ソフトウェア工学 73－7
（1990．7．19）

動画生成のための並行動作モデル

宮本雅之　花田恵太郎　吉川耕平

シャープ株式会社
コンピュータシステム研究所

コンピュータ・アニメーションにおける動きの表現のため、アニメーションを複数登場物の並行動作系ととらえるモデルを提案する。また、このモデルに基づくアニメーション記述言語とその処理系(Easy: Event-driven animation system)の紹介を行う。このモデルの特徴は、登場物の動きの発生やストーリの展開をイベント駆動により表現する点にある。これにより、i)登場物の動きの同期の表現が容易になり、ii)実時間アニメーションにおける、ユーザ等の他システムと同期したストーリ展開の表現が可能となる。

A MODEL OF CONCURRENT MOTIONS FOR COMPUTER ANIMATION

Masayuki MIYAMOTO　Keitaro HANADA　Kouhei YOSHIKAWA

Computer Systems Laboratories
SHARP Corporation

2613-1, Ichinomoto, Tenri, Nara 632, Japan

A model for representing motions in computer animation is introduced. We consider an animation to be a system of concurrent processes, and propose a new model for representing the processes and their synchronization. The main distinctive feature of the model is its message sending mechanism, which enables the construction of animation stories based on non-deterministic motions of highly independent actors - an actor is taken to be an object with internal states and its own motions. An animation description language based on the model is also introduced. This work, we believe, enlarges the scope of applicability of computer animation.

1.はじめに

キーフレーム法等のコンピュータアニメーション作成で一般的に用いられている手法によると、アニメーションのストーリ展開が固定的であり、ストーリへのユーザの参加や他システムとの同期、あるいは動きのシミュレーション等の表現が不可能であるという問題点がある。我々は、動きを記述し、その記述からアニメーションを生成することによりこの問題点を解消する。

アニメーションを記述･生成するためには、

i) 形状モデリング
ii) 登場物やカメラの動きと同期の指定
iii) レンダリング
　　陰面処理、陰影処理、…

といった技術が必要となる。i)、iii)に関してはCG技術の蓄積があり[1, 2]、ii)に関しては[3, 4, 5]等の研究がある。本論文では、ii)に関して検討した新しいモデルの紹介を行う。

2. 系の時間発展の記述技術

前章のii) 登場物やカメラの動きと同期の指定技術は複数登場物からなる系の時間発展の記述技術ととらえることができる。

2.1. 時間発展の記述方式 I

時間発展の最も直接的な記述方法は、系の状態を決定する状態変数の値の時間変化を時間の関数として指定する方法である(図1参照)。ここで時間tは離散的な値(整数)を取るものとする。キーフレーム法に基づくアニメーションはこの範疇に入る。

しかし、この方法では、

i) 登場物の内部状態の変化
　例えば、ある登場物の速度が一定値を越えた
ii) 登場物間の相互関係の変化
　例えば、二つの登場物が衝突した

といったイベントを発生時刻に置き換えて記述する必要があり、記述の持つ意味が不明確になるだけでなく、ストーリの局所的な変更が時間軸の大域に渡り影響するという問題点を持つ。

また、

iii) 系の外部(ユーザ等の外部系)からの刺激

に対応した動きの表現が不可能である。

2.2. 時間発展の記述方法 II

上記i)、ii)、iii)のイベントを表現可能な記述方法として、図2(1)式に示す方法がある。fは時刻tにおける状態x(t)からi)、ii)のイベントの発生を検知し、時刻t+1における状態x(t+1)を決定する。exは、外部から入ってくる刺激(上記iii)のイベント)である。図2(1)式は状態変数毎に記述すると図(2)式のようになるが、$x_i(t+1)$に$x_j(t)(j \neq i)$が関係しており、各x_iが従属しあった記述となっている。

$x(t) = m(t)$

あるいは

$x_i(t) = m_i(t)$
$(i = 1, 2, \cdots, n)$

ここで、tは離散時間パラメタ

図1. 時間発展の記述方式 I (直接的表現)

