東方地霊殿 AI システムの改良

計算工学専攻 M2 いで (@ide_an)
http://www.usamimi.info/~ide/

概要

2012 年前期研究報告会において発表した東方地霊殿自動プレイ AI についてシステムの改善を行った。1 つ目は対象の実行ファイルと同一のプロセスで実行させることによるパフォーマンス向上である。これを実現するために DLL インジェクションを用いた。2 つ目は AI のスクリプト化である。AI の作成は試行錯誤を伴うため、書換えのし易いように外部のスクリプトとして切り離した。AI 自体の改善をした訳ではないので注意されたし。

1 従来のシステムと本稿について

昨年度の前期研究報告会において東方地霊殿をプレイする AI について発表した [1]。この AI のプログラムはプレイ対象である東方地霊殿のプログラム (以後対象プログラムと称す) とは別プロセスで起動し、定期的に対象プログラムのプロセスのメモリを読み出して自機や弾の位置などを取得し、その情報を元に AI が自機の操作を決定し、キー入力を送信することで自動プレイを実現している。

このシステムによって AI による東方地霊殿のプレイが実現することが示されたが、いくつか問題があった。

- 別プロセスのメモリを読み出す操作はオーバーヘッドが大きい。このため AI の処理がゲームの処理に間に合わないことがある。
- メモリの読み出しの間もゲームの処理は進み続けるため、読み出しの間にメモリの内容が変わっていく。そのため AI プログラムが取得したゲーム情報が整合性を取れていないおそれがある。
- AI が C++ でハードコーディングされているため AI の修正がしづらい。だるい。めんどい。

上記の問題のうち始めの 2 つは AI プログラムが対象プログラムとは別のプロセスで実行されることに起因している。本稿では上記の問題を解決するために AI プログラムと対象プログラムを同一プロセスで実行させ、また AI をスクリプト言語 Lua によって記述できるようにした。

2 同一プロセスでの実行

2.1 なぜ同一プロセスか?

別プロセスのメモリを読む場合 Win32API では ReadProcessMemory 関数を用いる。読み出すメモリのサイズにもよるがこの関数のオーバーヘッドは概して大きく、この関数を 1 フレーム[*1]に数千回呼ぶ従来のシステムではプログラムの実行時間のかなりを占めていた。

もし AI のプログラムが対象プログラムと同一のプロセスで実行できるなら、メモリの読み出しのためにわざわざ ReadProcessMemory 関数を呼ぶ必要はなく、ポインタを経由した読み出しができる[*2]。よって従来のシステムが背負っていたオーバーヘッドを解消することができる。

AI のプログラムを対象プログラムと同一プロセスで動かす場合、AI のプログラムをゲーム処理と同一のスレッドで動かすか否かという選択がある。別スレッドで動かす場合はゲームの処理と AI プログラムの処理が同期しないため、冒頭に挙げた問題点の 2 番目については解決しない。一方、同一ス

[*1] 60fps なのでだいたい 16msec
[*2] ポインタの値、つまり指すアドレスはあらかじめ解析で得たアドレスを直に指定する

レッドで動かす場合は AI プログラムがゲーム処理の中に組み込まれ、AI の処理の間ゲームの処理が待機するため 2 番目の問題が解決される[*3]。今回は同一スレッドで動かすことを目指す。

2.2 DLL インジェクションとは

AI のプログラムを対象プログラムと同一のプロセスで実行させるためには DLL インジェクションを用いる。

DLL(Dynamic Link Library) とは実行時にリンクされるライブラリのことである[*4]。通常の (静的な) ライブラリはビルドして実行ファイルを生成する際にライブラリがリンクされる。DLL の場合はビルド時にはどの関数がどの DLL に含まれているか (つまりどの DLL をロードする必要があるか) の情報のみをリンクし、ライブラリの実行コード自体はプログラムの実行時に DLL ファイルを探索してロードすることで初めてリンクされる。

対象プログラムが実行時に読み込む DLL として自作の DLL を読みこませることができれば、その DLL に含まれる実行コードは対象プログラムと同一のプロセスで動かせる。あとは実際にそのコードを実行すればよい。これを実現するのが DLL インジェクションである。

Windows の場合 DLL インジェクションを実現する方法はいくつかある[*5]。一番直感的な方法としては対象プログラムがロードする DLL を調べておき、その DLL のダミーを用意してロードさせるというのがあるだろう。別の方法としては Win32API の CreateRemoteThread 関数を利用して対象プロセスに任意の DLL をロードさせるというのがある。今回は後者の方法を用いた[*6]。

