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

Proposal of system generates state transitions based scenario in game program

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

我々は PS2Linux 向けのオープンなゲームプログラム環境に関する研究を行っている。ゲームプログラムにおけるオブジェクトやシナリオ、プログラムが走るマシン環境などを単位としてまとめ、その振る舞いを個別のゲームプログラムの状態遷移として記述する。ゲーム全体の振舞を、個別の状態遷移の積としてコンパイルするシステムを新たな手法として提案する。旧 PS や Game Boy Advance などの移植作業を通して、新しいシステムの有効性に付いて考察する。

1 はじめに

我々はこれまで、リアルタイム・プログラムやユーザ・インタフェースの教育の一環として、「PlayStation ねっとやろうぜ」、PS2 Linux 、GBA (Game Boy Advance) などのゲーム機向けのゲームプログラミングを行ってきた。しかし、これらの専用機器によるプログラミングは、OS の API を使ったもの、あるいは、GUI ツールキットを使用したものとは異なり、独自なプログラミング手法と、そのライブラリ (ゲームプログラミングに特化したフリーでオープンなフレームワーク) が必要であることがわかってきた。

ゲームプログラミングでは、Graphics に関する部分が大きく、特に、PS2 などのゲームに特化したアーキテクチャでは、その特化した部分用のエンジンを構成する部分に労力がそそがれる。しかし、そのような部分は (大学生などの限られた能力では時間はかかるが)、2D イメージや 3D ポリゴンを表示する基本的なエンジンとして実現することができる。そして、そのエンジンを用いて、簡単なオブジェクトを動かすことまでは、初めて触る学部学生でも 3ヵ月程度で可能である。

しかし、面白いゲーム、完成されたゲームを作るためには、3ヵ月では難しく、少なくとも 1 年はかかると言うのが我々の経験である。ゲームの面白さは、高度な Graphics とは別な部分にあり、その部分により多くの労力を要求される。しかし、この部分を習得するのに時間がかかることが、ゲームプログラミングを中心とした学生実験からわかってきた。

Graphics 以外のゲームシステムの中核を成すのは状態遷移系の設計、実現、保守である。学生によって実現されたゲームシステムの状態遷移系は非常に大きな case 文や if 文によって構成され、入力や内部パラメータの変化を常に監視している。しかし、巨大な case 文や if 文は当然のごとく、プログラムの保守性を悪くしてしまう。実際、パズルゲームのような場合に、パズルのルールを守っているかどうかを保証することは難しく、また、バグを見つけた場合にどこを修正すれば良いかを見つけることは容易ではない。

ただし、状態遷移系自体は、ゲーム機に対する依存性は低く、実際、PS, PS2, GBA と同じゲームの移植を行った際は、状態遷移系に対する変更は、まったく必要なかった。

また、これらの状態遷移は、ゲームに登場するオブジェクト (キャラクタや 3D ポリゴンの部品) とは別に定義されることが観察されている。ゲームオブジェクトをオブジェクト指向言語を用いて記述することは、容易に思い付くことであり、実際に、C++ 等のオブジェクト指向言語を用いて実装されているゲームもある。しかし、極めてリアルタイム性が要求される部分はオブジェクト指向言語的なプログラムにはならない。

なぜならオブジェクト指向言語の記述は本質的にオブジェクト内に閉じているが、ゲームの状態は複数のオブジェクトにまたがって変更されるからであ

[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

る。例えば、ミサイルがボスに当たったときに、その処理は、ミサイル内部でもボス内部にも閉じてはいない。特に、時間制約、例えば、爆発シーケンスなどは、ミサイルとボスと同時に変化する状態変化であり、本質的にオブジェクト指向的な状態遷移とは異なる。

また、リアルタイム性の強いゲームでは、ダイナミックにオブジェクトを作ること、あるいは、GC することは受け入れられないことである。システムの柔軟性のためには、実行中に様々な判断をする方が良いが、それでは、十分な速度は得られない。

オブジェクト指向設計によるゲームシステムでは、各オブジェクトは、インスタンス変数によるリンクにより木構造を持つことになり、イベント処理は、その木構造をたどるインタプリタ的なアプローチになる。これは、汎用的な処理であり、それ自体は見通しが良い。しかし、これは、毎回、つまり、画面表示タイミング (1/60 秒) ごとに、すべての木構造を追跡することになる。描画処理のために一回は行う必要があるが、それを複数回 (例えば、ミサイルの衝突毎に) に行うことはゲームエンジンのパフォーマンスを落とす結果となる。

そこで本稿では、グラフィカルオブジェクトや物語、ルールといったゲームを構成する主要なオブジェクトを同一のモデル (ゲームモデル) として捕らえ、それらの振る舞いとシステムの状態遷移系との関連に着目し、状態遷移系のもつ問題点を除去し、保守性、移植性を向上させるコンパイラベースのフレームワークを提案する。

本稿は 8 節で構成され、第 2 節では我々が提案してきたフレームワークについて、第 3 節ではゲームシステムの状態遷移系について説明し、第 4 節で現状の問題点を列挙する。第 5 節では 4 節で挙げた問題を改善するための提案を行い、第 6 節では第 5 節における提案を実際のゲームシステム開発と対比しながら考察する。第 7 節では今後の実装に関して述べ、最後、第 8 節でまとめる。

