらの「オブジェクト・データ」に対して命令を実行することで、「列車」や「ポイント」を操作します。

後述する「イベント登録」もこの「オブジェクト・データ」を使います。

無料の「スターターキット」には「信号」や「ターンテーブル」などの一部パーツが入っていませんが、収録されているパッケージを購入することで、利用できるようになります。

図4-2-2　スクリプトで操作できる「ポイント」や「信号機」

■ 引数の「ev」には何が入っている？

2つ目の「ev引数」には、イベント実行時にどのイベントが「トリガー」になったのかを判別するための「識別用　文字列」が入ります。

この「ev引数」の文字列を「if-else」で「条件分岐」することで、「イベント・トリガー」ごとに異なる処理を記述します。

■ 引数の「param」には何が入っている？

3つ目の「param引数」には、処理に利用するパラメータが、辞書(dict)型で格納されています。

同じ「オブジェクト」でも、異なるイベントを処理すると、「値」はもちろんのこと、「キー項目」も変化します。

*

「存在しないデータ」を参照しようとすると「エラー」が発生するので、利用には注意が必要です。

中のデータを安全に確認したい場合は、「キー」と「値」を「スクリプトLOG」に出力する関数を作って、どのタイミングからでも使えるようにします。

リスト4-2-2　Dict型表示関数

```
# dict型のキーと値を表示
def showDict(param):
    # 個数の確認
    str_len = str(len(param))
    vrmapi.LOG("  param[" + str_len + ']')
    # キー、値の表示
    for k, v in param.items():
        vrmapi.LOG("   " +str(k)+" : "+str(v))
```

■「vrmevent関数」の登録と実行

「レイアウト・スクリプト」の「vrmevent関数」は、デフォルトで、ビュワー起動時に一度だけ「init」が呼び出されるようになっています。

それ以外のイベントは「SetEvent関数」を実行することで、イベントが登録されて実行されるようになります。

*

デフォルトで登録されているイベントと「各オブジェクト」が登録可能なイベントは、「オブジェクト」の種類によって異なります。

たとえば、レイアウト自身には、次のイベントを登録することができます。

●SetEventTime

「SetEventTime」はビュワー開始から指定時間後に発生するイベント。

「ev引数」の文字列は「time」。

●SetEventTimer

SetEventTimerは指定間隔で繰り返し発生するイベント。

「ev引数」の文字列は「timer」。

●SetEventAfter

「SetEventAfter」は実行時点から指定時間後に発生するイベント。

「ev引数」の文字列は「after」。

●SetEventFrame

「SetEventFrame」は画面描画の1フレームごとに発生するイベント。

「ev引数」の文字列は「frame」。

●SetEventKeyDown

「SetEventKeyDown」はキーボードのA～Zとテンキーの0～9キーが押されたときに発生するイベント。

登録する英字は大文字ですが、"「Shift」押しなし"で動きます。

「キー」は「param引数」の「keycode」から取得できます。

「ev引数」の文字列は「keydown」です。

*

基本的には「init」部分に「SetEvent関数」を登録します。

イベントを組み合わせることで特定のイベント後に有効にしたり、「キー」を押してから時間差で実行したりもできます。

*

以上を踏まえて、以下のサンプル・スクリプトを実行してみましょう。

リスト4-2-3　イベント・サンプル・スクリプト

```
#LAYOUT
import vrmapi

def vrmevent(obj,ev,param):
    if ev == 'init':
        # ビュワー起動時に一度だけ呼び出し
        vrmapi.LOG(ev + " : Hello, World!")
        # 指定キーを入力すると発生
        obj.SetEventKeyDown('A')
        # 指定間隔で繰り返し発生
        obj.SetEventTimer(3.0)
        # ビュワー開始から指定時間後に発生
        obj.SetEventTime(2.0)
```

```
        # フレームごとに発生
        #obj.SetEventFrame()
        # param確認
        showDict(param)
    elif ev == 'broadcast':
        vrmapi.LOG(ev)
    elif ev == 'timer':
        vrmapi.LOG(ev + "：3秒ごと")
        # param確認
        showDict(param)
    elif ev == 'time':
        vrmapi.LOG(ev + "：2秒後")
        # param確認
        showDict(param)
        # 登録から指定時間後に発生
        obj.SetEventAfter(2.0)
    elif ev == 'after':
        vrmapi.LOG(ev + "：登録から2秒後")
        # param確認
        showDict(param)
    elif ev == 'frame':
        # 表示が無限に流れるので省略
        #vrmapi.LOG(ev + "：1フレームごと")
        dummy = 1
    elif ev == 'keydown':
        str_key = param['keycode']
        vrmapi.LOG(ev + "：" + str_key)
        # param確認
        showDict(param)

# dict型のキーと値を表示
def showDict(param):
    # 個数の確認
    str_len = str(len(param))
    vrmapi.LOG("  param[" + str_len + ']')
    # キー、値の表示
    for k, v in param.items():
        vrmapi.LOG("    " +str(k)+"："+str(v))
```

それぞれのイベント実行時に「param引数」の配列を表示します。

*

実行結果は、次のようになります。

リスト4-2-4 「イベント・サンプル」実行結果

```
### Startup ###
init：Hello, World!
  param[2]
   eventid：708
   eventtime：0.0
keydown：A
  param[3]
   eventid：709
   eventtime：1.4412082999988343
   keycode：A
time：2秒後
  param[3]
   eventid：711
   eventtime：2.0040875999984564
   time：2.0
timer：3秒ごと
  param[3]
   eventid：710
   eventtime：3.004905500005407
   time：3.0
after：登録から2秒後
  param[3]
   eventid：741
   eventtime：4.005916000001889
   time：2.0
timer：3秒ごと
  param[3]
   eventid：710
   eventtime：6.008160100005625
   time：3.0
```

1段落目は「イベント文字列＋文字」、**2段落目**は「パラメータの個数」、**3段落目**に「パラメータ

のキーと値」を表示しています。

イベントは共通して「eventid」と「eventtime」が格納されていますが、「time」「timer」「after」イベントには「time」の値が格納され、「keydown」イベントには「keycode」が、独自に格納されることが分かりました。

図4-2-2　VRM-NXの車両

4-3　「VRM-NX」で列車の自動運転

「VRM-NX」で、列車の「発車」と「停車」を「自動センサ」パーツと「Python」を使って自動で行なう方法を紹介します。

*

線路に重ねて配置した「自動センサ」を列車が通ると減速して停車し、一定時間後に発車するというプログラムを繰り返し実行します。

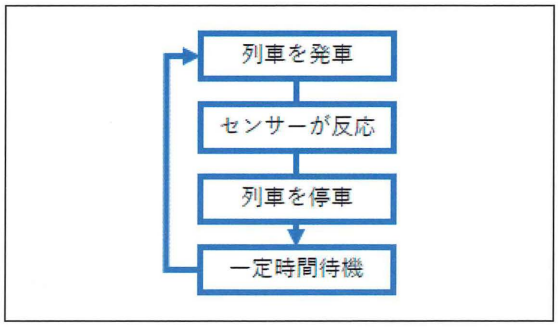

図4-3-1　自動運転のプログラム

■ レイアウトの準備

まずは、列車を動かすレイアウトを準備します。

*

「直線」と「曲線」を組み合わせた「環状線」（オーバル・レイアウト）を作ります。

上側に停車位置として駅ホームを配置し、駅ホームの左側、時計回りの進行方向手前に「自動センサ」を配置します。

列車は、自動運転させるための1編成を、時計回りになるように配置します。

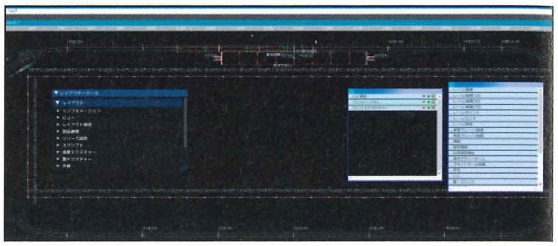

図4-3-2　オーバル・レイアウト

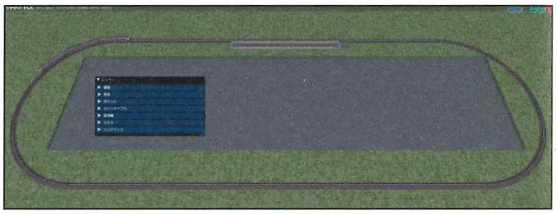

図4-3-3　ビュワー表示

■ 起動時に列車を発車させる

列車をビュワー起動時から走行させるようにするには「AutoSpeedCTRL」関数を使います。

「AutoSpeedCTRL」関数を使うと、指定した距離を走りながら速度を変化させます。

「第一引数」にはmm（ミリメートル）換算の「走行距離」、「第二引数」には「編成」に設定した「最高速度に対する速度比」を、0～1の浮動小数で指定します。

「AutoSpeedCTRL関数」の使用例

```
# 距離600mmで最高速度の75%に変更
VRMTrain.AutoSpeedCTRL(600.0, 0.75)
```

■ センサを反応させる

線路に重ねて配置した「自動センサ」パーツ(以下：センサ)は列車を検知すると「catchイベント」を発行します。

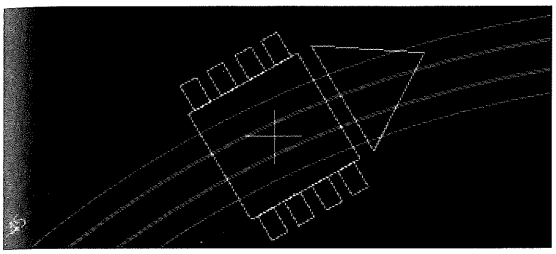

図4-3-4 「線路」に重ねた「センサ」

検知した列車はイベントの「obj引数」に対して「GetTrain」関数を使って「編成オブジェクト」を取得することができます。

リスト4-3-1 「自動センサ」での「編成オブジェクト」取得例

```
elif ev == 'catch':
    # 編成オブジェクトを取得
    tr = obj.GetTrain()
    # センサ検知ログ
    tn = tr.GetNAME()
    sn = obj.GetNAME()
    vrmapi.LOG(sn + "が" + tn + "検知")
```

■ 列車を「停車」させる

列車の「停車」は「発車」と同様の「AutoSpeedCTRL」をセンサの「catchイベント」と組み合わせて使います。

停車位置を駅ホームに合わせたいときは、センサの位置を調節するか、「AutoSpeedCTRL」関数の走行距離を変更します。

■ 列車を「待機・再出発」させる

列車を「停車」させてから一定時間経過後に発車させるには、センサの「catchイベント」で「SetEventAfter」関数を使います。

列車の「afterイベント」に出発用の「AutoSpeedCTRL」関数を設定することで、「SetEventAfter」関数の引数で指定した時間の経過後に再出発します。

時間指定は「列車が停車してからの待機時間」ではなく、「センサが反応してからの経過時間」になります。

そのため、停車するまでの「減速時間」を含めた時間を設定します。

■「Python」を書き込む

以上の内容を実現するためのスクリプトを、「列車」と「センサ」にそれぞれ記載します。

リスト4-3-2 列車用スクリプト

```
if ev == 'init':
    # 距離600mmで最高速度の75%に変更
    obj.AutoSpeedCTRL(600.0, 0.75)
elif ev == 'broadcast':
    dummy = 1
elif ev == 'timer':
    dummy = 1
elif ev == 'time':
    dummy = 1
elif ev == 'after':
    # 距離600mmで最高速度の75%に変更
    obj.AutoSpeedCTRL(600.0, 0.75)
```

リスト4-3-3　センサ用スクリプト(抜粋)

```
elif ev == 'catch':
    # 編成オブジェクトを取得
    tr = obj.GetTrain()
    # 距離1100mmで速度を0
    tr.AutoSpeedCTRL(1100, 0.0)
    # 20秒後にafterイベント実行
    tr.SetEventAfter(20.0)
```

記述できたら、レイアウトを保存して、「運転ボタン」を押してみましょう。

*

ビュワー起動後に列車が走り出し、レイアウトを回ります。

センサを踏むと減速し、ホームへ停車します。

一定時間停車後にまた出発して、以降は同じ動作を繰り返します。

■「列車」や「駅」を増やしてみる

この自動運転方法は「センサ」と「列車」が連携していますが、それぞれの仕組みは単体で完結しています。

そのため、複製して増やすことができます。

「駅ホーム」「列車」「センサ」をレイアウトの点対称となるように複製してみましょう。

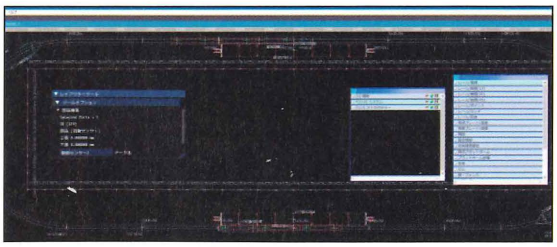

図4-3-5　「駅」と「列車」と「センサ」を追加

ビュワーを起動すると、2つの列車が同じように走ります。

2編成の速度が同じで充分な距離が空いていれば、永久に走り続けます。

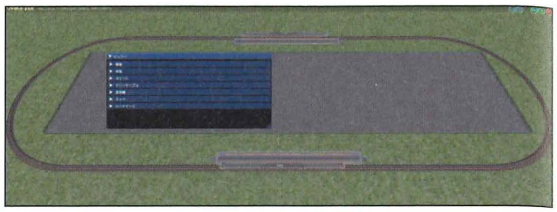

図4-3-6　ビュワー表示

このように1つの「オーバル・レイアウト」内に同一行動の列車や停車位置を増やすことは簡単にできます。

1つの「オーバル・レイアウト」に複数の列車を配置して駅の停車を行なう場合は、駅停車中に後続の列車がぶつからないような間隔を空けて配置してください。

列車の密度を上げるには、実際の「鉄道運行計画」同様に、列車の速度を上げたり停車時間を短くしたりして調節しましょう。

■ 車両の電源をONにする

「VRM-NX」では列車の「ヘッドライト」や「ルームライト」などのON/OFF制御が可能です。

車両の各種電源ステータスは「編成オブジェクト」(VRMTrain)の中にある「車両オブジェクト」(VRMCar)に対して車両ごとに変更命令を記述する必要があります。

