メッセージ指向ゲーム開発言語
「司エンジン」ガイドブック第３版（v2.2 対応）

本書はゲーム開発に最適化して設計されたオリジナルのプログラミング言語「tsukasa」と、その実行環境である「司エンジン」を、豊富なサンプルコードを用いて初歩からゲームの作り方を解説するガイドブックです。

司エンジンはコード注入型ステートマシンを特徴とするメッセージ指向プログラミング言語で、極めてゲームが作りやすい設計になっており、既存のゲームエンジンに対して圧倒的な生産性を実現します。

司エンジンはオープンソースで開発されていて、最新版のパッケージをネットからダウンロードしてゲーム開発を始めることができます。同人、商業にかかわらず、使用料は一切かかりません。

巻末には最新版の仕様に対応したリファレンスマニュアルが収録されています。

【注意】

・本書は司エンジン v2.2 をベースにしています。
・最低限の Ruby 文法の知識が必要です（本書では解説しません）。
・司エンジンは Windows 専用です。

※本書は２０１６年１１月１３日に発刊された同人誌「ポスト・ゲームエンジン tsukasa 言語が示す次世代ゲームエンジンアーキテクチャ」の内容を改訂し、v2.2 に対応させた物です。

メッセージ指向ゲーム開発言語「司エンジン」ガイドブック

土屋つかさ 著
染井吉野ゲームズ 編

第３版
（v2.２対応）

メッセージ指向ゲーム開発言語

司エンジン ガイドブック

土屋つかさ・著
染井吉野ゲームズ・編

オリジナルプログラミング言語「tsukasa」＆
無料ゲームフレームワーク「司エンジン」の
サンプルコード解説＆リファレンスマニュアル！

メッセージ指向ゲーム開発言語

「司エンジン」ガイドブック

第3版（V2．2対応版）

オリジナルプログラミング言語「tsukasa 言語」＆
無料ゲームフレームワーク「司エンジン」の
解説＆リファレンスマニュアル！

土屋つかさ　著

サークル染井吉野　刊

はじめに

本書は、ゲーム開発に最適化されたメッセージ指向ゲーム開発言語「tsukasa」の解説を通して、現代のゲームエンジンアーキテクチャが内包している課題を検討し、次世代のゲームエンジンアーキテクチャの姿を示す事を目的に書かれました。

巻末にはtsukasa言語の実装系である「司エンジン」のリファレンスマニュアルが収録されており、ネットから開発環境をダウンロードすれば、今すぐ多数のサンプルを実行して挙動を確認できます。

本書は以下の4部から構成されています。

第1部：メッセージ指向ゲーム開発言語「tsukasa」

現代のゲームプログラミングが抱える課題を列挙し、それを解決する為にtsukasa言語が採用したアプローチについて解説します。

第2部：司エンジンの使い方

tsukasa言語の機能を、サンプルコードを見ながら1つずつ解説します。

第3部：より高度なプログラミング

tsukasa言語の複数の機能を組み合わせた、より複雑な処理を解説します。

第4部：tsukasa言語リファレンスマニュアル（v2.2対応版）

tsukasa言語のインターフェイスを網羅したリファレンスマニュアルです。

筆者は、tsukasa言語／司エンジンが採用しているアーキテクチャが、将来的にあらゆるゲームエンジンにおいてスタンダードなスタイルになると確信しています。是非本書を読んで、筆者が確信する未来が本当に来るのかどうか、検証してみてください。

土屋つかさ　2017年2月14日

目次

はじめに 2

第1部　メッセージ指向ゲーム開発言語「tsukasa」..... 5

1　一般的なゲームアーキテクチャ 6
2　既存ゲームアーキテクチャの課題と司エンジンのアプローチ 8
3　まとめ 10

第2部　司エンジンの使い方 11

tsukasa 言語の概要と構造 12
サンプル 1：コントロールの操作 14
サンプル 2：コントロールを移動させる 16
サンプル 3：コントロールを曲線移動させる 17
サンプル 4：フェードイン・アウト 18
サンプル 5：コントロールの生成 19
サンプル 6：フェードトランジション 21
サンプル 7：より自然なフェードトランジション 23
サンプル 8：ルールトランジション 25
サンプル 9：ユーザー定義コマンドの宣言と実行 28
サンプル10：データストアへの読み書きと動的なテキスト出力 30
サンプル11：条件分岐 32
サンプル12：ループ 34
サンプル13：ボタンの制御とスクリプトの追加読み込み 35
サンプル14：サウンドの再生 37
サンプル15：セーブとロード 38
サンプル16：【tks スクリプト 1】文章の表示 40
サンプル 17：【tks スクリプト 2】文章を装飾する 42

第3部　より高度なプログラミング 45

1：様々なボタンの作成 46
2：コントロールのアニメーション 51
3：文字列の表示とポーズ処理 52

第4部　tsukasa 言語リファレンスマニュアル (v2.2 対応版) 55

司エンジン導入手順.....56
tsukasa 言語記法解説 58
tks スクリプト記法解説..... 60
Control.....62
Layoutable (内部モジュール) 71
Layout.....72
Window.....73
Drawable (内部モジュール) 75
Image.....76
Input..... 80
Data.....81
DrawableLayout.....82
Clickable (内部モジュール) 84
ClicakbleLayout..... 86
Sound.....87
Char..... 89
TextPage..... 90
TileMap.....93
Shader.....95
RuleTransition 96
標準ユーザー定義コマンド (default_script.rb) 97
ヘルパーコマンド (helper_script.rb) 101
テキストウィンドウ (text_layer_script.rb) 104
起動時に自動的に生成されるコントロール群.....107
司エンジンのフォルダ構成 108
組み込み定数一覧.....111

補足.....114

おわりに 115

奥付.....116

第1部
メッセージ指向ゲーム開発言語「tsukasa」

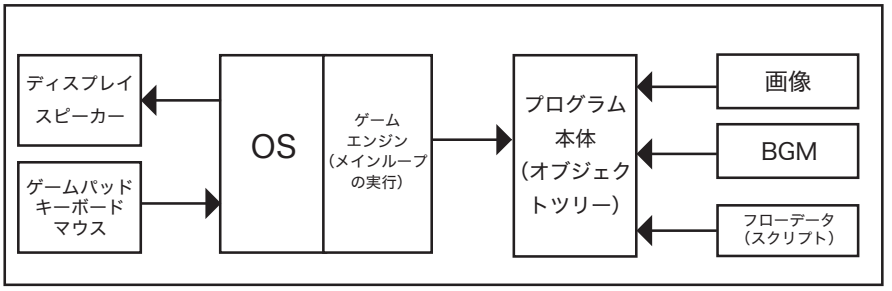
tsukasa 言語は「近年のゲーム開発が高コスト化しているのは、既存のゲームアーキテクチャがゲーム開発の大規模化にスケールできいないからで、解決の為には新しいアプローチの言語が必要」というコンセプトの基に設計されました。

第1部では、これまで使われてきた一般的なゲームアーキテクチャについて解説し、そのアーキテクチャが内包している課題と、その課題に対して tukasa 言語が採用したアプローチを説明します。

1　一般的なゲームアーキテクチャ

ここではまず、ゲームプログラミングの一般的なスタイル、その中でも根幹となるコア部分のゲームアーキテクチャについて、概念レベルで説明します。作られるゲームの規模や動作環境にかかわらず、近代のゲームプログラミングではほぼ同じ方法で実装されています。

実行中のゲームプログラム（以下プログラム）の概念図を示し、以下、メインループ／ステートマシン／スクリプトを中心に、一般的なゲームアーキテクチャについて解説します。



（1）メインループ

プログラムは動作している間、OSから60分の1秒ごとに繰り返し呼び出されます。呼び出されたプログラムは内部状態を更新し、更新された状態に応じて画面を描画し、OSに処理を返します。この定期的な呼び出しのことを「メインループ」と言います。また、秒間に60回画面が更新されることを60FPS（frame per second）と呼称します。

60分の1秒を単位としているのは、過去の一般的なディスプレイが60Hz（秒間60回更新）で駆動していたことに由来しています（※1）。1回の呼び出しにかかる時間が60分の1秒を超えた場合、描画が間に合わないので処理落ちが発生してしまいます。その為、プログラムは可能な限り60分の1秒以内に更新を完了しOSに処理を戻すことが求められます。

Windowsなどイベント駆動式のOSでは、このようなメインループ処理を構築しにくい為、通常はゲームエンジンがOSの挙動をラップして代わりにメインループ処理を制御します。ゲーム専用機の場合は、OSとゲームエンジンの機能は融合していて、OSが直接メインループ処理を提供します。

次に、メインループの呼び出し時に、プログラムが行う作業を説明します。

プログラムはファイルシステムにアクセスし、ゲームに必要なリソースデータを取得します。リソースデータには画像やBGM、ポリゴンモデルの頂点情報やキャラステータスのような数値データなどがあります。また、ゲームの進行制御に必要な「スクリプト（後述）」と呼ばれる特殊な性質を持つデータもリソースデータに含まれます。

プログラムはゲームパッドやマウスなどの入力デバイスから値を取得し、リソースデータの情報を加味して、ゲームの状態を更新します。そしてその結果を出力デバイスであるディスプレイやスピーカーに出力します。これを秒間60回行うことによって、ユーザーに連続したゲーム体験を提供しているのです。

（※1）スマートフォンが主要なゲームハードとなった現在では、常に安定した60FPSを実現することがハードスペック的に困難なため、主要なゲームエンジンでは可変FPSを採用しています。可変FPSでは、メインループの呼び出し間隔が不定になり、プログラムは前フレームから今フレームまでの経過時間を計算して、適切な処理を実行します。

（2）ステートマシン

次に、ファイルシステムから読み込んだリソースデータが、どのようにメモリ上に格納されているかを説明します。

一般的なゲームプログラムでは、ゲームを構成する各要素をオブジェクトとして生成しています。オブジェクトには画像やBGMなど、リソースデータを直接表す物の他、主人公キャラのように複数のリソースデータの集合として作られるものや、「タイトル画面」のような論理単位をさすものもあります。

これらのオブジェクトはステートマシン（state transition machine. 状態遷移機械）と呼ばれる構造になっています。個々のオブジェクトは内部でステート（状態）を持ち、自身が今どのステートにあるかに応じて、そのフレームでの処理が決定されます。

生成されたオブジェクトは、単一のオブジェクトツリーに登録されます（※2）。プログラムは1回の呼び出しごとにこのオブジェクトツリーを全探査し、全てのオブジェクトに対してステートの更新処理を命令します。

アクションゲームの主人公キャラのオブジェクトについて考えてみましょう。このオブジェクトは、ステートとして「通常」と「ジャンプ中」の2種類を持っています。

現在は「通常」が自身のステートになっています。このステートの時は、ユーザーからのジャンプボタンの入力を受け付けています。

ジャンプボタンが押されると、オブジェクトのステートが「ジャンプ中」に切り替わります。同時にジャンプがスタートし、キャラが上方向へ放物線移動し始めます。この間は、ユーザーからのジャンプボタンの入力は（既にジャンプしている最中なので）受け付けていません。

ジャンプが終わり、キャラが元の高さに戻ると、ステートは「通常」に戻り、再びジャンプボタンの入力を受け付けるようになります。このように、複数のステートごとに処理を変えることで、ゲームの複雑な挙動を制御しているのです。

（3）スクリプト（制御構造を持つフローデータ）

ゲームは他のコンテンツメディアに対して決定的に異なる、インタラクティブであるという特徴を持っています。コンテンツがユーザーの行動に応じて、動的に変化するのです。

コンテンツを動的に変化させるためには、コンテンツを構成するデータが制御構造を持つ必要があります。例えば、ノベルゲームにおいてはシナリオデータが制御構造を持っています。ユーザーの選択によってストーリーが分岐する場合、その分岐処理はデータで表現されていなければなりません（※3）。

このデータは、コンテンツ全体で莫大な容量になることが多く、その時点で使う部分だけをメモリに載せ、使い終わった物から破棄するロジックが必要です。このように使われていくデータのことをフローデータと呼びます。ノベルゲームでは、プレイヤーが読み終わったシナリオデータは不要なので、メモリ上から順次破棄して、新たなシナリオデータを読み込んでいきます。

ゲームプログラミングでは、このような制御構造を持つフローデータとして「スクリプト」と呼ばれる簡易な言語を使用します。ノベルゲームに限らず、ほとんどのゲームはこのようなスクリプトを使って作られています。

（※2）キャラクターのように描画要素を持つ物と、BGMのように描画要素を持たない物でツリーのノードを大別する傾向があります。
（※3）分岐処理のような制御箇所をデータではなく、プログラムで持つというアプローチもありますが、ゲームコンテンツではよく似た制御が必要になるシーンが無数に存在し、またその発生箇所が不定期なため、一般的にはスクリプト側で設定します。

2　既存ゲームアーキテクチャの課題と司エンジンのアプローチ

ここまで、一般的なゲームアーキテクチャについて解説してきました。この一般的なアーキテクチャは、商業ゲームの最初期に採用されて以来、今日まで抜本的な変化を経ず使われてきました。

しかし近年、このアーキテクチャはひとつの限界に達しつつあります。現在のゲームプログラミングは、その初期とは比較にならないほど複雑化しつつあり、また、当時とは比較にならないほど大量の情報を処理しなければならなくなっています。そのため、このアーキテクチャではその複雑性と情報量を吸収しきれなくなりつつあるのです。

またこの傾向は、今後も継続することは確実で、ゲーム開発のうえでボトルネックとなりつつあります。

司エンジンは、この課題を解決する方法を示すために開発されたゲームエンジンです。

司エンジンが採用している、ゲームプログラミングに最適化された tsukasa 言語によって、複雑化、増量化していくゲームのリソースに対してより良くスケールする開発環境を提供します。

ここからは、一般的なゲームアーキテクチャが抱える課題と、それに対して司エンジン／tsukasa 言語が採用したアプローチを説明します。

（1）メインループ

課題：複数フレームにまたがる処理の記述

ゲームでは画面の再描画のために、秒間６０回のメインループ処理が実行されます。プログラムは処理落ちを起こさせないために、６０分の1秒以内に処理をＯＳに返さなければなりません。しかし、オブジェクトが取るほとんどの挙動は複数フレームにまたがる処理であり、このメインループ方式と相性がよくありません。

メインループ方式で複数フレームにまたがる処理を行う場合、あるフレームで処理が完了しなかった多くの情報を保存しておいて、次フレームの再呼び出し時にその情報を取得させなければなりません。この処理はゲームのロジックとは無関係にも関わらず、ソースコード上に頻出し、コードのメンテナンス性を低下させます。

tsukasa 言語のアプローチ：隠蔽されたメインループの採用

メインループの処理を、tsukasa 言語は処理系の中に隠蔽します。そのため、プログラマーはメインループ処理を意識する必要がありません。そして複数フレームにまたがる処理についても、継続して必要となる情報を、処理系が自動的に保存し、取得します。

（2）ステートマシン

課題：ステートの組み合わせ爆発

ステートマシンは、オブジェクトが持つ多様なステートを管理するのに適切なものですが、近年のゲームでは1つのオブジェクトが取り得るステートの数が膨大になっています。場合によっては、次にどのステートに遷移するのかを判断するロジックが数千行にわたってしまい、バグの温床と化すことがあります。

例えばスーパーマリオブラザーズの場合、主人公には見た目に応じて「通常」「キノコで大型化」「ファイアマリオ」「スター」という４種類のステートを取り得ます。また、移動時のステートとして「歩く」「走る」

「ジャンプ」「泳ぐ」があります。「蔦を昇る」「ポールを下りる」などの特殊な移動ステートもあります。

また、大型化時には「しゃがむ」ことができて、「しゃがみながらジャンプする」ことも可能です。つまり、1つのステートマシンが同時に複数のステートを持つことになります。また、あるステートを遷移する際に、現在のステートだけでなく、過去に取っていたステート情報に依存する場合もあります。くわえて、他オブジェクトとのインタラクションにも依存する場合があります。

このように、ゲームのオブジェクトは非常に多数のステートを持つうえ、遷移判定が相互に依存するため、ステートの遷移を判定するロジックはとても複雑かつ巨大な物になる傾向があります。

tsukasa 言語のアプローチ：プッシュダウンオートマトン方式ステートマシンの採用

tsukasa 言語で生成されるオブジェクトは、すべて、プッシュダウンオートマトンという方式のステートマシンになっています。プッシュダウンオートマトンとは、内部にプログラムを格納するスタックを持っているステートマシンです。プログラムスタックの中にあらかじめ、そのステートマシンに実行させたいプログラムを格納しておくことで、メインループ処理の際に、ステートマシンはスタックからプログラムを取得して、プログラムに対応するステートを実行し、その後そのプログラムを破棄します。

プログラムのスタックには、必要に応じて複数のプログラムを動的にスタックすることができます。また複数のステートにまたがる処理の場合は、ステートマシンが自律的にそのプログラムをスタックに再格納します。これによって、ステートの遷移を判定するロジックを巨大にすることなく、小さなサイズに保てるようになります。

（3）スクリプトの課題とアプローチ

課題：汎用的なゲームスクリプト言語の不在

商業ゲーム開発において、制御構造を持つフローデータ、すなわちスクリプト言語とその処理系は、これまでスタンダードとなりうる実装、仕様が発展しませんでした。そのため、開発タイトル毎にプログラマがゼロから実装し、作業コストを費やすことが少なくありませんでした。

最近ではＬｕａのようなオープンソース系スクリプト言語が採用される例もありますが、ゲームのスクリプト処理系では特別な処理が必要となる場合があります。たとえばノベルゲームでは、「画面に表示されている所でスクリプトの評価を休止する」という処理が必要になり、それを設計レベルで採用しているのは吉里吉里やＮＳｃｒｉｐｔｅｒなど、一部のノベルゲーム用スクリプト言語に限定されています。

tsukasa 言語のアプローチ：メッセージ指向言語の採用

tsukasa 言語は、ゲーム開発用のネイティブ言語であり、同時にスクリプト言語しても機能するように設計されました。その際、言語アーキテクチャにはメッセージ指向を採用しました。メッセージ指向は Smalltalk 言語などで採用されている、プログラムをオブジェクト間でやりとりされるメッセージと捉えるアプローチです。

tsukasa 言語では、記述されている全てのプログラムはコマンドの列として解釈され、コマンドの列はそれぞれが対象とするオブジェクトに対して動的に送信されます。送信されたコマンド列は、それらのオブジェクトのプログラムスタックに格納され、各オブジェクトが処理される際に実行されます。

全てのオブジェクトは疑似マルチスレッド的な挙動をとりますが、このコマンド送信方式によって、オブジェクト間の通信がフラットに行え、また、使用済みのコマンドが破棄されることで自動的にフローが維持されます。

3　まとめ

既存のゲームエンジンが採用しているゲームアーキテクチャを確認し、そのアーキテクチャが内包している課題と、その課題に対して tsukasa 言語がどのようなアプローチを採用しているかについて説明しました。

第2部からは、tsukasa 言語の実装系である「司エンジン」のサンプルコードを見ながら、実際に tsukasa 言語でゲームを作る方法を説明していきます。

司エンジンはスターターキットがインターネット上で配布されており、ダウンロード後すぐに起動できるようになっています。是非ご自身で司エンジンを起動してサンプルコードを確認し、tsukasa 言語の持つポテンシャルを確認してください。

【コラム】tsukasa 言語のミッションは「ゲーム開発速度の圧倒的な向上」

tsukasa 言語が目指しているのは「ゲーム開発速度の圧倒的な向上」です。

商業ゲーム開発では、プランナーのアイデアをプログラマーがテスト実装して確認できるようなるまでに非常に長い期間を必要とする傾向にあります。ＰＣやゲーム機の性能向上によってこの期間はさらに長大化し続けていて、その結果、ゲーム開発は常に莫大な手戻りが発生する危険性を抱えた作業になってしまっています。極端な話、実際に動作を確認できるプログラムが、リリース直前まででできあがらない場合すらあるのです。

この問題は、ゲーム開発者の間ではそれなりに共有されてはいますが、その原因については、ゲームがポリゴンモデルや背景データなど、製作に時間がかかるリソースデータを大量に必要とするコンテンツだからとされ、一般的には回避できないコストだと考えられています。

しかし、それは原因の一面でしかなく、より大きな原因が見過ごされてきたと筆者は考えています。

著者は、この問題の根源はリソースデータの製作コストではなく、現行のプログラミング言語のアーキテクチャと、デジタルゲームのアーキテクチャの相性が悪すぎる為に発生するボトルネックに起因していると考えました。その仮説を実証するモデルとして設計されたのが、tsukasa 言語なのです。

tsukasa 言語は「金曜の夜にゲームギミックのアイデアを思いついたプランナーが、土日で動作するデモを作り、月曜朝の会議で披露できる」ぐらいの速度感を目指しています。本来ゲームとはこれくらい簡単に作れる筈の物で、実際に tsukasa 言語を触って貰えれば分かると思いますが、この目標は達成出来ていると考えています（でも、土日は休みましょうね）。

第2部
司エンジンの使い方

司エンジンはオープンソースで開発されている、tsukasa 言語の実装系です。

tsukasa 言語はメッセージ指向を採用したプログラミング言語で、一般的なプログラミング言語に慣れている人は、初めはその挙動に戸惑うかもしれません。

第２部では、サンプルコードを例に、tsukasa 言語が持つ多彩な機能を紹介します。

tsukasa 言語の概要と構造

tsukasa 言語は汎用的なゲーム開発を目的に設計されたメッセージ指向のコンピューター言語です。「メッセージ指向言語」というのはオブジェクト指向言語の特殊な形式で、全体的にはオブジェクト指向言語に似ていながらも、一部で大きな違いがあります。その為、Ruby や C++ などのオブジェクト指向言語を知っている人は、当初は tsukasa 言語の挙動に戸惑うかもしれません。

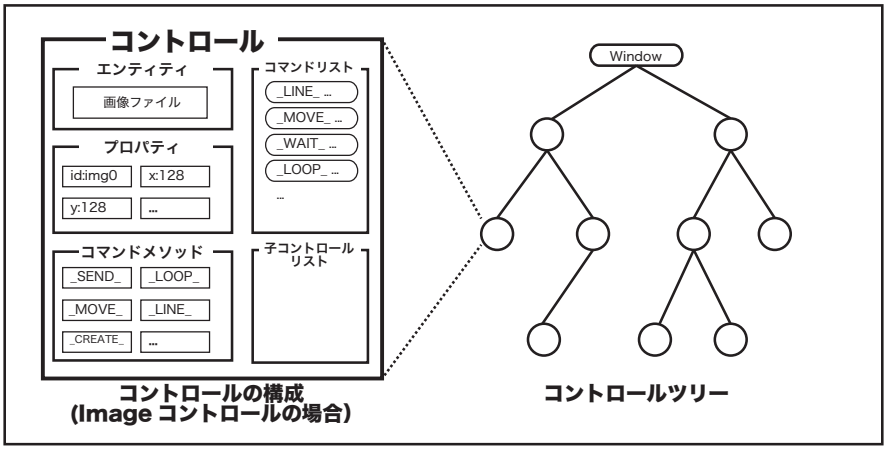
ここでは tsukasa 言語のコア概念「コントロール」「コントロールツリー」「コマンド」について解説します。

コントロール

tsukasa 言語では、ゲームを構成する全ての要素を「コントロール」という単位で管理します。

コントロールには画像ファイルを管理するコントロール、BGM ファイルを管理するコントロール、複数のコントロールを制御するためのコントロール等があります。コントロールの種類によって機能の差異がありますが、全て統一されたルールで制御できます。

コントロールの構成要素を以下に示します。



コントロールの各要素は以下の用途で使われます。

構成要素	意味
エンティティ	画像や BGM など、ゲームで使用するリソースファイル。
プロパティ	画像を配置する座標、透明度、可視設定などの値
コマンドメソッド	そのコントロールに対して行えるコマンド群
コマンドリスト	外部から送信された、そのコントロールが処理する予定のコマンドのリスト
子コントロールリスト	そのコントロールが下位に保持しているコントロールのリスト

