アクションゲーム記述に特化した言語

西 森 丈 俊[†] 久 野 靖[†]

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」「プロジェクトのたび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，本論文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開発することを目標とし，そのためのサーベイと開発している言語について述べ，有用性を実際のゲーム開発プロジェクトでの使用から評価する．

An Action Game-oriented Programming Language

Taketoshi Nishimori[†] and Yasushi Kuno[†]

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 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. And we evaluate effectiveness of our scripting system in commercial video game development project.

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，つねに新しい製品を開発することが要求されている．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おり，テレビゲーム開発のための技術研究もさかんに行われるようになっている[3),8),13)]．

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」をバランス良く，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に頼るのではなく，スクリプト言語☆で記述する開発システムを用意し，プログラマ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と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

† 筑波大学大学院ビジネス科学研究科
Graduate School of Business Sciences, University of Tsukuba

☆ 本論文で提案する言語はコンパイル作業をともないゲームごとに必要なライブラリを用意しなければならないため，必ずしもスクリプト言語とは呼べない．しかし，業界ではゲーム企画者がゲームシステムの記述に使用する言語を，C 言語等と区別するために，スクリプト言語と呼ぶことが一般的である．本論文でもゲームシステムの記述に使用する言語をスクリプト言語と呼ぶ．

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 なものとなりやすく，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たびに書き起こすため，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．

よって，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が分離でき，なおかつ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．これらのことから筆者らは，上記要件を満たす新しい言語と処理系の開発を行っている．本論文では先行研究や現場のサーベイについて述べた後，この言語と言語の評価について述べる．

本論文は次のような構成となっている．2 章では，目標とするアクションゲーム記述システムの要件と，先行研究や既存ツールが要件を満足するかどうかについて議論する．3 章では，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る．4 章では，開発した言語について解説する．5 章では，本スクリプト言語システムを実際のゲーム開発プロジェクトで使用した結果について報告し，システムの評価をする．最後に 6 章で今後の課題について述べ，7 章でまとめを行う．

2. ゲーム記述システムの要件と先行研究

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あげる特徴を持つため，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」等多くの状態を持ち，状況やプレーヤの操作等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的に動作する．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 1 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，プレイ中の状況に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，次のようなものがあげられる．

アニメーションオーサリングツール
ALICE[2)]，Obliq-3D[9)]はアニメーション記述に特化した言語とその実行系である．キャラクタを自律的に動作させたり，複数のオブジェクトを組み合わせて複雑なアニメーションをさせたりすることができる．そのため言語はオブジェクトの階層構造や並行処理の記述が容易な設計になっている．しかし，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．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[12)]，HSP[15)]，Tonyu[14)]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よりも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．

記述システムエンジン
Lua[7)]，Small[11)]，PSL[10)]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．

アニメーションオーサリングツールはゲームシステムの記述を目標としていないため，キャラクタ間の通信の記述性は考慮されていない．また，商用の開発を考慮していないため，制作したものをアプリケーションとして配布したり，アプリケーションに組み込むためにグラフィクス部分を分離したりすることができない．

統合的なゲーム開発環境は開発の困難さを解消するために専用のライブラリを用意している．そのため，環境がライブラリに依存し，記述システムを分離することができない．

記述システムエンジンはアプリケーションの部分的な記述やカスタマイズ用という位置づけであるため，並行性やキャラクタ間通信の記述性が考慮されていない．

全体として，これら既存の研究で先にあげた要件をすべて満たしているものはない．

3. 現場で開発，使用されたシステムのサーベイ

言語設計とシステム開発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，競争が激しく制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，公表の許可をいただけたのは 5 社 6 タイトルのシステム（**表 1**）と少ない．また，本章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．

3.1 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，連続技や分岐技等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とるために，ダメージ，当たり判定時間，硬直時間（プレーヤ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を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 体 基本的にゲーム内で登場するキャラクタは 2 体のみである．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．

表 1 調査したシステム
Table 1 Survayed systems.

名前	ゲームのタイプ	プラットフォーム
K1	対戦格闘	家庭用ゲーム機
K2		
K3		
A	キャラクタアクション	家庭用ゲーム機
S1	シューティング	携帯電話
S2		業務用機

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，本章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．

3.1.1 ゲーム K1，K2

K1 と K2 の記述例を**図 1** にあげる．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とる．

`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，ゲーム中の状態やイベントを条件とした指定が可能である．

`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プログラム実行時に読み込み，文法解析を行ってキャラクタを動作させる．テキストは `[action]` 部と `[action_table]` 部に分かれ，`[action]` 部がアク

```
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
Fig. 1 Scripting examples of K1 System (left) and K2 System (right).

file 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
Fig. 2 A scripting example of K3 system.

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 遷移表を指定する．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 1 対 1 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する．末期的になるとプロジェクトメンバの誰もスクリプト全体を把握できない状況になる．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検出することが難しい．

3.1.2 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，日本語文字が扱える，管理がしやすいという利点がある．いったんフィルタを通すため，実行の前に単純な間違いを報告させることもできる．

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

```
char Car : Obj {
 void set_rotvel(float rot, float vx, float vz) {
   set_rot([0, rot, 0]);
   set_vel([vx, 0, vz]);
 }
 @main() {
   load_poly("car_model");
   play_motion("drive_car");
   set_pos([10, 0, 10]);
   set_rotvel(0, 0, 2);
  _D_WAIT(100);
   set_rotvel(45, 2, 2);
  _D_WAIT(50);
   set_rotvel(0, 0, 2);
  _D_WAIT(100);
   set_rotvel(-180, 0, -2);
  _D_WAIT_IF(get_pos().y < -100);
   hide();
   Bom.enable_bom();
 }
}
```

