ゲームプログラムからの一部の仕様の抽出に関する考察

金城 拓実 [1], 河野 真治 [2]
Takumi KINJO, Shinji KONO

私達は家庭用ゲームマシン向けのオープンなゲーム開発環境に関する研究を行ってきた [5][3][4]。ゲームマシンは、特殊なアーキテクチャで実現されており、ハード性能を追求するとプログラムコードは必然的にハードへの強く依存する。強い依存性のあるコードは汎用性がなく再利用は難しい。ここで我々はプログラムの Demonstration を保存しながら technology mapping [2] を実現する手法を提案したい。

1 はじめに

我々はこれまで、「PlayStation ねっとやろうぜ」(PlayStation)、PS2Linux(PlayStation2) 、GBA (Game Boy Advance) などの家庭用ゲームマシン (以下ゲームマシン) 向けのゲーム作成実験をリアルタイム・プログラムやユーザ・インタフェースの学生実験の一環として行ってきた。

実験は、ゲームマシンのアーキテクチャや専用 API の理解から始まり、グラフィックの描画処理を構築し、作成するゲームに見合った API を実装する。実験生はこれらを 4ヵ月の実験期間中に行わなければならならないが、大抵の場合は、オブジェクトの描画や、制御、オブジェクト同士の接触判定の処理が動く程度でしか達成できない。それらは実行できるゲームの一部分ではあるが、単体ではゲームとは呼ばない。

[プレイ] 他者との競いや共闘である。プレーヤーは他者とスコアについて、またはゲームの技術的な面で競うことになる。

[ルール] 競いはルールの上で行われる。ゲームに制約を設けることであり、テレビゲームではオブジェクト制御の制限である。

[シナリオ] ゲームの流れや区切りである。テレビゲームではゲームの仮想的な物語であったりする。などが計算機上で仮想的に行われることがテレビゲームに足る条件と思われる。

また、[プレイ][ルール][シナリオ] は相互の関連がある (図 1)。シナリオは例えば場面であり、場面が変わればルールが変わる。プレーヤーからの入力はルールによる処理により、プレーヤーをルール内に留める。シナリオはまた、敵キャラクターを登場させる順番であり、プレイにより変化する。

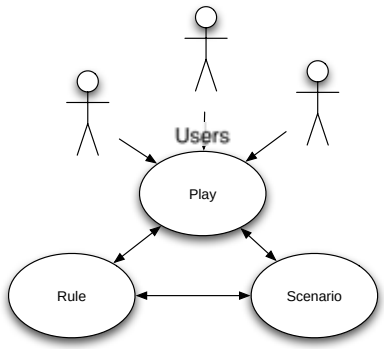

図 1: User, Play, Rule, Scenario の相互関係

我々はこれまでの経験から、実験生がゲームを作りきれないのは、過去の実験によって得られたノウハウや成果物を生かせない原因があると考えている。

オブジェクトの座標変換やオブジェクト同士の接触判定は、実装方法にもよるが、ノウハウはハードやアーキテクチャとは独立である。半面、ゲーム作りのノウハウは実行できる生きたプログラムの中に含まれている。しかしゲームマシンの入れ代わりに伴い過去のコードや成果物は古くなり、現行のシステムでは実行できないものもある。動かないプログラムからノウハウを取り出すのは難しい。

近年みられるハードウェアの劇的な進歩は、このようなソフトウェアの陳腐化の構図を招いてきた。ゲーム開発も同様に技術対応に時間を割く半面、限られた時間の中で、どのようなゲームを作成するかを迫

[1]琉球大学理工学研究科情報工学専攻
kinjo@cr.ie.u-ryukyu.ac.jp
Intelligent System Engineering, Graduate School of Engineering and Science, University of Ryukyu

[2]琉球大学工学部情報工学科
kono@ie.u-ryukyu.ac.jp
Infomation Engineering, faculty of engineering, University of Ryukyu

られたとき、過去の成果物を現行のシステムに移植することで速いゲームのリリースを行っていると予想できる。

そこで我々は、比較的小さな仕様に切り出したゲームの一部 (Demonstration) を繋ぎ合わせてゲームを作成する手法を提案したい。

今回、実際のゲームプログラム Super Dandy [1]から Demonstration の抜き出しと汎用計算機上への移植を行った。

本稿では、過去に行った Adapter(後述) を用いた移植について問題点を挙げ、今回行った Demonstration ベースの移植について説明し、双方の手法について比較したい。

