東方地霊殿の自動プレイプログラムの作成

計算工学専攻 M1 いで (@ide_an)
http://www.usamimi.info/~ide/

1 概要

今回私は東方地霊殿 (製品版 ver1.00a)[*1]を自動プレイするプログラムを作りました。このプログラムでは地霊殿プロセスのメモリから自機や弾などの情報を読み取り、AI によって次に自機が取るべき操作を決定し、ゲーム上の自機を操作します。

1.1 動機

去年の冬コミ (C81) で東方のプレイ AI に関する同人誌[*2]を買いまして、その本に感化されて自分も東方の AI を作ろうと思ったのが動機です。

この同人誌では画像処理を用いて弾っぽいところを見つけ出し、その近辺を回避するような AI を組むという手法を用いていましたが、PC のスペック上の問題やメモリ解析についての興味から私はメモリ解析によって自機や弾などの情報を取得する方法を選びました。

1.2 開発スケジュール

地霊殿のメモリから様々な情報を取得するにあたってあらかじめ地霊殿のメモリについて解析を行いました。メモリの解析は 1 月上旬〜中旬と 3 月の 2 回に分けて行いました[*3]。

プログラミングは 3 月の始めから 4 月中旬にかけて行いました。始めの 1 ヶ月は C#で開発していたのですが、パフォーマンス不足やマルチスレッドプログラミングに関する混乱により 4 月に入ってから C++ に移植して開発を続行しました。

[*1] http://www16.big.or.jp/~zun/html/th11top.html

[*2] 「AI にダブルスポイラーをプレイさせる本」(http://trial-run.net/archives/2235)

[*3] 分けたというかメモリ解析に行き詰って作業が止まったというのが本当のところ orz

1.3 自動プレイの流れ

自動プレイは以下の処理を繰り返し行うことで実現してます。

1. 地霊殿プロセスから自機情報、敵弾、敵、フレームカウントを読み取る。フレームカウントが更新していれば以降の処理をする
2. 取得した情報に基づいて次に自機がすべき操作を AI によって決定する
3. 決定した操作をキー入力に変換して地霊殿プロセスに送信する

1. については 2 節で、2. については 3 節でそれぞれ述べていきます。

2 メモリ解析

地霊殿のプロセスから自機などの情報を取得するために地霊殿のメモリを解析するということを行いました。ここではメモリ解析とは何か、どのようにやったのかを見ていきます。

2.1 変数はどこにある？中身はどこにある？

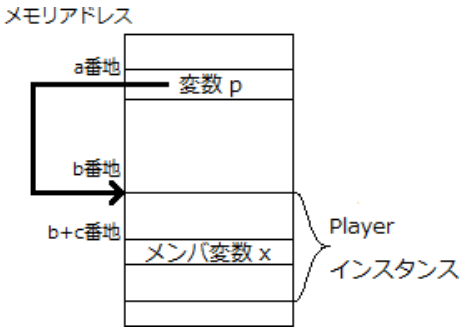
以下のようなプログラムについて考えてみます。

```
#include <iostream>
class Player{//自機クラス
public:
    int x;
    int y;
    Player(int x,int y){
        this->x = x;
        this->y = y;
    }
};
```

```
Player* p;
int main(void){
    //自機を動的に生成
    p = new Player(10,20);
    //p を使った処理（とりあえず x を表示）
    std::cout << p->x << std::endl;
    delete p;//p を成仏
    return 0;
}
```

このプログラムは自機クラスのインスタンスを生成して、そのインスタンスを使うプログラムです。このプログラムではグローバル変数 p が自機クラスのインスタンスを指しています。このとき変数 p や p が指しているインスタンス、メンバ変数 x などがメモリのどこにあってどのように表現されているかを割り出すのがメモリ解析です。

