独自のシーンを作る (2) - 画像の表示と描画順序

蒼竜 (@soryu_rpmakermv)*

前回は，ゲームの各シーンを構成する基本要素に注目し，それに基づいて RPG ツクールのゲームエンジン上で動作させるオリジナルのシーンの最小構成について検討した．現在の時点で，自作シーンに新しいウィンドウや文字を表示するところまでできているだろう．今回は，まず画像の表示について検討し，さらにシーンの終了処理を取り込むことでオリジナルシーン作成の基礎を完成させる．

1　ゲーム画面への画像の表示

前回，ウィンドウの作成とシーンへの配置方法について見たが，ここではそれと同等に重要である，画像ファイルを読み込んでシーンへ配置する方法について見ていく．ソースコード 1(`SoR_sm3.js`) は，前回のウィンドウを表示するコード `SoR_sm2.js` を元に画像ファイルを 1 つ読み込んで画面へ表示させるように追加したものである．js ファイル本体はサーバー上[1)]に置いた．

ソースコード 1: 'SoR_sm3.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
16  this.createTestWindow();
17  this.createSprites();
18 }
19
20 Scene_Test.prototype.start = function() {
21  Scene_Base.prototype.start.call(this);
22 }
23
24 Scene_Test.prototype.update = function() {
25  Scene_Base.prototype.update.call(this);
26 }
27
28 // main window
29 Scene_Test.prototype.createTestWindow = function
     (){
30   this._window = new Window_Base(Graphics.
        width/2-120, Graphics.height/2-120,
        240,240);
31   this._window.setBackgroundType(0);
32   this.addWindow(this._window);
33 }
34
35 Scene_Test.prototype.createSprites = function() {
36   const img = ImageManager.loadSystem("
       GameOver");
37   this._picture = new Sprite(img);
38   this.addChild(this._picture);
39 }
40
41
42 ////////////////////////////////////////////
43 Scene_Boot.prototype.start = function() {
44   DataManager.setupNewGame();
45   SceneManager.goto(Scene_Test);
46 }
```

画像の表示方法も書き方は違えど，本質的にはウィンドウの表示方法と同じで，

1. ファイルを指定して画像を読み込む
2. 読み込んだものを使って Sprite オブジェクトを作成 (シーンに配置できる形にする)
3. シーンに配置する

のような手順を踏む．具体的にはソースコード中の 35 行目の createSprite 関数が該当し，create(初期化処理) の段階で次のような処理を 1 回行っている．

基本的な画像ファイルの表示

```
const img = ImageManager.loadSystem("読み込むファイル名 (拡張子不要)");
this._picture = new Sprite(img);
this.addChild(this._picture);
```

始めの `ImageManager.loadSystem` がファイルを読み込むための関数であるが，これは `loadSystem` という名

*http://dragonflare.dip.jp/dcave/
1) `http://dragonflare.dip.jp/dcave/articles/SoR_sm3.js`

が示すようにRPGツクールのimgフォルダ下のSystemフォルダの中にあるファイルを読み込むというものになっている．loadの後に続く文字列を変えると別のフォルダを参照できる．例えば，`loadPicture`とすればPicturesフォルダの中から読み込むようになる．様々なファイルを読み込むための関数の名前はコアスクリプトを参照されたい．

> **問題 1** コアスクリプトの`xxx_managers.js`の中を開いて，`ImageManager`で始まる関数を探せばRPGツクールの標準で設定されている様々な画像フォルダの中を参照する関数が定義されていることが分かる．
> 他にどのような関数が利用できるかを調査せよ．(xxx部分はMV/MZで異なる)

RPGツクールのゲームエンジンにおいては，読み込んだ画像をそのまま画面に配置することはできない．実際の画像を画面上に表示する手続きは，**スプライト (Sprite)** と呼ばれるシーンに配置するための透明な物体を介して行われる．つまり図1のように，読み込んだ画像を一旦スプライトに貼り付けておいて，そのスプライトを画面に配置するという形態をとることになる．

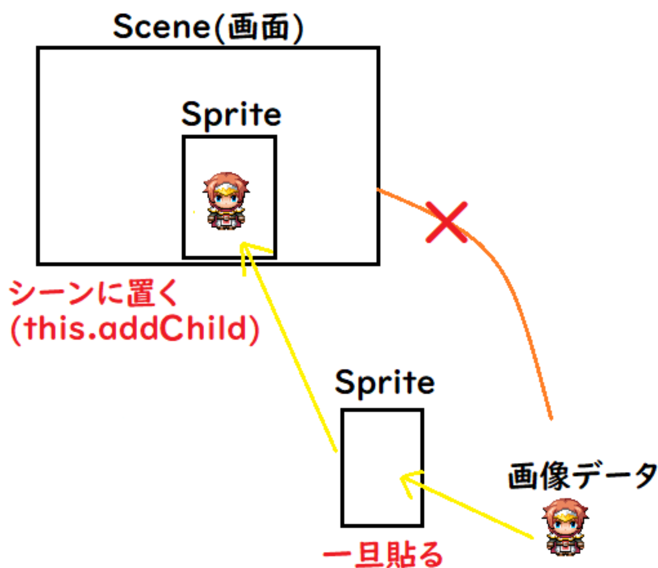

図 1: Spriteを介したシーンへの画像配置

ソースコード上では，`this._picture`という変数にスプライトの情報が保持されて,「このシーン (Scene_Test) で扱うスプライト」というものになる．そして，そのスプライトを現在のシーンに`addChild`することによって，画面上に読み込んだ画像が表示される．

図2がこのソースコードの実行結果となる．読み込んだファイルはGameOverつまりゲームオーバー画面で表示される画像だったため，ゲーム起動して早々ゲームオーバーになるというお粗末な結果である．

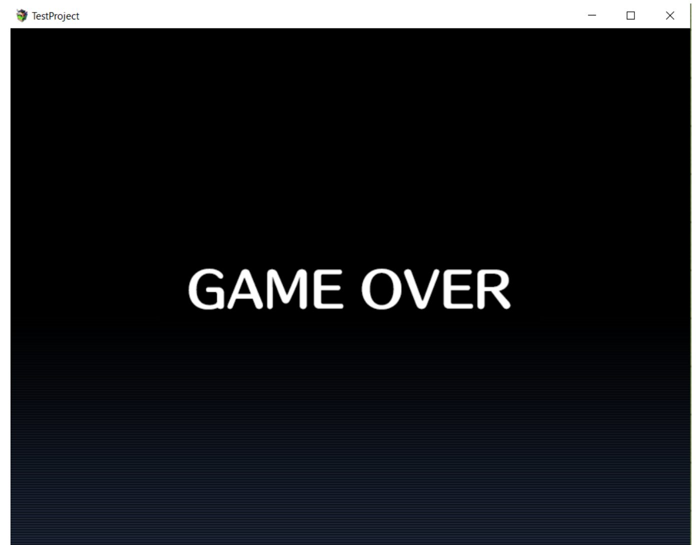

図 2: SoR_sm3.js の実行結果 (画像の表示)

これでひとまず画像を読み込んで表示する方法について分かったわけだが，よくよくソースコードを見てみると，前回作成したウィンドウの作成・表示を行った関数`createTestWindow`がそのまま残っているのだが，図2を見る限りウィンドウが見当たらない．

> **問題 2** ウィンドウはどこへ行ってしまったのだろうか？ 例えば，画像を表示したことで消えてしまったのだろうか？

この問題を解決するためには，画像やウィンドウの表示方法についてもっと細かく知る必要がある．RPGツクールのゲームエンジンにおいて，表示される画像やウィンドウがどのように管理されているのか次節で詳しく見てみよう．

2 スプライト・ウィンドウの管理

ここまでで，画像 (スプライト) やウィンドウをゲーム画面上に表示する基本的な方法は分かった．ウィンドウを表示する時にはウィンドウオブジェクトを生成してからaddWindowを，画像を表示する時は画像を渡したスプライトオブジェクトを生成してからaddChildを現

在のシーンに対して実行することで比較的容易に実現できた．

しかし，表示されるスプライトやウィンドウがゲームエンジン上でどのように管理されているのかをもう少し知っておくことは実用的なオリジナルシーンを自作していく過程で避けては通れないであろう．作成した `SoR_sm3.js` を使って，シーンに配置されたスプライト・ウィンドウがどうなっているのかを見ていこう．

2.1　シーンに配置されたスプライト・ウィンドウの管理

まずは，`SoR_sm3.js` を少しだけ修正して配置されたスプライト・ウィンドウがどう管理されているかを追いかけてみる．次のソースコード 2 のように，`SoR_sm3.js` の start 関数の最後に `console.log(this);` を追加する．こうすることによって，this つまり現在のシーン `Scene_Test`(のオブジェクト) の情報がコンソールに書き出される．

ソースコード 2: 'SoR_sm3-2.js'

