ゲームシステムのアーキテクチャと開発環境

– 開発効率と処理効率のために –

2014 年 2 月 18 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 2 月 18 日	板垣 衛	（初稿）

■ 目次

■ 概略

本書は、一連のドキュメントのインデックスである。

筆者自身の経験・得意分野・関心に基づいて、幾つかの面においてのゲーム開発の方法論を多数のドキュメントにまとめている。

いずれも、過去の実務上の取り組みを文書化するようなものではなく、新たなシステムや方法論としてまとめているものである。とくにマルチスレッドによる並行処理を最適化した開発を強く意識している。

多数のドキュメントは、下記の内容に分類される。

・ ゲームシステム
・ 開発環境／プロジェクト管理
・ プログラミング

本書では、これらのドキュメントで扱っている事項の要約を示す。

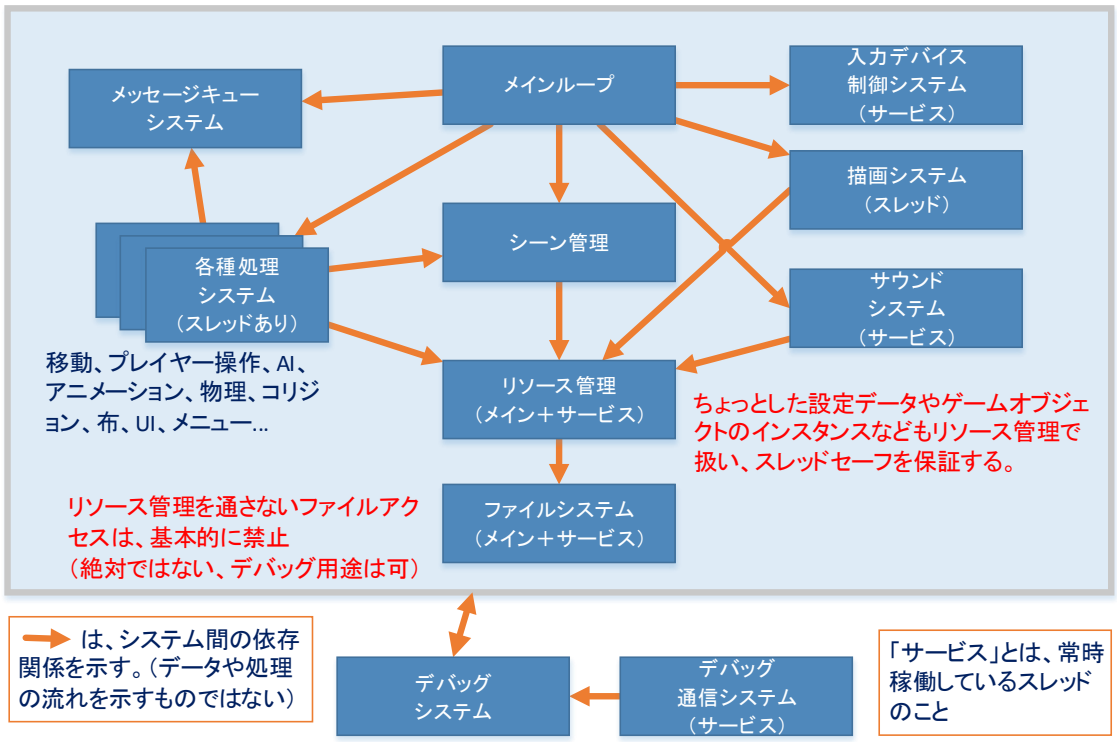
なお、各ドキュメントは基本的に「草案レベル」である。
実証に基づいた内容のものもあるが、確実性が得られていない要求仕様レベルの内容も混在している。多くは、ゲームシステムとその開発環境を構築するための「提案書」である。

また、本書中に記述した一般的な技術用語や一般的なツール名などを文末の索引にまとめ、少しでも関心事を参照し易いようにしている。

■ 基本ゲームシステムのアーキテクチャ

幾つかのドキュメントでは、基本ゲームシステムの設計を行っている。そのアーキテクチャの全体像を示す。

▼ 各種システムの関係



■ 基本ゲームシステムの設計思想・概略

基本ゲームシステムの設計思想・概略を示す。

▼ 【安全性】「役割」をベースにした分かり易いシステムの切り分け

個々の開発者が独自の判断で危うい処理を組み込むようなことがないように、「役割」の明確なシステムを用意する。

● 役割①：リソースの生成／削除の役割はリソースマネージャ

いかなる都合があっても、直接リソースの生成／削除を行ってはいけない。そのような操作は必ずリソースマネージャに依頼する。

リソースマネージャを通すことで、リソースの多重構築や不正なタイミングでのリソース削除などを防ぐことはもとより、マルチスレッドに最適化されたスレッドセーフなリソースアクセス手段を提供し、メモリ再配置も安全に行えるものとする。

そのため、グラフィックデータはもちろんのこと、サウンドデータや各種設定データなども含めて、およそメモリ管理される大部分の要素をリソースマネージャが管理する。

また、リソースマネージャはファイルマネージャやメモリマネージャと連携し、各処理系がほとんど意識することなく、高度なリソース管理を行う。例えば、先行読み込み（キャッシュ）されたリソースの瞬間的な構築や、別シーン（ムービー再生や割り込み可能なミニゲームなど）のためのメモリブロックの解放とその後の再構築といった処理を行う。

● 役割②：座標を持ったオブジェクトを管理する役割はシーンマネージャ

いかなる都合があっても、シーンマネージャに登録せずに座標を持ったオブジェクトを扱ってはいけない。オブジェクトには必ず親子関係があるものと意識し、シーンマネージャに登録して扱う。

マップも、キャラも、エフェクトも、頭上アイコンも、SE も、見えないファンクションボックス（イベント発生ポイント）も、定点カメラも、全てシーンマネージャに登録する。「シーングラフ」が全てのオブジェクトの親子関係に責任を持つ。

手に持った剣や炎エフェクトを体にまとまったモンスター、プロペラ音を出し続けるヘリコプター、動くエレベーター上に配置されたファンクションボックスなどの親子関係の解消は、全て共通処理で行う。

これにより、各処理系で同じような処理を実装することや、一部の処理系が正しく親子関係に基づいて処理できていないった問題を防ぐ。

なお、ここで言う「シーングラフ」とは、描画のためのものではなく、処理のためのもの。ファンクションボックスや SE のような非描画オブジェクトも含めて、シーン上に配置されるすべての要素をシーングラフで一元管理し、要素間の関係を明確にする。