コントロールツリー

全てのコントロールは「コントロールツリー」と呼ばれる単一のツリーに登録されます。コントロールツリーの最上位は「Window」という、ゲームそのものを表す特別なコントロールが設定されています。

コマンド

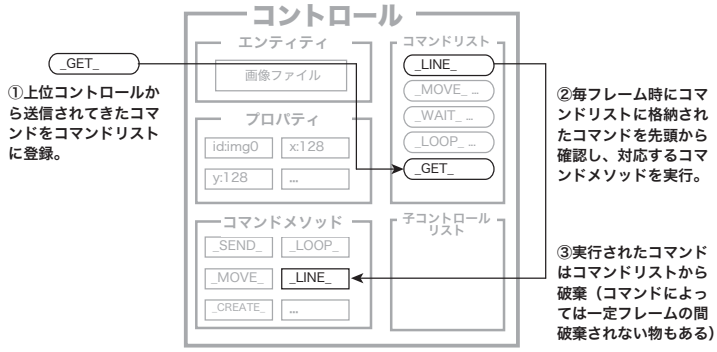
コントロールは外部から「コマンド」を発行することで操作できます。コマンドはツリーの上位から下位に向けて送信されます（下位から上位に送信する方法もあります）。

送信されたコマンドは即時に実行されるわけではなく、コマンドリストと呼ばれるキューに格納されます。すでにキューにコマンドが格納されている場合、その後ろに連結します。1度に複数のコマンドを送信することもできます。

毎フレーム時の挙動

tsukasa 言語では、コントロールツリーを毎フレーム巡回し、全てのコントロールに対して更新処理をかけます。その後、描画要素を持つコントロールについては画面に描画します。

更新処理をかけられた個々のコントロールは、一種の仮想計算機（VM:Virtual Machine）として動作します。コマンドリストに格納されているコマンド群を自身に対するプログラムであるとみなし、個々のコマンドに対応するコマンドメソッドを実行します。



コマンドの送信と実行

実行されたコマンドは自動的にコマンドリストから取り除かれます。コマンドによっては、条件を満たすまでコマンドリストに残り続ける物もあります。例えば、コントロールのプロパティに値を設定する _SET_ コマンドは、1回実行されるとリストから削除されますが、プロパティの値を連続的に遷移させる _MOVE_ コマンドは、指定されたフレーム数の間はリストに残り続けます。

また、tsukasa言語では条件分岐や繰り返しなどの制御構文もコマンドメソッドとして実装しています。そのためプログラムの制御構造自体を、コマンドとして送信することができます。「ソースコードそのものをコントロール間で送受信できる」と考えると、イメージしやすいかもしれません。

まとめ

tsukasa 言語では個々のコントロールが、いわば「再プログラム可能な状態遷移機械」として作られており、それによって、ゲームプログラミング特有の「個々のオブジェクトが膨大な状態を管理しなければならず実装が複雑になる」という問題が最小になるように設計されています。

また、全てのコントロールが単一のツリーに登録されている為、任意のノード以下をまるごと不可視にするなどして、ゲームプログラミングで頻出するシーンの管理なども容易に実装することが期待できます。

次項から、サンプルコードを例に tsukasa 言語のコードの書き方を解説していきます。

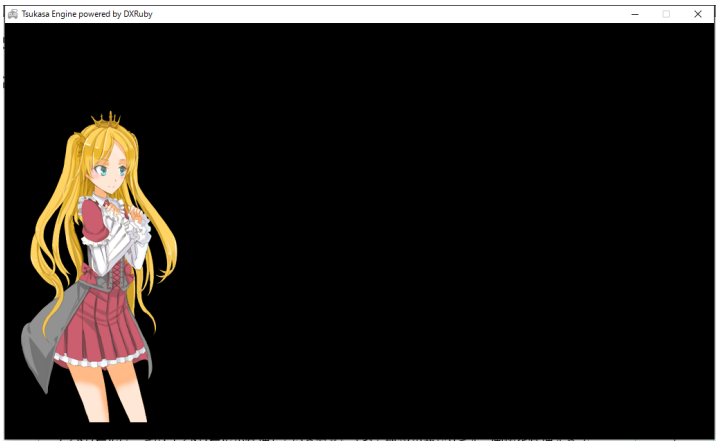
サンプル 1：コントロールの操作

コントロールを操作して、画面に画像を表示させてみます。コントロールは tsukasa 言語においてゲームを構成する要素を管理する最小の単位です。

サンプルコード (sample_1_1.rb)

```
_SEND_ :img0 do
  _SET_ path: "./resource/char/A-1.png"
end
```

実行すると画面に女の子の画像が表示されます。



コントロールとは

tsukasa 言語では、ゲームを構成する全ての要素を「コントロール」という単位で管理しています。「画像1つにつき1つのコントロールが用意されている」と考えると分かりやすいかと思います。

画像以外でも、例えばＢGMやＳＥのように絵を持たない物もコントロールです。テキストウィンドウもコントロールですし、その中に含まれる文字1つ1つも、それぞれコントロールです。

コントロールは、そのコントロールが管理している要素によって種類が異なります。画像を管理するコントロールは Image と言います。音声ファイルを管理するコントロールは Sound です。他にもテキストウィンドウを管理するコントロールや、複数のコントロールのグルーピングを目的としていてそれ自体は要素を持たないコントロール等もあります。

司エンジンの起動直後は、以下の5つのコントロールがあらかじめ用意されています。ユーザーは必要に応じて新たなコントロールを生成できます。

id 名	コントロールの種類

base	Image
img0	ImageI
img1	Image
img2	Image
text0	TextPage

コントロールには id という個別の名前を設定して、それぞれを区別します。

base/img0/img1/img2 は Image コントロールです。中身は空で、画像へのファイルパスを指定することで画像を表示します。base は背景画像を、img0/img1/img2 にはキャラクター画像を設定します。これは単に描画順の話で、機能上の区別はありません（描画順については後述します）。

text0 はテキストウィンドウを管理するコントロール TextPage です。tsukasa 言語ではゲームを構成する要素は全てコントロールなので、テキストウィンドウもコントロールです。また、コントロールは複数のコントロールを子要素として持つことで、より複雑なコントロールを構築できます。

解説

それでは改めてコードを見てみましょう。

```
_SEND_ :img0 do
  _SET_ path: "./resource/char/A-1.png"
end
```

このスクリプトでは、_SEND_ コマンドを使って img0 というコントロールにコマンドを送信しています。_SEND_ コマンドの第1引数にコントロール名（ここでは img0）を指定すると、do 〜 end で囲まれた箇所（Ruby 用語で「ブロック」と言います）が、指定したコントロールに送信されます。ここでは1行だけ送信していますが、複数のコマンドをまとめて送信する事も出来ます。

img0 コントロールに送信しているコマンドは以下になります。

```
_SET_ path: "./resource/char/A-1.png"
```

SET は全てのコントロールが持っている組み込みのコマンドで、コントロールが持つプロパティに値を設定します。プロパティはコントロールの各種情報を保持していて、それらを書き換えることでコントロールを制御できます。

プロパティの指定は以下の記法で行います。":" を忘れないでください（Ruby のハッシュ簡略記法です）

```
[ プロパティ名 ]: [ 設定する値 ]
```

ここでは path というプロパティに画像ファイルへのパスを設定し、それによって画像が表示されたわけです。今回は path のみを更新しましたが、1回の _SET_ で複数のプロパティを同時に更新できます。

コントロールごとに設定出来るプロパティの種類は変わります。それぞれのコントロールがどのようなプロパティを持っているかについては、リファレンスマニュアルを参照してください。

サンプル２：コントロールを移動させる

表示した Image を動かしてみます。サンプル1のスクリプトの、ブロックの中に1行追加します。ブロック内に記述したコマンドは、全て img0 コントロールに送信されます。

サンプルコード (sample_1_2.rb)

```
_SEND_ :img0 do
  _SET_ path: "./resource/char/A-1.png"
  _MOVE_ [120, :out_quart], x: [0, 500], y: [0, -100]
end
```

実行すると、画像が右上方向に移動します。移動速度が徐々に遅くなっているのが分かります。

解説

今回追加した行は如何になります。

```
  _MOVE_ [120, :out_quart], x: [0, 500], y: [0, -100]
```

追加したのはこの1行です。"_MOVE_" はコントロールを移動させるコマンドです。
第1引数に配列を設定し、第1引数で移動フレーム数 (120)、第2引数でイージングオプションを設定しています。

"x: [0, 500], y: [0,-100]" は各プロパティの始点と終点を指定しています。これによって、コントロールは [0,0] から [500,-100] に移動します (y がマイナスなのがわかりにくくてすみません。元の画像は上部に大きな空白があるのです)。

補足：イージングオプション

物体は移動する時に慣性がかかるので、力をかけると最初はゆっくり動き始め、途中から等速になり、停止する時にまたゆっくりになります。また、例えば投球のフォームでは腕が早く動く所とゆっくり動く所があります。このような動きの緩急を擬似的に再現するのがイージングです。

第1引数に設定されている配列の２番目の要素を削除するとイージングなし、つまり等速直線運動になります。違いを確認してみてください。

イージングをかけるとメリハリの利いた移動表現が可能になります。_MOVE_ コマンドでは約３０種類のイージングオプションを用意していて、状況に適した慣性移動を設定できます。詳しくはリファレンスを参照してください。

サンプル３：コントロールを曲線移動させる

次は線形補完です。くだけた言い方をすると曲線移動です。

サンプルコード (sample_1_3.rb)

```
_SEND_ :img0 do
  _SET_ path: "./resource/button_normal.png"
  _PATH_ 300,
    x: [  0, 30,240, 60,410,450],
    y: [  0, 90,150,480,120,480]
end
```

実行すると、画像がぐにゃぐにゃとカーブを描きながら移動します。

解説

PATH は曲線移動をするためのコマンドです。第1引数で指定したフレーム数をかけて、各オプションの値を配列の値に沿って滑らかに遷移させていきます。

値の遷移式には3次Bスプライン関数を使用しています。遷移式は任意の物を設定できます（設定方法についてはリファレンスを参照）が、標準で用意しているのは3次Bスプライン関数のみです。

３次Bスプライン関数は直感的に座標を指定できるのが特徴です。ただし、指定する座標は補助点になるため、実際の曲線はその座標を通過しないという欠点があります。様々な値を設定して、使い方を探ってみてください。

サンプル 4：フェードイン・アウト

画像のフェードイン・アウトをしてみましょう。

サンプルコード (sample_1_4.rb)

```
_SEND_ :base do
  _SET_ path: "./resource/bg_sample.png"
end

_SEND_ :img0 do
  _SET_ path: "./resource/button_over.png", x: 200, y: 100
  _MOVE_ 180, alpha: [255,   0]
  _MOVE_ 360, alpha: [  0, 255]
end
_SEND_ :img1 do
  _SET_ path: "./resource/button_normal.png", x: 100, y: 200
  _MOVE_ 180, alpha: [255,   0]
  _MOVE_ 360, alpha: [  0, 255]
end
```

実行すると、背景画像の手前で、２つの重なった正方形がフェードアウトして消え、その後フェードインして再び表示されます。"base" は予め用意されている背景用の Image コントロールです。描画順が１番後ろになっていること以外に、他の Image と違いはありません。

解説

画像を読み込んだ img0 と img1 それぞれにコマンドを送信しています。それぞれのコントロールに送信するスクリプトはファイル名と座標以外同じ物なので、ここでは img0 の中身について説明します。

```
  _SET_ path: "./resource/button_over.png", x: 200, y: 100
```

まず画像ファイルを読み込みます。x と y は画像の描画座標を指定します。

```
  _MOVE_ 180, alpha: [255,   0]
  _MOVE_ 360, alpha: [  0, 255]
```

フェードイン／アウトをするには、移動と同じ _MOVE_ コマンドを使います。alpha が透明度をあらわすプロパティです。180 フレームかけて 255 から 0 まで遷移させます。

MOVE コマンドは終了するまでコマンドリストに残り、コントロールの処理はそこで終了します。それ以降のコマンドは _MOVE_ が終了するまで実行されません。

フェードアウトが終わったら、今度は 360 フレームかけて再びフェードインさせます。

サンプル５：コントロールの生成

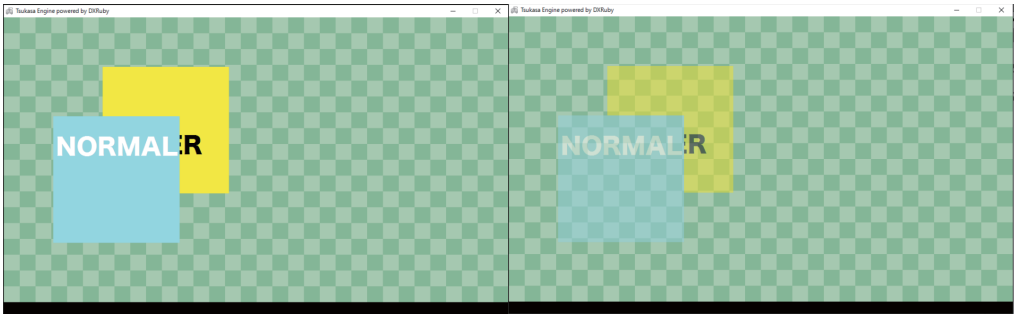
サンプル４では、２つの画像に対して同じコード（フェードアウト／待機／フェードイン）を記述していて冗長でした。また、画像それぞれに透明度を設定していたため、２つの画像が重なっている所で、下の四角形が透けて見えてしまっていて、不自然な表示になっていました。

今回は DrawableLayout コントロールを使ってこれらに対応します。

サンプルコード (sample_1_5.rb)

```
_SEND_ :base do
  _SET_ path: "./resource/bg_sample.png"
end

_CREATE_ :DrawableLayout, id: :test0,
  x: 100, y: 100, width: 512, height: 512, z: 4000 do
  _CREATE_ :Image, path: "./resource/button_over.png", x: 100
  _CREATE_ :Image, path: "./resource/button_normal.png", y: 100
end

_SEND_ :test0 do
  _MOVE_ 180, alpha: [255,   0]
  _MOVE_ 360, alpha: [  0, 255]
end
```

見た目はほぼ前回と同じなのですが、正方形の画像が重なり合っている箇所が透けていません。２枚の画像をグルーピングして1枚として扱っているからです。



解説

```
_CREATE_ :DrawableLayout, id: :test0,
  x: 100, y: 100, width: 512, height: 512, z: 4000 do
  _CREATE_ :Image, path: "./resource/button_over.png", x: 100
```

```
  _CREATE_ :Image, path: "./resource/button_normal.png", y: 100
end
```

ここでは2つのImageコントロールをグルーピングするためにDrawableLayoutというコントロールを生成しています。これは透明なImageコントロールのような物だと考えてください。このコントロールは、img0/1のように標準では用意されていないので、_CREATE_コマンドを使って新たに生成します。

_CREATE_コマンドは、第1引数で生成するコントロールの種類（ここではDrawableLayout）を指定します。その後、キーワード引数で必要なパラメータを設定します。ここでは順番にid、X座標、Y座標、コントロールの幅、高さ、描画順序を指定しています（個々のパラメータの意味についてはリファレンスマニュアル参照）。

do～endブロック内に記述されたスクリプトは、コントロールが生成されると同時にそのコントロール上で実行されます。ここでは画像コントロールを2つ生成し、それぞれファイルパスと座標を指定しています。ここで指定した座標はDrawableLayoutの左上を原点とした物になります。

```
_SEND_ :test0 do
  _MOVE_ 180, alpha: [255,   0]
  _MOVE_ 360, alpha: [  0, 255]
end
```

生成したtest0に対してコマンドを送信します。個々のImageコントロールではなく、それらをグルーピングしているtest0にコマンドを送信しているので、同じコードを2回書く必要がありません。このようにグルーピングすることで複数のコントロールを効率的に管理できます。

補足：コントロールの親子関係

全てのコントロールは複数の子コントロールを持つことができます。描画機能を持つコントロールは親コントロールに対して相対座標で管理されます。親コントロールの可視不可視設定は子コントロールにも伝搬します。また、子コントロールの更新処理を停止／再開することもできます。シーン管理やメニュー表示など、様々な利用法を考えてみてください。

サンプル６：フェードトランジション

ここから数回にかけてトランジションについて解説します。トランジションというのは、ある画像を別の画像に滑らかに切り替える演出のことを言います

ノベルゲームでは様々な方式のトランジションを使いますが、まずはキャラ立ち絵のポーズや表情を切り替える際に使うフェードトランジションから見ていきます。フェードトランジションは、元の画像をフェードアウトするのと同時に新しい画像をフェードインさせることで画像を自然に切り替える手法です。

サンプルコード (sample_1_6.rb)

```
_SEND_ :base do
  _SET_ path: "./resource/bg_sample.png"
end
_SEND_ :img0 do
  _SET_ path: "./resource/char/B-1.png", x: 250
end
_SEND_ :img1 do
  _SET_ path: "./resource/char/B-2.png", x: 250, visible: false
end

_WAIT_ input: {key_push: K_SPACE}

_SEND_ :img0 do
  _MOVE_ 360, alpha: [255, 0]
end
_SEND_ :img1 do
  _SET_ visible: true
  _MOVE_ 360, alpha: [0,255]
end
```

起動するとキャラの立ち絵画像が表示されます。スペースキーを押すとその画像がフェードアウトし、別の画像がフェードインしてきます。

解説

```
_SEND_ :base do
  _SET_ path: "./resource/bg_sample.png"
end
_SEND_ :img0 do
  _SET_ path: "./resource/char/B-1.png", x: 250
end
_SEND_ :img1 do
  _SET_ path: "./resource/char/B-2.png", x: 250, visible: false
end
```

まずImageコントロールに画像を設定します。visibleプロパティはコントロールの可視設定を行う物で、falseを設定するとそのコントロール（及び子コントロール）を描画対象から外します。その為、この段階ではbase0の背景画像とimg0のB-1.pngのみが画面に表示されます。

```
_WAIT_ input: {key_push: K_SPACE}
```

_WAIT_コマンドでキー入力を待機します。_WAIT_コマンドではハッシュで指定した条件が成立するまで待機します。条件が成立するまでコマンドリストの探査は中断され、_WAIT_コマンドより後にコマンド群が存在しても、それらは実行されません。

ユーザーからの入力はinputプロパティにハッシュを指定して行います。ここでは入力を確認する為にkey_push条件にK_SPACEを設定しています。K_SPACEはスペースキーを表すキーコード定数です。キーコード定数はキーボード上の各キーに対応する値が設定された定数です（リファレンスマニュアル参照）。複数のキーを同時に判定の対象にすることもできます。

```
_SEND_ :img0 do
  _MOVE_ 360, alpha: [255, 0]
end
_SEND_ :img1 do
  _SET_ visible: true
  _MOVE_ 360, alpha: [0,255]
end
```

スペースキーを押下すると待機状態が解除され、フェードを開始します。img0は透明度を255から0へ、img1は可視設定をONにしてから透明度を0から255に遷移させています。360フレームかけてトランジションが実行されます。

まとめ

ゲームではタイミングの制御がとても重要です。_WAIT_コマンドは非常に多くの条件項目を持ち、ユーザーの入力やゲームの現在の状態を細かく判定できます。詳細についてはリファレンスマニュアルを参照してください。

サンプル７：より自然なフェードトランジション

サンプル６のフェードトランジションでは、２つの立ち絵がどちらも半透明になっている時、よく見ると立ち絵画像が重なっている箇所で背景画像が透けて見えてしまっています。どちらの絵も半透明になっているのだから背景が透けて見えるのは当然なのですが、ノベルゲームのキャラ立ち絵切り替えで使う場合、このままだと少し違和感があります。

一般的なノベルゲームでは、このようにキャラ立ち絵が透けて背景が見えてしまうようなことは起きず、より自然なフェードトランジションになっています。どうすればこれを実現できるのでしょう？

国産の代表的なＡＤＶフレームワークである「吉里吉里２」では、「backray」という機能でこれを実現しています。backray は大まかに以下のような仕組みになっています。

１・現在の画面をまるごと1枚の画像としてコピーし、それ（以下Aとする）を1番手前に表示する

２・Aの背後に隠れた各画像を、トランジション後に表示したい物に差し替える

３・Aをフェードアウトさせる

吉里吉里２では上記のような手法で極めて自然、かつ低コストで（乱暴に言えば透明度の演算コストを半分にして）画面全体のトランジションを実現しています。スクリプトの記述も直感的で、大変優れた実装、設計思想であると言えます。

司エンジンでは、吉里吉里２の backray とほぼ同じ機能を _TO_IMAGE_ コマンドで実現します。このサンプルでは、_TO_IMAGE_でトランジションを行う方法について解説します。

サンプルコード (sample_1_7.rb)

```
_SEND_ :base do
  _SET_ path: "./resource/bg_sample.png"
end
_SEND_ :img0 do
  _SET_ path: "./resource/char/B-1.png", x: 250
end
_WAIT_ input: {key_push: K_SPACE}

_TO_IMAGE_ :test0, height: 1024, width: 600

_SEND_ :img0 do
  _SET_ path: "./resource/char/B-2.png"
end

_SEND_ :test0 do
  _MOVE_ 360, alpha: [255, 0]
end
```

実行すると、背景画像が透けることなく、自然にトランジションが行われているのが分かります。

解説

途中までは前回と同じです。今回は img1 は使いません。

```
_TO_IMAGE_ :test0, height: 1024, width: 600
```

_TO_IMAGE_ はデフォルトで用意されているユーザー定義コマンドです。_TO_IMAGE_ を実行すると以下のような処理が実行されます。

1・オプションで指定したサイズの Image コントロール（以下Bとする）を第1引数で指定した id 名で生成する。
2・Bに自身と自身の子コントロールを全て描画する
3・Bを自身の子コントロールリストの末端にZ指定 1000000 で追加する

「現在のコントロール」というのは、そのスクリプトが実行されているコントロールを表します。フラットな位置に記述されたコマンドは、全てコントロールツリーのルートである Window コントロールに送信されています。

Window コントロール上で _TO_IMAGE_ を実行することで、全ての描画コントロールをコピーした Image コントロールを生成し、それを子コントロールリストの末端に追加します。この時点で画面に表示されている画像は test0 であり、base と img0 は test0 の背後に隠れています。

```
_SEND_ :img0 do
  _SET_ path: "./resource/char/B-2.png"
end
```

test0 によって他のコントロールが隠れているので、その間に img0 が保持している立ち絵を差し替えます。path にファイルパスを設定すると、自動的に画像を読み込み直します。

```
_SEND_ :test0 do
  _MOVE_ 360, alpha: [255, 0]
end
```

test0 に _MOVE_ コマンドを送信します。test0 の透明度がフェードで 0 に遷移していくと、徐々に背後の画像が浮かび上がってきます。その間 img0 の透明度は変化しないので、背景が透けることはなく、自然なフェードトランジションを実現できます。

補足

フェードトランジションが終わった後は test0 を削除し、これを繰り返すことで、立ち絵を次々と切り替えていけます。

今回はルートコントロールの Window に対して _TO_IMAGE_ コマンドを実行したので、画面全体が対象になりましたが、もちろんツリーの一部に対して実行することも可能です。

サンプル８：ルールトランジション

次はトランジションのバリエーションであるルールトランジションについて説明します。

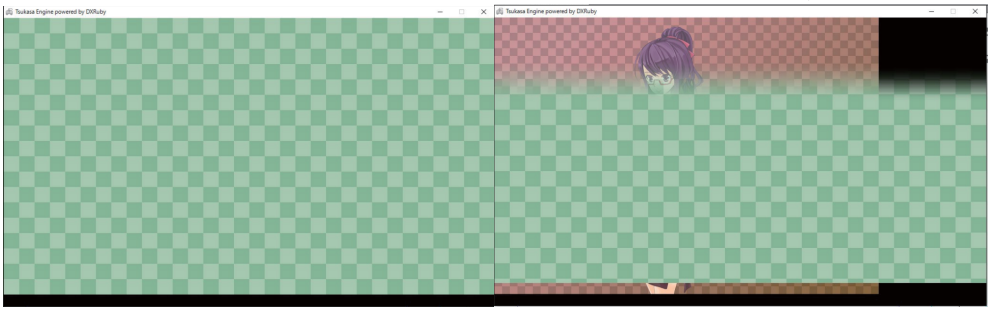
ルールトランジションは、予めルール画像と呼ばれるグラデーションの画像を用意しておき、そのグラデーションデータに応じて画面にフェードをかける物です。