しかし、「レイアウト・オブジェクト」の「GetTrainList」関数を利用することで、レイアウト内の全編成・全車両をまとめて属性変更することができます。

*

以下の内容を「レイアウト・オブジェクト」に記載すると、「ダミー編成以外の列車」の電灯やパンタグラフを「ON」にします。

リスト4-3-4 全列車の電源をONにする

```
#LAYOUT
import vrmapi

def vrmevent(obj,ev,param):
    if ev == 'init':
        dummy = 1
        setAllCarPower()
    elif ev == 'broadcast':
        dummy = 1
    elif ev == 'timer':
        dummy = 1
    elif ev == 'time':
        dummy = 1
    elif ev == 'after':
        dummy = 1
    elif ev == 'frame':
        dummy = 1
    elif ev == 'keydown':
        dummy = 1

# レイアウト内の全車両を電源ON
def setAllCarPower():
    # 編成リストを新規編成リストに格納
    trList = vrmapi.LAYOUT().GetTrainList()
    # 新規編成リストから編成を繰り返し取得
    for tr in trList:
        # ダミー編成は強制無効
        if tr.GetDummyMode():
            setTrainSwitch(tr, False)
        else:
            setTrainSwitch(tr, True)

# 編成の車両電灯系を操作
def setTrainSwitch(tr, sw):
    # 車両数を取得
    len = tr.GetNumberOfCars()
    # 車両ごとに処理
    for i in range(0, len):
        # 車両を取得
        car = tr.GetCar(i)
        # 方向幕灯
        car.SetRollsignLight(sw)
        # 室内灯
        car.SetRoomlight(sw)
        # LED
        car.SetLEDLight(sw)
        # 運転台室内灯
        car.SetCabLight(sw)
        # パンダグラフ個数確認
        k = car.GetCountOfPantograph()
        for j in range(0, k):
            # パンタグラフ昇降
            car.SetPantograph(j,sw)
        # 先頭車両処理
        if i == 0:
            # ヘッドライト
            car.SetHeadlight(sw)
        # 最後尾車両処理
        if i == len - 1:
            # テールライト
            car.SetTaillight(sw)
        # 蒸気機関車用(テンダーも対象)
        if car.GetCarType() == 1:
            # 煙
            car.SetSmoke(sw)
```

図4-3-7「電源ON」(左)と「電源OFF」(右)

*

今回はセンサで「発車」と「停車」を繰り返す基本的な自動運転システムを作りました。

4-4 追い越しのある列車の自動運転

前節は、「VRM-NX」で、駅に一定時間停車する列車の自動運転を紹介しました。

今回は、編成を追加して、「特急列車」と「普通列車」による、追い越しのある自動運転を紹介します。

■ レイアウトの準備

レイアウトは、前回の駅が上下にある環状線（オーバル・レイアウト）を改造して、上側にポイントの待避線を追加します。

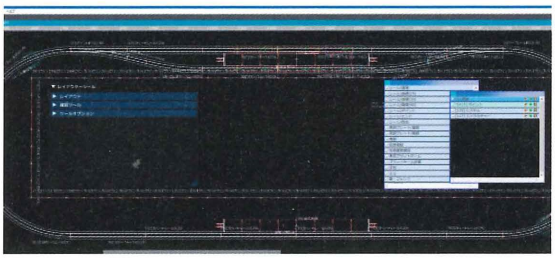

図4-4-1　待避線付きオーバル・レイアウト

「自動センサ」パーツ（以下、「センサ」）は、レイアウト左上と右下に、それぞれ、「上駅」「下駅」と名前を付けます。

分岐ポイントは「分岐A」「分岐B」の名前にします。

列車は、「特急列車」と「普通列車」の2編成を用意します。

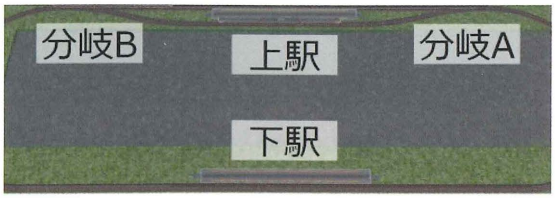

図4-4-2　ビュワー表示

■ パターン・ダイヤ

今回は、「特急列車」が「普通列車」を駅で追い越す、「緩急結合ダイヤ」を実現します。

周期的に繰り返すため、「パターン・ダイヤ」の一種になります。

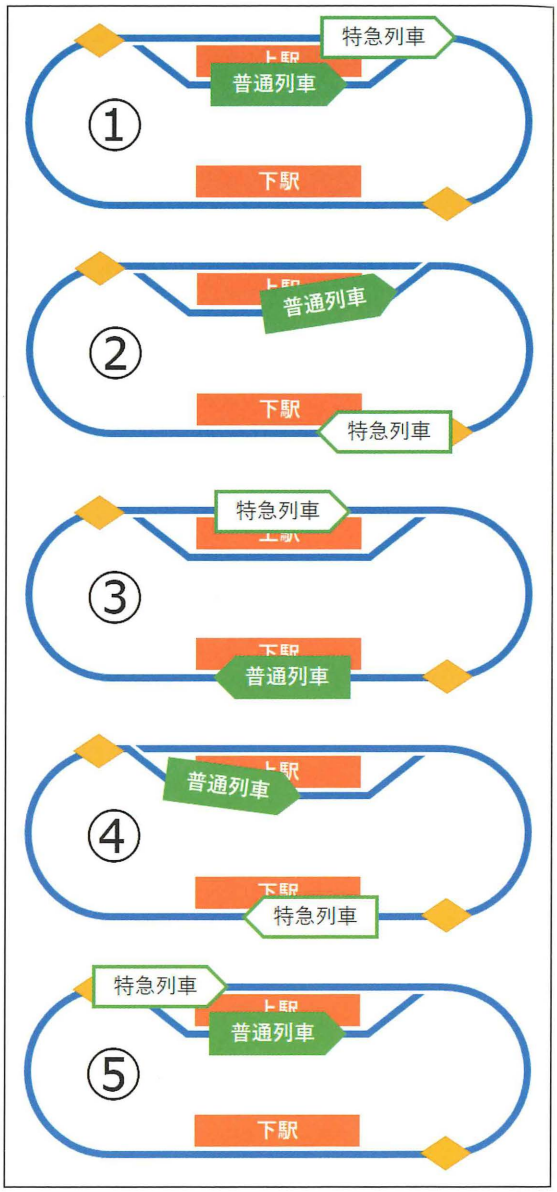

図4-4-3　パターン・ダイヤ

[1]「特急列車」が「上駅」を先発。
[2]「特急列車」が「下駅センサ」を踏んだら「分岐A」を待避線側に切り替えて「普通列車」が発車。
「特急列車」は「下駅」を通過。
[3]「特急列車」が「上駅」に、「普通列車」は「下駅」に一定時間停車。
[4]「普通列車」は「上駅センサ」を踏むと、「分岐B」を待避線に切り替えて、「上駅」に停車。
[5]「特急列車」が「上駅センサ」を踏むと、「分岐B」「分岐A」のポイントを本線側に切り替えて、上駅に一定時間停車。

*

以降は、[1]に戻って繰り返す。

■ スクリプト

関連するそれぞれの「スクリプト」を掲載します。

*

先頭行の「def vrmevent_xxx」は、レイアウト内の「パーツID」を示しています。

スクリプトをコピーするときは、「パーツID」と「スクリプト内のID」が一致しているか、確認してください。

●特急列車の「スクリプト」

特急列車の「スクリプト」には、「ビュワー起動時」と、駅停車後の「タイマ・イベント」で呼ばれる、「出発用速度制御」の「スクリプト」を記載します。

リスト4-4-1　特急列車のスクリプト

```
def vrmevent_3(obj,ev,param):
    if ev == 'init':
        vrmapi.LOG(obj.GetNAME() + " 出発")
        obj.AutoSpeedCTRL(600.0, 0.75)
    elif ev == 'after':
        vrmapi.LOG(obj.GetNAME() + " 出発")
        obj.AutoSpeedCTRL(600.0, 0.75)
```

●普通列車の「スクリプト」

普通列車の「スクリプト」は、駅停車後の「タイマ・イベント」で呼ばれる、「出発用速度制御」の「スクリプト」のみを記載します。

②のときとは別の「センサ・イベント」が、直接、車両を出発させます。

リスト4-4-2　普通列車のスクリプト

```
def vrmevent_42(obj,ev,param):
    if ev == 'init':
        dummy = 1
    elif ev == 'after':
        vrmapi.LOG(obj.GetNAME() + " 出発")
        obj.AutoSpeedCTRL(600.0, 0.75)
```

●「下駅」センサのスクリプト

「下駅」センサは、「特急列車」なら「下駅」を「通過」し、「普通列車」なら下駅に「停車」します。

「特急列車」と「普通列車」の識別は、センサの「GetTrain関数」で取得した列車オブジェクトの「GetNAME関数を使」い、「名前が「特急列車」かどうか」の「if条件」で判断します。

また、「特急列車」の場合は、「execStart関数」で「普通列車」に対する「出発判定」を実施します。

リスト4-4-3　「下駅」センサのスクリプト

```
#OBJID=149
import vrmapi
def vrmevent_149(obj,ev,param):
    if ev == 'init':
        dummy = 1
    elif ev == 'catch':
        # オブジェクト定義
        tr = obj.GetTrain()
        trn = tr.GetNAME()
        obn = obj.GetNAME()
```

```
        # 特急列車か
        if trn == '特急列車':
            # 特急列車通過処理
            vrmapi.LOG(trn+ " "+obn+"通過")
            l = vrmapi.LAYOUT()
            # 列車[42]のセンサ[118]停車判定
            execStart(l.GetTrain(42), 118)
        else:
            # 普通列車停車処理
            vrmapi.LOG(trn+" "+obn+"停車")
            tr.AutoSpeedCTRL(1050, 0.0)
            # 15秒後に出発
            tr.SetEventAfter(15.0)
```

●「上駅」センサのスクリプト

「上駅」センサは「列車名」を識別して「上駅のポイント切り替え」を行ないます。

「普通列車」は「分岐B」を「待避線」に切り替えて停車。

「特急列車」は「分岐A・B」両方を切り替えて、「停車」と「出発タイマー」を定義します。

リストリスト4-4-4　「上駅」センサのスクリプト

```
def vrmevent_118(obj,ev,param):
    if ev == 'init':
        dummy = 1
    elif ev == 'catch':
        # オブジェクト定義
        tr = obj.GetTrain()
        trn = tr.GetNAME()
        obn = obj.GetNAME()
        # ポイント定義
        l = vrmapi.LAYOUT()
        p1 = l.GetPoint(51)
        p1n = p1.GetNAME()
        p2 = l.GetPoint(37)
        p2n = p2.GetNAME()
        # ATS登録
        setATS(obj,tr)
```

```
        # 特急列車か
        if tr.GetNAME() == '特急列車':
            # 特急列車分岐切替
            p1.SetBranch(0)
            p1b = str(p1.GetBranch())
            vrmapi.LOG(p1n+"["+p1b+"]")
            p2.SetBranch(0)
            p2b = str(p2.GetBranch())
            vrmapi.LOG(p2n+"["+p2b+"]")
            # 特急列車停車処理
            vrmapi.LOG(trn+" "+obn+"停車")
            tr.AutoSpeedCTRL(1100, 0.0)
            # 18秒後に発車
            tr.SetEventAfter(18.0)
        else:
            # 普通列車分岐切替
            p1.SetBranch(1)
            p1b = str(p1.GetBranch())
            vrmapi.LOG(p1n+"["+p1b+"]")
            # 普通列車停車処理
            vrmapi.LOG(trn+" "+obn+"停車")
            tr.AutoSpeedCTRL(1100, 0.0)
```

●レイアウトの「スクリプト」

「レイアウト・スクリプト」は、「起動時」に「普通列車」に「センサ情報」を登録します。

「setATS関数」は、列車オブジェクトがもつ連想配列に"section"文字列をキーとして、「上駅センサ・オブジェクト」を格納します。

これは「execStart関数」の実行時に、「上駅」に普通列車が存在しているかどうかを、引数の「パーツID」と「上駅センサ」の「GetID関数」での「値比較」で確認します。

さらに、「完全停車」後に出発させるため、「センサ確認」後に速度が「0」かどうかも確認しています。

*

「出発処理」が完了したら、「連想配列」を「del」で削除します。

リストリスト4-4-5 レイアウトのスクリプト

```
def vrmevent(obj,ev,param):
    if ev == 'init':
        # 起動時初期登録
        sn = obj.GetATS(118)
        tr = obj.GetTrain(42)
        setATS(sn, tr)

# 列車の連想配列にセンサobjを格納
def setATS(sn,tr):
    k = "section"
    d = tr.GetDict()
    d[k] = sn
    trn = tr.GetNAME()
    snn = d[k].GetNAME()

# 普通列車出発判定
def execStart(tr,snID):
    k = "section"
    d = tr.GetDict()
    trn = tr.GetNAME()
    # 配列[section]が有り対象センサか
    if('section' in d) and
(d[k].GetID()==snID):
        # 列車が停車しているか
        if tr.GetSpeed() == 0.0:
            # 分岐切替
            l = vrmapi.LAYOUT()
            p2 = l.GetPoint(37)
            p2.SetBranch(1)
            p2n = p2.GetNAME()
            p2b = str(p2.GetBranch())
            obn = d[k].GetNAME()
            vrmapi.LOG(p2n+"["+p2b + "]")
            # 出発
            vrmapi.LOG(trn+" "+obn+"出発")
            tr.AutoSpeedCTRL(600.0, 0.75)
            # センサ情報削除
            del d[k]
        else:
            vrmapi.LOG(trn + " 未停車")
    else:
        vrmapi.LOG(trn + " 対象section外")
```

■実行結果

「ビュワー」起動時のスクリプト「LOGウインドウ」のログは以下になります。

*

センサ名を「上駅」「下駅」にしたことで、「センサ・イベント時の動作が分かりやすくなります。

リストリスト4-4-6 「スクリプトLOG」ウインドウ

```
特急列車 出発
特急列車 下駅通過
分岐A[1]
普通列車 上駅出発
普通列車 下駅停車
分岐B[0]
分岐A[0]
特急列車 上駅停車
普通列車 出発
特急列車 出発
分岐B[1]
普通列車 上駅停車
特急列車 下駅通過
普通列車 未停車
分岐B[0]
分岐A[0]
特急列車 上駅停車
特急列車 出発
(繰り返し)
```