```
char Bom : Obj {
 int start;
 void enable_bom() { start = 1; }
 @main() {
   start = 0;
  _D_WAIT_IF(start == 0);
   load_poly("bom_model");
   play_motion("effect_bom");
  _D_WAIT(40);
   hide();
 }
}
```

図 3 A のシステムの記述例
Fig. 3 A scripting example of A System.

ションによっては必要のない列が増えることになるため，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 すべ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，という点があげられる．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の手を借りなければならない．

3.2 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．

3.2.1 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しているため，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がバイナリファイルに変換する．コンパイル後のバイナリデータはゲームプログラム上に実装された仮想マシンにより実行される．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`☆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ルーチンを持つことができ，実体化と同時に処理を開始する．「`@名前 ()` {...}」形式のルーチンは自由に定義でき，「`_D_CHANGE(名前)`」とすることで処理の流れを変更できる．これにより状態遷移機械の記述が可能になっている．

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，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．すべてのメンバが 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 1 つしかなく，複数のスレッドをキャラクタに持たせることはできない．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，メソッドのオーバ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複数のプログラマにより無秩序に追加されたため，一貫性がなく，システム全体がゲームに依存した ad hoc な構成となっているため，結果的に記述システムの分離性が悪くなっている．

☆ この char はゲームのキャラクタ（character）から来ており，クラスと実体を同時に定義するキーワードである．

```
enemy CAR
  pos swidth/2, top
  dir 90deg
  speed 2.0
* 100
  turn left 45deg
* 50
  turn right 45deg
* 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, 10m, 0, 10m);
 wait(100);
 set_roll(0, 0x0000, 0);
 set_speed(0, 0, 2);
 wait(50);
 set_roll(0, 0x4000, 0);
 set_speed(0, 2, 2);
 wait(100);
 set_roll(0, 0x0000, 0);
 set_speed(0, 0, 2);
}
car1_hurt () {
 move_roll(8, 0x400, 0x4100, 0);
 wait 8;
 move_roll(16, -0x800, 0x3f00, 0);
 wait 8;
}
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);
 wait 5:00;
}
```

図 4 S1（左）と S2（右）のシステムの記述例
Fig. 4 Scripting examples of S1 System (left) and S2 System (right).

3.3 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現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．

1 つのキャラクタが並行な複数の処理をする 単純なキャラクタの場合であれば，1 つの処理を繰り返すだけ，ということもあるが，高度な振舞いをするキャラクタは，複雑なパターンの移動と攻撃を同時処理，といったように複数の処理を並行に実行する必要がある．

多数のキャラクタが連携することがある 敵キャラクタが集まって，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．

3.3.1 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．

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 つの部分に分かれており，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「すべての処理終了後」「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ほかに「`* damaged`（ダメージを受けたとき）」や「`* common`」（つね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` 等の制御構文が用意されている．また，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，プレーヤ機のパワーアップシステムや，どの組合せのキャラクタどうし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（グローバル変数やシステムが用意しているメソッドのみ）であるため，記述能力は低い．

また，分かりにくい文法や構造がいくつかあり記述誤りの原因になりやすい．たとえば，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かかわらず，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．

3.3.2 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およ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☆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つねにこのセクションの 1 つを実行する．各セクションの数値だけの行や，wait(100); という記述は指定フレーム待つ処理である．wait(10:0); とすることで，秒単位での待ち時間指定や 5:00; と直接数値を書いて「指定のワールド時間まで」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も容易である．別のキャラクタを用意してユーザ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，システムが用意していない任意のイベント処理も可能である．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の動作もできる．セクションの切替え時には 4 つまで引数を渡すこともできる．

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良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．

3.4 比　　較

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」の記述の容易さを比較したものを**表 2** にあげる．

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，スレッドは各キャ

表 2　比較表
Table 2　Comparing table.

ゲーム名	状態機械	並行	通信
K1	△	×	×
K2	△	×	×
K3	△	×	×
A	○	△	△
S1	△	○	△
S2	○	△	△

○　記述が容易
△　記述はできるが制限あり
×　記述が難しい

☆ 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．

```
ロボット # ロボット定義始まり
  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)
    else
     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
Fig. 5 An example program of robot and controller.

ラクタに 1 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．

キャラクタへ直接メッセージを送る方法はシステムに用意されている範囲だけで，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 1 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，異なるゲームへの適用性がある．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．

4. 開発中の言語の概要

2 章および 3 章で述べた既存システムは 2 章であげた要件を十分に満たさないため，商用のアクションゲーム開発に適したものであるとはいえない．本章では筆者らが開発中の，2 章であげた要件を満たすような新しい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．

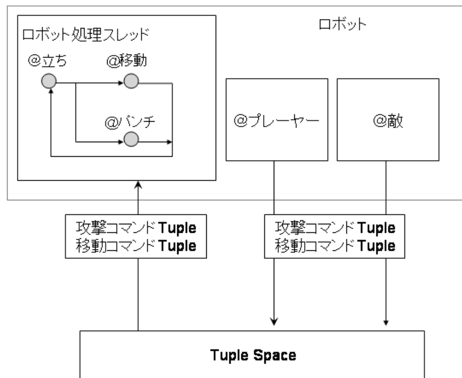

図 6 図 5 の構造図
Fig. 6 A figure of example's structure.

