Slime's Sky Travel

~Game Development Summary~

2025.11.22

日下　拓海

目次

1. 基本情報

ゲーム基本情報

ゲームタイトル : Slime's Sky Travel

コアコンセプト : ゲーム世界の速度を切り替えながら進む空中アクションゲーム

対応プラットフォーム : PC(macOS、Windows)

使用言語 : C++20

制作期間 : 2025.8 - 2025.11

制作人数 : 1 人

ゲーム開発における役割 : 全工程を担当しました

ゲームアピールポイント(3 点)

・ゲーム世界の速度を切り替えることで自分に合った速度感でプレイ可能。速度を速くすると、クリアタイムを縮めることができる一方で操作が難しくなったり、ギミックが速くなったりし、ゲーム性にメリハリが生まれます。

・オンラインランキングに対応したタイムアタックやシークレットスターといったこだわり抜いたゲームモードに加え、UI、ステージ、テクスチャを JSON 化し、ホットリロードに対応させる等、開発効率向上を目的とした機能があります。

・イージーモードによる救済を手厚くすることでアクションが苦手な方にも遊んでいただけるようにしています。また、高難易度モードを導入して上級者にも楽しんでいただけるように設計しています。

開発者情報

開発者名 ： 日下　拓海

所属 ： 弘前大学理工学部電子情報工学科

連絡先 ： takumi.090414528@gmail.com

プレイ動画/紹介動画

https://youtu.be/WSxkKcqc_a4?si=GeOHlOTH5CimOSOS

本作のプレイ映像と、基本ルール、特徴、実装したモード等に関する紹介をまとめた 3 分 40 秒の動画になります。

2. ゲーム概要

基本ルール

プレイヤーはゲーム世界の速度を切り替えながら、各ステージに 3 つ存在しているアイテムを全て集めてゴールを目指します。総ライフ数が 6 であり、落下するごとに 1 つのライフが失われます。全てのライフがなくなった時点でゲームオーバーになります。

コアゲームループ

ステージ選択フィールド上を移動してステージを解放/開始→初期速度の設定→ゲームスタート→空中アクション→アイテムの位置を確認し取得→ゴール/星の獲得→フィールドに戻り次ステージを探索

操作方法

W/A/S/D : 移動

T : 世界の速度変更(1x,2x,3x)

ESC : セーブ/ゲーム終了

ゲームモード一覧

・ノーマルモード

通常のゲームモードになります。できるだけ多くの星を獲得するために最速でのクリアを目指してあらゆるギミックを乗り越えていきます。

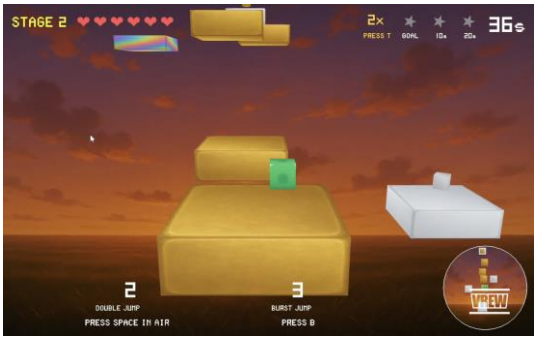
・タイムアタックモード

ステージのクリアタイムを測定/記録できるモードになります。最速クリアタイムの更新を目的にして行います。このモードにはリプレイ機能がついており、自分のプレイを見返して振り返ることが可能です。また、リプレイ中には一時停止、巻き戻し、早送り、映像速度の変更を行うことができるため、自分のプレイ/ルートどりをじっくりと観察/研究することが可能になっております。また、測定したタイム/リプレイはサーバーに送信され、オンラインランキングに掲載されます。ゲーム内掲示板からタイムが早い人のプレイを閲覧することができるため、プレイの研究になります。