```
Scene_Test.prototype.start = function() {
  Scene_Base.prototype.start.call(this);
  console.log(this); //追加
}
```

これを追加したコードで実行した結果を表したものが図 3 である．確かに `Scene_Test` と最初に表示され，シーンを構成する膨大な情報が列挙されていることが分かる．

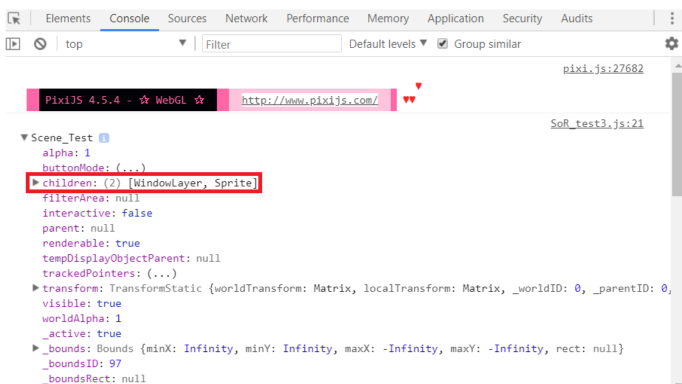

図 3: コンソール出力した Scene_Test の情報

ここで注目するべきは，図中に赤枠で囲った `chlidren` という部分である．この `chlidren` に，addChild や addWindow で追加した要素が含まれている．`SoR_sm3.js` ではスプライトとウィンドウを 1 つずつシーンに追加したため，合計で 2 つの要素が存在することは理にかなっている．また，この `chlidren` のタブをクリックすると，図 4 のようにさらに詳細な情報も確認できる．

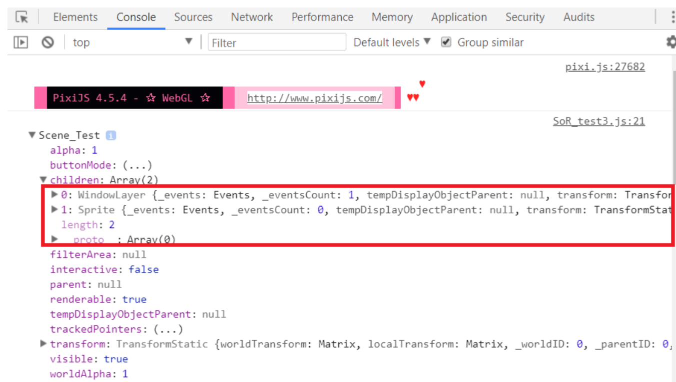

図 4: コンソール出力した Scene_Test の情報 2

ここで重要なことは，`chlidren` とは図 5 のように登録された要素を順に末尾へ格納して管理する配列になっており，この配列の**先頭にある要素から順に描画処理がなされていく**ことである．今，0 番に WindowLayer(ウィンドウ) があり，1 番に Sprite(画像) が登録されていることから，ウィンドウを表示した後に，画像を表示するという処理が行われていることが分かる．結果として**問題 2** に対する解答は，**先に描画されていたはずのウィンドウは，後から描画されたゲームオーバー画像に覆い隠されてしまっていた**ということになる (ウィンドウの描画自体は確かに行われている)．

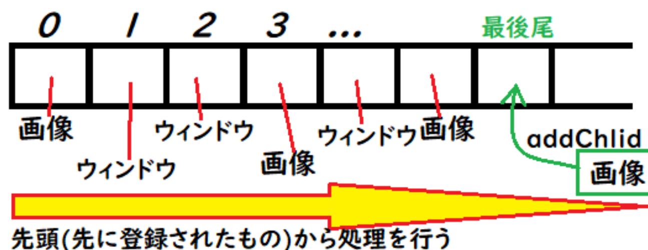

図 5: シーンに配置されたウィンドウ・スプライトを管理する配列 (`children`) の模式図

今 `SoR_sm3.js` を見返してみると，create 関数内で，`this.createTestWindow();` を呼び出した後に `this.createSprites();` を呼び出している．即ち，ウィンドウ，スプライトの順でシーンに addChild(Window) が行われていることになっている．この両者の呼び出す順番を入れ替えてしまえば，シーンに配置する順番が入れ替わるので,「画像の上にウィンドウを表示する」ことができるだろうと予想される．

ところが実はそうはいかない．入れ替え後の実行結果の画面を図 6 であるが，この 2 つの関数呼び出しを入れ

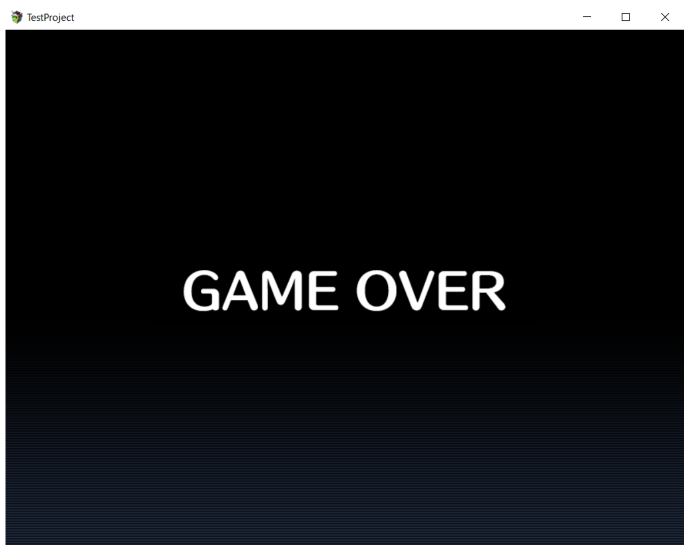

図 6: ウィンドウ・スプライトの呼び出し順序を入れ替えた SoR_sm3.js の実行結果

替えたとしても，その結果は図 2 と全く同じで，ゲームオーバー画像にウィンドウが覆い隠された状態になったままである．

前回，オリジナルシーンにウィンドウを作成・描画するための注意として，「**忘れずに this.createWindowLayer(); を呼び出しておかなければならない**」というくだりがあった．これは，Scene_Test を作る際の土台として継承しておいた Scene_Base で定義されているものとして内容については詳しく触れなかったが，実はこの this.createWindowLayer(); を読み解くことで画面への描画順序を制御する技法が分かる．

2.2　Scene_Base にあったウィンドウレイヤー

次に示すソースコード 3 は，コアスクリプトの Scene_Base.prototype.createWindowLayer と，前回よりウィンドウを描画するために用いた Scene_Base.prototype.addWindow 部分のみを抜粋したものである．

まず，addWindow の方を見てみると実はスプライトをシーンに配置する際に用いた addChild を呼び出しているだけだということが分かる．しかし，現在のシーン (this) に直接配置しているのではなく，_windowLayer というシーン下にある別の空間へ割り当てているということが分かる．それを踏まえて，createWindowLayer の方を見てみよう．

ソースコード 3: createWindowLayer と addWindow