● 役割③：シーンの切り替えの役割はメッセージキューとシーンマネージャ

いかなる都合があっても、各処理系の判断で直接シーンの切り替えを行ってはいけない。シーン切り替えの要求はメッセージキューを通し、シーンマネージャの判断のもとに切り替えを行う。

例えば、同一フレームで「メインメニューの呼び出し」「ゲームオーバー」「イベント発生点への到達」が同時に起こったとしても、それぞれの処理系で実行可能な状況かどうかは判断せず、メッセージキューに要求を出すだけとする。

メッセージキューに登録された要求は、フレームの最終処理で、シーンマネージャがまとめてチェックし、状況的に可能で、かつ、もっとも優先度の高いものを採択して実行する。

これにより、各処理系が多数の競合要素にアクセスして状況判断するようなことを無くする。シンプルに要求だけ出せば、ゲームの基本システムが適切な状況判断をする。複雑な依存関係と追跡しにくい処理分散を防ぐ。

▼ 【生産性】統一的なシステムで開発・制作効率の向上

統一的なシステムにより、プログラム開発者には迷うことの少ない分かり易い開発手段を提供し、コンテンツ制作者にはトライアンドエラーのサイクルを短縮するシステムを提供する。

● 統一システム①：「役割」をベースにした独立性の高い開発

基本システムの明確な役割分担により、システムの依存関係を減らし、独立性の高い開発を可能とする。

リソースマネージャ、シーンマネージャによって提供される安全性の高いシステムにより、各処理系の責任範囲を局所化し、開発の負担を軽減する。

また、メッセージキューにより、他の処理系との依存関係を作らず、独立した処理系として開発することを可能とする。

● 統一システム②：リソースマネージャとファイルマネージャによる動的リロード

基本システムにより、多くのリソースがランタイム中の動的リロードに対応する。

リソースマネージャによって提供される、スレッドセーフなリソースアクセスの手段により、各処理系はほとんど意識することなく、ランタイム中の動的なリソースのリロードを安全に行うことができる。（スクリプトのように、スクリプトエンジン側の対

応が必要な要素もあるので、全てに対応しきれるわけではないが）

ファイルマネージャは専用のオーサリングデータを扱う。パッチシステムのようなイメージで、動的リロードするデータだけをまとめたオーサリングデータを所定の位置におけば、ファイルマネージャからの働きかけにより、自動的にリロードが開始される。これにより、リロード対象を選別するよなユーザーインターフェースも不要とする。

● 統一システム③：頻繁なデータ構造変更に対応するゲームデータ処理機構

ゲームデータのデータ構造に対して、ランタイム時にバージョン整合を取る仕組みを提供することで、開発者とデータ制作者の双方の気苦労を軽減する。

ゲームデータの構造は変化し易いが、構造変更は安易に行えないのが通例である。構造変更時は、システム更新とデータ更新のタイミングを合わせるなど、何かと気苦労も多い。また、一部の整合をうしなったことにより、バグチェックのために過去バージョンのシステムを動作させてみるといったこともできなくしてしまう。

このような不便な問題を基本システムが解消する。先にシステム側でデータ構造を更新しても、古いデータを読み込んで動作することを可能とすることで、リリースのタイミングに対する気遣いを軽減する。新しいバージョンのデータを古いシステムで使用する場合も同様である。

▼ 【生産性】拡張性を考慮したデバッグシステム

デバッグ機能の拡充は、プログラム開発者の開発効率とコンテンツ制作者の生産効率を向上させる。そのためには、柔軟で拡張性に高いデバッグシステムが不可欠である。デバッグ機能の生産性が、システム全体の生産性につながる。

● デバッグシステム①：ユーザーインターフェースを分離した基本デバッグシステム

デバッグ機能の内部処理とユーザーインターフェースを切り分けることで、効率的なデバッグ機能の開発を可能とする。

基本的には「Windows のコマンドプロンプト」のように、コマンドベースでデバッグ機能を拡張する。例えば、「CPU ON」というコマンドを入力すると、CPU 使用率がゲーム画面上に表示される。これに対して、下記のようなユーザーインターフェースのアプローチをとることができる。

- ゲーム上のデバッグコマンドプロンプトで直接コマンドをキーボード入力。
- デバッグメニューのメニュー項目とデバッグコマンドとの関連づけ。その外部データ

化。
- ゲームコントローラーの特定の操作とデバッグコマンドとの関連づけ。その外部データ化。
- ネットワーク通信を介してデバッグ用アプリケーションと連携。

● デバッグシステム②：標準的なユニットテスト機能

ユニットテスト機能を標準化し、デイリービルドで毎日挙動が変わっていないことを再確認。これにより、クリティカルな処理に影響が及ぶ更新を早期発見し、開発効率の向上につなげる。

ユニットテストの機能を標準化し、プログラマーは、ある関数呼び出しに対して期待通りの結果が返ることをチェックするプログラムを書いておく。例えば、「EXPECT_EQ(funcAdd(10, 20), 30)」（funcAdd(10,20)の結果は 30 とイコールになることを期待する）のような書式である。

開発プロジェクトのビルドターゲット「ユニットテスト」でビルドされたプログラムは、プログラム中に書かれたユニットテストを一斉に実行し、エラーがあったかどうかを実行元に返す。Windows で実行しているなら環境変数 %ERRORLEVEL% に結果が返る。これを利用して、Jenkins などの CI ツールを使用し、毎日ビルドとユニットテストを自動実行し、実行結果を通知する。

また、ユニットテスト機能は任意の実行にも対応する。

例えば、ゲーム中のあるステージ上の時に実施したいユニットテストを用意し、それをデバッグメニューから実行して結果を確認できるようにする。

これにより、「ランタイムユニットテスト」として主要なテストを用意し、QA 作業を効率化するためのルーチンワークにするといった使い方ができる。

▼ 【パフォーマンス】マルチスレッドに最適化

マルチコア環境での動作を意識し、最適な並列・並行処理を行う基本システムを構築する。

● マルチスレッド①：シーンマネージャによる最適な並行化

シーングラフに基づき、依存関係にないオブジェクトの処理は並行実行する。

シーンマネージャは並行実行可能なオブジェクトを判断し、最適な処理スケジューリングを行う。

各オブジェクトはシーンマネージャから更新処理が呼び出されたら依存関係を深く気にせず、自身の処理に専念すればよい。ただし、更新処理は「移動」フェーズ、「衝突」フェーズ、「判断」フェーズに分かれ、それぞれの処理を実装する。若干手間はかかるものの、各フェーズでは前フェーズでの全オブジェクトの依存関係が解消されていることを前提に、安全な並行処理を実現する。