■「鉄道模型シミュレーター」ならでは

今回は、「特急列車」と「普通列車」を使った「追い越しのある自動運転」を作りました。

*

このような、1つの本線に速度の異なる列車を走らせることは、現実の鉄道模型では実現難易度が高く、「鉄道模型シミュレーター」ならではの魅力の一つと言えるでしょう。

4-5 閉塞と自動列車停止装置の構築

「閉塞」と「自動列車停止（Automatic Train Stop：以下、「ATS」）システム」を構築します。

■「VRM-NX」によるATSの構築

レイアウトは「閉塞」として利用する「センサ間隔」と「設置数」をコンパクトな面積に纏（まと）めるため、「立体交差」を用いた「周回レイアウト」を作成します。

*

列車は、①1駅のみ停車する「特急列車」1編成と、②2駅に停車して「特急列車」の発車後に出発する「普通列車」2編成を、用意します。

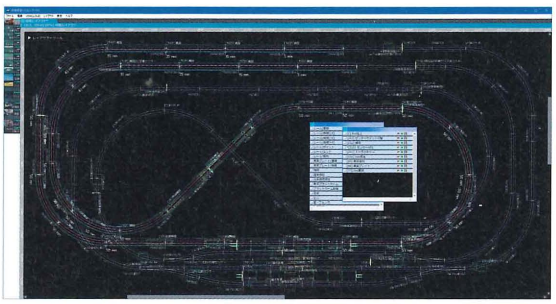

図4-5-1　立体交差レイアウト

図4-5-2　レイアウト・ビュワー

■ ATSの動作

構築するATSは、以下のように動作します。

[手順]

[1]「ビュワー」起動時に（a）列車の「ATSセンサ順路」と「初期位置情報」を設定します。

[2]列車が走行中に「ATSセンサ」を踏むと、「列車の位置情報」を「ATSセンサ順路」の「次位置」へ移動します。

[3]移動後の「ATSセンサ」の「さらに次のセンサ」を参照して、「前方列車の有無」を確認します。

[4]「前方列車」を確認したら列車を停車させ、「前方列車」の有無をタイマーで監視します。

[5]「前方列車」が不在になれば、監視を終了して列車を発車します。

上記の動作により、列車は「ATSセンサ」1つにつき1列車しか進入しないため、「閉塞」が成立します。

●Pythonスクリプト

それぞれのパーツに記述する「Pythonスクリプト」を掲載します。

*

先頭行の「def vrmevent_xxx」はレイアウト内の「パーツID」を示しています。

スクリプトをコピーするときは、レイアウトとスクリプト内の「パーツID」が一致しているか、確認してください。

*

なお、ATS以外に自動運転用のコードも内包しています。

●「特急列車」のスクリプト

「特急列車」のスクリプトには「ビュワー」起動時に、（a）「ATS情報の登録」を行ない、（b）「afterイベント」で「ATSの監視」を記述します。

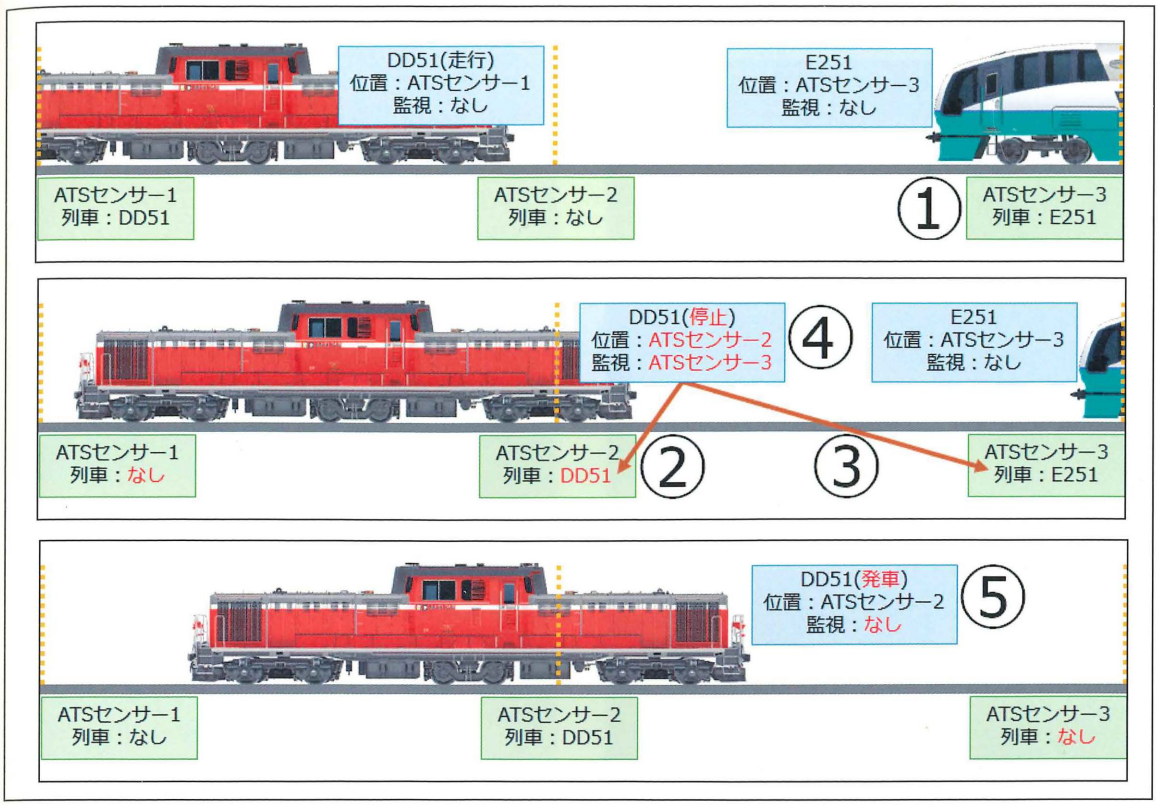

図4-5-3　ATSの動作

リスト4-5-1　「特急列車」のスクリプト

```
def vrmevent_162(obj,ev,param):
    if ev == 'init':
        # ATS順路登録
        d = obj.GetDict()
        d["route"] = [221, 224, 225, 253,
255, 227, 228, 257, 229, 259, 230, 261, 231]
        # ATS順路位置登録
        d["route_now"] = 0
        i = d["route"][d["route_now"]]
        # ATS列車登録
        s = vrmapi.LAYOUT().GetATS(i)
        ats_d = s.GetDict()
        ats_d["train"] = obj
        # 初期自動発車
        obj.AutoSpeedCTRL(200,1.0)
    elif ev == 'after':
        d = obj.GetDict()
        if("next_ats" in d):
            vrmapi.LOG("ATS監視開始")
            chkNextATS(obj, d["next_ats"],
param["eventid"])
        elif("go_auto" in d):
            # 発車
            obj.AutoSpeedCTRL(200,1.0)
            del d["go_auto"]
        else:
            vrmapi.LOG("ATS監視対象なし")
```

●「普通列車」のスクリプト

「普通列車」のスクリプトも、「特急列車」とほぼ同じです。

「ATS順路」が「本線」から「待避線側」になっています。

リスト4-5-2　「普通列車」のスクリプト

```
def vrmevent_270(obj,ev,param):
    if ev == 'init':
        # ATS順路登録
        d = obj.GetDict()
        d["route"] = [223, 224, 225, 253,
255, 226, 228, 257, 229, 259, 230, 261, 231]
        # ATS順路位置登録
        d["route_now"] = 5
        i = d["route"][d["route_now"]]
        # ATS列車登録
      l = vrmapi.LAYOUT()
        s = l.GetATS(i)
        ats_d = s.GetDict()
        ats_d["train"] = obj
        # ATS初期監視
        d["on_ats"] = l.GetATS(257)
        obj.SetEventAfter(2.0)
    elif ev == 'after':
        d = obj.GetDict()
        if("next_ats" in d):
            #vrmapi.LOG("ATS監視開始")
            chkNextATS(obj, d["next_ats"],
param["eventid"])
        elif("on_ats" in d):
            #vrmapi.LOG("出発監視開始")
            chkOnATS(obj, d["on_ats"], par
am["eventid"])
        else:
            vrmapi.LOG("ATS監視終了")
```

●「ATSセンサ」のスクリプト

「ATSセンサ」は一定間隔で複数配置します。

間隔は狭いほど多数の列車を動かすことができますが、最低でも「列車編成長＋ATSでの緊急停止制動距離」以上を空けてください。

「センサ・パーツ」は「ビュワー画面」では表示されないため、「架線柱」などの「目印」も一緒に配置します。

＊

「列車」を検知したらレイアウトの「setTrainOnATS」イベントを実行します。

「ATSセンサ」は複数配置するため、処理はなるべく書かずに、複製しやすいように実装します。

リスト4-5-3　「ATSセンサ」のスクリプト

```
def vrmevent_224(obj,ev,param):
    if ev == 'init':
        dummy = 1
    elif ev == 'catch':
        setTrainOnATS(obj)
```

●「駅センサ」のスクリプト

「駅センサ」のスクリプトは「自動運転処理」を記述しています。

「特急列車」と「普通列車」の識別を行ない、駅の「停車」と「発車」、「ポイントの切り替え」を制御します。

「出発時」の「ポイント切り替え」は、「分岐器直前に切り替え機能のみを有するセンサ」を独立配置しています。

リスト4-5-4　「ATSセンサ」のスクリプト

```
def vrmevent_264(obj,ev,param):
    if ev == 'init':
        dummy = 1
    elif ev == 'catch':
        #dummy = 1
        tra = obj.GetTrain()
        tra_n = tra.GetNAME()
        # ポイント定義
        l = vrmapi.LAYOUT()
        p = l.GetPoint(120)

        if(tra_n == "特急列車"):
            # 特急列車分岐切替
```

```
            p.SetBranch(0)
            tra.AutoSpeedCTRL(1000, 0.0)
            # 自動発車
            tra_d = tra.GetDict()
            tra_d["go_auto"] = 1
            tra.SetEventAfter(12.0)
        else:
            # 普通列車分岐切替
            p.SetBranch(1)
            tra.AutoSpeedCTRL(1000, 0.0)
            tra_d = tra.GetDict()
            tra_d["on_ats"] = l.GetATS(225)
            tra.SetEventAfter(12.0)
```

リスト ポイント切替センサのスクリプト

```
elif ev == 'catch':
    # ポイント定義
    l = vrmapi.LAYOUT()
    p = l.GetPoint(119)
    p.SetBranch(1)
```

●「レイアウト」のスクリプト

「レイアウト」のスクリプトは、センサと列車から呼ばれる関数処理を記述します。

特殊な点としては「ResetEvent」と「SetEvent After」が挙げられます。「

前方のATSセンサ監視処理」は「監視開始から終了まで一定間隔で繰り返し実行する」ものですが、「VRM-NX」の仕様ではマルチスレッドやWait処理がなく、時間イベントの監視機構はシングルタスクで動作するため、並列で「Afterイベント」を定義できません。

そこで、イベント中に「ResetEvent」と「SetEventAfter」を定義することで、繰り返し処理を実現しています。

リスト4-5-5 「レイアウト」のスクリプト

```
# 列車位置情報を更新
def setTrainOnATS(ats):
    # オブジェクト定義
    tra = ats.GetTrain()
    ats_d = ats.GetDict()

    # 1.ATSのDictに列車を登録
    ats_d["train"] = tra

    # 2.列車[ルート[現在地]]からATSを取得
    tra_d = tra.GetDict()
    route_ary = tra_d["route"]
    l = vrmapi.LAYOUT()
    ats_bef = l.GetATS(route_ary[tra_d["ro
ute_now"]])
    ats_bef_d = ats_bef.GetDict()
    # 前ATSのDictから列車を削除
    del ats_bef_d["train"]

    # 3.列車の位置を移動
    ats_id = ats.GetID()
    tra_d["route_now"] += 1
    # 配列を超えた場合
    if(tra_d["route_now"] >= len(route_ary)):
        # 最初に戻る
        tra_d["route_now"] = 0

    # 4.ATS
    # 更に次を確認
    route_nex = tra_d["route_now"] + 1
    # 配列を超えた場合
    if(route_nex >= len(route_ary)):
        # 最初に戻る
        route_nex = 0
    # 次のATSを取得
    ats_nex = l.GetATS(route_ary[route_nex])
    ats_nex_d = ats_nex.GetDict()
    # Dict["train"]があるか
    if ("train" in ats_nex_d):
```

```
        # 緊急停止
        tra.AutoSpeedCTRL(200,0.0)
        # 監視対象
        tra_d["next_ats"] = ats_nex
        # 監視イベント設定
        tra.SetEventAfter(2.0)

# 監視対象ATSにTrainが居なくなれば発車
def chkNextATS(tra, ats, evid):
    # ats.Dict["train"]がないか
    ats_d = ats.GetDict()
    if ("train" not in ats_d):
        # 監視解除
        tra_d = tra.GetDict()
        del tra_d["next_ats"]
        vrmapi.LOG("ATS解除")
        # 発車
        tra.AutoSpeedCTRL(200,1.0)
    else:
        # 監視延長
        tra.ResetEvent(evid)
        tra.SetEventAfter(2.0)

# 監視対象ATSにTrainがあれば発車
def chkOnATS(tra, ats, evid):
    # ats.Dict["train"]があるか
    ats_d = ats.GetDict()
    if ("train" in ats_d):
        ats_tra = ats_d["train"]
        ats_tra_n = ats_tra.GetNAME()
        # 特急列車のみ反応
        if ats_tra_n == "特急列車":
            # 監視解除
            tra_d = tra.GetDict()
            del tra_d["on_ats"]
            vrmapi.LOG("出発進行")
            # 発車
            tra.AutoSpeedCTRL(200,1.0)
        else:
            # 監視延長
            tra.ResetEvent(evid)
            tra.SetEventAfter(2.0)
            vrmapi.LOG("普通列車通過中")
    else:
        # 監視延長
        tra.ResetEvent(evid)
        tra.SetEventAfter(2.0)
        vrmapi.LOG("駅停車中")
```

*

「ATS」や「閉塞」の概念は、実際の鉄道にも利用されているシステムで、本物はさらに高度で複雑な機構を有しています。

*

しかし、それらの基礎システムである列車の「位置検出」や「衝突防止システム」を、たったこれだけのプログラムとソフトウェアで擬似的に再現できる「シミュレータ・ソフト」は希少です。

*

「VRM-NX」では「信号機」や「踏切」の機能も実装できます。

プログラミング可能な特徴を生かして、「ジオラマ」だけでなく、「ATC」や「移動閉塞」などの「鉄道システム」のシミュレーションにもチャレンジしてみてはいかがでしょうか。