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と，自律的に動く敵）が定義されている．

4.1 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「======」☆☆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"）はキャラクタの表示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．

各スレッドはアニメーションでのコマにあたる「フレーム」を処理の単位とする．スレッドは 1 フレームごとに進行時間分だけの処理を完了して，他の処理の終わっていないスレッドへ制御を移譲しなければならない．他のスレッドへ処理を委譲するために，サンプルプログラム中では sync という命令を使っている．また，特定のフレーム数や条件成立中だけ処理を行わない命令に wait がある．フレーム内におけるスレッドの実行順はキャラクタの生成順で，優先順位を直接指定することはできない．

図 5 ではセパレータによりロボットの動作のほかに 2 つのスレッドが定義されている．1 つはプレーヤによる入力操作の処理で，もう 1 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．

4.2 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 [5]で提案された通信システムである．

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いう．通信システムとして Tuple Space を採用した理由は次の 2 点である．

処理の流れが分離される 汎用的プログラミング言語で通信処理を記述する場合，各キャラクタに想定される通信を受け付けるメソッドを持たせる（**図 7**）．そのため，受信側は通信に応じたメソッドが多くでき，処理の流れが自身以外のキャラクタにも渡るためプログラムを複雑にしやすい．

一方，Tuple Space を用いる場合，通信処理は送信側と受信側の双方とも Tuple Space への読み書きで記述される．よって，送信側，受信側ともに処理の流れを分断せずに記述でき，複雑な通信で

☆ 例の中の「体力：int」は変数宣言のサンプルである．ゲームではもう 1 つスレッドを作り，ダメージを受ける等の通信を受けて増減するように実装される．

☆☆ '=' は 3 つ以上．

```
public class Robot implements Runnable {
  static final int NUL = -1;
  static final int IDOL = 0;
  static final int PUNCH = 1;
  static final int WALK = 2;
  int state = IDOL, dir;

  public Robot() { model("human_model"); }

  /* 処理の委譲 */
  void sync() { /* ... */ }

  /* ライブラリ */
  void model(String name) { /* ... */ }
  void repeatMotion(String name) { /*...*/ }
  void playMotion(String name) { /*...*/ }
  boolean isEndOfMotion() { /*...*/ }
  void turn(int dir) { /*...*/ }
  void advance(int speed) { /*...*/ }

  /* メッセージ受信 */
  synchronized public void punch() {
    if (state == IDOL) state = PUNCH;
  }
  synchronized public void walk(int dir) {
    if (state == IDOL || state == WALK) {
      state = WALK;
      this.dir = dir;
    }
  }

  public void run() {
    for (;;) {
      switch (state) {
        case IDOL:
          repeatMotion("idol_motion");
          while (state == IDOL) sync();
          break;
        case PUNCH:
          playMotion("attack_motion");
          while (!isEndOfMotion()) sync();
          state = IDOL;
          break;
        case WALK:
          repeatMotion("run_motion");
          while (state == WALK) {
            turn(dir);
            advance(15);
            state = IDOL;
            sync();
          }
          state = IDOL;
          break;
      }
    }
  }
  public static void create() {
    (new Thread(new Robot())).start();
  }
};
```

図 7 Java 言語で記述したロボットの例
Fig. 7 A Java program of a robot sample.

あってもプログラムの簡潔さを維持しやすい．

シンプルで理解しやすい Tuple Space は非同期的な通信や相手を特定しない通信も記述できる機能を持ちながらもシンプルで理解しやすい．スクリプト言語を使って記述する人間はプログラミングに精通していないため，理解しやすい通信システムが必要である．

Tuple Space への操作には次にあげる「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．

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．たとえば，「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 1 つ取り除く．Tuple Space 上に適合する Tuple がない場合，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 がない場合であっても処理を停止せず，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．たと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)
```

1 行目で書き込んだ Tuple は 4 行目 in 命令の識別子と引数の数と型がマッチする．よってこの in 命令により Tuple Space から取り出される．同様に，2 行目で書き込んだ Tuple は 3 行目の in 命令がマッチする．3 行目の?のみの指定は，何か引数があればよい，という意味になる．

読み書きのタイミング 2 つ以上のスレッドが同じフレームに同じ Tuple を in により取得しようとした場合，取得できるスレッドは 1 つだけで，フレーム内でのスレッドの実行順序に依存する．同様に，あるフレームで書き込んだ Tuple が同じフレームに他のスレッドから取得できるかどうかも，スレッドの実行順序に依存する．

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．このとき，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．

図 7 のように Tuple Space のな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．

プレーヤまたは敵のスレッドは，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合せ処理をしなければならないが，Tuple Space を使う場合，トリガとなる Tuple を in 命令で読み込む，という単純な記述で表現できる．

4.3 ライブラリ

図 5 の中の「人間（"human_model"）」や「振り向く（d，360）」はゲームごとに用意するライブラリの呼び出しである．

ライブラリの実装はプログラマに任されており本言語では提供されない．ライブラリはすべてアトミックで，wait や sync のように処理を委譲する機能を持たせることはできない．

図 5 のために用意したライブラリは次のように実装されている．

人間 () 引数の名前の 3D モデルデータを表示する初期化を行う．

モーション () 引数の名前のアニメーションデータの再生を開始する．最後までアニメーションを再生したら最後の状態を保つ．

リピートモーション () 引数の名前のアニメーションデータの再生を開始する．最後までアニメーションを再生したら最初に戻る．

モーション再生中 () アニメーション再生中かどうかを調べ論理値を返す．「モーション ()」によりアニメーションが最後まで再生されると真を返す．

振り向く () 引数で与えられる方向へ指定角速度で向きを変える処理を 1 フレーム分だけ行う．

進む () 引数で与えられる速度で進む処理を 1 フレーム分だけ行う．

ライブラリの実装を行うプログラマは次のようなスクリプトファイルを作り実行時に C++関数を呼び出すようにコンパイラに指示する．