```
1  Scene_Base.prototype.createWindowLayer =
       function() {
2   var width = Graphics.boxWidth;
3   var height = Graphics.boxHeight;
4   var x = (Graphics.width - width) / 2;
5   var y = (Graphics.height - height) / 2;
6   this._windowLayer = new WindowLayer();
7   this._windowLayer.move(x, y, width, height);
8   this.addChild(this._windowLayer);
9  };
10
11 Scene_Base.prototype.addWindow = function(
       window) {
12     this._windowLayer.addChild(window);
13 };
```

シーンへの描画順序を操るトリックとして重要な箇所は 6 行目と 8 行目にある．実は，RPG ツクールのゲームエンジンではウィンドウを管理するための専用の空間が_windowLayer として定義されている．この_windowLayer をシーンに配置することで，addWindow によって_windowLayer に割り当てられたウィンドウたちが_windowLayer の管理の下でシーン上に描画される．

先に見た図 3 において，children の先頭 (0 番目の) 要素に WindowLayer とあったのはこのためである．実際には，Scene_Test の children としてウィンドウが置かれているわけではなくて，_windowLayer の children としてウィンドウが置かれているということになる．話が複雑になってきたところで，図 5 に示した模式図を修正して，状況を整理しておこう．

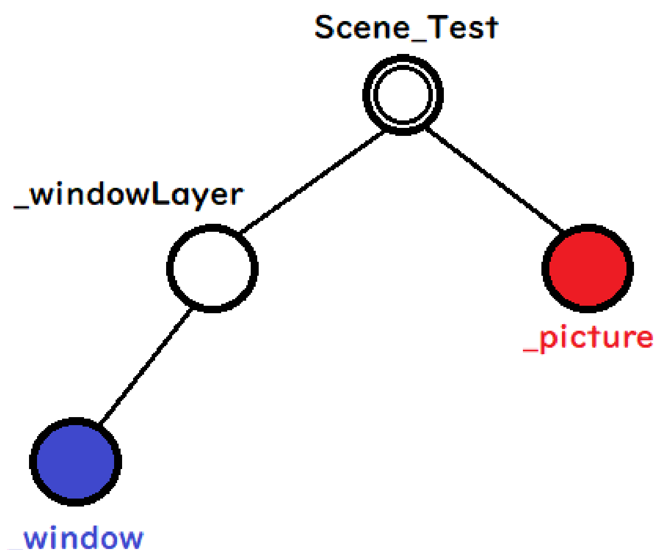

図 7: シーンに配置されたウィンドウ・スプライトを管理する木構造の模式図

「登録されたものから処理を行う」というのは正しいのだが，その管理構造の本質は図 5 のような線形 (一直

線状) の簡単なものではない．正確には木構造 (Tree) となっており，現在の Scene_Test の状況は図 7 のような形状で表される．

一般的な木構造の読み取りは後で述べるとして，先にこの図に限った話をしておこう．木のてっぺん (**根 (root)** という) にあたる Scene_Test から見て左側へまず下っていくように木を辿ればよい．左側へ最下層まで辿ると，_window があるためにこれをまずシーンに描画する．_window より下には何もないため，一旦木をさかのぼる．Scene_Test から見て右側の_picture はまだ辿っていないので辿る．そして_picture をシーンに描画する．

シーンに新たに割り当てられた要素は，木の左側から右側へ順に吊るされていくと考えればよい．_windowLayer は，初期化時点の create 関数で最初に this.createWindowLayer(); を呼び出して生成されているので，Scene_Test) の直下，左側へ吊るされている．その後，addChild でシーンにスプライトを割り当てているため，_picture が Scene_Test) の直下，右側へ吊るされている．

ここで，_window は，ソースコード 3 の 6-8 行目によれば，addWindow を使うことによって_windowLayer に割り当てられていたことを思い出そう．すると，ここで，_window は否応なく_windowLayer の直下に吊り下げられる．前節で検証し失敗したように，**自作したウィンドウとスプライトの登録の順番を入れ替えを行ったところでそれらの描画順序を入れ変えることはできない**ということが分かる．

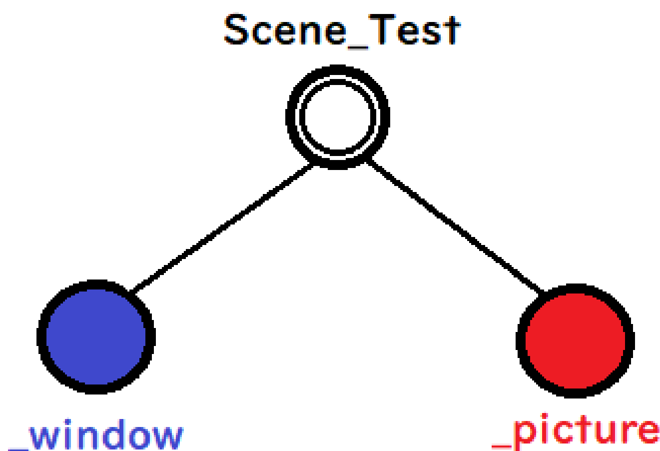

図 8: addWindow を用いずにシーンにスプライト・ウィンドウを登録した場合の木構造

ウィンドウをシーンに配置する際，addWindow ではなくて直接 addChild としてしまえば，図 8 のようなウィンドウをシーンの直下に管理した状態で描画することができる．そうすれば，図 5 で見たような線形にデータが並んだ配列と全く同じ状態になる．したがって，前節で検証した描画関数の呼び出し順序を入れ替えることによって，スプライトとウィンドウの描画順序を入れ替えることができるだろう．

しかしながらその方法だけに慣れてしまうと，他と干渉のない完全なオリジナルシーンならば問題は無いが，**既存のコアスクリプトで定義されたシーン (戦闘やステータス画面など) を改変する場合**に不都合となる．

そこで，次に画像の描画管理についても少しコアスクリプトを眺めてみたい．だがゲームエンジンの本格的なデータ管理構造を眺める前に，一般的な木構造の基本的な事項を確認しておくことにする．

2.3 木構造

2.3.1 木構造の基本

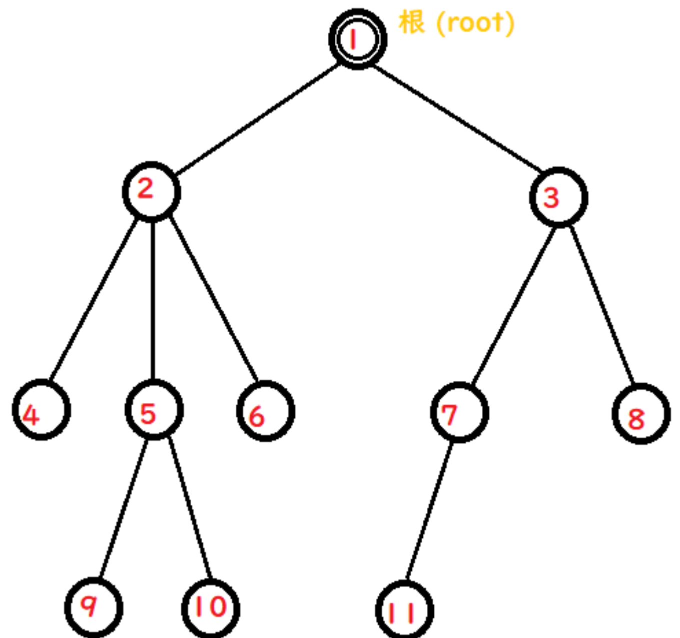

図 9: 一般的な木構造

図 9 のような，複数の**頂点** (節，またはノード) が親子関係を持つ構造を**木構造** (ツリー構造) という．コンピュータ上でデータを管理するための構造の 1 つとして計算機科学で扱われることが多い．主な用語を 1 つ 1 つ確認していくと，家系図を真似ているかのような印象を受ける．

木は上から下へ向かって参照するものとする．ある頂点 v(図中の頂点のどれでもよい) から見て，その上部に接続している頂点を v の**親** (parent)，下部に接続してい

る頂点を v の**子** (child) であるという．親を持たない頂点を**根 (root) と**，子を持たない頂点を**葉 (leaf)** という．

また，ある頂点 v に対して，根からの距離を v の**高さ** (または，深さ) という．特に，全ての頂点の高さの最大値を**木の高さ**と言う．

例題 図 9 の木において，
根 ... 1
頂点 2 の親 ... 1　　頂点 2 の子 ... 4,5,6
葉 ... 4, 6, 8, 9, 10, 11
頂点 2 の高さ ... 1　　頂点 8 の高さ ... 2
木の高さ ... 3

RPG ツクールのコアスクリプトで見かける `xxx.addChild(v)` とは，`xxx` を親ノードと見て子ノード `v` を繋ぐという意味である．

Web 上のページも，内部的にはいくつかの描画領域に分かれていて，そのうちのどれかを親として見て，追加 (描画) したい画像やコンテンツを子として吊り下げるようになっている．

2.3.2　深さ優先探索 (Depth First Search)

木構造となっている全てのデータを探索する最も単純な方法の 1 つとして，深さ優先探索 (DFS) が挙げられる．

このアルゴリズムは，木の根から出発点として，**葉に辿り着くことを最優先に探索して**全ての木の頂点を探索するものである．処理の概要は，疑似コード 1 に示す．

疑似コード 1 木構造上の深さ優先探索

