東方花映塚 AI 自作ツール『花 AI 塚』の開発とその前後

社会 S1 @ide_an
http://www.usamimi.info/~ide/

概要

対戦型シューティングゲーム『東方花映塚』の AI を作成できるツール『花 AI 塚』を開発し、公開しました。花 AI 塚では Lua で AI を記述して動かすことができます。本稿では花 AI 塚の開発に至る経緯や花 AI 塚で用いられている技術、そして花 AI 塚の今後について述べていきます。

1 はじめに

1.1 花 AI 塚とは?

花 AI 塚[*1]は対戦型シューティングゲーム『東方花映塚』[*2]で自作の AI を動かすツールです。AI は Lua[*3]のスクリプトとして記述し、自作 AI 同士で戦わせることもできます。東方花映塚の製品版 ver 1.00 と ver 1.50a の両方に対応しています。

1.2 開発動機

根本的な動機としては技術的自己満足と承認欲求がありますが、他にもいくつか動機があります。

花映塚を再興したい

花映塚は 9 年前のゲームですが、それだけの年月を経てもなお魅力のあるゲームです。私は花 AI 塚を通して、AI を作り AI 同士で戦わせるという新たな花映塚の楽しみ方を提示し、プレイヤーコミュニティに改めて花映塚へ注目を集めさせることができないかと考えました。そしてあわよくば花映塚続編への期待感を生み出し、神主が花映塚 2 に着手するようにならないかと・・・。

STG AI プラットフォームを構築したい

東方におけるシューティングゲーム AI(STG AI) ツールの事例としては紅魔郷 AI[1] やダブルスポイラー AI[2]、地霊殿 AI[3] などがありますが、これらはいずれも AI のアルゴリズムをハードコーディングしており自由に AI を書き換えることができませんでした。東方以外の STG AI の事例としては『白い弾幕くん』[4] があります。このゲームも AI はハードコーディングされており、AI を改造するためにはソースのリビルドが必要となります[*4]。

このようにこれまでの STG AI はハードコーディングされたものが多く、自由に AI アルゴリズムを書き換え、気軽に動かして試せる環境は見受けられませんでした。AI を作りたい人が AI 以外のコードに囚われることなく AI を作るためには、こういった環境があるべきです。

そうして作られたのが花 AI 塚です。

1.3 なぜ花映塚?

私が今回花映塚を対象にしたのは花映塚が他の東方作品と違って『対戦型』のシューティングゲームだからです。このことは AI プラットフォームを考えた場合に以下の魅力があります。

- AI の楽しみ方が多様になる
- 他の人と競う楽しみがある

他の東方作品のように 1 人のプレイヤーがシステムの用意した敵と戦い続けるタイプのゲームの場合、このシステムの上で AI を作っても、その AI のプレイを見て楽しむという楽しみ方しかできませんでした。しかし、対戦型のゲームの場合 AI とシステムが用意した敵と対戦する以外にも、AI と自分との対戦、AI と他の人との対戦、AI と他の人が作った AI との対戦といったように楽しみ方の幅が広がります。特に最後の「AI と他の人が作った AI との対戦」は AI 作成者同士で競う楽しみがあり、AI 作成者同士の切磋琢磨によってより良い AI を生み出す可能性があります。

2 花 AI 塚前史

ここでは花 AI 塚開発に至るまでの経緯を述べていきます。私が東方 AI に手を染めるに至ったそもそもの発端から振り返っていこうと思います。

2.1 東方 AI 同人誌との遭遇

2011 年 12 月、卒論の最中に C81 に赴いた私は「AI に『ダブルスポイラー』をプレイさせる本」[2] という同人誌に出会いました。この本ではスクリーンキャプチャからゲーム状態を推定して動く AI の開発と性能評価を行っており、この本に感化されて私も東方をプレイする AI を作りたいと思うようになりました。

---

*1 http://www.usamimi.info/~ide/programe/touhouai/
*2 http://www16.big.or.jp/~zun/html/th09top.html
*3 http://www.lua.org/
*4 ソースが公開されているため、リビルドは可能

2.2 地霊殿 AI 開発

私が初めてプレイした東方作品は東方地霊殿で、それゆえこのゲームには思い入れがありました。東方 AI のターゲットを選ぶ際にもその思い入れが働き、地霊殿の AI を作ることに決めました。

先述の同人誌ではスクリーンキャプチャからゲーム状態を推定していましたが、私は違うアプローチをとりました。私のアプローチは実行中のゲームプログラムのメモリからゲーム状態を読み出すというもので、これを実現するにはゲーム状態がメモリ上にどのように配置され、どのように使われるかを知る必要がありました。2012 年 1 月と 3 月にこれらの解析を行い、3 月から 4 月の頭にかけてこの解析結果に基いて AI を実装しました。この AI を新勧展示で展示し、その年の前期研究報告会で発表しました [3]。

2.3 花映塚 AI に向けて動き始める

時は流れて 2013 年 9 月、地霊殿 AI の改良として DLL インジェクションを試してみました。以前の地霊殿 AI は別プロセスから地霊殿プロセスのメモリを読む実装だったのですが、これには実行速度が遅いという問題があり、解決策として DLL インジェクションが使えるのではと考え実装しました。これによって高速化に成功したのですが、このとき私は AI をスクリプトに置き換えても十分な実行速度が得られるのではと考えました。ここから花 AI 塚のアイディアを思いつき、これを実現できるか検証すべく 11 月に地霊殿 AI への Lua 組込みを行いました。そしてこの 2 つの試みについて 2013 年の後期研究報告会で発表しました [5]。

3 開発概要

本節以降では花 AI 塚の開発および技術的側面について見ていきます。

3.1 花 AI 塚の仕組み

花 AI 塚は対戦の間、以下の処理を毎フレーム行うことで AI を動作させています。

1. 花映塚プロセスのメモリを読み出して、ゲーム状態を取得する
2. Lua 側に提供しているグローバル変数を更新する。このグローバル変数からゲーム状態が得られる
3. Lua スクリプトで定義した main 関数を呼び出す。このスクリプトではそのフレームで送信するキー入力を決めることができる
4. Lua スクリプトでの決定に基いてキー入力を花映塚プロセスに反映する

1 を行うためにはゲーム状態がメモリ上のどこに配置され、どのように表現されているかを知る必要があります。そのため開発にあたっては、まず花映塚の解析を行うこととなります。

