アクションゲーム記述に特化した言語

西森丈俊　久野　靖

筑波大学大学院ビジネス科学研究科

テレビゲームソフトウェア開発は近年大規模化しており、トライアンドエラーを繰り返しながら開発を進めることが難しくなっている。そのため開発プロジェクトでは通常スクリプト言語システムを導入するが、限られた時間で製品を開発しなければならない理由から記述システムはad hocなものになり、「新しいゲームシステムへの適用が難しい」「プロジェクトの度に最初から記述システムを作り直さなければならない」という問題が発生する。そこで、本稿ではアクションゲームを開発するのに適した分離性や効率の良い記述システムを制作することを目標とし、そのためのサーベイと制作している言語について述べる。

Action Game-Oriented Programming Language

Taketoshi NISHIMORI, Yasushi KUNO

Graduate School of Business Sciences, University of Tsukuba

As the complexity of video game software grew, it became difficult to develop them through traditional tiral-and-error processes. To overcome the problem, many farms have developed scripting languages to help game designers to modify their design quickly, without help from the programmers. However, most of these languages have ad hoc design and closely tied to specific game, making it difficult to reuse the system for other games. This paper presents a survey of existing scripting language targeted to commercial video game development, and our effort toward general, game-independent scripting language.

1　はじめに

テレビゲームソフトウェアの開発は従来から競争が激しく、常に新しい製品を開発することを要求されている。さらに、近年のハードウェアの性能向上により、より高次元な表現を実現することも重要な要件となってきている。このため開発は数年前に比べるとはるかに大規模なものとなってきており、ゲームの開発方針はできるだけ開発早期に決定することが必要となる。

一方、面白いアクションゲームを作るためには、ゲームルールやキャラクタの振る舞い等の「ゲームシステム」とグラフィクスやアニメーションやサウンドといった「表現要素」をバランスよく、しかも効率を損なわずに構成しなければならない。

アクションゲームシステムが面白いかどうかは作って遊んでみなければ検証ができず、ゲームシステムと表現要素のバランスや効率も実際に動かしてみなければ評価が難しい。このため、ゲーム開発の進行はトライアンドエラーの繰り返しとなるが、早期にゲームシステムの方針を決定したいという要求と、開発中にゲー

ムシステムについて何度も繰り返し実験を行いたいという要求を同時に満たすことは困難である。

このことを解消するために開発の現場では、開発しているゲームシステムの実装をプログラマーに頼るのではなく、スクリプト言語で記述するような開発システムを用意し、プログラマーに代わってゲームシステムの担当者が直接記述と実装を行うことが一般的な方法となっている。プログラマーの手を借りずにゲームシステムの実験ができれば、ゲームシステムを担当する人間と、表現要素を担当する人間が直接ゲーム内容についてやり取りをすることができ、トライアンドエラーで開発できる範囲を広げることになる。

しかし、限られた時間内に製品を作らなければならないことから、記述システムはゲームプログラム本体と強く結合した ad hoc なものとなり易く、記述システム部分の再利用は難しくなる。さらに、記述システムをプロジェクトの度に書き起こすため、最もトライアンドエラーを繰り返したい開発の初期段階で使用することができなくなる。

筆者らは、ゲームシステムの記述が容易な言語と、表現要素やゲームプログラム本体とは分離でき、効率を損なわないシステムがあれば、より面白いゲームやより新しいゲームを作るための有効な開発ツールとなると考え、アクションゲームを対象とした記述システムの開発を行っている。前記の議論から、アクションゲームの記述システムが満たすべき要件は次の３つとなる。

- 記述システムのゲームの他の部分からの分離性
- ゲーム実行時の効率性
- アクションゲームシステムの記述の容易さ

ここで、特にアクションゲームは以下に挙げる特徴をもつため、これらの記述の容易さが記述システムの言語に必要な要素となる。

状態遷移 アクションゲームシーン中のキャラクタは「移動中」「ジャンプ中」「物を拾っている最中」と等多くの状態を持ち、状況やプレーヤーの操作などに応じてこれらの間を遷移する。

並行性 ゲーム進行中は複数のキャラクタが自律して動作する。また「拳銃を相手に向けつつ一定間隔で撃ちながら走る」のように一つのキャラクタ内部で複数の処理が並行に動作する必要もある。

各キャラクタ間の様々な通信 ゲームシーン中の各キャラクタは相互に通信を行う。また、プレーヤーキャラクタとそれ以外のキャラクタとの当たり判定処理や、プレイ中のシチュエーション応じたリアクション処理等の相互作用も必要となる。

アクションゲームの記述が行える既存のツール類としては、次のようなものが挙げられる。しかしこれらは、上記の条件を十分に満たさないため、現状のゲーム開発プロジェクトでの使用が難しい。

アニメーションオーサリングツール ALICE[2] 等のアニメーション記述に特化した言語は、これらの言語はアニメーションを記述することに注力しており、インタラクティブなアプリケーションの記述には適していない [3]。また、グラフィクスの部分と言語システムの部分の分離はできない。

