Spark project

—Flash によるゲーム開発を支援するフレームワーク Xelf—

1　背景

近年、Flash のリッチインターネットアプリケーション（RIA）や、特にゲーム開発での利用が増えてきている。しかしながら、ActionScript には、ゲームライブラリやゲームフレームワークと呼ばれるものは存在せず、Flash ゲーム開発者は全てを自分で作り上げなければならない。グラフィック関連が充実している Flash とはいえ、これはかなりの労を要する。そのため、折角ゲーム開発をしようと思い立っても、初心者にとっては、そもそも開発にあたって何をすれば良いのか分からず、敷居が高いという問題が存在する。また、初心者ではない人にとっても、毎回、ゲームのオリジナリティとは関係ない部分の開発で時間がかかってしまい、効率が悪いという問題が存在する。

2　目的

背景で述べたような、Flash ゲーム開発に存在する問題を解決するために、ゲーム開発に共通で必要な処理をライブラリとしてまとめたフレームワーク「Xelf」を開発するのが本プロジェクトの目的である。Xelf によって、ゲーム開発の効率の向上と敷居の低減を図る。

ところで、他言語では数多くのゲームライブラリが存在するが、その多くが「見た目」の部分に重点が置いる。ゲームの構造や、ロジックをどう管理するのか、といった部分については、あまり考えられておらず、開発者任せである。そのため、実際のゲーム開発でこれらのライブラリを使用しても、グラフィック以外の部分で、開発者が自らカバーしなければならない様々な問題が浮かび上がってくる。これではゲーム開発が楽になったとは言えない。

Xelf では、このような問題を無くすため、ゲーム全体の設計のサポート、記述するコード量の削減、フレームワークへの依存性の低減といったことも考慮し、実用に耐えうるフレームワークを開発する。

3　開発内容

ゲーム内では、「ユーザーからの入力」や「アイテムをゲットした」など、様々なイベントが発生する。従来のゲーム開発では、これらのイベントが発生しているかどうかを逐一チェックして処理を行うといったコードを書くことが多かった。しか

し、段々と規模が大きくなってくると、イベントチェックのための条件式や条件分岐、そして実際の処理が混在し、見通しが非常に悪くなるという問題が存在した。そこで Xelf ではイベント駆動による実行方式を採用し、これらの処理を簡潔かつ直感的に記述できるようにした。

従来のゲーム開発では、上記のような問題に加えて、サウンドや描画、スコア処理といったような、性質は異なりながらも関連する処理がひとつのクラスやメソッドに集中してしまうことが多く、責務が大きくなり、保守や再利用が困難になるという問題が存在した。そこで Xelf では、ゲームの構成要素を「キャラクタ」「インスタンス」「タスク」の 3 つに切り分け、性質の異なる処理や責務を適切に分離して記述できるようにした。

「キャラクタ」とは敵やプレイヤー、弾やスコアといったように、画面上に登場する物から抽象的なものも含め、ゲームを構成する大まかな要素のことを指す。キャラクタがイベントに対してどのような処理を行うか、ということを実際に記述するための要素が「タスク」である。キャラクタは複数のタスクを持つ事ができ、移動の処理を行うタスク、描画の処理を行うタスク、スコアの処理を行うタスク、といったように、処理の性質ごとに異なるタスクを作成して組み合わせることが可能である。これにより、タスクごとの責務が明確になり、見通しが良く、再利用もしやすくなるメリットが生まれる。一部の処理のみが違うような敵を沢山作る場合、継承やコンポジションを使わずとも、その処理を担当するタスクだけを入れ替え、後は全て同じタスクを使いまわすことで対処できる（図 1)。

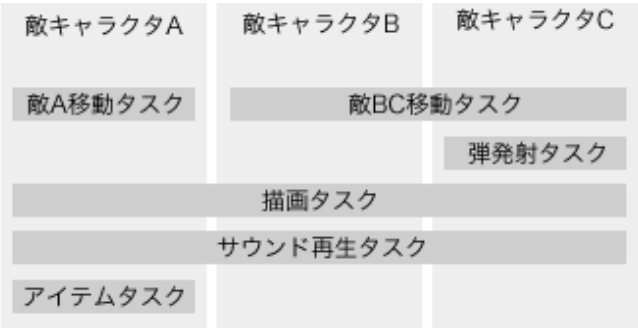

図 1: タスク組み合わせの例

しかしここで、これらのオブジェクトをどのようにして管理するのか、という問題や、オブジェクト間の依存関係はどうするのか、という問題が新たに浮上する。この問題の解決を開発者に任せてはフレームワークの価値が薄れてしまう。そこで Xelf では「Dependency Injection（DI）コンテナ」をオブジェクト管理に採用し、オブジェクトの管理や依存関係に関するコードを開発者が 1 行も書かなくて済むようにした。