記述の独立性を高め、時間発展記述の容易性、再利用性を高めるため、本論文では図2(3)式に示すような記述モデルを紹介する。すなわち、$x_i(t+1)$は$x_i(t)$とtにおいて存在するイベント(上記i)、ii)、iii)のイベント)のみから定まるものとする。言い換えると、アニメーションを複数の登場物(x_i)がイベントevにより同期を取りつつ並行動作を行う系ととらえて記述を行う。ただし、evの発生機構に関しては図2(3)式とは別に用意する必要がある。

(1)　$x(t+1) = f(t, x(t), ex)$

(2)　$x_i(t+1) = f_i(t, x(t), ex)$
$(i = 1, 2, \cdots, n)$
exは系外部からの刺激

(3)　$x_i(t+1) = f_i'(t, x_i(t), ev)$
$(i = 1, 2, \cdots, n)$
evはイベント(exも含む)

図2. 時間発展の記述方式 II
(イベントをとらえた記述方式(方式 I を含む))

3. 関連研究とモデルへの要件

3.1. 関連研究

時間発展の記述方式IIの範疇に入るアニメーション記述に関する研究として、Director [3] やASAS [4]、Paradise [5] 等がある。

ASASはLispを基にしたアニメーションの記述･生成システムである。アニメーション登場物の動きは、一単位時間に一度実行される手続きとして記述する。手続きには他の登場物へのメッセージ送付式が記述可能である。メッセージ送付式の役割は、送信先登場物に対してメッセージ名と一意に対応した手続きの起動を要求することにある。

一方、Paradiseのモデルでは、登場物は環境察知能力を持ち、自分の回りに存在する登場物やメッセージを能動的に取り込むことができる。

登場物は、回りの環境から取り込んだ情報と自分の状態から次の行動を自主的に決定する。

3.2. モデルへの要件

上で述べた、ASAS、Paradiseにおけるメッセージ通信は以下のように異なるモデルに基づいている。

i) 要求メッセージ通信

ASASでは、Smalltalk 80[6]に代表されるいわゆるオブジェクト指向言語におけるメッセージ通信を採用している。メッセージの送信登場物(あるいは、記述を行う人)は、受信登場物の行う動作を知っていて、メッセージ送信により仕事の要求を行う。メッセージを受信した登場物は、メッセージ名から一意に定まる手続きを実行する。

要求メッセージ通信は、メッセージとその効果が一意に対応しており、固定的なストーリ展開をもつアニメーションの記述に適する。しかし、このモデルでは、登場物の動きは、要求の受理により起動されるため、自律的な動きの表現には適さない。また、要求メッセージ送信には送信先を指定する必要があるため、送信側の記述が受信側に従属してしまう。このため、記述の独立性が失われ再利用性が低下する。

ii) データの放送([7])

Paradiseでは、メッセージ送信の目的はメッセージ中のデータを転送することにある。メッセージ名から一意に定まる手続きは存在せず、受信登場物はメッセージの内容を自由に操作でき、自身の行動規則に従い行動の決定をする。メッセージは特定の登場物に向けて発信されるものではなく、大域的に放送され任意の登場物が受信可能である。

これによりこのモデルは、登場物の自律的な動きの表現に適する。しかし、行動規則の発火の順序制御が困難であるため、固定的なストーリ展開をもつアニメーションの記述には適さない。

以上で述べたように、要求メッセージ通信モデルとデータの放送モデルは、それぞれ補完しあう長所と短所を持つ。

我々は、固定的なストーリ展開の表現能力と、自律的な動きの表現能力の双方をアニメーション記述言語への要件とした。以下では、要求メッセージ通信とデータの放送の双方の通信機構を持ち、双方の長所を併せ持つモデル述べる。

4. 並行動作モデル

この章では、アニメーションを複数登場物の並行動作系ととらえるモデルについて述べる。主要な概念は、動作を行う単位であるアクタと、アクタ間の通信手段であるデマンドとイベント、及び、動作の基本単位時間としてのティックである。

4.1. アクタ

アクタは動作を行う基本的な単位であり、アニメーションは並行に動作するアクタの集まりである。アクタは位置や形状等の属性と、属性の変化の仕方等を定めたメソッドを持つ。またデマンドを保持するデマンドプールを持つ。メソッドはデマンドによる起動要求があるとプロセスとして立ち上がる。アクタは複数のプロセスを同時に(同一ティックに)複数走行可能である。起動されたプロセスはアクタが持つプロセスリストに登録され走行状態となり、終了のデマンドを受け取ると削除される。プロセスリストにあるプロセスについては、各ティック毎に、対応するメソッドに記述された手続きが実行される(図3)。