余談ですが、ルールトランジションというのは吉里吉里で使われている用語で、一般的な物ではないかもしれません。

サンプルコード (sample_1_8.rb)

```
_CREATE_ :DrawableLayout, id: :test0, width: 800, height: 600 do
  _CREATE_ :Image, path: "./resource/bg_test.jpg"
  _CREATE_ :Image, path: "./resource/char/B-1.png", x: 250
end

_CREATE_ :Image, id: :test1, path: "./resource/bg_sample.png" do
  _CREATE_ :RuleTransition, id: :rule0, vague: 40,
           path: "./resource/rule/horizontal_rule.png"
  _GET_ :shader, control: :rule0 do |shader:|
    _SET_ shader: shader
  end
end

_WAIT_ input: {key_push: K_SPACE, mouse: :push}

_SEND_ :test1 do
  _SEND_ :rule0 do
    _MOVE_ 240, counter:[0,255]
    _DELETE_
  end

  _WAIT_ count: 240
  _DELETE_
end
```

スペースキーを押すと上から下にフェードがかかり、新しい背景画像とキャラが表示されます。



解説

今回使用しているルール画像を以下に示します。これは透明度に0～255を設定したpng画像です。ルール画像では透明度以外のパラメータは無視されます。ユーザーは任意のルール画像を作成できます。

ではコードを見ていきましょう。

```
_CREATE_ :DrawableLayout, id: :test0, width: 800, height: 600 do
  _CREATE_ :Image, path: "./resource/bg_test.jpg"
  _CREATE_ :Image, path: "./resource/char/B-1.png", x: 250
end
```

トランジション後の画像（茶色背景＋キャラ立ち絵）を生成します。基本的に先に宣言した方がより奥に表示されます。

```
_CREATE_ :Image, id: :test1, path: "./resource/bg_sample.png" do
  _CREATE_ :RuleTransition, id: :rule0, vague: 40,
            path: "./resource/rule/horizontal_rule.png"
  _GET_ :shader, control: :rule0 do |shader:|
    _SET_ shader: shader
  end
end
```

トランジション前の画像(緑色背景）を test1 という id 名で生成します。この時、付与ブロックを使って _CREATE_ と _SET_ と _GET_ の2つのコマンドを実行します。

_CREATE_では RuleTransition を rule0 という id 名で生成します。これはルール画像を保持し、ルールトランジションを処理する為のコントロールです。vague はルールトランジションを実行する際の境界の曖昧さを指定するプロパティです。

その後、_GET_でその rule0 コントロールの shader プロパティからシェーダーを取得し、_SET_で shader プロパティに設定します。これによって、test1 に対して rule0 がルールトランジションとして適用されるようになります。

```
_WAIT_ input: {key_push: K_SPACE, mouse: :push}
```

キー入力を待った後に、test1 にコマンドを送信します。

```
_SEND_ :test1 do
  _SEND_ :rule0 do
    _MOVE_ 240, counter:[0,255]
    _DELETE_
  end
```

test1 内部で更に rule0 に _MOVE_ コマンドを送信します。240 フレームかけて counter の値を 0 から 255 に遷移させます。これによって、test1 に対してルールトランジションが適用され、画面の上から下に画像が切り替わっていきます。

MOVE が終わるとコントロールを削除する _DELETE_ コマンドが実行されます。これによって rule0 が削除されます。

```
  _WAIT_ count: 240
  _DELETE_
end
```

test1 の方でも 240 フレーム待機した後に自分自身を削除します。これによってトランジションが完了しました。

まとめ

ルールトランジションは工夫次第で大きなインパクトを持つ演出をユーザに提供できるため、多くのノベルゲームで活用されています（うずまきに飲み込まれるように場面が転換する演出を見た事ある方も多いのではないでしょうか。あれもルールトランジションです）。効果的なルール画像を作ってみてください。

補足

ルール画像によるトランジションには、DXRuby が提供している HLSL アクセス機能 shader を利用しています。同じことを ruby の機能のみで実現しようとすると、恐らくパフォーマンスがまったく追いつきません。shader は DXRuby で Ruby ゲーム開発をする際の大きなアドバンテージと言えるでしょう。

現在の Shader へのアクセス機能はまだ限定的ですが、将来的にはピクセルを直接いじるタイプのトランジションや、遷移前と遷移後の画像を混在させるタイプのトランジションなども用意したいと思っています。

サンプル９：ユーザー定義コマンドの宣言と実行

サンプル８のコードを見ると分かる通り、多少複雑なことをしようとすると、スクリプトはすぐに記述が何行にもわたってしまいます。ルールトランジションを行う度に同じスクリプトを書いていたら、あとで修正が必要になった時にその全部を直すことになって面倒ですよね。

このような場合、tsukasa 言語では幾つかのコマンドをまとめた「ユーザー定義コマンド」を作り、必要に応じてそれを呼びだすことができます。

サンプルコード (sample_1_9.rb)

サンプルコードはこちら。挙動自体はサンプル８とまったく同じです。

```
# ルール画像を設定するユーザー定義コマンド
_DEFINE_ :set_rule do |options|
  _CREATE_ :RuleShader, id: options[:id], vague: options[:vague] || 40,
            path: options[:path]
  _GET_ :shader, control: options[:id] do |shader:|
    _SET_ shader: shader
  end
end
_CREATE_ :DrawableLayout, id: :test0, width: 800, height: 600 do
  _CREATE_ :Image, path: "./resource/bg_test.jpg"
  _CREATE_ :Image, path: "./resource/char/B-1.png", x: 250
end
_CREATE_ :Image, id: :test1, path: "./resource/bg_sample.png" do
  set_rule id: :rule0, vague: 40,
           path: "./resource/rule/horizontal_rule.png"
end
_WAIT_ input:{key_push: K_SPACE, mouse: :push}
# ルールトランジションを実行するユーザー定義コマンド
_DEFINE_ :go_rule do |options|
  _SEND_ :test1 do
    _SEND_ options[:rule_id] do
      _MOVE_ options[:time], counter:[0,255]
      _DELETE_
    end
    _WAIT_ count: options[:time]
    _DELETE_
  end
end
go_rule rule_id: :rule0, time: 240
```

解説

それでは解説していきます。前回と同じ箇所については省略します。

```
# ルール画像を設定するユーザー定義コマンド
_DEFINE_ :set_rule do |options|
  _CREATE_ :RuleShader, id: options[:id], vague: options[:vague] || 40,
           path: options[:path]
  _GET_ :shader, control: options[:id] do |shader:|
    _SET_ shader: shader
  end
end
```

まずルール画像を設定するためのユーザー定義コマンドを宣言します。

ユーザー定義コマンドの宣言には _DEFINE_ コマンドを使用します。第1引数でコマンドの名前 set_rule をシンボルで指定し、ブロックでコマンドの中身を記述します。ブロック引数には、ユーザー定義コマンドを呼び出した際のハッシュが設定されます。ここでは id、vague、path が取得できます。

```
_CREATE_ :Image, id: :test1, path: "./resource/bg_sample.png" do
  set_rule id: :rule0, vague: 40,
           path: "./resource/rule/horizontal_rule.png"
end
```

test1 の生成時に先ほど定義した set_rule コマンドを実行します。

```
# ルールトランジションを実行するユーザー定義コマンド
_DEFINE_ :go_rule do |options|
  _SEND_ :test1 do
    _SEND_ options[:rule_id] do
      _MOVE_ options[:time], counter:[0,255]
      _DELETE_
    end
    _WAIT_ count: options[:time]
    _DELETE_
  end
end
```

今度はルールトランジションを実行するユーザー定義コマンドを宣言します。

```
go_rule rule_id: :rule0, time: 240
```

宣言したコマンドを呼びだし、ルールトランジションが実行されます。

補足

ユーザー定義コマンドを定義する _DEFINE_ 自身もコマンドなので、コントロールに送信できます。その場合、そのコントロール内でのみ使用できるユーザー定義コマンドになります。ルートで宣言したユーザー定義コマンドは、全てのコントロールで使用できます。

サンプル10：データストアへの読み書きと動的なテキスト出力

主人公の名前を任意に設定できるタイプのゲームでは、その名前を内部で保存し、必要に応じて取り出せる必要があります。tsukasa 言語では、「データストア」という仕組みでこれを実現します。

ここではデータストアへの値の読み書きの基本と、データストアに格納した値を取得してテキストウィンドウに出力する方法を説明します。

サンプルコード (sample_1_10.rb)

```
_SET_ [:_ROOT_, :_TEMP_], name_a: "土屋"
_SEND_ :text0 do
  _GET_ :name_a, control: [:_ROOT_, :_TEMP_] do |name_a:|
    _TEXT_ name_a
  end
  _TEXT_ "「_SET_ コマンドはデータストアに値を格納します」"
end
_END_PAUSE_

_GET_ :name_a,  control: [:_ROOT_, :_TEMP_] do |name_a:|
  _SET_ [:_ROOT_, :_TEMP_], name_b: name_a
end
_SEND_ :text0 do
  _FLUSH_
  _GET_ :name_b, control: [:_ROOT_, :_TEMP_] do |name_b:|
    _TEXT_ name_b
  end
  _TEXT_ "「_GET_ コマンドはデータストアから値を取得します」"
end
```

実行すると、先頭に「土屋」という文字が追加されてテキストが表示されます。

解説

```
_SET_ [:_ROOT_, :_TEMP_], name_a: "土屋"
```

SET の第1引数にパス名を指定すると、そのコントロールのプロパティへ書き込みが行われます。ここでは、Data コントロールの _TEMP_ コントロールを選択し、その name_a プロパティに "土屋" を設定しています。

Data コントロールのプロパティは、ユーザーが任意に作成できるので、ゲーム中に使用する値を保存するのに使用できます。標準では _TEMP_/_SYSTEM_/_LOCAL_ の3種類が用意されています。

```
_SEND_ :text0 do
  _GET_ :name_a, control: [:_ROOT_, :_TEMP_] do |name_a:|
    _TEXT_ name_a
  end
  _TEXT_ "「_SET_ コマンドはデータストアに値を格納します」"
end
_END_PAUSE_
```

SET があれば _GET_ もあります。_SET_ がコントロール／データストアのプロパティに値を設定するのに対し、_GET_ は逆にコントロール／データストアのプロパティから値を取得します。

第1引数に name_a を、control プロパティに [:_ROOT_, _TEMP_] を設定することで、_TEMP_ コントロールの name_a プロパティの中身を取得し、ブロック引数に渡しています。第1引数は配列にも対応していて、複数の値を1度に取得できます。

ここでは name_a に格納されていた「土屋」を出力しています。

```
_GET_ :name_a,  control: [:_ROOT_, :_TEMP_] do |name_a:|
  _SET_ [:_ROOT_, :_TEMP_], name_b: name_a
end
```

ここでは name_a の値を別のプロパティ name_b に設定し直しています。

```
_SEND_ :text0 do
  _FLUSH_
  _GET_ :name_b, control: [:_ROOT_, :_TEMP_] do |name_b:|
    _TEXT_ name_b
  end
  _TEXT_ "「_GET_ コマンドはデータストアから値を取得します」"
end
```

コピーした name_b をテキストウィンドウに出力します。

サンプル11：条件分岐

これまで説明してきたように、tsukasa 言語ではコントロールへの制御は、そのコントロールに対するコマンドの送信で行います。コマンドの中には条件分岐を扱うコマンドやループを扱うコマンドもあり、これらを使うことでコントロールのより複雑な制御を行えます。条件分岐の動作を確認しましょう。

サンプルコード (sample_1_11.rb)

```
_SEND_ :text0 do
  _TEXT_ " Ｘか Ｚのキーを押してください "
  _LINE_FEED_
end
_WAIT_ input:{key_down: [K_X,K_Z]}
_CHECK_INPUT_ key_down: [K_X] do
  _SEND_ :text0 do
    _TEXT_ " Ｘキーが押されました "
  end
end
_CHECK_INPUT_ key_down: [K_Z] do
  _SEND_ :text0 do
    _TEXT_ " Ｚキーが押されました "
  end
end
```

キーボードの「X」か「Z」を押すと、押したキーに応じたテキストが表示されます。

解説

```
_SEND_ :text0 do
  _TEXT_ " Ｘか Ｚのキーを押してください "
  _LINE_FEED_
end
_WAIT_ input:{key_down: [K_X,K_Z]}
```

テキストを1行表示してから _WAIT_ コマンドで待機状態に入ります。key_down は指定したキーが押下されると成立します。ここでは X と Z キーに対応するキーコード定数 K_X と K_Z を指定しています。

```
 _CHECK_INPUT_ key_down: [K_X] do
  _SEND_ :text0 do
    _TEXT_ " Ｘキーが押されました "
  end
end
```

```
_CHECK_INPUT_ key_down: [K_Z] do
  _SEND_ :text0 do
    _TEXT_ " Zキーが押されました "
  end
end
```

_CHECK_INPUT_ はキー入力の条件分岐を行うコマンドで、条件が成立している時のみブロックを実行します。条件判断の書式は _WAIT_ と同じです。_WAIT_ と違い、成立の有無に拘わらず処理が先に進みます。

1つ目の _CHECK_INPUT_ ではXキー、2つ目の _CHECK_INPUT_ ではZキーが押下されているかをチェックし、押下されていれば対応するテキストを出力します。

補足

条件分岐には、このサンプルで紹介した _CHECK_INPUT_ コマンドの他に _CHECK_ コマンドがあります。_CHECK_INPUT_ コマンドがユーザーからの入力を判定するのにたいし、_CHECK_ コマンドはコントロールやデータストアのプロパティの値を判定するのに使用します。詳細はリファレンスマニュアルを参照してください。

tsukasa 言語では、他の言語の条件分岐にあるような else 文に相当する物がありません。その代わり、ほとんどの条件項目にはその not 版が用意されているので、条件の正否によって異なる処理をさせたい場合、上記のように _CHECK_/_CHECK_INPUT_ を重ねて実行してください。

サンプル12：ループ

tsukasa 言語では、ループ文も、条件分岐と同じくコマンドの一種になります。ループを使うことでより複雑なゲームを作れるようになります。

サンプルコード (sample_1_12.rb)

```
_LOOP_ 3 do
  _LOOP_ 3 do
    _SEND_ :text0 do
      _TEXT_ " ＊ "
    end
  end
  _SEND_ :text0 do
    _LINE_FEED_
  end
end
```

実行すると " ＊ " が3個ずつ3行表示されます。

解説

```
_LOOP_ 3 do
```

ループを行うコマンドはそのものズバリの _LOOP_ です。第1引数に数字を指定するとその回数ブロックを再実行します。指定しない場合は無限にブロックを再実行します。

```
  _LOOP_ 3 do
    _SEND_ :text0 do
      _TEXT_ " ＊ "
    end
  end
```

次の _LOOP_ も3回繰り返し、「＊」を3個出力しています。これで1行分の出力がされました。

```
  _SEND_ :text0 do
    _LINE_FEED_
  end
```

改行コマンドを出力してブロックを終了します。

サンプル13：ボタンの制御とスクリプトの追加読み込み

ノベルゲームでは、選択肢の表示からコンフィグ設定まで、ユーザーからのクリックを受け付けたいタイミングが様々な箇所で登場します。ここではマウスクリックを受け付けて、押されたボタンによって異なるスクリプトを読み込んでみます。

サンプルコード (sample_1_13.rb/sample_1_13a.rb/sample_1_13b.rb)

今回のサンプルはファイルが3つに分かれています。

```
//sample_1_13.rb
_SET_ [:_ROOT_, :_TEMP_], click: nil
_IMAGE_BUTTON_ :button1, x: 50, y: 150 do
   _DEFINE_ :on_key_push_user do
     _SET_ [:_ROOT_, :_TEMP_], click: :left
   end
end

_IMAGE_BUTTON_ :button2, x: 450, y: 150 do
   _DEFINE_ :on_key_push_user do
     _SET_ [:_ROOT_, :_TEMP_], click: :right
   end
end

_WAIT_ [:_ROOT_, :_TEMP_], not_equal: {click: nil}

_CHECK_ [:_ROOT_, :_TEMP_], equal:{click: :left} do
  _INCLUDE_ "./script/sample/sample_1_13a.rb"
end
_CHECK_ [:_ROOT_, :_TEMP_],equal:{click: :right} do
  _INCLUDE_ "./script/sample/sample_1_13b.rb"
end
```

```
//sample_1_13a.rb
_SEND_ :text0 do
  _TEXT_ "左のボタンが押されました"
end
_END_PAUSE_
```

```
//sample_1_13b.rb
_SEND_ :text0 do
  _TEXT_ "右のボタンが押されました"
end
_END_PAUSE_
```

実行するとボタンが2つ表示され、押したボタンに応じてテキストが表示されます。

解説

```
_SET_ [:_ROOT_, :_TEMP_], click: nil
_IMAGE_BUTTON_ :button1, x: 50, y: 150 do
```

_IMAGE_BUTTON_ は標準で用意されているユーザー定義コマンドで、最低限の機能を持ったボタンを表示します。button1 という id 名を設定し、ＸＹ座標を設定しています。

```
   _DEFINE_ :on_key_push_user do
     _SET_ [:_ROOT_, :_TEMP_], click: :left
   end
end
```

_IMAGE_BUTTON_ の内部にコマンドを定義することで、ボタンの状態に応じてそのコマンドがコールバックできます。ここでは on_key_push_user コマンドを定義して、ボタン画像の上でマウスの左クリックが行われた場合の処理を記述しています。_TEMP_ データストアの click プロパティに、button1 は :left を、button2 は :right を設定します。

```
_WAIT_ [:_ROOT_, :_TEMP_], not_equal: {click: nil}
```

待機状態に入ります。not_equal は指定したプロパティが指定した値でなくなったら成立します。ここでは、_TEMP_ データストアの click プロパティに値が書き込まれるのを待ちます。ユーザーがボタンをクリックすれば click プロパティに値が設定され、それによって nil ではなくなるため、待機状態は終了します。

```
_CHECK_ [:_ROOT_, :_TEMP_], equal:{click: :left} do
  _INCLUDE_ "./script/sample/sample_1_13a.rb"
end
_CHECK_ [:_ROOT_, :_TEMP_],equal:{click: :right} do
  _INCLUDE_ "./script/sample/sample_1_13b.rb"
end
```

CHECK コマンドで click プロパティの値を判別し、対応するブロックを実行します。_INCLUDE_ は指定したファイルパスを新たなスクリプトとして宣言した位置に挿入します。

補足

INCLUDE はスクリプトファイルの挿入であって切り替えではないので、読み込んだスクリプトの処理が全て終了すると、元のスクリプトに戻って来ることに注意してください。

サンプル14：サウンドの再生

tsukasa 言語ではサウンドの管理もコントロールを介して行います。

サンプルコード (sample_1_14.rb)

```
_CREATE_ :Sound, path: "./resource/music/easygoing.ogg", id: :test0
_SEND_ :text0 do
  _TEXT_ " 初期化終了。スペースキーを押してください "
end
_END_PAUSE_
_SEND_ :test0 do
  _PLAY_ 0, fadetime: 5
end
_SEND_ :text0 do
  _FLUSH_
  _TEXT_ " 無限ループ中。スペースキーを押してください "
end
_END_PAUSE_
_SEND_ :test0 do
  _STOP_ fadetime: 5
end
```

スペースキーを押すと音楽ファイルがフェードインしながら再生され、その後無限ループします。その後もう1度スペースキーを押すとフェードアウトしながら停止します。

解説

```
_CREATE_ :Sound, path: "./resource/music/easygoing.ogg", id: :test0
```

tsukasa 言語ではＢＧＭやボイス、ＳＥを問わず、音源ファイルは Sound コントロールに登録します。ここでは "easygoing.ogg" という ogg ファイルを読み込み、test01 という id を設定しています。

```
_SEND_ :test0 do
  _PLAY_ 0, fadetime: 5
end
```

test01 コントロールにコマンドを送信します。_PLAY_ コマンドは音源ファイルを再生します。第1引数はループ回数を示し、０を設定すると無限ループになります。fadetime はフェードインに要する秒数を指定します。同様に _STOP_ コマンドは再生を停止します。

サンプル15：セーブとロード

tsukasa 言語ではノベルゲームで使われる一般的なセーブロードの機能を用意しています。

サンプルコード (sample_1_15.rb)

```
_SEND_ :text0 do
  _TEXT_ " XかZのキーを押してください "
  _LINE_FEED_
end

_WAIT_ input:{key_down: [K_X,K_Z]}
_CHECK_INPUT_ key_down: [K_X] do
  _SET_ [:_ROOT_, :_SYSTEM_], data0: K_X
end
_CHECK_INPUT_ key_down: [K_Z] do
  _SET_ [:_ROOT_, :_SYSTEM_], data0: K_Z
end

_SYSTEM_SAVE_  "./datastore/system_data.dat"
_HALT_
_SET_ [:_ROOT_, :_SYSTEM_], data0: "dummy"
_SYSTEM_LOAD_  "./datastore/system_data.dat"
_HALT_

_CHECK_ [:_ROOT_, :_SYSTEM_], equal: {data0: K_X} do
  _SEND_ :text0 do
    _TEXT_ " Xキーが押されました "
  end
end
_CHECK_ [:_ROOT_, :_SYSTEM_], equal: {data0: K_Z} do
  _SEND_ :text0 do
    _TEXT_ " Zキーが押されました "
  end
end
_END_PAUSE_
```

XかZのキーを押すと、押したキーに応じた文字列が表示されます。サンプル11と挙動は同じですが、内部的には一旦データをセーブし、それを再び読み込んで判定しています。

解説

条件分岐の項で説明した箇所については省略します。

```
_CHECK_INPUT_ key_down: [K_X] do
```

```
  _SET_ [:_ROOT_, :_SYSTEM_], data0: K_X
end
_CHECK_INPUT_ key_down: [K_Z] do
  _SET_ [:_ROOT_, :_SYSTEM_], data0: K_Z
end
```

押されたキーに応じて _SET_ コマンドで _SYSTEM_ データストアの data0 にキーコード定数を設定します。_TEMP_ データストアではなく _SYSTEM_ データストアを指定しているのは、_TEMP_ データストアに格納した値はセーブの対象にならないためです（詳細はリファレンスマニュアル参照）。

```
_SYSTEM_SAVE_  "./datastore/system_data.dat"
_HALT_ # 1フレーム送る
```

_SYSTEM_SAVE_ コマンドは、_SYSTEM_ の値を第1引数で指定したファイル名で保存します。

```
_SET_ [:_ROOT_, :_SYSTEM_], data0: "dummy"
```

セーブロードが実行されていることを確認する為、_SYSTEM_ の data0 プロパティにダミーの文字列を設定し、先に設定した値を削除しておきます。

```
_SYSTEM_LOAD_  "./datastore/system_data.dat"
_HALT_ # 1フレーム送る
```

保存したデータストアは _SYSTEM__LOAD_ コマンドで読み込みます。第1引数は _SYSTEM_SAVE_ と同じ物です。これによって、_SYSTEM_ にデータが読み込まれ、data0 プロパティの中身が "dummy" を設定する前に戻ります。

```
_CHECK_ [:_ROOT_, :_SYSTEM_], equal: {data0: K_X} do
  _SEND_ :text0 do
    _TEXT_ " Xキーが押されました "
  end
end
```

CHECK で data0 プロパティの中身を確認し、内容に応じてテキストを出力します。equal は指定したプロパティの値と比較し、同じであれば成立します。

補足

司エンジンは Ruby 上で動いており、エンジンが解釈する tsukasa 言語は実際には ruby スクリプトなので、_SYSTEM_SAVE_/_SYSTEM_LOAD_ コマンドを使わずとも、直接 Ruby でファイルを扱うことでセーブロードが行えます（というか、セーブロードに限らず、理屈では Ruby でできることは全部できます）。必要に応じて使い分けてください。