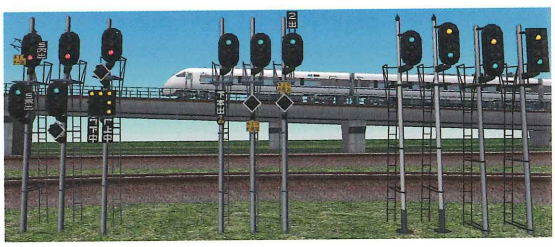

図4-5-4　VRM-NXの信号機

4-6 「機回し」による「折返し自動運転」

「機関車」と「客車」による「機回し」の「折り返し自動運転」を構築します。

■「VRM-NX」で「機回し」

レイアウトはU字型の線路の終端に、機回し線のある「機回し駅」を用意します。

「列車」は、牽引する「客車」と交換用に2種類の「機関車」を準備します。

「機関車」は「機関車交換」らしさのため、「ディーゼル機関車」と「電気機関車」の2種を選びました。

反対側の駅には、「機回し」と同様の動きを行なう「機関車交換駅」を用意していますが、こちらは次回の「機関車交換」で解説します。

■「機回し」の順序

「機回し」とは、鉄道の終着駅などで折り返し運転を行なう際に、客車を牽引する「機関車」の位置を、逆方向に付け替える作業です。

＊

機回しは、以下の順序で行なわれます。

[1]「列車」が駅に停車。
[2]「機関車」を「客車」から切り離す。
[3]「機関車」を「機関車待避線」へ進行。
[4]「機回し線」にポイントを切り替え。
[5]「機関車」を「折返し機回し線」へ進行。
[6]「機関車」を「本線」で停車。
[7]「本線」にポイントを切り替え。
[8]「機関車」を折り返し、「客車」と連結。
[9]「列車」が駅を出発。

図4-6-1 「機回し線」レイアウト

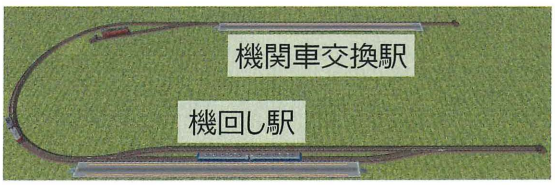

図4-6-2 レイアウト・ビュワー表示

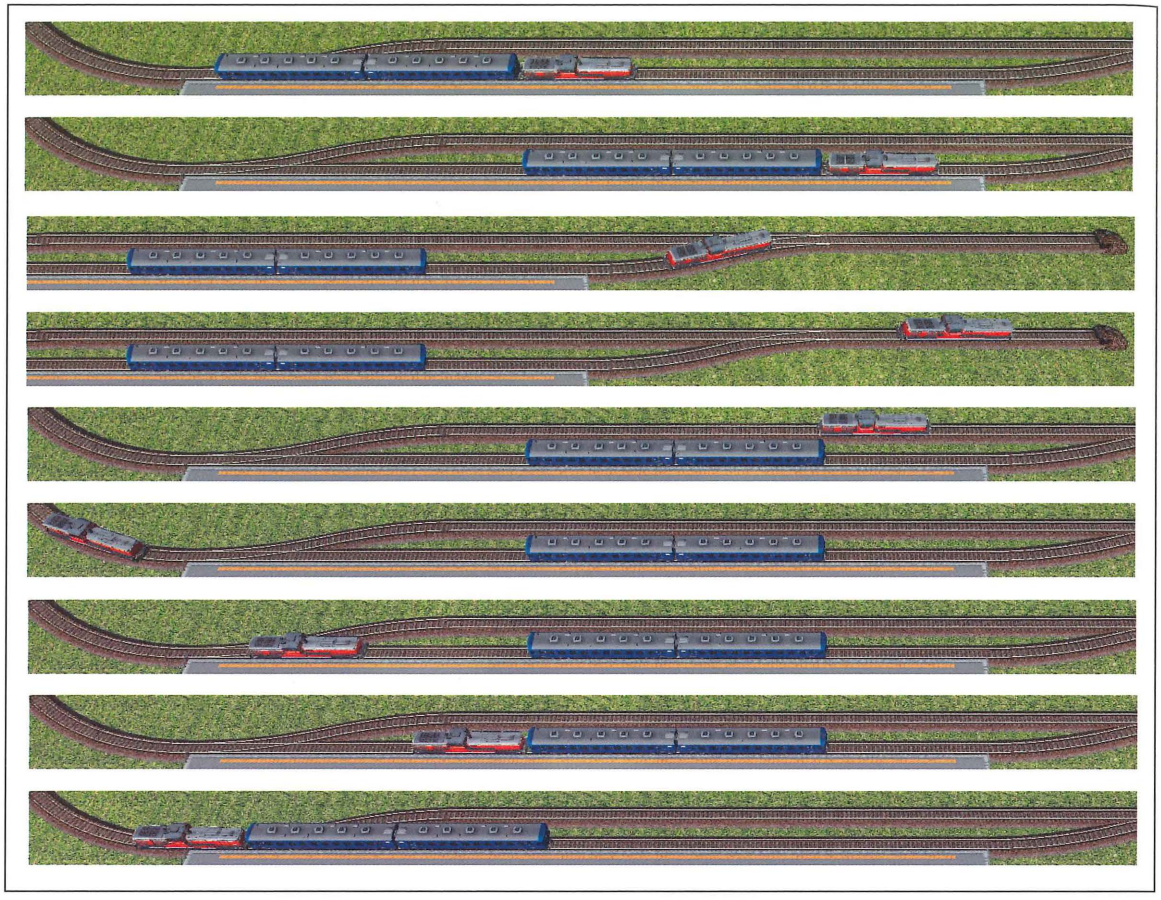

図4-6-3　「機回し」の順序

「VRM-NX」では、上記の動作を、「センサ・イベント」と「タイマ・イベント」を組み合わせて実行します。

「列車」の切り離し(解放)は「SplitTrain」関数、連結は「列車」を最高速度の25％以下で近接させることで、自動的に連結します。

■ スクリプト

●レイアウトのスクリプト

「レイアウト」のスクリプトには引数の値に応じて、列車を動作させる関数を記載します。

列車には、「関数の呼び出し」だけを記載して、同じ振る舞いをさせるようにします。

リスト4-6-1　レイアウトのスクリプト

```
def setTrainPattern(uid, tra):
    tra_n = tra.GetNAME()
    if uid == 1:
        vrmapi.LOG(tra_n + " 反転")
        tra.Turn()
    elif uid == 2:
        vrmapi.LOG(tra_n + " 機関車切離し")
        # 1両目で編成を分割
        tra.SplitTrain(1)
    elif uid == 102:
        vrmapi.LOG(tra_n + " 連結準備徐行")
        # 距離40mmで速度20%に変更
        tra.AutoSpeedCTRL(40, 0.2)
    elif uid == 104:
        vrmapi.LOG(tra_n + " 構内徐行")
        # 距離200mmで速度40%に変更
```

```
        tra.AutoSpeedCTRL(200, 0.4)
    elif uid == 110:
        vrmapi.LOG(tra_n + " 進行")
        # 距離600mmで速度100%に変更
        tra.AutoSpeedCTRL(600, 1.0)
    else:
        # 上記以外のUIDが指定された
        vrmapi.LOG(str(uid) +":設定無し")
```

●列車のスクリプト

「列車」のスクリプトには、(a)「タイマ・イベント」でレイアウトの「setTrainPattern」を呼び出す命令と、(b)「連結」したとき(ev='couple')に、「反転」と「自動出発」を行なうスクリプトを記載します。

*

「param['eventUID']」は、「VRM-NX」の最新ビルドで新しく追加されたパラメータです。

今までの「タイマ・イベント」は、オブジェクトに一つしか同時実行できませんでしたが、「eventUID」を振り分けることで、「タイマ・イベント」の並列実行が可能となります。

今回のは、「eventUID」を動作の識別IDに見立て、時間をズラして定義することで「連結解除」、「折返し」などの複合動作を実行させています。

リスト4-6-2 列車のスクリプト

```
#OBJID=367
import vrmapi
def vrmevent_367(obj,ev,param):
    if ev == 'init':
        # 初期自動発車
        obj.AutoSpeedCTRL(200,1.0)
    elif ev == 'after':
        # イベント呼び出し
        setTrainPattern(param['eventUID'], obj)
    elif ev == 'couple':
        # 反転
        obj.SetEventAfter(3.0, 1)
        # 進行
        obj.SetEventAfter(3.1, 110)
```

●「機回し」センサのスクリプト

今回使うスクリプトのポイントは、(a)センサの「GetForward」関数で「列車の向き」を、「GetNumberOfCars」関数で編成の車両数を判断しているところです。

また、「駅に来た列車」か「機回し中の機関車」かを判断し、列車に対して一連の動作を「タイマ・イベント」で命令することにあります。

これらのイベントのタイミングは、「駅の距離」や「列車の速度」によって正しく動作しない場合があるため、実際に動かして「時間」や「速度」を調節します。

リスト4-6-3 センサ①のスクリプト

```
elif ev == 'catch':
    # 列車を取得
    tra = obj.GetTrain()
    # 列車の向きを取得
    f = obj.GetForward()
    # 列車の車両数を取得
    n = tra.GetNumberOfCars()
    # 順方向
    if f == 1:
        # 本線側へポイント切替
vrmapi.LAYOUT().GetPoint(432).SetBranch(0)
        # 車両数が1両より多い(駅停車)
        if n > 1:
```

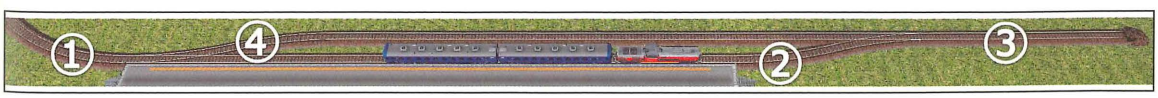

図4-6-4 「機回し」駅構内センサ位置

```
            # 距離800mmで停車
            tra.AutoSpeedCTRL(800, 0.0)
            # 7秒後に切り離し
            tra.SetEventAfter(7.0, 2)
            # 7.5秒後に構内徐行
            tra.SetEventAfter(7.5, 104)
    else:
        # 単機(機回し中)
        if n == 1:
            # 距離200mmで停車
            tra.AutoSpeedCTRL(200, 0.0)
            # 3秒後に反転
            tra.SetEventAfter(3.0, 1)
            # 3.5秒後に構内徐行
            tra.SetEventAfter(3.5, 104)
            # 8秒後に連結準備徐行
            tra.SetEventAfter(8.0, 102)
```

リスト4-6-4　センサ②のスクリプト

```
elif ev == 'catch':
    # ポイント切替
vrmapi.LAYOUT().GetPoint(437).SetBranch(1)
```

リスト4-6-5　センサ③のスクリプト

```
elif ev == 'catch':
    # 列車の向きを取得
    f = obj.GetForward()
    # 順方向
    if f == 1:
        # ポイント切替
vrmapi.LAYOUT().GetPoint(437).SetBranch(0)
    else:
        # 列車を取得
        tra = obj.GetTrain()
        # 距離130mmで停車
        tra.AutoSpeedCTRL(130, 0.0)
        # 4秒後に反転
        tra.SetEventAfter(4.0, 1)
        # 4.5秒後に進行
        tra.SetEventAfter(4.5, 110)
```

リスト4-5-6　センサ④のスクリプト

```
elif ev == 'catch':
    # ポイント切替
vrmapi.LAYOUT().GetPoint(432).SetBranch(1)
    # 方向転換のため事前減速
    tra = obj.GetTrain()
    tra.AutoSpeedCTRL(200, 0.7)
```

●好きなシーンを再現

「機回し」や「機関車交換」は「客車牽引」が減ってしまった昨今では、ごくわずかな場所でしか見ることができなくなりました。

そうした「昔の運用」を再現したり、現実にはなかった自分の好きな「機関車」と「客車」の組み合わせを動かして鑑賞することができるのも、「鉄道模型シミュレーターNX」ならではと言えるでしょう。

＊

今回は、製品版の車両を利用しました。

図4-6-5　製品版の車両

4-7 「機関車交換」を用いた「折返し」自動運転

前節で、「客車」を牽引する「機関車」の位置を逆方向へ付け替える「機回し」を紹介しました。今回は「機関車交換」を用いた「折返し」自動運転を紹介します。

■「VRM-NX」で「機関車交換」

レイアウトはU字型の線路の終端に機関車用の待避線を用意した「機関車交換駅」を用意します。

「列車」は牽引する「客車」と「機関車交換」に用いる2種類の「機関車」を準備します。

「機関車」は「機関車交換らしさ」を表現するのため、「ディーゼル機関車」と「電気機関車」の2種を選びます。

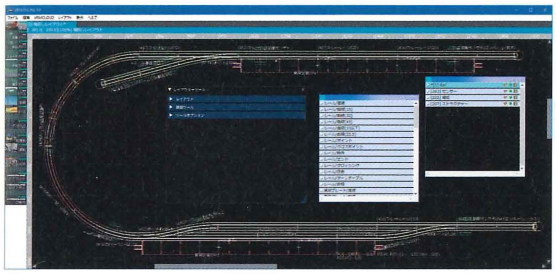

図4-7-1 機関車交換レイアウト

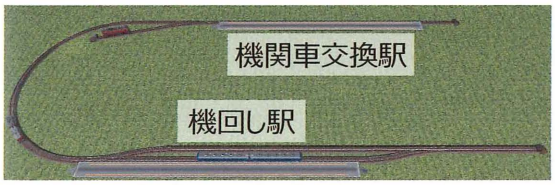

図4-7-2 レイアウト・ビュワー表示

■「機関車交換」の順序

「機関車交換」とは、「客車」や「貨車」を牽引する「機関車」を交代させるための作業です。

実際には、「運転区の境界」や、「電化・非電化区間を跨(また)ぐ貨物列車」などで見られます。

今回紹介するような「折り返し」も、同時に行なう「機関車交換」は「五稜郭駅」で行なわれていた「寝台列車」での、「電気機関車」と「ディーゼル機関車」のものが有名です。

実際の鉄道では、順行方向での付け替えが多いです。

「折り返し」を兼ねる「機関車交換」は、「機回し」と比べると複数の機関車が必要ですが、機回し線が不要で、作業時間も短縮できる利点があります。

*

「機関車交換」は、以下の順序で行なわれます。

[1] 「列車」が駅に停車。
[2] 「機関車」と「客車」を切り離し。
[3] 「機関車」を前方へ退避。
[4] 「ポイント」を待避線に切り替え。
[5] 待避線の「機関車」を「客車」の反対側へ連結。
[6] ポイントを本線に切り替え。
[7] 「列車」が駅を出発。
[8] 「列車」出発後にポイントを待避線へ切り替えて、残った「機関車」を待避線へ待避。