このプログラムの場合のメモリ配置はおおよそ以下のようになります。



グローバル変数の場合、その変数が配置されるアドレスは固定されたアドレスに定まっています[*4]。なので変数 p は毎回プログラムを実行する度に同じアドレス a に配置されます。一方 Player インスタンスが配置されるアドレス b は固定されておらず、プログラムの実行時に決まります[*5]。このアドレスから Player のメンバ変数 x までのオフセット c は固定されていますが、b が実行時に決まる以上プログラムの外から x の値を参照するのに x のアドレスを決め打ちにしてアクセスすることはできません。外から x にアクセスするには a 番地に入っている値 (=b) を読み取って (b+c) 番地に入っている値を読むという手順を踏むことになります。以上から外からメンバ変数 x を参照するにはアドレス a とオフセット c を知る必要があることが分かります。

2.2 解析の流れ

どうやって a と c を求めるかということですが、おおよそ以下の手順で求めていきます。

1. プログラム実行中のメモリを解析して (その実行時の) メンバ変数 x のアドレスを特定する
2. 1. で特定したアドレスにメモリブレークポイントを設定し、メンバ変数 x を読み書きするアセンブリコードの場所を特定する
3. プログラムを逆アセンブルし、2. で特定した場所からコードを遡って読んでいき、a と c を割り出す

まず 1. ですが、ここではプロセスメモリデバッガというツールを使い、プログラム実行中のメモリの変化から見つけたいデータにマッチするような変化の特徴を持つアドレスを探します。例えば自機の X 座標を抱えているメンバ変数であればゲーム中での自機の横移動操作に応じて値が増減するはずなので、メモリの中からそのような値の増減が起こるアドレスを探していけば良いわけです。

次に 2. ですがこの作業もプロセスメモリデバッガを使って行います。プログラムの中でメンバ変数を読んだり書いたりする操作はコンパイルされた機械語ではそれぞれ「メモリの中のあるアドレスから値を読み出す」、「メモリの中のあるアドレスに値を書き込む」といった操作になります。メモリブレークポイントとは特定のアドレスについてこれらのメモリ読み書きの操作を行った時に、その操作がプログラムのどこで行われたものかを記録する機能です。これによってプログラムのどの部分でメンバ変数に対する操作が行われるかが分かります。

[*4] 変数の配置場所が固定されるかどうかは変数のエクステント (寿命) によって決まる。グローバル変数のようにプログラムの実行の最初から最後まで生きている変数は固定される。ローカル変数はそうではない。

[*5] 動的に確保された領域だから。new や malloc などを用いて確保される領域はヒープ領域と呼ばれ、実行時に配置が決まる。

最後の 3. は逆アセンブルについては逆アセンブラを使えば簡単にできますが、そこから先の解析は地道にアセンブリコードを読んでいかないといけません。読んでもよく分からないところはデバッガを使って調べていきます。

2.3 地霊殿解析の流れ

地霊殿の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 [*6] プロセスメモリデバッガ
OllyDbg [*7] デバッガ
PeRdr [*8] 逆アセンブラ

解析はまず自機の情報を取得することから始めました。手順は前節で述べたのと同様に行い、自機情報の取得方法を確立しました。

次に当たり判定処理を行なっているコードの特定を行いました。ここでは自機の座標が配置されているアドレスに対してメモリブレークポイントを設定し、ブレークポイントに引っかかった場所の中から実行時の状況やデバッガによる介入、アセンブリの読解を元に該当するコードを特定しました。当たり判定処理は浮動小数点演算が多用されているのでこのあたりの命令に関する知識が必要になりました。

当たり判定処理では弾や敵などの当たり判定情報を扱っているので、この情報がどうやって流れてくるかをアセンブリやデバッガを頼りに辿っていくことで敵弾や敵の情報の取得方法を確立しました[*9]。このあたりの解析のおおまかな手順は

1. 当たり判定ルーチンに至るまでの呼び出し履歴を遡る (関数ポインタを介した呼び出しの場合もあるためデバッガの支援が必要)
2. コンテナを走査するコードにたどり着いたのでコンテナの構造を解析
3. さらに呼び出し履歴を遡った結果タスクリスト[*10]と思しきデータ構造にたどり着いたのでタスクリストのノードの初期化ルーチンを探す (ノードが関数ポインタを保持していて、その値が定数になっているのでこれを手がかりにアセンブリコードから検索)
4. ノードの初期化ルーチンを解析して敵弾などを格納するコンテナへのポインタを特定[*11]

