独自のシーンを作る (1)

蒼竜 (@soryu_rpmakermv)*

本稿 (RPG ツクールプログラミング系統) では，ある程度の基礎的なプログラミング (言語や技量の高低は問わない) にいて知識のある者を対象に，特定の処理を RPG ツクール上で自作するためのプログラミングの基礎事項をまとめるものである．Javascript の慣習や，RPG ツクールの細かい仕様に関わる事項はなるべく排して，読者がゲーム一般における必要な機能を自作する簡便な足掛かりになることを目指す．

1　はじめに

RPG ツクールでオリジナルの機能を追加しようとするならば，コアスクリプトを熟読して全体を理解しなければならないと考えてしまうかもしれない．それはある意味では正しいが，全くそうであるというわけでもない．

確かに，既存の機能に機能を付け加えたり，修正したりするという場合には膨大で複雑なゲームエンジンのコアスクリプトを熟知しておかなければなかなか手を出せない．一方，自分のオリジナルのシーンやウィンドウを作成して表示する，という点に絞ればコアスクリプトに精通していなくとも基本的なコードの構造が分かればツクールプログラミングを始めることは可能である．

ここではオリジナルシーンの作成を目標として，まずは一般的なゲームにおける最小限の処理の構成を考えてから，それらをどのようにすれば RPG ツクールで表現できるかに着目し，オリジナルシーンへのウィンドウや文字の基本的な描画方法を探る．

2　シーンの構造

2.1　ゲームの「シーン」とは

RPG ツクールでは，マップ画面，メニュー (装備やスキル)，戦闘といったゲーム内のあらゆる場面をシーン (Scene) と呼ぶ．ボタンの入力を受け付けたり，文字や数字を画面に表示したり，といった様々な処理はこのシーンごとに管理されていて，ゲームの状態に応じてこのシーンが移り変わっていくことでゲームが進んでいく．これは RPG ツクールに限ったことではなく，一般のゲームプログラミングにおいても類似な形式でゲームの管理が行われることが多いと類推される．

2.2　一般的なゲームのシーンの基本構造

ここで独自に新しいシーンを作成するために，ゲームにおける一般的な 1 場面でどのような処理が行われているかを考える．図 1 がその最も基本的な骨組みを表したものである．特に重要と思われる部分には赤で着色をしている．

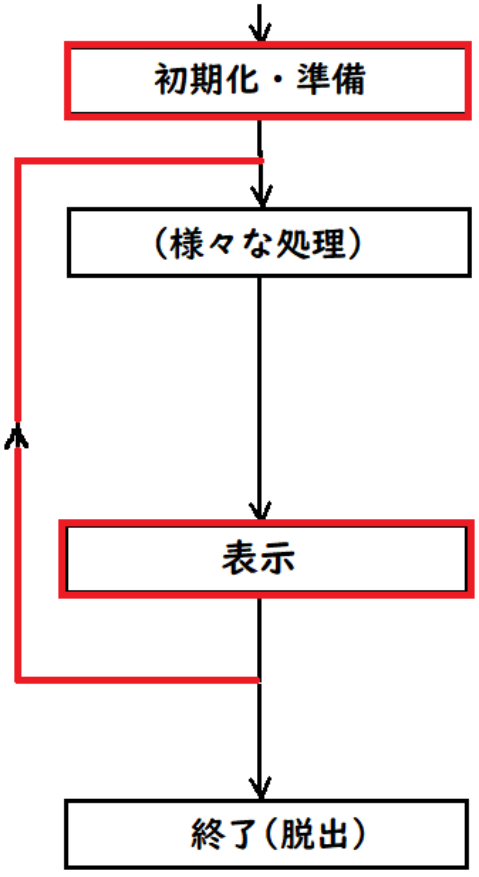

図 1: ゲーム中のある 1 場面の処理の基本構造

*http://dragonflare.dip.jp/dcave/

まず，シーンが始まるにあたって必要な処理は初期化である．これはシーンの処理の最初に 1 度呼ばれるもので,「どんな項目を表示するのか」,「シーンで使う変数の初期値はいくつか」を設定する段階である．これがゲーム中のステータス画面であるならば,「誰のステータス画面なのか」といった情報を準備する処理が含まれるだろう．