サンプル16：【tks スクリプト 1】文章の表示

司エンジンでは、tsukasa 言語の他に、拡張子が .tks の「tks スクリプト」という言語を処理できます。

tks スクリプトはノベルゲームなど、テキストの出力をメインで行うゲームに最適化して設計されています。これまでのサンプルでは、テキストを出力する為に、毎回 text0 コントロールにメッセージを送信し、_TEXT_ コマンドを実行していましたが、tks スクリプトではその記述を大幅に省略できます。

tks スクリプトは司エンジン内部で tsukasa 言語に変換され、透過的に処理されます。サンプル16と17では、tks スクリプトの挙動を紹介します。

サンプルコード (sample_1_16.tks)

```
（※１）１ページ目です。スペースキーを押すと
次のページに進みます。[ep]

（※２）２ページ目です。
改行が自動的に反映されます。[ep]

//（※３）"//" はコメント行として無視されます。

（※４）３ページ目です。[_LINE_PAUSE_]“_LINE_PAUSE_”コマンドを使うと、
行の途中で [_LINE_PAUSE_] クリック待ちに出来ます。[lp]
略称として lp も [lp] 使用できます。[ep]

（※５）４ページ目です。
@ _SEND_ :text0 do
@   _TEXT_ " 行頭に @ が書かれた行はコマンドブロックになります "
@ end
@   _LINE_PAUSE_ # コマンドブロック内でのコメントは "#" を使います
@ _SEND_ :text0 do
@   _LINE_FEED_
@ end
  _SEND_ :text0 do
    _TEXT_ " インデントされた行もコマンドブロックと見なされます "
  end
  _END_PAUSE_
  _SEND_ :text0 do
    _FLUSH_
  end
```

実行すると画面の下に文字列が表示されます。表示の途中でスペースキーを押すとテキストが全て表示されます。アイコンが出ている状態でスペースキーを押すと改ページが実行されます。

解説

（※1）1ページ目です。スペースキーを押すと
次のページに進みます。[ep]

（※2）2ページ目です。
改行が自動的に反映されます。[ep]

文字列は、標準テキストウィンドウに送信され、画面に表示されます。改行位置も反映されます。空行は改ページを表します。"[ep]" は後述するインラインコマンドで、ページ末の入力待ちアニメアイコンを表示したのち、ユーザーの入力があるまで待機します。

（※4）3ページ目です。[_LINE_PAUSE_]“_LINE_PAUSE_”コマンドを使うと、
行の途中で [_LINE_PAUSE_] クリック待ちに出来ます。[lp]
略称として lp も [lp] 使用できます。[ep]

"[" 〜 "]" で囲まれている箇所は「インラインコマンド」と言います。インラインコマンドではtsukasa 言語が用意しているコマンドを1個記述できます。コマンドを使うことで様々なインタラクションをゲームに組み込むことができます。

ここで使っているコマンドは "_LINE_PAUSE_" です。"_LINE_PAUSE_" は、行の途中でクリック待ちアイコンを表示し、ユーザーの入力があるまで待機します。これを使うと、文の途中でもユーザーの入力を待つことができます。"lp" は "_LINE_PAUSE_" の別名のコマンドで、まったく同じ機能になります。

"[" と "]" の間に改行を入れることはできないので注意してください。

（※5）4ページ目です。

```
@ _SEND_ :text0 do
@   _TEXT_ " 行頭に @ が書かれた行はコマンドブロックになります "
@ end
@   _LINE_PAUSE_ # コマンドブロック内でのコメントは "#" を使います
@ _SEND_ :text0 do
@   _LINE_FEED_
@ end
```

行頭に "@" が置かれた行は「コマンド行」と呼びます。コマンド行では tsukasa 言語を記述できます。このように、通常のテキスト出力の記述方法を、tks スクリプト内に混在させることもできます。

```
  _SEND_ :text0 do
    _TEXT_ " インデントされた行もコマンドブロックと見なされます "
  end
  _END_PAUSE_
  _SEND_ :text0 do
    _FLUSH_
  end
```

行頭がタブ文字、あるいは半角空白が2個以上置かれた行も、同じくコマンド行として解釈します。好きな記述方法を使ってください。

サンプル 17：【tks スクリプト 2】文章を装飾する

次は文章を装飾してみます。テキストウィンドウとして使用している TextPage コントロールは非常に多彩なカスタマイズ能力があり、製作するゲームの雰囲気にあわせたテキスト表現が可能です。

サンプルコード (sample_1_17.tks)

```
文字装飾デモです。行の途中で [_CHAR_SET_  color:[255,0,0]] フォントの色を変えます [ep]

@   _CHAR_SET_  font_name: "ＭＳ Ｐ明朝", color:[255,255,255]
フォントの種類を変えたり [_CHAR_SET_  line_height: 64, font_name: "ＭＳＰ ゴシック"]
文字サイズを [_CHAR_SET_ size:16] 変える事も [_CHAR_SET_  size:64] できます。[ep]
@   _CHAR_SET_ size:32, line_height: 32

他にも様々な設定が可能です。[lp]
@    _WAIT_FRAME_ 10
例えば文字の表示速度を変えたり、[lp]
@    _WAIT_FRAME_ 3
[_CHAR_RUBI_ "エクスカリバー"] ルビを振る事もできます。[ep]
```

文字の色が変わったり、文字サイズが変更されます。文字の表示速度も変化します。

解説

```
文字装飾デモです。行の途中で [_CHAR_SET_  color:[255,0,0]] フォントの色を変えます [ep]
```

インラインコマンドで _CHAR_SET_ コマンドを実行し、以降に表示される文字の色を [255,0,0] に変更します。_CHAR_SET_ コマンド自身はデフォルトで用意されているユーザー定義コマンドで、標準の TextPage コントロール（ここでは text0）に _SET_ コマンドを送信します。

```
@   _CHAR_SET_  font_name: "ＭＳ Ｐ明朝", color:[255,255,255]
```

次の行はコマンド行です。font_name でフォントをＭＳＰ明朝に、color で文字色を白に変更します。

```
フォントの種類を変えたり [_CHAR_SET_  line_height: 64, font_name: "ＭＳＰ ゴシック"]
文字サイズを [_CHAR_SET_ size:16] 変える事も [_CHAR_SET_  size:64] できます。[ep]
```

行の途中でフォントをＭＳＰゴシックに戻し、行の高さ（line_height）を変更します。次の行では文章の途中で size プロパティで文字のサイズを変えています。

行の高さを、フォントサイズを変えた行ではなく、その1つ前の行のインラインコマンドで書いている

のは、行の高さが直前の行の改行時に決定されるからです。tsukasa 言語ではこのようにコマンドの送信タイミングが挙動に強く影響する傾向があるので、意識しておくと開発が進みやすいかと思います。

```
@   _CHAR_SET_ size:32, line_height: 32

他にも様々な設定が可能です。[lp]
```

コマンド行で書式を戻します。先述したたように line_height は設定した次の行から反映されます。

```
@    _WAIT_FRAME_ 10
例えば文字の表示速度を変えたり、[lp]
```

_WAIT_FRAME_ は文字毎の表示間隔フレーム数を変更するユーザー定義コマンドです。これによって描画速度を制御します。

```
@    _WAIT_FRAME_ 3
[_CHAR_RUBI_ " エクスカリバー "] ルビを振る事もできます。[ep]
```

描画速度を元に戻した後、ルビを表示させます。ルビは本来文字が描画される座標を原点として、オフセット座標と文字幅間隔を指定して描画します。

まとめ

文字列の描画はあらゆるゲームでの必須要素であり、ゲーム全編を通してユーザーが注目し続けるＵＩでもあります。また、ゲームの世界観を表現する為に、描画方法を細かく調整する必要があります。tks スクリプトを使用すると、最低限の記述で大量のテキストを処理できるようになります。

TextPage コントロールは、柔軟な装飾能力を持ち、ユーザーの求めるテキスト表現が可能です。ここで紹介した以外にも様々なプロパティがあるので、リファレンスマニュアルを確認してください。

また、文字レンダラという機能を利用することで、１文字単位での描画をプログラミングすることもできます。標準では「２フレーム単位で文字をフェードインさせながら表示させ、ユーザーが入力するとハーフフェードアウトする」という文字レンダラを使用しています。この文字レンダラは script/sample/Utility_script.rb に収録されており、また第３部でも解説もしていますので、それらを参考に新しい文字表示表現を考えてみてください。

【コラム】日本のゲームプログラミング＆ゲームプログラマーの特殊性

ゲームプログラミングは、ビジネスソフトのプログラミングと根本的に異なっている箇所が多く、ゲームソフトのプログラマーにはゲーム開発に特化した能力が求められます。とはいっても、これはビジネスソフトのプログラマーよりゲームソフトのプログラマーの方が高度な知識・技術を必要とするという意味ではありません。そうではなく、両者の成り立ちの違いによる物です。

国内の商業ゲームプログラミングは、そもそもがハッカー文化の上に成り立ってきました。ゲームメーカーはゲームを作るために、ゲームが大好きで、かつ、自作ゲームの為に独学でプログラミングを学んできた筋金入りのハッカーを何人も（独自言語のコンパイラを自作できるような、ビジネスアプリの現場でも早々いない人達を何人もです）スカウトします。そして彼らに最高の環境と報酬を与え、あとは彼らがコードを組み上げるのをひたすら待つというスタイルを繰り返しました。その結果、これまで日本発の独創的なゲームが多数製作されてきましたが、逆に、体系的なプログラマ教育手法を確立するような流れが作られないまま今日に至ったのです。

この職人主義的なスタイルは、比較的最近までは成立していました。しかし、ゲームメーカー内の開発ラインが増え、またゲームメーカー自体の数が増えるうちに、ゲーム開発に特化したプログラマーを必要な人数揃える事が、どのメーカーでも徐々に困難になってきました。また、ゲームハードが進化するごとに、プログラマーに必要とされる知識が増加、高度化し、その条件を満たすプログラマーの確保はさらに難しくなりました。

国内大手メーカーの多くは、開発の高コスト化に対処する為に、再利用可能なゲーム開発フレームワークの設計に取り組みました。しかし、ゲームメーカーに所属しているプログラマーは、上記のようにゲーム開発に特化した才能の持ち主であり、汎用的なフレームワークを作り上げるスキルとモチベーションを持ち合わせていなかった為、いずれの例でも、大きな成功をおさめることはできませんでした。

現在、商業ゲームフレームワークの市場は Unity と UnrealEngine という海外製品の２強状態になっています。国内でも多くのメーカーが自社でのエンジン開発を諦め、いずれかの製品を使用しています。

しかし、これらのツールを採用しても、最終的にプログラマーが作り込むことになるコード自体は従来と変わっておらず、プロジェクトマネジメントのコストは下げられても、ゲーム開発に特化したプログラマーが足りない事によって発生するボトルネックを解消することはできません。

このような問題を解決するために、司エンジンが開発されたのです。

第3部
より高度なプログラミング

第３部では、複数のコントロールやコマンドを組み合わせることで、より複雑な処理を実現する方法を解説します。

司エンジンスターターキットには、ここに挙げた物以外にも、多数のサンプルコードが収録されているので、実際に動作を確認し、自作ゲームへの応用方法を考えてみて下さい。

1：様々なボタンの作成

ゲームを構成する重要な要素の一つに「ボタン」があります。ボタンはユーザーのインタラクションをシステムが受け取るために使用され、ゲーム中で様々な形状、演出を伴って登場します。

ボタンのバリエーションはきわめてに多岐にわたります。例えばゲーム画面の任意の場所で右クリックすることでメニューを表示させる場合も、広義では「透明な全画面サイズのボタン」として実装できます。

tsukasa言語ではボタンに相当する組み込みコントロールは用意されていません。代わりにClickableLayoutコントロールを使ってボタンの挙動をプログラミングできます。その為、手軽にボタンを表示できるわけではありませんが、柔軟なカスタマイズ能力を提供しています。

この項では「基本的なボタン」「コリジョンを用いたボタン」「カラーキー画像を用いたボタン」の3つのサンプルを紹介します。

サンプル1：基本的なボタン (sample_2_1_1.rb)

```
_CREATE_ :ClickableLayout, x: 100, y: 100, shape:[0,0,256,256], width: 256, height:
256, id: :test01 do
  _CREATE_ :TileMap,
    width: 256, height: 256 do
    _SET_ map_array: [[0]]
    _SET_TILE_ 0, path: "./resource/button_normal.png"
    _SET_TILE_ 1, path: "./resource/button_over.png"
    _SET_TILE_ 2, path: "./resource/button_key_down.png"
  end

  _DEFINE_ :inner_loop do
    _CHECK_MOUSE_ :cursor_over do
      _SEND_(0){ _MAP_STATUS_ 1}
    end
    _CHECK_MOUSE_:cursor_out do
      _SEND_(0){ _MAP_STATUS_ 0}
    end
    _CHECK_MOUSE_:key_down do
      _SEND_(0){ _MAP_STATUS_ 2}
    end
    _CHECK_MOUSE_:key_up do
      _SEND_(0){ _MAP_STATUS_ 1}
    end
    _HALT_
    _RETURN_ do
      inner_loop
    end
  end
  inner_loop
end
```

実行すると四角いボタンが表示されます。マウスカーソルを画像の上に載せると色が変わり、クリックすると更に変わります。クリックしたままカーソルを画像の外に移動させるなどして、ボタンの挙動を確認してください。

サンプル１：解説

ユーザー定義コマンド button を定義したのち、そのコマンドを実行します。button に設定した id/x/y のオプションは、button コマンドが呼びだされた時に options ハッシュの中に格納され、内部で見ることができます。

ではユーザー定義コマンド button の中身を見ていきましょう。

```
_CREATE_ :ClickableLayout, x: 100, y: 100, shape:[0,0,256,256], width: 256, height:
256, id: :test01 do
```

最初にベースとなる ClickableLayout コントロールを生成します。このコントロールは Layout コントロールにマウスカーソルとの衝突判定を行う機能が付加された物です。Layout コントロールは、自身は描画要素を持たず、子コントロールをグルーピングする為に用意されているコントロールです。

```
  _CREATE_ :TileMap,
    width: 256, height: 256 do
    _SET_ map_array: [[0]]
    _SET_TILE_ 0, path: "./resource/button_normal.png"
    _SET_TILE_ 1, path: "./resource/button_over.png"
    _SET_TILE_ 2, path: "./resource/button_key_down.png"
  end
```

TileMap コントロールを生成します。このコントロールは、マップタイルの画像からなる配列と、そのマップタイルの配置位置を示す二次元配列から、平面マップを表示させるコントロールです。ここでは、縦１マス×横１マスの変則的なマップを表示し、ボタンの画像としています。_SET_TILE_ で読み込んだ画像をタイルとして登録します。

```
  _DEFINE_ :inner_loop do
    _CHECK_MOUSE_ :cursor_over do
      _SEND_(0){ _MAP_STATUS_ 1}
    end
    _CHECK_MOUSE_:cursor_out do
      _SEND_(0){ _MAP_STATUS_ 0}
    end
    _CHECK_MOUSE_:key_down do
      _SEND_(0){ _MAP_STATUS_ 2}
    end
    _CHECK_MOUSE_:key_up do
      _SEND_(0){ _MAP_STATUS_ 1}
    end
    _HALT_
```

```
    _RETURN_ do
      inner_loop
    end
  end
```

ボタンの挙動を管理するユーザー定義コマンド inner_flame を定義します。_CHECK_MOUSE_ コマンドでマウスカーソルとの衝突状態を判定し、それに応じて MapTile が表示する画像を切り替えます。

判定の後は _HALT_ で1フレーム送ったのち、自分自身を再実行対象に指定して終了します。このコマンドは1度実行されると自分自身を永久に実行します。ボタンに限らず、ループ処理が必要な場合は、このテクニックが利用できます。

```
  inner_loop
end
```

定義したコマンドを実行します。

サンプル2：コリジョンを使ったボタン (sample_2_1_2.rb)

二つ目はコリジョンの形状を数値で指定するボタンです。

```
# ボタンコントロール
_CREATE_ :ClickableLayout, width: 256, height: 256, id: :button1,
      shape: [128,128,128] do
  _CREATE_ :Image, id: :normal, width:256, height:256 do
    _CIRCLE_ x: 128,  y: 128, r: 128, color: C_BLUE, fill: true
    _TEXT_ x:80, y:120, text: "NORMAL", color: [0,255,0]
  end
  _CREATE_ :Image, id: :over, visible: false, width:256, height:256 do
    _CIRCLE_ x: 128,  y: 128, r: 128, color: C_YELLOW, fill: true
    _TEXT_ x:80, y:120, text: "OVER", option: {color: [0,0,0]}
  end
  _CREATE_ :Image, id: :key_down, visible: false, width:256, height:256 do
    _CIRCLE_ x: 128,  y: 128, r: 128, color: C_GREEN, fill: true
    _TEXT_ x:80, y:120, text: "DOWN", option: {color: [0,0,0]}
  end
  _DEFINE_ :inner_loop do
    _CHECK_MOUSE_:cursor_over do
      _SEND_(:normal)  {_SET_ visible: false}
      _SEND_(:over)    {_SET_ visible: true}
      _SEND_(:key_down){_SET_ visible: false}
    end
    （中略）
    _HALT_
    _RETURN_ do
      inner_loop
    end
  end
```

```
  inner_loop
end
```

実行すると青色の丸いボタンが表示されます。このボタンの当たり判定は見た目と同じになっています。丸の縁あたりでカーソルを移動させると、縁に沿った当たり判定があるのがわかります。

サンプル２：解説

```
_CREATE_ :Layout, width: 256, height: 256, id: :button1,
      shape: [128,128,128] do
```

shape プロパティに設定した値に応じた形状のコリジョンがコントロールに設定され、その形状に沿って衝突判定が検出できるようになります。ここでは [128,128] を中心点とした、半径 128 ドットの円形のコリジョンを設定しています。

collision_shape には円の他に点、三角形、四角形を指定できます。詳しくはリファレンスマニュアルを参照してください。

```
  _CREATE_ :Image, id: :normal, width:256, height:256 do
    _CIRCLE_ x: 128,  y: 128, r: 128, color: C_BLUE, fill: true
    _TEXT_ x:80, y:120, text: "NORMAL", color: [0,255,0]
  end
```

Image コントロールの _CIRCLE_/_TEXT_ コマンドを使って、ボタンの画像を直接生成しています。Image コントロールには他にも図形描画をサポートするコマンドが多数用意されています。デモの作成などで一時的に図形が必要な場合などに役立ちます。

注意しておきたいのは、衝突判定に使用されているのはあくまで collision_shape で設定した円であって、画面に表示されている円は関係ないという点です。

```
  _DEFINE_ :inner_loop do
    _CHECK_MOUSE_:cursor_over do
      _SEND_(:normal)  {_SET_ visible: false}
      _SEND_(:over)    {_SET_ visible: true}
      _SEND_(:key_down){_SET_ visible: false}
    end
```

一つ目のサンプルと同じ様に inner_loop コマンドを定義しています。_CHECK_MOUSE_ コマンドでカーソル状態を確認し、状況に応じて３つの子要素に対して表示／非表示を設定することで画像を切り替えています。表示する画像に対してより細かい演出をつけたい場合は、このやり方がオススメです。

サンプル３：カラーキーを使ったボタン (sample_2_1_3.rb)

最後はカラーキー画像で当たり判定箇所を指定するボタンです。

```
_CREATE_ :ClickableLayout,
        width: 256,
        height: 256,
        id: :test01,
        shape: [0,0,256,256],
        colorkey_id: :normal,
        colorkey_border:200 do
  _CREATE_ :Image, path: "./resource/star_button.png",
    id: :normal
 （中略：画像の生成、イベント記述部はサンプル２と同じ）
end
```

実行すると星形の画像が表示され、画像の縁に沿って当たり判定が設定されているのが確認できます。colorkey_border の値を変えると、星形の縁のどこで判定が発生するかの境界が変化します。

サンプル３：解説

```
        colorkey_id: :normal,
        colorkey_border:200 do
```

colorkey_id プロパティを介して、カラーキーとして使用する画像（normal）を ClickableLayout コントロールに登録します。マウスの衝突判定が成立した場合、さらに normal を確認し、カーソル座標にあるピクセルの透明度が colorkey_border よりも大きい場合、衝突したとみなします。

まとめ

主なボタンの記述方法について説明しました。

状態に応じた画像をいちいち宣言したり、毎回画像の可視不可視を個別に設定しているのが煩雑な記述に見えるかもしれませんが、これは（Image コントロールに限定せず）任意のコントロールをボタンの描画要素として扱えるように ClickableLayout コントロールが設計されているためです。例えば、マウスが上に載るとくるくる回転したり、ぼんやりと光に包まれるボタンが作れるでしょう。様々な活用法を考えてみてください。

司エンジンでは、一般的な挙動をとるボタンコントロールを簡単に生成するためのヘルパーコマンドが用意されています。詳しくはリファレンスマニュアルを参照してください。

2：コントロールのアニメーション

tsukasa 言語では画像を切り替えるタイミングをスクリプトで指定することでアニメーションを実現します。TileMap コントロールを使うと、簡単にアニメーションを実現できます。

サンプルコード (sample_2_2.rb)

実行すると画面の左上端に本がめくれるアニメーションが表示されます。以下要点のみ説明します。

解説

```
_CREATE_ :TileMap,
  map_array: [[0]], size_x: 1, size_y: 1, width:32, height:32, id: :test01 do
  _SET_TILE_GROUP_ path: "./resource/icon/icon_4_a.png",
    x_count: 4, y_count: 1
```

_SET_TILE_GROUP_ コマンドでアニメーションさせる画像を読み込みます。x_count/y_count は画像の分割数です。path で指定した画像をこの数に分割して画像配列に登録します。ここでは、icon_4_a.png をX方向に4分割して、マップタイルとして格納しました。

```
  _DEFINE_ :inner_loop do
    _MAP_STATUS_ 0
    _WAIT_ count: 5
    _MAP_STATUS_ 1
    _WAIT_ count: 5
    _MAP_STATUS_ 2
    _WAIT_ count: 5
    _MAP_STATUS_ 3
    _WAIT_ count: 5
    _RETURN_ do
      inner_loop
    end
  end
```

ボタンと同じ様に、毎フレーム呼び出す inner_loop コマンドを定義します。inner_loop コマンドは、表示する画像の更新と、その後の5フレーム待機とを繰り返し、画像が一周したら自分自身を再度実行対象に指定して終了します。

```
  inner_loop
end
```

定義した inner_loop コマンドを実行します。以後コントロールは inner_loop を実行し続けます。

3：文字列の表示とポーズ処理

TextPage コントロールが表示する個々の文字を Char コントロールとして管理することで、文字単位での細かい制御が可能になっています。これは、逆に言えば、文字単位で細かい制御をするためにはスクリプトを作り込む必要があるということです。

ここでは、text_layer_script.rb 内に標準で用意されている _TEXT_WINDOW_ コマンドの処理を例にテキストウィンドウの制御を解説します。

なお、説明のためにスクリプトを簡略化しているので、元のコードも参照してください。

文字レンダラによる文字描画

_TEXT_WINDOW_ コマンドでは、TextPage コントロールの生成時に _CHAR_RENDERER_ というユーザー定義コマンドを定義しています。このコマンドは文字レンダラとして扱われ、TextPage コントロールが内部で文字を生成した際、このコマンドが渡されます。

以下、文字レンダラの実装を解説します。

```
# 文字レンダラ
_DEFINE_ :_CHAR_RENDERER_ do
  # フェードイン
  _MOVE_ [20, :in_quint], alpha: [0,255] do
    # キー入力判定
    _CHECK_INPUT_  key_down: K_RCONTROL,
                   key_push: K_SPACE,
                   mouse: :push do
      # α値を初期化
      _SET_ alpha: 255
      _BREAK_
    end
  end
```

２０フレームかけて文字の透明度を０から２５５へフェードインさせます。フェードイン中にスペースキーかマウスボタンが押下された際は、透明度を２５５に直接設定してフェードを終了します。

```
# 待機フラグが下がるまで待機
_WAIT_ [:_ROOT_, :_TEXT_WINDOW_TEMP_], equal: {_SLEEP_: false}
# キー入力伝搬を防ぐ為に１フレーム送る
_HALT_
```