2 経緯

現行のマシンに、過去の成果物であるゲームプログラム Super Dandy を移植した。労力を最小にするためと、コード自体が成果物であるという考えから、なるべくコードに変更を加えないという方針で移植を行った。

ゲームプログラムを異なる機種に移植するにはアーキテクチャへの依存性を解決する。ゲームプログラムはアーキテクチャや専用 API に強く依存するのは必然である。ゲームマシンは特殊なアーキテクチャでありリソースの制約が厳しい。ゲームプログラム

```
ActiveBuff = GsGetActiveBuff();
GsSetWorkBase(GpuPacketArea[nActiveBuff]);
GsClearOt(0,0,&WorldOT[nActiveBuff]);
sp = sprite;
for(i=0;i<=count; i++ , sp++) {
  pageno = sptable[spview[i].no].page;
  sp->tpage =GetTPage(
    0,0,640 + sgo_tpx[pageno],sgo_tpy[pageno]);
  ...
  GsSortFastSprite(sp,&WorldOT[nActiveBuff],0);
  ...
}
DrawSync(0);
VSync(0);
GsSwapDispBuff();
GsSortClear(0,0,0,&WorldOT[nActiveBuff]);
GsDrawOt(&WorldOT[nActiveBuff]);
```

図 2: Super Dandy の描画処理部分

に高度な表現能力を求めるとき、ゲームマシン・アーキテクチャの駆使が要求される。

図 2 は Super Dandy の描画処理を行うコードである。このゲームプログラムは、「PlayStation ねっとやろうぜ」の開発環境を用いて作成され、Gs の接頭辞の付く関数が PlayStation のグラフィックス API である。Super Dandy はこのような API 依存のコードは一つのファイルにまとめられている。

PS2Linux は、「PlayStation ねっとやろうぜ」のようなゲーム作成用の API は用意しておらず、ioctl のような低級な API を用意している。これらの API からゲーム作成用の API を作成すること自体はそれ程難しくない。しかしアーキテクチャやインタフェースなどのギャップを埋める実装 (Adapter) が必要である。Super Dandy は API 依存のコードを一つのファイルに集める方針で実装されており、Adapter の適用は容易に行うことができた。

```
 1void draw_sprite(GsSPRITE* sp) {
 2  ps2utilSprite *obj;
 3  ...
 4  obj = &sprite_obj[sprite_counter++];
 5  ps2util_sprite_Set_basicAttribute(
 6    obj,
 7    ADJS_SCREEN_X + (ushort)sp->x * ADJS_SCREEN_W,
 8    ADJS_SCREEN_Y + (ushort)sp->y * ADJS_SCREEN_H,
 9    (ushort)((float)sp->w * ((float)sp->scalex / \\
ONE_f)) * ADJS_SCREEN_W,
10    (ushort)((float)sp->h * ((float)sp->scaley / \\
ONE_f)) * ADJS_SCREEN_H,
11    (ushort)sp->u+ADJS_TEX_X,
12    (ushort)sp->v+ADJS_TEX_Y,
13    (ushort)sp->w-5, (ushort)sp->h-0,
14    SPRITE_PRIO_FOREGROUND );
15  obj->attribute.rot = (float)sp->rotate / ONE_f \\
/ ANGLE_HALF180_f * M_PI;
16  ...
17  ps2util_sprite_Set_pivot( obj,
18    (ushort)((float)sp->mx*((float)sp->scalex/ONE_f\\
)) * ADJS_SCREEN_W,
19    (ushort)((float)sp->my*((float)sp->scaley/ONE_f\\
)) * ADJS_SCREEN_H);
20  ps2util_sprite_Request( obj );
21}
```

図 3: Adapter のコード

図 3 のコードは PS2Linux で作成したゲーム作成用の API を用いた Adapter 実装の一部分である。ps2util の接頭辞が付いている関数が PS2Linux の API で、`ADJS_SCREEN_X` や `ADJS_TEX_X` などは、ゲーム実行時の画像のずれや、画像の表示位置のずれを修正するための定数である。15 行目の演算はスプライト[2]の回転値を求めている。PS は浮動小数点