2.3 DLL インジェクションの実践

CreateRemoteThread 関数を用いた DLL インジェクションの詳細を見ていく。この方法では対象プログラムに DLL を読み込ませるだけのプログラムと DLL 本体の 2 つを用意する。それぞれ順番に見ていこう。

2.3.1 DLL を読み込ませる

DLL を読み込ませる手順は以下のとおり。

1. 読み込む DLL の名前をおく領域を確保する。この領域は対象プログラムのプロセス上に作るので VirtualAllocEx 関数を用いる。
2. 読み込む DLL の名前を確保した領域に書き込む。別プロセスにある領域への書き込みなので WriteProcessMemory 関数を用いる。
3. DLL をロードする関数である LoadLibrary 関数[*7]のアドレスを取得する。
4. CreateRemoteThread 関数を用いて対象プログラムのプロセスで LoadLibrary 関数を呼ばせる[*8]。
5. 確保した領域を解放する。

要は対象プログラムが LoadLibrary 関数を使って自作の DLL を読み込む、という挙動をさせたい訳だ。ただし LoadLibrary 関数に食わせる引数の値も対象プログラムのプロセス上に置いておく必要がある。DLL を読み込ませるプログラムそれ自体は対象プログラムとは別プロセスであるため、手順 1, 2 のようにして引数の値を対象プログラムのプロセス上に用意してやらないといけない。

以上のことを実現する関数をプログラム 1 に示す。この関数は引数で指定されたプロセスに対して `DLL_NAME` で指定された dll を読み込ませる。ただしこの dll は対象プログラムと同一のディレクトリにあるものとする。

[*3] この場合 AI がトロいとゲーム自体が処理落ちする

[*4] *nix 系で同様のものとして共有ライブラリ (*.so) がある

[*5] なので詳しくはググれ

[*6] この方法を選んだのは VsyncPatch[2] を参考にしていたという都合がある

[*7] 実際には文字列の型に char を使うか WCHAR を使うかによって呼ぶ関数名が違う。本稿では WCHAR を使うため文字列絡みの関数には末尾に W が付く

[*8] CreateRemoteThread 関数に渡す関数のアドレスは対象プログラムのプロセス上でのアドレスだ。一般に異なるプロセス同士では関数のアドレスが同じである保証はないが、kernel32.dll はどのプロセスでも同じアドレスにロードされるため kernel32.dll に属する LoadLibrary 関数のアドレスも呼び出し元プロセスと対象プログラムのプロセスとで同じになる。そのためこの呼び出しができるのだ。闇である

プログラム 1　DLL インジェクションを行う関数 (エラー処理は省略)

```
1  #define DLL_NAME L"hoge.dll"
2  BOOL InjectDll(HANDLE process)
3  {
4    WCHAR filename[0xff];
5    DWORD filename_size = sizeof(filename)*sizeof(WCHAR);
6    GetModuleFileNameW(NULL,filename,filename_size);
7    wcscpy(wcsrchr(filename,'\\')+1,DLL_NAME); //ここまでで書き込みたいDLLの名前を用意
8    PWSTR remote_memory =
9      (PWSTR)VirtualAllocEx(process,NULL,filename_size,MEM_COMMIT,PAGE_READWRITE); //手順 1.
10   WriteProcessMemory(process,remote_memory,filename,filename_size,NULL); //手順 2.
11   PTHREAD_START_ROUTINE thread_routine =
12     (PTHREAD_START_ROUTINE)GetProcAddress(
13       GetModuleHandle(_TEXT("kernel32")),"LoadLibraryW"); //手順 3.
14   HANDLE thread = CreateRemoteThread(
15     process,NULL,0,thread_routine,remote_memory,CREATE_SUSPENDED,NULL); //手順 4.
16   ResumeThread(thread);
17   WaitForSingleObject(thread,INFINITE); //スレッドの終了を待ってからメモリを解放する
18   CloseHandle(thread);
19   VirtualFreeEx(process,remote_memory,filename_size,MEM_RELEASE); //手順 5.
20   return TRUE;
21 }
```