また、花映塚の対戦の開始や終了の検知も毎フレーム行っています。対戦の開始時に Lua スクリプトのロードを行い、終了時にアンロードを行っています。

花映塚プロセスのメモリを読むためのアプローチとしては以下の 2 種類があります。

- 花映塚プロセスとは別のプロセスとしてプログラムを動かし、Win32API の ReadProcessMemory 関数を用いてメモリを読む
- 花映塚プロセスと同一のプロセスとしてプログラムを動かし、通常のポインタ操作と同様にしてメモリを読む

前者は技術的難易度が低いですが、ReadProcessMemory 関数のオーバーヘッドが大きく処理落ちを起こしやすいという問題があります。その点後者はメモリ読み出しのオーバーヘッドが小さく実行速度が速いため、花 AI 塚では後者のアプローチを取っています。

3.2 スケジュール

花 AI 塚の開発スケジュールを表 1 に示します。

解析に関しては実装に踏み切れるまでの解析が終わったのは 3 月 2 日ごろですが、実装開始後も詳細を知りたい場合などに解析を行うこともありました。そのため実質的には解析には 2 ヶ月程度かかっています。

実装に関しては本当は 1 週間早く ver 1.0 を公開する予定だったのですが、Lua のバインディングまわりの見直しによってずれ込んだという事情がありました。

4 解析

解析にあたっては以下のツールを使用しました。

- うさみみハリケーン[*5](プロセスメモリデバッガ)
- IDA Pro Free[*6](逆アセンブラ)
- OllyDbg[*7](デバッガ)

プロセスメモリデバッガは主に解析の初期において、メモリ上の値の変化を元にデータのアドレスを特定する目的で使うことが多かったです。解析を進めていくと逆アセンブラとデバッガを主に使うようになります。IDA Pro は逆アセンブルしたプログラムの制御構造をグラフ

[*5] http://www.vector.co.jp/soft/win95/prog/se375830.html
[*6] https://www.hex-rays.com/products/ida/
[*7] http://www.ollydbg.de/

日付	開発関連	その他
1 月 25 日	花映塚解析開始	
1 月 27 日		修論提出
2 月 6 日		修論発表
2 月 26 日	オブジェクト関連ほぼ解析完了	
3 月 2 日	対戦開始・終了検知方法確立	
3 月 14 日	花 AI 塚実装開始	
4 月 1 日		就職
4 月 25 日	ver 1.0 公開	
4 月 27 日		例大祭 11 チェックツール公開
5 月 5 日	ver 1.1 公開	

表 1　花 AI 塚開発スケジュール (いずれも 2014 年)

として表現してくれるなど様々な強力な機能を備えています。

花 AI 塚において解析を行う目的はゲーム状態を取得することです。AI を作る場合にはゲーム状態として、自機や弾、敵など画面に表示される各種オブジェクトの情報が必要となります。また、AI が生成した自機操作をゲームに反映するためにプレイヤーの入力の格納の仕方を知る必要があります。花 AI 塚で取得しているゲーム状態は表 2 の通りです。

また、実装においては対戦の開始時 (あるいは終了時) だけ実行されるコードを知る必要があります。この情報は対戦開始や終了を検知するために使います。これに関しては自機オブジェクトの生成や削除に関するコードを追うことで見つけることができました。

4.1　自機座標の特定

表 2 で挙げたように、AI を作るためにはさまざまなオブジェクトに関する情報が必要となります。弾や敵など、自機以外のオブジェクトは 1P,2P それぞれに複数個存在し得ます。これはすなわち複数個のオブジェクトを管理するなんらかのデータ構造 (コレクション) があるということです。コレクションの解析にいきなり挑むのは難しいです。なのでまず自機に関する情報を解析し、それを手がかりに各種コレクションの解析を進めていくというアプローチを取りました。

自機座標は自機の情報の中でもプレイヤーの入力による制御がしやすく、なおかつコレクションの解析での手

*8 弾幕を構成する際に、不可視な敵を用意し、その敵に弾配置をさせることがある。このような敵を「擬似的な敵」と呼んでいる。妖夢の C2 などが該当。

自機	位置、当たり判定、自機キャラクター、ライフ、カードアタックレベル、ボスカードアタックレベル、チャージ現在値、チャージ最大値、チャージ速度、移動速度、スペルポイント、神主 AI か?
弾 (通常弾)	位置、速度、当たり判定、有効か?、爆風で消せる弾か?
弾 (レーザー)	位置、速度、当たり判定、有効か?
弾 (EX アタック)	位置、速度、当たり判定、弾種別 (主にどのキャラクターの EX かでの区別)
敵	位置、速度、当たり判定、有効か?、ボスか?、幽霊か?、活性化したか?、リリーか?、擬似的な敵か?[*8]
アイテム	位置、速度、当たり判定、有効か?、アイテム種別
その他	スコア、ラウンド数、勝利ラウンド数、難易度、リプレイ再生か?、プレイヤー入力、チャージタイプ

表 2　ゲーム状態として取得する情報

がかりになるため、まず自機座標を取得する方法を確立することから始めました。

大まかには以下の手順で進めていきました。

1. プロセスメモリデバッガを用いてある対戦における自機座標アドレスを特定する
2. 上記で求めた自機座標アドレスにウォッチポイントを設定し、このアドレスを参照するアセンブリコードを洗い出す
3. 洗いだしたアセンブリコードの中からわかりやすそうなコードを見つけ、そのコードの呼び出し元へ遡るようにコードを下から上に読んでいく
4. どこかしらで自機座標を間接的に参照できるグローバル変数が見つかるので、そのグローバル変数の配置されているアドレスを調べる

この手順では自機データはヒープ領域に配置されていて、ポインタ型のグローバル変数が自機データの配置されたアドレスを指しているという仮定を置いています。これはつまり `Player*`型のグローバル変数 `hoge` に

```
hoge = new Player();
```