[1]1998 年の実験生によって作成された PlayStation 向けのシューティングゲーム。「PlayStation ねっとやろうぜ」のツール・キットで作成され、ハードウェアのスプライト描画機能を用いている。

[2]画像オブジェクト。VRAM に画像を DMA 転送することで描画する。PlayStation,GBA はアーキテクチャで実現している。PlayStation2 はスプライト専用のアーキテクチャは持たないが代替できるアーキテクチャはある。

数演算を苦手としていたため、浮動小数点数ではなく整数の回転値を持っていたが、PS2 は整数よりも浮動小数点数の演算に特化したアーキテクチャを持っており、回転値は浮動小数点数で持ったほうが速い演算を期待できる。

Adapter による方法はオリジナルのゲームプログラムに API の明確な切り分けが要求される。これにはある程度の経験が必要である。また、API の相違を埋めるための余分な演算やコードが必要な他、アーキテクチャ内臓の専用 FPU やコプロセッサ等を用いた演算等の処理がオリジナルのコードに含まれると、Adapter のようなアプローチでは解決できない。

Adapter によるゲームプログラムの移植は GAME BOY ADVANCE(R) にも行われた。GBA アーキテクチャは FPU を内臓しておらず、浮動小数点数演算はソフトウェア演算されるのでゲームの処理速度は著しく低下した。

オブジェクト指向言語は依存性を弱めることが期待できるが、method search や meta level での実行を実現するために余分な条件判断を増やしてしまう。また、オブジェクト指向の実装はインタプリティブ・アプローチになる。処理は生成されたオブジェクトを逐次実行することで行われる。これは変化に強いプログラミング・スタイルだが、オブジェクトをマネージメントするための余計な実装を増やす。ゲームに登場するオブジェクトの状態がオブジェクト自身に閉じないこともコードを複雑にする原因である。動的なオブジェクト生成や GC(garbage collect) もリアルタイム性の求められるプログラムやリソース制約の厳しいゲームマシンへの組込み用途には適さない。

3 Demonstration

2 に述べたようにゲームプログラムはアーキテクチャや専用 API に依存するのは必然であるが、いずれにせよゲームプログラムはアーキテクチャへの依存部分と非依存部分とに分離することはできる。

しかしチューニングされた PlayStation で動作するプログラムと PlayStation2 で動作する同じゲームプログラムがあった場合、双方はアーキテクチャへの依存部分と非依存部分とに分離しても非依存部分のコードは完全には一致しない (図 4)。

完全に一致するコードでは 2 でも述べたように、Adapter API が必要であり、Adapter を用いたコードはアーキテクチャにチューニングできない。従って、アーキテクチャに非依存なコードを完全に保存する移植は適当ではない。

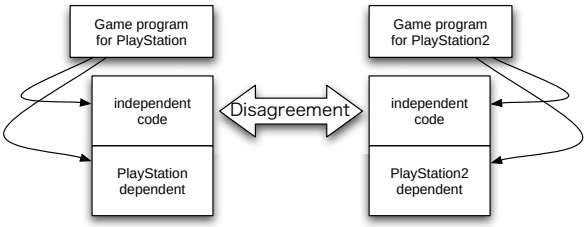

図 4: 非依存部分のコードの不一致

しかし、図 5 のように細分化されたコード同士には、部分的に一致するものも出てくる。このようなコードの断片は、プログラムの内部状態の一部分を切り取ったものである。一つの断片について単体での実行が可能であるなら、これらはプログラムの動作検証を行うためのテストコードであり、テストコードはゲームの仕様の一部を満たすことからゲームの Demonstration となる。