「VRM-NX」では上記の動作を「センサ・イベント」と「タイマ・イベント」を組み合わせて実行します。

「列車」の解放(切り離し)は「SplitTrain」関数を使います。「連結」は列車を「最高速度の25%以下で近接」させることで、自動的に行なわれます。

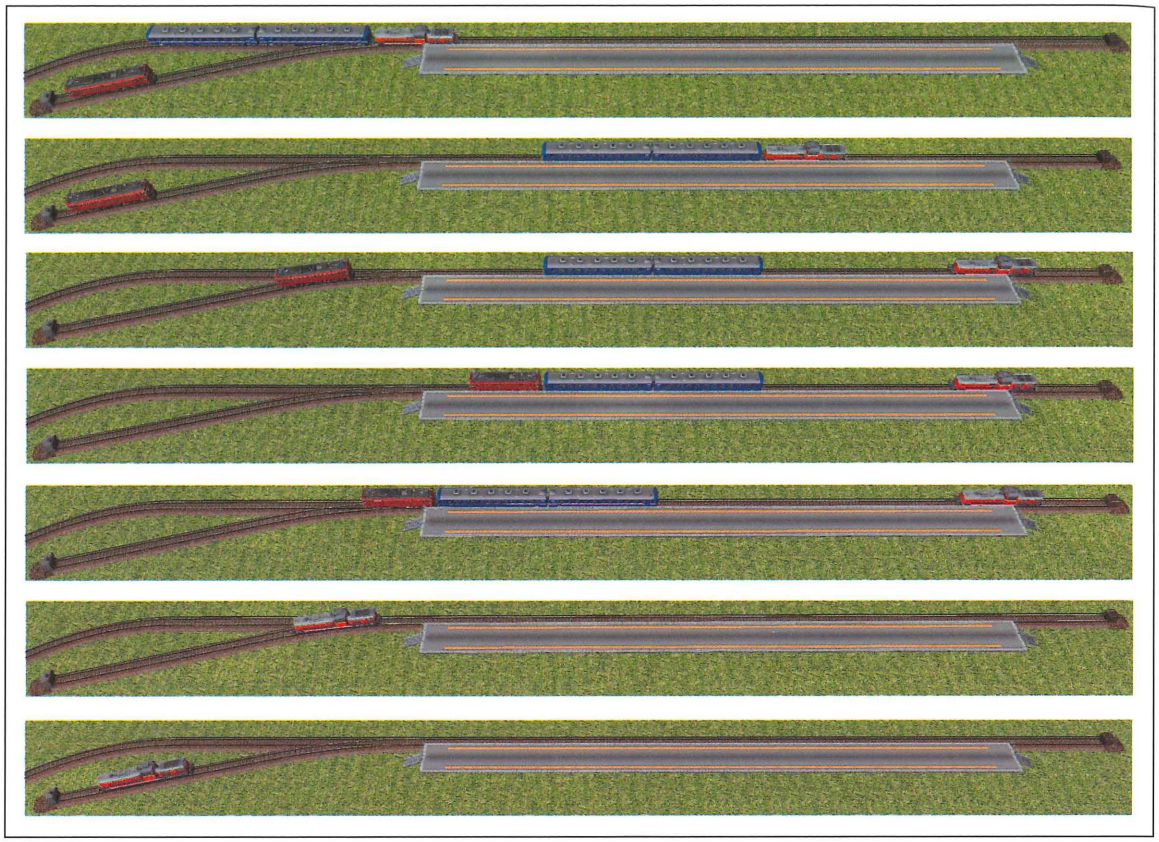

図4-7-3　機関車交換の順序

■ スクリプト

●レイアウトのスクリプト

「レイアウト」のスクリプトには、引数の値に応じて列車を動作させる「setTrainPattern」関数を記載します。

列車には「関数の呼び出し」だけを記載して、同じ振る舞いをさせるようにします。

リスト4-7-1　レイアウトのスクリプト

```
def setTrainPattern(uid, tra):
    tra_n = tra.GetNAME()
    if uid == 1:
        vrmapi.LOG(tra_n + " 反転")
        tra.Turn()
    elif uid == 2:
        vrmapi.LOG(tra_n + " 機関車切離し")
        # 1両目で編成を分割
        tra.SplitTrain(1)
    elif uid == 102:
        vrmapi.LOG(tra_n + " 連結準備徐行")
        # 距離40mmで速度20%に変更
        tra.AutoSpeedCTRL(40, 0.2)
    elif uid == 104:
        vrmapi.LOG(tra_n + " 構内徐行")
        # 距離200mmで速度40%に変更
        tra.AutoSpeedCTRL(200, 0.4)
    elif uid == 110:
        vrmapi.LOG(tra_n + " 進行")
        # 距離600mmで速度100%に変更
        tra.AutoSpeedCTRL(600, 1.0)
    else:
        # 上記以外のUIDが指定された
        vrmapi.LOG(str(uid) +":設定無し")
```

●列車のスクリプト

「列車」のスクリプトには「タイマ・イベント」でレイアウトの「setTrainPattern」を呼び出す命令と連結したとき(coupleイベント)に、反転と自動出発を行なうスクリプトを記載します。

「param['eventUID']」は「VRM-NX」のアップデートで新しく追加されたパラメータです。今までの「タイマ・イベント」はオブジェクトにつき一つしか同時実行できませんでしたが、「event UID」を振り分けることで「タイマ・イベント」の並列実行が可能となります。

*

今回はこの「eventUID」を識別IDとして使い、時間をズラして定義することで、「連結解除」や「折り返し」などの「複合時間差動作」を実現します。

リスト4-7-2 列車のスクリプト

```
def vrmevent_367(obj,ev,param):
    if ev == 'init':
        # 初期自動発車
        obj.AutoSpeedCTRL(200,1.0)
    elif ev == 'after':
        # イベント呼び出し
        setTrainPattern(param['eventUID'], obj)
    elif ev == 'couple':
        # 反転
        obj.SetEventAfter(3.0, 1)
        # 進行
        obj.SetEventAfter(3.1, 110)
```

■「機関車交換」センサのスクリプト

「機関車交換」スクリプトの要点は、(1)切り離した「機関車」を次の「列車」で呼び出して連結するため、切り離された「機関車」を「列車」出発後に待避線へ移動させることと、(2)次回呼び出しのために「センサ・②」の連想配列にオブジェクトを登録する処理が必要です。

「センサ・②」の処理内では「"wait_loco"」文字列をキーとして「機関車」オブジェクトを登録しています。

リスト4-7-3 センサ①のスクリプト

```
elif ev == 'catch':
    # 列車を取得
    tra = obj.GetTrain()
    # 列車の向きを取得
    f = obj.GetForward()
    # 列車の車両数を取得
    n = tra.GetNumberOfCars()
    # 順方向のみ
    if f == 1:
        # ポイント切替
vrmapi.LAYOUT().GetPoint(448).SetBranch(0)
        # 車両数が1両より多い(駅停車)
        if n > 1:
            # 距離800mmで停車
            tra.AutoSpeedCTRL(800, 0.0)
            # 8秒後に切り離し
            tra.SetEventAfter(8.0, 2)
            # 8.1秒後に構内徐行
            tra.SetEventAfter(8.1, 104)
```

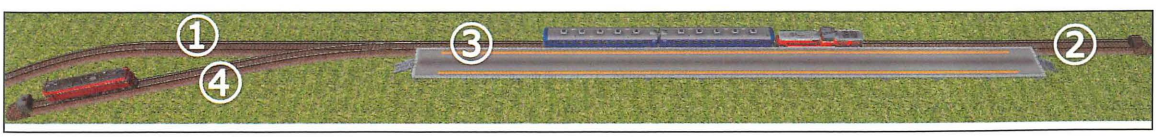

図4-7-4 「機関車交換」駅構内センサ位置

リスト4-7-4　センサ②のスクリプト

```
if ev == 'init':
    # 待機機関車の初期登録
    d = obj.GetDict()
    d["wait_loco"] = vrmapi.LAYOUT().GetTr
ain(421)
    d["wait_loco"].Turn()
elif ev == 'catch':
    d = obj.GetDict()
    tra = obj.GetTrain()
    f = obj.GetForward()
    # 順方向
    if f == 1:
        # A機関車新規登録
        vrmapi.LOG("機関車新規登録")
        d["wait_loco"] = tra
    else:
        # A機関車停止
        tra.AutoSpeedCTRL(100, 0)
        # B待機機関車呼び出し
        loco = d["wait_loco"]
        loco.Turn()
        loco.AutoSpeedCTRL(200, 0.4)
        # 3秒後にB連結準備徐行
        loco.SetEventAfter(3.0, 102)
        # 10秒後にB機関車反転
        loco.SetEventAfter(10.0, 1)
        # 10.1秒後にB機関車進行
        loco.SetEventAfter(10.1, 110)
        # 15秒後にA機関車反転
        tra.SetEventAfter(15.0, 1)
        # 15.1秒後にA機関車を待避線へ
        tra.SetEventAfter(15.1, 104)
```

リスト4-7-5　センサ③のスクリプト

```
elif ev == 'catch':
    # ポイント切り替え対象取得
    l = vrmapi.LAYOUT()
    p = l.GetPoint(448)
    # 列車を取得
    tra = obj.GetTrain()
    # 列車の向きを取得
    f = obj.GetForward()
    # 列車の車両数取得
    n = tra.GetNumberOfCars()
    # 順方向
    if f == 1:
        if n == 1:
            # 機関車単機なら待避線へ
            p.SetBranch(1)
        else:
            # 列車なら本線方向へ
            p.SetBranch(0)
    else:
        # 機関車のみ
        if n == 1:
            # 連結準備徐行
            tra.AutoSpeedCTRL(100, 0.2)
```

リスト4-7-6　センサ④のスクリプト

```
elif ev == 'catch':
    tra = obj.GetTrain()
    f = obj.GetForward()
    # 順方向
    if f == 1:
        # ポイント切替
vrmapi.LAYOUT().GetPoint(448).SetBranch(1)
    else:
        # 機関車停車
        tra.AutoSpeedCTRL(160, 0)
```

■ 昔のような客車牽引の再現も

「機回し」や「機関車交換」は、客車牽引が減ってしまった昨今では、ごくわずかな場所でしか見ることができなくなりました。

そうした昔の運用を再現したり、現実にはなかった自分の好きな機関車と客車の組み合わせを動かして鑑賞することも、「鉄道模型シミュレーター NX」の楽しみ方と言えるのではないでしょうか。

＊

今回は、製品版の車両を利用しました。

図4-7-5　製品版の車両

4-8　転車台を使った方向転換

「転車台(ターンテーブル)」を用いた「方向転換」を、「Pythonスクリプト」で自動運転する方法を紹介します。

図4-8-1　転車台

■「転車台」を使った「方向転換」

レイアウトは「機回し」用の機回し線と「転車台」のある駅を用意します。

反対側に位置する駅には「デルタ線」を内包する駅を用意します。

「列車」は、牽引する「客車」と「方向転換」に用いる「蒸気機関車」を準備します。

「列車」は、「転車台」と「デルタ線」を使って、常に「蒸気機関車」が先頭かつ前向き方向の状態で両駅間を往復します。

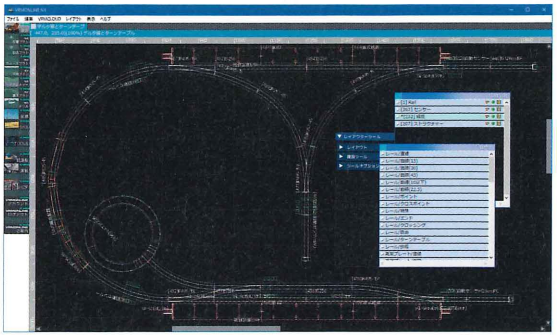

図4-8-2　ターンテーブル・レイアウト

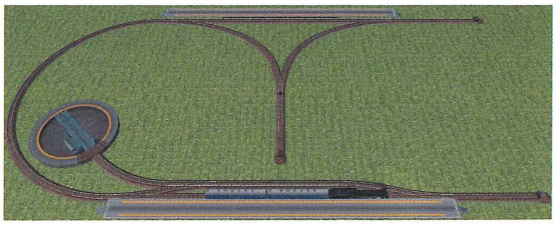

図4-8-3　「レイアウト・ビュワー」表示

■「転車台」を使った「方向転換」の順序

「転車台」とは、「蒸気機関車」のような片方向だけに運転台がある車両の向きを変える鉄道設備です。

終着駅や車両基地に設置されており、終着駅では「機回し」も同時に行なわれます。

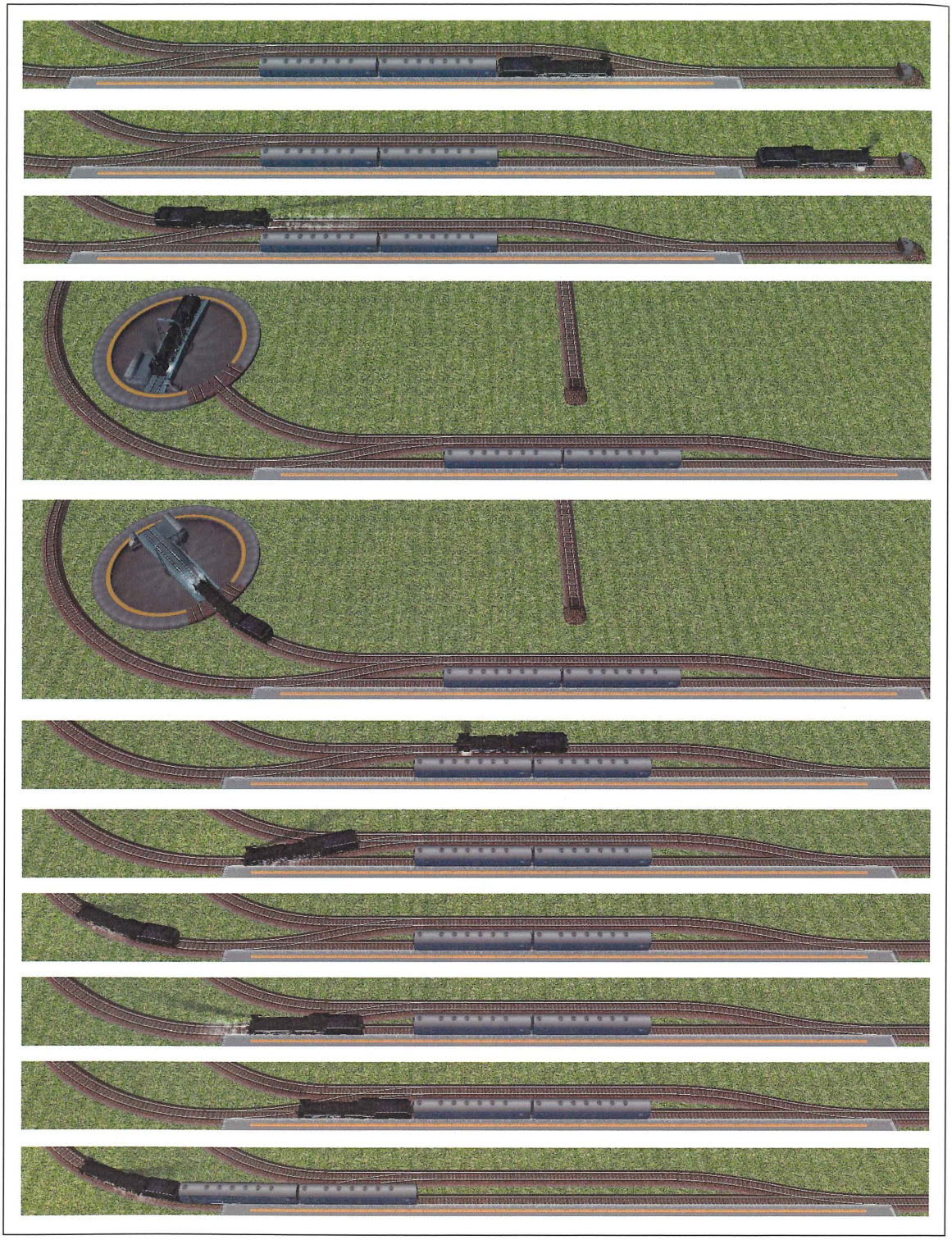

図4-8-4　「転車台」を使った「方向転換」の順序

実際にある転車台の設置場所は、駅によってさまざまなので、方向転換の順序も異なります。

*

今回の「転車台」の「方向転換」は以下の順序で行ないます。

[1]「列車」が駅に停車。
[2]「機関車」と「客車」を切り離し。
[3]「機関車」を前方へ退避。
[4]「ポイント」を待避線に切り替え。
[5]待避線の「機関車」を後進で「転車台」に載せる。
[6]「転車台」を180度回転。
[7]「転車台」から「機関車」を機回し線中央まで後進。
[8]ポイントを切り替えて「機関車」を本線まで前進。
[9]ポイントをホーム側へ切り替えて「機関車」を後進させて「客車」を連結。
[10]汽笛を鳴らしてホームを出発。