のように代入を行っていて、`hoge` のアドレスと `Player` クラスのインスタンスにおける自機座標の相対アドレ

スを調べることで自機座標の絶対アドレスが分かるはずだ、ということです。

手順 1 でやることをもう少し細かく見ていきましょう。自機の座標はプレイヤーの操作にしたがって変化します。たとえばプレイヤーが右移動の操作をすれば自機の X 座標は増加するはずです。メモリ上の値の変化が起きた場所を検索する方法がプロセスメモリデバッガには用意されています。これを用いてプレイヤーの操作に合わせて変化しているメモリを絞り込むことで、自機座標のアドレスを見つけることができます。ただし、このようにして見つかるアドレスは、対戦のたびに変化してしまいます[*9]。一方、グローバル変数 hoge 自体のアドレスは変わらないので、手順 2 以降ではこの hoge のアドレスを探しています。

手順 3 ではコードの呼び出し元を遡って行くとありますが、コードの呼び出し元はアセンブリコードを読むだけで分かるとは限りません。そのためデバッガを使って実行時に実際の呼び出し元を調べます。こうして遡るうちにタスクシステムと思しきものが見つかります。自機座標の更新処理などはタスクシステムにおける 1 つのタスクとして扱われていて、このタスクを表すオブジェクトが自機データへのポインタを持っていました。このタスクオブジェクトがどこで生成されているかを調べ、そのコードの周辺を調べた結果、自機データへのポインタを持つグローバル変数を見つけました。こうして自機座標を取得する方法を確立することができました。

4.2 謎のコレクション

自機座標の取得法が確立されたので、これを手がかりに他のオブジェクトのコレクションを探しました。ここでは以下の仮説を立てました。

- 自機と敵とがぶつかるときには当たり判定処理が行われる。このとき、自機座標ないし自機座標を加工した値を持つ変数が参照されるはずだ。
- 弾やアイテムについても同じことが言えるはずだ。

この仮説に基づき、自機座標やそれに関連した変数のアドレスに対してウォッチポイントを設置し、当たり判定が発生しそうな状況とそうでない状況とでウォッチポイントがヒットする場所を比較しました。その結果あるコレクションを走査するコードを見つけました。このコレクションの各要素と自機との当たり判定処理を行っていることが分かりました。このとき私は早とちりして、このコレクションが敵を格納しているコレクションだと考えていました。しかし調べているうちにこのコレクションがスタックであることが判明しました。

敵にせよ弾にせよアイテムにせよ、これらをスタックを用いて管理するというのは考えにくいです。しかもこのスタックは毎フレーム、ゲーム状態の更新後に空にしていることが分かりました。つまりこのスタックはゲーム状態更新の際に一時的に使っているだけなのです。このスタックには敵や弾の当たり判定情報が積まれていて、この要素と自機との当たり判定処理を行うことで、自機対敵の当たり判定処理と自機対弾の処理とをまとめて行っていたわけです。このような振る舞いであることが分かったので、このスタックへ push しているコードを探すことで本当の敵コレクションや弾コレクションを見つけることができました。

この謎コレクションには 4 日くらい振り回されたので記憶に残りました。

4.3 白弾の識別

花映塚には白弾と呼ばれる弾があります。敵を倒すとこの敵のまわりの白弾が消え、その分だけ対戦相手に弾が降り注ぎます。人間が花映塚をプレイする場合、回避経路を作ったり相手を攻撃したりするためには弾が白弾かどうかの識別が攻略において重要です。

花映塚の解析において苦労したのは白弾の識別と EX アタックでした。EX アタックについてはパラメータがデータではなくコードに埋め込まれていることが多いために抽出するのが面倒という大変さでしたが、白弾の識別は純粋に識別方法を見つけ出すことが大変でした。どれほど大変かというと花 AI 塚 ver 1.0 では識別ができず、ver 1.1 に持ち越されるほどでした。

解析が困難な理由としては、ゲーム画面で観測される現象についてその現象が起こる瞬間をデバッガで追いづらいということがあります。ゲーム画面で現象が観測されるのは現象が起きるよりも後です。一方で現象が起きる前と現象が起きた後のコードの実行の流れの違いを知るには、ゲーム画面で現象が観測される前からデバッガで少しずつコードの実行を進める必要があります。これを実際にやるのは困難で、うまく現象の発生前後の比較ができないために確度の低い仮説しか立てられず解析が進みませんでした。

もう一つ解析が困難な理由としては、現象が起きたときにどのようにコードの実行が変わるか想像しづらいというのがあります。白弾を識別するには、弾が白弾であるときだけ実行されるコードを探してその手がかりを探すのですが、白弾のときだけ処理することとは一体何で、それがどのコードに対応しているかが分からない状態が続いたのも解析が滞った原因として大きいです。

[*9] 対戦のたびに new で自機オブジェクトを生成しなおしているから。生成のたびにアドレスが変わりうる。

解析にあたっては以下のアプローチを試しました。

- 弾のライフサイクルに関するフィールドを監視。弾が消えるとき[*10]にこのフィールドが書き換わるので、書き換えを行っているコードから白弾に関わるコードを探す。白弾に関わるコードにたどり着くまでの条件分岐を調べて、白弾を識別する条件を絞り込む。
- 弾の生成時に実行されるコードの中から白弾のときのみ実行されるコードを見つけ出し、そのコードに至るまでの条件分岐から白弾を識別する条件を絞り込む。
- 敵を倒して白弾が消えるときにはエフェクトが発生する。このエフェクトは「弾」アイテムを拾ったときにも発生するので、まず「弾」アイテム取得に関するコードを調べ、エフェクト生成に関する関数を特定する。弾の更新ルーチン経由で実行されるコードのうちこの関数の呼び出しを含むコードが白弾に関わるコードであると仮定して白弾を識別する条件を絞り込む。

最初のアプローチではいくつか白弾の識別に使えそうなフィールドを見つけたのですが、例外的なケースがあって完全に白弾とそれ以外を識別することができず断念しました。2 番目のアプローチは識別できる条件を調べようとしましたが、この条件に関わるコードが複雑で解析を断念しました。この 2 つのアプローチに共通する問題は、白弾に関わるコードかどうかの確度が低いまま解析を進めたということです。条件分岐からの条件絞り込みに断念するほどの例外や複雑さがあるのは、その条件が白弾識別以外のことを含んでいたからと考えられます。よって注目していたコードが本当に白弾のときだけ実行されるコードだったのか疑わしいのです。

最後のアプローチを試した結果、白弾の識別に使えるフィールドを特定することができました。このアプローチは他と較べて注目したコードが白弾に関わるコードであるという確度が高く、そのおかげでうまくいったと考えています。

