生成 AI 支援による非プログラマのゲーム実装
——ワークフローとプロンプト設計の実践的考察

東京藝術大学大学院映像研究科アニメーション専攻

学籍番号 3225365

彭一蘭／Yilan Peng

生成 AI 支援による非プログラマのゲーム実装

――ワークフローとプロンプト設計の実践的考察

Game Implementation by Non-Programmers with Generative AI Support: A Practice-Based Examination of Workflow and Prompt Design

要旨：

本研究は、プログラミングの基礎知識を持たない筆者が、生成 AI を活用してインディーゲーム『チーズ抜きでお願い』を単独で実装した開発プロセスを記録・考察したものである。AI 支援による開発ワークフローを三類型に整理した上で、「ケンタウロス・モデル」という人間が設計判断を担い、AI が実装を支援する対話型協働を、非プログラマーの立場から実践した。

実践を通じて、非プログラマー向けの協働フレームワークとして、三段階の開発プロセスと「What＋How」に基づくプロンプト設計フレームワークを提案した。後者は、意図の数値化・明示化と、境界条件の宣言および実装構想の提示によって AI の出力方向を能動的に制御するものである。また、標準的な実装手法が機能しない場面において、技術的複雑性を意図的に低減する「邪道的アプローチ」の有効性も示した。

本研究は単一事例に基づく探索的研究であり、急速に進化する生成 AI ツールの中で、非プログラマーが主導権を保持しながら AI と協働した特定時期の実践記録として位置づけられる。

1. 序論

1-1. 背景：生成 AI によるゲーム開発の協働パラダイム

近年、生成 AI によるテキスト・画像・コード生成の進展は、デジタルコンテンツ制作における分業構造を大きく変えつつある。とりわけソフトウェアおよびゲーム開発の分野では、大規模言語モデル（LLM）[1]のコード生成能力によって、プログラミング知識を持たない個人クリエイターであっても自然言語を通じてプログラム実装に関与できるようになり、「アイデア」と「実装」の間に長らく存在していた技術的障壁が次第に低減されている。

本稿で着目する協働構造を議論しやすくするため、近年のツールエコシステムと公開されている実践事例の観察をもとに、筆者は AI 支援によるゲームプログラミング開発のワークフローを、「実装プロセスにおける人間の介入深度（Human-in-the-loop / Human-out-of-the-loop）」と「成果物の修正可能性」という二軸に基づいて、以下の三類型として整理する。

1）プロンプト・ネイティブ・クリエイター （ The Prompt-Native Creator ）

このモデルは、Google AI Studio、Claude、DreamCore といった生成 AI プラットフォームや高度に自動化されたツールを中心とするものである。クリエイターは自然言語あるいは高度な専門的プロンプトを入力し、AI によって直接、実行可能なゲームのプロトタイプや完成品を生成する。このような、AI が生成したコードの内部構造を精査せず、直感的な対話と結果の選択によって開発を進める実践は、近年 Andrej Karpathy（2025）らによって「Vibe Coding[2]」とも呼称され、技術コミュニティで広く認知されつつある。このアプローチは使用者のプログラミング技術レベルを問わず成立し得るものであり、参入障壁を大きく下げるという点で極めて有利である。

しかし、このモデルにおいて、コード生成のレベルでは基本的に「Human-out-of-the-loop（人間が開発ループの外にいる）」状態にある。マルチターンのプロンプトを通じて結果を取捨選択することは可能であるものの、生成プロセスがブラックボックス化しているため、

1 大規模言語モデル（LLM）：大量のテキストデータを学習し、自然言語の生成・理解・翻訳などを行う AI モデルの総称。ChatGPT や Claude などがこれにあたる。

2 Vibe Coding：Andrej Karpathy が X（旧 Twitter）上で 2025 年 2 月に提唱した語。元の投稿ではこのスタイルを "fully giving in to the vibes" と表現している。

非プログラマーのクリエイターにとっては、下層のロジックを再構築したり継続的なイテレーションを行ったりすることが難しい場合が多い。

2) インダストリアル・インテグレーター（The Industrial Integrator）

このモデルは、専門的なプログラマーが GitHub Copilot3や MCP（Model Context Protocol）4などの技術を用い、AI を IDE（統合開発環境）5内部に直接組み込むことによって実現される。AI はリアルタイムでコードの補完、アルゴリズムの最適化、テスト支援などを行い、継続的なレビューとデバッグの中で開発効率を高める。

本モデルは「Human-in-the-loop（人間が開発ループの中にいる）」に分類される典型である。同時に、生成されたコードの品質を審査・制御するためには、使用者に高度なプログラミング能力とエンジニアリング知識が求められる。

3) ケンタウロス・エンジニア（The Centaur Engineer）

この概念は、チェス分野における人間と AI の協働実験に由来する6。すなわち、人間が戦略的判断や意図の誘導を担い、AI が戦術的な計算や具体的な実行を担うという、高度に統合された協働形態である。アルベスとシプリアーノ(2023)は、カスパロフの概念をソフトウェア開発に応用し、人間が目標設定・設計判断・検収を担い、AI が局所的な実装や高速な試行錯誤を行うことで、高頻度の対話を通じて開発を進めるワークフローとなる。

この形態も「Human-in-the-loop（人間が開発ループの中にいる）」の協働方式に属する。その核心的な特徴は、実装の工程を対話形式で AI にアウトソースしつつ、ロジックとユーザー体験に関する主導権を人間が保持する点である。

---

3 GitHub Copilot：GitHub が提供する、AI によるプログラミング支援ツール。IDE にプラグインとして組み込み、コード記述時に周辺の文脈を参照しながら、次に来るコードの補完・提案や雛形生成を行う。

4 MCP：AI モデルが外部ツールやデータ（例：ファイル、データベース、アプリケーション等）と、安全かつ標準化された方法で接続し、必要な文脈情報を受け渡すためのオープンプロトコル。

5 IDE（統合開発環境）：コードの記述、実行、デバッグ（バグ修正）、補完、プロジェクト管理など、ソフトウェア開発に必要な機能を統合した開発用アプリケーション（例：Visual Studio Code、Godot Editor のスクリプト機能など）。

6 チェス分野におけるケンタウロス：1997 年に IBM の「Deep Blue」に敗れた元チェス世界王者カスパロフが、翌 1998 年に提唱した概念。人間と AI が対立するのではなく、半人半馬の怪物（ケンタウロス）のようにチームを組む形式を指す。人間が「直感・戦略」を、コンピュータが「演算」を担うことで、最強の AI 単体にも勝利できるとされた。

本研究では、上記三つのうち、第三の「ケンタウロス・モデル」に着目する。ただし、その主体を「非プログラマーのクリエイター」に限定する。すなわち、コードを作る能力を持たないクリエイターが、いかにしてこの高強度の「Human-in-the-loop」型協働を通じて技術の壁を乗り越え、カスタマイズされた芸術表現を実現し得るのかを探究するものである。

1-2. パラダイム動機：創作における主導権と AI がもたらす安心感

美術大学に在籍する大学院生である筆者の専門は、ゲームにおけるグラフィックリソースの制作、ゲームメカニクス、プレイヤーの感情と体験デザインにあり、プログラム開発に関しては未経験である。これまでの創作はプログラマーとの協働によって行われてきた。しかし、今回の『チーズ抜きでお願い』の開発においては、AI を活用してプログラムエンジンの誘導やコード生成を行い、単独での開発を試みることを選択した。これは、「創作主権」と「プロジェクトのリスク管理」という現実的な観点に基づく判断である。