それが終わると，シーンの処理が始まる．図中には様々な処理とあるが，その中でも重要なものが画面への「表示」となるだろう．これが無ければ画面には何も表示されない．そして最も重要なことは，この処理は**繰り返し実行される** (図中では赤線が示している) ということである．画面は常に何らかの処理を行って更新がなされており，ある条件を満たしてシーンが終了するまで決して処理が止まることはない．

ゲームシーンの基本構造

- 初期化
- 表示などの処理（行いたい命令）
- 繰り返し

ここまでをまとめると，ゲーム中の処理に必要な最小限の構成は上のようになる．この 3 点を RPG ツクールでのプログラミング手法に対応させて，基本的な自作シーンの作成を行っていく．

3　RPG ツクールでのシーンの基本構造

前節までの内容をふまえて，RPG ツクール (MV 以降) でのオリジナルシーンの作成のための基本実装を検討する．テストコードを用いて実験を行うため，空の新規プロジェクトを準備することを推奨する．

3.1　最小構成コードのテスト

いきなりであるが，RPG ツクールでのオリジナルシーンを作成するためのコードの基本形をソースコード 1 に示した．js ファイル本体はサーバー上[1] にある．

これは前節で考えたゲームシーンの基本構造に対応する，RPG ツクールのゲーム中のシーンを記述する最小構成のコードとなる．説明のために，わざと `console.log()` を仕込んでコンソールへの出力を試みている．

[1] `http://dragonflare.dip.jp/dcave/articles/SoR_sm.js`

ソースコード 1: 'SoR_sm.js'

```
1  function Scene_Test(){
2   console.log("call")
3   this.initialize.apply(this, arguments);
4  }
5
6  Scene_Test.prototype = Object.create(Scene_Base.
       prototype);
7  Scene_Test.prototype.constructor = Scene_Test;
8
9  Scene_Test.prototype.initialize = function() {
10  console.log("initialize");
11  Scene_Base.prototype.initialize.call(this);
12 }
13
14 Scene_Test.prototype.create = function() {
15  console.log("create");
16  Scene_Base.prototype.create.call(this);
17  this.createWindowLayer();
18 }
19
20 Scene_Test.prototype.start = function() {
21  console.log("start");
22  Scene_Base.prototype.start.call(this);
23 }
24
25 Scene_Test.prototype.update = function() {
26  console.log("update");
27  Scene_Base.prototype.update.call(this);
28 }
29
30 //////////////////////////////////////
31 Scene_Boot.prototype.start = function() {
32  DataManager.setupNewGame();
33  SceneManager.goto(Scene_Test);
34 }
```

実験のために，ゲームを起動しようとすると直ちにオリジナルシーンが呼び出されるようにしている．そこでさっそくであるが問題である．

問題 1 これを実行した時に，コンソール (ゲーム起動中に F8 または F12 を押す) には何が表示されるか？

ヒント： console.log() の" " の中に書かれた文字列が表示されるのだが，果たしてそれらがどのように (順番，回数) で出力されるだろうか．

問題 1 の答え合わせとして，図 2 にこのコードの実行結果を示した．

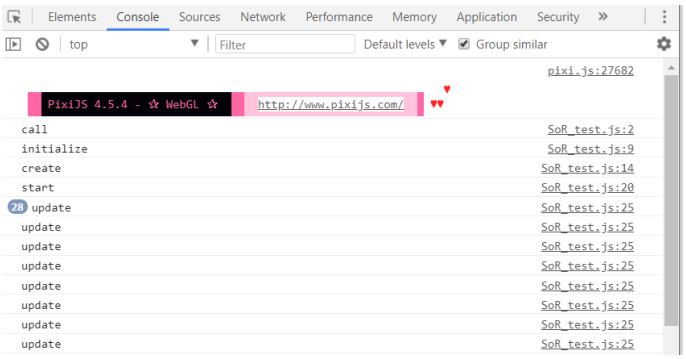

図 2: SoR_test.js の実行結果 (コンソール画面)