4.4 バージョンごとの差異について

花 AI 塚 ver 1.0 の時点では花映塚 ver 1.50a のみをサポートしていましたが、ver 1.1 で花映塚 ver 1.00 もサポートしました。複数のバージョンに対応するためにはまず花映塚の実行ファイルからそのバージョンを識別することが必要ですが、これについては花映塚のウィンドウタイトルに表示される文字列がバージョン番号を含んでいるので、この文字列を抽出して調べれば OK です。

バージョン間の差異についてはグローバル変数のアドレスが違う程度でデータ構造の違いはありませんでした。グローバル変数のアドレスの違いについても配置されるページが違うだけで、変数間の相対的な位置は変わっていないので対応は楽でした。

5 実装

花 AI 塚の実装においてやることは

- ゲーム状態を毎フレーム取得すること
- 適切なタイミングで Lua スクリプトのロード・アンロードを行うこと
- Lua スクリプトからゲーム状態にアクセスできるようにすること
- 自機操作をゲーム状態に反映すること

の 4 つです。これらのタスクは「ゲーム状態の監視と介入」と「Lua との連携」に分類できます。

ゲーム状態の監視のアプローチはいくつかありますが、3.1 節で書いた通り今回は DLL インジェクションを用いて花映塚のプログラムと同一のプロセスで AI システムを動かしてゲーム状態の監視を行いました。6 節ではゲーム状態監視の実装について述べていきます。

Lua との連携に関しては単に AI を動かすという機能を実現するだけではなく、セキュリティに関する注意も必要です。花 AI 塚では赤の他人が書いたスクリプトを動かすことを想定しています。そのためどんなスクリプトが動いてもシステムの破壊や情報漏洩が起きないようにするための仕組みが必要となります。7 節では Lua との連携について述べていきます。

6 ゲーム状態の監視

DLL インジェクションでは、

1. 自分が書いたプログラムを DLL として対象のプロセスにロードさせる
2. 対象のプロセスから DLL 内の関数が呼ばれるようにする (パラサイトルーチンの挿入)

といったことを行います。1 については [5] で解説した通りの実装をしました。ここでは 2 のパラサイトルーチンの挿入や、ゲーム状態取得に関する実装ノウハウについて書いていきます。

6.1 パラサイトルーチンの挿入

DLL インジェクションで DLL をプロセスへロードした後、パラサイトルーチンの挿入を行います。プロセス

[*10] 白弾でなくても画面外に出たりボムを使用したりすると弾が消え、フィールドが書き換わる。

のメモリには対象のプログラム自身も書かれており、メモリ上にあるプログラムを書き換えることで対象のプログラムの挙動を変えることができます。

今回の実装では

- 毎フレーム自機操作をゲーム状態に反映する前に実行するコード
- 対戦の開始時だけ実行するコード
- 対戦の終了時だけ実行するコード

の 3 箇所にパラサイトルーチンを仕込みます。具体的にどこがこういったコードに該当するかは解析の時点で調べています。これらの場所でそれぞれ

- AI システムの毎フレームの処理を行う関数
- スクリプトのロードと AI 初期化を行う関数
- スクリプトのアンロードと AI の後始末を行う関数

を呼び出します。

花 AI 塚でのパラサイトルーチン挿入を行うコードをプログラム 1 に示します。これは花映塚 ver 1.50a において AI の処理を毎フレーム行うようにパラサイトルーチンを仕込むコードです。実際にはいくつかのファイルに分散しており、`TH9ver1_5aMonitor` は `TH9Monitor` の子クラスとして定義されています。パラサイトルーチンを挿入するコードが `InjectOnFrameUpdate` メソッドで、このメソッドを呼ぶとメモリ上のプログラムを書き換えて毎フレーム `OnFrameUpdateVer1_5` 関数が呼ばれるようにします。変数 `code` の値が書き換え後の機械語のコードです[*11]。

`OnFrameUpdateVer1_5` 関数について見ていきます。この関数には`__declspec(naked)` 修飾子[*12]がついていますが、これはこの関数がコンパイルされて機械語になるときに余計なコードが追加されないようにするものです。通常関数がコンパイルされると関数の前処理や後処理[*13]を行うコードが追加されます。`__declspec(naked)` を付けた関数ではこれらのコードが追加されません。この修飾子を使う理由はパラサイトルーチンの挿入元で使っているレジスタをなるべく書き換えないようにするためです。関数の冒頭ではレジスタを保存するコードが、関数の終わりにはレジスタを復帰するコードがそれぞれインラインアセンブラで記述されています。これらのコードによってパラサイトルーチンの挿入元でのレジスタの状態を復元し、以後のプログラム実行に矛盾が発生しないようにしています[*14]。

対戦開始時や対戦終了時に関するパラサイトルーチン挿入も `InjectOnFrameUpdate` メソッド及び `OnFrameUpdateVer1_5` 関数と同様のコードで実現できます。

6.2 実装ノウハウ

ここではゲーム状態取得の実装ノウハウについて見ていきます。ここで紹介するものは以下のとおり。

- ゲーム状態のデータ構造に型付けする
- グローバル変数群を構造体へマップする

まず「ゲーム状態のデータ構造に型付けする」について見ていきます。解析で得られる各オブジェクトの情報は「自機オブジェクトの先頭アドレスから何バイト後ろに float 型で自機 X 座標が格納されている」というようにオブジェクトの先頭からのオフセットとそのフィールド自身の型として表されてます。これをそのままゲーム状態取得を行うコードに落とし込むとフィールドごとのオフセット定数をそこら中のコードに書き込むことになって、とてもメンテしづらくなってしまいます。これを防ぐために自機オブジェクトや弾オブジェクトといったオブジェクトごとに構造体を定義します。

プログラム 2　ゲーム状態のデータ構造 (弾オブジェクト) に対する型付け (花 AI 塚のコードから抜粋)

```
#pragma pack(push, 1)
struct Bullet{
    char unknown1[0xD38];
    struct Size2D size;
    char unknown2[0xD4C - 0xD38 - sizeof(Size2D)];
    struct Vector3D position;
    struct Vector3D velocity;
    char unknown3[0xDBE - 0xD58 - sizeof(Vector3D)];
    unsigned int status;
    char unknown4[0x10BC - 0xDBE - sizeof(int)];
    unsigned int unknown_status;
    short bullet_type;
    char unknown5[0x10C4 - 0x10C0 - sizeof(short)];
};
#pragma pack(pop)
```