学生作品や個人制作などの、非商用プロジェクトにおいて、協力してくれるプログラマーを見つけることはしばしば困難を伴う。

有償で実装を外注するモデルを選んだ場合、実行力は担保されるが、商業的展望に乏しい学生の実験的作品として、市場水準の高額な報酬を将来的な収益で回収することは現実的ではない。

一方、無償の純粋な協力関係に基づく共創モデルを選んだ場合、創作主権の譲渡とすり合わせのリスクに直面する。プロジェクトが双方または複数人による共創となると、筆者は協力者の美的嗜好や技術的慣習を尊重する必要がある。互いのデザイン理念、好み、要求、能力のマッチングが一致しない場合、筆者はしばしば元の案を修正したり、妥協したりせざるを得ない。作りながら調整を重ねていく反復型の創作プロセスは、協力者にとって作業負担や心理的ストレスの要因となる。また、契約などによる拘束がない場合、こうした理由や個人的事情によって協力者が途中で離脱するリスクも高まる。

これに対して、非プログラマーが AI ツールを用いて実装を進めることは、コミュニケーションコストの増加こそ想定されるものの、低いリスクかつ高いコントロール性を持つ選択肢となり得る。AI は感情を持たず、デザイン上の発言権も主張しないため、筆者にとっては非常に高い心理的安全性を備えた試行錯誤の環境が得られる。

本研究では、AI が支援を行う環境下で、人間のプログラマーの関与なしに、非プログラマーであるクリエーターが自らのペースで構想を忠実に、かつ妥協することなく脳内のデ

ザイン構想を再現できるかどうかを検証することを目的とする。

2. 開発対象と環境構築

2-1. ゲーム概要と技術要件分析

本研究の実践対象は、インディーゲーム『チーズ抜きでお願い』である。本作品は、完全なナラティブ構造、マルチブランチ体験、そして独自仕様のインタラクションを備えた、小規模ながらも非テンプレート型のゲーム作品である。

2-1-1. ゲームのテーマとコアメカニクス

本作は、「写真を撮られるのが嫌いな人が、撮影を拒否できない」というジレンマをテーマとしている。この「逃げられず、尊重されない」という無力感と苦痛を表現するため、筆者は以下の三つのコア・ゲームプレイモジュールを設計した。

1) ステルスモジュール

プレイヤーはカメラマンと先生の視線が逸れた隙を狙ってクリックし、列から押し出るように移動し、ファインダーの視界から逃れる。このモジュールでは、ステート管理、発見判定、成功／失敗条件の処理、およびそれに応じた分岐演出への接続が必要となる。

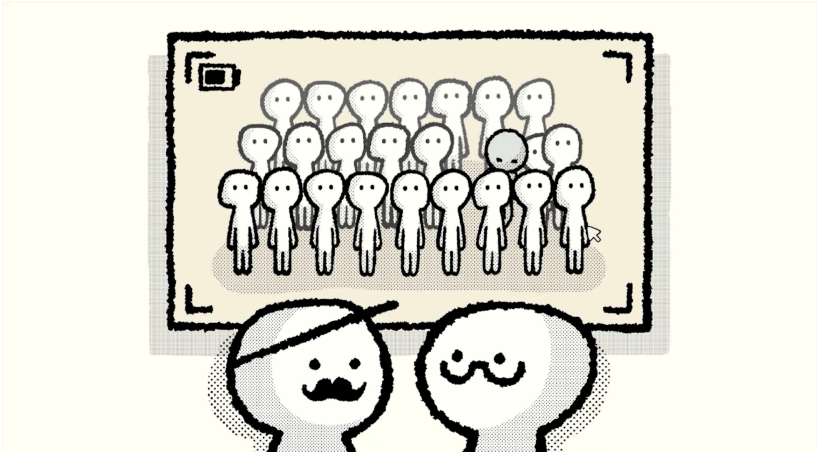

図 1　プレイヤー移動中

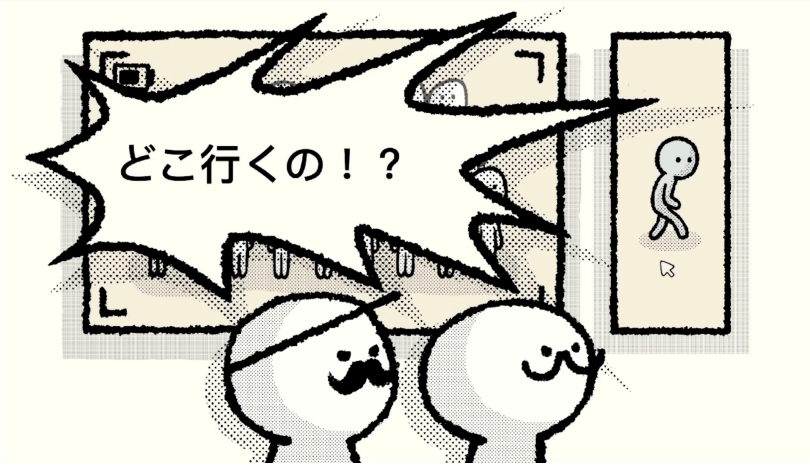

図 2　エンディング演出の一つ

2）交渉モジュール

プレイヤーは、複数の台詞吹き出しをスマートフォンの画面上にドラッグ＆ドロップして送信し、会話イベントをトリガーする。

図 3　送受信

この実装には、UI インタラクションシステム、物理演算によるドラッグ検出、ダイアログツリー、およびマルチエンディングを支える分岐管理構造が必要である。

3）QTE[7]演出モジュール

プレイヤーは、画面に浮かぶ「思考の吹き出し」を素早くクリックして、勇気を蓄積する。吹き出し内のテキストは、クリックに伴って徐々に文字化けし、最終的には別の語句へと変化することで、「言葉にできない思いや、意図通りに伝えられない心理状態」をシミュレー

