マルチスレッドによるゲームループ管理

– マルチスレッドに最適化したゲームシステムを統括 –

Draft

2014 年 3 月 13 日　初稿

板垣 衛

■ 改訂履歴

稿	改訂日	改訂者	改訂内容
初稿	2014 年 3 月 13 日	板垣 衛	（初稿）

Draft

■ 目次

Draft

■ 概略

マルチスレッドに最適化した各サブシステムを統括するメインループを設計する。

ファイルマネージャ、リソースマネージャ、シーンマネージャ、入力デバイスマネージャ、サウンドマネージャといった主要なサブシステム、および、描画スレッドを統括し、ゲームシステム全体を制御する。

■ 目的

本書は、マルチスレッドに最適化した各サブシステムを統括し、マルチスレッドゲームシステムのアーキテクチャを確立することを目的とする。

各サブシステムは、別紙の「開発を効率化するためのファイルシステム」、「開発の効率化と安全性のためのリソース管理」、「ゲーム全体を円滑に制御するためのシーン管理」、「反応性と安全性を考慮した入力デバイス管理」、「リソース管理を最適化するためのサウンドシステム」にて詳細を説明する。

また、別紙の「「サービス」によるマルチスレッドの効率化」で示す「並行ジョブスケジューラ」、「効率化と安全性のためのロック制御」で示す「スレッドセーフなシングルトン」と「ローカルシングルトン」、「効果的なデバッグログとアサーション」で示す「コールポイント」（プロファイラ機能部分）、「ユニットテストと継続的ビルド」で示す「ユニットテスト」も活用する。

本書のメインループを含め、これらのサブシステムが密接に連携することで、マルチスレッドの効果を最大限に発揮する。

■ 要件定義

▼ 基本要件

本書が扱うシステムの基本要件は下記のとおり。

- ゲーム全体の初期化処理と終了処理を行うものとする。
  - メモリ割り当てや各サブシステムの初期化／終了、常駐スレッドの起動／終了などを扱う。

Draft

・ フレームレートと処理時間の管理を行うものとする。

・ 各サブシステムの処理を順次実行するものとする。
  - ➢ 不整合を起こさない処理順序を保証する。
  - ➢ マルチスレッドに最適化した処理順序とする。

・ 描画スレッドを用意獅子、メインスレッドと描画スレッドのリレー処理を行うものとする。

・ 「デバッグポータル」に対応するものとする。
  - ➢ デバッグビルドのゲームを起動した時に最初に起動する画面。
  - ➢ 「特定の章」や「特定のレベル」、「特定のイベント」、「特定の戦闘」といった「シーン」を直接起動するためのショートカットメニュー画面。
  - ➢ これにより、QA 作業やコンテンツ制作作業を効率化する。

・ 「ソフトリセット」に対応するものとする。
  - ➢ ゲーム中、特定のコントローラ操作により、ソフトリセットを発動し、「デバッグポータル」に戻ることができる。
  - ➢ ソフトリセットはゲームの完全な初期化を行い、何度繰り返しても安定動作を保証する。
  - ➢ 信頼あるソフトリセット機構は、QA 作業を効率化する。

・ 「簡易ソフトリセット」に対応するものとする。
  - ➢ ゲーム中、特定のコントローラ操作により、間にソフトリセットを発動し、ゲームの初期状態に戻ることができる。
    - 例えば「特定のイベントを繰り返し確認する」といった作業を効率化するために使用する。
  - ➢ 「完全な初期化」は行わず、「デバッグポータルで選択した初期状態に戻すだけ」という処理を行う。
    - 「完全な初期化」は大量のファイル読み込みで処理時間がかかるので、それを省いた「簡易ソフトリセット」は「繰り返し確認」の作業を効率化する。
    - 「リソースの自動リロード機能」と組み合わせて活用することにより、コンテンツ制作作業を効率化する。

      ・ 「リソースの自動リロード機能」は、ランタイムでリソースのデータを読み直す機能。別紙の「開発を効率化するためのファイルシステム」、「開発の効率化と安全性のためのリソース管理」、「効果的なランタイムアセット管理」にて解説。

・ 「ユニットテスト」に対応するものとする。
  - ➢ ゲーム起動時に自動的に一通りのユニットテストを実行して自動的に終了し、エラー数を返す処理に対応する。

Draft

▼ 要求仕様／要件定義

以下、本書が扱うシステムの要件を定義する。なお、要件として不確定の要求仕様も併記する。

また、要件が複雑なため、一部処理仕様レベルの内容も織り交ぜて記載する。

・ ゲーム上の全てのサブシステムの初期化と終了、および、ゲーム中でのそれらの処理順序を管理するのが、基メインループの基本的な処理要件。

● システム間の依存関係

ゲームループは、あらゆるサブシステムを使用する。

- 基本的に、サブシステム側はゲームループに依存しない。
  - ただし、ゲームループは適正な処理順序でサブシステムを実行する必要があり、サブシステム側の処理はそれを前提として成り立つ。
- サブシステム側からゲームループへの要求は、専用のメッセージキューを用い、ゲームループに直接アクセスしない。
  - 主な要求は、「フレームレート変更要求」、「ソフトリセット要求」、「ゲーム終了要求」。
  - シーンの切り替え（「戦闘シーンに入る」や「タイトルに戻る」など）は、シーンマネージャ内で完結するため、ゲームループは関与しない。
    - 「タイトルに戻る」のタイミングでゲームの状態を安定化させたい場合は、シーンマネージャで処理を完結させず、ゲームループを一旦終了させるように要求すると良い。
    - 通常、ゲームループの終了要求は、「デバッグポータル」などの別のゲームループにつなぐ時に使用する。それを利用し、もう一度ゲームにつなぎ直すことで、再初期化を行う。
    - 再初期化により、メモリ状態などが完全にクリーンになると、安定動作が保証される。
    - この方法は、動作が安定する代わりに遅くなる。
  - ゲームループは、他のサブシステムと異なり、シングルトンではない。そのため、他のサブシステムから強引に参照することもできない。必ずメッセージキューの API を通して連携する。

● 全体の処理の流れに関する要件

ゲームの全体の流れは、「main 関数開始」⇒「初期化処理」⇒「ゲームループオブジェクト処理」⇒「終了処理」⇒「main 関数終了」となる。

Draft

● ゲームループオブジェクトに関する要件

「ゲームループオブジェクト」は、「ゲームループ」の処理が記述されたオブジェクト（クラス）。

- 「ゲームループ」は単なる関数ではなく、オブジェクトとして扱う。
- ソフトリセットに応じてゲームループを切り替える。
- 主に、「通常ゲーム処理ループ」、「デバッグポータル処理ループ」、「ユニットテスト処理」といった種類のゲームループオブジェクトを扱う。

● 通常ゲーム処理ループに関する要件

「ゲームループオブジェクト」の一つとして、「通常ゲーム」の処理ループを制御する。

- ファイルマネージャ、リソースマネージャ、シーンマネージャ、入力デバイスマネージャ、サウンドマネージャ、並行ジョブスケジューラ、描画スレッドなど、全てのゲーム要素を扱う。