フェードイン終了後は、_TEXT_WINDOW_TEMP_ データストアの _SLEEP_ プロパティが false になるまで待機します。_SLEEP_ プロパティは文字が表示される前に true に設定されていて、ユーザーがクリック待ち時にキーを押下すると false に更新されます。

```
    # ハーフフェードアウト
```

```
      _MOVE_ 60, alpha: [255, 128] do
        # キー入力判定
        _CHECK_INPUT_ key_down: K_RCONTROL,
                      key_push: K_SPACE,
                      mouse: :push do
          # α値を初期化
          _SET_ alpha: 128
          _BREAK_
        end
      end
    end
```

ここから後処理になります。６０フレームかけて透明度を１２８まで下げます。フェード中にキーが押下された場合、α値を 128 に強制的に変更し、フェードを終了します。

まとめ

_CHAR_RENDERER_を用いることで、文字の描画を生成から削除まで完全にプログラミングできます。ゲームのフレーバーに合わせて様々な表現方法を考えてみてください。

ポーズ処理

ノベルゲームではテキストを表示している最中もキー入力を監視し、状況に応じて処理する必要があります。_TEXT_WINDOW_ではそれらの処理を、先に説明した文字レンダラと、ユーザーの入力を待つポーズ処理に集約させています。以下、ユーザー定義コマンドである _PAUSE_ を解説します。

```
_DEFINE_ :_PAUSE_  do
  # テキストレイヤのクリック待ち
  _GET_ :_DEFAULT_TEXT_PAGE_, datastore: :_TEMP_ do |_DEFAULT_TEXT_PAGE_:|
    _SEND_ _DEFAULT_TEXT_PAGE_ do

      （※ TextPage コントロールに送信されるコマンドブロック）

    end
  end
```

_PAUSE_は大きく二つの処理から構成されています。まず_GET_で_DEFAULT_TEXT_PAGE_プロパティに格納されているデフォルトのTextPageコントロールのidを取得し、そのコントロールにコマンドブロックを送信します。まず、コマンドブロックを送信した後の処理を見てみましょう。

```
  # スリープフラグを立てる
  _SET_ [:_ROOT_, :_TEXT_WINDOW_TEMP_], _SLEEP_: true
  # スリープフラグが下りるまで待機
  _WAIT_ [:_ROOT_, :_TEXT_WINDOW_TEMP_], equal: {_SLEEP_: false} do
    _CHECK_INPUT_ key_down: K_RCONTROL do
      _BREAK_
    end
```

```
  end
end
```

_TEXT_WINDOW_TEMP_ データストアの _SLEEP_ プロパティに true を設定し、それが false になるまで待機します。これによって、スクリプトの読み込みはユーザーの入力があるまで待機状態になります。

では、テキストウィンドウに送信されたコマンドブロックを見ていきましょう。

```
    # 最後の文字のフェードイン待ち
    _WAIT_ count: 28 do
      # キーの押下を判定
      _CHECK_INPUT_ key_down: K_RCONTROL,
                    key_push: K_SPACE,
                    mouse: :push do
        # キー押下のクリアを待機
        _WAIT_  do
          _CHECK_INPUT_ key_down: K_RCONTROL,
                        key_up: K_SPACE,
                        mouse: :up do
            _BREAK_
          end
        end
        _BREAK_
      end
    end
```

全ての文字のフェードインが終わるのを待つために２８フレーム待機します。

まだフェードインが終わっていない間にキーの押下があった場合、_WAIT_ を強制的に終了させるのですが、正確には押下されたキーが解放（指が離れた）タイミングにしたいので、_WAIT_ を入れ子にしています。

```
     # キー押下待機
    _WAIT_ do
      _CHECK_INPUT_ key_down: K_RCONTROL,
                    key_push: K_SPACE,
                    mouse: :push do
        _BREAK_
      end
    end
    # ウェイクに移行
    _SET_ [:_ROOT_, :_TEXT_WINDOW_TEMP_], _SLEEP_: false
  end
end
```

キー入力があるまで待機します。キー入力があれば _TEXT_WINDOW_TEMP_ データストアの _SLEEP_ プロパティに false を設定します。これによって、それぞれの文字はハーフフェードアウトを開始し、ルートの待機は解除されてスクリプトの処理が再開されます。

第4部
tsukasa 言語リファレンスマニュアル
(v2.2 対応版)

- 司エンジン導入手順
- tsukasa 言語記法解説
- tks スクリプト記法解説
- 標準コントロール
  - Control
  - Layoutable
  - Layout
  - Window
  - Drawable
  - Image
  - Input
  - Data
  - DrawableLayout
  - ClickableLayout
  - Sound
  - Char
  - TextPage
  - TileMap
  - Shader
  - RuleTransition
- 標準ユーザー定義コマンド (default_script.rb)
- ヘルパーコマンド (helper_script.rb)
- テキストウィンドウ (text_layer_script.rb)
- 起動時に自動的に生成されるコントロール群
- 司エンジンのフォルダ構成
- 組み込み定数一覧

司エンジン導入手順

司エンジンを導入するには、司エンジンスターターキットを使う方法と、開発環境をインストールする方法があります。

・導入方法A：司エンジンスターターキットを使う
すぐに動作検証をする為のオールインワンパックです。環境が全てパッキングされているので、手軽に動作を確認できます。

・導入方法B：開発環境をインストールする
必要な各種のファイルをインストールして完全な開発環境を構築します。本格的な開発を行う場合はこちらを推奨します。

以下、それぞれの導入方法について説明します。

導入方法A：司エンジンスターターキットを使う

・以下のファイルをダウンロードして展開してください。
http://someiyoshino.main.jp/file/tsukasa/tsukasa_starter_kit_v2_2.zip
・ルートフォルダにある main.exe を実行するとサンプルコードが起動します。main_dev.exe をプロンプトから起動すると、標準出力が有効になります（デバッグ用）。
・ルートフォルダにある first.rb ファイルを修正することで、自作のスクリプトを動かすことができます。

導入方法B：開発環境をインストールする

開発環境を導入するには以下の5つの手順を実行します。あらかじめ DirectX をインストールしておいてください（司エンジンは Windows 専用です）

1・ruby のインストール

Ruby をインストールします。司エンジンは Ruby 2.2 以降で動作します。下記のサイトで「Ruby 2.2.4」をクリックし、ダウンロードしたファイルを実行するとインストールできます。
rubyinstaller for windows（http://rubyinstaller.org/downloads/）
現在動作を確認している Ruby のバージョンは下記になります。

```
ruby 2.2.4p230 (2015-12-16 revision 53155) [i386-mingw32]
```

【注意】司エンジンが使用するライブラリのバージョン依存のため、ver2.1 以前の Ruby では正常に動作しません（ver2.3 以降を使う際には、Ayame のアップデートを確認してください）。

2・DXRuby のインストール

mirichi 氏が製作した ruby 用 DirectX ラッパーフレームワーク「DXRuby」をインストールします。下記のサイトから最新版をダウンロードし、install.rb をダブルクリックするとインストールできます。

Project DXRuby（http://dxruby.osdn.jp/）

現在動作を確認している DXRuby のバージョンは DXRuby 1.4.4 になります。

3・司エンジンの配置

司エンジンのコードを配置します。

下記のファイルをダウンロードして展開し任意の場所に配置します（現状の最新版です）。

http://someiyoshino.main.jp/file/tsukasa/tsukasa_v2_2.zip

4・parslet のインストール

Ruby のパーサーライブラリである parslet をインストールします。司エンジンでは tks スクリプトをパースするのに使用しています。

parslet は RubyGem に登録されているので、Ruby がインストールされた状態で、コマンドプロンプトから以下のように入力すれば、自動的にインストールされます。

```
gem install parslet
```

5・Ayame/Ruby のインストール

DirectMusic ライブラリ「ayame」の Ruby ラッパーである「ayame/ruby」をインストールします。

以下のＵＲＬからダウンロードします。バージョンは Ayame/Ruby0.0.3 ruby2.2.x 用になります。

http://dxruby.osdn.jp/files/ayameruby003-mswin32-ruby22.zip

ダウンロードした ayameruby003-mswin32-ruby22.zip を解凍し、Ayame.dll と ayame.so の二つのファイルを以下の場所に配置します。

ファイル	配置ディレクトリ
Ayame.dll	main.rb と同じディレクトリ
ayame.so	Ruby がインストールされたディレクトリ（デフォルトでは C:¥Ruby22）配下の ¥lib¥ruby¥site_ruby¥2.1.0¥i386-msvcrt

6・動作確認

全てのインストールが正常に行われていれば、コマンドプロンプトでカレントドライブを司エンジンの開発フォルダに移動し、以下のように入力するとサンプルが起動します。script フォルダ内にある first.rb にスクリプトを記述すれば反映されます。

```
ruby main.rb
```

インストールの説明は以上になります。

tsukasa 言語記法解説

ここでは、tsukasa 言語の記法について解説します。tsukasa 言語は Ruby の内部 DSL として実装されていて、内部的には Ruby 式として評価されます。ここに書かれていない仕様については、ruby 言語に準じます。

ファイル名

tsukasa 言語のファイルは拡張子を ".rb" にして下さい。テキスト形式は UTF-8 のみ受け付けます。

予約語

1文字目が "_" で始まる識別子は全て予約語になります。また、ruby の予約語も使用できません。

コマンド

tsukasa 言語では全ての処理をコマンドで記述します。コマンドの書式は以下になります。

```
[ コマンド名 ] [ 第一引数 ],[ ハッシュ ] do |[ ブロック引数 ]|
  ブロック
end
```

・コマンドは、そのコマンドがコマンドリストに格納されていたコントロールに対して実行されます。トップレベルで実行されたコマンドは Window が対象になります。
・第1引数、ハッシュ、ブロックはそれぞれコマンドごとに省略できるもの、出来ないものがあります。詳細はそれぞれのコマンドを参照してください。
・ブロック引数にはハッシュが設定されます。コマンドによって生成される値の他、ユーザー定義コマンドの呼び出し時に設定された値が渡される場合があります。詳細は各コマンドを確認して下さい。

[プログラマ向け解説]

tsukasa 言語ではすべての文をコマンドの実行として記述します。この「すべての文」というのは比喩ではなく、tsukasa 言語では条件分岐や繰り返し構文に相当する文もコマンド扱いになります。
また、すべてのコマンドは上記のインターフェイスを踏襲していて、例外はありません。

コメント

"#" 以降はコメントになります（Ruby に準じます）。

色配列と座標

幾つかのコマンドでは引数に色配列を要求する物があります。色配列は [r,g,b] の３要素、あるいは [a,r,g,b] の４要素からなる配列です。「a= 透明度（α値）,r= 赤 ,g= 緑 ,b= 青」に相当します。

tsukasa 言語内で用いる座標系は、常に左上を原点とします。

コントロールへの相対パス

・コマンドによっては、操作の対象とするコントロールへの相対パスを指定できる物があります。コントロールへの相対パスは配列として表現します。配列には以下の物が使用出来ます。

要素	内容
[シンボル]	コントロールの id
[整数]	子コントロールのインデックス
:_ROOT_	ルートコントロール（Window）
:_PARENT_	親コントロール

・コントロールへの相対パスが指定されていたら、司エンジンはパスに沿ってコントロールを検索します。
・シンボルが指定された場合、そのシンボルを検索し、最初に見つかった物が返ります。
・整数が指定された場合、その数字をインデックスとみなし、直下の子コントロールに添え字でアクセスします。マイナスの整数を指定した場合、コントロールリストの末端からアクセスします。
・:_ROOT_ が指定された場合、コントロールツリー最上位のコントロールが返ります。
・:_PARENT_ が指定された場合、現在のコントロールの親にあたるコントロールが返ります。
・パスが省略された場合、あるいは nil が設定された場合、自コントロールが返ります。
・直下の子コントロールを指定する場合、配列ではなく直接記述できます。

ruby コードの使用

司エンジン上で処理される tsukasa 言語は Ruby の内部ＤＳＬであり、実際には ruby コードです。その為、スクリプト中に ruby のコードを記述することができます。これにより複雑な記述が可能になりますが、利用の際には以下の点に注意してください。

tsukasa 言語が実行タイミングで処理されるのに対し、ruby コードはそのコードが含まれるブロックが評価されるタイミングで実行されます。Ｃ言語で言う所のプリプロセス処理の扱いになります。

ローカル変数／インスタンス変数などは保存されません。特に、ループやウェイト処理内で ruby コードを記述すると、直感と異なる挙動になりやすいので注意してください。

ruby コードが実行されるドメインはスクリプトパーサークラス内であり、コントロールクラス内ではありません。実行しているコントロールのプロパティにアクセスしたい場合は _GET_ コマンドを使用します。

tks スクリプト記法解説

tks スクリプトは、司エンジンでノベルゲームの開発を効率化するために用意されている tsuaksa 言語のラッパー言語です。ファイルは読み込まれた時点で自動的に tsukasa 言語に変換され、内部では透過的に扱われます。tks という名称に由来はありません。"TsuKaSa" と覚えてください。

ファイル名

tks スクリプトのファイルは拡張子を ".tks" にして下さい。スクリプトファイルが _INCLUDE_ で読み込まれる際、拡張子が ".tks" のファイルは自動的に tsukasa 言語に変換されます。

tks スクリプトの構成要素

tks スクリプトの全ての行は、行頭の識別子によって以下のどれかに分類されます。

種類	説明
コメント行	評価されず無視される。
コマンド行	tsukasa 言語をそのまま記述できる
テキスト行	ゲーム画面に出力する文字列をフラットに記述できる
空行	改ページ指定
ラベル行	_LABEL_ コマンドの短縮記法（「ユーザー定義コマンド」の項で解説）

コメント行

行の先頭に "//" が置かれた行はコメントとして無視されます。

コマンド行

行の先頭が "@"、あるいはインデント（空白２文字かタブ文字）で始まる行はコマンド行になります。"@" もしくはインデントを除去した残りの文字列を、tsukasa 言語とみなして評価します。

テキスト行

他の行に当てはまらない行は全てテキスト行と見なされます。テキスト行に記述された文字列は _TEXT_ コマンドに変換され、デフォルトのテキストウィンドウに送信されます。行の終わりでは改行コード（_LINE_FEED_）が自動的に挿入されるので、テキストウィンドウに出力する文字列をフラットに記述できます。

空行

空行は改ページ指示とみなし、デフォルトのテキストレイヤに _FLUSH_ コマンドを送信します。

改ページ前、あるいは行の途中でユーザーの入力を待ちたい場合は後述するインラインコマンドで "[ep]"、"[lp]" を使用してください。

コマンド行とテキスト行の連携について

テキスト行に記述された文字列は、内部で _TEXT_ コマンドに変換されます。

コマンド行とテキスト行は内部では透過的に扱われるため、例えば以下のようにコードを混在させることができます。

```
@   _CHECK_ :_SYSTEM_, equal: {data0: K_X} do
Xキーが押されました
@   end
```

上記の tks スクリプトは内部的には以下のような tsukasa 言語に変換されます。

```
_CHECK_ :_SYSTEM_, equal: {data0: K_X} do
  _GET_ :_DEFAULT_TEXT_PAGE_, datastore: :_TEMP_ do |_DEFAULT_TEXT_PAGE_:|
    _SEND_ _DEFAULT_TEXT_PAGE_ do
      _TEXT_ " Xキーが押されました "
    end
  end
end
```

インラインコマンド

テキスト行中に "[" 〜 "]" で囲んだ箇所を「インラインコマンド」と呼びます。インラインコマンドの中には tsukasa 言語を記述できます。

ただし、一つのインラインコマンド内に記述出来るのは1コマンドのみになります。また、インラインコマンド中に改行を挟むことはできないので注意してください。

Control

コントロールツリーに登録される全てのコントロールのベースになります（基本的に、ユーザーが直接Controlを使用することはありません）。全てのコントロールはControlの持つプロパティ / コマンドを使用できます。

プロパティ

プロパティ	説明
id	[シンボル] コントロールの識別名
child_update	[真偽値] 子コントロールの更新処理を行うかどうか（初期値：true）

・idを省略した場合はコントロール名が設定されます。

・chiled_updateにfalseが設定された場合、子コントロールがコマンドリストの処理を行わなくなります（描画は行われます）。シーン管理で子コントロールの処理を一時的に停止する場合に使います。

コマンド：コントロールの生成と削除

CREATE

オプション	説明
第1引数	[シンボル] 生成するコントロールのクラス名
ハッシュ	コントロールに渡すオプション群
ブロック	あり（ブロック引数無し）

・指定したコントロールを生成します。

・第1引数で指定したコントロールを生成してハッシュの値で初期化し、現在のコントロールの子コントロールとしてツリーに追加します。

・ブロックに記述したコマンド群は、コントロールの生成後に送信されます。送信されたコマンド群は、コントロールがツリーへ追加された直後に（つまり、フレームをまたぐことなく）実行されます。

DELETE

オプション	無し
第1引数	（任意）[配列][シンボル] 送信先のコントロールパス

・現在のコントロールを削除します。

・第1引数を設定している場合、パスで指定されたコントロールを削除します。パスの指定方法については_SEND_コマンドを参照してください。

・DXRubyオブジェクトを保持している場合、それらにdisposeを発行します。

_DEFINE_PROPERTY_

オプション	無し

ハッシュ	キー：プロパティ名　値：初期値

・コントロールに動的にプロパティを追加します。複数のプロパティを同時に追加できます。
・追加したプロパティは _SET_/_GET_/_MOVE_ などのコマンドで操作できます。ただし、動的に追加したプロパティは _QUICK_SAVE_/_QUICK_LOAD_ の保存対象にならないので注意してください。

コマンド：プロパティの設定と取得

SET

オプション	説明
第1引数	（任意）[シンボル配列] コントロールへの相対パス
ハッシュ	キー：[シンボル] プロパティ名　値：設定する値

・コントロールの各プロパティに値を設定します。複数のプロパティに値を同時に設定可能です。
・第1引数にコントローラへの相対パスが設定されている場合は指定されたコントローラを対象にします。

```
# コントロールのX座標、Y座標を設定する
_SET_ x: 100, y: 100
#:_TEMP_ コントロールの値を更新する
_SET_ [:_ROOT_, :_TEMP_], flag_a: true
```

GET

オプション	説明
第1引数	[配列][シンボル] プロパティ名
ハッシュ	
control	（任意）[シンボル配列] コントロールへの相対パス
ブロック	あり（ブロック引数：第1引数で指定されたプロパティ名からなるハッシュ）

・コントロールのプロパティの値を取得します。複数のプロパティの値を同時に取得できます。
・第1引数で指定したプロパティの値が、ブロック引数のハッシュに同名のキーに値として格納されます。
・control オプションにコントローラへの相対パスが設定されている場合は、パスで指定されたコントローラを対象にします。

```
# サンプルコード
  _GET_ :id do |options|
    _PUTS_ options[:id]
  end
  # このように記述することもできます。
  _GET_ :id do |id:|
    _PUTS_ id
  end
```

・また、第1引数のプロパティ名を以下のように配列で追加のオプションを指定できます。コントロール

への相対パスを指定するとそのコントロールのプロパティを取得します。取得名をシンボルで指定すると、ブロック引数にその値で取得できるようになります。複数のコントロールから値を取得する際、名前が被る際に使用します。

```
_GET_ [[ プロパティ名 , コントロールへの相対パス , 取得名 ], [……], [……]] do 〜 end
```

コマンド：条件判定

CHECK

オプション	説明
第1引数	（任意）[シンボル配列] コントロールへの相対パス
ハッシュ	（任意）キー：[シンボル] 判定条件　値：[ハッシュ] 比較式
ブロック	あり（ブロック引数無し）

・コントロールのプロパティの値を判定します。判定条件で指定した条件で、比較で指定したコントロールのプロパティと任意の値とのペアを比較し、一つでも条件が成立していればブロックを実行します。
・第1引数にコントロールへの相対パスが設定されている場合は、パスで指定されたコントロールを判定の対象にします。

判定条件と比較式

・条件判定は、ハッシュのキーで指定した判定条件と、その値として指定する比較式の組で記述します。
・比較式はハッシュで、キーにプロパティ名、値に比較対象の値を設定します。
・用意されている判定種類は以下になります。

判定条件	処理
:equal	指定したプロパティと、値が同値であれば成立。
:not_equal	指定したプロパティと、値が同値でなければ成立。
:under	指定したプロパティが、値より小さければ成立。
:over	指定したプロパティが、値より大きければ成立。

_CHECK_BLOCK_

オプション	説明
ブロック	あり（ブロック引数無し）

・_CHECK_ のバリエーションです。ブロックの有無判定に使用します。
・実行中のコマンドがブロックを付与して呼び出されている場合は、付与ブロックを実行します。

コマンド：ループ処理

LOOP

オプション	説明
第1引数	（任意）[整数] ループ回数
ブロック	あり (ブロック引数 [end: 指定したループ回数　now: 現在のループ回数])

・コマンドブロックを繰り返し実行します。ループ回数を指定した場合はその回数繰り返したのちループを終了します。設定されていない場合、_BREAK_ 等で抜けるまで無限に繰り返します。
・再度の実行はフレームをまたがずに実行されるので、適切に _HALT_ を実行しないと場合によっては無限ループになるので注意してください。
・【プログラマ向け】ruby のループと異なり、ローカルの値は保存されないので注意してください。

MOVE

オプション	説明
第1引数	[配列][フレーム数 , イージングの種類（省略時 :liner）]
ハッシュ	キー：遷移させたいプロパティ　値：[配列][遷移開始時 , 終了時の値]
ブロック	あり（ブロック引数無し）

・_LOOP_ のバリエーションです。コントロールの任意のプロパティを第一引数で指定したフレーム数かけて遷移させます。
・プロパティは、コントロールが持つ全てのプロパティが対象となります。ただし、整数が設定できないプロパティを対象とした場合、動作は未定義になります。
・今フレームでコマンドが終了しない場合、ブロックを実行した後、_HALT_ が実行され、次フレームに再実行されます。
・第1引数でイージングの種類を指定した場合、値に応じて加速度を踏まえた移動が行えます。指定できるオプションは以下の３２種類です（参考： http://easings.net/ja）。

イージングオプション		
:liner（初期値。線形移動）		:swing
:in_quad	:out_quad	:inout_quad
:in_cubic	:out_cubic	:inout_cubic
:in_quart	:out_quart	:inout_quart
:in_quint	:out_quint	:inout_quint
:in_expo	:out_expo	:inout_expo
:in_sine	:out_sine	:inout_sine
:in_circ	:out_circ	:inout_circ
:in_back	:out_back	:inout_back
:in_bounce	:out_bounce	:inout_bounce
:in_elastic	:out_elastic	:inout_elastic

PATH

オプション	説明
第1引数	[配列][フレーム数 , 補間の種類（省略時 :spline）]
ハッシュ	キー：遷移させたいプロパティ　値：[配列] 遷移させる値
ブロック	あり（ブロック引数無し）

・_LOOP_ のバリエーションです。ハッシュのキーで指定したコントロールの各プロパティを、値に設定している配列の各数値を基準点として、第1引数で指定したフレーム数かけて補間遷移させます。
・補間の種類には line と spline の2種類があり、line の場合は直線移動、spline の場合はBスプラインによる曲線移動が行われます。Bスプラインの場合、一般に基準点上は通過しないので注意してください。

NEXT

オプション	説明
無し	

・現在のループの残りの処理を飛ばし、ループを頭から再実行します。
・_LOOP_/_MOVE_/_PATH_ の付与ブロック内で使用できます。

BREAK

オプション	説明
なし	

・現在のループを終了し、次のコマンドを実行します。
・_LOOP_/_MOVE_/_PATH_ の付与ブロック内で使用できます。

コマンド：ユーザー定義コマンド

DEFINE

オプション	説明
第1引数	[シンボル] コマンド名
ハッシュ	無し
ブロック	ユーザー定義コマンドの本体

・ユーザー定義コマンドを登録します。登録したコマンドは組み込みコマンドと同じように呼びだせます。
・特定のコントロール内で定義したコマンドは、そのコントロール内でのみ呼びだせます。別々のコントロール内で同名のコマンドを定義した場合、それらは区別されます。