7　QTE：Quick Time Event.ゲーム内で特定のイベント中に画面上に指示が表示され、プレイヤーが制限時間内に指定されたボタンを押すことで進行するゲームメカニクス。

トする。

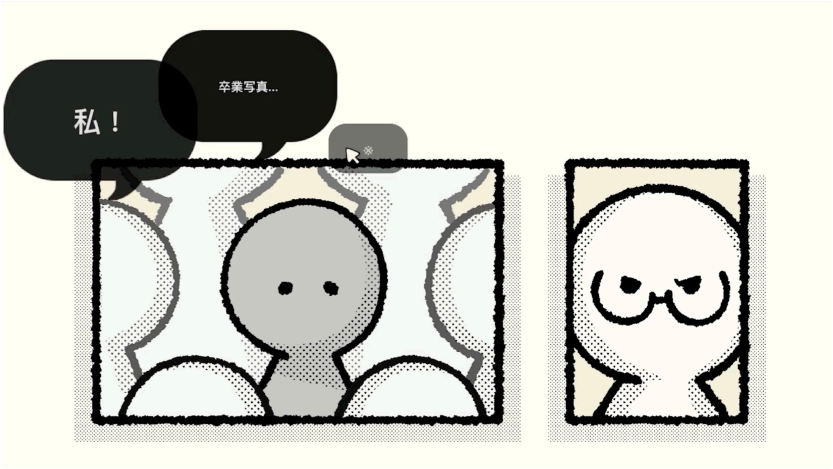

図 4　QTE 演出モジュール

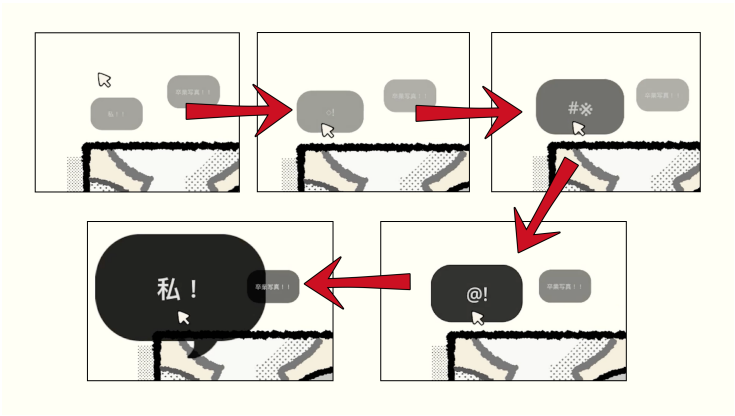

図 5　クリックによる吹き出しの変化

この実装には、カスタムされたテキスト状態制御と、セリフの演出管理を行うディレクター構造が必要となる。

2-1-2. 全体アーキテクチャと非プログラマーの課題

本作は、コアとなるゲームプレイに加えて、オープニングのチュートリアル、レベル選択、リトライ／放棄ロジック、多言語ローカライズ切り替え、エンディング演出、全体を通した演出管理構造など、完全なプレイフローと複数の補助的インタラクションを含んでいる。したがって、エンジニアリングの観点からは、本作は複数のシーンが密接に結合したシステム構造を有する作品であると言える。

非プログラマーである筆者にとっての困難は、これらの設計要件の多くが「非標準的」かつ「定型化されていない」ものである点にあった。そのため、Web 上においても汎用的なチュートリアルや実装例がほとんど存在せず、問題解決のための情報にアクセスすること

が極めて難しいという課題が生じた。

2-2.「ケンタウロス開発モデル」の確立

上述のような複雑な開発要件を踏まえ、筆者は既存の AI 支援開発ソリューションを評価した。その結果、最終的に「全自動生成」モデルを除外し、「ケンタウロス・モデル」の必要性を明確にした。

2-2-1. プロンプト・ネイティブの全自動生成の不適合性

Google AI Studio などのツールは、ゲームの直接生成をサポートしているものの、いわゆる「一言で生成」されるプロジェクトの成果物は、スネークゲーム、プラットフォーマー、モグラ叩きなど、既に確立されたゲームルールやフレームワークを持つジャンルのテンプレート的な複製であることが多い。

独創性の高い事例も存在するが、非プログラマーの開発者は結果を待つ立場に置かれ、生成されたゲームの内部ロジックや演出効果を精密に制御することができない。

また、こうしたツールで生成されるゲームの多くは Web または HTML5 ベースであり、構造的にも規模的にも限定的である（例：カウントダウンやスコア制を基盤としたミニゲームなど）。そのため、本作が求めるようなフルスケールのエンジニアリング構造や、Steam へのパブリッシュ要件を満たすには不適であり、直接生成型のワークフローには限界がある。

2-2-2. プラグイン統合のリスク

Copilot や MCP を用いて AI をゲームエンジンに直接統合する方法は、使用者にコードの審査とデバッグを行うプログラミング能力を要求する。初学者にとっては、こうした方法もまた生成プロセスが不可視なブラックボックス的アプローチとなる。

AI によってコードに加えられた変更が、思わぬ連鎖的エラーを引き起こした場合、非プログラマーの開発者がその原因を特定し、エラーを修正し、安定した状態へロールバックすることは困難である。このように、可視性の欠如は学習コストだけでなく、リスク対応の難易度にも直結する。

2-2-3. 適切なケンタウロス開発フロー

以上を踏まえ、独自仕様の非テンプレート的インタラクションロジックや、複合的なシーン構成、そして作品全体に対する制御性を同時に確保するために、「ケンタウロス・モデル」の開発フローを採用した。すなわち、Web ベースの大規模言語モデル（LLM）との対話セッションを通じて、操作ガイド、コード、修正提案を逐次取得し、それらを手動でエンジン内に移植しながら段階的に検証するという開発フローである。

より泥臭くとも言えるが、制御可能性の高い手法である。この方式により、コアファイルの誤改変リスクを低減できるとともに、各変更箇所とその実行結果を逐一確認することが求められるため、非プログラマーにとっては安全性と理解度の両立が図られるアプローチとなった。

2-3. 具体的なツールの選定と理由

2-3-1. ゲームエンジン：Godot Engine 4.5.1

Unity、UE4、UE5 などのゲームエンジンと比較して、Godot のノード（Node）アーキテクチャは論理構造が明快であり、コンポーネント化された設計と優れたローカライズ対応は、初めてゲームエンジンを使用する非プログラマーにとって非常に親和性が高い。

また、Godot が採用するスクリプト言語 GDScript は、文法が Python に近く、筆者にとって読みやすく、ロジックの理解もしやすいため、「手動で移植＋段階的検証」という開発フローにおいて非常に適している。

2-3-2. AI モデル：ChatGPT 4o / 4.5（Plus 版）

Claude、Gemini、ChatGPT を比較検討した結果、筆者は ChatGPT を選択した。その主な理由は、ChatGPT Plus が持つ以下の特性にある：

- 長大なコンテキストウィンドウ
- 優れたコンテキスト保持能力
- 複数ウィンドウ間での記憶保持（クロスウィンドウ記憶）
- 長期的かつ多モジュール型の開発に最適化された「プロジェクト・対話グループ」機能

この「プロジェクト・対話グループ」機能により、異なる対話を分類・整理し、それぞれに共通の背景ファイルを設定することが可能となる。

筆者には、ゲームを機能単位で分割し、各セクションを個別に開発・管理するという実践上の要件があったが、この機能によって、統一された文脈の下で、複数の対話ウィンドウを用いて異なる機能の開発を並行かつ整然と進めることが可能となった。

これにより、複数の機能が一つの対話内で混在することによって生じる対話の肥大化や論理の破綻を防ぎ、長期的な開発における安定性を高める結果となった。

３．実制作プロセスとワークフロー

本章では、筆者が『チーズ抜きでお願い』の開発において実際に採用した作業フローを記録する。このフローは、非プログラマーという条件下で実装を推進しつつ、ミスによるコストを最小限に抑えるため、開発の過程で徐々に定着したものである。

3-1. 全体プロセスの概観

本作では、ブロックごとの反復型開発モデルを採用しており、全体のプロセスは以下の 3 段階に分けられる。

1) コンテキスト同期と構造策定

プロジェクト全体の要件を整理し、開発の順序、エンジニアリングの骨組み、AI による分割協働の実現可能性を確定する。

2) 実装ループ（主要工程）

機能ブロック単位で、「役割宣言とプロジェクト状態の同期 - 指示とコードの取得 - 実際およびコードの移植 - 修正と最適化 - バージョン管理とリスクコントロール」のサイクルを繰り返す。

