眺めるローグライク

吉田月輝

自己紹介

- 名前
  吉田月輝（るなき）
- 出身大学
  京都産業大学外国語学部
- 年齢
  21歳(2003/04/22)
- 得意な言語
  C#
- 好きな一言

桜井政博さん
「とにかくやれ！」

好きなゲーム(好きな理由)

・スマブラ→努力が反映される

・音ゲー→失敗は全部自分のせい

・TD→同じ目標でも様々な方法がある

技術スキル

- 使用言語
  C#(UniRxを使ったリアクティブ設計経験あり)

- 使用エンジン
  Unity

- 開発ツール
  VS Code/GitHub/SourceTree(一年間の個人開発で活用)

制作実績

- タイトル:「**Dimension Core**」
- ジャンル: ローグライクストラテジー
- 制作期間: 1ヶ月半
- 使用言語/エンジン: C#/Unity
- プラットフォーム: IOS/Android
- 制作人数: 一人
- 意識したこと: 単一責任の原則、命名規則

なぜDimension Coreを作ったか

- ローグライクストラテジーを作りたい
- しかしローグライクやストラテジーは複雑で初心者には敷居が高い
- そこで「できる限りシンプルなローグライクストラテジー」を作ろうと考えた

企画の工夫

- キャラクターは図形

図形はこの世で最もシンプルなオブジェクトだと考えた

- 宇宙のイメージ

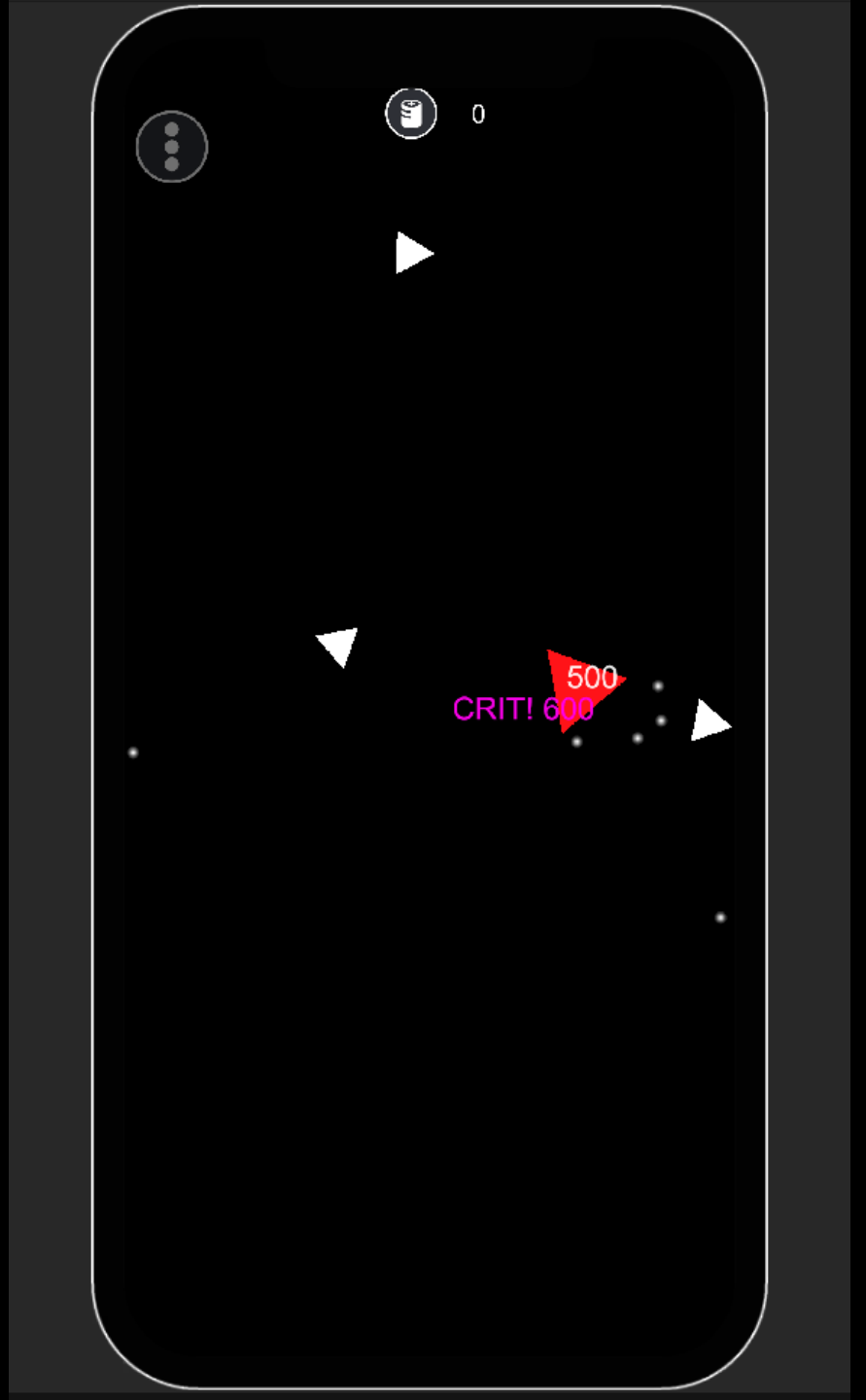
図形は宇宙空間を漂う星のイメージ。図形が自動で漂い、衝突することで戦闘を行う

- 世界はモノクロに

視認性を保ちつつ、宇宙を演出

Dimension Coreのコンセプト

- Dimension Coreは「眺める」ローグライク
- 操作せず、漂う図形を見守る
- 複雑になりがちな戦闘を「眺める」だけにすることでシンプルに



Github & プレイ動画

- GitHub

- YouTube

開発時の工夫①
UniRxによる定期処理の改善

- 目標
持続ダメージや持続回復のような、「一定時間ごとに処理を行う」実装がしたい。

- 課題
従来のUpdate()ベースの実装では

1タイマー変数を加算、リセットする必要がある
2条件分岐が多く、保守性が低い
3終了処理や破棄時の処理が複雑

開発時の工夫①
UniRxによる定期処理の改善

```
// ================================
// 継続回復
// ================================
public void TakeHealOverTime(float healValue, float duration, float interval)
{
    // 一定間隔で回復
    Observable.Interval(System.TimeSpan.FromSeconds(interval))
        // 継続時間が0になったら終了
        .TakeWhile(_ => duration > 0)
        .Subscribe(_ =>