```
s ⇐ 根となる頂点
while まだ探索していない頂点がある do
  if s に子ノードが存在して　かつ 未探索である then
    s の子ノードへ辿る
    s ⇐ 参照中の頂点
    s を探索済みとする
  else
    s の親ノードへ戻る (s ⇐ 親ノード)
  end if
end while
```

実際に図 9 の木構造に対して DFS を適用し，木に含まれる全ての頂点を探索してみる．探索は木の根から始まるため，$s = 1$ となる．ここで頂点 1 の子を調べると，頂点 2 と 3 がある．ここでは便宜上，候補となる頂点が複数存在する場合は**番号の小さいほう (左側にあるもの) から優先的に探索する**，とルールの取り決めをしておこう．

すると，頂点 2 をまず参照する ($s = 2$)．頂点 2 の子は頂点 4,5,6 だから次は頂点 4 を参照することになる ($s = 4$)．ところが，頂点 4 に子ノードが存在しないため，一旦頂点 2 へ戻ってからまだ探索していない頂点 5 へと進む ($s = 5$)．頂点 5 には，頂点 9 と 10 の子ノードが存在するため順に探索する．その後，頂点 2 へ戻ってから頂点 6 を探索する．すると頂点 2 から新たに探索できる頂点がなくなるため，さらに頂点 1(根) へ戻って次は頂点 3 へ進み，同様に最後まで探索を進めていく．

つまり，この木に DFS を適用したときの頂点の探索順は **1, 2, 4, 5, 9, 10, 6, 3, 7, 11, 8** となる．

このアルゴリズムと類似なものとして，根に近いもの (高さの小さい) 頂点から順に始めて，徐々に木の葉へ向かって捜索を広げるように探索を行う**幅優先探索** (Breadth First Search) というものもある．おそらくこの後の本題である,「スプライトやウィンドウの RPG ツクールのゲーム画面上への描画順序を理解する」という点では DFS のみを理解できていれば十分だと考えられるため，余力のある読者は各自で調べていただきたい．

2.4　コアスクリプトから見る描画順定義

さて，木構造に関する準備体操を終えたところで，いよいよ本題である RPG ツクールのゲーム画面上への RPG ツクールの標準機能を用いた，スプライトやウィンドウの描画順序の制御について考えていく．RPG ツクールの標準機能を用いたスプライトというのは，具体的にはピクチャや遠景，マップチップといったものが挙げられる．

ゲーム画面 (実行中のシーン) への描画は，`addChild` した順に行われるということを 2.1 節および 2.2 節で確認した．しかし RPG ツクールでピクチャを表示すれば，常に一番手前に表示されるし，遠景はマップチップよりも奥でなければならない．描画する，しないはエディタからの命令によって行われるため，ただ `addChild` された順に描画したのではゲーム画面が滅茶苦茶になることは明らかである．そこで，コアスクリプトでスプライトの描画順序をどう制御しているかを調べてみる．

ソースコード 4: スプライトに関するコアスクリプト