● デバッグポータル処理ループに関する要件

「ゲームループオブジェクト」の一つとして、「デバッグポータル」の処理ループを制御する。

- QA・制作に便利な「シーン」のショートカット起動用画面。
  - 「タイトルからゲーム開始」、「特定のレベルからゲーム開始」、「特定の戦闘からゲーム開始」、「特定のイベントシーンからゲーム開始」といった様々なショートカットメニューが並ぶ。
  - 「デバッグ用セーブデータ」を用いて、「特定のゲーム進行状態」、「特定のプレイヤーレベル」、「特定のパーティ編成」、「特定の装備状態」、「特定の所持アイテム」といった、任意のゲーム状況を作ることができる。
  - 状況作成の融通が利くように、デバッグ用セーブデータに加えて、「処理コマンド」も扱えるものとする。
    - 「プレイヤーレベル変更」、「アイテム追加」などのコマンドを用意する。
- ファイルマネージャ、リソースマネージャ、入力デバイスマネージャ、並行ジョブスケジューラ、描画スレッドなど、必要最低限のシステムを扱う。

Draft

- 「デバッグビルド」でビルドしたゲームは、ゲームを起動すると、最初にこの「デバッグポータル」画面となる。
  - 「リリースビルド」では直接ゲームが起動する。
  - この「デバッグポータルループオブジェクト」自体が削除される。
- ゲーム中にソフトリセットすると、この「デバッグポータル」画面に戻る。

● ユニットテスト処理ループに関する要件

「ゲームループオブジェクト」の一つとして、「ユニットテスト」の処理を制御する。

- あらかじめ定義された多数の「ユニットテスト」を自動実行する。
  - CI ツールを使用したデイリービルドで自動実行するなどして活用する。
  - ユニットテストに関する詳細は、別紙の「ユニットテストと継続的ビルド」を参照。
- 本来処理ループは必要なく、一回テストしたらすぐに終了する。
  - テストに必要なサブシステムの初期化のために、「ゲームループオブジェクト」化する。
- ファイルマネージャ、リソースマネージャ、シーンマネージャ、並行ジョブスケジューラなど、ユニットテストに使用するものは初期化する。
  - ゲーム中の状況に近い方が良い。
- 「ユニットテストビルド」でビルドしたゲームは、ゲームを起動すると、一通りの「ユニットテスト」を自動的に実行して自動的に終了する。
- 自動ユニットテストの結果は、プログラムの実行元に返す必要がある。
  - ユニットテストに失敗した件数を、main 関数の return 値として返す。
  - ゲームループオブジェクトが保持する「結果値」を、main 関数の return 値にするなどの処理を行う。

● ソフトリセットとゲームループ切り替えに関する要件

「ソフトリセット」の仕組みにより、「デバッグポータルからゲーム」、「ゲームからデバッグポータル」といった、ゲームループの切り替えを行う。

- 「ソフトリセット」は、ゲーム中、特定のコントローラ操作によって発動する。
  - 「多数のボタンを同時に押す」、「デバッグメニューからソフトリセットを呼び出す」など。

Draft

- 「デバッグポータル」でメニューが実行された場合なども、ゲームに切り替えるためにソフトリセット処理が実行される。
- 「ゲームループオブジェクト」を切り替えるための、最も外側の処理ループがあり、ソフトリセット時はその処理ループに立ち戻って切り替えを行う。
- 「ゲームループ要求メッセージキュー」を通じて、ソフトリセットの要求と、次に起動する「ゲームループオブジェクト」を指定する。
- 「デバッグポータル」から「ゲーム」に切り替える際、「最初のシーン」も「ゲームループ要求メッセージキュー」を通じて指定される。
  - 「最初のシーン」は、「デバッグ用セーブデータ」のロードと「処理コマンド」で構築する。
  - この「最初のシーン」の情報はキューから削除せず、「簡易ソフトリセット」（初期化を行わず、シーンを初期状態に戻すだけのソフトリセット）に備える。
- 「ゲーム」から「デバッグポータル」に復帰した際、「ゲームループ要求メッセージキュー」に残る情報に基づいて、メニュー実行時の状態を再現する。
  - メニューカーソルが指す項目や、リストのスクロール位置などを再現する。
  - 繰り返しのメニュー実行や、隣接するメニューへの切り替えを簡単に操作できるようにする。
- ソフトリセットの仕組みにより、ゲームループはほぼ完全な初期化を行う。
  - これにより、ソフトリセット後のゲームの安定動作を保証する。
  - ゲームループオブジェクトの初期化処理は、「メモリマネージャ」を含むほとんど全てのゲーム要素を初期化する。
  - メモリマネージャの初期化に当たり、メモリマネージャへの巨大なメモリ割り当てのやり直しが厳しい場合、メモリ割り当てだけはゲーム起動初回の一度だけにして、ゲームループの初期化のつど、内容をリセットする。
- 「ゲームループオブジェクト」および「ゲームループ要求メッセージキュー」は、少しでもメモリ状態を不安定にさせないように、一切ヒープメモリを使用しない。
  - ローカル変数（てスタック領域）を用いる。
  - 配置 new をうまく使うことで、ローカル変数として確保したバッファを使用して、任意のゲームループオブジェクトのインスタンスを生成することができる。
  - ゲームループ要求メッセージキューも同様にローカル変数を活用し、そのポインタだけを static 変数で公開するような「ローカルシングルトン」を活用する。
  - 「ローカルシングルトン」および「配置 new を活用したインスタンス生成方法」については、別紙の「効率化と安全性のためのロック制御」に記載している「シングルトン」の説明を参照。
- 「リリースビルド」では、「ソフトリセット」の操作を無効にする。

Draft

● 簡易ソフトリセットに関する要件

ゲーム中、簡易ソフトリセットにより、素早くシーンを初期状態に戻すことができる。

- 「簡易ソフトリセット」は、ゲーム中、特定のコントローラ操作によって発動する。
  - 「多数のボタンを同時に押す」､「デバッグメニューから簡易ソフトリセットを呼び出す」など。
- ソフトリセット時は、「ゲームループ要求メッセージキュー」に残る情報に基づいて、「最初のシーン」を再構築する。
  - 最初のシーンを構築するための「デバッグ用セーブデータ」のロードと「処理コマンド」を実行する処理に戻す。
- 「簡易ソフトリセット」を行うための、ゲームループの一つ外側の処理ループがある。
  - 「最初のシーン」を構築するための「デバッグ用セーブデータ」のロードと「処理コマンド」の実行は、この処理ループの初期化処理で行う。
- 簡易リセット処理は、サブシステムの初期化を行わないため、高速なリセットが可能。
  - 「繰り返し戦闘を行う」､「繰り返しイベントシーンを確認」といった作業を効率化する。
  - シーンの前後関係を無視した強引なリセットを行うため、適切に処理しないとメモリにゴミが残る可能性もある。
    - ・ リソースマネージャとシーンマネージャの基本的な仕組みにより、そのような問題は発生しないはずだが、特殊なリソースがあると、問題を起こす可能性もある。
- 簡易リセットの仕組みを利用し、「デバッグポータル」まで戻らずに、ゲーム中のデバッグメニューなどから、「イベントシーンの起動」や「別レベルへの直接移動」､「戦闘シーンの起動」などに対応することができる。
- 「リリースビルド」では、「簡易ソフトリセット」の操作を無効にする。