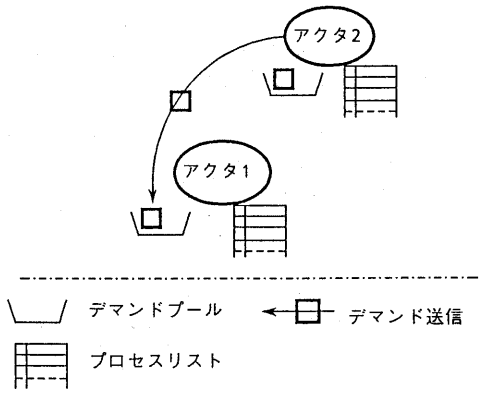

図3 アクタとデマンド

4.2. ティックとアクタの動作

ティックは系の大域的な時計の最小単位であり、全てのアクタは1ティックに1動作を行う。アクタの1動作は、デマンド処理とプロセスの実行からなる。

i) デマンド処理

アクタはデマンドプールに届いている各デマンドに対して以下の処理を行う。

●プロセス起動デマンドの場合

対応するメソッドがプロセスとして走行していないならばプロセスの起動(プロセスリストへの登録)を行う。

対応するメソッドがプロセスとして走行しているならば、そのプロセスを終了(プロセスリストからの削除)し、新たに起動しなおしを行う。

●プロセス終了デマンドの場合

対応するメソッドがプロセスとして走行しているならば、そのプロセスを終了する。
対応するメソッドがプロセスとして走行していないならば、何もしない。

ii) プロセスの実行
プロセスリストにある各プロセスの実行、すなわち対応するメソッドに記述された手続きの実行を行う。

4.3. デマンドとイベント

デマンド、イベントともにアクタ間の通信を行うための概念であり、共に名前と任意個のパラメタ並びからなるメッセージを他のアクタに届けるために用いられる。デマンドは前章で述べた要求メッセージ通信を行い、イベントはデータの放送を行う。

4.3.1. デマンド

デマンド通信(図3)は送信先アクタに対してプロセスの起動·終了要求を行うものであり、以下の特徴を持つ。

●送信先を指定する1対1通信
デマンド送信では送信先アクタを指定してメッセージを送出する。送信先として指定できるのは1つのアクタだけである。

●デマンドの処理は1ティック後
デマンドは送信先アクタのデマンドプールに蓄えられ、送信ティックの次のティックの動作時に処理される。

●プロセスの起動·終了要求
デマンド送信に用いられるメッセージの名前は送信先アクタの持つメソッドに対応し、パラメタ並びはメソッドの引数に対応する。

デマンドの送信の形式を以下に示す。

i) プロセス起動要求
send(送信先アクタ名,メソッド名,
パラメタ並び)

ii) プロセス終了要求
remove(送信先アクタ名,メソッド名)

4.3.2. イベント

イベント通信(図4)は以下の特徴を持つ

●宛先を指定しない通信
イベントは他のアクタに対してデータ(例えばアクタの内部状態)を送信したり、動作のタイミング(アクタの衝突が起こった)を知らせたりするために使われるメッセージの通信である。イベントは、全てのアクタからアクセス可能な大域的なイベントプールに送信され、保持される。

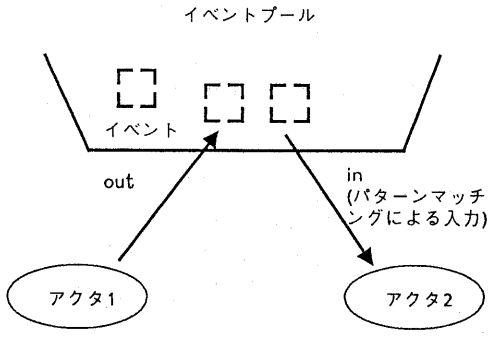

図4　イベント通信

●パターンマッチングによる読み込み
イベントプールからのイベントの読み込みは、名前とパターン並びを指定して行う。ここでパターン並びとは、値と変数から成る列であり、イベント中から名前が同じであり、さらにパターン並びと順番通りに合致する(値と値はそれらが同じ値なら合致する。変数と値はそれらの型が同じなら合致する。)イベントを読み込む。イベントプール中にパターンが合致するイベントが存在しなければ、読み込みは失敗する。パターンが合致するイベントが複数個存在する場合はそれらの集合が読み込まれる。イベントは読み込みによりイベントプールから削除されず、複数アクタにより読み込み可能である。