プログラム 2 では弾オブジェクトに対応する構造体を定義しています。ゲーム状態として意味のあるフィールドの他に、オフセットを調整するためのフィールドがあります (`unknown1` のように数字 N を用いて `unknownN` で表されるもの)。また、`unknownN` フィールドでのオフセット調整以外に余計なオフセット調整が行われないようにするために構造体定義の前後に`#pragma pack`[*15]を

[*11] 実際の書き換え後のコードは 0 で埋めてる部分に `OnFrameUpdateVer1_5` への相対アドレスが補完される。

[*12] Microsoft 独自拡張。http://msdn.microsoft.com/ja-jp/library/h5w10wxs.aspx

[*13] 具体的にはスタック領域の確保や解放など。

[*14] `mov eax, 1` とあるのはパラサイトルーチンの挿入元が eax=1 であることを期待していたためだが、実はこのコードを書く必要がない (どうせ復帰される) ことに気づいてしまった orz

[*15] Microsoft 独自プラグマ。gcc でも使えるらしい。http://msdn.microsoft.com/ja-jp/library/2e70t5y1.aspx

プログラム 1　パラサイトルーチンを仕込むコード (花 AI 塚のコードから抜粋、コメントを追加)

```
void TH9Monitor::SetJumpTo(char* code, int from, int to)
{
    *((int*)code) = to - from; // 相対アドレスを計算
}
void TH9Monitor::WriteCode(char* inject_to, char* new_code, size_t size)
{
    DWORD old_protect;
    ::VirtualProtect(inject_to, size, PAGE_EXECUTE_READWRITE, &old_protect); // 権限の書き換え
    ::memcpy_s(inject_to, size, new_code, size);                             // コードの書き換え
}
int __declspec(naked) OnFrameUpdateVer1_5(void)
{
    __asm{
        pushad;
        pushfd;
    }
    if(monitor){
        monitor->OnFrameUpdate(); // 毎フレームのAIシステムの処理を行う
    }
    __asm{
        popfd;
        popad;
        mov eax, 1; // パラサイトルーチン挿入元の期待する値に合わせる。実は要らない(脚注参照)
        ret;
    }
}
void TH9ver1_5aMonitor::InjectOnFrameUpdate(void)
{
    /**
    .text:00420290                  retn
    を
    .text:00420290                  call OnFrameUpdate ;; return 1
    .text:00420295                  retn
    に書き換える
    */
    char* inject_to = address::addr_on_frame_update.ver1_5; // パラサイトルーチン挿入先のアドレス
    char code[] = {
        0xE8, 0, 0, 0, 0, // call OnFrameUpdate
        0xC3              // retn
    };
    SetJumpTo(code + 1, (int)(inject_to + 5), (int)OnFrameUpdateVer1_5);
    WriteCode(inject_to, code, sizeof(code));
}
```

指定しています。このプラグマは構造体などを定義するときのアライメントを変更するもので、ここではアライメントを 1 にしています。この指定がないと、フィールドのアドレスが 4 や 8 の倍数になるようにオフセットを調整するため、自由にオフセットを指定できません。

このようにゲーム状態を表すオブジェクトに構造体で型付けしてやることで、フィールドのオフセットに関するコードを構造体定義に集約することができます。

次に「グローバル変数群を構造体へマップする」について見ていきましょう。解析においてはグローバル変数は絶対アドレスによる位置で得られます。よってゲーム状態のグローバル変数にアクセスするには

```
int* hoge = (int*) 0x293; // グローバル変数のアドレス
printf("%d\n", *hoge);
```

のようにグローバル変数へのポインタを定義して参照すればできます。が、このように書くとグローバル変数 1 つ 1 つに対してアドレス定数を定義することになり、メンテナンスしづらくなります。ところで、グローバル変数はある程度まとまって配置されます[*16]。となると構造体を使ってグローバル変数群をまとめて扱うというやり方が考えられます[*17]。

実際にグローバル変数群を構造体にまとめたのがプログラム 3 です。ゲーム状態のデータ構造に対して構造体を定義したときと同じように、unknownN フィールドでのアドレス調整や#pragma pack によるアライメント変更を行っています。このように構造体にまとめることによってグローバル変数のアドレスについての情報はほとんどがこの構造体に集約され、アドレスの定数として残るのは構造体の先頭アドレスだけとなります[*18]。

[*16] 静的領域と呼ばれる場所に配置される。

[*17] 別にまとまってなくても適用できると思うけどまぁまとまってたほうが精神衛生上よろしい。

[*18] もうひとつの利点としては複数バージョンへの対応のしやすさがある。4.4 節でも述べたように花映塚 ver 1.00 と ver 1.50a とでグローバル変数のアドレスは異なるものの、各グローバル変数の間のオフセットはバージョン間で変わらない。よって構造体は同じまま、構造体の先頭アドレスを変えるだけで両方のバージョンに対応することができる。が、花 AI 塚 ver 1.1 実装の際には果たして本当にそれでうまくいくか不安だったので ver 1.00 用と ver 1.50a 用とでそれぞれ構造体を定義してしまったという・・・。

プログラム 3　ゲーム状態のグローバル変数群から構造体へのマップ (花 AI 塚のコードから抜粋)

```
#pragma pack(push, 1)
struct Th9GlobalVer1_0{
    struct raw_types::Board board[2];//49ED94
    char unknown1[0x49EE3C - 0x49ED94 - sizeof(raw_types::Board)*2];
    struct raw_types::ExAttackContainer* ex_attack_container;//49EE3C
    char unknown2[0x49EE90 - 0x49EE3C - sizeof(raw_types::ExAttackContainer*)];
    unsigned int round;//49EE90
    char unknown3[4];
    unsigned int round_win[2];//49EE98
    char unknown4[0x49EEAC - 0x49EE98 - sizeof(int)*2];
    unsigned int difficulty;//49EEAC
    char unknown5[0x49EEC4 - 0x49EEAC - sizeof(int)];
    unsigned int play_status;
    char unknown6[0x4A3E18 - 0x49EEC4 - sizeof(int)];
    struct raw_types::KeyState key_states[3];
    char unknown7[0x4AA0B0 - 0x4A3E18 - sizeof(raw_types::KeyState)*3];
    int hwnd; //4AA0B0
    char unknown8[0x4AA104 - 0x4AA0B0 - sizeof(int)];
    int** d3d8;//4AA104
    int** d3d8_device;//4AA108
    char unknown9[0x4AA53C - 0x4AA108 - sizeof(int**)];
    char charge_types[2];
};
#pragma pack(pop)
// 一番最初のグローバル変数の絶対アドレスだけ指定すれば他のグローバル変数へのマップも行われる
struct Th9GlobalVer1_0* const globals_ver1_0 = reinterpret_cast<struct Th9GlobalVer1_0*>(0x49ED94);
// たとえば難易度を表すグローバル変数へのアクセスは以下のようになる
bool is_lunatic = (globals_ver1_0->difficulty == 3);
```