といった感じでした。

2.4 解析結果

解析の結果得られたメモリ配置を図 1〜図 5 に挙げていきます。

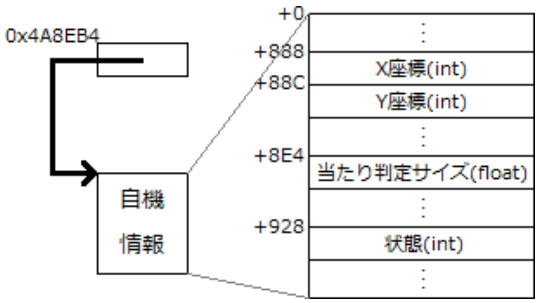


図 1　自機情報のメモリ配置

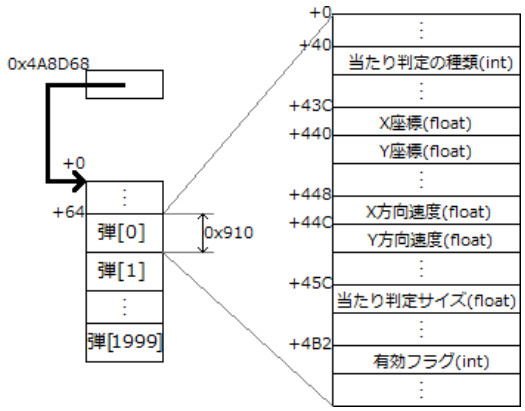


図 2　敵弾情報のメモリ配置

[*6] http://www.vector.co.jp/soft/win95/prog/se375830.html
[*7] http://www.ollydbg.de/
[*8] http://sourceforge.net/projects/perdr/
[*9] この辺の解析で一度挫折した結果冒頭に述べたようなスケジュールに・・・。