● マルチスレッド②：ジョブキューイングによる最適な並列化

多数の並行処理要件は、実行をキューイングし、ハードウェアに合わせて最適な並列処理で実行する。

シーンマネージャなどから要求される多数の並行処理要件をそのままマルチスレッドで処理せず、「ジョブ」として処理をキューイングする。実行環境に搭載されるコア数に応じて、並列実行の数を調整し、最適なパフォーマンスが得られるようにする。

● マルチスレッド③：リソースアクセス とリード・ライトロック

リソースマネージャが提供するリード・ライトロック機構により、スレッドセーフな処理を最大限に効率化する。

リソースマネージャが管理するリソースの一つ一つがリード・ライトロック機構を備える。並行処理要件の多くは「読み込み」であるため、読み込みが重なってもロックは発生せず、「書き込み」のタイミングでだけロックされる仕組みを提供する。

また、リソースアクセス時のプログラミングモデルが、そのようなロックが必然的に行われるものとして、プログラミング時の安全性を高める。

リソースマネージャはバックグラウンドでのリソース構築を行うが、リソースが可視化されるタイミング、および、リソースを削除するタイミング、リソースのメモリ再配置を行うタイミングを、ゲームループ中の一点に凝縮する。これにより、並行処理がブロックされる機会を極力軽減する。

▼ 【パフォーマンス】パフォーマンスに最適化した基本処理

頻繁に利用されてパフォーマンスに影響する基本システムを、よりよいパフォーマンスとなるように設計する。

● パフォーマンス①：パフォーマンスに最適化したメモリマネージャ

高速なメモリ確保／解放と自由なメモリブロック構築、メモリ再配置、参照カウン

タ、デバッグ情報の保持をメモリマネージャがサポートする。

メモリマネージャはスラブアロケータとヒープメモリのハイブリッドとして最適化する。

また、複数のメモリマネージャを持つことを可能とし、これにより、物理的なメモリブロックを分割可能にする。内部では論理的なメモリブロックも管理し、任意のカテゴリ（例えば「マップ用」や「メニュー用」など）でメモリの使用量制限や警告を可能とする。

ヒープメモリは通常のヘッダー部＋データ部の構造ではなく、管理部を専用のページフレームに割り当てることでメモリ再配置（コンパクション）を可能とする。管理部には参照カウンタも持ち、メモリマネージャのレベルでスマートポインタの管理を可能とする。（ガベージコレクションはとくに目的としない）

メモリ管理はページフレーム管理を基本として、管理部、各サイズ別のスラブ、ヒープに動的にページフレームを割り当てる。

- **パフォーマンス②：バージョン管理機構を備えたバイナリゲームデータ**

ゲームデータはバイナリデータで扱い、バージョンが一致する時はバイナリデータをそのままコピーすることで済ませる。

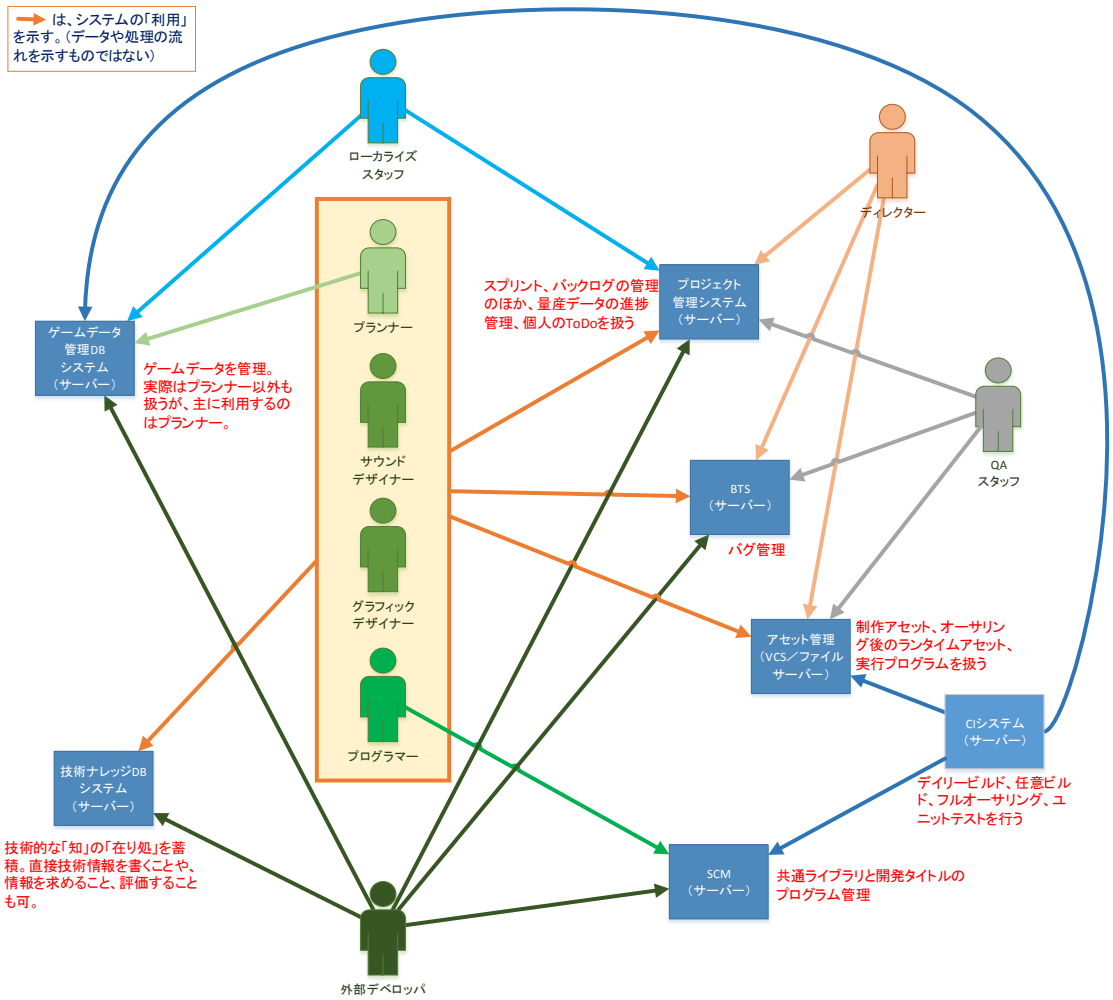
ゲームデータはバージョン管理機構を備え、バージョンが不一致の時は項目ごとのコピーを行う。