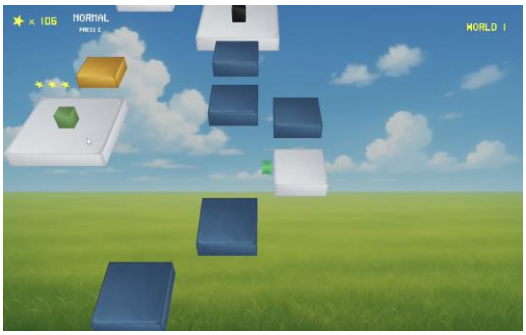
・シークレットスターモード

ゲームクリア後に解放される、上級者向けのコンテンツになります。世界の速度が3x で固定かつ変更不可能な Max Speed Star、世界が暗闇に包まれ、自分の足元しか見ることができなくなる Shadow Star、スライムの目線(FPS 視点)になってステージを攻略する Immersive Star の 3 種類のシークレットスターが存在しています。

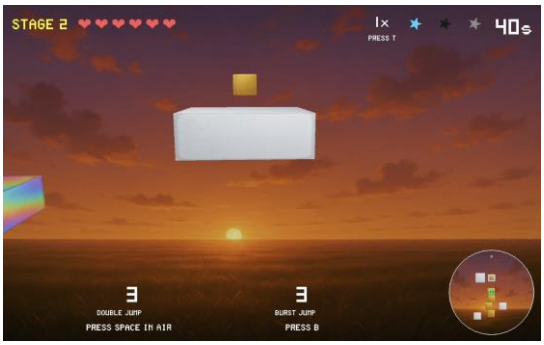
実際のゲーム画像



基本のノーマルモード



ギミックが体験できるステージ選択フィールド



Immersive Star モード

プレイ時間の目安(プレイヤーの習熟により異なります)

1 ステージ ： 1 分～5 分

想定クリア時間 ： 10 分～30 分

想定プレイヤー像

・アクションゲームが苦手な方(イージーモード対応)

・空中アクション/アトラクション系のゲームを好んでいる方

・短い時間でゲームクリアまで楽しみたい方

※詳しいルールや各モードの実際のプレイは、紹介動画で解説しています。

3. 技術概要

使用技術

エンジン/フレームワーク：

・自作エンジン（GLFW + OpenGL）

言語：

・C++20

主なライブラリ：

・GLFW（ウィンドウ管理・入力処理）
・OpenGL（グラフィックス）
・GLM（数学ライブラリ）
・SDL2_mixer（オーディオ）
・nlohmann/json（JSON 処理）
・stb_image（画像処理）

開発環境：

・クロスプラットフォーム対応（Windows / macOS）

ビルド・ツール類：

・CMake（ビルドシステム）

・cppcheck（静的解析ツール）
・Git / GitHub

主なシステム・モジュール

・**GameLoop** ： メインゲームループを管理し、毎フレームの更新・描画順序を制御します。タイトル画面、Ready 画面、ステージクリア画面などの画面遷移も統合管理します。
・**StageManager** ： ステージの読み込み・解放・進行状況を管理します。JSON ファイルからステージデータを読み込み、ステージ間の遷移を処理します。
・**PhysicsSystem** ： 物理演算と衝突判定を担当します。重力方向の変更に対応し、プレイヤーとプラットフォームの衝突判定を重力方向に応じて処理します。
・**ReplayManager** ： タイムアタックモード用のリプレイシステムです。プレイ記録を JSON 形式で保存・読み込みし、クリア後に再生可能にします。
・**GameState** ： ゲーム全体の状態を統合管理する構造体です。プレイヤー、カメラ、アイテム、スキル、進行状況、リプレイ、UI などの各サブシステムの状態を保持します。