```
1  Spriteset_Base.prototype.initialize = function() {
2    Sprite.prototype.initialize.call(this);
3    this.setFrame(0, 0, Graphics.width, Graphics.
       height);
4    this._tone = [0, 0, 0, 0];
5    this.opaque = true;
6    this.createLowerLayer();
7    this.createToneChanger();
8    this.createUpperLayer();
9    this.update();
10 };
11
12 Spriteset_Base.prototype.createLowerLayer =
       function() {
13   this.createBaseSprite();
14 };
15
16 Spriteset_Base.prototype.createUpperLayer =
       function() {
17   this.createPictures();
18   this.createTimer();
19   this.createScreenSprites();
20 };
21
22 Spriteset_Base.prototype.createBaseSprite =
       function() {
23   this._baseSprite = new Sprite();
24   this._baseSprite.setFrame(0, 0, this.width, this.
       height);
25   this._blackScreen = new ScreenSprite();
26   this._blackScreen.opacity = 255;
27   this.addChild(this._baseSprite);
28   this._baseSprite.addChild(this._blackScreen);
29 };
30
31
32 Spriteset_Base.prototype.createPictures = function
       () {
33   var width = Graphics.boxWidth;
34   var height = Graphics.boxHeight;
35   var x = (Graphics.width - width) / 2;
36   var y = (Graphics.height - height) / 2;
37   this._pictureContainer = new Sprite();
38   this._pictureContainer.setFrame(x, y, width,
       height);
39   for (var i = 1; i <= $gameScreen.maxPictures();
       i++) {
40       this._pictureContainer.addChild(new
             Sprite_Picture(i));
41   }
42   this.addChild(this._pictureContainer);
43 };
44
45 Spriteset_Base.prototype.createTimer = function()
       {
46     this._timerSprite = new Sprite_Timer();
47     this.addChild(this._timerSprite);
48 };
49
50 Spriteset_Base.prototype.createScreenSprites =
       function() {
51     this._flashSprite = new ScreenSprite();
52     this._fadeSprite = new ScreenSprite();
53     this.addChild(this._flashSprite);
54     this.addChild(this._fadeSprite);
55 };
```

ソースコード 4 は，コアスクリプト中の RPG ツクールの標準機能を用いたスプライトの描画を制御するコードの一部分を抜粋したものである．

ソースコード 4 にあるのは，`Spriteset_Base` というゲームエンジン全体に渡る基礎的な描画のためのクラスである．ここにも初期化を示す initialize 関数が当然存在する．この initlialize 関数を眺めてみると，6 行目と 8 行目に `createLowerLayer();` と `createUpperLayer();` という重要な関数が存在することが分かる．

その内容は 12-20 行目に記述されている．まず，`createLowerLayer();` ではさらに `createBaseSprite();` という関数が呼び出されている (22-29 行目)．この関数を見てみると，`addChild` が実行されている．

`this._blackScreen` という名前からして，ゲーム画面の一番後ろで画面全体を黒くするためのものだと推測できる．しかし，黒い画面を直接貼り付けるわけではなく，仮のスプライト `this._baseSprite` を用意してそこへ紐づけるようになっている．つまり，2.2 節で見た `_windowLayer` と同じ手法である．

一方，`createUpperLayer();` の方を見てみると，`createPictures();`，`createTimer();` と `createScreenSprites();` という 3 つの関数呼び出しがある．Picture や Timer とあるから，RPG ツクールに慣れた者であれば馴染みのある名前だろう．ピクチャやタイマーを画面上に描画するための処理が書かれている．これら 3 つの関数をよく見てみると，`addChild` が実行されている．

`createPictures();` では，エディタで設定される一通りの番号のピクチャの管理が行われている．ここでもやはり直接描画するのではなく，仮の描画空間を用意しておいて，そこで全てのピクチャを割り当てるようになっている．

一方で，タイマー描画を扱う `createTimer();` や，画面のフラッシュ、フェード等を扱う `createScreenSprites();` では仮の中間スプライトを挟まないで直接 `addChild` を行っていると見える．

ところで，コードだけを追いかけていても全容がなかなか把握しづらい．現在の `Spriteset_Base` の状況を先ほどの木構造を使って整理してみよう．ソースコード 4 から読み取れた `addChild` による各オブジェクトの親子関係を現した木構造を図 10 に示している．

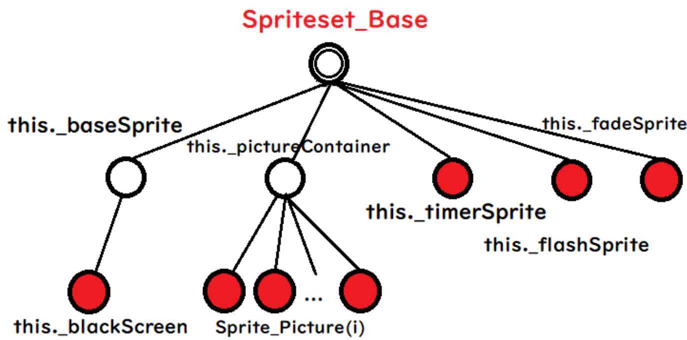

図 10: 抽出した Spriteset_Base の構造

今，Spriteset_Base というクラスを見ているので Spriteset_Base という要素が木の根に位置している．現在の this は Spriteset_Base となる．赤丸は，実際に描画されるデータが入っている頂点を現している．

this.addChild が実行された順に Spriteset_Base へ繋がれていくと考えれば，ソースコード 4 の実行順から考えて，createLowerLayer(); で呼び出されて生成された this._baseSprite が先に Spriteset_Base の子として接続し，その後に createUpperLayer(); で呼び出されて生成された this._pictureContainer, this._timerSprite らが続いていく．

そして，黒背景である this._blackScreen は this._baseSprite の子として，エディタで設定されたピクチャの 1 枚 1 枚である Sprite_Picture(i) は this._pictureContainer の子として接続されている．

ここで，この木に DFS を 2.3 節でやったように適用して各要素がゲーム画面上に描画される順を求めてみよう．左側の要素 (先に登録された順) を優先的に辿っていけば，黒背景があって，その上にピクチャを描画し，タイマー表示をさらにその上に行う．そして，画面のフラッシュ効果が乗り，一番上に画面のフェードイン・アウトの効果が乗ることとなる．

問題 3 ソースコード 4 を見る限り，マップチップや遠景，キャラクター画像はこの木には含まれていないように見えるが，実際にはこの木構造に正しく登録されている．どのように処理が行われているのだろうか？

Spriteset_Base のコード上では，黒背景を描画するために this._baseSprite を作成した後，すぐに this._blackScreen を作成して子ノードとして登録している．ピクチャについても同様に，this._pictureContainer を作成するや否や Sprite_Picture(i) を作成して子ノードとして登録している．これだとあまり意味がないようにも思えるが，この構造が生きてくるのは**後からスプライトを指定の改装へ登録する**場合である．

例えば，RPG ツクール MV のエディタからピクチャを登録できる個数は最大で 100 である．もし,「1000 枚のピクチャを処理するプラグインを作成する」ことになればどうすればよいだろうか？「Sprite クラスに画像を貼り付けて addChild する」が安直な解答だが，それだけだとピクチャが思わぬ階層に表示される (例えば，キャラクターの下に追加ピクチャが来る) 事例が想定される．

101 枚以上のピクチャを自分で登録したければ，this._pictureContainer の下に登録，つまり this._pictureContainer.addChild(picture); のように書けば，this._pictureContainer の描画順序はコアスクリプトで事前に上層付近と定義されているため，自分でスクリプトを介して登録したピクチャも，エディタから登録したピクチャと同様の扱いができるはずである．

このように，木構造のイメージを持って扱うスプライトやウィンドウの描画階層を意識すれば，自作するオリジナルシーンでも必要なスプライトやウィンドウの描画順を自在に操ることができるだろう．

3　自作シーンでのスプライト・ウィンドウの描画管理

ここまで来れば，矛盾のない描画順序で作られたオリジナルシーンを作成できるだろう．改めて Scene_Test を使った実験をしてみよう．

3.1　シーンの子要素の順序付け

次のソースコード 5(SoR_sm4.js) は，SoR_sm3.js を元に拡張したものである．js ファイル本体はサーバー上[2]に置いた．

今度の Scene_Test では，コアスクリプトを真似して，create 関数の最初に描画するオブジェクトたちのおおまかな描画順序をの決定を行うための仮のスプライトを.createBackSpriteLayer(); および createFrontSpriteLayer(); にて予め定義している．その後に，実際に描画のための addChild を行う関数を呼び出していて，SoR_sm3.js よりいくつか描画物を追加している．

[2] http://dragonflare.dip.jp/dcave/articles/SoR_sm4.js

ソースコード 5: 'SoR_sm4.js'