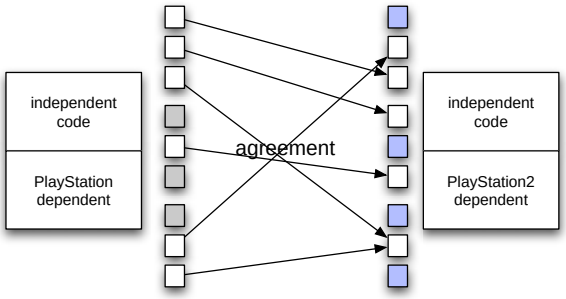

図 5: 細分化したコードの部分的な一致

4 Demonstration extraction and poring

実際のゲームプログラムからゲームオブジェクトに着目し、Demonstration を抜き出しを行った。

ゲームプログラムはオブジェクトや、モジュール化を意識して作成しない限りゲームオブジェクトを構成する要素は図 6 のようにコードの中に散らばって存在する。Super Dandy も同様にオブジェクトを意識しないコードとなっており、オブジェクトの情報は最初から与えられていないため、自機 (player plane と呼ぶ) を構成するコードの抜き出しを行った際は、抜き出しは player plane の動作を保存しながらコー

ド中の不要な箇所を削っていく消去法で行った。

コードの抜きだしは program slicing[1] では static slicing と呼ばれる手法が適用可能だと思われる。static slicing はいくつかのパラメータを保存しながらそのパラメータに影響を及ぼす箇所を抜き出す。しかし、元のオブジェクトの情報が予測できないか、または無く、オブジェクトのコードがプログラム全体に散らばっているコードには、パラメータに着目する方法の slicing は困難であると思われる。

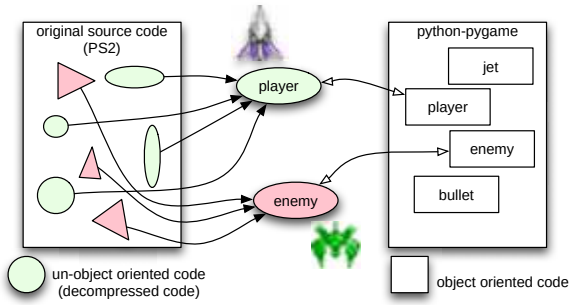

図 6: Demonstration の抜き出し

抜き出した Demonstration は、動作を保存しながら汎用計算機上に移植した。移植には python script を使用し、入出力を実現するために python の SDL ラッパー (pygame) を用いた。

Demonstration の作成は単体での動作検証はもちろんだが、先に述べたゲームであることを満たす条件 [ルール][競い][シナリオ] や、実際に面白いゲームであるかは実行しなければ分からない。Demonstration の作成には速いサイクルでの変更、実行が要求される。従って Demonstration はスクリプト言語による作成が適当である。python がオブジェクト指向言語である点は Demonstration の抜き出しを、ゲームオブジェクトをターゲットにしたことと、十分なリソースをもつ汎用計算機上の実行環境であることとから、ゲームプログラムの Demonstration はオブジェクト指向言語による記述も適当であると判断した。オブジェクト指向言語を用いたため、インタプリティブ・アプローチになっているが、これも汎用計算機上であるので問題はない。PS2Linux 上の切り出されたコードは、オリジナルのコードと比べるとシンプルな仕様を持つ。

例えば、図 7 はプレーヤーが操作する飛行機の噴射のアニメーションを実現する部分である。`jiki` が飛行機の位置情報、`pad` はパッドの矢印ボタン入力の情

```
1  jiki.x+=pad[0].right-pad[0].left;
2  jiki.y+=pad[0].down-pad[0].up;
3  ...
4  bania++;
5  bania%=3;
6  DefSprite (12,0,16*bania,160,16,16,480);
7  count++;
8  PutSprite(count,jiki.x+8,jiki.y+32,12);
```

図 7: 噴射の Demonstration コードの一部

報である。ボタンが押されていれば 1 が入る。`bania` は噴射のフレームカウンタで、カウンタが 3 進む毎に、違う画像を (アニメーションのコマ) を DefSprite 関数で定義する。`count` は、フレーム内で PutSprite 関数を実行した回数を表す。そして PutSprite 関数で画像 (スプライト) を描画する。PutSprite はスプライトを表示する関数であるが、「スプライトを表示する」こと自体はアーキテクチャ独立である。

python では図 8 のように記述できた。

```
1    self.__oldAnimSeq = self.__animSeq
2    if self.__frame % self.__ANIMRATE == 0:
3        self.__animSeq += 1
4        if self.__animSeq > 3:
5            self.__animSeq = 0
6        self.__gameObj.changeMaterial(self.__JET[\\
self.__animSeq])
7    ...
8    self.__gameObj.putAt(x, y)
9    self.__frame += 1
```