● ゲームの終了要求に関する要件

OS からのゲームの終了が要求されることがある。

- ゲームを終了させる場合は、「次のゲームループオブジェクト」を指定せずに、ソフトリセットを行うことで対応する。

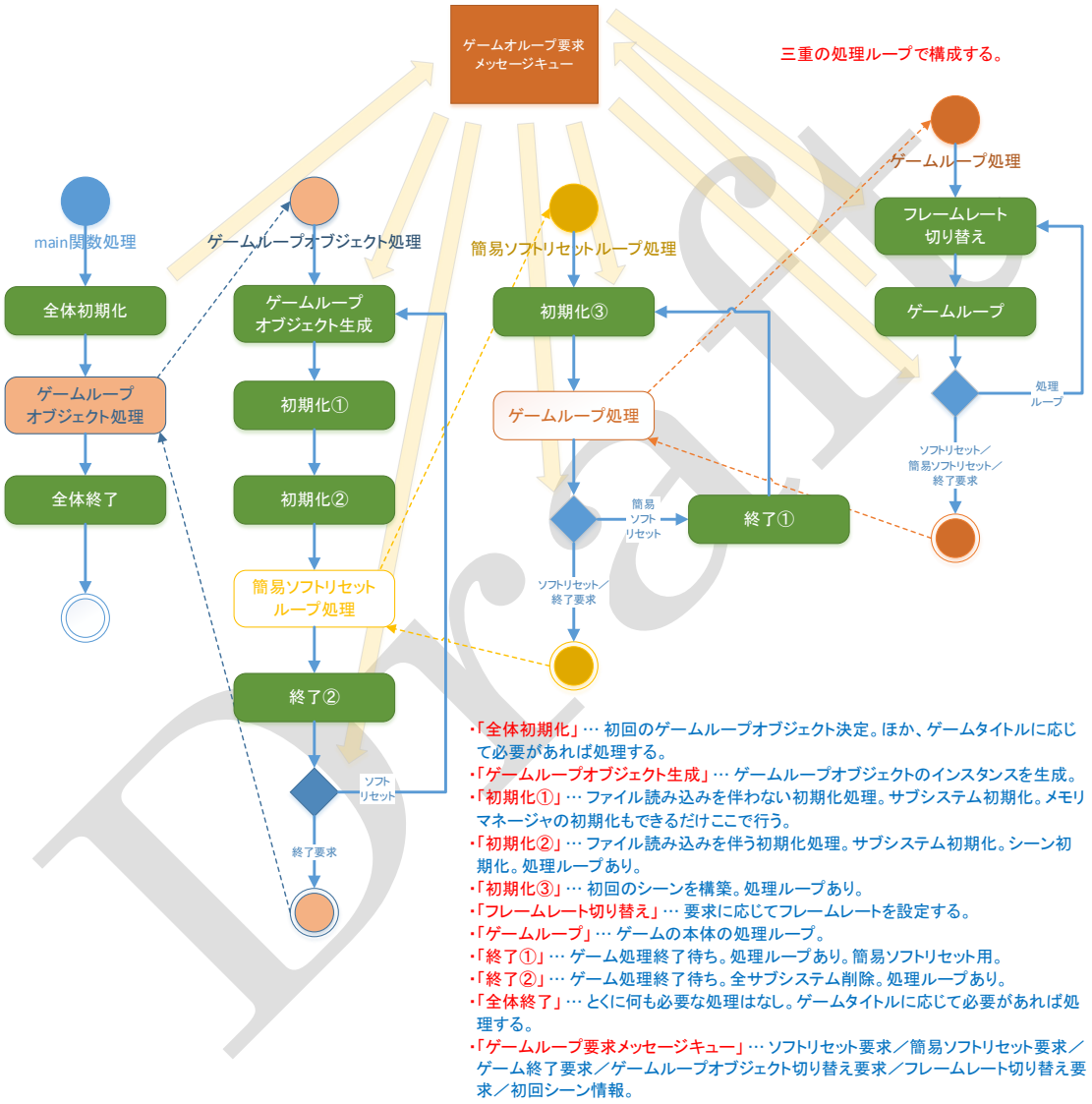
● ゲームループの処理フェーズに関する要件

以上の要件を総合し、ゲームループの処理フェーズとしてまとめる。

Draft

ゲームループは、「全体初期化」⇒［ 「ゲームループオブジェクト生成」⇒「初期化処理①」⇒「初期化処理②」⇒｛ 「初期化処理③」⇒（ 「フレームレート切り替え処理」⇒「ゲームループ処理」 ）⇒「終了処理①」 ｝⇒「終了処理②」 ］⇒「全体終了」の、10 の処理フェーズで構成する。

ゲームループの各処理フェーズのフロー：



・「全体初期化」… 初回のゲームループオブジェクト決定。ほか、ゲームタイトルに応じて必要があれば処理する。
・「ゲームループオブジェクト生成」… ゲームループオブジェクトのインスタンスを生成。
・「初期化①」… ファイル読み込みを伴わない初期化処理。サブシステム初期化。メモリマネージャの初期化もできるだけここで行う。
・「初期化②」… ファイル読み込みを伴う初期化処理。サブシステム初期化。シーン初期化。処理ループあり。
・「初期化③」… 初回のシーンを構築。処理ループあり。
・「フレームレート切り替え」… 要求に応じてフレームレートを設定する。
・「ゲームループ」… ゲームの本体の処理ループ。
・「終了①」… ゲーム処理終了待ち。処理ループあり。簡易ソフトリセット用。
・「終了②」… ゲーム処理終了待ち。全サブシステム削除。処理ループあり。
・「全体終了」… とくに何も必要な処理はなし。ゲームタイトルに応じて必要があれば処理する。
・「ゲームループ要求メッセージキュー」… ソフトリセット要求／簡易ソフトリセット要求／ゲーム終了要求／ゲームループオブジェクト切り替え要求／フレームレート切り替え要求／初回シーン情報。

- 処理フェーズ①：全体初期化
  - 基本的には、「ゲームループ要求メッセージキュー」のインスタンスを生成し、最初の要求を設定するだけ。
  - 要求には、「初回のゲームループオブジェクト」と「初回のシーン」が設定される。
  - 必要があれば、メモリマネージャに対する（巨大な）メモリ割り当てを行う。
- 処理フェーズ②：ゲームループオブジェクト生成

- 「ゲームループ要求メッセージキュー」に基づいて、要求された「ゲームループオブジェクト」のインスタンスを生成し、ゲームループを開始可能な状態にする。

➢ 処理フェーズ③：初期化①

- ローディングマークを表示できない期間の初期化処理。
- 基本的に、ファイル読み込みを伴わない初期化のみを実行。
- 処理ループを構成する必要はなく、直列的に各初期化処理を実行すればよい。
- 最低限、ローディングマークの描画が開始できる状態にする。
- 描画のために、最低限のシェーダーとローディングマークの読み込みが必要になるが、できれば、それらのファイルはプログラム（.elf ファイル）に埋め込んでおき、このフェーズでのファイルロード処理を徹底的に排除する。

  ・ 一例として、objcopy コマンドで必要なファイルをオブジェクトファイル化（.o ファイル）すると、実行ファイル（.elf ファイル）にリンクすることができる。
  ・ リンクされたオブジェクトファイルは、C 言語のプログラムから extern 宣言で、バイナリデータ（char 型の配列）として参照できる。
  ・ メモリに常駐することになるので、必要最小限のデータに留める。