2 ゲームシステム開発の概要

ゲームシステム開発では関与するアクターを次の 2 つ に大別する。一つは、ゲームシステム設計やコーディングの作業を行うアクターで、これを SE と呼ぶ。他方は、登場するキャラクターの作成を行い、物語の台本やゲームのルール定義といった作業を行うアクターで、これを Designer と呼ぶ。

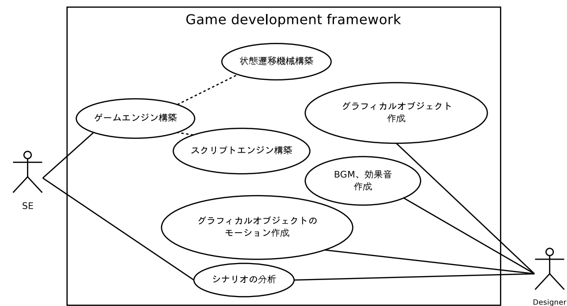

図 1: ゲームシステム開発の概略

SE の構築するゲームエンジン は ゲームシステムに依存し、Designer の作成するグラフィカルオブジェクト、BGM といったものをゲームの進行に沿って処理、もしくは出力する。Designer はゲームエンジンの能力を理解した上でグラフィカルオブジェクトや BGM の作成を行い、ゲーム世界におけるフィールドやストーリーにそれらを配置する。このことから、SE と Designer の両者は構築するゲームシステムを十分分析し、その作業は完全に同期されている必要がある。

そして両者の接点となるのがスクリプティングである。SE は可能な限りデータ主導型のゲームエンジンを目指すべきであり、スクリプティングのためのインタプリタを実装する必要がある [1]。Designer は作成したグラフィカルオブジェクトや BGM といったデータをスクリプティングによってゲームシステムに組み込んでいく。しかし、メインループで行われる 3-D グラフィカルオブジェクトの制御や、それらの衝突検知といった極めてリアルタイム性を要求される箇所にはインタプリタを導入できない。もしインタプリタを導入できない箇所に、変更を加えなければならないときはゲームエンジン自体を組み直す必要があった。

3 従来方法における状態遷移系の実現

3 節では、あるゲームを仮定しながら従来方法における状態遷移系の実現について説明する。

図 2 はあるゲームシステムの状態遷移系の一部であり、TITLE、SELECT を経てゲームの本編へと場面が移り変わる状態遷移を表している (図 4)。図 2 中、button 構造体は入力パラメータを保持する構造体で、start, up, down ボタンの情報を保持すると仮定する。また scene が保持する TITLE, SELECT,

ENDLESS, PUZZLE パラメータは、ゲームの場面を表すものとする。

```
switch(scene) {
case TITLE:
  if (button.start)
    scene = SELECT;
  break;
case SELECT:
  static int sl = SL_ENDLESS;
  if (button.down)
    (sl == SL_ENDLESS)?
      sl = SL_PUZZLE : SL_ENDLESS;
  if (button.up)
    (sl == SL_ENDLESS)?
      sl = SL_PUZZLE : SL_ENDLESS;
  if (button.start) {
    if (sl == SL_ENDLESS)
     scene = ENDLESS;
    else if (sl == SL_PUZZLE)
     scene = PUZZLE;
  }
  break;
```

図 2: if 文と case 文による状態遷移系

従来は図 2 のように、状態遷移系を if 文や case 文の集合によって実現していた。しかしこの実現方法では状態数の成長に従って、状態を表すフラグや状態の分岐を表現できなくなっていく。

4 従来設計手法における問題点

これまでゲームシステム開発の従来方法におけるフレームワークについて説明した。しかしながら、従来方法の開発フローではシステムの心臓部である状態遷移系に関する問題点があった。

- 状態遷移系の実現
- 状態遷移系の保守
- シナリオと状態遷移系との関連付け

5 提案

そこで我々は、ゲームに登場するグラフィカルオブジェクトの振る舞いや、ゲームのルール、場面の遷移などに着目し、そこから状態遷移系を自動的に生成するシステムを提案する。

5.1 ゲームモデルの構造

ゲームモデルは、ゲームの場面、ルール、画面に登場するグラフィカルオブジェクトなど、ゲームシステム上に存在するオブジェクトを表現するものである。オブジェクト指向パラダイムにおけるオブジェクトの概念とも似ているが、以下の点でオブジェクトとは異なる性質をもつ。

- ゲームシステムの事象を表す
- 発生から消滅まで自律的に振る舞う
- 各々が干渉しあう
- 各々は自身の状態をもつ

1 つのゲームモデルにはメインループの 1 サイクルに 1 タスクが割り当てられている。メインループのサイクル時間は画面のリフレッシュレートと同じで、一般的なビデオゲームにおいては 1/30sec であり、その間にゲームのある場面に登場するゲームモデルはパラメータの更新を行い、必要であるならば画面へ出力する。