RETURN

オプション	説明
ブロック	あり（ブロック引数：無し）

・コマンドの実行を終了し、呼び出し元に戻ります。
・ブロックが付与されている場合、呼び出し元に戻った直後にそのブロックを評価します。

YIELD

オプション	説明
ハッシュ	実行するブロックに渡される。

・コマンドの呼び出し時に付与されたブロックを実行します。
・_YIELD_ が実行される際に付与ブロックが存在しない場合、例外が発生します。_CHECK_BLOCK_ を用いて付与ブロックの存在を判定できます。

ALIAS

オプション	無し
ハッシュ	
original_name	[シンボル] 既存のユーザー定義コマンド名
new_name	[シンボル] 新規に設定するユーザー定義コマンド名の別名

・定義済みのユーザー定義コマンドに別名を設定します。old オプションに設定したコマンドを、new オプションで設定した名前でも呼び出せるようにします。
・コマンドAに別名Bを与え、Aを再定義して内部からBを呼び出すことで、既存のユーザー定義コマンドをラップすることができます。
・組み込みコマンドには別名を設定できません。

コマンド：コマンドブロックの送信

SEND

オプション	説明
第1引数	（任意）[シンボル配列] コントロールへの相対パス
ハッシュ	
interrupt	[真偽値] コマンドリストの先頭に送信するかどうか（初期値 :false）
ブロック	送信するコマンドブロック（引数は _SEND_ に渡したハッシュ全て）

・第1引数にパス指定した子コントロールにコマンドブロックを送信します。
・送信されたコマンドブロックは、そのコントロールのコマンドリストの末端に連結されます。interrupt オプションに true が設定されている場合はコマンドリストの先頭に挿入します。

_SEND_ALL_

オプション	説明
第1引数	（任意）[シンボル] コントロールＩＤ
ハッシュ	
interrupt	[真偽値] コマンドリストの先頭に送信するかどうか（初期値 :false）
ブロック	送信するコマンドブロック

・直下の全ての子コントロールに対して同内容の _SEND_ を実行します。
・第1引数にコントロールＩＤが設定されている場合、同名の子コントロール群のみを送信対象とします。
・interrupt は _SEND_ と同様に適用されます。

コマンド：スクリプトファイル処理

INCLUDE

オプション	説明
第1引数	[文字列] 挿入するスクリプトファイルへのパス
ハッシュ	
parser	[シンボル] ファイルの評価時に使用するパーサー（初期値：nil）
ブロック	無し

・第1引数で指定されたスクリプトファイルを読み込み、コマンドが実行された位置に挿入します。
・スクリプトファイルには tsukasa 言語形式の .rb ファイルか、tks スクリプト形式の .tks ファイルを指定できます。.tks ファイルを読み込んだ場合は、自動的に tsukasa 言語形式に変換されます。
　また parser オプションで任意のパーサーを指定できます。現状では ":tks" のみ対応しています。

PARSE

オプション	説明
第1引数	[文字列] パースするスクリプト
ハッシュ	
parser	スクリプトの評価時に使用するパーサー（初期値：nil）
ブロック	無し

・第1引数で指定された文字列を tsukasa 言語として評価します。
・parser オプションが指定されている場合、指定されたパーサーで文字列を置換したのち、tsukasa 言語として評価します。現状では ":tks" のみ対応しています。

```
# サンプルコード
test = <<"EOF"
テスト
  _PUTS_ "test"
EOF
_PARSE_ test, parser: :tks
```

SERIALIZE

オプション	説明
第1引数	（任意）[配列] デシリアライズ対象のコマンドリスト
ハッシュ	
control	（任意）[シンボル配列] コントロールへの相対パス
ブロック	引数名 command_list でシリアライズされたコマンドリストを受け取る

・コントロールツリーを再構築可能な配列の形式で取得／設定します。いわゆるシリアライズ／デシアリアライズを可能にします。
・第1引数を設定しない場合、実行されたコントロール配下のツリーをコマンドリスト配列としてシリアライズし、ブロックに渡します。取得した配列を第1引数に設定した場合、その配列をデシリアライズしてコントロールツリーを再構築します。

・control オプションにコントロールへの相対パスが設定されている場合は、パスで指定されたコントロールを取得／設定の対象にします。
・コマンドリスト配列は [（シンボル）コマンド名 ,（ハッシュ）設定オプション] のペアからなる配列です。

EXIT

オプション	無し

・アプリを終了します。このコマンドを実行すると、そのフレームの終了時にアプリを終了します。

HALT

オプション	無し

・フレームの終了を示すコマンドです。このコントロールにおけるコマンドリストの探査を中断し、今フレーム中の更新処理を強制的に終了します。

_SCRIPT_PARSER_

オプション	説明
第1引数	無し
ハッシュ	
ext_name	[シンボル] ファイル識別子
path	[文字列] パーサーのファイルパス
parser	[シンボル] 登録するパーサークラス
ブロック	無し

・主にエンジン開発者が使用するコマンドです。変換が必要なスクリプトを使用する際の置換パーサーを動的に設定します。
・path で指定した ruby ソースコードファイルを require_relative で読み込みます。
・ext_name で指定した拡張子のファイルが _INCLUDE_ で読み込まれた際に、parser で指定した置換パーサークラスを実行します。
・標準ではこのコマンドを使用して tks スクリプトファイル用のパーサー TksPerser を登録しています。

コマンド：デバッグ用

EVAL

オプション	説明
第1引数	実行する Ruby 式（文字列）

・デバッグ用のコマンドです。第一引数で指定した文字列を Ruby 式として評価します。
・コマンドとプレーンの ruby 式が混在した場合、その実行順が直感と一致しない事がありますが、_EVAL_ を使えばスクリプトの見た目に沿ったタイミングで Ruby 式を実行できます。

PUTS

オプション	説明
第1引数	コンソール出力する値
ハッシュ	コンソール出力する値のハッシュ

・デバッグ用のコマンドです。第1引数もしくはハッシュで指定した値をコンソールに出力します。

デバッグコマンド

・デバッグ時に各種情報を出力するためのコマンド群です。実行すると、各種情報を標準出力に出力します。通常運用時には使用しないでください。

_DEBUG_TREE_

・そのコントロール配下のコントロールをツリー上に出力します。

_DEBUG_PROP_

・そのコントロールのプロパティの一覧を出力します。

_DEBUG_COMMAND_

・そのコントロールが保持しているコマンドリストを出力します。

Layoutable（内部モジュール）

・他のコントロールのベースになる特殊なコントロールです。ユーザーが生成することはできません。
・配置座標、サイズなどを管理します。

プロパティ

プロパティ	説明
x	[整数] コントロールの X 座標（省略時0）
y	[整数] コントロールの Y 座標（省略時0）

コマンド

無し

Layout

・描画機能を持つコントロールをグループ化するためのコントロールです。
・Layout の座標を更新すると、保持している子コントロールはその座標を原点として相対配置されます。

ベースコントロール（このコントロールの機能を全て使用できる）

Layoutable

プロパティ

無し

コマンド

無し

Window

・ゲーム自体を表すオブジェクトとして機能するコントロールです。通常は、コントロールツリーのルートコントロールになります。
・ウィンドウに依存する各種プロパティの取得／設定が可能です。マウスの値についても管理しています。

ベースコントロール（このコントロールの機能を全て使用できる）

```
Layout
Clickable
```

プロパティ

オプション	説明
mouse_x	[整数] マウスカーソルX座標
mouse_y	[整数] マウスカーソルY座標
mouse_offset_x	[整数] マウスカーソルX座標の前フレームからのオフセット増分値
mouse_offset_y	[整数] マウスカーソルY座標の前フレームからのオフセット増分値
mouse_wheel_pos	[整数] マウスホイールの現在値
mouse_enable	[真偽値] マウスカーソル表示有無
auto_close	[真偽値]「閉じる」ボタンを押した際にアプリを終了するか（初期値 true）
caption	[文字列] ウィンドウのタイトルバーに表示する文字列
bgcolor	[色配列] ウィンドウの背景色
icon_path	[文字列] ウィンドウのタイトルバーに表示するアイコンのファイルパス
cursor_type	[マウスカーソル定数] マウスカーソル形状
full_screen	[真偽値] フルスクリーンモード
screen_modes	(読み出し専用)[配列] フルスクリーン化可能な解像度のリスト([X幅 , Y幅 , リフレッシュレート] からなる配列)
inactive_pause	[真偽値] ウィンドウが非アクティブの時に更新処理を行うか（初期値 :true）
_INPUT_API_	[Object] システム用：ベースＡＰＩ（初期値：DXRuby::Input）
_WINDOW_API_	[Object] システム用：ベースＡＰＩ（初期値：DXRuby::Window）

・inactive_pause が true の場合、子コントロールを含めて全ての更新処理がスキップされます。描画は最終状態の物が行われます。
・cursor_type オプションで指定できるマウスカーソル定数は以下になります。

マウスカーソル定数	意味
IDC_APPSTARTING	標準の矢印カーソルと小さい砂時計カーソル
IDC_ARROW	標準の矢印カーソル
IDC_CROSS	十字カーソル
IDC_HAND	ハンドカーソル
IDC_HELP	矢印と疑問符

IDC_IBEAM	アイビーム（縦線）カーソル
IDC_NO	禁止カーソル（円に左上から右下への斜線）
IDC_SIZEALL	4 方向の矢印カーソル
IDC_SIZENESW	右上と左下を指す両方向矢印カーソル
IDC_SIZENS	上下を指す両方向矢印カーソル
IDC_SIZENWSE	左上と右下を指す両方向矢印カーソル
IDC_SIZEWE	左右を指す両方向矢印カーソル
IDC_UPARROW	上を指す垂直の矢印カーソル
IDC_WAIT	砂時計カーソル

・ベースＡＰＩのオプションはデバッグ時の利用を想定しています。内部処理に DXRuby 互換オブジェクトを設定する際に使います。

コマンド

_CHECK_REQUESTED_CLOSE_

オプション	説明
ブロック	あり

・ウィンドウの「閉じる」ボタンが押されている場合、付与ブロックを実行します。
・「閉じる」ボタンが押された際の挙動を制御したい場合、auto_close プロパティに false を設定し、このコマンドを使用してください。

RESIZE

オプション	説明
ハッシュ	
width	[整数] X幅
height	[整数] Y幅

・ウィンドウサイズを変更します。

Drawable（内部モジュール）

・他のコントロールのベースになる特殊なコントロールです。ユーザーが生成することはできません。
・描画機能を管理します。

プロパティ

プロパティ	説明
visible	[真偽値] コントロールを表示する（省略時 : true）
scale_x	[実数] 横の拡大率（省略時1）
scale_y	[実数] 縦の拡大率（省略時1）
center_x	[整数] 回転、拡大の中心点（省略時画像の中心）
center_y	[整数] 回転、拡大の中心点（省略時画像の中心）
alpha	[整数] アルファ値 (0 ～ 255)（省略時２５５）
blend	[シンボル] 合成方法（:alpha、:add、:add2、:sub）（省略時 :alpha）
	:none 透明色、半透明色もそのまま上書き
	:add ソースにアルファ値を適用
	:add2 背景に 255- アルファ値を適用
	:sub アルファ値を全ての色の合成
	:sub2 RGB の色をそれぞれ別々に合成
angle	[実数]360 度系で画像の回転角度を指定。
z	[整数] 描画順指定
offset_sync	[真偽値] 画時の指定座標 x/y に、画像の center_x/y で指定した位置が来るように補正される（省略時 false）
shader	[DXRuby::Shader] シェーダーとして使用するオブジェクト

・angle と、scale_x/y が同時に指定された場合は、拡大率が先に適用されます。
・shader は、Shader コントロールの shader プロパティから取得した値を設定します。サンプルコードを参考にしてください。

コマンド

無し

Image

・画像を保持し、描画と移動を管理するコントロールです。

ベースコントロール（このコントロールの機能を全て使用できる）

```
Control
Drawable
```

プロパティ

プロパティ	説明
path	[文字列] 読み込む画像ファイルのパス
width	[整数] 単色画像Xサイズ（初期値：1）
height	[整数] 単色画像Yサイズ（初期値：1）
color	[色配列] 単色画像背景色（初期値：[0,0,0,0]）
_RENDERTARGET_API_	[Object] システム用：ベースＡＰＩ（初期値：DXRuby::Rendertarget）
_FONT_API_	[Object] システム用：ベースＡＰＩ（初期値：DXRuby::Font）

・path で指定した画像を読み込んで、コントロールを初期化します。
・path を指定しなかった場合、width/height/color を元に単色画像を生成します。
・ベースＡＰＩのオプションはデバッグ時の利用を想定しています。内部処理に DXRuby 互換オブジェクトを設定する際に使います。

コマンド

DRAW

プション	説明
第1引数	[配列][シンボル] 描画対象のコントロールへのパス
ハッシュ	
x	[整数] 描画Ｘ座標
y	[整数] 描画Ｙ座標
scale	[数値] 拡大縮小率（省略時：拡大縮小しない）
ブロック	（任意）コマンドブロック

・第1引数で指定したコントロールを自身の [x,y] に描画します。
・scale が設定されている場合、設定した拡大縮小率が適用されます。

_SAVE_IMAGE_

オプション	説明
第1引数	[文字列] ファイルパス名
ハッシュ	
format	[定数] 保存する画像のフォーマット形式 (初期値 :FORMAT_PNG)
ブロック	無し

・現在の画像を format で指定したフォーマット形式でファイルに保存します。format には FORMAT_JPG/FORMAT_PNG/FORMAT_BMP/FORMAT_DDS の４種類を設定できます。
・_TO_IMAGE_ 組み込みユーザー定義コマンドと連携し、セーブ画面用のスクリーンショットサムネイル画像を作成することを意図しています。

LINE

オプション	説明
第1引数	無し
ハッシュ	
x1,y1,x2,y2	[整数] 直線の始点と終点
color	[色配列] 描画色
ブロック	無し

・[x1,y1]-[x2,y2] に color で指定した色の直線を描画します。

BOX

オプション	説明
第1引数	無し
ハッシュ	
fill	[真偽値] 内部を塗りつぶす (省略時 : false)
x1,y1,x2,y2	[整数] 四角形の始点と終点
color	[色配列] 描画色
ブロック	無し

・[x1,y1]-[x2,y2] に color で指定した色の矩形のラインを描画します。
・fill に true が設定されている場合、矩形の中を塗りつぶします。

CIRCLE

オプション	説明
第1引数	無し
ハッシュ	
fill	[真偽値] 内部を塗りつぶす (省略時 : false)
x,y,r	[整数] 円の中心座標と半径
color	[色配列] 描画色
ブロック	無し

・座標 [x,y] を原点に半径 r の円のラインを color で指定した色で描画します。
・fill に true が設定されている場合、円の中を塗りつぶします。

TRIANGLE

オプション	説明
第1引数	無し
ハッシュ	
fill	[真偽値] 内部を塗りつぶす（省略時：false）
x1,y1,x2,y2,x3,y3	[整数] 三角形の頂点座標
color	[色配列] 描画色
ブロック	無し

・[x1,y1]-[x2,y2]-[x3,y3] を頂点とし、color で指定した色の三角形のラインを描画します。
・fill に true が設定されている場合、矩形の中を塗りつぶします。

TEXT

オプション	説明
第1引数	[文字列] 描画する文字列
ハッシュ	
x,y	[整数] テキストを描画する座標（初期値：0）
size	[整数] 文字サイズ（初期値：24）
font_name	[文字列] フォント名（初期値："")
weight	[整数] 太さ（0〜10。初期値：4）
italic	[真偽値] 斜体（初期値：false）
color	[色配列] 描画色（初期値：[0,0,0,0]）
option	（任意）描画オプションを設定したハッシュ
ブロック	無し

・座標 [x,y] に、第一引数で指定した文字列を描画します。
・option を設定してより多彩な装飾が出来ます。詳細については Char コントロールを参照してください。

FILL

オプション	説明
第1引数	[色配列] 描画色
ハッシュ	無し
ブロック	無し

・全体を指定色で塗りつぶします。

CLEAR

オプション	無し

・全体を描画色を [0,0,0,0] で塗りつぶします。

PIXEL

オプション	説明

第1引数	無し
ハッシュ	
x,y	[整数] 描画色を取得／設定する座標
color	[色配列] 描画色
ブロック	色を取得するブロック（引数名：color）

・座標 [x,y] への色の取得／設定を行います。
・color が設定されている場合、指定座標にその描画色を設定します。
・ブロックが設定されている場合、指定座標の描画色をブロック引数の corlor オプションに設定し、ブロックを呼びだします。

COMPARE

オプション	**説明**
第1引数	無し
ハッシュ	
x,y	[整数] 描画色を比較する座標
color	[色配列] 比較する描画色
ブロック	条件成立時に実行するブロック

・座標 [x,y] の描画色と、color で設定した描画色を比較します。
・描画色が等しい場合、ブロックを実行します。

Input

・キーボード、十字キー、ゲームパッドなどの入力を受け付けるコントロールです。
・Input コントロールは _CHECK_ コマンドを拡張し、キーやボタンの押下状態をチェックできます。

ベースコントロール（このコントロールの機能を全て使用できる）

Control

プロパティ

プロパティ	説明
pad_number	[整数] 対象とするパッド番号（初期値：0）
x	[整数] 方向キーのX増分（-1,0,1 のいずれか）
y	[整数] 方向キーのY増分（-1,0,1 のいずれか）
_INPUT_API_	[Object] システム用：ベースＡＰＩ（初期値：DXRuby::Input）

・x/y プロパティは pad_number で指定した方向キーの値を取得します。
・ベースＡＰＩのオプションはデバッグ時に DXRuby 互換オブジェクトを設定することを想定しています。

コマンド

CHECK

・Input コントロールの _CHECK_ コマンドでは、以下の比較条件が追加で使用できます。

比較条件	説明	
:key_push	キーが押下された	
:not_key_push	キーが押下されていない	
:key_down	キーが継続押下されている	
:not_key_down	キーが継続押下されていない	
:key_up	キーが解除された	
:not_key_up	キーが解除されていない	
:pad_push	パッドボタンが押された	
:pad_down	パッドボタンが継続押下されている	
:pad_release	パッドボタンが解除された	
:mouse	マウス状態取得。値として以下のシンボルの配列を受け付ける。	
	:push	左クリック
	:down	左継続クリック
	:up	左クリック解除
	:right_down	右クリック
	:right_push	継続クリック
	:right_up	右クリック解除

・パッドボタンの判定は、pad_number プロパティで指定したパッドを対象にします。

Data

・任意の値を設定／取得できる特殊なコントロールです。
・ゲーム全体で共有する値を一時的に格納するのに用います。Data コントロールをシリアライズ／デシアライズすることで、容易にゲームデータのセーブ／ロードが可能です。

ベースコントロール（このコントロールの機能を全て使用できる）

Control

プロパティ

・Data コントロールは特定のプロパティを持ちません。その代わりに、_SET_で任意のプロパティ名を指定すると、そのプロパティを新規に追加します（内部的には _DEFINE_PROPERTY_ が実行されます）。
・追加されたプロパティは _GET_で値を取得することが可能になり、また _SERIALIZE_ 実行時にシリアライズされるプロパティの対象になります。
・一度も _SET_ されたことのないプロパティ名で _GET_ が実行された場合、例外が発生します。必ず先に _SET_で値を設定してください。

コマンド

無し

DrawableLayout

・描画機能を持つコントロールをグループ化するためのコントロールです。Layout と異なり自身が描画機能を持ち、単独でも単色の Image として機能します。
・内部的には DXRuby::RenderTarget クラスのインスタンスを保持していて、配下の子コントロールをそのオブジェクト上に描画します。
・DrawableLayout に透過や回転などを設定した場合、配下の子コントロールにも適用されます。

ベースコントロール（このコントロールの機能を全て使用できる）

Control
Drawable

プロパティ

オプション	説明
width	[整数] コントロールのX幅
height	[整数] コントロールのY幅
bgcolor	[色配列] 背景色（初期値：[0, 0, 0, 0]）
relative_x	[整数] 原点からの相対描画X座標（初期値：0）
relative_y	[整数] 原点からの相対描画Y座標（初期値：0）
_RENDERTARGET_API_	[Object] システム用：ベースＡＰＩ（初期値：DXRuby::Rendertarget）
_FONT_API_	[Object] システム用：ベースＡＰＩ（初期値：DXRuby::Font）

・width/height は必須です。
・DrawableLayout は毎フレーム bgcolor で指定した色で更新されます。
・relative_x/y に値を設定すると、設定した座標を原点と見なして相対位置に描画します。
・ベースＡＰＩのオプションはデバッグ時に DXRuby 互換オブジェクトを設定することを想定しています。

コマンド

LINE

オプション	説明
第1引数	無し
ハッシュ	
x1,y1,x2,y2	[整数] 直線の始点と終点
color	[色配列] 描画色
z	[整数] 描画順
ブロック	無し

・[x1,y1]-[x2,y2] に color で指定した色の直線を描画します。

BOX

オプション	説明
第1引数	無し
ハッシュ	
fill	[真偽値] 内部を塗りつぶす（省略時：false）
x1,y1,x2,y2	[整数] 四角形の始点と終点
color	[色配列] 描画色
z	[整数] 描画順
ブロック	無し

・[x1,y1]-[x2,y2] に color で指定した色の矩形のラインを描画します。
・fill に true が設定されている場合、矩形の中を塗りつぶします。

CIRCLE

オプション	説明
第1引数	無し
ハッシュ	
fill	[真偽値] 内部を塗りつぶす（省略時：false）
x,y,r	[整数] 円の中心座標と半径
color	[色配列] 描画色
z	[整数] 描画順
ブロック	無し

・座標 [x,y] を原点に半径 r の円のラインを color で指定した色で描画します。
・fill に true が設定されている場合、円の中を塗りつぶします。

TEXT

オプション	説明
第1引数	[文字列] 描画する文字列
ハッシュ	
x,y	[整数] テキストを描画する座標（初期値：0）
size	[整数] 文字サイズ（初期値：24）
font_name	[文字列] フォント名（初期値："")
weight	[整数] 太さ（0〜10。初期値：4）
italic	[真偽値] 斜体（初期値：false）
color	[色配列] 描画色（初期値：[0,0,0,0]）
option	（任意）描画オプションを設定したハッシュ
ブロック	無し

・座標 [x,y] に、第一引数で指定した文字列を描画します。
・option を設定してより多彩な装飾が出来ます。詳細については Char コントロールを参照してください。

Clickable（内部モジュール）

・他のコントロールのベースになる特殊なコントロールです。ユーザーが生成することはできません。
・マウス座標及びマウスクリックとの衝突判定を検出する機能を追加します。
・Clickable をインクルードしたコントロールは _CHECK_ コマンドを拡張し、点、矩形、円、三角形での判定の衝突判定の他、マスク画像を用いたドット単位での抜き色判定が出来ます。

プロパティ

オプション	説明
cursor_x	[読み取り専用][整数] コントロールの原点座標からのカーソル相対X座標
cursor_y	[読み取り専用][整数] コントロールの原点座標からのカーソル相対Y座標
shape	[配列]DXRuby::Sprite で衝突判定する形状。（省略時：[0,0]）
	2 要素 [x, y] 1 ピクセルの点
	3 要素 [x, y, r] 中心 (x, y) から半径 r のサイズの円
	4 要素 [x1, y1, x2, y2] 左上 (x1, y1) と右下 (x2, y2) を結ぶ矩形
	6 要素 [x1,y1,x2,y2,x3,y3] (x1, y1) 〜 (x2, y2) 〜 (x3, y3) を結ぶ三角形
colorkey_id	[シンボル] マスク画像として使用する子コントロールの ID
colorkey_border	[整数] 判定を受け入れる境界値（初期値：255）
_INPUT_API_	[Object] システム用：ベースＡＰＩ（初期値：DXRuby::Input）
_SPRITE_API_	[Object] システム用：ベースＡＰＩ（初期値：DXRuby::Sprite）