3) 統合

分散開発された独立シーンを一貫したゲームフローとして結合し、ローカライズおよびエクスポート処理を行う。

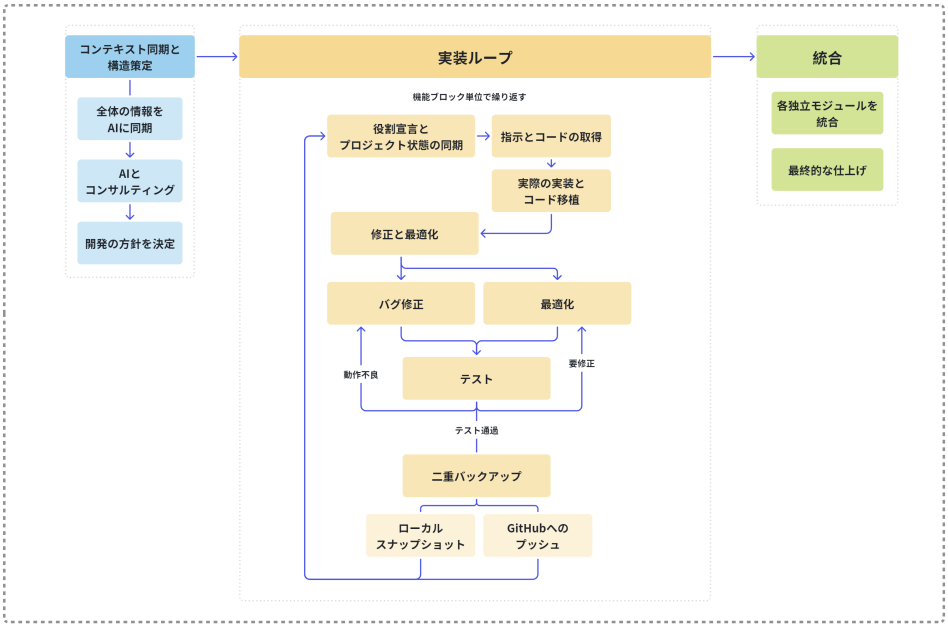

図 6　全体のプロセス

3-2.　段階 1：コンテキスト同期と構造策定の詳細

正式なプログラム実装を開始する前に、筆者はまず AI にプロジェクト全体の情報を同期し、開発構造の計画についての相談を行った。

1) 入力内容

作品テーマとゲーム全体のフロー、主要機能モジュール（ステルス、交渉、QTE）、使用予定のゲームエンジン、筆者のプログラミングスキルレベル。

2) コンサルティング

AI に対して、合理的な開発順序とシーン分割に関する提案を求めた。また、開発中によく使用される基本用語の解説を依頼し、不確定な要素（例：後工程でのシーン連結の実現性、BGM や効果音の追加タイミング、バージョン管理の方法など）についても質問を行った。

3) 意思決定

以上のプロセスを経て、「独立シーンごとの開発＋メイン制御シーンによる統合」という戦略を採用する方針を確立した。

ゲームプレイモジュールと全体構造に基づいて独立シーンを個別に構築・プレテストし、

その後、通過したもののみを連結工程に進めることで、

- 単一の大規模シーン内で複数のロジックを同時に処理することによるカップリングの上昇と回帰テストの難化
- AI とのやりとりにおいて、プロジェクト状況の過度な情報共有が必要となることによるコミュニケーションコストの増大

というようなリスクを回避する狙いがあった。

3-3. 段階 2：実装ループの詳細

アーキテクチャを確立した後、開発は機能ブロック単位で反復的に推進された。各機能ブロックの開発は、以下の 5 つのステップに従って行われた。

1) Step 1: 役割宣言とプロジェクト状態の同期

新たな機能モジュールに関する対話ウィンドウを開く際、筆者はまず AI に対して自身の技術的前提条件を明示する（例:「私は完全にプログラミングとゲームエンジンの素人です。バカとして扱い、わかりやすく明確な手順を指示してください」）。

さらに、そのシーンにすでに実装されている機能がある場合には、現在のノード構造、実装済みのコードや機能の詳細、使用中のリソース名などを AI に伝達し、旧機能との競合やリソース名の不一致によるエラーを未然に防ぐ。

2) Step 2: 指示とコードの取得

AI に現在の目標を AI に送信する。実装したいインタラクションや条件分岐、演出、使用素材などの要件を AI に具体的かつ詳細に説明し、それに基づく実装案、エンジン内での操作手順、コードの生成を依頼する。抽象的な議論やオーバーエンジニアリングを減らすため、「最小限の実行可能プロトタイプを目標として生成してください」というプロンプトを AI に補足する。

3) Step 3: 実際の実装とコード移植

AI が提示した手順に従い、Godot エンジン内でノードツリーの構築、美術リソースのインポート、各種プロパティの設定、スクリプトの作成およびコードの貼り付けを実施する。

作業中に不明点が発生した場合は、直ちに AI に詳細な再説明を求めることで対応した。

コードの移植に際しては、

- ノードパスと命名の整合性
- シグナル接続の正確性
- 依存リソース（スクリプト、音声、美術素材など）の存在確認

というような点に重点的に照合する。

4) Step 4: 修正と最適化

一通りの構築とコード移植を完了後、通常はバグ修正と最適化の両面からイテレーションが必要となる。

（a）動作しない場合、バグ修正ステップに入る

- **フィードバック**：ゲーム実行後の不具合について、エンジンによるエラーログと筆者の自然言語による観察（例：「クリックしてもキャラクターが動かない」「操作後にクラッシュする」）を AI に報告する。
- **再生成**：AI が原因分析と修正用の指示・コードを提供
- **修正・テスト・繰り返し**：修正後の再テストを繰り返し、正常動作を確認するまでループする。

（b）動作はするが表現と機能が満足できない場合、最適化ステップに入る

- **フィードバック**：現在の挙動を AI に説明し、改善したいポイントや期待する効果を補足（例：「状態切り替えが早すぎる」「便利で数値調整の仕組みを加えたい」など）
- **再生成**：AI が既存コードを踏まえて最適化用の修正案を出力
- **修正・テスト・繰り返し**：再テストの結果を踏まえてさらに改善を重ねる。新たなバグが出た場合は再び修正ステップに戻る。

5) Step 5: バージョン管理とリスクコントロール

AI によるコード生成は予測不可能な変更をもたらすことがある。そのため、新機能の追加や旧機能の最適化により、既存の正常動作が破壊されるリスクは高い。

安定して動作するバージョンにロールバックできない事態を避けるため、「機能ブロック

内での高速ロールバック」と「機能完成後の段階的アーカイブ」というニーズに応じて、二重バックアップ体制を採用した。

(a) ローカルスナップショット

開発中の機能ブロックで「安定して動作する」状態に達し、小さな進捗があるたびに、例えば、基礎バージョンの動作が実装済み、ある最適化の動作が確認でき、ある判定ロジックが安定した際には、プロジェクトフォルダ全体を複製し、バージョン管理フォルダに保存する。

この方式は、エラーが複合的に蓄積し、構造が把握困難となった際でも、安定版に迅速に戻れる利点がある。コードの精読やリファクタリングが困難な非プログラマーにとって、安定状態にロールバックして再構築するほうが効率的なことも多い。

(b) GitHub[8]へのプッシュ

機能が完成し、当面の修正・最適化の予定がないと判断した時点で、その状態を GitHub にプッシュし、段階的なバージョンとして記録する。

これは将来的な保守や誤操作、ローカルデータの破損に対する復元手段として機能する。

3-4. 段階 3：統合の詳細

各独立モジュール（ステルス、交渉、QTE、失敗演出、オープニング、エンディング）が安定して動作した後、筆者は引き続き、前述の Step 1〜5 に基づく協働方式を採用し、AI によるエンジン操作指示およびコード生成のサポートのもとで、統合作業と最終的な仕上げ、あるいは収束処理を行った。