通常、バージョン不整合の問題はデータをスクリプト形式でデータを管理することで解消できるが、それでは読み込みのパフォーマンスが悪いので、バージョンが一致する時はそのままバイナリデータを転送して済ませればよい構造とする。（実際はコピー後に文字列データやネストする不定長データのポインター変換がある。）

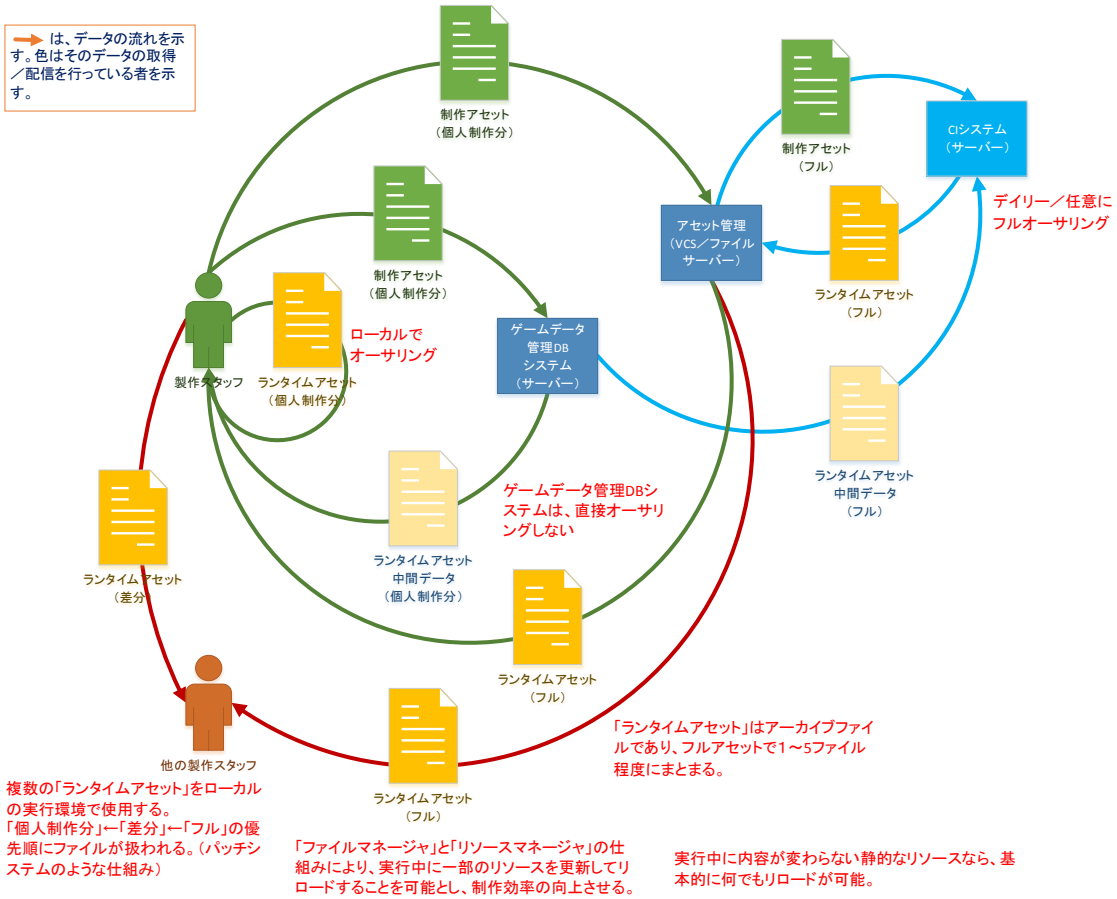
■ 開発環境の全体像

幾つかのドキュメントでは、開発環境の設計を行っている。その全体像を示す。

▼ 各種管理システムと制作スタッフの関係



▼ アセット管理



■ 開発環境の概略

開発環境の概略を示す。

▼ 【生産性・安全性・保守性】分散 SCM による効果的なプログラム開発

分散 SCM を活用し、生産性と安全性、保守性の高い開発環境を構築する。
なお、分散 SCM は Git の利用を想定している。

- **生産性：ローカルリポジトリを活用した障害発生時にも止まらない開発進行**

分散 SCM にはローカルリポジトリがあるため、サーバーの障害発生時にもコミッ

ト作業が停止しない。

障害発生中は、ローカル間でコミットを渡し合うことも可能。

障害復旧の際は、バックアップを活用するよりも、開発現場にあふれるローカルリポジトリから、最新の状態に近いものをピックアップして済ませることができるため、復旧も素早い。

● 生産性：ローカルコミットとブランチの活用による自己都合に合わせた開発進行

分散 SCM にはローカルリポジトリがあるため、重要なビルドの最中も気にせずコミットを進めることができる。また、強力なブランチ機能により、緊急修正コミットの要請にも迅速に応じることができる。

確実に動作するビルドが必要な際、スタッフの無造作なコミットを禁止することがあるが、分散 SCM ではその必要がない。

また、個人の作業を進めている最中に、重要なビルド向けに修正差分のみの緊急コミットが要請されることがある。このような場合、作業中の途中状態をローカルコミットし、ブランチを切り替えて行う。修正コミットを行ったら、またブランチを切り替えて、元の作業に戻るという流れを取る。

ブランチの切り替えは、Subversion などに比べて、圧倒的に高速で安全である。

● 安全性：トピックブランチを活用した安全なチーム開発進行

個人の作業の際は、必ずトピックブランチを作成して行う。一通りの機能の実装が済んだら、トピックブランチをグループリーダーに提出し、グループリーダーの判断で問題がなければ本流にマージする。

分散 SCM の強力なブランチ機能を活用し、個人の作業の際は、必ず作業用ブランチ（「トピックブランチ」と呼ぶ）を作成する。

作業が完了したら、直接本流にマージせず、グループリーダーに提出（プルリクエスト）する。グループリーダーは内容を確認した後に、問題がなければ本流にマージする。問題があればトピックブランチの再修正を要請する。手間はかかるが、安全性が高まり、チーム内の適正な連携も強化される。

このようなトピックブランチの活用により、途中状態のコミットをサーバーに送信（プッシュ）しても他者に影響がないため、障害に備えることができる。その上、本流のリビジョンは常に安定動作する状態を保つことができる。