設計の方針・意識したこと

・レイヤードアーキテクチャの採用

アプリケーション層、ゲームロジック層、グラフィックス層、物理層、入出力層、コア層に分離しました。上位レイヤーは下位レイヤーにのみ依存し、保守性と拡張

性を確保しています。

・SOLID 原則の遵守
単一責任の原則に基づき、各クラス・関数は 1 つの責任のみを持つことを心がけて設計しています。GameUpdater（更新）、GameRenderer（描画）、InputHandler（入力）などに分離しているため、既存コードを変更せずに新機能を追加できる設計にしています。

・状態管理の分離設計
GameState で各サブシステムの状態（PlayerState、CameraState、ItemState、SkillState 等）を分離しています。状態の変更が他システムに影響しにくく、デバッグとテストが容易になります。

・std::variant による型安全な多態性
プラットフォームシステムで、静的・移動・回転・テレポートなど 9 種類のプラットフォームタイプを std::variant で統合管理しています。仮想関数のオーバーヘッドを避けつつ、型安全性を確保します。

・データ駆動設計によるステージ管理
ステージデータを JSON ファイルで管理しています。コード変更なしでステージの追加・編集が可能になります。JsonStageLoader で読み込み、StageManager で統合管理しています。

・リプレイシステムの実装
タイムアタックモードでプレイ記録をフレーム単位で保存・再生できます。リプレイデータを JSON 形式で保存し、クリア後に再生可能とします。

テスト・品質向上のために使ったツール／工夫

・Cppcheck による静的解析 ： CMake に統合し、未初期化変数や到達不能コードを検出します。
・ 統一されたエラーハンドリングシステム ： ErrorHandler クラスでエラー処理を統一しています。GLFW、レンダラー、ファイル I/O などのエラーを一貫してログ出力し、デバッグを容易にします。
・デバッグ出力の制御機能 ： debug_config.h で ENABLE_DEBUG_OUTPUT フラグを定義しています。開発時は有効、リリース時は無効化でき、パフォーマンスへの影響を最小化しています。
・コマンドライン引数によるデバッグ機能 ： 起動時にステージ番号やエンディングシーケンスを直接指定可能です。テストやデバッグを効率化しています。
・ステージエディタ機能 ： ゲーム内でステージを編集可能にしています。リアルタイムでパラメータを調整し、テストと調整の効率を向上させています。

データ構造・データ駆動

・ステージ構成の JSON 外部化 ： ステージごとのプラットフォーム配置、アイテム位置、ギミック設定を JSON ファイルで定義。ゲーム起動時に JsonStageLoader で読み込み、動的に生成。コード変更なしでステージの追加・編集が可能です。
・セーブデータの JSON 管理 ： プレイヤーの進行状況、アンロック状態、タイムアタック記録などを JSON 形式で保存・読み込み。SaveManager で管理し、データの永続化を実現しています。
・UI 設定の外部化 ： UI 要素の位置、色、サイズなどの設定を ui_config.json で管理してゲーム実行中にファイル変更を検知して自動リロードし、調整を効率化しています。
・リプレイデータの JSON 保存 ： タイムアタックモードのリプレイデータを JSON 形式で保存して、フレーム単位のプレイヤー位置・速度・アイテム収集状態を記録し、再生時に再現しています。

拡張性

・プラットフォームタイプの拡張性

PlatformSystem で std::variant とテンプレート関数を使用。新しいプラットフォームタイプを追加する際は、構造体を定義して PlatformVariant に追加するだけで対応可能です。

・スキルシステムの拡張性

SkillState でスキルを管理。新しいスキルを追加する際は、SkillState にメンバ変数を追加するだけで対応可能です。

・レイヤードアーキテクチャによる拡張性

各レイヤーが独立しているため、新機能を追加する際は、新しいレイヤーまたは既存レイヤー内に新システムを追加するだけで対応可能です。

4. 開発プロセス

開発プロセスの概要(タイムライン)

2025.08-09：コア実装・全ステージ作成

2025.11：リファクタリング・追加機能開発・提出準備

開発開始時、夏季休暇だったため週 50 時間以上を目処に作業を行っていました。

開発体制・担当範囲

開発体制 ： 個人開発

担当範囲 ： 企画、設計、実装、デバッグなど全ての工程を担当しました。(Github にコラボレーターがいますがそちらは WindowsPC を借りて、自分で作業をしたものになります。)