「VRM-NX」では上記の動作を「センサ・イベント」と「タイマ・イベント」を組み合わせて実行します。

「列車」の解放(切り離し)は「SplitTrain」関数を使います。
「連結」は列車を「最高速度の25％以下で近接」させることで自動的に行なわれます。

■ スクリプト

●レイアウトのスクリプト

「レイアウト」のスクリプトには引数の値に応じて列車を動作させる「setTrainPattern」関数を記載します。

列車には「関数の呼び出し」だけを記載して「uid引数」の違いだけで異なるイベントを実行します。

リスト4-8-1　レイアウトのスクリプト

```
def setTrainPattern(uid, tra):
    tra_n = tra.GetNAME()
    if uid == 1:
        vrmapi.LOG(tra_n + " 反転")
        tra.Turn()
    elif uid == 2:
        vrmapi.LOG(tra_n + " 機関車分離")
        tra.SplitTrain(2)
    elif uid == 3:
        vrmapi.LOG(tra_n + " 警笛")
        tra.PlayHorn(0)
    elif uid == 102:
        vrmapi.LOG(tra_n + " 警戒")
        tra.AutoSpeedCTRL(40, 0.2)
    elif uid == 104:
        vrmapi.LOG(tra_n + " 抑制")
        tra.AutoSpeedCTRL(200, 0.4)
    elif uid == 110:
        vrmapi.LOG(tra_n + " 進行")
        tra.AutoSpeedCTRL(600, 1.0)
```

●列車のスクリプト

「列車」のスクリプトには、「タイマ・イベント」でレイアウトの「setTrainPattern」を呼び出す命令と連結したとき(coupleイベント)に、「反転」と「自動出発」を行なうスクリプトを記載します。

*

今回は少し演出にこだわり、警笛を鳴らしてゆっくりとホームを出発し、その後、全力加速、という手順を再現しました。

リスト4-8-2　列車のスクリプト

```
def vrmevent_367(obj,ev,param):
    if ev == 'init':
        # 初期自動発車
        obj.AutoSpeedCTRL(200,1.0)
    elif ev == 'after':
        # 列車制御関数呼び出し
        setTrainPattern(param['eventUID'],
```

```
obj)
    elif ev == 'couple':
        # 反転
        obj.SetEventAfter(2.0, 1)
        # 警笛
        obj.SetEventAfter(2.5, 3)
        # ゆっくり出発
        obj.SetEventAfter(8.0, 102)
        # 進行
        obj.SetEventAfter(12.0, 110)
    elif ev == 'split':
        dummy = 1
```

●「転車台」のスクリプト

「転車台」スクリプトは「機関車」が「転車台」に差し掛かったことを示す「'trainin'」イベントと「転車台」が止まったときに発生する「'stop'」イベントを使います。

＊

「'trainin'」イベントによって「機関車」を「転車台」の中央に停車するように減速させ、時間差で「MoveTurntablePos」関数を使い「機関車」を反転させます。

リスト4-8-3　転車台のスクリプト

```
def vrmevent_496(obj,ev,param):
    if ev == 'init':
        dummy = 1
    elif ev == 'after':
        # 180度回転
        if obj.GetTurntablePos() == 0:
            obj.MoveTurntablePos(12)
        else:
            obj.MoveTurntablePos(0)
    elif ev == 'stop':
        vrmapi.LOG("回転終了")
        tra = obj.GetTrain()
        # 徐行
        tra.AutoSpeedCTRL(200, 0.2)
    elif ev == 'trainin':
        vrmapi.LOG("転車台進入:" + str(para
m['route']) + "=" + str(param['trainid']))
        tra = vrmapi.LAYOUT().GetTrain(par
am['trainid'])
        # 停車(距離注意)
        tra.AutoSpeedCTRL(60, 0.0)
        # 回転
        obj.SetEventAfter(4.0, 1)
```

「転車台」駅構内センサのスクリプト

「機関車」を「機回し」させるために4つのセンサを組み合わせて自動運転させます。

列車の向きを「GetForward」で、車両数を「GetNumberOfCars」関数で識別しています。

今回選択した「蒸気機関車」(C57)は「機関車」と「炭水車(テンダー)」の2両扱いになるため、切り離しの車両数に注意してください。

リスト4-8-4　センサ①のスクリプト

```
elif ev == 'after':
    # ポイント定義
    vrmapi.LAYOUT().GetPoint(432).SetBran
```

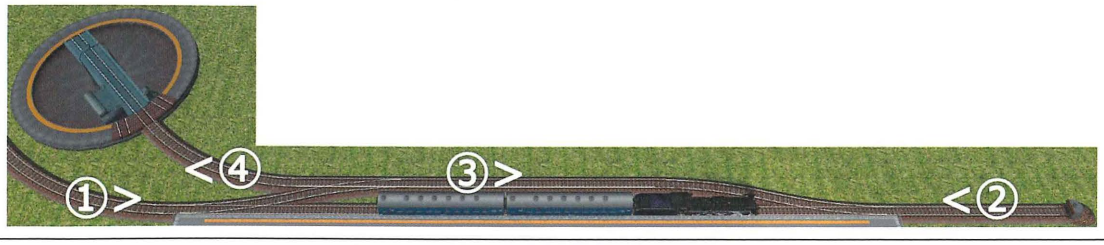

図4-8-5　「機関車交換」駅構内センサの位置

```
ch(0)
elif ev == 'catch':
    # 列車を取得
    tra = obj.GetTrain()
    # 列車の向きを取得
    f = obj.GetForward()
    # 列車の車両数を取得
    n = tra.GetNumberOfCars()
    # 順方向
    if f == 1:
        # ポイント定義
        vrmapi.LAYOUT().GetPoint(432).SetB
ranch(0)
        # 列車(駅停車)
        if n > 2:
            # ポイント定義
            vrmapi.LAYOUT().GetPoint(494).
SetBranch(0)
            # 列車停車
            tra.AutoSpeedCTRL(700, 0.0)
            # 切り離し
            tra.SetEventAfter(8.0, 2)
            # 単機抑制
            tra.SetEventAfter(8.5, 102)
    else:
        # 単機
        if n == 2:
            # 停車
            tra.AutoSpeedCTRL(80, 0.0)
            # 折返し
            tra.SetEventAfter(6.0, 1)
            # 折返し注意(連結用)
            tra.SetEventAfter(6.5, 102)
            # 6秒後ポイント切り替え
            obj.SetEventAfter(6.0)
```

リスト4-8-5　センサ②のスクリプト

```
elif ev == 'catch':
    f = obj.GetForward()
    tra = obj.GetTrain()
    # 巡行のみ
    if f == 1:
        # ポイント定義
        l = vrmapi.LAYOUT()
        l.GetPoint(494).SetBranch(1)
        l.GetPoint(437).SetBranch(0)
        # 転車台減速
        #tra.SetEventAfter(7, 102)
    else:
        # 停車
        tra.AutoSpeedCTRL(110, 0.0)
        # 反転
        tra.SetEventAfter(6.0, 1)
        # 進行
        tra.SetEventAfter(6.5, 104)
```

リスト4-8-6　センサ③のスクリプト

```
elif ev == 'after':
    # ポイント切り替え
    l = vrmapi.LAYOUT()
    l.GetPoint(432).SetBranch(1)
    l.GetPoint(437).SetBranch(1)
elif ev == 'catch':
    # 列車を取得
    tra = obj.GetTrain()
    # 列車の向きを取得
    f = obj.GetForward()
    # 順方向
    if f == 1:
        # 停車
        tra.AutoSpeedCTRL(120, 0.0)
        # 折返し
        tra.SetEventAfter(6.0, 1)
        # 単機抑制
        tra.SetEventAfter(6.5, 102)
        # 6秒後ポイント切り替え
        obj.SetEventAfter(6.0)
```

リスト4-8-7　センサ④のスクリプト

```
elif ev == 'catch':
    # 列車を取得
    tra = obj.GetTrain()
    # 列車の向きを取得
    f = obj.GetForward()
    # 順方向
    if f == 1:
        # 徐行
        tra.AutoSpeedCTRL(100, 0.2)
```

*

一現在、般的な列車は前後に運転席が付いているので「転車台」を使うシーンは貴重なものとなりました。

今でも「転車台」が実際に使われているのは、「東武鉄道」や「真岡鐵道」などの観光用の蒸気機関車を運行している路線がほとんどです。

そうした、それぞれの駅の運用を再現したり、自分の好きな蒸気機関車と客車を組み合わせて鑑賞することも、「VRM-NX」と自動運転の楽しみ方と言えるでしょう。

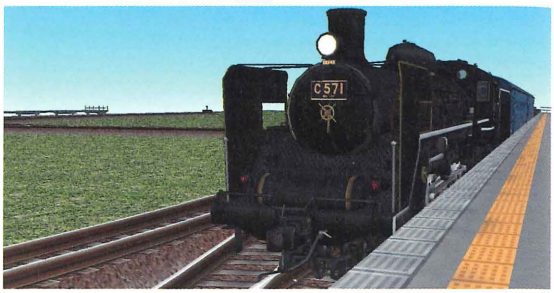

図4-8-6　ビュワーでリアルに再現

4-9 踏切を動かそう

「鉄道模型シミュレーター NX」(以下、「VRM-NX」)は、「列車」や「分岐ポイント」の他に、「信号機」や「踏切」などの鉄道設備を「Python」で動かすことができます。

今回は、列車を検出するセンサを組み合わせて、踏切を制御します。

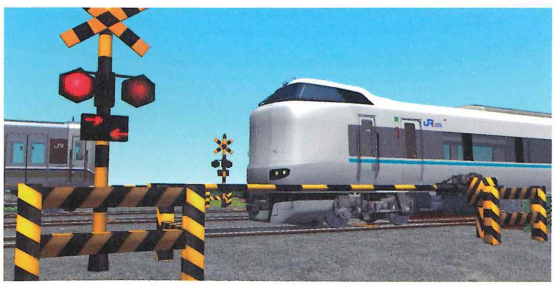

図4-9-1　踏切を制御する

■「警報機」と「遮断機」

VRM-NXの踏切では、「警報機」と「遮断機」を動かすことができます。

「警報機」は踏切警標に各種警報装置が付いたもので、“カンカンカン…”という警報音を鳴らし、警報灯を交互に発光させます。

種類によっては、列車がどちらから来るのかを知らせる「矢印」(列車進行方向指示器)が付いています。

「遮断機」は列車が近付いてきたときに線路への通行を制限するもので、一般的な棒状のものが開閉する「腕木式」が実装されています。

図4-9-2　踏切の警報機と遮断機

■「踏切」を動かす命令

「踏切を動かすための命令」は、主に以下の2つです。

(A)「SetCrossingStatus」は、警報機の「警報装置」と「遮断機」の開閉状態を設定します。

Int型引数に「2」を入れると警報機が鳴り、遮断機が降下します。

「1」を入れると警報音が止まり、遮断機が上がります。

(B)「SetCrossingSign」は、一部の警報機に付いている矢印「列車進行方向指示器」の表示を設定します。

「1」は左向き、「2」は右向き、「3」は両方向、「0」で表示無しとなります。

*

「SetCrossingSign」は設定しただけでは表示されず、「SetCrossingStatus」と連動して表示されます。

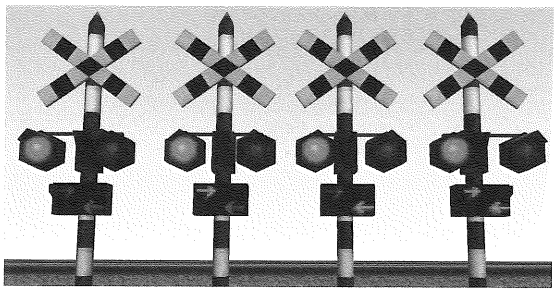

図4-9-3　「列車方向指示器」の例(左から、引数=0,1,2,3)

リスト4-9-1　「踏切」のスクリプト例

```
def vrmevent_1(obj,ev,param):
    if ev == 'init':
        # 方向指示器(3:両方向)
        obj.SetCrossingSign(3)
        # 踏切開閉状態(2:動作)
        obj.SetCrossingStatus(2)
```