```
extern void 人間 "human" (string)
extern void 振り向く "turn" (int，int)
```

""で囲まれた名前は C 言語プログラムの関数名を指定する．C 言語側の実装は次のようにする．

```
// obj:スクリプトオブジェクト
// arg:呼び出し時の引数列
```

```
// res:返却値格納先
 void
 human(SObj* obj, Tuple* arg, void* res)
 {
   Tuple::iterator i = arg->iterator();
   const char* name = i.get_string();
   // name のデータをロード
 }
 void
 turn(SObj* obj, Tuple* arg, void* res)
 {
   Tuple::iterator i = arg->iterator();
   int dir = i.get_int();  i.next();
   int spd = i.get_int();
   // obj を dir 方向へ spd で移動
 }
```

4.4 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合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どうし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
```

上記のeach文の内部はすべての「ロボット」インスタンスの組合せについて実行される．たとえば，ゲーム中に「create　ロボット」として実体化されたキャラクタr1，r2，r3がいるとき，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，全ロボットはつねに距離が9以下にならないように調整され続ける．

4.5 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見なす．また，句点と「.」，読点と「,」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」等日本語のキーワードがある．
- 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[4)]を使用している．

5. 言語システムの評価

開発したスクリプト言語システムを，(有) 娯匠☆において複数の人間が戦うアクションゲームの初期段階の開発に使用した．プロトタイプとしてのゲームができあがった時点で，開発中に記述されたスクリプトプログラムの調査と記述にかかわった開発者へのインタビューを通じて評価を行った．

評価の基準は次の2つとした．

- アクションゲームについての言語の記述性の良さ
- アクションゲーム開発者にとっての言語の理解のしやすさ

5.1 開発プロジェクトの状況

プロジェクトは家庭用ゲーム機上のアクションゲームの開発を目的としており，本言語はその方向性や開発の方法を固めるためのプロトタイプを制作する段階に使用した．

開発期間はおよそ4カ月で，記述に携わったのは

☆ http://www.goshow.co.jp

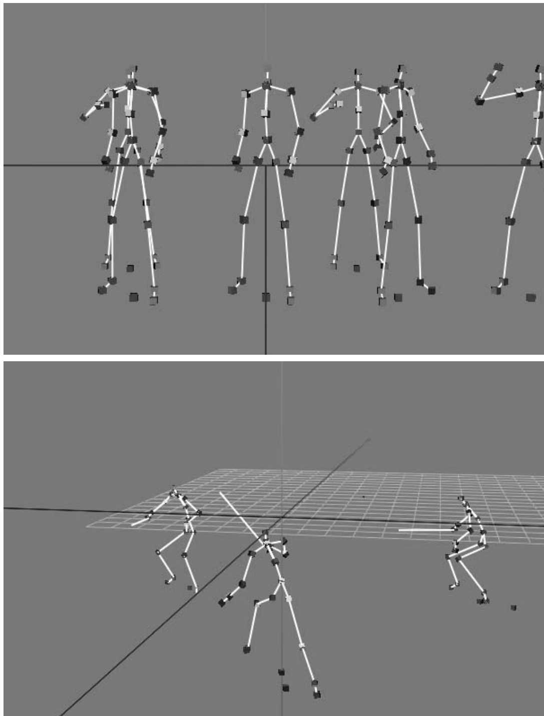

図 8 キャラクタ選択中（上）とプレイ中（下）の画面
Fig. 8 Screen shots in selecting character (upper) and playing (bottom).

ゲーム企画担当の 3 人である．3 人のうち 2 人は BASIC，Perl 等のプログラミング経験があり，変数の使い方や，GOTO，IF 文といった基本的な制御の流れとそのコントロールについての知識を持っていた．また，1 人は本言語が初めて体験するプログラミング言語であった．3 人とも並行処理を記述するプログラミングに関しては初体験であった．

図 8 は開発したゲームの画面である☆．開発したゲームは複数の人間同士が武器を持って戦う格闘タイプのアクションゲームで，プレーヤの目的は自分の攻撃技で相手にダメージを与えて倒すことである．ゲームは操作するキャラクタを選択する場面から始まり，その後ゲームプレイモードになる．敵を全員倒せば次のシーンへ移り，逆にプレーヤが敵に倒されるとゲームオーバとなり，キャラクタ選択画面へ戻る．

スクリプト言語による開発は，本言語システムの入門用のテキストと，著者らの作った図 5 のような簡単なスクリプトを企画担当者に渡し，分からないところは随時質問してもらうようにすることで開始した．

表 3 プログラム規模
Table 3 A scale of program.