【参考】objcopy コマンドを使用した外部ファイルの埋め込み例：

【text.txt】

```
Test Text for ObjCopy!
```

【main.cpp】

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern char _binary_test_txt_start[];//test.txt の先頭の位置
extern char _binary_test_txt_end[];//test.txt の終端+1 の位置
extern char _binary_test_txt_size[];//test.txt のサイズ
//_binary_ + (ファイル名) + _ + (拡張子) + _start / _end / _size というシンボルが objcopy で作られる
//Cygwin の場合は注意！
//GCC でコンパイルしても、Visual C++ と同じく、コンパイル時に
//自動的にシンボル名の先頭にアンダースコアを付加する。
//そのため、C/C++のコード上では先頭のアンダースコアを付けない。

int main(const int argc, const char* argv[])
{
	const char *txt_top = _binary_test_txt_start;//test.txt の先頭の位置を取得
	const char *txt_end = _binary_test_txt_end;//test.txt の終端+1 の位置を取得
	const size_t txt_size = reinterpret_cast<size_t>(_binary_test_txt_size);//test.txt のサイズを取得

	//test.txt の内容をコピー
	char* txt = new char[txt_size + 1];
	strncpy(txt, txt_top, txt_size);
	txt[txt_size] = '¥0';

	//text.txt の内容を表示
	printf("test.txt = ¥"%s¥"¥n", txt);

	//バッファを破棄
	delete txt;
```

Draft

```
    return EXIT_SUCCESS;
}
```

【ビルド①：objcoopy コマンドで text.txt からオブジェクトファイル test_txt.o を生成】

```
$ objcopy -I binary -O elf32-i386 -B i386 test.txt test_txt.o      ← -O, -B は適切なオプションを指定
```

【ビルド②：コンパイル＆リンク】

```
$ g++ -c main.cpp                                                  ← main.cpp コンパイル
$ g++ main.o test_txt.o -o main                                    ← リンク
```

【実行】

```
$ ./main
```

【実行結果】

```
test.txt = "Test Text for ObjCopy!"        ← text.txt の内容が実行ファイルに組み込まれて
                                             表示されたことがわかる
```

【シンボルの確認：objdump コマンドで test_txt.o の シンボルを確認】

```
$ objdump -t -s -h test_txt.o
```

【シンボルの確認結果：Cygwin + GCC 4.8.2 + objdump 2.23】

```
test_txt.o:     ファイル形式 elf32-i386

セクション:
索引名          サイズ      VMA       LMA       File off  Algn
  0 .data         00000016  00000000  00000000  00000034  2**0
                  CONTENTS, ALLOC, LOAD, DATA
SYMBOL TABLE:
00000000 l    d  .data  00000000 .data
00000000 g       .data  00000000 _binary_test_txt_start
00000016 g       .data  00000000 _binary_test_txt_end
00000016 g       *ABS*  00000000 _binary_test_txt_size