具体的には、ディレクターシーンを新たに作成し、

- 分岐パートにおける「再体験／放棄」ループの制御
- 各分岐の完了フラグに応じた選択画面の非表示処理
- 多言語設定（ローカライズ言語）のシーン間同期処理
- グローバルステート（全体状態変数）の設計と運用
- 必要なシグナルの送受信とイベント管理の実装

8 Github：コードやプロジェクトファイルをオンライン上に保存・管理するためのプラットフォーム。変更のたびに履歴が記録されるため、過去の任意の状態に戻すことができる。

など、プロジェクト全体に共通する機能を一元的に統合した。

4. 制作における課題と解決策

第 3 章で述べたワークフローは、開発の基本的なフレームワークを提供したものの、実際の制作過程では、依然として多くの困難に直面した。それらは主に以下の 4 つの原因に分類できる。

- 自然言語の曖昧さによって生じる、AI による要求解釈のズレ
- AI の標準化バイアスによって、実装および保守の難易度が上昇する問題
- AI の知識の陳腐化や文脈の断絶による、指示やコードの無効化
- AI による継ぎ接ぎ的な修正傾向と、非プログラマーのコードレビュー／リファクタリング能力の不足との衝突によって引き起こされる、機能の不安定化

本章では、これらの問題に対して筆者が実際に遭遇した具体的な事例と、それぞれに対する対応策を提示する。

4-1. 課題 1：自然言語の曖昧さと数値化の必要性

1) 問題の説明

美術的バックグラウンドを持つクリエイターとして、筆者は仕様を伝える際、しばしば「感覚的」かつ「包括的」な表現（例：「素早く移動する」「吹き出しの端が水面の波のように揺れる」）を用いてしまう傾向がある。しかしながら、プログラムの実装は明確なパラメータ・閾値・計算可能なルールに依拠しているため、こうした曖昧な指示は、AI 側にとって「推測」による実装を余儀なくさせ、結果として意図と乖離したアウトプットを生むことになる。

2) 典型的な実例と解決過程

たとえば交渉モジュールにおける SMS シーンでは、「吹き出しをドラッグ→メッセージ送信→相手の返信が生成される」という一連のインタラクションを実装する必要があった。筆者は、新着メッセージが常に画面下部から出現し、古いメッセージは上方向へ押し上げられ、一定の間隔と左右位置を維持するというような、リアルなチャットアプリに類似した効

果を再現したいと考えていた。

当初、筆者が AI に与えた指示は以下のような自然言語による記述だった：

「主人公のメッセージでも母のメッセージでも、新着メッセージであれば常に最下部に現れ、既存メッセージはその分上に押し上げられる。吹き出し同士は適切な間隔を保って表示される。」

この抽象的な表現に基づき AI が生成したコードでは、吹き出しの位置が乱れる、画面外に飛び出す、指定位置に正しく生成されないといった問題が頻発し、繰り返しデバッグとフィードバックを行っても安定した実装には至らなかった。

問題の一因は、「上に押し上げる」「適切な間隔」といった定量化されていない曖昧な語句にあり、もう一因は、SMS 画面自体が複雑なノード階層とレイアウトルールを伴う UI 構造であったため、一見合理的に見える座標操作が、実行時にはエンジンの自動整列機能によって混乱されていた点にあった。

最終的に、筆者は AI に対し「エンジンの自動レイアウト機能でメッセージを自動で押し上げることを諦めてください」と指示し、代わりに数値・計算式による配置の実装を依頼する方向に転換した。

すなわち、新規メッセージの生成座標を常に一定位置に固定し、既存メッセージの Y 座標を「y =元の位置 − 新しい吹き出しの高さ − 固定間隔」という数式で算出、順次上方向に配置し直すという方法を AI に明確にし、指導とコードを生成するよう要求した。このように位置関係が数式として定義されたことにより、AI の出力するレイアウトロジックは安定し、筆者の意図通りの表示挙動がようやく実現された。

同様の問題は、シェーダーによる質感表現の実装にも見られた。たとえば、筆者は「線が手描き風に震えるような演出」を希望していたが、AI は初期段階で sin/cos 関数による周期的な波形変位を採用する傾向があり、結果として機械的で不自然な演出となっていた。

筆者はこの「震え」の比喩を幾何学的な動作記述へと再定義し、AI には「頂点が X 軸方向に対して、高周波・小振幅・ランダム性を伴う変位を行うようにしてほしい。高周波・小振幅・ランダム性は調整可能にすること。」というように依頼を変更した。これにより、期待する手描きのような自然な揺らぎに近づけることができた。

3) 一般化された対応策

意識的に、曖昧な感覚的表現と視覚的比喩を具象的な数値的表現に翻訳し、パラメータ、

閾値、範囲、周波数/振幅など、明確な変数関係を用いて表現する。

4-2. 課題 2：AI の「正攻法」バイアスと「邪道」の実践

1) 問題説明

実装方式を指定せずに要件のみを提示した場合、AI は教科書的で標準化された、汎用性の高い実装案（いわゆる「正攻法」）を優先して提示する傾向がある。しかし、こうした標準的な手法は、ノード階層の深さや設定手順の複雑さを伴いやすく、保守コストも高くなりがちである。また、既存の実装との整合性が取れない場合、エラーの原因にもなりやすい。コードのレビューやリファクタリングを行う能力を持たない非プログラマーは、このような状況において原因を特定できず、試行を重ねても機能を実現できないという状態に陥る。

2) 典型的な困難と解決過程

交渉モジュールの SMS シーンにおいて、「文字数に応じて吹き出しのサイズを自動的に変化させる」機能を実装する際、AI は Godot 標準の NinePatchRect ノード（9 スライス）の使用を提案した。この手法はエンジニアリング的には合理的であるが、テクスチャ領域の分割、アンカー設定、ノードのネストといった複雑な構成を必要とする。

筆者は指示に従って繰り返し試みたが、階層の競合や引き伸ばしによる変形の問題により、実装は安定しなかった。

この機能は既存の「メッセージ生成と上方向移動」のレイアウトロジックと密結合しており、NinePatchRect 側のノード構造を変更すると、上方向移動の処理が正常に動作しなくなる可能性があった。すでに上方向移動ロジックの安定化に多大なコストを費やしていた筆者にとって、既存構造を維持したまま全面的に作り直すことは現実的ではなかった。

最終的に筆者は、AI から有効な解決策を引き出すことを断念し、自ら「邪道」な代替案を採用した。すなわち、無段階の自動リサイズという連続性を放棄し、あらかじめ固定の高さを持つ複数の吹き出し素材（1 行用・2 行用・3 行用・4 行用）を用意し、実行時に文字数の区間に応じて if-else 分岐で対応するテクスチャへ切り替える方式である。

```gdscript
# 估算 rank：按文字长度粗略判断几倍高气泡
func _pick_rank_by_text_len(text: String) -> int:
    var lines = int(ceil(float(text.length()) / CHARS_PER_LINE))
    if lines <= 1:
        return 1
    elif lines == 2:
        return 2
    elif lines == 3:
        return 3
    else:
        return 4

# 玩家 / 家长 各自的贴图选择函数（不再串用）
func _apply_player_texture_by_rank(bubble: Sprite2D, rank: int) -> void:
    if rank == 1:
        bubble.texture = player_tpl.texture
    elif rank == 2 and player_tex_2x:
        bubble.texture = player_tex_2x
    elif rank == 3 and player_tex_3x:
        bubble.texture = player_tex_3x
    elif rank >= 4 and player_tex_4x:
        bubble.texture = player_tex_4x
    else:
        bubble.texture = player_tpl.texture
```