統合的なゲーム開発環境 Keel[4]、HSP[5]、Tonyu[3] 等の統合的なゲーム開発システムは、アニメーションオーサリングツールと比べてインタラクティブ性の高いアニメーションや、ゲームを制作することに特化しており、ゲームシステムの実験やプロトタイピングには十分な効果が期待できる。特に、Keel は完成度の高いライブラリを備えており、高度な開発をすることができる。しかし、アニメーションオーサリングツールの場合と同様、表現要素との分離ができないため、実際のゲーム開発プロジェクトへの適用は難しい。

記述システムエンジン Lua[6]、PSL[7] は記述エンジン単体として利用できるゲームもしくはマルチメディアアプリケーション開発用のスクリプト言語とその実装である。前提とするライブラリはなく、スクリプトを実行する仮想機械もプラットフォームに依存しないため、上記２つと比較して記述システムの分離性が高い。その反面、使用する言語が非並行手続き言語で、記述のしやすさという要素を十分に満足しない。

これらのことから筆者らは、上記の要件を満たす新しい言語の設計とシステムの制作を行っている。

以下、２章で実際に業界内で開発利用された記述システムの調査の報告とそれらの評価を述べ、３章で現在製作中の言語について解説する。

表 1:調査したシステム

名前	ゲームのタイプ	プラットフォーム
K 1	対戦格闘	家庭用ゲーム機
K 2		
K 3		
A	キャラクタアクション	家庭用ゲーム機
S 1	シューティング	携帯電話
S 2		業務用機

2 現場で使用されているシステム

言語設計とシステム製作の参考とするために、実際の現場で使用されているシステムが前述した要件をどのように解決しているかを調査した。調査はアクションゲームを開発している会社や知人にシステムに関する資料の提示を依頼する、という形式で行ったが、競争が激しく製作物の多くが社外秘となる業界のため、実際に提示していただいた数は多くなく、公表の許可いただけたのは5社6タイトルのシステム（表1）と少ない。また、本節で紹介するものについても社名やタイトル、具体的なゲーム内容については伏せて紹介する。以下、表1の順に説明し、最後にまとめを行う。

2.1 対戦格闘ゲームのシステム

ゲームK1，K2，K3は3D対戦格闘ゲームである。対戦格闘ゲームには、次のような特徴がある。

多くのアクションを持つ 多くのアクション(技や移動のアニメーション)を用意することで、遊び方のバリエーションが広がることになるため、各キャラクタは多くのアクションを持つように作られる。

複雑な状態の遷移がある アクションはそれ自体で完結しているものではなく、連続技や分岐技などのバラエティに富んだ変化をする。対戦格闘ゲームの大まかなルールは1対1で対戦し、相手をより早く倒すという単純なものであるので、ゲームに深みを持たせるために複雑なアクションの変化が必要となる。

細かいパラメータ指定が必要 アクションの個性を出しながら、アクション全体のバランスを取るために、ダメージ、当たり判定時間、硬直時間（プレーヤーが操作不能な時間帯）やキャンセル（硬直を強制的にスキップする）といった属性やパラメータはゲームシステムの担当者が容易にコントロールできる必要がある。

キャラクタは2人 基本的にゲーム内で登場するキャラクタは2人のみである。K1、K2、K3システムの言語はこのことを前提とした記述形式になっている。

対戦格闘ゲームの開発はアクションの制作と調整が中心となることから、本節で紹介する記述システムもアクションの設定や動作の記述が容易になるように特化している。

2.1.1 ゲームK1，K2

K1とK2の記述例を図1に挙げる。K1とK2は、テキスト形式でゲームシステムを記述する。どちらも1つのアクション（技）についての記述であり、実際のゲーム用に書かれたものは、これがアクションの数だけ列挙されている。

K1のシステムは、記述したテキストファイルをC言語プリプロセッサによりC言語プログラム用の数値テーブルに変換し、そのテーブルをゲームプログラム本体が読み取りながらキャラクタを動作させる。各アクションは`MH`というアクション名を引数とするマクロで始まる。アクションには次のようなマクロを記述する。

MH_HIGH 上段攻撃という属性の指定。引数にはダメージや攻撃判定時間等の攻撃に関するパラメータを取る。

mh_yarare このアクションの攻撃が成功した場合の相手のリアクションの指定。`HS`、`HD`等のパラメータはリアクションのタイプを指定している。

mh_cont_shift このアクションから他のアクションへの遷移の指定。`MIDDLE_ATK`は中段攻撃入力を、`MN_M61_H3`は遷移先のアクションを意味する。最後の`13,32`はこの遷移が許可される時間帯を指定している。遷移の指定はこのような単純なもの以外にも「相手が攻撃をガードしたら」「この攻撃がＨＩＴしたら」等、ゲーム中のシチュエーションを条件とした指定が可能である。

```
MH(mh_n07_rp2)
  MH_HIGH(PUNCH_L,
      0,12,0,35,11,
      ATF_NORMAL,TACHI,TACHI)
  mh_yarare(HS,HD,HM,HD,DA,
    80,100*DEF_TOBI,90,120*DEF_TOBI,
    66,100*DEF_TOBI,100,100*DEF_TOBI,
    100,100*DEF_TOBI,100,100*DEF_TOBI)
  mh_cont_shift(MIDDLE_ATK,MN_M61_H3,13,32)
  mh_cont_shift(1F_HIGH_ATK,MN_N07_RP3,23,32)
  mh_cont_shift(JUST_HIGH_ATK,MN_M61_H2,13,26)
  pp_sound(10, sn_2_kaze02)
```

```
[action]
name "H_JKD_STEP_LHK"
motion "H_JKD_STEP_LHK" ""
shift button 39 39
attack 18 asi_l -1 15 +1 hi none fr
yarare damage_hi_large -1 damage_hi_counter -1
use_action_table Kamae_L

[action_table]
root Kamae_L "JKD_KAMA_LOOP_NEW_L"
  file btn_a 0 "L_JKD_L_STEP_LOW_K"
    file btn_y 0 "H_JKD_L_STEP_HK" end
  end
  file btn_x 0 "M_JKD_L_SIDE_K" end
end
```