7　Lua との連携

花 AI 塚は C++ で実装されており、Lua の処理系として Lua 5.1.4[*19]を使用しています。Lua のようなスクリプト言語をプログラムに組み込んでいく上での問題として以下のものがあります。

- どのように API を提供するか
- セキュリティをどう守るか

まず前者ですが、C++ 側で用意したオブジェクトをどう Lua 側に提供するかは設計の好みの問題だけでなく、その設計を実現する実装の困難さや実行時のパフォーマンスにも関わる大きな問題です。実装の困難さを緩和するためにバインディングライブラリを利用するというのは有力な選択肢ですが、往々にしてパフォーマンスとのトレードオフとなります。バインディングライブラリを利用するかどうかは API 設計にも大きく影響します。そのためバインディングに関しての問題に触れてから API 設計について述べていきます。

後者のセキュリティについてですが、Lua の標準ライブラリにはファイルの削除など、悪用するとシステムの破壊ができるようなものも含まれているので、こういったライブラリが使えないようにスクリプトの実行に際して制限を課す必要があります。ここでは花 AI 塚ではどのような制限を課し、どう実装したかについて述べていきます。

7.1　バインディング

過去の報告書 [5] で Lua を組み込んだときはバインディングライブラリを使いませんでした。そのためバインディングライブラリの使用がどれだけパフォーマンスに影響するかは未知数でした。

当初、バインディングライブラリとして Luabind[*20]を使用していました。C++ 側でゲーム状態をラップしたクラスを用意しており、このクラスを Luabind でバインドして Lua 側に提供するという実装で進めていました。しかしおおよそシステムができあがった 4 月 14 日[*21]、実際に実用レベルの AI スクリプトを動かしてみたところ、著しく実行速度が遅いことが発覚しました。これによりバインディングまわり、さらには API 設計を見直すことになり、バインディングライブラリを使用しない実装、オブジェクト指向的でない API に変えることになりました。

STG AI を作るというタスクにはパフォーマンスの問題が重くのしかかります。まず 60FPS を維持できるように 16msec 以内に[*22]AI に関するすべての処理を行わないといけません。スクリプトを実行する上でこの条件は厳しく、なるべくオーバーヘッドを小さくする必要があります。また、STG(特に弾幕 STG) では大量の弾が発生します。よって C++ 側で管理するゲーム状態オブジェクトも大量に生成されますし、Lua 側において

[*19] 最新の 5.2 系ではなく 5.1 系を使用しているのは後述する Luabind が公式に対応しているのが 5.1 系までなのと、LuaJIT が 5.1 系互換であることを気にしたため。

[*20] http://www.rasterbar.com/products/luabind.html

[*21] ちなみに花 AI 塚 ver 1.0 公開が 4 月 25 日。

[*22] 2 つの AI を対戦させる場合なら 8msec 以内。

もそうです。少量のオブジェクトしか扱わなければバインディングライブラリによるオーバーヘッドはさほど問題になりませんが、大量のオブジェクトを扱う場合このオーバーヘッドは無視できません。

Luabind のパフォーマンスについて未知数だったとはいえ、花 AI 塚の実装に取り掛かる前に調査することはできたはずです。あらかじめ以前開発した地霊殿 AI[5] に対して Luabind を適用してパフォーマンスを調べるべきだったというのが今回の反省点です。

ところでバインディングライブラリのオーバーヘッドは何だったのでしょうか。Luabind の場合 Lua 側でフィールドにアクセスするとそのオブジェクトの持つメタテーブルの`__index` フィールドへのアクセスを経て、C++ 側の該当するフィールドへのアクセスを行います。メタテーブルの`__index` フィールドを経由したアクセスは Lua のオブジェクトが直接持つフィールドへのアクセスよりも 2〜2.5 倍程度遅く [6]、これがオーバーヘッドの原因の 1 つと考えられます。また、C++ 側から Lua 側へのオブジェクトの型変換もオーバーヘッドの原因と考えられます。前者のオーバーヘッドは [6] によると処理系を本家 Lua 実装から LuaJIT に置き換えることでかなり軽減できるようです。よって LuaJIT を採用することでバインディングライブラリのパフォーマンスの問題を回避できた可能性があります。

7.2 API の設計・実装

バインディングライブラリを使用しないという選択をしたことは API 設計にも影響します。各オブジェクトがメソッドを持つように実装するのがかなりの手間になり、C++/Lua 間での型変換の扱いも面倒になります。また、Luabind の場合は shared_ptr に対応していて C++/Lua 間のオブジェクトの所有権の問題が楽になるのですが、バインディングライブラリ無しではやはり面倒です。こういった面倒を避けるように API を考えると、オブジェクト指向的な API を諦めざるを得なくなります。

Lua 上でクラスベースのオブジェクト指向を実現しようとすると往々にしてメタテーブルを駆使することになりますが、7.1 節の最後で述べたようにこれがバインディングライブラリが遅い原因の 1 つです。となるとパフォーマンスを気にするならやはりオブジェクト指向を諦めざるを得ません[*23]。

花 AI 塚の API では C++ 側のゲーム状態オブジェクトをラップして提供するということはしていません。Lua 側へは単にゲーム状態の値をフィールドに持つテーブル型のオブジェクトを提供しています。

花 AI 塚の API で提供している関数は 2 つだけです。1 つは当たり判定処理の関数で、もう 1 つは自機操作を送信する関数です。ゲーム状態の取得はグローバル変数を経由して行います。AI のスクリプトは対戦の開始時にロードされ、毎フレーム main 関数が呼ばれ、対戦が終了するとアンロードされるというライフサイクルを送ります。