● 透明性：トピックブランチと作業タスクの関連づけ

トピックブランチは常に何らかの作業タスクに基づいて作成する。これにより、スタッフの作業状況を分かり易くする。

トピックブランチには任意の名前を付けることができる。プロジェクト管理システムのタスク ID や、BTS のバグ／チケット ID を用いて分かり易い名前にする。

数名の作業を要するタスクの場合、「機能ブランチ」という扱いで、数名で共通利用するブランチを用意する。

● 保守性：サブモジュールを活用した共通ライブラリの管理

サブモジュール機能により、リポジトリの一部のディレクトリを別のリポジトリ（ブランチ）の参照にすることができる。

この機能を活用し、共通ライブラリを別リポジトリに分けて管理し、各タイトル向けのリポジトリは共通ライブラリのリポジトリを参照するように構成する。

複数のサブモジュールを扱っていても、「サブモジュールの更新」という一回の操作で全てを最新の状態にすることができる。

● 生産性：ローカルリポジトリを活用した効率的な遠隔地開発

遠隔地開発では、頻繁に中央のリポジトリにアクセスせずにローカルコミット主体で作業することで、通信速度の影響を受けずに作業することができる。

拠点内の共通リポジトリを扱うスタイルも良い。ある程度コミットがまとまったところで、中央リポジトリと同期する。

トピックブランチ、機能ブランチを用いた開発は、このようなケースでも問題をあまり起こさず進行できる。

▼ 【生産性】ローカルオーサリングによる効率的なコンテンツ制作

デイリービルドをベースにしつつ、手元の制作コンテンツ（アセット）をローカルオーサリングして素早く実機確認を行えるような、生産性の高い環境を構築する。

● 生産性：ローカルオーサリングによる差分制作で即時実機確認

その日そのチームまたは個人が制作しているコンテンツだけをまとめたローカルオーサリングを行うことで、制作したコンテンツをすぐに実機で確認できる。

ローカルオーサリングデータは複数扱うことができるため、その日のうちに更新されたビルドプログラムを使用する場合は、そのビルド向けにまとめられた差分オーサリングデータもいっしょに扱う。「（デイリー）フルオーサリングデータ」＋「差分オーサリングデータ」＋「ローカルオーサリングデータ」で扱う。

パッチシステムと同様のファイルシステムにより、それぞれのオーサリングデータ内に重複するファイルは、よりローカルに近いものが実機上で扱われる。

- **生産性：ローカルオーサリングの受け渡しで即時関連データを分配**

チーム間で依存関係のあるコンテンツを作成している場合、他のチームからローカルオーサリングデータを受け取ることですぐに自チームの環境に反映できる。

ローカルオーサリングデータはチーム間で受け渡しても良い。

- **生産性：ローカルオーサリングの入れ替えで効率的な共有マシンの利用**

ローカルオーサリングデータは１ファイルにまとまっている上、フルオーサリングデータの内容を破損することがないため、共有マシンを交代で利用する時に安全で素早く受け渡す事ができる。

交代の際は、ローカルオーサリングファイルを置き直すだけで良い。

- **生産性：ローカルオーサリングによるランタイムの動的なコンテンツのリロード**

特定のローカルオーサリングデータは、ランタイムでリロードさせることができる。

この時、使用するオーサリングファイルは「フルオーサリングデータ」＋「ローカルオーサリングデータ」＋「動的リロードオーサリングデータ」となる。ユーザーがデバッグメニューなどから動的リロードを実行すると、「動的リロードオーサリングデータ」の存在をチェックし、そこに含まれるファイルが構築済みのものであれば、リソースマネージャは自動的に再構築を行う。

リソースマネージャが提供するスレッドセーフ機構により、ランタイムでリソースが一時的に破棄されても（一時的に構築中状態に戻されても）、動作が停止することなく安全に処理される。ただし、スクリプトやコリジョン（物理演算用）など、正確にリロードが反映されないものもある。

▼ 【生産性・安全性】ゲームデータ DB システムによる効率的なデータ制作

ゲームデータを Excel などのファイルではなく、RDB＋ドキュメント DB で管理するシステムを構築し、同系統データの作成作業も複数名で同時進行できるような環境を構築する。また、バージョン管理の仕組みも備え、生産性と安全性を確立する。

● 生産性：ゲーム全体を横断するゲームデータ管理

ゲームデータ作成者（プランナー）は、ファイル読み替えの区分を気にせず、常に全域的なデータとして扱うことができる。

例えば、ゲーム進行に応じてあちこちの町に移動する NPC のデータを作成する場合、その NPC の時系列に沿ったデータ作成をすれば良い。オーサリングに際しては、どのようにデータを区切って（あるいは重複させて）まとめるか、プログラマーが調整する。

● 安全性：バージョン管理による安全なゲームデータ作成

バージョン管理の機能を備え、コミットの履歴や、任意のリビジョンのデータやコメントを確認することができる。

SCM ツールと似たバージョン管理を行う。

● 安全性：トピックブランチと局所的な編集ロックによる、安全な作業

トピックブランチの機能を備え、他者に影響を与えずにローカルオーサリングを可能にする。また、局所的な編集ロック機能により、編集の競合を防ぐ。

分散 SCM ツールと似たトピックブランチの管理を行う。
マージ機能は備えないので、編集ロックを行い、編集ロックしたデータしかコミットできない。

● 生産性：Excel エクスポート／インポートによる編集しやすいインターフェース

データ編集の際は、Excel にエクスポートして行い、編集後にインポートする。Excel を用いて、編集しやすいデータ入力が可能。

ただし、万能ではない。着色セルや罫線、欄外のコメントなど、多くの装飾が DB に記録できない（ある程度は保持するようにする）。データを削除したい場合は、Excel から行を削除するのではなく、削除データを明示するなどの手間もあるが、統一システ

ムとして慣れれば済むので、バージョン管理などの管理機能の利便性の方を重視する。

- **保守性：スケールアウト可能な分散システムとして構築**

Web サーバー＋RDB＋ドキュメント指向 DB でシステムを構築し、スケールアウトによる容量・パフォーマンスの増強、および、レプリケーション（クラスタリング）による障害対策を行う。

トピックブランチの作成、データのエクスポート／インポート、コミットなどの操作は、Web システムで構成する。内部で Excel を扱える必要があるので Microsoft の IIS (ASP.Net) を使用。DB 部分は PostgreSQL (or MySQL) + MongoDB といったシステム構成を想定。

▼【コスト削減】ゲームデータ DB システムによる効果的なローカライズ

ゲームデータ DB システムの機能により、ローカライズのコストを削減する。

- **コスト削減：複数タイトル管理による、シリーズタイトルの翻訳効率化**

シリーズタイトルは再翻訳不要なデータも多少ある上、前作の翻訳ポリシーの影響を受ける。ゲームデータ DB システムは複数タイトルを管理するので、前作のローカライズデータを同システム上で検索して参考にすることができる。