図1:K１（左）とK２（右）のシステムの記述例

pp_sound 効果音属性の指定。このアクションが始まって10フレーム目に効果音 **sn_2_kaze02** を鳴らす、という指定である。

一方、K２のシステム（図1右）は、記述したテキストをそのままデータとして読み込み、ゲーム実行時に文法解析を行い、動作が実行される。テキストは [action] 部と [action_table] 部に分かれ、[action] 部がアクションの動作の記述部分、[action_table] 部が他のアクションへ遷移するときの条件と遷移先を列挙した遷移表となっている。アクション部の属性の記述は次のようになっている。

name,motion K１の MH と同じくアクション名もしくはアニメーションデータ名。

shift 遷移についての属性の指定。button はボタン入力により遷移可能というパラメータで、39,39 は遷移可能期間のパラメータである。

attack,yarare K１と同じく攻撃属性、攻撃が成功したときのリアクション属性の指定。

use_action_table 遷移のテーブルを指定する。ボタン入力による遷移の許可は **shift** で指定するが、遷移自体の記述は [action_table] 部の遷移表で記述する。

遷移表はK１同様にシチュエーションに応じた条件が使用できるが、さらに例のように条件を階層化して細かい設定が可能になっている。

K１、K２どちらのシステムも、単純な命令やパラメータの列記による記述であるため、スクリプトを書くユーザーは、特にプログラミング言語についての経験を必要としない。また、記述システムのプログラマーにとっても、記述システムがシンプルであるためシステムの環境を充実させることが容易である。特にK２には実行しながらスクリプトを修正できるデバッグ環境が用意されている。アクションとスクリプト上の状態が一対一で対応しているので記述の構造も理解しやすい。

一方で、システムの機能追加や修正を続けていくと、複雑で細かいパラメータ指定がスクリプト内に氾濫し、末期的な状況になると誰もスクリプト全体を把握できなくなる。また、K１はエラー報告の実装が難しく、K２も実行時解釈のため、実行の前に記述間違いを発見することが難しい。

2.1.2 ゲームK３

同じ対戦格闘ゲームであるが、K３はK１，K２と異なり、テキストではなく Microsoft Excel を使い表で記述される。表は縦方向にアクション、横方向にパラメータ指定という構成になっている（図2）。Excel の表は、CSV に変換後プログラムが読める形式のデータに専用フィルタによって変換され、ゲーム実行時にデータとして読み込まれ処理される。K１やK２と比較すると、表形式のため見やすい、日本語が扱える、管理がしやすいという利点がある。いったんフィルターを通すため、実行の前に単純な間違いを報告させることができる。

逆に、アクションにバリエーションを持たせるためにパラメータを追加していくと表が大きくなり、アク

ションによっては必要のない列が増えることにもなるため、可読性が悪くなる。また、表では表現が難しい構造の記述や指定には向かない。

さらにＫ１，Ｋ２，Ｋ３全てに共通することとして、各アクションに対して用意している機能以外は書くことができない、という点が挙げられる。任意にデータを定義することや、制御フローを管理することはできず、ユーザーが新しいパラメータ指定や処理を必要とする場合、必ず記述システムプログラマーの手を借りなければならない。

2.2 キャラクタアクションゲームのシステム

前述の対戦格闘ゲームは登場キャラクタは２人と決まっていたが、以下では３体以上のキャラクタがシーンに登場するゲームのシステムについて述べる。

2.2.1 ゲームＡ

Ａはゲームシステム以外に、ゲームの間のデモンストレーションのシーン等にも使用するため、より通常のプログラミング言語に近い形式になっている。

Ａのシステムは、テキストファイルをコンパイラに通してバイナリファイルを出力する。スクリプトの実行処理は、バイナリデータをゲーム中に読み込ませ、ゲームプログラム上に実装された仮想マシンにより行われる。スクリプトはコンパイル単位で独立しており、実行時に必要に応じて他のスクリプトを読み込み、実行を切り替えることも可能である。