```
function Scene_Test(){
        this.initialize.apply(this, arguments);
}

Scene_Test.prototype = Object.create(Scene_Base.prototype);
Scene_Test.prototype.constructor = Scene_Test;

Scene_Test.prototype.initialize = function() {
    Scene_Base.prototype.initialize.call(this);
}

Scene_Test.prototype.create = function() {
    Scene_Base.prototype.create.call(this);
        this.createBackSpriteLayer();
        this.createWindowLayer();
        this.createFrontSpriteLayer();


        this.createSprites();
        this.createButton();
        this.createDamage();
        this.createTestWindow();
}

Scene_Test.prototype.start = function() {
        Scene_Base.prototype.start.call(this);
}

Scene_Test.prototype.update = function() {
    Scene_Base.prototype.update.call(this);
}


//layer for backgronud images
Scene_Test.prototype.createBackSpriteLayer = function(){
  this._backlayer = new Sprite();
  this.addChild(this._backlayer);
}
//layer for foreground images
Scene_Test.prototype.createFrontSpriteLayer = function(){
  this._frontlayer = new Sprite();
  this.addChild(this._frontlayer);
}

// main window
Scene_Test.prototype.createTestWindow = function(){
    this._window = new Window_Base(Graphics.width/2-120, Graphics.height/2-120, 240,240);
        this._window.setBackgroundType(0);
    this.addWindow(this._window);

    this._window2 = new Window_Base(Graphics.width/2+70, Graphics.height/2-120, 180,80);
        this._window2.setBackgroundType(0);
    this.addWindow(this._window2);
}

Scene_Test.prototype.createSprites = function() {
        const img = ImageManager.loadSystem("GameOver");
        this._picture = new Sprite(img);
        this._backlayer.addChild(this._picture);
}

Scene_Test.prototype.createButton = function() {
        const img = ImageManager.loadSystem("ButtonSet");
        this._button = new Sprite(img);
        this._frontlayer.addChild(this._button);
}

Scene_Test.prototype.createDamage = function() {
        const img = ImageManager.loadSystem("Damage");
        this._damage = new Sprite(img);
        this._frontlayer.addChild(this._damage);
}

//////////////////////////////////////////
Scene_Boot.prototype.start = function() {
        DataManager.setupNewGame();
    SceneManager.goto(Scene_Test);
}
```

このコードを実行したシーンを映したものが図 11 である．3 つの System 以下にある画像と，2 つのウィンドウが描画されている．

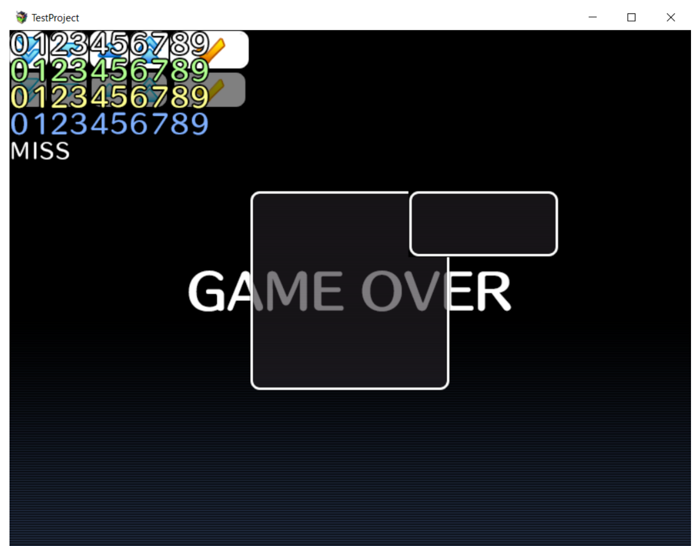

図 11: SoR_sm4.js を実行したゲーム画面

特に，ゲームオーバー画面は最背面に来るように描画されている．他のピクチャやウィンドウはそれよりも前面へ描画されている．そのような描画順になるよう制御するために書かれたのが，35-43 行目にあるコードである．描画順を決めるためだけの，空のスプライト this._backlayer と this._frontlayer が定義され，this.addChild で Scene_Test に紐づけられている．

登録された順，かつ木構造を DFS で辿る要領で描画順が決定されるため，なるべく背面に描画したいものは this._backlayer の子として，前面に描画したいものは this._frontlayer の子として登録すればよいということになる．

DFS の挙動をもう一度思い出せば，this._backlayer

の子として登録しておけば，登録したタイミングが処理の後の方であっても，`this._frontlayer` より後の描画になることは絶対にない．もちろん，`this._backlayer` に登録されたものの中では低い優先順位 (後の方での描画) となる．

このように，作成するシーンが実際にゲーム上で動くときに，ウィンドウやスプライトがどのように描画されるかを木構造を意識してコードを書けば，考えた通りのデザインを持つシーンが作成しやすくなるだろう．

3.2 子要素の順序管理

描画順序の制御について，やや応用的な話題に触れよう．次のソースコード 6(`SoR_sm5.js`) は，`SoR_sm3.js` を元に拡張したものである．js ファイル本体はサーバー上[3)]に置いた．

ソースコード 6: 'SoR_sm5.js'