ワークフロー・ツール

・バージョン管理 ： Git/Github を用いました。master ブランチとテスト用の stg ブランチで開発を行い、機能ごとに細かくコミットすることを心がけておりました。

・タスク管理 ： Notion を用いました。現在のタスクを優先度で振り分けてその順に実装を進めたり、開発における課題とその解決について都度まとめたり、開発スケジュールの管理を行ったりしていました。

開発における課題とその解決

開発中に直面した課題とその解決について、課題→解決→効果→学びという 4 つの観点から紹介します。

技術的課題①：ハードコードと再起動に依存したワークフロー

問題 ： ステージや UI を調整するたびにコードを書き換え、ビルド＆ゲーム再起動を行っていました。ステージ構成はコード／ファイル固定、UI 座標もハードコードだったため、細かい調整に時間がかかり、試行回数が稼げない状態でした。

解決 ： ステージ構成・テクスチャ・サウンド・UI レイアウトなどを JSON で定義するデータ駆動構成に変更しました。また、ゲーム起動中に JSON ファイルの更新を監視し、保存されたタイミングでホットリロードして即座に反映する仕組みを実装しました。

・ステージ：ステージ管理用 JSON を編集 → 保存 → 起動中のゲーム内ステージに即反映

・UI：UI 座標・サイズ等を JSON から読み込み → 同様にホットリロード対応

効果 ： ステージのギミック配置や UI の位置調整を再起動なしで連続して試せるようになり、細かいチューニングにかかる時間を大幅に削減できました。その結果、

ステージ・UI の試行回数が増え、全体の完成度向上にもつながりました。

学び ： ゲーム開発では、コンテンツをハードコードせずデータ駆動にし、編集→確認サイクルを短くすることが生産性を大きく上げると実感しました。将来的にデザイナーやレベルデザイナーと協業する際にも、JSON などの外部データで制御できる構成は有効だと感じました。

技術的課題②：リプレイ再生の滑らかさとデータ量の両立

問題 ： タイムアタックモードでリプレイ機能を実装する際、毎フレーム記録するとデータ量が増え、記録間隔を粗くすると再生時にカクつきが発生しました。60FPS で記録すると 30 秒のプレイで約 1,800 フレームとなり、ファイルサイズが大きくなります。一方、0.1 秒間隔（10FPS）で記録すると容量は抑えられますが、そのまま再生すると動きが不自然に見えます。

解決 ： タイムスタンプベースの線形補間システムを実装しました。記録は 0.1 秒間隔で行い、再生時に 2 フレーム間の位置・速度を補間します。具体的には、再生時刻が 2 フレーム間にある場合、補間係数 t を計算し、glm::mix で線形補間します。位置・速度は連続値として補間し、アイテム取得状態は離散値として中間点で切り替えます。また、巻き戻し・早送りにも対応し、再生速度を変更しても同じ補間アルゴリズムで動作します。

効果 ： データ量を約 6 分の 1 に削減しつつ、60FPS で滑らかに再生できるようになりました。記録は粗くても、補間によりユーザーは違いを感じません。さらに、任意の再生速度に対応し、リプレイの視認性が向上しました。
学び ： データ量と品質のバランスを取る設計の重要性を実感しました。タイムスタンプベースの設計により、フレームレートに依存しない再生が可能になり、将来のフレームレート変更にも対応できます。連続値と離散値を適切に扱うことで、補間の精度と実装の簡潔さを両立できました。

技術的課題③：タイムアタックのオンラインランキング導入によるゲームの単調さの改善

問題 ： タイムアタックモードを実装したものの、ローカルでの自己記録更新だけでは、何度も同じステージをプレイする動機が弱く、ゲームが単調になりがちでした。プレイヤーは自分のベストタイムを更新しても、他のプレイヤーとの比較ができないため、モチベーションが維持しにくい状況でした。また、上位プレイヤーの