[*10] タスクシステムでググれ
[*11] なんでか知らないけどノードのメンバだけでなくグローバル領域にもこれらのポインタが格納されていた

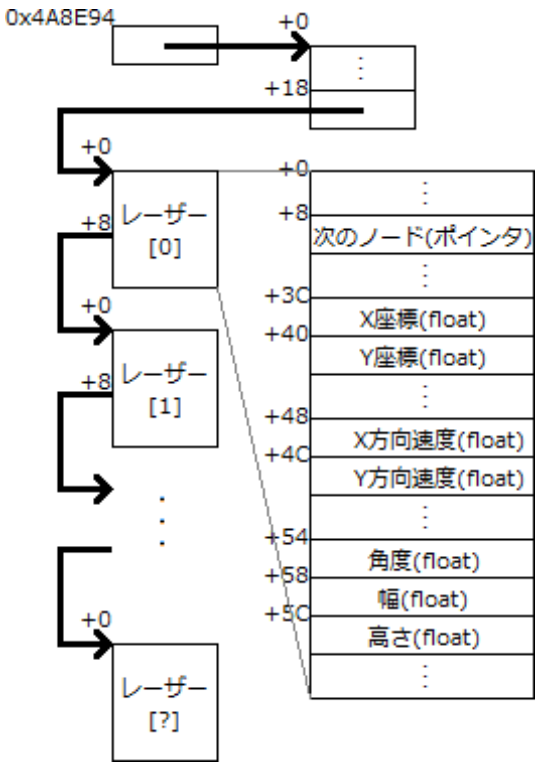


図 3　敵レーザー情報のメモリ配置

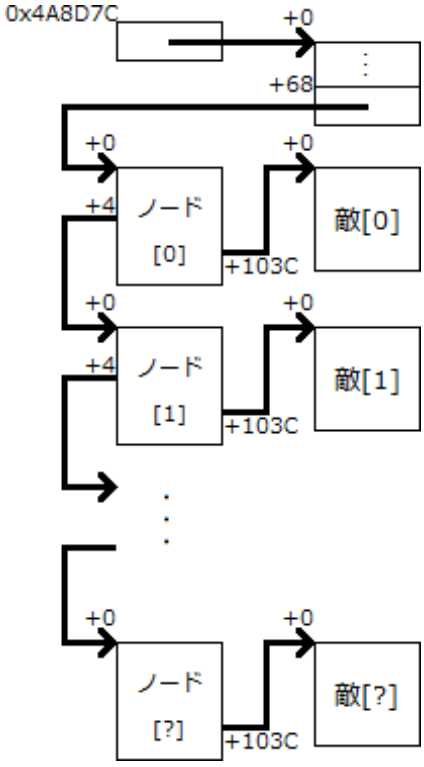


図 4　敵コンテナのメモリ配置

図 5　敵情報のメモリ配置

3　AI

前節でのメモリ解析によって地霊殿プロセスから自機や敵弾の位置など各種情報を取得できるようになったので、これらの情報を元に敵弾などを回避する AI を作成しました。

3.1　アイディア

自機操作のポリシーとして以下のことを考えました。

脊髄反射的な回避　数フレーム先までの弾の動きを予測したとき、弾とぶつかる可能性が低い操作を行う。動かなくても弾に当たる恐れがなければ動かない。

積極的な回避　現在のフレームにおいて自機の近辺で最も弾密度の少ないところを探し、なるべく弾密度の薄いルートで移動する

各移動動作[*12]についてこれらのポリシーを数値化し、これらから作成した評価値を求め、最も評価値の大きい動作を選択するというのが今回の AI の動作です。

[*12] 上下左右と停止。高速移動縛りで斜め移動は行わない

評価値 $E^{\text{動作}}$ は以下の式で求めています。$p_*^{\text{動作}}$ が各種ポリシーの数値、W が重み付けです。

$$E^{\text{動作}} = p^{\text{動作}}_{\text{脊髄反射的}} \left(1 + W p^{\text{動作}}_{\text{積極的}}\right)$$

基本的には脊髄反射的な回避を重視していて、それを補佐する形で他のポリシーを使用しています。

各種ポリシーの数値化について見ていきましょう。脊髄反射的ポリシーは以下のように数値化しています。

$$p^{\text{動作}}_{\text{脊髄反射的}} = \max\left(0, 1 - \sum_{n=1}^{N} \sum_{i} f^{\text{動作}}(n, b_i)\right)$$

N : 先読みフレーム数

b_i : 当たり判定 (敵、弾、レーザーなど)

$$f^{\text{動作}}(n, b) = \begin{cases} \dfrac{1}{2^n} & \left(\begin{array}{l}\text{動作を続けると}\\ n \text{ フレーム後に}\\ \text{当たり判定 } b \text{ と}\\ \text{衝突するとき}\end{array}\right) \\ 0 & (\text{otherwise}) \end{cases}$$

積極的ポリシーについては弾が薄い場所にたどり着く経路を計算し、その経路に沿って移動する動作は $p^{\text{動作}}_{\text{積極的}} = 1$、そうでない動作については $p^{\text{動作}}_{\text{積極的}} = 0$ を割り当てています。経路計算の詳細ですが、自機の近傍をタイルに分割したフィールドを考え、各タイルにそのタイル内にある当たり判定の面積の和をコストとして割り当て、一番コストの小さいタイルから自機までの経路を A*アルゴリズムによって求めています。

積極的ポリシーに対する重み付け W は決め打ちで $W = 0.01$ としています。

3.2 戦果

作成した AI の成果ですが、現状では Easy クリアもできない感じです。

要因としては

- 長期的な予測に基づく回避ができない [*13]
- 初見殺しで死ぬ。経験による学習をしない [*14]
- 弾やレーザーのサイズ変化を予測できない[*15]

といったことが挙げられます。

その一方である程度大回りな回転避けも実現できました。これは積極的な回避ポリシーによる効果と考えられます。

4 今後の展望

AI に関しては大局的な回避ができていないという問題があり、このあたりを改善できるようなモデルが欲しいと思っています。また、アイテム回収や敵への攻撃を重視した立ち回りなど回避だけに留まらないプレイができたらいいなと思っています。が、これらを実現する気力が・・・。まぁ前向きに善処します。

[*13] 霊烏路空のメガフレアで巨大弾に押しつぶされるなど

[*14] 勇儀の三歩必殺の泡弾など

[*15] さとりのテリブルスーヴニールのレーザーや勇儀の 2 回目の通常弾幕の赤レーザーなど