        {   // HPが満タンでない場合
            if (stats.CurrentHP.Value < stats.MaxHP)
            {   // 回復
                TakeHeal(healValue);
            }
            // 残り時間を減らす
            duration -= interval;
        })
        .AddTo(this);
}
```

より宣言的・安全・簡潔な記述が可能なUniRxを採用

Interval:一定時間ごとの実行を簡潔に

TakeWhile:終了条件を記述し安全にループ

AddTo(this) GameObjectの破棄に応じて自動でDispose(メモリリーク防止)

開発時の工夫①
UniRxによる定期処理の改善

項目	Before(Update方式)	After(UniRx方式)
可読性	タイマー制御が複雑	1行で明確に記述
保守性	条件分岐・変数が多い	条件も処理も一箇所に集約
安全性	破棄時の処理忘れあり	AddToで自動破棄管理
再利用性	処理の使い回しは難しい	Observableで再利用しやすい

- 学び

リアクティブ設計はUIだけでなく、ゲームロジックにも効果的。
今後はUniTaskやStateパターンなども活用し、より柔軟なゲーム設計を目指す。

開発時の工夫②
単一責任の原則(SRP)によるクラス分割

- 目標

戦闘時に登場する図形クラスに

・ステータス管理
・アニメーション制御
・移動処理
・衝突判定
・戦闘処理
・エフェクトの発生

を実装する

- 課題

初期は全ての機能を1つのスクリプトにまとめていたため、保守性が低く、バグも発生しやすかった

コードは400行を越えており可読性が低く、どこに何の処理があるか分かりにくい状態だった

開発時の工夫②
単一責任の原則(SRP)によるクラス分割

- 改善内容:機能ごとにクラスを明確に分離

機能	担当クラス	説明
図形本体	Shape	死亡処理や各ハンドラの初期化など
戦闘中のステータス管理	ShapeBattleStats	HPや攻撃力などのデータ保持
アニメーション制御	ShapeAnimationHandler	DOTweenでのアニメーション
移動制御	ShapeMovementHandler	物理計算による自動移動制御
衝突制御	ShapeCollisionHandler	衝突時のイベント処理
戦闘処理	ShapeCombatHandler	攻撃や回復の管理
エフェクト処理	ShapeEffectHandler	衝突エフェクトの生成管理

- 結果

改善前は400行以上あったShapeクラスが、改善後は100行程度まで減った

今後の目標

- 拡張性と保守性の高いシステム設計を実現するために、SOLID原則やデザインパターンを深く学び、コンポーネントの適切な分割と依存関係の最適化を実践する。

- イベント駆動設計やDI（依存性注入）を活用し、柔軟で再利用可能なアーキテクチャを構築することで、チーム開発の効率化に貢献する。