●イベントの読み込みは1ティック後のみ可能
イベントはそれが送信されたティックの次のティックでのみ読み込み可能であり、そのティックのすべてのアクタの動作後はイベントプールから消滅する。

イベント通信の形式を以下に示す。

i) イベント出力
out(イベント名,パラメタ並び)

ii) イベント入力
in(イベント名,パターン並び)

5. アニメーションの記述と実行
(Easy: Event-driven animaton system)

5.1. アニメーションの記述

前章で述べたモデルに基づくアニメーション記述言語を概説する。

複数アクタからなる系の時間発展の記述には、

·登場するアクタ
·アクタの行う動き
·アクタの動きの開始と終了タイミング

を記述する必要がある。以下では、これらをモジュール性高く記述するために、シナリオ、キャラクタ、ウオッチャという3種類のアクタを導入する。

各アクタの記述は二つの部分に分かれる。一つはアクタの属性の宣言部であり、もう一つはメソッドの定義部分である。メソッドはプロセスとして実行される動きの記述であり、各ティックに逐次的に実行される文の列である。ここで、文とは、アクタの属性値の変化、デマンドの送信、イベントの送信·読み込み等やこれらを条件分岐や繰り返しを用いて構造化したものである。

i) シナリオ

シナリオとは、すべてのアニメーションストーリにただ一つ存在するアクタであり、他のアクタの管理を行う。シナリオアクタは、そのアニメーションに登場する他の全てのアクタを属性として持つ。シナリオのメソッドをシーンと呼ぶ。シーンはアニメーションにおける一つの幕に対応し、複数のシーンは同時にプロセスとして実行されることはない。

ii) キャラクタ

キャラクタは画面上で可視化され、回転や移動、あるいは歩く、笑うといった目にみえる動きを行うアクタである。キャラクタは属性として、位置、形状、速度等を持つことができる。また、サブキャラクタを属性として持つことができ、この関係によりキャラクタは階層構造を構成可能である。

iii) ウオッチャ

ウオッチャはキャラクタ間の制約や系の外部との同期等を表現するために導入したアクタである。

例えば、二つのキャラクタの衝突の検知を考える。衝突検知の手続きをキャラクタの一方あるいは双方のメソッド中に記述すると、一方のキャラクタの記述が他方に従属してしまうだけでなく、その衝突検知の手続きは特定の二つのキャラクタの衝突の検知にのみに有効となり、一般性が損なわれる。

そこで、衝突検知等の制約監視はウオッチャという特殊なアクタを導入し、そのメソッドとして記述することにする。例えば、衝突検知ウオッチャのメソッドは、監視対象のキャラクタに対応した二つの引数を持つ。このメソッドは、各ティックに二つのキャラクタが衝突しているかどうかの検査を行う。もし、衝突を検知したなら、衝突イベントの送出を行う。このようなウオッチャを導入することにより、煩雑な衝突検知の手続き等を局所化できるだけでなく、キャラクタの記述を制約の記述やその存在から独立させることができる

また、ウオッチャはアニメーションの系をユーザ等の外部の系と同期させるためのインタフェース(外部系の監視役)として利用できる。外部の系を監視するウオッチャは、各ティックにおいて外部からのメッセージを受取り、それを解釈し適当なイベントやデマンドをアニメーションン系の中に送出する。また、系外部に対して適当なメッセージを送出する。このようなウオッチャを用いることにより、アニメーション記述における外部依存部分の局所化が可能となる。

5.2. アクタ記述の実行(処理系の動き)

前節において、シナリオ、キャラクタ、ウオッチャの三種類のアクタを導入した。これらのアクタの実行に関し、実行順序とメッセージ(デマンドとイベント)の存在時間に関する制約を課し、各アクタ群の持つ意味を明確にする。