簡単な AI スクリプトの例で見ていきましょう。プログラム 4 は左右往復を繰り返す AI のスクリプトです。3 行目で 2 つのグローバル変数 `game_sides` と `player_side` を使用して自機の X 座標を取得しています。前者は 1P/2P それぞれのゲーム状態を格納する配列、後者は AI が 1P/2P のどちらで動くかを表す変数です。自機の X 座標と現在の移動方向に基いて次に行う自機操作を決定し、`sendKeys` 関数で操作を送ります[*24]。

プログラム 4　左右往復を繰り返す AI のスクリプト

```
1  local move_right = true;
2  function main ()
3    local p_x = game_sides[player_side].player.x;
4    if move_right and p_x > 100 then
5      move_right = false;
6    elseif not(move_right) and p_x < -100 then
7      move_right = true;
8    end
9    if move_right then -- send key
10     sendKeys(2 ^ 7); -- move right
11   else
12     sendKeys(2 ^ 6); -- move left
13   end
14 end
```

グローバル変数は毎フレームゲーム状態が変わるたびに更新されます。このときなるべく既に存在しているテーブルを使いまわすようにしています。これは新たなオブジェクトの生成を避け、ガーベジコレクション (GC) が行われる頻度を下げるためです。Lua はインクリメンタル GC を採用していますが、それでも GC による停止時間はパフォーマンスを悪化させます。オブジェクトは作ってすぐにゴミになることが多いですが、なるべくゴミを作らずに使いまわすことで GC の発生頻度を抑えています。ただ、これには弊害があります。例えば敵 A のデータが `enemies[1]` に格納されていたとして、

```
1  local enemy_A = enemies[1];
```

上のようにローカル変数 `enemy_A` からも敵 A のデータを参照するようにしたとします。1 フレーム経過すると `enemies[1]` が更新されますが、このとき 1 フレーム前のオブジェクトに上書きして使い回します。上書きした

[*23] Lua 側のオブジェクト 1 個ずつにメソッドを直接持たせるというアプローチもあり得るが、オブジェクトの生成コストが大きくなりオーバーヘッドとなることが予想される。

[*24] 引数がビットマスクで指定する仕様になっているのはどうせ適当な仕様でもそれをラップするマシなライブラリがあれば問題ないだろ、という判断の結果。

結果が敵 A のデータとは限りません。そしてローカル変数 enemy_A から参照される値も敵 A のデータとは限らないということになります。これは直感的ではない意味論なので API ドキュメントで注記した上で、enemy_A のようにフレームをまたいでグローバル変数の要素を参照し続けるのは非推奨としています。

7.3 セキュリティ

花 AI 塚では自分で書いたスクリプトを動かすだけでなく他人が書いたスクリプトを動かすこともあります。そのスクリプトが実は悪意のあるコードだった、という可能性は十分あります。危ないコードを安全に実行できる環境をサンドボックスといいますが、花 AI 塚の AI スクリプトの実行もサンドボックスの中で実行する必要があります。

花 AI 塚のサンドボックスはスクリプトから呼べるライブラリを安全なものだけに制限する[*25]ことで危険なコード実行を防いでいます。安全なライブラリとしては

- 数学関数
- Lua の中で閉じているライブラリ

逆に危険なライブラリとしては

- 外部コマンド実行
- DLL のロード
- ファイル操作 (読み書き、削除、作成、移動など)

が挙げられます。

安全上は前者のライブラリのみを呼べるようにすべきですが、実用上困ることがあります。たとえば AI スクリプトのデバッグ時にはデバッグログを吐きたいですし、AI スクリプトが複雑になってきたら別ファイルに分けて読み込みたいです。しかしこれらの機能はファイルの読み書きを行えるという点で危険を伴います。これらの機能に関しては読み書きできるファイルのパスを制限し、AI スクリプトと同一のディレクトリにあるファイルのみを読み書きできるようにしています。これによって不正なファイル操作の危険性を抑えています[*26]。

Lua に対するサンドボックスの実装をプログラム 5 に示します。[7] の実装を元にし、setfenv 関数を用いて[*27]スクリプトから利用できるライブラリを制限しています。C++ で直接サンドボックスを実装するのがだるそうだったので、Lua で実装したサンドボックスを C++ から実行しています[*28]。C++ からスクリプトをロードする際には loadfile_safe 関数 (15 行目で定義) でロードします。このとき setfenv 関数によってスクリプトから呼べるライブラリが制限されます。

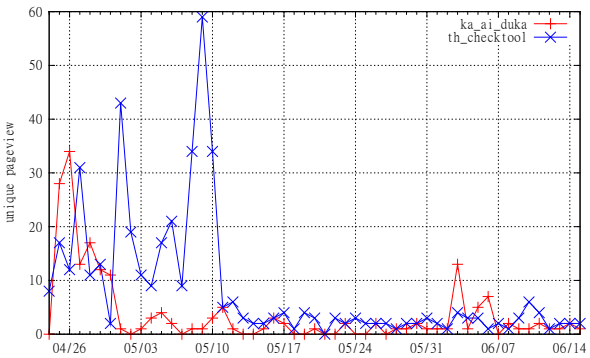

図 1　花 AI 塚と例大祭 11 チェックツールのページ別訪問数の推移。後者のピーク時に前者が釣られて上がるといった振る舞いは見られない。

8 公開とこれから

8.1 公開と反応

花 AI 塚は 4 月 25 日に ver 1.0 が、5 月 5 日に ver 1.1 が公開されました (表 1)。公開の際は Twitter と東方幻想板[*29]のツールスレにおいてアナウンスを行いました。幻想板における反応は皆無に近く、Twitter での反応もツール作成者同士での反応が多く、ユーザーとしての反応が少なく感じられました。

花 AI 塚公開と同時期に例大祭 11 チェックツールの公開も行いましたが、これによる相乗効果は特に見られませんでした (図 1)。それぞれのユーザー層が重ならなかったのか、それともページ誘導がなかったからなのかは定かではありません。

8.2 今後

花 AI 塚の公開に対する反応は薄かったですが、花 AI 塚に需要がなかったかどうかはまた別の問題です。仮に需要があったとして、反応が薄かった理由として以下のことが考えられます。