・shape に値を設定すると、指定した図形を当たり判定として使用します。通常は4要素で矩形を指定するのが多いと思われますが、他にも点、円、三角形を設定することもできます。
・Layout コントロール自体が持つ矩形情報と shape が持つ図形情報は独立しているので注意して
・colorkey_id が設定されている場合、図形の衝突判定に加えて、指定した子コントロールを用いたカラーキー衝突判定を行います。その場合は、カーソル座標のドットを取得して、そのドットのα値が colorkey_border 値以上であれば、衝突したとみなします。
・ベースＡＰＩのオプションはデバッグ時に DXRuby 互換オブジェクトを設定することを想定しています。

コマンド

CHECK

・ClickableLayout コントロールの _CHECK_ コマンドでは、以下の比較条件が追加で使用出来ます。

判定条件	内容
:collision	マウスのカーソル座標との衝突状態とマウスボタンの押下状態を識別する

・ClickableLayout コントロールがマウスのカーソル座標と衝突しているかを判定し、またその時のマウスボタンの押下状態を識別します。値として以下のシンボルの配列を受け付け、いずれかが成立している場合にブロックを実行します。

シンボル	説明

:cursor_over	カーソルが指定範囲に侵入した場合成立。
:cursor_out	カーソルが指定範囲の外に移動した場合成立。
:cursor_on	コントロール上にカーソルがある場合成立。
:cursor_off	コントロール上にカーソルがない場合成立。
:key_push	マウスボタンがコントロール上で押下された場合成立。
:key_down	マウスボタンがコントロール上で継続押下された場合成立。
:key_down_out	マウスボタンがコントロール外で押下された場合成立。
:key_up	マウスボタン押下がコントロール上で解除された場合成立。
:key_up_out	ウスボタン押下がコントロール外で解除された場合成立。
:right_key_push	マウス右ボタンがコントロール上で押下された場合成立。
:right_key_down	マウス右ボタンがコントロール上で継続押下された場合成立。
:right_key_down_out	マウス右ボタンがコントロール外で押下された場合成立。
:right_key_up	マウス右ボタン押下がコントロール上で解除された場合成立。
:right_key_up_out	マウス右ボタン押下がコントロール外で解除された場合成立。

ClicakbleLayout

・Clickable モジュールをコントロール化した物です。
・ボタンやクリッカブルマップなど、当たり判定が必要な描画要素を作る場合に使用します。

ベースコントロール（このコントロールの機能を全て使用できる）

Layout
Clickable

Sound

・BGMやSEを再生する為のコントロールです。ogg/wav ファイルを利用できます。
・1つのコントロールにつき1つの音源ファイルを管理します。

ベースコントロール（このコントロールの機能を全て使用できる）

`Control`

プロパティ

プロパティ	説明
path	[文字列] 音源ファイルへのパス

・音源を再生する為には、事前に path プロパティに音源ファイルを指定しておく必要があります。

コマンド

PLAY

プロパティ	説明
第1引数	（任意）再生ループ回数（初期値：1／ゼロなら無限ループ）
ハッシュ	
fadetime	フェード秒数
ブロック	無し

・音源を再生します。`fadetime` が指定されている場合、その秒数をかけてフェードします。

STOP

プロパティ	説明
第1引数	無し
ハッシュ	
fadetime	フェード秒数
ブロック	無し

・音源を停止します。`fadetime` が指定されている場合、その秒数をかけてフェードします。

PAUSE

プロパティ	説明
第1引数	無し
ハッシュ	
fadetime	フェード秒数

ブロック	無し

・音源を一時停止します。fadetime が指定されている場合、その秒数をかけてフェードします。

RESUME

プロパティ	説明
第1引数	無し
ハッシュ	
fadetime	フェード秒数
ブロック	無し

・音源の一時停止を解除します。fadetime が指定されている場合、その秒数をかけてフェードします。

VOLUME

プロパティ	説明
第1引数	音量（初期値：９０）
ハッシュ	
fadetime	フェード秒数
ブロック	無し

・音量を変更します。fadetime が指定されている場合、その秒数をかけてフェードします。

PAN

プロパティ	説明
第1引数	定位（−１００〜１００／初期値：０）
ハッシュ	
fadetime	フェード秒数
ブロック	無し

・音の定位を変更します。fadetime が指定されている場合、その秒数をかけてフェードします。

Char

・文字列描画を管理するコントロールです。

ベースコントロール（このコントロールの機能を全て使用できる）

`Control/Drawable`

プロパティ

オプション	説明
char	[文字列] 表示する文字列（初期値：nil）
size	[整数] 文字サイズ（初期値：２４）
font_name	[文字列] 文字フォント名（初期値：" ＭＳ 明朝 "）
weight	[整数] 太字の割合（初期値：false（無効））
italic	[真偽値] イタリックのオンオフ（初期値：false）
aa	[真偽値] アンチエイリアスのオンオフ（初期値 true）
color	[配列] 色（初期値 [255,255,255]）
edge	[真偽値] 縁文字のオンオフ（初期値 true）
edge_color	[配列] 縁文字：縁の色（初期値 [0, 0, 0]）
edge_width	[整数] 縁文字：縁の幅（初期値 2）
edge_level	[整数] 縁文字：縁の濃さ（初期値 16）
shadow	[真偽値] 影のオンオフ（初期値 true）
shadow_edge	[真偽値] 影：影の縁文字（初期値：false）
shadow_color	[配列] 影：影の色（初期値 [0, 0, 0]）
shadow_x	[整数] 影：オフセットＸ座標（初期値 0）
shadow_y	[整数] 影：オフセットＹ座標（初期値 0）
image_path	[文字列] 表示する画像へのファイルパス（初期値 nil）
_IMAGE_API_	[Object] システム用：ベースＡＰＩ（初期値：DXRuby::Image）
_FONT_API_	[Object] システム用：ベースＡＰＩ（初期値：DXRuby::Font）

・`char` に設定された1文字以上の文字列を、各プロパティの値を元に表示します。プロパティを更新すると、リアルタイムに文字の設定が更新されます。生成される画像サイズは、縦幅は文字サイズに、横幅は文字列と指定したフォントに依存します。縁文字などの文字装飾が設定されている場合、その装飾分が生成される画像サイズにプラスされます。`image_path` が設定されている場合、パスで指定された画像に設定された文字装飾をかけて表示します。この時、`char` で設定された文字は無視されます。

コマンド

CLEAR

・設定されているテキストをクリアします。`char` プロパティに `nil` を設定するのと同じです。

TextPage

・複数行からなるテキストウィンドウを管理するコントロールです。Char より高度な文字列制御を行います。TextPage コントロールは非常に多くの機能を持っています。具体的な使い方についてはサンプルコードを参照してください。また、多くの機能がユーザー定義コマンドによって実現されています。標準ユーザー定義コマンドの項も参考にしてください。

ベースコントロール（このコントロールの機能を全て使用できる）

Layout

プロパティ

ページの設定

オプション	説明
line_spacing	[整数] 行間のY幅（初期値：12）
character_pitch	[整数] 文字間のX幅（初期値：0）
line_height	[整数] 行の高さ（初期値：32）
rubi_size	[整数] ルビ文字の文字サイズ（初期値：12）
rubi_offset_x	[整数] ベース文字の原点からのルビ文字オフセットX座標（初期値：0）
rubi_offset_y	[整数] ベース文字の原点からのルビ文字オフセットY座標（初期値：-12）
rubi_pitch	[整数] ルビ文字間のX幅（初期値：12）
indent	[整数] インデント幅設定。2行目以降の開始X座標（初期値：0）
_FONT_API_	[Object] システム用：ベースＡＰＩ（初期値：DXRuby::Font）

・ベースＡＰＩのオプションはデバッグ時に DXRuby 互換オブジェクトを設定することを想定しています。

文字の設定（Char に直接渡されます）

オプション	説明
size	[整数] 文字サイズ（初期値：24）
font_name	[文字列] 文字フォント名（初期値：" ＭＳ 明朝 "）
weight	[整数] 太字の割合（初期値：false（無効））
italic	[真偽値] イタリックのオンオフ（初期値：false）
aa	[真偽値] アンチエイリアスのオンオフ（初期値 true）
color	[配列] 色（初期値 [255,255,255]）
edge	[真偽値] 縁文字のオンオフ（初期 false）
edge_color	[配列] 縁文字：縁の色（初期値 [0, 0, 0]）
edge_width	[整数] 縁文字：縁の幅（初期値 2）
edge_level	[整数] 縁文字：縁の濃さ（初期値 16）
shadow	[真偽値] 影のオンオフ（初期値 false）

shadow_edge	[真偽値] 影 : 影の縁文字 (初期値 : false)
shadow_color	[配列] 影 : 影の色 (初期値 [255, 255, 255])
shadow_x	[整数] 影 : オフセットX座標 (初期値 8)
shadow_y	[整数] 影 : オフセットY座標 (初期値 8)

コマンド

_INSERT_CHAR_

オプション	説明
第1引数	[文字列] 表示する文字列

・第一引数に指定した文字列に応じた Char コントロールを生成し、最終行の末端に連結します。
・コントロール内で _CHAR_RENDERER_ ユーザー定義コマンドが定義されている場合、生成した Char コントロールに _CHAR_RENDERER_ で定義したコマンドブロックを送信します。

_INSERT_CHAR_IMAGE_

オプション	説明
第1引数	[文字列] 画像へのファイルパス
ハッシュ	
:width	[整数] 文字として扱う際のX幅 (初期値 : 0)
:height	[整数] 文字として扱う際のY幅 (初期値 : 0)

・第1引数に指定したパスの画像に応じた Char コントロールを生成し、最終行の末端に連結します。外字の表示などに使用します。外字の表示などに使用します。
・width オプションを指定した場合、そのサイズの文字が描画された物とみなし、次の文字の描画X座標が補正されます。height オプションを指定した場合、文字の下揃え位置が調整されます。
・コントロール内で _CHAR_RENDERER_ ユーザー定義コマンドが定義されている場合、生成した Char コントロールに _CHAR_RENDERER_ で定義したコマンドブロックを送信します。

_INSERT_COMMAND_

オプション	説明
ハッシュ	
:width	[整数] 文字として扱う際のX幅 (初期値 : 0)
:height	[整数] 文字として扱う際のY幅 (初期値 : 0)
:skip	[真偽値]width で指定した値を文字送りに適用するか (初期値 : false)
ブロック	送信するコマンドブロック (ブロック引数 : x/y)

・付与ブロックを最終行の末端に送信します。文字列中に任意のアイコンなどを表示するのに使用します。
・width オプションを指定した場合、そのサイズの文字が描画された物とみなし、次の文字の描画X座標が補正されます。height オプションを指定した場合、文字の下揃え位置が調整されます。
・skip に true が設定された場合、文字送りが行われません。
・ブロック引数には、連結された要素が描画されるべき座標が x/y で与えられます。

TEXT

オプション	説明
第1引数	[文字列] 表示する文字列

・第1引数で指定した文字列を出力します。
・内部的には受け取った文字列を _CHAR_ コマンドに分解し、コマンドリストにスタックし直しています。
・コントロール内で _CHAR_WAIT_ ユーザー定義コマンドが定義されていた場合、_CHAR_ をスタックし直す際に、1文字ごとに挿入します。

RUBI

オプション	説明
第1引数	ルビ文字列

・第1引数に設定したルビ文字列を表示します。内部的にはルビ文字用に TextPage を生成し、オフセット描画させています。ルビ文字の表示する位置や書式については、プロパティで設定してください。

_LINE_FEED_

オプション	無し

・改行します。コントロール内で _LINE_WAIT_ ユーザー定義コマンドが定義されていた場合、文字の直後にコマンドを実行します。

FLUSH

オプション	無し

・ページをクリアします。

TileMap

・登録されているタイル画像を、指定した二次元配列に応じた２Ｄマップを構築します。
・内部的には DXRuby のタイルマップ機能のラッパーになります。

ベースコントロール（このコントロールの機能を全て使用できる）

DrawableLayout

オプション

オプション	説明
image_array	[配列] タイル画像 (DXRuby::Image)（初期値 : 空）
map_array	[二次元配列] タイル番号 (初期値 : 空）
map_x	マップ内描画開始Ｘ座標 (初期値 : 0）
map_y	マップ内描画開始Ｙ座標 (初期値 : 0）
size_x	Ｘ方向に描画するマップタイルの個数 (初期値 : 1）
size_y	Ｙ方向に描画するマップタイルの個数 (初期値 : 1）
z	描画順 (初期値 : 0）
_FONT_API_	[Object] システム用 : ベースＡＰＩ (初期値 : DXRuby::Font）

・image_array 配列の各要素にタイルになる画像を登録し、そのインデックス (タイル番号と呼びます) を map_array 二次元配列に設定すると、自動的に二次元のマップとして描画します。
・size_x/size_y に設定された分、Ｘ／Ｙ方向にタイルが表示されます。
・map_x/map_x に値が設定された場合、その座標を原点として相対表示します。これによって、コントロール自身の描画サイズより大きな２Ｄマップをスクロール表示できます。
・コントロールの自身の描画サイズは width/height で設定します (初期値３２×３２)。
・ベースＡＰＩのオプションはデバッグ時に DXRuby 互換オブジェクトを設定することを想定しています。

コマンド

_SET_TILE_

オプション	説明
第1引数	[整数] インデックス番号
ハッシュ	
path	[文字列] 画像ファイルへのパス
ブロック	無し

・タイル画像を第1引数で指定したタイル番号として登録します。既にその番号に別のタイル画像が登録されてる場合は上書きします。
・第1引数が省略されている場合、image_array の末端に追加されます。

_SET_TILE_GROUP_

オプション	説明
第1引数	[整数] インデックス番号
ハッシュ	
path	[文字列] 画像ファイルへのパス
x_count	[整数] X方向分割数（初期値：1）
y_count	[整数] Y方向分割数（初期値：1）
share_switch	[真偽値] 分割後のタイルチップ画像を共有管理とするか（初期値：false）
ブロック	無し

・path で指定した画像ファイルを x_count/y_count で指定した個数で分割してタイル画像を生成し、第1引数で指定したインデックス番号を起点に、左上から右下の順でタイル画像を登録します。
・第1引数が省略されている場合、image_array の末端に追加されます。
・share_switch に true を設定した場合、分割された個々の画像を共有でメモリ管理します（通常は false を指定してください）。

_MAP_STATUS_

オプション	説明
第1引数	[整数] タイル番号
ハッシュ	
x	map_array 上のX番目位置
y	map_array 上のY番目位置
ブロック	タイル番号を取得するブロック（引き数名：status）

・マップ座標 [x,y] に設定されているタイル番号を設定／取得します。
・第1引数が設定されていればタイル番号とみなして、マップ座標 [x,y] のタイル番号更新します。
・第1引数が設定されていない場合、マップ座標 [x,y] のタイル番号をブロック引数の status オプションに設定し、ブロックを実行します。

Shader

・DXRuby::Shader を管理するコントロールです。単体で使用することはありません。
・このクラスを継承して、Image/DrawableControl に適用するカスタムシェーダーを作成できます。
・RuleTransition コントロール、HorrorTextShader コントロールなどのコードを参照して下さい。

ベースコントロール（このコントロールの機能を全て使用できる）

Control

プロパティ

プロパティ	説明
shader	[DXRuby::Shader] シェーダーオブジェクト

コマンド

無し

RuleTransition

・ルールトランジションを実現するコントロールです。

ベースコントロール（このコントロールの機能を全て使用できる）

Shader

プロパティ

プロパティ	説明
path	[文字列] ルール画像へのファイルパス
counter	[整数] シェーダーの適用割合（0〜255。初期値：0）
vauge	[整数] シェーダーの曖昧度（初期値40）

コマンド

無し

使用方法

・path でルール画像を指定して読み込んだ後、shader プロパティ経由で DXRuby::shader オブジェクトを取得し、ルールトランジションを設定したいコントロールに設定します。
・以後は _MOVE_ コマンドなどで counter の値を変更すると、対象のコントロールにシェーダーが適用されます。
・サンプルコードを参照してください。

標準ユーザー定義コマンド（default_script.rb）

・実行環境に依存する機能を利用するために、pulgin_script/default_script.rb 内に、複数のユーザー定義コマンドが定義されています。これらのコマンドは起動時に自動的に読み込まれ、使用できます。

ガベージコレクション制御関連

・Ruby のガベージコレクション制御メソッド群のラッパーコマンドです。詳細については Ruby のリファレンスマニュアルを参照してください。

_GC_GARBAGE_COLLECT_

オプション	説明
ハッシュ	
full_mark	マイナーＧＣを動作させる場合は false（省略時：true）
immediate_sweep	LazySweep を行い場合は false（省略時：true）

・ガベージコレクタを強制的に動作させます。GC.start のラッパーコマンドです。

_GC_ENABLE_

オプション	説明
無し	

・ガベージコレクションの実行を許可します。GC.enable のラッパーコマンドです。

_GC_DISABLE_

オプション	説明
無し	

・ガベージコレクションの実行を禁止します。GC.disable のラッパーコマンドです。

_GC_LATEST_GC_INFO_

オプション	説明
第1引数	[シンボル] 取得したい情報
ブロック	あり（引数は取得したい情報）

・ガベージコレクションの最新の情報をハッシュで取得します。第一引数を設定した場合はその値のみを取得します。GC.latest_gc_info のラッパーコマンドです。

_GC_STATUS_

オプション	説明
第1引数	[シンボル] 取得したい情報
ブロック	あり（引数は取得したい情報）

・ガベージコレクションの統計情報をハッシュで取得します。第一引数を設定した場合はその値のみを取得します。GC.stat のラッパーコマンドです。

計測関連

_RUNNNING_TIME_

オプション	説明
ブロック	あり（ブロック引数　:time）

・アプリ起動時からの経過ミリ秒を取得します。コマンドを実行すると付与ブロックの time オプションに値が格納されます。

FPS

オプション	説明
第1引数	[整数] 秒間描画更新回数
ブロック	あり（ブロック引数　:fps）

・画面の秒間描画更新回数（fps/frame per second）を設定／取得します。
・第1引数が設定されている場合、その値を秒間描画更新回数に設定します。ブロックが設定されている場合、ブロック引数の fps オプションに現在の秒間描画更新回数を設定してブロックを実行します。

_CAPTURE_SS_

オプション	説明
第1引数	[文字列] 保存する画像ファイルのファイル名を含むパス
ハッシュ	
format	[定数] 保存する画像のフォーマット（初期値：FORMAT_PNG）
ブロック	無し

・現在の画面をキャプチャし、画像ファイルとして保存します。
・format には画像フォーマット定数 FORMAT_JPG/FORMAT_PNG/FORMAT_BMP/FORMAT_DDS のいずれかを指定します。それぞれ jpg/png/bmp/dds 画像フォーマットを表します。
・内部的には DXRuby::Window#get_screen_shot のラッパーになります。

ファイルダイアログ関連

_OPEN_FILENAME_

オプション	説明
第1引数	[文字列] ダイアログのタイトルバーに表示する文字列
ハッシュ	
filter	[文字列] 表示するファイルの種類と説明用の文字列

ブロック	ファイルパスを取得するブロック

・ファイルオープンダイアログを表示します。ファイルが指定された場合にはファイルのフルパスが、指定されなかった場合は nil をブロック引数の第1引数に設定し、ブロックを実行します。
・filter の初期値は "[[" すべてのファイル (*.*)", "*.*"]["JPG ファイル (*.jpg)", "*.jpg"]]" になります。

_SAVE_FILENAME_

オプション	説明
第1引数	[文字列] ダイアログのタイトルバーに表示する文字列
ハッシュ	
filter	[文字列] 表示するファイルの種類と説明用の文字列
ブロック	ファイルパスを取得するブロック

・ファイルセーブダイアログを表示します。ファイルが指定された場合にはファイルのフルパスが、指定されなかった場合は nil をブロック引数の第1引数に設定し、ブロックを実行します。
・filter の初期値は "[[" すべてのファイル (*.*)", "*.*"]["JPG ファイル (*.jpg)", "*.jpg"]]" になります。

_FOLDER_DIALOG_

オプション	説明
第1引数	[文字列] ダイアログのタイトルバーに表示する文字列
ハッシュ	
default_dir	[文字列] 初期表示時に選択されるディレクトリパス
ブロック	ファイルパスを取得するブロック

・フォルダダイアログを表示します。フォルダが指定された場合にはフォルダのフルパスが、指定されなかった場合は nil をブロック引数の第1引数に設定し、ブロックを実行します。

マウス／ゲームパッド関連

_PAD_CONFIG_

オプション	説明
第1引数	無し
ハッシュ	
pad_number	[整数] パッド番号（初期値：0）
pad_code	[定数] ボタン定数
key_code	[定数] キーコード定数
ブロック	無し

・パッドとキーの割り当てを変更します。設定したボタンとキーは、片方が押された場合に、条件判定でもう片方も押された物とみなします。

フォント管理

_INSTALL_FONT_

オプション	説明
第1引数	[文字列] フォントファイルへのファイルパス

・第1引数で指定した TrueType フォントファイルを一時的にインストールします。
・元々インストールされているフォントファイルと透過的に使用できます。

_INSTALL_PRERENDER_FONT_

オプション	説明
第1引数	[文字列] プリレンダフォントファイルへのファイルパス
ハッシュ	
font_name:	[文字列] 識別用フォント名

・第1引数で指定したプリレンダフォントファイルを一時的にインストールし。
・フォントは font_name で指定した名前で登録されます。詳細は「プリレンダフォント」 の項を参照してください。

キャッシュ管理

_IMAGE_REGIST_

オプション	説明
第1引数	[文字列] 画像へのファイルパス

・第1引数で指定した画像を画像キャッシュに登録します。Image コントロールを生成するより前に画像をキャッシュ登録できます。

_IMAGE_DISPOSE_

オプション	説明
第1引数	[文字列] 画像へのファイルパス

・第1引数で指定した画像を画像キャッシュから削除します。

ヘルパーコマンド（helper_script.rb）

plugin_script/helper_script.rb には、ゲーム開発を効率化するために用意されたヘルパーコマンドが登録されています。これらのコマンドは起動時に自動的に読み込まれ、使用できます。

ヘルパーコマンド

WAIT

オプション	説明
第1引数	（任意）[シンボル] データストア名
ハッシュ	(_CHECK_ で使用できる物を全て使用可能)
count:	（任意）[整数] ループ回数
input:	[ハッシュ] (_CHECK_INPUT_ で使用できる物が全て使用可能)
ブロック	あり (ブロック引数無し)

・_LOOP_ に幾つかの機能を追加したヘルパーコマンドです。
・_LOOP_ と同じようにブロックを繰り返し実行しますが、1回のブロックの実行が終了した直後に必ず _HALT_ を実行します。これにより、ブロックがフレーム毎に実行されるようになります。
・また、ループを終了する条件として _CHECK_ で使用する条件を指定できます。count は通常の _LOOP_ と同じループ回数指定、input は _CHECK_INPUT_ で使用する条件を指定できます。

_TO_IMAGE_

オプション	説明
第1引数	[配列][シンボル] コントロールへの相対パス
ハッシュ	
z	[整数] 生成した Image の描画順序 (省略時 Fixnum::INFINITY)
visible	[真偽値] 新規に作成した Image の可視設定 (省略時 true)
scale	[整数] 生成する画像の拡大率 (省略時：拡大しない)
widh/height	[整数] 生成 Image のサイズ
ブロック	（任意）あり

・Image コントロールを生成し、第1引数で指定したコントロールをそれに描画した後、コントロールリストの末端に追加します。主にトランジション用の元画像や、サムネイル画像の作成に使います。
・ブロックが設定されている場合、ブロックを生成した Image コントロールに送信します。

_CHECK_INPUT_

オプション	説明
ハッシュ	（任意）キー：[シンボル] 判定条件　値：[ハッシュ] 比較式
ブロック	あり (ブロック引数無し)

・標準の Input コントロールを判定する為のラッパーコマンドです。[:_ROOT_, :_INPUT_] に対して _

CHECK_ コマンドを実行します。

汎用ボタン

_BUTTON_BASE_

オプション	説明
第1引数	[シンボル] 作成するボタンコントロールのＩＤ
ハッシュ	（生成される ClickableLayout に渡す）
ブロック	あり

・ClickableLayout を用いた汎用的なボタンコントロールを実装する為のヘルパーコマンドです。通常は下記の _IMAGE_BUTTON_/_TEXT_BUTTON_ を使用します。