■「センサ」と組み合わせて使う

「踏切」は、列車の接近をセンサで検知して踏切を動かし、列車が通過したあとに動作を止めます。

VRM-NXでもセンサと組み合わせて踏切を開閉することができます。

*

今回は踏切を閉じる用のセンサ①と、開ける用のセンサ②を用意します。

センサ①は列車の通過速度によって位置を調整します。

センサ②は踏切通過直後に設置します。

*

「方向表示器」を使う場合、左右の向きを決めるための基準となる「基準警報機」を決めます。

「基準警報機」の対面に置かれる警報機の矢印は、プログラム上では反転するので、注意してください。

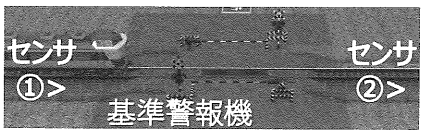

図4-9-4　踏切センサ配置例

■スクリプト

●センサ①のスクリプト

「センサ①」のスクリプトには列車を検知した際に、「SetCross」関数を呼びます。

単線区間のように線路上を双方向から列車が行き来する場合は、誤作動を避けるために、列車の向きが順方向でのみ動くようにします。

引数「1」は基準警報機から見て左から来る列車を示します。

右から来る列車のセンサには「10」を入れます。

リスト4-9-2　「センサ①」のスクリプト(抜粋)

```
elif ev == 'catch':
    dummy = 1
    # 列車の向きを取得
    f = obj.GetForward()
    # 順方向のみ
    if f == 1:
        SetCross(470, 1)
```

●センサ②のスクリプト

「センサ②」のスクリプトも①と同じ「SetCross」関数を呼びます。

異なるのは、引数が「-1」である点です。

右から来る列車のセンサには、「-10」を設定します。

起動時(init)に「SetSNSMode(1)」を実行することで、センサの実行タイミングを「先頭車両の通過直前」から「最後尾車両の通過後」に変更します。

これによって、編成長が違っても、必ず列車が踏切を通過してから開くようになります。

リスト4-9-3　「センサ②」のスクリプト

```
if ev == 'init':
    dummy = 1
    obj.SetSNSMode(1)
elif ev == 'catch':
    dummy = 1
    # 列車の向きを取得
    f = obj.GetForward()
    # 順方向のみ
    if f == 1:
        SetCross(470, -1)
```

●「踏切」のスクリプト

「踏切」のスクリプトには、動作に必要な情報を「基準警報機」に記載します。

「踏切」をグループ化して処理を分離することで、最小限の定義でレイアウト内に複数の踏切を設置することができます。

リスト4-9-4　踏切のスクリプト

```
#OBJID=470
import vrmapi
def vrmevent_470(obj,ev,param):
    if ev == 'init':
        dummy = 1
        # 主オブジェクト変数初期化
        l = vrmapi.LAYOUT()
        d = obj.GetDict()
        # 踏切が検知している編成
        d['closs_tr_cnt'] = 0
        # 1踏切の部品グループ
        d['closs_obj_id']=[470,477,471,478]
        # 部品グループの種別
        #   1:警報機(正)
        #  -1:警報機(向)
        #   2:遮断機
        d['closs_obj_type'] = [1, -1, 2, 2]
    elif ev == 'open':
        dummy = 1
    elif ev == 'close':
        dummy = 1
```

●レイアウトのスクリプト

レイアウトのスクリプトには、センサ情報の入力と、踏切の動作を記述します。

センサからの引数の値は、「d['closs_tr_cnt']」で管理されます。

この値は「1桁目」に左から来た列車、「2桁目」に右から来た列車をそれぞれ定義していて、最大9編成まで管理できます。

複々線の対応や、踏切動作中に反対から列車が来たときに方向表示器の表示を更新することができます。

*

「SetCrossingStatusGP」関数は、「踏切グループ」のオブジェクトを「for文」を用いて処理します。

「警報機」と「遮断機」の違いや、「警報機」の「正面」か「対面」かで、方向表示器の表示を切り分けて処理しています。

リスト4-9-5　レイアウトのスクリプト

```
# センサ情報を踏切に入力
#   obj_id 基準警報機の部品ID
#   センサ引数
def SetCross(obj_id, fd):
    l = vrmapi.LAYOUT()
    obj = l.GetCrossing(obj_id)
    d = obj.GetDict()
    # 列車管理カウントを追加
    d['closs_tr_cnt']=d['closs_tr_cnt']+fd
    # 列車管理カウントが0
    if d['closs_tr_cnt'] == 0:
        # 開ける
        SetCrossingStatusGP(obj, 0)
        vrmapi.LOG("i=" + str(d['closs_tr_
cnt']))
    else:
        # 踏切状態がON以外
        if obj.GetCrossingStatus() != 2:
            # 遮断機閉める
            SetCrossingStatusGP(obj, 9)
        # 方向表示計算
        tr_l = d['closs_tr_cnt'] // 10
        tr_r = d['closs_tr_cnt'] % 10
        vrmapi.LOG("i=" + str(d['closs_tr_
cnt']) + " L=" + str(tr_l) + " R=" + str(
tr_r))
        # 左右あり
        if (tr_l != 0) and (tr_r != 0):
            SetCrossingStatusGP(obj, 3)
        # 右のみ
        elif (tr_l == 0) and (tr_r != 0):
            SetCrossingStatusGP(obj, 2)
        # 左のみ
        elif (tr_l != 0) and (tr_r == 0):
            SetCrossingStatusGP(obj, 1)

# 踏切の状態を表現
def SetCrossingStatusGP(obj, status):
    l = vrmapi.LAYOUT()
    d = obj.GetDict()
    i = 0
    # 踏切グループを繰り返し
    for obj_id in d['closs_obj_id']:
        # 踏切オブジェクトを定義
        c = l.GetCrossing(obj_id)
        # 踏切オブジェクト種類を定義
        t = d['closs_obj_type'][i]
        # 開ける
        if status == 0:
            # 警報機
            if t == 1 or t == -1:
                c.SetCrossingStatus(0)
            # 遮断機
            elif t == 2:
                c.SetCrossingStatus(1)
        # 閉める
        elif status == 9:
            c.SetCrossingStatus(2)
        # 方向表示(左)
        elif status == 1:
            # 警報機
            if t == 1:
                c.SetCrossingSign(1)
            # 警報機(対面)
            elif t == -1:
                c.SetCrossingSign(2)
        # 方向表示(右)
        elif status == 2:
            # 警報機
            if t == 1:
                c.SetCrossingSign(2)
```

```
        # 警報機(対面)
        elif t == -1:
            c.SetCrossingSign(1)
    # 方向表示(両方)
    elif status == 3:
        c.SetCrossingSign(3)
```

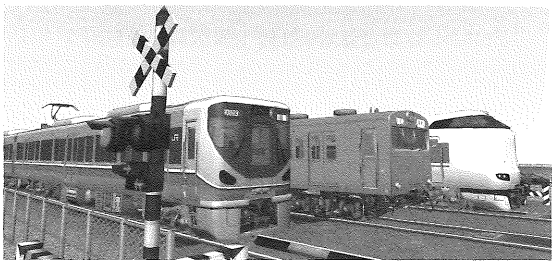

図4-9-5　踏切をリアルに再現

4-10　「ImGui」を使った「自作コントローラ」

「レイアウター」と「ビュワー」のインターフェイスに「ImGui」を採用しています。

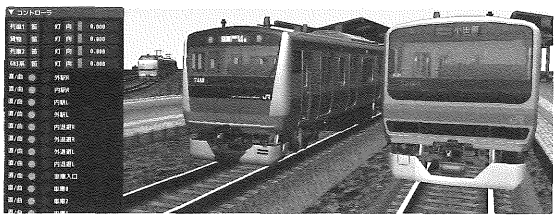

図4-10-1

■「VRM-NX」で「ImGui」を使う

「ImGui」とは、**ゲーム画面内でゲーム設定などを操作するためのインターフェイスを提供する、軽量で直感的な「汎用GUIライブラリ」**で、その扱いやすさからソフトウェア開発者がデバッグ用に利用することがあります。

「VRM-NX」ではこの「ImGui」の機能をPythonで扱えるようにAPIを提供しており、レイアウター画面で自由なインターフェイスを作れるようになっています。

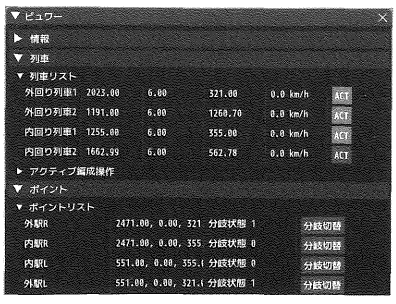

図4-10-2　「レイアウター」画面で「インターフェイス」が作れる

■「ImGuiウインドウ」を描画する

「ImGui」を描画するには「フレーム・イベント」の「メイン・ループ」内で、毎回処理する必要があります。

「フレーム・イベント」を使うため、起動時に「SetEventFrame」関数を呼び出します。

*

まずは「空ウインドウ」を描画してみます。

ウインドウは「Begin」関数と「End」関数をセットで使います。

「Begin」関数の引数にはウインドウの「識別子」と「表示名」を入れます。

*

以下のコードを、「レイアウト・スクリプト」に記述します。

リスト4-10-1　空ウインドウ・スクリプト

```
#LAYOUT
import vrmapi
def vrmevent(obj,ev,param):
    if ev == 'init':
        # ImGui用イベント定義
        obj.SetEventFrame()
    elif ev == 'broadcast':
        dummy = 1
    elif ev == 'timer':
        dummy = 1
```

```
    elif ev == 'time':
        dummy = 1
    elif ev == 'after':
        dummy = 1
    elif ev == 'frame':
        # ここでフレーム毎にImGuiを描画
        vrmapi.ImGui().Begin("w1","Sample
Window")
        vrmapi.ImGui().End()
    elif ev == 'keydown':
        dummy = 1
```

図4-10-3 ImGuiで「空ウインドウ」表示

「ImGuiコントロール」を描画する

こんどはウインドウ内に、「ボタン」や「チェック・ボックス」を追加してみましょう。

*

「ImGui」の利点の一つに、「描画とイベントが同じ場所に書ける」ことが挙げられます。

処理を「if文」として記述することで、ボタンのクリックが発生したことを検知できます。

処理内に変数を定義してイベント発生を検知することで、リアルタイムな挙動を返すことができます。

*

以下のコードは、公式のサンプル（https://vrmcloud.net/nx/script/script/imgui/index.html）を改修したものです。

「一般的なボタン」「チェック・ボックス」「ラジオ・ボタン」「数値入力ボックス」「スライド・バー」が利用できます。

リスト4-10-2 サンプル表示スクリプト

```
#LAYOUT
import vrmapi

# globalで初期化
btn = 0
chk = [0]
nNum = [0]
fNum = [0.0]
radio0 = [0]

def vrmevent(obj,ev,param):
    if ev == 'init':
        obj.SetEventFrame()
    elif ev == 'broadcast':
        dummy = 1
    elif ev == 'timer':
        dummy = 1
    elif ev == 'time':
        dummy = 1
    elif ev == 'after':
        dummy = 1
    elif ev == 'frame':
        global btn
        global chk
        global nNum
        global fNum
        global radio0
        # 毎回の「vrmapi.ImGui()」が面倒なので変
数gを定義
        g = vrmapi.ImGui()
        g.Begin("w1","Sample Window")
        g.Separator()
        g.Text("ボタンテスト")
        if g.Button("b1", "ボタンその1"):
            btn = btn + 1
        g.Text("ボタンその1を" + str(btn) +
"回クリックした")
        g.Separator()
```

```
        if g.CollapsingHeader("head1", "He
ader"):
            g.Text("チェックボックステスト")
            g.Checkbox("chk1", "チェックA",
chk)
            g.Text("chk =>"+str(chk[0]))

        if g.TreeNode("tree1", "Tree-A"):
            g.PushItemWidth(160.0)
            g.InputFloat("f1", "浮動小数点数
", fNum)
            g.InputInt("n1", "整数", nNum)
            g.Text("[nNum, fNum] =>"+str(f
Num + nNum))
            g.PopItemWidth()
            g.TreePop()

        if g.TreeNode("tree2", "Tree-B"):
            g.RadioButton("r0", "ラジオボタ
ンテスト0", radio0, 0)
            g.RadioButton("r1", "ラジオボタ
ンテスト1", radio0, 1)
            g.RadioButton("r2", "ラジオボタ
ンテスト2", radio0, 2)
            g.RadioButton("r3", "ラジオボタ
ンテスト3", radio0, 3)
            g.RadioButton("r4", "ラジオボタ
ンテスト4", radio0, 4)
            g.Text("radio0 =>"+str(rad
io0))
            g.TreePop()

        g.End()
    elif ev == 'keydown':
        dummy = 1
```

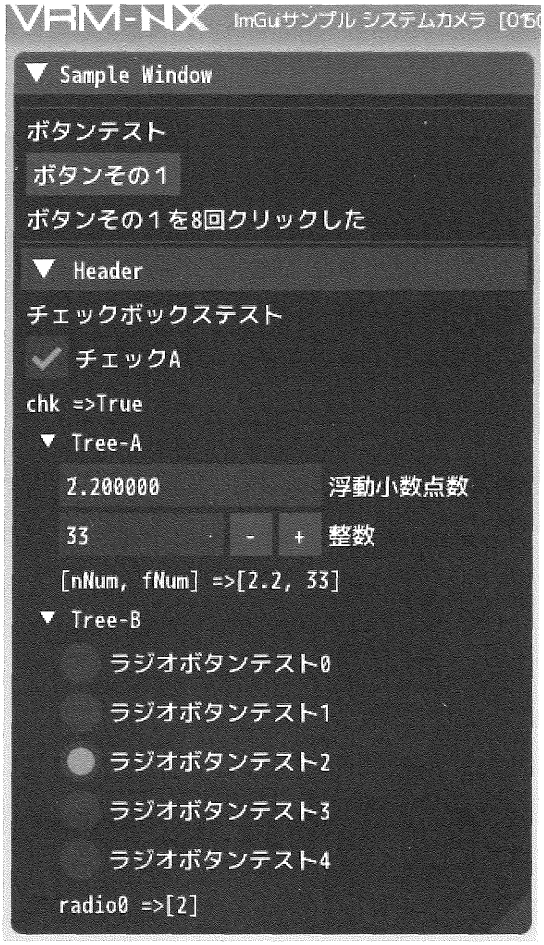

図4-10-4　ImGuiでサンプル表示

■「ImGui」で「自作コントローラ」を作

サンプルでは固定文字や変数を表示させていましたが、「VRM-NX」では列車の情報や、ポイントの切り替えを「ImGui」から操作できます。

＊

今回は列車を複数制御できる「自作コントローラ」を作ってみましょう。

実装する「自作コントローラ」を図4-10-5に示します。

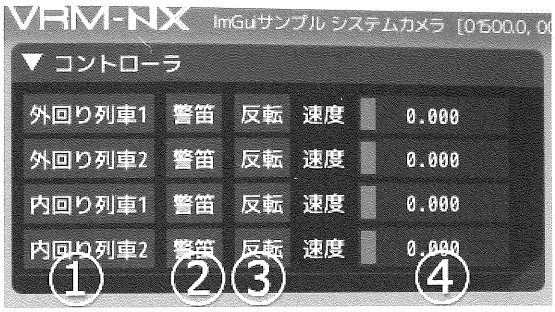

図4-10-5 自作コントローラ

コントローラは、主に4つの機能で構成されています。

①ボタンは**編成名**を表示し、クリックすると操作対象の列車になる
②クリックで**警笛**を鳴らす
③クリックで進行方向を切り替える
④スライド・バーで**列車の速度**を調整する