シナリオ、キャラクタ群、ウオッチャ群はこの順番で実行される。キャラクタ群、ウオッチャ群におけるアクタの実行順は任意とする。アニメーションの開始時にはまずシナリオの実行から始められ、ティックはシナリオの実行後に更新される。

メッセージの存在時間に関しては、以下のように修正·詳細化する。イベントは、その送信アクタの属する群のすべてのアクタの実行の終了直後から、次のティックの処理における送信アクタの属する群のすべてのアクタの実行の終了直後までイベントプールに存在する。同様に、デマンドは、その送信アクタの属する群のすべてのアクタの実行の終了直後から次のティックの処理における送信アクタの属する群のすべてのアクタの実行の終了直後までの間に、受信アクタにより処理される。以上の機構を図5に示す。

上記概念に基づくアニメーション並行記述言語Easy(Event-driven Animation System)を設計し、その実装を行った。この言語では、キャラクタ、ウオッチャの定義のために、まずクラスの定義を行い、個々のキャラクタ、ウオッチャはそのクラスのインスタンスとして宣言される。メソッドの定義は、クラス定義の一部として行われる。メソッドの記述はSTART部とBODY部を持つ。START部は、メソッドがプロセスとして起動されたティックに実行され、BODY部はプロセスが終了するまで毎ティック実行される。

Easy言語よる記述は、C言語による記述に変換される。このC言語による記述をコンパイルし

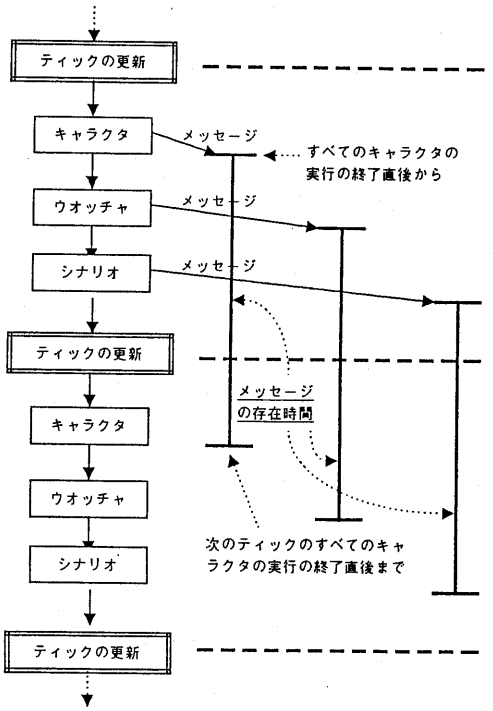

図5. アクタの処理順と
メッセージの存在時間

Easyのライブラリとリンクすることにより実時間アニメーションのコードが生成される。

付録にアニメーション記述の例を上げる。

6. まとめ

アニメーションをアクタと呼ぶ登場物の並行動作系とみなすモデルの提案と、このモデルに基づく記述言語Easyの概説を行った。このモデルは次の特徴を持つ。

●他に従属せずに動作する独立性の高いアクタを用いてアニメーションを表現する。
●アクタの動きの同期のために、デマンドとイベントという2種類の通信機能を提供する。デマンドは、受信アクタに対して動きの命令を行う通信であり、イベントは、アクタの状態や制約の破綻等を他のアクタへ知らせる通信である。デマンドは、確定的な動きの表現に適し、イベントはアクタの自律的な動きの表現に適する。

これらの特徴により、
●アクタの部品化による、動きの再利用
●記述時には発生時間を決定できない動きの表現が可能となる。

本研究は、通商産業省工業技術院大型プロジェクトの一環として(財)情報処理相互運用技術協会(INTAP)が新エネルギー・産業技術総合開発機構からの委託をうけ、シャープ株式会社がINTAPからの再委託研究として実施したものである。

参考文献

[1] Thalmann, N. and Thalmann, D.: Computer Animation, Springer-Verlag (1985).

[2] 安居院, 中嶋, 大江: コンピュータアニメーション, 産報出版 (1983).