図 7　吹き出しサイズを変化させるコードと用いたアートリソース

この方法は精密な自動サイズ調整を犠牲にする一方で、ロジックが単純で動作も安定しており、NinePatchRect の設定や密結合によって生じる不具合を効果的に回避することができた。

3) 一般化された対応策

標準的な実装案の試行錯誤コストが過大である場合、あるいは既存構造との密結合により根本的な再設計が困難であり、繰り返し試行しても実装に至らない場合には、リソース数の増加やプリセット素材の活用によってロジックを単純化する「邪道的な代替案」を積極的に検討すべきである。

4-3. 課題 3：バージョンの不整合とコンテキストの断絶

1) 問題説明

長期間の開発において、AI との協働では主に二種類のズレが生じる。第一に、バージョン情報の不整合であり、旧バージョンのエンジンに基づいた、実際には使用できない指示やコードが出力される問題である。第二に、コンテキストの断絶であり、対話が長期化した場合やウィンドウを切り替えた際に、AI がプロジェクトの現状を自動的に引き継げず、開発の進捗が連続しない、あるいは生成されたコードが既存機能と整合しないといった問題である。

2) 典型的な困難と解決過程

バージョンの不整合として最も典型的に現れるのは、AI が Godot 3.x 系の記法でコード

を提示し、それが Godot 4.5.1 では適用できないケース、あるいは操作指示に含まれるエンジン機能や設定項目が現行バージョンに存在しないケースである。

このような問題に遭遇した場合、筆者はまず AI に対して使用しているエンジンバージョンを明示する。また、「該当項目が存在しない」と判断された場合には、エンジン内の検索機能および公式ドキュメントとの照合結果を基準とし、「その内容は 4.5.1 には存在しないため、4.x 系での代替手法を提示してほしい」とフィードバックする。

コンテキストの断絶は、新しい対話で開発を継続する場合や、新機能の実装を開始する際に毎回発生する問題である。第 3-3 節 Step1 の「プロジェクト状態の同期」は、この問題に対処するために設けられたステップである。

筆者は各新規対話の冒頭において、現在のノード構造、主要なスクリプトおよびリソースの命名規則、実装済み機能とその動作状態を整理して提示し、AI が開発の背景を把握したうえで作業に入れるようにする。また、既存機能との境界を侵さないよう、その都度 AI に対して明示的に指示する。

コンテキストの断絶自体は技術的に高度な問題ではない。プロジェクト状態の同期は難易度こそ低いが、単調で時間を要する作業である。しかし、この手順を踏むことで、後続の補足説明にかかるコミュニケーションコストと手戻りの発生率を大幅に低減することができる。

4-4. 課題 4：修正の累積とリファクタリング不能性

1) 問題説明

バグ修正の過程で、AI はしばしば条件分岐の追加や一時変数の導入といった、パッチワーク的な手法で機能を動作させようとする。修正が繰り返し積み重なると、コード構造は冗長かつ脆弱になりやすい。一方、非プログラマーはコードレビュー・リファクタリング・保守の能力を持たないことが多いため、既存の実装に修正を重ね続けるこの方式は、直すほどバグが増え、問題が連鎖し、元々動いていた機能まで動かなくなるという制御不能な状態を引き起こしやすい。

2) 一般化された対応策

修正が「パッチの累積・構造の脆弱化・問題の連鎖」という傾向を示し始めたら、より安定した時点まで戻り、実装の方針を立て直してから再度進めることが有効である。第 3-3 節

Step5 の二重バックアップ方式に従ってバージョンをアーカイブしておくことで、迅速なロールバックと安定バージョンへの回復できる余地を維持できる。

5. AI 協働を安定化させるプロンプト設計提案

本章では、前章までの実践を踏まえ、自然言語の曖昧さと AI の標準化バイアスを克服するため、本研究では非プログラマーが指示を出す際に「What（意図の記述）＋How（実装構想）」という記述フレームワークに従うことを提案する。

5-1. プロンプトの基本式：What＋How

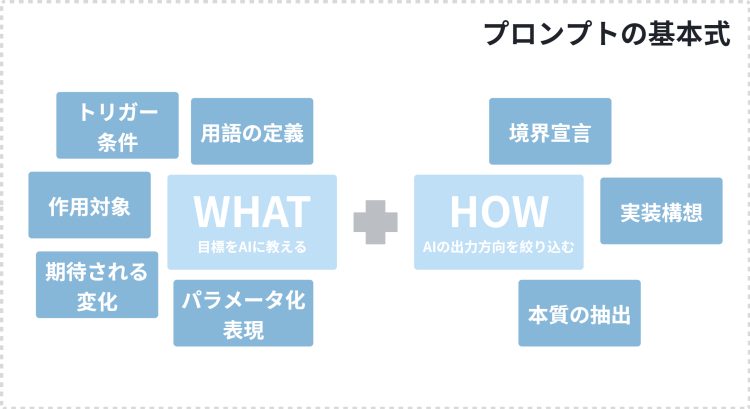

図 8　プロンプトの基本式：What＋How

5-1-1. What：客観的かつ数値化された目標記述

What パートは、「どのような結果を達成したいか」を明確に定義する役割を担う。曖昧さを避けるため、感覚的・主観的な形容表現はできる限り控え、検証可能なルールと数値による記述を優先すべきである。

具体的には、以下の 5 要素を含めることを推奨する。

- 用語の定義：要件に含まれる口語的・非厳密な表現について、その意味と適用範囲を AI に明示する。
- トリガー条件: プレイヤーがどのような操作を行ったか、またはゲームがどのような状態にあるかを記述する。

- 作用対象: 対象となる具体的なノード名、変数名、またはキャラクターオブジェクトを明示する。
- 期待される変化: 動作の変化、視覚的フィードバック、または判定ロジックの変更内容を具体的に示す。
- パラメータ化表現: 曖昧な形容表現を、具体的な数値や数式へと変換する。

例	Bad	Good
	「スペースキーを押すともっと速く動く」	「プレイヤーがスペースキーを押した時、キャラクターの移動速度を基礎速度の 1.5 倍にする」
	「吹き出しの間隔をいい感じに保つ」	「吹き出しの間隔を 12px に固定する」

5-1-2. How：境界宣言と実装構想

How パートの核心は「AI の提案への盲目的な依存を打破する」ことにある。非プログラマーは、自身の状況に適した制御性の高いアウトプットを得るために、AI の出力方向を能動的に絞り込む必要がある。

1）境界宣言

What で記述した要件に対し、AI は非プログラマーには制御しづらい「既存コードのリファクタリング」や「複雑な自動レイアウトシステムの導入」といった手法を提案することがある。こうした方式は既存機能の安定性を損ないやすい。既存の実装を保護するため、「何をしてはいけないか」を AI に明示することが重要である。

例	「既存の上方向移動ロジックはそのままに、色変化の機能のみを差分的に追加してください。元のコードはリファクタリングしないでください。」
	「自動レイアウトコンテナ（Container）は使用しないでください。」