Ａの記述例を図３に示す。構文はC++言語やJava言語に近い。クラスはメソッドとメンバ変数により構成され、キーワード`class`の代わりに`char`[1]を使用することでクラスを実体化できる。実体化したキャラクタはクラス名そのものでアクセスが可能である。また、継承が用意されており、差分プログラミングが可能となっている。クラスは`main()`というスレッドとして実行される関数を持つことができ、実体化されると動作を開始する。この`main()`関数内でのみ待つ命令や同期命令を記述できる。メソッドの定義や呼び出しもC++言語やJava言語と同じように記述可能でき、システムに対して新しい機能を追加することも容易である。全てのメンバがpublicであるので、`char GLOBAL`というようなクラスを定義して、グローバル変数の定義もできる。

しかし、スレッドは各キャラクタに一つしかなく、状態遷移機械を記述するには`switch`文などを使ったプログラミング上の工夫が必要となる。スレッドは`main()`内にしか同期命令を記述できないため、スレッド中の処理を関数に分割する場合にも記述に工夫が必要になる。また、実体化の手段が`char`として定義するのみであるため、スクリプト初期化時以外にキャラクタを生成することができない。継承に関しても、メソッドはオーバーライドができないため、限定された用途にしか使用できない。さらに、このシステムに用意されたライブラリは開発プロジェクト中に複数のプログラマにより無秩序に追加されたため、ライブラリに一貫性が無く、システムはゲームに依存したad hocな構成となり、結果的に記述システムの分離性が悪くなっている。

[1]この言語の場合、`char`はＣ言語やJava言語のそれと違い、クラスと実体を同時に定義するキーワードである。

2.3 シューティングゲームのシステム

ゲームＳ１、Ｓ２はシューティングゲームである。シューティングゲームには、次のような特徴がある。

多くの種類のキャラクタが存在する シューティングゲームでは、数多くのキャラクタがゲームシーン中にあらわれては消えることを繰り返すため、キャラクタの生成/廃棄機構が不可欠である。

キャラクタが複数の処理を並行に処理する 単純なキャラクタの場合であれば一つの処理を繰り返すだけ、ということもあるが、高度な振る舞いをするキャラクタは複数の移動処理と攻撃処理が並行に動作する等の必要がある。

多数のキャラクタが連携することがある 小さなキャラクタが多く集まって、特定のフォーメーションで自機に迫る、というようなシーンはシューティングゲームで多く見られる。

2.3.1 ゲームＳ１

Ｓ１ではテキストファイルでゲームシステムを記述する。そのテキストファイルをコンパイラによってJava言語プログラムへ変換し、ゲームプログラム本体と一緒にコンパイルすることで、ゲームに組み込まれる。

ファイル名	種別	判定	判定モデル	値	攻撃方向	高さ	種別	...
JAB	_pnch	8	右手＋右肘	3	NORMAL	H	NORMAL	...
RLRP	_pnch	19	左手＋左肘	17	NORMAL	H	NORMAL	...
MAWAK	_kick	22	左足＋左膝	19	RIGHT	H	NORMAL	...

図 2:K３のシステムの記述例

```
char Car : Obj {
  int flag;
  void start() { flag = 1; }
  void stop() { flag = 0; }
  @main() {
    flag = 0;
    model("car_model");
    _D_WHILE (flag == 0)
    _D_WHILE_END
    play_motion("drive_car");
    _D_WHILE (flag != 0)
    _D_WHILE_END
    play_motion("break_car");
  }
}
```

```
char Director {
  @main() {
    _D_WAIT(60);
    Car.start();
    _D_WAIT(60);
    Car.stop();
  }
}
```

図 3:Aのシステムの記述例

図４の左側はＳ１のあるザコキャラクタの定義の例である。このザコキャラクタは「--------」により２つの部分にわかれており、上側は移動の処理をするスレッド、下側は攻撃をするスレッドとなっている。処理は上から順に実行され、途中の「* 数値」の形の記述は、指定時間待つ命令である。また「* after」や「* killed」はイベントハンドラで、それぞれ「全ての処理終了後」「殺されたとき」というイベント発生時の処理を記述する。イベントハンドラには他に「* damaged（ダメージを受けたとき）」や「* common」（常に呼ばれるハンドラ）がある。

このシステムの言語にはif、while、goto等の制御構造が用意されている。また、クラスにはメンバ変数やグローバル変数を任意に定義できるようになっており、比較的自由な構造の記述が容易である。コンパイラ方式のため実行前に間違いをチェックすることもできる。

一方、プレーヤー機のパワーアップシステムや、どの組み合わせのキャラクタ同士で当たり判定を行うかという重要なゲーム要素のいくつかは、このシステムがあらかじめ用意しているものであり、スクリプトで記述することはできない。キャラクタ間の通信手段についても、システム側が用意している限定されたものだけ（位置の設定やキャラクタの向き等）であるため、記述能力は低い。また、急造のシステムであるため分かりづらい文法や構造がある。例えば、キャラクタの位置を得るのは「x」や「y」というシステム変数の参照である一方、位置の設定は「pos newx,newy」と命令形で指定するという非対称性や、ループ文やif-else文の内部では処理を待つ命令が使えない等の制約が存在する。特に後者の制約のためにＳ１のシステムはループや分岐の制御文があるにも関わらず、状態遷移機械を記述するのが難しい。goto文を使うこともできるが、Ｓ１システムのgotoはBASIC言語と同じく、制御の流れを任意にコントロールできるものなので、使い方を間違えるとバグの温床になる。