セクション .data の内容:
 0000 54657374 20546578 7420666f 72204f62  Test Text for Ob
 0010 6a436f70 7921                        jCopy!
```

- このフェーズで初期化すべきものは、「メモリマネージャ」、「ファイルマネージャ」、「リソースマネージャ」、「並行ジョブスケジューラサービス」、「描画スレッド」、および、最低限のシェーダーとローディングマーク。
- 「メモリマネージャ」については、ソフトリセットの動作を安定させるために、メモリマネージャに対する（巨大な）メモリ割り当てのみは最初の「全体初期化」でやっておき、ここではメモリマネージャの状態のみをリセットする。

  ・ メモリリークがあっても強引にリセットし、完全に初期状態にする。

➢ 処理フェーズ④：初期化②

- ローディングマークを表示可能な初期化処理。
- 処理ループを構成し、ローディングマークを表示しながら初期化処理を行う。
- ローディングマークを円滑に表示するために、各初期化処理はジョブとして投入し、並行処理する。
- このフェーズで初期化すべきものは、「入力デバイスマネージャ」、「サウンドマネージャ」、「シ

Draft

ーンマネージャ」、および、その他のゲーム要素。

- 初期化完了後の「最初のシーン」を「タイトルシーン」に限定するのは NG。「デバッグポータル」の活用として、直接「移動シーン」や「戦闘シーン」、「イベントシーン」を起動する場合がある。

➢ 処理フェーズ⑤：初期化③

- ゲームループに入った時の「最初のシーン」を準備する。
- ゲームループ中に「簡易ソフトリセット」が実行された場合、完全な終了処理と初期化処理をやり直さず、この処理フェーズに戻される。これにより、「繰り返し確認」の作業を効率化する。
- 通常は「タイトルシーン」の起動準備として、シーンマネージャのタイトルシーンを構築する。
- 「デバッグポータル」により、「特定のレベル」や「特定のイベント」が選ばれた場合、「移動シーン」や「イベントシーン」を構築する。
- 基本的には、シーンマネージャに対して「シーン切り替え要求」を出してゲームループに切り替えれば、シーンマネージャはシーンの構築を始める。
- 一部のシーンは、元になるシーンの上に成り立つものがある。例えば、特定の移動シーンを前提としたイベントシーンなどがある。この対応として、この処理フェーズは処理ループを構成し、ベースになるシーンの構築を済ませることができる。
- デバッグポータルでは、「特定のパーティ編成」や「特定のレベル」「特定の装備」「特定の所持アイテム」などのシチュエーションが指定される場合がある。これは、デバッグ用セーブデータのロードに特定のコマンド（所持アイテム追加、レベル変更など）を組み合わせて実現する。そのセーブデータ読み込みとコマンド実行もこのフェーズで行う。

➢ 処理フェーズ⑥：フレームレート切り替え処理

- フレームレート切り替え要求に基づいて、30fps → 60fps などのフレームレート切り替えを行う。
- OS の画面が呼び出された際のポーズ中や、ムービー中などに切り替えを行う。

➢ 処理フェーズ⑦：ゲームループ処理

- ゲーム本体の処理ループ。
- 処理フローなどの詳しい要件と説明は別途記述。

➢ 処理フェーズ⑧：終了処理①

- 「簡易ソフトリセット」専用の終了処理。この処理のあと、「処理フェーズ③：初期化③」につながる。
- 簡易ソフトリセット時以外は呼び出されない。
- 通常はとくに何もしなくて良いが、「処理フェーズ③：初期化③」に処理を戻す上で障害となる要素があるなら、このフェーズで調整しておく。

Draft

- 処理フェーズ⑨：終了処理②
  - ソフトリセットのための処理。
  - 「処理フェーズ②：初期化②」と「処理フェーズ①：初期化①」で初期化したサブシステム、スレッド・サービスを終了する。
  - 「メモリマネージャ」については、ソフトリセットの動作を安定させるために、「全体初期化」でメモリマネージャに割り当てられた（巨大な）メモリまでは破棄しない。
  - 「メモリマネージャ」は、終了処理の一環として、割り当て中のメモリをすべてダンプ出力する。
    - 終了処理を通しても解放されなかったメモリは、リークしたメモリとみなす。
    - 通常、終了処理の最後には、メモリマネージャのメモリは全て解放されていなければならない。
    - メモリマネージャは、デバッグモードでは、メモリ割り当てを行ったスレッドと処理の名前、割り当てを行ったゲーム時間を記録している。
  - 処理完了後は、新たな処理ループの「処理フェーズ①：初期化①」に戻る。
- 処理フェーズ⑩：全体終了処理
  - ほぼ何もすることはない。
  - 必要があれば、メモリマネージャに割り当てていた（巨大な）メモリを開放する。

● ゲームループ要求メッセージキューに関する要件

各処理からゲームループに対する要求は、直接ゲームループを操作するのではなく、「ゲームループ要求メッセージキュー」を通す。

- 扱われる要求は、「フレームレート変更要求」、「ゲームループ切り替え要求（ソフトリセット要求）」、「簡易ソフトリセット要求」、「ゲーム終了要求」。
- シングルトンにより、どの処理からでも要求を出すことができる。
- プログラム起動時に、「ゲームを起動するか」、「デバッグポータルを起動するか」といった決定も、この要求を設定することによって処理する。
  - 「全体初期化処理」のタイミングでインスタンスを生成するとともに、要求をセットする。
  - ゲームループ間に跨がって共有する唯一の情報となるため、他のサブシステムのようにゲームループの最初に初期化したりはしない。