行数	8,910 行（コメント，空行含む）
クラス数	32
スレッド数	2 つのクラスが 8．残りは 1 つ
状態数	平均約 5．最大 21（人間クラス）

この開発は，開発者らがそれまでに体験してきたものと違い，ゲームシステムの実装をすべて企画担当者が行わなければならない．そのため，ある程度のプログラミング技術が必要となり，使い始めの時期は言語の理解や記述方法での試行錯誤が続いた．

しかし，使用から 1 カ月程度で攻撃や防御を使い分ける簡単なプログラムを動かせるようになり，自分たちの意図するゲームを作るために必要な機能を要望する意見も出るようになった．また，企画担当者 3 人の間で「ゲーム全体の流れ」「ロボット」「エフェクトや AI☆☆」という分担ができた．各企画担当者は必要なキャラクタを記述し Tuple Space を使ってお互いのプログラムを連携させるという形態で開発が進められた．

開発の後半時期になると，ゲームシステムに関してはプログラマはまったく関与せず，企画担当者からの要求をこなしていく，という進行体制となった．ゲーム内容はつねに変化していたが，プログラマはゲーム内容についてほとんど理解していなかった．また，グラフィクスやアニメーションデータを実機上で確認するツールもスクリプト言語で記述され，開発の環境も企画担当者らがある程度自由に構成するようになった．

5.2 記述されたプログラムの調査

表 3 に開発で作られたプログラムの規模を示す．スクリプトプログラムはクラスの数に比べてやや大きい．これはサブルーチンを定義する機能が言語にないため，同じような記述を繰り返している部分が多くあるためである．

クラスと並行処理の使われ方

記述された主なクラスについて**表 4** に，2 つのスレッドが定義されたクラス 8 個についてのスレッドの使われ方を**表 5** にまとめた．

人間クラスは 2 つのスレッドを持っているが，プレーヤの入力処理や敵の思考処理のクラスからのコマンドを受けて動作するため，実際には 3 つのスレッドで 1 人の人間の処理が記述されていた．

ゲーム全体の進行は，進行を管理するクラスがキャラクタの生成や廃棄の処理を行い，ゲーム中に発生す

☆ 画面は PC 上でのテスト用のもので，実際の画面は家庭用機上で 3D グラフィクスを駆使したものとなっている．

☆☆ 業界ではプレーヤの操作によらないキャラクタのコントロールプログラムを AI と呼ぶ．

表 4　クラスの記述内容
Table 4　Discriptions of significant classes.

クラスのタイプ	記述内容
人間，剣，柱，馬車	ゲーム中に登場しアクションを行う，ゲームシステムにかかわるクラス
ゲームシステム上必要な処理	ゲーム中には直接登場しないが，ゲームシステムの処理上記述されたクラス．プレーヤの入力もしくは敵ロボットの思考の結果を人間キャラクタにコマンドとしてメッセージ送信する処理やコリジョンの処理
効果処理	カメラ，照明設定，ビジュアルエフェクト，サウンドエフェクトの処理クラス．ゲームシステムに直接かかわらないが，ゲーム中のイベントに反応するキャラクタ
ゲームシステムにかかわらないキャラクタ	キャラクタ選択時に登場する飾りの人間や，ゲームの舞台となる背景
ゲームの進行処理	キャラクタ選択処理，ゲーム全体の流れの管理

表 5　2 つスレッドを持つクラスの処理内容
Table 5　Discriptions of classes including 2 threads.

クラス	各スレッドの処理内容
人間，剣，柱	自身のアクション
	コリジョン処理から送られるメッセージ処理
馬車	自身のアクション
	人間から送られるの乗り降りのメッセージ処理
敵の思考処理	思考（AI）処理
	キャラクタの状態や位置情報等の思考処理に必要な情報の収集
コリジョンの処理	キャラクタの体どうしの排他処理
	攻撃判定処理

るイベントメッセージを受信することで進行の流れを処理するという記述がされていた．

Tuple Space の使われ方

主な Tuple Space の使われ方を**表 6** に，実行時の主な Tuple についてのアクセス量を**表 7** に示す．

現在，本言語はサブルーチンの定義や呼び出しができず，クラス外の変数へのアクセスもできないため，キャラクタ間通信や情報の共有には必ず Tuple Space が必要となり，スクリプトプログラム中にも Tuple Space へのアクセス記述が非常に多い．

そのため，取り忘れや書き忘れのバグが多く発生していた．それを防ぐために 3 人の間では Tuple の引数型や引数の数は識別子ごとに完全に決めていたようで，同じ識別子で違う型や数の引数をとる Tuple は使われていなかった．

このゲームでは相手の状態に応じた処理やアクションをするルールが多く盛り込まれている．プログラム中では各キャラクタがつねに Tuple Space 上に自分の状態を書き込み，他のキャラクタは次にあげるような記述でその Tuple をチェックして適した処理を選択する，という記述箇所が多く見られた．

```
each ロボット, A
 if A!=self and
     !inp(状態, A, "死亡") and
     !inp(状態, A, "無敵") and
     inp(状態, A, "ダウン")
   # ...
 end
end
```

また，状態 Tuple だけではゲームシステム上でのキャラクタの状態を網羅できず,「状態 2」という Tuple も使われていた．

上記の例では，可読性のために引数に文字列を使用している．このため実行時は文字列のコピーや比較が頻繁に発生しており，効率性を損なう原因となっている．

敵ロボ Tuple は Tuple Space 上にある間，敵が存在していることを意味するが,「全敵ロボについての処理」をするために次のように記述すると，Tuple Space 上から敵ロボ Tuple が消えてしまい，以降フラグとし

表 6 Tuple Space の主な使われ方
Table 6 Uses of Tuple Space.