- 花 AI 塚を周知する適切なメディアを使わなかった
- 花 AI 塚を使いこなせるまでの難易度が高かった

前者に関して具体的に適切なメディアとは何かは分かりませんが、もっと知れ渡りやすいメディアはあると思い

[*25] ホワイトリスト方式。

[*26] パスを制限しても危険性は残る。たとえばファイルを大量に作成してリソースを食い潰すといったことが考えられる。ただ、こういった攻撃はすぐにバレるし復旧できるので今回このリスクは受容している。

[*27] Lua 5.1 系の話。5.2 系では setfenv ではなく load の第 4 引数で指定する。

[*28] 正直バッドな感じが漂っているので C++ で直接サンドボックスを実装するようにしたい、構文ハイライト効かないし。

[*29] 東方関連の掲示板。http://jbbs.shitaraba.net/computer/41116/

プログラム 5　サンドボックスの実装 (花 AI 塚のコードから、一部省略)

```
bool Initialize(::lua_State* ls)
{
    const char * src =
        "env␣=␣{};\n"
        "local␣function␣validate_path(filename)\n"
        "␣␣if␣type(filename)␣~=␣'string'␣then\n"
        "␣␣␣␣return␣false\n"
        "␣␣end\n"
        "␣␣return␣string.match(filename,␣'^[%w_%-]+%.%w+$')␣~=␣nil;\n"
        "end\n"
        "local␣function␣absolute_path(filename)\n"
        "␣␣return␣script_dir..filename;\n" // script_dirはスクリプトファイルのあるディレクトリパス
        "end\n"
        // 制限版loadfile実装。ファイルパスの制限、bytecode実行回避、使えるライブラリの制限。
        "local␣function␣loadfile_safe(filename)\n"
        "␣␣if␣not␣validate_path(filename)␣then\n"
        "␣␣␣␣return␣nil,␣'invalid␣filename';\n"
        "␣␣end\n"
        "␣␣filename␣=␣absolute_path(filename);\n"
        "␣␣local␣fp␣=␣io.open(filename,␣'rt');\n"
        "␣␣local␣magic␣=␣fp:read(4);\n"
        "␣␣if␣magic:byte(1)␣==␣0x1B␣then\n"
        "␣␣␣␣return␣nil,␣'binary␣code␣prohibited';\n"
        "␣␣end\n"
        "␣␣local␣f,␣err␣=␣loadfile(filename);\n"
        "␣␣if␣not␣f␣then\n"
        "␣␣␣␣return␣nil,␣err;\n"
        "␣␣end\n"
        "␣␣setfenv(f,␣env);\n"
        "␣␣return␣f;\n"
        "end\n"
        // 制限版io.open実装。ファイルパスの制限。
        "local␣function␣io_open_safe(filename,␣mode)\n"
        "␣␣if␣not␣validate_path(filename)␣then\n"
        "␣␣␣␣return␣nil,␣'invalid␣filename';\n"
        "␣␣end\n"
        "␣␣if␣not␣mode␣then\n"
        "␣␣␣␣mode␣=␣'r';\n"
        "␣␣end\n"
        "␣␣filename␣=␣absolute_path(filename);\n"
        "␣␣return␣io.open(filename,␣mode);\n"
        "end\n"
        "local␣safe_lib␣=␣[[\n"
        "assert␣error␣ipairs␣next␣pairs␣pcall␣select␣tonumber␣tostring␣type␣unpack␣_VERSION␣xpcall\n"
        // 途中省略。スクリプトからアクセスできる関数やオブジェクトの名前を列挙している
        "HitType␣ItemType␣ExAttackType␣CharacterType␣ChargeType\n"
        "]];\n"
        "for␣s␣in␣string.gmatch(safe_lib,␣'[a-zA-Z0-9_%.]+')␣do\n"
        "␣␣local␣s1,␣s2␣=␣string.match(s,␣'([a-zA-Z0-9_]+)%.([a-zA-Z0-9_]+)');\n"
        "␣␣if␣not␣s2␣then\n"
        "␣␣␣␣env[s]␣=␣_G[s];\n"
        "␣␣else\n"
        "␣␣␣␣if␣not␣env[s1]␣then\n"
        "␣␣␣␣␣␣env[s1]␣=␣{}\n"
        "␣␣␣␣end\n"
        "␣␣␣␣env[s1][s2]␣=␣_G[s1][s2];\n"
        "␣␣end\n"
        "end\n"
        "env.loadfile␣=␣loadfile_safe;\n"
        // 実際には制限版dofileも用意しているが省略。
        "env.io.open␣=␣io_open_safe\n"
        "env._G␣=␣env;\n";
    ::luaL_loadbuffer(ls, src, ::strlen(src), "sandbox");
    if(::lua_pcall(ls, 0, 0, 0)){
        return false;
    }
    return true;
}
```

ます。ただ、半分は技術的自己満足から始めたプロジェクトなのであまり大々的に宣伝したいかというと微妙なところです。後者に関しては、花映塚 AI を作るチュートリアルを用意していこうかと考えています。

また、花 AI 塚の API 仕様に関してはパフォーマンス上の妥協が少なからずあり、この辺を解決できるといいかなと思っています。方針としてはスクリプト実行のパフォーマンスを向上させることで API 仕様の制約を緩める余裕を作ることを考えています。パフォーマンス向上の策としては LuaJIT の導入などがあります。ただ、これで向上するパフォーマンスと API 仕様を綺麗にすることで落ちるパフォーマンスとがうまくバランスするかは予想しがたいので、当分はチュートリアル作成に注力するつもりです。

参考文献

[1] @aki33524: “東方紅魔郷 AI”, http://www.slideshare.net/aki33524/ai-32089294 (2014).

[2] trial-run: “AI に『ダブルスポイラー』をプレイさせる本”, http://trial-run.net/archives/2235 (2011).

[3] @ide_an: “東方地霊殿の自動プレイプログラムの作成”, http://www.usamimi.info/~ide/programe/touhouai/ (2012).

[4] shinichiro.h: “白い弾幕くん”, http://shinh.skr.jp/sdmkun/ (2001).

[5] @ide_an: “東方地霊殿 AI システムの改良”, http://www.usamimi.info/~ide/programe/touhouai/ (2013).

[6] “lua-users wiki: Object benchmark tests”, http://lua-users.org/wiki/ObjectBenchmarkTests (2012). Accessed at 2014/6/8.

[7] “lua-users wiki: Sandboxes”, http://lua-users.org/wiki/SandBoxes (2012). Accessed at 2014/6/13.