- ゲーム起動時の最初のシーンを「タイトルシーンにするか」、「移動シーンにするか」といった決定も、この要求を設定することによって処理する。
  - 直接ゲームを起動する場合、「全体初期化処理」のタイミングで「タイトルシーン」の要求を

Draft

セットする。

- 要求の内容に基づいて、処理ループの開始状態が決定する。
  - 外側の処理ループ「ゲームループオブジェクト生成処理」では、要求されたゲームループに応じたゲームループオブジェクトを生成する。
  - 中間の処理ループ「初期化③処理」では、要求に基づいてゲームの最初のシーンを構築する。
  - 内側の処理ループ「フレームレート切り替え処理」では、要求に基づいてフレームレートを変更する。

- 要求の内容に基づいて、処理ループの終了が決定する。
  - この時、複数の要求があった場合の選別を行う。
  - 「簡易ソフトリセット要求」があった場合は、内側の処理ループが終了し、中間の処理ループにより、「初期化③処理」につながる。
  - 「ソフトリセット要求」があった場合は、中間の処理ループが終了し、外側の処理ループにより、「ゲームループオブジェクト生成処理」につながる。
  - 「ゲーム終了要求」があった場合は、外側の処理ループが終了し、main 関数が終了する。

- 「キュー」と称しつつも、キュー以外の情報も扱う。
  - 「デバッグポータル」の選択メニューとメニューのスクロール状態を扱う。
    - 「ゲーム」から「デバッグポータル」にソフトリセットで戻った時に、メニュー選択時の状態を再現するのに必要な情報。
    - ゲーム中も消さずに保持する。
  - ゲームの「最初のシーン」の情報を扱う。
    - 「デバッグポータル」から選択されたメニューに基づくシーンの再現に必要な情報。
    - シーンの指定（「タイトルシーン」、「移動シーン」、「イベントシーン」など）、ロードすべきデバッグ用セーブデータ、処理コマンド（プレイヤーレベル変更や所持アイテム追加など）を保持する。
    - 「簡易ソフトリセット」時に、再度この情報を必要とするため、ゲーム中も消さずに保持する。
    - ゲーム中のデバッグメニューから、任意のイベントシーンの起動や戦闘シーンの起動を行う場合は、この「最初のシーン」の情報を書き換えて簡易ソフトリセットを実行すれば良い。

● フレームレートと処理時間管理に関する要件

ゲームループ処理は、フレームレートと処理時間を管理する。

Draft

- フレームレートと処理時間の情報はシングルトンでまとめられ、どこからでも参照できる。
  - 「処理時間オブジェクト」としてまとめる。

- ゲーム中の処理は、全般的に、「フレームベース」ではなく「時間ベース」で扱うことを基本とする。
  - これにより、フレームレートが変動しても、アニメ―ションのレートが変わらない。
  - 「フレームベース」の処理が必要な個所は極力局所的な処理とし、都度時間をフレーム数に換算する。

- 「処理時間」は、ゲーム全体の経過時間を表す。
  - ゲームループ処理開始時からカウントを始める。(「初期化処理③」終了後のタイミング)
  - dobule 型で管理。単位は「秒」。

- フレーム毎の「経過時間」も保持する。
  - 前のフレームの処理にかかった時間が格納される。
  - float 型で管理され、単位は「秒」。
  - 通常は 30fps, 60fps で固定された値（1/30sec, 1/60sec）が格納されるが、処理落ちするとそれより長い時間になることがある。
  - この「経過時間」は、「アップデート処理」でシーンオブジェクトの時間を進めるために使用されるが、そのままは使用されない。
    - 「シーンオブジェクト」ごとに「処理時間レート」があり、「経過時間」×「処理時間レート」がその「シーンオブジェクトの経過時間」として扱われる。
    - この「シーンオブジェクトの経過時間」はシーンマネージャが計算し、「アップデート処理」呼び出し時に引数で受け渡す。
    - これにより、オブジェクトごとのスローモーション処理などを行う。

- デバッグ用に、全体の「処理時間レート」も設定可能とする。
  - シーンオブジェクトごとの処理時間レートではなく、全体のレート。
  - 例えば、2.0 を設定すると、毎フレームの経過時間が 2.0 倍となり、全てのオブジェクトが倍速で動作するようになる。1.0 未満にすればその逆。
  - シーンオブジェクトごとの処理時間の掛け合わせは通常どおり行われるので、各オブジェクトの動作時間の比率に変わりはない。

- デバッグ用に、「経過時間」を固定化する設定も可能とする。
  - ムービー用に毎フレームキャプチャーするような場合に使用する。
  - どんなに処理落ちしても、フレームレート分の処理時間の進行に固定する。

Draft

- フレーム毎の「経過時間」と共に、フレームごとの「経過フレーム数」も保持する。
  - 基本的に、参考用の情報。処理に直接用いるようなことはしない。
  - 通常は 1.0 で固定されるが、処理落ちしたり、フレームレートが変更されたりすると、異なる値になる。
  - 経過フレーム数は、「基準フレームレート」と現在のフレームレートに基づいてカウントされる。

- ゲーム中に「フレームレート」を変更することは可能だが、別途絶対に変動しない「基準フレームレート」を扱う。
  - 基準フレームレートは、フレームレートが途中で変更されても変わらない固定値。
  - 基準フレームレートも処理時間オブジェクトから参照できる。
  - フレームレートが途中で変更されても、常に基準フレームレートに基づいてフレーム数（経過フレーム数）をカウントする。