2.3.2　DLL の関数を呼ばせる

先の節で述べた手順によって対象プログラムのプロセスに DLL がロードされる。次はロードした DLL の中にあるプログラムを実行させたい。DLL がロードされた際には DLL の中で定義された DllMain 関数が呼び出される。DLL の中の関数を一度呼ぶだけでいいというのであれば単に DllMain 関数内から呼び出せば済むだろう。しかし今回やりたいことはもう少し手間が必要だ。

AI のプログラムを組み込む場合は毎フレームのゲームの処理が行われる度に DLL の中にある関数を呼ばせるようにしないといけない。これを実現するためにはメモリ上にロードされている実行コードを書き換える必要がある。

コードの書換えとしてはコード領域の適当な隙間に以下のアセンブリに相当する機械語を挿入し,

```
pushad ;汎用レジスタを退避
pushfd ;ステータスレジスタを退避
call [呼びたい関数のアドレス]
popfd ;ステータスレジスタを復元
popad ;汎用レジスタを復元
[ジャンプ元の元々の命令]
jmp [ジャンプ元の1つ後の命令のアドレス]
```

DLL の関数の呼び出しを挿入したい箇所の機械語を上記のアセンブリ断片へのジャンプ命令に書き換える。

ここで問題になるのがどこに DLL の関数の呼び出しを挿入するかである。AI の処理は毎フレームのゲームの処理の度に呼ばれて欲しい訳であるから、当然毎フレーム実行されるコードを探す必要がある。そしてゲームのプレイ中以外 (タイトル画面やリプレイ再生画面など) では実行されないコードであることが望ましい。

一般にゲームプログラムの構造は

- プレイヤーの入力を受け取る
- ゲーム状態を更新する
- 表示に反映する

を毎フレーム繰り返すループとなっている。このうちプレイヤーの入力を受け取る段階は入力取得まわりの API の呼び出しがあるため、逆アセンブルしたコードから該当するコードを見つけやすい[*9][*10]。

解析した結果、東方地霊殿においてはプレイヤーから受け取った入力をゲーム状態の更新に用いる入力に変換するコードが存在し、このコードはリプレイ再生時やポーズ時、タイトル画面などでは実行されないことが分かった。このコードを挿入先として

[*9] 表示に反映する段階でも特徴的な API 呼び出しはあるだろうが、入力を受け取る段階の方が使われる可能性のある API の種類が少なく探しやすい

[*10] とはいえ DirectInput を叩くオブジェクト指向的なコードはメソッド呼び出しが vtable を介して行われるため読みづらい。東方地霊殿においては DirectInput と Win32API の GetKeyboardState を併用しているため見つけやすかったという側面もある

プログラム 2　DLL の関数の呼び出しコードの挿入

```
void SetJumpTo(char* code, int from, int to) // 相対アドレスを求める補助関数
{
    *((int*)code) = to - from; // ひどいキャストだ
}
int InjectCode()
{
  /*
   左のコードを右のコードに書き換える。
   00436D20 8BC1         mov eax, ecx        00436D20 60           pushad
   00436D22 E9B9F2FFFF   jmp 00435FE0h       00436D21 E8********   call 呼びたい関数
   00436D27 CC           int 3          ==>  00436D26 61           popad
   00436D28 CC           int 3               00436D27 8BC1         mov eax, ecx
   00436D29 CC           int 3               00436D29 E9********   jmp 00435FE0h
   00436D2A CC           int 3          ; アセンブリ断片へのジャンプを廃して挿入先の
   00436D2B CC           int 3          ; コード領域にアセンブリ断片を丸ごと書き込んでいる。
   00436D2C CC           int 3          ; ステータスレジスタまわりの処理を忘れていたが、
   00436D2D CC           int 3          ; 別に問題なく動いているので省略した。
  */
  char inject_code[] = { // 書き込みたいコード(機械語)
    0x60,           // pushad
    0xE8,0,0,0,0,   // call ****
    0x61,           // popad
    0x8B,0xC1,      // mov eax,ecx
    0xE9,0,0,0,0    // jmp ****
  };
  char* ptr = (char*)0x00436D20; // 書き込み先の先頭アドレス
  // call命令のオペランド(呼びたい関数の相対アドレス)を生成
  SetJumpTo(inject_code+2,(int)(ptr+6),(int)OnFrameUpdated);
  // jmp命令のオペランド(ジャンプ先の相対アドレス)を生成
  SetJumpTo(inject_code+10,(int)(ptr+sizeof(inject_code)),0x00435FE0);
  DWORD old_protect; // メモリ周りの権限を変更して書き換えできるようにする
  VirtualProtect(ptr,sizeof(inject_code),PAGE_EXECUTE_READWRITE,&old_protect);
  for(int i=0;i<sizeof(inject_code);++i){ // 機械語を書き換える
    ptr[i] = inject_code[i];
  }
  return 0;
}
```