2.3.2 ゲームＳ２

Ｓ２のシステムはテキストファイルでゲームシステムを記述する。そのテキストファイルをコンパイラによってＣ言語の数値テーブルへ変換し、ゲームプログラム本体と一緒にコンパイル及びリンクすることで、ゲーム本体に組み込む。テーブルはゲームプログラム

上に実装されたインタプリタ用のコード[2]として解釈実行される。

図4右はS2による、車キャラクタの定義例である。この中で`car1()`、`car1_hurt()`、`car_crash()`はこのシステムでセクションと呼ばれる処理単位で、ゲーム中のキャラクタは常にこのセクションの一つを実行する。各セクションの数値だけの行や、「5:00」という記述は「指定のワールド時間まで」待つ処理である。明示的に「`wait` 時間」と記述することで「指定時間だけ待つ」こともできる。

セクションは通常キャラクタの初期化時に指定するが、「傷ついたとき」「死んだとき」等のあらかじめシステムで用意されているイベント時の処理ルーチンとしても使用できる。図4の`car1()`セクション中にある次の2つの命令がそれに相当する。

```
suffer_section(car1_hurt)
dying_section(car1_crash)
```

さらに、セクションは任意に切り替えることが可能なため、状態遷移機械の実装や、別のキャラクタを用意してユーザーが決めた任意の条件を監視させ、条件成立時にセクションを切り替えることで、システムで用意されているもの以外のイベントハンドリングも可能である。また、他のセクションを呼び出して、処理が終わったらもとのセクションへ戻ってくる、というサブルーチンコール的な動作もできる。セクションの切り替え時には4つまで引数を渡すことができるので、パラメータを持つメッセージを送受信するような機構も記述可能である。

このシステムは記述の自由度が高く、K2のような実行中にスクリプトを変更して再実行するようなデバッグシステムも用意されている。プログラマ側からもこのシステムに対しての機能追加が容易にできるようになっており、1ゲームのために作られたシステムとしては大変よくできたシステムである。

しかし、キャラクタが複数の並行する処理を持つ構造を記述するには、シングルスレッドで記述するときと同じプログラミング上の工夫が必要である。また、ユーザーが任意にイベントハンドリングをする場合はゲーム中に直接登場しないキャラクタを作る必要があり、任意定義のイベントが多い場合、管理するキャラクタも増えることになる。

[2]コンパイラが出力するテーブルデータは機械語に近い命令語の列である。

2.4 評価

6つのゲームの記述システムの言語について、「状態機械」「並行」「通信」の記述の容易さを評価したものを表2に挙げる。

K1、K2、K3は状態機械の列挙であるが、任意の条件で状態遷移したり、任意のキャラクタ間の通信を記述することはできない（記述システムのプログラマーに用意してもらわなければならない）。また、並行処理の記述もできない。

Aは全体的にカバーしているが、状態機械は限定された関数(`main`関数)の中でしか記述することができない。スレッドは各キャラクタに一つ、という制限つきである。通信機能はメソッド呼び出しとグローバル変数の2種類で、ある程度自由度があるが、他の実体を特定する方法が、スクリプトで定義した識別子のみであるため、「自分に一番近いキャラクタに攻撃」という動作の記述をするにはテクニックが必要である。

S1はループ文中や`if`文中に他のキャラクタと同期するために処理を停止する命令が使えないため、`while`と`switch`を組み合わせて状態遷移機械を記述するテクニックが使用できない。通信もキャラクタへ直接メッセージを送る方法は位置や方向の設定というシステムに制限された範囲だけで、他はグローバル変数を使用するしかない。しかし、キャラクタ内部で複数の並行処理を記述することができる。

S2は状態遷移機械は記述しやすい。スレッドはAと同じく各キャラクタに一つ、という制限つきである。通信の記述能力がやや弱い。

AとS2以外は、言語自体がゲームシステムに依存しているため、ゲームルールの変更をすると記述システムや言語も変更しなければならない可能性がある。

逆にAとS2の記述システムは汎用性が高く、様々なゲームに適用可能である。しかし、この2つのシステムは簡単にシステムに機能追加ができる、という理由から、ゲーム開発中は記述言語に対するライブラリが無節操に追加されてしまい、システム全体としては一貫性のないad hocなものとなってしまっている。

3 開発中の言語の概要

1章で述べたように、商用のアクションゲーム開発に適したシステムはない。また、2章にあるように、

```
enemy ZAKO
  pos swidth/2, top
  dir 90deg
  speed 2.0
* 100
  turn left 45deg
* 50
  dir shipdir
* 100
  dir up
* after
  dead if out
* killed
  create BOM : pos x, y
-----------------
* 40 loop
  create TAMA : pos x, y
* 30 endloop
end
```

```
car1 () {
  hit_point ( 5 );
  model(MODEL_CAR_TAXI);
  suffer_section (car1_hurt);
  dying_section (car1_crash);
  set_posi(-30m, 0, 10m);
  set_roll(0, 0x4000, 0);
  set_move_mode (MOVE_AIM_STOP,MOVE_AIM_STOP);
  move_posi(8:00, 40m, 0, 10m);
  10:00;
}
car1_hurt () {
  move_roll ( 8, 0x400, 0x4100, 0 );
  8;
  move_roll ( 16, -0x800, 0x3f00, 0 );
  16;
}
car1_crash () {
  hit_point ( -1 );  /*** 無敵にしておく ***/
  move_posi_rel ( 3:00, 0, 0, 2m );
  move_roll_rel ( 1:00, 0, 0x2000, 0 );
  5:00;
}
```