プレイスタイルを学ぶ機会もなく、技術向上のきっかけが限られていました。

解決 ： オンラインランキングシステムを実装しました。libcurl を使用して HTTP 通信を行い、バックエンド API と非同期でデータを送受信します。具体的には、ステージクリア時にタイム記録とリプレイデータを自動送信し、ステージ選択画面のランキングボードから他プレイヤーの記録を閲覧できるようにしました。ネットワークエラー時は自動リトライ機能を実装し、最大 5 回まで 2 秒間隔で再試行します。

効果 ： オンラインランキング機能により、プレイヤー間の競争が生まれ、ゲームのリプレイ性が大幅に向上しました。自分の順位を確認し、上位プレイヤーのタイムを目標にすることで、継続的なプレイの動機が生まれました。また、リプレイ機能により、上位プレイヤーのプレイスタイルを学習でき、技術向上にもつながりました。結果として、タイムアタックモードが単なる自己記録更新から、コミュニティ全体での競争へと発展し、ゲームの単調さが解消されました。

学び ： ゲームのリプレイ性を高めるには、単なる機能追加だけでなく、プレイヤー間の競争や学習機会を提供することが重要だと実感しました。非同期通信の実装により、UI の応答性を保ちながらネットワーク処理を行う設計の重要性を学びました。また、エラーハンドリングとリトライ機能の実装により、ネットワーク環境が不安定な場合でもユーザー体験を損なわない設計の必要性を理解しました。オンライン機能は、ゲームの単調さを解消し、長期的なエンゲージメントを生み出す重要な要素であることを実感しました。

5. まとめ

本作品で達成できたこと

・ゲーム世界の速度を切り替えながら進む空中アクションというコンセプトを、ノーマル／タイムアタック／シークレットスターといった複数モードに落とし込み、PC（macOS／Windows）向けのプレイアブルなタイトルとして完成させました。
・C++20 と自作のレイヤードアーキテクチャを用いて、ゲームループ・ステージ／UI・リプレイ・オンラインランキングなど、ゲームプレイ全体を一人で設計・実装しました。
・UI／ステージ／テクスチャ／サウンドの JSON データ化とホットリロード、Cppcheck やエラーハンドリング／デバッグ出力の仕組みなどを整えることで、個人

開発でも効率よく反復できる開発環境を構築しました。

本作品を開発中に学んだこと

・コンテンツをハードコードせず JSON などの外部データで管理し、ホットリロード可能にすることで、ステージや UI の調整サイクルが大幅に短縮され、ゲームの完成度向上に直結することを実感しました。
・レイヤードアーキテクチャや状態管理の分離、SOLID 原則を意識したクラス設計により、機能追加やリファクタリングが行いやすくなり、拡張性と保守性を両立できると学びました。
・リプレイ機能の実装を通じて、タイムスタンプ＋補間による「データ量と再生品質のバランスを取る設計」や、フレームレートに依存しない仕組み作りの重要性を理解しました。
・イージーモードやシークレットスターモードの設計を通じて、アクションが苦手なプレイヤーから上級者まで、幅広い層が楽しめる難易度設計の大切さを学びました。

今後の展望

・ステージ数やギミックのバリエーションを拡充し、特に演出を強化することで、ゲームとしての体験をさらに厚くしていきたいです。
・レベルエディタなど開発用ツールを整備し、JSON＋ホットリロードの仕組みを発展させることで、将来的にはレベルデザイナーなど他職種とも協業しやすい環境を目指したいです。
・オンラインランキングやリプレイ共有などネットワーク周りの堅牢化を進めつつ、今後の機能追加やプラットフォーム拡張も見据えてアーキテクチャを継続的にブラッシュアップしていきたいです。