先に述べた実行コードの書換えを行うことで AI プログラムを毎フレーム実行させることができるようになった。この書換えを行う処理はプログラム 2 のようになる。このコードでは OnFrameUpdated 関数を毎フレーム呼ばせるようにしている。DllMain 関数が呼び出された際に InjectCode 関数を呼び出すことで AI プログラムのための DLL インジェクションは成就する。

2.4　パフォーマンスについて

従来の別プロセスからゲーム状態を監視する AI システムと今回作成した DLL インジェクションを用いて同一プロセスでゲーム状態を監視する AI システムについてパフォーマンスの測定を行った。ここでは東方地霊殿 Extra 開始から古明地こいしの最初のスペル[*11]までについて毎フレームのゲーム状態取得にかかる時間を測定した (表 1)。

DLL インジェクションによって従来の AI システムに較べて平均的なケースについて 30〜40 倍、最大 100 倍の高速化が達成できたことが分かる。

3　AI のスクリプト化

今までの経験上 AI を書くというのは試行錯誤を伴うものだ。答えとして考えたモデルがまずいのか、それともモデルのパラメータを選び間違えただけなのか、分からない故に AI のコードを大なり小なり何度も書き換える。そうして何度も書き換える度にコンパイラにお伺いを立てないといけないのはとてもしんどい。そこで AI のコードを AI システムのプログラムから切り離してスクリプトとして記述できるようにしようという考えに至るのはごく自然なことだろう。

アプリケーションに組み込むことを前提として作られたスクリプト言語はいくつもある。今回は組込

[*11] 表象「夢枕にご先祖総立ち」。AI がここでゲームオーバーすることが多い

表 1　ゲーム状態取得にかかる時間の測定結果 (1 フレーム当たりの時間。単位はすべて μsec)

測定環境	実装	平均	標準偏差	中央値	最小値	最大値
CPU: Core2 Duo 3.0GHz/RAM: 4GB	DLL インジェクション	98	93	74	17	629
	別プロセスから監視	3358	595	3328	1992	8728
CPU: Celeron 1.2GHz/RAM: 4GB	DLL インジェクション	229	171	183	50	2600
	別プロセスから監視	6613	1349	6222	4853	14809

みについての情報が容易に手に入る点とスクリプト言語の中でも実行速度が速い点から Lua を採用した。

3.1　速度

STG のプログラムは毎フレームその時のゲーム状態について自機と弾などとの当たり判定処理を行う。それに対して STG をプレイする AI はその時のゲーム状態だけでなく数フレーム先のゲーム状態を予測した上での当たり判定処理も行うため、計算量はより大きくなる傾向がある。

Lua はスクリプト言語の中でもかなり実行速度の速い言語として知られているが、無論 C/C++ ほどには速くないため計算量にはより気をつけるべきだ。具体的な目安として 1 フレームに 4 千回以上当たり判定処理を行うと処理落ちが発生するようになる[*12][*13]。自機から遠い弾について先読みを行わないなど不要な当たり判定処理を減らせば、60fps を維持した状態で AI によるプレイが実現できる。

3.2　AI API の構成

AI を記述する上で必要なものとしては

1. 現在のゲーム状態に関する情報の取得。自機や弾、敵の位置や速度の取得等。
2. 自機と弾などとの当たり判定処理。
3. 自機に対する操作の指定。

がある。通常のゲームにおけるスクリプト化と違ってゲーム上のオブジェクトをスクリプト側から書き換える必要はない。

今回作成した AI システムの場合は Lua 側で定義した main 関数を毎フレーム呼び出す。この関数は自機操作の入力を戻り値として返すものとし、この戻り値を元に AI システムが自機を操作する。自機や弾などの情報は Lua 側に対してグローバル変数として提供する。この変数の中身は単に位置や速度などの値が入ったテーブル[*14]やその配列で、main 関数の呼び出しの直前に更新される。これらによって 1. や 3. については満たされる。