図 4:S 1 （左）と S 2 （右）のシステムの記述例

表 2:評価表

ゲーム名	状態機械	並行	通信
K 1	△	×	×
K 2	△	×	×
K 3	△	×	×
A	△	△	△
S 1	△	○	△
S 2	○	△	△

○ 記述が容易。

△ 記述はできるが制限あり。

× 記述が難しい。

現場で使用されているシステムは再利用性が低い。本章では筆者らが開発中の、1 章での要求を満たすような新しい記述システムの言語について解説する。

本言語は、クラスのみから構成されるオブジェクト指向言語である。クラスは 1 つ以上のスレッドで構成され、スレッドはクラスを実体化した時点から実行を開始する。スレッドは 1 つの状態遷移機械を持ち、状態は実際のゲーム中の動作を定義する。スレッドやキャラクタ間の通信には後述する Tuple Space や、each 文を用いる。

図 5 が本言語の具体的な記述例である。この例の構造を図にしたものが図 6 となっている。この例では、立ち、歩き、パンチといったアクションをするロボットと、ロボットを実際に操作する 2 つの処理（レバーとボタンで操作するプレーヤーと、自律的に動く敵）が定義されている。

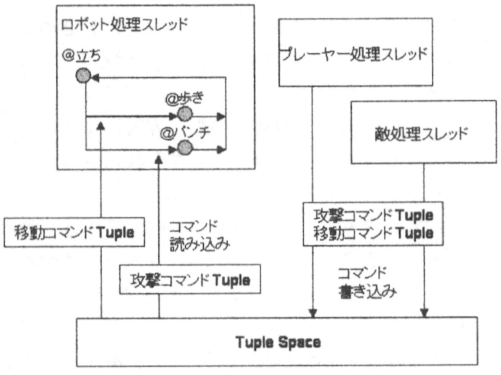

図 6:図 5 の構造図

```
ロボット
  体力:int
  @開始
    人間 ("human_model")
    体力= 100
    goto  @立ち
  @立ち
    リピートモーション ("idol_motion")
    while true
      if inp(攻撃コマンド,self)
        goto @パンチ
      elif rdp(移動コマンド,self,?)
        goto @歩く
      end
      sync
    end
  @パンチ
    モーション ("attack_motion")
    wait モーション再生中 ()
    goto @立ち
  @歩く
    リピートモーション ("run_motion")
    while rdp(移動コマンド,self,?d:float)
      振り向く (d,360)
      進む (15)
      sync
    end
    goto @立ち
==============
@プレーヤー
  in(プレーヤー開始,self)
  while true
    if レバー (0)
      out(移動コマンド,self, レバー向き (0))
    elif レバーオフ (0)
      inp(移動コマンド,self, レバー向き (0))
    end
    if ボタンオン (0)
      out(攻撃コマンド,self)
    elif
      inp(攻撃コマンド,self)
    end
    sync
  end
==============
@敵
  in(敵開始,self)
  while true
    times 4
      out(移動コマンド,self, 乱数 (0,360))
      wait 60
      inp(移動コマンド,self,?)
    end
    out(攻撃コマンド,self)
    wait 60
  end
end  #ロボット定義終わり
```

図 5:ロボットとそれを操作するサンプルプログラム

3.1 クラス

クラスは次の形式で記述する。

```
クラス名
  #クラスの定義
終わり (または end)
```

クラス内部は「======」[3]というセパレータにより複数のブロックに区切られ、1つのブロックが1つのスレッドを意味する。スレッド中の状態は「@名前」として定義する。状態内では処理は上から下に実行され、状態の最後の文まで処理が進むとスレッドは停止する。状態間の移動は「goto @状態名」による。ブロックの先頭に記述した状態がスレッドの初期状態である。スレッドは「create ロボット」のようにクラスを実体化した時点から実行を開始する。

図 5 の例において、ロボットクラスは次の 4 つの状態を持ち、各状態の処理は次のようになっている。

- @開始。初期化を行う。人間 ("human_model") はキャラクタのモデルの指定をしている。
- @立ち。何もせず立っている状態。コマンドに応じて分岐処理をする。「リピートモーション」は "idol_motion"というアニメーションを繰り返し再生する。
- @パンチ。パンチをしている状態。アニメーションが終わったら@立ちへ戻る。
- @歩く。移動状態。移動するコマンドがロボットに与えられていないなら@立ちへ戻る。「振り向く」や「進む」はライブラリで、図 5 の場合、d 度の方向へ最高 360 度/秒の速さで向きを変え、15/秒の速さでその方向へ前進する。

プログラム中に sync という命令があるが、これはキャラクタの処理を同期するための命令である。また、wait は指定条件成立中は処理を止める命令である。wait にはこのほかに、指定時間だけ待つ、という機能もある。