  - 例えば、基準フレームレートが 30fps なら、フレームレートが 30fsp の時は 1.0 でカウントされるが、60fps の時は 0.5 でカウントされる。
  - また、例えば、基準フレームレートが 60fps なら、フレームレートが 60fsp の時は 1.0 でカウントされるが、30fps の時は 2.0 でカウントされる。

- 初期のフレームレートは、「最初のシーン」に設定されている情報に基づき、「初期化処理③」のタイミングでセットする。
  - 「シーン」を扱わないゲームループでは、タイトル規定のフレームレートを設定する。

- フレームレートは、描画スレッドの処理途中で変更されるため、描画スレッドは（そのフレームの）処理開始時のフレームレートをフリップ完了まで維持する。
  - あまりに処理が早く終わりすぎた場合は、処理時間に基づいて一回フリップをスキップする必要もある。（30fps 動作時）
  - このフリップタイミングの調整は、描画スレッド開始時からの処理時間ではなく、「前回のフリップからの経過時間」で判断する。

- 「CPU 使用率」などの「パフォーマンス計測」の記録にも用いる。
  - 後述する「コールポイント」処理と組み合わせ、任意の処理のパフォーマンス（処理時間）も記録し、集計する。

● ゲームループの処理フローに関する要件

ゲームループは、マルチスレッドに最適化した処理順序で処理する。

- 「並行ジョブスケジューラ」を活用した fork-join モデルでマルチスレッド処理を最適化する。

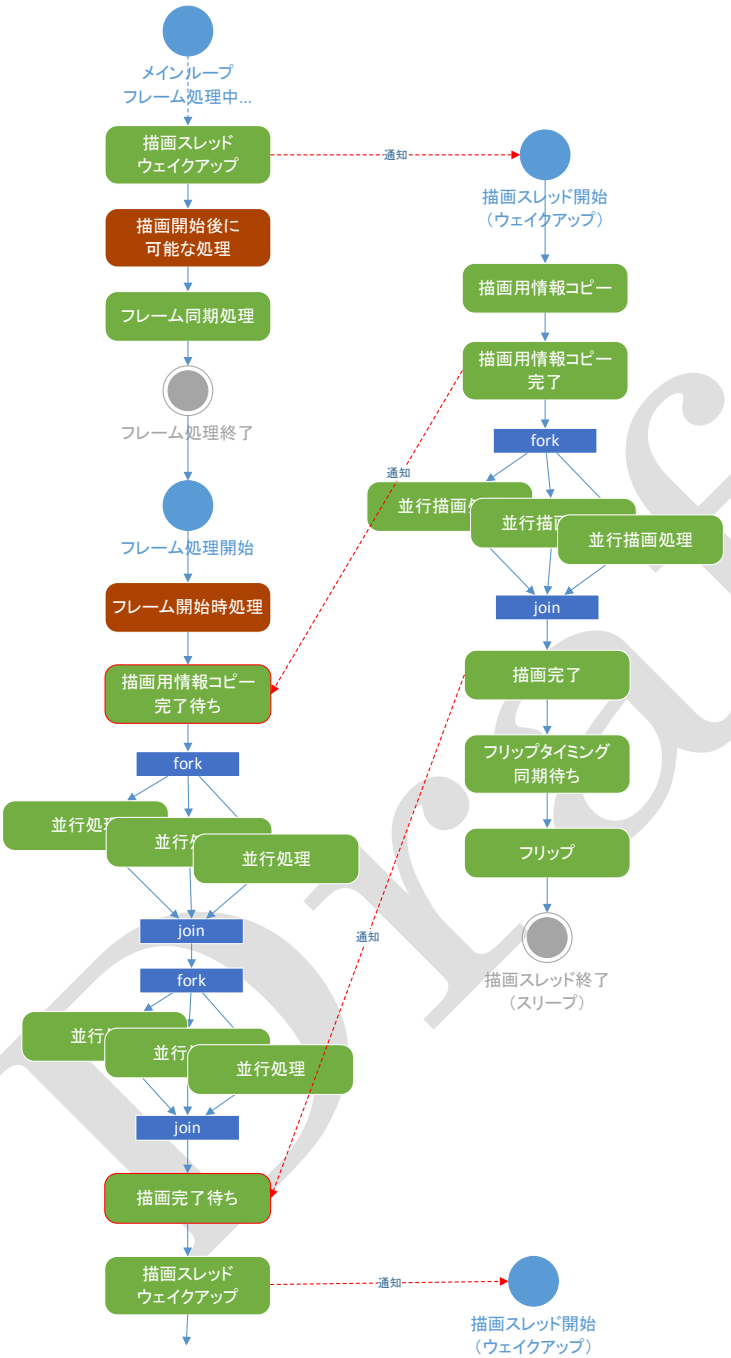
Draft

- ゲームループのメインループ（メインスレッド側の処理）と描画スレッドとの処理リレーを考慮し、可能な限り処理をオーバーラップさせる。
  - 描画スレッド処理開始後に回せるメインループの処理を極力増やす。
  - メインスレッドは、メインループが少しでも早く次の処理に入れるように、描画ための情報コピー処理を最優先で行う。

- メインループは、描画スレッドの情報コピー待ち、描画完了待ちをきちんと行い、誤ったタイミングでの情報更新を絶対に行わないようにする。
  - どんなに確実に間に合う処理であっても、処理が遅れたりタイミングがずれたりする可能性はゼロではない。

Draft

処理フローのイメージ：



● サブシステムと常駐スレッドに関する要件

ゲームループは多数のサブシステム、および、常駐スレッドと連携する。

➢ ゲームループは、主に下記のサブシステムと連携する。

- メモリマネージャ

- ファイルマネージャ
- リソースマネージャ
- シーンマネージャ
- 入力デバイスマネージャ
- サウンドマネージャ
- デバッグ制御システム
- 他、物理エンジンなど

➢ ゲームループは、主に下記の常駐スレッドと連携する。

- 描画スレッド
- 並行ジョブスケジューラ
- 長期並行ジョブスケジューラ

● 描画スレッドに関する要件

描画スレッドは、常駐スレッドとして処理する。

➢ メインループと並行処理するために、描画処理はスレッド化する。

➢ 描画スレッドの優先度は、メインスレッドと同じに設定する。

➢ 描画スレッドを常駐スレッドにする理由は下記のとおり。

- 毎フレーム必ず必要なスレッドなので、スレッド生成のコスト、リソース確保をわずかにでも抑え、安定動作させる。
- 複数の描画スレッドが必要になることはない。

  ・ 活用できなくもないが、滅多に必要な状況が発生しないばかりか、このために所要メモリ量が増大し、処理も複雑化するので、割に合わない。

- 描画スレッドの処理状況によってはメインスレッドをブロックする必要もあるため、常駐スレッドで確かな同期を行う。

  ・ スレッド生成時の隙間で同期のタイミングを逃すなどの問題を生じない。

● 並行ジョブスケジューラに関する要件

並行ジョブスケジューラは、並行処理可能なアップデート処理や描画処理が投入される汎用スケジューラ。

➢ 「コア数 － 1～2」程度の並行処理を上限とし、対象ハードウェアに最適なパフォーマンス調整を行う。

Draft

- 並行処理数を超える多数の処理をジョブとしてキューイングし、適切な優先度判定と依存関係判定のもとスケジューリングし、順次ジョブを実行する。
  - ジョブ用のスレッドは常にプールしており、ジョブを実行するたびに、空いているスレッドを再利用する。
- 並行ジョブスケジューラサービスのスレッドの優先度は、メインスレッド・描画スレッドよりも高めに設定する。
  - 並行ジョブスケジューラサービスは、ジョブの実行状態を監視して、溜まっているジョブを次々投入するスレッド。
- 並行ジョブスケジューラによって処理されるジョブのスレッド優先度は、メインスレッド・描画スレッドと同じに設定する。
- 並行ジョブスケジューラに関する詳細は、別紙の「「サービス」によるマルチスレッドの効率化」を参照。

● 長期並行ジョブスケジューラに関する要件

パフォーマンス目的ではなく、AI の経路探索のように、数フレームかけて計算したい処理のためのジョブスケジューラも用意する。

- 長期並行ジョブスケジューラの並行ジョブ数（スレッド数）は、ゲームの要件に合わせて設定する。
- 並行ジョブスケジューラと異なり、ジョブのプールはせず、実行できない時は予約時に失敗させるか、優先度の低いジョブを中断する。
  - この状況がほとんど発生しないように並行ジョブ数を調整する。
- 長期並行ジョブスケジューラサービスのスレッドの優先度は、メインスレッド・描画スレッドよりも高めに設定する。
  - プールするジョブはないが、基本的に並行ジョブスケジューラと同じ仕組みで動作する。
- 長期並行ジョブスケジューラによって処理されるジョブのスレッド優先度は、メインスレッド・描画スレッドより低く設定する。