図 8: python に移植された Demonstration

言語仕様によりコードの概観は変わるが、内容は同じであり、フレームカウンタが 3 進む毎に表示する画像を入れ換える部分は変わらない。だが、python 側には PutSprite 関数に当たる描画はなく、生成されたスプライトは消滅するまで描画されたままである。putAt メソッドでスプライトの描画位置を更新することができ、画面のリフレッシュ時に putAt された位置にスプライトが描画される。このような描画方式の相違も移植を難しくする原因である。

この小規模の記述が一つのゲームオブジェクトの仕様である。この記述はメインルーチンから呼ぶことで Demonstration になる。このような小さな仕様で記述されたコードの移植は易しい。

しかしオリジナルのコードから Demonstration を切り出すのは難しい。Super Dandy のオリジナルのコードから図 7 の箇所を特定するには、オリジナルのコードにある程度の情報が求められた。しかし、Super Dandy は変数名などに統一されたルールはな

く、他のオブジェクトの記述は混同して同一の関数に集められており、コードからの類推は困難だった。

5 Test surface verification

4 では Demonstration を汎用計算機に移植する作業について説明したが、汎用計算機からゲームマシンへの移植も可能だと思われる。

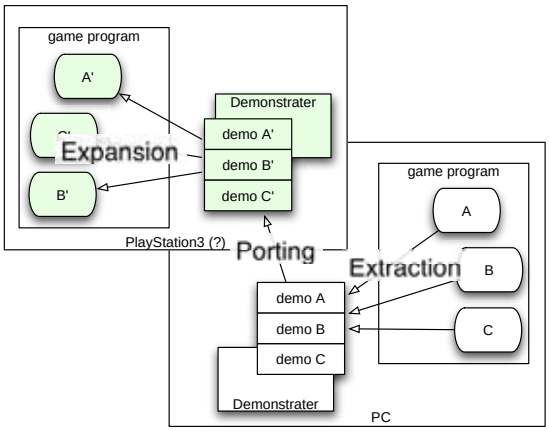

図 9: Demonstration ベースのゲームプログラム移植

4 は手作業による Demonstration の抜き出しや移植であったが、これらの作業は自動化されて然るべきか、または、作業のためのツールは作成できると思われる。自動化されれば、エミュレータ環境の開発なしに、汎用計算機で作成された Demonstration からゲームマシンへの移植を容易に行うことができる (図 9)。

重要性のあるツールの一つとして Demonstration のシーケンス (3) の検証系が挙げられる。

Demonstration は手作業による移植であるので、オリジナルの Demonstration のシーケンスは完全に保存されていない。しかしオリジナルの Demonstration のシーケンスは完全に保存されなくても移植後の Demonstration はオリジナルのそれとほぼ同様に動作するように見える。これは移植された Demonstration は、同様に動作したと言えるに適当なシーケンスが保存されていたためと考えられる。このようなゲームオブジェクトの移植に適当なシーケンスは保存されなければならず、これらは検証されなければならないと思われる。また、手作業では移植不可能な多くの Demonstration から成るゲームプログラムである場合も同様にテストサーフェースの保存を自動的に検証する系は必要である。

6 まとめ

今回、実際のゲームプログラムから Demonstration の切り出しを行い移植を行った。Demonstration は小さな仕様でまとめられた動作可能なゲームプログラムの一部であるため、移植における動作検証は手作業でも行える。しかし多くの Demonstration から成る場合は手作業は限界があることからある程度の自動化や検証を行う系などは必要である。

参考文献

[1] Weiser, M. Program Slicing,. *Proceedings of the Figth International Conference on SoftwareEngineering*, pp. 439-449(1981).

[2] 佐渡山陽, 河野真治. Continuation Based C による Technology Mapping のサポート. FIT, Sep,2002

[3] 佐渡山陽, 河野真治. Continuation Based C による PlayStation2(R)Vector Unit のシミュレーション. 日本ソフトウェア科学会 19 回大会論文集, June,2002

[4] 島袋仁, 河野真治. C with Continuation と、その PlayStation への応用. 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS), May,2000

[5] 河野真治, 金城拓実. ゲームプログラムのシナリオに基づいた状態遷移系を生成するシステムの提案. 日本ソフトウェア科学会 21 回大会論文集, Sep,2004