```
1  function Scene_Test(){
2   this.initialize.apply(this, arguments);
3  }
4  Scene_Test.prototype = Object.create(Scene_Base.
       prototype);
5  Scene_Test.prototype.constructor = Scene_Test;
6
7  Scene_Test.prototype.initialize = function() {
8   Scene_Base.prototype.initialize.call(this);
9  }
10
11 Scene_Test.prototype.create = function() {
12  Scene_Base.prototype.create.call(this);
13  this.createBackSpriteLayer();
14  this.createWindowLayer(); /////<-- initialization
15  this.createFrontSpriteLayer();
16
17  this.createSprites();
18 }
19
20 Scene_Test.prototype.start = function() {
21  Scene_Base.prototype.start.call(this);
22  this._windowLayer.children.sort(compare);
23 }
24
25 Scene_Test.prototype.update = function() {
26  Scene_Base.prototype.update.call(this);
27 }
28
29
30 //layer for backgronud images
31 Scene_Test.prototype.createBackSpriteLayer =
       function(){
32  this._backlayer = new Sprite();
33  this.addChild(this._backlayer);
34 }
35 //layer for foreground images
36 Scene_Test.prototype.createFrontSpriteLayer =
       function(){
37  this._frontlayer = new Sprite();
38  this.addChild(this._frontlayer);
39 }
40
41 Scene_Test.prototype.createSprites = function() {
42  this.characters = [];
43  const img = ImageManager.loadCharacter('
       Actor1');
44
45  for(let i=0;i<8;i++){
46    this.characters.push(new Sprite(img));
47    this.characters[i].setFrame(48+i%4 *144, Math.
         floor(i/4)*192, 48, 48);
48    this.characters[i].x = 320;
49    this.characters[i].y = 400-i*36;
50    this._frontlayer.addChild(this.characters[i]);
51  }
52
53 }
54
55
56
57 /////////////////////////////////////////
58 Scene_Boot.prototype.start = function() {
59         DataManager.setupNewGame();
60     SceneManager.goto(Scene_Test);
61 }
```

これは，キャラクター画像 (Actor1.png) を読み込んで，図 12 のようにハロルドを先頭に画面下から上へ並ぶように配置したものである．ところが，後から描画したキャラクターが前のキャラクターの上部に表示されてしまい，不自然な絵面になってしまっている．

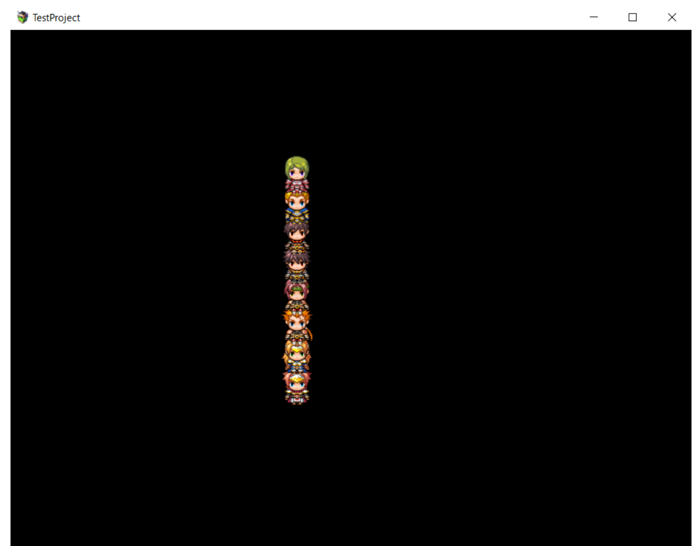

図 12: キャラクターを下から上へ縦一列に並べた画面

描画順番を逆にすればいいという思うかもしれないが，実際のゲーム画面だと，キャラクターはゲーム中に画面内を移動してしまうかもしれない．そうなると，最初に規定した描画順序では正しい描画ができなくなってしまう．

そこで，描画順序を好きなタイミングで入れ替える方法を試しておこう．初期化処理の最終段階である start

[3)] `http://dragonflare.dip.jp/dcave/articles/SoR_sm5.js`

関数を，次のソースコード 7 のように変更し，さらに compare() という関数を付け足す．

ソースコード 7: 'start 関数の変更'