[3] Kahn, K. M. and Hewitt, C.: Dynamic Graphics Using Quasi Parallelism, Computer Graphics, Vol. 12, No. 3, pp. 357-362 (1978).

[4] Reynolds, C. W.: Computer Animation with Scripts and Actors, Computer Graphics, Vol. 16, no. 3, pp. 289-296 (1982).

[5] 内木, 丸一, 所: 行動シミュレーションに基づいたアニメーションシステムParadise, コンピュータソフトウエア, Vol. 4, No.2, pp. 24-38 (1987).

[6] Goldberg, A. and Robinson, D.: Smalltalk 80-The Language and its implementation, Addison Wesley (1983).

[7] Gelernter D.: Generative Communication in Linda, ACM Trans. Program. Lang. Syst., Vol. 7, No. 1, pp. 80-112 (1985).

付録

Easy言語によるアニメーションの記述例を示す。

A1. アニメーションの内容

登場物:人、石

人はユーザからのキー入力を受けて、画面上を上下左右に移動する。人と石が衝突すると、人は倒れる。

A.2. キャラクタの記述(図A.1)

i) 人(man)クラスの定義

●属性(状態変数)として以下の3つを持つ。

·count: 移動のためのティックカウンタ

·step: 移動を行うティック間隔

·width: 1回の移動量

画像パーツを扱うコンポーネント(compo)は上下左右の各々の方向への歩行表現に4種類ずつと、倒れた姿の表現のための計17種類からなる。

●メソッドとして以下の2つを持つ。

·move: 指定された方向に動き続ける。

START部では、指定された移動の方向から、X,Y方向の座標の変化量、初期画像を設定する。BODY部では、stepティックに1回座標を変更する。変更時には画像を切り換えて、歩く様子を表現する。

·fall: 倒れる

画像を倒れた姿に切り換え、即座に終了する。

ii) 石(stone)クラスの定義

石はただ存在しているだけであり、メソッドは持たない。

A.3. ウオッチャの記述(図A.2)

i) キー入力監視ウオッチャ(key)の定義

ユーザからのキー入力を受けて、意味の有るキー入力の場合は対応するイベントを出力する。

ii) 衝突監視ウオッチャ(crash)の定義

引数で指定した2つのアクタが衝突したときにイベントを出力する。各アクタの座標と表示画像の大きさから衝突を検知する。

A.4. シナリオの記述(図A.3)

i) キャラクタ、ウオッチャの生成

人アクタと石アクタを一つずつ生成し、初期位置等の設定を行う。

```
defclass    man=            /*  人のクラス定義    */
{
    int count;              /*  キャラクタ変数の定義 */
    int step;
    int width;
    compo  left0 = {    /*  平常時の人の姿を定義 */
        parts = man__left__image__0;
                            /*  画像パーツの指定      */
        x = 0;  y = 0;          /*  オフセットの指定 */
    }
    compo  left1 = {
        parts = man__left__image__1;
        x = 0;  y = 0;
    }
    .
    .
    .
    compo  down3 = {
        parts = man__down__image__2;
        x = 0;  y = 0;
    }
    compo  fall = {     /*  倒れた人の姿を定義 */
        parts = man__fallen__image;
        x = 0;  y = 0;
    }
}

def__method                 /*  移動メソッドの定義   */
    move(int dir : int vx, int vy)

{
START:          /*  プロセス起動時に実行される部分 */
    count = 0;
    vx= vy =0;
    switch (dir)    /*  引数 dir の値によって進行方向を
    {                                       決定    */
    case LEFT:
        vx = -width ;
        compo__state=left0;
        break ;
    case RIGHT:
        vx = width ;
        compo__state=right0;
        break ;
    case UP :
        vy = -width ;
        compo__state=up0;
        break ;
    case DOWN:
        vy = width ;
        compo__state=down0;
        break ;
    }
BODY:               /*  毎ティック実行される部分    */
    if(++count > step)/*  stepティックに1回移動    */
    {
        count = 0;
        x +=vx;   y +=vy;       /*  座標の更新     */
        if(++compo__state % 4 == 3)
            compo__state -= 4;
                /*  4枚の画像を切り替えることに
                    より歩行を表現する    */
    }
}

def__method fall()
{
START:
    compo__state = fall;    /*  画像を切り替え    */
    stop__method();         /*  プロセスを終了*/
BODY:
}




defclass    stone=          /*  石のクラス定義          */
{
    compo  normal = {    /*  石の姿を定義  */
        parts = stone__image;
        x = 0;  y = 0;
    }
}
```