ユーザーに対するタイトルごとのアクセス権設定もあり、翻訳者や外部デベロッパが共通利用できるシステムとする。

- **コスト削減：全体検索機能で同じ名詞を異なる翻訳にするような問題を軽減**

DB 全体のテキスト検索に対応し、翻訳作業を強力にサポートする。

- **コスト削減：重複テキストのピックアップで無駄のない翻訳**

ゲーム内にはどうしても同じテキストが含まれる。検索機能は重複するテキストをピックアップして、翻訳不要なものとしてマーキングすることができる。（そのマーキング作業は先に日本語版制作者が行っておく）

重複ピックアップは、何も翻訳専用の機能ではなく、日本語データ作成時にも役立つ。

● **コスト削減：言語の依存関係ピックアップで並行翻訳作業を効率化**

最初から国際化したタイトルを開発する際、英訳が済んだそばから日本語が追加・変更されたり、スペイン語訳までされた後にその翻訳元の英語が修正されたりといったことが起こりえる。言語の依存関係に基づいて、翻訳後に元のテキストが追加・変更されたものをピックアップする機能を備える。

▼ 【生産性・品質向上】技術ナレッジ DB システムによる効率的な技術共有

様々な場所にある「技術」を検索するためのシステムを構築し、各開発者が必要な情報を素早く得て、生産性と品質向上につなげる。

● **生産性：マニュアルよりも「知」の所在を明確に**

読まれるかどうかわからないマニュアルを細かく作成するよりも、技術情報の「知」の在処を DB に簡単に記録する。

「技術ナレッジ DB」自体はリファレンスマニュアル的な性質のものではなく、「知」を登録する側も検索する側も極めて手軽に使えるものである必要がある。なので、そこには手の込んだ説明を書くのではなく、「ある技術」について「誰に聞けば分かるのか？」「どのプログラムを見れば参考になるのか？」「どのマニュアル・書籍を読むのが最適か？」といった内容を記録して扱う。取っ掛かりとして、「簡単に使える」ことが何より重要。

● **品質向上：「食べログ」的評価システムで「知」を洗練する**

登録されている「知」に対しては、コメントの書き込みや評価、さらなる詳細情報の要求などを可能とする。これにより、一度登録された「知」の情報は、後からどんどん増強され、情報の品質が高まる。

Web システムとして構成し、目的を絞った検索以外にも、「高評価な技術」「話題の技術」なども表示可能。「知」の情報には幾つもカテゴリを設定でき、「プログラム一般」「C++言語」「DCC ツール」といった任意のカテゴリを追加したり、組み合わせたりして扱える。

また、評価システムは「いい」「悪い」を扱うだけではない。「賛同者」により、投稿者以外にも「知」のサポートが可能であることを明示する。

● **品質向上：「Yahoo!知恵袋」的質問システムで「知」を要求する**

得たい「知」を要求することにも対応し、利便性を向上させる。

このような要求に対して得られた回答もまた重要な「知」のデータベースである。このような「知」の「要求」から始まることにより、システムの活性化を期待する。

▼ 【透明性】開発フェーズに合わせたアジャイル管理ツールによる連携の効率化

社内開発から外部デベロッパによる開発にいたるまで、アジャイル開発をサポートするWeb ツールを構築する。ただし、開発フェーズに合わせた二つの開発ツールを用意する。これにより、余計な手間を極力省きつつ、明確に状況把握が可能なシステムにする。手間がかからず、使い続けることが可能なシステムが、プロジェクト内の透明性を生む。

● **透明性：コンセプト～プリプロダクションフェーズを支えるバックログシステム**

コンセプトフェーズ、プリプロダクションフェーズは、スクラムのバックログを扱うシステムを活用。

無造作なバックログ登録→厳選・統合・分割→優先度設定→スプリント化・チーム編成→個人タスク化をサポートする。

初めから「タスク」に着眼しているTrac やRedmine などの既存のタスク管理ツールを利用するのではなく、もっと粒度の粗い要件から扱えるものとする。

個人タスクレベルでは複数のスプリントのタスクを抱えるスタッフもいる。スタッフは、判明しているタスク全体に優先度を割り当て、その順に作業予定日を設定する。カレンダー（休日予定など）を登録しておいて、自動的に作業予定日を割り振る機能も設ける。

スプリントとタスクが関連付いていれば、スプリントから足が出るタスクがわかるため、スプリントのリーダーとスタッフはどう調整するかを交渉する。

他者のタスク間との依存関係も設定できるが、それによる作業予定の調整は計算する機能は設けない。「間に合うかどうか？」や「順序が逆転しているかどうか？」などを伝えるだけのシステムとして、スタッフ間で予定の調整を交渉できるようにする。

このように、単純な機能の実装によって、交渉・コミュニケーションの手がかりとすることがこのツールのポイントである。

● **透明性：プロダクションフェーズを支える量産進行管理**

プロダクションフェーズは、量産のためのシステムを用いて、一つ一つタスク登録

しない（してもいいが必須ではない）。量産データのセクション間での連携もサポートする。

予定している量産データがどれだけあり、どれだけ達成できているかに着目したツール。ある量産データに対して作業項目と依存関係がある。例えば、モデル→テクスチャ→モーション→サウンドのような作業が設定され、前工程が完了したものをリストアップして作業することができる。

データが変更されて影響を受けることも捕捉できるものとする。例えば、「モーション完了」→「サウンド作成・完了」→「やっぱりモーション修正」といった流れが発生した場合、終わったはずのサウンドを再調整しなければならないことがこのシステムを通して判明する。

全体をまとめた達成度としても集計可能であるため、ディレクターが進行状況を把握することにも適切なツールとなる。

- **透明性：ポストプロダクションフェーズを支える BTS**

ポストプロダクションフェーズは基本的に BTS 利用。重要な提出タイミングなどの管理には、プリプロダクションフェーズと同様のバックログシステムを用いる。

このシステムの利用にあたって、BTS に対する特殊な要件はとくにない。

- **透明性：外部デベロッパとの連携にも利用**

外部デベロッパにもシステムを利用して貰い、状況確認をし易いようにする。

このような Web ベースのプロジェクト管理ツールを標準化し、外注管理の手法もある程度標準化する。

■ プログラミング方法論の概略

幾つかのドキュメントでは、プログラミングの方法論を説明している。

とくにC++言語を用いてゲームプログラミングする際に有効な手法や注意事項をまとめている。

一般的なプログラミング方法論というより、チームによるゲームプログラミングを強く意識している。チームメンバーでの共通認識を深めて意思疎通し易くすることや、全体的な技術力の底上げのために、文書化したものである。