結果を見てみると，call, initialize, create, start が順に 1 回ずつ呼ばれ，その後に update が繰り返し出力されている．ちなみに，本来のゲーム画面はどうなっているのかというと図 3 のようになっている．画面に何かを表示するコードは一切記述していないので，当然こちらは真っ黒な何もない画面である．

図 3: SoR_test.js の実行結果 (ゲーム画面)

話をコンソール上での出力結果に戻して，なぜこうなるのかを理解することが，オリジナルシーンを自分で作成するために必要なことであり，ここで少しコアスクリプトの基本を理解する必要が生じる．

これは，実験のために作成したシーン Scene_Test が，RPG ツクールのあらゆるシーンを管理する基本のシーンである Scene_Base を元にして作成されていることに起因する．もっと言えば，Scene_Base がこのような挙動になるように記述されているからである．

ゆえに，この部分を理解しておくことが，あらゆるオリジナルシーンの自作のための出発点ということになる．

3.2　基本構造の詳細

問題 1 の結果を踏まえて，さきほどのコードの重要な点を確認していこう．ソースコード 1 から，シーンの骨組みだけを抜き出したものがソースコード 2 である．オリジナルのシーンに必要な関数は高々，initialize, create, start, update の 4 つであり，これは基にしているコアスクリプト中の Scene_Base の方で，シーン起動時にこれらの名前で定義された関数が呼び出されるように記述されている．

ソースコード 2: 'SoR_sm.js の骨組み'

```
1  function Scene_Test(){
2   this.initialize.apply(this, arguments);
3  }
4
5  Scene_Test.prototype = Object.create(Scene_Base.
       prototype);
6  Scene_Test.prototype.constructor = Scene_Test;
7
8  Scene_Test.prototype.initialize = function() {
9   Scene_Base.prototype.initialize.call(this);
10 }
11
12 Scene_Test.prototype.create = function() {
13  Scene_Base.prototype.create.call(this);
14  this.createWindowLayer();
15 }
16
17 Scene_Test.prototype.start = function() {
18  Scene_Base.prototype.start.call(this);
19 }
20
21 Scene_Test.prototype.update = function() {
22  Scene_Base.prototype.update.call(this);
23 }
```

initialize, create, start は**問題 1** によれば順に 1 回ずつ呼び出されていることから，それぞれが基本構造の「初期化」の部分にあたると考えられる．一方，これらの後に update が繰り返し呼ばれていることから，update が基本構造の「繰り返し」の部分を担っていると考えられる．これらを踏まえてそれぞれをより細かく見ていく．

3.3　基本構造：シーンの立ち上げ

3.3.1　初期化：initialize

ソースコード 2 の 1 行目から 10 行目までが，シーンの立ち上げに相当する部分である．他からこの Scene_Test が呼び出されると，1-3 行目の関数が呼ばれ，そこから

Scene_Test.prototype.initialize が呼び出されるようになっている．5 行目が，「Scene_Base の構造を引き継いだシーンを作成する」ことを意味している．

この辺りについてはオブジェクト指向と呼ばれるプログラミングの概念の理解が必要になるが，本稿は根本的にプログラミングの教育を目的としているわけではないので，それらについては省略する (ひとまず本稿で目指す目的のためには，取り急ぎ必要ではない)．

ところでこの **initialize** だが，日本語では「**初期化する**」という意味である．つまり，この initialize でシーンに必要な初期設定を行うということになる．具体的にはコアスクリプトの各シーンの initialize を参照してみるとよい．

3.3.2　初期化：create

RPG ツクールでは，初期化にあと 2 つの工程が定義されている．その 1 つが，12-15 行目にある create である．create は initalize の処理が終わった後に 1 回呼び出されている．RPG ツクールのコアスクリプトによれば，create では画面に表示したい文字や描画したい画像を管理するための空間を作成する場所とされている．