2）実装構想

「こういう方法でできるのではないか」という仮説を提示し、非プログラマーにとって理解しやすく保守しやすいロジックを AI が採用するよう誘導する。その実装アイデアが正攻法でなく、単純すぎるように見えたとしても、標準的な手法が機能しない局面では予想外の効果を発揮することがある。確信が持てない場合は、疑問形で AI にその実現可能性を確認

してもよい。

例	「NinePatchRect ノードを使わず、行高の異なる PNG を私が提供するので、文字数（<10, 10-20, >20）に応じてテクスチャを切り替える方式で実装できますか？」
	「プレイヤーが連続してクリックするにつれ心拍音が徐々に明瞭になり、クリックを止めると徐々に消えていく演出を実装したいです。心拍音を常時・人の耳に聞こえない音量で再生しておき、クリックのたびに一定量音量を上げ、停止後は初期音量に向けて徐々に下げ、上限音量も設定できるようにする、という実装は可能ですか？」

3）補足：視覚現象から数理的本質へ

実装構想を考える際、視覚的な見た目にとらわれず、動作の数学的な本質から捉え直すことが有効な場合がある。経験豊富な開発者にとっては直感的なアプローチかもしれないが、非プログラマーにとっては普段の思考や表現とは異なる発想である。

たとえば、吹き出しの上方向移動は視覚的には「新着メッセージが来ると古いメッセージが上に押し上げられる」ように見える。しかしこの表現は AI に対し、複雑な UI コンテナによる自動レイアウトロジックの使用を誘発しやすい。

その数学的な本質を整理すると、「新着メッセージは常に固定座標に出現し、既存メッセージの Y 座標は新着メッセージの高さと固定間隔の分だけ減算される」となる。この理解に基づけば、次のようなより明確なプロンプトを書くことができる。

「自動レイアウトは使わず、公式 Y＝A － B で新しい座標を計算し、吹き出しの位置を直接更新してください。」

5-2. 小結

本章で提示した「What＋How」フレームワークは、非プログラマーが協働において「受動的な受容者」から「能動的な設計者」への転換を支援するものである。

What の数値化は自然言語の曖昧さを解消し、How における能動的な実装構想はロジック構築における人間の主導権を確立する。これにより、AI の過剰設計に起因するメンテナンス困難の問題を効果的に軽減することができる。

また、プロンプトの構成要素を明確化することで、非プログラマーである制作者の開発への関与を深め、プロジェクト全体に対する制御力を高めることが可能となる。

6. 非プログラマーにおける「ケンタウロス」協働モデルの限界と意義

本章では、前章までの開発実践に基づき、非プログラマーと AI による「ケンタウロス」協働モデルが、実際の制作においてどのような有効性を持つのかを客観的に検討する。本モデルは技術的ハードルを低減する一方で、効率や労力の面では顕著な制約を伴う。しかし同時に、個人の創作領域の拡張やスキル習得の補助という点において、代替しがたい価値も有している。

6-1. 限界とコスト

1) 効率の相対性

AI を用いた開発支援は一般に効率向上の手段として捉えられるが、本事例においてはその前提が必ずしも成立しない。コードの査読能力を持つプログラマーにとって、AI はコード補完を高速化する有効な補助となる。一方で、非プログラマーである筆者の場合、開発プロセスは「対話による要件記述 → 手動でのコード適用 → テストとエラー収集 → エラーのフィードバックと修正要求 → 再テスト」という反復ループに依存せざるを得ない。その結果、開発効率は専門プログラマーによる手動コーディングを下回り、将来的に普及が見込まれる人手を介さない Vibe Coding 的な開発形態と比較しても劣るものとなる。

2) 高い認知的負荷と反復作業

本モデルは制作者に高い認知的・精神的負荷を課す。第一に、自然言語による意図を論理的な記述へと変換する作業は、継続的な思考負荷を伴う。第二に、長時間の対話はコンテキストの保持を困難にしやすく、さらに AI はエンジン内の実際のプロジェクト状態を直接参照できないため、存在しないノードの生成、変数関係の混同、既存機能と競合するコードの出力といったハルシネーションが生じやすい。

これらに対処するため、制作者はプロジェクトのノード構造や既存コードといった状態情報を手作業で整理し、対話のたびに繰り返し提示する必要がある。プロジェクトの規模が拡大するにつれて、この状態同期の作業は単調で時間を要するものとなる。

3) 外部依存の不確実性

本モデルは大規模言語モデル（LLM）に強く依存しており、そのバージョン更新はユー

ザー体験に予測困難な変動をもたらす。筆者の事例では、モデルの更新（GPT-4o から後続バージョンへの移行）によりコード生成能力に顕著な低下は見られなかったものの、出力テキストが冗長化し、文脈から意図を読み取る能力が低下する傾向が確認された。

その結果、制作者はより明示的で詳細な指示を行う必要に迫られ、「不要な説明を省く」といった制御指示も機能しにくくなった。さらに、冗長な応答はコンテキスト長を急速に消費するため、対話の分割や再初期化が頻繁に必要となり、その都度プロジェクト状態の再同期が求められる。こうした、使用習慣の変更を強いられ、ツールの更新によってかえって作業負担が増加する状況は、制作者に一定の精神的負担をもたらす。

6-2. 意義

1) 可能性とアクセシビリティの拡大

本モデルの最も重要な意義は、「制作不可能」から「制作可能」への転換を実現した点にある。長期的な技術力の構築には依然として体系的なプログラミング学習が重要であるが、現時点においては、エンジン操作の経験がない制作者であっても、一定の規模と複雑なインタラクションを持つ作品を完成させることが可能となっている。また、既存のチュートリアルへの依存から脱却し、AI との対話を通じて自作品に固有のメカニクスを探索・実装できる点も重要な意義である。

2) 創作主権の回帰

従来のチーム開発の価値を前提とした上で、本モデルは特に個人的な表現欲求が強く実験性の高い作品において、制作者の主導権を強く保持する手段となる。従来、非プログラマーは技術的制約やコミュニケーションの問題により設計の妥協を強いられることが多かった。AI を介することで、制作者は技術的な実装手段を自ら担うことができ、他者との調整を必要とせず、自身のペースと当初の意図に沿って作品を完成させることが可能となる。

3) 実践駆動型のスキル習得

教育的観点において、本モデルは実践駆動型の学習効果を持つ。AI は感情的な負担を与えない指導者として機能し、非プログラマーがコードやエンジンに対して抱く心理的障壁を大きく低減する。制作者は自身のプロジェクトを推進する過程で、実際のエラーへの対処を通じて基礎的なプログラミング概念（変数、条件分岐、シグナル、状態管理など）に繰り

返し接触し、理解を徐々に蓄積していく。この過程で、制作者は複雑な要件を分解・抽象化する計算論的思考を実践的に身につけていく。

自身の制作課題を動機の源とするこうした学習は熱意を維持しやすく、さらに分割開発から統合・エクスポートまでの一連の工程を経験することで、エンジニアリング全体の構造に対する俯瞰的な理解が形成される契機ともなる。

7. 結論

本研究は、筆者が修士課程 1 年次に制作したインディーゲーム『チーズ抜きでお願い』の開発プロセスを事例とし、プログラミングの基礎知識を持たない条件下において、生成 AI を活用してプログラム機能の実装およびエンジニアリング統合をいかに実現したかを記録・分析したものである。