▼ 【連携性】チーム開発の開発効率に影響するコーディングの説明

チーム内での開発に対する「意識」を共有し、安定した開発を行うための手法を説明する。

● 連携性：チーム開発のためのコーディング手法

基礎的のことながら、見落としがちなコーディング上の注意点を示す。

とくにメモリ事情に厳しいゲーム開発において気にかけるべき事項をまとめている。安易なプログラミングがどのような影響を及ぼすかを説明する。

▼ 【連携性】基礎的なプログラミングテクニックと用語の説明

チームで開発を行う上で、「共通認識」があったほうが意思の疎通を取りやすいことは間違いない。チーム開発を少しでも円滑なものにするために、基礎的なプログラミングテクニックや用語を説明する。

● 連携性：基礎的なプログラミングテクニック

チーム開発を少しでも円滑にするための、基礎的なプログラミングテクニックを抑える。

断片的なTipsながら、プログラミングの際に直面する「コーディング」、「分析」、「処理」に関するテクニックを説明する。

● 連携性：オブジェクト指向とC++

C++プログラマーが、一般的なオブジェクト指向の参考書を読み解くために、両者

の用語の違いを抑える。

例えば、デザインパターンの解説などは、一般的なオブジェクト指向の用語で書かれていることが多い。Java や C#も、どちらかと言えば、一般的なオブジェクト指向の用語に基づいている。しかし、C++言語は対応する用語があるものの、違う用語を用いている。

一般的な用語と C++言語との対応を説明し、オブジェクト指向の基礎を身につけることで、チーム内で「クラス図」などの共通言語を利用した開発を行い易くする。

▼ 【連携性】チーム開発の際に確認すべき事項

開発プロジェクトによってローカルルールというものはある。C 言語の標準ライブラリだからといって安易に使用してもよいものではない。プロジェクトに配属された際には確認して気をつけなければならない事項を説明する。

● 連携性：プログラミング禁則事項

開発プロジェクトによって「禁則事項」となり得る要素を説明する。

とくにゲーム開発では禁止されることが多い要素とその対処方法を説明する。とくに STL に関しては、利用を危険視すべき要素と、使っても問題ないであろう要素をわけて説明する。

▼ 【生産性・品質向上】効果的なプログラミングテクニックの説明

より実践的な、やや高度な C++言語のプログラミングテクニックを説明する。とくにゲームプログラミングに有効なものをまとめている。

● 生産性・品質向上：効果的なテンプレートテクニック

C++のテンプレートをゲーム開発に役立てる方法を説明。

テンプレートはゲームプログラミングでも非常に便利に活用することができることを説明する。

また、粗悪なコピペプログラムを題材に、いかにして最適化していくかというサンプルを丁寧に解説。STL のアルゴリズム活用、C++11 のラムダ式活用で更に洗練された最適化まで説明する。

最後には、テンプレートのより高度なテクニックを説明する。ゲームプログラミン

グに抜群の生産性とパフォーマンスをもたらす方法を提示する。

- **品質向上：デザインパターンの活用**

GoF による 23 のデザインパターンを説明。ゲームプログラミングに有効なパターンも多い。

ゲームプログラミングにおいても非常に有効性の高いプログラミングパターンを、具体的なサンプルプログラムを添えて説明する。

アダプターパターン、オブザーバーパターン、プロキシーパターン、デコレーターパターンなどは、ゲームプログラミングにも有効活用できる。

- **品質向上：プレイヤーに不満を感じさせないための乱数制御**

ゲームでは頻繁に用いられる乱数について、意外と難しい乱数制御の扱い方を説明。

ゲームでは、プレイヤーが「確率」を強く意識する場面がある。そのような場面でなるべく公平な乱数となるようにする方法を示す。

また、乱数の理解のために、乱数の性質も説明する。

▼【安定性・品質向上】マルチスレッドプログラミングの説明

マルチコアが一般的になった現在では、ゲームプログラミングでもマルチスレッドの活用が加速している。最適なマルチスレッドプログラミングのために理解すべきことと、プログラミングの方法論を提示する。

- **安定性・品質向上：マルチスレッドプログラミングの基礎**

マルチスレッドプログラミングを行う上で、基礎となる用語や仕組み、技術を説明。

プログラムが動作するハードウェア、OS、プログラムの動作原理といった基本事項の理解が、適切なマルチスレッドプログラミングには不可欠。

アトミック操作の失敗やデッドロックといったマルチスレッドプログラミングで起こりえる一般的な問題とその解決方法も解説。

高度なところでは、コンパイラによる最適化や CPU の最適化（アウト・オブ・オーダー実行）の影響で、一見してハードウェアやコンパイラの不具合を疑うような不可解な問題を引き起こすことがあることと、その解決方法も解説する。

割り込み処理のようなマルチスレッドとの競合要素や、メイン CPU 以外のプロセッサを用いた並列処理についても説明する。

● **安定性・品質向上：安全性と効率性のためのロック制御**

マルチスレッドの処理を安全かつ効率的にするための仕組みを提案する。

Scoped Lock Pattern や、リード・ライトロックの活用、更には、スレッドセーフなシングルトンパターンなどを解説。

■ ドキュメント一覧

以下、一連のドキュメントをカテゴリ別に列挙する。
各ドキュメントは、カテゴリごとのフォルダに配置している。

▼ インデックス

・ ゲームシステムのアーキテクチャと開発環境（本書）

▼ ゲームシステム系

＜ゲームループ管理＞
・ マルチスレッドによる**ゲームループ管理**
・ 安全性をのための**メッセージキュー管理**とイベントドリブン

＜ファイルシステム＞
・ 開発を効率化するための**ファイルシステム**

＜メモリ管理＞
・ ゲーム制御のための**メモリ管理**方針
・ ヒープメモリとスラブアロケータを併用した**メモリ管理**
・ 様々なメモリ管理手法と**共通アロケータインターフェース**