製品標準コントローラは、操作対象となっている編成しか「速度」や「方向転換」の操作ができませんが、この自作コントローラを使うと、俯瞰視点で複数の編成を一度に操作できます。

リスト4-10-3 自作コントローラ・スクリプト

```
#LAYOUT
import vrmapi

def vrmevent(obj,ev,param):
    if ev == 'init':
        # ImGui描画用変数の登録
        setImGuiObjState()
        # フレームイベントの登録
        obj.SetEventFrame()
    elif ev == 'broadcast':
        dummy = 1
    elif ev == 'timer':
        dummy = 1
    elif ev == 'time':
        dummy = 1
    elif ev == 'after':
        dummy = 1
    elif ev == 'frame':
        # 自作コントローラの描画
        drawImGui()
    elif ev == 'keydown':
        dummy = 1

# ImGui描画用変数の登録
def setImGuiObjState():
    # 編成リストを新規編成リストに格納
    trainList = vrmapi.LAYOUT().GetTrainList()
    # 新規編成リストから編成を繰り返し取得
    for tra in trainList:
        di = tra.GetDict()
        # 速度
        di['imgui_vol'] = [0.0]
        # 向き
        di['imgui_sw'] = [0]

# 自作コントローラの描画
def drawImGui():
    g = vrmapi.ImGui()
    g.Begin("con","コントローラ")
    # 新規編成リストを作成
    trainList=list()
    # 編成リストを新規編成リストに格納
    vrmapi.LAYOUT().ListTrain(trainList)
    # 編成一覧を参照
    for train in trainList:
        # 編成コントローラ
        drawImGuiTraCon(g, train)
    g.End()

# 編成コントローラ
def drawImGuiTraCon(g, tra):
    # ①編成名ボタン
    if g.Button("bt1" + str(tra.GetID()),
tra.GetNAME()):
        # 編成操作視点
        tra.SetView()
```

```
    g.SameLine()

    # ②警笛ボタン
    if g.Button('bt2' + str(tra.GetID()),
"警笛"):
        # 警笛実行
        tra.PlayHorn(0)
    g.SameLine()
    # 編成内のパラメータを取得
    di = tra.GetDict()

    # 電圧変数取得
    vlary = di['imgui_vol']
    # ③反転ボタン
    if g.Button('bt3' + str(tra.GetID()),
"反転"):
        # 方向転換実行
        tra.Turn()
        # 速度スライドバーを0で再描画
        vlary[0] = 0.0
        g.SliderFloat('vl' + str(tra.Get
ID()), '', vlary, 0, 1.0)
    g.SameLine()

    g.Text("速度")
    g.SameLine()
    # スライドバーサイズ調整
    g.PushItemWidth(100.0)
    # ④速度スライドバー
    if g.SliderFloat('vl' + str(tra.Get
ID()), '', vlary, 0, 1.0):
        # 電圧反映
        tra.SetVoltage(vlary[0])
    # サイズリセット
    g.PopItemWidth()
```

*

今回のスクリプトは、①初期設定を行なう「setImGuiObjState」関数、②毎回フレームイベントから呼び出される「drawImGui」関数、③「drawImGui」から編成単位で呼び出される「drawImGuiTraCon」関数――の3つで構成されています。

編成の一覧は「ListTrain」で取得したものを「for文」で読み込むことで、新しい編成をレイアウトに追加しても、コードの変更なしでウインドウに追加されます。

また、操作対象の編成オブジェクト自身にImGui用のイベント変数を定義することで、サンプルのような「global関数」を定義することなく動作するように設計されていることが特徴です。

■「ImGui」で自由にウインドウの構築を

「Pythonスクリプト」は「無料のスターターキット」(http://www.imagic.co.jp/hobby/products/vrmnx/start/starter/)でも動きます。

編成の他にもポイントやカメラなどほとんどのパラメータを扱うことができます。

皆さんも「ImGui」で、いろいろなウインドウを構築してみてください。

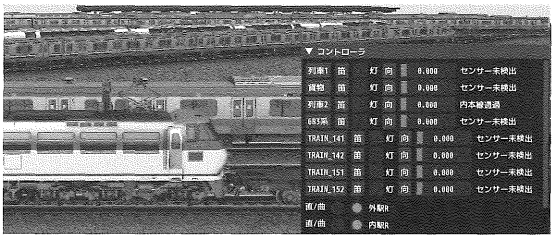

図4-10-6　コントローラを自在に作成できる

4-11 VRM-NXの「外」と連携しよう

従来の「鉄道模型シミュレーター」で実装されていたスクリプトは、ゲーム内のオブジェクトを操作するためのものでした。

しかし、最新作の「鉄道模型シミュレーターNX」(以下、「VRM-NX」)では、「Pythonスクリプト」を使うことで、ゲーム内だけでなくゲームの「外」と連携することができます。

図4-11-1　スクリプトでゲーム外と連携

■ゲームの「外」とつながる

「外との連携」には大きく2つの意味があります。

①スクリプトの連携

PythonをVRM-NX内だけでなく、外部ファイルに実装したり、外部ライブラリを読み込むことができます。

②データの連携

VRM-NX内の情報を外部に出力したり、入力したりすることができます。

■ 自作モジュールのメリット

「レイアウト・ファイル」にPythonを記述すると、同じ処理を別のレイアウトで実装するときも、毎回同じものを作る必要があります。

また、「編集ダイアログ」(レイアウト・スクリプト・エディタ)を開いている間は、他の情報を閲覧できず、パーツIDを調べたり他のスクリプトを確認することができないため、作業効率が大きく低下します。

*

「呼び出し部分」など最低限の「Pythonスクリプト」以外を、「自作モジュール」として外部ファイルに記載することで、複数のレイアウトから関数を利用できるようになります。

さらに、「Visual Studio Code」などの汎用エディタを使うことで、コーディングの効率が大きく向上します。

図4-11-2　「Visual Studio Code」を使う

●「自作モジュール」の読込み方法

「自作モジュール」はレイアウト・ファイルと同じディレクトリに「pyファイル」を作り、「import」でファイル名を読み込むことで利用できます。「pyファイル」の文字コードは「UTF-8」で保存してください。

*

以下の例は、レイアウト起動時に列車の一覧リストを出力するスクリプトです。

リスト4-11-1はレイアウト・スクリプトのメイン処理に記述している例、「リスト2」は関数を独立させた例、「リスト3」と「リスト4」は同じ処

理を自作モジュール「vrmapiex.py」に分離して記述した例です。

いずれの記述も実行結果は「リスト5」になります。

リスト4-11-1　レイアウト・スクリプト直接記述

```
import vrmapi

def vrmevent(obj,ev,param):
  if ev == 'init':
    # 編成リストを新規編成リストに格納
    tList = vrmapi.LAYOUT().GetTrainList()
    # 編成を繰り返し取得
    for t in tList:
      # 編成ID
      id = t.GetID()
      # 編成名
      name = t.GetNAME()
      # 出力
      vrmapi.LOG(str(id) + '-' + name)
  elif ev == 'broadcast':
    dummy = 1
```

リスト4-11-2　レイアウト・スクリプト別関数記述

```
import vrmapi

def vrmevent(obj,ev,param):
  if ev == 'init':
    # 関数名で呼出し
    showTrainList()
  elif ev == 'broadcast':
    dummy = 1

def showTrainList():
  # 編成リストを新規編成リストに格納
  tList = vrmapi.LAYOUT().GetTrainList()
  # 新規編成リストから編成を繰り返し取得
  for t in tList:
    # 編成ID
    id = t.GetID()
    # 編成名
    name = t.GetNAME()
    # 出力
    vrmapi.LOG(str(id) + '-' + name)
```

リスト4-11-3　レイアウト・スクリプト(呼出のみ)

```
import vrmapi
# 自作モジュールを呼出し
import vrmapiex

def vrmevent(obj,ev,param):
  if ev == 'init':
    # モジュール名.関数名で呼出し
    vrmapiex.showTrainList()
  elif ev == 'broadcast':
    dummy = 1
```

リスト4-11-4　vrmapiex.py

```
# vrmapiを利用するのでvrmapiをimport
import vrmapi

def showTrainList():
  # 編成リストを新規編成リストに格納
  tList = vrmapi.LAYOUT().GetTrainList()
  # 新規編成リストから編成を繰り返し取得
  for t in tList:
    # 編成ID
    id = t.GetID()
    # 編成名
    name = t.GetNAME()
    # 出力
    vrmapi.LOG(str(id) + '-' + name)
```

リスト4-11-5　実行結果(例)

```
string : 10-251系
string : 54-キハ85系
string : 64-683系
string : 25-787系
```

＊

「自作モジュール」以外にも、ファイル操作用の「shutil」モジュールや「pathlib」などの「一般モジュール」を呼び出すことができます。

ただし、「Socket」などの一部モジュールは利用に制限があり、それ以外のモジュールについても基本的には「VRM-NX」の動作保証外のため、意図しない動きに注意してください。

■ 情報をファイル出力する

次は、「VRM-NX」の情報を外部に出力してみましょう。

＊

列車の一覧を、「ログ・ウィンドウ」ではなく「TrainList.txt」ファイルに出力する、「writeTrainList」関数を作ります。

リスト4-11-6 vrmapiex.py ファイル出力関数

```
# vrmapiを利用するのでvrmapiをimport
import vrmapi

def writeTrainList():
  # 編成リストを新規編成リストに格納
  tList = vrmapi.LAYOUT().GetTrainList()
  # 文字連結用
  stream = list()
  # 新規編成リストから編成を繰り返し取得
  for t in tList:
    # 編成ID
    id = t.GetID()
    # 編成名
    name = t.GetNAME()
    # 出力
    stream.append(str(id) + '-' + name + '\n')
  # 結合
  text = ''.join(stream)
  # ファイルパス
  path_w = 'TrainList.txt'
  # ファイル出力
  with open(path_w, mode='w') as f:
    f.write(text)
```

ビュワーを一度実行すると、「レイアウト・ファイル」と同じフォルダに、「テキスト・ファイル」が出力されます。

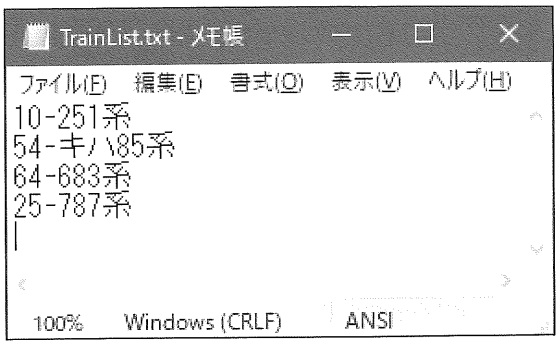

図4-11-3　TrainList.txt出力結果

保存されていないレイアウトで相対パスを使った場合は、「カレント・ディレクトリ」が意図しない場所に設定されてしまいます。

そのため、「ファイル操作」をする場合は、「レイアウト・ファイル」を保存してから実行してください。

■ 「外」に広がる無限の可能性

「プログラム」との「データ」のやり取りを「ファイル形式」で行なう方法は、**「ファイル連携」**と言い、一般的なシステム間連携でも使われている手法です。

この「ファイル連携」を「タイマ・イベント」などで定期的に実行することで、「擬似的なリアルタイム連携」が可能になります。

＊

出力した情報ファイルはさらに別のプログラムで「ネットワーク通信」に変換したり、「Webサービス」と連携することができます。

＊

反対に、「コントローラやプログラムからの操

作命令」「自動運転用のダイヤ」を「VRM-NX」にファイル読み込みさせることで、「列車の運転」や「ポイントの切り替え」も可能でしょう。

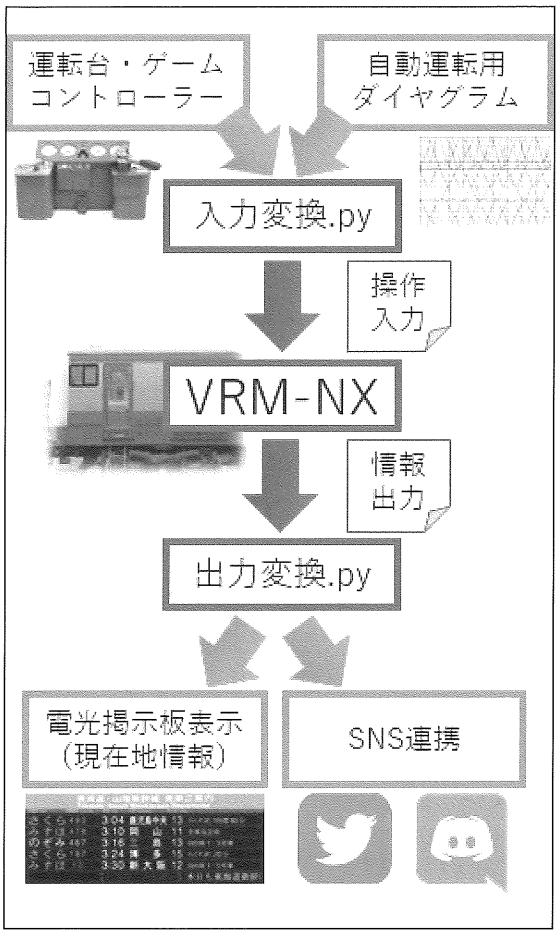

図4-11-4　VRM-NXと「外」との連携例

このように「外」とつながることができる「VRM-NX」は、単なる“鉄道模型のシミュレーター“にとどまらない「Pythonプラットフォーム」として、「いろいろなサービスと連携できる無限の可能性をもつ」と言えるでしょう。

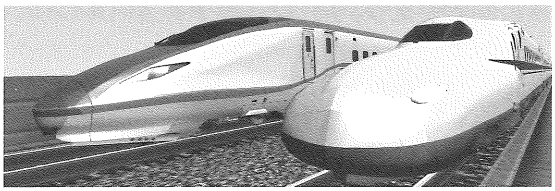

図4-11-5　PythonでVRMの世界が広がる