[3] '=' は3つ以上。

sync や wait 命令により、他の並行動作しているキャラクタに処理を切り替えゲーム全体の同期処理を行う。

図5ではセパレータによりロボットの動作の他に2つのスレッドが定義されている。一つはプレーヤーによる入力操作の処理で、もう一つはプログラムによる自動コントロール処理である。

どちらもロボットに対してコマンドを送ることでロボットを操作するが、プレーヤーのスレッドは、人間の入力に応じてコマンドを発行し、敵のスレッドは、1秒おきにランダムな方向へ4回移動後、1回攻撃、というコマンドを発行する。

スレッド中の最初の状態定義の直前にはC++言語やJava言語と同じようにメンバ変数を宣言できる。この変数はキャラクタ内部ではスレッドを問わずどこでも参照できる。例では体力を定義して、最初に１００を代入している。

3.2 Tuple Spaceによる通信

Tuple Spaceは分散システム上のプログラミング言語LINDA[1]で提案された通信システムである。Tuple Spaceを利用した通信はシンプルだが、構造化されたメッセージを選択的に受信したり、相手を特定しない通信をすることが可能なので、様々なキャラクタ同士で複雑な通信を行うアクションゲームには適している。

Tuple Spaceは Tupleという構造化されたメッセージを読み書きする場所のことを言う。Tuple Spaceへの操作には「out」「in」「rd」「inp」「rdp」があり、これらを使用することで、キャラクタやスレッドは通信を行える。各命令は次のような処理をする。

out out は Tuple Space へ Tuple を書き込む。第1引数は Tuple の識別子であり必須だが、それ以降については任意である。例えば、「メッセージ」という識別子で引数として 10 と 20 を持つ Tuple を Tuple Space に書き込む場合次のようにする。

```
out(メッセージ, 10, 20)
```

Tuple Space は識別子の重複検査を行わず、同じ内容の Tuple が Tuple Space 上に2つ以上存在できる。

in in は Tuple Space 上にある Tuple のうち、識別子と引数の数、引数の値が一致する Tuple を一つ取り除く。Tuple Space 上に適合する Tuple が無い場合、一致する Tuple が得られるまで処理を停止する。(メッセージ,10,20) という Tuple を Tuple Space から取り除く場合は次のようにする。

```
in(メッセージ, 10, 20)
```

rd rd は in と同様に Tuple Space から読み出すが、Tuple を Tuple Space から取り除かない。

inp,rdp inp,rdp は in,rd と違って、適合する Tuple がない場合にも処理を停止せず、代わりに読めたか否かを真偽値により返す。Tuple が存在するかどうかの条件式として使用できる。

```
if inp(メッセージ, 10, 20)
  print "メッセージ,10,20 を読めました"
end
```

匿名の引数 out 以外の読み込み操作は、Tuple を識別するための識別子以外に値ではなく匿名を与えることができる。例えば次のプログラムの実行後は a=3,b=1 となる。

```
out(メッセージ,1,2)
out(メッセージ,3,4,5)
in(メッセージ,?a:int,?,5)
in(メッセージ,?b:int,4)
```

3行目の in 命令は (メッセージ,int,4) にマッチする Tuple を取りに行くため、2行目で Tuple Space に書き込んだ Tuple を読む。6行目は、?のみ指定しているがこの場合、引数はあれば良い、という意味になる。

図5の例では「移動」や「攻撃」というロボットに対してアクションを要求するコマンドは、プレーヤースレッドや敵スレッドから Tuple として書き込まれ、「ロボット」により in 命令と rdp 命令で取得され処理される。各コマンド Tuple には「self」という自分を識別するための値を入れておき、コマンド Tuple が他のロボットによって処理されることを防いでいる。ロボットのコマンド Tuple の処理は次のようになっている。

- 「@立ち」状態のときに攻撃コマンド Tuple が Tuple Space にあると、「@パンチ」へ遷移する。この時、攻撃コマンド Tuple は Tuple Space から取り除かれる。

- 「@立ち」状態のときに移動コマンド Tuple が Tuple Space にあると、「@歩き」状態へ遷移する (Tuple は取り除かない)。移動コマンドが Tuple Space にある間は「@歩き」状態で移動処理をする。移動コマンドは移動方向が必要なため引数として移動方向を持つ。

Tuple Space の無い言語でこのサンプルの処理を書く場合、コマンドを受け付けるための関数や、コマンドをキューに溜める処理が必要となるが、図 5 の場合、Tuple Space への操作と if 文により、処理の流れを分断せずに記述できている。

プレーヤーまたは敵スレッドは、スレッドの先頭で (プレーヤー開始,?character) または (敵開始,?character) という Tuple を待っている。よって、プレーヤーを 1 体、敵を 2 体ゲームシーン中に登場させたいときは次のようにする。

```
out(プレーヤー開始、create ロボット)
out(敵開始、create ロボット)
out(敵開始、create ロボット)
```

この例でも、Tuple Space を使わない場合は、ロボットのクラスには、フラグ変数を用いた待ち合わせ処理をしなければならないが、Tuple Space を使う場合、トリガーとなる Tuple を in 命令で読み込む、という単純な記述で表現できる。