＜リソース管理＞
・ 開発の効率化と安全性のための**リソース管理**

＜シーン管理＞
・ ゲーム全体を円滑に制御するための**シーン管理**

＜デバイス管理＞
・ 反応性と安全性を考慮した**入力デバイス管理**

＜デバッグシステム＞

・ **デバッグ制御システム**

・ **ユニットテスト**と継続的ビルド

・ 効果的な**デバッグログ**と**アサーション**

＜ゲームデータ管理＞

・ **ゲームデータ**仕様

・ **ゲームデータ管理** DB システム

・ ローカライズのための**テキスト管理**構造

＜シリアライズ＞

・ セーブデータのための**シリアライズ**処理

＜スクリプト管理＞

・ スクリプトの生産性向上のための**プロパティマップ**

＜カメラシステム＞

・ **カメラ処理**の効率化手法

＜サウンドシステム＞

・ サラウンドとリソース管理を効率化するための**サウンドシステム**

＜イベントシステム＞

・ 効果的な**イベントストリーミングシステム**

＜レベル管理＞

・ オープンワールドのための**レベル管理**

＜AI＞

・ プランナーのための **AI システム**考察

▼ 開発環境系

＜プロジェクト管理＞

・ ゲーム開発のための**プロジェクト管理**

・ **プロジェクト管理** Web システム

・ **技術ナレッジ** DB システム

＜アセット管理＞

・ 効果的な**ランタイムアセット管理**

＜開発環境＞

・ 複数タイトルにまたがる効率的な**フレームワーク管理**

▼ プログラミング系

＜プログラミング Tips＞

・ チーム開発のための**コーディング手法**
・ 本当にちょっとした**プログラミング Tips**
・ オブジェクト指向と C++
・ プログラミング**禁則事項**
・ 効果的な**テンプレートテクニック**
・ **デザインパターン**の活用
・ プレイヤーに不満を感じさせないための**乱数制御**

＜マルチスレッド制御＞

・ **マルチスレットプログラミング**の基礎
・ 効率化と安全性のための**ロック制御**
・ **「サービス」**によるマルチスレッドの効率化

■ 【目的別】ドキュメント一覧

以下、一連のドキュメントを目的別にまとめ直して列挙する。
複数の目的にまたがるドキュメントもある。

▼ ゲームシステムに関するドキュメント

＜基本システム＞

・ マルチスレッドによる**ゲームループ管理**
・ ゲーム全体を円滑に制御するための**シーン管理**
・ 安全性をのための**メッセージキュー管理**とイベントドリブン
・ 開発の効率化と安全性のための**リソース管理**
・ 開発を効率化するための**ファイルシステム**
・ 反応性と安全性を考慮した**入力デバイス管理**
・ サラウンドとリソース管理を効率化するための**サウンドシステム**

＜基本仕様＞

・ ゲーム制御のための**メモリ管理**方針
・ **ゲームデータ**仕様
・ **カメラ処理**の効率化手法

＜拡張仕様＞

・ オープンワールドのための**レベル管理**
・ 効果的な**イベントストリーミングシステム**
・ プランナーのための **AI システム**考察

＜デバッグシステム＞

・ **デバッグ制御システム**
・ 効果的な**デバッグログ**と**アサーション**

▼ 開発環境に関するドキュメント

＜アセット管理＞

・ 効果的な**ランタイムアセット管理**
・ 開発を効率化するための**ファイルシステム**

＜ツール関係＞

・ **ゲームデータ**仕様
・ **ゲームデータ管理 DB** システム
・ ローカライズのための**テキスト管理**構造
・ 技術ナレッジ DB システム

＜プログラム関係＞

・ 複数タイトルにまたがる効率的な**フレームワーク管理**
・ **ユニットテスト**と継続的ビルド

▼ プロジェクト管理に関するドキュメント

＜プロジェクト管理＞

・ ゲーム開発のための**プロジェクト管理**
・ 複数タイトルにまたがる効率的な**フレームワーク管理**

＜管理ツール＞

・ **プロジェクト管理** Web システム

・ **技術ナレッジ DB** システム

▼ プログラミングに関するドキュメント

＜プログラミングの基本と規約＞

・ チーム開発のための**コーディング手法**
・ 本当にちょっとした**プログラミング Tips**
・ オブジェクト指向と C++
・ プログラミング**禁則事項**
・ **マルチスレットプログラミング**の基礎
・ 複数タイトルにまたがる効率的な**フレームワーク管理**

＜プログラミングテクニック／ライブラリ＞

・ 効果的な**テンプレートテクニック**
・ **デザインパターン**の活用
・ プレイヤーに不満を感じさせないための**乱数制御**
・ セーブデータのための**シリアライズ**処理
・ スクリプトの生産性向上のための**プロパティマップ**

＜処理仕様＞

・ **ゲームデータ**仕様
・ **カメラ処理**の効率化手法
・ **「サービス」**によるマルチスレッドの効率化
・ 効率化と安全性のための**ロック制御**
・ ゲーム制御のための**メモリ管理**方針
・ ヒープメモリとスラブアロケータを併用した**メモリ管理**
・ 様々なメモリ管理手法と**共通アロケータインターフェース**
・ 開発の効率化と安全性のための**リソース管理**

＜デバッグ＞

・ **デバッグ制御**システム
・ **ユニットテスト**と継続的ビルド
・ 効果的な**デバッグログ**と**アサーション**

■ ほとんど扱っていない事

一連のドキュメントは、基本ゲームシステム、データ処理、開発環境、プログラミング技術についてしかまとめていない。ゲーム開発には重要な要素がまだまだいくらでもある。例えば、以下のような重要な要素があるが、ほいとんど触れていない。

・ 3D グラフィック全般（ツール、アセット管理、データ構造、アニメーション、描画）
・ 2D グラフィック全般（ツール、アセット管理、データ構造、アニメーション、描画）
・ ライティング（ツール、アセット管理、データ構造、描画）
・ エフェクトシステム（パーティクルエフェクトシステムなど）
・ ポストエフェクト（アニメーション、描画）
・ グラフィック全般（シェーダー、描画）
・ シェーダー管理
・ UI／HUD（メニュー）システム
・ カメラ制御
・ サウンド（制御、データ）
・ ストリーミングシステム（音声、映像）
・ 物理演算・布処理
・ ネットワーク関係
・ 汎用／内製スクリプトエンジン
・ DCC ツール／グラフィックツール関係（プラグインなど）
・ サウンドツール関係
・ イベントシーン制作ツール
など...

■ 【課題】今後考えたい事

現状まとめきれていないが、今後考えていきたい課題も多い。

とくに、効果的な QA で品質を向上させるシステムは検討したいところ。

例えば、「A/B テスト」を応用したシステムなどは考えてみたい。「幾つかのパターンでプレイヤーが期待通りの行動を行うのはどれか？」といったことを試行し、自動的に集計されるようなシステムがあると、品質向上に貢献できそうである。何にせよ、プレイの反応がプランナーに迅速にフィードバックされるような仕組みは考えたい。

■■以上■■

ゲームシステムのアーキテクチャと開発環境

以　上