記述性や拡張性、利便性などを考慮し、フレームワークへの設定は全て統一的に

XML で記述可能にした。ActionScript では E4X が採用されているので、XML との親和性が高い事も理由のひとつである。

ただ、設定自体が複雑になってしまっては、やはりフレームワークの価値が薄れてしまう。そこで Xelf では、「Convention over Configuration」（CoC）を採用し、規約に沿った開発をしていればほとんど設定を記述しなくていいようにした。例えばタスクは、イベントが起きた際に処理を行うためのメソッドを登録しなければならなが、これは単純作業でコード量も増えてしまう。そこで、「on + イベントのタイプ」又は「イベントのタイプ + Handler」という名前で、引数なし或いは Message 型の引数をひとつ取り、戻り値が void であるメソッドを宣言すると、自動で登録を行うという CoC が用意されている（図 2）。

```
public class SampleTask
{
    // m.addMessageListener('hoge', onHoge); が自動で行われる
    public function onHoge():void
    {
    }
    // m.addMessageListener('fuga', fugaHandler); が自動で行われる
    public function fugaHandler():void
    {
    }
}
```

図 2: メッセージハンドラ自動登録の例

ゲーム開発において意外と面倒なのがシーンの制御である。シーンとは、タイトル画面やゲーム画面、ゲームオーバー画面やランキング画面といったように、ゲーム中に出現する場面のことを指す。ゲームでは、これらのシーン間を遷移することによってもゲームが進行する。しかし、このようなシーン内に登場するオブジェクトをどのように管理して、切り替えるのか、という問題が存在した。そこで Xelf では、シーン制御機構を導入し、シーン単位での実行を可能にした。

4　従来の技術との相違

Xelf は、グラフィックやサウンド関連の機能に特化した一般的なゲームライブラリとは違い、ゲーム全体の設計や構造についても考慮して開発されたフレームワークである。キャラクタやタスクを用いた設計方針で、「とりあえずタスクを作ればゲームが動く」という道筋を示すことにより、ゲーム開発の初期段階で挫折してきた初心者でも使いやすいフレームワークになっている。

勿論、Xelf は初心者だけでなく、良くゲーム開発をしている開発者もターゲットにしており、既存のゲーム開発に良く存在した、コードの複雑化、責務の集中、面

倒なオブジェクト管理やシーン遷移といった様々な問題に取り組み、解決を図っている。独自のアーキテクチャや、DI などの技術を取り入れることにより、簡潔かつ直感的にゲーム開発が行えるようになっている。開発者は基本的にタスクの作成のみに集中すればよく、タスクを開発していくだけでゲームが動作するようになっている。また、推奨する設計方針に従って開発を進めていれば、自然と他のオブジェクトへの依存度が低く、再利用がしやすいクラスやタスクが作成できるようになっている。

フレームワークを利用すると必ずといって良いほど直面する問題が、フレームワークに関するコードや設定の肥大化である。特に、XML による設定その傾向が強い。Xelf では、DI や CoC などを取り入れることで、フレームワークを利用していながらも、フレームワークに関連するコードや設定を殆ど書かずに、最低限で済むようになっている。これはフレームワークへの依存度が低いということにも繋がり、開発が行いやすくなっている。

5　期待される効果

Xelf の登場によって、Flash によるゲーム開発の敷居が大幅に低減し、多くの人がゲーム開発を行いやすくなることが第一に期待される。また、敷居が低減するだけでなく、Xelf が目指すコードや設定の軽量化によって、従来に比べて開発効率や保守性が向上することも期待される。

6　普及の見通し

Xelf は現在、開発版を配布しており、オープンソースで開発を続けている。そこで、多くの人からフィードバックを得つつ、更なる機能強化や利便性の追求を行っていく。

Xelf は、初心者だけでなく、全ての Flash ゲーム開発者をターゲットとしたものであり、多くの人が利用し、恩恵を受けることの出来るフレームワークになる予定である。

7　開発者名

- 新藤　愛大 (立教池袋高等学校)

(参考) 開発者 URL

- http://www.libspark.org/