ソースコード 2 を見ると，create の中では `this.createWindowLayer();` というものが呼び出されている．これは Scene_Base で元々定義されている関数で，画面にウィンドウや文字を描画するために必要なものである．ここでの説明はややこしくなるため，詳細は後ほど示す．

3.3.3　初期化：start

初期化のもう 1 つの工程に，17-19 行目に現れる start がある．これは RPG ツクールでシーンを起動する際に，あらゆる初期化や準備が終わって，実際にシーンが始まる直前に 1 度呼ばれるものとなっている．つまり create よりも後に呼び出される．

シーンを開始するまさに直前の最期の初期化処理としてこの start がある．この時点での詳細な説明は避けるが，文字や画像などの画面への表示順序などの関係から明示的に初期化が複数にステージ分けされているものと考えられる．

3.4　基本構造：繰り返し行われる処理

一連の初期化処理が終わると，いよいよシーンがゲーム画面に描画され始め様々な処理が行われる．ゲーム画面は，刻々と進んでいくゲームの状況に応じて表示内容を変更しなければならないため，常に状況を見張るために繰り返し行われる処理がある．その部分が 21-23 行目の update となる．ソースコード上では実質的に何もしていないことになっているが，ここに自分で実現したい様々な処理を書き加えていくことが基本となる．

また，このシーンを抜けない限り，update は非明示的に繰り返し実行される関数であるため，現実的にはここに「ある条件を満たしたらシーンを抜ける」という処理も作らなくてはならない．

> **問題 2**「基本構造」として挙げた 4 つの関数を書かなかったらどうなってしまうか？
>
> **ヒント：** initialize, create, start, update は，Scene_Base で予め定義されている関数である．

4　自作シーン中で画面に文字を表示する

さて，このままでは基本シーンとしてあまりに寂しい．そこで基本例として図 4 のように画面にウィンドウを作成し，その中へ文字を書き込んで表示するというコードを作成してみる．これはソースコード 2 に必要な処理を書き込むだけで簡単に実現できる．