また、図 3 に示すように、ゲームモデルは自身のパラメータと状態、ネストされた子を持ち、一種のディレクトリ構造で表現することができる。

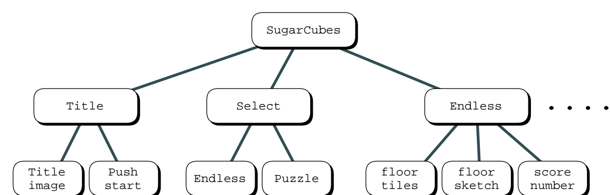

図 3: ゲームモデルのディレクトリ構造

6 ゲームモデルによる開発手法

6 節では図 4 に示すゲームに基づいてゲームモデル型のゲームシステム開発手法について説明する。

図 4 は、あるゲームシステムをゲームモデルによって表している。ゲームは Endless mode と Puzzle mode があり、Title , Mode select , Endless , Puzzle , Puzzle game over, Endless game over の 4 つの場面があり、それらをシーンと呼ぶ。ここで Mode select

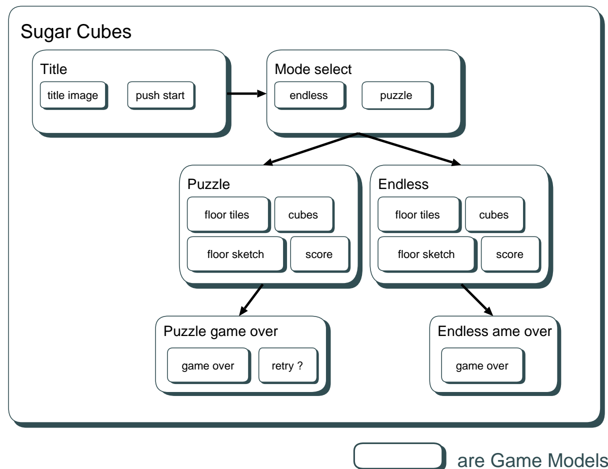

図 4: ゲームモデルの概念

シーンに注目すると, Mode select シーンは endless, puzzle の 2 つのゲームモデルがあり、2 つはスプライトと呼ばれるグラフィカルオブジェクトである。Mode select シーンから Puzzle シーン, もしくは Endless シーンへの遷移のトリガはユーザからの入力であり、あるトリガによってゲームモデルが遷移することをルールと呼ぶ。

以上を本フレームワークに提案する、モデルを用いて記述したコードを図 5 に示す.

図 5 中、model はゲームモデルを宣言するキーワードであり、TITLE シーン, SELECT シーン, さらに SELECT シーンの SL_ENDLESS ステート、SL_PUZZLE ステートの各ゲームモデルを表現している。また、transition は goto のように model によって定義されたゲームモデルへ遷移するためのキーワードである。その他は全て C と同じ構文によって表現される。

一見、ステートパターンを用いたオブジェクト指向言語のようであるが、本システムの提案する変換器により、コードは図 2 に示す C に変換される。C に変換することで変換器は C の移植性をそのまま継承できると考えられる。

7 今後

ゲームエンジンの構築においてはオブジェクトは扱い辛くオブジェクトをよりゲームシステムに特化した型でゲームモデルとして再定義する必要があった。その結果ゲームモデルはオブジェクトのようにパラメータとメソッドだけでなく自身の状態と、自身の子を含んでいるモデルであることが分かった。

しかしルールを表現するにあたり、どのゲームモデルがルールを強いるべきなのか明白に分からない問題もいくつかある。

```
model TITLE {
  if (button.start)
    transition SELECT;
}
model SELECT {
  model SL_ENDLESS {
    if (button.down)
      transition SL_PUZZLE;
    elsif (button.up)
      transition SL_PUZZLE;
    elsif (button.start)
      transition PUZZLE;
  }
  model SL_PUZZLE {
    if (button.down)
      transition SL_ENDLESS;
    elsif (button.up)
      transition SL_ENDLESS;
    elsif (button.start)
      transition ENDLESS;
  }
}
```

図 5: model による状態遷移系

8 まとめ

本システムの提案するフレームワークによれば、図 5 に示すように、model を用いて各ゲームモデルを表現し、transition によりそのゲームモデルから次に遷移するゲームモデルへの状態遷移を表現できる。これは次のような利益をもたらすと思われる。

- 移植性の向上
- リアルタイムな部分におけるシナリオの変更
- オブジェクト指向設計のインタプリタ性を除去

つまり、変換器による C への変換により、本システムは C の移植性を継承しており、シナリオとゲームシステムの状態遷移系の関連付けを容易に設計で

き、これにより従来困難とされてきた、リアルタイムな部分におけるシナリオの変更も可能にする。また、オブジェクト指向設計によりシナリオを記述し、オブジェクト指向設計のインタプリタ的な処理を変換器を用いることで、コンパイラ的なアプローチにより状態遷移系を設計することができる。

参考文献

[1] M. DELOURA, 川西 裕幸 (監訳), Game Programming Gems, ボーンデジタル (2002-9), ISBN4-939007-28-6