使われ方	Tuple の識別子	内容
グローバル変数	ターゲット	カメラが注視するキャラクタを保持
	選択キャラ	プレーヤの選択したキャラクタ名を保持.
	プレーヤロボット	プレーヤキャラクタを保持
キャラクタの状態を公開	状態	「攻撃中」「死亡」等のキャラクタの状態を保持．外部からキャラクタの状態を調べるために利用される.
キャラクタ存在フラグ	敵ロボ	敵キャラクタを保持する．敵の数だけ Tuple Space 上にある
メッセージ送信	コマンド	人間キャラクタに「移動」「攻撃」等のコマンドをメッセージとして送信する.
	攻撃判定，攻撃当たった	コリジョン処理へ攻撃中を通知する，もしくはコリジョン処理から攻撃があたったことを通知するメッセージ
	カメラ揺らし	カメラへのメッセージ.
	選択	キャラクタ選択処理からイベントへの選択決定メッセージ.
	イベント終了	ゲームが終了したときにゲーム進行の管理処理へ送られるメッセージ.
クラス変数	表示人数	あるクラスのインスタンスを生成するたびにインクリメントされる変数．キャラクタの位置設定に使用している.
初期化時の引数	位置情報，誰	キャラクタ生成前に Tuple Space に書き込み，生成後の初期化処理で取得するパラメータ.「位置情報」は初期位置,「誰」は表示するモデルデータ名を引数に持つ.

表 7 実行時の主な Tuple のアクセス量
Table 7 An access measure of significant Tuples.

Tuple の識別子	OUT	READ	HIT	HITRATE	LEFT
ターゲット	82	230	134	0.5826	0.7035
選択キャラ	4	60	4	0.0667	0.8687
プレーヤロボット	2	33733	33677	0.9983	0.8664
状態	1909	1057087	40365	0.0382	5.2192
敵ロボ	6	3864	2058	0.5326	0.3238
コマンド	16061	214676	53174	0.2477	1.6599
攻撃判定	437	12483	437	0.0350	0.0315
攻撃当たった	67	60322	112	0.0019	0.6861
カメラ揺らし	19	18919	19	0.0010	0.0000
選択	3	19248	3	0.0002	0.0000
イベント終了	2	527	2	0.0038	0.0000
表示人数	12	12	12	1.0000	0.0007
位置情報	20	44	20	0.4545	0.0000
誰	2	11	2	0.1818	0.0000

計測は 13849 フレーム.
プログラム中で使用されていた Tuple は全部で 111 種類

OUT out の回数
READ Tuple Space へ読みにいった回数
in，rd で処理が停止している間も数える.
HIT READ が成功した回数
HITRATE READ が成功した割合 (HIT/READ)
LEFT 1 フレームの処理終了後に残っていた Tuple 数の平均

表 8 主な意見
Table 8 Principle opinions.

言語の印象	IF や GOTO があって BASIC のようだった.
	体で覚えた（プログラミング初心者）.
開発進行について	プログラマに頼まずにゲームシステムの実験ができた．逆にどこまでスクリプトで書いてもよいのかが分からない.
	キャラクタの操作やカメラの実験はやりやすかった.
	スクリプト全体に影響を与える変更はすぐにはしにくい.
	スクリプトを自由に書けるためには技術や経験が必要.
並行処理について	処理を追いかけてバグを見つけるのが難しい.
	クラスに分けるかスレッドに分けるか 1 スレッドで処理するかで迷う.
	1 キャラクタ内での複数の並行処理という考え方をしたことがなかった.
Tuple Space について	Tuple の上書き機能が欲しい.
	ブロードキャストとして正しく動作する機能が欲しい.
	書いた Tuple が次のフレームにならないと受信されない点が問題になることがある.
	値のみのマッチングではなく，条件式によるマッチング機能も欲しい.
	スレッドの処理順を指定したい.

て使えなくなる問題がおこる[1].

```
while inp(敵ロボ, ?A)
  # 敵ロボ (A) に対する処理
end
```

そのため，次のような冗長な記述をする箇所も見られた.

```
each ロボット, A
  if rdp(敵ロボ, A)
    # 敵ロボ (A) に対する処理
  end
end
```

カメラ揺らし Tuple はフィールドの柱が倒れたとき等にカメラを振るわせるために使われるメッセージ用の Tuple である．カメラをコントロールする処理はつねにこの Tuple が書き込まれたかどうかを見張っている．このため書き込み回数に比べ極端に読み出し回数が多い．このような使われ方の Tuple も多く見られた．また，カメラ揺らしメッセージは送信後ただちに受信，処理されるようになっており，1 フレーム処理終了後に残っていることはない．生成時に初期化のためのパラメータとして使われる位置情報，誰 Tuple も同様である.

その他の記述の傾向

- ほぼすべての識別子は日本語文字で記述されていた．予約語についても英字，日本語文字の両方を用意していたが，おおむね日本語文字の方が使われている．アルファベットについては全角を使う人と半角を使う人がいたが，1 人で両方使っていることはなかった.
- メンバ変数の多くが，GOTO による状態遷移時に状態間でのパラメータの受け渡しに使われていた.
- 人間キャラクタは CSV データをスクリプト上から利用できるライブラリを用いて，データ主導のコントロールをするようになっていた．CSV データにはコマンドに対応した技データの名前や，技のダメージ量等が含まれており，スクリプトプログラムはそのデータから人間キャラクタの動作を決定していた.
- 3 つ以上のスレッドが 1 つのクラスに定義された場合誤動作するバグがシステムにあったため，スレッドを 3 つ以上使用することを避けていた.