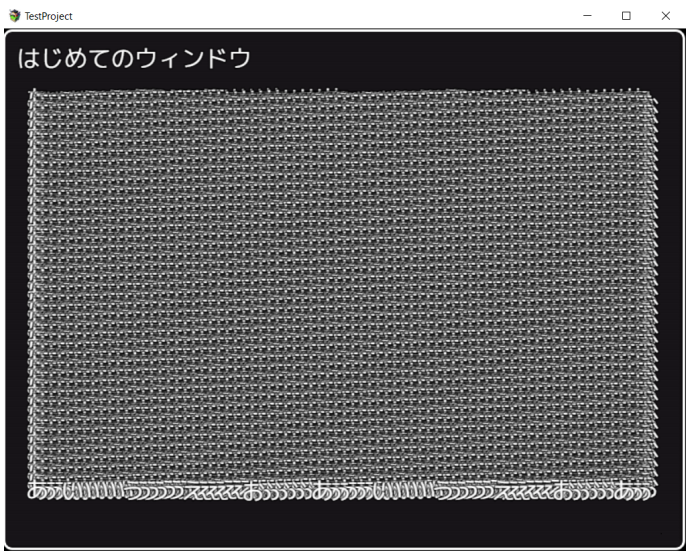

図 4: 文字を書き込んだウィンドウを描画した画面

図 4 のように表示するためのソースコード 3 に示した．同様に，js ファイル本体はサーバー上[2] にある．

[2] `http://dragonflare.dip.jp/dcave/articles/SoR_sm2.js`

ソースコード 3: 'SoR_sm2.js'

```
1  function Scene_Test(){
2    this.initialize.apply(this, arguments);
3  }
4
5  Scene_Test.prototype = Object.create(Scene_Base.
       prototype);
6  Scene_Test.prototype.constructor = Scene_Test;
7
8  Scene_Test.prototype.initialize = function() {
9    Scene_Base.prototype.initialize.call(this);
10 }
11
12 Scene_Test.prototype.create = function() {
13  Scene_Base.prototype.create.call(this);
14  this.createWindowLayer(); /////<- initialization
15  this.createTestWindow();
16 }
17
18 Scene_Test.prototype.start = function() {
19  Scene_Base.prototype.start.call(this);
20  this.DrawGraffiti();
21 }
22
23 Scene_Test.prototype.update = function() {
24  Scene_Base.prototype.update.call(this);
25 }
26
27 // main window
28 Scene_Test.prototype.createTestWindow = function
       (){
29  this._window = new Window_Base(0, 0, Graphics.
        width, Graphics.height);
30  this._window.setBackgroundType(0);
31  this.addWindow(this._window);
32 }
33
34 //for draw
35 Scene_Test.prototype.DrawGraffiti = function() {
36  this._window.drawText('はじめてのウィンドウ',
       0, 0, this._window.width);
37
38  const ar=['あ','い','う','え','お'];
39  for(let i=0; i<10000; i++){
40   this._window.drawText(ar[i%5], 10+72*i%750,
        48+i, this._window.width);
41  }
42 }
43
44 ////////////////////////////////////////
45 Scene_Boot.prototype.start = function() {
46  DataManager.setupNewGame();
47  SceneManager.goto(Scene_Test);
48 }
```

このコードを用いて，オリジナルシーンにウィンドウを新しく作成し，そこへ文字を書き込んでみよう．

4.1　ウィンドウレイヤーの定義

目標としては，自作のシーンに文字を表示させたいのだが，シーンそのものへ直接書き込むことはできない．シーンとはゲーム中のある場面の全体を管理するものとして作られているため，表示したいものがあれば，**表示するための何かに貼り付けてから**シーンに置くという形をとる．

ところが，ウィンドウを画面 (シーン) に配置するためには，予め 14 行目の create 関数内にある `this.createWindowLayer();` という呼び出しを忘れずに行わなければならない．これは，RPG ツクールにおいてシーン全体にわたるウィンドウの表示方式 (実装) によるものであり，画面にウィンドウを表示するための管理を一括しているものである．これを忘れてしまうと，いくら正しくウィンドウ作成のコードを書いても図 5 のように実行ができないため，**コアスクリプトの詳細までもを理解せずとも，これだけは忘れないで呼び出すように**しなければならない．

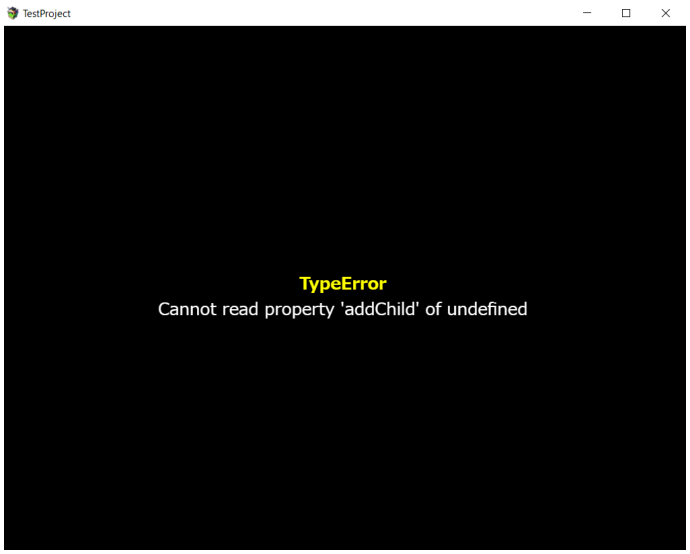

図 5: `this.createWindowLayer();` をコードから抜き去った時の実行結果 (エラー出力)

4.2　新規ウィンドウの作成

新しいウィンドウを作成して，そこに文字を書き込んだものをシーンに置くという方法でやってみよう．ソースコード 3 の 28-32 行目には `createTestWindow()` という関数が新たにあり，そこにウィンドウを作成するための記述を置いた．これは 15 行目を見てみると，create の中で呼び出されていることが分かる．つまり，初期化処理の一部であることが分かる．

create や update のようにコアスクリプトで定義されているものは，**他からの再定義によって継ぎ足しが起こりやすい**．よって，コードの見栄えの観点から直接コードを書き込まず，自分で適当に作った関数の中に新しい処理を書いた方がよい．