_IMAGE_BUTTON_

オプション	説明
第1引数	[シンボル] 作成するボタンコントロールのＩＤ
ハッシュ	
width/height	コントロールのサイズ（省略時 256）
上記以外	_BUTTON_BASE_ に渡される。
ブロック	あり

・最低限の機能を持ったボタンコントロールを生成します。
・コマンドが付与されている場合、生成されたボタンコントロールに送信されます。
・以下のプロパティで各状態の時に表示する画像を指定できます。省略時は標準の物になります。

オプション	説明
:normal	通常時の画像
:over	カーソルがコントロールに重なった時の画像
:down	マウスがクリックされている時の画像

・ボタンのサンプルコードとして利用してください。また、幾つかのサンプルコードでも使用しています。

_TEXT_BUTTON_

オプション	説明
ハッシュ	
width	[整数] ボタンX幅（初期値：128）
height	[整数] ボタンY幅（初期値：32）
text	[文字列] 表示文字列（初期値："）
out_color	[色配列] カーソルがボタン外にある時の背景色（初期値：[0,0,0]）
in_color	[色配列] カーソルがボタン上にある時の背景色（初期値：[255,255,0]）
上記以外	_BUTTON_BASE_ に渡される。
ブロック	あり（ブロック引数無し）

・文字列を伴うボタンを生成します。汎用的なボタンコントロールとして使用できます。

・ブロックが付与されている場合、生成時に送信します。
・ボタンのサンプルコードとして利用してください。また、幾つかのサンプルコードでも使用しています。

コマンド：ゲーム情報のセーブ／ロード

_SYSTEM_SAVE_

オプション	説明
第1引数	ファイルの保存先パス

・第1引数で指定したファイルを作成し、[:_ROOT_,:_SYSTEM_] の値をシリアライズして保存します。

_SYSTEM_LOAD_

オプション	説明
第1引数	ファイルの保存先パス

・第1引数で指定したファイルを取得し、[:_ROOT_,:_SYSTEM_] の値をシリアライズして再設定します。

_LOCAL_SAVE_

オプション	説明
第1引数	ファイルの保存先パス

・第1引数で指定したファイルを作成し、[:_ROOT_,:_LOCAL_] の値をシリアライズして保存します。

_LOCAL_LOAD_

オプション	説明
第1引数	ファイルの保存先パス

・第1引数で指定したファイルを取得し、[:_ROOT_,:_LOCAL_] の値をシリアライズして保存します。

_QUICK_SAVE_

オプション	説明
第1引数	ファイルの保存先パス

・第1引数で指定したファイルを作成し、コマンドが実行されたコントロールと、その下にあるコントロールのツリー全ての値をシリアライズして保存します。
・個々のコントロールが保持しているコマンドは保存されません。その為、TileMap コントロールなど、コマンドによるアニメーションを前提とするコントロールは正しく機能しないので注意してください。

_QUICK_LOAD_

オプション	説明
第1引数	ファイルの保存先パス

・第1引数で指定したファイルを取得し、実行されたコントロール上にデシアライズします。

テキストウィンドウ（text_layer_script.rb）

・以下のコマンドは TextPage コントロールを使いやすくするためのヘルパーメソッドです。これらのコマンドは、実行されると _ROOT_ のコントロールに追加された _DEFAULT_TEXT_PAGE_ プロパティに格納された id のコントロールを対象として処理を実行します。このプロパティは標準では自動で生成される :text0 が設定されています。

_TEXT_WINDOW_

オプション	説明
第1引数	[シンボル] 作成するテキストウィンドウの ID
ハッシュ	TextPage コントロールに渡す物と同じ
ブロック	あり

・TextPage コントロールを、標準的な _CHAR_WAIT_/_LINE_WAIT_/_CHAR_RENDERER_ を定義した状態で生成します。これらのコマンドを記述するためのサンプルコードとしてもお使いください。
・ブロックがある場合、生成された TextPage コントロールに送信されます。

PAUSE

オプション	説明
ブロック	あり

・TextPage コントロールの処理を一時的に停止し、同時にメインの処理を待機状態に移行します。
・ブロックが設定されている場合、TextPage コントロールの末行に送信します。
・キー入力などを検知すると、待機状態が解除されます。

_LINE_PAUSE_/lp

オプション	無し

・_PAUSE_ のバリエーションになります。行クリック待ちアイコンを表示して _PAUSE_ を実行します。待機状態が解除されると、行クリック待ちアイコンは削除されます。
・lp は短縮記述用の名称です。

_END_PAUSE_/ep

オプション	無し

・_PAUSE_ のバリエーションになります。ページクリック待ちアイコンを表示して _PAUSE_ を実行します。待機状態が解除されると、ページクリック待ちアイコンは削除されます。
・ep は短縮記述用の名称です。

_CHAR_SET_

オプション	説明
ハッシュ	TextPage コントロールへ渡される

・_DEFAULT_TEXT_PAGE_ に対して _SET_ を実行します。

_CHAR_RUBI_

オプション	説明
ハッシュ	TextPage コントロールへ渡される

・_DEFAULT_TEXT_PAGE_ に対して _RUBI_ を実行します。

_CHAR_IMAGE_

オプション	説明
第1引数	[文字列] 画像へのファイルパス

・第一引数で指定した画像を文字のビットマップとみなし、_DEFAULT_TEXT_PAGE_ に追加します。

_WAIT_FRAME_

オプション	説明
第1引数	[整数] 待機フレーム数

・_DEFAULT_TEXT_PAGE_ に登録されている _CHAR_WAIT_ コマンドを更新します。

ユーティリティーコマンド：ラベル

・既読フラグを管理するために、スクリプト中にラベルを埋め込むサポートコマンドです。

LABEL

オプション	説明
第1引数	無し
ハッシュ	
chapter	[シンボル] ラベル
id	[整数] 同 chapter 内の一意な値
ブロック	ラベルを付与するブロック

・_LABEL_ が実行されると、chapter と id の情報がグローバルストアに既読フラグとして記録され、その後にブロックが実行されます。
・chapter と id の組はゲーム全体で一意であるとされ、この機構を利用して既読フラグを管理します。
・chapter を省略した場合、直前に使用された chapter を設定した物とみなします。最初は必ず設定する必要があります。
・id を省略した場合、同じ chapter 内の最大の数値+1を設定した物とみなします。その chapter が一度も指定されていなかった場合は0になります。

ラベル行（tks スクリプトのオプション機能）

・tks スクリプトの補助機能として、既読フラグを管理する _LABEL_ ユーザー定義コマンドの省略記法

としてラベル行がサポートされています。

ラベル書式

・先頭が * で始まる行がラベル行として扱われます。ラベル行は以下のように記述します。

```
* [chapter] [id]
```

・chapter、id はそれぞれ _LABEL_ コマンドのそれぞれの引数に対応します。id は省略可能です。

起動時に自動的に生成されるコントロール群

・司エンジンが起動すると、自動的に default_script.rb/utility_script.rb を読み込む他、以下のコントロールを自動的に生成します。ユーザーはこれらのコントロールを自由に使用できます。

ID	コントロール	補足
:text0	TextPage	
:base	Image	Z値：0
:img0	Image	Z値：1000
:img1	Image	Z値：2000
:img2	Image	Z値：3000
:_INPUT_	Input	汎用的にキー入力を受け付ける
:_SYSTEM_	Data	環境全体で共有するデータストアを想定
:_LOCAL_	Data	個別の環境で使用するデータストアを想定
:_TEMP_	Data	一時的に使用するデータストアを想定
:_TEXT_WINDOW_TEMP_	Data	_TEXT_WINDOW_ 用のデータストア

司エンジンのフォルダ構成

司エンジン開発フォルダルート直下のフォルダとファイル構成は以下になります（ファイルについては最低限の物のみ説明しています）。

tsukasa フォルダ

main.exe　……実行ファイルです。実行すると main.rb を実行します
main_dev.exe　……main.exe にデバッグ版です。標準出力用のウィンドウを表示します。
main.rb　……ruby のソースコードです。司エンジンの初期設定を行い、first.rb を司スクリプトとして実行します。
first.rb　……最初に実行される司スクリプトファイルです。初期状態では "./script/demo.rb" を読み込みます。このファイルを書き換えることで、任意の司スクリプトを実行できます。
rakefile　……rake 用のスクリプト。現在は test タスク /spec タスクのみ用意
.rspec　……RSpec の実行時オプション格納ファイル
GemFile　……Bundler 用のインストールパッケージ記述ファイル
GemFile.lock　……Bundler で使用
README.md　……readme ファイル。概要と更新履歴を収録。
init.rb　※内部で使用
Ayame.dll　※ Ruby で使用
LIBEAY32.dll　※ Ruby で使用
libffi-6.dll　※ Ruby で使用
msvcrt-ruby220.dll　※ Ruby で使用

tsukasa/script フォルダ

・司スクリプト /tks スクリプトを格納するフォルダです。ゲームで使用されるスクリプトを配置することを想定としています。

tsukasa/script/sample フォルダ

・サンプルコードが収納されています。
demo.rb　……サンプルコードのランチャーです。
block フォルダ　……ブロック崩しゲーム
demo_game フォルダ　……ノベル脱出ゲーム
jump_action フォルダ　……ジャンプアクションゲーム
nomaid フォルダ　……野メイド
sample フォルダ　……サンプルコード群

tsukasa/doc フォルダ

・ガイドブックとドキュメントを格納するフォルダです。

tsukasa/datastore フォルダ

・セーブデータなど保存を想定しているフォルダです。初期状態では空です。

tsukasa/plugin_control フォルダ

・カスタムコントロール用の ruby ファイルを格納するフォルダです。このフォルダに配置された .rb ファイルは自動的に ruby プログラムとして読み込まれます。

HorrorTextShader.rb　……カスタムシェーダーサンプル

tsukasa/plugin_script フォルダ

・ユーザー定義コマンド用の司スクリプトを格納するフォルダです。このフォルダに配置された .rb ファイルは自動的に tsukasa 言語として読み込まれます。

default_script.rb　……Ruby/DXRuby のラッパーコマンド群
helper_script.rb　……tsukasa のラッパーコマンド群
text_layer_script.rb　……TextPage のヘルパーコマンド群

tsukasa/resource フォルダ

・画像ファイルや音声ファイルの格納を想定したフォルダです。サンプルで使用するファイルが収納されています。

char フォルダ　……立ち絵画像
Fonts フォルダ　……プリレンダ済みフォント
icon フォルダ　……アイコン画像
music フォルダ　……BGM
rule フォルダ　……トランジション用ルール画像

tsukasa/system フォルダ

・司エンジンの ruby ソースコードが格納されています。

Tsukasa.rb　……司エンジンを構成する最低限のコードを読み込みます。

tsukasa/lib フォルダ

※ Ruby で使用するシステムフォルダ。

tsukasa/test フォルダ

・ユニットテスト（Minitest）用の ruby ソースコードが格納されています。

tsukasa/spec フォルダ

・ユニットテスト（Rspec）用の ruby ソースコードが格納されています。

tsukasa/tools フォルダ

・司エンジンをサポートする外部ツールが格納されています。

FontDataMaker.rb　……TrueTypeフォントデータを司エンジンで使用できるプリレンダフォントデータに変換します。

ConvertFont.rb　……FontDataMaker.rb の内部で使用しています。

組み込み定数一覧

キーコード定数	キーボード上の表記
K_ESCAPE	Esc
K_1	1
K_2	2
K_3	3
K_4	4
K_5	5
K_6	6
K_7	7
K_8	8
K_9	9
K_0	0
K_MINUS	- *2
K_EQUALS	= *1
K_BACK	Backspace
K_TAB	Tab
K_Q	Q
K_W	W
K_E	E
K_R	R
K_T	T
K_Y	Y
K_U	U
K_I	I
K_O	O
K_P	P
K_LBRACKET	[
K_RBRACKET	]
K_RETURN	Enter
K_LCONTROL	左 Ctrl
K_A	A
K_S	S
K_D	D
K_F	F
K_G	G
K_H	H
K_J	J
K_K	K
K_L	L

キーコード定数	キーボード上の表記
K_SEMICOLON	;
K_APOSTROPHE	' *1
K_GRAVE	` *1
K_LSHIFT	左 Shift
K_BACKSLASH	¥
K_Z	Z
K_X	X
K_C	C
K_V	V
K_B	B
K_N	N
K_M	M
K_COMMA	,
K_PERIOD	.
K_SLASH	/ *2
K_RSHIFT	右 Shift
K_MULTIPLY	* *3
K_LMENU	左 Alt
K_SPACE	Space
K_CAPITAL	Caps Lock
K_F1	F1
K_F2	F2
K_F3	F3
K_F4	F4
K_F5	F5
K_F6	F6
K_F7	F7
K_F8	F8
K_F9	F9
K_F10	F10
K_NUMLOCK	NumLock *3
K_SCROLL	ScrollLock
K_NUMPAD7	7 *3
K_NUMPAD8	8 *3
K_NUMPAD9	9 *3
K_SUBTRACT	- *3
K_NUMPAD4	4 *3
K_NUMPAD5	5 *3

キーコード定数	キーボード上の表記
K_NUMPAD6	6 *3
K_ADD	+ *3
K_NUMPAD1	1 *3
K_NUMPAD2	2 *3
K_NUMPAD3	3 *3
K_NUMPAD0	0 *3
K_DECIMAL	. *3
K_F11	F11
K_F12	F12
K_F13	F13 *1
K_F14	F14 *1
K_F15	F15 *1
K_KANA	カタカナひらがな
K_CONVERT	変換
K_NOCONVERT	無変換
K_YEN	¥
K_NUMPADEQUALS	= *3 *1
K_PREVTRACK	^
K_AT	@
K_COLON	:
K_UNDERLINE	_ *1
K_KANJI	半角 / 全角
K_NUMPADENTER	Enter *3
K_RCONTROL	右 Ctrl
K_NUMPADCOMMA	, *3 *1
K_DIVIDE	/ *3
K_RMENU	右 Alt
K_PAUSE	PauseBreak
K_HOME	Home
K_UP	↑
K_LEFT	←
K_RIGHT	→
K_END	End
K_DOWN	↓
K_INSERT	Insert
K_DELETE	Delete
K_LWIN	左 Windows
K_RWIN	右 Windows

キーコード定数	キーボード上の表記
K_APPS	アプリケーション
K_BACKSPACE	Backspace
K_NUMPADSTAR	* *3
K_LALT	左 Alt
K_CAPSLOCK	CapsLock
K_NUMPADMINUS	- *3
K_NUMPADPLUS	+ *3
K_NUMPADPERIOD	. *3
K_NUMPADSLASH	/ *3
K_RALT	右 Alt
K_UPARROW	↑
K_PGUP	PageUp
K_LEFTARROW	←
K_RIGHTARROW	→
K_DOWNARROW	↓
K_PGDN	PageDown

*1　日本で使われている標準的な 109 キーボードに存在しないキーです。

*2　テンキー (Numpad) にあるキーではなくメインキーボード側のキーです。

*3　メインキーボードにあるキーではなくテンキー (Numpad) にあるキーです。

マウスボタン定数	
M_LBUTTON	左ボタン
M_MBUTTON	中ボタン
M_RBUTTON	右ボタン

パッド定数
P_UP
P_LEFT
P_RIGHT
P_DOWN
P_BUTTON0
P_BUTTON1
P_BUTTON2
P_BUTTON3
P_BUTTON4
P_BUTTON5
P_BUTTON6
P_BUTTON7
P_BUTTON8
P_BUTTON9
P_BUTTON10
P_BUTTON11
P_BUTTON12
P_BUTTON13
P_BUTTON14
P_BUTTON15
アナログ左スティックのデジタル入力
P_L_UP
P_L_LEFT
P_L_RIGHT
P_L_DOWN
アナログ右スティックのデジタル入力
P_R_UP
P_R_LEFT
P_R_RIGHT
P_R_DOWN
アナログ POV のデジタル入力
P_D_UP
P_D_LEFT
P_D_RIGHT
P_D_DOWN

色定数
C_BLACK
C_RED
C_GREEN
C_BLUE
C_YELLOW
C_CYAN
C_MAGENTA
C_WHITE
C_DEFAULT

※ `system/Constants.rb` も参照してください。

補足

★本書で解説していない項目について

・カスタムコントロールの自作方法や、レンダリング済みフォントデータの作成方法、ユニットテストの方法などの ruby の本格的な知識が必要な物については、tsukasa/doc フォルダ内に配置されたテキストファイルに概要を記載しているので、ご確認下さい。

★今後のアップデート予定

ゲームのリリース方法／暗号化／パッキング

・司エンジンで作成したゲームはそのままリリースすることが可能ですが、現状ではソースコードが隠蔽されず、画像ファイルなどのリソースデータもプレーンなままになってしまいます。
・吉里吉里ではリソースファイルを暗号化し、かつ1つのファイルにパッキングする機能があります。司エンジンでも将来的には類似のアプローチを採用するつもりです。

３Dへの対応

・ＡＤＶにおける２Ｄと３Ｄのカットシーンは見た目がまったく異なりますが、キャラを表示し、移動させ（３Ｄではカメラ側も移動する）、見た目を切り替える（２Ｄではトランジション、３Ｄではモーションの指定）という点で見れば、記述する内容はそこまで大きな違いがありません（演出する軸が1次元増えるので、作業労力はとんでもなく増えますが）。なので、将来的には３Ｄにも対応したいと考えています。

ネイティブ実装

・現在開発している司エンジンの Ruby 実装は、第1部で説明した通り、既存のゲームプログラミングが抱える課題を解決する手法についての実証モデルです。Ruby で実装したのは、著者が使い慣れている言語なのと、Ruby 用のゲームフレームワーク DXRuby が非常に使いやすい為です。
・現状でも充分実用になると考えていますが、言語仕様を作り込んだ結果、Ruby の内部 DSL であることが制約を生む箇所がいくつか発生してきました。ネイティブ実装にすることでこれらを解決できるかもしれません。また、現状の司エンジンは Windows 上でしか動きませんが、ネイティブ実装にすることで移植性を高められるかもしれません。Unity/UnrealEngine への移植も検討しています。

おわりに

中学生時代の著者が初めて入手したパソコンは8ｂｉｔのMSX2+規格マシン「Ｐａｎａｓｏｎｉｃ A1−WSX」でした。脇にフロッピーのスロットがあるキーボード一体型のプラスチックボディのデザインが、大人のホビーの雰囲気があって大好きでした。

MSXは電源を入れるとMSX−BASICが立ち上がり、そのままプログラミングができるようになっていました。月刊の専門誌には読者が投稿したゲームのソースコードが何作も掲載され、意味も分からないまま1文字ずつ打ち込んで遊んでいました。

ある日、そのように打ち込んだゲームの中で、街を発展させるＳＬＧに猛烈にはまり、内部パラメータの成長率をいじってみたいと思ったのが、土屋がプログラミングを始めたきっかけでした。ＢＡＳＩＣの解説書を片手にソースコードを解析し、ついに成長率を設定するロジックを見つけ、普通ではあり得ない成長速度で街が発展していくのを見て、「これでどんなゲームでも作れる」と感じた物でした。

その後、ＢＩＯＳコール、Ｚ８０マシン語、ＶＤＰアクセス、メモリスロット切り替えなど、MSXの内部仕様を可能な限り勉強し、その思いはより確かな物になりました。当時はメモリも６４ＫＢしかなかったこともあり、パソコンの全ての領域まで自分の手が届く気になれたのです。

時は移り変わり、現代。パソコンの能力は指数的に向上しました。ＣＰＵ速度やメモリ容量だけでなく、フォトリアルな3D映像やＰＣＭ音源で、ゲームの表現力は信じられない進化を遂げました。しかし、逆にゲームは大変に作りにくくなってしまいました。我々は望む望まざるに関わらず、電源を入れればＢＡＳＩＣが起動した時代から、既に、遥か遠くにやってきてしまったのです。

ただし、ゲームを作るのが難しい理由は、一般に言われるように画像やポリゴンモデル、音楽を作るコストが上がったからだけではなく、プログラミング言語についてより根深い問題があるのではないかということを、著者は長年悶々と考え続けていました。その結果作ったのが司エンジンなのです。

著者は大手ゲームメーカーでプランナーをしていた期間があり、ゲームプログラマーの知り合いも多いのですが、彼らは職人肌、天才気質の方が多く、著者が抱えている問題領域を肌感覚として理解してもらえず「理屈を並べる時間があったら今あるツールで一個でも多くゲームを作りなさい」と言われることが大半でした。本書の完成によって、ようやくこう答えることができます。「これを読んでください、土屋の言いたい事が全部書いてあります」と。

まだ検証の為の機能実装が終わった段階で、実用性の証明はこれからになりますが、一歩一歩進んで行ければと思っています。その為には多くのフィードバックが必要です。ぜひ司エンジンを使ってゲームを作ってみてください。

最後に謝辞を。DXRuby の作者である mirichi さんには、土屋の無茶なお願い（DirectX の標準機能にない袋文字対応とか）に対し、真摯に DXRuby をアップデートし続けて頂きました。DXRuby がなければ、そもそも司エンジンを作ることはありませんでした。ありがとうございました。DXRuby のヘビーユーザーであるあおいたくさんにはスクリプトパーサーの実装、ＢＧＭサンプル音源の提供、野メイドの移植許可など多くの協力を頂きました。ハイドさんには開発初期にコミッターとして様々なアイデアを頂きました。また、ここに書ききれないほどの多くの方に記事レビューでご協力いただきました。協力してくださった全ての皆さんに感謝します。

最初に自前のゲームエンジンを作りたいと考え始めたのは、もう１０年以上前のことになります。ここまで来るのに１０年かかりました。もう少し早く形にしたかったという気持ちはありますけれども、とはいえ、これは始まりにすぎません。tsukasa 言語／司エンジンの今後の発展に、どうぞご期待ください。

それではまたお会いしましょう。土屋つかさでした。

奥付

メッセージ指向ゲーム開発言語「司エンジン」ガイドブック第3版 (v2.2 対応)

2016 年 6 月 24 日
　第1版「メッセージ指向ゲーム開発言語「司エンジンガイドブック」」発行
2016 年 11 月 13 日
　第2版「ポスト・ゲームエンジン　tsukasa 言語が示す次世代ゲームエンジンアーキテクチャ」発行
2017 年1月 31 日
　第3版第1刷「メッセージ指向ゲーム開発言語「司エンジン」ガイドブック第3版 (v2.2 対応)」発行
2017 年 2 月 14 日
　第3版第2刷発行（第1刷から v2.2 正式リリースまでの修正分を反映）

著者　土屋つかさ
　　TwitterID:@t_tutiya
　　E-mail:skip081-nowornever@yahoo.co.jp
発行　サークル染井吉野
印刷　株式会社ＰＯＰＬＳ

※本書は２０１６年 11 月 13 日に発刊された同人誌「ポスト・ゲームエンジン　tsukasa 言語が示す次世代ゲームエンジンアーキテクチャ」の内容を改訂し、v2.2 に対応させた物です。

ソースコードについて

　司エンジンは GitHub 上（https://github.com/t-tutiya/tsukasa/）でオープンソース開発を行っています。最新版のコードについてはこちらを確認してください。

著者紹介：土屋つかさ

　ライトノベル作家／デジタル・アナログゲームデザイナー／プログラマー。
　中学時代にＭＳＸ２+とＴＲＰＧに出会い、プログラミングとアナログゲームにハマる。ＳＥ職を経てスクウェアエニックスにゲームプランナーとして所属、その後角川スニーカー文庫よりライトノベル作家としてデビュー。現在はフリーで文筆活動をこなしつつ時間を見つけて司エンジンの開発を続けている。
　最近はゲーム関連のノベライズを多くこなし、「ゲームもプログラムも理解しているラノベ作家」として上手いこと業界での立ち位置を確保できないものかと何年ももがいている。ノベライズ作に「小説　名探偵コナン（ノベライズ担当）」「シュタインズゲートゼロ（シナリオライターとして参加）」など。
※本書の商業出版化オファー or よろずテクニカルライティングのお仕事お待ちしています！>＜

・本書は著作権上の保護を受けています。本書の一部あるいは全部について、著作者から文書による許諾を得ずに、いかなる方法においても無断で複写、複製することは禁じられています。