2. の当たり判定処理については厄介だ。東方地霊殿においては当たり判定の形状にはいくつか種類があり、2 つのオブジェクトの当たり判定処理を行う際にはそれぞれの当たり判定の種類に応じてロジックを切り替える必要がある。当たり判定処理のコードを Lua に移植するのは面倒臭いし、何より 1. に関して当たり判定に関する情報をより多く取得できるようにしなければならない (つまり面倒臭い)。よって当たり判定処理については C++ 側で実装された関数として Lua 側に提供し、当たり判定まわりの詳細は隠蔽するという考えに行き着く。

AI の記述の場合、現在の自機や弾の位置における当たり判定だけでなく数フレーム先で予測される位置での当たり判定処理がしたくなる。が、敵や弾などの情報を単なるグローバル変数で提供し、Lua 側から C++ 側へデータの書き換えが反映されない仕組みとして作った手前、弾の座標を書き換えた上で当たり判定処理を行う、といったことが自然にはできない[*15]。

[*12] CPU:Celeron SU2300 1.20GHz RAM: 4GB での場合

[*13] 東方地霊殿において画面に現れる弾の個数は多く見積もっても 2 千個程度なので余裕だと思うかもしれないが、全弾について先読みを行う残念な AI 実装では (先読みフレーム数) × (自機の取り得る位置の数) が掛け算されるため越えることがある

[*14] Lua におけるデータ構造。配列と連想配列を兼ねる

[*15] やっつけ仕事のツケを払わされていると言える

現状では敵や弾に ID を持たせることにし、当たり判定処理については自機との当たり判定だけを行う以下の関数を Lua 側に提供している。

```
hitTestAgainstPlayer(id, offset_x, offset_y)
```

id に当たり判定を行う敵ないし弾の ID を指定し、offset_x と offset_y で指定した分だけ自機の座標をずらした状態で当たり判定を行う。これによって弾と自機がそれぞれ移動した後の状態での当たり判定処理ができる[*16]。

4 今後の課題

AI のスクリプト化については従来のシステムがスクリプト化を想定してないために問題を招いていたところもある。今後 AI システムを作成する際にはスクリプト化を想定して作る必要がある。

もう一つ AI のスクリプト化に関して触れなかったこととしてセキュリティの問題がある。現時点では AI システムの作者である筆者が AI のスクリプトを記述し自分の環境で動かしているためセキュリティに気を使う必要性は低い。しかし今後他人が書いた AI のスクリプトを動かすといったことが起きるようになると話は変わる。Lua の標準ライブラリがそのまま AI スクリプトから使えてしまうと任意のファイルを書き換えるなどセキュリティ上まずい操作ができてしまう。それを防ぐために AI スクリプトから使える標準ライブラリ関数を制限したり、実装を差し替えるなどする必要がある。

5 関連研究

東方関連で AI のスクリプト化に関する事例としては sweetie 氏による緋想天/非想天則 AI[3] や心綺楼 AI[4] がある。それぞれスクリプト言語として Lua や mruby を使用している。格闘ゲームの場合は弾幕 STG と較べて画面中にあるオブジェクトが少ないため、当たり判定の計算量の問題に直面する可能性は低いと考えられる。また、これらのツールではセキュリティ上の都合から標準ライブラリの制限や差し替えが行われている。

組込み言語を持つアプリケーションは解析対象としても興味深い。東方心綺楼はそれ自体 Squirrel を組込み言語として使用している。こういった組込み言語の上に構成されたゲームの解析は難しいようだ。@s_yukikaze 氏による綺録帖 [5] では東方心綺楼のプログラムが組込み言語側に提供しているテーブルの内容を別プロセスから取得している。ソースコードが公開されており、具体的なアプローチや解析の困難さの一端を知ることができるだろう。

[*16] 弾の回転については考慮していない。これを考慮すると回転軸はどこかという話になり当たり判定の詳細を知らないとスクリプトが書けないというジレンマに陥る

参考文献

[1] @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/diary/2012_6.html (2012).

[2] ◆ swmpLV/75E：“VsyncPatch”, http://thwiki.info/?VsyncPatch%B2%F2%C0%E2 (2008). Accessed at 2013/11/16.

[3] sweetie: “th123_aiVer0.95”, http://resemblances.click3.org/?p=1387 (2012). Accessed at 2013/11/18.

[4] sweetie: “th135_ai v1.01”, http://resemblances.click3.org/?p=1574 (2013). Accessed at 2013/11/18.

[5] @s_yukikaze: “綺録帖 Rev.5”, https://twitter.com/s_yukikaze/status/369076399304237056 (2013). Accessed at 2013/11/22.