5.3 インタビュー

インタビューは人数が少ないこととできるだけ多くの情報を得るために，ミーティング形式とした．イン

タビューに参加したのは，スクリプトプログラムの記述を担当した3人と，バランス調整作業をしたデザイナ1人の4人である．主な意見を**表8**にあげる．

言語の理解しやすさ

プログラミング経験のあった2人にとっては「IFやGOTOがあってBASICのようだった」という意見が出されたことから見て，基本的な処理や状態遷移の記述は理解しにくいものではなかったといえる．キャラクタが自律的に動作しながらお互いに通信しあってゲームが進むという点についても，ゲームはそういうものだ，という認識があるため自然に受け入れられたようだ．Tuple Spaceも簡単な説明とサンプルで十分に理解できたようで，プログラミング経験のあった2人からは言語機能の理解が難しかったという意見はなかった．

一方，プログラミング初心者であるもう1人は，「体で覚えた」という答えのとおり，かなり試行錯誤を繰り返しながらなんとか理解を進めていたようである．

開発進行について

プログラマに頼まずにゲームシステムの実験ができた，という意見や，キャラクタの細かい操作システムやアクション中のカメラの動きを簡単に実験ができた，という意見が得られた．これらの意見からキャラクタの操作やカメラは実験が頻繁に繰り返されたことが分かる．

しかし，実験のための変更がスクリプトプログラム全体に影響を及ぼすような場合はすぐに実験というわけにいかなかった，といった意見や，スムーズな開発のためには記述テクニックや経験が必要という意見もあり，必ずしも意図する実験をすべて自由に行えたわけではなかった．

また，自由度が高くなった分，スクリプトでの実装とプログラマによるC言語での実装の境界線がはっきりしない，という意見もあった．これは筆者らも含めてゲームシステムすべてをスクリプト言語で記述する開発形態が初めてであったため，手探り状態で開発を進めていたことが理由である．

並行処理について

並行処理を使って記述したプログラムのバグの修正や，記述したい動作をどのようにスレッドに割り当てるかという点が難しい部分だったようである．

また，お互いに通信しあうキャラクタのうち1つだけを消してしまうというバグにも悩まされていたようで，この点を解消するために，敵ロボTupleのようなキャラクタが存在するかどうかを保持するTupleを用意して適切にキャラクタが消されるようにする記述をしていた．

フレーム内でのスレッドの処理順序を指定したいという意見もあった．これは，アクションゲームではゲーム進行に対してキャラクタの反応が少しでも遅れるとゲームバランス上致命的となってしまうので，処理とメッセージの流れをコントロールしたいという理由による．

Tuple Spaceについて

Tuple Spaceシステムはスクリプトプログラム中で頻繁に使用されており，注文や問題指摘も多かった．

Tupleの上書き機能が欲しいという意見は，グローバル変数として使用するTupleの書き込みは必ず一度`in`して`out`しなければならないという管理上の面倒さが理由となっている．

ブロードキャスト機能は，ある処理で`in`しているTupleを他の処理でも必要となった場合に，2つの処理とも正しく`in`することができない[1)]，という理由からあれば便利という意見としてあげられた．

スレッドの実行順によっては`out`した次のフレームでないと`in`できない通信遅延の問題の指摘は，スレッドの処理順序を指定したいという意見と同様，処理の遅延によるゲームバランス上の問題から出された意見である．

その他

その他次のような意見が得られた．

- ローカル変数は積極的には使わなかった．状態間やスレッド間で共有できるメンバ変数の方が多く使用した．
- 実数の変数だと思って代入していたら整数型で困ったことがよくあった．
- 「キャラクタ型なので…できません」といったコンパイル時エラーでデータに型があることを理解した（プログラミング初心者の話）．
- 配列変数が欲しい．特に列ごとに型の違う配列が使いたい．
- Tuple Spaceの通信遅延の問題はプログラムを動かしてみて気がついた．その他にもコンパイルしたり動かしたりして初めて理解することが多かった．
- キャラクタの「状態」に応じた処理を記述したい場合が多いので，他のキャラクタのメンバ変数にアクセスしたい．
- 実行しているときに，どのキャラクタが生きているかという情報が欲しい．
- 処理が軽くなるならどんな手段でも使いたい．
- 多種多様な通信に1つのTuple Spaceを使用して

いるので通信を構造化するために Multiple Tuple Space[6]システムが必要か尋ねたが，特に興味を持った様子がなかった．

5.4 システムの評価

開発で作られたスクリプトプログラムの調査とインタビューから次のような評価ができる．

- 言語の構文や機能は素直に受け入れられていた．実際の記述でも実装した機能はほぼすべて使われていた．
- 開発の状況やインタビューから，トライアンドエラーの幅を広げるという目的はある程度達成されたと考えられる．しかし，より自由な開発のためには記述テクニックや経験が必要である．
- サブルーチンを定義できず，記述プログラム中には冗長な部分が多い．
- 並行処理による記述はバグを特定することが難しく，そのための環境が整備されていない．
- Tuple Space による通信の理解は難しくない．しかし，グローバル変数，キャラクタの状態の取得といった記述の冗長さや，通信遅延やブロードキャストの記述が難しい，という問題がある．また，Tuple Space へ書き込む Tuple は識別子によって引数の型や数を決めていたことから，識別子ごとに引数の型や数を宣言することで，コンパイル時に間違い検出ができる機能の導入が考えられる．
- 状態 Tuple の氾濫を防ぐために，各キャラクタの状態を Tuple Space なしに参照できる機能か，現在の状態遷移の記述も含めた包括的な状態を取り扱える機能が必要である．
- 「処理が軽くなるならどんな手段でも使いたい」という意見があるように，より効率の良いシステムが要求されている．特に可読性のために文字列を使用している箇所が多く，定数や列挙型といった機能が必要となっている．
- ドキュメントが整備されていないため，実際に動かさないと分からないことが多い．しかし，なんとか動くシステムといくつかの動くサンプルがあれば，初心者であっても教科書や先生なしでプログラミングは習得できる（場合もある）．