改めて createTestWindow() を見てみると，さっそくウィンドウの作成を行う部分が現れる．

基本的な新規ウィンドウの作成

```
this._window = new Window_Base(X 座標，Y 座標，横幅，縦幅);
this.addWindow(this._window);
```

作りたいウィンドウの大きさや場所を指定して，new Window_Base とすることで，単純な何もないウィンドウが作成される．作成したウィンドウを，オリジナルのシーン (Scene_Test) で使うために this._window へ代入している．この時点で，this._window は **Scene_Test が持っているウィンドウ**となる．

これだけではまだウィンドウは表示されない．作成したウィンドウを表示するためには，それを明示的に配置する必要がある．続く this.addWindow(this._window); を実行することで，作成中のシーン Scene_Test に this._window が置かれることになり，図 6 のように画面にウィンドウが出現する．

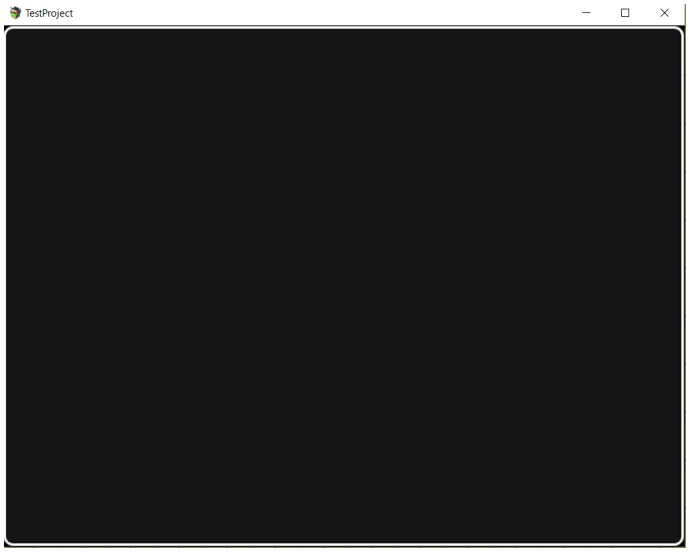

図 6: 空のウィンドウを描画したゲーム画面

30 行目にある this._window.setBackgroundType(0); は，ウィンドウを表示する時のオプションの設定で，RPG ツクールのエディタでの通常，暗くする，透明に対応する．敢えて明示的に示したが，書かなくても通常のウィンドウとして表示される．

4.3　作成したウィンドウ内に文字を書き込む

それでは，作成したウィンドウに文字を書き込んでみよう．ここではシーン起動時に 1 回だけ任意の文字列を書き込むということにする．

ソースコード 3 の例では，シーン開始の直前に実行される start 内にウィンドウへ文字を描画する処理を記述した．先ほどと同様に，start では DrawGraffiti() という関数の呼び出しだけを行い，35-42 行目において，DrawGraffiti() が行う処理として文字列の描画を記述している．文字列の描画は，**作成したウィンドウに対して** drawText() という関数を実行することで行う．

ウィンドウへの文字の描画

```
this._window.drawText('描画したい文字列', X 座標，Y 座標，文字列の最大描画幅);
```

this._window というのが，先ほどテスト用シーン Scene_Test に作成したウィンドウであり，そこへ文字列を描画するということで this._window.drawText となる．これを使えば，図 4 で示したようにこのウィンドウに対して文字列を自由に描画することが可能である．

5　練習問題

5.1　複数のウィンドウ

2 つ以上のウィンドウを作成し，それらを互いに重なり合わないようにシーン上に配置したコードへ変更せよ．

5.2　表示する文字を更新する

this._window に描画された文字等は，

```
this._window.contents.clear();
```

を実行することで消去できる．

Scene_Test.prototype.update を中心に書き加えて，一定間隔おき (時間でなくてよい) に異なる文字列が描画されるコードへ変更せよ．

ヒント：「一定間隔」が分かるように数をどこかで数えておかなければ…