```
1  Scene_Test.prototype.start = function() {
2    Scene_Base.prototype.start.call(this);
3    this._frontlayer.children.sort(compare);
4  }
5
6  function compare(a, b){
7    const val = a.y−b.y;
8
9    if (val > 0) return 1;
10   else if (val < 0) return −1;
11   else return 0;
12 }
```

このコードを SoR_sm5.js に追加して実行した結果が，図 13 である．今度はハロルドが手前に居るような自然な描画となっている．

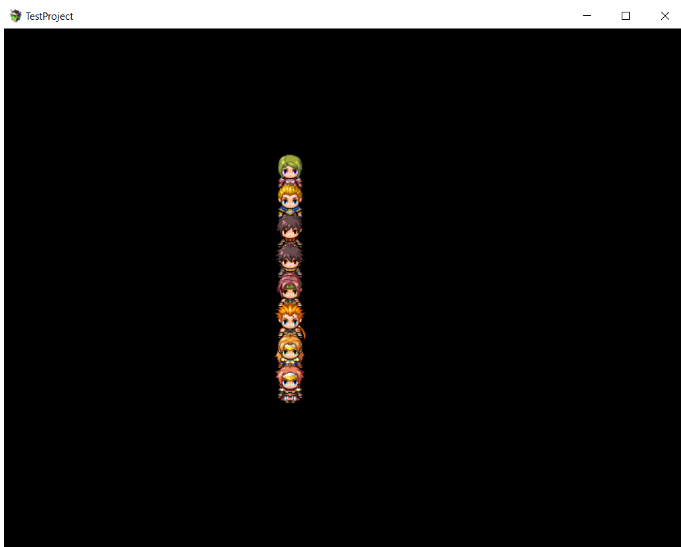

図 13: コード修正後のキャラクターの整列

さて，何をしたのかコードを見てみよう．とはいえ，本質的なことは 3 行目にある 1 文のみである．これはソート (並べ替え) を行う関数である．

子要素の並べ替え

this._frontlayer.children.sort(比較関数);

ここまでの内容を理解していれば，この呼出し命令の意味の解釈は容易で，「Scene_Test の子ノード _frontlayer に紐づけられた子要素 (children) を並べ替えよ」となる．_frontlayer の children とはつまり，実際に描画されている画像 (スプライト) ということになる．

ただし並べ替えとは本来，数値 (大きい，小さい) や文字 (アルファベット昇順，降順) のような**大小関係が明確なものに対して行われる操作**であり,「キャラクターの画像を並び替える」ことを命令しても，コンピュータは理解できない．そこで**比較関数**と呼ばれる**大小関係が不明瞭なもの同士の大小関係を規定する**関数を作成し，関数名を sort 関数の引数に渡すことでいろいろな変数・オブジェクトに対して並べ替えが可能になる．

ソースコード 7 では 6-12 行目の compare という名前で作られた関数が比較関数にあたる．javascript に慣れた読者であればアロー関数を使えば簡単に記述できるが，本稿は javascript に慣れた者のための講座ではないこと，他オブジェクト指向言語への可搬性，および初学者への可読性に配慮して明示的に記述している．

compare 関数は整数値を sort 関数へ返し，sort 関数はその結果の正負を見てデータの順序を決定する．6 行目の compare 関数の仮引数 (a, b) は，sort 関数が選んだ任意の 2 つのデータ (つまり this._frontlayer の子要素のキャラクター画像である．7 行目では，キャラクター画像の y 座標の差を調べていて，その結果を元に順序を決定している．戻り値の意味は次のとおりである．

比較関数の戻り値

- 戻り値が 正 → a は b より後ろ
- 戻り値が 負 → a は b より前
- 戻り値が 0 → 同値として並べ替えなし (実装による)

例では y 座標の差を調べているが，これが正であるときにキャラクターの画像データ a は children のより後ろへ置かれる．つまり，並べ替えた後には y 座標の大きいものほど後から描画されるようになる．

これに基づいて，sort 関数が与えられた比較関数に基づいて全てのデータの組み合わせ (a, b) について検証し，データの順列を決定している．したがって，順序を評価するための計算式を変更すれば，どのようなデータに対しても順序を定義できるということである．

一部のスクリプトでは，各データに z 座標に相当する変数を勝手に定義して，描画の優先順序を整数で指定し，必要に応じて並べ替えるという手法をしているものもある．作成したデータに対する工夫や独自の定義を施すことで，描画順位は意のままにコントロールできることを覚えておこう．ただし，コアスクリプトを変更する類のスクリプトを書く際には，あまりに勝手な定義を付け加えてしまうと競合を起こしておかしな動作になってしまう可能性があるので，よく検討の上で実装しよう．

4 練習問題

4.1 プロジェクト内の自作フォルダからの画像読み込み

ImageManager というクラスが RPG ツクールのプロジェクト内の画像を管理している．問題 1 のように，標準で設定されているフォルダ (./img/animations や ./img/faces) の中にある画像を読み込むための関数は提供されている．

自作シーンのためにプロジェクト内に作成した**新しいフォルダへ必要画像を格納した時**，どのようにすればそれらをゲーム内で描画できるか？

4.2 Scene_Test の木構造

1. SoR_sm4.js において，描画されるウィンドウやスプライトの順序を現す木構造を図 10 を参考に示せ．
2. this._frontlayer に登録されるスプライトの中でも，必ず最も背面へ描画したい画像があるとする．しかし，その画像はいつ addChild できるか分からない (create 関数の時点では用意できない，または描画の必要がない) らしい．Scene_Test にどのような木構造を作成しておけば，これを実現できるか？

4.3 描画物の描画順序の動的入れ替え

次のソースコード 8(SoR_sm6.js) は，5 つのウィンドウをでたらめにレイアウトしたものである．js ファイル本体はサーバー上[4]に置いた．実際にゲームを起動してみると，ウィンドウは画面全体を覆うものが 1 つ描画されているのみである．

このシーンで義されているウィンドウを，次の方法でコードを修正し，全てのウィンドウが画面内に描画されていることが確認できるようにせよ．

1. 各ウィンドウに定義されている変数 z を手で調整し，z の値に応じて並べ替えを行う
2. z を使わず，**ウィンドウの描画面積のより小さいものが手前に描画される**に並べ替えを行う

[4] http://dragonflare.dip.jp/dcave/articles/SoR_sm6.js

ソースコード 8: SoR_sm6.js'

```
1  function Scene_Test(){
2   this.initialize.apply(this, arguments);
3  }
4  Scene_Test.prototype = Object.create(Scene_Base.
       prototype);
5  Scene_Test.prototype.constructor = Scene_Test;
6
7  Scene_Test.prototype.initialize = function() {
8   Scene_Base.prototype.initialize.call(this);
9  }
10
11 Scene_Test.prototype.create = function() {
12  Scene_Base.prototype.create.call(this);
13  this.createBackSpriteLayer();
14  this.createWindowLayer(); /////<- initialization
15  this.createSprites();
16
17
18  this.createTestWindow();
19  this.createTestWindow2();
20  this.createTestWindow3();
21  this.createTestWindow4();
22  this.createTestWindow5();
23 }
24
25 Scene_Test.prototype.start = function() {
26  Scene_Base.prototype.start.call(this);
27 }
28
29 Scene_Test.prototype.update = function() {
30  Scene_Base.prototype.update.call(this);
31 }
32
33 //layer for backgroud images
34 Scene_Test.prototype.createBackSpriteLayer =
       function(){
35  this._backlayer = new Sprite();
36  this.addChild(this._backlayer);
37 }
38
39 // main window
40 Scene_Test.prototype.createTestWindow = function
       (){
41  this._window = new Window_Base(Graphics.
        width/2-120, Graphics.height/2-120,
        240,300);
42  this._window.setBackgroundType(0);
43  this._window.z = 4;
44  this.addWindow(this._window);
45 }
46 Scene_Test.prototype.createTestWindow2 =
       function(){
47  this._window2 = new Window_Base(Graphics.
        width/2+100, Graphics.height/2-20, 480,320)
        ;
48  this._window2.setBackgroundType(0);
49  this._window2.z = 2;
50  this.addWindow(this._window2);
51 }
52
53 Scene_Test.prototype.createTestWindow3 =
       function(){
54  this._window3 = new Window_Base(Graphics.
        width/4-120, Graphics.height/2+120,
        320,260);
55  this._window3.setBackgroundType(0);
56  this._window3.z = 3;
57  this.addWindow(this._window3);
58 }
59
```

```
60 Scene_Test.prototype.createTestWindow4 =
       function(){
61  this._window4 = new Window_Base(0, 0, 500,400)
        ;
62  this._window4.setBackgroundType(0);
63  this._window4.z = 1;
64  this.addWindow(this._window4);
65 }
66
67 Scene_Test.prototype.createTestWindow5 =
       function(){
68  this._window5 = new Window_Base(0, 0,
        Graphics.width,Graphics.height);
69  this._window5.setBackgroundType(0);
70  this._window5.z = 0;
71  this.addWindow(this._window5);
72 }
73
74
75 Scene_Test.prototype.createSprites = function() {
76  const img = ImageManager.loadSystem("
        GameOver");
77  this._picture = new Sprite(img);
78  this._backlayer.addChild(this._picture);
79 }
80
81 ////////////////////////////////////////
82 Scene_Boot.prototype.start = function() {
83  DataManager.setupNewGame();
84  SceneManager.goto(Scene_Test);
85 }
```