3.3 複数のキャラクタへのアクセス

複数のキャラクタに対して通信や処理を行うために each 文というループ文がある。これを使用することで、マルチキャストのような処理や、ある特定のキャラクタの組み合わせの処理をすることができる。

次の例は前述のロボットのサンプルに続くプログラムでロボット同士がめり込まないようにする処理を行う。

```
ロボットコリジョン
  @体コリジョン処理
    while true
      each ロボット,a b
        離す (a,b,9)
      end
      sync
    end
end
```

「@体コリジョン処理」の処理は each によりすべてのロボットの組合せについて実行される。例えば、ゲーム中に「作る　ロボット」として実体化されたキャラクタ r1,r2,r3 がいる時、次の記述は「a=r1,r2,r3」というループ処理をする。

```
each ロボット,a
 #処理
end
```

また、次の記述は「(a,b)=(r1,r2)(r1,r3)(r2,r3)」という重複しない組合せのループ処理をする。

```
each ロボット,a b
 #処理
end
```

「@体コリジョン処理」では、各ロボット間の距離が 9 以下にならないようにしている (「離す」は 2 つのキャラクタが指定距離以下なら位置を離すライブラリ手続き)。sync を含めた無限ループとなっていることで、全ロボットは常に距離が 9 以下にならないように調整され続ける。

3.4 その他

その他の雑多な特徴として次のようなものがある。

- 半角文字のある全角文字 (アルファベット、数字等) はすべて同一文字とみなす。また、「。」と「.」、「、」と「,」等も同一視する。
- while や in 等のキーワードには「条件ループ」「取る」など日本語のキーワードもある。
- 各状態内でローカル変数を任意に宣言できる。each の変数もローカル変数である。
- コンパイラは C プログラムを出力し、出力された C プログラムはゲームプログラムと一緒にコンパイルして実行される。コンパイラの実装には SableCC[9] を使用している。

3.5 今後の課題

- 現在はコンパイラが C プログラムを出力し、それを実際のゲームプログラムと一緒にコンパイル、リンクするという形式のため、ダイナミックにコードをメモリにロードすることが難しい。そこで仮

想機械を制作し、コンパイラはその仮想機械用のコードを出力する、という構成への変更を予定している。

- 記述性を優先して効率性についてはまだ研究の余地が多い。例えば、Tuple と Tuple Space を用いた通信の実装は他の言語のメソッド呼び出し等と比べて、あまり高速な方法ではない。また、仮想機械による実装の効率性はまだ検証していない。

- Tuple Space による通信の記述は自由度が高い反面、引数の型や数のエラーチェックが難しい。「取り忘れ、書き忘れ Tuple」のような論理的なエラーも発生しやすい。デバッグシステムや実行時のエラーチェックの強化で対応していこうと考えている。

- 図5の「@歩く」に見られるようなループは、スクリプトが大きくなってくると何度も現れる。また、大部分はロボットと同じだがある一部分だけ違うキャラクタを定義したい、といったことはゲームではよく見られる。この点については、継承のような機構や、マクロ等を用意することを検討している。

- 現在ライブラリとしている関数やメソッドは、特にモジュール化等はされておらず、前述したAゲームやS２ゲームと同じく、システム全体が ad hoc なものとなる可能性を持つ。この問題の解決のために、ゲームプログラム側から提供される機能を整理できるシステムが必要である。

4 まとめ

本稿では、既存のアクションゲームを開発している現場で実際に使用されたゲームシステムを記述するシステムを紹介し、アクションゲームのゲームシステムを記述する言語について述べた。言語はまだ開発途中であり課題も多く、今後更に検討と改良が必要である。

謝辞 本研究で製作した言語システムは、娯匠[4]のご協力をいただき、実際の家庭用ゲームソフト開発プロジェクトで、研究と並行して利用していただいています。

また、２章の現場のシステムの調査で、多数のゲーム企業や開発者の方々に貴重な資料やご意見をいただきました。

[4] http://www.goshow.co.jp

参考文献

[1] David Gelernter: Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems, 7(1):pp.80-112, 1985.*

[2] Matthew Conway , Steve Audia , Tommy Burnette , Dennis Cosgrove , Kevin Christiansen. Alice: lessons learned from building a 3D system for novices. *Proceedings of the CHI 2000 conference on Human factors in computing systems April 2000.*

[3] 長 慎也. 『Tonyu - アニメーション作成に特化したプログラミング言語と開発ツール』夏のプログラミング・シンポジウム pp.99-103, 2001.

[4] Keel. サイバーステップ株式会社. http://www.cyberstep.com/keel/index.html

[5] 悠黒喧史, 奥山喜正, おにたま: 「HSP2.55 Windows95/89/2000/Me/XP プログラミング入門」秀和システム (2001)

[6] The Programming Language Lua. http://www.lua.org

[7] Portable Scripting Language. http://rsn.gamedev.net/psl/

[8] ASM Software - Simple Game Scripting Engine. http://home.tu-clausthal.de/~ifmar/asmsoftware/sgse.html

[9] Etienne M. Gagnon, Laurie J. Hendren. SableCC, an Object-Oriented Compiler Framework. *TOOLS98 conference, 1998.*