調査したスクリプトプログラム中の記述の冗長性や，インタビュー時の意見から，本スクリプト言語システムにはゲームシステムの記述の容易性という面でまだ問題点が多い．しかし，ゲームを作れないという意見はなく，問題点を解決していくことで，より改善することが可能である．また，開発状況やインタビューからトライアンドエラーの幅を広げる効果を確認できた．

以上のことから，未完成だが改善すべき余地が多くあり有望なアプローチであるといえる．

6. 今後の課題

評価にあげた問題点の解決以外に次のような課題が残されている．

- 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とリンクするという形式のため，分離性のために機械独立なものにすることが難しい．そこで仮想機械を開発し，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．
- 現在のスクリプト言語用ライブラリは，コンパイラが出力した C プログラムにライブラリ関数呼び出しコードが埋め込まれるため，他のプログラマから自由にライブラリを設計し，実装することが難しい．ゲームプログラム側から提供される機能を容易に組み込んだり差し替えたりできる機能が必要である．
- 今回のゲーム開発プロジェクトでの使用だけでは，他のタイプのゲームの開発での有用性が評価できない．より広い範囲に適用できるシステムとするために，格闘タイプとは違ったタイプのゲーム開発プロジェクトでの適用が必要である．

7. ま と め

本論文では，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．また，開発した言語を現場の開発プロジェクトに使用して評価を行った．システムは有望ではあるが課題も多く，今後さらに検討と改良が必要である．

謝辞 本システムは（有）娯匠のご協力をいただき，ゲーム開発プロジェクトで使用していただきました．

また，3 章の現場のシステムの調査に関しては，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．

最後に本論文の内容について有益なご指摘をいただいた差読者の方に感謝いたします．

参 考 文 献

1) Rowstron, A. and Wood, A.: Solving the Linda multiple rd problem using the copy-collect primitive, *Sciences of Computer Pro-*

gramming (1997).

2) Conway, M., Audia, S., Cosgrove, T.B.D. and Christiansen, K.: Alice: lessons learned form building a 3D system for novices, *Proc. CHI 2000 conference on Human factors in computing systems* (2000).
3) Deloura, M.A.: *Game Programming Gems*, Charles River Media (2000).
4) Gagnon, E. and Hendren, L.: *SableCC — an object-oriented compiler framework* (1998).
5) Gelernter, D.: Generative Communication in Linda, *ACM Trans. Prog. Lang. Syst.*, Vol.7, No.1, pp.80–112 (1985).
6) Gelernter, D.: Multiple Tuple Space in Linda, *Proc. Conf. on Parallel Architectures and Languages Europe* (*PARLE 89*), Vol.365 of Lecture Notes in Computer Sciences, No.1, pp.20–27 (1989).
7) Ierusalimschy, R., de Figueiredo, L.H. and Celes., W.: Lua-an extensible extension language, *Software: Practice & Experience*, Vol.26, No.6, pp.635–652 (1996).
8) Moller, T., Hainess, E. and Akenine-Moller, T.: *Real-Time Rendering* (*2nd*), A K Peters Ltd. (2002).
9) Najork, M.A. and Brown, M.H.: Obliq-3D: A High-Level, Fast-Turnaround 3D Animation Systems, *IEEE Trans. Visualization and Computer Graphics*, Vol.1, No.2, pp.175–193 (1995).
10) Porter, B.: Portable Scripting Languages. http://www.rsn.gamedev.net/psl
11) Riemersma, T.: THE SMALL SCRIPTING LANGUAGE, *Dr. Dobb's Journal* (1999).
12) サイバーステップ株式会社：Keel. http://www.cyberstep.com/keel/index.html
13) 宮沢　篤，駒野目裕久：アーケードゲームのテクノロジ，プログラミングシンポジウム，Vol.37, No.17, pp.175–193 (1996).
14) 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).
15) 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).

(平成 15 年 2 月 18 日受付)
(平成 15 年 7 月 9 日採録)

西森　丈俊（学生会員）

1969 年生．1994 年東京理科大学理工学部数学科卒業．同年ナムコ（株）入社．家庭用ゲーム開発に従事．1996 年よりドリームファクトリー（株）入社．2001 年よりフリーランス．

久野　　靖（正会員）

1956 年生．1984 年東京工業大学理工学研究科情報科学専攻博士後期課程単位取得退学．同年東京工業大学理学部情報科学科助手．筑波大学経営システム科学専攻講師，同助教授を経て現在同教授．プログラミング言語，プログラミング環境，ユーザインタフェース，情報教育に興味を持つ．著書に『UNIX による計算機科学入門』（丸善），『入門 JavaScript』（アスキー）等がある．日本ソフトウェア科学会，ACM，IEEE Computer Society 各会員．