- 以上の差を除けば、ジョブスケジューラの仕組み・動作は、通常の並行ジョブスケジューラと同じ。

● 常駐スレッドに関する要件

常駐スレッドの実行が開始されるタイミングは、「条件変数」などの「モニター」の

Draft

仕組みを利用してウェイクアップする。

- 待ち受け中はできる限りスリープし、ビジーウェイトを用いない。
- ウェイクアップ通知のタイミングに瞬時に反応できるように、ポーリングとスリープを交互に行うような処理もできるだけ避ける。
- 条件変数を使用した最適な待ち受けとウェイクアップ手法は、別紙の「「サービス」によるマルチスレッドの効率化」を参照。

● 入力デバイス制御に関する要件

入力デバイスの状態は、ゲームループの最初に入力デバイスマネージャから取得し、同一フレーム中はその情報に基づいて処理する。

- 実際のところ、ゲームループ側に入力デバイスの情報をコピーするのではなく、入力デバイスマネージャに、そのフレームの状態を確定するように通知する。
- 実際の処理は、ゲームループ中の各処理で、「論理デバイス」を通じて入力デバイスの情報を参照する。
- 入力デバイスマネージャ側でそのフレームの状態が確定しているので、フレームの処理中にデバイスの状態が変わることはない。
- 入力デバイスマネージャに関する詳細は、別紙の「反応性と安全性を考慮した入力デバイス管理」を参照。

● パフォーマンス計測（プロファイラ）に関する要件

前述の「処理時間オブジェクト」は、パフォーマンス計測の集計にも対応し、柔軟なプロファイラとして活用できるものとする。

- メインループの先頭から、最後の「フレーム同期処理」（処理時間を調整する処理）の直前までの処理時間を、フレームレートの処理時間（1/30sec or 1/60sec）で割った割合を CPU 使用率とする。
- 描画スレッドも同様に、描画スレッドの開始から、最後の「フリップ同期タイミング待ち」の直前までの処理時間を、フレームレートの処理時間（1/30sec or 1/60sec）で割った割合を CPU 使用率とする。

Draft

- 本当のプロファイラではないので、厳密なスレッドの稼働時間ではなく、スリープ時間を含めた計測となる。
  - スリープも含めて、時間がかかっている処理が見つかれば良いので、CPU 負荷よりも処理時間が重要。
  - コアごとの処理負荷なども特に計測しない。

- CPU 使用率は、そのフレームの値を記録するだけではなく、「1 秒間」や「5 秒間」といったスパンで集計し、問題を発見し易くする。
  - 集計時間はデバッグメニューなどから任意に変更可能とする。
  - 集計時間内での、最大や平均を集計する。
  - 「95%」などのしきい値を設け、「しきい値を超えたフレーム数の割合」「しきい値を超えた時の平均」などを集計し、少しでも問題を分析し易いような集計の仕組みを実装する。
  - パフォーマンスの集計方法については、別紙の「本当にちょっとしたプログラミング Tips」にも詳しく説明している。

- 処理時間オブジェクトに対して、何らかのキーと処理時間を送れば、メインループと描画スレッドに限らず記録し、集計する。
  - 例えば、メインループは「キー = 0」、描画スレッドは「キー = 1」といった設定で記録を行う。
  - これ以外にも、例えば「funcA 関数の処理時間」といった任意の計測を記録し、集計させることができる。
  - これを利用し、重い処理上位 10 件などを画面に表示する仕組みを設ける。
  - 「コールポイント」という、一連の処理ブロックを宣言するデバッグ機能の仕組みを利用して、任意の処理時間計測を強化する。
    - 「コールポイント」は、処理ブロックの開始と終了を自動的に記録する仕組みである。
    - 主にデバッグログに活用する仕組みである。
    - この処理を利用し、処理ブロックの開始から終了の処理時間を計測し、処理時間オブジェクトへの記録を自動化する。
    - コールポイントは、ブロック宣言時に「処理名」を与える。この処理名とスレッド名を CPU 使用率と共に表示するようにすると、分かり易くなる。
    - コールポイントはネストも扱える。子の処理時間は必ず親の処理時間に反映されるので、親子関係も明確に分かった方が分かり易くなる。
    - コールポイントの処理時間を記録するに当たっては、例えば、__FILE__ マクロと __LINE__ マクロを組み合わせて作成した文字列のポインタを利用する。ポインタがそのままキー（ユニークな ID）として利用可能な値となる。
    - コールポイントに関する詳細は、別紙の「効果的なデバッグログとアサーション」を参照。

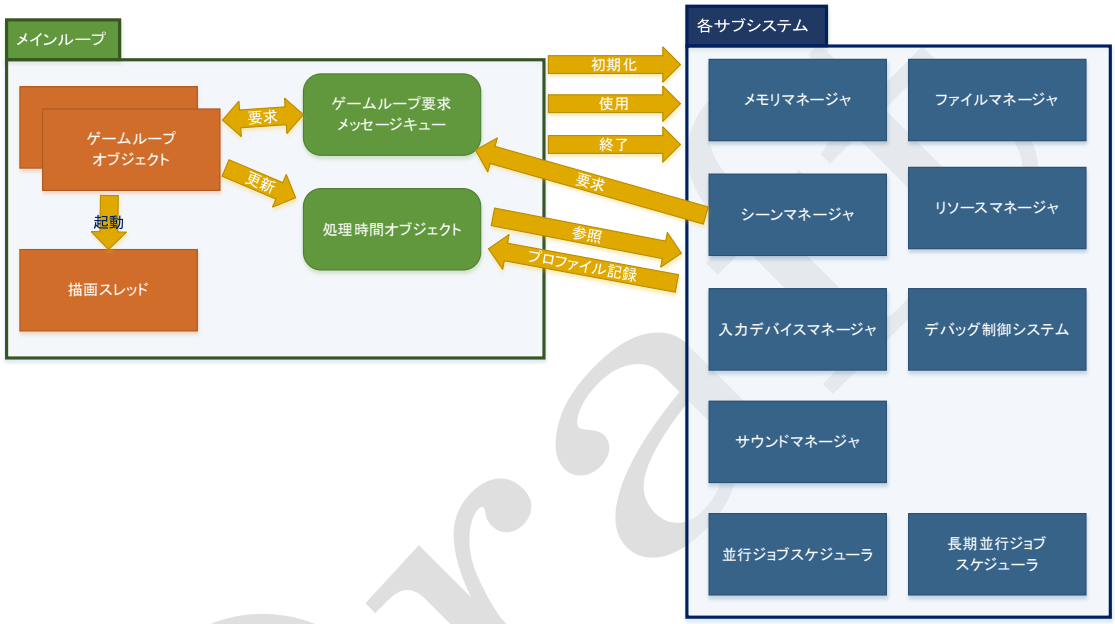
Draft

■ 仕様概要

▼ システム構成図

要件に基づくシステム構成図を示す。

ゲームループのシステム構成図：



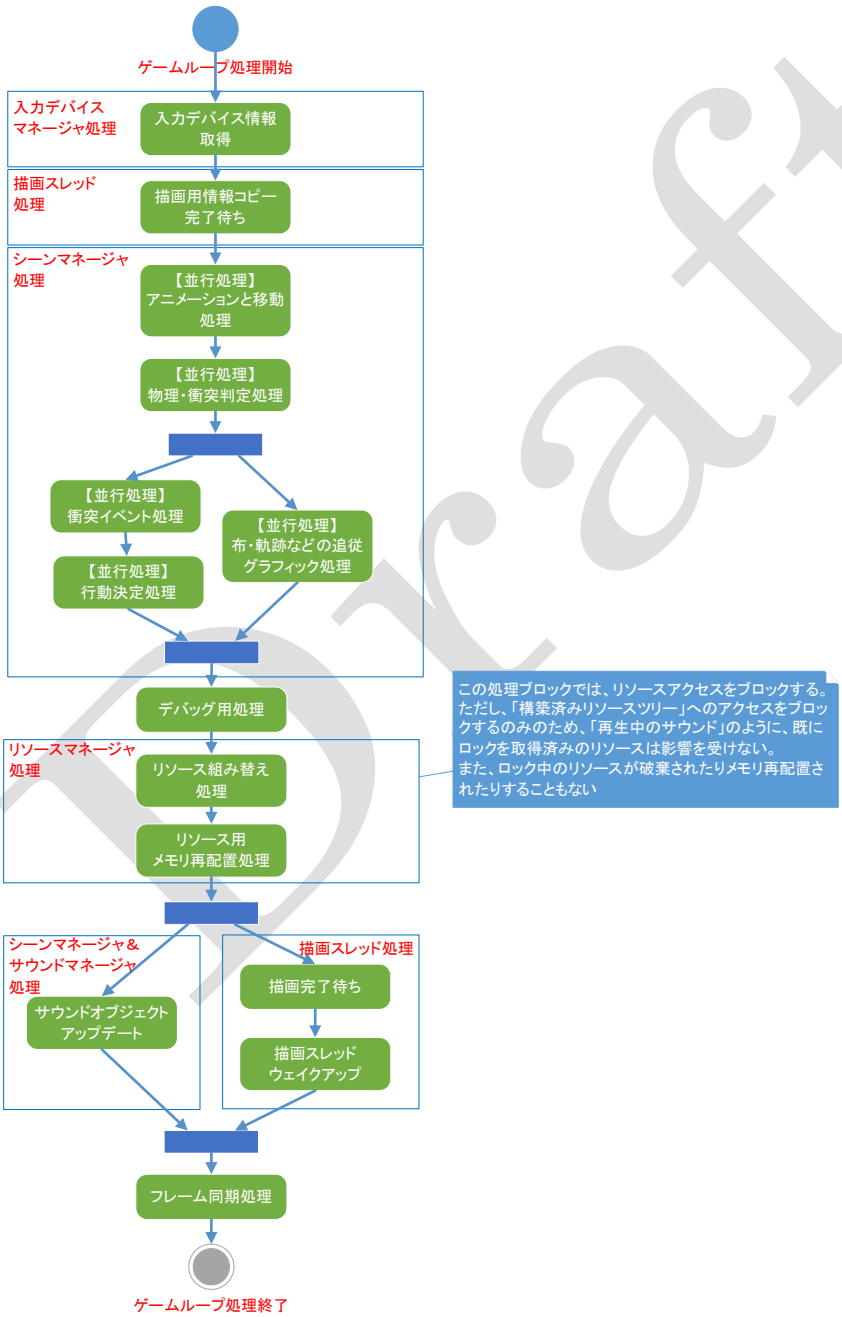
Draft

■ 処理仕様

▼ 処理フロー

マルチスレッドに最適化したゲームループ内の処理フローを示す。

ゲームループ内の処理フローのイメージ：

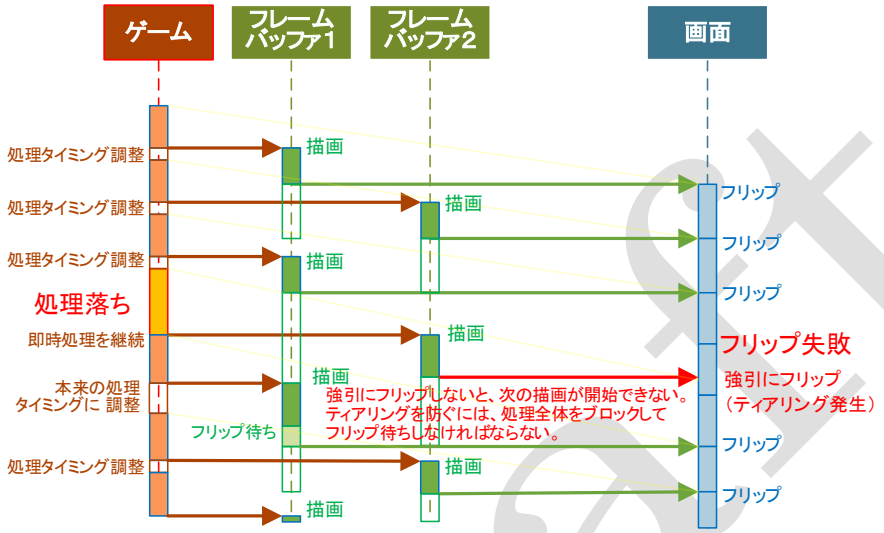


Draft

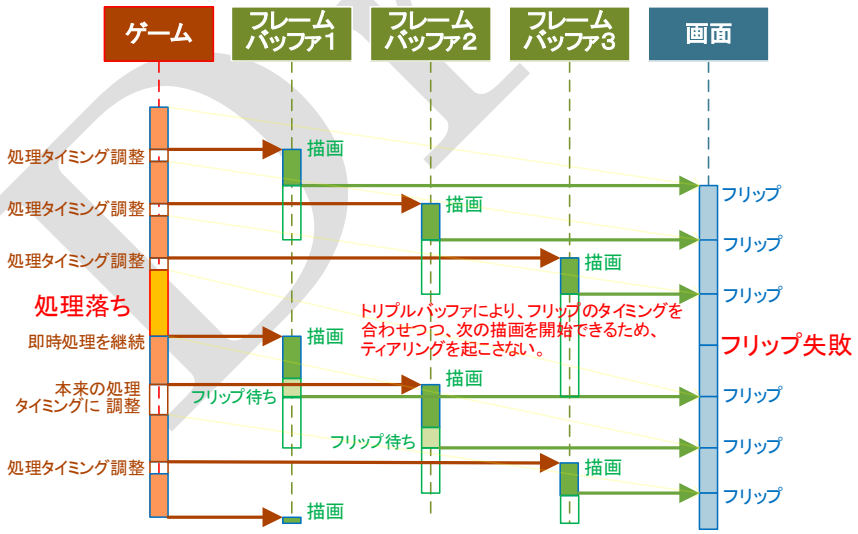
▼ 処理落ち対策：トリプルバッファ

処理落ちの発生が少しでも描画に影響しないように、できる限り、トリプルバッファを採用する。

処理落ち時の描画処理（ダブルバッファ時）：



処理落ち時の描画処理（トリプルバッファ時）：



■■以上■■

■ 索引

索引項目が見つかりません。

Draft

マルチスレッドによるゲームループ管理

以　上

Draft