実際の制作プロセスの振り返りを通じて、本研究は非プログラマーに向けた協働フレームワークを提示した。マクロな観点では、「コンテキスト同期と構造策定—実装ループ—統合」という三段階のワークフローによって開発を進める手法を整理した。ミクロな観点では、「What＋How」に基づくプロンプト記述プロトコルを提案し、制作者が数値化・論理化された形式で AI に設計意図を伝達する方法を示した。

本研究の核心的な貢献は、完全なエンジニアリング構造を持つプロジェクトの実践を通じて、非プログラマーの協働における普遍的な課題である「デザイン意図の論理的翻訳」に対し、具体的かつ操作可能な戦略を提示した点にある。具体的には、①意図の数値化、②境界条件の明示的な宣言、③標準的な実装手法が機能しない場合の、意図的な複雑性の低減、の三点である。特に第三の点は、プログラマーにとっては直感的である一方、非プログラマーにとっては「制御性のために規範的な実装をあえて採用しない」という判断自体が大きな障壁となる。これら三つの要素が組み合わさることで、非プログラマーが理解し、保守し、かつ主導権を保持できるワークフローが成立する。

本研究は探索的研究であり、方法論の観点からいくつかの限界を有する。第一に、本研究は単一プロジェクト、特定の AI モデルおよびエンジンバージョンに基づくケーススタディであり、定量的検証を伴っていない。第二に、ツールチェーンの選択において、本研究は主に Web ベースの対話型 AI に依存しており、IDE プラグインやプロジェクトレベルのコン

テキスト共有ツールなど、より高度な開発環境は導入していない。また、既存の協働モデルとの体系的な比較評価も行っていない。したがって、本稿で提示したフレームワークは、能力や環境が制約された状況における実践的な参照モデルとして位置付けられるものであり、最適化された開発手法や標準的なツールチェーンを提示するものではない。

最後に、本研究の執筆期間は、生成 AI 技術が急速に進展する過渡期と重なっている。本プロジェクトの開発当時は、Vibe Coding の概念や高度な自律性を持つ AI エージェント[9]（AI Agent）は、一般のクリエイターに広く普及していたとは言い難い。しかし、執筆後期においては、これらの新たなパラダイムが急速に発展している状況が確認された。将来的に、これらの高度に自動化されたツールが、非プログラマーに対して「開発効率」と「低レイヤーの制御性」を両立したホワイトボックス的な開発体験を提供できるかどうかは、現時点では不明であり、本研究の射程外でもある。このことは、本稿が前提とする「高強度の人為的介入」に基づく開発モデルが、ツールの進化に伴い、近い将来に過度に煩雑なものと見なされる可能性を示唆している。

それにもかかわらず、本稿は、この特定の技術的転換期において、技術的背景を持たない個人がいかにルールを構築し、AI と協働していったかという実践の過程を記録したものである。草の根的な実践事例として、技術的制約に直面した創作者が主導権を獲得しようとする試みを示しており、これこそが本研究の最終的な到達点である。

文献

- カスパロフ, ガルリ(2017)『DEEP THINKING〈ディープ・シンキング〉人工知能の思考を読む』染田屋茂訳、日経 BP。
- Alves, Paolo; Cipriano, Bruno P. (2023) "The Centaur Programmer: How Kasparov's Advanced Chess Spans Over to the Software Development of the Future", 『arXiv』 https://arxiv.org/abs/2304.11172 (2026 年 1 月確認)

<Web 上の記事>

- Karpathy, Andrej (2025) 「I'm mostly just "vibe coding" now...」、『X (旧 Twitter)』 https://x.com/karpathy/status/1886192184808149383 (2026 年 3 月確認)

9 AI エージェント：AI Agent. AI が人間の指示を待たず、目標に基づいて自律的にタスクを計画・実行するシステムの総称。

- ソムタム田井（2025）「『ゲームプログラムの 8〜9 割はすでに AI が作る。必要なのは人間の“審美眼”』LV5 日野社長が若手クリエイターを激励」、『ファミ通.com』https://www.famitsu.com/article/202505/41242（2026 年 1 月確認）
- shioa(2025)「１．はじめに：p5.js × 生成 AI でパックマンを作ろう」、『note』https://note.com/nice_llama936/n/n1f884905fe43（2026 年 1 月確認）
- shioa(2025)「２．非エンジニアでもパックマンを作れる？ 生成 AI×ゲームプログラミングの可能性」、『note』https://note.com/nice_llama936/n/n8788e6589022（2026 年 1 月確認）
- ノトフ/NEIGHBOR CEO(2025)「DreamCore（ドリームコア）ゲーム制作ガイド」、『note』https://note.com/notf/n/n5c7559a80025（2026 年 1 月確認）
- 桃生篤（株式会社 Toumu）(2025)「【AI 活用事例】「『ゲーム開発が劇的に変わる』日本企業 51%が AI 活用中。あなたの会社でも使える 3 つのヒント」」、『note』https://note.com/toumu0208/n/n6309d8c1d4f5（2026 年 1 月確認）
- yuji_yasuhara(2024)「AI を使ったゲーム制作」、『Qiita』https://qiita.com/yuji_yasuhara/items/ba8ffd6d8973eafcc117（2026 年 1 月確認）

動画

- けまいチャンネル(2025)『【マジ驚愕】ChatGPT でヴァンサバライクローグライク開発してみた結果…爆速で弾幕ゲー完成してヤバい…【AI ゲーム制作】』『note』https://www.youtube.com/watch?v=xcpXmf8aqoc（2026 年 1 月確認）
- AI ゲームクリエイターズ(2024)『AI を活用して Unity でゲームプロトタイプを作る！40 代会社員が AI を頼りまくってゲームリリースに挑戦!? #2 AI の指示通りに作ればゲームプロトタイプは作れるのか…』https://www.youtube.com/watch?v=iEF6uh2VC9A（2026 年 1 月確認）
- イラスト 55＠ クリエイティブ情報を紹介(2025)『AI で誰でも爆速ゲーム制作！プログラミング未経験でも作れる本格アクションゲーム【バイブコーディング入門】』https://www.youtube.com/watch?v=oqWLx9CLRGY（2026 年 1 月確認）
- いまにゅの AI プログラミング塾(2025)『【爆速検証】AI だけで 3 つのミニゲーム作れる？v0・Replit・Claude を徹底比較！』https://www.youtube.com/watch?v=uNbcI6YeNoU（2026 年 1 月確認）

- AI ゲームクリエイターズ(2025)『【Unity MCP】AI エージェントが Unity を直接操作！Cursor との連携で開発効率爆上げ～初心者必見の Unity MCP 導入方法～』https://www.youtube.com/watch?v=5i8ryI7E_2E (2026 年 1 月確認)
- 草履虫飼養員(2025)「人，咪在古代支了个摊儿｜AI 经营游戏（猫が古代で屋台を開いたよ｜AI 経営ゲーム)」、『小紅書(レッド)』 http://xhslink.com/o/84CeRz1gcDP (2026 年 1 月確認)
- 人類不只有二极管思維(2025)「用 gemini 做了一个反乌托邦文本交互游戏（Gemini でディストピア風テキストアドベンチャーゲームを作ってみた)」、『小紅書(レッド)』http://xhslink.com/o/59KwSlsbd7E (2026 年 1 月確認)
- 純良二遛子(2025)「我用 Gemini 搓了一个小游戏（Gemini でミニゲームを作ってみた)」、『小紅書(レッド)』http://xhslink.com/o/5AZa57X829H （2026 年 1 月確認)