図A.1. キャラクタの記述例

キー入力を監視するウオッチャと、人と石の衝突を監視するウオッチャを生成する。

ii) シーンの記述

シーン開始時には、人を活動状態、可視状態にし、石を可視状態にし、2つのウオッチャを起動する。

通常動作時は以下を行う。

·CRASHというイベントを受けると人の移動を止め、倒れさせる。

·MOVEというイベントを受けると新たに移動を起動する。

```
def_watcher key() /*  キー入力監視ウオッチャ定義 */
{
    int k;
START:
BODY:
    k = realtime_getchar();
                /*  待ちを行わないキー入力呼出  */
    switch(k)
    {
    case 'h':                /*  'h' キーが押された場合
        out("MOVE", LEFT);  LEFT というパラメタを
        break;                持つMOVEという名前の
    case 'j':                 イベントを出力    */
        out("MOVE", DOWN);
        break;
    case 'k':
        out("MOVE", UP);
        break;
    case 'l':
        out("MOVE", RIGHT);
        break;
    }
}

def_watcher crash(ACTOR a, ACTOR b)
                /*  衝突検知ウオッチャの定義  */
{   /*      引数で渡された2つのキャラクタが衝突
            したときイベントを出力  */
    struct pr_size *asxy, *bsxy;
START:
BODY:
    asxy = &(a->compo[a->compo_state].parts
            ->picture->pr_size);
    bsxy = &(b->compo[b->compo_state].parts
            ->picture->pr_size);
              /*  2つのキャラクタの大きさを求める  */
    if((a->x+asxy->x > b->x) &&
       (a->x < b->x+bsxy->x) &&
       (a->y+asxy->y > b->y) &&
       (a->y <b->y+bsxy->y))
    /*  2つのキャラクタの領域が重なりを持つ場合  */
    {
        out("CRASH", a, b);
            /*  CRASHという名前のイベントを出力 */
    }
}
```

図A.2. ウオッチャの記述例

```
make_actor   man    man1 = {
            /*  manクラスのキャラクタ man1 を生成 */
    step = 160;     width = 20;     count = 0;
    x = -120;   y = 800;
    compo_state = normal; /*  初期画像の指定  */
};

make_actor   stone   stone1 = {
        /*  stoneクラスのキャラクタ stone1 を生成 */
    x = 400;   /*  キャラクタの初期位置(X方向)設定 */
    y = 600;   /*  キャラクタの初期位置(Y方向)設定 */
};

make_watcher    crash   crash1;
        /*  crashクラスのウオッチャ crash1 を生成  */
make_watcher    key     user;
        /*  keyクラスのウオッチャ user を生成  */
set_first_scene scene1;
        /*  シーン scene1 から実行を開始 */

scene scene1                /*  シーン scene1 の定義  */
{
    int dir;  /* シーン中で使用するローカル変数    */
START:                          /*  シーンの初期設定 */
    activate(man1);
            /*  キャラクタ man1 を活性化    */
    visualize(man1);
            /*  キャラクタ man1 を可視化    */
    visualize(stone1);
    watch_start(crash1, man1, stone1);
/*  ウオッチャ crash1(キャラクタ man1 とキャラクタ
    stone1 の衝突を検知するのプロセス)の起動*/
    watch_start(user);
            /*  ウオッチャ user のプロセスを起動    */
BODY:           /*  毎ティック実行される部分    */
    if(in("CRASH"))
        /*  「衝突」というイベントが存在するなら  */
    {
        remove(man1, "move");
  /*  キャラクタ man1 に move プロセスの終了要求  */
        send(man1, "fall");
    /*  キャラクタ man1 に fall プロセスの起動要求 */
    }
    if(in("MOVE", @dir))
        /*  「動け」というイベントが存在するなら  */
    {
        send(man1, "move", dir);
    /*  キャラクタ man1 に move プロセスの起動要求*/
    }
}
```

図A.3. シナリオの記述例